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

**Term-work Report**

|  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| **Term-work** | *01* | | | | |  |  | | | | |
| **Student Name** | Abhinav Sushil Pawar | | | | |  |  | | | | |
| **SRN** | 01FE21BEC087 | | | | **Roll Number** | | 248 | | **Division** | B | |
| **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** | | Singly linked list | | | | | O(n) | | | |
| Recursive | **1** | | Array | | | | | O(5n) | | | |
| **2** | | Stack | | | | | O(2n) | | | |
| Back tracking | **1** | | 2D array | | | | | O(9n) | | | |
| **2** | | n/a | | | | | n/a | | | |
| Divide and conquer | **1** | | Array | | | | | O(nlogn) | | | |
| **2** | | 2D array | | | | | O(n2) | | | |
| Dynamic programming | **1** | | Array | | | | | O(n) | | | |
| **2** | | Singly linked list | | | | | O(n) | | | |
| Greedy | **1** | | Array | | | | | O(n2) | | | |
| **2** | | Array | | | | | O(n2) | | | |
| Branch and bound | **1** | | 2D array | | | | | O(n2) | | | |
| **2** | | Array | | | | | O(n2) | | | |
| Brute force | **1** | | Character array | | | | | O(mn) | | | |
| **2** | | Array | | | | | O(n) | | | |
| Randomized | **1** | | Array | | | | | O(n) | | | |
| **2** | | n/a | | | | | n/a | | | |

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

*I was able to solve most, if not all, of the problems. The challenge was to learn a very large set of new concepts which were unknown to me in a very short period of time.*

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

*Better understanding of each of the algorithms with a simple example to understand its working and flow of instructions; and most importantly, calculation of time complexity would be the best assistance I could get for this work.*

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

*I should make sure that I have expertise on basic operations of all the data structure. Also, clarity on calculation of time complexity is required very much.*

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
| **Modularity** |  | **Documentation** |  | **Indentation** |  | **Programming practices** |  |
| **Type of Algorithm:** *Iterative* | | | | | | | |
| **Details of the algorithm:** | | | | | | | |
| *Iteration (meaning, to repeat) refers to a sequence of instructions where an algorithm needs to be executed repeatedly until a desired result is obtained, which is terminated by a terminating condition. Iterative algorithms find wide range of applications, some of which are to generate sequences, searching a key element in a list, etc.*  *Input format:*  *First line contains number of array elements to be entered*  *Subsequent lines contain the array elements*  *Last line is the key element* | | | | | | | |
| **Code for example 1:** | | | | | | | |
| *#include<stdio.h>*  *#define MAX 100*  *void read(int array[],int n)*  *{*  *for(int i=0;i<n;i++){ //f(n)=n+1*  *scanf("%d",&array[i]);*  *}*  *}*  *int search(int array[], int n,int key)*  *{*  *int j,found=0;*  *for(j=0;j<n;j++)*  *{*  *if(array[j]==key)*  *{*  *found=1;*  *break;*  *}*  *}*  *if(found==1)*  *return j;*  *else*  *return -1;*  *}* | | | | | | | |
| **Sample Input:** | | | | | | | |
| *5*  *23*  *54*  *65*  *32*  *56*  *32* | | | | | | | |
| **Sample Output:** | | | | | | | |
| *3* | | | | | | | |
| **Time complexity calculation:** | | | | | | | |
| First for loop runs for (n+1) times  Second for loop runs for (n+1) times  Total (2n)  Hence, Time complexity: O(n) | | | | | | | |

|  |
| --- |
| **Code for example 2:** |
| **#include<stdio.h>**  **struct node**  **{**  **int data;**  **struct node \*next;**  **};**  **typedef struct node \*NODE;**  **NODE head, newnode, temp, cur,next,prev; //Global declaration**  **NODE head=NULL;**  **NODE getnode()**  **{**  **newnode=(NODE)malloc(sizeof(struct node));**  **if(newnode==NULL)**  **{**  **printf ("Memory not allocated for node! \n");**  **exit(0);**  **}**  **newnode->next=NULL;**  **return newnode;**  **}**  **void findmiddle(NODE head)**  **{**  **NODE middle=head,fast=head,slow=head;**  **if(head==NULL)**  **printf("SLL is empty\n");**  **else**  **{**  **while(fast!=NULL && fast->next!=NULL) // && fast->next->next!=NULL) --> if first number of the middle two nos is to be assigned (in case of even number of numbers)**  **{**  **slow=slow->next;**  **fast=fast->next->next;**  **}**  **middle=slow;**  **}**  **printf("Middle element is %d\n",middle->data);**  **}**  **void display\_SLL(NODE head)**  **{**  **if(head==NULL)**  **{**  **printf ("No data available in Linked list!");**  **}**  **temp=head;**  **printf("\n\nDisplaying numbers from nodes:\n");**  **do**  **{**  **printf ("%d\t",temp->data);**  **temp=temp->next;**  **}while(temp->next!=NULL);**  **printf ("%d ",temp->data);**  **}**  **NODE reverse\_nodes(NODE head,int n)**  **{**  **cur=head;**  **prev=head;**  **while(cur!=NULL)**  **{**  **next=cur->next;**  **cur->next=prev;**  **prev=cur;**  **cur=next;**  **}**  **head=prev;**  **}**  **NODE insertend(int num)**  **{**  **newnode=getnode();**  **newnode->data=num;**  **if(head==NULL)**  **{**  **head=newnode;**  **}**  **else**  **{**  **temp=head;**  **while(temp->next!=NULL)**  **temp=temp->next;**  **temp->next=newnode;**  **}**  **//printf("Entered the element %d in a node\n",newnode->data);**  **return(head);**  **}**  **void main()**  **{**  **int choice,num,middle;**  **while(1)**  **{**  **printf("\n1:Insert\n2:Findmiddle\n3:Display\n4:exit\n");**  **scanf("%d",&choice);**  **switch(choice)**  **{**  **case 1: scanf("%d",&num);**  **newnode=insertend(num); break;**  **case 2: findmiddle(head);**  **break;**  **case 3: display\_SLL(head);break;**  **case 4: exit(0); break;**  **default: printf("invalid choice!\n");**  **}**  **}**  **}** |
| **Sample Input:** |
| *1 3*  *1 4*  *1 5*  *1 7*  *1 9*  *1 6*  *3*  *2*  *4* |
| **Sample Output:** |
| *Displaying numbers from nodes:*  *3 4 5 7 9 6*  *Middle element is 7* |
| **Time complexity calculation:** |
| *Complexity of while loop here turns out to be*  *O(n)* |

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
| **Modularity** |  | **Documentation** |  | **Indentation** |  | **Programming practices** |  |
| **Type of Algorithm:** *Recursive* | | | | | | | |
| **Details of the algorithm:** | | | | | | | |
| *Recursion in simple terms mean that a function is being called in itself; or “A function calling itself”. Recursion is used to compute the factorial of a number, Tower of Hanoi, some sorting algorithms etc. But the time complexity is very high of recursion.* | | | | | | | |
| **Code for example 1:** | | | | | | | |
| *//Bubble sort using recursion*  *#include <stdio.h>*  *#define MAX 1000*  *void swap(int \*a, int \*b)*  *{*  *int temp = \*a;*  *\*a = \*b;*  *\*b = temp;*  *}*  *void bubblesort(int arr[], int n)*  *{*  *if (n == 1) // Base case*  *return;*  *int count = 0;*  *for (int i=0; i<n-1; i++)*  *{*  *if (arr[i] > arr[i+1]) //ascending order*  *{*  *swap(&arr[i], &arr[i+1]);*  *count++;*  *}*  *}*  *if (count==0)*  *return;*  *bubblesort(arr, n-1); //upto nth element is sorted, sort remaining*  *}*  *void read(int array[],int n)*  *{*  *for(int i=0;i<n;i++){ //f(n)=n+1*  *scanf("%d",&array[i]);*  *}*  *}*  *void display(int array[], int n)*  *{*  *for (int i=0; i < n; i++)*  *printf("%d ", array[i]);*  *printf("\n");*  *}*  *void main()*  *{*  *int array[MAX];*  *int n;*  *scanf("%d",&n);*  *read(array,n);*  *bubblesort(array, n);*  *printf("Sorted array : \n");*  *display(array, n);*  *}* | | | | | | | |
| **Sample Input:** | | | | | | | |
| *5*  *45 32 89 12 56* | | | | | | | |
| **Sample Output:** | | | | | | | |
| *Sorted array :12 32 45 56 89* | | | | | | | |
| **Time complexity calculation:** | | | | | | | |
| In recursion if a function calls itself m times, then the time complexity is mn.  Here, in this example, we can see that the bubblesort() function is being called once for each element sorting.  Hence, its complexity = 5n | | | | | | | |

|  |
| --- |
| **Code for example 2:** |
| *//Reverse the elements of a stack*  *#include<stdio.h>*  *#define MAX 1000*  *int sp=-1;*  *void push(int stack[],int item)*  *{*  *if(sp==MAX-1) //overflow*  *{*  *printf("Stack overflow\n");*  *return;*  *}*  *sp++;*  *stack[sp]=item;*  *}*  *void display(int stack[])*  *{*  *int item,count=0;*  *while(sp!=-1)*  *{*  *item=pop(stack);*  *printf("%d ",item);*  *count++;*  *}*  *printf("\n");*  *sp+=count;*  *}*  *int pop(int stack[])*  *{*  *if(sp==-1) //isempty*  *{*  *printf("Stack underflow\n");*  *return;*  *}*  *int item=stack[sp];*  *sp--;*  *return item;*  *}*  *void reverse(int stack[])*  *{*  *int temp;*  *if(sp!=-1)*  *{*  *temp=pop(stack);*  *printf("temp=%d",temp);*  *reverse(stack);*  *insert\_at\_bottom(stack,temp);*  *}*  *}*  *void insert\_at\_bottom(int stack[],int item)*  *{*  *if(sp==-1)*  *push(stack,item);*  *else*  *{*  *int temp=pop(stack);*  *insert\_at\_bottom(stack,item);*  *push(stack,temp);*  *}*  *}*  *void main()*  *{*  *int stack[MAX];*  *int n,item;*  *scanf("%d",&n);*  *for(int i=0;i<n;i++) //Complexity=n*  *{*  *scanf("%d",&item);*  *push(stack,item);*  *}*  *display(stack); //Complexity of this instruction is of order = n*  *item=pop(stack); //Complexity of this instruction is of order = constant*  *display(stack); //Complexity of this instruction is of order = n*  *reverse(stack); //Complexity of this instruction is of order = 2^n +2^n*  *display(stack);*  *}* |
| **Sample Input:** |
| *5*  *89 45 87 12 23* |
| **Sample Output:** |
| *23 12 87 45 89* |
| **Time complexity calculation:** |
| *Complexity of reverse() function is of order = 2^n +2^n*  *Hence, complexity = O(2n)* |

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
| **Modularity** |  | **Documentation** |  | **Indentation** |  | **Programming practices** |  |
| **Type of Algorithm:** *Back tracking* | | | | | | | |
| **Details of the algorithm:** | | | | | | | |
| *In backtracking, we incrementally build up a candidate to the solution, but as soon as it is found that it is not the solution, we go back to where we assumed that it was the solution, and delete the backtrack.*  *It is a convenient technique for parsing, knap sacking, crosswords, Sudoku etc.* | | | | | | | |
| **Code for example 1:** | | | | | | | |
| *//the requirements for Sudoku solution: (non-duplication across rows,columns, and boxes)*  *#include <stdio.h>*  *#include <stdlib.h>*  *#define N 9*  *int checkelement(int partsudoku[N][N], int row,int col, int num)*  *{*  *for (int x = 0; x <= N-1; x++) //Check for identical numbers in row*  *{*  *if (partsudoku[row][x] == num)*  *return 0;*  *}*  *for (int x = 0; x <= N-1; x++) //Check for identical numbers in column*  *{*  *if (partsudoku[x][col] == num)*  *return 0;*  *}*  *int rowBeg,colBeg;*  *rowBeg = row - row%3;*  *colBeg = col - col%3;*  *for (int i = 0; i < 3; i++) //Check for identical number in 3x3 box*  *{*  *for (int j = 0; j < 3; j++)*  *{*  *if (partsudoku[i+rowBeg][j+colBeg] == num)*  *return 0;*  *}*  *}*  *return 1; //No identical elements*  *}*  *int solveSudoku(int partSudoku[N][N], int row, int col)*  *{*  *if (row == N - 1 && col == N) //check if reached the end of sudoku (last row (index=8), and last+1 column (index=9))*  *return 1; //solved*  *if (col == N) //check if we reached the end of particular row (last+1 column (index=9))*  *{*  *row++; //reinitialise indices for next row*  *col = 0;*  *}*  *if (partSudoku[row][col] > 0) //checks if element at this position is >0*  *return solveSudoku(partSudoku, row, col + 1); //if true,check for next col*  *for (int num = 1; num <= N; num++) //check for all 9 numbers, one by one, to enter in grid*  *{*  *if (checkelement(partSudoku, row, col, num)==1)*  *{*  *partSudoku[row][col] = num; //assuming entered element is correct*  *if (solveSudoku(partSudoku, row, col + 1)==1) //Check for next column*  *return 1;*  *}*  *//Assumption is wrong -------. Backtrack*  *partSudoku[row][col] = 0; //Hence remove the incorrect assumption*  *}*  *return 0;*  *}*  *void display(int array[N][N])*  *{*  *for (int i=0;i<N;i++)*  *{*  *for (int j=0;j<N;j++)*  *{*  *printf("%d ",array[i][j]);*  *}*  *printf("\n");*  *}*  *}*  *void main()*  *{*  *int partSudoku[N][N] = {*  *{ 3, 0, 6, 5, 0, 8, 4, 0, 0 },*  *{ 5, 2, 0, 0, 0, 0, 0, 0, 0 },*  *{ 0, 8, 7, 0, 0, 0, 0, 3, 1 },*  *{ 0, 0, 3, 0, 1, 0, 0, 8, 0 },*  *{ 9, 0, 0, 8, 6, 3, 0, 0, 5 },*  *{ 0, 5, 0, 0, 9, 0, 6, 0, 0 },*  *{ 1, 3, 0, 0, 0, 0, 2, 5, 0 },*  *{ 0, 0, 0, 0, 0, 0, 0, 7, 4 },*  *{ 0, 0, 5, 2, 0, 6, 3, 0, 0 }*  *};*  *int status = solveSudoku(partSudoku, 0, 0); //start by checking 1st element*  *if (status==1)*  *display(partSudoku);*  *else*  *printf("No solution found!");*  *}* | | | | | | | |
| **Sample Input:** | | | | | | | |
| * *(Incomplete Sudoku is system initialised)* | | | | | | | |
| **Sample Output:** | | | | | | | |
| *3 1 6 5 7 8 4 9 2*  *5 2 9 1 3 4 7 6 8*  *4 8 7 6 2 9 5 3 1*  *2 6 3 4 1 5 9 8 7*  *9 7 4 8 6 3 1 2 5*  *8 5 1 7 9 2 6 4 3*  *1 3 8 9 4 7 2 5 6*  *6 9 2 3 5 1 8 7 4*  *7 4 5 2 8 6 3 1 9* | | | | | | | |
| **Time complexity calculation:** | | | | | | | |
| *Complexity of checkelement() function is straightforward n2 as there is a nested for loop[ inside the function.*  *Complexity of solvesudoku() function gets complex because of recursion. At the worst case, where we cannot find a solution to the Sudoku or we find at the last step, the complexity becomes 9n.* | | | | | | | |

|  |
| --- |
| **Code for example 2:** |
| *<Write/paste your code here>*  *<code should be well documented and indented>* |
| **Sample Input:** |
| *<Sample Input>* |
| **Sample Output:** |
| *<Sample Output>* |
| **Time complexity calculation:** |
|  |

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
| **Modularity** |  | **Documentation** |  | **Indentation** |  | **Programming practices** |  |
| **Type of Algorithm:** *Divide and conquer* | | | | | | | |
| **Details of the algorithm:** | | | | | | | |
| *The basic idea behind divide and conquer algorithm is to decompose a given problem into smaller but similar and simpler sub problems, which can eventually be solved using basic conditions. Since we divide the problem into smaller chinks to solve it, the name is ‘Divide and conquer’*  *Divide and conquer finds applications I some arrangement of data, e.g. binary search, or in merge sort, quick sort etc.* | | | | | | | |
| **Code for example 1:** | | | | | | | |
| *//Binary search*  *//applicable for some arrangement of data*  *//reduce the search space --> go to middle*  *#include<stdio.h>*  *#define MAX 1000*  *void read(int array[],int n)*  *{*  *for(int i=0;i<n;i++){*  *scanf("%d",&array[i]);*  *}*  *}*  *int binary\_search(int array[],int n,int key)*  *{*  *int flag=0;*  *int start=0,end=n-1,mid=0;*  *while(start<=end)*  *{*  *mid=(start+end)/2;*  *if(array[mid]==key)*  *{*  *flag=1;*  *break;*  *}*  *else if(array[mid]<key)*  *start=mid+1;*  *else*  *end=mid-1;*  *}*  *if(flag==1)*  *return mid;*  *else*  *return -1;*  *}*  *void main()*  *{*  *int array[MAX];*  *int n,key,flag;*  *scanf("%d",&n);*  *read(array,n);*  *scanf("%d",&key);*  *int found\_index=binary\_search(array,n,key);*  *if(found\_index!=-1)*  *printf("element found at index %d\n",found\_index);*  *else*  *printf("element not found!");*  *}* | | | | | | | |
| **Sample Input:** | | | | | | | |
| *8*  *1 2 3 4 5 6 7 8*  *5* | | | | | | | |
| **Sample Output:** | | | | | | | |
| *Element found at index 4* | | | | | | | |
| **Time complexity calculation:** | | | | | | | |
| *The area of searching the key elements gets reduced to half its previous data range as the loop runs. This means that it gets binomially reduced, which is opposite of exponential.*  *Hence the time complexity of binary search is O(nlog)* | | | | | | | |

|  |
| --- |
| **Code for example 2:** |
| *//Matrix multiplication using Strassens Algorithm*  *#include<stdio.h>*  *#define MAX 100*  *int resMatrix[MAX][MAX];*  *int resMatrix2[MAX][MAX],resMatrix3[MAX][MAX],resMatrix4[MAX][MAX];*  *void addmatrix(int matrixA[MAX][MAX],int matrixB[MAX][MAX],int row,int col)*  *{*  *for(int i=0;i<row;i++)*  *{*  *for(int j=0;j<col;j++)*  *{*  *resMatrix[i][j] = matrixA[i][j] + matrixB[i][j];*  *}*  *}*  *}*  *void mulmatrix(int matrixA[MAX][MAX],int row1,int col1,int matrixB[MAX][MAX],int row2,int col2)*  *{*  *if(row2==col1)*  *{*  *for(int i=0;i<row1;i++)*  *{*  *for(int j=0;j<col2;j++)*  *{*  *resMatrix[i][j]=matrixA[i][0]\*matrixB[0][j];*  *if (col1>1){*  *resMatrix2[i][j]=matrixA[i][1]\*matrixB[1][j];*  *if(col1>2){*  *resMatrix3[i][j]=matrixA[i][2]\*matrixB[2][j];*  *if(col1>3)*  *resMatrix4[i][j]=matrixA[i][3]\*matrixB[3][j];*  *}*  *}*  *}*  *}*  *if(col1>1){*  *addmatrix(resMatrix,resMatrix2,row1,col2);*  *if(col1>2){*  *addmatrix(resMatrix,resMatrix3,row1,col2);*  *if(col1>3)*  *addmatrix(resMatrix,resMatrix4,row1,col2);*  *}*  *}*  *}*  *else*  *{*  *printf("Matrix multiplication not possible!\n");*  *exit(0);*  *}*  *}*  *void readmatrix(int matrix[MAX][MAX], int row, int col)*  *{*  *for(int i=0;i<row;i++)*  *{*  *for(int j=0;j<col;j++)*  *{*  *scanf("%d",&matrix[i][j]);*  *}*  *}*  *}*  *void displaymatrix(int matrix[MAX][MAX],int row, int col)*  *{*  *for(int i=0;i<row;i++)*  *{*  *printf("\t\t");*  *for(int j=0;j<col;j++)*  *{*  *printf(" %d ",matrix[i][j]);*  *}*  *printf("\n");*  *}*  *}*  *void main()*  *{*  *int matrixA[MAX][MAX],matrixB[MAX][MAX];*  *int row1,col1,row2,col2;*  *scanf("%d%d",&row1,&col1);*  *readmatrix(matrixA,row1,col1);*  *printf("\nMatrix A: \n");*  *displaymatrix(matrixA,row1,col1);*  *scanf("%d%d",&row2,&col2);*  *readmatrix(matrixB,row2,col2);*  *printf("\nMatrix B: \n");*  *displaymatrix(matrixB,row2,col2);*  *mulmatrix(matrixA,row1,col1,matrixB,row2,col2);*  *printf("\n\nResultant multiplied matrix: \n");*  *displaymatrix(resMatrix,row1,col2);*  *}* |
| **Sample Input:** |
| *2*  *2*  *1 2*  *3 4*  *2*  *2*  *5 6*  *7 8* |
| **Sample Output:** |
| *Matrix A:*  *1 2*  *3 4*  *Matrix B:*  *5 6*  *7 8*  *Resultant multiplied matrix:*  *19 22*  *43 50* |
| **Time complexity calculation:** |
| *There are 2 for loops in the function. But the iteration are controlled by the condition. The complexity of each for loop is n.*  *Nested for loop has complexity O(n^2).* |

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
| **Modularity** |  | **Documentation** |  | **Indentation** |  | **Programming practices** |  |
| **Type of Algorithm:** *Dynamic programming* | | | | | | | |
| **Details of the algorithm:** | | | | | | | |
| *Dynamic programming is very similar to recursion, except when the problem is broken down into smaller sub problems, the intermediate results are store. This enables the function to use these intermediate results to find next intermediate result, to finally calculate the end result.*  *Dynamic programming finds its applications in finding shortest path, optimal search solutions, finding n-th element of an arithmetic sequence, finding a sequence where the current element depends the previous element etc.* | | | | | | | |
| **Code for example 1:** | | | | | | | |
| *//Golomb sequence using dynamic programming*  */\*Golomb sequence is a sequence where n-th term is equal to number of times n appears in the sequence.*  *\*/*  *#include<stdio.h>*  *void Generate\_Golomb(int n)*  *{*  *int G[n+1];*  *G[1]=1; //base case*  *for(int i=1;i<n;i++)*  *{*  *G[i+1]=1+G[(i+1)-G[G[i]]];*  *}*  *printf("Golomb sequence with position of each element:\n");*  *display(G,n);*  *}*  *void display(int array[],int n)*  *{*  *printf("Sequence: ");*  *for(int i=1;i<n;i++){*  *printf(" %d ",array[i]);*  *if(i>9) printf(" ");*  *}*  *printf("\nPosition: ");*  *for(int i=1;i<n;i++)*  *printf(" %d ",i);*  *}*  *void main()*  *{*  *int n;*  *scanf("%d",&n);*  *Generate\_Golomb(n);*  *}* | | | | | | | |
| **Sample Input:** | | | | | | | |
| *27* | | | | | | | |
| **Sample Output:** | | | | | | | |
| *Golomb sequence with position of each element:*  *Sequence: 1 2 2 3 3 4 4 4 5 5 5 6 6 6 6 7 7 7 7 8 8 8 8 9 9 9*  *Position: 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 21 22 23 24 25 26* | | | | | | | |
| **Time complexity calculation:** | | | | | | | |
| *There is only a single for loop in any function of the written program. Hence the time complexity is simply O(n)* | | | | | | | |

|  |
| --- |
| **Code for example 2:** |
| *//Finding nth element in Fibonacci Series using Dynamic Programming*  *#include<stdio.h>*  *struct node //Structure declaration*  *{*  *int data;*  *struct node \*next;*  *};*  *typedef struct node \*NODE; //alias name for structure*  *NODE head, newnode, temp, cur,next,prev,beforeprev; //Global declaration*  *NODE head=NULL;*  *NODE getnode() //Creating a new node*  *{*  *newnode=(NODE)malloc(sizeof(struct node));*  *if(newnode==NULL) //Overflow condition (maximum memory reached)*  *{*  *printf ("Memory not allocated for node! \n");*  *exit(0);*  *}*  *newnode->next=NULL; //assigning link of created node to NULL*  *return newnode;*  *}*  *NODE insertend(int num) //Insert newnode at the end of SLL*  *{*  *newnode=getnode(); //Creating newnode*  *newnode->data=num; //assign data into node*  *if(head==NULL) //SLL is empty*  *{*  *head=newnode; //newnnode is the first node*  *}*  *else //SLL exists*  *{*  *temp=head;*  *while(temp->next!=NULL) //traverse till the end of SLL*  *temp=temp->next;*  *temp->next=newnode; //last node*  *}*  *return(newnode);*  *}*  *void display\_SLL(NODE head)*  *{*  *if(head==NULL) //empty SLL*  *{*  *printf ("No data available in Linked list!");*  *return;*  *}*  *temp=head;*  *printf("Displaying numbers from nodes: ");*  *do //traversing throughout the SLL (since it is not empty)*  *{*  *printf ("%d ",temp->data);*  *temp=temp->next;*  *}while(temp!=NULL);*  *}*  *int fibonacci(int n)*  *{*  *beforeprev=NULL;*  *prev=insertend(0);*  *cur=insertend(1);*  *for (int i=2;i<=n;i++)*  *{*  *int buffer\_fib = cur->data + prev->data;*  *beforeprev=prev;*  *prev=cur;*  *cur=insertend(buffer\_fib);*  *}*  *return (cur->data);*  *}*  *int main ()*  *{*  *int n,nth\_fib;*  *scanf("%d",&n);*  *nth\_fib=fibonacci(n);*  *display\_SLL(head);*  *printf("\n%d\n", nth\_fib);*  *}* |
| **Sample Input:** |
| *15* |
| **Sample Output:** |
| *rDisplaying numbers from nodes: 0 1 1 2 3 5 8 13 21 34 55 89 144 233 377 610*  *610* |
| **Time complexity calculation:** |
| *The program, as a whole, runs on iterative analysis, where Fibonacci() function has a for loop, functions related to linked lists have while loop.*  *The overall time complexity of the program turns out to be O(n).* |

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
| **Modularity** |  | **Documentation** |  | **Indentation** |  | **Programming practices** |  |
| **Type of Algorithm:** *Greedy* | | | | | | | |
| **Details of the algorithm:** | | | | | | | |
| *Greedy algorithm works by finding locally optimal solutions of each part. So that we can reach to the globally optimal solution. Greedy algorithm builds up the solution piece-by piece, always choosing the next piece that offers to be the most optimal solution.*  *A very popular example of greedy algorithm is finding of minimum spanning tree. Some other applications of this algorithm are job sequencing with a deadline, shortest path, activity selection problem.* | | | | | | | |
| **Code for example 1:** | | | | | | | |
| *//Find the minimum number of currency notes/coins for the amount in the denominations (given in notes[]) by reading the amount.*  *#include<stdio.h>*  *#define MAX 100*  *int notes[]={2000,500,200,100,50,20,10,5,2,1}; //10*  *int denomination[MAX];*  *void Minimum\_change(int sum,int n)*  *{*  *for(int i=0;i<n;i++)*  *{*  *denomination[i]=0;*  *while(sum>=notes[i])*  *{*  *sum=sum-notes[i];*  *denomination[i]++;*  *}*  *}*  *display(&notes,&denomination,n);*  *}*  *void display(int\* notes, int\* denomination,int n)*  *{*  *int change=0;*  *printf("Notes \t Denomination \n");*  *for(int i=0;i<n;i++)*  *{*  *printf("%d \t %d \n",\*notes,\*denomination);*  *(notes)++;*  *(denomination)++;*  *change+=\*denomination;*  *}*  *printf("Total number of notes: %d\n",change);*  *}*  *void main()*  *{*  *int sum,n=sizeof(notes)/sizeof(notes[0]);*  *scanf("%d",&sum);*  *Minimum\_change(sum,n);*  *}* | | | | | | | |
| **Sample Input:** | | | | | | | |
| *1531* | | | | | | | |
| **Sample Output:** | | | | | | | |
| *Notes Denomination*  *2000 0*  *500 3*  *200 0*  *100 0*  *50 0*  *20 1*  *10 1*  *5 0*  *2 0*  *1 1*  *Total number of notes: 6* | | | | | | | |
| **Time complexity calculation:** | | | | | | | |
| **I***n the function Munimumchange(), there is a while loop nested inside the for loop, the condition for which is (sum>=notes[i]). Hence the complexity does not remain linear, but changes to*  *O(n^2).* | | | | | | | |

|  |
| --- |
| **Code for example 2:** |
| *// selection sort using greedy algorithm*  *#include <stdio.h>*  *#define MAX 1000*  *void swap(int \*a, int \*b)*  *{*  *int temp=\*a;*  *\*a=\*b;*  *\*b=temp;*  *}*  *void selectionSort(int arr[],int n)*  *{*  *int min\_index;*  *for(int i=0;i<n-1;i++)*  *{*  *min\_index=i;*  *for(int j=i+1;j<n;j++)*  *{*  *if(arr[j]<arr[min\_index]) //sorted the subarray*  *min\_index=j; //Ascending order*  *if(min\_index!=i)*  *swap(&arr[min\_index],&arr[i]);*  *}*  *}*  *for(int i=0;i<n;i++)*  *printf("%d ",arr[i]);*  *}*  *void read(int\* array,int n)*  *{*  *for(int i=0;i<n;i++)*  *{*  *scanf("%d",array);*  *(array)++;*  *}*  *}*  */\**  *void display(int\* arr, int n)*  *{*  *for(int i=0;i<n;i++)*  *{*  *printf("%d ",\*arr);*  *(arr)++;*  *}*  *}*  *\*/*  *void main()*  *{*  *int n,array[MAX];*  *scanf("%d",&n);*  *read(&array,n);*  *selectionSort(array,n);*  *}* |
| **Sample Input:** |
| *6*  *48 26 15 95 87 53* |
| **Sample Output:** |
| *26 48 15 53 87 95* |
| **Time complexity calculation:** |
| *The function selectionsort() contains a nested for loop. This makes the complexity of the program*  *O(n^2).* |

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
| **Modularity** |  | **Documentation** |  | **Indentation** |  | **Programming practices** |  |
| **Type of Algorithm:** *Branch and Bound* | | | | | | | |
| **Details of the algorithm:** | | | | | | | |
| *Branch and bound is a method for solving optimization problems by breaking them down into smaller sub-problems and using a bounding function to eliminate sub-problems that cannot contain the optimal solution.*  *Applications: Combinatorial optimization problems, Heuristic function construction, minimum case combination* | | | | | | | |
| **Code for example 1:** | | | | | | | |
| */\*Let there be N workers and N jobs. Any worker can be assigned to perform any job, incurring some cost that may vary depending on the work-job assignment. It is required to perform all jobs by assigning exactly one worker to each job and exactly one job to each agent in such a way that the total cost of the assignment is minimized.*  *\*/*  *#include<stdio.h>*  *#include<limits.h>*  *void main()*  *{*  *int workers,jobs,min\_cost;*  *scanf("%d",&workers);*  *scanf("%d",&jobs);*  *int cost[workers][jobs];*  *for(int i=0;i<workers;i++)*  *{*  *for(int j=0;j<jobs;j++)*  *{*  *scanf("%d",&cost[i][j]);*  *}*  *}*  *for(int i=0;i<workers;i++)*  *{*  *printf("\n");*  *for(int j=0;j<jobs;j++)*  *{*  *printf("%d\t",cost[i][j]);*  *}*  *printf("\n");*  *}*  *int fix\_index[jobs];*  *for(int m=0;m<jobs;m++)*  *fix\_index[m]=-1;*  *for(int i=0;i<workers;i++)*  *{*  *printf("\n Worker %d: ",i);*  *min\_cost=INT\_MAX;//cost[0][0];*  *for(int j=0;j<jobs;j++)*  *{*  *int k=0;*  *do*  *//for(int k=0;k<=fix\_index[i-1];k++)*  *{*  *if(j!=fix\_index[k])*  *{*  *if(cost[i][j]<min\_cost)*  *{*  *min\_cost=cost[i][j];*  *printf("Min cost = %d, at index i,j = %d,%d\n",min\_cost,i,j);*  *fix\_index[i]=j;*  *}*  *}*  *k++;*  *}while(k<=fix\_index[i-1]);*  *}*  *}*  *int total\_cost=0;*  *for(int i=0;i<workers;i++)*  *{*  *for(int j=0;j<jobs;j++)*  *{*  *if(fix\_index[j]==i)*  *{*  *printf("Min cost = %d, at index i,j = %d,%d\n",cost[i][j],i,j);*  *total\_cost+=cost[i][j];*  *}*  *}*  *}*  *printf("Total cost = %d\n",total\_cost);*  *}}* | | | | | | | |
| **Sample Input:** | | | | | | | |
| *3 3*  *10 6 8*  *5 7 9*  *10 6 1* | | | | | | | |
| **Sample Output:** | | | | | | | |
| *Minimum cost = 6, at index (I,j) = 0,1*  *Minimum cost = 5, at index (I,j) = 1,0*  *Minimum cost = 1, at index (I,j) = 2,2*  *Total cost = 12* | | | | | | | |
| **Time complexity calculation:** | | | | | | | |
| *Because of nested for loop, the time complexity turns out to be*  *O(N^2)* | | | | | | | |

|  |
| --- |
| **Code for example 2:** |
| *// traveling salesman problem*  *#include <limits.h>*  *#define MAX\_CITIES 10*  *int graph[MAX\_CITIES][MAX\_CITIES];int numCities;*  *int min(int a, int b) {*  *return (a < b) ? a : b;*  *}*  *int tsp(int mask, int currentCity)*  *{*  *if (mask == (1 << numCities) - 1)*  *{        // All cities visited, return to starting city*  *return graph[currentCity][0];*  *}*  *int minDistance = INT\_MAX;*  *for (int nextCity = 0; nextCity < numCities; nextCity++)*  *{*  *if ((mask & (1 << nextCity)) == 0)*  *{*  *int distance = graph[currentCity][nextCity] + tsp(mask | (1 << nextCity), nextCity);*  *minDistance = min(minDistance, distance);*  *}*  *}*  *return minDistance;*  *}*  *void main()*  *{*  *printf("Enter the number of cities: ");*  *scanf("%d", &numCities);*  *printf("Enter the adjacency matrix:\n");*  *for (int i = 0; i < numCities; i++)*  *{*  *for (int j = 0; j < numCities; j++)*  *{*  *scanf("%d", &graph[i][j]);*  *}*  *}*  *int startingCity = 0;*  *int initialMask = 1 << startingCity;*  *int shortestDistance = tsp(initialMask, startingCity);*  *printf("Shortest distance: %d\n", shortestDistance);*  *}* |
| **Sample Input:** |
| *3*  *1 2 3*  *4 5 6*  *7 8 9* |
| **Sample Output:** |
| *Shortest distance: 21* |
| **Time complexity calculation:** |
| *There is a for loop in tasp() function. Hence the time complexity of tsp function is O(n).*  *But to read the matrix, nested for loops are use, which makes the complexity of the program O(n^2).* |

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
| **Modularity** |  | **Documentation** |  | **Indentation** |  | **Programming practices** |  |
| **Type of Algorithm:** *Brute force* | | | | | | | |
| **Details of the algorithm:** | | | | | | | |
| *Branch and bound is a method for solving optimization problems by breaking them down into smaller sub-problems and using a bounding function to eliminate sub-problems that cannot contain the optimal solution.*  *Applications: Combinatorial optimization problems, Heuristic function construction, minimum case combination* | | | | | | | |
| **Code for example 1:** | | | | | | | |
| *//Search a string pattern in a line of strings*  *#include <stdio.h>*  *#include <string.h>*  *#define MAX 100*  *int bruteForce(char \*source, char \*find, int slen, int plen)*  *{*  *int j,k;*  *for (int i=0;i<=slen-plen;i++)*  *{*  *for (j=0,k=i;(source[k]==find[j])&&(j<plen);j++,k++);*  *if (j==plen)*  *return i;*  *}*  *return -1;*  *}*  *int main() {*  *char original[MAX], key[MAX];*  *int res;*  *printf("Enter Search String:");*  *gets(original);*  *printf("Enter Pattern String:");*  *gets(key);*  *res = bruteForce(original, key, strlen(original), strlen(key));*  *if (res == -1) {*  *printf("Search pattern is not available\n");*  *} else {*  *printf("Search pattern available at the location %d\n", res);*  *}*  *return 0;*  *}* | | | | | | | |
| **Sample Input:** | | | | | | | |
| *This is a sample statement to search*  *ment* | | | | | | | |
| **Sample Output:** | | | | | | | |
| *Search pattern available at the location 22* | | | | | | | |
| **Time complexity calculation:** | | | | | | | |
| *Since there is a while loop in a for loop, the complexity of outer for loop is O(n), and that of while loop is O(m).*  *Hence complexity is O(mn)* | | | | | | | |

|  |
| --- |
| **Code for example 2:** |
| *//Linear search*  *#include<stdio.h>*  *void read(int array[],int n)*  *{*  *for(int i=0;i<n;i++)*  *{*  *scanf("%d",&array[i]);*  *}*  *}*  *void linear\_search(int array[],int n,int key)*  *{*  *int count=0;*  *for(int i=0;i<n;i++)*  *{*  *if(array[i]==key)*  *{*  *printf("%d found at index %d\n",key,i);*  *count++;*  *}*  *}*  *if(count==0)*  *printf("%d is not present in the array\n",key);*  *else*  *printf("%d is present %d times in the array",key,count);*  *}*  *void main()*  *{*  *int n,key;*  *scanf("%d",&n);*  *int array[n];*  *read(array,n);*  *scanf("%d",&key);*  *linear\_search(array,n,key);*  *}* |
| **Sample Input:** |
| *8*  *54 89 65 32 53 48 15 68*  *53* |
| **Sample Output:** |
| *53 found at index 4*  *53 is present 1 times in the array* |
| **Time complexity calculation:** |
| *Linearsearch() function contains a single for loop.*  *The complexity of the program is O(n)* |

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
| **Modularity** |  | **Documentation** |  | **Indentation** |  | **Programming practices** |  |
| **Type of Algorithm:** *Randomized* | | | | | | | |
| **Details of the algorithm:** | | | | | | | |
| *An algorithm that uses random numbers to decide what to do next anywhere in its logic is called Randomized Algorithm.* | | | | | | | |
| **Code for example 1:** | | | | | | | |
| *//Random prime numbers*  *#include <stdio.h>*  *#include<time.h>*  *int isprime(int n)*  *{*  *if(n % 2 == 0 || n == 1)*  *return 0;*  *for(int i = 3; i\*i <= n; i += 2) {*  *if(n % i == 0)*  *return 0;*  *}*  *return 1;*  *}*  *int randPrime(void)*  *{*  *int r;*  *while(!isprime((r = rand())));*  *return r;*  *}*  *int main(int argc, char \*\*argv)*  *{*  *int i;*  *srand(time(0));*  *for(i = 0; i < 10; i++) {*  *printf("%d\n", randPrime());*  *}*  *return 0;*  *}* | | | | | | | |
| **Sample Input:** | | | | | | | |
| *-* | | | | | | | |
| **Sample Output:** | | | | | | | |
| *28219*  *10607*  *9533*  *23671*  *13757*  *25321*  *1103*  *18731*  *14633*  *32059* | | | | | | | |
| **Time complexity calculation:** | | | | | | | |
| *Because of single for loop and single while loop, the time complexity turns out to be*  *O(N)* | | | | | | | |