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

**Term-work Report**

|  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| **Term-work** | *01* | | | | |  |  | | | | |
| **Student Name** | Harsh A Patil | | | | |  |  | | | | |
| **SRN** | 02FE21BEC035 | | | | **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 programs for each algorithm type, and express the time complexity of each program.   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** | | Queue | | | | | O(n) | | | |
| Recursive | **1** | | Array | | | | | O(n2) | | | |
| **2** | | Array | | | | | O(n) | | | |
| Back tracking | **1** | | 2D Array | | | | | O(9n) | | | |
| **2** | | Array | | | | | O(2n) | | | |
| Divide and conquer | **1** | | Array | | | | | O(nlog) | | | |
| **2** | | Array | | | | | O(n logn) | | | |
| Dynamic programming | **1** | | Single Linked List | | | | | O(n) | | | |
| **2** | | Array | | | | | O(n) | | | |
| Greedy | **1** | | Array | | | | | O(n2) | | | |
| **2** | | Array | | | | | O(n2) | | | |
| Branch and bound | **1** | | 2D Array | | | | | O(n2) | | | |
| **2** | | 2D Array | | | | | O(n2) | | | |
| Brute force | **1** | | Array | | | | | O(n) | | | |
| **2** | | Array | | | | | O(mn) | | | |
| Randomized | **1** | | Array | | | | | O(n) | | | |
| **2** | | Array | | | | | O(n) | | | |

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.* | | | | | | | |
| **Code for example 1:** | | | | | | | |
| *Input format:*  *First line contains number of array elements to be entered*  *Subsequent lines contain the array elements*  *Last line is the key element*  *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;*  *}*  *void main()*  *{*  *int array[MAX];*  *int n,key;*  *scanf("%d",&n);*  *read(array,n);*  *scanf("%d",&key);*  *int index=search(array,n,key);*  *printf("%d\n",index);*  *}* | | | | | | | |
| **Sample Input and Output:** | | | | | | | |
|  | | | | | | | |
| **Time complexity calculation:** | | | | | | | |
| First for loop = 2n+2 times  Int j,found = 1 times  Second for loop = 2n+2 times  Total = 4n+5  Hence , Time complexity : O(n) | | | | | | | |

|  |
| --- |
| **Code for example 2:** |
| *#include <stdio.h>*  *#define size 5 // size of the queue*  *int arr[size];*  *int front = 0;*  *int rear = 0;*  *int isQueueEmpty() {*  *if (front == rear)*  *return 1;*  *else*  *return -1;*  *}*  *void dequeue() {*  *if (isQueueEmpty() == 1)*  *printf("Queue is Empty.\n");*  *else {*  *printf("Dequeued element = %d\n", arr[front]);*  *front++;*  *}*  *}*  *int isQueueFull() {*  *if (rear == size)*  *return 1;*  *else*  *return -1;*  *}*  *void enqueue(int val) {*  *if (isQueueFull() == 1)*  *printf("Queue is Full\n");*  *else {*  *arr[rear] = val;*  *rear++;*  *}*  *}*  *void printQueue() {*  *if (isQueueEmpty() == 1) {*  *printf("Queue is Empty.\n");*  *} else {*  *printf("Queue elements: ");*  *for (int i = front; i < rear; i++) {*  *printf("%d ", arr[i]);*  *}*  *printf("\n");*  *}*  *}*  *int main() {*  *int choice, value;*  *while (1) {*  *printf("\nQueue Operations:\n");*  *printf("1. Enqueue\n");*  *printf("2. Dequeue\n");*  *printf("3. Print Queue\n");*  *printf("4. Exit\n");*  *printf("Enter your choice: ");*  *scanf("%d", &choice);*  *switch (choice) {*  *case 1:*  *printf("Enter the value to enqueue: ");*  *scanf("%d", &value);*  *enqueue(value);*  *break;*  *case 2:*  *dequeue();*  *break;*  *case 3:*  *printQueue();*  *break;*  *case 4:*  *printf("Exited.\n");*  *return 0;*  *default:*  *printf("Invalid choice.\n");*  *}*  *}*  *return 0;*  *}* |
| **Sample Input and Output:** |
|  |
| **Time complexity calculation:** |
| *Time Complexity of enqueue and dequeue turns out to be O(n).* |

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
| **Modularity** |  | **Documentation** |  | **Indentation** |  | **Programming practices** |  |
| **Type of Algorithm:** *Recursive* | | | | | | | |
| **Details of the algorithm:** | | | | | | | |
| *Recursion in simple terms means 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 for recursion.* | | | | | | | |
| **Code for example 1:** | | | | | | | |
| Eg 1: Bubble Sort :  #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)  {  printf("enter the array elements:\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;  printf("enter the size:");  scanf("%d",&n);  read(array,n);  bubblesort(array, n);  printf("Sorted array : \n");  display(array, n);  } | | | | | | | |
| **Sample Input and Output:** | | | | | | | |
|  | | | | | | | |
| **Time complexity calculation:** | | | | | | | |
| The bubblesort function implements the Bubble Sort algorithm, which has a worst-case and average time complexity of O(n2).  The for loop runs for 2n+2 times.  Therefore, the overall time complexity of the code is O(n2). | | | | | | | |

|  |
| --- |
| **Code for example 2:** |
| //Find largest of Number in array  #include <stdio.h>  int large(int[], int, int);  int main()  {  int size;  int largest;  int list[20];  int i;  printf("Enter size of the list:");  scanf("%d", &size);  printf("Enter the elements of the list:\n");  for (i = 0; i < size; i++)  {  scanf("%d", &list[i]);  }  if (size == 0)  {  printf("Empty list\n");  }  else  {  largest = list[0];  largest = large(list, size - 1, largest);  printf("\nThe largest number in the list is: %d\n", largest);  }  }  int large(int list[], int position, int largest)  {  if (position == 0)  return largest;  if (position > 0)  {  if (list[position] > largest)  {  largest = list[position];  }  return large(list, position - 1, largest);  }  } |
| **Sample Input and Output:** |
|  |
| **Time complexity calculation:** |
| *Time complexity of the code is O(n).* |

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
| **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, crossword, 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 and Output:** | | | | | | | |
|  | | | | | | | |
| **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 O(9n). | | | | | | | |

|  |
| --- |
| **Code for example 2:** |
| //Subset Sum Problem  #include <stdio.h>  void printSubset(int subset[], int size) {  printf("Subset: ");  for (int i = 0; i < size; i++) {  printf("%d ", subset[i]);  }  printf("\n");  }  void subsetSumUtil(int set[], int subset[], int n, int sum, int index, int subsetSize) {  if (sum == 0) {  printSubset(subset, subsetSize);  return;  }  if (index >= n || sum < 0) {  return;  }  subset[subsetSize] = set[index];  subsetSumUtil(set, subset, n, sum - set[index], index + 1, subsetSize + 1);  subsetSumUtil(set, subset, n, sum, index + 1, subsetSize);  }  void subsetSum(int set[], int n, int sum) {  int subset[n];  subsetSumUtil(set, subset, n, sum, 0, 0);  }  int main() {  int n, sum;  printf("Enter the size of the set: ");  scanf("%d", &n);  int set[n];  printf("Enter the elements of the set:\n");  for (int i = 0; i < n; i++) {  scanf("%d", &set[i]);  }  printf("Enter the desired sum: ");  scanf("%d", &sum);  subsetSum(set, n, sum);  return 0;  } |
| **Sample Input and Output:** |
|  |
| **Time complexity calculation:** |
| *The number of possible subsets of a set with size n is 2n. Therefore, the code has a time complexity of O(2n).* |

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
| **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 chunks to solve it, the name is ‘Divide and Conquer’.*  *Eg. Binary search, or in merge sort, quick sort etc.* | | | | | | | |
| **Code for example 1:** | | | | | | | |
| //Binary Search  #include<stdio.h>  #define MAX 1000  void read(int array[],int n)  {  printf("enter the array elements:\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;  printf("enter the size:");  scanf("%d",&n);  read(array,n);  printf("enter the key to be found:");  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 and Output:** | | | | | | | |
|  | | | | | | | |
| **Time complexity calculation:** | | | | | | | |
| The area of searching the key elements gets reduced to half of 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:** |
| //Merge Sort  #include <stdio.h>  void merge(int arr[], int left[], int leftSize, int right[], int rightSize) {  int i = 0, j = 0, k = 0;  // Compare elements from both subarrays and merge them in sorted order  while (i < leftSize && j < rightSize) {  if (left[i] <= right[j]) {  arr[k] = left[i];  i++;  } else {  arr[k] = right[j];  j++;  }  k++;  }  // Copy the remaining elements of the left subarray, if any  while (i < leftSize) {  arr[k] = left[i];  i++;  k++;  }  // Copy the remaining elements of the right subarray, if any  while (j < rightSize) {  arr[k] = right[j];  j++;  k++;  }  }  void mergeSort(int arr[], int size) {  if (size < 2) {  return; // Base case: Array is already sorted or empty  }  int mid = size / 2;  // Create left and right subarrays  int left[mid];  int right[size - mid];  // Copy elements to left and right subarrays  for (int i = 0; i < mid; i++) {  left[i] = arr[i];  }  for (int i = mid; i < size; i++) {  right[i - mid] = arr[i];  }  // Recursively sort the left and right subarrays  mergeSort(left, mid);  mergeSort(right, size - mid);  merge(arr, left, mid, right, size - mid); // Merge the sorted subarrays  }  void printArray(int arr[], int size) {  for (int i = 0; i < size; i++) {  printf("%d ", arr[i]);  }  printf("\n");  }  int main() {  int size;  printf("Enter the size of the array: ");  scanf("%d", &size);  int arr[size];  printf("Enter the elements of the array: ");  for (int i = 0; i < size; i++) {  scanf("%d", &arr[i]);  }  mergeSort(arr, size);  printf("Sorted array: ");  printArray(arr, size);  return 0;  } |
| **Sample Input and Output:** |
|  |
| **Time complexity calculation:** |
| *The merging step, where two sorted subarrays are combined, takes O(n) time. The division step occurs recursively, halving the array size at each level, which results in a logarithmic number of levels.*  *Therefore, the overall time complexity of Merge Sort is O(n log n).* |

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
| **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:** | | | | | | | |
| //Finding nth element in Fibonacci Series using Dynamic Programming  #include<stdio.h>  #include<stdlib.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;  printf("enter the size:");  scanf("%d",&n);  nth\_fib=fibonacci(n);  display\_SLL(head);  printf("\nnth element is:%d\n", nth\_fib);  } | | | | | | | |
| **Sample input and Output:** | | | | | | | |
|  | | | | | | | |
| **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). | | | | | | | |

|  |
| --- |
| **Code for example 2:** |
| /\*Golomb sequence is a sequence where n-th term is equal to number of times n appears in the sequence.  \*/  #include<stdio.h>  #include<stdlib.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;  printf("enter the size:");  scanf("%d",&n);  Generate\_Golomb(n);  } |
| **Sample Input and Output:** |
|  |
| **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)* |

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
| **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]);  printf("enter the amount:");  scanf("%d",&sum);  Minimum\_change(sum,n);  } | | | | | | | |
| **Sample Input and Output:** | | | | | | | |
|  | | | | | | | |
| **Time complexity calculation:** | | | | | | | |
| In the function Minimumchange(), 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(n2). | | | | | | | |

|  |
| --- |
| **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]);  }  }  printf("Sorted Array:");  for(int i=0;i<n;i++)  printf("%d ",arr[i]);  }  void read(int\* array,int n)  {  printf("enter array elements:");  for(int i=0;i<n;i++)  {  scanf("%d",array);  (array)++;  }  }  void main()  {  int n,array[MAX];  printf("enter size of array:");  scanf("%d",&n);  read(&array,n);  selectionSort(array,n);  } |
| **Sample Input and Output:** |
|  |
| **Time complexity calculation:** |
| *The function selectionsort() contains a nested for loop. This makes the complexity of the program*  *O(n2).* |

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
| **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:** | | | | | | | |
| // min cost of workers  #include<stdio.h>  #include<limits.h>  void main()  {  int workers,jobs,min\_cost;  printf("enter the Number of workers:");  scanf("%d",&workers);  printf("enter the number of jobs:");  scanf("%d",&jobs);  printf("enter the cost matrix:\n");  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  {  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 and Output:** | | | | | | | |
|  | | | | | | | |
| **Time complexity calculation:** | | | | | | | |
| Because of nested for loop, the time complexity turns out to be  O(n2). | | | | | | | |

|  |
| --- |
| **Code for example 2:** |
| //Travelling Salesman  #include <stdio.h>  #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;  }  int 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);  return 0;  } |
| **Sample Input and Output:** |
|  |
| **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(n2).* |

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
| **Modularity** |  | **Documentation** |  | **Indentation** |  | **Programming practices** |  |
| **Type of Algorithm: Brute Force** | | | | | | | |
| **Details of the algorithm:** | | | | | | | |
| *The brute force algorithm, also known as the exhaustive search algorithm, is a straightforward approach that systematically checks all possible solutions to a problem in order to find the optimal solution. The name "brute force" refers to the idea of using sheer computational power to try all possible solutions without employing any specific optimization techniques.*  *Brute force algorithms are used in various applications and scenarios. Some of the applications include Combinatorial problems,Cryptanalysis,Exhaustive search problems etc* | | | | | | | |
| **Code for example 1:** | | | | | | | |
| //Linear search  #include<stdio.h>  void read(int array[],int n)  { printf("enter the array elements:");  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;  printf("enter the size:");  scanf("%d",&n);  int array[n];  read(array,n);  printf("enter the key:");  scanf("%d",&key);  linear\_search(array,n,key);  } | | | | | | | |
| **Sample Input and Output:** | | | | | | | |
|  | | | | | | | |
| **Time complexity calculation:** | | | | | | | |
| Linearsearch() function contains a single for loop.  The complexity of the program is O(n) | | | | | | | |

|  |
| --- |
| **Code for example 2:** |
| //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 and Output:** |
|  |
| **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). |

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
| **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:** | | | | | | | |
| // Generating random numbers in range  #include <stdio.h>  #include <stdlib.h>  #include <time.h>  void random(int n, int p, int q);  int main() {  int n, p, q;  printf("Enter the number of random numbers to generate: ");  scanf("%d", &n);  printf("Enter the lower bound (P): ");  scanf("%d", &p);  printf("Enter the upper bound (Q): ");  scanf("%d", &q);  random(n, p, q);  return 0;  }  void random(int n, int p, int q) {  int i, num;  srand(time(NULL));  for (i = 0; i < n; i++) {  num = rand() % (q - p + 1) + p;  printf( "%d\n", num);  }  } | | | | | | | |
| **Sample Input and Output:** | | | | | | | |
|  | | | | | | | |
| **Time complexity calculation:** | | | | | | | |
| The loop runs for n iterations, generating one random number in each iteration. Therefore, the time complexity is linearly proportional to the number of random numbers to generate, which is O(n). | | | | | | | |

|  |
| --- |
| **Code for example 2:** |
| *// C program for the above approach*  *#include <math.h>*  *#include <stdio.h>*  *#include <stdlib.h>*  *#include <time.h>*    *// Function to randomly generates password*  *// of length N*  *void randomPasswordGeneration(int N)*  *{*  *// Initialize counter*  *int i = 0;*    *int randomizer = 0;*    *// Seed the random-number generator*  *// with current time so that the*  *// numbers will be different every time*  *srand((unsigned int)(time(NULL)));*    *// Array of numbers*  *char numbers[] = "0123456789";*    *// Array of small alphabets*  *char letter[] = "abcdefghijklmnoqprstuvwyzx";*    *// Array of capital alphabets*  *char LETTER[] = "ABCDEFGHIJKLMNOQPRSTUYWVZX";*    *// Array of all the special symbols*  *char symbols[] = "!@#$^&\*?";*    *// Stores the random password*  *char password[N];*    *// To select the randomizer*  *// inside the loop*  *randomizer = rand() % 4;*    *// Iterate over the range [0, N]*  *for (i = 0; i < N; i++) {*    *if (randomizer == 1) {*  *password[i] = numbers[rand() % 10];*  *randomizer = rand() % 4;*  *printf("%c", password[i]);*  *}*  *else if (randomizer == 2) {*  *password[i] = symbols[rand() % 8];*  *randomizer = rand() % 4;*  *printf("%c", password[i]);*  *}*  *else if (randomizer == 3) {*  *password[i] = LETTER[rand() % 26];*  *randomizer = rand() % 4;*  *printf("%c", password[i]);*  *}*  *else {*  *password[i] = letter[rand() % 26];*  *randomizer = rand() % 4;*  *printf("%c", password[i]);*  *}*  *}*  *}*    *// Driver Code*  *int main()*  *{*  *// Length of the password to*  *// be generated*  *int N = 10;*    *// Function Call*  *randomPasswordGeneration(N);*    *return 0;*  *}* |
| **Sample Input and Output:** |
|  |
| **Time complexity calculation:** |
| *The loop runs N times, generating one character of the password in each iteration.*  *Therefore, the time complexity is linearly proportional to the length of the password to be generated, which is O(n).* |