Data Structures Applications Lab (21EECF201) [0-0-2]
Term-work Report

Term-work	01				
Student Name	Abhijeet Jadhav				
SRN	01FE21BEI045	Roll Number	138	Division	A-II

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,
6. 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	0(n)
	2	Linked list	$0(n^2)$
Recursive	1	Array	0(n)
	2	-	0(2^n)
Back tracking	1	Array	0(k+m-n)
	2	-	0(n!)
Divide and conquer	1	Array	0(logn)
	2	Array	0(nlogn)
Dynamic programming	1	Array	0(n)
	2	Array	0(n+k)
Greedy	1	Array	0(x*n)
	2	-	0(log(min(l,b,h))
Branch and bound	1	Array	0(n)
	2	Array	0(n)
Brute force	1	Array	0(k)
	2	-	--
Randomized	1	Array	0(n)
	2	Array	0(n)

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

Yes, absolutely the problems that I discovered and learned were very much exciting and were a challenge to myself the improve in DSA

What assistance do you need to learn this term work better?
Assistance was the website where I got interesting questions over the topic and the discussion of various solution with friends.

What are the areas you think you should work on to be able to make this solution better?
Over recursion I guess the difficulty was to trace and come up with a solution which would take a considerable amount of time

Modularity		Documentation		Indentation		Programming practices	
Type of Algorithm: Iterative							
Details of the algorithm:							

An iterative algorithm is a computational approach that solves a problem or finds an solution by using iteration, or the repetition of a sequence of stages

Applications:
1. Numerical Analysis: solving linear equations, numerical integration, and differential equations.
2. Optimisation: employed in optimisation issues to identify the best approach or minimize/maximize a specific objective function.

Steps involved in algorithm
1. Initialization of a variable
2. Setting a terminating point where the iteration should end
3. Update the initialized variable.
4. Check for the terminating point.
5. Check the output.

Code for example 1:
Q. Find the factorial of number?

```
include <stdio.h>

void factorial(int n)
{
   int product = 1;
   for (int i = 1; i <= n; i++)
   {
      product = product * i; // multiplying the product in a loop
   }
   printf("Output:\n");
   printf("%d", product); // printing the output
}

void main()
{
   int n;
   printf("Enter number:\n");
   scanf("%d", &n); // Entry of number
   factorial(n);    // function call
}
```

Sample Input:

```
Enter Number:
5
```

Sample Output:

```
Output:
120
```

Time complexity calculation:

```
void factorial(int n)
{
   int product = 1;    ->1
   for (int i = 1; i <= n; i++)    ->n+1
   {
      product = product * i;    ->n
   }
   printf("Output:\n");    ->1
   printf("%d", product);    ->1
}
```

Time =1+n+1+n+1+1
        =2n+4
Time complexity =0(n)

Code for example 2:
Q. bubble sort

```
#include <stdio.h>
#include <time.h>
#include <stdlib.h>
struct node
{
   int data;
   struct node *link;
};
struct node *cur, *prev;
struct node *newnode;

struct node *create_node(int n)
{
   newnode = (struct node *)malloc(sizeof(struct node));
   if (newnode == NULL)
   {
      printf("NO memory allocated");
   }
   newnode->data = n;
   newnode->link = NULL;
   return newnode;
}
struct node *insert_end(struct node *head, int n)
{
   struct node *cur;
   newnode = create_node(n);
   if (head == NULL)
   {
      head = newnode;
   }
   else
   {
      cur = head;
      while (cur->link != NULL)
      {
         cur = cur->link;
      }
      cur->link = newnode;
   }
   return head;
}

void display(struct node *head)
{
   struct node *cur;
   if (head == NULL)
   {
      printf("List empty\n");
   }
   else
   {
      cur = head;
      while (cur != NULL)
      {
            printf("%d->", cur->data);
            cur = cur->link;
        }
    }
}

void bubble(struct node *head, int n)
{
    cur = head;
    prev = NULL;
    for (int i = 0; i < n; i++)
    {
        while (cur->link != NULL)
        {
            prev = cur;
            cur = cur->link;
            if (cur->data < prev->data)
            {
                int temp = prev->data;
                prev->data = cur->data;
                cur->data = temp;
            }
        }
        cur = head;
        prev = NULL;
    }
}

struct node *list_make(FILE *fp, int n, struct node *head)
{
    int x;
    for (int i = 0; i < n; i++)
    {
        fscanf(fp, "%d", &x);
        head = insert_end(head, x);
    }
    return head;
}

void random(FILE *fp, int n, int l, int h)
{
    int r;
    srand(time(NULL));
    for (int i = 0; i < n; i++)
    {
        r = rand() % (h - l) + l;
        fprintf(fp, "%d ", r);
    }
    rewind(fp);
}

int main()
{
    struct node *head = NULL;
    int n, l, h;
    FILE *fp;
    fp = fopen("List.txt", "w+");
```

```
   printf("Enter how many numbers:\n");
   scanf("%d", &n);
   printf("Enter range l,H\n");
   scanf("%d %d", &l, &h);
   random(fp, n, l, h);
   head = list_make(fp, n, head);
   display(head);
   bubble(head, n);
   printf("\n");
   display(head);
}
```

Sample Input:

7 4 1 2 5 8 9 6 3

Sample Output:

1 2 3 4 5 6 7 8 9

Time complexity calculation:

```
void bubble(struct node *head, int n)
{
   cur = head;
   prev = NULL;
   for (int i = 0; i < n; i++)
   {
      while (cur->link != NULL)
      {
         prev = cur;
         cur = cur->link;
         if (cur->data < prev->data)
         {
            int temp = prev->data;
            prev->data = cur->data;
            cur->data = temp;
         }
      }
      cur = head;
      prev = NULL;
   }
}
```

Time complexity =0(n^2)

Modularity		Documentation		Indentation		Programming practices	

Type of Algorithm: Recursion

Details of the algorithm:

Recursion is a programming technique where a function calls itself, either directly or indirectly, to solve a problem. It offers an alternative and often more effective approach compared to iteration.

The process of recursion involves a function invoking itself with different parameter values. This allows the function to break down a complex problem into smaller, similar subproblems, eventually leading to a solution.

Application:
1. Mathematical calculation
2. Tree and graph traversal
3. Divide and conquer algorithm

Steps involved in recursion:
1. Define the function with required parameters.
2. Call the same function through recursive call.
3. Find the terminating condition for the function sot that the recursion terminates

Code for example 1:
Q. Finding Maximum number in array?

```
#include <stdio.h>

int max_number(int arr[], int n)
{
   if (n < 0)
   {
      return 0; // return 0 if arr index is less then 0
   }
   int max = max_number(arr, n - 1); // calling own same function
   if (arr[n] > max)              // checking for max value
   {
      max = arr[n]; // changing max value
   }
   return max;
}

void main()
{
   int arr[] = {2, 4, 1222, 34, 1, 56, 0, 465, 4};
   int n = sizeof(arr) / sizeof(int);
   int max = max_number(arr, n - 1);            // calling function
   printf("Maximum number in array is:%d\n", max); // output
}
```

Sample Input:

{2, 4, 122, 34, 1, 56, 0, 465, 4}

Sample Output:

```
Maximum number in array is:465
```

Time complexity calculation:

```
int max_number(int arr[], int n)   ->T(n)
{
   if (n < 0)    ->1
   {
        return 0;
    }
    int max = max_number(arr, n - 1);   ->T(n-1)
    if (arr[n] > max)                ->1
    {
        max = arr[n];               ->1
    }
    return max;                      ->1
}
Total Time =T(n)=T(n-1)+4
T(n)=T(n-2)+4+4
.
T(n)=T(n-k)+4k
If n-k=0(base case)
n=k
time complextity=0(n)
```

Code for example 2:
Q. Fibonacci series?

```
#include <stdio.h>

int fibonacci_series(int n)
{
    if (n == 0)
    {
        return 0;
    }
    if (n == 1)
    {
        return 1;
    }
    return fibonacci_series(n - 1) + fibonacci_series(n - 2); // calling the same function(recursion)
}

void main()
{
    int n;
    printf("Enter number for Fibonacci series:\n");
    scanf("%d", &n);               // input
    n = fibonacci_series(n);       // function call
    printf("The output is:%d\n", n); // output
}
```

Sample Input:

```
Enter number for Fibonacci series:
5
```

Sample Output:
The output is: 3
Time complexity calculation:

```
int fibonacci_series(int n)
{
   if (n == 0)   ->1
   {
      return 0;
   }
   if (n == 1)  ->1
   {
      return 1;
   }
   return fibonacci_series(n - 1) + fibonacci_series(n - 2);   ->T(n-1)+T(n-2)
}
```

Total time =T(n)=T(n-1)+T(n-2)+2
T(n)=T(n-2)+T(n-3)+4
.
T(n)=T(n-k)+T(n-k+1)+2k
Time complexity=0(2^n)

Modularity		Documentation		Indentation		Programming practices	

Type of Algorithm: Back Tracing

Details of the algorithm:

Backing is a algorithmic method to find every possible combination using recursion to solve the required problem. It finds the solution one piece at a time by removing the solution which cannot satisfy the condition of the questions

Applications of backtracking:

1. Its used in decision making problems.
2. Its also used for optimization of the problem.
3. It also used for enumeration type problem where all the feasible solutions are required.

Code for example 1:
Q. Finding subset of a set

```
#include <stdio.h>
#include <time.h>

void rec(int a[], int b[], int m, int n, int k)
{
   for (int i = 0; i < k; i++)
   {
      printf("%d ", b[i]);
   }
   printf("\n");
   for (int i = m; i < n; i++)
   {
      b[k] = a[i];
      rec(a, b, i + 1, n, k + 1);
   }
}

int main()
{
   int a[4];
   printf("how many numbers in array:\n");
   int n;
   scanf("%d", &n);
   printf("Enter array elemnts:\n");
   for (int i = 0; i < n; i++)
   {
      scanf("%d", &a[i]);
   }
   int b[n];
   printf("subsets:\n");
   rec(a, b, 0, n, 0);
}
```

Sample Input:

```
How many numbers in array:
3
Enter array elements:
1 2 3
```

Sample Output:

```
1
1 2
2
```

Time complexity calculation:

```
void rec(int a[], int b[], int m, int n, int k)
{
    for (int i = 0; i < k; i++)  ->k+1
    {
        printf("%d ", b[i]);    ->k
    }
    printf("\n");
    for (int i = m; i < n; i++)    ->n+1
    {
        b[k] = a[i];                        ->n
        rec(a, b, i + 1, n, k + 1);    ->T(n-1)
    }
}
```

Total time =1+1+k+0(n-m)+0(1)*(n-m)
0(k)+0(n-m)
Time complexity =0(k+m-n)

Code for example 2:
Q. permutation of strings

```
#include <stdio.h>
#include <string.h>

void swap(char *x, char *y)
{
    char temp;
    temp = *x;
    *x = *y;
    *y = temp;
}

void per(char *a, int l, int r)
{
    int i;
    if (l == r)
        printf("%s\n", a);
    else
    {
        for (i = l; i <= r; i++)
        {
            swap((a + l), (a + i));
            per(a, l + 1, r);
            swap((a + l), (a + i));
        }
    }
}
int main()
{
    char str[100];
    printf("Enter string:\n");
    gets(str);
    int n = strlen(str);
    per(str, 0, n - 1);
    return 0;
}
```

Sample Input:
Enter string: DSA
Sample Output:
DSA DAS SDA SAD ASD ADS
Time complexity calculation:
T(n)=n*T(n-1) T(n)=n*(n-1)*(n-2)…..2*1 =n!

Modularity		Documentation		Indentation		Programming practices	
Type of Algorithm: Divide and Conquer							
Details of the algorithm:							
Divide and conquer approach is a problem solving algorithm where we divide the problem into smaller sub problems and solve them independently then combine their solution to obtain the final result.							

It follows recursive approach where the problem is recursively divided into sub problems until they can be solved directly

Applications:
1.Sorting Algorithms- Merge sort, Quick Sort.
2. Searching Algorithms- Binary Search.
3. Numerical Algorithms-Matrix Multiplication

It involves 3 main steps
Step 1. Divide: the problem is divided into smaller and more manageable sub problems.
Step 2. Conquer: Then Problem is solved independently which can be easily manageable
Step 3. Combine: The solutions to the sub problems are then combine to form the required solution.

Code for example 1:
Q. binary search

```
#include <stdio.h>
#include <stdlib.h>

void sort(int a[], int n)
{
    for (int i = 0; i < n; i++)
    {
        for (int j = 0; j < n - i - 1; j++)
        {
            if (a[j] > a[j + 1])
            {
                int temp = a[j];
                a[j] = a[j + 1];
                a[j + 1] = temp;
            }
        }
    }
}

void binary_search(int arr[], int l, int h, int k)
{
    if (l == h)
    {
        printf("Unsuccesfull");
    }
    int mid = (h + l) / 2;
    if (arr[mid] == k)
    {
        printf("Successfull");
        exit(0);
    }
    if (arr[mid] > k)
    {
        h = mid - 1;
    }
    else
    {
        l = mid + 1;
    }
    binary_search(arr, l, h, k);
}
```

```
void main()
{
   int arr[] = {2, 1222, 34, 1, 56, 0, 465, 4};
   int n = sizeof(arr) / sizeof(int);
   sort(arr, n);
   printf("Enter the element u wanrt to search:\n");
   int k;
   scanf("%d", &k);
   binary_search(arr, 0, n - 1, k);
}
```

Sample Input:

```
Enter the element u want to search:
4
```

Sample Output:

```
Successful
```

Time complexity calculation:

```
void binary_search(int arr[], int l, int h, int k)
{
   if (l == h)
   {
      printf("Unsuccesfull");
   }
   int mid = (h + l) / 2;
   if (arr[mid] == k)
   {
      printf("Successfull");
      exit(0);
   }
   if (arr[mid] > k)
   {
      h = mid - 1;
   }
   else
   {
      l = mid + 1;
   }
   binary_search(arr, l, h, k);  ->T(n/2)
}
```

Total time =T(n)=T(n/2)+0(1)
Using master's theorem
Time complexity=0(logn)

Code for example 2:
Merge sort

```
#include <stdio.h>
void disp(int a[], int n)
{
   for (int i = 0; i < n; i++)
   {
       printf("%d ", a[i]);
   }
   printf("\n");
}

void merge(int arr[], int l, int m, int r)
{
   int i, j, k;
   int n1 = m - l + 1;
   int n2 = r - m;

   int L[n1], R[n2];

   for (i = 0; i < n1; i++)
       L[i] = arr[l + i];

   for (j = 0; j < n2; j++)
       R[j] = arr[m + 1 + j];

   i = 0;
   j = 0;
   k = l;

   while (i < n1 && j < n2)
   {
       if (L[i] <= R[j])
       {
           arr[k] = L[i];
           i++;
       }
       else
       {
           arr[k] = R[j];
           j++;
       }
       k++;
   }

   while (i < n1)
   {
       arr[k] = L[i];
       i++;
       k++;
   }

   while (j < n2)
   {
       arr[k] = R[j];
       j++;
       k++;
   }
}

void mergeSort(int arr[], int l, int r)
{
   if (l < r)
   {
       int m = l + (r - l) / 2;
       mergeSort(arr, l, m);
       mergeSort(arr, m + 1, r);
       merge(arr, l, m, r);
   }
}

int main()
{
   int arr[] = {2, 4, 6, 32, 57, 782, 4, 0, 34, 4};
   int n = sizeof(arr) / sizeof(int);
   printf("Original array: ");
   disp(arr, n);
   mergeSort(arr, 0, n - 1);
   printf("Sorted array: ");
   disp(arr, n);
   return 0;
}
```

Sample Input:

```
2, 4, 6, 32, 57, 782, 4, 0, 34, 4
```

Sample Output:

```
0, 2, 4, 4, 4, 6, 32, 34, 57, 782,
```

Time complexity calculation:

```
void mergeSort(int arr[], int l, int r)
{
    if (l < r)
    {
        int m = l + (r - l) / 2;          ->1
        mergeSort(arr, l, m);             ->T(n/2)
        mergeSort(arr, m + 1, r);         ->T(n/2)
        merge(arr, l, m, r);              ->n
    }
}
```

T(n)=2T(n/2) +0(n)
Using master's theorem
T(n)=aT(n/b)+0((n^k)*(logn^p))
$T(n) = 0(n\^(\log_b a))*\log^{P+1} n)$
=0(nlogn)

Modularity		Documentation		Indentation		Programming practices	

Type of Algorithm: Dynamic Programming

Details of the algorithm:

This algorithm uses the previously computed values to find the next desired output so that its does not need to compute the required value again. It reduces the time required to find the solution and thereby reducing the usage of time and space.

It is far better suited then recursion where recursion calculates the values again and waste precious time.

It stores the results of the sub problems and avoid recompilation.

It is called "dynamic programming" to create a sense of mathematical optimization.

Applications:
1.Combinatorial Optimization: Travelling Salesmen Problem, Knapsack problem ,Coin change.
2. Resource Allocation: Job Assignment.
3. Pathfinding and Routing: Dijkstra's algorithm.
4. Economics and Finance: to optimize investment strategies, portfolio management etc.

Steps:
1.Break down the problem into smaller sub-problems.
2.Solve the sub-problems and store the results
3.Use the stored results for further computations if required to obtain the final desired output.

Code for example 1:
Q.Fibonacci series

```
#include <stdio.h>

int fibo(int a[], int n)
{
   for (int i = 2; i < n; i++)
   {
      a[i] = a[i - 1] + a[i - 2];
   }
}

void main()
{
   int n;
   printf("Enter number for Fibonacci:\n");
   scanf("%d", &n);
   int a[n];
   a[0] = 0;
   a[1] = 1;
   fibo(a, n);
   for (int i = 0; i < n; i++)
   {
      printf("%d ", a[i]);
   }
}
```

Sample Input:

```
Enter number for Fibonacci :
5
```

Sample Output:

```
0 1 1 2 3
```

Time complexity calculation:

```
int fibo(int a[], int n)
{
   for (int i = 2; i < n; i++)   ->n-1
   {
      a[i] = a[i - 1] + a[i - 2];  ->n-2
   }
}
Total time=n-1+n-2
Time complexity=0(n)
```

Code for example 2:
Combination(nCr)

```
#include <stdio.h>

int min(int i, int k)
{
   if (i < k)
   {
      return i;
   }
   return k;
}

int bin(int n, int k)
{
   int C[k + 1];
   for (int i = 1; i < k + 1; i++)
   {
      C[i] = 0;
   }
   C[0] = 1;
   for (int i = 1; i <= n; i++)
   {
      for (int j = min(i, k); j > 0; j--)
         C[j] = C[j] + C[j - 1];
   }
   return C[k];
}

int main()
{
   int n, k;
   printf("Enter n and k for nCk:\n");
   scanf(" %d%d", &n, &k);
   printf("%d ", bin(n, k));
   return 0;
}
```

Sample Input:
Enter n and k for nCk: 5 2
Sample Output:
10
Time complexity calculation:

```
int bin(int n, int k)
{
   int C[k + 1];   ->1
   for (int i = 1; i < k + 1; i++)   ->k+2
   {
      C[i] = 0;    ->k+1
   }
   C[0] = 1;    ->1
   for (int i = 1; i <= n; i++)    ->n+1
   {
      for (int j = min(i, k); j > 0; j--)    ->n+1
         C[j] = C[j] + C[j - 1];
   }
   return C[k];
}
Total time=0(n+k)
```

Modularity		Documentation		Indentation		Programming practices	

Type of Algorithm: Greedy

Details of the algorithm:

Greedy algorithm is a technique where the solution is build piece by piece always choosing the optimal piece that offers the obvious and immediate benefit and also considering the larger picture or the consequences of that choice.

Application of greedy Algorithm:

1. Huffman coding
2. Coin change problem
3. Interval scheduling

Steps involved

1. Define the problem clearly
2. Identify the sub problems
3. Determine the choice that u want to apply
4. Checking the choice made
5. Update the solution

Code for example 1:
Q. Job Sequencing

```
#include <stdio.h>
#include <stdlib.h>

typedef struct JOB
{
    char j;
    int time;
    int profit;
} job;

void sort(job a[], int n)
{
    for (int i = 0; i < n; i++)
    {
        for (int k = 0; k < n - i - 1; k++)
        {
            if (a[k].profit < a[k + 1].profit)
            {
                job temp = a[k];
                a[k] = a[k + 1];
                a[k + 1] = temp;
            }
        }
    }
}

void max_profit(job arr[], int n, int x)
{
    int max = -1, c = 0;
    printf("JOb should be:\n");
    for (int i = 0; i < x && i < n; i++)
    {
        for (int j = 0; j < n; j++)
        {
          if (arr[j].profit > max && arr[j].time >= i)
          {
             max = arr[j].profit;
             c = j;
          }
       }
       max = -1;
       arr[c].profit = 0;
       printf("%c ", arr[c].j);
    }
}

void main()
{
    job arr[] = {{'a', 2, 95},
               {'b', 1, 23},
               {'c', 2, 17},
               {'d', 4, 35},
               {'e', 3, 15}};
    int n = sizeof(arr) / sizeof(job);
    sort(arr, n);
    printf("enter deadline works:\n");
    int x;
    scanf("%d", &x);
    max_profit(arr, n, x);
}
```

Sample Input:

```
Enter deadline works:
3
```

Sample Output:

```
b,a,d,
```

Time complexity calculation:

Total time= O(1) + O(x) * (O(n) * O(1) + O(1) + O(1) + O(1) + O(1))
= O(1) + O(x) * O(n)
=0(x*n)

Code for example 2:
Q. Find max volume cube formed by a cuboid

```
#include <stdio.h>

int find_gcd(int a, int b)
{
    int gcd = 0;
    for (int i = 1; i <= a && i <= b; i++)
    {
        if (a % i == 0 && b % i == 0)
        {
            gcd = i;
        }
    }
    return gcd;
}

void max_volume(int l, int b, int h)
{
    int cub_l = find_gcd(l, find_gcd(b, h));
    int num = l / cub_l;
    num = (num * b) / cub_l;
    num = (num * h) / cub_l;
    printf("It can form %d numbers of cubes with side length = %d", num, cub_l);
}

void main()
{
    int l, b, h;
    printf("Enter length breadth height of cuboid:\n");
    scanf("%d%d%d", &l, &b, &h);
    max_volume(l, b, h);
}
```

Sample Input:

```
Enter length breadth height of cuboid:
4 6 8
```

Sample Output:

```
It can form 24 numbers of cubes with side length = 2
```

Time complexity calculation:

```
void max_volume(int l, int b, int h)
{
    int cub_l = find_gcd(l, find_gcd(b, h));
    int num = l / cub_l;
    num = (num * b) / cub_l;
    num = (num * h) / cub_l;
    printf("It can form %d numbers of cubes with side length = %d", num, cub_l);
}
```

Time complexity=0(log(min(l,b,h))

Modularity		Documentation		Indentation		Programming practices	
Type of Algorithm: Branch and bound							

Details of the algorithm:

Branch and bound is an algorithm design paradigm which is generally used for solving combinatorial optimization problems. These problems are typically exponential in terms of time complexity and may require exploring all possible permutations in worst case. The Branch and Bound Algorithm technique solves these problems relatively quickly.
Applications of backtracking:

1. Its used in decision making problems.
2. Its also used for optimization of the problem.

It also used for enumeration type problem where all the feasible solutions are required

Code for example 1:
Q. Find max number in a array

```
#include <stdio.h>

int maxElement = -1;

void findMax(int arr[], int n, int currentIndex)
{
   if (currentIndex == n)
   {
      return;
   }

   if (arr[currentIndex] > maxElement)
   {
      maxElement = arr[currentIndex];
   }

   findMax(arr, n, currentIndex + 1);
}

int main()
{
   int arr[] = {5, 9, 2, 7, 3};
   int n = sizeof(arr) / sizeof(arr[0]);

   findMax(arr, n, 0);

   printf("Maximum element: %d\n", maxElement);

   return 0;
}
```

Sample Input:

```
5,9,2,7,3
```

Sample Output:

```
Maximum element:9
```

Time complexity calculation:

Total time =T(n)=T(n-1)+0(n)
T(n)=0(1)+0(1)+.....0(1)
T(n)=0(n)

Code for example 2:
Q. Sum of given array

```
#include <stdio.h>

int sum(int a[], int n)
 {
   if (n == 0) {
      return 0;
   }

   int cur_ele = a[0];
   int remainingsum = sum(a + 1, n - 1);

   return cur_ele + remainingsum;
}

void main()
{
   int n,i,sum1;
   printf("Enter n:");
   scanf("%d",&n);
   int a[n];
   printf("\nEnter elements: ");
   for(i=0;i<n;i++)
   {
     scanf("%d",&a[i]);
   }
    sum1=sum(a,n);
    printf("\nThe total sum is: ");
   printf("%d ",sum1);
}
```

Sample Input:

```
Enter n:
5
Enter elements:
1 2 3 4 5
```

Sample Output:

```
The total sum is:
15
```

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(0) + n * O(1)
T(n) = O(1) + n * O(1)
= O(1) + O(n)
= O(n)

Modularity		Documentation		Indentation		Programming practices	

Type of Algorithm: Brute force
Details of the algorithm:
Brute force algorithm is a technique that searches for every possible combination of the solution. It is used for solving small sized problems. Brute force is small and inefficient for large sized data. For such problems they have higher time complexities and are not feasible. The term "brute force" signifies an approach that relies on raw computational power rather than clever strategies or optimizations to solve a problem. Applications 1. Password Cracking 2. String Matching: Knuth-Morris-Pratt or Boyer-Moore algorithms. 3. Chess and Game Playing: In games like chess to evaluate all possible moves and their consequences to determine the best move Steps: 1. Define the problem. 2. Identify the range of possible solutions and constraints. 3. Create loop or nested loop to iterate through all possible combinations. Apply constraints to obtain the desired solution.
Code for example 1: Q. String length
(code below)

```
#include <stdio.h>

int str_length(char str[])
{
   int i = 0;
   while (str[i] != '\0')
   {
      i++;
   }
   return i;
}

void main()
{
   char str[100];
   printf("Enter string:\n");
   gets(str);
   int n = str_length(str);
   printf("Length of string=%d", n);
}
```

Sample Input:
Enter string: asdfg
Sample Output:
Length of string=5
Time complexity calculation:

```
int str_length(char str[])
{
   int i = 0;
   while (str[i] != '\0')
   {
      i++;    ->k
   }
   return i;
}
Time complexity =0(k)
```

Code for example 2:
Q. Finding pattern

```
#include <stdio.h>
#include <string.h>

void pattern(char str[100], char p[10], FILE *fp)
{
   int status = 1, p1 = 0, l1, l2, j, i;
   while (!feof(fp))
   {
      fgets(str, 50, fp);
      l1 = strlen(str);
      l2 = strlen(p);

      for (i = 0; i < l1 - l2; i++)
      {
         for (j = 0; j < l2; j++)
         {
            if (str[i + j] != p[j])
            {
               status = 0;
               break;
            }
         }
         if (j == l2 && status == 1)
         {
            p1 = 1;
            break;
         }
         status = 1;
      }
      if (p1 == 1)
      {
         printf("%s", str);
      }
      p1 = 0;
   }
}

void main()
{
   int n, l1, l2, i, j, k, status = 1, p1 = 0;
   char str[100], p[10];
   FILE *fp;
   fp = fopen("test.txt", "r");
   printf("Enter pattern to be found: ");
   gets(p);
   printf("\nThe lines with the required pattern: \n");
   pattern(str, p, fp);
}
```

Sample Input:

```
Enter pattern to be found:
Abcd
```

Sample Output:

```
The lines with the required pattern:
Abcdeggr
Jdsbkabcddg
ckdnabcdde
```

Time complexity calculation:

The string array the time complexity depends on the string and no of lines in the file

Modularity		Documentation		Indentation		Programming practices	

Type of Algorithm: Randomized

Details of the algorithm:

These are the algorithms that uses random numbers at any part of the algorithm which decides the next step of the algorithm. These are non-deterministic. They may not always terminate with the right answer.
The term "randomized" refers to the incorporation of randomness or random elements into the process.
Applications:
1. Computational Optimization: finding the minimum or maximum of a function.
2. Password, captcha generation.
3. Sorting algorithms: Quick Sort.

Steps:
1. Define the problem.
2. Identify the areas where randomization can be applied.
3. Include the randomization Strategy.
4. Analyse the results.
5. Test and verify the solution.

Code for example 1:
Q. Random password generator

```
#include <stdio.h>
#include <time.h>

char generate()
{
  char str1[26] = "QWERTYUIOPASDFGHJKLZXCVBNM";
  char str2[26] = "qwertyuioplkjhgfdsazxcvbnm";
  char spec[10] = "!@#$%^&*()+";
  char num[10] = "0123456789";
  int x, y;
  char c;
  x = rand() % 4;
  switch (x)
  {
  case 0:
    y = rand() % 26;
    c = str1[y];
    return c;
  case 1:
    y = rand() % 26;
    c = str2[y];
    return c;
  case 2:
    y = rand() % 10;
    c = spec[y];
    return c;
  case 3:
    y = rand() % 10;
    c = num[y];
    return c;
  }
}
```

```
void main()
{

    printf("Enter how many character password u want:\n");
    int n;
    srand(time(NULL));
    scanf("%d", &n);
    char pass[n];
    for (int i = 0; i < n; i++)
    {
        pass[i] = generate();
        printf("%c", pass[i]);
    }
}
```

Sample Input:

```
Enter how many character password u want:
5
```

Sample Output:

```
F3#g2
```

Time complexity calculation:

Time complexity=0(n)

Code for example 2:
Q. randomly rearrange the the array

```
#include <stdio.h>
#include <time.h>

void swap(int *a, int *b)
{
    int temp = *a;
    *a = *b;
    *b = temp;
}

void randomize(int a[], int n)
{
    srand(time(NULL));
    for (int i = n - 1; i >= 0; i--)
    {
        int j = rand() % (i + 1);
        swap(&a[i], &a[j]);
    }
}

void main()
{
    int a[] = {1, 2, 3, 4, 5, 6, 7, 8, 9, 10};
    int n = sizeof(a) / sizeof(int);
    for (int i = 0; i < n; i++)
    {
        printf("%d ", a[i]);
    }
    printf("\n");
    randomize(a, n);
    for (int i = 0; i < n; i++)
    {
        printf("%d ", a[i]);
    }
}
```

Sample Input:

```
1, 2, 3, 4, 5, 6, 7, 8, 9, 10
```

Sample Output:

```
4 2 3 6 9 8 5 1 7 0
```

Time complexity calculation:

```
void randomize(int a[], int n)
{
   srand(time(NULL));
   for (int i = n - 1; i >= 0; i--)
   {
      int j = rand() % (i + 1);
      swap(&a[i], &a[j]);
   }
}
```

Time complexity=0(n)