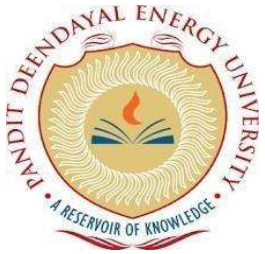


SCHOOL OF TECHNOLOGY
PANDIT DEENDAYAL ENERGY UNIVERSITY
GANDHINAGAR, GUJARAT, INDIA

Computer Science & Engineering

LAB File

(2023-24)

Design and Analysis

of

Algorithm Lab

(20CP209P)

Student Name: Sameeksha Gupta

Enrollment No.: 22BCP343

Semesters: 4

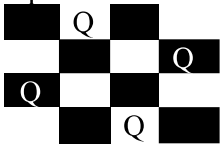
Division: 5

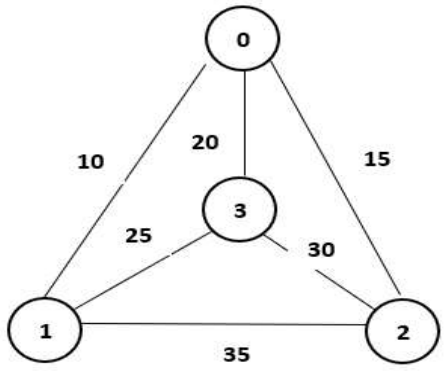
Group: 9

Instructor: Rajendra Chaudhari

List of Practical

Exp. No.	Experiment Title	Date	Signature
1	Implement the following sorting in C programming language. a) Bubble sort b) Insertion sort c) Selection sort Now, measure the execution time and the number of steps required to execute each algorithm in best case, worst case, and average case.		
2	Implement the following sorting in C programming language. d) Merge sort e) Quick sort f) Radix sort Now, measure the execution time and the number of steps required to execute each algorithm in best case, worst case, and average case.		
3	Use singly linked lists to implement integers of unlimited size. Each node of the list should store one digit of the integer. You should implement addition, subtraction, multiplication, and exponentiation operations. Limit exponents to be positive integers. What is the asymptotic running time for each of your operations, expressed in terms of the number of digits for the two operands of each function?		
4 (I) 4 (II)	Implement a city database using unordered lists. Each database record contains the name of the city (a string of arbitrary length) and the coordinates of the city expressed as integer x and y coordinates. Your program should allow following functionalities: a) Insert a record, b) Delete a record by name or coordinate, c) Search a record by name or coordinate. d) Pint all records within a given distance of a specified point. Implement the database using an array-based list implementation, and then a linked list implementation. Perform following analysis: a) Collect running time statistics for each operation in both implementations. b) What are your conclusions about the relative advantages and disadvantages of the two implementations? c) Would storing records on the list in alphabetical order by city name speed any of the operations? d) Would keeping the list in alphabetical order slow any of the operations?		

5	**[Greedy Approach]** Implement interval scheduling algorithm. Given n events with their starting and ending times, find a schedule that includes as many events as possible. It is not possible to select an event partially. For example, consider the following example: **Event** — **Starting time** — **Ending time** A — 1 — 3 B — 2 — 5 C — 3 — 9 D — 6 — 8 Here, maximum number of events that can be scheduled is 2. We can schedule B and D together.		
6	**[Divide and Conquer]** Implement both a standard $O(n^3)$ matrix multiplication algorithm and Strassen's matrix multiplication algorithm. Using empirical testing, try and estimate the constant factors for the runtime equations of the two algorithms. How big must n be before Strassen's algorithm becomes more efficient than the standard algorithm?		
7	**[Dynamic Programming]** implement the Floyd Warshall Algorithm for All Pair Shortest Path Problem. You are given a weighted diagraph $G = (V, E)$, with arbitrary edge weights or costs c_{vw} between any node v and node w. Find the cheapest path from every node to every other node. Edges may have negative weights. Consider the following test case to check your algorithm: v — w — c_{vw} 0 — 1 — -1 0 — 2 — 4 1 — 2 — 3 1 — 3 — 2 1 — 4 — 2 3 — 2 — 5 3 — 1 — 1 4 — 3 — -3		
8	**[Backtracking]** Solve the n queens' problem using backtracking. Here, the task is to place n chess queens on an n x n board so that no two queens attack each other. For example, following is a solution for the 4 Queen' problem. 		

9	**[Branch and Bound]** Given a set of cities and distance between every pair of cities, the problem is to find the shortest possible tour that visits every city exactly once and returns to the starting point. Solve this problem using branch and bound technique. For example, consider the following graph:  A Travelling Salesman Problem (TSP) tour in the graph is $0-1-3-2-0$. The cost of the tour is $10+25+30+15=80$.		
10 (I)	To design and solve given problems using different algorithmic approaches and analyze their complexity. Your friends are starting a security company that needs to obtain licenses for n different pieces of cryptographic software. Due to regulations, they can only obtain these licenses at the rate of at most one per month. Each license is currently selling for a price of \$100. However, they are all becoming more expensive according to exponential growth curves: in particular, the cost of license j increases by a factor of $r_j > 1$ each month, where r_j is a given parameter. This means that if license j is purchased t months from now, it will cost $100r_j t$. We will assume that all the price growth rates are distinct; that is, $r_i \neq r_j$ for licenses $i \neq j$ (even though they start at the sameprice of \$100). The question is: Given that the company can only buy at most one license a month, in which order should it buy the licenses so that the total amount of money it spends is as small as possible? Give an algorithm that takes the n rates of price growth $r_1, r_2, \ldots, r_n$, and computes an order in which to buy the licenses so that the total amount ofmoney spent is minimized. The running time of your algorithm should be polynomial in n.		
(II)	Suppose you are given an array A with n entries, with each entry holding a distinct number. You are told that the sequence of values $A[1], A[2], \ldots, A[n]$ is unimodal. That is, for some index p between 1 and n, the values in the array entries increase up to position p in A and then decrease the remainder of the way until position n. (So if you were to draw a plot with the array position j on the x-axis and the value of the entry $A[j]$ on the y-axis, the plotted points would rise until x-value p, where they'd achieve their maximum value, and then fall from there on). You'd like to find the "peak entry" p without having to read the entire array - in fact, by reading as few entries of A as possible. Show		

	how to find the entry p by reading at most $O(logn)$ entries of A.		

Instruction's

i. Make all the programs using C language
ii. You are allowed to use only gcc compiler and command prompt for running the programming
iii. With each program you have to print your name and enrollment number.
iv. In each lab after making the programs, paste the code (in text format) with the output (snapshot of the output) in this file.
v. You have to submit only soft-copy of the lab manual.

Exp.No. : 1

Implement the following sorting in C programming language.

a) Bubble sort

CODE:
```
#include<stdio.h>
void main(){
    printf("-Sameeksha Gupta 22BCP343--\n");
    printf("Enter the number of elements which you have to enter: ");
    int n;
    scanf("%d", &n);
    int arr[n-1];
    printf("Enter elements in the array: ");
    for(int i  = 0; i<n ; i++){
        scanf("%d", &arr[i]);
    }
    BubbleSort(arr, n);
    printf("Sorted array: ");
    for (int i = 0; i < n; i++)
        printf("%d ", arr[i]);
    printf("\n");
}
void BubbleSort(int arr[],int n){
    for(int i = 0; i<n; i++){
        for(int j = 0; j<n-i-1; j++){
            if(arr[j]>arr[j+1]){
                int t = arr[j];
                arr[j] = arr[j+1];
                arr[j+1] = t;
            }
        }
    }
}
```

OUTPUT:

```
C:\Users\samee\Documents\PDEU\SEM 4\LAB\DAA\practice>a.exe
-Sameeksha Gupta 22BCP343--
Enter the number of elements which you have to enter: 5
Enter elements in the array: 5
4
3
2
1
Sorted array: 1 2 3 4 5
```

Best Case:

- The best case scenario for bubble sort is when the input array is already sorted.
- In this case, the algorithm will only need to iterate through the array once without performing any

swaps because no elements need to be rearranged.

- Time complexity: O(n), where n is the number of elements in the array.
- Number of steps: *n*.

Worst Case:

- The worst case scenario for bubble sort is when the input array is sorted in reverse order.
- In this case, the algorithm will have to compare and swap each adjacent pair of elements in every pass until the array is sorted.
- Time complexity: O(n^2), where n is the number of elements in the array.
- Number of steps: *n2*.

Average Case:

- In the average case, the input array is neither already sorted nor sorted in reverse order. It is a random arrangement of elements.
- The average case time complexity of bubble sort is also O(n^2).
- The number of steps in the average case can vary depending on the exact arrangement of elements, but it will be proportional to .

Insertion sort

CODE:

```
#include<stdio.h>
void main(){
   printf("-Sameeksha Gupta 22BCP343--\n");
   int n;
   printf("Total number of elements that you want to enter: ");
   scanf("%d", &n);
   int arr[n-1];
   printf("Enter the elements in the arrays: ");
   for(int i=0; i<n; i++){
      scanf("%d", &arr[i]);
   }

   InsertionSort(arr, n);
   printf("Sorted array:");
   for(int i=0; i<n; i++)
      printf("%d ", arr[i]);
   printf("\n");
}

void InsertionSort(int arr[],int n){
   for(int i=1; i<n; i++){
      int current = arr[i];
      int j = i-1;
      while(j>=0 && arr[j]>current){
         arr[j+1] = arr[j];
         j--;
      }
      arr[j+1]= current;
```

```
    }
}
```

OUTPUT:

```
C:\Users\samee\Documents\PDEU\SEM 4\LAB\DAA\practice>a.exe
-Sameeksha Gupta 22BCP343--
Total number of elements that you want to enter: 5
Enter the elements in the arrays: 5
4
3
2
1
Sorted array:1 2 3 4 5
```

Best Case:

- The best case scenario for insertion sort is when the input array is already sorted.
- In this case, the inner loop of the algorithm will never execute because there are no elements to the left of the current element that are greater than it.
- Time complexity: O(n), where n is the number of elements in the array.
- Number of steps: *n*.

Worst Case:

- The worst case scenario for insertion sort is when the input array is sorted in reverse order.
- In this case, each element needs to be compared and shifted to the beginning of the array.
- Time complexity: O(n^2), where n is the number of elements in the array.
- Number of steps: *n2*.

Average Case:

- In the average case, the input array is not sorted but also not in reverse order.
- The average case time complexity of insertion sort is also O(n^2).
- The number of steps in the average case can vary depending on the exact arrangement of elements, but it will be proportional to *n2*.

Selection sort

CODE:

```
#include<stdio.h>
void main(){
    printf("Sameeksha Gupta 22BCP343\n");
    printf("Enter the total number of elements which you want to enter: ");
    int n;
    scanf("%d", &n);
    int arr[n-1];
    printf("Enter the elements in the array: ");
    for(int i=0; i<n; i++){
        scanf("%d", &arr[i]);
    }
    SelectionSort(arr, n);
    printf("Sorted array:");
    for(int i=0; i<n; i++)
        printf("%d ", arr[i]);
}
void SelectionSort(int arr[], int n){
    for(int i=0; i<n; i++){
        int minimum = i;
        for(int j=i+1; j<n; j++){
            if(arr[j]<arr[minimum]){
                minimum = j;
            }
        }
        int t = arr[i];
        arr[i] = arr[minimum];
        arr[minimum]= t;
    }
}
```

OUTPUT:

```
C:\Users\samee\Documents\PDEU\SEM 4\LAB\DAA\practice>a.exe
Sameeksha Gupta 22BCP343
Enter the total number of elements which you want to enter: 5
Enter the elements in the array: 5
4
3
2
1
Sorted array:1 2 3 4 5
C:\Users\samee\Documents\PDEU\SEM 4\LAB\DAA\practice>
```

Selection Sort:

Best Case:

- The best case scenario for Selection Sort is when the array is already sorted.
- However, even in the best case, Selection Sort has a time complexity of O(n^2) because it still needs to scan the entire array to find the minimum element.
- Number of steps: *n2*.

Worst Case:

- The worst case scenario for Selection Sort is when the array is sorted in reverse order.
- Like the best case, Selection Sort has a time complexity of O(n^2).
- Number of steps: *n2*.

Average Case:

- Similar to the worst and best cases, the average case time complexity of Selection Sort is O(n^2).
- Number of steps: *n2*.

Exp.No. : 2

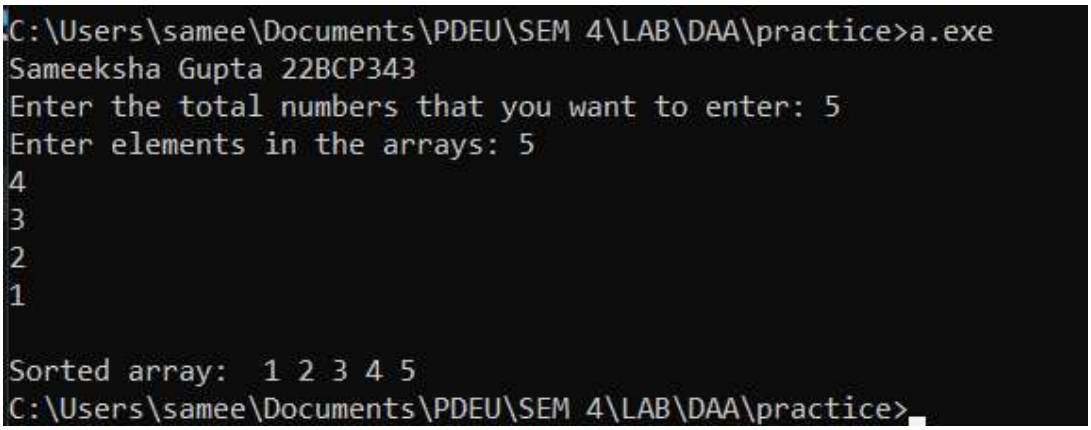
Merge sort

CODE:

```
#include<stdio.h>
void main(){
    printf("Sameeksha Gupta 22BCP343\n");
    printf("Enter the total numbers that you want to enter: ");
    int n;
    scanf("%d", &n);
    int arr[n-1];
    printf("Enter elements in the arrays: ");
    for(int i=0; i<n; i++){
        scanf("%d", &arr[i]);
    }
    mergesort(arr, 0, n-1);

    printf("\nSorted array:  ");
    for(int i = 0; i < n; i++) {
        printf("%d ", arr[i]);
    }
}
void mergesort(int arr[], int left,int right){
    if(left<right){
    int mid= (left+right)/2;
    mergesort(arr, left, mid);
    mergesort(arr, mid+1, right);
    mergersort(arr, left, mid, right);
    }
}
void mergersort(int arr[], int left, int mid, int right){
    int i,j,k;
    int n1= mid-left+1;
    int n2=right-mid;
    int arr1[n1], arr2[n2];
    i=0,j=0,k=left;
    for(int i=0;i<n1;i++){
        arr1[i]=arr[left+i];
    }
    for(int i=0; i<n2;i++){
        arr2[i]=arr[mid+i+1];
    }
    while(i<n1&& i<n2){
        if(arr1[i]<arr2[j]){
            arr[k]= arr1[i++];
        }
        else{
            arr[k]=arr2[j++];
```

```
        }
        k++;
    }
    while(i<n1){
        arr[k++]=arr1[i++];
    }
    while(j<n2){
        arr[k++] = arr2[k++];
    }
}
```

OUTPUT:



Best Case:

- The best case scenario for merge sort is when the array is already sorted.
- In this case, merge sort will still divide the array into halves recursively, but it will perform fewer comparisons during the merge step.
- Time complexity: O(n log n), where n is the number of elements in the array.
- Number of steps: $O(n \log n)$.

Worst Case:

- The worst case scenario for merge sort occurs when the array is sorted in reverse order.
- Merge sort will still divide the array into halves recursively, and each merge step will require a significant number of comparisons.
- Time complexity: O(n log n), where n is the number of elements in the array.
- Number of steps: $O(n\log n)$.

Average Case:

- In the average case, the input array is not sorted but also not in reverse order.
- Merge sort exhibits consistent behavior regardless of the arrangement of elements.
- Time complexity: O(n log n), where n is the number of elements in the array.
- Number of steps: $O(n\log n)$.

Quick sort

CODE:

```
#include<stdio.h>
void main(){
    printf("Sameeksha Gupta 22BCP343\n");
    int n;
    printf("Total number of elements that you want to enter: ");
    scanf("%d", &n);
    int arr[n-1];
    printf("Enter the elements in the arrays: ");
    for(int i=0; i<n; i++){
        scanf("%d", &arr[i]);
    }

    quicksort(arr, 0, n-1);
    printf("Sorted array:");
    for(int i=0; i<n; i++)
        printf("%d ", arr[i]);
    printf("\n");
}
void quicksort(int arr[], int left, int right){
    if(left<right){
        int partitionIndex = partition(arr, left, right);
        quicksort(arr, left, partitionIndex-1);
        quicksort(arr, partitionIndex+1, right);
    }
}
int partition(int arr[],int left, int right){
    int pivot = arr[left];
    int pivotIndex = right;
    for(int i=right; i>left; i--){
        if(arr[i]>pivot){
            int temp = arr[pivotIndex];
            arr[pivotIndex]= arr[i];
            arr[i]= temp;
```

```
            pivotIndex--;
        }
    }
    int temp = arr[left];
    arr[left]= arr[pivotIndex];
    arr[pivotIndex]= temp;
    return pivotIndex;
}
```

OUTPUT:

```
C:\Users\samee\Documents\PDEU\SEM 4\LAB\DAA\practice>a.exe
Sameeksha Gupta 22BCP343
Total number of elements that you want to enter: 5
Enter the elements in the arrays: 5
4
3
2
1
Sorted array:1 2 3 4 5
```

Best Case:

- In the best-case scenario, the pivot divides the array into two subarrays of roughly equal size.
- Each partitioning step divides the array into two halves, so the partitioning process requires $O(n)$ time.
- After partitioning, the quicksort algorithm recursively sorts the two subarrays.
- The recurrence relation for the best-case time complexity of Quick Sort is $T(n)=2T(n/2)+O(n)$, which results in a time complexity of $O(n\log n)$.

Worst Case:

- In the worst-case scenario, the pivot consistently divides the array into one subarray of size 1 and one subarray of size $n-1$.
- Each partitioning step results in one subarray with $n-1$ elements and another subarray with 1 element, leading to $O(n)$ time complexity for partitioning.
- However, in the worst case, the algorithm doesn't efficiently divide the array, resulting in $O(n2)$ time complexity.
- The recurrence relation for the worst-case time complexity of Quick Sort is $T(n)=T(n-1)+O(n)$, which results in a time complexity of $O(n2)$.

Average Case:

- In the average case, the pivot divides the array into two subarrays of approximately equal size, leading to balanced partitioning.
- On average, the partitioning step takes $O(n)$ time, similar to the best case.
- Therefore, the average-case time complexity of Quick Sort is also $O(n\log n)$.

Radix sort

CODE:

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

```
int *count_sort(int *arr, int size, int exponent) {
    int *f_a ;
    f_a = (int*)malloc(sizeof(int) * 10);
    int sum = 0;

    for(int i = 0; i < 10; i++) {
        f_a[i]=0;
    }

    for(int i = 0; i < size; i++) {
        f_a[(arr[i]/exponent) % 10]++;
    }

    for(int i = 0; i < 10; i++) {
        sum = sum + f_a[i];
        f_a[i] = sum;
    }

    int *new_arr;
    new_arr = (int*)malloc(sizeof(int) * size);
    int pos;
    for(int i = size-1; i >= 0; i--){
        pos = f_a[(arr[i]/exponent)%10]-1;
        new_arr[pos] = arr[i];
        f_a[(arr[i]/exponent)%10]--;
    }

    return new_arr;
}

int maximum(int *arr, int length) {
    int max=0;
    for( int i = 0 ; i < length; i++)
    {
        if(arr[i]>max)
            max=arr[i];
    }
    return max;
}

int *radix_sort(int *arr, int size)
{
    int max_ele = maximum(arr, size);
    int exp = 1;
    while(exp <= max_ele)
    {
        arr = count_sort(arr, size, exp);
        exp = exp * 10;
```

```
    }
    return arr;
}

void main() {

    printf("-Sameeksha Gupta 22BCP343--\n");
    int arrayLength;
    printf("Enter Array Size : ");
    scanf("%d", &arrayLength);

    int inputArray[arrayLength];
    int *outputArray;
    outputArray = (int*)malloc(sizeof(int) * arrayLength);

    printf("Enter %d integers:\n", arrayLength);
    for (int i = 0; i < arrayLength; i++) {
        scanf("%d", &inputArray[i]);
    }

    printf("Original array: ");
    for (int i = 0; i < arrayLength; i++) {
        printf("%d ", inputArray[i]);
    }

    outputArray = radix_sort(inputArray, arrayLength);

    printf("\nSorted array: ");
    for (int i = 0; i < arrayLength; i++) {
        printf("%d ", outputArray[i]);
    }
}
```

OUTPUT:

```
C:\Users\samee\Documents\PDEU\SEM 4\LAB\DAA\practice>a.exe
Sameeksha Gupta 22BCP343
Total number of elements that you want to enter: 5
Enter the elements in the arrays: 5
4
3
2
1
Sorted array:1 2 3 4 5
```

Best Case:

The best-case scenario for radix sort occurs when all the elements have the same number of digits, and thus, the distribution across buckets in each pass is uniform. In this case, the time complexity is O(n * k), where n is the number of elements in the array, and k is the number of digits in the maximum element.

Worst Case:

The worst-case scenario happens when there's a significant difference in the number of digits among elements, leading to non-uniform distribution across buckets in each pass. In the worst case, the time complexity is O(n * k), where n is the number of elements in the array, and k is the number of digits in the maximum element.

Average Case:

The average-case time complexity is also O(n * k), similar to the best and worst case. This is because radix sort processes each digit of each element in the array, which takes linear time relative to the number of elements and the number of digits.

Exp.No. : 3

Use singly linked lists to implement integers of unlimited size. Each node of the list should store one digit of the integer. You should implement addition, subtraction, multiplication, and exponentiation operations. Limit exponents to be positive integers.

What is the asymptotic running time for each of your operations, expressed in terms of the number of digits for the two operands of each function?

CODE:

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

struct Node {
   int data;
   struct Node* next;
};

struct Node* insert(struct Node* head, int data, int flag) {
   struct Node* newNode = (struct Node*)malloc(sizeof(struct Node));
   newNode->data = data;
   newNode->next = NULL;
   if (!flag) {
      if (head == NULL) {
         return newNode;
      } else {
         struct Node* curr = head;
         while (curr->next != NULL) {
            curr = curr->next;
         }
         curr->next = newNode;
         return head;
      }
   } else {
      if (head == NULL) {
         return newNode;
      } else {
         newNode->next = head;
         head = newNode;
         return head;
      }
   }
}

struct Node* addition(struct Node* head1, struct Node* head2) {
   struct Node* addList = NULL;
   int carry = 0;
   while (head1 != NULL && head2 != NULL) {
      int digOne = head1->data;
      int digTwo = head2->data;
      int sum = digOne + digTwo + carry;
      carry = sum / 10;
      sum = sum % 10;
      addList = insert(addList, sum, 1);
      head1 = head1->next;
      head2 = head2->next;
```

```
    }
    while (head1 != NULL) {
        int digit = head1->data + carry;
        carry = digit / 10;
        digit = digit % 10;
        addList = insert(addList, digit, 1);
        head1 = head1->next;
    }
    while (head2 != NULL) {
        int digit = head2->data + carry;
        carry = digit / 10;
        digit = digit % 10;
        addList = insert(addList, digit, 1);
        head2 = head2->next;
    }
    if (carry > 0) {=
        addList = insert(addList, carry, 1);
    }
    return addList;
}

struct Node* subtraction(struct Node* head1, struct Node* head2, int isNo1Greater) {
    struct Node* subList = NULL;
    int borrow = 0;

    while (head1 != NULL && head2 != NULL) {
        int digOne = head1->data;
        int digTwo = head2->data;
        int sub = digOne - digTwo + borrow;

        if (sub < 0) {
            borrow = -1;
            sub += 10;
        } else {
            borrow = 0;
        }
        subList = insert(subList, sub, 1);

        head1 = head1->next;
        head2 = head2->next;
    }

    while (head1 != NULL) {
        int digit = head1->data + borrow;
        if (digit < 0) {
            borrow = -1;
            digit += 10;



         } else {
            borrow = 0;
        }
        subList = insert(subList, digit, 1);
        head1 = head1->next;
    }
```

```
    if (borrow < 0) {
        subList = insert(subList, borrow, 1);
    }

    if (!isNo1Greater) {
        subList->data = -(subList->data);
    }

    return subList;
}

struct Node* multiply(struct Node* num1, struct Node* num2) {
    struct Node* result = NULL;

    int multiplier = 0;
    struct Node* temp = num2;
    while (temp != NULL) {
        multiplier = multiplier * 10 + temp->data;
        temp = temp->next;
    }

    for (int i = 0; i < multiplier; i++) {
        result = addition(result, num1);
    }

    return result;
}

struct Node* power(struct Node* base, struct Node* exponent) {
    struct Node* result = NULL;

    // Initialize result with value 1
    result = insert(result, 1, 0);

    // Convert the exponent linked list to integer
    int exp = 0;
    struct Node* temp = exponent;
    while (temp != NULL) {
        exp = exp * 10 + temp->data;
        temp = temp->next;
    }

    // Multiply base by itself 'exp' times
    for (int i = 0; i < exp; i++) {
        result = multiply(result, base);
    }

    return result;
}

void printList(struct Node* head) {
    while (head != NULL) {
        printf("%d", head->data);
        head = head->next;
    }
    printf("\n");
```

```
}

int main() {
    printf("-Sameeksha Gupta 22BCP343--\n");
    int no1, no2;
    printf("Enter any two numbers : ");
    scanf("%d %d", &no1, &no2);
    int isNo1Greater = 1;
    if (no1 < no2) {
        isNo1Greater = 0;
    }
    struct Node* head1 = NULL;
    struct Node* head2 = NULL;
    while (no1 > 0) {
        int rem = no1 % 10;
        head1 = insert(head1, rem, 0);
        no1 = no1 / 10;
    }
    while (no2 > 0) {
        int rem = no2 % 10;
        head2 = insert(head2, rem, 0);
        no2 = no2 / 10;
    }
    printf("Addition of Two Linked Lists : ");
    struct Node* add = addition(head1, head2);
    printList(add);
    struct Node* sub = NULL;

    if (isNo1Greater) {
        sub = subtraction(head1, head2, isNo1Greater);
    } else {
        sub = subtraction(head2, head1, isNo1Greater);
    }

    printf("Subtraction of Two Linked Lists : ");
    printList(sub);

    struct Node* mul = multiply(head1, head2);
    printf("Multiplication of Two Linked Lists : ");
    printList(mul);

    printf("Exponential result: ");
    struct Node* expResult = power(head1, head2);
    printList(expResult);

    return 0;
}
```

OUTPUT:

```
C:\Users\samee\Documents\PDEU\SEM 4\LAB\DAA>a.exe
-Sameeksha Gupta 22BCP343--
Enter any two numbers : 34
12
Addition of Two Linked Lists : 46
Subtraction of Two Linked Lists : 22
Multiplication of Two Linked Lists : 1020
Exponential result: 649101104047232963254697678821870 84

C:\Users\samee\Documents\PDEU\SEM 4\LAB\DAA>
```

Addition:
n the addition function, we traverse both linked lists simultaneously once, which takes O(d), where d is the maximum number of digits between the two operands. Therefore, the asymptotic running time for addition is O(d).

Subtraction:
Similar to addition, subtraction also traverses both linked lists once, which takes O(d), where d is the maximum number of digits between the two operands. Therefore, the asymptotic running time for subtraction is also O(d).

Multiplication:
In the multiply function, we perform addition 'n' times, where 'n' is the value of the multiplier, which is represented by the second operand. The value of the multiplier can have at most 'd' digits, where 'd' is the maximum number of digits in the second operand. Therefore, the maximum number of iterations of addition is O(d). Hence, the overall time complexity for multiplication is O(d).

Exponentiation:
In the power function, we multiply the base by itself 'n' times, where 'n' is the value of the exponent. Similar to multiplication, the value of the exponent can have at most 'd' digits, where 'd' is the maximum number of digits in the second operand. Therefore, the maximum number of iterations of multiplication is also O(d). Thus, the overall time complexity for exponentiation is O(d).

In summary, the asymptotic running time for each operation is as follows:

- Addition: O(d)
- Subtraction: O(d)
- Multiplication: O(d)
- Exponentiation: O(d)

Where 'd' is the maximum number of digits among the operands.

4
1)

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <math.h>
#include <time.h>

typedef struct record {
   char *cityName;
   int x;
   int y;
} record;

typedef struct database {
   int size;
   record *records;
} database;

void printDatabase(database *db) {
   for(int i = 0; i < db->size; i++) {
      printf("%s %d %d\n", db->records[i].cityName, db->records[i].x , db->records[i].y);
   }
}

void insertRecord(database *db) {
   db->records = (record*)realloc(db->records, (db->size+1)*sizeof(record));
   if(!db) {
      printf("Memory allocation failed...Exiting");
      return;
   }
   else {
      db->records[db->size].cityName = (char*)malloc(25*sizeof(char));

      printf("Enter city name: ");
      scanf("%s", db->records[db->size].cityName);
      printf("Enter x-coordinate: ");
      scanf("%d", &db->records[db->size].x);
      printf("Enter y-coordinate: ");
      scanf("%d", &db->records[db->size].y);

      db->size++;

      printf("Record inserted\n");
   }
}

void deleteRecord(database *db) {
   char ch;
   printf("Delete by (n)ame or (c)oordinate? n/c ");
   scanf(" %c", &ch);

   if(ch == 'n') {
      char cityName[25];
      printf("Enter city name to delete: ");
      scanf("%s", cityName);
```

```
        for(int i = 0; i < db->size; i++) {
            if(strcmp(db->records[i].cityName, cityName) == 0) {
                free(db->records[i].cityName); // deallocate the char array

                for(int j = i; j < db->size - 1; j++) {
                    db->records[j] = db->records[j + 1];
                }

                db->records = realloc(db->records, (db->size-1)*sizeof(record));
                db->size--;
                printf("Record deleted\n");
                return;
            }
        }
        printf("City not found.\n");
    }
    else if(ch == 'c') {
        int deleteX, deleteY;
        printf("Enter x coordinate to delete: ");
        scanf("%d", &deleteX);
        printf("Enter y coordinate to delete: ");
        scanf("%d", &deleteY);

        for(int i = 0; i < db->size; i++) {
            if(db->records[i].x == deleteX && db->records[i].y == deleteY) {
                free(db->records[i].cityName);

                for(int j = i; j < db->size - 1; j++) {
                    db->records[j] = db->records[j + 1];
                }

                db->records = realloc(db->records, (db->size-1)*sizeof(record));
                db->size--;
                printf("Record deleted\n");
                return;
            }
        }
        printf("Coordinate not found.\n");
    }
    else {
        printf("Invalid option...Exiting\n");
    }
}

void searchRecord(database *db) {
    char ch;
    printf("Search by (n)ame or (c)oordinate? n/c ");
    scanf(" %c", &ch);

    if(ch == 'n') {
        char cityName[25];
        printf("Enter city name to search: ");
        scanf("%s", cityName);

        for(int i = 0; i < db->size; i++) {
```

```
            if(strcmp(db->records[i].cityName, cityName) == 0) {
                printf("%s %d %d\n", db->records[i].cityName, db->records[i].x , db->records[i].y);
            }
        }
    }
    else if(ch == 'c') {
        int searchX, searchY;
        printf("Enter x coordinate to search: ");
        scanf("%d", &searchX);
        printf("Enter y coordinate to search: ");
        scanf("%d", &searchY);

        for(int i = 0; i < db->size; i++) {
            if(db->records[i].x == searchX && db->records[i].y == searchY) {
                printf("%s %d %d\n", db->records[i].cityName, db->records[i].x , db->records[i].y);
            }
        }
    }
    else {
        printf("Invalid option...Exiting\n");
    }
}

int main() {
    database db = {0, NULL};
    printf("-Sameeksha Gupta 22BCP343-\n");

    int ch;

    do {
        printf("\nCity Database Menu:\n");
        printf("1) Insert Record\n");
        printf("2) Delete Record\n");
        printf("3) Search Record\n");
        printf("4) Print Database\n");
        printf("0) Quit\n");
        printf("Enter your choice: ");
        scanf("%d", &ch);

        clock_t start, end;
        double cpu_time_used;

        // Start timing
        start = clock();

        switch(ch) {
            case 1:
                insertRecord(&db);
                break;
            case 2:
                deleteRecord(&db);
                break;
            case 3:
                searchRecord(&db);
                break;
            case 4:
```

```
            printDatabase(&db);
            break;
        case 0:
            printf("Exiting program.\n");
            break;
        default:
            printf("Invalid choice. Please try again.\n");
    }

    // End timing
    end = clock();
    cpu_time_used = ((double) (end - start)) / CLOCKS_PER_SEC;
    printf("Time taken for the operation: %f seconds\n", cpu_time_used);

} while (ch != 0);

// Free dynamically allocated memory
for(int i = 0; i < db.size; i++) {
    free(db.records[i].cityName);
}
free(db.records);

return 0;
}
```

OUTPUT:

```
C:\Users\samee\Documents\PDEU\SEM 4\LAB\DAA>a.exe
-Sameeksha Gupta 22BCP343-

City Database Menu:
1) Insert Record
2) Delete Record
3) Search Record
4) Print Database
0) Quit
Enter your choice: 1
Enter city name: surat
Enter x-coordinate: 3
Enter y-coordinate: 4
Record inserted
Time taken for the operation: 6.607000 seconds

City Database Menu:
1) Insert Record
2) Delete Record
3) Search Record
4) Print Database
0) Quit
Enter your choice: 1
Enter city name: ahmd
Enter x-coordinate: 2
Enter y-coordinate: 4
Record inserted
Time taken for the operation: 4.478000 seconds

City Database Menu:
1) Insert Record
2) Delete Record
3) Search Record
4) Print Database
0) Quit
Enter your choice: 3
Search by (n)ame or (c)oordinate? n/c n
Enter city name to search: surat
surat 3 4
Time taken for the operation: 7.632000 seconds
```

```
City Database Menu:
1) Insert Record
2) Delete Record
3) Search Record
4) Print Database
0) Quit
Enter your choice: 2
Delete by (n)ame or (c)oordinate? n/c n
Enter city name to delete: surat
Record deleted
Time taken for the operation: 3.787000 seconds

City Database Menu:
1) Insert Record
2) Delete Record
3) Search Record
4) Print Database
0) Quit
Enter your choice: 4
ahmd 2 4
Time taken for the operation: 0.002000 seconds

City Database Menu:
1) Insert Record
2) Delete Record
3) Search Record
4) Print Database
0) Quit
Enter your choice:
```

4

2)

```
#include<stdio.h>
#include<stdlib.h>
#include<string.h>
#include<math.h>
#include<time.h>

typedef struct record {
    char *cityName;
    int x;
    int y;
} record;

typedef struct database {
    int size;
    record *records;
} database;
```

```
void printDatabase(database *db) {
   for(int i = 0; i < db->size; i++) {
      printf("%s %d %d\n", db->records[i].cityName, db->records[i].x , db->records[i].y);
   }
}

void insertRecord(database *db) {
   db->records = (record*)realloc(db->records, (db->size+1)*sizeof(record));
   if(!db) {
      printf("Memory allocation failed...Exiting");
      return;
   }
   else {
      db->records[db->size].cityName = (char*)malloc(25*sizeof(char));

      printf("Enter city name: ");
      scanf("%s", db->records[db->size].cityName);
      printf("Enter x-coordinate: ");
      scanf("%d", &db->records[db->size].x);
      printf("Enter y-coordinate: ");
      scanf("%d", &db->records[db->size].y);

      db->size++;

      printf("Record inserted\n");
   }
}

void deleteRecord(database *db) {
   char ch;
   printf("Delete by (n)ame or (c)oordinate? n/c ");
   scanf(" %c", &ch);

   clock_t start, end;
   double cpu_time_used;

   // Start timing
   start = clock();

   if(ch == 'n') {
      char cityName[25];
      printf("Enter city name to delete: ");
      scanf("%s", cityName);

      for(int i = 0; i < db->size; i++) {
         if(strcmp(db->records[i].cityName, cityName) == 0) {
            free(db->records[i].cityName);
            // deallocate the char array

            for(int j = i; j < db->size - 1; j++) {
               db->records[j] = db->records[j + 1];
            }

            db->records = realloc(db->records, (db->size-1)*sizeof(record));
            db->size--;
```

```
            // End timing
            end = clock();
            cpu_time_used = ((double) (end - start)) / CLOCKS_PER_SEC;
            printf("Record deleted\n");
            printf("Time taken: %f seconds\n", cpu_time_used);
            return;
        }
    }
    printf("City not found.\n");
}
else if(ch == 'c') {
    int deleteX, deleteY;
    printf("Enter x coordinate to delete: ");
    scanf("%d", &deleteX);
    printf("Enter y coordinate to delete: ");
    scanf("%d", &deleteY);

    for(int i = 0; i < db->size; i++) {
        if(db->records[i].x == deleteX && db->records[i].y == deleteY) {
            free(db->records[i].cityName);

            for(int j = i; j < db->size - 1; j++) {
                db->records[j] = db->records[j + 1];
            }
            db->records = realloc(db->records, (db->size-1)*sizeof(record));
            db->size--;

            // End timing
            end = clock();
            cpu_time_used = ((double) (end - start)) / CLOCKS_PER_SEC;
            printf("Record deleted\n");
            printf("Time taken: %f seconds\n", cpu_time_used);
            return;
        }
    }
    printf("Coordinate not found.\n");
}
else {
    printf("Invalid option...Exiting\n");
}
}

void searchRecord(database *db) {
    char ch;
    printf("Search by (n)ame or (c)oordinate? n/c ");
    scanf(" %c", &ch);

    clock_t start, end;
    double cpu_time_used;

    // Start timing
    start = clock();

    if(ch == 'n') {
        char cityName[25];
        printf("Enter city name to search: ");
```

```
        scanf("%s", cityName);

        for(int i = 0; i < db->size; i++) {
            if(strcmp(db->records[i].cityName, cityName) == 0) {
                printf("%s %d %d\n", db->records[i].cityName, db->records[i].x , db->records[i].y);
            }
        }
    }
    else if(ch == 'c') {
        int searchX, searchY;
        printf("Enter x coordinate to search: ");
        scanf("%d", &searchX);
        printf("Enter y coordinate to search: ");
        scanf("%d", &searchY);

        for(int i = 0; i < db->size; i++) {
            if(db->records[i].x == searchX && db->records[i].y == searchY) {
                printf("%s %d %d\n", db->records[i].cityName, db->records[i].x , db->records[i].y);
            }
        }
    }
    else {
        printf("Invalid option...Exiting\n");
    }

    // End timing
    end = clock();
    cpu_time_used = ((double) (end - start)) / CLOCKS_PER_SEC;
    printf("Time taken: %f seconds\n", cpu_time_used);
}

void searchAround(database *db) {
    char ch;
    printf("Search by (n)ame or (c)oordinate? n/c ");
    scanf(" %c", &ch);

    clock_t start, end;
    double cpu_time_used;

    // Start timing
    start = clock();

    if(ch == 'n') {
        char cityName[25];
        int radius;
        printf("Enter city name to search around: ");
        scanf("%s", cityName);
        printf("Enter search radius: ");
        scanf("%d", &radius);

        for(int i = 0 ; i < db->size; i++) {
            if(strcmp(db->records[i].cityName, cityName) == 0) {
                int x1 = db->records[i].x, y1 = db->records[i].y;
                for(int j = 0; j < db->size; j++) {
                    if(j == i) continue;
```

```
                if(sqrt(pow(db->records[j].x-x1, 2) + pow(db->records[j].y-y1, 2)) <= radius) {
                    printf("%s %d %d\n", db->records[j].cityName, db->records[j].x , db->records[j].y);
                }
            }
            // End timing
            end = clock();
            cpu_time_used = ((double) (end - start)) / CLOCKS_PER_SEC;
            return;
        }
    }
    printf("No such record found");
  }
  else if(ch == 'c') {
    int searchX, searchY, radius;
    printf("Enter x coordinate to search around: ");
    scanf("%d", &searchX);
    printf("Enter y coordinate to search around: ");
    scanf("%d", &searchY);
    printf("Enter search radius: ");
    scanf("%d", &radius);

    for(int i = 0; i < db->size; i++) {
        if(db->records[i].x == searchX && db->records[i].y == searchY) {
            for(int j = 0; j < db->size; j++) {
                if(j == i) continue;

                if(sqrt(pow(db->records[j].x-searchX, 2) + pow(db->records[j].y-searchY, 2)) <= radius) {
                    printf("%s %d %d\n", db->records[j].cityName, db->records[j].x , db->records[j].y);
                }
            }
            // End timing
            end = clock();
            cpu_time_used = ((double) (end - start)) / CLOCKS_PER_SEC;
            return;
        }
    }
    printf("No such record found");
  }
  else {
    printf("Invalid option...Exiting\n");
  }
}

void main() {
  database db = {0, NULL};

  int ch;
  printf("-Sameeksha Gupta 22BCP343--");

  do {
    printf("\nCity Database Menu:\n");
    printf("1) Insert Record\n");
    printf("2) Delete Record\n");
    printf("3) Search Record\n");
    printf("4) Search Around\n");
    printf("5) Print Database\n");
```

```
        printf("0) Quit\n");
        printf("Enter your choice: ");
        scanf("%d", &ch);
        printf("\n");

        switch(ch) {
            case 1:
                insertRecord(&db);
                break;
            case 2:
                deleteRecord(&db);
                break;
            case 3:
                searchRecord(&db);
                break;
            case 4:
                searchAround(&db);
                break;
            case 5:
                printDatabase(&db);
                break;
            case 0:
                printf("Exiting program.\n");
                break;
            default:
                printf("Invalid choice. Please try again.\n");
        }

    } while (ch != 0);
}
```

OUTPUT:

```
C:\Users\samee\Documents\PDEU\SEM 4\LAB\DAA>a.exe
-Sameeksha Gupta 22BCP343--
City Database Menu:
1) Insert Record
2) Delete Record
3) Search Record
4) Search Around
5) Print Database
0) Quit
Enter your choice: 1

Enter city name: surat
Enter x-coordinate: 3
Enter y-coordinate: 6
Record inserted

City Database Menu:
1) Insert Record
2) Delete Record
3) Search Record
4) Search Around
5) Print Database
0) Quit
Enter your choice: 3

Search by (n)ame or (c)oordinate? n/c n
Enter city name to search: surat
surat 3 6
Time taken: 2.204000 seconds

City Database Menu:
1) Insert Record
2) Delete Record
3) Search Record
4) Search Around
5) Print Database
0) Quit
Enter your choice: 5

surat 3 6
```

```
City Database Menu:
1) Insert Record
2) Delete Record
3) Search Record
4) Search Around
5) Print Database
0) Quit
Enter your choice: 2

Delete by (n)ame or (c)oordinate? n/c n
Enter city name to delete: surat
Record deleted
Time taken: 2.383000 seconds

City Database Menu:
1) Insert Record
2) Delete Record
3) Search Record
4) Search Around
5) Print Database
0) Quit
Enter your choice: 5


City Database Menu:
1) Insert Record
2) Delete Record
3) Search Record
4) Search Around
5) Print Database
0) Quit
Enter your choice:
```

1. **Advantages and Disadvantages:**

- Array-based List:
  - Advantages: Random access, faster search in sorted arrays.
  - Disadvantages: Costly insertion and deletion.
- Linked List:
  - Advantages: Cheap insertion and deletion.
  - Disadvantages: Linear search, memory overhead of pointers.

2. **Storing Records in Alphabetical Order:**

- Storing records in alphabetical order by city name could speed up search operations when searching by name. However, it would slow down insertion and deletion operations as the list may need to be rearranged.

3. **Effect of Alphabetical Ordering on Operations:**

- Alphabetical ordering could slow down insertion and deletion operations due to the need for rearranging the list. However, it could speed up search operations when searching by name as it enables binary search in the array-based implementation. In the linked list implementation, binary search is not feasible, so the effect may not be as significant.

[Greedy Approach]

5)Implement interval scheduling algorithm. Given n events with their starting and ending times, find a schedule that includes as many events as possible. It is not possible to select an event partially. For example, consider the following example:

Event	Starting time	Ending time
A	1	3
B	2	5
C	3	9
D	6	8

Here, maximum number of events that can be scheduled is 2. We can schedule B and D together.

CODE:

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

struct Event {
   char name;
   int start;
   int end;
};

int compare(const void *a, const void *b) {
   return ((struct Event*)a)->end - ((struct Event*)b)->end;
}

int interval_scheduling(struct Event *events, int n) {
   qsort(events, n, sizeof(struct Event), compare);

   struct Event *schedule = malloc(n * sizeof(struct Event));
   if (schedule == NULL) {
      fprintf(stderr, "Memory allocation failed.\n");
      exit(EXIT_FAILURE);
   }

   int schedule_index = 0;
   int last_end_time = INT_MIN;

   for (int i = 0; i < n; i++) {
      if (events[i].start >= last_end_time) {
         schedule[schedule_index++] = events[i];
         last_end_time = events[i].end;
      }
   }

   printf("Schedule:\n");
   for (int i = 0; i < schedule_index; i++) {
      printf("%c\t%d\t%d\n", schedule[i].name, schedule[i].start, schedule[i].end);
   }

   free(schedule);
```

```
    return schedule_index;
}

int main() {
    printf("—Sameeksha Gupta 22BCP343--\n");

    int n;
    printf("Enter the number of events: ");
    scanf("%d", &n);

    struct Event *events = malloc(n * sizeof(struct Event));
    if (events == NULL) {
        fprintf(stderr, "Memory allocation failed.\n");
        exit(EXIT_FAILURE);
    }

    printf("Enter the events in the format (name start_time end_time):\n");
    for (int i = 0; i < n; i++) {
        scanf(" %c %d %d", &events[i].name, &events[i].start, &events[i].end);
    }

    int max_events = interval_scheduling(events, n);
    printf("Maximum number of events that can be scheduled: %d\n", max_events);

    free(events);

    return 0;
}
```

OUTPUT:

```
C:\Users\samee\Documents\PDEU\SEM 4\LAB\DAA>a.exe
Sameeksha Gupta 22BCP343
Enter the number of events: 4
Enter the events in the format (name start_time end_time):
a 3 5
b 2 4
c 1 5
d 4 7
Schedule:
b       2       4
d       4       7
Maximum number of events that can be scheduled: 2
```

6) Stressen Matrix

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

void add_matrices(int **A, int **B, int **C, int n) {
    for (int i = 0; i < n; i++) {
        for (int j = 0; j < n; j++) {
            C[i][j] = A[i][j] + B[i][j];
        }
    }
}

int **allocate_matrix(int n) {
    int **matrix = (int **)malloc(n * sizeof(int *));
    for (int i = 0; i < n; i++) {
        matrix[i] = (int *)malloc(n * sizeof(int));
    }
    return matrix;
}

void standard_matrix_multiply(int **A, int **B, int **C, int n) {
    for (int i = 0; i < n; i++) {
        for (int j = 0; j < n; j++) {
            C[i][j] = 0;
            for (int k = 0; k < n; k++) {
                C[i][j] += A[i][k] * B[k][j];
            }
        }
    }
}

void print_matrix(int **matrix, int n) {
    for (int i = 0; i < n; i++) {
        for (int j = 0; j < n; j++) {
            printf("%d ", matrix[i][j]);
        }
        printf("\n");
    }
}

void input_matrix(int **matrix, int n) {
    printf("Enter matrix values row by row:\n");
    for (int i = 0; i < n; i++) {
        printf("Row %d: ", i + 1);
        for (int j = 0; j < n; j++) {
            scanf("%d", &matrix[i][j]);
        }
    }
}

// Strassen's matrix multiplication algorithm
void strassen_matrix_multiply(int **A, int **B, int **C, int n) {
    if (n == 1) {
        C[0][0] = A[0][0] * B[0][0];
        return;
```

```
}

int newSize = n / 2;

int **A11 = allocate_matrix(newSize);
int **A12 = allocate_matrix(newSize);
int **A21 = allocate_matrix(newSize);
int **A22 = allocate_matrix(newSize);
int **B11 = allocate_matrix(newSize);
int **B12 = allocate_matrix(newSize);
int **B21 = allocate_matrix(newSize);
int **B22 = allocate_matrix(newSize);
int **C11 = allocate_matrix(newSize);
int **C12 = allocate_matrix(newSize);
int **C21 = allocate_matrix(newSize);
int **C22 = allocate_matrix(newSize);

// Partitioning input matrices into submatrices
for (int i = 0; i < newSize; i++) {
    for (int j = 0; j < newSize; j++) {
        A11[i][j] = A[i][j];
        A12[i][j] = A[i][j + newSize];
        A21[i][j] = A[i + newSize][j];
        A22[i][j] = A[i + newSize][j + newSize];
        B11[i][j] = B[i][j];
        B12[i][j] = B[i][j + newSize];
        B21[i][j] = B[i + newSize][j];
        B22[i][j] = B[i + newSize][j + newSize];
    }
}

int **temp1 = allocate_matrix(newSize);
int **temp2 = allocate_matrix(newSize);

// Strassen's recursive calls
standard_matrix_multiply(A11, B11, temp1, newSize);
standard_matrix_multiply(A12, B21, temp2, newSize);
add_matrices(temp1, temp2, C11, newSize);

standard_matrix_multiply(A11, B12, temp1, newSize);
standard_matrix_multiply(A12, B22, temp2, newSize);
add_matrices(temp1, temp2, C12, newSize);

standard_matrix_multiply(A21, B11, temp1, newSize);
standard_matrix_multiply(A22, B21, temp2, newSize);
add_matrices(temp1, temp2, C21, newSize);

standard_matrix_multiply(A21, B12, temp1, newSize);
standard_matrix_multiply(A22, B22, temp2, newSize);
add_matrices(temp1, temp2, C22, newSize);

// Combining submatrices into the result matrix
for (int i = 0; i < newSize; i++) {
    for (int j = 0; j < newSize; j++) {
        C[i][j] = C11[i][j];
        C[i][j + newSize] = C12[i][j];
```

```
            C[i + newSize][j] = C21[i][j];
            C[i + newSize][j + newSize] = C22[i][j];
        }
    }
}

int main() {
    int n;
    int **A, **B, **C_standard, **C_strassen;

    printf("-- Matrix Multiplication --\n");
    printf("Enter size of matrices: ");
    scanf("%d", &n);

    A = allocate_matrix(n);
    B = allocate_matrix(n);
    C_standard = allocate_matrix(n);
    C_strassen = allocate_matrix(n);

    printf("Input values for matrix A:\n");
    input_matrix(A, n);
    printf("Input values for matrix B:\n");
    input_matrix(B, n);

    // Standard matrix multiplication
    clock_t start_standard = clock();
    standard_matrix_multiply(A, B, C_standard, n);
    clock_t end_standard = clock();
    double time_standard = ((double)(end_standard - start_standard)) / CLOCKS_PER_SEC;

    // Strassen's matrix multiplication
    clock_t start_strassen = clock();
    strassen_matrix_multiply(A, B, C_strassen, n);
    clock_t end_strassen = clock();
    double time_strassen = ((double)(end_strassen - start_strassen)) / CLOCKS_PER_SEC;

    printf("\nResult Matrix C (Standard Multiplication):\n");
    print_matrix(C_standard, n);
    printf("\nResult Matrix C (Strassen's Multiplication):\n");
    print_matrix(C_strassen, n);

    printf("\nTime taken for standard multiplication: %f seconds\n", time_standard);
    printf("Time taken for Strassen's multiplication: %f seconds\n", time_strassen);

    // Determining threshold value for Strassen's algorithm
    printf("\nThreshold value for Strassen's algorithm:\n");
    printf("For n > %d, Strassen's algorithm becomes more efficient.\n", (int)(pow(2, 14)));

    // Free allocated memory
    for (int i = 0; i < n; i++) {
        free(A[i]);
        free(B[i]);
        free(C_standard[i]);
        free(C_strassen[i]);
    }
    free(A);
```

```
    free(B);
    free(C_standard);
    free(C_strassen);

    return 0;
}
```

Output:

```
Sameeksha Gupta 22BCP343
-- Matrix Multiplication --
Enter size of matrices: 2
Input values for matrix A:
Enter matrix values row by row:
Row 1: 1 2
Row 2: 3 4
Input values for matrix B:
Enter matrix values row by row:
Row 1: 1 2
Row 2: 3 4

Result Matrix C (Standard Multiplication):
7 10
15 22

Result Matrix C (Strassen's Multiplication):
7 10
15 22

Time taken for standard multiplication: 0.000000 seconds
Time taken for Strassen's multiplication: 0.001000 seconds
```

7. Floyd Warshell

```
#include <stdio.h>
#define INF 999

void printMatrix(int matrix[][100], int n);
// Implementing Floyd-Warshall algorithm
void floydWarshall(int graph[][100], int n) {
    int matrix[100][100], i, j, k;

    for (i = 0; i < n; i++)
        for (j = 0; j < n; j++)
            matrix[i][j] = graph[i][j];

    // Adding vertices individually
    for (k = 0; k < n; k++) {
        for (i = 0; i < n; i++) {
            for (j = 0; j < n; j++) {
                if (matrix[i][k] + matrix[k][j] < matrix[i][j])
                    matrix[i][j] = matrix[i][k] + matrix[k][j];
            }
        }
    }
    printMatrix(matrix, n);
}

void printMatrix(int matrix[][100], int n) {
    printf("Shortest distances between every pair of vertices:\n");
    for (int i = 0; i < n; i++) {
        for (int j = 0; j < n; j++) {
            if (matrix[i][j] == INF)
                printf("%5s", "INF");
            else
                printf("%5d", matrix[i][j]);
        }
        printf("\n");
    }
}

int main() {
    int n;
    printf("--Sameeksha Gupta 22BCP343-\n");
    printf("Enter the number of vertices: ");
    scanf("%d", &n);

    int graph[100][100];

    printf("Enter the adjacency matrix (enter %d rows, each containing %d elements, use 'INF' for
infinity):\n", n, n);
```

```
    for (int i = 0; i < n; i++) {
        for (int j = 0; j < n; j++) {
            char input[10];
            scanf("%s", input);


            if (strcmp(input, "INF") == 0)
                graph[i][j] = INF;
            else
                graph[i][j] = atoi(input);
        }
    }

    floydWarshall(graph, n);

    return 0;
}
```

Output:

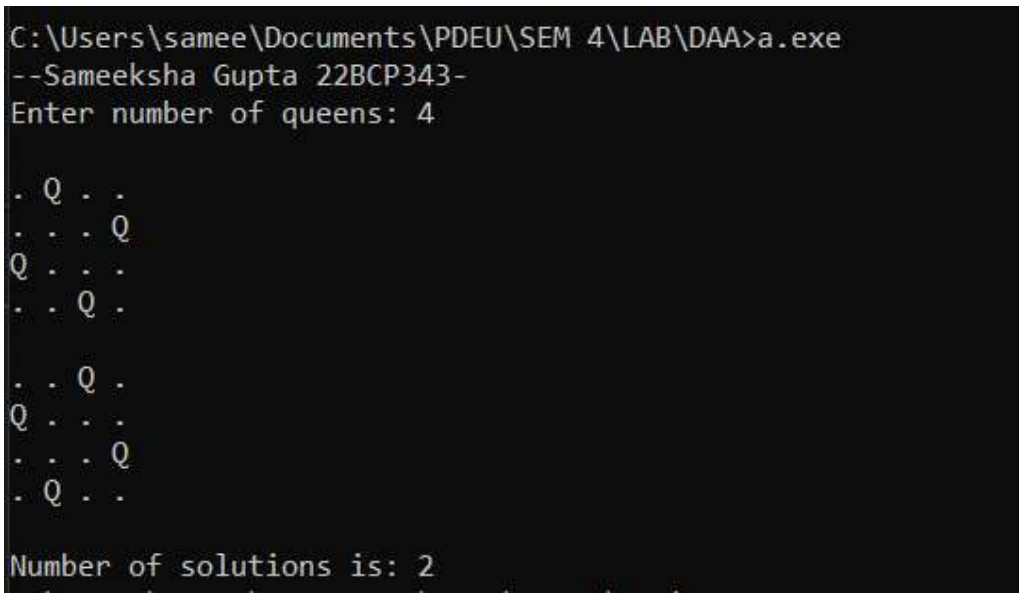
```
C:\Users\samee\Documents\PDEU\SEM 4\LAB\DAA>a.exe
Sameeksha Gupta
Enter the number of vertices: 4
Enter the adjacency matrix (enter 4 rows, each containing 4 elements, use 'INF' for infinity):
0 5 INF 10
INF 0 3 INF
INF INF 0 1
INF INF INF 0
Shortest distances between every pair of vertices:
    0    5    8    9
  INF    0    3    4
  INF  INF    0    1
  INF  INF  INF    0
```

8) N-Queen problem:
Code:

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

typedef struct {
    char **matrix;
    int size;
} Matrix;

Matrix createMatrix(int n) {
    Matrix m = {NULL, n};
    m.matrix = (char **)malloc(m.size * sizeof(char *));
    for (int i = 0; i < m.size; i++) {
        m.matrix[i] = (char *)malloc(m.size * sizeof(char));
    }
    for (int i = 0; i < m.size; i++) {
        for (int j = 0; j < m.size; j++) {
            m.matrix[i][j] = '.';
        }
    }
    return m;
}

bool isSafe(int row, int col, Matrix *board) {
    // check column above the cell
    for (int i = row - 1; i > -1; i--) {
        if (board->matrix[i][col] == 'Q')
            return false;
    }
    // check left diagonal above cell
    for (int i = row - 1, j = col - 1; i > -1 && j > -1; i--, j--) {
        if (board->matrix[i][j] == 'Q')
            return false;
    }
    // check right diagonal above cell
    for (int i = row - 1, j = col + 1; i > -1 && j < board->size; i--, j++) {
        if (board->matrix[i][j] == 'Q')
            return false;
    }
    return true;
}

int backtrack(int row, Matrix *board, int n, int count) {
    if (n == 0) {
        printf("\n");
```

```
        for (int i = 0; i < board->size; i++) {
            for (int j = 0; j < board->size; j++) {
                printf("%c ", board->matrix[i][j]);
            }
            printf("\n");
        }
        return count + 1;
    }
    for (int i = row; i < board->size; i++) {
        for (int j = 0; j < board->size; j++) {
            if (isSafe(i, j, board)) {
                board->matrix[i][j] = 'Q';
                count = backtrack(i + 1, board, n - 1, count);
                board->matrix[i][j] = '.';
            }
        }
    }
    return count;
}

int main() {
    int n;
    printf("--Sameeksha Gupta 22BCP343-\n");
    printf("Enter number of queens: ");
    scanf("%d", &n);
    Matrix board = createMatrix(n);
    int count = backtrack(0, &board, n, 0);
    printf("\nNumber of solutions is: %d", count);
    return 0;
}
```

Output:


C:\Users\samee\Documents\PDEU\SEM 4\LAB\DAA>a.exe
--Sameeksha Gupta 22BCP343-
Enter number of queens: 4
. Q . .
. . . Q
Q . . .
. . Q .
. . Q .
Q . . .
. . . Q
. Q . .
Number of solutions is: 2

9.)TSP

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

#define N 10

int n;
int graph[N][N];
int minCost = INT_MAX;
int finalPath[N];


int calculateLB(int path[], int level, int visited[])
{
   int lb = 0;
   for (int i = 0; i < n; i++)
   {
      if (!visited[i])
      {
         int minDist = INT_MAX;
         for (int j = 0; j < n; j++)
         {
            if (!visited[j] && graph[i][j] < minDist)
            {
               minDist = graph[i][j];
            }
         }
         lb += minDist;
      }
   }
   return lb;
}

void tsp(int level, int cost, int path[], int visited[])
{
   if (level == n)
   {
     if (graph[path[level - 1]][0] != 0)
      {
         int totalCost = cost + graph[path[level - 1]][0]; // Total cost of the tour
        if (totalCost < minCost)
         {
            minCost = totalCost;
            for (int i = 0; i < n; i++)
            {
```

```
                finalPath[i] = path[i];
            }
        }
    }
    return;
}

for (int i = 0; i < n; i++)
{
    if (!visited[i] && graph[path[level - 1]][i] != 0)
    {
        visited[i] = 1;
        path[level] = i;
        int lowerBound = calculateLB(path, level, visited);
        if (cost + graph[path[level - 1]][i] + lowerBound < minCost)
        {
            tsp(level + 1, cost + graph[path[level - 1]][i], path, visited); // Recur for city i
        }
        visited[i] = 0;
    }
}
}

int main()
{
    printf("--Sameeksha Gupta 22BCP343-”);
    printf("Enter the number of cities: ");
    scanf("%d", &n);

    printf("Enter the distances between the cities in the format 'source city, destination city,
distance':\n");

    int source, destination, distance;
    for (int i = 0; i < n * (n - 1) / 2; i++)
    {
        scanf("%d %d %d", &source, &destination, &distance);
        graph[source][destination] = distance;
        graph[destination][source] = distance;    }

    int path[N], visited[N] = {0};
    path[0] = 0;
    visited[0] = 1;

    tsp(1, 0, path, visited);
```

```
    printf("The minimum cost of the tour is: %d\n", minCost);
    printf("The tour path is: ");
    for (int i = 0; i < n; i++)
    {
        printf("%d ", finalPath[i]);
    }
    printf("%d\n", finalPath[0]);

    return 0;
}
```

Output:

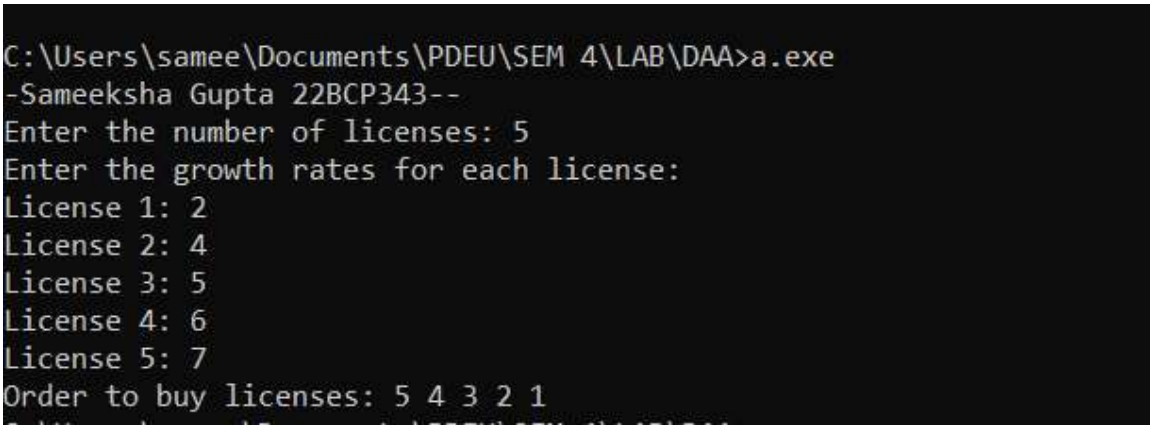
```
C:\Users\samee\Documents\PDEU\SEM 4\LAB\DAA>a.exe
--Sameeksha Gupta 22BCP343-
Enter the number of cities: 4
Enter the distances between the cities in the format 'source city, destination city, distance':
0 1 10
1 2 35
2 0 15
0 3 25
2 3 30
1 3 25
The minimum cost of the tour is: 80
The tour path is: 0 1 3 2 0
```

10.)
a)

```
#include <stdio.h>


struct License
{
   int id;
   float growth_rate;
};


void swap(struct License *a, struct License *b)
{
   struct License t = *a;
   *a = *b;
   *b = t;
}


int partition(struct License arr[], int low, int high)
{
   float pivot = arr[high].growth_rate;
   int i = (low - 1);

   for (int j = low; j <= high - 1; j++)
   {
      if (arr[j].growth_rate > pivot)
      {
         i++;
         swap(&arr[i], &arr[j]);
      }
   }
   swap(&arr[i + 1], &arr[high]);
   return (i + 1);
}

void quickSort(struct License arr[], int low, int high)
{
   if (low < high)
   {
      int pi = partition(arr, low, high);
      quickSort(arr, low, pi - 1);
      quickSort(arr, pi + 1, high);
```

```
    }
}

void buy_licenses(struct License licenses[], int n)
{
    quickSort(licenses, 0, n - 1);
    printf("Order to buy licenses: ");
    for (int i = 0; i < n; i++)
    {
        printf("%d ", licenses[i].id);
    }
}

int main()
{
    printf("Name: Sameeksha Gupta\nRoll No.: 22BCP343\n");
    int n;
    printf("Enter the number of licenses: ");
    scanf("%d", &n);

    struct License licenses[n];
    printf("Enter the growth rates for each license:\n");
    for (int i = 0; i < n; i++)
    {
        licenses[i].id = i + 1;
        printf("License %d: ", i + 1);
        scanf("%f", &licenses[i].growth_rate);
    }

    buy_licenses(licenses, n);

    return 0;
}
```

Output:


C:\Users\samee\Documents\PDEU\SEM 4\LAB\DAA>a.exe
-Sameeksha Gupta 22BCP343--
Enter the number of licenses: 5
Enter the growth rates for each license:
License 1: 2
License 2: 4
License 3: 5
License 4: 6
License 5: 7
Order to buy licenses: 5 4 3 2 1

10)
b)

```
#include <stdio.h>

void find_p(int array[], int start, int end)
{
   if (start == end)
   {
      printf("Peak value is: %d\n", array[start]);
      return;
   }

   int mid = start + (end - start) / 2;

   if (array[mid] > array[mid + 1])
      find_p(array, start, mid);
   else
      find_p(array, mid + 1, end);
}

int main()
{
   printf("Name: Sameeksha Gupta\nRoll No.: 22BCP343\n");
   int n = 0;
   printf("Enter number of elements of array: ");
   scanf("%d", &n);

   int array[n];
   printf("Enter numbers: ");
   for (int i = 0; i < n; i++)
   {
      scanf("%d", &array[i]);
   }

   find_p(array, 0, n - 1);

   return 0;
}
```

Output:

```
C:\Users\samee\Documents\PDEU\SEM 4\LAB\DAA>a.exe
Name: Sameeksha Gupta
Roll No.: 22BCP343
Enter number of elements of array: 5
Enter numbers: 2
4
1
6
3
Peak value is: 6
```