# **EXPERIMENT 1**

## **20CP209P – Design and Analysis of Algorithm Lab**

## **Aim:**

Implement Insertion Sort and Selection Sort and give complexity analysis

## **Code:**

### **Insertion Sort:**

#include <stdio.h>

#include <time.h>

void insertion\_sort(int arr[], int len);

int main(void)

{

    clock\_t start, end;

    int arr[] = {7,4,8,9,0,1,2,5,3,6};

    int len = sizeof(arr) / sizeof(int);

    start = clock();

    insertion\_sort(arr, len);

    end = clock();

    for (int i = 0; i < len; i++)

    {

        printf("%d ", i);

    }

    printf("\n");

    printf("time taken for execution: ", (double) (end - start));

    return 0;

}

void insertion\_sort(int arr[], int len)

{

    for (int i = 1; i < n; i++)

    {

        int key = arr[i];

        int j = i - 1;

        while (j >= 0 && arr[j] > key)

        {

            arr[j + 1] = arr[j];

            j--;

        }

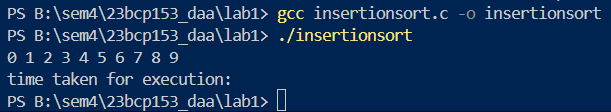
        arr[j + 1] = key;

    }

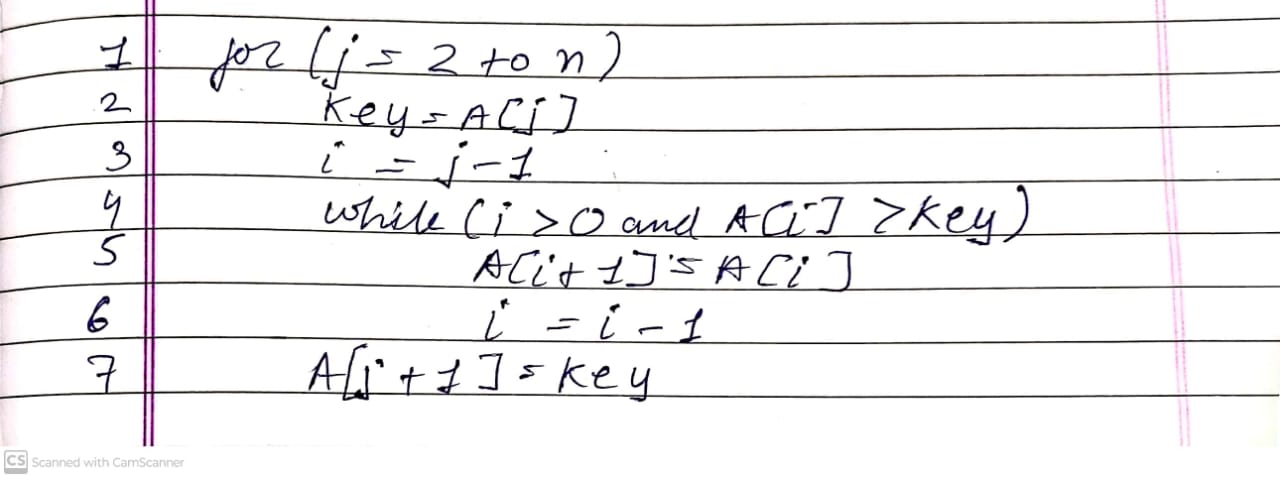
    return;

}

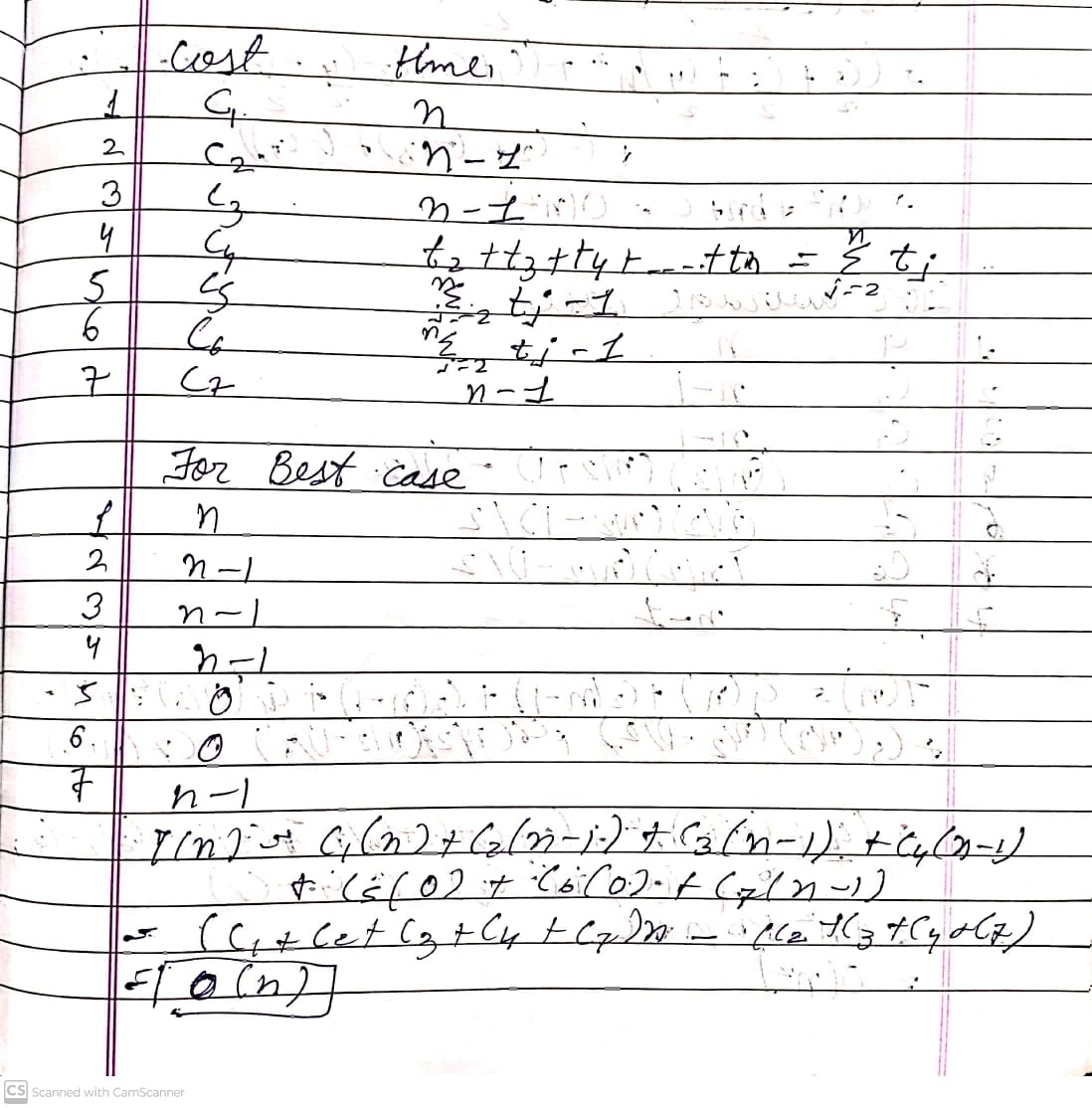
## **Output:**

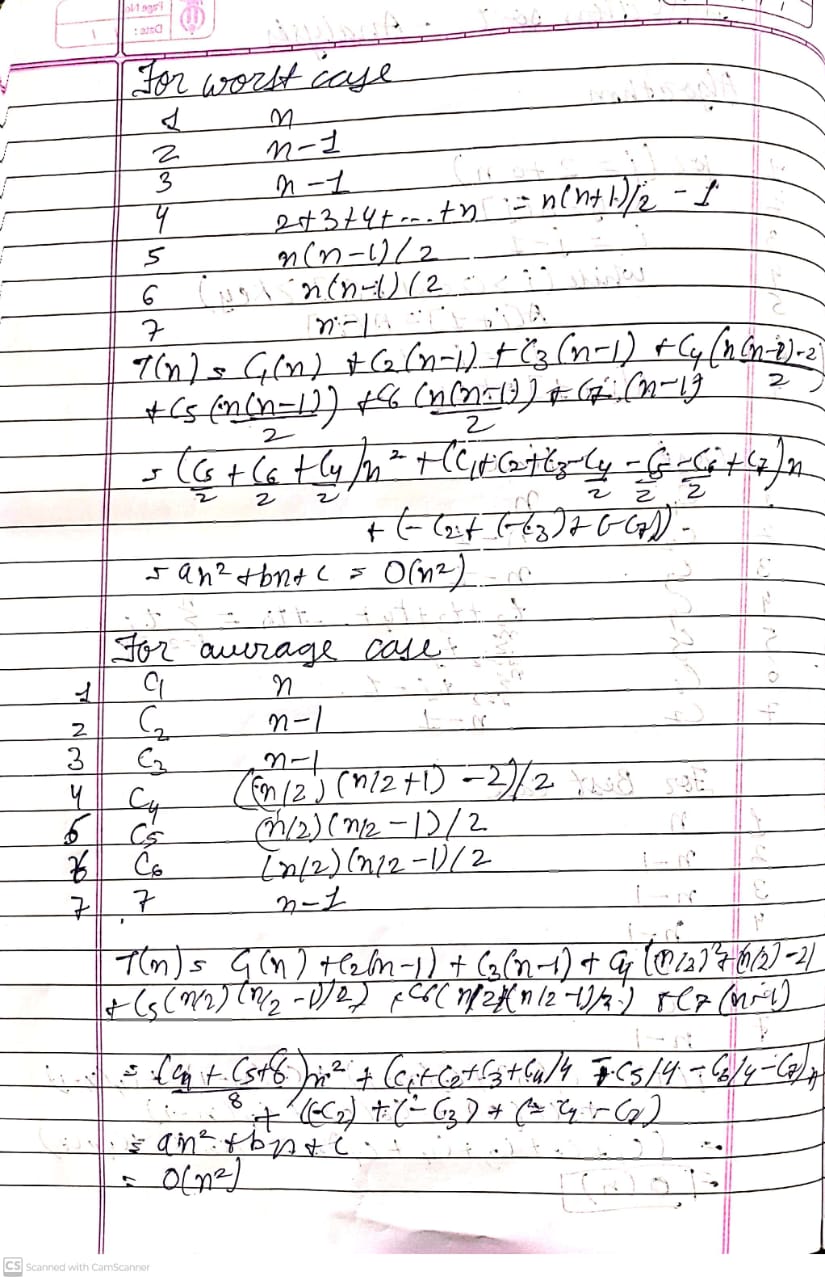


## **Algorithm:**



## **Complexity Analysis:**





## **Code:**

### **Selection Sort:**

#include <stdio.h>

#include <time.h>

void selection\_sort(int arr[], int len);

int main(void)

{

    clock\_t start, end;

    int arr[] = {7,4,8,9,0,1,2,5,3,6};

    int len = sizeof(arr) / sizeof(int);

    start = clock();

    selection\_sort(arr, len);

    end = clock();

    for (int i = 0; i < len; i++)

    {

        printf("%d ", i);

    }

    printf("\n");

    printf("time taken for execution: %f", (double) (end - start));

    return 0;

}

void selection\_sort(int arr[], int n)

{

    for (int i = 0; i < n; i++)

    {

        int min = i;

        for (int j = i; j < n; j++)

        {

            if (arr[j] < arr[min])

            {

                min = j;

            }

        }

        if (min != i)

        {

            int temp = arr[i];

            arr[i] = arr[min];

            arr[min] = temp;

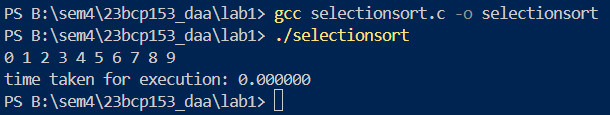
        }

    }

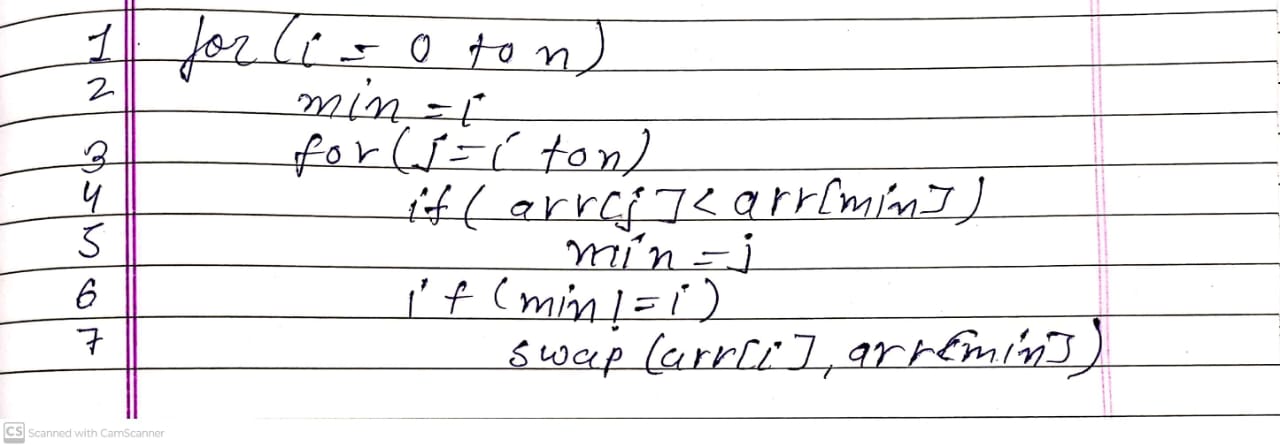
    return;

}

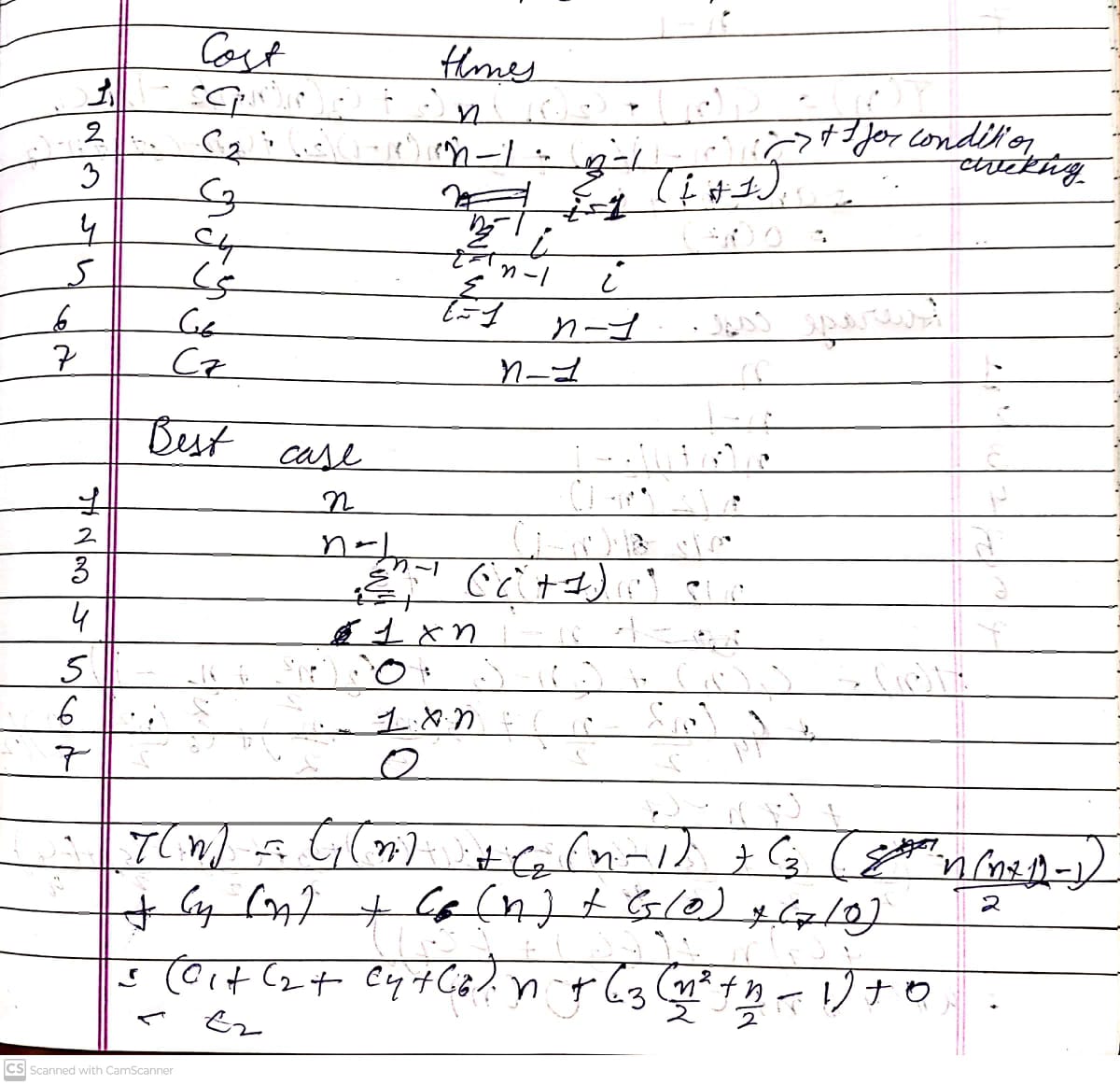
## **Output:**

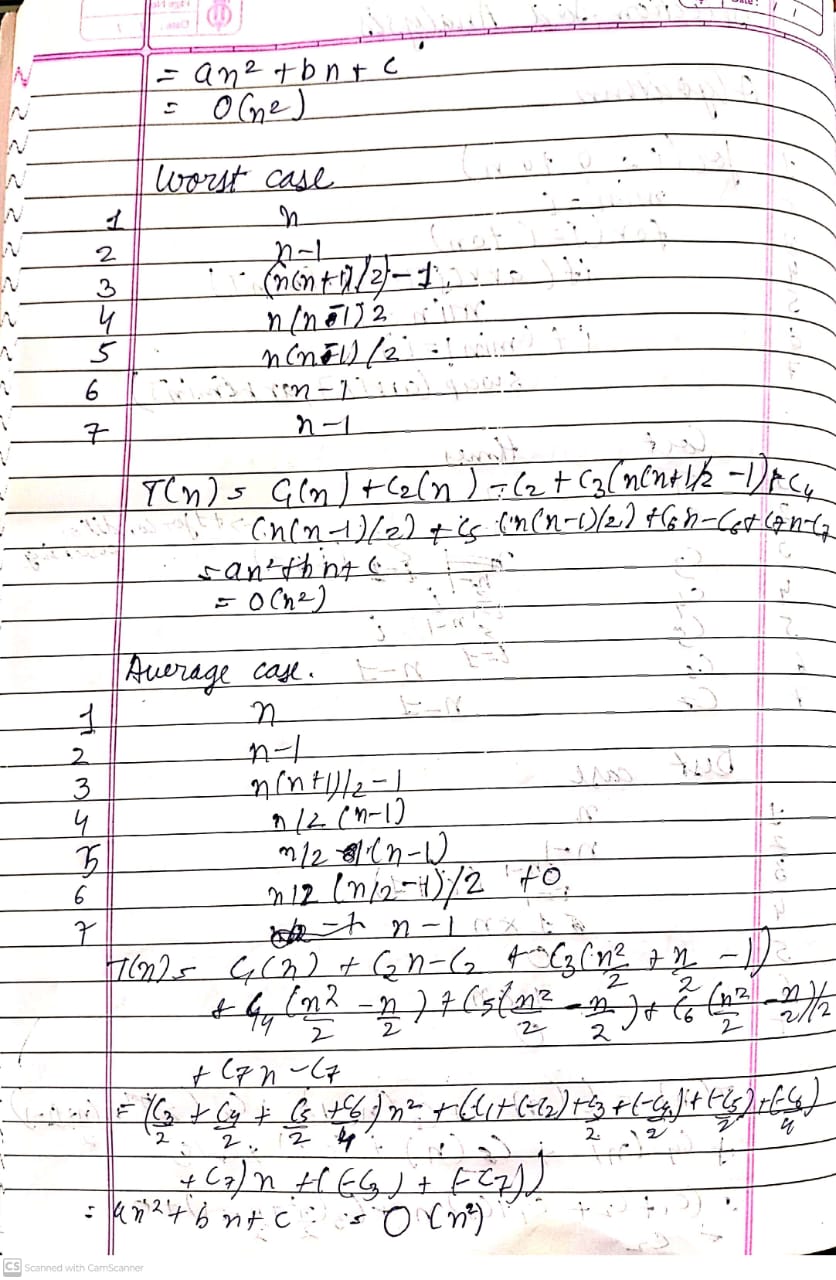


## **Algorithm:**



## **Complexity Analysis:**





# **EXPERIMENT 2**

## **20CP209P – Design and Analysis of Algorithm Lab**

## **Aim:**

Implement Merge Sort and Quick Sort and give complexity analysis

## **Code:**

### **Merge Sort:**

#include <stdio.h>

#include <time.h>

void merge\_sort(int arr[], int low, int high);

void merge(int arr[], int low, int mid, int high);

int main(void)

{

    clock\_t start, end;

    int arr[] = {7, 4, 8, 9, 0, 1, 2, 5, 3, 6};

    int len = sizeof(arr) / sizeof(int);

    int low = 0;

    int high = len - 1;

    start = clock();

    merge\_sort(arr, low, high);

    end = clock();

    for (int i = 0; i < len; i++)

    {

        printf("%d ", arr[i]);

    }

    printf("\n");

    printf("Time taken for execution: %f seconds\n", (double)(end - start) / CLOCKS\_PER\_SEC);

    return 0;

}

void merge\_sort(int arr[], int low, int high)

{

    if (low < high)

    {

        int mid = (low + high) / 2;

        merge\_sort(arr, low, mid);

        merge\_sort(arr, mid + 1, high);

        merge(arr, low, mid, high);

    }

}

void merge(int arr[], int low, int mid, int high)

{

    int i = low;

    int j = mid + 1;

    int k = 0;

    int arrB[high - low + 1];

    while (i <= mid && j <= high)

    {

        if (arr[i] <= arr[j])

        {

            arrB[k] = arr[i];

            i++; k++;

        }

        else

        {

            arrB[k] = arr[j];

            j++; k++;

        }

    }

    if (i > mid)

    {

        while (j <= high)

        {

            arrB[k] = arr[j];

            j++; k++;

        }

    }

    else if (j > high)

    {

        while (i <= mid)

        {

            arrB[k] = arr[i];

            i++; k++;

        }

    }

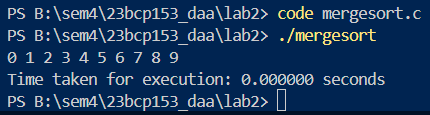
    for (int x = 0; x < (high - low + 1); x++)

        arr[low + x] = arrB[x];

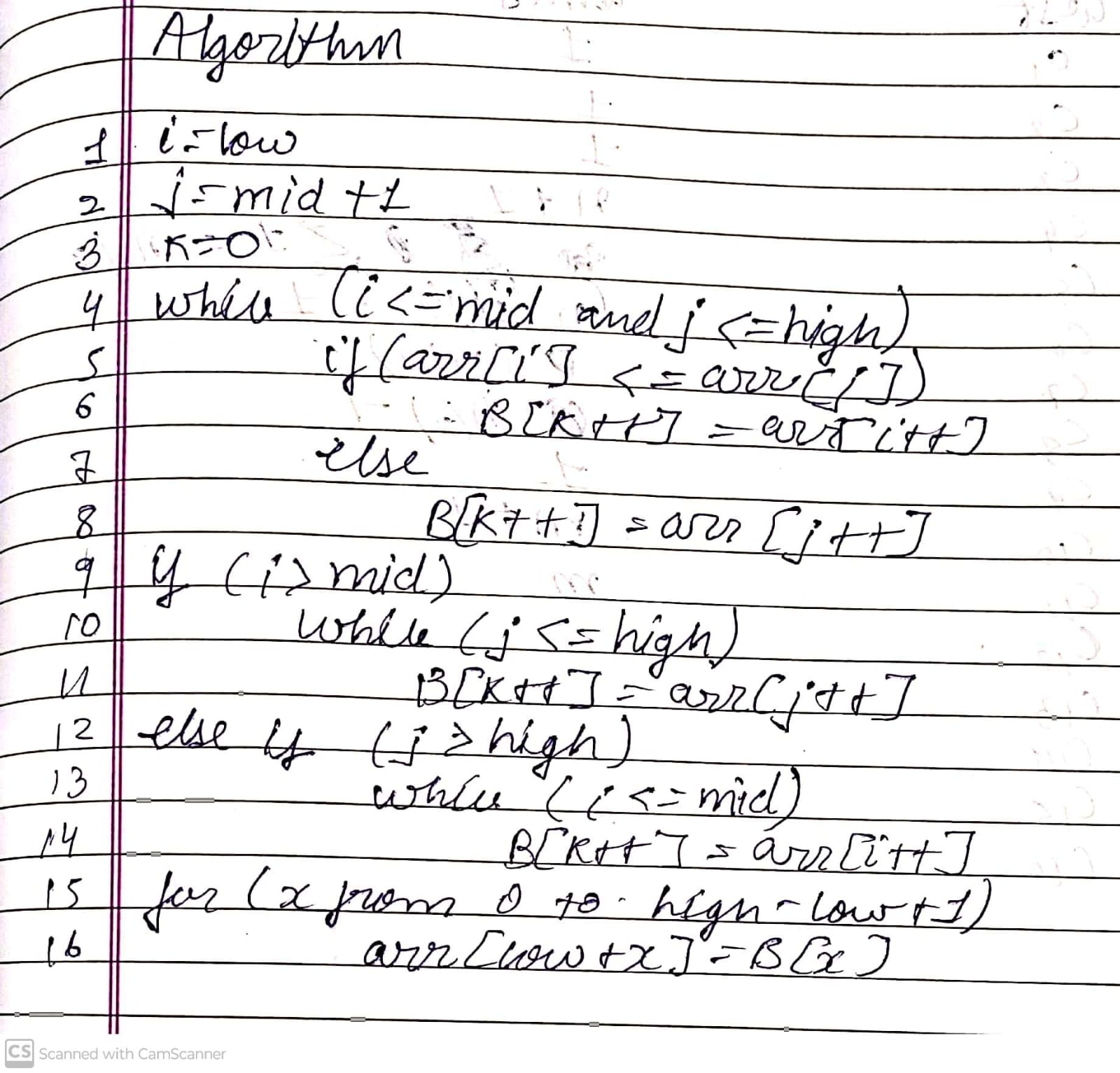
    return;

}

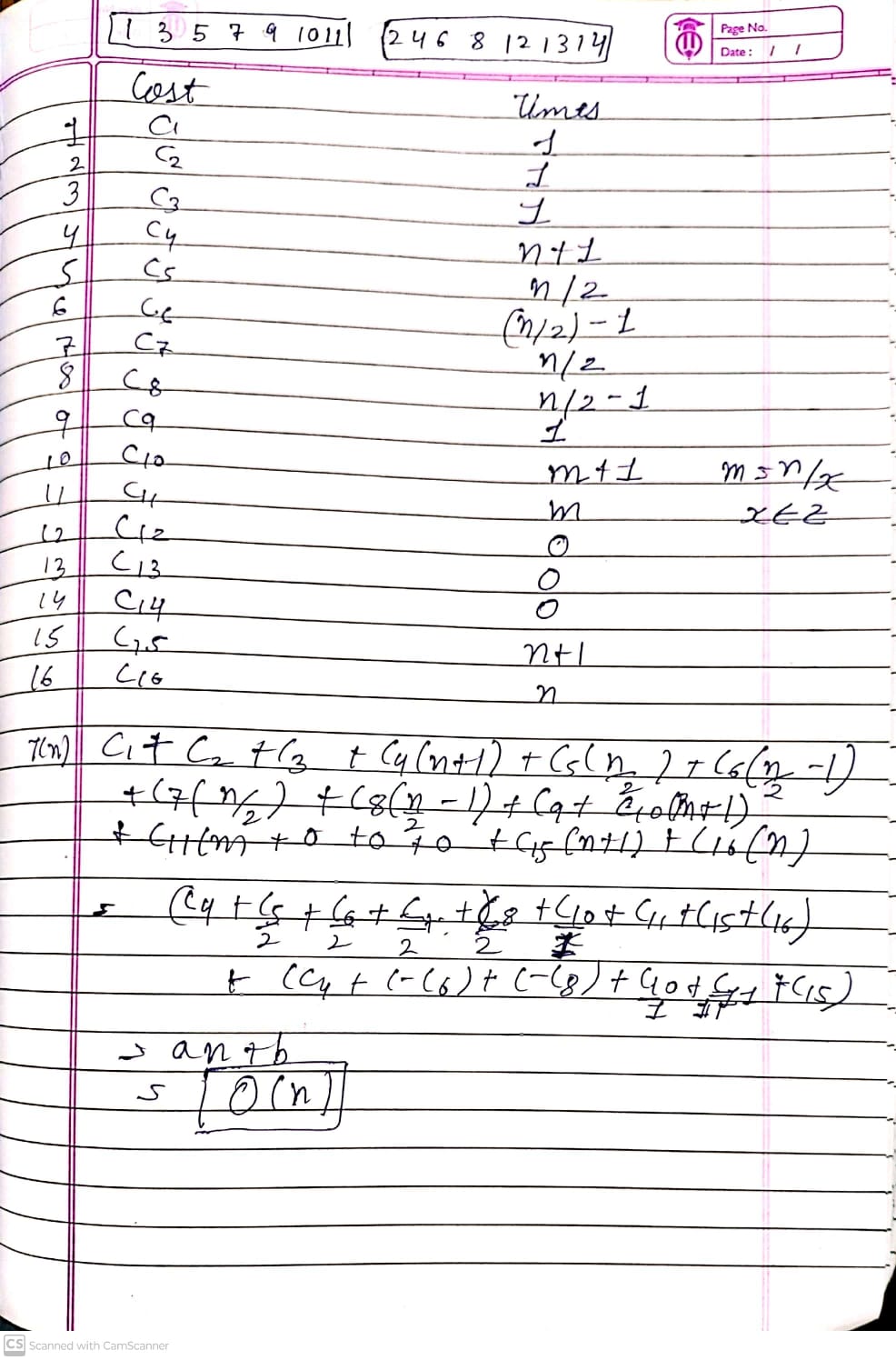
## **Output:**



## **Algorithm:**



## **Complexity Analysis:**



## **Code:**

### **Quick Sort:**

#include <stdio.h>

#include <time.h>

void quick\_sort(int arr[], int low, int high);

int partition(int arr[], int low, int high);

int main(void)

{

    clock\_t start, end;

    // int arr[] = {7, 4, 8, 9, 0, 1, 2, 5, 3, 6};

    int arr[] = {9, 8, 7, 6, 5, 4, 3, 2, 1, 0};

    int len = sizeof(arr) / sizeof(int);

    int low = 0;

    int high = len - 1;

    start = clock();

    quick\_sort(arr, low, high);

    end = clock();

    for (int i = 0; i < len; i++)

    {

        printf("%d ", arr[i]);

    }

    printf("\n");

    printf("Time taken for execution: %f seconds\n", (double)(end - start) / CLOCKS\_PER\_SEC);

    return 0;

}

void quick\_sort(int arr[], int low, int high)

{

    if (low < high)

    {

        int location = partition(arr, low, high);

        quick\_sort(arr, low, location - 1);

        quick\_sort(arr, location + 1, high);

    }

}

int partition(int arr[], int low, int high)

{

    int pivot = arr[low];

    int i = low; // i is start in lab algo

    int j = high; // j is end in lab algo

    while (i < j)

    {

        while (arr[i] <= pivot && i <= high)

            i++;

        while (arr[j] > pivot && j >= low)

            j--;

        if (i < j)

        {

            int temp = arr[i];

            arr[i] = arr[j];

            arr[j] = temp;

        }

    }

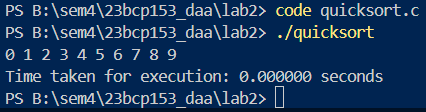
    arr[low] = arr[j];

    arr[j] = pivot;

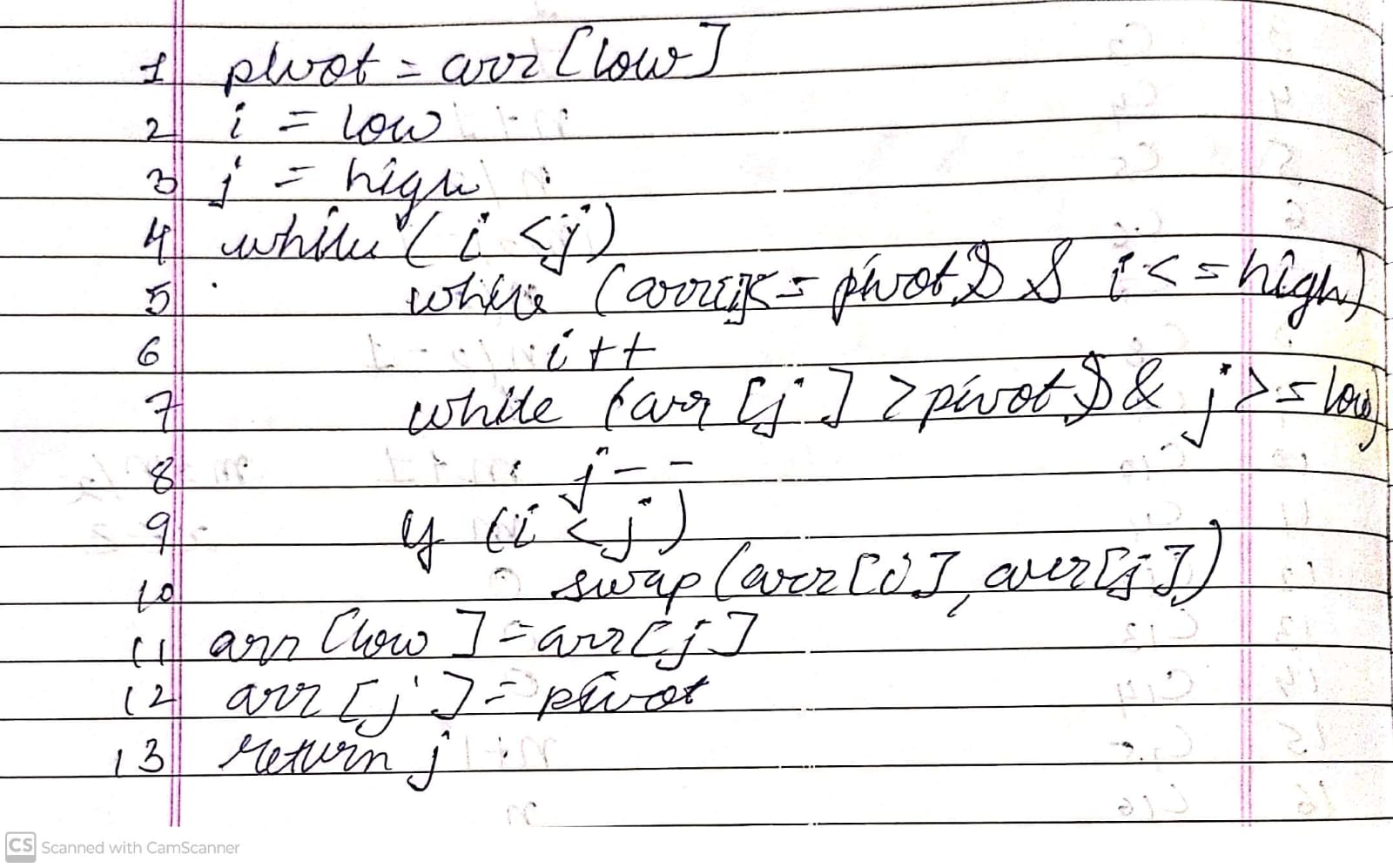
    return j;

}

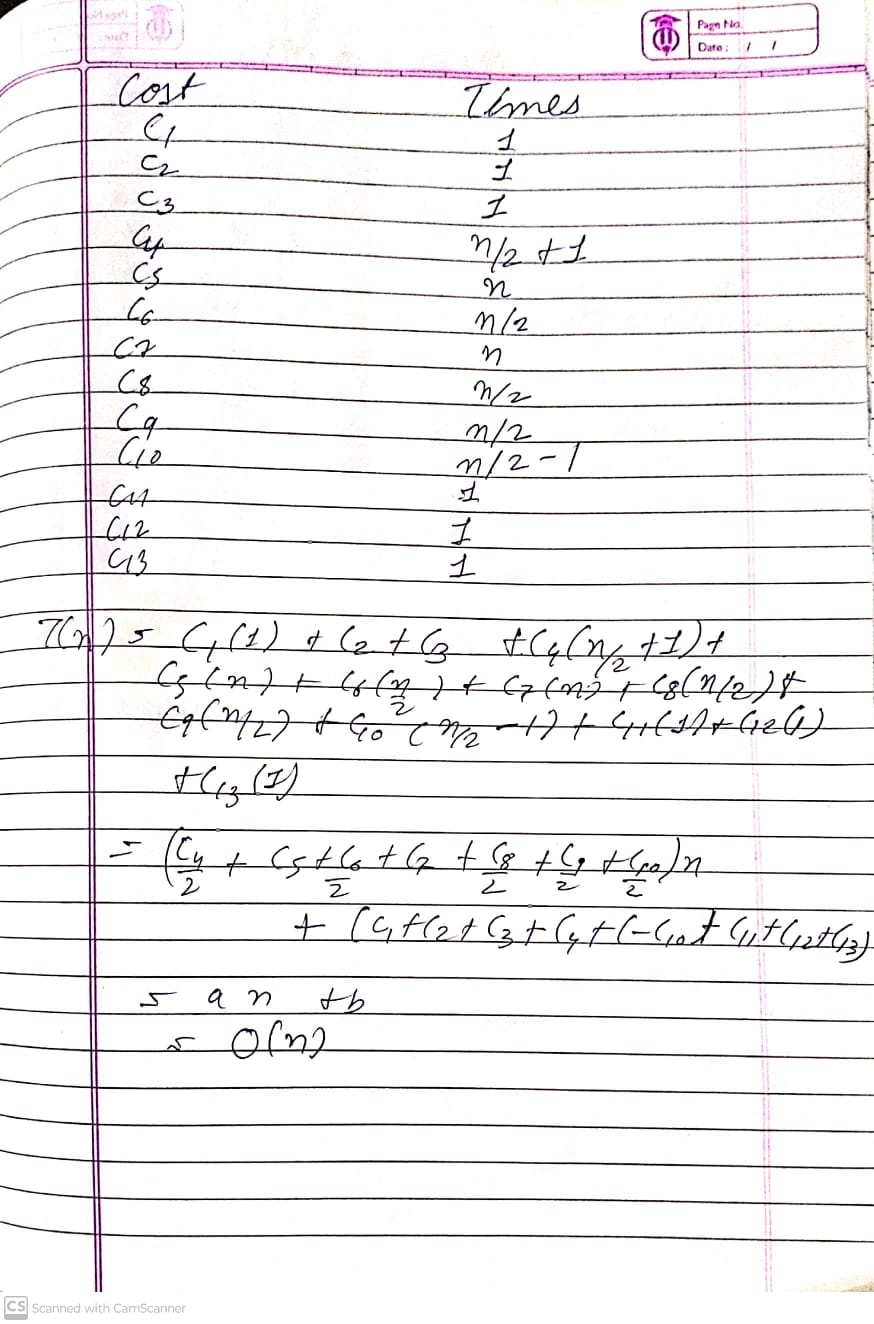
## **Output:**



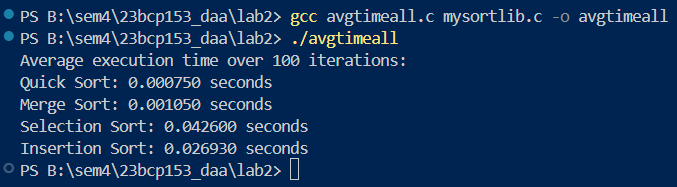
## **Algorithm:**



## **Complexity Analysis:**



## **Time Analysis of all sorting algorithms:**



Selection takes most time

Quick Sort takes least time

# **EXPERIMENT 3**

## **20CP209P – Design and Analysis of Algorithm Lab**

## **Aim:**

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:**

### **Addition:**

Node\* lladditer(Node\* head1, Node\* head2, int carry)

{

    Node\* revhead1 = reverselist(head1);

    // printlist(revhead1);

    Node\* revhead2 = reverselist(head2);

    // printlist(revhead2);

    Node\* result = NULL;

    int sum;

    while (revhead1 != NULL || revhead2 != NULL || carry)

    {

        sum = carry;

        if (revhead1)

        {

            sum += revhead1->data;

            revhead1 = revhead1->next;

        }

        if (revhead2)

        {

            sum += revhead2->data;

            revhead2 = revhead2->next;

        }

        carry = sum / 10;

        Node\* newnode = create\_node(sum % 10);

        newnode->next = result;

        result = newnode;

    }

    // printlist(result);

    if (result != NULL)

    {

        int size = 0;

        Node\* temp = result;

        while (temp != NULL) {

            size++;

            temp = temp->next;

        }

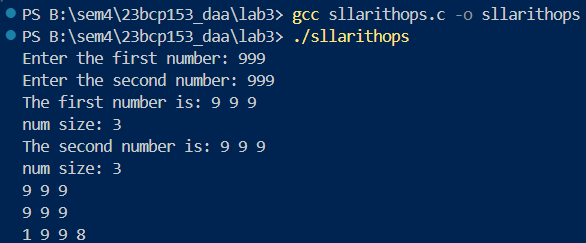
        result->size = size;

    }

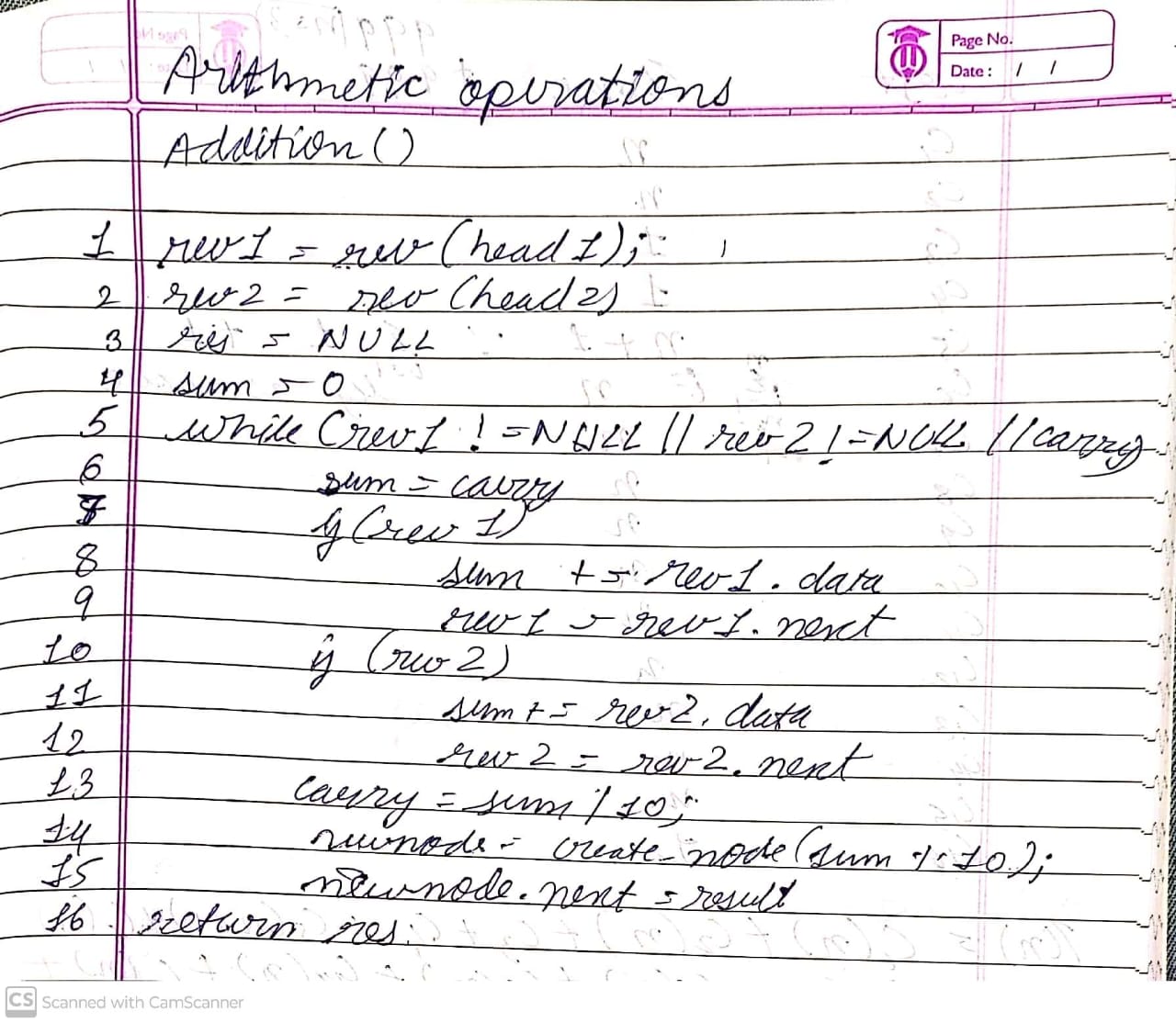
    return result;

}

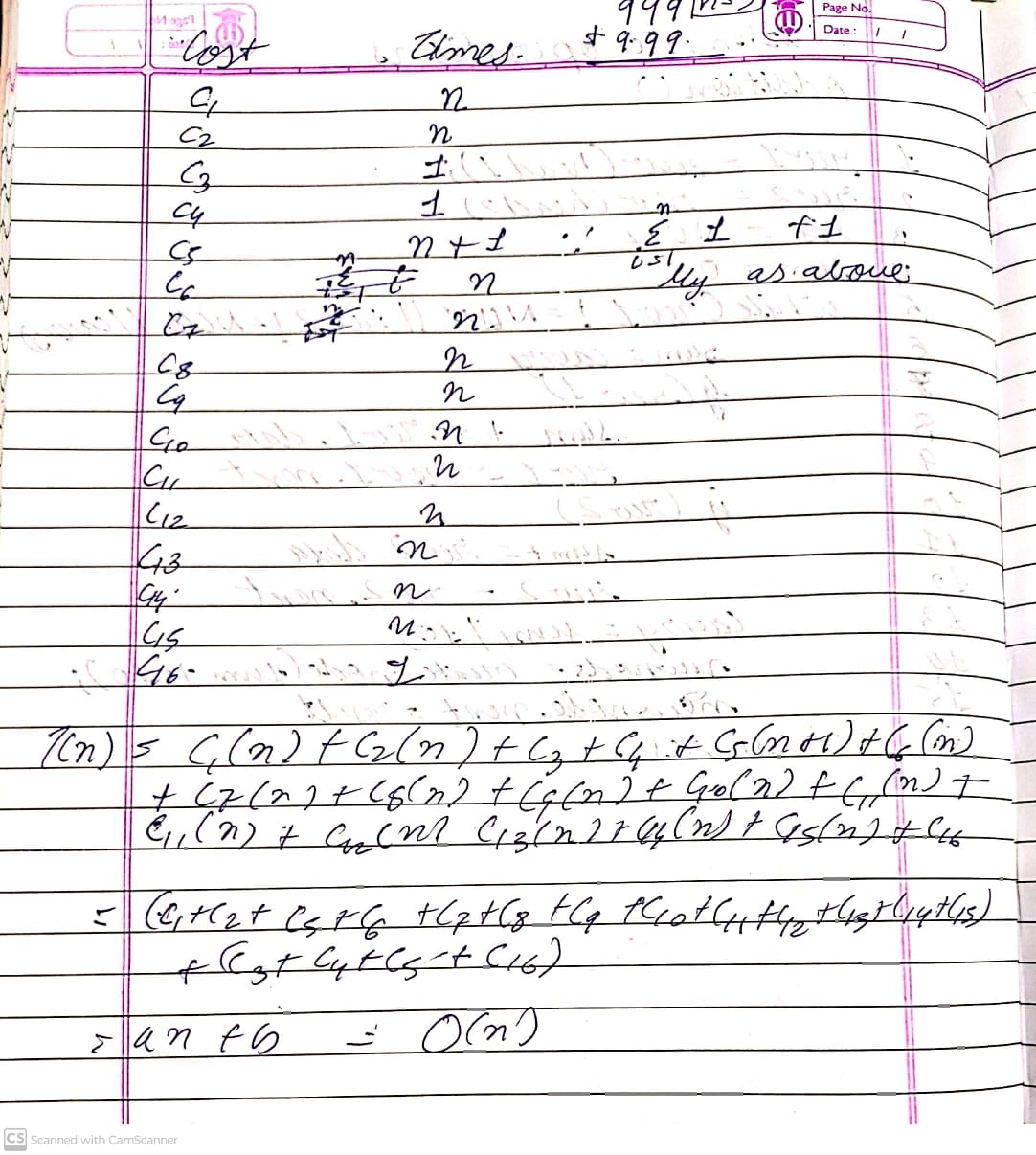
## **Output:**



## **Algorithm:**



## **Complexity Analysis:**



## **Code:**

### **Subtraction:**

Node\* llsubiter(Node\* head1, Node\* head2, int borrow)

{

    Node\* revhead1 = reverselist(head1);

    // printlist(revhead1);

    Node\* revhead2 = reverselist(head2);

    // printlist(revhead2);

    Node\* result = NULL;

    int diff;

    while (revhead1 || revhead2)

    {

        diff = borrow;

        if (revhead1)

        {

            diff += revhead1->data;

            revhead1 = revhead1->next;

        }

        if (revhead2)

        {

            if (diff >= revhead2->data)

            {

                diff -= revhead2->data;

            }

            else

            {

                borrow = -1;

                int fordiff = 10 - revhead2->data;

                diff += fordiff;

            }

            revhead2 = revhead2->next;

        }

        Node\* newnode = create\_node(diff);

        newnode->next = result;

        result = newnode;

    }

    if (result != NULL)

    {

        int size = 0;

        Node\* temp = result;

        while (temp != NULL) {

            size++;

            temp = temp->next;

        }

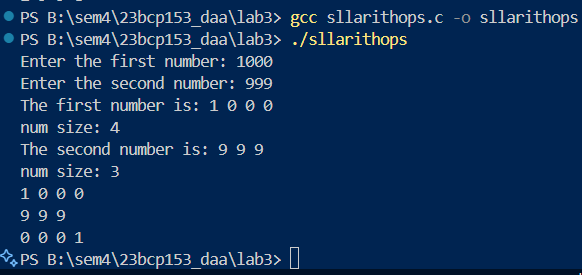
        result->size = size;

    }

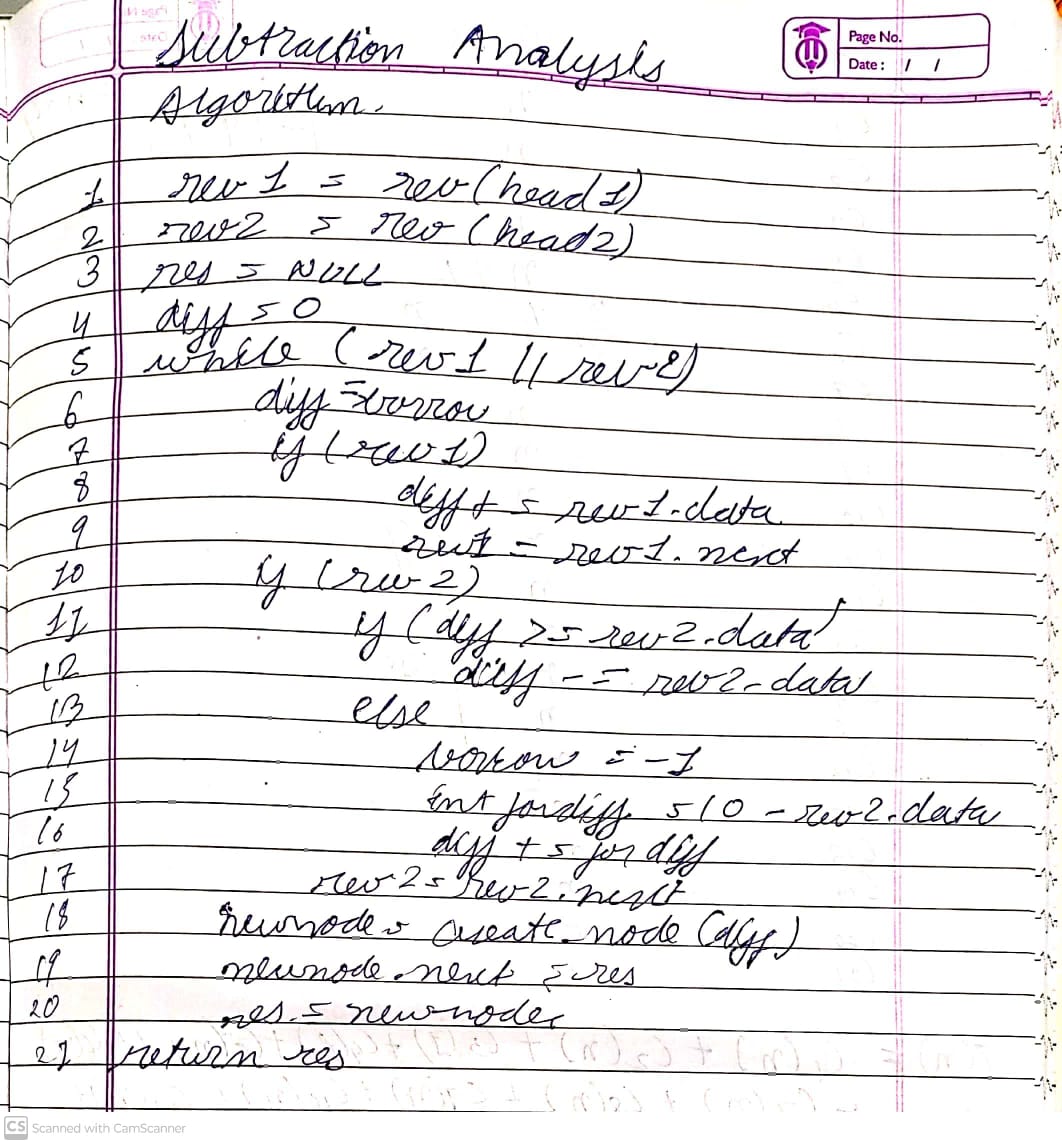
    return result;

}

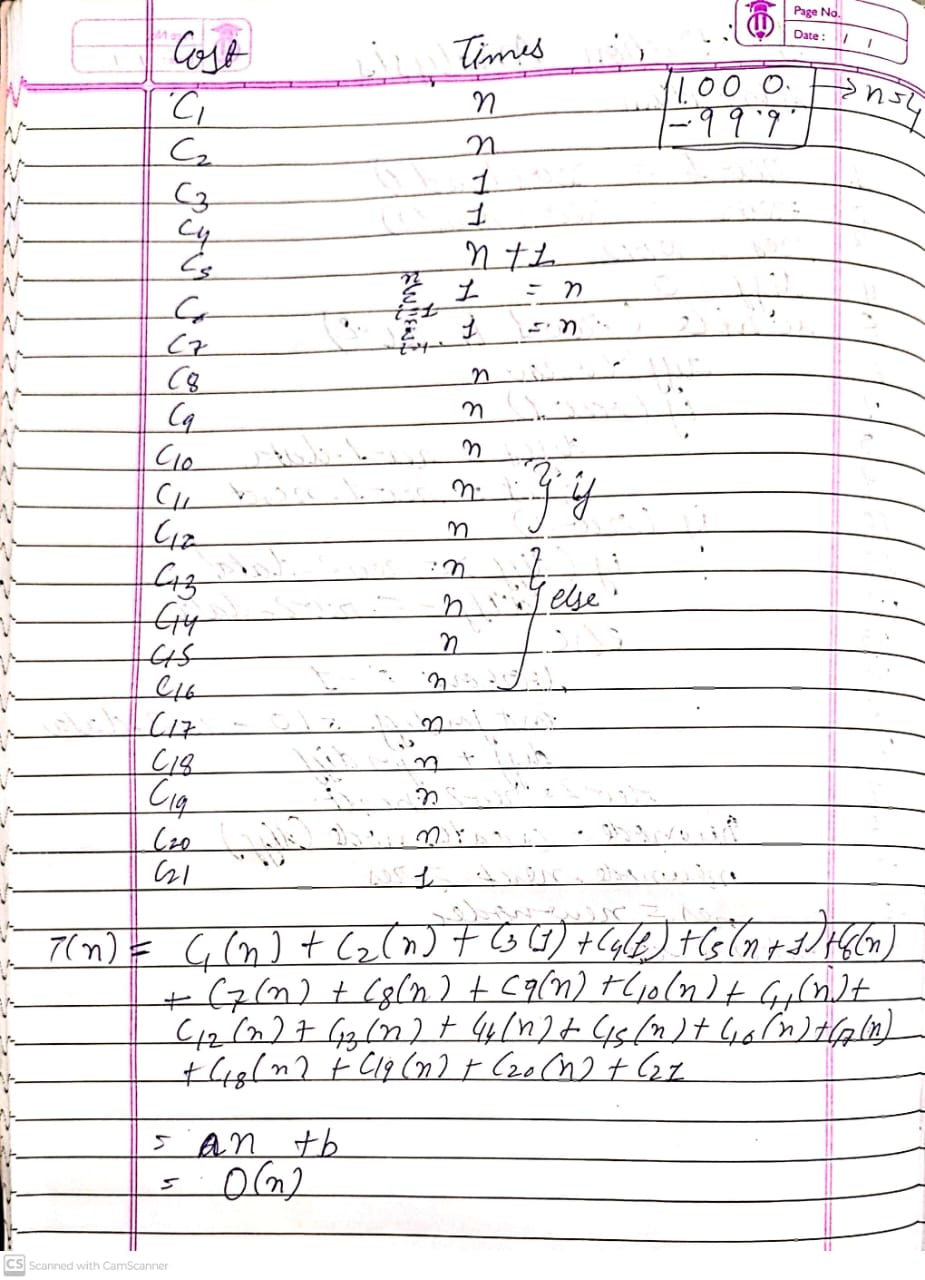
## **Output:**



## **Algorithm:**



## **Complexity Analysis:**



## **Code:**

### **Multiplication:**

Node\* llmuliter(Node\* head1, Node\* head2, int carry)

{

    Node\* revhead1 = reverselist(head1);

    // printlist(revhead1);

    Node\* revhead2 = reverselist(head2);

    // printlist(revhead2);

    Node\* final\_result = NULL;

    int product;

    Node\* trav1 = revhead1;

    Node\* trav2 = revhead2;

    int pad\_count = -1;

    while (trav2)

    {

        Node\* result = NULL;

        // since we are adding null - we need to change the if condition inside padding function

        // result = add\_padding\_back(result, ++pad\_count);

        // so i just found out that the add padding back function was useless

        result = add\_padding(result, ++pad\_count);

        trav1 = revhead1;

        carry = 0;

        while (trav1)

        {

            // printlist(result);

            // result = add\_padding(result, ++pad\_count);

            product = carry;

            product += trav1->data \* trav2->data;

            carry = product / 10;

            Node\* newnode = create\_node(product % 10);

            newnode->next = result;

            result = newnode;

            trav1 = trav1->next;

        }

        // bhai carry to dekho

        if (carry > 0)

        {

            Node\* newnode = create\_node(carry);

            newnode->next = result;

            result = newnode;

        }

        // printlist(result);

        final\_result = lladditer(final\_result, result, 0);

        trav2 = trav2->next;

    }

    // somehow reverselist function is changing head1 and just keeping it to be its first node

    // i.e. for 123 - it is making it 1

    // so i am retrieving head1 again with the reversed list

    // although i don't think this is good practice - it is working

    // i suspect that this is due to the fact that in reverselist function we are taking current = head

    // which may be changing the head - unintentionally - deepseek - deepthink r1 can help identify that

    // another approach could be to first create a copy of the head and then reverse both the lists

    // also freelist function needs to be implemented: 22:34 04-02-2025

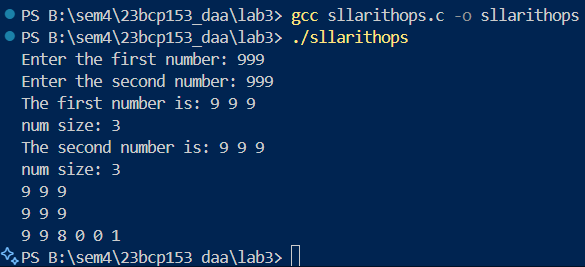
    head1 = reverselist(revhead1);

    head2 = reverselist(revhead2);

    return final\_result;

}

## **Output:**



## **Code:**

### **Exponential:**

Node\* llexpiter(Node\* head, int power)

{

    if (power < 0)

    {

        printf("Power less than 1 not supported");

        return NULL;

    }

    else if (power == 0)

    {

        return create\_node(1);

    }

    Node\* result = create\_node(1);

    for (int i = 0; i < power; i++)

    {

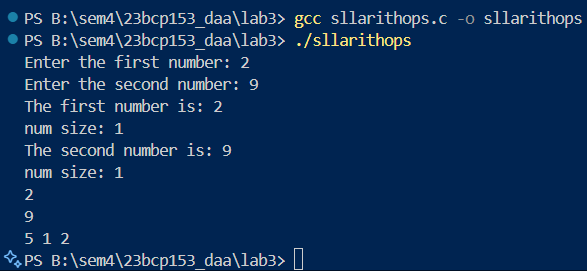
        result = llmuliter(result, head, 0);

    }

    return result;

}

## **Output:**



# **EXPERIMENT 4**

## **20CP209P – Design and Analysis of Algorithm Lab**

## **Aim:**

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?

## **Code:**

### **Array-Based:**

#include <stdio.h>

#include <stdlib.h>

#include <string.h>

#include <time.h>

#include <math.h>

typedef struct City {

    char\* name;

    int x;

    int y;

} City;

typedef struct Arrdb {

    City\*\* citiesarr;

    int size;

    int capacity;

} Arrdb;

City\* create\_city(char\* name, int x, int y);

Arrdb\* init\_arr\_db(int int\_cap);

void insert\_city\_arr(Arrdb\* arrdb, City\* city);

void delete\_by\_name(Arrdb\* arrdb, char\* name);

void delete\_by\_coordinates(Arrdb\* arrdb, int x, int y);

void print\_arr\_db(Arrdb\* arrdb);

void print\_within\_dist(Arrdb\* arrdb, int x, int y, int dist);

int main(void)

{

    Arrdb\* arrdb = init\_arr\_db(2);

    City\* city1 = create\_city("Ahmedabad", 6, 9);

    insert\_city\_arr(arrdb, city1);

    City\* city2 = create\_city("Mumbai", 9, 6);

    insert\_city\_arr(arrdb, city2);

    City\* city3 = create\_city("Delhi", 3, 4);

    insert\_city\_arr(arrdb, city3);

    City\* city4 = create\_city("MyCity", 5, 4);

    insert\_city\_arr(arrdb, city4);

    City\* city5 = create\_city("Kolkata", 9, 4);

    insert\_city\_arr(arrdb, city5);

    City\* city6 = create\_city("Chennai", 7, 9);

    insert\_city\_arr(arrdb, city6);

    City\* city7 = create\_city("Indore", 9, 9);

    insert\_city\_arr(arrdb, city7);

    print\_arr\_db(arrdb);

    delete\_by\_name(arrdb, "Delhi");

    print\_arr\_db(arrdb);

    delete\_by\_coordinates(arrdb, 9, 4);

    print\_arr\_db(arrdb);

    print\_within\_dist(arrdb, 5, 7, 10);

    return 0;

}

City\* create\_city(char\* name, int x, int y)

{

    City\* city = (City\*)malloc(sizeof(City));

    city->name = (char\*)malloc(strlen(name) + 1);

    city->name = name;

    city->x = x;

    city->y = y;

    return city;

}

Arrdb\* init\_arr\_db(int init\_cap)

{

    Arrdb\* arrdb = (Arrdb\*)malloc(sizeof(Arrdb));

    arrdb->citiesarr = (City\*\*)malloc(sizeof(City\*) \* init\_cap);

    arrdb->size = 0;

    arrdb->capacity = init\_cap;

}

void insert\_city\_arr(Arrdb\* arrdb, City\* city)

{

    if (arrdb->size == arrdb->capacity)

    {

        arrdb->citiesarr = (City\*\*)realloc(arrdb->citiesarr, sizeof(City\*) \* arrdb->capacity \* 2);

        arrdb->capacity \*= 2;

    }

    arrdb->citiesarr[arrdb->size] = city;

    arrdb->size++;

}

void delete\_by\_name(Arrdb\* arrdb, char\* name)

{

    if (arrdb->size == 0)

    {

        printf("Database is already empyt!");

        return;

    }

    for (int i = 0; i < arrdb->size - 1; i++)

    {

        if (strcmp(arrdb->citiesarr[i]->name, name) == 0)

        {

            free(arrdb->citiesarr[i]->name);

            // free(arrdb->citiesarr[i]->x);

            // free(arrdb->citiesarr[i]->y);

            // can't do the above as have not done malloc for the above i.e. not dynamically allocated (not pointers)

            free(arrdb->citiesarr[i]);

            for (int j = i; j < arrdb->size - 1; j++)

            {

                arrdb->citiesarr[j] = arrdb->citiesarr[j + 1];

            }

            arrdb->size--;

            printf("%s deleted successfully!\n", name);

            return;

        }

    }

    printf("City not found!\n");

    return;

}

void delete\_by\_coordinates(Arrdb\* arrdb, int x, int y)

{

    if (arrdb->size == 0)

    {

        printf("Database is already empyt!");

        return;

    }

    for (int i = 0; i < arrdb->size - 1; i++)

    {

        if (arrdb->citiesarr[i]->x == x && arrdb->citiesarr[i]->y == y)

        {

            char\* name = arrdb->citiesarr[i]->name;

            free(arrdb->citiesarr[i]->name);

            free(arrdb->citiesarr[i]);

            for (int j = i; j < arrdb->size - 1; j++)

            {

                arrdb->citiesarr[j] = arrdb->citiesarr[j + 1];

            }

            arrdb->size--;

            printf("%s deleted successfully! with coordinates (%d, %d)\n", name, x, y);

            return;

        }

    }

    return;

}

void print\_arr\_db(Arrdb\* arrdb)

{

    for (int i = 0; i < arrdb->size; i++)

    {

        printf("%s %d %d\n", arrdb->citiesarr[i]->name, arrdb->citiesarr[i]->x, arrdb->citiesarr[i]->y);

    }

    printf("Size of array database: %d\n", arrdb->size);

    printf("Capacity of array database: %d\n", arrdb->capacity);

}

void print\_within\_dist(Arrdb\* arrdb, int x, int y, int dist)

{

    if (arrdb->size == 0)

    {

        printf("Empty database");

        return;

    }

    printf("Cities within %d units of (%d, %d):\n", dist, x, y);

    for (int i = 0; i < arrdb->size; i++)

    {

        int diffx = arrdb->citiesarr[i]->x -x;

        int diffy = arrdb->citiesarr[i]->y -y;

        float distance = sqrt(pow(diffx, 2) - pow(diffy, 2));

        if (distance <= dist)

        {

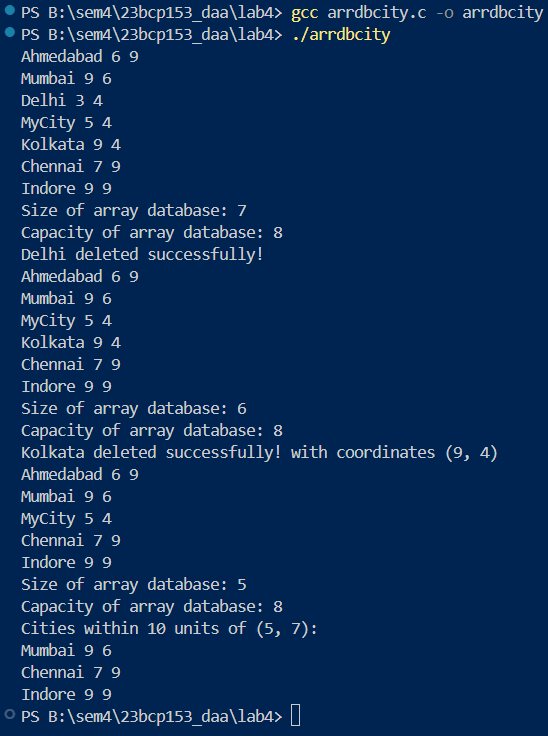
            printf("%s %d %d\n", arrdb->citiesarr[i]->name, arrdb->citiesarr[i]->x, arrdb->citiesarr[i]->y);

        }

    }

}

## **Output:**



## **Code:**

### **Linked-List-Based:**

#include <stdio.h>

#include <stdlib.h>

#include <string.h>

#include <time.h>

#include <math.h>

typedef struct City {

    char\* name;

    int x;

    int y;

} City;

typedef struct Node {

    City\* city;

    struct Node\* next;

} Node;

typedef struct Lldb {

    Node\* head;

    int size;

} Lldb;

City\* create\_city(char\* name, int x, int y);

Node\* create\_node(City\* city);

void insert\_city\_ll (Lldb\* lldb, City\* city);

void print\_ll\_db(Lldb\* lldb);

void delete\_by\_name(Lldb\* lldb, char\* name);

void delete\_by\_coordinates(Lldb\* lldb, int x, int y);

Node\* search\_by\_name(Lldb\* lldb, char\* name);

Node\* search\_by\_coordinates(Lldb\* lldb, int x, int y);

void print\_within\_dist(Lldb\* lldb, int x, int y, int dist);

int main(void)

{

    Lldb\* lldb = (Lldb\*)malloc(sizeof(Lldb));

    lldb->head = NULL;

    lldb->size = 0;

    City\* city1 = create\_city("Ahmedabad", 6, 9);

    insert\_city\_ll(lldb, city1);

    City\* city2 = create\_city("Mumbai", 9, 6);

    insert\_city\_ll(lldb, city2);

    City\* city3 = create\_city("Delhi", 3, 4);

    insert\_city\_ll(lldb, city3);

    City\* city4 = create\_city("MyCity", 5, 4);

    insert\_city\_ll(lldb, city4);

    City\* city5 = create\_city("Kolkata", 9, 4);

    insert\_city\_ll(lldb, city5);

    City\* city6 = create\_city("Chennai", 7, 9);

    insert\_city\_ll(lldb, city6);

    City\* city7 = create\_city("Indore", 9, 9);

    insert\_city\_ll(lldb, city7);

    print\_ll\_db(lldb);

    delete\_by\_name(lldb, "MyCity");

    print\_ll\_db(lldb);

    delete\_by\_coordinates(lldb, 9, 6);

    print\_ll\_db(lldb);

    Node\* somecity = search\_by\_name(lldb, "Delhi");

    printf("%s %d %d\n", somecity->city->name, somecity->city->x, somecity->city->y);

    somecity = search\_by\_coordinates(lldb, 9, 9);

    printf("%s %d %d\n", somecity->city->name, somecity->city->x, somecity->city->y);

    print\_within\_dist(lldb, 5, 7, 10);

    return 0;

}

City\* create\_city(char\* name, int x, int y)

{

    City\* city = (City\*)malloc(sizeof(City));

    city->name = (char\*)malloc(strlen(name) + 1);

    city->name = name;

    city->x = x;

    city->y = y;

    return city;

}

Node\* create\_node(City\* city)

{

    Node\* node = (Node\*) malloc(sizeof(Node));

    node->city = city;

    node->next = NULL;

    return node;

}

void insert\_city\_ll (Lldb\* lldb, City\* city)

{

    Node\* node = create\_node(city);

    node->next = lldb->head;

    lldb->head = node;

    lldb->size++;

}

void print\_ll\_db(Lldb\* lldb)

{

    Node\* trav = lldb->head;

    while(trav != NULL)

    {

        printf("%s %d %d\n", trav->city->name, trav->city->x, trav->city->y);

        trav = trav->next;

    }

    printf("Size of linked list (database): %d\n", lldb->size);

}

void delete\_by\_name(Lldb\* lldb, char\* name)

{

    Node\* trav = lldb->head;

    if (strcmp(trav->city->name, name) == 0)

    {

        // if city is at head

        lldb->head = trav->next;

        free(trav);

        lldb->size--;

        return;

    }

    while (trav || strcmp(trav->next->city->name, name) != 0)

    {

        if (trav->next == NULL)

        {

            printf("City not found\n");

            return;

        }

        if (strcmp(trav->next->city->name, name) == 0)

        {

            Node\* temp = trav->next;

            trav->next = trav->next->next;

            free(temp);

            lldb->size--;

            return;

        }

        trav = trav->next;

    }

    lldb->size--;

    return;

}

void delete\_by\_coordinates(Lldb\* lldb, int x, int y)

{

    Node\* trav = lldb->head;

    if (trav->city->x == x && trav->city->y == y)

    {

        // if city is at head

        lldb->head = trav->next;

        free(trav);

        lldb->size--;

        return;

    }

    while (trav || (trav->next->city->x != x && trav->next->city->y != y))

    {

        if (trav->next == NULL)

        {

            printf("City not found\n");

            return;

        }

        if (trav->next->city->x == x && trav->next->city->y == y)

        {

            Node\* temp = trav->next;

            trav->next = trav->next->next;

            free(temp);

            lldb->size--;

            return;

        }

        trav = trav->next;

    }

    lldb->size--;

    return;

}

Node\* search\_by\_name(Lldb\* lldb, char\* name)

{

    Node\* trav = lldb->head;

    while (trav || strcmp(trav->city->name, name) != 0)

    {

        if (trav->next == NULL)

        {

            printf("City not found\n");

            return NULL;

        }

        if (strcmp(trav->city->name, name) == 0)

        {

            return trav;

        }

        trav = trav->next;

    }

    return NULL;

}

Node\* search\_by\_coordinates(Lldb\* lldb, int x, int y)

{

    Node\* trav = lldb->head;

    while (trav || (trav->city->x != x && trav->city->y != y))

    {

        if (trav->next == NULL)

        {

            printf("City not found\n");

            return NULL;

        }

        if (trav->city->x == x && trav->city->y == y)

        {

            return trav;

        }

        trav = trav->next;

    }

    return NULL;

}

void print\_within\_dist(Lldb\* lldb, int x, int y, int dist)

{

    if (lldb->head == NULL)

    {

        printf("Database Empyt\n");

        return;

    }

    printf("Cities within %d units of (%d, %d):\n", dist, x, y);

    Node\* trav = lldb->head;

    while (trav)

    {

        int diffx = trav->city->x - x;

        int diffy = trav->city->y - y;

        float distance = sqrt(pow(diffx, 2) - pow(diffy, 2));

        if (distance <= dist)

        {

            printf("%s %d %d\n",  trav->city->name, trav->city->x, trav->city->y);

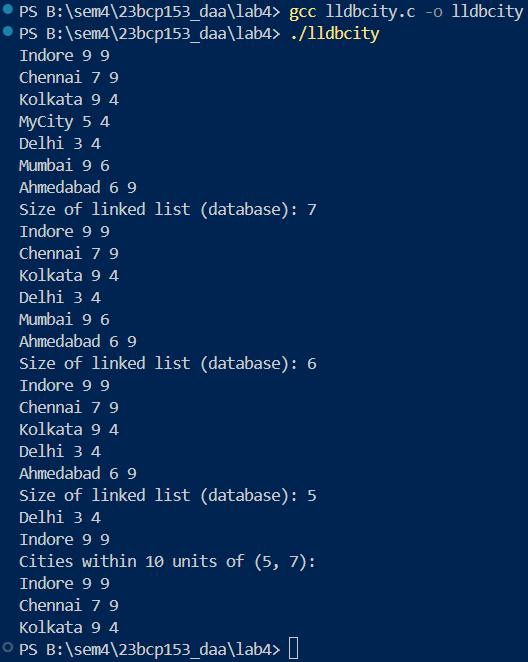
        }

        trav = trav->next;

    }

}

## **Output:**



## **Analysis:**

|  |  |  |
| --- | --- | --- |
| **Operation** | **Array** | **Linked List** |
| Insertion |  |  |
| Deletion |  |  |
| Search |  |  |
| Print |  |  |

**Advantages of Array based implementation:**

* Accessing element takes constant time
* No extra pointer needed for traversal

**Disadvantages of Array based implementation:**

* Resizing takes extra time
* Even after dynamic resizing – this particular implementation may have space unused

**Advantages of Linked List based implementation:**

* Has proper dynamic size
* Insertion is efficient as we insert at head

**Disadvantages of Linked List based implementation:**

* Accessing elements/ traversal has o(n) complexity

Yes, sorting would help in array-based implementation as operations can use binary search

Except insertion which would require shifting

Yes, alphabetical order may slow down the operations as shifting would be required in array based implementation

# **EXPERIMENT 5**

## **20CP209P – Design and Analysis of Algorithm Lab**

## **Aim:**

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:

## **Code:**

### **Interval-Scheduling:**

#include <stdio.h>

#include <stdlib.h>

typedef struct Process {

    int id;

    int start;

    int finish;

    int duration;

} Process;

int comp\_fin(const void\* a, const void\* b);

int comp\_st(const void\* a, const void\* b);

int comp\_dur(const void\* a, const void\* b);

void earl\_st(Process processes[], int n);

void sjf(Process processes[], int n);

void earl\_fin(Process processes[], int n);

int main(void)

{

    Process processes1[] = {

        {1, 1, 4, 4 - 1},

        {2, 3, 5, 5 - 3},

        {3, 0, 6, 6 - 0},

        {4, 5, 7, 7 - 5},

        {5, 3, 9, 9 - 3},

        {6, 5, 9, 9 - 5},

        {7, 6, 10, 10 - 6},

        {8, 8, 11, 11 - 8},

        {9, 8, 12, 12 - 8},

        {10, 2, 14, 14 - 2}

    };

    int n = sizeof(processes1) / sizeof(processes1[0]);

    earl\_fin(processes1, n);

    earl\_st(processes1, n);

    sjf(processes1, n);

    printf("\n~~~~~~~~~~~~~~~~~~~~~~~~~~~~~\n\n");

    // As per Cormen example

    Process processes2[] = {

        {1, 1, 4, 4 - 1},

        {2, 3, 5, 5 - 3},

        {3, 0, 6, 6 - 0},

        {4, 5, 7, 7 - 5},

        {5, 3, 9, 9 - 3},

        {6, 5, 9, 9 - 5},

        {7, 6, 10, 10 - 6},

        {8, 8, 11, 11 - 8},

        {9, 8, 12, 12 - 8},

        {10, 2, 14, 14 - 2},

        {11, 12, 16, 16 - 12}

    };

    int n2 = sizeof(processes2) / sizeof(processes2[0]);

    earl\_fin(processes2, n2);

    earl\_st(processes2, n2);

    sjf(processes2, n2);

    return 0;

}

// Greedy activity selection - cormen pg. 424 - pdf pg. 446

void earl\_fin(Process processes[], int n)

{

    qsort(processes, n, sizeof(Process), comp\_fin);

    printf("Selected processes -> Earliest Finish Time\n(printed instead of added in set)\n");

    printf("As per Cormen Greedy Approach\n");

    int last\_fin\_time = 0;

    for (int i = 0; i < n; i++)

    {

        if (processes[i].start >= last\_fin\_time)

        {

            printf("Process %d -> Start: %d, Finish: %d, Duration: %d\n", processes[i].id, processes[i].start, processes[i].finish, processes[i].duration);

            last\_fin\_time = processes[i].finish;

        }

    }

    return;

}

void earl\_st(Process processes[], int n)

{

    qsort(processes, n, sizeof(Process), comp\_st);

    printf("Selected processes -> Earliest Start Time\n(printed instead of added in set)\n");

    printf("As per Cormen Greedy Approach\n");

    int last\_fin\_time = 0;

    for (int i = 0; i < n; i++)

    {

        if (processes[i].start >= last\_fin\_time)

        {

            printf("Process %d -> Start: %d, Finish: %d, Duration: %d\n", processes[i].id, processes[i].start, processes[i].finish, processes[i].duration);

            last\_fin\_time = processes[i].finish;

        }

    }

    return;

}

void sjf(Process processes[], int n)

{

    qsort(processes, n, sizeof(Process), comp\_dur);

    printf("Selected processes -> Shortest Job first\n(printed instead of added in set)\n");

    printf("As per Cormen Greedy Approach\n");

    int last\_fin\_time = 0;

    for (int i = 0; i < n; i++)

    {

        if (processes[i].start >= last\_fin\_time)

        {

            printf("Process %d -> Start: %d, Finish: %d, Duration: %d\n", processes[i].id, processes[i].start, processes[i].finish, processes[i].duration);

            last\_fin\_time = processes[i].finish;

        }

    }

    return;

}

int comp\_fin(const void\* a, const void\* b)

{

    return (((Process \*)a)->finish - ((Process \*)b)->finish);

}

int comp\_st(const void\* a, const void\* b)

{

    return (((Process \*)a)->start - ((Process \*)b)->start);

}

int comp\_dur(const void\* a, const void\* b)

{

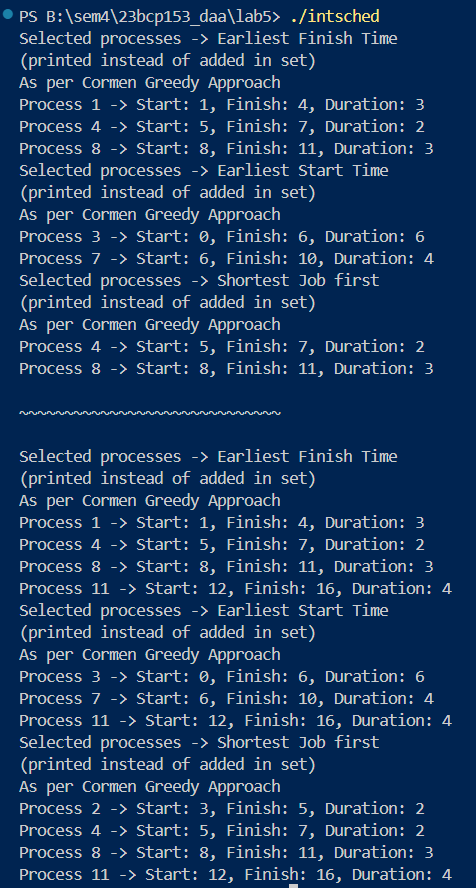
    return (((Process \*)a)->duration - ((Process \*)b)->duration);

}

// Details for qsort function

// https://www.w3schools.com/c/ref\_stdlib\_qsort.php#:~:text=The%20qsort()%20function%20sorts,h%3E%20header%20file.

## **Output:**



## **Code:**

### **Interval-Partitioning:**

#include <stdio.h>

#include <stdlib.h>

typedef struct Process {

    int id;

    int start;

    int finish;

    int duration;

} Process;

typedef struct  Node {

    Process process;

    struct Node\* next;

} Node;

typedef struct TT {

    Node\*\* classes;

    int n;

    int filled;

} TT;

int comp\_fin(const void\* a, const void\* b);

int comp\_st(const void\* a, const void\* b);

int comp\_dur(const void\* a, const void\* b);

TT\* init\_tt(int n);

void add\_proc\_to\_class(Node\*\* head, Process p);

void earl\_st(Process processes[], int n, TT\* mytt);

void earl\_fin(Process processes[], int n, TT\* mytt);

void sjf(Process processes[], int n, TT\* mytt);

void print\_tt(TT\* mytt);

int main(void)

{

    Process processes1[] = {

        {1, 1, 2, 2 - 1},

        {2, 1, 3, 3 - 1},

        {3, 1, 4, 4 - 1},

        {4, 2, 4, 4 - 2},

        {5, 3, 5, 5 - 3},

        {6, 4, 6, 6 - 4},

        {7, 4, 6, 6 - 4},

        {8, 6 , 7, 7 - 6},

        {9, 6, 8, 8 - 6},

        {10, 6, 8, 8 - 6}

    };

    int n = sizeof(processes1) / sizeof(processes1[0]);

    TT\* mytt;

    printf("Earliest Finish Time Partitioning:\n");

    mytt = init\_tt(n);

    earl\_fin(processes1, n, mytt);

    print\_tt(mytt);

    free(mytt->classes);

    free(mytt);

    printf("\nEarliest Start Time Partitioning:\n");

    mytt = init\_tt(n);

    earl\_st(processes1, n, mytt);

    print\_tt(mytt);

    free(mytt->classes);

    free(mytt);

    printf("\nShortest Job First Partitioning:\n");

    mytt = init\_tt(n);

    sjf(processes1, n, mytt);

    print\_tt(mytt);

    free(mytt->classes);

    free(mytt);

    return 0;

}

TT\* init\_tt(int n)

{

    TT\* mytt = (TT \*)malloc(sizeof(TT));

    mytt->classes = (Node\*\*)malloc(sizeof(Node \*) \* n);

    mytt->n = n;

    mytt->filled = 0;

    for (int i = 0; i < n; i++)

    {

        mytt->classes[i] = NULL;

    }

    return mytt;

}

void add\_proc\_to\_class(Node\*\* head, Process p)

{

    Node\* new\_node = (Node\*)malloc(sizeof(Node));

    new\_node->process = p;

    new\_node->next = \*head;

    \*head = new\_node;

    return;

}

int can\_place\_in\_class(Node\* head, Process p)

{

    Node\* curr = head;

    while(curr)

    {

        if (p.start < curr->process.finish && p.finish > curr->process.start)

        {

            return 0;

        }

        curr = curr->next;

    }

    return 1;

}

void earl\_st(Process processes[], int n, TT\* mytt)

{

    qsort(processes, n, sizeof(Process), comp\_st);

    for (int i = 0; i < n; i++)

    {

        int placed = 0;

        for (int j = 0; j < mytt->filled; j++)

        {

            if (can\_place\_in\_class(mytt->classes[j], processes[i]))

            {

                add\_proc\_to\_class(&mytt->classes[j], processes[i]);

                placed = 1;

                break;

            }

        }

        if(!placed)

        {

            add\_proc\_to\_class(&mytt->classes[mytt->filled], processes[i]);

            mytt->filled++;

        }

    }

    return;

}

void earl\_fin(Process processes[], int n, TT\* mytt)

{

    qsort(processes, n, sizeof(Process), comp\_fin);

    for (int i = 0; i < n; i++)

    {

        int placed = 0;

        for (int j = 0; j < mytt->filled; j++)

        {

            if (can\_place\_in\_class(mytt->classes[j], processes[i]))

            {

                add\_proc\_to\_class(&mytt->classes[j], processes[i]);

                placed = 1;

                break;

            }

        }

        if(!placed)

        {

            add\_proc\_to\_class(&mytt->classes[mytt->filled], processes[i]);

            mytt->filled++;

        }

    }

    return;

}

void sjf(Process processes[], int n, TT\* mytt)

{

    qsort(processes, n, sizeof(Process), comp\_dur);

    for (int i = 0; i < n; i++)

    {

        int placed = 0;

        for (int j = 0; j < mytt->filled; j++)

        {

            if (can\_place\_in\_class(mytt->classes[j], processes[i]))

            {

                add\_proc\_to\_class(&mytt->classes[j], processes[i]);

                placed = 1;

                break;

            }

        }

        if(!placed)

        {

            add\_proc\_to\_class(&mytt->classes[mytt->filled], processes[i]);

            mytt->filled++;

        }

    }

    return;

}

void print\_tt(TT\* mytt)

{

    for (int i = 0; i < mytt->filled; i++)

    {

        printf("Class no.: %d\n\t", i);

        Node\* curr = mytt->classes[i];

        while (curr)

        {

            printf("P%d - (%d-%d)   ", curr->process.id, curr->process.start, curr->process.finish);

            curr = curr->next;

        }

        printf("\n");

    }

    printf("\nTotal number of classes used: %d\n", mytt->filled);

}

int comp\_fin(const void\* a, const void\* b)

{

    return (((Process \*)a)->finish - ((Process \*)b)->finish);

}

int comp\_st(const void\* a, const void\* b)

{

    return (((Process \*)a)->start - ((Process \*)b)->start);

}

int comp\_dur(const void\* a, const void\* b)

{

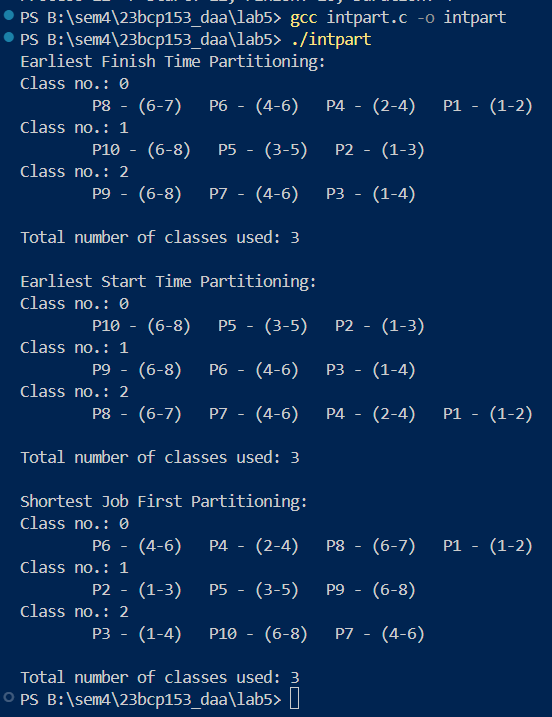
    return (((Process \*)a)->duration - ((Process \*)b)->duration);

}

// can implement using the below strategy

// https://leetcode.com/problems/divide-intervals-into-minimum-number-of-groups/editorial/

## **Output:**



# **EXPERIMENT 6**

## **20CP209P – Design and Analysis of Algorithm Lab**

## **Aim:**

Implement both a standard 𝑂(𝑛 matrix multiplication algorithm and Strassen’s matrix 3 ) 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?

## **Code:**

### **Strassen’s Algorithm:**

#include <stdio.h>

#include <stdlib.h>

#define fr(i, a, b) for (int i = a; i < b; i++)

void matmul(int arra[4][4], int arrb[4][4], int arrc[4][4]);

void add\_matrix(int size, int a[size][size], int b[size][size], int c[size][size]);

void sub\_matrix(int size, int a[size][size], int b[size][size], int c[size][size]);

void strassen\_multiply(int size, int a[size][size], int b[size][size], int c[size][size]);

void strassen\_4x4(int A[4][4], int B[4][4], int C[4][4]);

int main(void)

{

    int arra[4][4] = {{1, 2, 3, 4},

                      {5, 6, 7, 8},

                      {9, 10, 11, 12},

                      {13, 14, 15, 16}};

    int arrb[4][4] = {{1, 2, 3, 4},

                      {5, 6, 7, 8},

                      {9, 10, 11, 12},

                      {13, 14, 15, 16}};

    int arrc[4][4];

    matmul(arra, arrb, arrc);

    fr(i, 0, 4)

    {

        fr(j, 0, 4)

        {

            printf("%d ", arrc[i][j]);

        }

        printf("\n");

    }

    printf("\n"); printf("\n");

    strassen\_4x4(arra, arrb, arrc);

    fr(i, 0, 4)

    {

        fr(j, 0, 4)

        {

            printf("%d ", arrc[i][j]);

        }

        printf("\n");

    }

    return 0;

}

void matmul(int arra[4][4], int arrb[4][4], int arrc[4][4])

{

    fr(i, 0, 4)

    {

        fr(j, 0, 4)

        {

            arrc[i][j] = 0;

            fr(k, 0, 4)

            {

                arrc[i][j] += arra[i][k] \* arrb[k][j];

            }

        }

    }

    return;

}

void add\_matrix(int size, int a[size][size], int b[size][size], int c[size][size])

{

    fr(i, 0, size)

    {

        fr(j, 0, size)

        {

            c[i][j] = a[i][j] + b[i][j];

        }

    }

}

void sub\_matrix(int size, int a[size][size], int b[size][size], int c[size][size])

{

    fr(i, 0, size)

    {

        fr(j, 0, size)

        {

            c[i][j] = a[i][j] - b[i][j];

        }

    }

}

void strassen\_multiply(int size, int a[size][size], int b[size][size], int c[size][size])

{

    if (size == 2)

    {

        // this is base case for final 2x2 mat

        c[0][0] = a[0][0] \* b[0][0] + a[0][1] \* b[1][0];

        c[0][1] = a[0][0] \* b[0][1] + a[0][1] \* b[1][1];

        c[1][0] = a[1][0] \* b[0][0] + a[1][1] \* b[1][0];

        c[1][1] = a[1][0] \* b[0][1] + a[1][1] \* b[1][1];

    }

    else

    {

        int new\_size = size / 2;

        int a11[2][2], a12[2][2], a21[2][2], a22[2][2];

        int b11[2][2], b12[2][2], b21[2][2], b22[2][2];

        int c11[2][2], c12[2][2], c21[2][2], c22[2][2];

        int M1[2][2], M2[2][2], M3[2][2], M4[2][2], M5[2][2], M6[2][2], M7[2][2];

        int temp1[2][2], temp2[2][2];

        fr(i, 0, new\_size)

        {

            fr(j, 0, new\_size)

            {

                a11[i][j] = a[i][j];

                a12[i][j] = a[i][j + new\_size];

                a21[i][j] = a[i + new\_size][j];

                a22[i][j] = a[i + new\_size][j + new\_size];

                b11[i][j] = b[i][j];

                b12[i][j] = b[i][j + new\_size];

                b21[i][j] = b[i + new\_size][j];

                b22[i][j] = b[i + new\_size][j + new\_size];

            }

        }

        add\_matrix(new\_size, a11, a22, temp1);

        add\_matrix(new\_size, b11, b22, temp2);

        strassen\_multiply(new\_size, temp1, temp2, M1);

        add\_matrix(new\_size, a21, a22, temp1);

        strassen\_multiply(new\_size, temp1, b11, M2);

        sub\_matrix(new\_size, b12, b22, temp1);

        strassen\_multiply(new\_size, a11, temp1, M3);

        sub\_matrix(new\_size, b21, b11, temp1);

        strassen\_multiply(new\_size, a22, temp1, M4);

        add\_matrix(new\_size, a11, a12, temp1);

        strassen\_multiply(new\_size, temp1, b22, M5);

        sub\_matrix(new\_size, a21, a11, temp1);

        add\_matrix(new\_size, b11, b12, temp2);

        strassen\_multiply(new\_size, temp1, temp2, M6);

        sub\_matrix(new\_size, a12, a22, temp1);

        add\_matrix(new\_size, b21, b22, temp2);

        strassen\_multiply(new\_size, temp1, temp2, M7);

        add\_matrix(new\_size, M1, M4, temp1);

        sub\_matrix(new\_size, temp1, M5, temp2);

        add\_matrix(new\_size, temp2, M7, c11);

        add\_matrix(new\_size, M3, M5, c12);

        add\_matrix(new\_size, M2, M4, c21);

        sub\_matrix(new\_size, M1, M2, temp1);

        add\_matrix(new\_size, temp1, M3, temp2);

        add\_matrix(new\_size, temp2, M6, c22);

        fr(i, 0, new\_size)

        {

            fr(j, 0, new\_size)

            {

                c[i][j] = c11[i][j];

                c[i][j + new\_size] = c12[i][j];

                c[i + new\_size][j] = c21[i][j];

                c[i + new\_size][j + new\_size] = c22[i][j];

            }

        }

    }

}

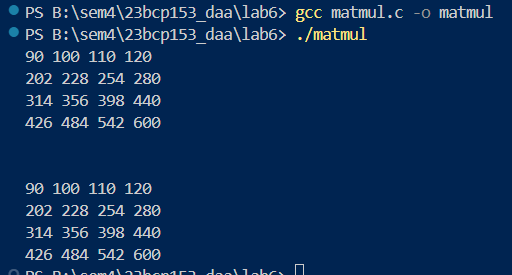
void strassen\_4x4(int A[4][4], int B[4][4], int C[4][4])

{

    strassen\_multiply(4, A, B, C);

}

## **Output:**



# **EXPERIMENT 7**

## **20CP209P – Design and Analysis of Algorithm Lab**

## **Aim:**

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 between any node 𝑣𝑤 𝑣 and node 𝑤. Find the cheapest path from every node to every other node. Edges may have negative edge weights.

## **Code:**

### **Floyd Warshall Algorithm:**

#include <stdio.h>

#include <stdlib.h>

#include <limits.h>

#include <string.h>

#define fr(i, a, b) for (int i = a; i < b; i++)

#define inf INT\_MAX

// note that floyd warhsall does not work for negative cycles

int \*\*create\_mat(int n);

void print\_mat(int \*\*mat, int n);

int \*\*init\_matr();

void copy\_matrix(int \*\*src, int \*\*dest, int n);

int \*\*\*floydwar\_with\_hist(int \*\*graph, int vertex);

void show\_hist(int \*\*\*hist, int vertex);

void reconstruct\_path\_recursive(int i, int j, int k, int \*\*\*hist, char \*path\_str);

void print\_all\_shortest\_paths(int \*\*\*hist, int vertex);

void free\_mat(int \*\*mat, int n);

void free\_hist(int \*\*\*hist, int vertex);

int main(void)

{

    int \*\*matr = init\_matr();

    int vertex = 4;

    // int vertex = 9;

    // int vertex = sizeof(matr);

    // the above will always return 4, as it is the size of the pointer

    // int vertex = sizeof(matr) / sizeof(matr[0]);

    // int x = sizeof(matr); printf("x===> %d\n", x);

    // int y = sizeof(matr[0]); printf("y===> %d\n", x);

    // printf("vertex ===> %d \n", vertex);

    // int \*\*new\_mat = create\_mat(vertex);

    // print\_mat(new\_mat, vertex);

    printf("The Graph: \n");

    print\_mat(matr, vertex);

    int \*\*\*hist = floydwar\_with\_hist(matr, vertex);

    printf("All the matrices (history using floyd warshall algo): \n");

    show\_hist(hist, vertex);

    printf("All the matrices (history using floyd warshall algo): \n");

    show\_hist(hist, vertex);

    printf("Shortest Paths:\n");

    print\_all\_shortest\_paths(hist, vertex);

    return 0;

}

int \*\*create\_mat(int n)

{

    int \*\*mat = (int \*\*)calloc(n, sizeof(int\*));

    if (mat == NULL)

    {

        printf("Memory Alloc failed!\n");

        return NULL;

    }

    fr(i, 0, n)

    {

        mat[i] = (int\*)calloc(n, sizeof(int));

        if (mat[i] == NULL)

        {

            printf("Memory Alloc failed!\n");

            return NULL;

        }

    }

    return mat;

}

void print\_mat(int \*\*mat, int n)

{

    fr(i, 0, n)

    {

        fr(j, 0, n)

        {

            if (mat[i][j] == inf)

            {

                printf("inf ");

            }

            else

                printf("%d ", mat[i][j]);

        }

        printf("\n");

    }

    printf("\n");

}

int \*\*init\_matr()

{

    // int matr[9][9] = {

    //     {0,1,1,1,0,0,0,0,0},

    //     {1,0,1,0,1,0,0,0,0},

    //     {1,1,0,1,1,1,1,0,0},

    //     {1,0,1,0,0,0,1,0,0},

    //     {0,1,1,0,0,0,0,0,1},

    //     {0,0,1,0,0,0,1,1,1},

    //     {0,0,1,1,0,1,0,1,0},

    //     {0,0,0,0,0,1,1,0,1},

    //     {0,0,0,0,1,1,0,1,0}

    // };

    int matr[4][4] = {

        {0, 2, inf, 5},

        {3, 0, inf, 4},

        {inf, 1, 0, inf},

        {inf, inf, 2, 0}

    };

    // the above is the graph for the problem

    int n = sizeof(matr) / sizeof(matr[0]);

    printf("n ===> %d\n", n);

    int \*\*new\_mat = create\_mat(n);

    if (!new\_mat)

    {

        printf("Memory Alloc failed\n");

        return NULL;

    }

    fr(i, 0, n)

    {

        fr(j, 0, n)

        {

            new\_mat[i][j] = matr[i][j];

        }

    }

    return new\_mat;

}

void copy\_matrix(int \*\*src, int \*\*dest, int n)

{

    fr(i, 0, n)

    {

        fr(j, 0, n)

        {

            dest[i][j] = src[i][j];

        }

    }

}

int \*\*\*floydwar\_with\_hist(int \*\*graph, int vertex)

{

    // here graph is matr or matr is graph

    int \*\*\*hist = (int \*\*\*)malloc(vertex + 1 \* sizeof(int\*\*));

    hist[0] = create\_mat(vertex);

    copy\_matrix(graph, hist[0], vertex);

    fr(k, 0, vertex)

    {

        hist[k + 1] = create\_mat(vertex);

        copy\_matrix(hist[k], hist[k + 1], vertex);

        fr(i, 0, vertex)

        {

            fr(j, 0, vertex)

            {

                if (hist[k][i][k] != inf && hist[k][k][j] != inf)

                {

                    int new\_dist = hist[k][i][k] + hist[k][k][j];

                    if (new\_dist < hist[k + 1][i][j])

                    {

                        hist[k + 1][i][j] = new\_dist;

                    }

                }

            }

        }

    }

    return hist;

}

void show\_hist(int \*\*\*hist, int vertex)

{

    fr(i, 0, vertex + 1)

    {

        printf("hist[%d]:\n", i);

        print\_mat(hist[i], vertex);

    }

}

void reconstruct\_path\_recursive(int i, int j, int k, int \*\*\*hist, char \*path\_str)

{

    // Base case: If k < 0, it means no intermediate node from 0 to vertex-1

    // was found on the path between the \*current\* i and j segment.

    // This implies a direct edge (or i==j, handled outside).

    if (k < 0)

    {

        // We only need to add the intermediate nodes. The start and end are handled outside.

        // If i and j were directly connected in the original graph check:

        // if (hist[0][i][j] != inf && hist[0][i][j] != 0 ) { } // No action needed here

        return;

    }

    // Check if the shortest path from i to j \*changed\* when node k was introduced.

    // We compare the distance in hist[k+1] (using nodes up to k)

    // with the distance in hist[k] (using nodes up to k-1).

    if (hist[k + 1][i][j] < hist[k][i][j])

    {

        // Yes, node 'k' is essential for the shortest path between i and j.

        // The path must go i -> ... -> k -> ... -> j.

        // Recursively find the path from i to k (using intermediates up to k-1).

        reconstruct\_path\_recursive(i, k, k - 1, hist, path\_str);

        // Append the intermediate node k to the path string.

        char buffer[20];

        sprintf(buffer, " -> %d", k);

        strcat(path\_str, buffer);

        // Recursively find the path from k to j (using intermediates up to k-1).

        reconstruct\_path\_recursive(k, j, k - 1, hist, path\_str);

    }

    else

    {

        // No, the shortest path from i to j did NOT require node 'k' at this stage.

        // The path is the same as the one found using intermediate nodes up to k-1.

        // Continue checking with the next lower intermediate node.

        reconstruct\_path\_recursive(i, j, k - 1, hist, path\_str);

    }

}

void print\_all\_shortest\_paths(int \*\*\*hist, int vertex)

{

    fr(i, 0, vertex)

    {

        fr(j, 0, vertex)

        {

            printf("Path from %d to %d: ", i, j);

            // Check if a path exists

            if (hist[vertex][i][j] == inf)

            {

                printf("No path\n");

            }

            else if (i == j)

            {

                printf("%d (Dist: 0)\n", i);

            }

            else

            {

                // Allocate a buffer for the path string. Size calculation can be tricky,

                // make it large enough (e.g., vertex \* (max digits + arrow len)).

                char path\_str[vertex \* 15]; // Adjust size if needed

                sprintf(path\_str, "%d", i); // Start path string with the source node 'i'

                // Call the recursive function to build the intermediate path nodes string.

                // Start checking from the highest possible intermediate node (vertex - 1).

                reconstruct\_path\_recursive(i, j, vertex - 1, hist, path\_str);

                // Append the final destination node 'j'.

                char buffer[20];

                sprintf(buffer, " -> %d", j);

                strcat(path\_str, buffer);

                // Print the reconstructed path and the final distance.

                printf("%s (Dist: %d)\n", path\_str, hist[vertex][i][j]);

            }

        }

        printf("\n"); // Add a newline after processing all paths from node i

    }

}

void free\_mat(int \*\*mat, int n)

{

    if (!mat) return;

    fr(i, 0, n)

    {

        free(mat[i]); // Free each row's columns

    }

    free(mat); // Free the row pointers

}

void free\_hist(int \*\*\*hist, int vertex)

{

    if (!hist) return;

    // Free matrices from hist[0] to hist[vertex]

    fr(k, 0, vertex + 1)

    {

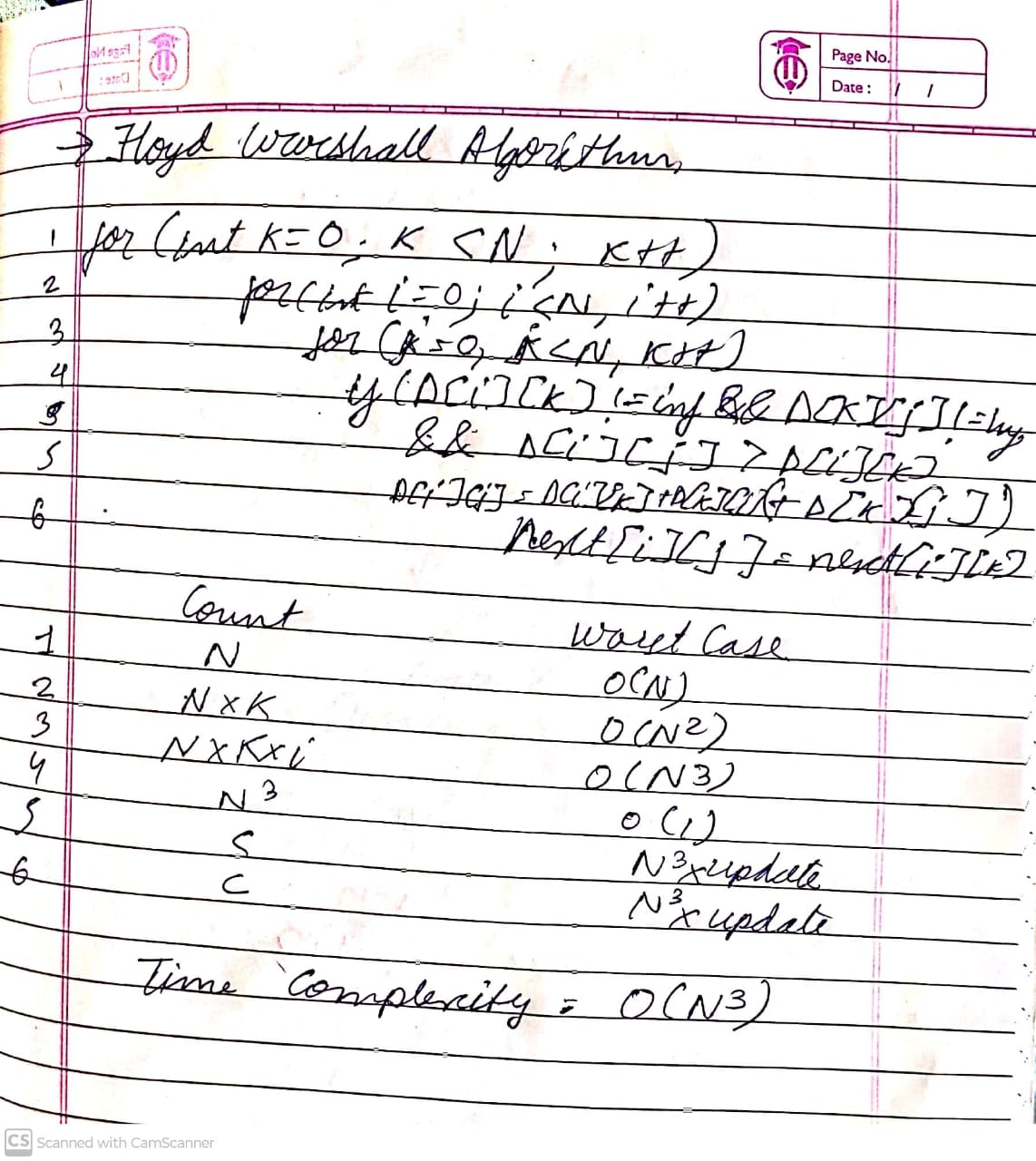
        free\_mat(hist[k], vertex);

    }

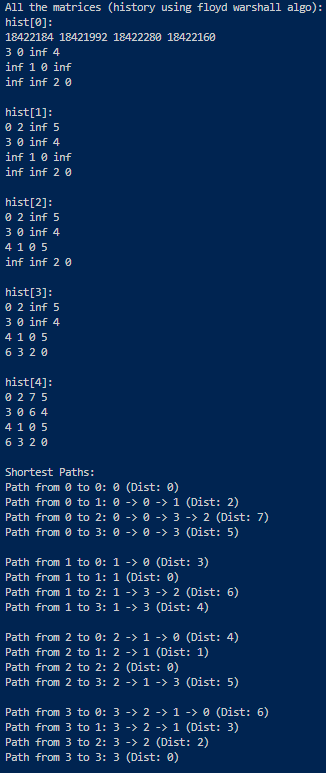
    free(hist); // Free the array of matrix pointers

}

## **Analysis:**



## **Output:**



# **EXPERIMENT 8**

## **20CP209P – Design and Analysis of Algorithm Lab**

## **Aim:**

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

## **Code:**

### **N Queens Problem:**

#include <stdio.h>

#include <stdlib.h>

#include <limits.h>

#include <stdbool.h>

#define fr(i, a, b) for (int i = a; i < b; i++)

#define N 4

// #define N 5

// #define N 6

// #define N 7

int total\_sol\_count = 0;

void print\_sol(int board[N][N]);

bool is\_safe(int board[N][N], int row, int col);

bool solveNQUtil(int board[N][N], int col);

int main(void)

{

    // int board[N][N] = {

    //     {0, 0, 0, 0},

    //     {0, 0, 0, 0},

    //     {0, 0, 0, 0},

    //     {0, 0, 0, 0}

    // };

    int board[N][N];

    fr (i, 0, N)

    {

        fr(j, 0, N) board[i][j] = 0;

    }

    if (solveNQUtil(board, 0) == false)

    {

        printf("Solution does not exist");

        return 0;

    }

    // print\_sol(board);

    printf("Total solutions found: %d\n", total\_sol\_count);

    return 0;

}

void print\_sol(int board[N][N])

{

    fr(i, 0, N)

    {

        fr (j, 0, N)

        {

            printf("%d", board[i][j]);

        }

        printf("\n");

    }

}

bool is\_safe(int board[N][N], int row, int col)

{

    int i, j;

    fr (i, 0, col)

    {

        if (board[row][i])

        {

            return false;

        }

    }

    for (i = row, j = col; i >= 0 && j >= 0; i--, j--)

    {

        if (board[i][j])

        {

            return false;

        }

    }

    for (i = row, j = col; j >= 0 && i < N; i++, j--)

    {

        if (board[i][j])

        {

            return false;

        }

    }

    return true;

}

bool solveNQUtil(int board[N][N], int col)

{

    if (col >= N)

    {

        total\_sol\_count++;

        printf("\nsol\n");

        print\_sol(board);

        return true;

    }

    // changes might be required here

    // Flag to track if any solution is found from this column onwards.

    // Initialize to false. It will be set to true if any recursive call

    // down the line finds a solution.

    bool res = false;

    for (int i = 0; i < N; i++)

    {

    // if it is safe to place the queen at position i, col -> place it

        if (is\_safe(board, i, col))

        {

            board[i][col] = 1;

            // printf("row: %d\n", i);

            // print\_sol(board);

            // printf("\n");

            // if (solveNQUtil(board, col + 1))

            // {

            //     return true;

            // }

            // Recur to place the rest of the queens for the next column (col + 1).

            // Crucially, we use 'res = solveNQUtil(...) || res;'

            // This calls the function for the next column AND combines its result

            // (true if a solution was found down that path) with any previous

            // results found by trying other rows in this \*current\* column.

            // We do NOT return immediately.

            res = solveNQUtil(board, col + 1) || res;

            // backtrack if the above condition is false

            board[i][col] = 0; // BACKTRACK

        }

    }

    // return false;

    // Return the final result 'res'. It will be true if any placement

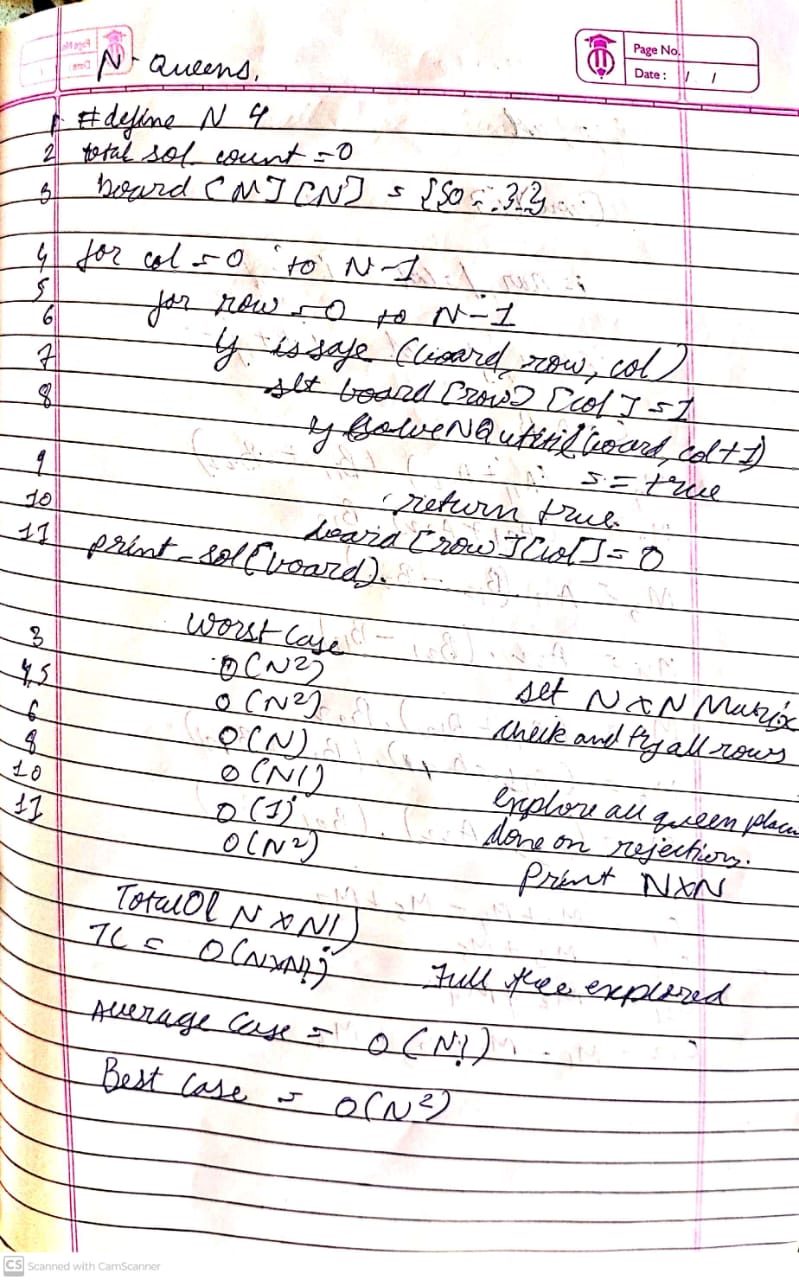
    // in this column 'col' led to at least one solution down the recursion path.

    // It will be false if no placement in this column led to any solution.

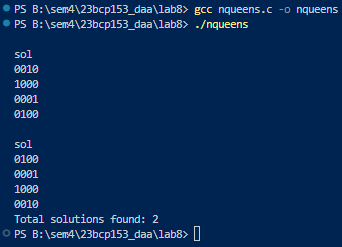
    return res;

}

## **Analysis:**



## **Output:**



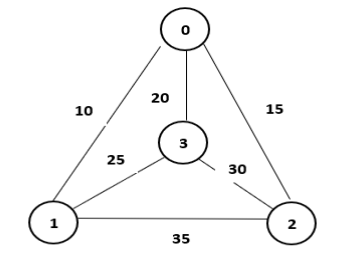
# **EXPERIMENT 9**

## **20CP209P – Design and Analysis of Algorithm Lab**

## **Aim:**

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

## **Code:**

### **Travelling Salesman Problem (Branch and Bound):**

#include <iostream>

#include <vector>

#include <limits> // Use <limits> for numeric\_limits

#include <algorithm>

#include <numeric>

#include <vector>

#include <queue> // For priority queue

#define fr(i, a, b) for (int i = a; i < b; i++)

#define pii pair<int,int>

using namespace std;

// Use numeric\_limits for infinity for better type safety and clarity

const int INF = numeric\_limits<int>::max();

// Structure to represent a node in the state-space tree

struct TSPNode {

    vector<int> path;       // Path taken so far

    vector<bool> visited;   // Visited cities marker

    int cost;               // Cost of the path so far

    int lower\_bound;        // Estimated lower bound for the total tour cost

    int level;              // Number of cities visited (path.size())

    // Constructor

    TSPNode(int n) : visited(n, false), cost(0), lower\_bound(0), level(0) {}

    // Custom comparator for the priority queue (min-heap based on lower\_bound)

    bool operator>(const TSPNode& other) const {

        return lower\_bound > other.lower\_bound;

    }

};

// Function to calculate the lower bound for a given node

// A simple lower bound: current cost + sum of minimum outgoing edge from each unvisited city

int calculate\_lower\_bound(const TSPNode& node, int n, const vector<vector<int>>& graph)

{

    int bound = node.cost;

    int last\_city = node.path.back();

    for (int i = 0; i < n; ++i)

    {

        // If city 'i' is unvisited OR it's the starting city and we need to return

        if (!node.visited[i])

        {

             // Find the minimum cost edge leaving city 'i' to any \*other\* city

             int min\_edge = INF;

             // If it's the last node in the path, find min edge to unvisited or start

             if (i == last\_city)

             {

                 for (int j = 0; j < n; ++j)

                 {

                      // Must go to an unvisited city OR back to start if it's the last hop

                      if (j == node.path[0] && node.level == n)

                      { // Check if it's the last step

                          if (graph[i][j] != INF)

                          {

                             min\_edge = min(min\_edge, graph[i][j]);

                          }

                      }

                      else if (!node.visited[j] && i != j && graph[i][j] != INF)

                      {

                         min\_edge = min(min\_edge, graph[i][j]);

                      }

                 }

             }

             else

             { // For other unvisited cities, find the absolute minimum outgoing edge

                for(int j=0; j<n; ++j)

                {

                    if (i != j && graph[i][j] != INF)

                    {

                        min\_edge = min(min\_edge, graph[i][j]);

                    }

                }

             }

            // If an unvisited city has no way out (shouldn't happen in complete graph)

            // or if it's the last node with no path back to start

            if (min\_edge == INF)

            {

                return INF; // This path is infeasible or bound calculation failed

            }

            bound += min\_edge;

        }

    }

     // A slightly simpler version (often used, might be less tight but still valid):

     // Sum minimum edge cost for all unvisited nodes, regardless of where they go

     /\*

     int simpler\_bound = node.cost;

     for(int i=0; i<n; ++i) {

         if(!node.visited[i]) {

             int min\_edge = INF;

             for(int j=0; j<n; ++j) {

                 if(i != j && graph[i][j] != INF) {

                     min\_edge = min(min\_edge, graph[i][j]);

                 }

             }

             if(min\_edge == INF) return INF; // Infeasible

             simpler\_bound += min\_edge;

         }

     }

     // Add cost from last visited node back to start if it's the last leg needed

     if (node.level == n -1 && !node.visited[0]) { // Check if only start is left

         int last = node.path.back();

         if(graph[last][0] != INF) {

              simpler\_bound += graph[last][0]; // Already partially counted, tricky

         } else {

              return INF; // No path back to start

         }

     }

     return simpler\_bound; // Let's use the simpler bound for clarity

     \*/

     // Let's stick to the slightly more accurate one calculated first.

     // If the path is nearly complete, the last leg must go back to start

     if (node.level == n)

     {

         int return\_cost = graph[last\_city][node.path[0]];

         if (return\_cost == INF) return INF; // No path back

         // The bound IS the final cost here

         return node.cost + return\_cost;

     }

     return bound;

}

// --- Simplified Lower Bound Implementation ---

// Calculate the lower bound: cost so far + sum of min edges from unvisited nodes

int calculate\_simplified\_lower\_bound(int current\_cost, int current\_city, int n, const vector<bool>& visited, const vector<vector<int>>& graph)

{

    int bound = current\_cost;

    for (int i = 0; i < n; ++i)

    {

        // Consider unvisited cities

        if (!visited[i])

        {

            int min\_val = INF;

            for (int j = 0; j < n; ++j)

            {

                // Find the cheapest edge leaving this unvisited city 'i'

                // The destination 'j' can be any other city (visited or unvisited)

                // or the starting city if 'i' is the last city visited in a partial tour

                if (i != j && graph[i][j] != INF)

                {

                   // Simple version: min edge to \*any\* other node

                    min\_val = min(min\_val, graph[i][j]);

                }

            }

             if(min\_val == INF) return INF; // This city is stranded

             bound += min\_val;

        }

    }

     // Special handling for the edge FROM the current\_city

     // Find the minimum edge from current\_city to an UNVISITED city

     int min\_from\_current = INF;

     bool can\_reach\_unvisited = false;

     for(int j = 0; j < n; ++j) {

        if (!visited[j] && graph[current\_city][j] != INF)

        {

            min\_from\_current = min(min\_from\_current, graph[current\_city][j]);

            can\_reach\_unvisited = true;

        }

     }

     // If we can't reach any unvisited node from current city (and tour not complete)

     // it might mean we only need to go back to start

     if (!can\_reach\_unvisited && count(visited.begin(), visited.end(), false) == 1 && !visited[0])

     {

         if (graph[current\_city][0] != INF)

         {

            // The only unvisited node is the start node

            // The lower bound should include the cost back to start

            // Note: The loop above already added min cost \*from\* start (if unvisited)

            // Let's refine the loop logic slightly:

            // bound = current\_cost

            // sum min outgoing edge for all nodes k (!visited)

            // sum min incoming edge for all nodes k (!visited)

            // return bound = current\_cost + (sum\_min\_out + sum\_min\_in) / 2 (Held-Karp idea - complex)

            // Stick to simpler: Sum of min outgoing from unvisited

            // Let's recalculate simpler bound:

            bound = current\_cost;

            for(int i=0; i<n; ++i)

            {

                // For the current node, find min edge to UNVISITED node

                if(i == current\_city)

                {

                    int min\_edge\_curr = INF;

                     for(int j=0; j<n; ++j)

                     {

                         if(!visited[j] && graph[i][j] != INF)

                         {

                             min\_edge\_curr = min(min\_edge\_curr, graph[i][j]);

                         }

                     }

                     // If only start node is left unvisited

                     if (min\_edge\_curr == INF && count(visited.begin(), visited.end(), false) == 1 && !visited[0])

                     {

                          if(graph[i][0] != INF)

                          {

                             min\_edge\_curr = graph[i][0];

                          }

                          else

                          {

                             return INF; // Cannot return to start

                          }

                     }

                     else if (min\_edge\_curr == INF && count(visited.begin(), visited.end(), false) > 0)

                     {

                          return INF; // Cannot reach any other unvisited node

                     }

                     if(min\_edge\_curr != INF) bound += min\_edge\_curr; // Add cost from current node onwards

                }

                // For OTHER unvisited nodes, find the absolute minimum outgoing edge

                else if (!visited[i])

                {

                     int min\_edge\_other = INF;

                     for(int j=0; j<n; ++j)

                     {

                          if(i != j && graph[i][j] != INF)

                          {

                              min\_edge\_other = min(min\_edge\_other, graph[i][j]);

                          }

                     }

                     if(min\_edge\_other == INF) return INF; // Stranded node

                     bound += min\_edge\_other;

                }

            }

            return bound;

         }

         else

         {

             return INF; // Can't get back to start

         }

     }

    return bound; // Return the calculated simple bound

}

int main(void)

{

    // Example Graph (Cost Matrix)

    // graph[i][j] = cost from city i to city j

    // Use INF if no direct path (or for diagonal i == j)

     vector<vector<int>> graph = {

        {INF, 10, 8, 9, 7},

        {10, INF, 10, 5, 6},

        {8, 10, INF, 8, 9},

        {9, 5, 8, INF, 6},

        {7, 6, 9, 6, INF}

    };

    int n = graph.size(); // Number of cities

    // Use graph2 for testing

    // vector<vector<int>> graph2 = {

    //     {INF, 10, 15, 20},

    //     {5, INF, 9, 10},

    //     {6, 13, INF, 12}, // Changed 9 to 13 based on common examples for cost 35

    //     {8, 8, 9, INF}

    // };

    // vector<vector<int>> graph = graph2; // Uncomment to use graph2

    // n = graph.size();

    // Priority Queue for B&B nodes (min-heap based on lower\_bound)

    priority\_queue<TSPNode, vector<TSPNode>, greater<TSPNode>> pq;

    // Global minimum cost found so far

    int min\_cost = INF;

    vector<int> final\_path;

    // Create the root node (starting at city 0)

    TSPNode root(n);

    root.level = 1;

    root.path.push\_back(0); // Start at city 0

    root.visited[0] = true;

    root.cost = 0;

     // Calculate initial lower bound (sum of min edges from all nodes) - Adjusted calculation

    root.lower\_bound = calculate\_simplified\_lower\_bound(root.cost, 0, n, root.visited, graph);

    // root.lower\_bound = 0; // Initial cost is 0

    // for(int i=0; i<n; ++i) {

    //     int min\_row = INF;

    //     for (int j=0; j<n; ++j) {

    //         if (i != j && graph[i][j] != INF) {

    //             min\_row = min(min\_row, graph[i][j]);

    //         }

    //     }

    //      if(min\_row == INF && n > 1) { // Check for infeasible graphs early

    //          cout << "Graph is not strongly connected or has isolated nodes." << endl;

    //          return 1;

    //      }

    //      if (min\_row != INF) root.lower\_bound += min\_row;

    // }

    if (root.lower\_bound != INF) {

        pq.push(root);

    } else if (n > 0) {

         cout << "Initial lower bound is INF. Problem might be infeasible." << endl;

    }

    cout << "Starting Branch and Bound TSP..." << endl;

    if (n <= 1) {

         cout << "Minimum Cost: 0" << endl;

         cout << "Path: 0" << endl;

         return 0;

    }

    // Branch and Bound main loop

    while (!pq.empty()) {

        // Get the node with the lowest lower\_bound

        TSPNode current\_node = pq.top();

        pq.pop();

        // --- Pruning Step ---

        // If the current node's lower bound is already worse than the best solution found, prune it.

        if (current\_node.lower\_bound >= min\_cost) {

            // cout << "Pruning node. Lower bound (" << current\_node.lower\_bound

            //      << ") >= min\_cost (" << min\_cost << ")" << endl;

            continue;

        }

        // --- Goal Check ---

        // If all cities have been visited (path length = n)

        if (current\_node.level == n) {

            // We need to add the cost of returning to the starting city (city 0)

            int last\_city = current\_node.path.back();

            int return\_cost = graph[last\_city][0]; // Cost from last city back to start

            if (return\_cost != INF) {

                int total\_cost = current\_node.cost + return\_cost;

                // Found a new best solution

                if (total\_cost < min\_cost) {

                    min\_cost = total\_cost;

                    final\_path = current\_node.path;

                    // Add the starting city to the end to show the cycle

                    final\_path.push\_back(0);

                    cout << "Found new best solution: Cost = " << min\_cost << ", Path = ";

                     for(int city : final\_path) cout << city << " "; cout << endl;

                }

            }

            // No further branching needed from a complete tour node

            continue;

        }

        // --- Branching Step ---

        // Explore neighbors (unvisited cities)

        int current\_city = current\_node.path.back();

        for (int next\_city = 0; next\_city < n; ++next\_city) {

            // If the next city is not visited and there's an edge

            if (!current\_node.visited[next\_city] && graph[current\_city][next\_city] != INF)

            {

                // Create a child node representing the extended path

                TSPNode child\_node = current\_node; // Copy parent state

                child\_node.level++;

                child\_node.path.push\_back(next\_city);

                child\_node.visited[next\_city] = true;

                child\_node.cost = current\_node.cost + graph[current\_city][next\_city];

                // Calculate the lower bound for the child node

                child\_node.lower\_bound = calculate\_simplified\_lower\_bound(child\_node.cost, next\_city, n, child\_node.visited, graph);

                 // cout << "  Exploring edge " << current\_city << "->" << next\_city

                 //      << ", Cost: " << child\_node.cost

                 //      << ", Lower Bound: " << child\_node.lower\_bound << endl;

                // --- Pruning before adding to queue ---

                // Only add the child to the queue if its bound is promising

                if (child\_node.lower\_bound < min\_cost) {

                    pq.push(child\_node);

                } else {

                    // cout << "  Pruning child path (bound " << child\_node.lower\_bound << " >= min\_cost " << min\_cost << ")" << endl;

                }

            }

        }

    }

    // Output the result

    if (min\_cost == INF) {

        cout << "\nNo feasible solution found." << endl;

    } else {

        cout << "\n------------------------------------" << endl;

        cout << "Optimal Minimum Cost: " << min\_cost << endl;

        cout << "Optimal Path: ";

        for (int i = 0; i < final\_path.size(); ++i) {

            cout << final\_path[i] << (i == final\_path.size() - 1 ? "" : " -> ");

        }

        cout << endl;

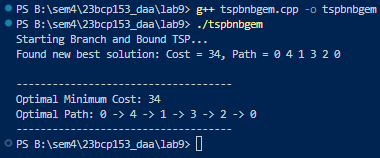
        cout << "------------------------------------" << endl;

    }

    return 0;

}

## **Output:**



# **EXPERIMENT 10**

## **20CP209P – Design and Analysis of Algorithm Lab**

## **Aim:**

To design and solve given problems using different algorithmic approaches and analyze their complexity.

1. 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 each month, where is a given parameter. This means that if license is 𝑗 > 1 𝑟 𝑗 𝑗 purchased 𝑡 months from now, it will cost 100𝑟 . We will assume that all the price 𝑗 𝑡 growth rates are distinct; that is, 𝑟 for licenses (even though they start at the 𝑖 ≠ 𝑟 𝑗 𝑖 ≠ 𝑗 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 , and 1 , 𝑟 2 , . . . , 𝑟 𝑛 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.
2. 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], . . . , 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(𝑙𝑜𝑔 𝑛) entries of 𝐴.

## **Code:**

### **A (License):**

#include <stdio.h>

#include <stdlib.h>

#include <math.h>

#define fr(i, a, b) for (int i = a; i < b; i++)

typedef struct License {

    int id;

    int cost;

} License;

int comp\_desc(const void \*a, const void \*b);

int main(void)

{

    int r[] = {7, 8 , 9, 2, 3, 4, 5, 10, 3, 11};

    int n = sizeof(r) / sizeof(r[0]);

    License arr[9];

    fr (i, 0, n)

    {

        arr[i].id = i;

        arr[i].cost = r[i];

    }

    qsort(arr, n, sizeof(License), comp\_desc);

    int ans[n];

    fr (i, 0, n)

    {

        ans[i] = arr[i].id;

    }

    double final\_cost = 0;

    fr (i, 0, n)

    {

        int t = i;

        int rate = arr[i].cost;

        printf("Month => %d, Job => %d\n", i + 1, ans[i]);

        // final\_cost += (100 \* rate \* t);

        final\_cost += (100 \* pow(rate, t));

    }

    printf("Final Cost => %.2f\n", final\_cost);

    return 0;

}

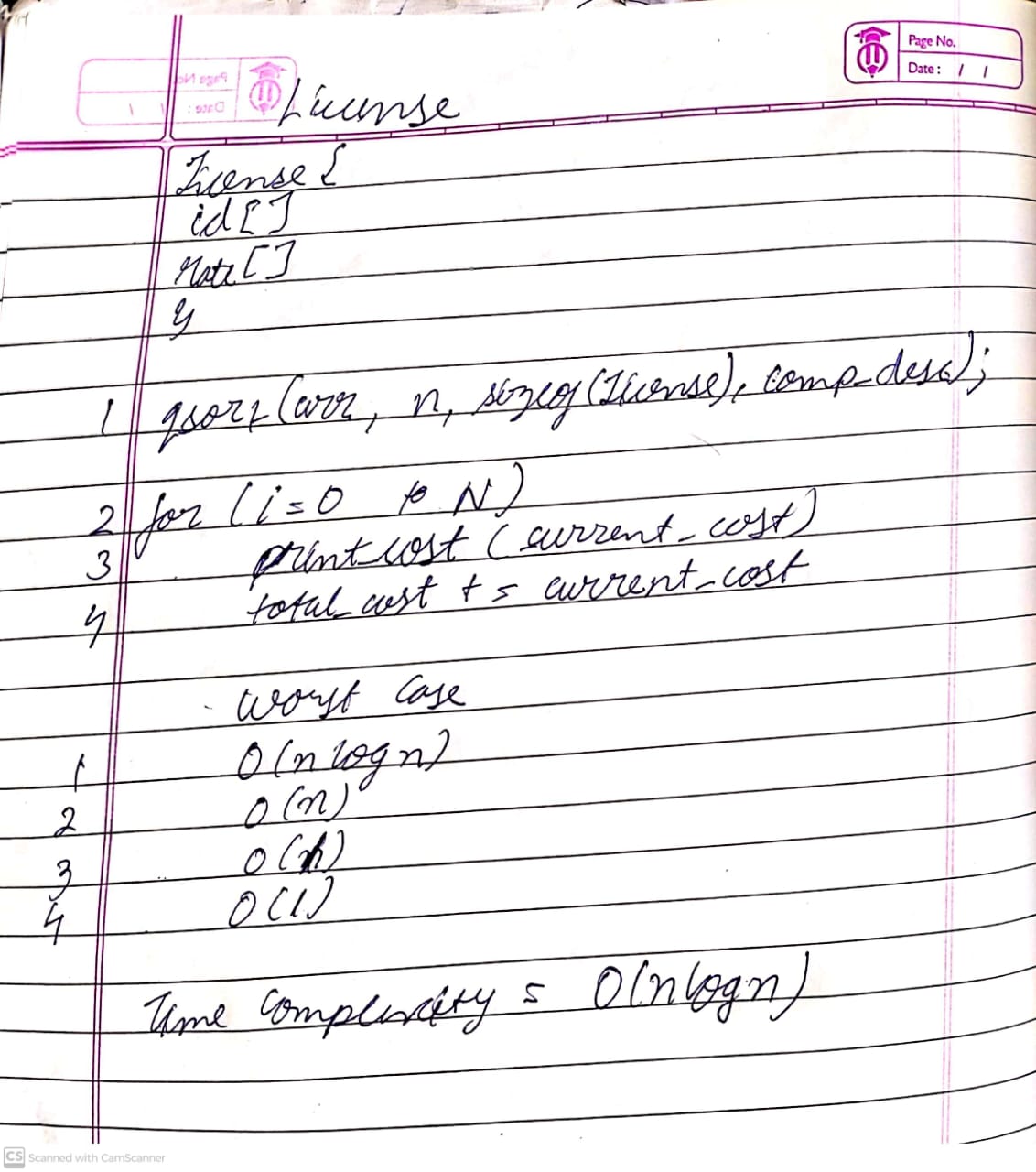
int comp\_desc(const void \*a, const void \*b)

{

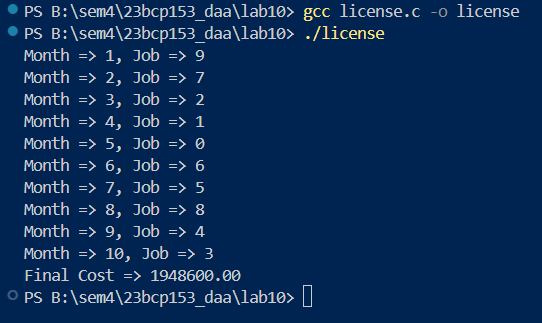
    return(((License \*)b)->cost - ((License \*)a)->cost);

}

## **Analysis:**



## **Output:**



## **Code:**

### **B (Unimodal):**

#include <stdio.h>

#include <stdlib.h>

int main(void)

{

    int ans[] = {144, 206, 282, 576, 580, 1395, 1096, 1081, 694, 623};

    int n = sizeof(ans) / sizeof(ans[0]);

    int i = 0, j = n - 1;

    int idx = 0;

    int iter = 0;

    while (i < j)

    {

        iter++;

        int mid = (i + j) / 2;

        printf("%d\t", iter);

        if (ans[mid] > ans[mid + 1] && ans[mid] > ans[mid - 1])

        {

            idx = mid;

            break;

        }

        else if (ans[mid] < ans[mid + 1])

        {

            i = mid + 1;

        }

        else

        {

            j = mid - 1;

        }

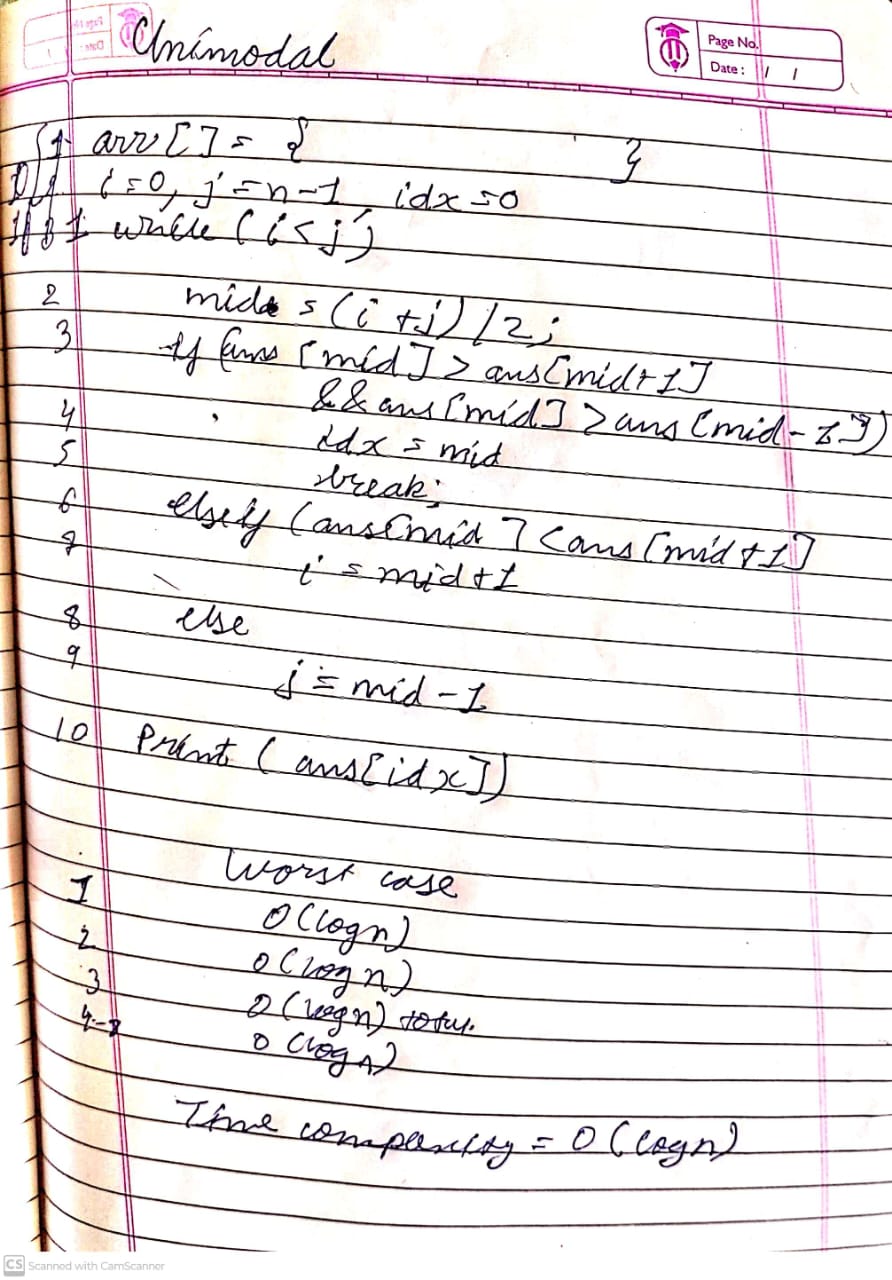
    }

    printf("\nUnimodal peak found at index %d with value %d\n", idx, ans[idx]);

    return 0;

}

## **Analysis:**



## **Output:**

