# **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--;

        }

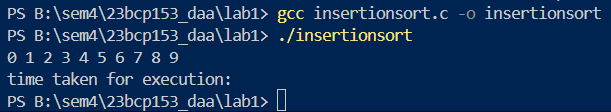
        arr[j + 1] = key;

    }

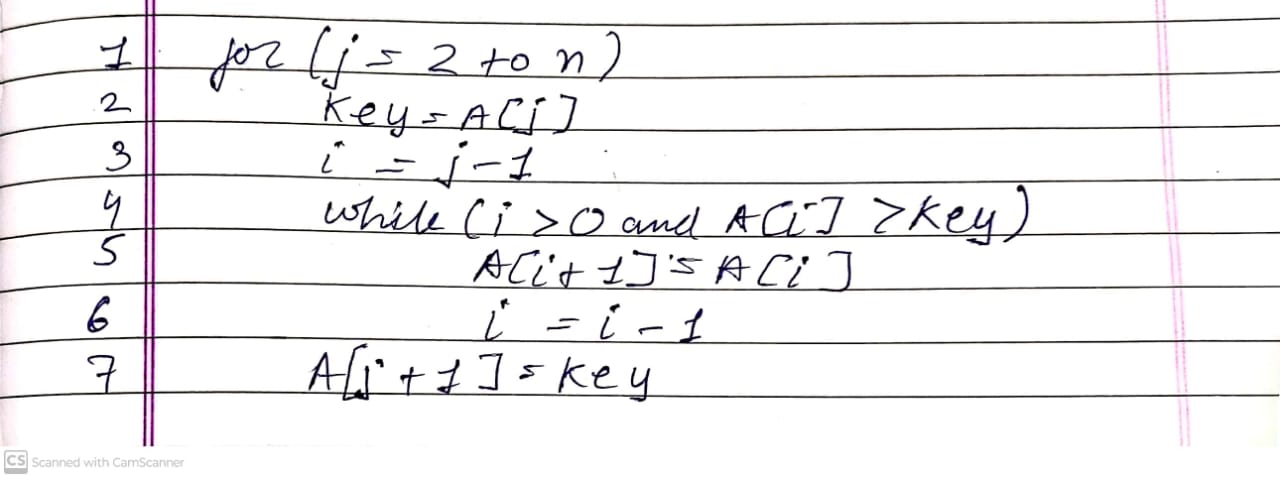
    return;

}

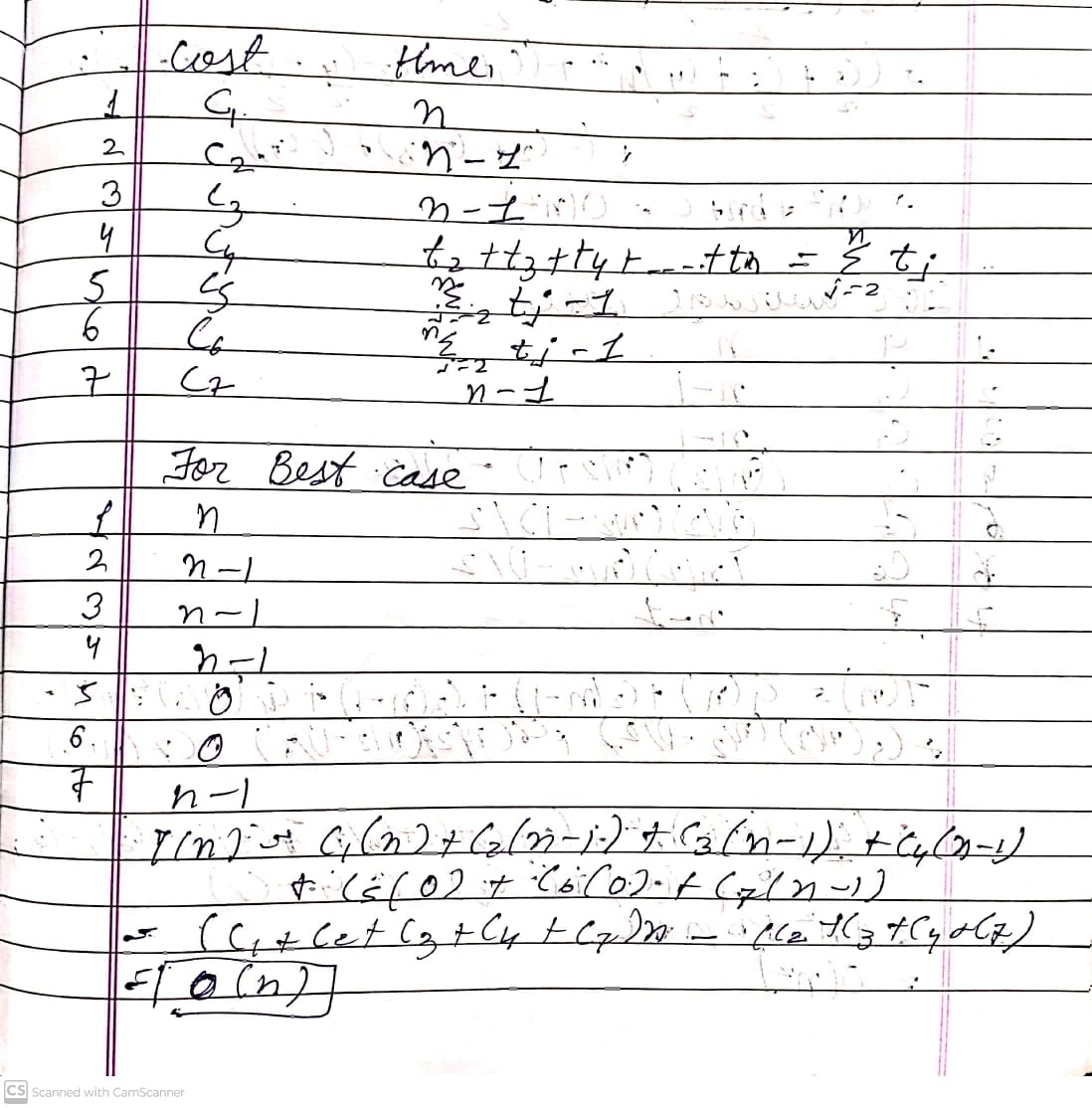
## **Output:**

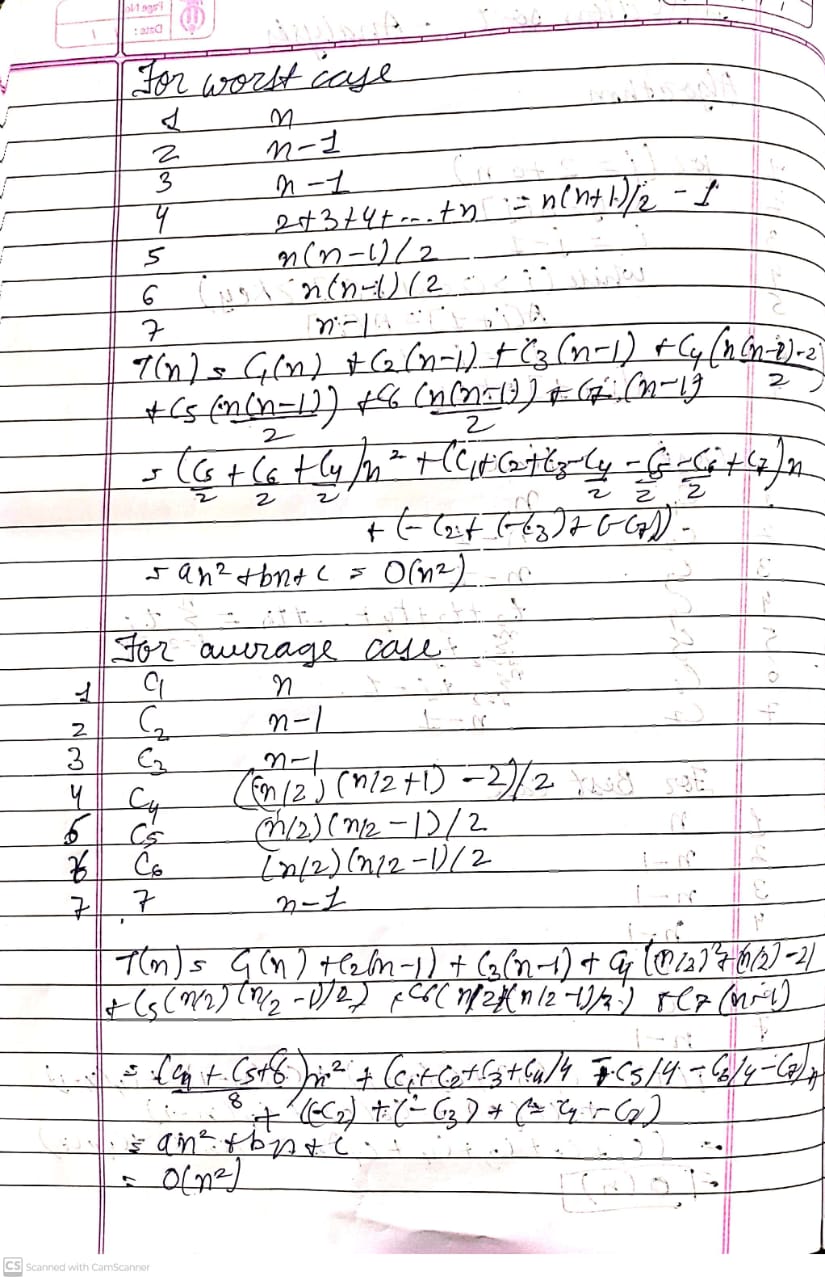


## **Algorithm:**



## **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;

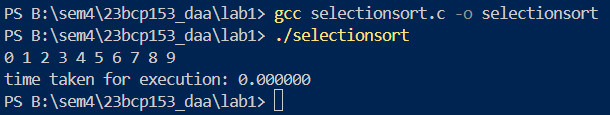
        }

    }

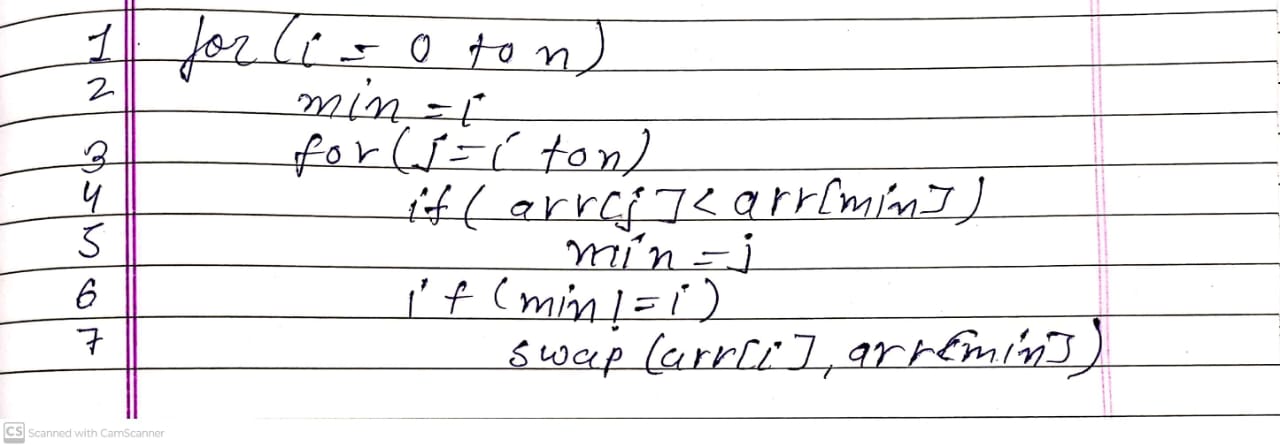
    return;

}

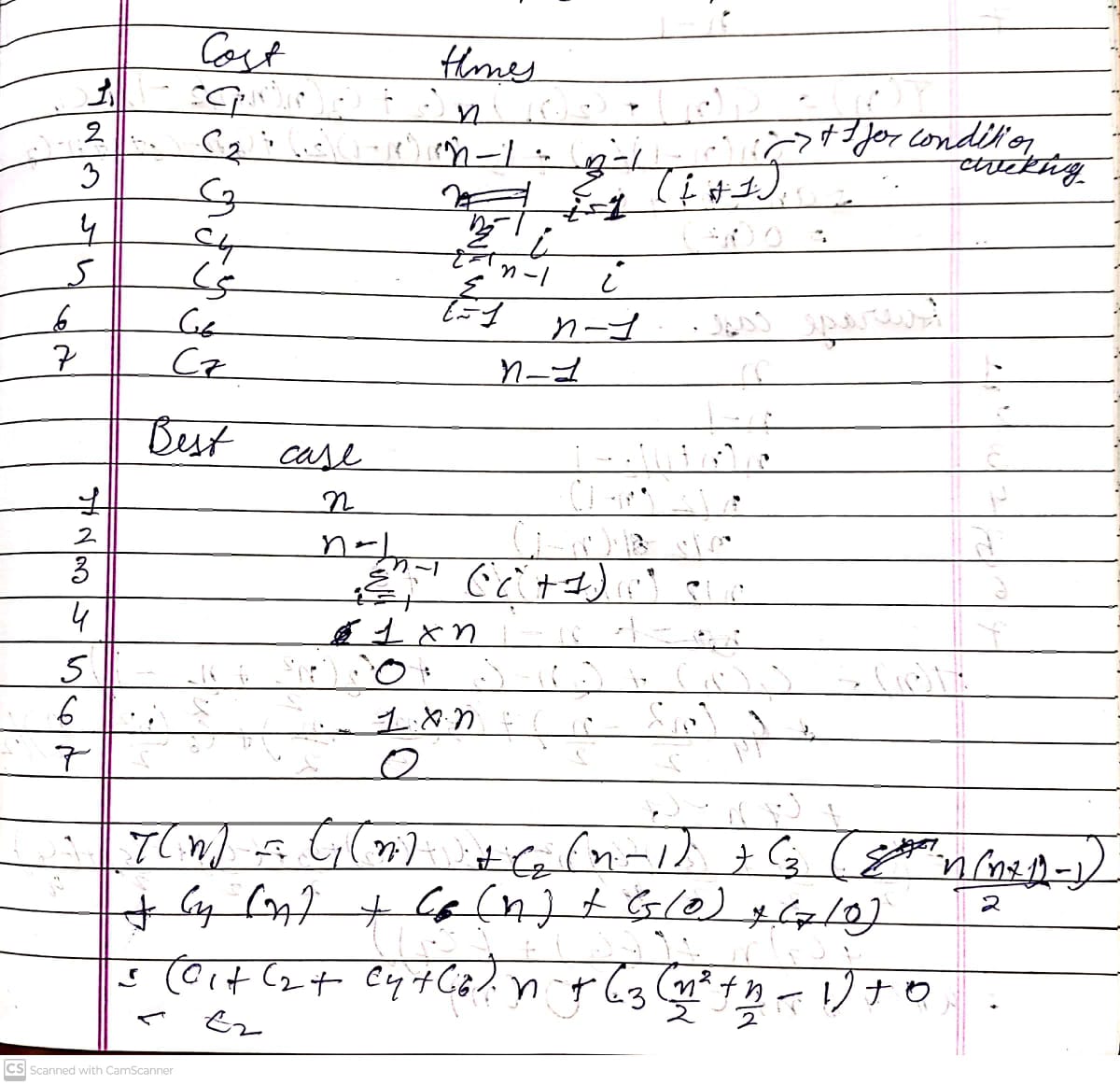
## **Output:**

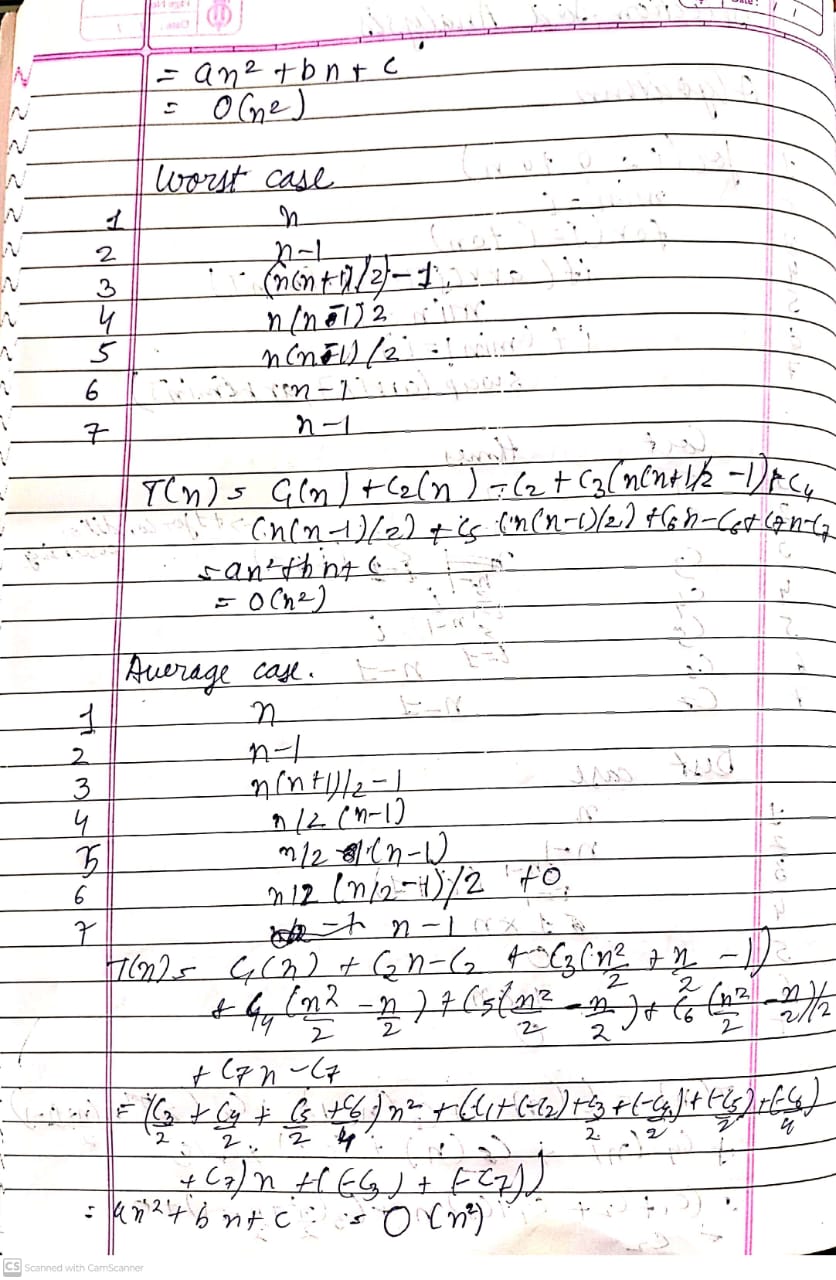


## **Algorithm:**



## **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++;

        }

    }

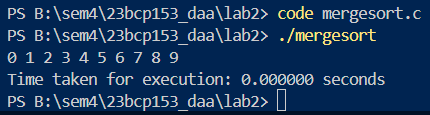
    for (int x = 0; x < (high - low + 1); x++)

        arr[low + x] = arrB[x];

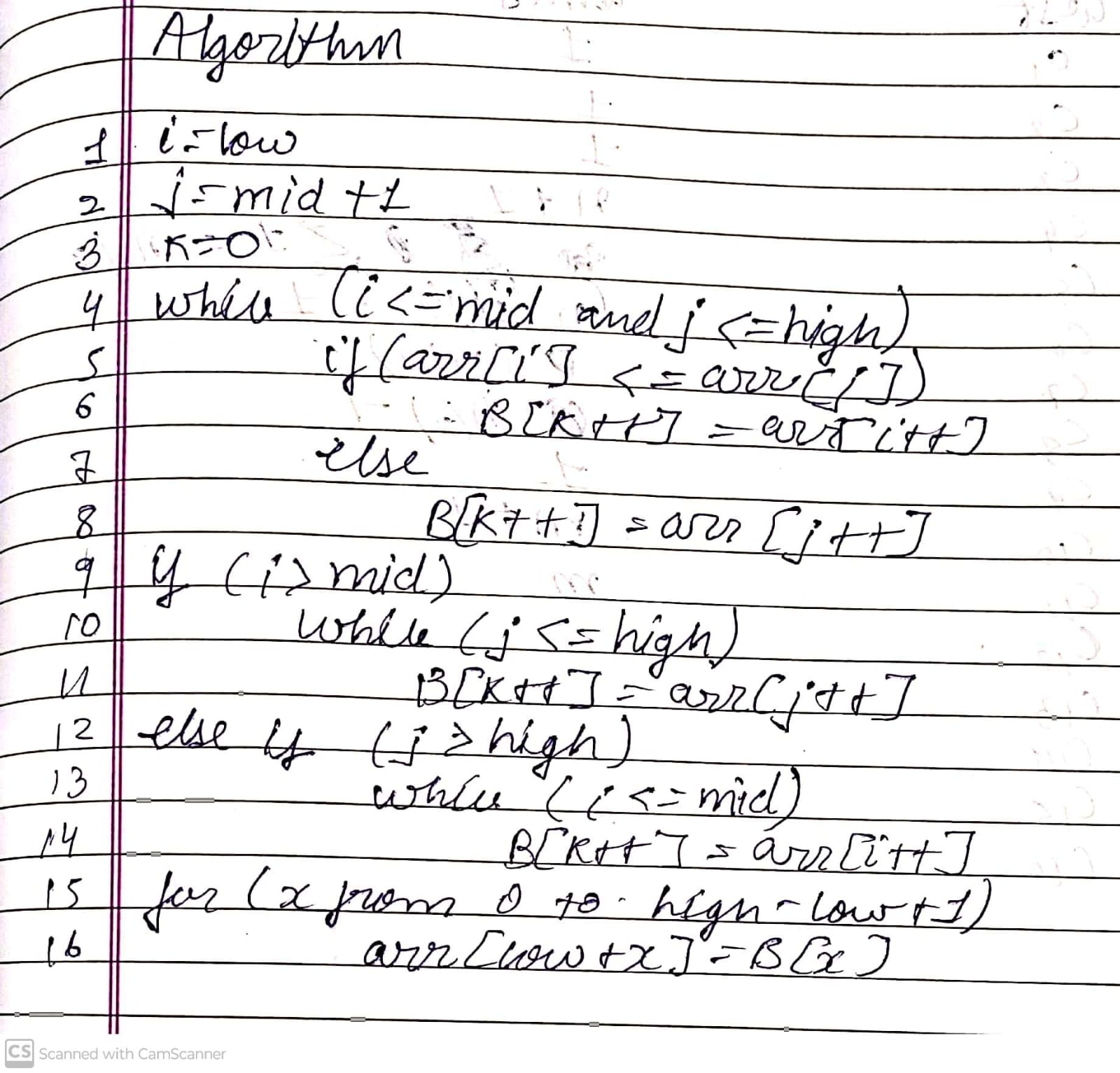
    return;

}

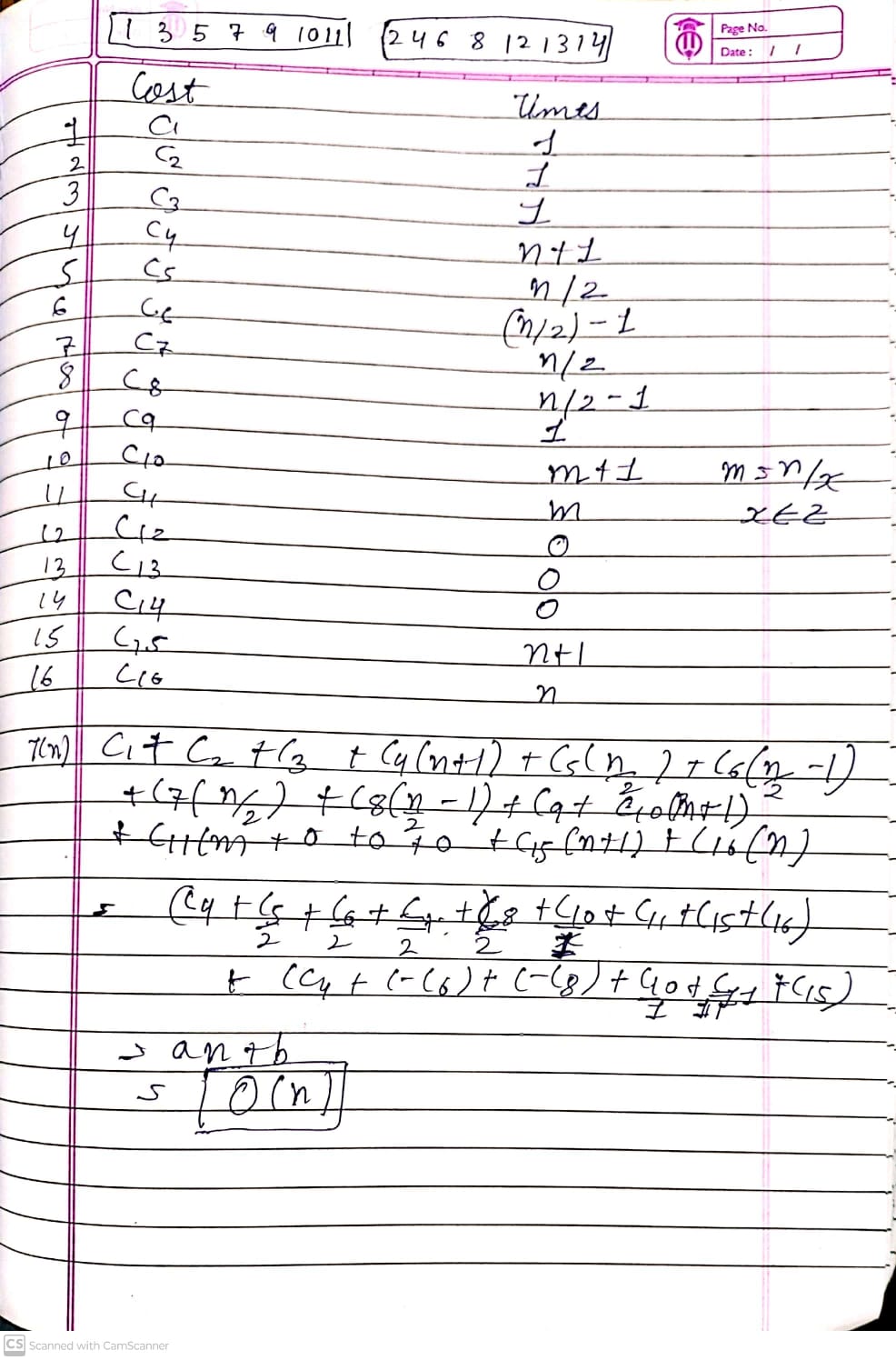
## **Output:**



## **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;

        }

    }

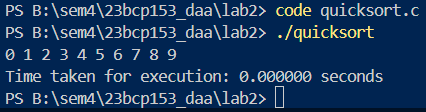
    arr[low] = arr[j];

    arr[j] = pivot;

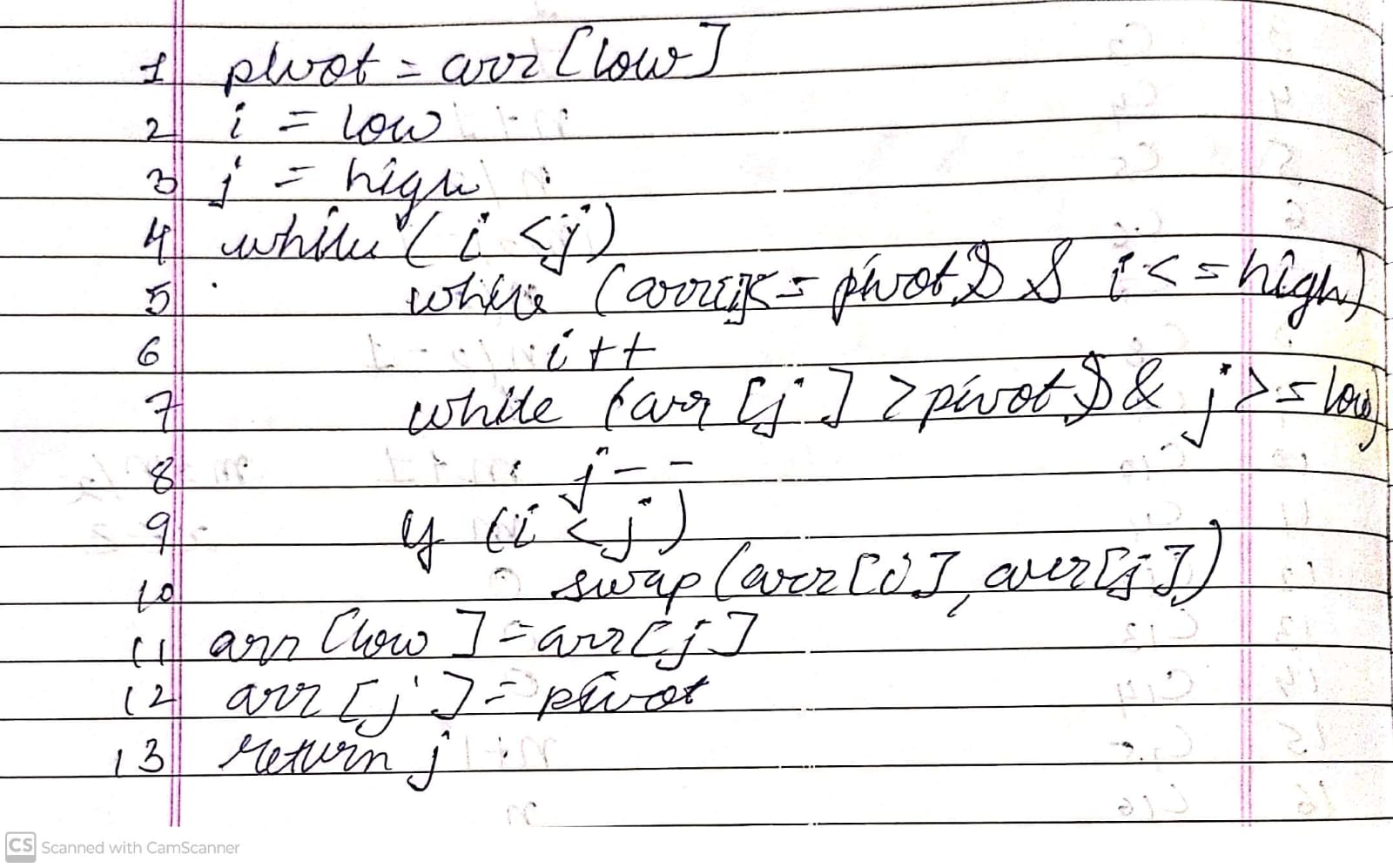
    return j;

}

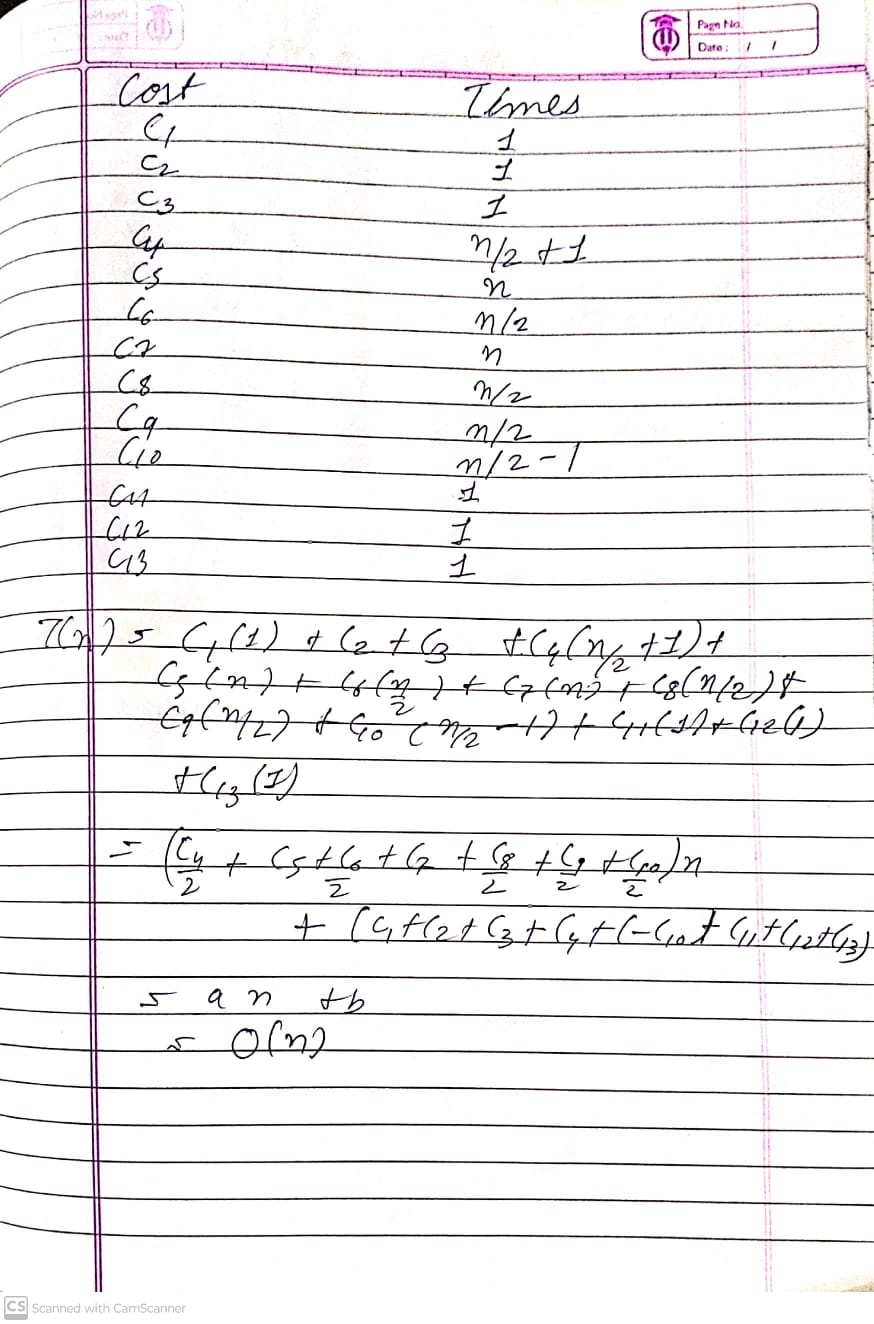
## **Output:**



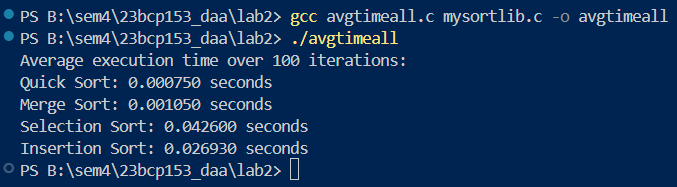
## **Algorithm:**



## **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;

        }

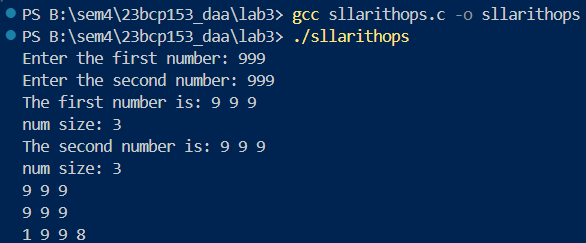
        result->size = size;

    }

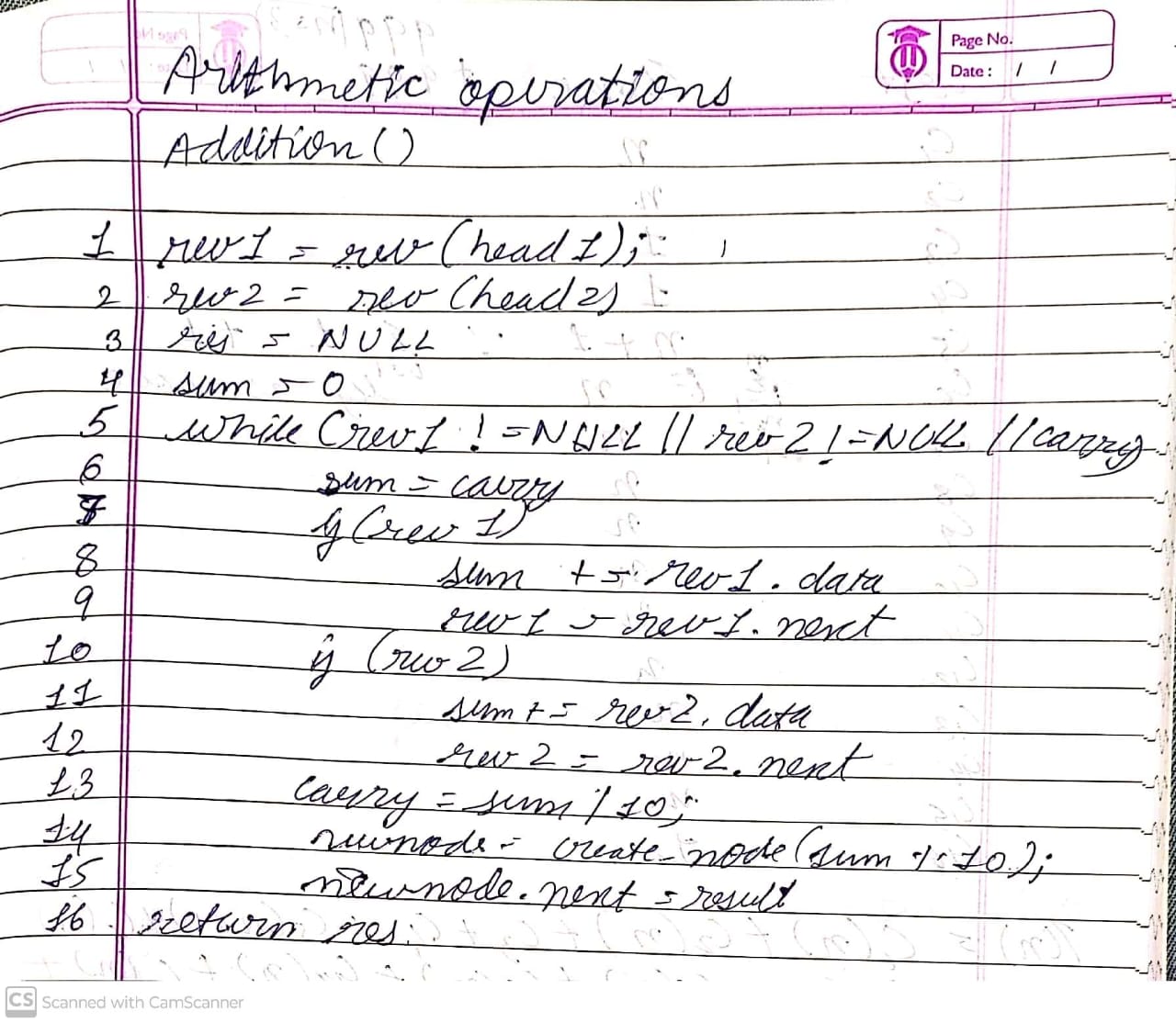
    return result;

}

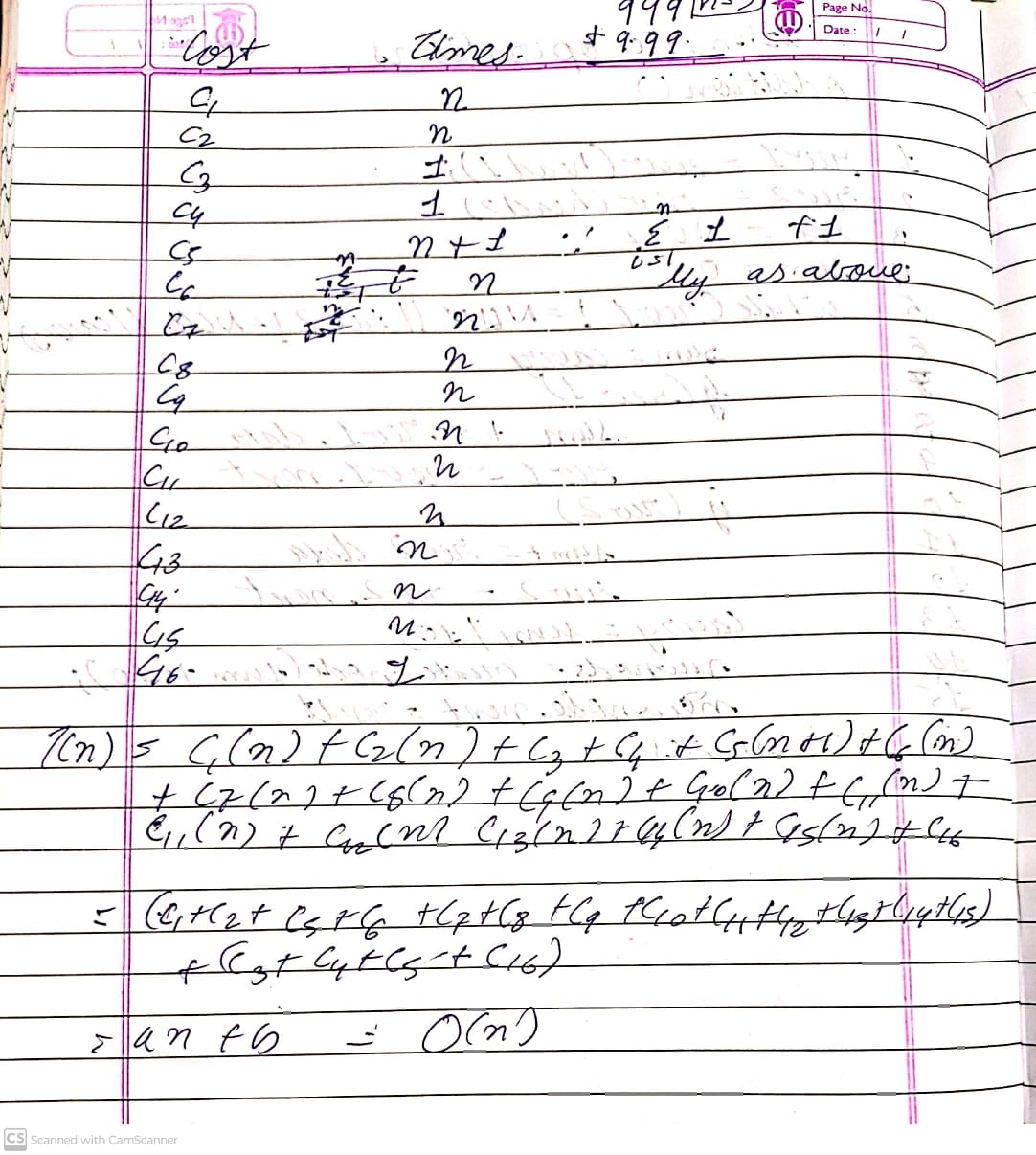
## **Output:**



## **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;

        }

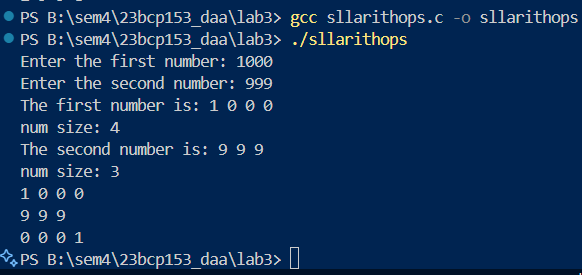
        result->size = size;

    }

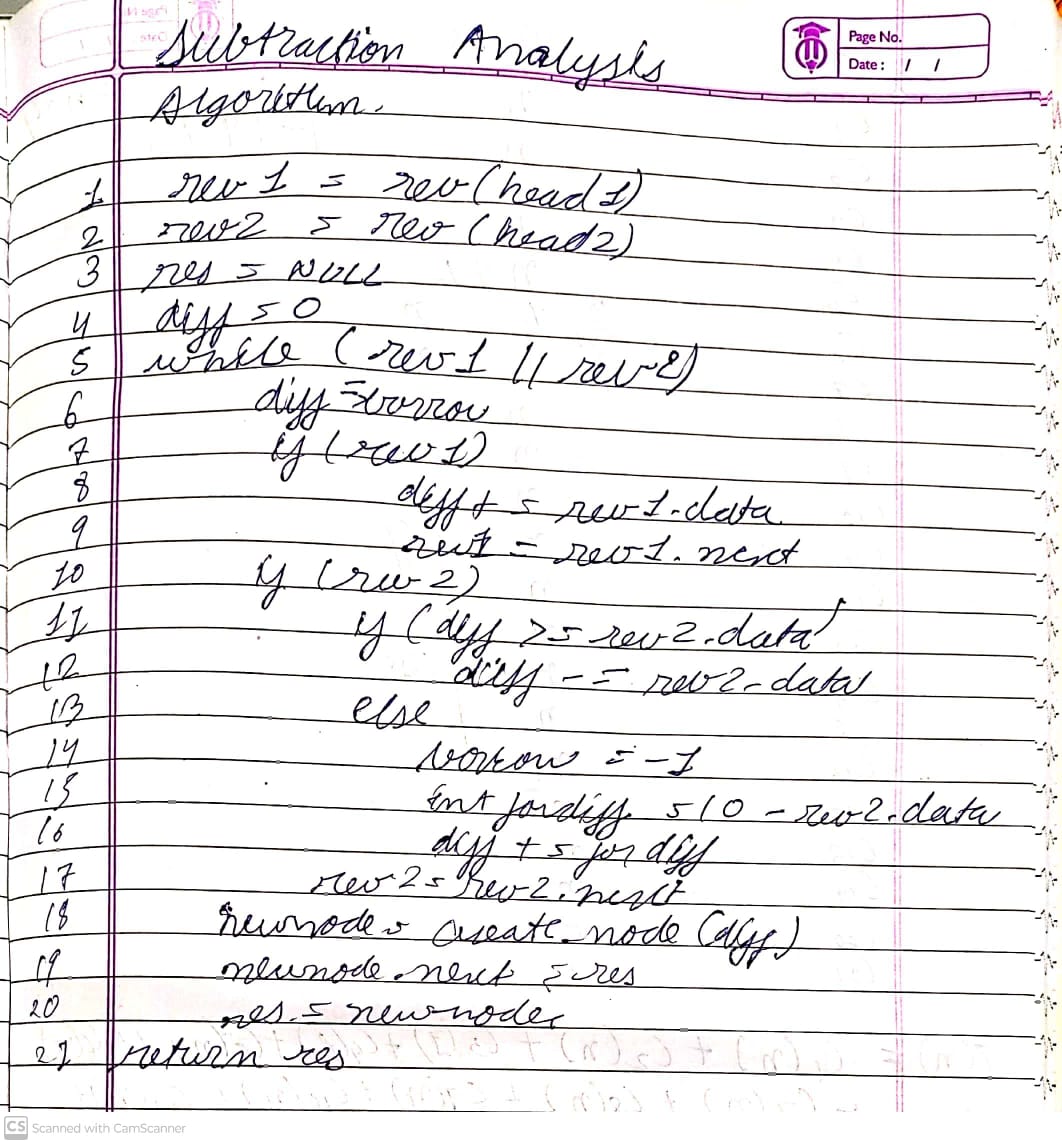
    return result;

}

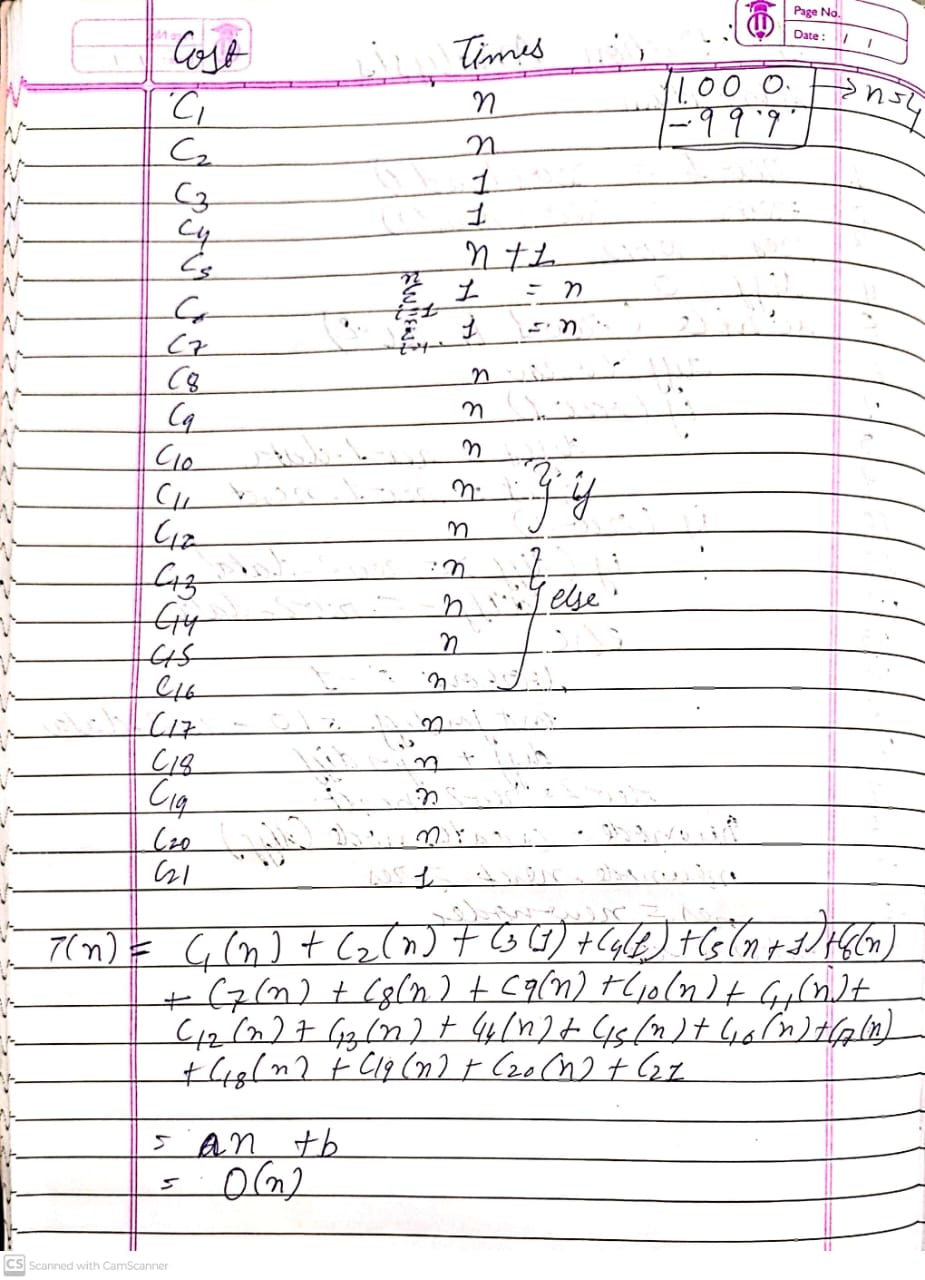
## **Output:**



## **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

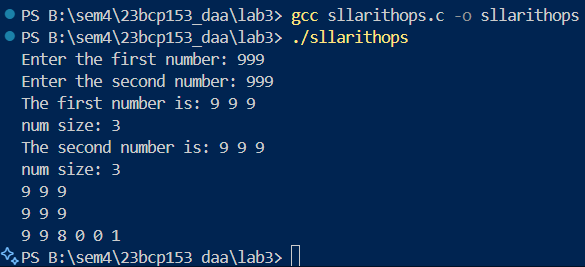
    head1 = reverselist(revhead1);

    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++)

    {

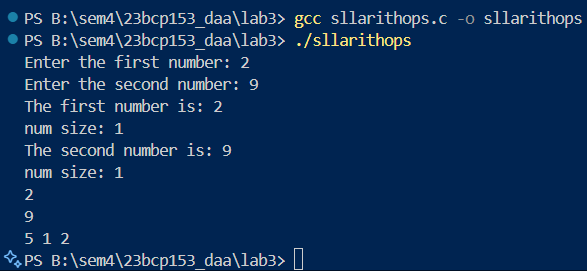
        result = llmuliter(result, head, 0);

    }

    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)

        {

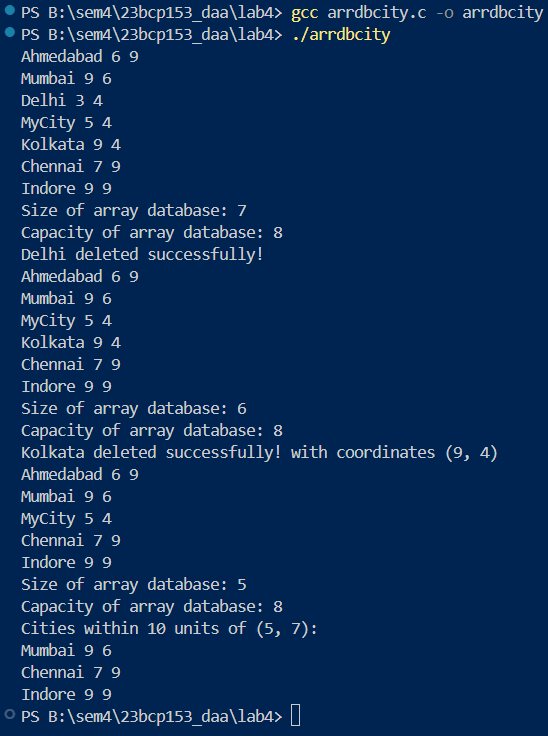
            printf("%s %d %d\n", arrdb->citiesarr[i]->name, arrdb->citiesarr[i]->x, arrdb->citiesarr[i]->y);

        }

    }

}

## **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);

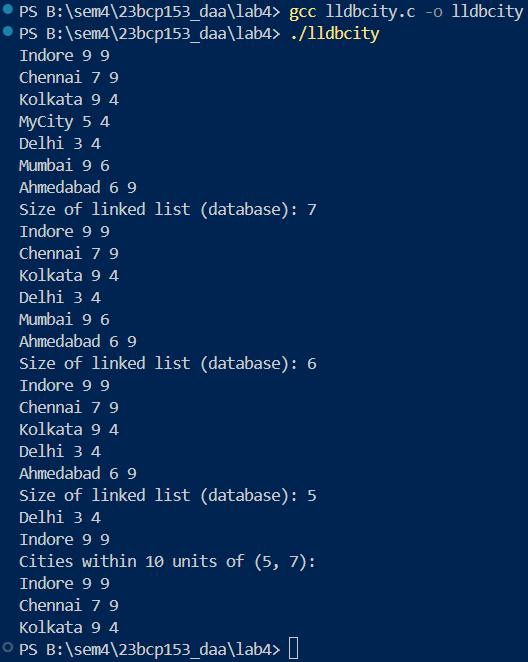
        }

        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