Practical 1: Sorting Arrays-I

Aim

Implement the **Insertion sort** algorithm for an array. Measure the execution time and the number of steps required to execute each algorithm in best case, worst case and average case.

Theory

Insertion sort is a sorting algorithm that places an unsorted element at its suitable place in each iteration. Insertion sort works similarly as we sort cards in our hand in a card game.

We assume that the first card is already sorted then, we select an unsorted card. If the unsorted card is greater than the card in hand, it is placed on the right otherwise, to the left. In the same way, other unsorted cards are taken and put in their right place.

A similar approach is used by insertion sort.

Algorithm

To sort an array of size N in ascending order:

- Iterate from arr[1] to arr[n] over the array.
- Compare the current element (key) to its predecessor.
- If the key element is smaller than its predecessor, compare it to the elements before. Move the greater elements one position up to make space for the swapped element.

Program

```
import java.util.*;

public class InsertionSort {

   static void insertionSort(int array[]) {

      for (int i = 1; i < array.length; i++) {

         int key = array[i];

         for (int j=i-1 ; j>-1 ; j--){

            if ( array [j] > key ){

               array [j+1] = array [j];

               array[j] = key;

            }

         }

      }

   }


   public static void main(String[] args) {
```

```
        Scanner sc = new Scanner(System.in);

        System.out.print("Enter the number of elements: ");
        int length = sc.nextInt();

        int[] array = new int[length];

        for(int k = 0; k < length; k++) {
           System.out.printf("Enter element %d: ", k+1);
           array[k] = sc.nextInt();
        }
        System.out.print("Data you entered: ");
        for(int k = 0; k < length; k ++) {
           System.out.print(array[k] + " ");
        }
        insertionSort(array);
        System.out.print("\nAfter Sorting array: ");
        for(int k = 0; k < length; k ++) {
           System.out.print(array[k] + " ");
        }
    }
}
```

Output

```
Enter the number of elements: 7
Enter element 1: 43
Enter element 2: 67
Enter element 3: 55
Enter element 4: 12
Enter element 5: 98
Enter element 6: 5
Enter element 7: 23
Data you entered: 43 67 55 12 98 5 23
After Sorting array: 5 12 23 43 55 67 98
```

Analysis of Algorithm

i. Worst Case Complexity

Suppose, an array is in descending order, and you want to sort it in ascending order. In this case, worst case complexity occurs. Each element has to be compared with each of the other elements so, for every nth element, (n-1) number of comparisons are made.

Thus, the total number of comparisons = n(n-1) = $\mathbf{n^2}$

ii. Best case Complexity

When the array is already sorted, the outer loop runs for n number of times whereas the inner loop does not run at all. So, there are only **n** number of comparisons. Thus, complexity is linear.

iii. Average case Complexity

It occurs when the elements of an array are in jumbled order. So, we can't say how much comparisons will be made. Hence the average case complexity would be of order on $\mathbf{n^2}$,

Description	**Best Case**	**Average case**	**Worst case**
Occurrence	It occurs when Array is sorted.	It occurs when Array is randomly sorted.	It occurs when Array is reverse sorted, and tj =j
Complexity	O(n)	$O(n^2)$	$O(n^2)$

Aim

Implement the **Selection sort** algorithm for an array. Measure the execution time and the number of steps required to execute each algorithm in best case, worst case and average case.

Theory

The selection sort algorithm sorts an array by repeatedly finding the minimum element (considering ascending order) from the unsorted part and putting it at the beginning.

The algorithm maintains two subarrays in a given array.

- The subarray which already sorted.
- The remaining subarray was unsorted.

In every iteration of the selection sort, the minimum element (considering ascending order) from the unsorted subarray is picked and moved to the beginning of unsorted subarray.

After every iteration sorted subarray size increase by one and unsorted subarray size decrease by one.

After N (size of array) iteration we will get sorted array.

Algorithm

- selectionSort(array, size)
- repeat (size - 1) times
- set the first unsorted element as the minimum
- for each of the unsorted elements
- if element < currentMinimum
- set element as new minimum
- swap minimum with first unsorted position
- end selectionSort

Program

```
import java.util.*;

public class SelectionSort {

   static void selectionSort(int array[]) {

      for (int i = 0 ; i < array.length ; i++) {

         int min = Integer.MAX_VALUE;

         int index = 0;

         for (int j=i ; j<array.length ; j++){

            if (min > array[j]){

               min = array[j];

               index = j;

            }
```

```
            }
            array[index] = array[i];
            array[i] = min;
        }
    }

    public static void main(String[] args) {
        Scanner sc = new Scanner(System.in);

        System.out.print("Enter the number of elements: ");
        int length = sc.nextInt();

        int[] array = new int[length];

        for(int k = 0; k < length; k++) {
            System.out.printf("Enter element %d: ", k+1);
            array[k] = sc.nextInt();
        }
        System.out.print("Data you entered: ");
        for(int k = 0; k < length; k ++) {
            System.out.print(array[k] + " ");
        }

        selectionSort(array);

        System.out.print("\nAfter Sorting array: ");
        for(int k = 0; k < length; k ++) {
            System.out.print(array[k] + " ");
        }
    }
}
```

Output:

```
Enter the number of elements: 7
Enter element 1: 67
Enter element 2: 536
Enter element 3: 254
Enter element 4: 68
Enter element 5: 42
Enter element 6: 342
Enter element 7: 54
Data you entered: 67 536 254 68 42 342 54
After Sorting array: 42 54 67 68 254 342 536
```

Analysis of Algorithm

Cycle	Number of Comparison
1st	(n-1)
2nd	(n-2)
3rd	(n-3)
4th	(n-4)
5th	(n-5)
...	...
last	1

Number of comparisons: (n - 1) + (n - 2) + (n - 3) + ... + 1 = n (n - 1) / 2 nearly equals to $\mathbf{n^2}$.

Whether it is Best/Worst/Average case both loops would execute. Hence, the complexity would be of order of $\mathbf{n^2}$.

Description	Best Case	Average case	Worst case
Occurrence	It occurs when Array is sorted.	It occurs when Array is randomly sorted.	It occurs when Array is reverse sorted, and $t_j = j$
Complexity	$O(n^2)$	$O(n^2)$	$O(n^2)$

Practical 2: Sorting Arrays-II

Aim

Implement the **Merge sort** algorithm for an array. Measure the execution time and the number of steps required to execute each algorithm in best case, worst case and average case.

Theory

Merge Sort is one of the most popular sorting algorithms that is based on the principle of **Divide and Conquer** Algorithm.

Here, a problem is divided into multiple sub-problems. Each sub-problem is solved individually. Finally, sub-problems are combined to form the final solution.

A similar approach is used by merge sort where we divide an array into smaller subarrays, sorting each subarray, and then merging the sorted subarrays back together to form the final sorted array.

Algorithm

- Start
- Declare array and left, right, mid variable
- Perform merge function.

```
if left > right
        return
mid= (left+right)/2
mergesort(array, left, mid)
mergesort(array, mid+1, right)
merge(array, left, mid, right)
```

- Stop

Program

```
import java.util.*;

class MergeSort {

    void merge(int arr[], int p, int q, int r) {

        int n1 = q - p + 1;

        int n2 = r - q;

        int L[] = new int[n1];

        int R[] = new int[n2];

        for (int i = 0; i < n1; ++i){

            L[i] = arr[p + i];

    }

        for (int j = 0; j < n2; ++j){

            R[j] = arr[q + 1 + j];

    }
```

```
        int i = 0, j = 0;
        int k = p;
        while (i < n1 && j < n2) {
            if (L[i] <= R[j]) {
                arr[k] = L[i];
                i++;
            }
            else {
                arr[k] = R[j];
                j++;
            }
            k++;
        }
        while (i < n1) {
            arr[k] = L[i];
            i++;
            k++;
        }
        while (j < n2) {
            arr[k] = R[j];
            j++;
            k++;
        }
    }

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

            sort(arr, l, m);
            sort(arr, m + 1, r);
            merge(arr, l, m, r);
        }
```

```
        }

        static void printArray(int arr[]) {
                int n = arr.length;
                for (int i = 0; i < n; ++i){
                        System.out.print(arr[i] + " ");
    }
                System.out.println();
        }

        public static void main(String args[]) {
    Scanner sc = new Scanner(System.in);

    System.out.print("Enter the number of elements: ");
    int length = sc.nextInt();

    int[] array = new int[length];

    for(int k = 0; k < length; k++) {
        System.out.printf("Enter element %d: ", k+1);
        array[k] = sc.nextInt();
    }

                System.out.println("Given Array");
                printArray(array);

                MergeSort ob = new MergeSort();
                ob.sort(array, 0, array.length - 1);

                System.out.println("\nSorted array");
                printArray(array);
        }
}
```

Output

```
Enter the number of elements: 6
Enter element 1: 65
Enter element 2: 45
Enter element 3: 32
Enter element 4: 78
Enter element 5: 6
Enter element 6: 626
Given Array
65 45 32 78 6 626

Sorted array
6 32 45 65 78 626
```

Analysis of Algorithm

A merge sort consists of several passes over the input.

1st pass - merges segments of size 1

2nd pass - merges segments of size 2

ith pass - merges segments of size 2^{i-1}

Thus, the total number of passes is *logn*. As merge showed, we can merge two sorted segments in linear time, which means that each pass takes *O(n)* time. Since there are [$log_2 n$] passes, the total computing time is ***O(nlogn)***.

The time complexity of Merge Sort is *O(nlog(n))* in all 3 cases (worst, average, and best) as merge sort always divides the array into two halves and takes linear time to merge two halves.

Description	**Best Case**	**Average case**	**Worst case**
Complexity	O(nlog(n))	O(nlog(n))	O(nlog(n))

Aim

Implement the **Quick sort** algorithm for an array. Measure the execution time and the number of steps required to execute each algorithm in best case, worst case and average case.

Theory

Quicksort is a sorting algorithm based on the **divide and conquer** approach where an array is divided into subarrays by selecting a pivot element (element selected from the array).

While dividing the array, the pivot element should be positioned in such a way that elements less than pivot are kept on the left side and elements greater than pivot are on the right side of the pivot.

The left and right subarrays are also divided using the same approach. This process continues until each subarray contains a single element.

At this point, elements are already sorted. Finally, elements are combined to form a sorted array.

Algorithm

```
quickSort(array, leftmostIndex, rightmostIndex)
    if (leftmostIndex < rightmostIndex)
        pivotIndex <- partition(array,leftmostIndex, rightmostIndex)
        quickSort(array, leftmostIndex, pivotIndex - 1)
        quickSort(array, pivotIndex, rightmostIndex)

partition(array, leftmostIndex, rightmostIndex)
    set rightmostIndex as pivotIndex
    storeIndex <- leftmostIndex – 1
    for i <- leftmostIndex + 1 to rightmostIndex
    if element[i] < pivotElement
        swap element[i] and element[storeIndex]
        storeIndex++
    swap pivotElement and element[storeIndex+1]
return storeIndex + 1
```

Program

```
import java.util.*;
class QuickSort {
    static void swap(int[] arr, int i, int j) {
        int temp = arr[i];
        arr[i] = arr[j];
        arr[j] = temp;
```

```
    }
    static int partition(int[] arr, int low, int high) {

        int pivot = arr[high];
        int i = (low - 1);

        for (int j = low; j <= high - 1; j++) {
            if (arr[j] < pivot) {
                i++;
                swap(arr, i, j);
            }
        }
        swap(arr, i + 1, high);
        return (i + 1);
    }
    static void quickSort(int[] arr, int low, int high) {
        if (low < high) {
            int pi = partition(arr, low, high);
            quickSort(arr, low, pi - 1);
            quickSort(arr, pi + 1, high);
        }
    }
    static void printArray(int[] arr, int size) {
        for (int i = 0; i < size; i++){
            System.out.print(arr[i] + " ");
  }
        System.out.println();
    }

public static void main(String[] args){
    Scanner sc = new Scanner(System.in);
    System.out.print("Enter the number of elements: ");
    int n = sc.nextInt();
```

```
        int[] array = new int[n];
        for(int k = 0; k < n; k++) {
           System.out.printf("Enter element %d: ", k+1);
           array[k] = sc.nextInt();
        }
                System.out.println("Given Array");
                printArray(array, n);
                quickSort(array, 0, n - 1);
                System.out.println("\nSorted array: ");
                printArray(array, n);
           }
}
```

Output:

```
Enter the number of elements: 5
Enter element 1: 4
Enter element 2: 617
Enter element 3: 7
Enter element 4: 54
Enter element 5: 21
Given Array
4 617 7 54 21

Sorted array:
4 7 21 54 617
```

Analysis of Algorithm

1. **Worst Case Complexity**

   It occurs when the pivot element picked is either the greatest or the smallest element. This condition leads to the case in which the pivot element lies in an extreme end of the sorted array. One sub-array is always empty and another sub-array contains n - 1 elements. Thus, quicksort is called only on this sub-array.

2. **Best Case Complexity**

   It occurs when the pivot element is always the middle element or near to the middle element.

3. **Average Case Complexity**

   It occurs when the above conditions do not occur.

Description	**Best Case**	**Average case**	**Worst case**
Complexity	O(nlog(n))	O(nlog(n))	$O(n^2)$

Practical 3: Applications of Linked Lists

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?

Program

1. **Addition**

```
package PRACTICAL3;

public class Addition {
   static Node head1, head2;

   static class Node {
      int data;
      Node next;
      Node(int d) {
         data = d;
         next = null;
      }
   }

   void addTwoLists(Node first, Node second) {
      Node start1 = new Node(0);
      start1.next = first;
      Node start2 = new Node(0);
      start2.next = second;

      addPrecedingZeros(start1, start2);
      Node result = new Node(0);
      if (sumTwoNodes(start1.next, start2.next, result) == 1) {
         Node node = new Node(1);
         node.next = result.next;
         result.next = node;
      }
      printList(result.next);
   }

   // Adds lists and returns the carry
   private int sumTwoNodes(Node first, Node second, Node result) {
      if (first == null) {
         return 0;
      }
      int number = first.data + second.data + sumTwoNodes(first.next, second.next, result);
      Node node = new Node(number % 10);
      node.next = result.next;
```

```
    result.next = node;
    return number / 10;
}

// Appends preceding zeros in case a list is having lesser nodes than the other one
private void addPrecedingZeros(Node start1, Node start2) {
    Node next1 = start1.next;
    Node next2 = start2.next;
    while (next1 != null && next2 != null) {
        next1 = next1.next;
        next2 = next2.next;
    }
    if (next1 == null && next2 != null) {
        while (next2 != null) {
            Node node = new Node(0);
            node.next = start1.next;
            start1.next = node;
            next2 = next2.next;
        }
    } else if (next2 == null && next1 != null) {
        while (next1 != null) {
            Node node = new Node(0);
            node.next = start2.next;
            start2.next = node;
            next1 = next1.next;
        }
    }
}

// Function to print linked list
void printList(Node head) {
    while (head != null) {
        System.out.print(head.data + " ");
        head = head.next;
    }
    System.out.println("");
}

public static void main(String[] args) {
    Addition list = new Addition();

    // creating first list
    head1 = new Node(7);
    head1.next = new Node(4);
    head1.next.next = new Node(1);
    head1.next.next.next = new Node(6);
    head1.next.next.next.next = new Node(7);
    System.out.print("First list :");
    list.printList(head1);

    // creating second list
    head2 = new Node(4);
```

```
        head2.next = new Node(3);
        System.out.print("Second list : ");
        list.printList(head2);

        System.out.print("Added list : ");
        list.addTwoLists(head1, head2);
    }
}
```

Output

```
First list :7 4 1 6 7
Second list : 4 3
Added list : 7 4 2 1 0
```

2. Subtraction

```
package PRACTICAL3;

import java.lang.*;

class Subtraction {
    static Node head;
    boolean borrow;

    static class Node {
        int data;
        Node next;
        Node(int d){
            data = d;
            next = null;
        }
    }

    int getLength(Node node){
        int size = 0;
        while (node != null) {
            node = node.next;
            size++;
        }
        return size;
    }

    Node paddZeros(Node sNode, int diff){
        if (sNode == null) {
            return null;
        }
        Node zHead = new Node(0);
        diff--;
        Node temp = zHead;
        while ((diff--) != 0) {
```

```
            temp.next = new Node(0);
            temp = temp.next;
        }
        temp.next = sNode;
        return zHead;
    }

    Node subtractLinkedListHelper(Node l1, Node l2) {
        if (l1 == null && l2 == null && borrow == false)
            return null;

        Node previous
                = subtractLinkedListHelper(
                (l1 != null) ? l1.next
                        : null,
                (l2 != null) ? l2.next : null);

        int d1 = l1.data;
        int d2 = l2.data;
        int sub = 0;
        if (borrow) {
            d1--;
            borrow = false;
        }
        if (d1 < d2) {
            borrow = true;
            d1 = d1 + 10;
        }
        sub = d1 - d2;
        Node current = new Node(sub);
        current.next = previous;
        return current;
    }
    Node subtractLinkedList(Node l1, Node l2) {
        if (l1 == null && l2 == null){
            return null;
        }
        int len1 = getLength(l1);
        int len2 = getLength(l2);

        Node lNode = null, sNode = null;
        Node temp1 = l1;
        Node temp2 = l2;
        if (len1 != len2) {
            lNode = len1 > len2 ? l1 : l2;
            sNode = len1 > len2 ? l2 : l1;
            sNode = paddZeros(sNode, Math.abs(len1 - len2));
        }
```

```
        else {
            while (l1 != null && l2 != null) {
                if (l1.data != l2.data) {
                    lNode = l1.data > l2.data ? temp1 : temp2;
                    sNode = l1.data > l2.data ? temp2 : temp1;
                    break;
                }
                l1 = l1.next;
                l2 = l2.next;
            }
        }
        borrow = false;
        return subtractLinkedListHelper(lNode, sNode);
    }

    static void printList(Node head){
        Node temp = head;
        while (temp != null) {
            System.out.print(temp.data + " ");
            temp = temp.next;
        }
    }

    public static void main(String[] args) {
        Node head = new Node(3);
        head.next = new Node(8);
        head.next.next = new Node(5);
        System.out.print("First list : ");
        printList(head);

        Node head2 = new Node(4);
        head2.next = new Node(3);
        System.out.print("\nSecond list : ");
        printList(head2);

        Subtraction ob = new Subtraction();
        Node result = ob.subtractLinkedList(head, head2);
        System.out.print("\nSubtracted list : ");
        printList(result);
    }
}
```

Output

```
First list : 3 8 5
Second list : 4 3
Subtracted list : 3 4 2
```

3. Multiplication

```
package PRACTICAL3;

public class Multiplication {
   static class Node {
      int data;
      Node next;
      Node(int data){
         this.data = data;
         next = null;
      }
   }

   static long multiplyTwoLists(Node first, Node second) {
      long N = 1000000007;
      long num1 = 0, num2 = 0;

      while (first != null || second != null){
         if(first != null){
            num1 = ((num1)*10)%N + first.data;
            first = first.next;
         }
         if(second != null) {
            num2 = ((num2)*10)%N + second.data;
            second = second.next;
         }
      }
      return ((num1%N)*(num2%N))%N;
   }

   static void printList(Node node){
      while(node != null) {
         System.out.print(node.data);
         if(node.next != null)
            System.out.print(" ");
         node = node.next;
      }
      System.out.println();
   }

   public static void main(String args[]) {
      Node first = new Node(9);
      first.next = new Node(4);
      first.next.next = new Node(6);
      System.out.print("First List : ");
      printList(first);

      Node second = new Node(8);
```

```
        second.next = new Node(4);
        System.out.print("Second List : ");
        printList(second);

        System.out.print("Multiplied list : ");
        System.out.println(multiplyTwoLists(first, second));
    }
}
```

Output

```
First List : 9 4 6
Second List : 8 4
Multiplied list : 79464
```

Analysis of Algorithm

For the singly linked list implementation of integers of unlimited size, the asymptotic running time for each of the operations would be as follows:

- **Addition**: O(n + m), where n and m are the number of digits for the two operands. This is because each node of the linked list must be traversed in order to add each corresponding digit, and the operation will take longer if the operand with more digits is used.
- **Subtraction**: O(n + m), where n and m are the number of digits for the two operands. This is similar to addition, as each node of the linked list must be traversed in order to subtract each corresponding digit.
- **Multiplication**: O(n * m), where n and m are the number of digits for the two operands. This is because for each digit of the first operand, the entire second operand must be traversed in order to multiply each corresponding digit.

Practical 4: City Databases using Linked List

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) Point 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

Program

a. Array based Implementation

```
public class CityDatabaseArray {

   private static final int INITIAL_CAPACITY = 10;
   private String[] cityNames;
   private int[] xCoords;
   private int[] yCoords;
   private int size;

   public CityDatabaseArray() {
      cityNames = new String[INITIAL_CAPACITY];
      xCoords = new int[INITIAL_CAPACITY];
      yCoords = new int[INITIAL_CAPACITY];
      size = 0;
   }

   // Inserts a record with the given name and coordinates
   public void insert(String name, int x, int y) {
      if (size >= cityNames.length) {
         resize();
      }
      cityNames[size] = name;
      xCoords[size] = x;
      yCoords[size] = y;
      size++;
   }

   // Deletes a record with the given name or coordinates
   public void delete(String nameOrCoord) {
      for (int i = 0; i < size; i++) {
         if (cityNames[i].equals(nameOrCoord) || (xCoords[i] + "," + yCoords[i]).equals(nameOrCoord)) {
            cityNames[i] = cityNames[size - 1];
            xCoords[i] = xCoords[size - 1];
```

```
            yCoords[i] = yCoords[size - 1];
            size--;
            return;
        }
    }
}

// Searches for a record with the given name or coordinates and returns its index, or -1 if not found
public int search(String nameOrCoord) {
    for (int i = 0; i < size; i++) {
        if (cityNames[i].equals(nameOrCoord) || (xCoords[i] + "," + yCoords[i]).equals(nameOrCoord)) {
            return i;
        }
    }
    System.out.println("City is:");
    return -1;
}

// Prints all records within the given distance of the specified point
public void printNearby(int x, int y, double distance) {
    for (int i = 0; i < size; i++) {
        double dx = xCoords[i] - x;
        double dy = yCoords[i] - y;
        double dist = Math.sqrt(dx*dx + dy*dy);
        if (dist <= distance) {
            System.out.println(cityNames[i] + " (" + xCoords[i] + "," + yCoords[i] + ")");
        }
    }
}

// Resizes the arrays to twice their current capacity
private void resize() {
    int newCapacity = 2 * cityNames.length;
    String[] newCityNames = new String[newCapacity];
    int[] newXCoords = new int[newCapacity];
    int[] newYCoords = new int[newCapacity];
    for (int i = 0; i < size; i++) {
        newCityNames[i] = cityNames[i];
        newXCoords[i] = xCoords[i];
        newYCoords[i] = yCoords[i];
    }
    cityNames = newCityNames;
    xCoords = newXCoords;
    yCoords = newYCoords;
}
```

```
    public static void main(String[] args) {
        CityDatabaseArray db = new CityDatabaseArray();

            db.insert("Berlin",50,60);
            db.insert("Tokyo",40 ,70);
            db.insert("Berlin",50,90);
            db.insert("Delhi",20 ,70);
            db.search("Berlin");
            db.printNearby(50,40,20);

    }
}
```

Output:

```
Berlin (50,60)
```

b. Linked List Implementation

```
    public class CityDataBaseLinked {
        String name;
        int x;
        int y;
        CityDataBaseLinked next;

        public CityDataBaseLinked(String name, int x, int y) {
            this.name = name;
            this.x = x;
            this.y = y;
            next = null;
        }

        public String toString() {
            return name + " (" + x + "," + y + ")";
        }
    }

    class CityDatabase {

        private CityDataBaseLinked head;
        private int size;

        public CityDatabase() {
            head = null;
            size = 0;
        }
```

```
    // Inserts a record with the given name and coordinates
    public void insert(String name, int x, int y) {
        CityDataBaseLinked newCity = new CityDataBaseLinked(name, x, y);
        newCity.next = head;
        head = newCity;
        size++;
    }

    // Deletes a record with the given name or coordinates
    public void delete(String nameOrCoord) {
        if (head == null) {
            return;
        }
        if (head.name.equals(nameOrCoord) || (head.x + "," + head.y).equals(nameOrCoord)) {
            head = head.next;
            size--;
            return;
        }
        CityDataBaseLinked curr = head;
        while (curr.next != null) {
            if (curr.next.name.equals(nameOrCoord) || (curr.next.x + "," +
curr.next.y).equals(nameOrCoord)) {
                curr.next = curr.next.next;
                size--;
                return;
            }
            curr = curr.next;
        }
    }

    // Searches for a record with the given name or coordinates and returns its index, or -1 if not found
    public CityDataBaseLinked search(String nameOrCoord) {
        CityDataBaseLinked curr = head;
        while (curr != null) {
            if (curr.name.equals(nameOrCoord) || (curr.x + "," + curr.y).equals(nameOrCoord)) {
                return curr;
            }
            curr = curr.next;
        }
        return null;
    }

    // Prints all records within the given distance of the specified point
    public void printNearby(int x, int y, double distance) {
        CityDataBaseLinked curr = head;
        while (curr != null) {
            double dx = curr.x - x;
            double dy = curr.y - y;
```

```
            double dist = Math.sqrt(dx * dx + dy * dy);
            if (dist <= distance) {
                System.out.println(curr);
            }
            curr = curr.next;
        }
    }

    // Returns the size of the database
    public int size() {
        return size;
    }

    public static void main(String[] args) {


        CityDatabase db = new CityDatabase();

        db.insert("New York", 0, 0);
        db.insert("Los Angeles", 100, 0);
        db.insert("Chicago", 50, 50);

        CityDataBaseLinked city = db.search("Chicago");
        if (city != null) {
            System.out.println("Found: " + city);
        }

        db.delete("New York");

        db.printNearby(0, 0, 50);

    }
}
```

Output:

```
Found: Chicago (50,50)
```

Analysis of Algorithms

a) Collect running time statistics for each operation in both implementations.

Insert Operation	Delete Operation	Search Operation	printNearby Operation
0(1)	0(N)	0(N)	0(N)

Overall, the worst-case time complexity of this implementation is O(n) for most operations, except for the insert operation which has an average-case time complexity of O(1) and a worst-case time complexity of **O(n)** if a resize is needed.

b) What are your conclusions about the relative advantages and disadvantages of the two implementations?

When it comes to searching and deleting elements, an implementation using an array is more efficient, whereas implementing a **linked list is more efficient** for inserting elements.

c) Would storing records on the list in alphabetical order by city name speed any of the operations?

If records are organized in the list according to the alphabetical order of the city name, it would **speed up the search** operation because the **binary search algorithm** could be used to access the elements.

d) Would keeping the list in alphabetical order slow any of the operations?

Inserting new elements into the list while maintaining alphabetical order would **slow down** the insertion operation as new elements would have to be added to the correct position to preserve the alphabetical order.

Practical 5: Matrix Multiplication using DnC

Aim

Implement both a **standard $O(n^3)$ matrix multiplication** algorithm and **Strassen's matrix multiplication** algorithm. Using empirical testing, try and estimate the constant factors for the runtime equations of the two algorithms. How big must n be before Strassen's algorithm becomes more efficient than the standard algorithm?

Algorithm

1. Standard Matrix Multiplication

For each row i of matrix A, and for each column j of matrix B:

a. Initialize a variable sum to zero.

b. For each index k from 1 to n:

- **i.** Multiply the element in the i-th row and k-th column of matrix A by the element in the k-th row and j-th column of matrix B.
- **ii.** Add the result to the variable sum.

c. Assign the value of the variable sum to the element in the i-th row and j-th column of matrix C.

d. Return the result matrix C.

The above algorithm follows the traditional method of matrix multiplication,

2. Strassen's Matrix Multiplication

- If the size of the matrices is 1, perform regular multiplication of the matrices to obtain the result matrix.
- If the size of the matrices is greater than 1, split the matrices into four smaller matrices of size n/2 x n/2.
- Recursively compute the products of the smaller matrices using the same algorithm.
- Combine the results of the smaller matrices to obtain the final result matrix.
- Create a result matrix C of size m x p, initialized with zeros.

Program

a. Standard Matrix Multiplication

```
package PRACTICAL5;

import java.util.*;

public class MatrixMultiply {
   public static void main(String[] args) {
      Scanner sc = new Scanner(System.in);
      int i,j,k;

      MatrixMultiply obj = new MatrixMultiply();

      System.out.print("Enter no. of rows of 1st array: ");
      int row1 = sc.nextInt();
      System.out.print("Enter no. of columns of 1st array: ");
      int col1 = sc.nextInt();
```

```
        System.out.print("Enter no. of rows of 2nd array: ");
        int row2 = sc.nextInt();
        System.out.print("Enter no. of columns of 2nd array: ");
        int col2 = sc.nextInt();

        if (col1 == row2){
            // First Array
            System.out.println("\nEnter elements of 1st array");
            int a[][] = new int[row1][col1];
            for (i=0; i<row1; i++){
                for (j=0; j<col1; j++){
                    a[i][j] = (int)(Math.random()*(20)+1);
                }
            }
            System.out.println("1st Array: "+Arrays.deepToString(a));

            // Second Array
            System.out.println("\nEnter elements of 2nd array");
            int b[][] = new int[row2][col2];
            for (i=0; i<row2; i++){
                for (j=0; j<col2; j++){
                    b[i][j] = (int)(Math.random()*(50-21+1)+1);
                }
            }
            System.out.println("2nd Array: "+Arrays.deepToString(a));

            // Multiplying Array
            int c[][] = new int[row1][col2];
            for (i = 0; i < row1; i++) {
                for (j = 0; j < col2; j++) {
                    for (k = 0; k < row2; k++){
                        c[i][j] += a[i][k] * b[k][j];
                    }
                }
            }
            System.out.print("\nThe Multiplied array is ");
            System.out.println(Arrays.deepToString(c));
        }

        else{
            System.out.println("\nArrays can't be multiplied!!");
        }
        sc.close();
    }
}
```

Output:

```
Enter no. of rows of 1st array: 3
Enter no. of columns of 1st array: 2
Enter no. of rows of 2nd array: 2
Enter no. of columns of 2nd array: 3

Enter elements of 1st array
1st Array: [[5, 10], [18, 14], [7, 19]]

Enter elements of 2nd array
2nd Array: [[5, 10], [18, 14], [7, 19]]

The Multiplied array is [[190, 305, 250], [310, 548, 614], [351, 552, 415]]
```

b. Strassen's matrix multiplication

```
package PRACTICAL5;

import java.util.*;

class MMStrassens {

    static int ROW_1 = 3,COL_1 = 2, ROW_2 = 2, COL_2 = 4;

    public static void printMat(int[][] a, int r, int c){
        for(int i=0;i<r;i++){
            for(int j=0;j<c;j++){
                System.out.print(a[i][j]+" ");
            }
            System.out.println();
        }
        System.out.println();
    }

    public static void print(String display, int[][] matrix,int start_row, int start_column, int end_row,int
end_column)
    {
        System.out.println(display + " =>\n");
        for (int i = start_row; i <= end_row; i++) {
            for (int j = start_column; j <= end_column; j++) {
                //cout << setw(10);
                System.out.print(matrix[i][j]+" ");
            }
            System.out.println();
        }
        System.out.println();
    }
```

```
    public static void add_matrix(int[][] matrix_A,int[][] matrix_B,int[][] matrix_C, int split_index)
    {
        for (int i = 0; i < split_index; i++){
            for (int j = 0; j < split_index; j++){
                matrix_C[i][j] = matrix_A[i][j] + matrix_B[i][j];
            }
        }
    }

    public static void initWithZeros(int[][] a, int r, int c){
        for(int i=0;i<r;i++){
            for(int j=0;j<c;j++){
                a[i][j]=0;
            }
        }
    }

    public static int[][] multiply_matrix(int[][] matrix_A,int[][] matrix_B)
    {
        int col_1 = matrix_A[0].length;
        int row_1 = matrix_A.length;
        int col_2 = matrix_B[0].length;
        int row_2 = matrix_B.length;

        if (col_1 != row_2) {
            System.out.println("\nError: The number of columns in Matrix A  must be equal to the number
of rows in Matrix B\n");
            int[][] temp = new int[1][1];
            temp[0][0]=0;
            return temp;
        }

        int[] result_matrix_row = new int[col_2];
        Arrays.fill(result_matrix_row,0);
        int[][] result_matrix = new int[row_1][col_2];
        initWithZeros(result_matrix,row_1,col_2);

        if (col_1 == 1){
            result_matrix[0][0] = matrix_A[0][0] * matrix_B[0][0];
        } else {
            int split_index = col_1 / 2;

            int[] row_vector = new int[split_index];
            Arrays.fill(row_vector,0);

            int[][] result_matrix_00 = new int[split_index][split_index];
            int[][] result_matrix_01 = new int[split_index][split_index];
            int[][] result_matrix_10 = new int[split_index][split_index];
            int[][] result_matrix_11 = new int[split_index][split_index];
            initWithZeros(result_matrix_00,split_index,split_index);
            initWithZeros(result_matrix_01,split_index,split_index);
            initWithZeros(result_matrix_10,split_index,split_index);
```

```
        initWithZeros(result_matrix_11,split_index,split_index);

        int[][] a00 = new int[split_index][split_index];
        int[][] a01 = new int[split_index][split_index];
        int[][] a10 = new int[split_index][split_index];
        int[][] a11 = new int[split_index][split_index];
        int[][] b00 = new int[split_index][split_index];
        int[][] b01 = new int[split_index][split_index];
        int[][] b10 = new int[split_index][split_index];
        int[][] b11 = new int[split_index][split_index];

        initWithZeros(a00,split_index,split_index);
        initWithZeros(a01,split_index,split_index);
        initWithZeros(a10,split_index,split_index);
        initWithZeros(a11,split_index,split_index);
        initWithZeros(b00,split_index,split_index);
        initWithZeros(b01,split_index,split_index);
        initWithZeros(b10,split_index,split_index);
        initWithZeros(b11,split_index,split_index);


        for (int i = 0; i < split_index; i++){
            for (int j = 0; j < split_index; j++) {
                a00[i][j] = matrix_A[i][j];
                a01[i][j] = matrix_A[i][j + split_index];
                a10[i][j] = matrix_A[split_index + i][j];
                a11[i][j] = matrix_A[i + split_index][j + split_index];
                b00[i][j] = matrix_B[i][j];
                b01[i][j] = matrix_B[i][j + split_index];
                b10[i][j] = matrix_B[split_index + i][j];
                b11[i][j] = matrix_B[i + split_index][j + split_index];
            }
        }

        add_matrix(multiply_matrix(a00, b00),multiply_matrix(a01, b10),result_matrix_00,
split_index);
        add_matrix(multiply_matrix(a00, b01),multiply_matrix(a01, b11),result_matrix_01,
split_index);
        add_matrix(multiply_matrix(a10, b00),multiply_matrix(a11, b10),result_matrix_10,
split_index);
        add_matrix(multiply_matrix(a10, b01),multiply_matrix(a11, b11),result_matrix_11,
split_index);

        for (int i = 0; i < split_index; i++){
            for (int j = 0; j < split_index; j++) {
                result_matrix[i][j] = result_matrix_00[i][j];
                result_matrix[i][j + split_index] = result_matrix_01[i][j];
                result_matrix[split_index + i][j] = result_matrix_10[i][j];
                result_matrix[i + split_index] [j + split_index] = result_matrix_11[i][j];
            }
        }
    }
```

```
        return result_matrix;
    }

    public static void main (String[] args) {
        int[][] matrix_A = { { 1, 1, 1, 1 },
                { 2, 2, 2, 2 },
                { 3, 3, 3, 3 },
                { 2, 2, 2, 2 } };

        System.out.println("Array A =>");
        printMat(matrix_A,4,4);

        int[][] matrix_B = { { 1, 1, 1, 1 },
                { 2, 2, 2, 2 },
                { 3, 3, 3, 3 },
                { 2, 2, 2, 2 } };

        System.out.println("Array B =>");
        printMat(matrix_B,4,4);

        int[][] result_matrix =  multiply_matrix(matrix_A, matrix_B);

        System.out.println("Result Array =>");
        printMat(result_matrix,4,4);
    }
}
```

Output:

```
Array A =>
1 1 1 1
2 2 2 2
3 3 3 3
2 2 2 2

Array B =>
1 1 1 1
2 2 2 2
3 3 3 3
2 2 2 2

Result Array =>
8 8 8 8
16 16 16 16
24 24 24 24
16 16 16 16
```

Analysis of Algorithm

The program implements the standard matrix multiplication algorithm in Java.

The time complexity of the **standard algorithm** is $\mathbf{O(n^3)}$, where n is the size of the input matrices. The constant factor for this algorithm will depend on the specific implementation and hardware used.

For **Strassen's algorithm**, the time complexity is $\mathbf{O(n^{log2(7)})}$, which is approximately $\mathbf{O(n^{2.81})}$. The constant factor for this algorithm is typically larger than the standard algorithm due to the additional overhead of recursive calls and matrix additions/subtractions.

The crossover point at which Strassen's algorithm becomes more efficient than the standard algorithm depends on the constant factors for each algorithm and the size of the input matrices. Generally, Strassen's algorithm becomes more efficient for large matrices (n > 100-200), but the specific crossover point will depend on the implementation and hardware used.

In practice, a hybrid algorithm that uses Strassen's algorithm for large matrices and the standard algorithm for small matrices is often used to achieve better performance across a range of input sizes.

Practical 6: Greedy - Kruskal using Union Find

Aim

To **Understand** and **Implement** the Kruskal's Algorithm using Union Find Greedy Approach, analyse space and time complexity of it.

Algorithm

1. **Sort** the edges of the graph by weight in non-decreasing order.
2. **Initialize** an empty set of edges S.
3. **Initialize** an empty Union-Find data structure with V disjoint sets, where V is the number of vertices in the graph.
4. For each edge **e = (u, v)** in the sorted list of edges:
   a. **Find** the sets that u and v belong to using the **find()** operation of the Union-Find data structure.
   b. If the sets are different, **add** e to S and **merge** the sets using the **union()** operation.
   c. If the sets are the same, **skip** e to avoid creating a cycle.
5. **Return** S, which contains the edges of the minimum spanning tree.

Program

```
import java.util.*;

public class Kruskal {
    public static void main(String[] args) {
        Scanner sc = new Scanner(System.in);

        System.out.print("Enter the number of vertices: ");
        int V = sc.nextInt();
        System.out.print("Enter the number of edges: ");
        int E = sc.nextInt();

        Graph graph = new Graph(V, E);

        // Adding edges
        for (int i = 0; i < E; i++) {
            System.out.println("Enter the source, destination, and weight of edge " + (i +
1) + ":");
            graph.edges[i].src = sc.nextInt();
            graph.edges[i].dest = sc.nextInt();
            graph.edges[i].weight = sc.nextInt();
        }

        graph.kruskal();
    }
}

class Graph {
    static class Edge implements Comparable<Edge> {
        int src, dest, weight;
        public int compareTo(Edge other) {
            return weight - other.weight;
        }
    }

    int V, E;
    Edge[] edges;
```

```
    Graph(int v, int e) {
        V = v;
        E = e;
        edges = new Edge[E];
        for (int i = 0; i < E; ++ i)
            edges[i] = new Edge();
    }

    int find(int[] parent, int i) {
        if (parent[i] == -1)
            return i;
        return find(parent, parent[i]);
    }

    void union(int[] parent, int x, int y) {
        int xset = find(parent, x);
        int yset = find(parent, y);
        parent[xset] = yset;
    }

    void kruskal() {
        Edge[] result = new Edge[V];
        int e = 0;
        int i = 0;
        for (i = 0; i < V; ++ i)
            result[i] = new Edge();

        Arrays.sort(edges);

        int[] parent = new int[V];
        Arrays.fill(parent, -1);

        i = 0;
        while (e < V - 1) {
            Edge next_edge = edges[i ++];
            int x = find(parent, next_edge.src);
            int y = find(parent, next_edge.dest);

            if (x != y) {
                result[e ++] = next_edge;
                union(parent, x, y);
            }
        }

        int finalWeight = 0;
        System.out.println("Edges in the MST :: ");
        for (i = 0; i < e; ++ i) {
            System.out.println(result[i].src + " - " + result[i].dest + ": " +
result[i].weight);
            finalWeight = finalWeight + result[i].weight;
        }
        System.out.println("Total Weight of MST :: " + finalWeight);

    }
}
```

Output:

```
Enter the number of vertices: 6
Enter the number of edges: 8
Enter the source, destination, and weight of edge 1:
0 1 4
Enter the source, destination, and weight of edge 2:
0 2 4
Enter the source, destination, and weight of edge 3:
1 2 2
Enter the source, destination, and weight of edge 4:
2 3 3
Enter the source, destination, and weight of edge 5:
2 4 4
Enter the source, destination, and weight of edge 6:
2 5 2
Enter the source, destination, and weight of edge 7:
3 4 3
Enter the source, destination, and weight of edge 8:
4 5 3
Edges in the MST ::
1 - 2: 2
2 - 5: 2
2 - 3: 3
3 - 4: 3
0 - 1: 4
Total Weight of MST :: 14
```

Analysis of Algorithm

Time Complexity:

The time complexity of Kruskal's algorithm is ***O(E log E)***, where E is the number of edges in the graph. This is because the algorithm sorts the edges in the graph by weight, which takes O(E log E) time using an efficient sorting algorithm such as **quick sort** or **merge sort**.

After the edges are sorted, the algorithm iterates through them in increasing order of weight and performs a union-find operation to determine whether adding the edge to the MST would create a cycle. The **union-find operation takes O(log V) time**, where V is the number of vertices in the graph.

Since Kruskal's algorithm performs the union-find operation at most E times, the total time complexity of the algorithm is **O(E log E + E log V)**, which can be simplified to **O(E log E)** since E >= V-1 in a connected graph.

Space Complexity:

The space complexity of the algorithm is O(V) to store the parent array in the union-find data structure.

Practical 7: Floyd Warshall – Dynamic Programming

Aim

To Implement the Floyd-Warshall Algorithm for All Pair Shortest Path Problem using Dynamic Programming.

Algorithm

1. Initialize distance and pred matrices with infinity and null values respectively, except for the diagonal elements which are initialized to 0.
2. For each vertex k from 1 to n, do the following:
   a. For each pair of vertices i and j from 1 to n, check if the path from i to k and then from k to j is shorter than the current path from i to j. If it is, update the distance and pred matrices accordingly.
3. Return the distance and pred matrices.

Program

```
public class FloydWarshall {
    static int INF = 9999;
    public static void main(String[] args) {
        int graph[][] = {
                {0, 5, INF, 10},
                {INF, 0, 3, INF},
                {INF, INF, 0, 1},
                {INF, INF, INF, 0}
        };
        floydWarshall(graph);
    }
    static void floydWarshall(int[][] graph) {
        int V = graph.length;
        int[][] matrix = new int[V][V];
        for(int i = 0; i < V; i ++) {
            for(int j = 0; j < V; j ++) {
                matrix[i][j] = graph[i][j];
            }
        }
        for(int i = 0; i < V; i ++) {
            for(int j = 0; j < V; j ++) {
                for(int k = 0; k < V; k ++) {
                    if(matrix[j][k] > matrix[i][k] + matrix[j][i]) {
                        matrix[j][k] = matrix[i][k] + matrix[j][i];
                    }
                }
            }
        }
        printMatrix(matrix);
    }
    static void printMatrix(int[][] matrix) {
        System.out.println("Resultant Matrix using Floyd Warshall is: ");
        int V = matrix.length;
        for(int i = 0; i < V; i ++) {
            System.out.print("[");
            for(int j = 0; j < V; j ++) {
                if(matrix[i][j] == INF) {
                    System.out.print("INF ");
                }
                else {
                    System.out.print(matrix[i][j] + " ");
```

```
                }
            }
            System.out.print("]");
            System.out.println();
        }
    }
}
```

Output:

```
Resultant Matrix using Floyd Warshall is:
[0 5 8 9 ]
[INF 0 3 4 ]
[INF INF 0 1 ]
[INF INF INF 0 ]
```

Analysis of Algorithm

Time Complexity:

The time complexity of the Floyd-Warshall algorithm is **$O(V^3)$**, where V is the number of vertices in the graph. The algorithm iteratively considers all possible intermediate vertices in the shortest path calculation, resulting in a **nested triple loop** that performs V^3 operations. This time complexity is independent of the specific graph structure or edge weights, making it a suitable algorithm for a wide range of graph problems.

Space Complexity:

The space complexity of the Floyd-Warshall algorithm is **$O(V^2)$**, where V is the number of vertices in the graph. This is because the algorithm requires a matrix of size V x V to store the shortest path distances between all pairs of vertices.

Practical 8: Backtracking - N Queens

Aim

Solve the $\boldsymbol{n}$ queens' problem using backtracking. Here, the task is to place $\boldsymbol{n}$ chess queens on an n x n board so that no two queens attack each other.

Algorithm

1. Initialize an empty chessboard of size N x N.
2. Start with the leftmost column and place a queen in the first row of that column.
3. Move to the next column and place a queen in the first row of that column.
4. Repeat step 3 until either all N queens have been placed or it is impossible to place a queen in the current column without violating the rules of the problem.
5. If all N queens have been placed, print the solution.
6. If it is not possible to place a queen in the current column without violating the rules of the problem, backtrack to the previous column.
7. Remove the queen from the previous column and move it down one row.
8. Repeat steps 4-7 until all possible configurations have been tried.

Program

```
import java.util.Scanner;

public class NQueenProblem {
    static int N;

    // print the final solution matrix
    static void printSolution(int board[][]) {
        for (int i = 0; i < N; i++) {
            for (int j = 0; j < N; j++) {
                System.out.print(" " + board[i][j] + " ");
            }
            System.out.println();
        }
    }

    // function to check whether the position is safe or not
    static boolean isSafe(int board[][], int row, int col) {
        int i, j;
        // Check for Same Row
        for (i = col - 1; i >= 0; i--) {
            if (board[row][i] == 1) {
                return false;
            }
        }
        // Check for Upper Diagonal
        for (i = row, j = col; i >= 0 && j >= 0; i--, j--) {
            if (board[i][j] == 1) {
                return false;
            }
        }
        // Check for Lower Diagonal
        for (i = row, j = col; j >= 0 && i < N; i++, j--) {
            if (board[i][j] == 1) {
                return false;
            }
        }
```

```
        return true;
    }

    static boolean solveNQueen(int board[][], int col) {
        if (col >= N) {
            return true;
        }
        for (int i = 0; i < N; i++) {
            if (isSafe(board, i, col)) {
                board[i][col] = 1;

                if (solveNQueen(board, col + 1)) {
                    return true;
                }
                // Backtrack if the above condition is false
                board[i][col] = 0;
            }
        }
        return false;
    }

    public static void main(String args[]) {
        Scanner sc = new Scanner(System.in);
        System.out.print("Enter size of board: ");
        N = sc.nextInt();
        sc.close();

        int[][] board = new int[N][N];
        for (int i = 0; i < N; i++) {
            for (int j = 0; j < N; j++) {
                board[i][j] = 0;
            }
        }

        if (!solveNQueen(board, 0)) {
            System.out.print("Solution does not exist");
            return;
        }

        printSolution(board);

    }
}
```

Output:

```
Enter size of board: 5
 1  0  0  0  0
 0  0  0  1  0
 0  1  0  0  0
 0  0  0  0  1
 0  0  1  0  0
```

Analysis of Algorithm

Time Complexity:

The time complexity of this code is **O(N!)** because for each column, we check all possible rows to place the queen, which means we have N choices in the first column, N-2 choices in the second column, N-4 choices in the third column, and so on. Therefore, the total number of possibilities will be **N * (N-2) * (N-4) * ... * 1**, which is equivalent to N!.

Space Complexity:

The space complexity of this code is **$O(N^2)$** because we are using a **two-dimensional array of size N x N** to represent the chessboard. Additionally, we are using recursive function calls that will occupy memory on the stack, and the maximum depth of the recursion tree will be N. Therefore, the space complexity will be $O(N^2 + N)$, which is equivalent to **$O(N^2)$**.