Data Structure Project Report

Name: Lu Jingyu

UCD: 20205767

BJUT: 20372216

Class:SE-2

Project 1-A

Selection Sort

Theory:

Select the largest element from the array to be sorted, store it at the end of the sequence, then place it at the end of the sequence, and repeat the process until you have the smallest element in the entire array. So, the whole array is sorted from smallest to largest

Code implementation

```
public static void getSort(int[] arr){
    int i=0;
    int k=0;
    int j=0;
    while (i< arr.length-1) {
        j = 0;
        k=0;
        while (k < arr.length-1-i) {
            if (arr[j] < arr[k+1]) {
                j = k + 1;
            }k++;
        }
        if (arr.length != j) {
            int x = arr[arr.length - 1-i];
            arr[arr.length - 1-i] = arr[j];
            arr[j] = x;
        }
        i++;
        }
    for (int n=0;n<arr.length;n++) {
        System.out.println(arr[n]);
    }
}
```

Code analysis

```
        while (i< arr.length-1) {
            j = 0;
            k=0;
```

The outermost loop is used to iterate over the array.

```
    while (k < arr.length-1-i) {
        if (arr[j] < arr[k+1]) {
            j = k + 1;
        }k++;
    }
```

The first minor loop under the major loop is used to find the largest element in the list and record its location, and fix the largest number at the end of the array

```
    if (arr.length != j) {
        int x = arr[arr.length - 1-i];
        arr[arr.length - 1-i] = arr[j];
        arr[j] = x;
    }
```

This if statement is used to place the largest number at the end of an unsorted list.

```
        int x = arr[arr.length - 1-i];
```

Stores elements that need to be swapped

```
        arr[arr.length - 1-i] = arr[j];
```

Equals the largest element of the current array to the last element of the sorted array

```
        arr[j] = x;
```

Because the current list has the largest identical element in both arrays.

So I need to assign the previously stored element that needs to be swapped to the position of the largest element before the swap.

```
        for (int n=0;n<arr.length;n++) {
            System.out.println(arr[n]);
        }
```

Use this for loop to check that the sorted array elements are in the correct order.

Selection Sort_2D

Code implementation

```
    public static void getSort_2D(int[][] arr){
        int i=0;
        int k=0;
        int j=0;
        while (i< arr.length-1) {
            j = 0;
            k=0;
            while (k < arr.length-1-i) {
                if (arr[j][0] < arr[k+1][0]) {
                    j = k + 1;
                }k++;
            }
            if (arr.length != j) {
                int[] x = arr[arr.length - 1-i];
                arr[arr.length - 1-i]= arr[j];
                arr[j] = x;
            }
            i++;
        }
```

Code analysis

Sorting a 2D array is basically the same as sorting a 1D array

It's just a little bit different when it's moving

```
            while (k < arr.length-1-i) {
                if (arr[j][0] < arr[k+1][0]) {
                    j = k + 1;
                }k++;
            }
```

Because when sorting a 2D array, I only need to compare the first element of each array.

```
            if (arr.length != j) {
                int[] x = arr[arr.length - 1-i];
                arr[arr.length - 1-i]= arr[j];
                arr[j] = x;
            }
            i++;
        }
```

Then sort the entire array by the size of the first element of each array. So, I need to move each array to the correct position as the first element moves.

```
    for (int n=0;n<arr.length;n++) {
        System.out.println(Arrays.toString(arr[n]));
    }
```

Use this for loop to verify that sorted arrays are in the correct order.

Insertion Sort

Theory:

Build an ordered sequence. For unordered data, scan from back to front in the sorted sequence, find the corresponding position and insert until all elements are inserted.

Code implementation:

```java
public static void getSort(int[] arr) {
    int i=1;
    int compareNum;
    while (i < arr.length) {
        compareNum=arr[i];
        int j=i;
        while (j>0){
            if (compareNum<arr[j-1]){
                arr[j]=arr[j-1];
            }else{
                break;
            }
            j--;
        }
        arr[j]=compareNum;
        i++;
    }
```

Code analysis:

```java
    int i=1;
    int compareNum;
    while (i < arr.length) {
```

Because I need to compare the element to the one before it and insert it, I iterate from the second element in the list.

```java
        compareNum=arr[i];
        int j=i;
```

Store the elements I want to compare

Assign the number of the index to another variable

```java
        while (j>0){
            if (compareNum<arr[j-1]){
                arr[j]=arr[j-1];
```

If the selected element is smaller than the element before it, it is equal to the element

before it, and then it moves forward, comparing pairs.

```
            }else{
                break;
            }
```

If not, jump out of the small loop into the large loop and continue iterating backwards through the entire array.

```
            arr[j]=compareNum;
            i++;
        }
```

At the end of the small loop, insert the stored number into the position where it needs to be inserted, that is, the position where there is no smaller element in front of the element

```
    for (int n=0;n<arr.length;n++) {
        System.out.println(arr[n]);
    }
```

Use this for loop to check that the sorted array elements are in the correct order. (But this code will not appear in the finished project)

Insertion Sort_2D

Code implementation:

Sorting a 2D array is basically the same as sorting a 1D array.

It's just a little bit different when it's moving.

```java
public static void getSort_2D(int[][] arr) {
    int i=1;
    int compareNum;
    int[] tempArr1;
    int[] tempArr2;
    while (i < arr.length) {
        compareNum=arr[i][0];
        tempArr1=arr[i];
        tempArr2=arr[i];
        int j=i;
        while (j>0){
            if (compareNum<arr[j-1][0]){
                arr[j]=arr[j-1];
                arr[j]=arr[j-1];
            }else{
                break;
            }
            j--;
        }
        arr[j]=tempArr1;
        arr[j]=tempArr2;
        i++;
    }
```

Code analysis:

```java
        while (j>0){
            if (compareNum<arr[j-1][0]){
```

Because when sorting a 2D array, I only need to compare the first element of each array.

```java
          arr[j]=arr[j-1];
          arr[j]=arr[j-1];
```

Then sort the entire array by the size of the first element of each array. So, I need to move each array to the correct position as the first element moves.

```
for (int n=0;n<arr.length;n++) {
    System.out.println(Arrays.toString(arr[n]));
}
```

Use this for loop to check that sorted arrays are in the correct order. (But this code will not appear in the finished project)

Challenges:

1. When I first wrote insertion sort, The idea was to compare the selected element to its predecessor and then immediately transpose it, so I did the comparison and

```
int temp;
int i=1
while (i< arr.length){
    if (arr[i]<arr[i-1]){
        temp=arr[i];
        arr[i]=arr[i-1];
        arr[i-1]=temp;
    }
    i++;
}
```

   sorting in one loop, The following code:

   However, in the subsequent detection, I found that the time complexity of such writing is inconsistent with that of insertion sort, so I needed to completely overturn my idea and start again.

2. During the loop, two cycles are required, one for traversing the arrays from front to back, and the other for comparing the currently selected element with its previous element. Therefore, it is a great challenge for me to carry out the two loops at the same time and in the opposite direction, but I later thought of assigning the same index to different cycle counters, and change the conditions of while loop, e.g. i < 10, i++ and j > 0, j--

Project 1-B

Quick Sort

Theory:

The first thing I need to do is find a number to use as a pivot number. So what I need to do is move the elements that are smaller than the base number to the left of the sequence, and move the elements that are larger than the base number to the right of the sequence. Then, the elements of the two partitions continue to find the pivot number of each partition according to the above two methods, and then move until there is only one number in each partition.

Code implementation:

```
public static void QuickSort(int arr[], int lo, int hi) {
    int pivotNum = arr[lo];
    int i = lo;
    int j = hi;
    int num;
    while (i < j) {
        while (arr[j] >= pivotNum && i < j) {
            j--;
        }
        while (arr[i] <= pivotNum && i < j) {
            i++;
        }
        if (i < j) {
            num = arr[j];
            arr[j] = arr[i];
            arr[i] = num;
        }
    }
    arr[lo] = arr[i];
    arr[i] = pivotNum;
    if(i>lo) {
        QuickSort(arr, lo, hi: j - 1);
    }
    if(j<hi) {
        QuickSort(arr, lo: j + 1, hi);
    }
```

Code analysis:

```
public static void QuickSort(int arr[], int lo, int hi) {
```

This method takes an array with the left and right edges as parameters.

```
    int pivotNum=arr[lo];
```

Select the leftmost number of the array as the pivot number.

```
    int i = lo;
    int j = hi;
    int num;
    while (i < j) {
```

All loops are performed under the condition that the left edge of the array is smaller than the right edge, preventing an infinite loop.

```
        while (arr[j] >= pivotNum && i < j) {
            j--;
        }
```

This loop is used to find elements that are smaller than the base value from back to front. (The reason why I must search from right to left is explained in the challenge)

```
        while (arr[i] <= pivotNum && i < j) {
            i++;
        }
```

This loop is used to find elements that are larger than the base value from back to front.

```
        if (i < j) {
            num = arr[j];
            arr[j] = arr[i];
            arr[i] = num;
        }
```

If all of the above conditions are met, the positions of the two elements are swapped in the same way as in selection sort.

If the left edge of the array is less than the right edge of the array, repeat the preceding steps

If the left edge of the array is equal to the right edge of the array, then the right and left

scans are the same number.

```
        arr[lo] = arr[i];
        arr[i] = pivotNum;
```

After that, the first number in the array, the pivot number, is swapped with the number that coincides with the left and right scans.

The elements to the left of the base value of the array are all less than the base value, and the elements to the right of the array are all greater than the base value, but the elements around the pivot value are not sorted, so it needs to recurse and perform the next quick sort.

```
        if(i>lo) {
            QuickSort(arr, lo,  hi: j - 1);
        }
        if(j<hi) {
            QuickSort(arr,  lo: j + 1, hi);
        }
```

The if statement is inserted before each recursion to create an infinite loop, it prevents us from recursing infinitely if the value of i is less than the left edge of the array and the value of j is greater than the right edge of the array.

```
        if(i>lo) {
            QuickSort(arr, lo,  hi: j - 1);
        }
```

This recursion quicksorts the array to the left of the reference value until each array partition contains only one element.

```
        if(j<hi) {
            QuickSort(arr,  lo: j + 1, hi);
        }
```

This recursion quicksorts the array to the right of the reference value until each array partition contains only one element.

```
    for (int n = 0; n < arr.length; n++) {
        System.out.println(arr[n]);
    }
```

This code is used to check if the output is correct. (But this code will not appear in the finished project)

Quick Sort 2D

```
int[][] arr_2D={{98,34,100,36,44,64,3,99,59},{20 ,88 ,55 ,91 ,14 ,58 ,25 ,29 ,44},{66 ,62 ,4 ,65 ,49 ,71 ,71 ,24, 12},
        {14 ,3 ,58 ,23 ,12 ,66 ,11 ,45, 36 },{55 ,64 ,35 ,24 ,85 ,73 ,33 ,85 ,46},{94 ,76 ,23 ,36 ,57, 26, 8 ,92 ,17},
        {85 ,68 ,52 ,34, 53, 93 ,4, 37, 34},{70 ,9 ,15, 42, 31, 16, 72, 61, 62 },{11, 38, 34, 21, 81, 9, 45, 68, 11},
        {20 ,83 ,27, 6 ,69 ,26 ,5 ,31 ,8 },{74 ,97 ,11 ,60, 1 ,68 ,14 ,27 ,46}};
for (int i = 0; i < arr_2D.length; i++) {
    int l=0;
    int h=arr_2D[0].length-1;
    QuickSort(arr_2D[i],l,h);
```

The 2D array requires only a single sort for each contained array, so I just need to extract each contained array with a for loop and sort it.

```
        for (int i = 0; i < arr_2D.length; i++) {
            int l=0;
            int h=arr_2D[0].length-1;
            QuickSort(arr_2D[i],l,h);
            System.out.println(Arrays.toString(arr_2D[i]));
        }
```

This code is used to check if the output is correct. (But this code will not appear in the finished project).

Challenge:

1. When I first wrote this code, I scanned the array from the left, but that can cause a problem

   e.g.

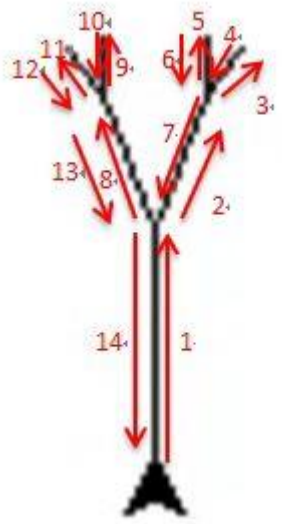
   ```
   public class test {
       int[] testArr={7,2,3,8,10};
       while (testArr[i]<=pivotNum){
           i++;
       }
       while (testArr[i]<=pivotNum){
           j--;
       }
   }
   ```

   When i do quicksort this way, with 7 as the pivot value, i and j will stop at 8, and then swap the pivot value and the scan stop value, and the array will become {8,2,3,7,10}, Not all the values to the left of the base value are less than the base value, so I need to scan from the right.

2. In the final recursion, I didn't put in an if statement at the beginning to limit the

range of i and j, this will eventually lead to errors: StackOverflowError, finally, by using system.out.println, I find that the values of i and j exceed the array bounds.

3. Double recursion was a big challenge for me, and I did a lot of work to understand the order in which double recursion was executed, here is a picture to make it easier to understand.



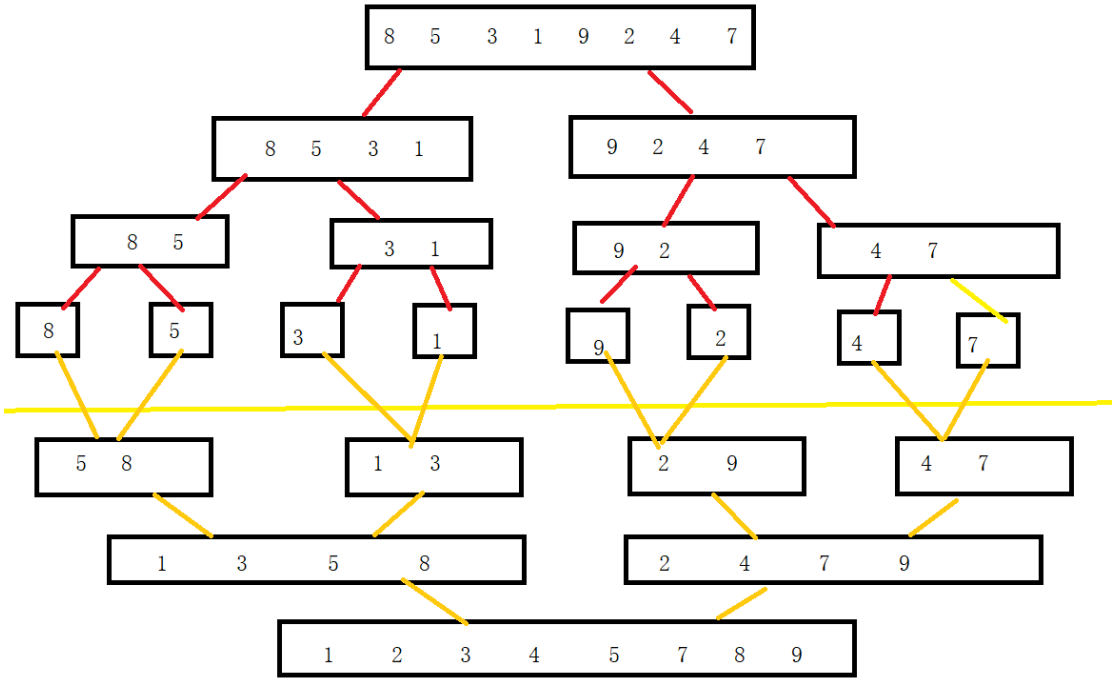
There are two recursions, one in which the branch grows to the left and one in which the branch grows to the right, and the number is the order in which the recursion is executed

Project 1-D

Merge Sort

Theory:

Order each subsequence first, and then order the segments of the subsequence. Merge the ordered subsequences to obtain a fully ordered sequence, and merge the two ordered tables into one ordered table



Code implementation:

```
public static int[] mergeSort(int[] arr,int l,int r){
    int mid=(l+r)/2;
    if (l<r){
        mergeSort(arr,l,mid);
        mergeSort(arr, mid+1,r);
        merge(arr,l,mid,r);
    }
    return arr;
}
public static void merge(int[] arr,int l,int m,int r){
    int[] tempArr=new int[r-l+1];
    int i=l;
    int j=m+1;
    int k=0;
    while (i<=m&&j<=r){
        if (arr[i]<=arr[j]){
            tempArr[k]=arr[i];
            k++;
            i++;
        }else{
            tempArr[k]=arr[j];
            k++;
            j++;
        }
    }
```

```
        while (i<=m){
            tempArr[k]=arr[i];
            k++;
            i++;
        }
        while (j<=r){
            tempArr[k]=arr[j];
            k++;
            j++;
        }
        for (int n=0;n<tempArr.length;n++) {
            arr[n+l]=tempArr[n];
        }
        }

}
```

Code analysis:

```
    public static int[] mergeSort(int[] arr,int l,int r){
```

This method takes an array with the left and right edges as parameters.

```
    public static int[] mergeSort(int[] arr,int l,int r){
        int mid=(l+r)/2;
        if (l<r){
            mergeSort(arr,l,mid);
            mergeSort(arr, mid+1,r);
            merge(arr,l,mid,r);
        }
        return arr;
    }
```

This method is used to split an array in half and recursively divide it into individual elements, then call and recursive merge method to sort and merge.

```
        int mid=(l+r)/2;
        if (l<r){
            mergeSort(arr,l,mid);
            mergeSort(arr, mid+1,r);
```

Recursively bisecting the array, and then bisecting the array again until it divides into individual elements.

```
public static void merge(int[] arr,int l,int m,int r){
```

This method is used to sort and merge arrays, accepts an array with its left, right, and middle values as parameters.

```
int[] tempArr=new int[r-l+1];
```

Creates an array to hold sorted elements of the same length as the input array.

```
int i=l;
int j=m+1;
```

First half and second half of the loop counter.

```
while (i<=m&&j<=r){
    if (arr[i]<=arr[j]){
        tempArr[k]=arr[i];
        k++;
        i++;
```

Compare the first element of the two separate arrays, place the smaller element in the temporary array, and traverse both backwards, if arr[i] is small, put it in, and go backwards.

```
    }else{
        tempArr[k]=arr[j];
        k++;
        j++;
    }
```

Else, add arr[j], and go backwards.

```
while (i<=m){
    tempArr[k]=arr[i];
    k++;
    i++;
}
```

Put the remaining elements of the left array into the temporary array.

```
while (j<=r){
    tempArr[k]=arr[j];
    k++;
    j++;
}
```

Put the remaining elements of the right array into the temporary array.

```
for (int n=0;n<tempArr.length;n++) {
    arr[n+l]=tempArr[n];
}
```

Copies the elements of the temporary array into the original array.

```
merge(arr,l,mid,r);
```

And then keep recursing.

```
int[] arr = {31, 33, 27, 15, 42, 11, 40, 5, 19, 21};
int l=0;
int r=arr.length-1;
mergeSort(arr,l,r);
System.out.println(Arrays.toString(arr));
```

This code is used to check if the output is correct. (But this code will not appear in the finished project)

MergeSort_2D

Same idea as quicksort

Challenge:

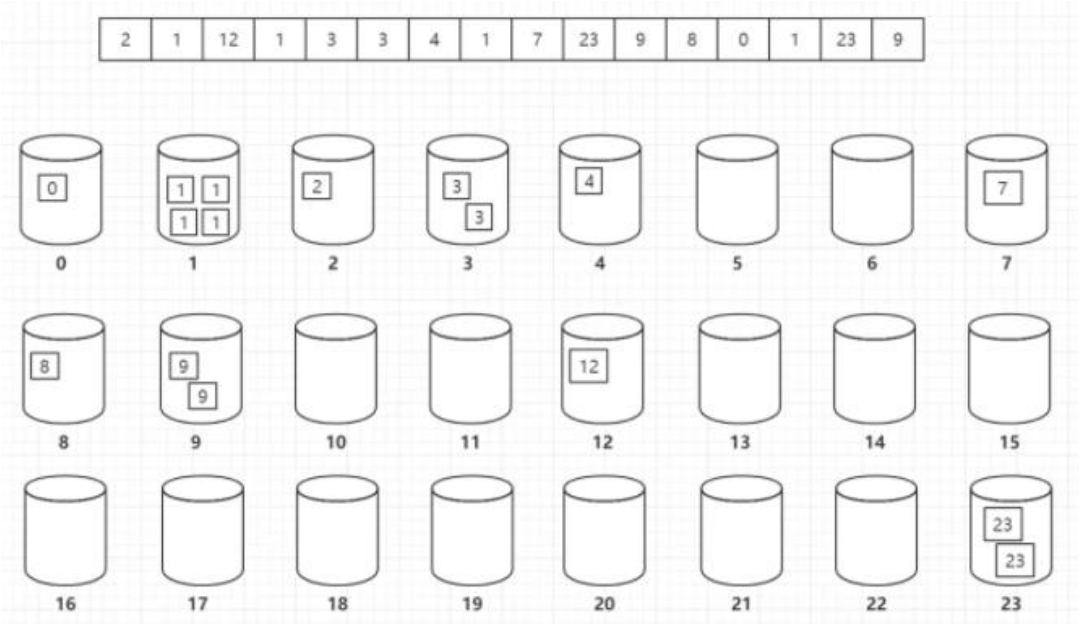
1. When I started writing the program, I wanted to take a step by step approach, splitting the array into individual elements with one method, sorting the array, and consolidating the array, but that made merging the elements step by step very difficult and required three methods
2. When sorting, I started with the idea of sorting separate arrays individually, that is, comparing the elements in an array with each other and then merging them, rather than comparing two separate and different array elements. As a result, the elements of the array are relatively sorted after being merged, e.g. {1,3,4,5,2,6,7}, rather than being sorted from smallest to largest

Project 1-D

Bucket Sort

Theory:

Divide the array into a finite number of buckets. Sort each bucket individually (it is possible to use another sort algorithm or continue to use bucket sort recursively).



Code implementation:

```
Integer[] result=new Integer[arrBucket[0].length];
int x=0;
int y=0;
int z=0;
for (int n=0;n<arrBucket[0].length;n++) {
    int k=1;
    int temp;
    while (k < arrBucket[0].length&&arrBucket[n][k]!=null) {
        temp=arrBucket[n][k];
        int m=k;
        while (m>0){
            if (temp<arrBucket[n][m-1]){
                arrBucket[n][m]=arrBucket[n][m-1];
            }else{
                break;
            }
            m--;
        }
        arrBucket[n][m]=temp;
        k++;
```

```
        Integer[] result=new Integer[arrBucket[0].length];
        int x=0;
        int y=0;
        int z=0;
        for (int n=0;n<arrBucket[0].length;n++) {
            int k=1;
            int temp;
            while (k < arrBucket[0].length&&arrBucket[n][k]!=null) {
                temp=arrBucket[n][k];
                int m=k;
                while (m>0){
                    if (temp<arrBucket[n][m-1]){
                        arrBucket[n][m]=arrBucket[n][m-1];
                    }else{
                        break;
                    }
                    m--;
                }
                arrBucket[n][m]=temp;
                k++;
            }
        }
        while (x<arrBucket.length&&arrBucket[0][0]!=null){
            y=0;
            z=0;
            while (y<arrBucket[0].length&&z<arrBucket[0].length) {
                if (result[y] == null) {
                    result[y] = arrBucket[x][z];
                    y++;
                    z++;
                } else {
                    y++;
                }

            }
            x++;
        }
        System.out.println(Arrays.toString(result));

    }
```

Code analysis:

```
    public static void getSort(int[] arr) {
        int countNum=11;
        int i=0;
        int j=0;
```

Because I know that the elements I want to sort are 0 to 100, so I need 11 buckets, 0 to 9,10 to 19, and so on, up to 100 to 119.

```
        Integer[][] arrBucket = new Integer[countNum][arr.length];
```

Use a two-dimensional array to create 11 arrays of the same length as the input array. (The reason for using the Integer type is mentioned in the challenge)

```
            int section=arr[i]/10;
```

Since there are only 11 buckets, I need to divide each element by 10 to determine

which bucket it belongs to.

```
while (i<arr.length){
    int section=arr[i]/10;
    if (section<1){
        section=0;
    }
```

Put everything less than 1 divided by 10 into the first bucket.

```
while (j<arr.length){
    if (arrBucket[section][j]==null){
        arrBucket[section][j]=arr[i];
        break;
    }else{
        j++;
    }
}
```

Put the element into the bucket. To prevent the element from being knocked out, the if statement is required to determine whether there are already elements in the current position of the array. Pick out the loop after you've found the location and put the element in, otherwise you keep going back to find the free location.

```
int x=0;
int y=0;
int z=0;
for (int n=0;n<arrBucket[0].length;n++) {
    int k=1;
    int temp;
    while (k < arrBucket[0].length&&arrBucket[n][k]!=null) {
        temp=arrBucket[n][k];
        int m=k;
        while (m>0){
            if (temp<arrBucket[n][m-1]){
                arrBucket[n][m]=arrBucket[n][m-1];
            }else{
                break;
            }
            m--;
        }
        arrBucket[n][m]=temp;
        k++;
    }
}
```

The elements in each bucket are sorted by insertion sorting. Because the bucket

storing elements is a two-dimensional array, the two-dimensional usage of insertion sorting is used.

```
Integer[] result=new Integer[arrBucket[0].length];
```

```
while (x<arrBucket.length&&arrBucket[0][0]!=null){
    y=0;
    z=0;
    while (y<arrBucket[0].length&&z<arrBucket[0].length) {
        if (result[y] == null) {
            result[y] = arrBucket[x][z];
            y++;
            z++;
        } else {
            y++;
        }

    }
    x++;
}
```

This code is used to place the sorted elements in each bucket into the same array.

Bucket Sort_2D

Same idea as quicksort

Challenge:

1. Because I am not allowed to use any standard libraries to complete this task, so ArrayLists are unusable, I need to set the length of each array to the length of the input array, which requires us to apply the Integer type so that all elements in the array are null at the beginning, which also makes it easy to locate and replace elements later.
2. Initially, I was going to create 11 arrays manually, but because that was too cumbersome, when I optimized the code, I came up with the idea of storing 11 arrays in a two-dimensional array.