PSS 3 Solutions

Sorting Problems

0) a) array with n equal elements [a a a a … a]

	Worst Case	Best Case
Quicksort	O(n)	O(n)
In-Place Quick-Sort	O(nlogn)	O(nlogn)
Insertion sort	O(n)	O(n)
Merge sort	O(nlogn)	O(nlogn)
Bubble sort	O(n)	O(n)

Quicksort
Pivot is a.
int[] temp is created
for loop runs n times without entering the if/else statements
int[] L = Arrays.CopyOfRange(temp,0,m) where m is still 0
Hence L contains only 1 element at index 0
int[] E = new int[k-m] m is still 0, k is equal to n
Arrays.fill(E,pivot) the whole array is n elements of pivot
int[] G = Arrays.copyOfRange(temp,k,n) where k is equal to n
Hence G contains only 1 element from n index
quickSort(L) call enters the base case immediately
quickSort(G) call enters the base case immediately
System.arrayCopy(L,0,S,0,m) m is still 0
System.arrayCopy(E,0,S,m,k-m) m is still 0, k is equal to n
System.arrayCopy(G,0,S,k,n-k) n-k is equal to 0
Time complexity: O(n)
Space complexity: O(n)

In-Place-QuickSort
Pivot is a.
While loop entered left is <= right
inner while loop entered:
    condition: left<pivot is not satisfied, stop
second inner while loop entered:
    condition: right>pivot is not satisfied, stop
if statement (left<=right) satisfied
    swaps two elements
// we do n/2 swaps in the outer while loop
Swaps the leftmost greater element with the last element - pivot
calls quickSortInplace(S,a,left-1)
calls quickSortInplace(S,left+1,b)
these calls contain all the elements together without the pivot
Time complexity: O(nlogn)
Space complexity: O(logn) which is the depth of recursion

Insertion Sort
for loop runs n−1 times
inner while loop condition is never satisfied, so the program does not enter it
Time complexity: O(n)
Space complexity: O(1)
Selection Sort
We make a call with array [a a a a … a]
for loop runs n−1 times
inner for loop runs n−1 times
if statement in the inner for loop is never satisfied, however both of the for loops run n−1 times
Time complexity: O(n^2)
Space complexity: O(1)

Bubble Sort
for loop entered
boolean swapped = false;
the inner for loop runs n−1 times
in the inner for loop if statement is never satisfied, so the boolean swapped stays false which breaks the for loop
Time complexity: O(n)
Space complexity: O(1)

b) sorted array in increasing order

	Worst Case	Best Case
Quicksort	$O(n^2)$	$O(n^2)$
In-Place Quick-Sort	$O(n^2)$	$O(n^2)$
Insertion sort	O(n)	O(n)
Merge sort	O(nlogn)	O(nlogn)
Bubble sort	O(n)	O(n)

QuickSort
The algorithm always reduces the array by 1 element only, as the L array stays empty, and G array contains n-1 elements after iteration.
We do partitioning (dividing into groups of less, equal, greater)
We run the algorithm n-1 times and do corresponding number of operations each time:
(n-1) + (n-2) + (n-) + … + 2 + 1
Time complexity: $O(n^2)$
Space complexity: O(n)

In-Place-QuickSort
Pivot is the last (greatest element)
Each time the call decreases the number of elements by 1 only
so we call recursively n-1 times performing n-1 operations
Time complexity: $O(n^2)$
Space complexity: O(n) (depth of recursion)

Insertion Sort
for loop runs n-1 times
the condition of the while loop is never satisfied
Time complexity: O(n)
Space complexity: O(1)

Selection Sort
outer for loop runs n-1 times
inner for loop runs n-1 times
If condition is never satisfied but the loop run anyway
Time complexity: $O(n^2)$
Space complexity: O(1)

Bubble Sort
outer for loop entered
Inner for loop runs n-1 times
boolean swapped stays false which breaks the outer loop
Time complexity: O(n)
Space complexity: O(1)

c) array reverse-ordered: sorted in decreasing order

	Worst Case	Best Case
Quicksort	$O(n^2)$	$O(n^2)$
In-Place Quick-Sort	$O(n^2)$	$O(n^2)$
Insertion sort	$O(n^2)$	$O(n^2)$
Merge sort	O(nlogn)	O(nlogn)
Bubble sort	$O(n^2)$	$O(n^2)$

QuickSort
With each iteration we decrease the size of the array by 1
performing for loop iteration each time
(n-1) + (n-2) + (n-3) … + 2 + 1 operations
Time complexity: $O(n^2)$
Space complexity: O(n)

In-place QuickSort
With each iteration we decrease the size of the array by 1
Performing for loop iteration each time
(n-1) + (n-2) + (n-3) … + 2 + 1 operations
Time complexity: $O(n^2)$
Space complexity: O(n)

Insertion Sort
Time complexity: $O(n^2)$

Merge Sort
Time complexity: O(nlogn)

Selection Sort
Time complexity: $O(n^2)$

d) sequence a, b, c, a, b, c… where a < b < c,in total 3n elements

	Worst Case	Best Case
Quicksort	O(n)	O(n)
In-Place Quick-Sort	Skip	Skip
Insertion sort	$O(n^2)$	$O(n^2)$
Merge sort	O(nlogn)	O(nlogn)
Bubble sort	$O(n^2)$	$O(n^2)$

1) QuickSort
Pivot is c.
The algorithm makes 3n−1 comparisons and partitions the array:
n pieces of c go to the array E, 2n pieces of a,b go to array G
Next we have a call quickSort(L): pivot now is b
The algorithm makes 2n−1 comparisons and partitions the array:
N pieces of b go to the array E, n pieces of a go to array L
Next, we have a call quickSort(L): pivot is a
The algorithm makes n−1 comparisons and partitions the array:
All the elements go to the array E
The algorithm merges all the partitions and returns
Total job done is: (3n−1) + (2n−1) + (n−1) which is O(n)

2) Solution:
Best case: c,b,a (the most balanced partitioning we can get)
Worst case: sorted sequence

3) Solution:
Guaranteed: O(nlogn) worst-case

4) Solution:
Guaranteed: O(nlogn) worst-case

5) Solution: Selection sort
Selection sort makes O(n) swaps which is the minimum among all sorting algorithms mentioned above.

6) a) 12, 13, 16, 18, 21, 25, 56, 123
b) 13, 31, 35, 42, 78

7) Answer: Bucket Sort: a set with fixed sized range

Coding

```
import java.util.ArrayList;
public class Union {
   public static ArrayList<Integer> computeUnion(int[] A, int[] B)
{
        int n = A.length + B.length;
        ArrayList<Integer> union = new ArrayList<>();
        int i = 0, j = 0, k = 0;
        while (i < A.length && j < B.length) {
            if (A[i] < B[j]) {
                union.add(A[i++]);
            } else if (A[i] > B[j]) {
                union.add(B[j++]);
            } else {
                union.add(A[i++]);
                j++;
            }
        }
        return union;
    }
    public static void main(String[] args) {
        int[] A = {1, 2, 4, 5, 6};
        int[] B = {2, 3, 5, 7};
        ArrayList<Integer> union = computeUnion(A, B);
        for (int i = 0; i < union.size(); i++) {
            System.out.print(union.get(i) + " ");
        }
    }
}
```

1) Bucket Sort

```
public static void bucketSort (int[] arr, int exp) {
        ArrayList<Integer>[] tempArray = (ArrayList<Integer>[])
new ArrayList[10];

        for(int i = 0; i < tempArray.length; i++) {
            tempArray[i] = new ArrayList<Integer>();
        }
        for(int i = 0; i < arr.length; i++) {
            tempArray[(arr[i]/exp)%10].add(arr[i]);
        }
        int nextIndex = 0;
        for(int i = 0; i < 10; i++) {
            for(int j = 0; j < tempArray[i].size(); j++) {
                arr[nextIndex++] = tempArray[i].get(j);
            }
        }
}
```

2) Radix Sort

```
public static void radixSort(int[] arr) {
        int max = arr[0];
        for(int i = 0; i < arr.length; i++) {
            if(arr[i] > max) {
                max = arr[i];
            }
        }
        for(int exp = 1; max/exp > 0; exp *= 10) {
            bucketSort(arr,exp);
        }
}
```

Main:

```
public static void main(String[] args) {
        int[] arr =
{1,18,112,239,85,14,6,12,23,239,5,116,47,1121,19};
        radixSort(arr);
        for(int i = 0; i < arr.length; i++) {
            System.out.println(arr[i]);
        }
}
```