**Ben Kuhn**

**PS 2: Part I**

## Problem 1: Sorting practice

**1-1)**

{2, 3, 12, 13, 34, 24, 50, 27}

**1-2)**

The inner do...while loop will be skipped 3 times (for elements 27, 34, and 50).

**1-3)**
{13, 3, 2, 24, 12, 27, 50, 34}

**1-4)**
{3, 2, 13, 12, 24, 27, 34, 50}

**1-5)**
{12, 3, 2, 13, 34, 27, 50, 24}

**1-6)**
{50, 2, 12, 3, 13, 24, 34, 27}

**1-7)**
{3, 13, 24, 27, 2, 34, 50, 12}

## Problem 2: Counting comparisons

**2-1)**
Selection sort makes 15 comparisons. For each index i from 0 to the length of the array, selection sort compares the element at i to every element with an index greater than i, regardless of whether the array is sorted or not. So the number of comparisons can be expressed by (n(n-1) / 2), or 15.

**2-2)**
Insertion sort makes 5 comparisons. For each index i from 1 to the length of the array, insertion sort compares the element at i to the element at i - 1. Since in a sorted array the element at i will always be greater than or equal to the element at i - 1, insertion sort will make no additional comparisons, so the total number of comparisons is n - 1 or 5.

**2-3)**
Mergesort will make 9 comparisons. I worked through the algorithm line-by-line to get this answer. For each half of the array, one comparison is made at the highest level of recursion and two comparisons are made at the next level of recursion. This sums to six comparisons, and three comparisons are made to merge the two halves of the array, making nine comparisons total.

**Problem 3: Comparing two algorithms**

**3-1)**

In the worst case (one in which there are no duplicates), the algorithm will make n - 1 passes through the array and during each pass will make an average of n / 2 comparisons. Therefore, c(n) is $(n^2 - n) / 2$. Since the $n^2$ term grows most quickly, the worst-case time efficiency is $O(n^2)$.

**3-2)**

If we consider the second part of the algorithm first, the algorithm passes through the array only once, making n - 1 comparisons during this pass. So the worst case time efficiency for this part of the algorithm is O(n). Now if we consider the first part of the algorithm, the worst case time efficiency of mergesort is O(nlogn). The time efficiency of mergesort supersedes the time complexity of the second part of the algorithm, so the time complexity of the overall algorithm is O(nlogn).

**Problem 4: Sum generator**

**4-1)**

(n * (n + 1) / 2)

**4-2)**

If we simplify the equation above, it is equivalent to $(n^2 + n) / 2$ or $n^2 / 2 + n / 2$. Ignoring the coefficients, the largest term is $n^2$, so the time efficiency is $O(n^2)$.

**4-3)**

```
    public static void generateSums(int n){
        int lastSum = 0;
        int sum = 0;

        for (int i = 1; i <= n; i++){
            sum = i + lastSum;
            lastSum = sum;
            System.out.println(sum);
        }
    }
```

**4-4)**

The number of operations for this implementation is linear with respect to n; during each iteration from i = 1 to i = n there are two computations done (setting sum equal to i + lastSum and setting lastSum equal to the new sum). So the number of operations is 2n, and the time efficiency is O(n).

**Problem 5: Stable and unstable sorting**

To demonstrate that selection sort is an unstable sorting algorithm, start with a four element array arr:

arr = {1, 5a, 5b, 2}

where 5a and 5b both represent the key 5. Selection sort proceeds as follows:

Step 1: {**1**, 5a, 5b, 2}  // 1 is already at the correct index
Step 2: {1, **2,** 5b, **5a**}  // swap 2 and 5a
Step 3: {1, 2, **5b**, 5a}  // no swap occurs because 5 == 5
At the conclusion of selection sort, 5a occurs after 5b.

**Problem 6: Practice with references**

**6-1)**

| **Expression** | **Address** | **Value** |
|---|---|---|
| x | 0x128 | 0x840 |
| x.ch | 0x840 | ‘h’ |
| y.prev | 0x326 | 0x400 |
| y.next.prev | 0x666 | 0x320 |
| y.prev.next | 0x402 | 0x320 |
| y.prev.next.next | 0x322 | 0x660 |

**6-2)**

```
x.next = y;
x.prev = y.prev;
y.prev = x;
x.prev.next = x;
```