Lab 2

1. Compute the number of primitive operations for each of the following algorithm fragments. The "increment counter" step has not been mentioned explicitly – be sure to take this into account in your computations. *Hint*: When faced with nested `for` loops, compute the number of steps required by the inner loop first, and then figure out the effect of the outer loop.

   A.
   ```
   sum ← 0
   for i ← 0 to n-1 do
      sum ← sum + 1
   ```

   B.
   ```
   sum ← 0
   for i ← 0 to n-1 do
      for j ← 0 to n-1 do
         sum ← sum + 1
   ```

2. Determine the asymptotic running time of the following procedure (an exact computation of number of basic operations is not necessary):

   ```
   int[] arrays(int n) {
      int[] arr = new int[n];
      for(int i = 0; i < n; ++i){
         arr[i] = 1;

      }
      for(int i = 0; i < n; ++i) {
         for(int j = i; j < n; ++j){
            arr[i] += arr[j] + i + j;
         }
      }
      return arr;

   }
   ```

3. Consider the following problem: As input you are given two sorted arrays of integers. Your objective is to design an algorithm that would merge the two arrays together to form a new sorted array that contains all the integers contained in the two arrays. For example, on input

   `[1, 4, 5, 8, 17], [2, 4, 8, 11, 13, 21, 23, 25]`

   the algorithm would output the following array:

   `[1,2,4,4,5,8,8, 11, 13, 17, 21, 23, 25]`

   For this problem, do the following:

   A. Design an algorithm `Merge` to solve this problem and write your algorithm description using the pseudo-code syntax discussed in class.
   B. Examining your pseudo-code, determine the asymptotic running time of this merge algorithm. Here, let *n* denote the sum of the lengths of the two arrays:
   `n = arr1.length + arr2.length`
   C. Implement your pseudo-code as a Java method `merge` having the following signature:
   `int[] merge(int[] arr1, int[] arr2)`
   Be sure to test your method in a `main` method to be sure it really works!

4. Below, an algorithm called *removeDups* is provided. Its purpose is to extract from the input list L a list M of all the distinct elements of L.
   A. Explain why the running time of *removeDups* is O(n^2) (remember to consider the running time of M.contains(x))
   B. Try using the technique shown in the solution to the Sum of Two Problem (i.e. using a HashMap) to improve running time to O(n). Be sure to *prove* that running time of your new algorithm is O(n).
   C. Prove your algorithm in B is correct – to do this, come up with a loop invariant I(i). *Hint.* At stage i, M contains the distinct elements contained in [L[0]…L[i]].

<u>Rules</u>: For B, you may *not* use any of the implementations of the Set interface in the Java libraries. If you use HashMap, you may assume that its `get`, `put`, and `containsKey` operations run in O(1) time.

```
Algorithm removeDups (L)
  Input a list L
  Output a list M containing the distinct elements of L
  M ← new list
  for i ← 0 to L.size() -1 do
     if not M.contains(L[i]) then
           M.add(L[i])
  return M
```

5. The following is the code for a sorting algorithm known as *BubbleSort*.

```
int[] arr = //initialized to have n elements
void sort(){
    int len = arr.length;
    for(int i = 0; i < len; ++i) {
      for(int j = 0; j < len-1; ++j) {
        if(arr[j] > arr[j+1]){
          swap(j,j+1);
        }
      }
    }
}
void swap(int i, int j){
    int temp = arr[i];
    arr[i] = arr[j];
    arr[j] = temp;
}
```

   A. Specify a best case and a worst case for BubbleSort.
   B. What are the best-case and worst-case running times for BubbleSort?
   C. Modify BubbleSort so that it has a best-case running time of O(n). Call your modified version BubbleSort1. Use the sort environment to verify that your modified version runs faster.
   D. Prove BubbleSort is correct (you may use your updated version for this if you want). *Hint:* Show that after the loop with i = 0, the element in position $n - 1$ is in final sorted order; after $i = 1$, the last two elements arr[n-2], arr[n-1] are in final sorted order. In general, after pass i, the elements arr[n-i-1]..arr[n-1] are in final sorted order. Let I(i) be the statement
      the elements arr[n-i-1]..arr[n-1] are in final sorted order.
   E. Show that BubbleSort is inversion-bound (you can use either version of the algorithm here). *Hint*: Suppose A is an input array of distinct integers, i<j, and A[i] > A[j] (in other words, (A[i], A[j]) is an inversion in A). Show that at some point during execution, BubbleSort will perform a comparison of A[i] and A[j] and then will swap A[i], A[j].

6. *Interview Question.* An array A holds n integers, and all integers in A belong to the set {0, 1, 2}. Describe an O(n) sorting algorithm for putting A in sorted order. Your algorithm may not make use of auxiliary storage such as arrays or hashtables (more precisely, the only additional space used, beyond the given array, is O(1)). Give an argument to explain why your algorithm runs in O(n) time.