Lab 3

1. Suppose we perform a sequence of n operations on a data structure in which the i-th operations cost i if i is an exact power of 2 and 1 otherwise. Use both Aggregate analysys method and Amortized analysis method to determine the amortized cost per operation.

2. Improve the BubbleSort implementation so that in the best case (which means here that the input is already sorted), the algorithm runs in O(n) time. Explain why your new version works --- in other words, prove that the best case running time of your code is O(n). Call your new Java file BubbleSort1.java.

3. Recall that in BubbleSort, at the end of the first pass through the outer loop, the largest element of the array is in its final sorted position. After the next pass, the next largest element is in its final sorted position. After the *i*th pass (i=0,1,2,…), the largest, second largest,…, i+1st largest elements are in their final sorted position. Use this observation to cut the running time of BubbleSort in half. Implement your solution in code, and prove that you have improved the running time in this way. Call your new Java file, which contains the improvements from this problem and the previous problem, BubbleSort2.java.

4. 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.