Alex Thurston - CIT 594 HW1 Theory Writeup

1. A binary search algorithm could perform this search with $O(\log n)$ worst case time complexity. Please see the attached TheoryCode.java file for the code and a thorough description of the search strategy.

2. (a) The outer loop will execute $(3 - n)$ times. Note that this means that the algorithm will run in constant time for $n \geq 3$, since neither loop will be entered and only primitive operations (variable assignment and comparison) will be performed. The inner loop will execute $n - \frac{n}{2}$ times, and it will run in constant time for $n \leq 0$.

   Considering this as a whole, both loops will be executed only when $n = 2$ or $n = 1$. Since we are discussing the growth rate of the function as n increases, this gives us $O(1)$ order of growth.

   (b) Assuming $n \geq 2$, the outermost loop will execute $n - 1$ times. The first nested loop will execute $(n - 1) \times (n - 1)$ times for the worst case where $i = n - 1$. The innermost nested loop will execute $j - 1$ times. In the worst case, $j - 1 = i^2 - 1 = (n - 1) \times (n - 1) - 1$. Disregarding constants and considering only the most significant terms, this becomes $n \times n^2 \times n^2 = n^5$. So this becomes $O(n^5)$ order of growth.

3. We say $f(x) = O(g(x))$ if there is a real constant $c > 0$ and an integer constant $n_0 \geq 1$ s.t. $f(x) \leq cg(x)$ for every $x \geq n_0$. Here, we can see that a constant $c = 3$ and an integer $n_0 = 1$ would give us $cg(x) = 3(x^4 + 5x^3 + 17x^2 + 13x + 5)$. Under these conditions, $f(x) \leq cg(x)$ for every $x \geq n_0$. Thus, it is true that $f(x) = O(g(x))$.

4. (a) Regardless of whether the algorithm is sorted or unsorted, the array will be passed into isSorted. isSorted checks the array from beginning to end element by element, so it runs in linear time. For an unsorted n-element array, this operation will take $n-1$ steps in the worst case, where the last two elements in the array are out of order.

   If the array is unsorted, it is then passed to linearSearch, which performs a similar element by element search from the beginning of the array to the end. Likewise, for an n-element array, this would take n steps in the worst case scenario, where the element does not exist in the array.

   Thus, for an unsorted array, efficientFind would take $(n - 1) + n = 2n - 1$ steps, and it would run in $O(n)$ (linear) time.

   (b) If the array is sorted, isSorted will take n-1 steps to verify this, since

it needs iterate through the entire array from front to back. Thus, this step runs in linear time. The array will then be passed to binarySearch, which performs a more efficient search that runs in logarithmic time.

Since we only consider the largest term when determining time complexity, efficientFind will still run in $O(n)$ time when the array is sorted.