CSCI 3104 Assignment 1

Jackson Chen

24 January 2016

1. (a)

```
ternarySearch( [1,3,5,10,12,15,32,91,125,132], 18)
ternarySearch( [10,12,15], 18)
False
```

(b) **Theorem:** The procedure `ternarySearch(a,k)` returns `True` if and only if k is contained in a, and returns `False` otherwise.

Base case: If a contains no elements, then `len(a) == 0` and the function returns `False`. This behavior is correct because an element k cannot be found in an array with no elements.

Induction Hypothesis: Assume `ternarySearch` works for an array with a range of elements from 0 to n, then `ternarySearch` works for an array of size $n+1$.

Proof: Since a is already sorted, if `k < a[m]` where m is `len(a)/3`, then `k < a[p]` for all `p > m`. Thus we only need to search the range [0, m) for k. The range [0, m) is necessarily smaller than the range of a and we have already assumed that `ternarySearch` works for all arrays of sizes smaller than $n+1$, therefore `ternarySearch` will work for the range [0, m).
If `a[m] <= k < a[2m]`, then we only need to search the range [m, 2m). The range [m, 2m) is necessarily smaller than the range of a and we have already assumed that `ternarySearch` works for all arrays of sizes smaller than $n+1$, therefore `ternarySearch` will work for the range [m, 2m).
If `k >= a[2m]`, then we only need to search the range [2m, n) where n is `len(a)`. `ternarySearch` will work for this range because of similar logic as stated above.

Therefore the theorem is correct by mathematical induction because it works for arrays of size 0 and for size $n+1$ if we assume that it works for arrays of up to size n.

(c)

$$T(n) = \begin{cases} c_0 & n \leq 2 \\ T\left(\frac{n}{3}\right) + c_1 & n > 2 \end{cases}$$

$$= T\left(\frac{n}{9}\right) + 2c_1$$

$$= T\left(\frac{n}{3^j}\right) + j \cdot c_1$$

The termination case is when $\frac{n}{3^j} \leq 1$ or $\log_3 n \leq j$

$$T(n) = c_0 + c_1 \cdot \log_3 n$$

$$\boxed{T(n) = \Theta(\log_3 n)}$$

2. (a)

```
def ksort(a):
    outofOrder = []
    for i in range(1, len(a)):
        for j in range(0, i):
            if (a[j] > a[i]):
                outofOrder.append(a[i])
                break
    return outofOrder
```

The time complexity is $T(n) \approx O(n^2)$.

(b) In, say, the jth step in insertion sort, a[j] is either in the sorted order, where insertion takes $O(1)$ time or unsorted order, where insertion takes $O(n)$ time.
Since there are k elements out of position, the running time would be $O(n \cdot k)$.

(c) Mergesort still continues to split the array into multiple two-element arrays and then proceeds to merge them back together, despite the fact that the array is already sorted. Mergesort splits arrays in at a logarithmic speed while using linear speed to merge, hence $O(n \log n)$. Because insertion sort is extremely efficient for almost sorted arrays, we could test to see how close the array is to being fully sorted and then choose the most efficient sorting algorithm based on the test.