Caio Brighenti
COSC 302 - Analysis of Algorithms – Spring 2019
Assignment 3

1. Maximum subarray
   Our algorithm, in pseudocode, is as follows.

```
maxSubarray(array A){

          // keep track of max
          max = 0
          // maximum subarray ending at A[j]
          currmax = 0

          for (i in A){
                    // add current element
                    currmax += A[i]

                    // if subarray goes under 0, start new subarray
                    currmax = max(currmax,0)

                    if (currmax > max)
                              max = currmax

                    }
                    return max
}
```

   This algorithm runs in linear time, as it visits each element in the array A once. Next, we briefly explain the algorithm's functionality.
   Explanation
   The algorithm consists of two persistent variables, and one loop. At all points, it keeps track of two sub array values: an overall maximum, stored in the variable max, and the maximum subarray ending at $A[j]$, stored in currmax. Both are initialized to 0, which is what we would expect for an empty array. Let us consider maintenance for each the variables throughout the loop, starting with currmax. At any index j, the maximum subarray ending at j must be equal to the maximum subarray ending at $A[j-1]$ plus the value at $A[j]$. In the case that $A[j]$ is positive, it is obvious that this continuous subarray will continue to grow. In the case that $A[j]$ is negative, the subarray will shrink, but if the subarray continues to be positive overall than it is still the greatest subarray up until $A[j]$, and is what is continued to $A[j+1]$. However, if $A[j]$ is sufficiently small, the subarray currmax may become smaller than 0, in which case we empty the subarray, as an empty subarray must a maximum with respect to a negative one. This is equivalent to choosing $A[j+1]$ as the new beginning of the subarray which will now be considered.
   We have shown the algorithm appropriately calculates the largest subarray ending at each index j in A. The algorithm will always set the variable max to be the largest of these subarrays. As a maximum subarray in A must end at an index j, than the variable max will have the largest subarray in A, and thus the algorithm is correct.

2. **Reccurence relations**

3. **Quick sort**
   By combining the quick sort algorithm with the insertion sort algorithm, we will have an algorithm composed of two smaller pieces. To show the overall running time for this algorithm, we consider each section separately and then combine the two.

Quick sort:
For simplicity, we consider quick sort with a balanced partition each time. Each level will thus divide n by 2, until we reach the level such that n goes to 1. By stopping when the subarrays at equal to k, however, we stop at the level $\frac{n}{k}$. Thus, the typical running time goes form $O(n \log n)$ to $O(n \log \frac{n}{k})$. This leaves $\frac{n}{k}$ arrays of length k which must be sorted by insertion sort.

Insertion Sort:
The running time of insertion sort is $O(n^2)$ in the worst and average case. We assume one of these two cases, as the best case is unreasonable in this case, given that it assumes the array is already sorted. Thus the total running time for sorting all unsorted arrays with insertion sort is $\frac{n}{k}O(k^2)$. Through algebra, this can be clearly simplified to $O(nk)$.

Overall running time:
As we first sort most of the array using quick sort, and sort the remainder with insertion sort, the overall running time is equal to the sum of both pieces. Thus, the overall running time is $O(n \log \frac{n}{k}) + O(nk) = O(n \log \frac{n}{k} + nk)$. Next we consider how to choose a value k

Choosing K:
It is plain to see that we need a value k such that $n \log n > n \log \frac{n}{k} + nk$. Let us thus solve for k in this inequality.

$$\begin{aligned}
n \log n &> n \log \frac{n}{k} + nk && && (1)\\
\log n &> \log \frac{n}{k} + k && \text{divide both sides by } n && (2)\\
\log n &> \log n - \log k + k && \text{log rules} && (3)\\
\log k &> k && \text{subtract } \log n \text{, algebra} && (4)
\end{aligned}$$

This is clearly impossible, and would suggest no such k exists. However, this is because we must consider the constant terms. We solve again, this time including the constant c_1 for quick sort and the constant c_2 for insertion sort.

$$\begin{aligned}
c_1 n \log n &> c_1 n \log \frac{n}{k} + c_2 nk && && (5)\\
c_1 \log n &> c_1 \log \frac{n}{k} + c_2 k && \text{divide both sides by } n && (6)\\
c_1 \log n &> c_1 \log n - c_1 \log k + c_2 k && \text{log rules} && (7)\\
\log k &> \frac{c2}{c1} k && \text{subtract } c_1 \log n \text{, algebra} && (8)
\end{aligned}$$

Thus, using the constant terms from quick sort and insertion sort we would be able to locate a value of k such that this combination of algorithm works more efficiently than simply using quick sort. In practice, we might also find such a k more easily through statistical simulation. That is, we might randomly generate high volumes of arrays to be sorted, and experiment with different values of k to experimentally determine a value that minimizes the runtime.