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.

$$n \log n > n \log \frac{n}{k} + nk \tag{1}$$

$$\log n > \log \frac{n}{k} + k \qquad \text{divide both sides by } n \tag{2}$$

$$\log n > \log n - \log k + k \qquad \text{log rules} \tag{3}$$

$$\log k > k \qquad \text{subtract } \log n\text{, algebra} \tag{4}$$

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.

$$c_1 n \log n > c_1 n \log \frac{n}{k} + c_2 nk \tag{5}$$

$$c_1 \log n > c_1 \log \frac{n}{k} + c_2 k \qquad \text{divide both sides by } n \tag{6}$$

$$c_1 \log n > c_1 \log n - c_1 \log k + c_2 k \qquad \text{log rules} \tag{7}$$

$$\log k > \frac{c2}{c1} k \qquad \text{subtract } c_1 \log n\text{, algebra} \tag{8}$$

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.

4. **Heap data structure**

   (a) How would you represent a d-ary heap in an array?

   We consider an array representation indexed by 1, such that the root of the heap is at $A[1]$. As any node will have d children, the children of the root must be in the range $A[2]$ to (inclusive) $A[d+1]$. Each of these d children will have d children themselves, giving the third level d^2 children, which will be in the range $A[d+2]$ through $d^2 + d + 1$. This pattern continues (the next level will have d^3 items, the next d^4, and so forth.)

We also need a system to calculate the parent and children of a node using its index i. Suppose that we can calculate the parent of a node i with $\lfloor \frac{i-2}{d+1} \rfloor$ and the children by $d(i-1)+j+1$, where j represents the $j-th$ child of i in the range $1 \leq j \leq d$. We can see that this procedure works by choosing any appropriate i, j, and d, and verifying that parent(child(i)) results in i.

(b) What is the height of a d-ary heap of n elements in terms of n and d?

To find the height, we leverage that each level of the tree contains d^l nodes, where l is the current level, as each node has d children. Thus the total number of nodes is equal to the sum $\sum_{l=0}^{h-1} d^l$, which simplifies to $d^h - 1$, where h is the height of the tree. Given that n is the total number of nodes, we have that $d^h = n$, removing the 1 for simplicity as we are considering asymptotic notation. By taking the log of this, we have that $h = \log_d n = \Theta(\frac{\log n}{\log d})$.

(c) Give an efficient implementation of EXTRACT-MAX in a d-ary max-heap. Analyze the running time in term d and n.

Algorithm:
We consider if the standard efficient procedure for EXTRACT-MAX on page 163 of book 1 for a binary heap works for a d-ary heap. In simple terms, that algorithm removes the root of the tree (the max item) and places the last leaf in its place. Then, the algorithm simply calls MAX-HEAPIFY on the whole tree. It is important to note that the procedure of extracting the max and placing the leaf at the top maintains the max heap property of the left and right subtrees. It is left to show that the MAX-HEAPIFY algorithm works also works in the d-ary case.
The MAX-HEAPIFY algorithm available on page 154 also clearly works in the case of a d-ary heap. On each recursive call, it simply compares the root to each of it's children, swapping it with the maximum child and recursively proceeding on the new subtree containing the swapped root. The only difference is that this algorithm now makes d comparisons, as opposed to 2. As this part of the overall algorithm works, then clearly the overall EXTRACT-MAX algorithm also works for a d-ary heap.
Running time
As the EXTRACT-MAX algorithm simply makes an operation in constant time, and then calls MAX-HEAPIFY, its overall running time is the running time of MAX-HEAPIFY plus a constant term. In the binary case, MAX-HEAPIFY has a running time of $O(h)$, where h is the height of the tree (page 156). We have shown the height of tree here to be $\Theta(\frac{\log n}{\log d})$. However, the algorithm also has increased overhead at each recursive call, which must now make d comparisons. Thus, the running time for algorithm is $O(1) + O(d\frac{\log n}{\log d}) \in O(d\frac{\log n}{\log d})$.

(d) Give an efficient implementation of INSERT in a d-ary max-heap. Analyze its running time in terms of d and n.

Algorithm:
As in the previous problem, we will use the INSERT algorithm for a typical max heap available in the book, on page 164. In simple terms, this algorithm increases the size of the heap by 1, and adds an infinitely small value to the last index in the array. Then, the algorithm makes a call to INCREASE-KEY to swap the new value to be inserted with the infinitely small value just added. We will assume that INCREASE-KEY works for a d-ary heap, as an algorithm for that is precisely what we will provide in the next part of this problem. For now, we assume that it works.
Running time:
The running time of INSERT has two parts: the operations within the function, and the running time of the call to INCREASE-KEY. The operations within INSERT are clearly constant, therefore this section is $\Theta(1)$. The INCREASE-KEY function, on the other hand, depends on the height of the tree. This will be explained in more detail in the next problem, but it is clear the algorithm's loop executes one per each level until the appropriate level is found. Therefore, in the worst case it will be $O(h) = O(d\frac{\log n}{\log d})$. As the running time for the first section was constant, than the overall running time will simply be $O(d\frac{\log n}{\log d})$.

(e) Give an efficient implementation of INCREASE-KEY(A, i, k), which flags an error if $k < A[i]$, but otherwise sets $A[i] = k$ and then updates the d-ary max-heap structure appropriately. Analyze its running time in terms of d and n.

As has been the case for our previous algorithms, the algorithm provided in the book works (mostly) unchanged for a d-ary heap. In short, this algorithm swaps the given key within the heap with the new value passed to the function, and then moves it up to the appropriate spot. In the case that the new value is lower than the key given, the algorithm returns an error. In order to move the new value up to its appropriate spot, the algorithm performs a while loop, checking in each iteration whether the value's parent is smaller than the value itself and if this condition is met swapping the parent with the value. Interestingly, this procedure works identically in this case as to the binary case, as the number of children has no bearing on the path taken by the new value up to its appropriate position. The only small modification needed is to make sure the parent index is calculated appropriately, but this is simply a constant operation.
Running time: To compute the running time for this algorithm, we consider the constant operations and the while loop in the algorithm. As the algorithm simply checks a condition and then performs one simple operation, this section is clearly $\Theta(1)$. The loop, however, iterates until the appropriate level is found for the new value. In the worst case, this will be the root of the heap. As such, the number of iterations will be equal to the height of the heap, as it must traverse from the bottom to the top. Thus, the overall running time is $\Theta(1) + O(h) = \Theta(1) + O(d\frac{\log n}{\log d}) \in O(d\frac{\log n}{\log d})$.