Mystery Code I

In line 05 the procedure maxthree is calling itself on a version of the input list with the kth element (a_k) removed. Assume it takes constant time to temporarily remove a_k from the list. (Doing this in constant time actually requires some extra details that we are hiding for clarity.)

```
00 maxthree($a_1, \ldots, a_n$) : list of $n$ positive integers, $n \geq 3$)
01     if ($n = 3$) return $a_1 + a_2 + a_3$
02     else
03         bestval = 0
04         for $k = 1$ to $n$
05             newval = maxthree($a_1, a_2, \ldots, a_{k-1}, a_{k+1}, \ldots a_n$)
06             if (newval > bestval) bestval = newval
07         end for
08         return bestval
```

(a) Describe (in English) what maxthree computes.

(b) Suppose that $T(n)$ is the running time of maxthree on an input array of length n. Give a recursive definition of $T(n)$.

(c) How many leaf nodes are there in the recursion tree for $T(n)$? Briefly explain.

(d) Does maxthree run in $O(2^n)$ time? Briefly explain why or why not.

Solution:

(a) maxthree computes the largest sum of 3 numbers in the list.

(b) $T(3) = c_1$
$T(n) = n \cdot T(n-1) + c_2 n + c_3$
There loop on lines 4 - 7 runs n times and makes a recursive call each time on a list one shorter. It also does some constant work inside and outside the loop.

(c) $\frac{n!}{3!}$ This happens since the last level is when the list is 3 and at each level there are children equal to the lenght of the list. So $n \cdot (n-1) \cdots 3$.

(d) Given that there are $\Theta(n!)$ leaves even with no other work it is easy to see that it is way more than $O(2^n)$ time.

Mystery Code III

Consider an array of n *distinct* real numbers $a_1, a_2, \ldots, a_n$. We say that the array has a *peak* at position k if the following two conditions hold for every position j between 2 and n:

(1) If $j \le k$, then $a_{j-1} < a_j$.

(2) If $j > k$, $a_{j-1} > a_j$.

Consider the following procedure to determine position of the peak of an array (assume that the array does indeed have a peak):

```
00 procedure FindPeak($a_1$, $a_2$, ..., $a_n$: array of real numbers)
01     if (n = 1)
02         return 1
03     if ($a_1 > a_2$)
04         return 1
05     else if ($a_n > a_{n-1}$)
06         return n
07     k = floor((1+n)/2)
08     if ($a_{k-1} > a_k$)
09         return FindPeak($a_1$, . . . , $a_{k-1}$)
10     else if ($a_k < a_{k+1}$)
11         return FindPeak($a_{k+1}$, . . . , $a_n$) + k
12     else
13         return k
```

(a) Consider the array $-1, 3, 6, 7, 0$. Trace the execution of the above pseudocode and show that it correctly returns the position of the peak.

(b) At line 07, what is the smallest value that n might contain? Why?

(c) Let $T(n)$ be the worst-case running time of the above pseudocode when the array has size n. Write a recurrence for $T(n)$, including the necessary base case(s). Assume that splitting the array (lines 09 and 11) takes constant time.

(d) What is the tightest big-O running time of FindPeak?

Solution:

(a) FindPeak([-1 3 6 7 0]) will not match any lines 1,3,5 so will compute the $k = 3$ and then take the else if and call FindPeak([7 0]) which will the if at line 3 and return 1. The original function will then return $1 + 3 = 4$.

(b) **3**, If n were 1, we would have returned on line 1. If n were 2, we would return on either line 4 or line 6 (because the first item is either greater than or less than the second/last). However on an input array with 3 elements whose peak is in the center, like [5, 6, 4], we can reach line 7.

(c) $T(1) = c_1$
$T(2) = c_2$
$T(n) = T(n/2) + c_3$

(d) If we consider the recursion tree of this code it is clear that at each level there is only one child and each node only does a constant amount of work. Thus the runtime will be simply the height of the tree. To compute this we can see that at each level the list gets smaller by one half so at most the height of the tree is $O(\log n)$.