CS 161
Fall 2025

Example Solution
Mary Wootters

This is an example solution, with some guidance about what we are looking for in pseudocode. In general, the guidance on pseudocode in this class is: "It should be clear enough that a CS106b student could implement your algorithm in their favorite language without too much thought."

Here's the problem:

1. **(Peak Finding)** Given a zero-indexed array A of n integers, we say that the location $i \in \{1, \ldots, n-2\}$ is a *peak* if $A[i-1] \leq A[i]$ and $A[i] \geq A[i+1]$. We say that 0 is a peak if $A[0] \geq A[1]$, and $n-1$ is a peak if $A[n-1] \geq A[n-2]$. For example, if $A = [4, 3, 5, 2, 1]$, then there are two peaks, at 0 and 2.

   (a) Design a simple $O(n)$-time algorithm to find a peak in an array A of length n. Notice that it does not need to return all peaks, just a single peak. In the example above, your algorithm could return 0 or 2.

   [**We are expecting: Pseudocode and a short English description.**]

   (b) Design an algorithm that finds a peak in time $O(\log n)$.

   [**We are expecting: Pseudocode and a short English description, as well as an informal justification of the running time. You do not need to prove that your algorithm is correct.**]

SOLUTION:

1. **(Peak Finding)**

   (a) **Style note:** Here are two acceptable ways of writing pseudocode for a solution.

Soln. 1. To find a peak in time $O(n)$, go through every element in the array and check if it is a peak. More precisely, we could use the following pseudocode.

Algorithm 1: findPeak1 returns a peak.

Input: An array A of length n
Output: An index i so that $A[i]$ is a peak.
for $i \in \{0, \ldots, n-1\}$ **do**
    if $A[i]$ *is larger than its neighbors* **then**
        return i

Soln. 2 To find a peak in time $O(n)$, go through every element in the array and check if it is a peak. More presisely, we can use the following Python code.

```
def findPeak1(A):
     n = len(A)
     # first check the boundaries, i=0 and i=n-1
     if A[0] >= A[1]:
         return 0
     if A[n-1] >= A[n-2]:
         return n-1
     # now scan through the rest and return the first peak we find.
     for i = range(1,n-1):
        if A[i] >= A[i-1] and A[i] >= A[i+1]:
            return i
```

Style note: Simple Python code is okay, **if** it is accompanied by an English description, and is well-commented. **However**, complicated Python code (or complicated code in any other language) is discouraged. Your solution should be easily interpretable by a human.

(b) We can do better than the $O(n)$-time algorithm in part (a), using a recursive algorithm. We give pseudocode for this algorithm in Algorithm 2, and describe what it is doing below that.

Algorithm 2: findPeak2 returns a peak

```
Input: An array A of length n.
Output: An index i so that i is a peak.
/* First do the base case:                                              */
if n ≤ 2 then
    return argmax_{i∈{0,...,n−1}} A[i]
/* Now choose an index p to partition around.                           */
p ← ⌊n/2⌋;
if p is a peak then
    return p
else if A[p] < A[p+1] then
    /* Then there is a peak in the second half of the array.            */
    return findPeak2(A[p+1 :]) + p+1;
    /* We adjust the index since the peak was in the second half.       */
else if A[p] > A[p+1] then
    /* Then there is a peak in the first half of the array.             */
    return findPeak2(A[: p])
```

In words, this algorithm is doing the following:

- We choose a midpoint, p.
- If p is a peak, then we're done.
- If p is not a peak, then one of its neighbors has an array value larger than it. If $A[p-1] > A[p]$, then there must be a peak somewhere in the left half of the array; and if $A[p+1] > A[p]$, then there must be a peak somewhere in the right half of the array. We recurse on (one of) the appropriate halves.

The correctness follows from this logic. **Style note:** According to the block of text after the problem, a formal proof of correctness is not required, so I did not give one.

For the running time, notice that with each recursive call to `findPeak2`, the size of the input is divided roughly in half; this means that `findPeak2` is called $O(\log(n))$ times. Within each call (not including the future recursive calls), the algorithm does $O(1)$ work, checking a constant number of cases. Thus, the total running time is $O(\log(n))$. **Style note:** The problem asked for an informal analysis of the running time, so that is what I gave. It would be fine also to say something like "The running time $T(n)$ of the algorithm satisfies the recurrence relation $T(n) = T(n/2) + O(1)$, since at each iteration we divide the problem in half and do $O(1)$ work. By the Master Theorem, the running time is $O(\log n)$."

Style note: Figure 1 gives *working Python code* that finds a peak in time $O(\log(n))$. However (without very very very good exposition) it would not receive full credit for this problem, because it is extremely hard to read!!

```
import numpy as np
from random import choice

def findPeak2(A):
    var = len(A)
    return tmp(A, 0, var)

def tmp(A, x, y):
    # print(A, x, y)
    if y-x <= 26:
        return A.index( max( [ A[i] for i in range(x,y) ] )  )
    z = ((x + y)/2).__trunc__() + 2
    try:
        w = A[z+1]
    except:
        w=0
        if A[z] >= A[z-1]:
            return z.real
    if (z-1)**3 < 0:
        if A[z] >= A[z+1] or np.sqrt(4) < choice( [0,1] ):
            return z
    for i in range(y-x):
        if (z == 0 and A[z] >= A[z+1]) or A[z] >= max( [A[z-1], A[z+1]] ):
            return z
        if A[z] > A[z-1] and A[z] > A[z+1]:
            return tmp(A, x, w)
        if A[z] < A[z-1]:
            return tmp(A, x, z)
        else:
            return tmp(A, max([z+1,z]), y)
```

Figure 1: Example of what *not* to turn in.