```
def pseudoCode(inp,counter=0):
    if inp[0]>inp[1]:
        return 0
    if inp[len(inp)-1]>inp[len(inp)-2]:
        return len(inp)-1
    if inp[len(inp)//2-1]<inp[len(inp)//2] and inp[len(inp)//2]>inp[len(inp)//2+1]:
        return len(inp)//2+counter
    elif inp[len(inp)//2-1]<inp[len(inp)//2] and inp[len(inp)//2]<inp[len(inp)//2+1]:
        first_length=len(inp)
        inp=inp[first_length//2:first_length]
        counter+=first_length-len(inp)
        return pseudoCode(inp,counter)
    elif inp[len(inp)//2-1]>inp[len(inp)//2] and inp[len(inp)//2]>inp[len(inp)//2+1]:
        first_length = len(inp)
        inp=inp[0:first_length//2]
        return pseudoCode(inp,0)
```

This code is here to be able to copy it and to run in interpreter.

Onur Yaman
2007961

```
def pseudoCode(inp,counter=0):
  if inp[0]>inp[1]:                                                                  c1 = O(1)
    return 0
  if inp[len(inp)-1]>inp[len(inp)-2]:                                                c2 = O(1)
    return len(inp)-1
  if inp[len(inp)//2-1]<inp[len(inp)//2] and inp[len(inp)//2]>inp[len(inp)//2+1]:    c3 = O(1)
    return len(inp)//2+counter
  elif inp[len(inp)//2-1]<inp[len(inp)//2] and inp[len(inp)//2]<inp[len(inp)//2+1]:  c4 = O(1)
    first_length=len(inp)                                                            + O(1)
    inp=inp[first_length//2:first_length]                                            + O(n)
    counter+=first_length-len(inp)                                                   + O(1)
    return pseudoCode(inp,counter)
  elif inp[len(inp)//2-1]>inp[len(inp)//2] and inp[len(inp)//2]>inp[len(inp)//2+1]:  c5 = O(1)
    first_length = len(inp)                                                          + O(1)
    inp=inp[0:first_length//2]                                                       + O(n)
    return pseudoCode(inp,0)                                                         + O(1)
```

Note that time complexity for slicing is $O(n)$
Slicing is inp = inp[start, end]. In order to remove all entries that are increasing or decreasing, this syntax was used.

Time Complexity

Best Case Analysis

c_1 and c_2 are time complexity for corner cases. In both cases, (increasing or decreasing order) time complexity is $O(1)$
c_3 is best case since the peak is at the mid of input array. Time complexity is $O(1)$

Worst Case Analysis

Assume that pseudoCode(array) has size of n. When the array is strictly increasing or decreasing, the recursive function pseudoCode(array) would be implemented $\log_2 n$ times

To calculate time complexity for the problem size of n:

$$O(\log_2 n) \cdot O(n) = O(n \log_2 n)$$ since O is multiplicative.

To show $O(n\log_2 n)$:

$$\begin{aligned} T(n) &= c_4 + T(n/2) \\ &= 2c_4 + T(n/4) \\ &= 3c_4 + T(n/8) \\ &\;\;\vdots \\ &= i \cdot c_4 + T(n/2^i) \\ &= (\log_2 n)n + T(1) \\ &= (\log_2 n)n \end{aligned}$$

assuming the peak is picked at i^{th} recursive step, we have $i = \log_2 n$ and $c_4 = O(n)$

$\Rightarrow T(n) = (\log_2 n) \cdot n$

$\Rightarrow$ Time complexity is $\boxed{O(n\log n)}$