Chapter 1

Introduction to Algorithms.

1.1 Peaks-Finding.

Example 1.1.1. Consider an n-length array A such that $A_1, A_2, \ldots, A_n \in \mathbb{R}$. We will say that A_j is a **peak** (local maximum) if it is greater than its neighbors. Namely, $A_i \geq A_{i\pm1}$ if $i \pm 1 \in [n]$. Whenever $i \pm 1$ is not in the range $[n]$, we will define the inequality $A_i \geq A_{i\pm1}$ to hold trivially. For example, for $n = 1$, $A_1 = A_n$ is always a peak.

Write an algorithm that, given A, returns the position of an arbitrary peak.

Example 1.1.2. Warming up. How many peaks do the following arrays contain?

1. $A[i] = 1 \; \forall i \in [n]$
2. $A[i] = \begin{cases} i & i < n/2 \\ n/2 - i & \text{else} \end{cases}$
3. $A[i] = i \; \forall i \in [n]$

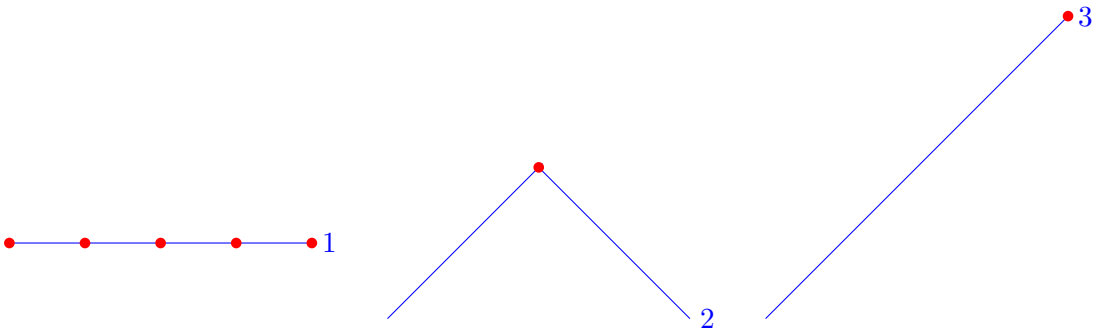

Figure 1.1: Illustration of the arrays on the smooth line, the peaks are marked in red.

1.2 Naive solution.

To better understand the problem, let's first examine a simple solution before proposing a more intriguing one. Consider the algorithm examining each of the

items A_i one by one.

Result: returns a peak of $A_1 ... A_n \in \mathbb{R}^n$
1 **for** $i \in [n]$ **do**
2 **if** A_i *is a peak* **then**
3 **return** i
4 **end**
5 **end**

Algorithm 1: naive peak-find alg.

Correctness. We will say that an algorithm is correct with respect to a given task if it computes the task for any input. Let's prove that the above algorithm is doing the job.

Proof. Denote by j the position of the first peak in A, we will prove that Algorithm 1 returns A_j.

Denote by i the current iteration step[1], and separate upon the following cases:

1. If $i < j$, then by the minimality of j (A_j was defined to be the first peak), it has to hold that A_i is not a peak. Therefore, the condition on line (2) is not satisfied $\Rightarrow$ the algorithm skips to line (4) and from there continues to the next iteration.

2. If $i = j$, then by definition of A_j, the condition on line (2) is satisfied, so the algorithm enters line (3) and returns A_j.

3. Else, consider the case when $i > j$. In that case, the algorithm will not reach the ith iteration. That is because the algorithm is defined to scan the elements in 'continuous' order, meaning that if the algorithm indeed gets into the i iteration, it implies that it passed through any $i' < i$ iteration and didn't return, specifically at the j iteration, which contradicts A_j being a peak.

□

Note that in the third bullet, in the case $i > j$, we proved a general claim about the behavior of simple 'for loop' programs. From now on, whenever we analyze a similar 'loop' that scans elements iteratively, we will not consider cases beyond the returning point and will be satisfied with proving analogous cases to 1 and 2. Also, in the exam/exercises, unless it is the main issue of the question, you are not expected to treat that case. (Spend your time wisely on what really matters.)

[1] In our course, in terms of terminology, usually 'step' = 'iteration' = 'iteration number' = 'turn' = 'index'.

Running Time. Question, How would you compare the performance of two different algorithms? What will be the running time of the naive peak-find algorithm? At the lecture, you will see a well-defined way to treat such questions, but for the sake of getting the general picture, let's assume that we pay for any comparison quanta of processing time and, overall, checking if an item in a given position is a peak, cost at most $c \in \mathbb{N}$ time, a constant independent on n.

Question, In the worst case scenario, how many local checks does peak-finding do? For the third example in Example 1.1.2, the naive algorithm will have to check each item, so the running time adds up to at most $c \cdot n$.

1.3 Naive alg. recursive version.

Now, we will show a recursive version of the navie peak-find algorithm for demonstrating how correctness can be proved by induction.

Result: returns a peak of $A_1 ... A_n \in \mathbb{R}^n$
if $A_1 \geq A[2]$ *or* $n = 1$ **then**
  return 1
end
return 1 + peak-find$(A_2, ..A_n)$

Algorithm 2: naive recursive peak-find alg.

Claim 1.3.1. Let $A = A_1, \ldots, A_n$ be an array, and $A' = A_2, A_3, \ldots, A_n$ be the $n-1$ length array obtained by taking all of A's items except the first. If $A_1 \leq A_2$, then any peak of A' is also a peak of A.

Proof. Let A'_j be a peak of A'. Split into cases upon on the value of j.

1. If $n - 1 > j > 1$, then $A'_j \geq A'_{j\pm1}$, but for any $j \in [2, n-2]$ we have $A'_j = A_{j+1}$ and therefore $A_{j+1} \geq A_{j+1\pm1} \Rightarrow A_{j+1}$ is a peak in A.
2. If $j' = 1$, then $A'_1 > A'_2 \Rightarrow A_2 \geq A_3$ and by combining the assumption that $A_1 \leq A_2$ we have that $A_2 \geq A_1, A_3$. So $A_2 = A'_1$ is also a peak.
3. The last case $j = n - 1$ is left as an exercise.

□

One can prove a much more general claim by following almost the same argument presented above.

Claim 1.3.2. Let $A = A_1, \ldots, A_n$ be an array, and $A' = A_{j+1}, A_{j+2}, \ldots, A_n$ be the $n - j$ length array. If $A_j \leq A_{j+1}$, then any peak of A' is also a peak of A.

We are ready to prove the correctness of the recursive version by induction using Claim 1.3.1.

1. Base, single element array. Trivial.
2. Assumption, Assume that for any m-length array, such that $m < n$ the alg returns a peak.

3. Step, consider an array A of length n. If A_1 is a peak, then the algorithm answers affirmatively on the first check, returning 1 and we are done. If not, namely $A_1 < A_2$, then by using Claim 1.3.1 we have that any peak of $A' = A_2, A_3, \ldots, A_n$ is also a peak of A. The length of A' is $n - 1 < n$. Thus, by the induction assumption, the algorithm succeeds in returning on A' a peak which is also a peak of A.

1.4 An attempt for sophisticated solution.

We saw that we can find an arbitrary peak at $c \cdot n$ time, which raises the question, can we do better? Do we really have to touch all the elements to find a local maxima? Next, we will see two attempts to catch a peak at logarithmic cost. The first attempt fails to achieve correctness, but analyzing exactly why will guide us on how to come up with both an efficient and correct algorithm.

Result: returns a peak of $A_1 ... A_n \in \mathbb{R}^n$
$i \leftarrow \lceil n/2 \rceil$
if A_i *is a peak* **then**
  return i
end
else
  return $i - 1 +$ find-peak$(A_i, A_{i+1}..A_n)$
end

Algorithm 3: fail attempt for more sophisticated alg.

Let's try to 'prove' it.

1. Base, single element array. Trivial.

2. Assumption, Assume that for any m-length array, such that $m < n$ the alg returns a peak.

3. Step. If $A_{n/2}$ is a peak, we're done. What happens if it isn't? Is it still true that any peak of $A_i, A_{i+1}, \ldots, A_n$ is also a peak of A? Consider, for example, $A[i] = n - i$.

1.5 Sophisticated solution.

The example above points to the fact that we would like to have a similar claim to Claim 1.3.2 that relates the peaks of the split array to the original one. Let's prove correction.

Proof. By induction.

1. Base, single element array. Trivial.

2. Assumption, Assume that for any m-length array, such that $m < n$ the alg returns a peak.

Result: returns a peak of $A_1 ... A_n \in \mathbb{R}^n$
$i \leftarrow \lceil n/2 \rceil$
if A_i *is a peak* **then**
  return i
end
else if $A_{i-1} \leq A_i$ **then**
  return $i+$ find-peak$(A_{i+1}..A_n)$
end
else
  return find-peak$(A_1, A_2, A_3..A_{i-1})$
end

Algorithm 4: sophisticated alg.

3. Step, Consider an array A of length n. If $A_{\lceil n/2 \rceil}$ is a peak, then the algorithm answers affirmatively on the first check, returning $\lceil n/2 \rceil$ and we are done. If not, then either $A_{\lceil n/2 \rceil} < A_{\lceil n/2 \rceil - 1}$ or $A_{\lceil n/2 \rceil} < A_{\lceil n/2 \rceil + 1}$. We have already handled the first case, that is, using Claim 1.3.2 we have that any peak of $A' = A_{\lceil n/2 \rceil + 1}, A_{\lceil n/2 \rceil + 2}, \ldots, A_n$ is also a peak of A. The length of A' is $n/2 < n$. So by the induction assumption, in the case where $A_{\lceil n/2 \rceil} < A_{\lceil n/2 \rceil - 1}$ the algorithm returns a peak. In the other case, we have $A_{\lceil n/2 \rceil} < A_{\lceil n/2 \rceil + 1}$ (otherwise $A_{\lceil n/2 \rceil}$ would be a peak). We leave finishing the proof as an exercise.

□

What's the running time? Denote by $T(n)$ an upper bound on the running time. We claim that $T(n) \leq c_1 \log(n) - c_2$.

Proof. By induction.

1. Base. For the base case, $n \leq 3$ we get that $c_1 \log(1) + c_2 = c_2$ on the other hand only a single check made by the algorithm, so indeed the base case holds (Choosing $c_2 \geq$ the cost of a single check).

2. Induction Assumption. Assume that for any $m < n$, the algorithm runs in at most $c_1 \log(m) + c_2$ time.

3. Step. Notice that in the worst case, $\lceil n/2 \rceil$ is not a peak, and the algorithm calls itself recursively immediately after paying c_2 in the first check. Hence:

$$\begin{aligned} T(n) &\leq c_2 + T(n/2) \leq c_2 + c_1 \log(\lceil n/2 \rceil) + c_2 \\ &= c_1 - c_1 + c_2 + c_1 \log(\lceil n/2 \rceil) + c_2 \\ &= c_1 \log(2) + c \log(\lceil n/2 \rceil) + 2c_2 - c_1 \\ &= c_1 \log(2(\lceil n/2 \rceil)) + 2c_2 - c_1 \end{aligned}$$

So, choosing $c_1 > c_2$ gives $2c_2 - c_1 < c_2$ and therefore:

$$T(n) \leq c_1 \log(n) + c_2$$

□