

Chapter 1

Introduction to Algorithms, Correctness and Efficiency.

Computer science differs from other scientific disciplines in that it focuses not on solving or making discoveries, but on questioning how good is our current understanding. The fact that one has successfully come up with an idea for a certain problem immediately raises the question of optimality. At the most basic level, we would like to answer what is the 'best' program that exists for a particular problem. To do so, we must have a notation that allows us to determine if an algorithm is indeed solving the task, quantify its performance, and compare it to other algorithms. In this chapter, we introduce this basic notation. The chapter is divided into two main parts: the first is about induction, a mathematical technique for proving claims, and the second presents asymptotic notation, which we use to describe the behavior of algorithms over large inputs.

Note 1: text for right-hand side of pages, it is set justified.

1.1 Peaks-Finding.

Example 1.1.1 (Leading Example.). Consider an n -length array A such that $A_1, A_2, \dots, A_n \in \mathbb{R}$. We will say that A_j is a peak (local minimum) if he's greater than his neighbors. Namely, $A_i \geq A_{i\pm 1}$ if $i \pm 1 \in [n]$. Whenever $i \pm 1$ is not in the range $[n]$, we will define the inequality $A_i \geq A_{i\pm 1}$ 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. $A[i] = i \ \forall i \in [n]$

1.2 Naive solution.

For capturing an understanding of the problem, let's study a simple solution before suggesting a much more interesting one. Consider the algorithm examining each

of the items A_i one by one.

Result: returns a peak of $A_1 \dots A_n \in \mathbb{R}^n$

```

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. Assume towards contradiction that there exists an n -length array A such that the algorithm peak-find fails to find one of its peaks, in particular, the Alg. returns $j' \in [n]$ such that $A_{j'}$ is not a peak. Denote by j the first position of a peak in A , and note that if the algorithm gets to line (2) in the j th iteration then either it returns j or A_j is not a peak.

Hence it must hold that $j' < j$. But a satisfaction of the condition on line (2) can happen only if $A_{j'}$ is a peak, which contradicts the minimality of j . \square

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? On 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 a quanta of processing time, and in overall the checking if an item in a given position is a peak we pay at most $c \in \mathbb{N}$ time, a constant independent on n .

Question, In the worst case scenario, how many times does peak-finding need to check if an item A_i is a peak? 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 naive peak-find algorithm for demonstrating how correctness can be proved by induction.

Result: returns a peak of $A_1 \dots A_n \in \mathbb{R}^n$

```

1 if  $A_1 \geq A[2]$  or  $n = 1$  then
2   return  $A[1]$ 
3 end
4 return  $1 + \text{peak-find}(A_2, \dots, A_n)$ 
```

Algorithm 2: naive recursive peak-find alg.

Claim 1.3.1. *Let $A = A_1, \dots, A_n$ be an array, and $A' = A_2, A_3, \dots, A_n$ 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 . If $n - 1 > j > 1$, then $A'_j \geq A'_{j \pm 1}$, but for any $j \in [2, n - 2]$ we have $A'_j = A_{j+1}$ and therefore $A_{j+1} \geq A_{j+1 \pm 1} \Rightarrow A_{j+1}$ is a peak in A . 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. The last case $j = n - 1$ is left as an exercise. \square

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 $A' = A_2, A_3, \dots, A_n$ 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 $A_1 \dots A_n \in \mathbb{R}^n$

```

1  $i \leftarrow \lceil n/2 \rceil$ 
2 if  $A_i$  is a peak then
3   | return  $i$ 
4 end
5 else
6   | return  $i - 1 + \text{find-peak}(A_i, A_{i+1} \dots A_n)$ 
7 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}, \dots, 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 1.3.1 that relates the peaks of the split array to the original one.

Result: returns a peak of $A_1 \dots A_n \in \mathbb{R}^n$

```

1  $i \leftarrow \lceil n/2 \rceil$ 
2 if  $A_i$  is a peak then
3   | return  $i$ 
4 end
5 else if  $A_{i-1} \leq A_i$  then
6   | return  $i - 1 + \text{find-peak}(A_i, A_{i+1} \dots A_n)$ 
7 end
8 else
9   | return  $\text{find-peak}(A_1, A_2, A_3 \dots A_{i-1})$ 
10 end
```

Algorithm 4: sophisticated alg.

1.6 Induction.

Suppose that a teacher, who is standing in front of his class, is willing to prove that he can reach the door at the corner. One obvious way to do so is to actually reach the door; that is, move physically to it and declare success. For small classes containing a small number of students, this protocol might even be efficient, lasting less than several seconds. But what if the class is really big, maybe the length and width of a football stadium? In that case, proving by doing might take time. So the obvious question to ask is, what else can we do? Is there a more efficient way to prove this?

Indeed, there is. Instead of proving that he can reach the door, he can prove that while he does not stand next to the door, nothing can stop him from keeping moving forward. If that is indeed the case, then it's clear that not reaching the door in the end would be a contradiction to being just one step away from it (why?), which, in turn, would also contradict being two steps away from it. Repeating this argument leads to a contradiction for the fact that the teacher was in the classroom at the beginning.

What is induction?

1. A mathematical proof technique. It is essentially used to prove that a property $P(n)$ holds for every natural number n .
2. The method of induction requires two cases to be proved:

- (a) The first case, called the base case, proves that the property holds for the first element.
- (b) The second case, called the induction step, proves that if the property holds for one natural number, then it holds for the next natural number.

3. The domino metaphor.

The two types of induction, their steps, and why it makes sense (Strong vs Weak) - Emphasize the change in the induction step.

Example 1.6.1 (Weak induction). *Prove that $\forall n \in \mathbb{N}, \sum_{i=0}^n i = \frac{n(n+1)}{2}$.*

Proof. Base: For $n = 1$, $\sum_{i=0}^1 1 = 1 = \frac{(1+1) \cdot 1}{2}$. Assumption: Assume that the claim holds for n . Step:

$$\begin{aligned} \sum_{i=0}^{n+1} i &= \left(\sum_{i=0}^n i \right) + n + 1 = \frac{n(n+1)}{2} + n + 1 \\ &= \frac{n(n+1) + 2 \cdot (n+1)}{2} = \frac{(n+1)(n+2)}{2} \end{aligned}$$

□

Example 1.6.2 (Weak induction.). *Let $q \in \mathbb{R}/\{1\}$, consider the geometric series $1, q, q^2, q^3, \dots, q^k, \dots$. Prove that the sum of the first k elements is*

$$1 + q + q^2 + \dots + q^{k-1} + q^k = \frac{q^{k+1} - 1}{q - 1}$$

Proof. Base: For $n = 1$, we get $\frac{q^{k+1}-1}{q-1} = \frac{q-1}{q-1} = 1$. Assumption: Assume that the claim holds for k . then: Step:

$$\begin{aligned} 1 + q + q^2 + \dots + q^{k-1} + q^k + q^{k+1} &= \frac{q^k - 1}{q - 1} + q^{k+1} = \frac{q^{k+1} - 1 + q^{k+1}(q - 1)}{q - 1} = \\ &= \frac{\textcolor{red}{q}^{k+1} - 1 + q^{k+2} - \textcolor{red}{q}^{k+1}}{q - 1} = \frac{q^{k+2} - 1}{q - 1} \end{aligned}$$

□

Example 1.6.3 (Strong induction). *Let there be a chocolate bar that consists of n square chocolate blocks. Then it takes exactly $n - 1$ snaps to separate it into the n squares no matter how we split it.*

Proof. By strong induction. Base: For $n = 1$, it is clear that we need 0 snaps. Assumption: Assume that for **every** $m < n$, this claim holds.

Step: We have in our hand the given chocolate bar with n square chocolate blocks. Then we may snap it anywhere we like, to get two new chocolate bars: one with some $k \in [n]$ chocolate blocks and one with $n - k$ chocolate blocks. From the induction assumption, we know that it takes $k - 1$ snaps to separate the

first bar, and $n - k - 1$ snaps for the second one. And to sum them up, we got exactly

$$(k - 1) + (n - k - 1) + 1 = n - 1$$

snaps.

□