Chapter 1

Induction and Asymptotic Notations.

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 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 do 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.

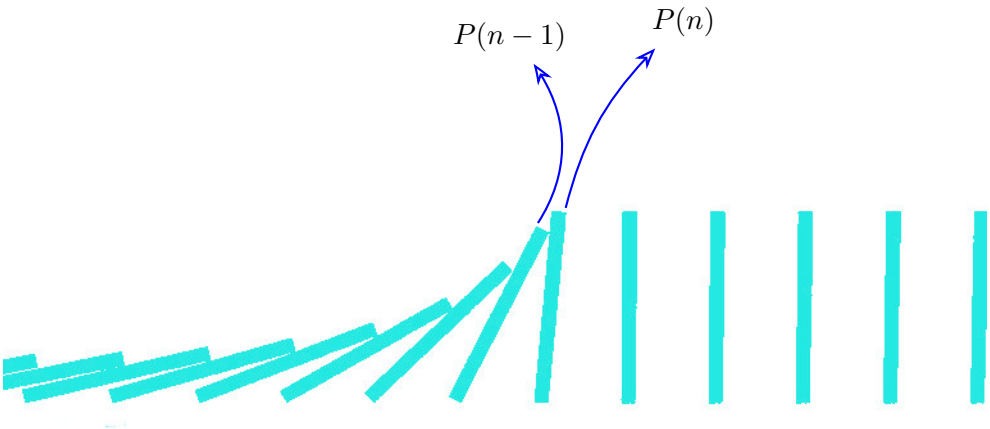

Figure 1.1: domino line falling.

1.1.1 Two Types of Induction Strategies.

We keep in mind two types of induction strategies: the first is Weak Induction, which refers to proofs that, in their step stage, only require assuming the correctness of the previous step, i.e. $P(n-1)$, in order to prove correctness for $P(n)$. This differs from the Strong Induction strategy, for which the step stage assumes the correctness of the claim for any value less than n, namely assuming that all $P(1), P(2), ..., P(n-1)$ are correct.

Example 1.1.3 and Example 1.1.2 demonstrate the use of weak induction to prove the formulas for arithmetic and geometric sums. Example 1.1.1 uses strong induction to determine the number of splits required to separate a chocolate bar.

Example 1.1.1 (Weak induction). Prove that $\forall n \in \mathbb{N}$:

$$\sum_{i=0}^{n} i = \frac{n(n+1)}{2}$$

Proof. By induction.

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

3. Step.

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

□

Example 1.1.2 (Weak induction.). Let $q \in \mathbb{R}^+/\{1\}$, consider the geometric series $1, q, q^2, q^3 ....q^k$. Prove that the sum of the first k elements is

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

Proof. By induction.

1. Base. For $n = 1$, we get $\frac{q^{k+1}-1}{q-1} = \frac{q-1}{q-1} = 1$.
2. Assumption. Assume that the claim holds for an integer k.
3. Step.

$$1 + q + q^2 + ... + q^{k-1} + q^k + q^{k+1} = \frac{q^k - 1}{q - 1} + q^{k+1}$$
$$= \frac{q^{k+1} - 1 + q^{k+1}(q-1)}{q-1}$$
$$= \frac{q^{k+1} - 1 + q^{k+2} - q^{k+1}}{q-1}$$
$$= \frac{q^{k+2} - 1}{q-1}$$

□

Example 1.1.3 (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.

1. Base. For $n = 1$, it is clear that we need 0 snaps.
2. Assumption. Assume correctness for **every** $m < n$.
3. 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.

□

1.2 Asymptotic Notations.

Definition 1.2.1. Let $f, g : \mathbb{N} \to \mathbb{R}^+$. We say that $f = O(g)$ if there exists $N \in \mathbb{N}$ and $c > 0$ such that for all $n \geq N$ we have $f(n) \leq c \cdot g(n)$.

Example 1.2.1. For example, if $f(n) = n + 10$ and $g(n) = n^2$, then $f = O(g)$ (Draw the graphs) for $n \geq 5$: $f(n) = n + 10 \leq n + 2n = 3n \leq n \cdot n = n^2$

Example 1.2.2. Also if $f(n) = 5n$ and $g(n) = n^2$, then $f(n) = O(g(n))$

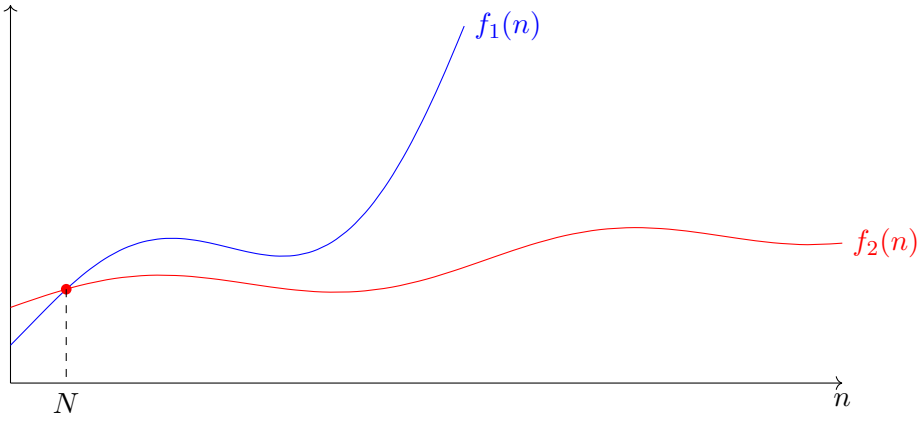

Figure 1.2: Here $f_1(x) \sim x^2 + \sin(x)$, and $f_2 \sim x + \sin(x)$, both defined up to additive and multiplicative terms. So, $f_1 = \Omega(f_2)$ and $f_2 = O(f_1)$.

Definition 1.2.2. Let $f, g : \mathbb{N} \to \mathbb{R}$ We say that $f = \Omega(g)$ if $g = O(f)$, equivalently, there exist $N \in \mathbb{N}$ and $c > 0$ s.t. $\forall n \geq N \Rightarrow cg(n) \leq f(n)$

Example 1.2.3. For example, if $f(n) = n + 10$ and $g(n) = n^2$, then $g = \Omega(f)$

Definition 1.2.3. Let $f, g : \mathbb{N} \to \mathbb{R}$, We say that $f = \Omega(g)$ if: there exist $N \in \mathbb{N}$ and $\exists c > 0$ s.t $\forall n \geq N \Rightarrow f(n) \geq c \cdot g(n)$.

Definition 1.2.4. Let $f, g : \mathbb{N} \to \mathbb{R}$, We say that $f = \Theta(g)$ if: $f = O(g)$ and $f = \Omega(g)$ That is, we say that $f = \Theta(g)$ if: $\exists N \in \mathbb{N}, \exists c_1, c_2 > 0$ s.t. $\forall n \geq N$ $c_1 \cdot g(n) \leq f(n) \leq c_2 \cdot g(n)$

Example 1.2.4. For every $f : \mathbb{N} \to \mathbb{R}$, $f(n) = \Theta(f(n))$

Example 1.2.5. If $p(n) = n^5$ and $q(n) = 0.5n^5 + n$, then $p(n) = \Theta(q(n))$

But why is this example true? This next Lemma helps for intuition:

Lemma 1.2.1. $\lim_{n\to\infty} \frac{f(n)}{g(n)} < \infty \Rightarrow f(n) = O(g(n))$

Proof. Assume that $l = \lim_{n\to\infty} \frac{f(n)}{g(n)} < \infty$. Then for some $N \in \mathbb{N}$ we have that for all $n \geq N$: $\frac{f(n)}{g(n)} < l + 1 \Rightarrow f(n) < (l + 1)g(n)$ Which is exactly what we wanted. □

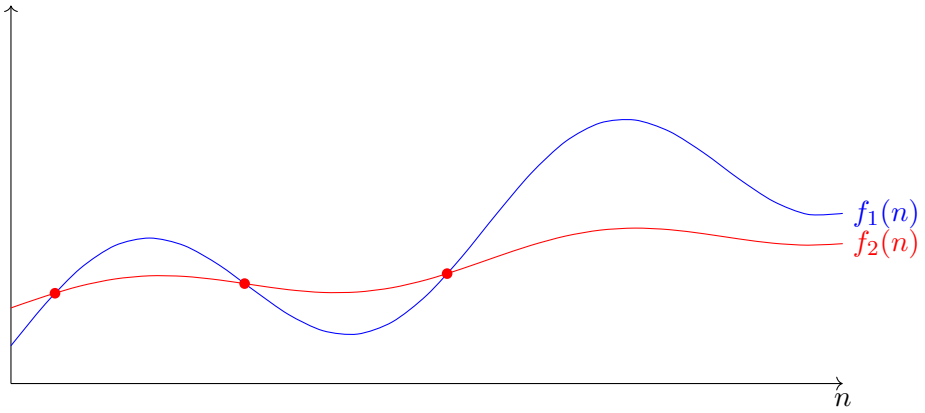

Figure 1.3: Here both $f_1(x)$ and $f_2(x)$ behave like $\sim x + \sin(x)$ up to additive and multiplicative terms. So, $f_1 = \Theta(f_2)$.

1.3 Examples with proofs.

Claim 1.3.1. $n = O(2^n)$

(This must seem very silly, but even though we have a strong feeling it's true, we still need to learn how to PROVE it)

Proof. We will prove by induction that $\forall n \geq 1$, $2^n \geq n$, and that will suffice.

1. Base. $n = 1$, so it is clear that: $n = 1 < 2 = 2^n$
2. Assumption. Assume that $n < 2^n$ for some n.
3. Step. We will prove for $n + 1$. It holds that:

$$n + 1 < 2^n + 1 < 2^n + 2^n = 2^{n+1}$$

□

Claim 1.3.2. *Let $p(n)$ be a polynomial of degree d and let $q(n)$ be a polynomial of degree k. Then:*

1. $d \leq k \Rightarrow p(n) = O(q(n))$ *(set upper bound over the quotient)*
2. $d \geq k \Rightarrow p(n) = \Omega(q(n))$ *(an exercise)*
3. $d = k \Rightarrow p(n) = \Theta(g(n))$ *(an exercise)*

Proof. Proof (Of 1) First, let's write down $p(n), g(n)$ explicitly:

$$p(n) = \sum_{i=0}^{d} \alpha_i n^i, \; g(n) = \sum_{j=0}^{k} \beta_j n^j$$

Now let's manipulate their quotient:

$$\frac{p(n)}{q(n)} = \frac{\sum_{i=0}^{d} \alpha_i n^i}{\sum_{j=0}^{k} \beta_j n^j} = \frac{\sum_{i=0}^{d} \alpha_i n^i}{\sum_{j=0}^{k} \beta_j n^j} \cdot \frac{n^{k-1}}{n^{k-1}} = \frac{\sum_{i=0}^{d} \alpha_i n^{i-k+1}}{\sum_{j=0}^{k} \beta_j n^{j-k+1}} \leq$$

$$\leq \frac{\sum_{i=0}^{d} \alpha_i}{\beta_k} < \infty$$

And now we can use the lemma that we have proved earlier. □

1.4 Logarithmic Rules.

Just a quick reminder of logarithmic rules:

1. $log_a x \cdot y = log_a x + log_a y$
2. $log_a \frac{x}{y} = log_a x - log_a y$
3. $log_a x^m = m \cdot log_a x$
4. Change of basis: $\frac{log_a x}{log_a y} = log_y x$

And so we get that:

Remark 1.4.1. For every $x, a, b \in \mathbb{R}$, we have that $log_a x = \Theta(log_b x)$

Example 1.4.1. Let $f(n)$ be defined as:

$$f(n) = \begin{cases} f\left(\lfloor \frac{n}{2} \rfloor\right) + 1 & \text{for } n > 1 \\ 5 & \text{else} \end{cases}$$

Let's find an asymptotic upper bound for $f(n)$. let's guess $f(n) = O(\log(n))$.

Proof. We'll prove by strong induction that : $f(n) < c\log(n) - 1$ for $c = 8$ And that will be enough (why? This implies $f(n) = O(\log(n))$).

1. Base. $n = 2$. Clearly, $f(2) = 6 < 8$
2. Assumption. Assume that for every m ¡ n, this claim holds.
3. Step. Then we get:

$$\begin{aligned} f(n) = f\left(\lfloor \frac{n}{2} \rfloor\right) + 1 &\leq c\log\left(\lfloor \frac{n}{2} \rfloor\right) + 1 \\ &\leq c\log(n) - c\log(2) + 1 \leq c\log(n) \quad \text{for } c = 8 \end{aligned}$$

□

1.5 Peaks-Finding.

Example 1.5.1 (Leading Example.). Consider an n-length array A such that $A_1, A_2, ..., A_n \in \mathbb{R}$. We will say that A_j is a peak (local minimum) if he's greater than his neighbors. Namely, $A_i \geq A_{i\pm1}$ 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 1.5.2. 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. $A[i] = i \; \forall i \in [n]$

1.6 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. 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. either returns $j' \in [n]$ such that $A_{j'}$ is not a peak or does not return at all (never reach line (3)). Let's handle first the case in which returning indeed occurred. Denote by j the first position of a peak in A, and note that if the algorithm gets to line (2) in the jth iteration then either it returns j or A_j is not a peak. Hence it must hold that $j' < j$. But satisfaction of the condition on line (2) can happen only if $A_{j'}$ is a peak, which contradicts the minimality of j. In the case that no position has been returned, it follows that the algorithm didn't return in any of the first j iterations and gets to iteration number $j + 1$, which means that the condition on line (2) was not satisfied in contradiction to the fact that A_j is a peak. □

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, 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 1.5.2, the naive algorithm will have to check each item, so the running time adds up to at most $c \cdot n$.

1.7 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 $A_1 ... A_n \in \mathbb{R}^n$
1 **if** $A_1 \geq A[2]$ *or* $n = 1$ **then**
2 | **return** 1
3 **end**
4 **return** 1 + peak-find$(A_2, ..A_n)$

Algorithm 2: naive recursive peak-find alg.

Claim 1.7.1. *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. 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. 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. □

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

Claim 1.7.2. *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 1.7.1.

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 1.7.1 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.

1.8 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.

Let's try to 'prove' it.

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.

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, $A[i] = n - i$.

1.9 Sophisticated solution.

The example above points to the fact that we would like to have a similar claim to Claim 1.7.2 that relates the peaks of the split array to the original one. Let's prove

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.

correction by induction.

Proof.
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 $A_{\lceil n/2 \rceil}$ 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 1.7.2 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 $T(n) \leq c_1 \log(n) - c_2$, let's prove it by induction.

Proof.

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:

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

□