The Big-Oh Notation[1]

1 The Big-Oh Notation

Let us fix a function $g : \mathbb{R}_{\geq 0} \to \mathbb{R}_{\geq 0}$ with domain and co-domain being the non-negative real numbers. $O(g)$ is a *set* of functions defined by g. Every element of this set are also functions $f : D \to \mathbb{R}_{\geq 0}$ where the domain $D \subseteq \mathbb{R}_{\geq 0}$ is a subset of the non-negative real numbers. In particular, the domain D could be $\mathbb{N}$, the set of natural numbers. One should think of g as being a "nice/simple" function, like $g(n) = n$ or $g(n) = n^2$. In the study of algorithms, we would like to understand for which such nice function g, does the running time function $T(n)$ lie in $O(g)$. Let's look at the definition.

Definition 1. *A function $f : D \to \mathbb{R}_{\geq 0}$ lies in $O(g)$ if there exists two* constants $a > 0$ *and $b \geq 0$ such that for all $n \geq b$, $f(n) \leq a \cdot g(n)$.*

In plain English, it means for "large enough" n (more precisely, $n \geq b$) $f(n)$ is at most a "scaled version" of $g(n)$ (more precisely, $a \cdot g(n)$).

Remark: *Although $O(g(n))$ is a set and $f(n)$ belongs in the set, the world often abuses and says $f(n) = O(g(n))$ when it means $f(n) \in O(g(n))$. This is important to keep in mind. $O(g(n))$ is a set, and can't be, for instance, added together. Yet, we will see us adding these things. In this lecture we will make precise what these mean – these are all definitions and used for convenience.*

Often when the function $g(n)$ is "simple", then one substitutes that function in the Big-Oh. For instance, $O(n)$ is the set of functions $O(g(n))$ where $g(n) = n$.

Example 1. Let $f(n) = 10n + 4$. We claim that $f(n) \in O(n)$. In order to show this, all we need to do is exhibit two constants $a > 0, b \geq 0$ such that for all $n \geq b$, we have $10n + 4 \leq a \cdot n$.

If we set $b = 4$, then we get for all $n \geq 4$, the LHS $10n + 4 \leq 10n + n \leq 11n$. Thus, the constants $b = 4, a = 11$ certify that $10n + 4 \in O(n)$.

Definition 2. *$f(n) \in \Omega(g(n))$ if there exists two* constants $a > 0$ *and $b \geq 0$ such that for all $n \geq b$, $f(n) \geq a \cdot g(n)$.*

In plain English, it means for "large enough" n (more precisely, $n \geq b$) $f(n)$ dominates some "scaled version" of $g(n)$ (more precisely, $a \cdot g(n)$).

Theorem 1. $f(n) \in O(g(n))$ if and only if $g(n) \in \Omega(f(n))$.

Proof. Suppose $f(n) \in O(g(n))$. That is, there exists $a > 0, b \geq 0$ such that for all $n \geq b$, $f(n) \leq a \cdot g(n)$. Dividing by a both sides (since $a > 0$) and rearranging, we get that $g(n) \geq (1/a) \cdot f(n)$ for all $n \geq b$. Thus the constant $a' := 1/a$ and $b' = b$ certifies that $g(n) \in \Omega(f(n))$. The other direction is similar and is left as an exercise. □

[1] *Lecture notes by Deeparnab Chakrabarty. Last modified : 14th Jan, 2025*
These have not gone through scrutiny and may contain errors. If you find any, or have any other comments, please email me at deeparnab@dartmouth.edu. Highly appreciated!

Definition 3. $f(n) \in \Theta(g(n))$ *if there exists three* constants $b \geq 0$ *and* $a_1, a_2 > 0$ *such that for all* $n \geq b$, $a_1 \cdot g(n) \leq f(n) \leq a_2 \cdot g(n)$.

Theorem 2. $f(n) = \Theta(g(n))$ if and only if $f(n) = O(g(n))$ and $f(n) = \Omega(g(n))$.

Proof. One direction is easy, and the other is easier. The easier direction is given $f(n) = \Theta(g(n))$, we directly get $f(n) = O(g(n))$ (certified by b, a_2) and $f(n) = \Omega(g(n))$ (certified by b, a_1).

The easy direction follows thus: since $f(n) = O(g(n))$, there exists constants $b_2 \geq 0, a_2 > 0$ such that for all $n \geq b_2$, $f(n) \leq a_2 \cdot g(n)$. Similarly, $f(n) = \Omega(g(n))$ implies constants $b_1 \geq 0, a_1 > 0$ such that for all $n \geq b_1$, $f(n) \geq a_1 \cdot g(n)$. But this means for all $n \geq b := \max(b_1, b_2)$, we have $a_1 \cdot g(n) \leq f(n) \leq a_2 \cdot g(n)$. □

It is sometimes important to show when some functions $f(n)$ ***are not*** $O(g(n))$ of some other function.

Example 2 (Counterexample)**.** If $f(n) = n$ and $g(n) = \frac{n^2}{3} - 7$, then $g(n) \notin O(f(n))$ or as is more commonly stated, $g(n) \neq O(f(n))$. To prove this, let us suppose for contradiction's sake that $g(n) = O(f(n))$. Therefore, there exists constants $a > 0$, $b \geq 0$ such that for all $n \geq b$, we have $\frac{n^2}{3} - 7 \leq a \cdot n$. But consider $n := \max(3(a+1), 7, b) + 1$. Then, we get

$$\begin{aligned} \frac{n^2}{3} = n \cdot \frac{n}{3} &> n \cdot \frac{3(a+1)}{3} && \text{since } n > 3(a+1) \\ = n \cdot (a+1) &= an + n > an + 7 && \text{since } n > 7 \end{aligned}$$

which in turn implies $\frac{n^2}{3} - 7 > an$ which is a contradiction, since $n > b$ as well.

Example 3 (Counterexamples)**.** (Here is a common thing which confuses many students). $4^n \neq O(2^n)$. Although $2n = O(n)$, when the n is in the exponent the constant next to n matters. That is, $2^{2n} \neq O(2^n)$. The reason is actually simple: 2^{2n} is not twice of 2^n but *square* of 2^n; and squaring takes us to a different order of magnitude.

To prove formally, again we use contradiction. Suppose not, and say $4^n = O(2^n)$. Then there exists $a > 0, b \geq 0$ such that for all $n \geq b$, $4^n \leq a \cdot 2^n$. Firstly, we may assume that $a \geq 1$ since $4^n \geq 2^n$ for $n \geq 0$ (2^n is an increasing function and so $2^n \geq 1$ for all $n \geq 0$ and $4^n = (2^n)^2$.)

Now choose $n := \max(b, \log_2 a) + 1$. Then, we must have $4^n = 2^n \cdot 2^n > a \cdot 2^n$, where the inequality follows from the fact that $n > \log_2 a$. Thus we get a contradiction.

Why are these notions useful? As mentioned in the first lecture, in the analysis of algorithms, we are interested in understanding the running times of algorithms. In particular, we wish to understand the function $T(n)$ with respect to the size parameter n. As a first cut, we are interested in the *asymptotic* behavior of $T(n)$, that is, the behavior of $T(n)$ as n grows larger and larger. The Big-Oh notation is perfect to capture this: we don't care about how T behaves in the small n regime, and we also don't care about the scaling factor of T. The latter is relevant in running time analysis since scaling boils down to "change of units". And what unit to use is not obvious. Should time be measured in number of BIT-ADDs? Should it be measured in number of comparisons? Or Wall Clock time? The Big-Oh notation tries to capture the "big-picture"[2]. In Section 3, we see some examples.

[2]This is not to say the small picture is not important. When we really want performance in our implementation of an algorithm, the constants are game-changers. Nevertheless, the really good algorithms often have wall clock times great too. Of course there are exceptions, and you should always implement your algorithm to check which is the case.

Remark: *An algorithm $\mathcal{A}$ is a **polynomial time** algorithm if there exists a constant $k \geq 1$ such that $T_{\mathcal{A}}(n) = O(n^k)$. If $k = 1$, the algorithm is a linear-time, or sometimes simply linear, algorithm. If $k = 2$, then the algorithm is a quadratic-time, or simply quadratic, algorithm.*

Useful Facts. The next few theorems are going to establish some useful properties of the Big-Oh notation. Make sure you understand the theorem statements (as we will be using them willy-nilly); indeed, in the first reading feel free to skip the proofs (but please do return back to them).

Theorem 3 (Some operations involving the notations)**.** Below all functions take non-negative values.

(a) If $f(n) \in O(g(n))$ and $g(n) \in O(h(n))$, then $f(n) \in O(h(n))$ as well.

(b) If $f(n) \in \Theta(g(n))$ and $g(n) \in \Theta(h(n))$, then $f(n) \in \Theta(h(n))$ as well.

(c) If $f(n) \in O(g_1(n))$ and $h(n) \in O(g_2(n))$, then $f(n) + h(n) \in O(g_1(n) + g_2(n))$.

(d) If $f(n) \in O(g_1(n))$ and $h(n) \in O(g_2(n))$, then $f(n) \cdot h(n) \in O(g_1(n) \cdot g_2(n))$.

(e) Let $f(n) \in O(g(n))$, and $h()$ is a *non-decreasing, non-negative, invertible* function of n. Recall composition of functions: $(f \circ h)(n) := f(h(n))$. Then, $(f \circ h)(n) \in O((g \circ h)(n))$.

Proof.

a. $f(n) = O(g(n))$ means there exists constants $a > 0, b \geq 0$ such that $f(n) \leq a \cdot g(n)$ for all $n \geq b$. $g(n) = O(h(n))$ means there exists constants $a' > 0, b' \geq 0$ such that $g(n) \leq a' \cdot h(n)$ for all $n \geq b'$. Thus, for all $n \geq \max(b, b')$, we have $f(n) \leq (a \cdot a') \cdot h(n)$. This proves that $f(n) = O(h(n))$.

b. Very similar proof as above, except there are three variables a_1, a_2, b to play with. Please do this yourself (recommended).

c. As before, there exists $a > 0, b \geq 0$ and $a' > 0, b' \geq 0$ such that for all $n \geq b$, we have $f(n) \leq a \cdot g_1(n)$ and for all $n \geq b'$ we have $h(n) \leq a' \cdot g_2(n)$. That is, for all $n \geq \max(b, b')$, we have $s(n) \leq a_1 g_1(n) + a_2 g_2(n) \leq \max(a_1, a_2)\,(g_1(n) + g_2(n))$. Thus, $s(n) = O(g(n))$.

d. As before, there exists $a, b \geq 0$ and $a', b' \geq 0$ such that for all $n \geq b$, we have $f(n) \leq a \cdot g_1(n)$ and for all $n \geq b'$ we have $h(n) \leq a' \cdot g_2(n)$. Therefore, for all $n \geq \max(b, b')$, we get $f(n) \cdot h(n) \leq (a \cdot a') \cdot (g_1(n) \cdot g_2(n))$.

e. There exists $a > 0, b \geq 0$ such that for all $n \geq b$, we have $f(n) \leq a \cdot g(n)$. Now since h is increasing, for all $n \geq h^{-1}(b)$ we have $h(n) \geq h(h^{-1}(b)) = b$. Therefore, For all $n \geq h^{-1}(b)$, we get $(f \circ h)(n) = f(h(n)) \leq a \cdot g(h(n)) = (g \circ h)(n)$. □

Remark: *Part (c) implies we can take the "Big Oh out of the summation". That is, if we have $F := \sum_{j=1}^{n} f_j$ and each $f_j(n) \in O(g_j(n))$ for some "nice function g_j", then $F \in O\left(\sum_{j=1}^{n} f_j\right)$ as well. It is important to be wary that we don't subtract. That is, if $f_1 \in O(g_1)$ and $f_2 \in O(g_2)$, then it is not necessarily true that $f_1 - f_2 \in O(g_1 - g_2)$. For instance, if $f_1(n) = 2n$ and $f_2(n) = n$, then both*

f_1 and f_2 are in $O(n)$, but $f_1 - f_2$ is not in $O(0)$ (which contains only the zero function).

The following theorem is very useful to argue about the Big-Oh relations for large classes of functions.

Theorem 4. Suppose $L := \lim_{n\to\infty} \left(\frac{f(n)}{g(n)}\right)$ exists. Then,

$$L < \infty \Leftrightarrow f(n) \in O(g(n)) \quad \text{and} \quad L > 0 \Leftrightarrow f(n) \in \Omega(g(n))$$

In particular, $0 < L < \infty \Leftrightarrow f(n) \in \Theta(g(n))$.

Proof. Let me prove the first statement and leave the second's analogous proof as an exercise – please, try it, it requires one extra idea. Suppose $L < \infty$. This means for any $\varepsilon > 0$, there is a number b_ε such that for all $n \geq b_\varepsilon$, we have

$$\left|\left(\frac{f(n)}{g(n)}\right) - L\right| < \varepsilon$$

In particular, we get for all $n > b_\varepsilon$, $\frac{f(n)}{g(n)} \leq L + \varepsilon$. That is, $f(n) \leq (L+\varepsilon)g(n)$. Choose $\varepsilon = 1$, $b := b_1$, $a := L + 1$ to get that for all $n \geq b$, $f(n) \leq a \cdot g(n)$. Note a, b are constants independent of n, and thus, $f(n) \in O(g(n))$.

On the other hand, if $f(n) = O(g(n))$, then there exists constants $a > 0, b \geq 0$ such that for all $n \geq b$, $f(n) \leq a \cdot g(n)$. That is, $\frac{f(n)}{g(n)} \leq a$. Therefore, if $\lim_{n\to\infty}(f(n)/g(n))$ exists, then it must be $L \leq a < \infty$. □

Let us illustrate the power of the above theorem. Take $f(n) = 4^n$ and $g(n) = 2^n$ and look at the ratio. We get $f(n)/g(n) = 4^n/2^n = 2^n$ which goes to ∞ as $n \to \infty$. Thus, $4^n \notin O(2^n)$. The above two theorems help a lot often. ***But be wary when limits don't exist.*** In particular, when asked to prove from ***first principles***, you cannot use the limit theorem.

As a corollary, we get the following useful fact: polynomials are in the Big-Oh class of the leading monomial (highest degree). I'll leave the proof as an exercise; in fact, try proving it using first principles as well.

Theorem 5. Suppose $p(n)$ is a degree d *polynomial*, that is, $p(n) = a_d n^d + a_{d-1}n^{d-1} + \cdots + a_1 n + a_0$. Then, $p(n) \in O(n^d)$.

2 Fun with Factorials and Logarithms

Recall the factorial function defined as $n! = 1 \cdot 2 \cdot 3 \cdots n$. What is its relation with n^n?

Theorem 6. $n! \in O(n^n)$ but not in $\Omega(n^n)$.

Proof. Let's look at $n!/n^n$. This equals $(1/n) \cdot (2/n) \cdots (n/n)$. These terms are all positive. Further, they are all ≤ 1 and thus, in particular, $n!/n^n < 1/n$, that is, it is less than the first term. Taking limits, we get $\lim_{n\to\infty}(n!/n^n)$ is 0. □

Theorem 7. $\log_2(n!) = \Theta(n \log_2 n)$.

Although $n! \neq \Theta(n^n)$, taking logs satisfies the $\Theta()$ relation. Again, this should not be surprising – we have seen some examples of functions such that $f(n) = \Theta(g(n))$ but $2^{f(n)} \neq O(2^{g(n)})$.

Proof. Let's now prove the claim. Once again, let's look at the rational function $r(n) := \frac{\log_2(n!)}{n \log_2 n}$.

First we note that the numerator is precisely (using log of product is sum of logs)

$$\log_2(n!) = \log_2(1 \times 2 \times \cdots \times n) = \log_2 1 + \log_2 2 + \cdots + \log_2 n = \sum_{i=1}^{n} \log_2 i$$

Now, since $\log_2 i \leq \log_2 n$ for all $i \leq n$ (since $\log_2$ is an increasing function), we get that the numerator of $r(n)$ is $\leq n \log_2 n$, which is the denominator of $r(n)$. This implies $r(n) \leq 1$ for all n. And in particular, $\lim_{n\to\infty} r(n) \leq 1$.

We now need to show that $\lim_{n\to\infty} r(n) >$ some positive number. This will show $\log_2(n!) \in \Theta(n \log_2 n)$. To see this, we note that for all $i \geq \lceil n/2 \rceil$, $\log_2 i \geq \log_2(n/2)$. Therefore, we get that the numerator of $r(n)$ is

$$\sum_{i=1}^{n} \log_2 i = \log_2 1 + \log_2 2 + \cdots + \log_2 n \geq \log_2(n/2) + \log_2(n/2+1) + \cdots + \log_2 n \geq \lfloor n/2 \rfloor \cdot \log_2(n/2)$$

Dividing by $n \log_2 n$, we get

$$r(n) \geq \frac{\lfloor n/2 \rfloor}{n} \cdot \frac{\log_2 n - 1}{\log_2 n}$$

As $n \to \infty$, the first fraction $\to \frac{1}{2}$ and the second fraction $\to 1$. Thus, $\lim_{n\to\infty} r(n) \geq 1/2$. □

Recall that given any natural number N, the number of bits required to describe it is $\lceil \log_2(N+1) \rceil$. In your problem set, you will show that $\lceil x \rceil = \Theta(x)$. Therefore, one requires $\Theta(\log_2 N)$ bits to describe a natural number N. The above theorem, therefore, gives the following corollary.

Theorem 8. For any number n, the number of bits required to write $n!$ is $\Theta(n \log_2 n)$.

We end with two more definitions: "little-oh" and "little-omega" notation. We will probably not use this notation much in CS, but it's good to know what they mean in case you see it somewhere.

Definition 4. *We say that* $f(n) = o(g(n))$ *if* $\lim_{n\to\infty} \left(\frac{f(n)}{g(n)}\right) = 0$. *We say that* $f(n) = \omega(g(n))$ *if* $\lim_{n\to\infty} \left(\frac{f(n)}{g(n)}\right) = \infty$.

The "little-oh" notation is often used to signify "very small errors". For instance, the *prime number theorem* says that the function $\pi(x)$ defined as the number of prime numbers $\leq x$ satisfies

$$\pi(x) = \frac{x}{\ln x} + o\left(\frac{x}{\ln x}\right)$$

Note that the first term on the RHS has no "Big-Oh"; it's precisely the function x divided by the natural logarithm (base e) of x. The second term is the error and it says that compared to $x/\ln x$, it is "much much

smaller". In particular, it shows that $x/\ln x$ is a very good approximation for the number of primes less than x, especially when x is "large".

Another example is the *harmonic sum*. For any natural number n, the nth Harmonic number is defined to be $H_n := 1 + \frac{1}{2} + \cdots + \frac{1}{n}$. It can be shown that

$$H_n = \ln n + \gamma + o(1)$$

where γ is a constant called the Euler-Mascheroni constant and is ≈ 0.577. The $o(1)$ means that it's a function of n which goes to 0 as $n \to \infty$.

3 Analyzing Algorithms with Big-Oh Notation

Consider the example we looked at in the last lecture.

```
1: procedure MAX(A[1 : n]):
2:     ▷ Returns j where A[j] ≥ A[i] for all 1 ≤ i ≤ n.
3:     rmax ← 1. ▷ Initialize running-max to 1
4:     for 2 ≤ j ≤ n do:
5:         if A[j] > A[rmax] then: ▷ If A[j] bigger than running-max, swap.
6:             rmax ← j
7:     return rmax.
```

Unspecified Constants. The size of the input was n, the number of elements in the array. To analyze the running time, we need to first figure how much time does each line take. Well, this question is *ill-defined* since we do not know the exact implementation of the algorithm. However, we will assume *certain atomic steps take some unspecified* ***constant*** *time*. That is, these steps are steps whose processing times do not depend on n at all. Since we do not know these constants, we will use $O(1)$ to denote their running time.

What does $O(1)$ mean? It is the set $O(g)$ where $g : \mathbb{R}_{\geq 0} \to \mathbb{R}_{\geq 0}$ is the *constant* function $g(x) = 1$ for all x. This set consists of all functions $f : D \to \mathbb{R}_{\geq 0}$ such that there exists constants $a > 0$, $b \geq 0$ s.t. for all $n \geq b$, we get $f(n) \leq a$. Now suppose $B := \max_{n \leq b} f(n)$ is the maximum value this function takes in the range $-0, b]$. This B is some fixed ***constant*** (could be a bazillion, we don't care); it does not grow with n. Therefore, combining the above two, we see that $f(n) \leq \max(a, B)$ for all n. Implying $f(n)$ is at most some *unspecified* constant.

Remark: *The running time of an algorithm on an input of size a constant (no matter how large the constant is) is going to be $O(1)$.*

Now we can answer what $T_{\text{MAX}}(n)$ is for the above algorithm. We already know there are at most $(n-1)$ for-loops, and in each for-loop, Lines 5 and 6 both take $O(1)$ time. Thus, the total time spent on the for-loops is $O(n)$ (Theorem 3.(c)). The time taken in Lines 3 and 7 is also $O(1)$. Thus, the total time is $O(n)$.

Let us look at one other example.

```
1: procedure FOO(A[1 : n]):
2:     for 1 ≤ i ≤ n do:
3:         for i ≤ j ≤ n do:
4:             if A[j] < A[i] then:
5:                 tmp ← A[i]
6:                 A[i] ← A[j]
7:                 A[j] ← tmp
8:     return A
```

✍

Exercise: *Before reading ahead, what do you think the above pseudocode does? Note how the absence of comments makes reading unpleasant.*

Let us analyze the running time (we don't need to know what the algorithm does to figure out the running time). Lines 5 to 7 each take $O(1)$ time. The total time taken in the loops is therefore Big-Oh of the number of loops. How many loops are there? There is an outer loop and there is an inner loop. The outer loop has n iterations. The ith outer loop, however leads to $(n - i + 1)$ inner loops. Therefore, the total number of times Lines 5 to 7 is executed is $\sum_{i=1}^{n}(n - i + 1)$. Now you can use your discrete math knowledge to prove that this is $O(n^2)$.

Remark: *Please do not ever say/write/thing something like "the running time is $O(n^2)$ because there are two for-loops." To appreciate this, analyze the running time of the above algorithm where Line 3 is replaced by "***for** $2^i \leq j \leq n$*:". The number of loops will become $\sum_{i=1}^{n} \max(0, n - 2^i + 1)$. This is* **not** *$O(n^2)$. In fact, it is $O(n \log n)$ — can you see why?*

Coming back to FOO, the total time in the for-loops is $O(n^2)$. Line 8 can be assumed to be $O(1)$ (if only pointer returned), or $O(n)$ (if all the entries of array returned). Doesn't really matter, since $O(n^2)$ plus this "small change" is still $O(n^2)$.

4 Abuse of Notation: what does it mean?

As if the abuse $f(n) = O(g(n))$ instead of $f(n) \in O(g(n))$ was not enough, in this course we will see statements of the form

$$\text{For every } n > 0, \quad f(n) \leq g(n) + O(h(n)) \tag{1}$$

Again recall, $O(h(n))$ is a *set* of functions. What does it mean to "add a set"!? In fact, this is most prevalent in recurrence inequalities. For instance, we see recurrence inequalities of the form

$$T(1) = O(1), \quad \text{and for all } n \geq 2,\ T(n) \leq T(n-1) + O(n) \tag{2}$$

What do these mean?

So, whenever you see a recurrence as above, what it really means is that that *there exists* a function $e(n)$, such that (a) for all $n \geq 2$, $T(n) \leq g(n) + e(n)$, and (b) $e(n) \in O(n)$.

Let us unpack what $e(n) \in O(n)$ means. It means that there exists positive constants a, b such that for all $n \geq b$, $e(n) \leq a \cdot n$. We are now going to simplify our lives a bit and see that we can actually state there exists a constant A such that

$$\forall n \geq 2, \quad e(n) \leq A \cdot n$$

How can we say this? Well, if $n \geq b$, we get this with $A = a$. Let $B = \max_{n<b} e(n)$; this is some constant. Note, that for all $2 \leq n \leq b$, $e(n) \leq B \leq B \cdot n$. Therefore, $A = \max(a, B)$ satisfies our needs.

Putting all together, the recurrence (2) actually means the existence of a constant A, B such that

$$T(1) \leq B, \quad \text{and} \quad T(n) \leq T(n-1) + A \cdot n, \quad \text{for all } n \geq 2$$

The reason for this abuse is brevity – writing out these constants and keeping track of them is tedious. One of the nice things about the Big Oh notation is the fact that we can express things succinctly. Of course brevity comes at a cost – awareness! Be wary, especially when you are solving recurrence inequalities with Big-Oh's and Theta's floating about. Let me point out a mistake which you should never make.

Theorem 9. Warning: This is a **wrong** theorem followed by an **erroneous** proof.
Consider the following recurrence inequality:

$$T(1) = O(1); \quad \forall n > 1, \quad T(n) \leq T(n-1) + O(1)$$

The solution to the above is $T(n) = O(1)$.

Wrong Proof of Wrong Theorem 9. We proceed by induction. The predicate $P(n)$ is true if $T(n) = O(1)$

Base Case: When $n = 1$, $T(n) = O(1)$. This is given to us.

We fix a $k \geq 1$ and assume $P(1), \ldots, P(k)$ are all true. That is, $T(m) = O(1)$ for all $m \leq k$. We need to show $P(k+1)$ is true. That is, we need to show $T(k+1) = O(1)$.

Well, $T(k+1) \leq T(k) + O(1)$. Since $P(k)$ is true, we get $T(k) = O(1)$. Therefore, $T(k+1) = O(1) + O(1)$. But adding a constant with another constant, is again a constant. So, $T(k+1) = O(1)$. Hence proved. □

The above theorem is wrong, because we know what the recurrence really means there exists some constants A, B such that

$$T(n) \leq \begin{cases} B & \text{if } n = 1 \\ T(n-1) + A & \text{if } n \geq 2 \end{cases}$$

what the solution to $T(n)$ is by the "opening up brackets" or the "kitty collection" method. For any $n \geq 2$,

$$\begin{aligned} T(n) &\leq T(n-1) + A \\ &\leq T(n-2) + A + A \\ &\vdots \\ &= T(1) + A(n-1) \leq An + (B - A) \quad = O(n) \end{aligned}$$

Where was the bug in the above induction proof? Whenever you see a wrong proof, you must find out where were it is wrong. And the mistake crept in due to the abuse of notation. Consider what the Theorem 9 is asserting : it says there is an (unspecified) constant $C \geq 0$ such that $T(n) \leq C$ for all n. When we try to apply induction, we must ***first agree upon this constant***. We don't know what it is, but it exists, and we stick with it. Base Case: $T(1) \leq C$ – fine. Induction Hypothesis: for all $1 \leq n < m$, we have $T(n) \leq C$ – sure. Now we need to prove $T(m) \leq C$ as well. What do we know – we know that $T(m) \leq T(m-1) + \Theta(1)$. Again what does this mean? It means there is some other unspecified constant $C' \geq 0$ such that $T(m) \leq C + C'$. But now we are in trouble – we can't say $T(m) \leq C$ unless $C' = 0$, and that cannot be assumed.

The Correct Induction Proof of the Correct Theorem. First agree upon the unspecified constant in the *recurrence*. So, $T(1) \leq C$ and $T(m) \leq T(m-1) + C$ for all m. Then assert (correctly) that $T(n) \leq Cn$ for all n. Base case: $T(1) \leq C$ is correct. The Induction hypothesis holds for all $1 \leq n < m$ and we need to show that it holds for $T(m)$. Well, $T(m) \leq T(m-1) + C \leq C(m-1) + C = Cm$. Therefore, $T(n) \leq Cn$ for all n, implying $T(n) = O(n)$. □