Recitation Week 2 - Complexity and Correctness

Abstract

In this course (and many more), we will encounter algorithms. As you've seen in Intro, a single task can have many different solutions and implementations. One of the things that differ between solutions is their **running time**, or **time complexity**. We continue here the discussion you had in class on this subject. We also add the notion of algorithm's **correctness**, as we would like to analyse only algorithms that solve the given problem for any given input, and show how we can prove this holds for a specific algorithm.

1 Asymptotic notations - O, Ω, Θ

Definition 1.1. ***(Reminder)*** *Let $f, g : \mathbb{N} \to \mathbb{R}^+$, We define:*

- $f(n) = O(g(n)) \iff \exists C > 0 \quad \exists n_0 \in \mathbb{N} \quad \forall n \geq n_0 \quad f(n) \leq Cg(n)$
- $f(n) = \Omega(g(n)) \iff \exists c > 0 \quad \exists n_0 \in \mathbb{N} \quad \forall n \geq n_0 \quad cg(n) \leq f(n)$
- $f(n) = \Theta(g(n)) \iff f(n) = O(g(n)) \land f(n) = \Omega(g(n))$

Draw schematic examples. For Θ preferably of a function that "goes nuts" betweet two lines, for example.

In essence, Big O notation implies that "eventually, g is larger that f". Similarly, Big Ω notation implies that "eventually, g is smaller than f". Another way to put it is that **asymptotically**, g is an upper bound for f (for O, lower bound for Ω). For Θ, it describes asymptotic **similarity**, in the sense that f "behaves like" g for large values of n. We will see some formalization of this idea later today.

1.1 Properties

General properties

Claim 1.2. *(Reflexive)*
For any f, $f(n) = O(f(n))$.

Proof. Choose $n_0 = 1, C = 1$, for any $n > n_0$ we have $f(n) \leq f(n)$. □

We said last week this also holds for Θ. Does it also hold for Ω? (*was noted in class*)

Claim 1.3. *(Antisymmetric)*
For any f, g, $f = O(g)$ iff $g = \Omega(f)$.

Remark 1.4. This is not exactly the same as an antisymmetric relation as you saw in Discrete Math! $f = O(g)$ and $g = O(f)$ does not imply $f = g$.

Proof. $\underline{f = O(g) \Rightarrow g = \Omega(f)}$: Let n_0, C with $n > n_0 \Longrightarrow f(n) \leq Cg(n)$. Choose $c = \frac{1}{C}$, thus $n > n_0 \Longrightarrow cf(n) \leq g(n)$
$\underline{g = \Omega(f) \Rightarrow f = O(g)}$: Similarly. □

Claim 1.5. *Symmetric* $f = \Theta(g) \iff g = \Theta(f)$

Proof.

$$f = \Theta(g) \iff \Big(f = O(g) \wedge f = \Omega(g)\Big) \iff \Big(g = \Omega(f) \wedge g = O(f)\Big) \iff g = \Theta(f)$$

□

Exercise 1.6. *Show that the relations O, Ω, Θ are transitive.*

(*Transitivity of O appears in Ex1*)

Remark 1.7. This justifies thinking about Θ as an equivalence relation. In the same way - $f = O(g)$ means "g is larger than f": This is an order relation (on equivalence classes).

Algebraic properties

Question 1.8. (Can do this using polls)
Let $f, g : \mathbb{N} \to \mathbb{R}^+$.
Prove or disprove the following claims:

1. $\lim_{n\to\infty} \frac{f(n)}{g(n)} = 0 \Longrightarrow g(n) = \Omega(f(n))$
2. $f(n) = \Theta\Big(f\big(\frac{n}{2}\big)\Big)$
3. $f(n) = O(g(n)) \Longrightarrow f^2(n) = O(g^2(n))$
4. $f(n) = \Theta(g(n)) \Longrightarrow 2^{f(n)} = \Theta\big(2^{g(n)}\big)$

Solution. (With possible examples)

1. True. Similar to question 1 in exercise 1.
2. False. Example - $f(n) = 2^n$
3. True. Case of a general claim $f = O(g), f' = O(g') \Longrightarrow f \cdot f' = O(g \cdot g')$.
4. False. Example - $f(n) = n, g(n) = 2n$.

Example 1.9. (Factorial)
Prove $n! = O(n^n)$

Proof. $n! = \prod_{i=1}^{n} i \leq \prod_{i=1}^{n} n = n^n$ □

We will see another proof, just to work our induction muscles:

Proof. (using induction) Choosing $C = 1$ and $n_0 = 0$, prove by induction on n that $\forall n > n_0 : n! \leq Cn^n$.

Base: for $n = 1$ this is immediate.

Assumption: Assume this holds for $n-1$, that is $(n-1)! \leq (n-1)^{n-1}$.

Step: For n, we have:

$$n! = n \cdot (n-1)! \leq n(n-1)^{n-1} \leq n \cdot n^{n-1} = n^n$$

And Bob's your uncle. □

Exercise 1.10. *(Appears in Exercise 1) If* $f(n) = O(g(n))$ *and* $\lim_{n\to\infty} f(n) = \infty$*, then* $\log(f(n)) = O(\log(g(n)))$*.*

Corollary 1.11. $\log(n!) = O(n \log(n))$

Proof. We showed that $n! = O(n^n)$, and $n! \xrightarrow{n\to\infty} \infty$, thus $\log(n!) = O(\log(n^n)) = O(n\log(n))$. □

2 Running time of algorithms

In computational tasks the difficulty lies in solving the given problem for large instances. The difficulty usually grows monotonically with the input size, so we would like to analyse it, meaning to calculate the amount of resources used by it as a **function of the input size**. To address this question we must define what resources we measure/count and the way in which we measure/count them.

Given an algorithm, we would like to know: what is the algorithm's

- Space complexity?
- Best time complexity?
- Worst time complexity?

By "space" we mean memory. Both memory and time are machine-dependent and as you saw in class, we would like to study algorithms, the abstraction of programs, that do not depend on the coding-language or the machine on which the instructions might run.

Preparatory Question. The asymptotic notations (O, Ω, Θ) are equivalence classes that "ignore" two parameters of functions:

1. behaviour on finite prefixes of inputs
2. constant scalings

As was said in class, this allows us to ignore machine-specific properties. What sort of such properties can we ignore this way? Can you think of an example showing the necessity of this approach?

An example to show the necessity of the first property being ignored can be algorithms with a constant-time pre-processing stage, like constructing several dictionaries in python. For small inputs this stage seem to make the algorithm inefficient, but its advantages become clear in larger inputs.

An example for the second property could be two machines, one on which the multiplication instruction is efficient while the addition instruction is inefficient, and another with these efficiencies reversed. To decide which of the following algorithms to use in order to double an integer depends on the machine we will use. Ignoring constant scaling these algorithms become equivalent, and we may think of any instruction as taking "one unit of time" for simplicity of analysis.

Algorithm 1 DoubleByFactor(n)

```
1: return $2 \cdot n$
```

Algorithm 2 DoubleByAddition(n)

```
1: return $n + n$
```

Example 2.1. (Bubble Sort)

Algorithm 3 *BubbleSort*$(A[0,\ldots,n-1])$

```
for $i$ from $0$ to $n-2$ do
    swapped = False
    for $j$ from $0$ to $n-2-i$ do
        if $A[j] > A[j+1]$ then
            swap$(A[j], A[j+1])$
            swapped = True
    if not swapped then
        Break
return $A$
```

Space Complexity:

The input takes a space of n.
How much **additional** space did we use? we only used i, j and a flag. This is a **constant** amount, lets say C. A constant is $O(1)$, thus the total space complexity is $S(n) = n + O(1) = O(n)$.

Best Time Complexity:

What is the "best case scenario"? If the input A is already sorted when the program starts. In this case, the **For** loop in row 3 will run n times, each time with a constant cost for each execution, so the running time will be $an + b$ for some constants a, b. This is a linear function, thus $T_{BEST}(n) = O(n)$.

Worst Time Complexity

The worst case: the array is sorted in reversed order. In that case, the **For** loop in row 1 will run n times. Within each run, the inner **For** loop will run i times, each time with a constant cost for each execution. So the total running time will be:

$$T_{WORST}(n) = cn + c(n-1) + c(n-2) + \ldots + c + d = \left(\sum_{i=1}^{n} ci\right) + d = O(n^2)$$

When we say "time complexity" we mean "worst time complexity". Normally want to guarantee an upper bound on an algorithm's running time. This is done by upper bounding (giving a big-O bound) its worst time complexity.

Now we know how to analyse the resources the algorithm takes. But we would like to analyse only **correct** algorithms. By correctness of an algorithm we mean it achieves the task at hand. In this case, that the output array is a sorted version of the input one.

We will prove this using an invariant - a property that is kept true in regard to some quantity. Here the invariant will be that the suffix (last part) of the array is sorted, and the quantity will be this suffix's length.

3. (If there is time) Explanation on Loop Invariant:

We can prove the correctness of iterative algorithms using an induction, which is proved on the iterations of the algorithm (each iteration is accounted as the “i” of the induction). This will be a key topic in your next course in Algorithms, but it is also important for our course. We can prove an algorithm's correctness using an induction, which is proved on the iterations of the algorithm (each iteration is accounted as the “i” of the induction). The computer science way to call this sort of proof is a “loop invariant”, which we will now cover more precisely.

A loop invariant is a property of a program loop that is true before (and after) each iteration. It is a logical assertion, sometimes checked within the code by an assertion call. Knowing a code's invariants is essential in understanding the effect of a loop and proving its correctness. There are three parts of a loop invariant that we need to show:

1. **Initialization**: It is true prior to the first iteration of the loop.
2. **Maintenance**: If it is true before an iteration of the loop, it remains true before the next iteration.
3. **Termination**: When the loop terminates, the invariant gives us a useful property that helps show that the algorithm is correct.

We'll see some examples:

1. **Example 1:**

```
int j = 9;
for(int i=0; i<10; i++)
j--;
```

What is the loop invariant?

i + j == 9

2. **Example 2:**

```
max = -INF (minus infinitity)
for (i = 0 to n-1)
  if (A[i] > max)
      max = A[i]
```

What is the loop invariant?

max is always maximum among the first i elements of array A.