Algorithms and Datastructures (HS2024)
Week 2

Rui Zhang

September 30, 2024

Contents

1 Revision Theory

1.1 Big-O Notation and Asymptotic Growth

1.1.1 Motivation

In computer science, we often deal with varying quantities of data that can get very large very quick. The exact computation time / runtime for a problem often depends on the size of the input data and varies from computer to computer. Thus it makes less sense to talk about the exact time an algorithm needs to complete and more sense to talk about how much the computation time increases

for every increase in input data size. As a consequence, we would like to leave out constant factors and non-dominant factors in runtime functions, as they are less relevant with increasing input size. Furthermore, we want to make our algorithm run with a certain runtime even in the **worst case** scenario. This is the intuition behind the "Asymptotic Notation" or "Big-O Notation".

1.1.2 Example

For Example, let us consider a program that takes a list of length n and goes through that list 4 times to do some operation. For each item the program operates on, we take t time. The total runtime is thus $4 * t * n$.

But as we have already said before, we want to leave out the constant factors. Intuitively, the runtime now becomes in the order of around n, as 4 is a constant factor as well as t, since t is dependent on the system the algorithm is running on.

This also makes sense if we consider the following thought: The runtime increases linearly as the input size increases - doubling the length of the input list would double the runtime. This is precisely reflected in the term n.

Formally, we write that the program's runtime is in $O(n)$.

1.1.3 Definition

The formal definition of the Big-O-Notation is as follows: Given a set of possible sizes of input N and a function f with $f : \mathbb{N} \to \mathbb{R}^+$ (which is the function which reflects the runtime of a give algorithm)

$$O(f) = \{g : N \to \mathbb{R}^+ \mid \exists C > 0 \; \forall n \in N \; g(n) \leq C * f(n)\}$$

So $O(f)$ is in reality the set of functions g which go from the set of possible input sizes to the set of positive real numbers, with the condition that there exists a constant factor C which, when multiplied with f, is always larger than or equal to g for all possible input sizes. g here also depicts the runtime of some algorithm for varying input sizes n.

Intuitively, it is the set of functions g that grow asymptotically **slower** than f.

Since we are talking about "slower", we usually use the following notation:

$$g \leq O(f)$$

instead of

$$g \in O(f)$$

1.1.4 Connection to Asymptotic Growth

We would not call it "Asymptotic Notation" if it had no connection to asymptotic growth. These two theorems from the exercises are very important.

It states that if a function g grows asymptotically faster than f, then $f \leq O(g)$. If a f grows asymptotically faster than g, then $g \leq O(f)$ and if they are asymptotically equivalent, then $g \leq O(f)$ and $f \leq O(g)$ both hold.

Figure 1: Useful theorems and connection to asymptotic growth

Theorem 1 (Theorem 1.1 from the script). *Let N be an infinite subset of $\mathbb{N}$ and $f : N \to \mathbb{R}^+$ and $g : N \to \mathbb{R}^+$.*

- *If $\lim_{n\to\infty} \frac{f(n)}{g(n)} = 0$, then $f \leq O(g)$ and $g \not\leq O(f)$.*
- *If $\lim_{n\to\infty} \frac{f(n)}{g(n)} = C \in \mathbb{R}^+$, then $f \leq O(g)$ and $g \leq O(f)$.*
- *If $\lim_{n\to\infty} \frac{f(n)}{g(n)} = \infty$, then $f \not\leq O(g)$ and $g \leq O(f)$.*

1.2 Star Search

Again, same as with Karatsuba and the Cows, it is important here to not memorize the exact algorithm, but to try and analyze the thought process and find key takeaways.

In this problem, we are given n people in a room and we want to find a so called *star*. A person is a star if and only if this person knows none of the other $n-1$ people and all of the other $n-1$ people know this person. To do so, we are allowed only to perform the following *elementary operation*: Asking one person if they know another. Note here, that the "knowing another person" relation is not symmetric, i.e. if one person p_1 knows another p_2, that does not immediately mean that p_2 knows p_1.

1.2.1 Naive Solution

The most simple way of doing this is by just asking every person whether they know each of the $n-1$ other people. Since we are asking each of the n people $n-1$ questions, that resultsin $n(n-1)$ questions in total.

1.2.2 Recursive Solution

Spoiler: this solution will not be better than the last *in the worst case*. But it reveals a very important idea that will be used again and again: *recursion* Recursion is often hard to understand in the beginning, but this example shows it quite well.

The idea is to rethink the problem of finding a star among n people in such a way, so that we can construct the solution of this problem from the solution of finding a star among $n-1$ people. We are essentially taking a problem and reducing its complexity by finding a way to construct its solution from the solution of the same problem, but for a smaller problem-size. In fact, this is exactly what we were also doing in Karatsuba's Algorithm.

The question we have to ask ourselves now is how we can construct the solution of this problem for problem-size n if we already have the solution for the same problem for $n-1$. So: **How can we find out the star among n people, if we know the star among $n-1$ people?**

The answer is to take the new person p_n that we are adding to the previous $n-1$ people, and ask them first if they know the current star. If they do not, then the current star is not the actual star of all n people. If they do, then the current star is only an actual star if they do not know the new n-th person that we added, so we ask the current star if they know this n-th person too. That is a total of 2 further questions.

If the current star is not actually the star of all n people after asking those 2 questions, then we will have to check whether the new person p_n we added is the new star. We do this by again by asking all $n-1$ people if they know the new person and if the new person knows any of the $n-1$ people. If someone does not know the new person, or the new person any of the $n-1$ people, then the new person is also not a star. This results in $2(n-1)$ further questions

To sum up, this is the outline of the algorithm:

1. find the star p_s in the group consisting of $n-1$ people (recursively)
2. add the $n-$th person to the group
3. check if p_s is still the star by asking 2 questions:
   (a) does p_s know the new person p_n?
   (b) does p_n know the star p_s?

   If (a) is answered with yes, or (b) is answered with no, then p_s cannot be the actual star of the group and then and only then do we move on with the last step:
4. If we have reached this step, then p_s is not the actual star. Then, maybe p_n is the star? We find out whether this is true by asking all $n-1$ other people if they know p_n and if p_n knows any of the other $n-1$ people. In total, those are $2(n-1)$ questions.

In essence, we start with 2 people and find out who is the star by asking two questions. We have now solved the problem for the problem size 2. We then take another person and add them to the group and proceed as above, finding out who is the star out of the 3 people. We then continue with 4, then with 5,... until we reach n people.

The worst-case runtime (worst-case number of questions we have to ask) is when everytime we add a person to the group, we realize that the current star is not a star, making us ask another $2(n-1)$ questions. As explained in the lecture, this results in $2(\frac{n(n-1)}{2}) = n(n-1)$ total asked questions. Again, since we always look at the worst-case of an algorithm, this algorithm is not much better.

1.2.3 Efficient Solution

Do not throw away the idea from the recursive solution yet! We may be able to optimize the idea further and get a solution that is actually better. What is the

problem with the recursive solution? It is obviously step (3). If we never had to execute step (3), then we would only need to ask 2 questions for the first two people (*) and for each person we add, we would need 2 further questions (**). That is $\underbrace{2}_{(*)} + \underbrace{2(n-2)}_{(**)} = 2n - 2 = 2(n-1)$ questions.

Indeed, we are able to preprocess the group of people such that out of the $n-2$ people we add one by one, there cannot be a star. That way, step 3 becomes useless, since its purpose of finding out whether the new person added is a star will always be answered with *no*.

We do this, by positioning the n people in a line, from left to right:

$$p_1, p_2, \ldots, p_n$$

Now, we begin removing person by person, starting from the right (p_n). But we do this via the following algorithm: We ask the person left of the right-most person (in this case p_{n-1})

(Q) Do you know the person to the right of you (in this case p_n)?

This single question achieves the following:

- (Q) is answered with *yes* then p_{n-1} cannot be a star!
- (Q) is anwered with *no* then p_n cannot be a star!

The best part is that no matter the answer, we can find one person, that cannot be the star by definition. This person can then be sent away and we carry on with the rest the same way. So we have one question per person sent away. We repeat until we have 2 people left ($n - 2questions$) and continue with the recursive solution. And this time, we will never execute step 3, resulting in a total of

$$n - 2 + 2(n-1) = 3n - 4$$

questions. Two things are to be taken away here:

1. *Do not throw away any ideas just because it does not seem better at first*
2. *identify weak spots of algorithms and see if you can avoid them via* ***preprocessing***

2 Exercises

2.1 Exercise Sheet 2 - Priority List

Again, top is most important, bottom is least important.

1. 2.2, 2.3 (in earlier exams, we have always had a quiz concerning O-Notation. So these two are very important. 2.3 also gives good practice from another perspective; also: Bonus.)

2. 2.4 (Induction is always good.)
3. 2.1a) (Exam level induction.)
4. 2.1b), 2.5 (very hard exercises. If you cannot get to the solution, still try to understand / remember the end result without having proven it)

3 Supplementary Exercises

3.1 Induction 1

Prove the following equation holds for all $n \in \mathbb{N}$:

$$\sum_{i=1}^{n} i^2 = \frac{n(n+1)(2n+1)}{6}$$

3.2 Induction 2

Prove the following equation holds for fixed $a, b \in \mathbb{R}$ for all $n \in \mathbb{N}$ via induction over n.

$$(a+b)^n = \sum_{i=0}^{n} \binom{n}{k} a^{n-k} b^n$$

3.3 O-Notation and Algorithms

This is an question adapted from exam FS23: For the following code snippet, derive the best-possible O-Notation for the number of times function f is called.

Algorithm 1 Random Algo

```
for i = 1...n do
    for j = 1...i^2 do
        f()
    end for
    f()
end for
```