MAT344 Lecture 7

2019/May/28

1 Announcements

2 This week

This week, we are talking about

1. The Pigeonhole principle
2. Algorithms
3. Introduction to complexity theory

3 Recap

Last time we talked about

1. Recursion
2. Induction

4 The Pigeonhole principle (Chapter 4.1 in [KT17])

If we have $n+1$ pigeons and n holes that we have to place all the pigeons in, there will be at least one hole with at least two pigeons in it. The mathematical way of saying this common sense observation is the following:

Proposition 4.1 (Proposition 4.1. in [KT17])**.** *If $f : X \to Y$ is a function and $|X| > |Y|$, then there exists an element $y \in Y$ and distinct elements $x, x' \in X$ such that $f(x) = f(x') = y$.*

While this seems like an obvious statement, it has many applications.

Exercise 4.2. *Show that at any party there are two people who have the same number of friends at the party assuming that all friendships are mutual.*

Solution: Let n be the number of people at the party. Each person can have $0, 1, \ldots, n-1$ friends present at the party. But if one person has no friends at the party, then there can't be a person who has $n-1$ friends at the party, since then they are friends with everyone at the party and friendships are mutual. So there are n people with $n-1$ possible numbers of friends at the party. By the pigeonhole principle, there are at least two people with the same number of friends present.

Exercise 4.3. *Show that if* 101 *integers are chosen from the set* $\{1, 2, \ldots, 200\}$ *then one of the chosen integers divides another.*

Solution: Let the chosen integers be $a_1, \ldots, a_{101}$. For each k, write $a_k = 2^{p_k} b_k$ for b_k odd. All of the 101 odd numbers b_k have to be from the 100 odd integers in $\{1, 3, 5, \ldots 199\}$, so by the pigeonhole principle there is m, l such that $b_m = b_l$. Either $p_m < p_l$ and therefore a_m divides a_l or $p_m > p_l$ and a_l divides a_m.

The pigeonhole principle also appears in many proofs of theorems that are far from obvious, for example:

Theorem 4.4 (Erdős-Szekeres, Theorem 4.2 in [KT17])**.** *If m and n are non-negative integers, then any sequence of $mn+1$ distinct real numbers either has an increasing subsequence of $m+1$ terms or it has a decreasing subsequence of $n+1$ terms.*

Proof. Let $\sigma = (x_1, x_2, \ldots, x_{mn+1})$ be a sequence of $mn+1$ distinct real numbers. For each $i = 1, 2, \ldots mn+1$, let a_i be the maximum number of terms in a increasing subsequence of σ with x_i as the first term. Also, let b_i be the maximum number of terms in a decreasing subsequence of σ with x_i as the last term. If some $a_i \geq m+1$ or $b_i \geq n+1$, we are done, so it remains to consider the case where $a_i \leq m$ and $b_i \leq n$ for all i. Since there are mn different ordered pairs of the form (a, b) with $1 \leq a \leq m, 1 \leq b \leq n$, by the Pigeon Hole principle that there must be integers $i_1 < i_2$ such that $(a_{i_1}, b_{i_1}) = (a_{i_2}, b_{i_2})$. Since all the numbers are distinct, either $x_{i_1} < x_{i_2}$ or $x_{i_1} > x_{i_2}$.

1. If $x_{i_1} < x_{i_2}$, we can extend any increasing subsequence starting with x_{i_2} by putting x_{i_1} at the start, so $a_{i_1} > a_{i_2}$, a contradiction.
2. If $x_{i_1} > x_{i_2}$, we can extend any decreasing subsequence ending with x_{i_1} by appending x_{i_2} at the end, so $b_{i_2} > b_{i_1}$, a contradiction.

Q.E.D.

5 Computing Fibonacci numbers

What if we wanted to somehow study how hard a computational problem is? Maybe we want to know how long it would take a computer to perform a certain computation. But what should an answer to a question like this look like? Consider computing Fibonacci numbers using the recurrence

$$F_n = F_{n-1} + F_{n-2}$$

with the initial conditions $F_0 = 0, F_1 = 1$.

You can sit down and after maybe a minute or so, (or if you make a computer do it, in less than a second) find that the 12th Fibonacci number is 144. How hard was that? The problem is, once you have the sentence:

"The 12th Fibonacci number is 144."

written down (or committed to the computer's memory), all you need to do is to look at your own writing and read the answer. If you have the answer in your hand, any problem becomes very easy.

The challenge is estimating how long it would take to compute the n-th Fibonacci number using the recurrence. One way of answering this question is to count (or estimate) the **number of operations** that you (or the computer) have to perform to find the answer. If we treat the addition of two integers as one operation[1] and count how many operations we need to perform. Consider this simple algorithm:

```
def fib(n):
  if n ≤ 1:
    return 1
  else:
    return fib(n − 1)+fib(n − 2)
```

Let us see how many operations will result when we ask the function to compute the 5th Fibonacci number

$$\begin{aligned}
\texttt{fib}(5) &= \texttt{fib}(4) + \texttt{fib}(3) \\
&= (\texttt{fib}(3) + \texttt{fib}(2)) + (\texttt{fib}(2) + \texttt{fib}(1)) \\
&= ((\texttt{fib}(2) + \texttt{fib}(1)) + \texttt{fib}(2)) + (\texttt{fib}(2) + \texttt{fib}(1)) \\
&= (((\texttt{fib}(1) + \texttt{fib}(0)) + \texttt{fib}(1)) + (\texttt{fib}(1) + \texttt{fib}(0))) + ((\texttt{fib}(1) + \texttt{fib}(0)) + \texttt{fib}(1))
\end{aligned}$$

[1] you probably see that this is ignores the difficulty that adding single-digit numbers is a lot easier than adding numbers with hundreds of digits, but we'll ignore this for now

and now using that $\texttt{fib}(1) = 1$ and $\texttt{fib}(0) = 0$, we conclude, using 7 operations that $F_5 = 5$. While this seems okay at this magnitude, it really isn't very efficient. Since $\texttt{fib}(n)$ refers to $\texttt{fib}(n-1)$ and $\texttt{fib}(n-2)$, we are almost doubling the number of terms at every step! This is what we would call exponential time complexity, and this is considered very bad. To compute $\texttt{fib}(n)$ this way, an upper bound for the number of additions is 2^n and it's not easy to come up with a smaller upper bound!

Notice that during the course of computing $\texttt{fib}(5)$, we computed $\texttt{fib}(2)$ three different times. This seems really inefficient, and if we are just a little bit smarter with the code and we remember each computation as we are performing them (like when we wrote it on a paper/the computer's memory), we can reduce the number of operations significantly. This technique is called dynamic programming. For example, if we are faced with the task of computing $\texttt{fib}(5)$, we could start by computing $\texttt{fib}(2) = \texttt{fib}(1) + \texttt{fib}(0) = 1 + 0 = 1$ (a single operation), then saving it to memory. Then we could proceed to compute $\texttt{fib}(3) = \texttt{fib}(2) + \texttt{fib}(1) = 1 + 1 = 2$ (here we are reading the value of $\texttt{fib}(2)$ from memory), which would take us another operation. We see that using this method, we would have to perform at most n additions.

6 Big Oh notation ([KT17], Chapter 4.3)

At this point it should be clear that the second way of computing the Fibonacci numbers is better, but how do we quantify this? It would likely be unfeasible to count *exactly* how many operations we would have to perform, and it also would not be completely relevant. Remember, we are trying to *study the problem*, not solve an any particular case of it. We don't care about how long the computation will *exactly* take, we just want to know how long it will take *approximately*. We want to know that if we run the code, is it going to return an output within seconds, hours, months, or maybe in a million years? So we really just want a **good upper bound** for the time.

Definition 6.1. *Let $f, g : \mathbb{N} \to \mathbb{R}$ be functions. We write*

$$f = O(g)$$

and say f is "Big Oh" of g when there is a constant C and an integer n_0 such that $f(n) \leq Cg(n)$ whenever $n > n_0$.

This definition is trying to say that if $f = O(g)$, then for large enough n, f is "asymptotically at most g".

For example, let f and g be the functions that describe the number of additions required by the two algorithms we used to compute the Fibonacci numbers. Note that we did not actually find how many additions they each took, but we found good appriximations for both. Then in the Big Oh language,

$$f = O(2^n)$$
$$g = O(n)$$

We have quantified the complexity of the two algorithms that let us compute Fibonacci numbers. But in fact, f is much bigger than g for large n, and this is captured by the "little oh" notation

Definition 6.2. *Let $f, g : \mathbb{N} \to \mathbb{R}_{\geq 0}$ be functions. We write*

$$f = o(g)$$

and say that f is "little oh" of g if $\lim_{n\to\infty} \frac{f(n)}{g(n)} = 0$.

In our example, we have $g = o(f)$.

References

[KT17] Mitchel T. Keller and William T. Trotter. *Applied Combinatorics*. Open access, 2017. Available at http://www.rellek.net/appcomb/. 1, 2, 3