MAT344 Lecture 8

2019/May/30

1 Announcements

2 This week

This week, we are talking about

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

3 Recap

Last time we talked about

1. Algorithms
2. Big Oh notation

4 Complexity classes

When thinking about asymptotic growth of functions, there are a few broad categories that are useful for keeping in mind. If we are considering a computational problem (whose input is given by an integer n), then if $h(n)$ is the function describing the number of operations necessary to solve the problem,

1. Constant functions. These are $h(n) = O(1)$. Meaning that $h(n) \leq C$ for all n. These are trivial problems.
2. Logarithmic functions. These are $h(n) = O(\log n)$. These are considered "easy" problems.
3. Polynomial functions. Examples of these include $n, n^2, \sqrt{n}, n^c$ for some constant c. These are $h(n) = O(n^c)$ for some constant c. Problems in this class are said to be "in $\mathcal{P}$", suggesting that they will take "polynomial time" to compute.
4. Exponential functions. Examples of these include 2^n, n^n, etc. These are the problems that are considered "very hard".

These are the categories we usually want to compare our other functions to.

5 Binary search

Let us consider a real-life application.

Exercise 5.1. *You are trying to look up a person's phone number in a phone book. A phone book is an alphabetized list of people's names, with their phone numbers next to them. The operation we are considering is "reading an entry in the phone book". Determine the difficulty of the problem, i.e. if the phone book contains n names, how many do we need to read to find the person we are looking for?*

Intuitively this seems like an "easy" problem. To prove it is actually easy, we have to come up with an algorithm, and a function $f(n)$ counting the number of operations required by the algorithm to find the solution. Then we want to give a good upper bound for the values of f for large n.

Here is one way: we start in the beginning and read the names in order until we find the one we are looking for. Let $f(n)$ be the function counting the number of names read by the algorithm. Then $f(n) = O(n)$, as in the worst case we have to read all n names, and even on average we have to read $\frac{n}{2}$ of them (note that both of these are in $O(n)$).

This would be considered an acceptable difficulty, but we can make it much better. The phone book is *sorted*, so if we open it at a random page about halfway, read a name, check if the person's name is before or after the name we just read in the in the alphabet then we have cut our options down by half. Let $g(n)$ denote the number of operations required by this algorithm. We see that we'll need to read approximately $\log_2(n)$ many names to find the phone number we are looking for, or, in Big Oh notation, $g(n) = O(\log(n))$. This confirms our intuition that we would much rather look up someone's phone number in a 100 person phone book than compute the 100th Fibonacci number.

This algorithm is called **binary search**. A more abstract version of this problem asks to find the position of an entry in a sorted array. Figure 1 illustrates how it proceeds in the case when we are trying to find the position of the entry 7 in an array with 16 entries.

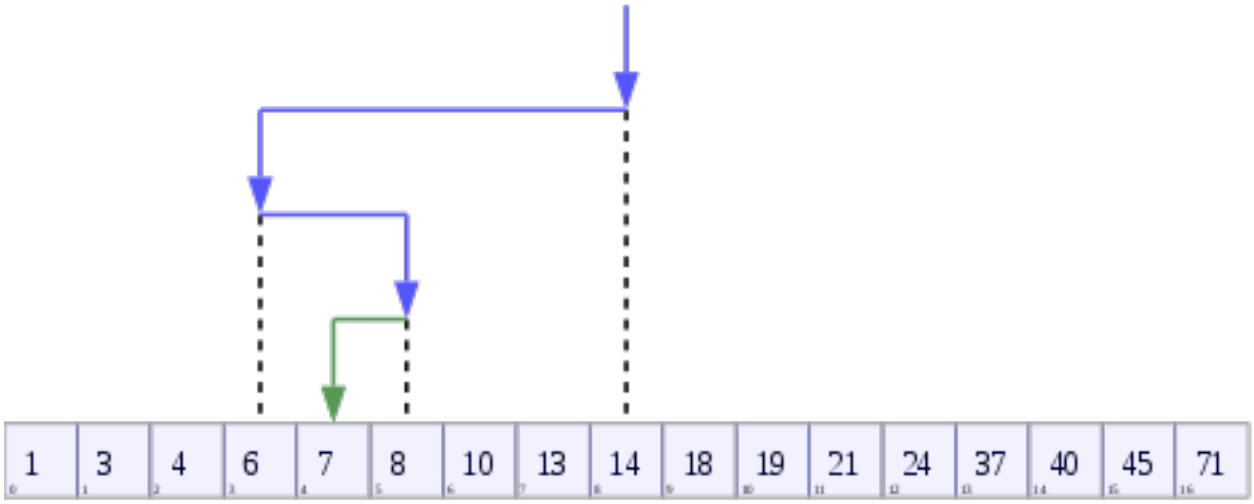

Figure 1: Binary seach

Let's modify the phone book example a little bit. What if you have a phone number and you want to find out who it belongs to? Since the phone book isn't sorted by phone numbers, we can't use binary search and the only thing you can do is to start reading the names and check if the number matches. This would be a $f(n) = O(n)$ difficulty. Making this a little more abstract, this is checking if a certain entry is in an unsorted array.

6 Sorting algorigthms

The $O(\log(n))$ difficulty of binary seach is *much better* than the $O(n)$ for reading all the entries in a list. So we always want to search sorted lists. But how difficult is it to sort a list?

Suppose we have a sequence of integers $a_1, \ldots a_n$ and we want to make a computer sort them. How are we going to assess the speed of a sorting algorithm? One reasonable operation we may want to count it the number of comparisons the algorithm has to make to sort the list.

7 Selection sort

Let's start with a simple idea: We first run through the sequence, we find the smallest element. We can achieve this by comparing a_1 to the following elements, until we find a smaller one, then we proceed comparing *that element* to

the remaining elements. Once we have performed $n-1$ comparisons, we will have found the smallest element in the list. We put this in the first place and repeat this for the remaining $n-1$ element list. It is not hard to see that altogether we have to perform

$$\sum_{i=0}^{n}(n-i)$$

comparisons. We recognize this formula as

$$\binom{n}{2} = \frac{n(n-1)}{2}$$

and this has complexity $O(n^2)$. This is not too bad (it's a polynomial time algorithm) but it will still take a while on longer lists (see https://www.youtube.com/watch?v=Ns4TPTC8whw for a demonstration of selection sort).

We could do much worse, for example: https://xkcd.com/1185/. The question you should have is: Can we do better?

8 Merge sort

One of the most important algorithms you learn in a computer science class is the Merge Sort algorithm. For a demonstration, see https://www.youtube.com/watch?v=XaqR3G_NVoo. The idea is to repeatedly subdivide a list into smaller lists (all the way until they just have one element) and then merge them back into longer lists. See Figure 2 for an illustration.

Since it is a more involved algorithm we include the pseudocode.

1. `MergeSort`$(a_1, \ldots, a_n)$:
   (a) If $n = 1$, then return a_1.
   (b) If $n = 2$, then return $\min(a_1, a_2), \max(a_1, a_2)$.
   (c) Let $k = \lceil n/2 \rceil$.
   (d) Let $L =$ `MergeSort`$(a_1, \ldots, a_k)$.
   (e) Let $R =$ `MergeSort`$(a_{k+1}, \ldots, a_n)$.
   (f) Return `Merge`(L, R)

2. `Merge`(L, R):
   (a) Let G be an empty list.
   (b) While both L and R are not empty:
      i. Let $L = \ell_1 \leq \ell_2 \leq \cdots$, and $R = r_1 \leq r_2 \leq \cdots$.
      ii. If $\ell_1 < r_1$, then add ℓ_1 to the end of G and remove ℓ_1 from L. Otherwise, add r_1 to the end of G and remove r_1.
   (c) Append L and R to the end of G
   (d) Return G

Merge sort is a really good algorithm and is used by many programming languages as the built-in sorting routine. Its main competitor is an algorithm called `QuickSort` which performs slightly better on average but the worst-case performance of it is worse than `MergeSort` (its complexity is $O(n^2)$. For a demonstration, see https://www.youtube.com/watch?v=es2T6KY45cA

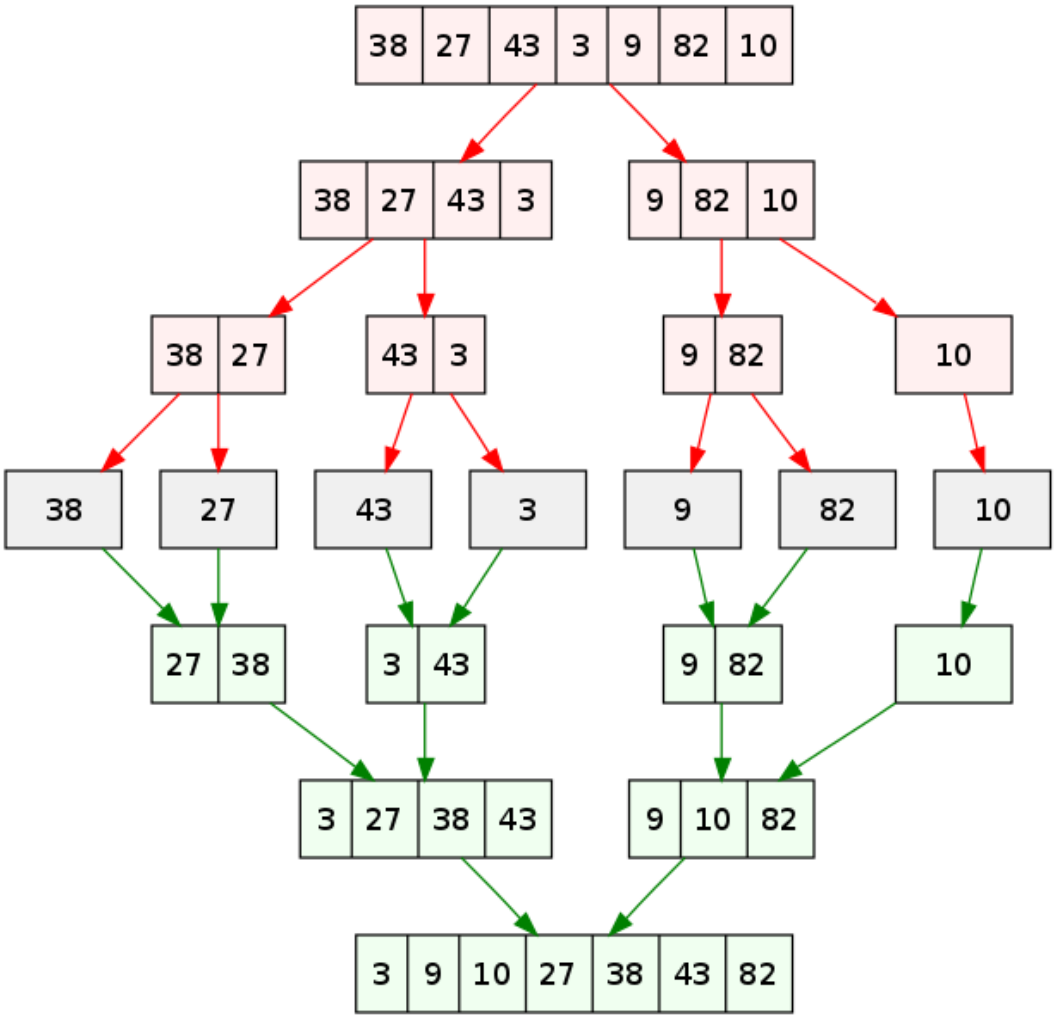

Figure 2: Merge sort

9 The complexity of MergeSort

What is the complexity of the algorithm? We're measuring the number of comparisons made in this algorithm. Let $f(n)$ denote the maximum number of comparisons made by the `MergeSort` algorithm on a list of n integers. We now that $f(1) = 0$, and $f(2) = 1$ just by inspecting the algorithm. For larger n, the number of comparisons depends on the number of comparisons made for smaller length inputs. We can come up with a recursive definition for $f(n)$:

$$\begin{aligned} f(n) &= f(\lceil n/2 \rceil) + f(n - \lceil n/2 \rceil) + \text{Complexity of } \texttt{Merge} \\ &\leq 2f(\lceil n/2 \rceil) + \text{Complexity of } \texttt{Merge} \end{aligned}$$

The complexity of the `Merge` operation on two lists of length at most k is going to be at most $2k$. Why? Let's think about the easy and hard cases for `Merge`. The easy case is if one list (say L) is bigger than all elements of the other list (say R). In that case, then the number of comparisons will be one comparison per item of L, which is at most k. In the hard case, let's say we have a sequence like this:

$$1, 3, 5, 7, \ldots$$

versus

$$2, 4, 6, 8, \ldots$$

In this case, we will have to make one comparison per element of both lists, which means at most $2k$ comparisons. Therefore we have

$$f(n) \leq 2f(\lceil n/2 \rceil) + 2\lceil n/2 \rceil.$$

This is a nice recursive bound on $f(n)$, but it doesn't tell us exactly how many comparisons the algorithm takes. But the point is, we usually don't care about the *exact* complexity of an algorithm – we'd rather like to understand the qualitative behaviour: we'd like to find out whether the complexity of an algorithm is closer to n comparisons, n^2 comparisions, or 2^n comparisons?

Let's try to come up with an estimate from the formula

$$f(n) \leq 2f(\lceil n/2 \rceil) + n$$

and $f(1) = 0$ and $f(2) = 1$.

We are going to show that $f(n) = O(n \log n)$.

Fix an n. Let's ignore the ceiling function to keep things simple. We are going to use the fact that we know that $f(1) = 0$.

$$\begin{aligned}
f(n) &\leq 2f\left(\frac{n}{2}\right) + n \\
&\leq 2\left(2f(\frac{n}{4}) + \frac{n}{2}\right) + n \\
&\leq 2\left(2\left(2f(\frac{n}{8}) + \frac{n}{4}\right) + \frac{n}{2}\right) + n \\
&= 2^3 f(\frac{n}{8}) + n + n + n \\
&\vdots \\
&\leq 2^k f(\frac{n}{2^k}) + \underbrace{n + n + \cdots + n}_{k \text{ times}}.
\end{aligned}$$

How large does k need to be in order for $\frac{n}{2^k}$ to be 1? Well, it needs to be at least $\lceil \log_2 k \rceil$. Therefore, we have that

$$f(n) \leq 2^{\lceil \log_2 n \rceil} f(1) + \lceil \log_2 n \rceil \cdot n \leq n(1 + \log_2 n).$$

And this is $O(n \log n)$, since for $n \geq 2$, $n \log_2 n \geq n$.

10 P=NP? (Chapter 4.4.3 in [KT17])

One of the most famous unsolved problems in mathematics and one of the Millenium Prize Problems is the question: Is $\mathcal{P} = \mathcal{NP}$? The $\mathcal{P}$ in this question is the one we saw before, algorithms that can be solved in polynomial time (i.e. they are $O(n^k)$ complexity for some k). The class $\mathcal{NP}$ is a little harder to describe, one way of interpreting it is that these are the problems that if someone gives you a claimed solution, you can *verify* that it is a solution *in polynomial time*, i.e. the problems that have easy to verify certificates. One example that might illustrate this is a Sudoku puzzle. It is easy to check that a filling of the Sudoku grid with numbers is a solution, and this difficulty only grows slowly (polynomially) as we increase the size of the grid. *Solving* a puzzle, on the other hand, seems much harder than that. The difficulty of all the known algorithms for solving Sudoku puzzles grow *exponentially* with increasing the size of the board.

All this is saying, however that we *do not know* if there is an efficient (polynomial time) algorithm for Sudoku puzzles. Maybe we just have not found it yet. It is in general extremely difficult to prove that there isn't an easy algorithm for doing something. We will not discuss this problem further, but feel free to read more about it in our textbook.

References

[KT17] Mitchel T. Keller and William T. Trotter. *Applied Combinatorics*. Open access, 2017. Available at http://www.rellek.net/appcomb/. 5