

Lecture 3: Introducing Recurrences

January 18, 2022

*Lecturer: Frederic Faulkner**Scribe: Aditya Diwakar & David Burns*

Homework 1 due Saturday and Homework 2 will be released Saturday. We recommend to regularly ask questions on Piazza and to go to Office Hours.

1 Fast Multiplication

In the previous lecture, we discussed how we can use a recursive strategy to multiply numbers by dividing the bits in half and then recursively multiplying.

Our original naive method required 4 recursive multiplications, such as:

$$\underbrace{T(n)}_{\text{number of steps for } n \text{ sized input}} = \underbrace{O(n)}_{\text{nonrecursive work}} + \underbrace{4T\left(\frac{n}{2}\right)}_{\text{multiplying } n/2 \text{ bits, 4 times}}$$

This comes out to $T(n) = O(n^2)$. We can make a faster method by only performing 3 recursive calls, giving a runtime of:

$$T(n) = O(n) + 3T\left(\frac{n}{2}\right)$$

...which comes out to $O(n^{\log_2 3}) \approx O(n^{1.59})$, a faster algorithm.

2 Solving Runtimes

Lemma 2.1. Let $f(n) = 1 + a + a^2 + \dots + a^n$, then

$$f(n) = \begin{cases} O(1) & a < 1 \\ O(n) & a = 1 \\ O(a^n) & a > 1 \end{cases}$$

Proof. For the case when $a > 1$, the conclusion of $O(a^n)$ comes from

$$\frac{a^{n+1} - 1}{a - 1}$$

When $a = 1$, the conclusion of $O(n)$ comes from $\sum_{i=1}^n 1 = n$ and $a < 1$, this is a geometric series and has a constant sum irrespective of n given by:

$$s = \frac{1}{1 - a}$$

This can be a strategy for converting certain equations to a Big-O form, but may not be the most useful for recursive definitions. For the naive recursive multiplication, the recurrence relationship is given (and expanded) as:

$$\begin{aligned} T(n) &= O(n) + 4T\left(\frac{n}{2}\right) \\ &= c \cdot n + 4T\left(\frac{n}{2}\right) \\ &= c \cdot n + 4\left(c \cdot \left(\frac{n}{2}\right) + 4T\left(\frac{n}{4}\right)\right) \\ &= cn \left(1 + \frac{4}{2}\right) + 16T\left(\frac{n}{4}\right) \\ &= cn \left(1 + \frac{4}{2}\right) + 16\left(c \cdot \left(\frac{n}{4}\right) + 4T\left(\frac{n}{8}\right)\right) \\ &= cn \left(1 + \frac{4}{2} + \left(\frac{4}{2}\right)^2\right) + 4^3 \left(\frac{n}{2^3}\right) \\ &\vdots \text{ after } i \text{ steps} \\ &= 4^i T\left(\frac{n}{2^i}\right) + cn \left(1 + \frac{4}{2} + \left(\frac{4}{2}\right)^2 + \cdots + \left(\frac{4}{2}\right)^i\right) \end{aligned}$$

The base case for the recurrence is $T(1)$ so what value i do we need such that $n/2^i = 1$? In other words:

$$\frac{n}{2^i} = 1 \implies 2^i = n \implies i = \log_2 n$$

Hence, the above running time simplifies to

$$\begin{aligned}
T(n) &= 4^{\log_2 n} T(1) + cn \underbrace{\left(1 + \frac{4}{2} + \left(\frac{4}{2}\right)^2 + \cdots + \left(\frac{4}{2}\right)^{\log_2 n - 1} \right)}_{\text{geometric with } a > 1; \text{ dominated by largest term}} \\
&= 2^{2 \cdot \log_2 n} + cn O \left(\left(\frac{4}{2}\right)^{\log_2 n - 1} \right) \\
&= n^2 + cn O \left(\left(\frac{4}{2}\right)^{\log_2 n - 1} \right)
\end{aligned}$$

Simplifying the second term shows:

$$cn \left(\frac{4}{2}\right)^{\log_2 n - 1} = cn \left(\frac{4^{\log_2 n - 1}}{2^{\log_2 n - 1}}\right) = cn \left(\frac{4^{\log_2 n - 1}}{O(n)}\right) = c \cdot \underbrace{4^{\log_2 n - 1}}_{O(n^2)} = cn^2 = O(n^2)$$

With this, we can finish the above:

$$T(n) = n^2 + O(n^2) = n^2 + c(n^2) = (c + 1)(n^2) = \boxed{O(n^2)}$$

We can follow this same process for the faster multiplication algorithm:

$$\begin{aligned}
T(n) &= 3T\left(\frac{n}{2}\right) + O(n) \\
&= 3\left(3T\left(\frac{n}{4}\right) + \frac{cn}{2}\right) + cn \\
&= 9T\left(\frac{n}{4}\right) + cn\left(1 + \frac{1}{2}\right) \\
&= 9T\left(3T\left(\frac{n}{8}\right) + \frac{cn}{4}\right) + cn\left(1 + \frac{3}{2}\right) \\
&= 27T\left(\frac{n}{8}\right) + cn\left(1 + \frac{3}{2} + \frac{9}{4}\right) \\
&= 3^3 T\left(\frac{n}{2^3}\right) + cn\left(1 + \frac{3}{2} + \left(\frac{3}{2}\right)^2\right)
\end{aligned}$$

Now that a pattern has been found in the recurrence, we can see that

$$T(n) = 3^i T\left(\frac{n}{2^i}\right) + cn \left(1 + \frac{3}{2} + \left(\frac{3}{2}\right)^2 + \cdots + \left(\frac{3}{2}\right)^i\right)$$

We now compute for the last value of i :

$$\frac{n}{2^i} = 1 \implies n = 2^i \implies i = \log_2 n$$

Finally, we can complete the simplification of the recurrence:

$$\begin{aligned} T(n) &= \underbrace{3^{\log_2 n}}_{n^{\log_2 3}} T(1) + cn \underbrace{\left(1 + \frac{3}{2} + \left(\frac{3}{2}\right)^2 + \cdots + \left(\frac{3}{2}\right)^{\log_2 n-1}\right)}_{\text{dominated by last term}} \\ &= n^{\log_2 3} + cn \left(\left(\frac{3}{2}\right)^{\log_2 n-1}\right) \\ &= n^{\log_2 3} + cn \left(\frac{3^{\log_2 n-1}}{2^{\log_2 n-1}}\right) \\ &= n^{\log_2 3} + cn \left(\frac{3^{\log_2 n-1}}{O(n)}\right) \\ &= n^{\log_2 3} + O(3^{\log_2 n}) \\ &= n^{\log_2 3} + O(n^{\log_2 3}) \\ &= n^{\log_2 3} + cn^{\log_2 3} = (c+1)n^{\log_2 3} = \boxed{O(n^{\log_2 3})} \end{aligned}$$

3 Towers of Hanoi

Time for a game! Towers of Hanoi is a game with 3 pegs and n disks. The goal is to get all of the n pegs onto another peg. It is illegal to put a larger disk onto a smaller disk.

If all the disks start on the left-most peg, then the strategy is to move all the top disks except the last onto another peg and then recursively solve the smaller sub-game with one fewer disk.

For each step, there are two additional recursive calls, both of which have a size of $n - 1$. Hence, $T(n) = 2T(n - 1) + O(1)$. We can solve this:

$$\begin{aligned}
 T(n) &= 2T(n - 1) + O(1) \\
 &= 2(2T(n - 2) + c) + c \\
 &= 2^2T(n - 2) + (1 + 2)c \\
 &= 2^3T(n - 3) + (1 + 2 + 4)c \\
 &= 2^4T(n - 4) + (1 + 2 + 4 + 8)c \\
 &= \vdots \\
 &= 2^iT(n - i) + (1 + 2 + \dots + 2^{i-1})c \\
 &= 2^nT(1) + O(2^n) \\
 &= 2^n + O(2^n) \\
 &= 2^n + c2^n = (c + 1)2^n = \boxed{O(2^n)}
 \end{aligned}$$

A less detailed explanation and solution to the game is given here because lots of resources are available online with visualizations, animations, etc.

4 Binary Search

Binary search is a search algorithm for sorted arrays. The idea is that if you are given a sorted list and a number, each time you look at the middle, you can rule out half the elements in the list. Hence, binary search has a runtime of:

$$\begin{aligned}
 T(n) &= T\left(\frac{n}{2}\right) + O(1) \\
 &= \left(T\left(\frac{n}{4}\right) + c\right) + c \\
 &= \vdots \\
 &= T\left(\frac{n}{2^i}\right) + ic \\
 &= T(1) + c \log n = O(\log n)
 \end{aligned}$$

An implementation of this algorithm would be as follows:

```
1 Function BinarySearch ( $L, x$ ) :  
2   if  $|L| = 1$  then  
3     return  $L[1] = x$   
4   if  $x > L \left\lceil \frac{|L|}{2} \right\rceil$  then  
5     return BinarySearch ( $L \left\lceil \frac{|L|}{2} : |L| \right\rceil$ )  
6   else  
7     return BinarySearch ( $L \left[ 0 : \frac{|L|}{2} \right]$ )
```

Remark: Remember, binary search requires the input array to be sorted. Otherwise, the most optimal way to find an element would be to do a linear scan over the list (with complexity $O(n)$).