Prep Work 10 - Recurrences

CS 234

due April 7, before class

0 Introduction

This assignment has 1 part: the recurrences.

This assignment is to be completed individually, but feel free to collaborate according to the course's external collaboration policy (which can be found in the syllabus).

The deliverables consist of one `.pdf` file. The deliverables should be submitted electronically by the deadline. Put any attribution text in the `.pdf` file.

Every file should be named like `FLast_cs234_pX.ext` where `F` is your first initial, `Last` is your last name, `X` is the assignment number, and `ext` is the appropriate file extension. For example, Liron Cohen's `.pdf` file should be given the name `LCohen_cs234_p10.pdf`. (Liron Cohen is researcher in constructive/computable logic and mathematics. When I was an undergrad, she taught me about ancestral logic!)

1 Pumping Lemma

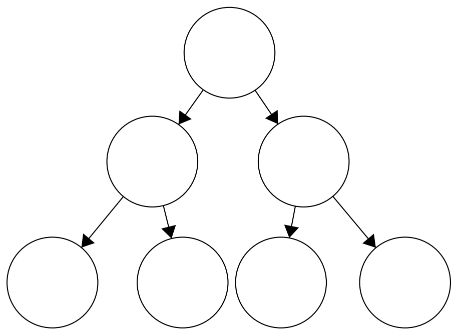
Read chapter G in the textbook. Then complete the following tasks in your `.pdf` submission. Clearly label your responses with the task number.

The tree method in the textbook does not quite show you where the "tree" comes in. Here is an alternative explanation that will match what I will show you in class. To make comparison clear, this example will use the same recurrence as the textbook example:

$$T(n) = \begin{cases} 0 & n \leq 1 \\ 2 \cdot T(\frac{n}{2}) + cn & n > 1 \end{cases}$$

The recursive case of this recurrence can be divided into two parts: the recursive call part ($2 \cdot T(\frac{n}{2})$), and the "local" part (cn). There is no work done in the non-recursive part of the recurrence so we will ignore it.[1]

First focus on the recursive call part. There is a coefficient of 2 mutliplying the recursive call to T. This comes from the algorithm making 2 recursive calls. Thus, we might imagine that the "call tree" of the algorithm looks like the following for large inputs n, where a given call at the root requires 2 calls represented as child nodes, and each of those children also needs 2 calls/has 2 children, etc.



Because this call tree branches into 2 at every node, it ends up having 2^n nodes per row (where the root is considered be in row 0).

Next we need to consider how tall the tree is. Note that n, the argument to T, is divided by 2 at each recursive call, and recursive calls only stop when $n \leq 1$. For simplicity, let's assume that the input n is a power of 2 so that we end at 1 exactly—if n is not a power of 2, we can always pretend it is by rounding it up to the next power of 2. Thus we see the following relation between the

[1]But beware! The non-recursive part cannot be ignored in general. This approach must be modified if the the work done here is not 0.

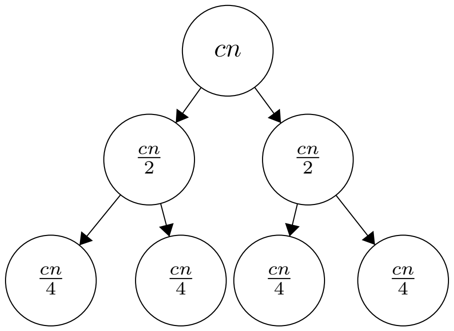
maximum row number m and the input n:

$$n \cdot \left(\frac{1}{2}\right)^m = 1$$
$$\iff n = 2^m$$
$$\iff log_2(n) = m$$

Thus m is $log_2(n)$. Since row numbering starts at 0, this means that there are $log_2(n) + 1$ rows. (Be careful! It is easy to make an off-by-one error here!)

Now that we know the entire shape of the tree, the final piece of the puzzle is to determine how much work is accomplished *locally* to each node. That is, how much work is performed at a node *not* counting any work from further recursve calls. This corresponds to the local part of the recurrence we picked out earlier: cn.

But it is not the case that there is cn work per node! The argument n changes at every row. Each row has half the argument of the previous, so in row r each node only performs $cn \cdot \left(\frac{1}{2}\right)^r$ work. Thus the first couple rows of the fully-filled call tree looks like the following:



The final step is sum this all together. The amount of work in a given row r is just the number of nodes in that row (2^r) times the amount of work per node in that row ($cn \cdot \left(\frac{1}{2}\right)^r$). Then the work represented by the whole tree is sum of the work over all the rows.

$$\sum_{r=0}^{log_2(n)} 2^r \cdot cn \cdot \left(\frac{1}{2}\right)^r$$
$$= \sum_{r=0}^{log_2(n)} cn$$
$$=(log_2(n) + 1) \cdot cn$$

This work is in $O(n \cdot log(n))$.

1. The above description of the tree method finds three quantities to combine into the final result. What properties do each of those three quantities measure?

Suppose for the next tasks that you have the following recurrence:

$$T(n) = \begin{cases} 0 & n \leq 1 \\ 4 \cdot T(\frac{n}{3}) + cn^2 & n > 1 \end{cases}$$

2. How many nodes will be in row r (starting from $r = 0$) of the call tree?
3. How tall will the call tree be?
4. How much work is done locally to each node of row r?
5. How should the above quantities be combined to get the work represented by the whole tree?

To compute these sums, it is necessary to know how to properly manipulate geometric series (sums of terms which differ by a common factor that is not equal to 0 or 1). The formula for such series is the following, where the common factor is x.

$$\sum_{i=0}^{m} a \cdot x^i = a \cdot \frac{x^{m+1} - 1}{x - 1}$$

6. How this formula is used in section G.6?
7. While the formula does not work when $x = 1$ because you would divide by 0, a different formula can still be derived that works with $x = 1$. What is this new formula? (Hint: Look back at the original summation, replace x with 1, and then simplify.)

To compute these sums, it is also necessary to know various logarithm identities. The textbook lists some identities, but one identity that is not listed is that $a^{log_b(c)} = c^{log_b(a)}$.

8. How this identity is used in section G.6?

Aside from using the tree method to directly derive a solution to a recurrence, one can also use induction induction to verify a guessed solution. This verification of guesses is an equally valid way of providing a solution to a recurrence.

9. Why is strong induction typically the correct choice for verifying a guessed solution to these divide-and-conquer recurrences?