CSCI 2041: Advanced Programming Principles
Analyzing the Clever Exponentiation Function

Gopalan Nadathur

This is the writeup I promised to put up about the analysis of the clever exponentiation function that we went through in the lecture on April 17.

Before I get started with this, I should repeat a couple of things that I said or at least hinted at in class; these are meant to set you at ease in case you are getting worried about being able to start such analyses on your own. The first thing is that not every kind of analysis is "obvious." If it were, there would not be much to education. In fact, we should look at education something that exposes us to new ways of looking at problems—more generally, to new ways of thinking about issues—so that we are better equipped the next time we look at similar problems or issues. The second thing is that any good course should be doing things that expand our minds, and there may be things in them that do this but are not really suitable material for an "in-class exam." This discussion is in that category. More precisely, with regard to this problem, you should generally understand that the divide-and-conquer approach as played out in the clever exponentiation function leads to an exponential reduction in the number of multiplications and you should be able to apply this kind of idea to other related problems. What you do not need to be able to do is reproduce or mirror the argument for a precise solution that I go through below with your "eyes closed."

Perhaps another, shorter way to say all the stuff above: relax as you read the rest of this note, try to enjoy what you are reading (yes, math *can* be exhilarating and eye-opening if viewed the right way!) and see if you can learn something from it that you can use at a later time.

Enough of "sermons," lets get to substance! Our problem was the following. We were given the OCaml function shown below:

```
let rec exp a b =
  match b with
  | 0 -> 1
  | n ->
      let n = exp a (b / 2) in
        n * n * (if b mod 2 = 0 then 1 else a)
```

Our goal was to write down a recurrence equation for the number of multiplications this code would carry out, expressed as a function of the second (exponent) argument and to then solve that equation.

One small clarification before we move on. In the lecture, I wrote down expressions for the amount of *time* the function would take, but here I seem to be counting only the number of multiplications. Thus, the obvious question: am I doing the same thing in this note when I count multiplications only? The answer is "yes." Here is a way to understand this: the other computations—the match, the call to functions, the evaluation of the if-then-else and the mod—all take place in the context of multiplications, except in the case of the last recursive call. What this means is that if we count the multiplications only, then we can get the complete running time by multiplying the result by a constant and then adding a constant (for the last call). Check this out and convince yourself that nothing is lost in focusing only on the number of multiplications, else ask questions on Forum.

The first part of our task, that of writing down the recurrence equation, is not too difficult to do. When the second argument to *exp*, which I will call b, is 0, there are no multiplications. On the other hand, when b is greater than 0, there are 2 multiplications plus however many multiplications there will be when *exp* is called with the second argument reduced to $\lfloor b/2 \rfloor$. Thus, using $T(b)$ to signify the number of multiplications when this second argument is b, we get the following recurrence equation:

$$T(b) = \begin{cases} 0 & \text{if } b = 0 \\ T(\lfloor b/2 \rfloor) + 2 & \text{if } b > 0. \end{cases} \qquad (1)$$

The tricky part, which we spent a fair part of the lecture time on, is understanding how to solve this recurrence equation. As usual, we can try to guess a solution by repeated expansion. If we do this, here is how things would proceed:

$$\begin{aligned} T(b) &= T(\lfloor b/2 \rfloor) + 2 \\ &= T(\lfloor (\lfloor b/2 \rfloor)/2 \rfloor) + 2 + 2 \\ &= T(0) + 2 + \ldots + 2 \\ &= 0 + 2 + \ldots + 2 \end{aligned}$$

Looking at this, we see that if we can figure out how many 2s there are in the last expression, then we can easily write a closed-form expression for $T(b)$: it would just be 2 multiplied by that number. However, coming up with a number for how many 2s there are going to be seems not so easy: for example, how can we relate the floor of the result of dividing the floor of $b/2$ by 2 to b? And this kind of nesting repeats even further, so this might seem to be a hopeless task.

Well, most interesting things seem hopeless when you look at them at first and the key is to stand back a little, think of what you are looking at in different ways and try to extract a structure; I would contend that it is *this* aspect, the challenge of discovering a structure, that makes things interesting in the first place.

So how should we proceed in this case? First let us consider the case where b is a power of 2. For example, suppose b is 2^2. How many recursive calls do we end up making? There would be 4 of them, these being with $b = 4$, $b = 2$, $b = 1$ and $b = 0$, respectively. Only 3 of these calls involve multiplications, so the factor for 2 in the expression for $T(2^2)$ would be 3. As another example, suppose $b = 2^4$. By a similar reasoning, we see that the number of recursive calls will be 6 and hence the multiplication factor would be 5. Thus, we seem to have a pattern at least for the case when b is a power of 2: if b is 2^k, then the multiplication factor is $k + 1$. Another way to phrase this is that the multiplication factor should be $log_2(b) + 1$ in this case.

However, we also want to deal with b when it is *not* a power of 2. Here the key observation is that all the numbers from 2^k up to one less than $2^{(k+1)}$ behave the *same way* from the perspective of the number of recursive calls. For example, consider the case where $b = 6$ at the outset. The recursive calls will be for $b = 6$, $b = 3$, $b = 1$ and $b = 0$, and the multiplication factor will be 3. More generally, a good conjecture seems to be that the multiplication factor will be $k + 1$ for all values of b such that $2^k \leq b < 2^{(k+1)}$. But how do we express this uniformly as a function of b? A slight change in the way we phrased our observation/conjecture for the case when b is a power of 2 helps in this: the multiplication factor is $(k + 1)$ for all values of b such that $2^k < b + 1 \leq 2^{(k+1)}$. If we phrase it this way, we can state the multiplication factor as $\lceil log_2(b + 1) \rceil$. Of course, this means that our analysis works only for those bs that can be characterized as belonging to such a range.

Since the smallest sensible value for k is 0 in this analysis, it must be the case that $2^0 = 1 < b + 1$, i.e. $b \geq 1$. For the only other value of b, that when $b = 0$, we will have to state the solution as a special case.

In summary, then, our conjecture of the solution to the recurrence equation is the following:

$$T(b) = \begin{cases} 0 & \text{if } b = 0 \\ 2 * \lceil log_2(b+1) \rceil & \text{if } b > 0 \end{cases} \qquad (2)$$

The solution that we have stated above is only a conjecture at this stage and we have to think of a way to verify it. For this, we have to pay more attention to the intuition underlying the process by which we arrived at our conjecture. If you look at the analysis more closely, it relied on the fact that any time we take a number between 2^k and $2^{(k+1)}$ and take the floor of the result of dividing it by 2, we get a number between $2^{(k-1)}$ and 2^k; of course, k has to be greater than 0 for this to make sense. This is essentially what allowed us to say that we would have $(k + 1)$ recursive calls to get down to a call with $b = 1$ (and then one more call with $b = 0$). This insight will obviously be useful in proving the correctness of our conjecture, so lets establish it as a lemma.

Lemma. For any number n such that $2^k \leq n < 2^{(k+1)}$ for some $k > 0$, it is the case that $2^{(k-1)} \leq \lfloor n/2 \rfloor < 2^k$.

Proof. We can write n as $2^k + c$ where c has the following property:

$$0 \leq c < 2^{(k+1)} - 2^k.$$

But from this it follows that

$$0 \leq \lfloor c/2 \rfloor < 2^k - 2^{(k-1)}. \qquad (3)$$

Now we observe that

$$\lfloor n/2 \rfloor = \lfloor (2^k + c)/2 \rfloor = 2^{(k-1)} + \lfloor c/2 \rfloor. \qquad (4)$$

Putting (3) and (4) together, it follows that

$$2^{(k-1)} \leq \lfloor n/2 \rfloor < 2^k$$

as we desired to show. □

We really want to use the lemma above to relate the number of recursive calls when b is the second argument to *exp* to the number of recursive calls when $\lfloor b/2 \rfloor$ is the second argument. That is done in the following simple corollary to the lemma.

Corollary. For $b > 0$, $\lceil log_2(b+1) \rceil = \lceil log_2(\lfloor b/2 \rfloor + 1) \rceil + 1$.

Proof. Since $b > 0$, we know that for some $k > 0$ the following holds: $2^k \leq b < 2^{(k+1)}$. For the k for which this property is satisfied, $\lceil log_2(b+1) \rceil = k + 1$. By the lemma that we have just proved, we also know that $2^{(k-1)} \leq \lfloor b/2 \rfloor < 2^k$. But this means that $\lceil log_2(\lfloor b/2 \rfloor + 1) \rceil = k$. Thus

$$\lceil log_2(\lfloor b/2 \rfloor + 1) \rceil + 1 = k + 1 = \lceil log_2(b+1) \rceil,$$

thereby verifying the claim. □

We now have all the ingredients in place to prove that our conjecture is in fact correct. This is stated and proved in the theorem below.

Theorem. The recurrence equation shown in (1) has the solution shown in (2).

Proof. As expected, we will use induction over natural numbers in this argument. One thing to note here is that the value of b in a recursive call could decrease by more than 1—in fact the recursive call uses $\lfloor b/2 \rfloor$—so this is a case where the stronger version of induction is more useful.

How does that form of induction work? Well, we get to assume that the conjecture in (2) holds for all $b' < b$ and we need to show it also holds for b. The assumption is vacuous for the case when $b = 0$ so we will prove that case separately—i.e. it will be our "base case"—and then we will treat the remaining cases.

The argument for $b = 0$ is actually quite easy: Looking at the code for the function we see that there are no multiplications in this case and that is what the solution in (2) also tells us.

Now for the argument when $b > 0$. This proceeds as follows:

$$\begin{array}{rcll} T(b) & = & T(\lfloor b/2 \rfloor) + 2 & \text{by the recurrence equations in (1)} \\ & = & 2 * \lceil log_2(\lfloor b/2 \rfloor + 1) \rceil + 2 & \text{by the inductive assumption} \\ & = & 2 * (\lceil log_2(\lfloor b/2 \rfloor + 1) + 1) & \text{by rearranging terms} \\ & = & 2 * \lceil log_2(b+1) \rceil & \text{by the corollary} \end{array}$$

The final equality is what we were after, so the proof is complete. □