COMP 220
Lecture Notes 02
Analyzing Simple Recursion

August 28, 2017

In COMP161 you mostly looked closely at the analysis of loops. Here we look at simple recursive functions and their analysis. If you want a refresher and/or some more perspective on the basics of recursion, then you should review chapter 7 of the text.

Repetition is as Repetition Does

The essential analysis task of analyzing a recursive function is the same as analyzing a loop because they both induce the same kind of property in out computations: repetition. Recursive functions do so by repeatedly calling themselves where loops manage it via the implicit repetition of C++ loop control structures. What this means is we must still look for three things:

1. The amount of work carried out independently of the repetition.
2. The amount of work carried out per repetition
3. The frequency of repetition.

The only difference between analyzing a recursive procedure and loop-based iterative procedure is where we go looking for these things.

Recursive Linear Search

In lecture notes 16 from COMP161[1] you looked at linear search implemented recursively. This implementation is shown again in figure 1 and utilizes the two function pattern that is typical with our design methodology for developing recursive functions for C++ vectors. The second function utilizes index parameters *fst* and *lst* in order to manage the recursive decomposition of the vector's index range.

[1] `https://jlmayfield.github.io/MC-COMP161/Lecture_Notes/comp161-lectureNotes-16.pdf`

To start our analysis we start at the top with the search which takes the vector and key and searches the entire vector. This procedure carries out $O(1)$ operations to get the vector size and then makes a recursive call. This means that the complexity is $O(1) + O(f)$ where $O(f)$ is the complexity of the recursive helper. At this point we're pretty much done with this procedure. Its complexity is exactly the complexity of the helper[2], i.e. $O(f)$. The trick is not not get hung up on the unknown function f just yet. Take a moment to come to a

[2] Do you see why?

```
int search(const std::vector<int>& data,int key) {
  return search(data, 0, data.size(), key);
}

int search(const std::vector<int>& data,int fst, int lst, int key)
    {
  if (fst >= lst) {
    return -1;
  } else if (data[fst] == key) {
    return fst;
  } else {
    return search(data, fst+1, lst, key);
  }
}
```

Figure 1: Search: Recursive Implementation

decisive conclusion about the analysis of the top level function that is independent of the details of the analysis of its helper.

Now that we're more or less done thinking about the top level function, we can continue with the helper. The worst case clearly occurs with the condition for the *if* and *else if* is false and we drop down to the recursive call in the *else*. When this happens then the procedure will perform exactly 4 operations prior to making the recursive call[3].

[3] >=, [], ==, and +

We now need to think about how recursion is carried out. The function gets called over and over until the base case is reached in which no recursive call is made. If n recursive calls are made, then $n - 1$ of those make a recursive call and 1 is the terminating base case. Combine this with our previous analysis, we known that 4 operations are done when the procedure makes a recursive call, but what about the final recursive call, the base case call?

The code clearly indicates that two situations can terminate the recursion. The first is covered by the *if* and occurs when the the index range specified by *fst* and *lst* is empty. This case requires a single operations. The next situation is when for a non-empty index range, the key is found at *fst*. This case requires three operations. Given that our concern is overall worst case complexity, we need to recognize that the terminating case that maximizes work is the first one because terminating there means we've searched an entire index range and not a part of a range.

We now known that one operation, the work of the terminating base case, is carried out outside the repetition and that four operations are carried out every time a recursive call is made. We're left the question of how many recursive calls are made? To determine this we can dig into the code logic. Every recursive call advances the value of *fst* by one. Recursion terminates when *fst*≥*lst*. In short, we're dealing with counting logic just like our loops.

$$\begin{aligned} fst + k &\geq lst \\ k &\geq lst - fst \end{aligned} \tag{1}$$

Now recall that $lst - fst + 1$ is the number of integers in the range of fst to lst if we exclude last. So a simpler way of state what we found is that *the recursion repeats once for each of the elements in the index range*. When searching an entire vector, this is exactly the size of the vector. We can now find the exact work done by this function and it's complexity. For a vector of size n we'll do $4n + 1$ operations with a complexity of $O(n)$.

Recurrence Relations

Our analysis of liner search mostly relied on feeling our way through the code and recognizing a pattern familiar from iteration. There is a way of analyzing the complexity of a recursive function that is more general from the perspective that it relies less on recognizing the kind of counting that typifies iteration.

Before we determined the frequency of repetition we recognized two crucial cases to the worst case complexity for our function.

1. The empty range base case that carries out a single operation
2. The recursive case that carries out four operations plus the cost of recursively searching the remainder of the vector.

We can express this knowledge of the complexity of search as a recursive function. Let n be the size of the index range[4] and $\mathcal{T}$ be the function that maps n to the number of operations needed for the worst case search. Then equation 2 expresses $\mathcal{T}$ recursively!

[4] n =*fst-lst*

$$\begin{aligned} \mathcal{T}(0) &= 1 \\ \mathcal{T}(n) &= \mathcal{T}(n-1) + 4 \end{aligned} \tag{2}$$

Equations like those shown in equation 2 are called Recurrence Relations and are incredibly useful in the analysis of algorithms. The problem is that they are *open form solutions* to our problem. They do not provide us with an exact, calculable solution. What we really want is a *closed form* solution where $\mathcal{T}$ is not on the right hand side of the equation. To do this we must solve the recurrence.

Solving Simple Recurrences via Unrolling

The recurrence relation shown in equation 2 is about as simple as they come. The easiest way to solve it is usually to just unroll the recursion and determine the underlying pattern. As has become the norm, you're goal is to establish an equation for $\mathcal{T}(n)$ after k levels of unrolling or recursion depth k. The initial equation gives us depth 1 and we take it from there.

$$\begin{aligned}
\mathcal{T}(n) &= \mathcal{T}(n-1)+4 \\
&= (\mathcal{T}(n-2)+4)+4 = \mathcal{T}(n-2)+2*4 \\
&= (\mathcal{T}(n-3)+4)+2*4 = \mathcal{T}(n-3)+3*4 \\
& \quad \ldots \\
&= (\mathcal{T}(n-k)+4)+4(k-1) = \mathcal{T}(n-k)+4k
\end{aligned}$$

To finish the process and solve the recurrence we must determine when the recurrence terminates, the max depth of the recursion. The base case occurs when $n = 0$ so the recurrence terminates when $n-k = 0$ or, solving for k, with a recursive depth of $k = n$. When this happens, then we substitute the base case for the known value of $\mathcal{T}(0)$ on the right and find the closed form solution found in equation 3.

$$\mathcal{T}(n) = 4n+1 \tag{3}$$

To be certain that we're correct we can do a quick proof by induction or at the very least check the core logic of the proof. When n is zero our closed form gives us $\mathcal{T}(0) = 1$. The base case holds. Now we assume everything works great for the first $n-1$ steps and verify that this implies the above equation.

$$\begin{aligned}
\mathcal{T}(n) &= \mathcal{T}(n-1)+4 \\
&= (4(n-1)+1)+4 \\
&= 4n-4+1+4 \\
&= 4n+1
\end{aligned}$$

Thus, by the principle of mathematical induction our closed form solution matches the open form recurrence relation.

Simple Recurrences

In lab we noticed that basic step counting loops will keep us firmly in the ballpark of linear time algorithms or worse. The same pattern holds true for these kind of simple recurrences. What if the base case has a fixed cost of a, each recursive call has a fixed cost of w, and each step steps towards the base case by some fixed amount s. Then we'd be dealing with a recurrence that fits the template shown in equation 5.

$$\begin{aligned}
\mathcal{T}(0) &= a \\
\mathcal{T}(n) &= \mathcal{T}(n-s)+w
\end{aligned} \tag{4}$$

$$\begin{aligned}
\mathcal{T}(n) &= \mathcal{T}(n-s)+w \\
&= \mathcal{T}(n-2s)+2w \\
&= \mathcal{T}(n-3s)+3w \\
&\dots \\
&= \mathcal{T}(n-ks)+kw
\end{aligned} \tag{5}$$

We now solve for k when $n-ks=0$ to get $k=\lceil \frac{n}{s} \rceil$ for a closed form of:

$$\mathcal{T}(n) = \left\lceil \frac{nw}{s} \right\rceil + a \tag{6}$$

For fixed base case cost a, step size s, and work per recursion w, this is linear time, $O(n)$.

A Different Recurrence

Let's look at another recurrence relation without any particular code attached to it. In equation 7 you see a recurrence that is similar to that of linear search but where the argument to the recursive call to $\mathcal{T}$ is $\frac{n}{2}$ instead of $n-1$.

$$\begin{aligned}
\mathcal{T}(0) &= 1 \\
\mathcal{T}(n) &= \mathcal{T}(\tfrac{n}{2})+4
\end{aligned} \tag{7}$$

If you've been paying attention to the material thus far, then you can guess what the analysis of this recurrence is going to yield. Let's proceed by unrolling and see what we see.

$$\begin{aligned}
\mathcal{T}(n) &= \mathcal{T}(\tfrac{n}{2})+4 \\
&= (\mathcal{T}(\tfrac{n}{4})+4)+4 = \mathcal{T}(\tfrac{n}{2^2})+4*2 \\
&= (\mathcal{T}(\tfrac{n}{8})+4)+2*4 = \mathcal{T}(\tfrac{n}{2^3})+4*3 \\
&= \dots \\
&= (\mathcal{T}(\tfrac{n}{2*2^{k-1}})+4)+4(k-1) = \mathcal{T}(\tfrac{n}{2^k})+4k
\end{aligned}$$

After k steps of recursion we see $\mathcal{T}(\frac{n}{2^k})+4k$ operations. The base case is reached when $\left\lfloor \frac{n}{2^k} \right\rfloor = 0$. A nicer way of saying this is in terms the second to last step where $\left\lfloor \frac{n}{2^{k-1}} \right\rfloor = 1$.[5] If n is a power of two we can solve this easily:

[5] Since $\left\lfloor \frac{n}{2^k} \right\rfloor = 0$, $\frac{n}{2^k} < 1$ (otherwise taking the floor would yield 1, not 0). Multiplying each side by 2, we have $\frac{n}{2^{k-1}} < 2$, so taking the floor of the left hand side yields 1.

$$\begin{aligned}
\frac{n}{2^{k-1}} &= 1 \\
n &= 2^{k-1} \\
\log_2 n &= k-1 \\
k &= \log_2 n + 1
\end{aligned} \tag{8}$$

Intuition tells us this probably holds true when n is not a power of two. You can verify this by writing a proof by induction[6] Or, you

[6] This relates to the "guess and check" method of solving recurrences: build an intuition for what you think the recurrence solves to, then verify it by trying to prove it with induction.

could solve it directly, though it is slightly trickier to deal with the ceiling and floor functions:

$$\begin{aligned} \left\lfloor \frac{n}{2^{k-1}} \right\rfloor &= 1 \\ n &\geq 2^{k-1} \\ \log_2 n &\geq k-1 \\ k &\leq \lfloor (\log_2 n) \rfloor + 1 = \lceil \log_2 (n) \rceil \end{aligned} \tag{9}$$

Thus the recurrence terminates at a depth of (at most) $\lceil \log_2 n \rceil$ yielding logarithmic complexity.

$$\begin{aligned} \mathcal{T}(n) &= 4 \lceil \log_2 n \rceil + 1 \\ &= 4 \lceil \log_2 (n) \rceil + 1 = O(\log n) \end{aligned} \tag{10}$$

Analysis to Code

Understanding the analysis of algorithms can really help you advance as a program because when you understand the efficiency limits of certain structures like basic counting loops and recursive procedures, then you immediately understand the impact of certain design strategies on efficiency. You also give yourself targets in the design process. If you are shooting for something logarithmic, then you know you can't just make little steps through a vector. You must find a way to take increasingly larger steps, to cut the problem size by a constant factor not a constant increment. Absent any other plan of attack, this kind of knowledge helps to give you a place to start.