Recurrence Relations & The Master Theorem

Suhas Arehalli

COMP221 - Spring 2024

1 Analyzing Runtimes of Recursive Functions

- If you attempt to use our previous techniques for determining runtime growth functions on a recursive function, you may have run into difficulty when you have to figure out the runtime of recursive calls!
- Consider MergeSort in Alg. 1

Algorithm 1 Mergesort, with Merge's structure implied. See the MergeSort notes for details on Merge!

```
function MergeSort(Array $A$)
    if $N \leq 1$ then
        return $A$
    end if
    $m \leftarrow \lceil \frac{N}{2} \rceil$
    $l \leftarrow$ MergeSort($A[1 \ldots m]$)
    $r \leftarrow$ MergeSort($A[m+1 \ldots N]$)
    return Merge($l, r, N$)
end function
```

- Let $T(n)$ represent the worst-case runtime growth function for MergeSort. Using the RAM model, we can see that this should be equal to the sums of the runtimes of each line!
- Merge, as we saw in the mergesort lecture, should take $\Theta(n)$ time, which dominates the constant time it takes to compute m in the first line and check the conditional.
- But it might be unclear how to handle the remaining two recursive calls.
  - In the the RAM model, function calls take time proportional to the time the code that composes the function takes
  - But since it's a recursive function, that's the entire block of code we're currently analyzing!
  - A somewhat tricky way to handle this is to observe that each line takes the amount of time it takes for Mergesort to handle an array of size $\lceil \frac{N}{2} \rceil$ — Luckily we defined T such that this is approximately (assuming N is even) $T(\frac{N}{2})$!

  - It's also critical to note that this is specifically handling the recursive case of MergeSort! If $N = 0, 1$, then the algorithm only takes constant time!

- This allows us to write a *recurrence relation* - A definition of a function that is written in terms of itself! — to define the time complexity of MergeSort!

$$T(n) = \begin{cases} 2T(\frac{n}{2}) + f(n), & n > 1 \\ g(n), & n \leq 1 \end{cases}$$

  Where $f(n) \in \Theta(n)$ and $g(n) \in \Theta(1)$.

- Giving names to functions in a particular time complexity class can sometimes get tedious, so you'll often see notation simplified to look like

$$T(n) = \begin{cases} 2T(\frac{n}{2}) + \Theta(n), & n > 1 \\ \Theta(1), & n \leq 1 \end{cases}$$

  which you should understand as a shorthand for the earlier definition!

- We can do the same for something like Binary Search! Consider Alg. 2

Algorithm 2 Binary Search

```
function BinarySearch(Array $A$, target $e$)
    if $N == 0$ then
        return NULL
    end if
    $m \leftarrow \lceil \frac{N}{2} \rceil$
    if $A[m] == e$ then
        return $m$
    else if $e \leq A[m]$ then
        return BinarySearch($A[1 \ldots m-1], e$)
    else
        return BinarySearch($A[m+1 \ldots N], e$)
    end if
end function
```

- Each recursive call will be on an input of size $\frac{N}{2}$, and additional work can be done in constant time, giving us a recurrence relation for BinarySearch's worst-case time complexity of

$$T(n) = \begin{cases} T(\frac{n}{2}) + \Theta(1), & N > 0 \\ \Theta(1), & N = 0 \end{cases}$$

- These recurrence relations are nice, but they don't give us neat descriptions of the algorithm's total runtime. We want to eventually have a non-recursive definition of the time complexity of BinarySearch or MergeSort.

- One method to do that is to *unroll* the recurrence. That is, substitute for the recursive call until we get a base case! For Binary Search

$$\begin{aligned}T(n) &= T(\frac{n}{2}) + \Theta(1)\\ &= (T(\frac{n}{4}) + \Theta(1)) + \Theta(1)\\ &= ((T(\frac{n}{8}) + \Theta(1)) + \Theta(1)) + \Theta(1)\end{aligned}$$

  Seeing this, we can generalize to, for some k less than or equal to when $\frac{n}{2^k}$ becomes a base case,

$$T(n) = T(\frac{n}{2^k}) + k\Theta(1)$$

  which suggests a strategy where we pick k such that $T(\frac{n}{2^k}$ becomes a base case. This is roughly when $k = log_2 n + 1$, since $\frac{n}{2n}$ rounds down to 0 (the number of elements in an array is an integer, after all), and thus we have that

$$\begin{aligned}T(n) &= \Theta(1) + (log_2 n + 1)\Theta(1)\\ &= \Theta(logn)\end{aligned}$$

  As we expect for binary search!

- We can also view this as a recursion tree, where we construct a tree where each node represents the work done by a recursive call, and it's children represent the recursive calls that that recursive call makes.

  For example, for MergeSort, we observe that in the first call, we do $\Theta(n)$ work outside of calls, and make two recursive calls, so we draw a tree with a root labeled $\Theta(n)$ to indicate the work done by the first call, and give it two children.

  Those children will do $\Theta(\frac{n}{2})$ work (dominated by a call of size $\frac{n}{2}$ to MERGE), and each make two calls, and so on until a base case is reached. We can then start to estimate the total amount of work done at each level of the tree (In this case, roughly $\Theta(n)$ work at each level!), and count the number of levels of the tree (Since the tree splits the problem size in half each level, roughly $\log_2 n$ levels!). See Fig. 1.

  Our goal with this tree is to visualize the recursive structure of the algorithm, where each node represents work done at some point during the algorithm's execution.

- We can think of the total work done by this algorithm as the sum of all of the values in the recursive tree, which here would look something like

$$\sum_{k=1}^{\log n} 2^k \Theta(\frac{n}{2^k})$$

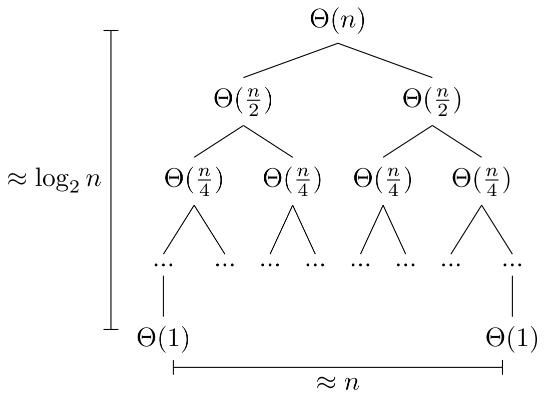

Figure 1: A recursion tree for a call to MergeSort of size n

Which, if we let n_0, c be the constants such that the unnamed function is $\Theta(\frac{n}{2^k})$, we can solve to get, for $n \geq n_0$

$$\begin{aligned}\sum_{k=1}^{\log n} 2^k \cdot c\frac{n}{2^k} &= \sum_{k=1}^{\log n} cn \\ &= cn \log n \in \Theta(n \log n)\end{aligned}$$

As we expect.

- We can also see that this is the product of the number of levels (represented by the index of the summation), and the amount of work done at each level (the argument of the summation), since each level was perfectly balanced (i.e., the k term cancelled out). This is what happens when our recursion tree is nicely balanced!

2 The Master Theorem

- Techniques like unrolling and inspecting recurrence trees are useful to try and find patterns to solve for the true time complexity growth functions of recursive algorithms, but they can be tedious to do.
- Luckily, many of these analyses go in very similar ways, to the point where we can prove the results of these analyses follow a specific template if we can coerce our recurrence relations into a particular form.
- This result is called the *Master Theorem*

Theorem 2.1 (The Master Theorem) *Given a recurrence relation*

$$T(n) = aT(\frac{n}{b}) + f(n)$$

where $a, b \in \mathbb{N}$*,*

Case 1: If $f(n) \in \Theta(n^{\log_b a})$, then $T(n) \in \Theta(n^{\log_b a} \log n)$

Case 2: If $f(n) \in O(n^c)$, where $c < \log_b a$, then $T(n) \in \Theta(n^{\log_b a})$

Case 3: If $f(n) \in \Omega(n^c)$ where $c > \log_b a$ and $af(\frac{n}{b}) \leq kf(n)$ for $0 \leq k \leq 1$, then $T(n) = \Theta(f(n))$

- To understand the Master Theorem, it's important to get a handle on the value $n^{\log_b a}$, which shows up in each case!
  - This is *the number of leaves (i.e., base cases!) in the recursive tree formed by a call of size n*
  - Since each base case takes $\Theta(1)$ time to complete, this means just solving all of the base cases takes $\Theta(n^{\log_b a})$ time!
  - To see why this is the number of leaves, you should observe that this is actually just

    $$\begin{aligned} n^{\log_b a} &= a^{\log_a (n^{\log_b a})} \\ &= a^{\log_b a \log_a n} \\ &= a^{\log_b n} \end{aligned}$$

    all by applying a few of logarithm properties ($a^{\log_a x} = x$, and the change-of-base formula).

    Then note that a is the *branching factor* of the tree (how many children each node will have) and $\log_b n$ is the depth of our recursive tree!

- It's also useful to think of what's happening in each case with respect to our recursion tree!
  1. In Case 1, we have something that looks like MergeSort! The amount of work done at each node ($f(n)$) is proportional to the amount of work done by all of the leaves in the subtree rooted at that node ($\Theta(n^{\log_b a})$, and so the amount of work done at each level will be proportional! In this case, our total runtime is the work done at each level ($\Theta(n^{\log_b a})$) multiplied by the number of levels ($\Theta(\log n)$)!
  2. In Case 2, the work done at each node ($f(n)$) is *strictly*[1] less than the work done by the leaves in it's subtree. This means that the work done in the tree is dominated by work done at the leaves! Thus, our runtime complexity is simply the cost of handling all of our base cases: $\Theta(n^{\log_b a})$!
  3. Case 3 is a bit more complex. Here, we have that the work done at each node strictly dominates the work done by the leaves it generates. This alone gets us nowhere though, since this just tells us that the leaves won't dominate and nothing about how to sort out all of the internal layers! We need an additional assumption: That for any given node, the work done by it's children ($af(\frac{n}{b})$) is only some fraction k of the work done at that node itself ($f(n)$).

     Together, this information tells us that the root's time complexity dominates: The base cases don't dominate, and the work at lower levels is just some fraction of work at a higher level, so our time complexity is just the work done at the highest level, $\Theta(f(n))$!

[1] Note that this is the purpose of c (and ε in Skiena's variant)! The exponent of n must be strictly smaller than $\log_b a$!