Lecture Notes for CSCI 311: Algorithms
Set 5-Recurrences

Professor Talmage

February 12, 2025

---

1 Recursion Trees

Recall `MergeSort`:

```
1: function MergeSort(X = [x_1, ..., x_n])
2:     if |X| ≤ 1 then return X
3:     end if
4:     Left = [x_1, ..., x_{⌊n/2⌋}]
5:     Right = [x_{⌊n/2⌋+1}, ..., x_n]
6:     L = MergeSort(Left)
7:     R = MergeSort(Right)
8:     return Merge(L, R)
9: end function
```

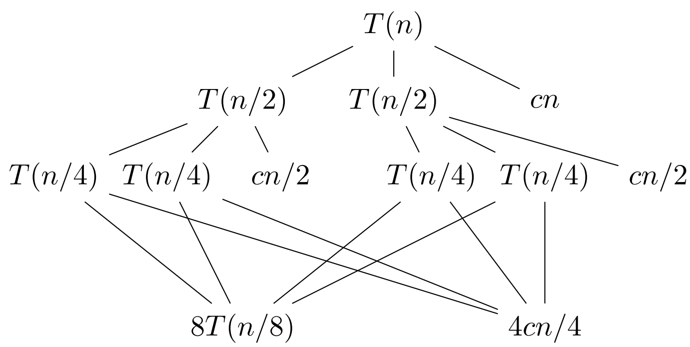
Consider the runtime of `MergeSort`. This is not entirely obvious, as we cannot express it without the runtime of the recursive calls to `MergeSort` on lines 6 and 7. If we name this, we can use it, however. Let $T(n)$ denote the runtime of `MergeSort` on input of size n. then

$$T(n) = T(\lfloor n/2 \rfloor) + T(n - \lfloor n/2 \rfloor) + O(n) + O(n) = 2T(n/2) + O(n)$$

- The two "halves" differ from $n/2$ by at most 1, so it works out to ignore the small difference. Another perspective is that this math works exactly when n is a power of 2, and the in-between cases do not cost exorbitantly more. See Section 4.6 of the textbook for detailed analysis.
- The two $O(n)$ terms come from splitting and merging the sorted lists. While it is possible to split in place and reduce that cost to $O(1)$, we see in the last equality that it does not reduce the asymptotic growth of $T(n)$ because of the $O(n)$ cost to merge two sorted lists.

This expression for $T(n)$, while it does contain all the costs of the pseudocode, is not sufficient. We need to reduce it further, eliminating the recursive dependency, to find a *closed-form expression* for $T(n)$. To do this, we will start by expanding it a few times.

- We need to replace the anonymous asymptotic functions, or we can break the substitution for $T(n)$ by growing constants hidden in the big-O.
- If a function is $O(n)$, then for sufficiently large n, it is less than or equal to cn, for some constant c, so we can replace $O(n)$ with cn.



Note that each node in the tree is either replaced by substituting the recurrence again or is non-recursive (closed form). If we add all the leaves, we will get the total value of the function.

1. Add the non-recursive costs at each level.
2. Look for a pattern. Here, each level sums to cn.
   - Logically, at each level of the tree, each element will be part of one split and one merge across all the recursive calls, so $O(n)$ is reasonable for the sum.
3. Sum all levels
   (a) Recursion stops when we get to $T(1)$. $T(1) = O(1)$, since we do not need to sort anything.
   (b) We reach $T(1)$ when $\frac{n}{2^i} = 1$, which is when $i = \log_2 n$.
   (c) We will have $2^i = 2^{\log_2 n} = n$ leaves, each $O(1)$ at that lowest level.
4. Total is thus cost per level times number of levels, plus the cost of the $T(1)$ leaves:

$$T(n) = (cn)(\log_2 n) + (n)(O(1)) = O(n \log n)$$

This is the *Recursion Tree* method for solving recurrences. It is good for building intuition when the recurrence behaves fairly nicely.

Aside: We will generally assume that $T(1) = O(1)$ for any algorithm. Thus, when solving recurrences, if there is no base case given, you can assume that once the input size is constant, the runtime will be constant.

Exercise: Find a closed-form, asymptotic upper bound for the following recurrences:

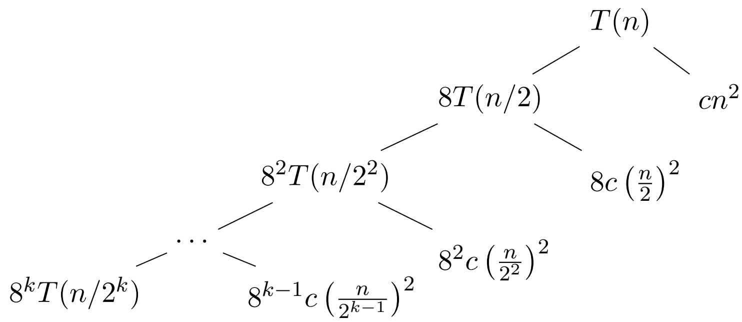
1. $T(n) = 8T(n/2) + \Theta(n^2)$
2. $T(n) = 3T(n/9) + \Theta(n^4)$
3. $T(n) = 4T(n/2) + n$

Solution:

1. First, we remove the anonymous asymptotic, limiting our attention to upper bounds (as directed): $T(n) \leq 8T(n/2) + cn^2$, for some constant $c > 0$. We can then draw a few levels of the recursion tree and look for patterns. It is important to be careful with the substitution at each level, and to keep the coefficient of each $T(...)$ term, as that coefficient grows, and apply

Solution, continued

it to both child nodes.



We can now solve for the number of levels, k. The base case is assumed to be $T(1) = O(1)$, so we need to find k s.t. $n/2^k = 1$. This is true when $k = \log_2 n$. Substituting this into the exponent of the coefficient 8, we get $8^{\log_2 n}O(1) = O(n^3)$ for the recursive portion of the runtime.

Next, we sum the non-recursive costs from each level:

$$\sum_{i=1}^{\log_2 n} 8^{i-1}c\left(\frac{n}{2^{i-1}}\right)^2 = cn^2 \sum_{j=0}^{\log_2 n-1} \left(\frac{8}{2^2}\right)^j$$
$$= cn^2 \sum_{j=0}^{\log_2 n-1} 2^j$$

If we extended this sum to infinity, it would diverge, so we cannot do that. What we can do is notice that the last term of the sum is $2^{\log_2 n-1} = n/2$. If we reverse the sum, we find that it equals $n/2 + n/4 + ... + 2 + 1$. This is a geometric series with a constant factor of $1/2 < 1$, so we can apply the geometric sum rule.

$$cn^2 \sum_{j=0}^{\log_2 n-1} 2^j = cn^2\left(\frac{n/2}{1-1/2}\right)$$
$$= cn^2(n) = cn^3$$

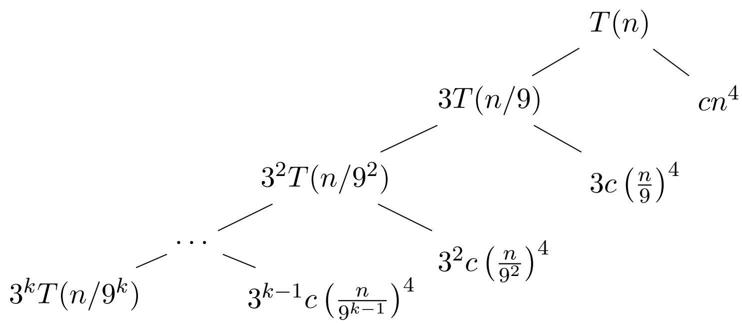
Finally, we add the recursive and non-recursive costs to get the total value of $T(n)$:

$$T(n) \leq O(n^3) + cn^3 = O(n^3)$$

2. $T(n) = 3T(n/9) + \Theta(n^4)$

   Again, we remove the anonymous asymptotic, yielding $T(n) \leq 3T(n/9)+cn^4$, for some constant c. Drawing the tree:

Solution, continued



Solving for the number of levels, we will continue at most until $n = 1$, which occurs when $n/9^k = 1$, or $k = \log_9 n$. The recursive cost is thus $3^{\log_9 n} * O(1) = O(n^{\log_9 3}) = O(n^{.5})$.

For the non-recursive costs, we sum across the levels:

$$\sum_{i=1}^{\log_9 n} 3^{i-1}\left(\frac{n}{9^{i-1}}\right)^4 = n^4 \sum_{i=1}^{\log_9 n} \frac{3^{i-1}}{9^{4(i-1)}}$$
$$= n^4 \sum_{i=1}^{\log_9 n} 3^{-7i+7}$$

This sum is a geometric series starting at one and with common factor 3^{-7}. We extend the sum to infinity for convenience (which adds only a very small constant factor), yielding $n^4 \frac{1}{1-3^{-7}} = O(n^4)$, since the fraction is a constant slightly more than 1.

To get the total value of the function, we add the recursive and non-recursive costs: $O(n^{.5}) + O(n^4) = O(n^4)$.

There are two other general methods to solve a recurrence we will use in this course:

1. Recursion Tree: Draw a few levels of expanding the recurrence, look for patterns, sum recursive (base case) and non-recursive costs.
2. Substitution: Once you have a guess for a bound, prove with induction.
   - Substitution can be used for guess and check, or pairs well with the recursion tree method to prove that that intuition is correct.
3. Master Theorem: Doesn't *always* apply, but usually does and is the easiest, most reliable method.

2 Substitution

Since induction is the inverse of recursion, induction is a great way to prove a property of a recurrence (such as a bound). These proofs follow a pretty standard formula. Note that we will prove the slightly stronger claim that $T(n) \le dg(n)$, instead of directly proving big-Oh, as defining d gives us a very useful lever.

Example: `Mergesort` We can verify our claim that $T(n) = O(n \log n)$. Recall that the runtime function of `MergeSort` is $T(n) = 2T(n/2) + O(n)$.

Claim 1. $T(n) = 2T(n/2) + cn = O(n \log n)$

Proof. We will show that $T(n) = 2T(n/2) + cn$ is less than or equal to $dn \log n$ for some positive constant d for all $n \geq 2$.

- BC: $T(2) = 2T(1) + c(2) \leq d(2)\log(2)$ for sufficiently large d. This follows since $T(1)$ is constant. We start our induction at 2, because $\log 1 = 0$. This means that we also need $n = 3$ as a base case, so that our dependence on $n/2$ will be covered.

  $T(3) = T(1) + T(2) + c(3) = 3T(1) + 5c \leq d(3)\log(3)$ for sufficiently large d.

- IH: Assume that for an arbitrary $n > 3$, for all $2 \leq k < n, T(k) \leq dk \log k$. (Note that we need strong induction to assume $n/2$.)

- IS:

$$\begin{aligned} T(n) &\leq 2T(n/2) + cn \\ &\leq 2(d\frac{n}{2}\log\frac{n}{2}) + cn \\ &= dn(\log n - \log 2) + cn \\ &= dn\log n + (c - d\log 2)n \end{aligned}$$

  This is $\leq dn \log n$ if $d \log 2 \geq c$, which is true for sufficiently large d. Thus, we have the claim by strong induction, and $T(n) = O(n \log n)$.

□

Exercise: Prove that $T(n) = 4T(n/2) + 18n + 5$ is $O(n^2)$.

Proof. We show that $T(n) \leq dn^2$ for $n \geq 1$.

- BC: $4T(1) + 18(1) + 5 = 4a + 23$ for some constant $T(1) = a$. This is less than or equal to $d(1^2)$ for $d \geq 4a + 23$, a constant.

- IH: Assume $T(k) \leq dn^2$ for all $1 \leq k < n$.

- IS:

$$\begin{aligned} T(n) &= 4T(n/2) + 18n + 5 \\ &\leq 4\left(d\left(\frac{n}{2}\right)^2\right) + 18n + 5 \\ &\leq dn^2 + 18n + 4 \\ &\not\leq dn^2 \end{aligned}$$

  The induction fails at this point. While we do have $T(n) = O(dn^2)$, we have increased the hidden constants in the big-Oh. At each level of recursion, these constants would grow, possible enough to be overall more than a constant factor. Besides, they are constants, so should not be growing.

We can fix this by changing our specific claim. If $T(n) \leq dn^2 - fn$, for d and f constants, then $T(n) = O(n^2)$. By subtracting the lower-order term, we are actually strengthening our claim, which is counterintuitively easier to prove.

- BC: $4T(1)+18(1)+5 = 4a+23$ for some constant $T(1) = a$. This is less than or equal to $d(1^2)-f(1)$ for appropriate d, f.

- IH: Assume $T(k) \leq dn^2 - fn$ for all $1 \leq k < n$.
- IS:

$$\begin{aligned} T(n) &= 4T\left(\frac{n}{2}\right) + 18n + 5 \\ &\leq 4\left(d\left(\frac{n}{2}\right)^2 - f(n/2)\right) + 18n + 5 \\ &\leq dn^2 - 2fn + 18n + 5 \\ &\leq (dn^2 - fn) + (18 - f)n + 5 \end{aligned}$$

If we choose $f = 23$, then for $n \geq 1$, $(18-f)n+5 \leq 0$, so we have $T(n) \leq dn^2 - fn$. Thus, by strong induction, $T(n) = O(n^2)$.

□

In general, if you are doing an inductive proof of a recurrence and find yourself with extra terms preventing you from showing the required inequality, strengthen your claim by subtracting a lower-order term that will cancel out the extra terms.

3 Master Theorem

Theorem 1. *Let $a \geq 1, b > 1$ be constants, $f(n)$ a positive function, and $T(n) = aT(n/b) + f(n)$, for $n \in \mathbb{Z}^+$. Then $T(n)$ is bounded as follows:*

1. *If $f(n) = O\left(n^{\log_b a - \epsilon}\right)$, then $T(n) = \Theta\left(n^{\log_b a}\right)$.*
2. *If $f(n) = \Theta\left(n^{\log_b a} \log^k n\right)$ for a constant $k > 0$, then $T(n) = \Theta\left(n^{\log_b a} \log^{k+1} n\right)$.*
   - *Most often, $k = 0$, and we have $f(n) = \Theta\left(n^{\log_b a}\right)$, which gives $T(n) = \Theta\left(n^{\log_b a} \log n\right)$.*
3. *If $f(n) = \Omega\left(n^{\log_b a + \epsilon}\right)$, then $T(n) = \Theta\left(f(n)\right)$.*
   - *if $af(n/b) \leq cf(n)$ for some constant $c < 1$ and sufficiently large n.*

Usage:

- Extract a, b, f from recurrence.
- Compare $f(n)$ vs $n^{\log_b a}$:
  1. $f(n)$ smaller: recursive cost dominates: $\Theta(n^{\log_b a})$.
  2. nearly equal, only log-factor difference: recursive and non-recursive costs balance, extra $\log n$ factor: $\Theta(n^{\log_b a} \log^{k+1} n) = \Theta(f(n) \log^{k+1} n)$.
  3. $f(n)$ larger: non-recursive costs dominate: $\Theta(f(n))$.

Exercise: Use the Master Theorem to find closed-form expressions for each of the following:

(a) $T(n) = 3T(n/9) + \Theta(n^4)$

(b) $T(n) = 2T(n/2) + 17n$

(c) $T(n) = 4T(n/2) + n^3$

(d) $T(n) = 3T(n/4) + n \log n$

(e) $T(n) = 2T(n/4) + \sqrt{n}$

Be careful: the Master Theorem does not always apply!

- For cases (1) and (3), there must be a polynomial difference between f and $n^{\log_b a}$.
- Consider $T(n) = 2T(n/2) + \frac{n}{\log n}$.
- Difference is a factor of $\log^{-1} n$, which is smaller than n^ϵ for any positive $\epsilon > 0$, but does not fit in case (2), since $k < 0$.
- Rule 3 also has an extra condition that can make the theorem not apply when a is large and b is small.