CS 6363.003.21S Lecture 4—January 28, 2021

Main topics are #asymptotic_notation and #recurrances.

Run Time Analysis

- Las time, we saw two different sorting algorithms, and they both happen to be based on divide-and-conquer. But uh, which one's better?
- It's a tricky question, because quicksort tends to run very well in practice thanks to how most computers are architected, but as we'll see, there are choices of array for which quicksort performs quite poorly. There's also the question of how big n should be before we start caring. All algorithms are fast when n = 2.
- For this class, we're going to focus on what happens when n grows very very large, and to make sense of running time growth relative to n's growth, we're going to focus on how fast running times grow *asymptotically*.

"Big-oh" notation

- Let's review the concept. Let f(n) : N ➔ R^+ be a positive function over the natural numbers; perhaps f(n) represents the running time of an algorithm whose input size is n.
- Let g(n) : N ➔ R^+ be some other positive function. Suppose we believe f to grow no quick than g *up to constant factors*. We say f(n) is *in* or *equals* O(g(n)) ("big-Oh g of n") if, after n grows large enough "to care", f(n) is smaller than some constant multiple of g(n).
- Formally, O(g(n)) is a set of functions defined as

$$O(g(n)) = \{f(n) : \text{there exist positive constants } c \text{ and } n_0 \text{ such that } 0 \le f(n) \le cg(n) \text{ for all } n \ge n_0\}.$$

- Note that c and n_0 are defined separately *for each choice of function f*. If you think a function f is in O(g(n)), you're free to choose any gargantuinely large choice of n_0 you'd like and set c to be as large as necessary to prove yourself correct. But you still need to make sure your choice of c continues to work for *all* n ≥ n_0.
- O-notation provide a *loose* upper bound on how fast a function grows. Think of it as a less-than-or-equal symbol. So 256n in O(n) (c = 256), but also n in O(n^2). The looseness is convenient when all you can prove is a running time like O(n^2 log n) even when the real running time is sometimes smaller, like O(n^2).

Other notations

- But maybe we don't always want a loose upper bound. Maybe we want a loose lower bound instead. For example, maybe we want to emphasize that an algorithm will always required *at least* a certain amount of time in the worst case. For that, we have big-Omega.

$$\Omega(g(n)) = \{f(n) : \text{there exist positive constants } c \text{ and } n_0 \text{ such that } 0 \le cg(n) \le f(n) \text{ for all } n \ge n_0\}.$$

- Or maybe, you know exactly how fast a function grows up to constant factors. For that, we have big-Theta. Set Theta(g(n)) = O(g(n)) cup Omega(g(n)). We say g(n) is an *asymptotically tight bound* for f(n) in this case.
- Finally, there are situations where a loose bound isn't what you want, and you'd instead like a strict bound. Informally, o(g(n)) ("little-oh g of n") contains functions that are in O(g(n)), but aren't in Theta(g(n)). These functions grow more slowly than g(n), and g(n) will eventually beat them, no matter what constant c you try to multiply by. little-omega is the lower bound counterpart.

Working with Asymptotics

- The course website contains some typeset lecture notes on asymptotic notation. I highly highly recommend you go over them as we're going to be applying a lot of simple rules without much fanfare pretty much any time we analyze an algorithm. But here are some highlights anyway.
- If f(n) in O(g(n)) and g(n) in O(h(n)), then f(n) in O(h(n)) as well.
- Suppose f_1(n) = O(g_1(n)) and f_2(n) in O(g_2(n)). Then,

$$c \cdot f_1(n) = O(f_1(n)) \text{ for any positive constant } c, \tag{3.1}$$
$$f_1(n) + f_2(n) = O(g_1(n) + g_2(n)), \tag{3.2}$$
$$f_1(n) \cdot f_2(n) = O(g_1(n) \cdot g_2(n)), \text{ and} \tag{3.3}$$
$$f_1(n) + f_2(n) = O(\max\{g_1(n), g_2(n)\}). \tag{3.4}$$

- It's pretty common to abuse notation and put asymptotics in the middle of equations or inequalities. Writing f(n) = O(g(n)) when we really mean f(n) in O(g(n)) is a common special case. The general rule when you see such a thing is that *for all* choices of functions on the left falling within the various asymptotic sets, there *must exist* a choice of functions on the right making the inequality true.
- Finally, there are some classes of functions that we really care about with asymptotic notation. *Polynomially bounded functions* are those f(n) = O(n^k). The bigger the k, the slower the function. Exponential functions grow faster and we generally try to avoid them: n^k = o(a^n) for any constant k and constant a > 1. Also, a^n = o(c^n) for any c > a > 0. Finally, polylogarithmically bounded functions are better than polynomials: log^ell n = o(n^k) for any constant ell and constant k > 0. Also, logs of different bases differ only by a constant, which gets absorbed by the big-Oh if the log appears as a factor in runtime (i.e., not in an exponent).
- To finish this review, let's analyze that procedure FibonacciMultiply from last Tuesday, but for the special case of n = m.

```
FIBONACCIMULTIPLY(X[0 .. m - 1], Y[0 .. n-1]):
  hold ← 0
  for k ← 0 to n + m − 1
      for all i and j such that i + j = k
          hold ← hold + X[i] · Y[j]
      Z[k] ← hold  mod 10
      hold ← ⌊hold/10⌋
  return Z[0..m + n − 1]
```

- Here, we note there are at most k + 1 ≤ 2n = O(n) choices of i and j per value k in the inner for loop. We spend O(1) (constant) time multiplying and adding a constant number of digits to hold in the loop, so the inner loop takes O(n) time per iteration of the outer loop. Setting Z[k] and removing the modulus from hold takes an additional constant amount of time, so still O(n) for all inner iterations.
- But then there are at most 2n - 1 = O(n) iterations of the outer loop, so the whole algorithm takes O(n) * O(n) = O(n^2) time.

Analyzing Divide-and-Conquer Algorithms

- Unfortunately, divide-and-conquer algorithms require a bit more work to analyze. Consider the mergesort from last Tuesday.

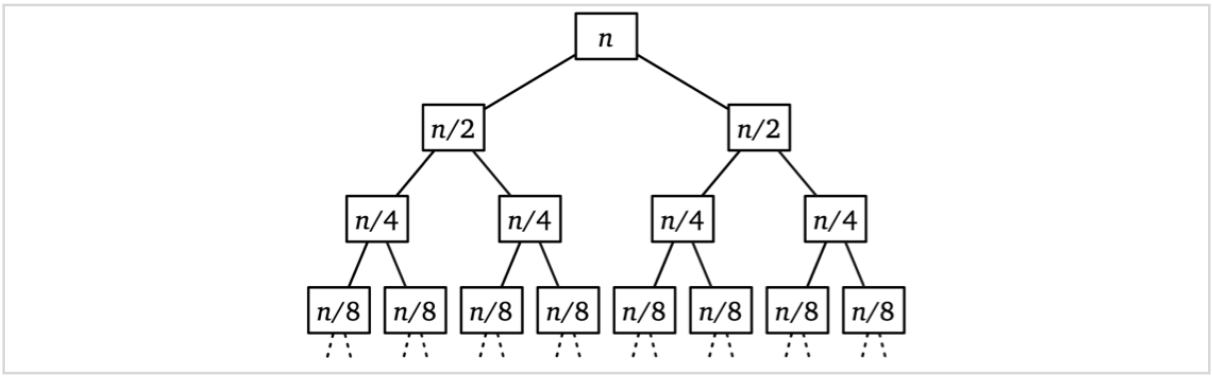
```
MERGESORT(A[1 .. n]):
  if n > 1
      m ← ⌊n/2⌋
      MERGESORT(A[1 .. m])          ⟨⟨Recurse!⟩⟩
      MERGESORT(A[m + 1 .. n])      ⟨⟨Recurse!⟩⟩
      MERGE(A[1 .. n], m)
```

```
MERGE(A[1 .. n], m):
  i ← 1;  j ← m + 1
  for k ← 1 to n
      if j > n
          B[k] ← A[i];  i ← i + 1
      else if i > m
          B[k] ← A[j];  j ← j + 1
      else if A[i] < A[j]
          B[k] ← A[i];  i ← i + 1
      else
          B[k] ← A[j];  j ← j + 1
  for k ← 1 to n
      A[k] ← B[k]
```

- To analyze any divide-and-conquer algorithm, including mergesort, we need to write and solve a recurrence for its running time.
- Let T(n) denote the worst-case running time for MergeSort(A[1 .. n]), whatever it is.
- MergeSort does an O(n) time for loop plus it takes the time to do two recursive calls on arrays of size ceil{n / 2} and floor{n / 2}. We can typically ignore the floors and ceilings in divide-and-conquer recurrences, so T(n) = T(n) = 2T(n / 2) + O(n) when n is sufficiently large, and T(Theta(1)) = Theta(1). See Erickson for a formal proof that we can remove the floors and ceilings.
- Now we need to figure out what T(n) in simpler terms. We need some asymptotic bound that is true for all T(n) following that recurrence.

Recursion Trees

- We can use a recursion tree to solve this and similar recurrences.
- A *recursion tree* is a rooted tree that describes the contributions to a recurrence or the time spent in a recursive algorithm.
- Each node is a recursive subproblem called at some point during the algorithm's execution.
- A node's subtrees are the recursive subproblems called by that node, and the root is the top-level call to the algorithm.
- So in MergeSort, for example, we have the root representing the top call, and each node except for the leaves gets two children.
- The *value* of each node is the time spent by the recursive subproblem *excluding* other recursive calls.
- So recall the definition of big-Oh. For large enough n, T(n) ≤ 2T(n / 2) + cn for some constant c.
- So, we write cn in the root node. Each child call works on a problem of size n/2 so these nodes have value cn / 2.
- In general, a node at depth i gets a value of cn / 2^i, and there are 2^i nodes of depth i. I want to emphasize we're using the same constant c at each level. If we let the constant change depending on n, then it's not really a constant and we can easily cheat for a better analysis.

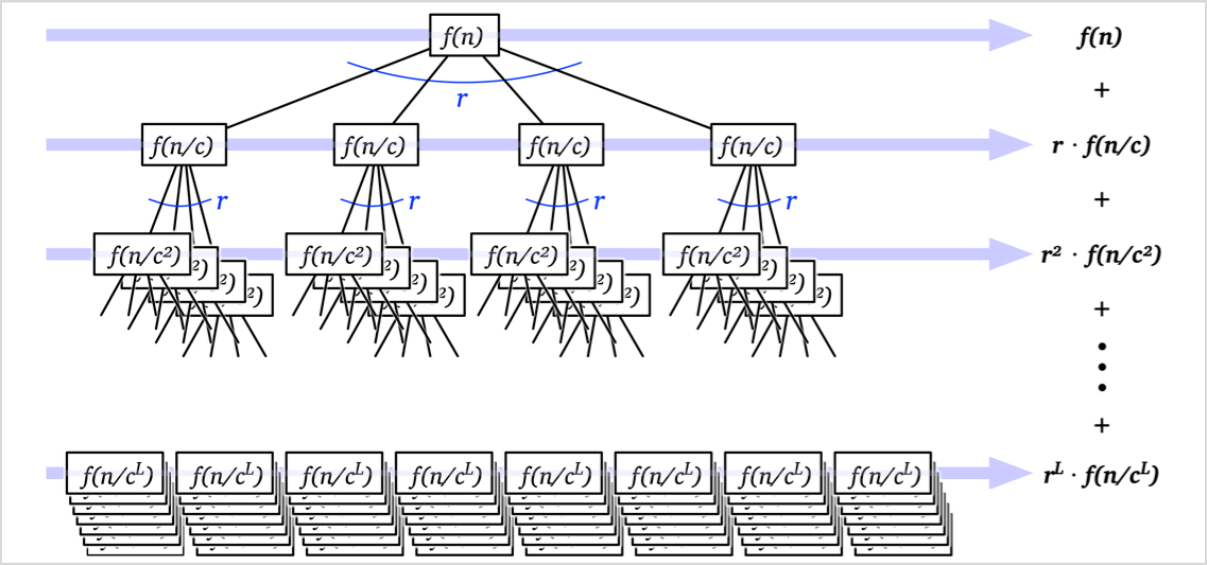


- The overall time spent by mergesort, the solution to T(n), is the time spent in all those recursive calls. We need to **sum** the value of all the nodes.
- The easiest way to evaluate this sum is to do so level-by-level. So what is the sum within each level?
- Each level (except the base cases) has a sum of exactly cn.
- We divide the problem size by 2 in each recursive call, so the depth or number of levels is lg n.
- So T(n) ≤ cn lg n. MergeSort runs in O(n log n) time.

- Let's discuss a more general case.
- Often, but not always, you'll be dealing with an algorithm that does f(n) non-recursive work

and makes r recursive calls, each on a subproblem of size n / c where c is a constant.

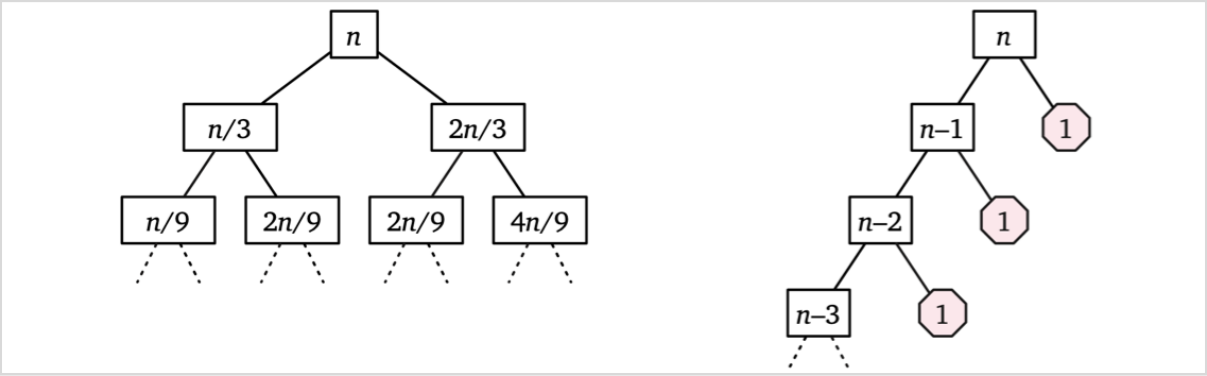
- These algorithms have run time recurrences that look like T(n) = r T(n / c) + f(n) with a base case of f(1) = Theta(1).
- In this case, each internal node of the recursion tree has r children.
- The root gets a value of f(n). The problem size at depth i is n / c^i, so those nodes get value f(n / c^i).
- Again, to compute T(n), we need to sum the values on each node, and its easiest do so level-by-level.
- There are r^i nodes at depth i, so the sum at that depth is r^i f(n / c^i).
- The easiest thing to do in practice is to draw the first few levels of the tree to see the big picture:



- The leaves of the recursion tree correspond to base cases of the algorithm. Since we're aiming for an asymptotic bound anyway, we'll assume the worst that the leaves go all the way down to instances of size n_0 = 1.
- T(n) is the sum of all node values, so
  - T(n) = sum_{i=0}^L r^i f(n / c^i) where L is the depth of the recursion tree.
- Our choice of n_0 = 1 means L = log_c n and there are r^L = r^{log_c n} = n^{log_c r} leaves. The values of the leaves sum to n^{log_c r} f(1) = \Theta(n^{log_c r}).
- There are three common cases where the level-by-level series (the sum over all node values) is easy to evaluate.
  - **Decreasing**: If the series *decays exponentially*, meaning each term is at most a constant < 1 times the previous one, then the sum is dominated by its first and largest term. T(n) = Theta(f(n)).
  - **Equal** If all the term in the series are equal, then T(n) = Theta(f(n) * L) = Theta(f(n) log n). Remember, the base of the log doesn't matter if you're just multiplying by it.
  - **Increasing**: If the series *grows exponentially*, meaning each term is at least a constant > 1 times the previous, then the sum is dominated by its last and largest term. T(n) =

Theta(n^{log_ c r}).

- Looking for these three specific cases, exponential decay, everything being equal, or exponential growth is a variant of the *Master Method* of solving recurrences taught in CLRS. However, I think working with the recursion trees is easier to remember. These three particular cases work almost every time you would use the master method, and recursion trees in general work in *more* situations than the master method.
- For example, recursion trees can be used in the case that not every recursive call is the same size.
- Suppose T(n) = T(n / 3) + T(2n / 3) + n. **[use the figure on the left]**



- Each *full* level of the recursion tree sums to n, and non-full levels sum to less.
- The tree has depth log_{3 / 2} n = O(log n), so T(n) = O(n log n).
- On the other hand, there at at least log_3 n = Omega(log n) full levels, so T(n) = Omega(n log n). Ah, so T(n) = Theta(n log n).
- We'll see more examples of unbalanced recursion trees on Tuesday.