COMP 285 (NC A&T, Spr '22) Lecture 3

Adapted From Virginia Williams' lecture notes. Additional credits: J. Su, W. Yang, Gregory Valiant, Mary Wootters, Aviad Rubinstein, Sami Alsheikh.

Asymptotics and Worst-Case Analysis

1 Asymptotic Notation

To talk about the running time of algorithms, we will use the following notation. $T(n)$ denotes the runtime of an algorithm on input of size n.

1.1 "Big-Oh" Notation:

Intuitively, Big-Oh notation gives an upper bound on a function. We say $T(n)$ is $O(f(n))$ when as n gets big, $f(n)$ grows at least as quickly as $T(n)$. Formally, we say

$$T(n) = O(f(n)) \iff \exists c, n_0 > 0 \text{ s.t } \forall n \geq n_0, 0 \leq T(n) \leq c \cdot f(n)$$

1.2 "Big-Omega" Notation:

Intuitively, Big-Omega notation gives a lower bound on a function. We say $T(n)$ is $\Omega(f(n))$ when as n gets big, $f(n)$ grows at least as slowly as $T(n)$. Formally, we say

$$T(n) = O(f(n)) \iff \exists c, n_0 > 0 \text{ s.t } \forall n \geq n_0, 0 \leq c \cdot f(n) \leq T(n)$$

1.3 "Big-Theta" Notation:

Intuitively, Big-Theta notation gives both a lower and upper bound on a function. We say $T(n)$ is $\Theta(f(n))$ if and only if $T(n) = O(f(n))$ and $T(n) = \Omega(f(n))$.

$$T(n) = O(f(n)) \iff \exists c_1, c_2, n_0 > 0 \text{ s.t } \forall n \geq n_0, 0 \leq c_1 f(n) \leq T(n) \leq c_2 f(n)$$

We can see that these notations really do capture exactly the behavior that we want - namely, to focus on the rate of growth of a function as the inputs get large, ignoring constant factors and lower order terms. As a sanity check, consider the following example and non-example.

Claim 1. *All degree-k polynomials*[1] *are* $O(n^k)$. *Proof.* Suppose $T(n)$ is a degree-k polynomial. That is, $T(n) = a_k n^k + \cdots + a_1 n + a_0$ for some choice of a_i's where $a_k \neq 0$. To show

[1] To be more precise, all degree-k polynomials T such that $T(n) \geq 0$ for all $n \geq 1$. How would you adapt this proof to be true for all degree-k polynomials T with positive leading coefficients?

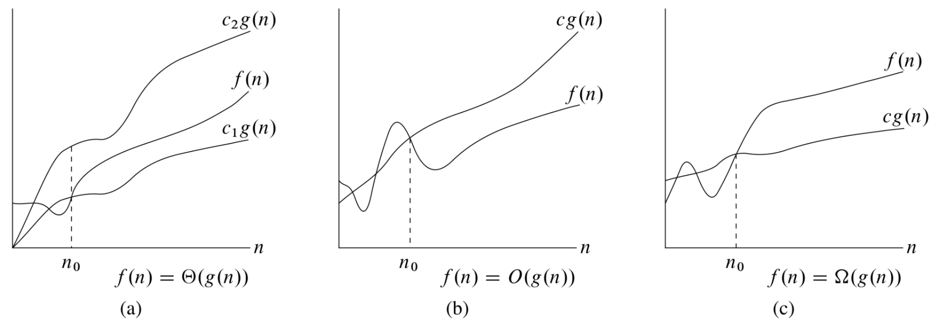


Figure 1: Figure 3.1 from CLRS - Examples of Asymptotic Bounds. (Note: In these examples, $f(n)$ corresponds to our $T(n)$ and $g(n)$ to our $f(n)$).

that $T(n)$ is $O(n^k)$ we must find a c and n_0 such that for all $n \geq n_0, T(n) \leq c \cdot n^k$. (Since $T(n)$ represents the running time of an algorithm, we assume it is positive.) Let $n_0 = 1$ and let $a^* = max_i|a_i|$. We can bound $T(n)$ as follows:

$$
\begin{aligned}
T(n) &= a_k n^k + \cdots + a_1 n + a_0 \\
&\leq a^* n^k + \cdots + a^* n + a^* \\
&\leq a^* n^k + \cdots + a^* n^k + a^* n^k \\
&= (k+1)a^* \cdot n^k
\end{aligned}
$$

Let $c = (k+1)a^*$ which is constant, independent of n. Thus, we've exhibited $c, n0$ which satisfy the Big-Oh definition, so $T(n) = O(n^k)$.

Claim 2. *For any $k \geq 1$, n^k is not $O(n^{k-1})$.* *Proof.* By contradiction. Assume $n^k = O(n^{k-1})$. Then there is some choice of c and $n0$ such that $n^k \leq c \cdot n^{k-1}$ for all $n \geq n_0$. But this in turn means that $n \leq c$ for all $n \geq n_0$, which contradicts the fact that c is a constant, independent of n. Thus, our original assumption was false and n^k is not $O(n^{k-1})$.

2 MergeSort

Recall the *Divide-and-conquer* paradigm from the second lecture. In this paradigm, we use the following strategy:

- Break the problem into sub-problemsn.
- Solve the sub-problems (often recursively)
- Combine the results of the sub-proboems to solve the big problem.

At some point, the sub-problems become small enough that they are easy to solve, and then we can stop recursing.

With this approach in mind, `MergeSort` is a very natural algorithm to solve the sorting problem.

The pseudocode is below:

```
MergeSort(A):
    n = len(A)
    if n <= 1:
        return A
    L = MergeSort( A[:n/2] )
    R = MergeSort( A[n/2:] )
    return Merge(L, R)
```

Above, we are using Python notation, so $A[: n/2] = [A[0], A[1], \ldots, A[n/2 - 1]]$ and $A[n/2 :] = [A[n/2], ..., A[n-1]]$. Additionally, we're using integer division, so n/2 means $\lfloor n/2 \rfloor$.

How do we do the `Merge` procedure? We need to take two sorted arrays, L and R, and merge them into a sorted array that contains both of their elements. See the slides for a walkthrough of this procedure.

```
Merge(L, R):
    m = len(L) + len(R)
    S = [ ]
    for k in range(m):
        if L[i] < R[j]:
            S.append( L[i] )
            i += 1
        else:
            S.append( R[j] )
            j += 1
    return S
```

Note: This pseudocode is incomplete! What happens if we get to the end of L or R? Try to adapt the pseudocode above to fix this.

As before, we need to ask: Does it work? And does it have good performance?

2.1 Correctness of MergeSort

Let's focus on the first question first. As before, we'll proceed by induction. This time, we'll maintain a *recursion invariant* that any time `MergeSort` returns, it returns a sorted array.

- **Inductive Hypothesis.** Whenever `MergeSort` returns an array of size $\leq i$, that array is sorted.
- **Base case.** Suppose that $i = 1$. Then whenever `MergeSort` returns an array of length 0 or length 1, that array is sorted. (Since all array of length 0 and 1 are sorted). So the Inductive Hypothesis holds for $i = 1$.

- **Inductive step.** We need to show that if `MergeSort` always returns a sorted array on inputs of length $leqi-1$, then it always does for length $\leq i$. Suppose that `MergeSort` has an input of length i. Then L and R are both of length $\leq i-1$, so by induction, L and R are both sorted. Thus, the inductive step boils down to the statement:

  "When Merge takes as inputs two sorted arrays L and R, then it returns a sorted array containing all of the elements of L, along with all of the elements of R."

  This statement is intuitively true, although proving it rigorously takes a bit of book-keeping. In fact, it takes another proof by induction! Check out CLRS Section 2.3.1 for a rigorous treatment.

- **Conclusion.** By induction, the Inductive hypothesis holds for all i. In particular, given an array of any length n, `MergeSort` returns a sorted version of that array.

2.2 Running time of MergeSort

Finally, we get to our first question in this lecture where the answer may not be intuitively obvious. What is the running time of `MergeSort`? In the next few lectures, we'll see a few principled ways of analyzing the runtime of a recursive algorithm. Here, we'll just go through one of the ways, which is called the `recursion tree` method.

The idea is to draw a tree representing the computation (see the slides for the visuals). Each node in the tree represents a subproblem, and its children represent the subproblems we need to solve to solve the big sub-problem. The recursion tree for `MergeSort` looks something like this:

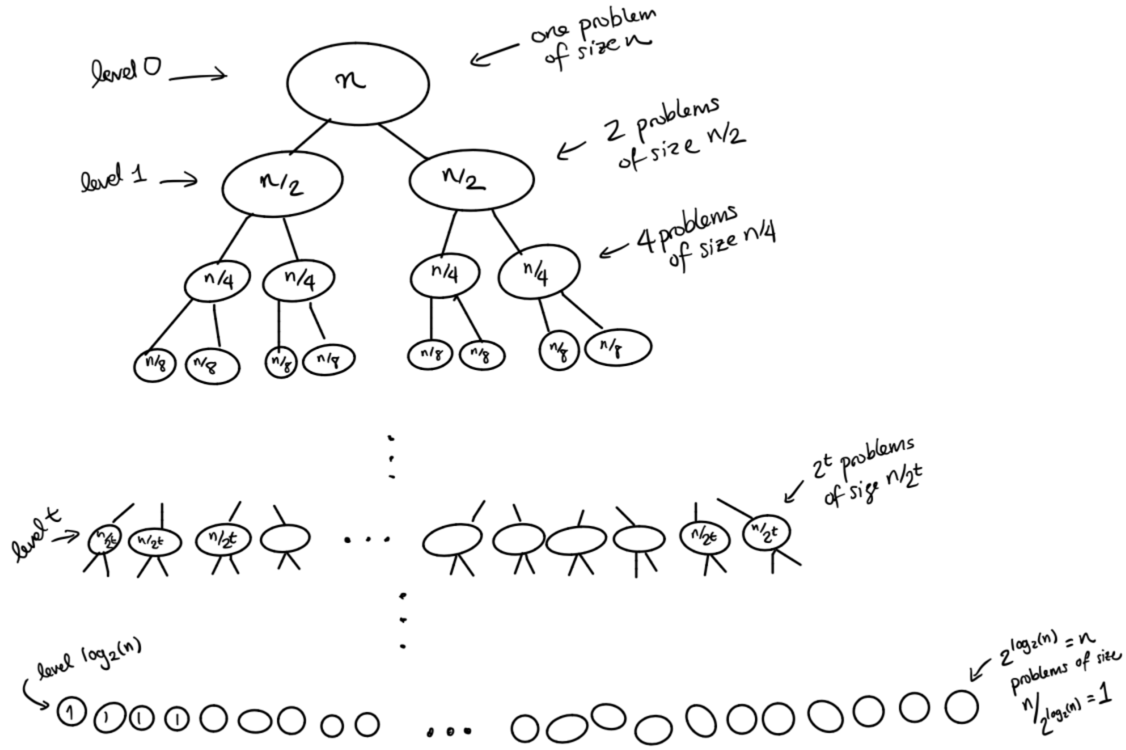
At the top (zeroth) level is the whole problem, which has size n. This gets broken into two sub-problems, each of size $n/2$, and so on. At the t'th level, there are 2^t problems, each of size $n/2^t$. This continues until we have n problems of size 1, which happens at the $log(n)$'th level.

Some notes:

- In this class, logarithms will **always** be base 2, unless otherwise noted.
- We are being a bit sloppy in the picture above: what if n is not a power of 2? Then $n/2^j$ might not be an integer. In the pseudocode above, we actually break a problem of size n into problems of size $\lfloor n/2 \rfloor$ and $\lceil n/2 \rceil$. Keeping track of this in our analysis will be messy, and it won't add much, so we will ignore it, and for now we will assume that n is a power of 2 [2]

In order to figure out the total amount of work, we will figure out how much work is done at each node in the tree, and add it all up. To that end, we tally up the work that is done

[2] To formally justify the assumption that n is a power of 2, notice that we can always sort a *longer* list of length $n' = 2^{\lceil log_2(n) \rceil}$. That is, we'll add extra entries, whose values are ∞, to the list. Then we sort the new list of length n', and return the first n values. Since $n' \leq 2n$ (why?) this won't affect the asymptotic running time. Also see CLRS Section 4.6.2 for a rigorous analysis of the original algorithm with floors and ceilings.



in a particular node in the tree—that is, in a particular call to `MergeSort`. There are three things:

1. Checking the base case
2. Making recursive calls (but we don't count the work actually done in those recursive calls; that will count in other nodes)
3. Running Merge.

Let's analyze each of these. Suppose that our input has size m (so that $m = n/2^j$ for some j).

1. Checking the base case doesn't take much time. For concreteness, let us say that it takes one operation to retrieve the length of A, and other operation to compare this length to 1, for a total of two operations.[3]
2. Making the recursive calls should also be fast. If we implemented the pseudocode well, it should also take a constant number of operations.

   Aside: This is a good point to talk about how we interpret pseudocode in this class. Above, we've written `MergeSort(A[:n/2])` as an example of a recursive call. This

[3] Of course, there's no reason that the "operation" of getting the length of A should take the same amount of time as the "operation" of comparing two integers. This disconnect is one of the reasons we use big-Oh notation.

makes it clear that we are supposed to recurse on the first half of the list, but it's not clear how we actually implement that. Our "pseudocode" above is in fact working Python code, and in Python, this implementation, while clear, is a bit inefficient. That is, written this way, Python will actually copy the first $n/2$ elements of the list before sending them to the recursive call. A much better way would be to instead just pass in pointers to the 0'th and $n/2-1$'st index in the list. This would result in a faster algorithm, but kludgier pseudocode. In this class, we generally will opt for cleaner pseudocode, as long as it does not hurt the asymptotic running time of the algorithm. In this case, our simple-but-slower pseudocode turns out not to affect the asymptotic running time, so we'll stick with this.

In light of the above Aside, let's suppose that this step takes $m+2$ operations, $m/2$ to copy each half of the list over, and 2 operations to store the results. Of course, a better implementation of this step would only take a constant number (say, four) operations.

3. The third thing is the tricky part. We claim that the Merge step also takes about m operations.

   Consider a single call to Merge, where we'll assume the total size of A is m numbers. How long will it take for Merge to execute? To start, there are two initializations for i and j. Then, we enter a for loop which will execute m times. Each loop will require one comparison, followed by an assignment to S and an increment of i or j. Finally, we'll need to increment the counter in the for loop k. If we assume that each operation costs us a certain amount of time, say $Cost_a$ for assignment, $Cost_c$ for comparison, $Cost_i$ for incrementing a counter, then we can express the total time of the Merge subroutine as follows:

$$2Cost_a + m(Cost_a + Cost_c + 2Cost_i)$$

   This is a precise, but somewhat unruly expression for the running time. In particular, it seems difficult to keep track of lots of different constants, and it isn't clear which costs will be more or less expensive (especially if we switch programming languages or machine architectures). To simplify our analysis, we choose to assume that there is some global constant c_{op} which represents the cost of an operation. You may think of c_{op} as $\max\{Cost_a, Cost_c, Cost_i, \ldots\}$. We can then bound the amount of running time for Merge as

$$2c_{op} + 4c_{op}m = 2 + 4m \text{ operations}$$

Thus, the total number of operations is at most

$$2 + (m+2) + 4m + 2 \leq 11m$$

using the assumption that $m \geq 1$. This is a very loose bound; for larger m this will be much closer to $5m$ than it is to $11m$. But, as we'll discuss more below, the difference between 5 and 11 won't matter too much to us, so much as the linear dependence on m.

Now that we understand how much work is going on in one call where the input has size m, let's add it all up to obtain a bound on the number of operations required for `MergeSort`. In a Merge of m numbers, we want to translate this into a bound on the number of operations required for `MergeSort`. At first glance, the pessimist in you may be concerned that at each level of recursive calls, we're spawning an exponentially increasing number of copies of `MergeSort` (because the number of calls at each depth doubles). Dual to this, the optimist in you will notice that at each level, the inputs to the problems are decreasing at an exponential rate (because the input size halves with each recursive call). Today, the optimists win out.

Claim 3. MergeSort *requires at most* $11n \log n + 11n$ *operations to sort* n *numbers.*

Before we go about proving this bound, let's first consider whether this running time bound is good. We covered in last lecture that more obvious methods of sorting, like `InsertionSort`, required roughly n^2 operations. How does $n^2 = n \cdot n$ compare to $n \cdot \log n$? An intuitive definition of $\log n$ is the following: "Enter n into your calculator. Divide by 2 until the total is ≤ 1. The number of times you divided is the logarithm of n." This number in general will be significantly smaller than n. In particular, if $n = 32$, then $\log n = 5$; if $n = 1024$, then $\log n = 10$. Already, to sort arrays of ≈ 103 numbers, the savings of $n \log n$ as compared to n^2 will be orders of magnitude. At larger problem instances of $10^6, 10^9$, etc. the difference will become even more pronounced! $n \log n$ is much closer to growing linearly (with n) than it is to growing quadratically (with n^2).

One way to argue about the running time of recursive algorithms is to use recurrence relations. A recurrence relation for a running time expresses the time it takes to solve an input of size n in terms of the time required to solve the recursive calls the algorithm makes. In particular, we can write the running time $T(n)$ for `MergeSort` on an array of n numbers as the following expression.

$$
\begin{aligned}
T(n) &= T(n/2) + T(n/2) + T(Merge(n)) \\
&\leq 2 \cdot T(n/2) + 11n
\end{aligned}
$$

There are a number of sophisticated and powerful techniques for solving recurrences. We will cover many of these techniques in the coming lectures. Today, we can actually analyze the running time directly.

Proof of Claim 3 Consider the recursion tree of a call to `MergeSort` on an array of n numbers. Assume for simplicity that n is a power of 2. Let's refer to the initial call as Level 0, the proceeding recursive calls as Level 1, and so on, numbering the level of recursion by its depth in the tree. How deep is the tree? At each level, the size of the inputs is divided in half, and there are no recursive calls when the input size is ≤ 1 element. By our earlier "definition", this means the bottom level will be Level $\log n$. Thus, there will be a total of $\log n + 1$ levels.

We can now ask two questions: (1) How many subproblems are there at Level i? (2) How large are the individual subproblems at Level i? We can observe that at the ith level, there will be 2^i subproblems, each with inputs of size $n/2^i$.

We've already worked out that each sub-problem with input of size $n/2^i$ takes at most $11n/2^i$ operations. Now we can add this up:

$$\begin{aligned}
\text{Work at Level } i &= (\text{number of subproblems}) \cdot (\text{work per subproblem}) \\
&\leq 2^i \cdot 11 \left(\frac{n}{2^i}\right) \\
&= 11n \text{ operations.}
\end{aligned}$$

Importantly, we can see that the work done at Level i is independent of $i - i$. It only depends on n and is the same for every level. This means we can bound the total running time as follows:

$$\begin{aligned}
\text{Total number of operations } &= (\text{operations per level}) \cdot (\text{number of levels}) \\
&\leq (11n) \cdot (\log n + 1) \\
&= 11n \log n + 11n
\end{aligned}$$

This proves the claim, and we're done!