1 Introduction

Today, we will introduce a fundamental algorithm design paradigm, Divide-And-Conquer, through a case study of the MergeSort algorithm. Along the way, we'll introduce guding principles for algorithm design, including Worst-Case and Asymptotic Analysis, which we will use throughout the remainder of the course. We will introduce Asymptotic ("Big-Oh") Notation for analyzing the run times of algorithms.

2 MergeSort and the Divide-And-Conquer Paradigm

The sorting problem is a canonical computer science problem. The problem is specified as follows: as input, you receive an array of n numbers, possibly unsorted; the goal is to output the same numbers, sorted in increasing order. Computer scientists care a lot about sorting because many other algorithms will use sorting as a subrouting. Thus, it is extremely important to find efficient algorithms for sorting lists, that work well in theory and in practice.

In this lecture, we'll assume that the elements given in the array are distinct. It is a worthwhile exercise to go through the notes carefully and see which aspects of our analysis would need to change if we allow for ties amongst elements.

There are a number of natural algorithms for sorting a list of numbers. One such algorithm is SelectionSort, which builds up a sorted array by traversing the elements of the given array and finds the minimum element remaining and removes the element and adds it to the end of the sorted array. We can argue that on input arrays of length n, SelectionSort requires roughly n^2 operations. The natural question to ask (which should become your mantra this quarter) is "Can we do better?". In fact, the answer to this question is "Yes". The point of today's lecture is to see how we might do better, and to begin to understand what "better" even means.

2.1 Divide-And-Conquer

The Divide-And-Conquer paradigm is a broad pattern for designing algorithms to many problems. Unsurprisingly, the pattern uses the following strategy to solve problems.

- Break the problem into subproblems
- Solve the subproblems (recursively)
- Combine results of the subproblems

This strategy requires that once the instances become small enough, the problem is trivial to solve (or it is cheap to find a solution through brute-force).

With this pattern in mind, there is a very natural way to formulate a Divide-And-Conquer algorithm for the sorting problem. Consider the following pseudocode[1] for MergeSort (in Algorithm 1).

Now, we need to describe the Merge procedure, which takes two sorted arrays, L and R, and produces a sorted array containing the elements of L and R. Consider the following Merge procedure (Algorithm 2), which we will call as a subroutine in MergeSort.

[1] A note on pseudocode: We will write our algorithms in pseudocode. The point of pseudocode is to be broadly descriptive – to convey your approach in solving a problem without getting hung up on the specifics of a programming language. In fact, one of the key benefits of using pseudocode to describe algorithms is that you can take the algorithm and implement it in any language you want based on your needs.

Algorithm 1: MERGESORT(A)

```
n ← length(A);
if n ≤ 1 then
    return A;
L ← MergeSort(A[1 : n/2]);
R ← MergeSort(A[n/2 + 1 : n]);
return Merge(L, R);
```

Algorithm 2: MERGE(L, R)

```
m ← length(L) + length(R);
S ← empty array of size m;
i ← 1; j ← 1;
for k = 1 → m do
    if L(i) < R(j) then
        S(k) ← L(i);
        i ← i + 1;
    else
        S(k) ← R(j);
        j ← j + 1;
return S;
```

Intuitively, MERGE loops through every position in the final array, and at the ith call, looks for the ith smallest element. (As a special case, consider the fact that the smallest element in S must be the smallest element in L or the smallest element in R). Because L and R are sorted, we can find the ith smallest element quickly, by keeping track of which elements we've processed so far. (The MERGE subroutine, as written, is actually incomplete. You should think about how you would have to edit the pseudocode to handle what happens when we get to the end of L or R.)

Now that we have an algorithm, the first question we always want to ask is: "Is the algorithm correct?" In this case, to answer this question, we will define an *invariant* which we will claim holds at every recursive call. The invariant will be the following: "In every recursive call, MERGESORT returns a sorted array." If we can prove that this invariant holds, it will immediately prove that MERGESORT is correct, as the first call to MERGESORT will return a sorted array. Here, we will prove that the invariant holds.

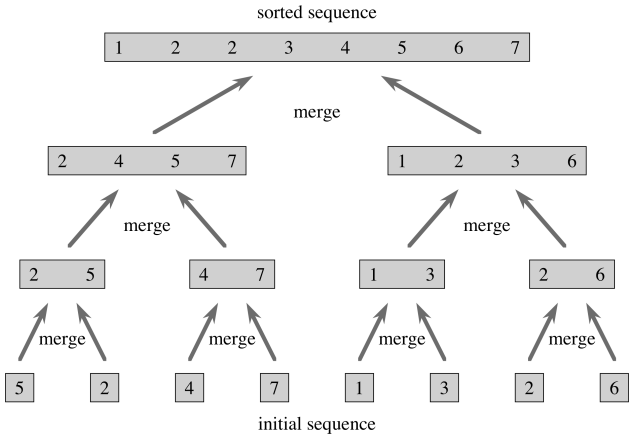

Figure 2.4 from CLRS
Schematic of the levels of recursive calls, or "recursion tree", and the resulting calls to MERGE

Proof of Invariant. By induction. Consider the base case, of the algorithm, when MergeSort receives ≤ 1 element. Any array of ≤ 1 element is sorted (trivially), so in this case, by returning A, MergeSort returns a sorted list.

To see the inductive step, suppose that after $n-1 \geq 1$ levels of recursive calls return, MergeSort still returns a sorted list. We will argue that the nth recursive call returns a sorted list. Consider some execution of MergeSort where L and R were just returned after $\leq n-1$ recursive calls. Then, by the inductive hypothesis, L and R are both sorted. We argue that the result of Merge(L, R) will be a sorted list. To see this, note that in the Merge subroutine, the minimum remaining element must be at either the ith position in L or the jth position of R. When we find the minimum element, we will increment i or j accordingly, which will result in the next remaining minimal element still being in the ith position in L or the jth position of R. Thus, Merge will construct a sorted list, and our induction holds. □

2.2 Running Time Analysis

After answering the question of correctness, the next question to ask is: "Is the algorithm *good*?" Well, that depends on how we define the "goodness" of an algorithm. In this class, as is typical, we will generally reframe this question as: "How does the running time of the algorithm grow with the size of the input?" Generally, the slower the running time grows as the input increase in size, the better the algorithm.

Our eventual goal is to argue about the running time of MergeSort, but this seems a bit challenging. In particular, each call to MergeSort makes a number of recursive calls and then calls Merge – it isn't immediately obvious how we should bound the time that MergeSort takes. A seemingly less ambitious goal would be to analyze how long a call to Merge takes. We will start here, and see if this gives us something to grasp onto when arguing about the total running time.

Consider a single call to Merge, where we'll assume the total size of S 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 then can bound the amount of running time for Merge as

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

Using the fact that we know that $m \geq 1$, we can upper bound this running time by $6m$ operations.

Now that we have a bound on the number of operations required 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 1. MergeSort *requires at most* $6n \log n + 6n$ *operations to sort* n *numbers.*[2]

Before we go about proving this bound, let's first consider whether this running time bound is good. We mentioned earlier that more obvious methods of sorting, like SelectionSort required roughly n^2 operations.

[2]In this lecture, all future lectures, unless explicitly state otherwise, log refers to $\log_2$.

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 $\approx 10^3$ 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 for MergeSort on an array of n numbers as the following expression.

$$\begin{aligned} T(n) &= T(n/2) + T(n/2) + T(\textsc{Merge}(n)) \\ &\leq 2 \cdot T(n/2) + 6n \end{aligned}$$

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.

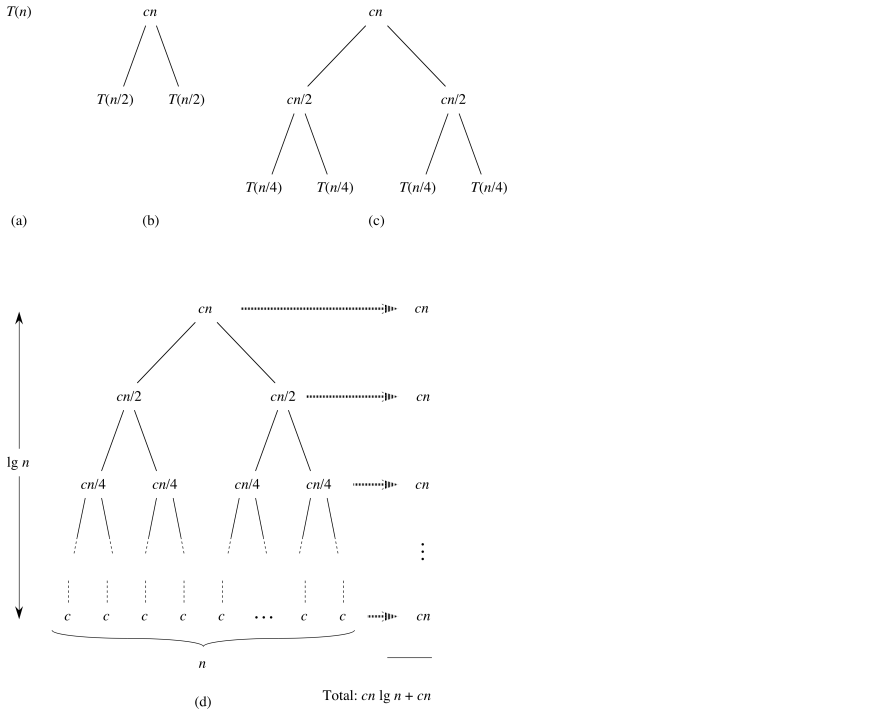

Figure 2.5 from CLRS – Analyzing the running time in terms of a Recursion Tree

Proof of Claim. Consider the recursion tree of a call to MergeSort on an array of n numbers. 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 it's depth in the tree. How deep is the tree? At each level, the size of the inputs is divided in half, and the recursion bottoms out 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 $n/2^i$. So how much work do we do overall at the ith level? First, we need to make two recursive calls – but the work done for these recursive calls will be accounted for by Level $i + 1$. Thus, we are really concerned about the cost of a call to Merge at the Level i. We can express the work per level as follows:

$$\begin{aligned}\text{Work at Level } i &= (\text{number of subproblems}) \cdot (\text{work per subproblem}) \\ &\leq 2^i \cdot 6\left(\frac{n}{2^i}\right) \\ &= 6n\end{aligned}$$

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

$$\begin{aligned}\text{Total RT} &= (\text{work per level}) \cdot (\text{number of levels}) \\ &\leq (6n) \cdot (\log n + 1) \\ &= 6n \log n + 6n\end{aligned}$$

□

3 Guiding Principles for Algorithm Design and Analysis

After going through the algorithm and analysis, it is natural to wonder if we've been too sloppy. In particular, note that the algorithm never "looks at" the input. For instance, what if we received the sequence of numbers $[1, 2, 3, 5, 4, 6, 7, 8]$? Clearly, there is a "sorting algorithm" for this sequence that only takes a few operations, but MergeSort runs through all $\log n + 1$ levels of recursion anyway. Would it be better to try to design our algorithms with this in mind? Additionally, in our analysis, we've given a very loose upper bound on the time required of Merge and dropped a number of constant factors and lower order terms. Is this a problem? In what follows, we'll argue that these are actually *features*, not bugs, in the design and analysis of the algorithm.

3.1 Worst-Case Analysis

One guiding principle we'll use throughout the class is that of *Worst-Case Analysis*. In particular, this means that we want any statement we make about our algorithms to hold for *every* possible input. Stated differently, we can think about playing a game against an adversary, who wants to maximize our running time (make it as bad as possible). We get to specify an algorithm and state a running time $T(n)$; the adversary then chooses an input. We win the game if even in the worst case, whatever input the adversary chooses (of size n), our algorithm runs in at most $T(n)$ time.

Note that because our algorithm made no assumptions about the input, then our running time bound will hold for every possible input. This is a very strong, robust guarantee. [3]

3.2 Asymptotic Analysis

Throughout our argument about MergeSort, we combined constants (think $Cost_a, Cost_i$, etc.) and gave very loose upper bounds (i.e. $6m \geq 4m + 2$). Why did we choose to do this? First, it makes the math much easier. But does it come at the cost of getting the "right" answer? Would we get a more predictive result if we threw all these exact expressions back into the analysis? From the perspective of an algorithm designer, the answer is to both of these questions is a resounding "No". As an algorithm designer, we want to come up with results that are broadly applicable, whose truth does not depend on features of a specific programming language or machine architecture. The constants that we've dropped will depend greatly on the language and machine on which you're working. For the same reason we use pseudocode instead of writing our algorithms

[3] In the case where you have significant domain knowledge about which inputs are likely, you may choose to design an algorithm that works well in expectation on these inputs (this is frequently referred to as Average-Case Analysis). This design pattern is less common and can lead to poor performance if you don't actually understand which inputs are likely.

in Java, trying to quantify the exact running time of an algorithm would be inappropriately specific. This is not to say that constant factors never matter in applications (e.g. I would be rather upset if my web browser ran 7 times slower than it does now) but worrying about these factors is not the goal of this class. In this class, our goal will be to argue about which strategies for solving problems are wise and why.

In particular, we will focus on *Asymptotic Analysis*. This type of analysis focuses on the running time of your algorithm as your input size gets very large (i.e. $n \to +\infty$). This framework is motivated by the fact that if we need to solve a small problem, it doesn't cost that much to solve it by brute-force. If we want to solve a large problem, we may need to be much more creative in order for the problem to run efficiently. From this perspective, it should be very clear that $6n(\log n + 1)$ is much better than $n^2/2$. (If you are unconvinced, try plugging in some values for n.)

Intuitively, we'll say that an algorithm is "fast" when the running time grows "slowly" with the input size. In this class, we want to think of growing "slowly" as growing as close to linear as possible. Based on this this intuitive notion, we can come up with a formal system for analyzing how quickly the running time of an algorithm grows with its input size.

3.3 Asymptotic Notation

To talk about the running time $T(n)$ of algorithms, we will use the following notation.

"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

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

"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

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

"Big-Theta" Notation:

$T(n)$ is $\Theta(f(n))$ if and only if $T(n) = O(f(n))$ and $T(n) = \Omega(f(n))$. Equivalently, we can say that

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

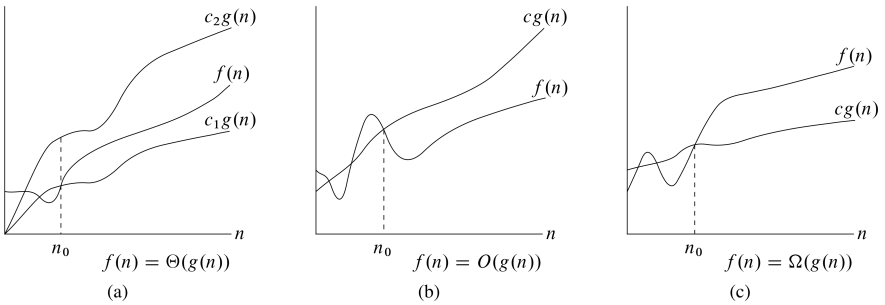

Figure 3.1 from CLRS – Examples of Asymptotic Bounds
(Note: In these examples $f(n)$ corresponds to our $T(n)$ and $g(n)$ corresponds to our $f(n)$.)

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 2. *All degree-k polynomials are $O(n^k)$.*

Proof of Claim. Suppose $T(n)$ is a degree-k polynomial. That is, $T(n) = a_k n^k + \ldots + a_1 n + a_0$ for some choice of a_i's where $a_k \neq 0$. To show that $T(n)$ is $O(n^k)$ we must find a c and n_0 such that for all $n > n_0$ $T(n) \leq c \cdot n^k$. Let $n_0 = 1$ and let $a^* = \max_i |a_i|$. We can bound $T(n)$ as follows:

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

Let $c = (k+1)a^*$ which is a constant, independent of n. Thus, we've exhibited a c, n_0 which satisfy the Big-Oh definition, so $T(n) = O(n^k)$. □

Claim 3. *For any $k \geq 1$, n^k is not $O(n^{k-1})$.*

Proof of Claim. By contradiction. Assume $n^k = O(n^{k-1})$. Then there is some choice of c and n_0 such that $n^k \leq c \cdot n^{k-1}$ for all $n > n_0$. 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 $O(n^{k-1})$. □