The purpose of this lecture is to give a brief overview of the topic of Algorithms and the kind of thinking it involves: why we focus on the subjects that we do, and why we emphasize proving guarantees. We also go through an example of a problem that is easy to relate to, that of finding the median of a set of n elements. This is a problem for which there is a simple $O(n \log n)$ time algorithm, but we can do better, using randomization, and also a clever deterministic construction. These illustrate some of the ideas and tools we will be using (and building upon) in this course.

Material in this lecture:

- Administrivia (see course policies on the webpage)
- What is the study of Algorithms all about?
- Why do we care about specifications and proving guarantees?
- Finding the median: A Randomized Algorithm in expected linear time.
- A deterministic linear-time algorithm.

1 Goals of the Course

This course is about the design and analysis of algorithms — how to design correct, efficient algorithms, and how to think clearly about analyzing correctness and running time.

What is an algorithm? At its most basic, an algorithm is a method for solving a computational problem. A recipe. Along with an algorithm comes a specification that says what the algorithm's guarantees are. For example, we might be able to say that our algorithm indeed correctly solves the problem in question and runs in time at most $f(n)$ on any input of size n. This course is about the whole package: the design of efficient algorithms, *and* proving that they meet desired specifications. For each of these parts, we will examine important techniques that have been developed, and with practice we will build up our ability to think clearly about the key issues that arise.

The main goal of this course is to provide the intellectual tools for designing and analyzing your own algorithms for problems you need to solve in the future. Some tools we will discuss are Dynamic Programming, Divide-and-Conquer, Hashing and other Data Structures, Randomization, Network Flows, and Linear Programming. Some analytical tools we will discuss and use are Recurrences, Probabilistic Analysis, Amortized Analysis, and Potential Functions.

There is also a dual to algorithm design: Complexity Theory. Complexity Theory looks at the intrinsic difficulty of computational problems — what kinds of specifications can we expect *not* to be able to achieve? In this course, we will delve a bit into complexity theory, focusing on the somewhat surprising notion of NP-completeness. We will additionally discuss some approaches for dealing with NP-complete problems, including the notion of approximation algorithms.

Another goal will be to discuss models that go beyond the traditional input-output model. In the traditional model we consider the algorithm to be given the entire input in advance and it just has to perform the computation and give the output. This model is great when it applies, but it is not always the right model. For instance, some problems may be challenging because they require

decisions to be made without having full information, and we will discuss online algorithms and machine learning, which are two paradigms for problems of this nature. In other settings, we may have to deal with computing quantities of a "stream" of input data where the space we have is much smaller than the data. In yet other settings, the input is being held by a set of selfish agents who may or may not tell us the correct values.

2 On guarantees and specifications

One focus of this course is on proving correctness and running-time guarantees for algorithms. Why is having such a guarantee useful? Suppose we are talking about the problem of sorting a list of n numbers. It is pretty clear why we at least want to know that our algorithm is correct, so we don't have to worry about whether it has given us the right answer all the time. But, why analyze running time? Why not just code up our algorithm and test it on 100 random inputs and see what happens? Here are a few reasons that motivate our concern with this kind of analysis — you can probably think of more reasons too:

Composability. A guarantee on running time gives a "clean interface". It means that we can use the algorithm as a subroutine in some other algorithm, without needing to worry whether the kinds of inputs on which it is being used now necessarily match the kinds of inputs on which it was originally tested.

Scaling. The types of guarantees we will examine will tell us how the running time scales with the size of the problem instance. This is useful to know for a variety of reasons. For instance, it tells us roughly how large a problem size we can reasonably expect to handle given some amount of resources.

Designing better algorithms. Analyzing the asymptotic running time of algorithms is a useful way of thinking about algorithms that often leads to nonobvious improvements.

Understanding. An analysis can tell us what parts of an algorithm are crucial for what kinds of inputs, and why. If we later get a different but related task, we can often use our analysis to quickly tell us if a small modification to our existing algorithm can be expected to give similar performance to the new problem.

Complexity-theoretic motivation. In Complexity Theory, we want to know: "how hard is fundamental problem X really?" For instance, we might know that no algorithm can possibly run in time $o(n \log n)$ (growing more slowly than $n \log n$ in the limit) and we have an algorithm that runs in time $O(n^{3/2})$. This tells us how well we understand the problem, and also how much room for improvement we have.

It is often helpful when thinking about algorithms to imagine a game where one player is the algorithm designer, trying to come up with a good algorithm for the problem, and its opponent (the "adversary") is trying to come up with an input that will cause the algorithm to run slowly. An algorithm with good worst-case guarantees is one that performs well no matter what input the adversary chooses. We will return to this view in a more formal way when we discuss lower bounds and game theory.

3 An example: Median Finding

One thing that makes algorithm design "Computer Science" is that solving a problem in the most obvious way from its definitions is often not the best way to get a solution. A simple example of this is median finding.

Recall the median. For a set of n elements, this is the element in this set that is the $n/2^{th}$ smallest, i.e., it has $n/2$ elements larger than it.[1] Given an unsorted array, how quickly can one find the median element? The definition does not give us an obvious clue: we can enumerate over all elements, and for each check if it is the median. This gives a $\Theta(n^2)$ time algorithm. Or one can sort, and then read off the median, which takes $O(n \log n)$ time using MergeSort or HeapSort (deterministic) or QuickSort (randomized).

Can one do it more quickly than by sorting? In this lecture we describe two linear-time algorithms for this problem: one randomized and one deterministic. More generally, we solve the problem of finding the kth smallest out of an unsorted array of n elements.

3.1 The problem and a randomized solution

Consider a problem that is slightly more general than median-finding:

> *Find the k^{th} smallest element in an unsorted array of size n.*

(Let's say all elements are distinct to avoid the question of what we mean by the kth smallest when we have equalities). One way to solve this problem is to sort and then output the kth element. We can do this in time $O(n \log n)$ if we sort using MergeSort, QuickSort, or HeapSort. Is there something faster – a linear-time algorithm? The answer is yes. We will explore both a simple randomized solution and a more complicated deterministic one.

The idea for the randomized algorithm is to start with the Randomized QuickSort algorithm (choose a random element as "pivot", partition the array into two sets Less and Greater consisting of those elements less than and greater than the pivot respectively, and then recursively sort Less and Greater). Then notice that there is a simple speedup we can make if we just need to find the kth smallest element. In particular, after the partitioning step we can tell which of Less or Greater has the item we are looking for, just by looking at their sizes. So, we only need to recursively examine one of them, not both. For instance, if we are looking for the 87th-smallest element in our array, and suppose that after choosing the pivot and partitioning we find that Less has 200 elements, then we just need to find the 87th smallest element in Less. On the other hand, if we find Less has 40 elements, then we just need to find the $87 - 40 - 1 = 46$th smallest element in Greater. (And if Less has size exactly 86 then we can just return the pivot). One might at first think that allowing the algorithm to only recurse on one subset rather than both would just cut down time by a factor of 2. However, since this is occuring recursively, it compounds the savings and we end up with $\Theta(n)$ rather than $\Theta(n \log n)$ time. This algorithm is often called Randomized-Select, or QuickSelect.

[1] We are deliberately ignoring what happens if n is odd, you can — and indeed, should (and will have to) — make this precise when you code it up, but for now it will be easier not to worry about this, since the ideas we get here can be all made perfectly precise.

QuickSelect: Given array A of size n and integer $1 \leq k \leq n$,

1. Pick a pivot element p at random from A.
2. Split A into subarrays LESS and GREATER by comparing each element to p as in Quicksort. While we are at it, count the number L of elements going in to LESS.
3. (a) If $L = k-1$, then output p. [always happens when $n = 1$]
   (b) If $L > k-1$, output QuickSelect(LESS, k).
   (c) If $L < k-1$, output QuickSelect(GREATER, $k-L-1$)

Theorem 1 *The expected number of comparisons for QuickSelect is at most $4n$.*

Before giving a formal proof, here's some intuition. If we split a candy bar at random into two pieces, the expected size of the larger piece is 3/4 of the bar. If the size of the larger subarray after our partition was always 3/4 of the array, then we would have a recurrence $T(n) \leq (n-1)+T(3n/4)$ which solves to $T(n) < 4n$. Now, this is not quite the case for our algorithm because $3n/4$ is only the *expected* size of the larger piece. That is, if i is the size of the larger piece, our expected cost-to-go is really $\mathbb{E}[T(i)]$ rather than $T(\mathbb{E}[i])$. (And the exercise below shows the latter could underestimate by a lot.) However, because the answer is linear in n, the average of the $T(i)$'s turns out to be the same as T(average of the i's). Let's now see this a bit more formally.

Proof (Theorem 1): Let $T(A, n, k)$ denote the expected time to find the kth smallest in an array A (of distinct elements) of size n. First we observe that this function does not depend on the array A. (This can be proved by induction.) Now let $T(n) = \max_k T(A, n, k)$. We will show that $T(n) < 4n$.

The proof is by induction. In the base case $n = 1$ and there are zero comparisons. In the general case it takes $n-1$ comparisons to split into the array into two pieces in Step 2. These pieces are equally likely to have size 0 and $n-1$, or 1 and $n-2$, or 2 and $n-3$, and so on up to $n-1$ and 0. The piece we recurse on will depend on k, but since we are only giving an upper bound, we can imagine that we always recurse on the larger piece. Let's assume for a moment that n is even. Therefore we have:

$$\begin{aligned} T(n) &\leq (n-1) + \frac{2}{n}\sum_{i=n/2}^{n-1} T(i) \\ &= (n-1) + \text{avg}\left[T(n/2), \ldots, T(n-1)\right]. \end{aligned}$$

We can solve this using the "guess and check" method based on our intuition above. Assume inductively that $T(i) \leq 4i$ for $i < n$. Then,

$$\begin{aligned} T(n) &\leq (n-1) + \text{avg}\left[4(n/2), 4(n/2+1), \ldots, 4(n-1)\right] \\ &\leq (n-1) + 4(3n/4) \\ &< 4n, \end{aligned}$$

and we have verified our guess. The case when n is odd is similar. ∎

Exercise 1: As in the intuition above, let i be random variable denoting the size of the larger piece. Show an increasing function $F : \mathbb{R}_+ \to \mathbb{R}_+$ such that $\mathbb{E}[F(i)] \gg F(\mathbb{E}[i])$.

4 A deterministic linear-time algorithm

What about a deterministic linear-time algorithm? For a long time it was thought this was impossible, and that there was no method faster than first sorting the array. In the process of trying to prove this formally, it was discovered that this thinking was incorrect, and in 1972 a deterministic linear time algorithm was developed by Manuel Blum, Bob Floyd, Vaughan Pratt, Ron Rivest, and Bob Tarjan.[2]

The idea of the algorithm is that one would like to pick a pivot deterministically in a way that produces a good split. Ideally, we would like the pivot to be the median element so that the two sides are the same size. But, this is the same problem we are trying to solve in the first place! So, instead, we will give ourselves leeway by allowing the pivot to be any element that is "roughly" in the middle: at least 3/10 of the array below the pivot and at least 3/10 of the array above. The algorithm is as follows:

DeterministicSelect: Given array A of size n and integer $k \leq n$,

1. Group the array into $n/5$ groups of size 5 and find the median of each group. (For simplicity, we will ignore integrality issues.)
2. Recursively, find the true median of the medians. Call this p.
3. Use p as a pivot to split the array into subarrays Less and Greater.
4. Recurse on the appropriate piece.

Theorem 2 *DeterministicSelect makes $O(n)$ comparisons to find the kth smallest in an array of size n.*

Proof: Let $T(n,k)$ denote the worst-case time to find the kth smallest out of n, and $T(n) = \max_k T(n,k)$ as before.

Step 1 takes time $O(n)$, since it takes just constant time to find the median of 5 elements. Step 2 takes time at most $T(n/5)$. Step 3 again takes time $O(n)$. Now, we claim that at least 3/10 of the array is $\leq p$, and at least 3/10 of the array is $\geq p$. Assuming for the moment that this claim is true, Step 4 takes time at most $T(7n/10)$, and we have the recurrence:

$$T(n) \quad \leq \quad cn + T(n/5) + T(7n/10), \tag{1}$$

for some constant c. Before solving this recurrence, lets prove the claim we made that the pivot will be roughly near the middle of the array. So, the question is: how bad can the median of medians be? But first, let's do an example.

> **Example 1:** Suppose the array has 15 elements and breaks down into three groups of 5 like this:
>
> $$\{1,2,3,10,11\}, \ \ \{4,5,6,12,13\}, \ \ \{7,8,9,14,15\}.$$
>
> In this case, the medians are 3, 6, and 9, and the median of the medians p is 6. There are five elements less than p and nine elements greater.

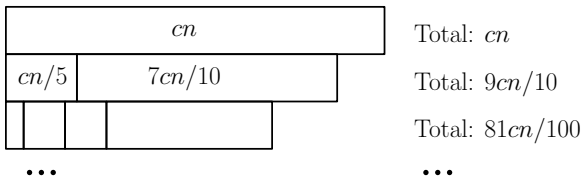
In general, what is the worst case? If there are $g = n/5$ groups, then we know that in at least $\lceil g/2 \rceil$ of them (those groups whose median is $\leq p$) at least three of the five elements are $\leq p$. Therefore,

[2] That's 4 Turing Award winners on that one paper!

the total number of elements $\leq p$ is at least $3\lceil g/2 \rceil \geq 3n/10$. Similarly, the total number of elements $\geq p$ is also at least $3\lceil g/2 \rceil \geq 3n/10$.

Now, finally, let's solve the recurrence. We have been solving a lot of recurrences by the "guess and check" method, which works here too, but how could we just stare at this and *know* that the answer is linear in n? One way to do that is to consider the "stack of bricks" view of the recursion tree discussed in the notes for Recitation #1.

In particular, let's build the recursion tree for the recurrence (1), making each node as wide as the quantity inside it:



Notice that even if this stack-of-bricks continues downward forever, the total sum is at most

$$cn(1 + (9/10) + (9/10)^2 + (9/10)^3 + \ldots),$$

which is at most $10cn$. This proves the theorem. ■

Notice that in our analysis of the recurrence (1) the key property we used was that $n/5+7n/10 < n$. More generally, we see here that if we have a problem of size n that we can solve by performing recursive calls on pieces whose total size is at most $(1-\epsilon)n$ for some constant $\epsilon > 0$ (plus some additional $O(n)$ work), then the total time spent will be just linear in n. This gives us a nice extension to our "Master theorem" from the notes to Recitation #1.

Theorem 3 *For constants c and $a_1, \ldots, a_k$ such that $a_1 + \ldots a_k < 1$, the recurrence*

$$T(n) \quad \leq \quad T(a_1 n) + T(a_2 n) + \ldots T(a_k n) + cn$$

solves to $T(n) = O(n)$.

Exercise 2: Show that for constants c and $a_1, \ldots, a_k$ such that $a_1 + \ldots a_k = 1$ and each $a_i < 1$, the recurrence

$$T(n) \quad \leq \quad T(a_1 n) + T(a_2 n) + \ldots T(a_k n) + cn$$

solves to $T(n) = O(n \log n)$. Show that this is best possible by observing that $T(n) = T(n/2)+T(n/2)+n$ solves to $T(n) = \Theta(n \log n)$.

Exercise 3: What happens if we split the elements into $n/3$ groups of size 3? Or n/k groups of size k for larger odd values of k?

Exercise 4: The rank of an element a with respect to a list A of n distinct elements is $|\{e \in A \mid e \leq a\}$ is the number of elements in A no greater than a. Hence the rank of the smallest element in A is 1, and the rank of the median is $n/2$. Given an unsorted list A and indices $i \leq j$, give an $O(n)$ time algorithm to output all elements in A with ranks lying in the interval $[i, j]$.