Introduction and Linear-time Selection

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. We will also practice writing and solving recurrence relations, which is a key tool for the analysis of algorithms.

Objectives of this lecture

In this lecture, we cover:

- 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.
- Analyzing a randomized recursive algorithm
- A deterministic linear-time algorithm for medians.

Recommended study resources

- CLRS, *Introduction to Algorithms*, Chapter 9, Medians and Order Statistics
- DPV, *Algorithms*, Chapter 2.4, Medians

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 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. 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. Algorithms that solve such problems are called *online* algorithms, which we will also discuss. 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 non-obvious 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 for a given problem, 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. An example of this is median finding. Recall the concept of the median of a set. For a set of n elements, this is the "middle" element in the set, i.e., there are exactly $\lfloor n/2 \rfloor$ elements larger than it. In computer sciencey terms, if the elements are zero-indexed, the median is the $\lfloor (n-1)/2 \rfloor^{\text{th}}$ element of the set when represented in sorted order.

Given an unsorted array, how quickly can one find the median element? Perhaps the simplest solution, one that can be described in a single sentence and implemented with one or two lines of code in your favorite programming language, is to sort the array and then read off the element in position $\lfloor (n-1)/2 \rfloor$, which takes $O(n \log n)$ time using your favorite sorting algorithm, such as 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 k^{th} smallest out of an unsorted array of n elements.

3.1 The problem and a randomized solution

Let's consider a problem that is slightly more general than median-finding:

Problem: Select-k / k^{th} Smallest

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

To remove ambiguity, we will assume that our array is zero indexed, so the k^{th} smallest element is the element that would be in position k if the array were sorted. Alternatively, it is the element such that exactly k other elements are smaller than it. Additionally, let's say all elements are distinct to avoid the question of what we mean by the k^{th} smallest when we have duplicates.

The straightforward way to solve this problem is to sort and then output the k^{th} element. We can do this in time $O(n \log n)$ if we sort using MergeSort, QuickSort, or HeapSort. However, today we want something faster than this. 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 key idea to obtaining a linear time algorithm is to identify and eliminate redundant/wasted work. Note that by sorting the array, we are not just finding the k^{th} smallest element for the given value of k, we are actually finding the answer for *every possible* value of k. So instead of completely sorting the array, it would suffice to "partially sort" the array such that element k ends up in the correct position, but the remaining elements might still not be perfectly sorted. This might remind you of Quicksort, which selects an element called the "pivot" and places it in the correctly sorted position in the array, and then moves all elements that are less than it to the left, and all elements that are greater to the right. In essence, this is partially sorting the array with respect to the pivot. Recursively sorting both sides then sorts the entire array.

So, what if we were to just run Quicksort, but skip some of the steps that do not matter? Specifically, note that if we run Quicksort and our goal is to output the k^{th} element at the end, that after partitioning the array around the pivot, we know which of the two sides must contain the answer. Therefore instead of recursively sorting both sides, we ignore the side that can not contain the answer and recurse just once. That's it!

More specifically, the algorithm chooses a random pivot and then partitions the array into two sets LESS and GREATER consisting of those elements less than and greater than the pivot respectively. 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. 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.

Algorithm 1: QuickSelect

Given array A of size n and integer $0 \le k \le n-1$,

1. Pick a pivot element p at random from A.
2. Split A into subarrays LESS and GREATER by comparing each element to p.
3. (a) If $|\text{LESS}| > k$, then return QuickSelect(LESS, k).

   (b) If $|\text{LESS}| < k$, then return QuickSelect(GREATER, $k - |\text{LESS}| - 1$)

   (c) If $|\text{LESS}| = k$, then return p. [always happens when $n = 1$]

Theorem 1

The expected number of comparisons for QuickSelect is at most $8n$.

Remark

We need to be careful when understanding what exactly we are measuring when analyzing a randomized algorithm. When we talk about the expected complexity of an algorithm, what we mean precicely is the expected value *over the random numbers used by the algorithm* of the number of comparisons for a *worst-case input*. We are not interested in analyzing the worst possible behavior of the random number generator, and we are not analyzing a random input (this is usually called *average-case behavior*).

Formally, let $T(n)$ denote the expected number of comparisons performed by QuickSelect on any (worst-case) input of size n. What we want is a recurrence relation that looks like

$$T(n) \le n - 1 + \mathbb{E}[T(X)],$$

where $n-1$ comparisons come from comparing the pivot to every other element and placing them into LESS and GREATER, and X is a random variable corresponding to the size of the subproblem that is solved recursively. We can't just go and solve this recurrence because we don't yet know what X or $\mathbb{E}[T(X)]$ look like yet.

Before giving a formal proof, here's some intuition. First of all, how large is X, the the size of the array given to the recursive call? It depends on two things: the value of k and the *randomly-chosen pivot*. After partitioning the input into LESS and GREATER, whose size adds up to $n-1$, the algorithm recurively calls QuickSelect on one of them, but which one? Since we are interested in the behavior for a *worst-case input*, we can assume pessimistically that the value of k will always make us choose the bigger of LESS and GREATER. Therefore the question becomes: if we choose a random pivot and split the input into LESS and GREATER, how large is the larger of the two of them? Well, possible sizes of the splits (ignoring rounding) are

$$(0, n-1), (1, n-2), (2, n-3), \ldots, (n/2-2, n/2+1), (n/2-1, n/2),$$

so we can see that the larger of the two is a random number from

$$n-1, n-2, n-3, \ldots, n/2+1, n/2.$$

So, the expected size of the larger half is about $3n/4$, again, ignoring rounding errors.

Another way to say this is that if we split a candy bar at random into two pieces, the expected size of the larger piece is 3/4 of the bar. Using this, we might be tempted to write the following recurrence relation, which is almost correct, but not quite.

$$T(n) \le n - 1 + T(3n/4).$$

If we go through the motions of solving this recurrence, we get $T(n) = O(n)$, but unfortunately, this derrivation is not quite valid. The reasoning is that $3n/4$ is only the *expected* size of the larger piece. That is, if X is the size of the larger piece, we have written a recurrence where the cost of the recursive call is $T(\mathbb{E}[X])$, but it was supposed to be $\mathbb{E}[T(X)]$, and these are not the same thing! (The exercise below shows the two could differ by a lot.)[1] Let's now see this a bit more formally.

Proof Theorem 1. To correct the proof, we need to correctly analyze the *expected value of* $T(X)$, rather than the value of T for the expected value of X. To do so, we can consider with what probabilities does X take on certain values and analyze the corresponding behavior of T. So, with what probability is X at most 3/4? This happens when the smaller of LESS and GREATER is at least one quarter of the elements, i.e., when the pivot is not in the bottom quarter or top quarter[2]. This means the pivot needs to be in the middle half of the data, which happens with probability 1/2. The other half the time, the size of X will be larger, at most $n-1$. Although this might sound rather loose, this is good enough to write down a good upper bound recurrence!

$$\mathbb{E}[T(X)] \le \frac{1}{2} T\left(\frac{3n}{4}\right) + \frac{1}{2} T(n)$$

This sounds like too loose of a bound, but we will see that it is good enough. Returning to our original recurrence, we can now correctly assert that

$$T(n) \le n - 1 + \left(\frac{1}{2} T\left(\frac{3n}{4}\right) + \frac{1}{2} T(n)\right),$$

and multiplying both sides by 2 then subtracting $T(n)$, we obtain

$$T(n) \le 2(n-1) + T\left(\frac{3n}{4}\right).$$

We can now use induction to prove that this recurrence relation satisfies $T(n) \le 8n$. The base case is simple, when $n = 1$ there are no comparisons, so $T(1) = 0$. Now assume for the sake of

[1] It turns out that these two quantities are equal if T is linear, which it is in this particular case. However, we can not make this assumption in the proof, because that is we are trying to prove in the first place! Assuming that T is linear to prove that T is linear would be circular logic.

[2] Note that we are picking 3/4 here purely for convenience. You could use any constant and still prove a linear bound on the number of comparisons.

induction that $T(i) \leq 8i$ for all $i < k$. We want to show that $T(k) \leq 8k$. We have

$$\begin{aligned}
T(k) &\leq 2(n-1) + T\left(\frac{3n}{4}\right), \\
&\leq 2(n-1) + 8\left(\frac{3n}{4}\right), && \textit{Use the inductive hypothesis} \\
&\leq 2n + 6n, \\
&= 8n,
\end{aligned}$$

which proves our desired bound.

□

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.[3]

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 (i.e., some kind of "approximate median"). We will use a technique called the *median of medians* which takes the medians of a bunch of small groups of elements, and then finds the median of those medians. It has the wonderful guarantee that the selected element is greater than at least 30% of the elements of the array, and smaller than at least 30% of the elements of the array. This makes it work great as an approximate median and therefore a good pivot choice! The algorithm is as follows:

Algorithm 2: 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 in the same way as Quickselect.

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

Theorem 2

DeterministicSelect makes $O(n)$ comparisons to find the k^{th} smallest element in an array of size n.

Proof of Theorem 2. Let $T(n)$ denote the worst-case number of comparisons performed by the DeterministicSelect algorithm on inputs of size n.

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:

$$T(n) \quad \leq \quad cn + T(n/5) + T(7n/10),$$

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. Suppose the array has 15 elements and breaks down into three groups of 5 like this:

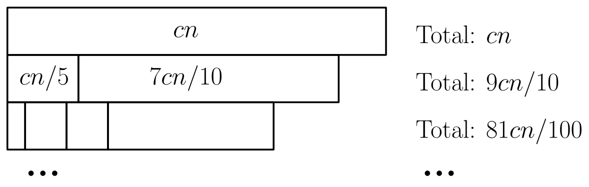
$$\{1,2,3,10,11\},\ \{4,5,6,12,13\},\ \{7,8,9,14,15\}.$$

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, 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 that you might have discussed in your previous classes.

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.

Theorem 3

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

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

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

Exercises: Selection Algorithms & Recurrences

Problem 1. Recall the attempted analysis of randomized quickselect where we accidentally assumed that $\mathbb{E}[T(X)] = T(\mathbb{E}[X])$. Let X be a random variable, and find an increasing function $F : \mathbb{R}_+ \to \mathbb{R}_+$ such that $\mathbb{E}[F(i)] \gg F(\mathbb{E}[i])$.

Problem 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

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

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 $T(n) = \Theta(n \log n)$.

Problem 3. Consider the median of medians algorithm. What happens if we split the elements into $n/3$ groups of size 3 instead? Or n/k groups of size k for larger odd values of k?

Problem 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 $[i, j]$.