Lecture 1

Introduction & Median Finding

Supplemental reading in CLRS: Section 9.3; Chapter 1; Sections 4.3 and 4.5

1.1 The Course

Hello, and welcome to 6.046 Design and Analysis of Algorithms. The prerequisites for this course are

1. *6.006 Introduction to Algorithms.* This course is designed to build on the material of 6.006. The algorithms in 6.046 are generally more advanced. Unlike 6.006, this course will not require students to implement the algorithms; instead, we make frequent use of pseudocode.

2. *Either 6.042 / 18.062J Mathematics for Computer Science or 18.310 Principles of Applied Mathematics.* Students must be familiar with beginning undergraduate mathematics and able to write mathematical proofs. After taking this course, students should be able not just to describe the algorithms they have learned, but also to prove their correctness (where applicable) and rigorously establish their asymptotic running times.

The course topics are as follows:

- Divide and Conquer
- Dynamic Programming
- Greedy Algorithms
- Graph Algorithms
- Randomized Algorithms
- Data Structures
- Approximation Algorithms.

The course textbook is *Introduction to Algorithms 3e.*, by Cormen, Leiserson, Rivest and Stein. We will usually refer to the textbook (or the authors) as "CLRS." In addition, Profs. Bruce Tidor and Dana Moshkovitz, who taught this course in spring of 2012, prepared hand-written notes for each lecture. What you are reading is an electronic, fleshed-out version of those notes, developed by Ben Zinberg with oversight by Prof. Moshkovitz under the commission of MIT OpenCourseWare.

1.2 Algorithms

Before our first example, let's take a moment to ask:

What is the study of algorithms, and why is it important?

CLRS offer the following answer: An algorithm is "any well-defined computational procedure that takes some [...] input and produces some [...] output." They point out that algorithms are an essential component not just of inherently electronic domains such as electronic commerce or the infrastructure of the Internet, but also of science and industry, as exemplified by the Human Genome Project or an airline that wishes to minimize costs by optimizing crew assignments.

Studying algorithms allows us both to understand (and put to use) the capabilities of computers, and to communicate our ideas about computation to others. Algorithms are a basis for design: they serve as building blocks for new technology, and provide a language in which to discuss aspects of that technology. The question of whether a certain algorithm is effective depends on the context in which it is to be used. Often relevant are the following two concerns:

- *Correctness.* An algorithm is said to be **correct** if, for every possible input, the algorithm halts with the desired output. Usually we will want our algorithms to be correct, though there are occasions in which incorrect algorithms are useful, if their rate of error can be controlled.

- *Efficiency.* The best algorithms are ones which not just accomplish the desired task, but use minimal resources while doing so. The resources of interest may include time, money, space in memory, or any number of other "costs." In this course we will mostly be concerned with time costs. In particular, we will try to give asymptotic bounds on the "number of steps" required to perform a computation as a function of the size of the input. The notion of a "step" is an imprecise (until it has been given a precise definition) abstraction of what on current computers could be processor cycles, memory operations, disk operations, etc. Of course, these different kinds of steps take different amounts of time, and even two different instructions completed by the same processor can take different amounts of time. In this course, we do not have the time to study these subtleties in much detail. More advanced courses, such as 6.854 Advanced Algorithms, take up the subject in further detail. Fortunately, the course material of 6.046 is extremely useful despite its limitations.

1.3 Order Statistics ("Median Finding")

Median finding is important in many situations, including database tasks. A precise statement of the problem is as follows:

Problem 1.1. Given an array $A = A[1,\ldots,n]$ of n numbers and an index i ($1 \le i \le n$), find the ith-smallest element of A.

For example, if $i = 1$ then this amounts to finding the minimum element, and if $i = n$ then we are looking for the maximum element. If n is odd, then setting $i = \frac{n+1}{2}$ gives the median of A. (If n is even, then the median will probably be some average of the cases $i = \lfloor \frac{n+1}{2} \rfloor$ and $i = \lceil \frac{n+1}{2} \rceil$, depending on your convention.)

Intuitive Approach. We could simply sort the entire array A—the ith element of the resulting array would be our answer. If we use MERGE-SORT to sort A in place and we assume that jumping to the ith element of an array takes $O(1)$ time, then the running time of our algorithm is

$$\begin{array}{rrr} \text{(MERGE-SORT)} & & O(n \lg n) \\ \text{(jump to the } i\text{th element)} & + & O(1) \\ \hline & & O(n \lg n) \quad \text{(worst case).} \end{array}$$

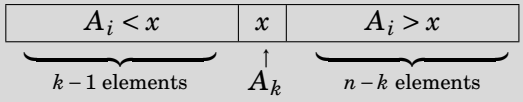
Can we expect to do better than this initial approach? If sorting all of A were a necessary step in computing any individual median, then the answer would be no. But as it stands, sorting all of A does much more work than we are asking for—it finds all n medians, whereas we only wanted one. So, even though the best sorting algorithms take $\Theta(n \lg n)$ time, it does not necessarily follow that median finding cannot be done faster.

It turns out that median finding can be done in $O(n)$ time. In practice, this is usually accomplished by a randomized algorithm with linear expected running time, but there also exists a deterministic algorithm with linear worst-case time. In a 1973 paper, Blum, Floyd, Pratt, Rivest and Tarjan proposed the so-called "median-of-medians" algorithm, which we present below.

For this algorithm, we will assume that the elements of A are distinct. This is not a very strong assumption, and it is easy to generalize to the case where A is allowed to have duplicate elements.[1]

Algorithm: SELECT(A, i)

1. Divide the n items into groups of 5 (plus any remainder).
2. Find the median of each group of 5 (by rote). (If the remainder group has an even number of elements, then break ties arbitrarily, for example by choosing the lower median.)
3. Use SELECT recursively to find the median (call it x) of these $\lceil n/5 \rceil$ medians.
4. Partition around x.* Let $k = \text{rank}(x)$.†



5.
   - If $i = k$, then return x.
   - Else, if $i < k$, use SELECT recursively by calling SELECT$(A[1, \ldots, k-1],\ i)$.‡
   - Else, if $i > k$, use SELECT recursively by calling SELECT$(A[k+1, \ldots, i],\ i-k)$.

* By "partition around x," we mean reorder the elements of A in place so that all elements prior to x are less than x (and consequently all elements after x are $\geq x$—in our case actually $> x$ since there are no duplicate elements). Partitioning can be done in linear time. We won't show the details here, but they can be found in §7.1 of CLRS.

† For a set S of distinct numbers, we define the **rank** of an element $x \in S$ to be the number k such that x is the kth-smallest element of S. Thus, in the set $\{5, 8, 2, 3\}$, the rank of 5 is 3.

‡The array $A[1, \ldots, k-1]$ should be passed by reference—there is certainly no need to waste time and memory by making a new copy.

[1] One example of such a way is to equip each element of A with satellite data. Replace each element $A[j]$ $(1 \leq j \leq n)$ with the pair $\langle A[j], j \rangle$, thus turning A into an array of ordered pairs. Then we can define an order on A by saying $\langle x_1, j_1 \rangle < \langle x_2, j_2 \rangle$ iff either $x_1 < x_2$, or $x_1 = x_2$ and $j_1 < j_2$. (This is the so-called "lexicographic order.") Since this order is strict, it is safe to feed into our algorithm which assumes no duplicate elements. On the other hand, once A is sorted in the lexicographic order, it will also be sorted in the regular order.

Note a few things about this algorithm. First of all, it is extremely non-obvious. Second, it is recursive, i.e., it calls itself. Third, it appears to do "less work" than the intuitive approach, since it doesn't sort all of A. Thus, we might suspect that it is more efficient than the intuitive approach.

1.3.1 Proof of Correctness

The strategy to prove correctness of an algorithm is a mix of knowing standard techniques and inventing new ones when the standard techniques don't apply. To prove correctness of our median-of-medians algorithm, we will use a very common technique called a loop invariant. A **loop invariant** is a set of conditions that are true when the loop is initialized, maintained in each pass through the loop, and true at termination. Loop invariant arguments are analogous to mathematical induction: they use a base case along with an inductive step to show that the desired proposition is true in all cases. One feasible loop invariant for this algorithm would be the following:

> At each iteration, the "active subarray" (call it A') consists of all elements of A which are between $\min(A')$ and $\max(A')$, and the current index (call it i') has the property that the i'th-smallest element of A' is the ith-smallest element of A.

In order to prove that this is a loop invariant, three things must be shown:

1. *True at initialization.* Initially, the active subarray is A and the index is i, so the proposition is obviously true.

2. *Maintained in each pass through the loop.* If the proposition is true at the beginning of an iteration (say with active subarray $A' = A'[1,\ldots,n']$ and active index i'), then the $(k-1)$- and $(n'-k)$-element subarrays depicted in step 4 clearly also have the contiguity property that we desire. Then, if step 5 calls $\textsc{Select}(A'', i'')$, it is easy to see by casework that the i''th-smallest element of A'' equals the i'th-smallest element of A', which by hypothesis equals the ith-smallest element of A.

3. *True at termination.* Since there is nothing special about the termination step (it is just whichever step happens to be interrupted by the returned value), the proof of maintenance in each pass though the loop is sufficient to show maintenance at termination.

Now that we have established the loop invariant, it is easy to see that our algorithm is correct. If the final recursive iteration of $\textsc{Select}$ has arguments A' and i', then step 5 returns the i'th-smallest element of A', which by the loop invariant equals the ith-smallest element of A. Thus we have proven that, if our algorithm terminates at all, then it terminates with the correct output. Moreover, the algorithm must terminate eventually, since the size of the active subarray shrinks by at least 1 with each recursive call and since $\textsc{Select}(A', i')$ terminates immediately if A' has one element.

1.3.2 Running Time

There is no single magic bullet for writing proofs. Instead, you should use nonrigorous heuristics to guide your thinking until it becomes apparent how to write down a rigorous proof.

In order to prove that our algorithm is efficient, it would help to know that the active subarray in each successive recursive call is much smaller than the previous subarray. That way, we are guaranteed that only relatively few recursive calls are necessary. Therefore, let's determine an upper bound on the size of A', the new active subarray, given that the old subarray, A, had size n.

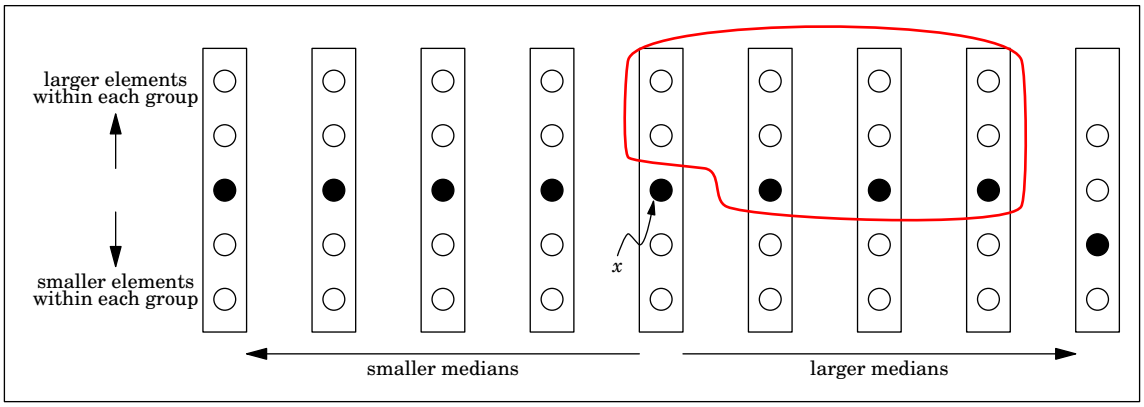


Figure 1.1. Imagine laying out the groups side by side, ordered with respect to their medians. Also imagine that each group is sorted from greatest to least, top to bottom. Then each of the elements inside the red curve is guaranteed to be greater than x.

As in step 3, let x be the median of medians. Then either A' doesn't contain any items greater than x, or A' doesn't contain any items less than x. However, as we will see, there are lots of elements greater than x and there are lots of elements less than x. In particular, each (non-remainder) group whose median is less than x contains at least three elements less than x, and each (non-remainder) group whose median is greater than x contains at least three elements greater than x. Moreover, there are at least $\lceil n/5 \rceil - 1$ non-remainder groups, of which at least $\left\lfloor \frac{1}{2} \lceil n/5 \rceil - 2 \right\rfloor$ have median less than x and at least $\left\lfloor \frac{1}{2} \lceil n/5 \rceil - 2 \right\rfloor$ have mean greater than x. Finally, the group with x as its median contains two elements greater than x and two elements less than x (unless $n < 5$, but you can check the following inequalities in that case separately if you like). Thus

$$\left(\#\text{ elts of } A \text{ less than } x\right) \geq 3\left(\left\lfloor \tfrac{1}{2} \lceil n/5 \rceil - 2 \right\rfloor\right) + 2 \geq 3\left(\tfrac{1}{2} \lceil n/5 \rceil - 2 - \tfrac{1}{2}\right) + 2 > \tfrac{3}{10} n - 6.$$

Similarly,

$$\left(\#\text{ elts of } A \text{ greater than } x\right) \geq \tfrac{3}{10} n - 6.$$

Therefore A' must exclude at least $\frac{3}{10} n - 6$ elements of A, which means that A' contains at most $n - \left(\frac{3}{10} n - 6\right) = \frac{7}{10} n + 6$ elements. See Figure 1.1 for an illustration.

Let's now put together all the information we know about the running time of SELECT. Let $T(n)$ denote the worst-case running time of SELECT on an array of size n. Then:

- Step 1 clearly takes $O(n)$ time.
- Step 2 takes $O(n)$ time since it takes $O(1)$ time to find the median of each 5-element group.
- Step 3 takes $T(\lceil n/5 \rceil)$ time since there are $\lceil n/5 \rceil$ submedians.
- Step 4 takes $O(n)$ time, as explained in §7.1 of CLRS.
- Step 5 takes at most $T\left(\frac{7}{10} n + 6\right)$ since the new subarray has at most $\frac{7}{10} n + 6$ elements.[2]

[2] Technically, we haven't proved that $T(n)$ is an increasing function of n. However, if we let $T'(n) = \max\{T(m) : m \leq n\}$, then $T'(n)$ is an increasing function of n, and any asymptotic upper bound on $T'(n)$ will also be an asymptotic upper bound on $T(n)$.

Thus

$$\boxed{T(n) \le T(\lceil n/5 \rceil) + T\left(\tfrac{7}{10}n + 6\right) + O(n).} \tag{1.1}$$

1.3.3 Solving the Recurrence

How does one go about solving a recurrence like (1.1)? Possible methods include:

- Substitution Method: make a guess and prove it by induction.
- Recursion Tree Method: branching analysis.
- Master Method: read off the answer from a laundry list of already-solved recurrences.[3]

We will use the substitution method this time. Our guess will be that $T(n)$ is $O(n)$, i.e., that there exists a positive constant c such that $T(n) \le cn$ for all sufficiently large n. We won't choose c just yet; later it will become apparent which value of c we should choose.

In order to deal with the base case of an inductive argument, it is often useful to separate out the small values of n. (Later in the argument we will see how this helps.) Since SELECT always terminates, we can write

$$T(n) \le \begin{cases} O(1), & n < 140 \\ T(\lceil n/5 \rceil) + T\left(\tfrac{7}{10}n + 6\right) + O(n), & \text{for all } n. \end{cases}$$

In other words, there exist positive constants a, b such that $T(n) \le b$ for $n < 140$ (for example, $b = \max\{T(m) : m < 140\}$) and $T(n) \le T(\lceil n/5 \rceil) + T\left(\tfrac{7}{10}n + 6\right) + an$ for all n. Once this is done, we can start our inductive argument with the base case $n = 140$:

Base case.

$$\begin{aligned} T(140) &\le T(\lceil 140/5 \rceil) + T\left(\tfrac{7}{10}140 + 6\right) + an \\ &\le b + b + an \\ &= 2b + an \\ &= 2b + 140a. \end{aligned}$$

Thus, the base case holds if and only if $2b + 140 \le c(140)$, or equivalently $c \ge a + \frac{b}{70}$.

Inductive step. Suppose $n \ge 140$. Then the inductive hypothesis gives

$$\begin{aligned} T(n) &\le c(\lceil n/5 \rceil) + c\left(\tfrac{7}{10}n + 6\right) + an \\ &\le c\left(\tfrac{n}{5} + 1\right) + c\left(\tfrac{7}{10}n + 6\right) + an \\ &= \tfrac{9}{10}cn + 7c + an \\ &= cn + \underbrace{\left(\frac{-cn}{10} + 7c + an\right)}. \end{aligned}$$

Thus all we need is for the part with the brace to be nonpositive. This will happen if and only if $c \ge \frac{10an}{n-70}$. On the other hand, since $n \ge 140$, we know $\frac{n}{n-70} \le 2$. (This is why we wanted a base case of $n = 140$. Though, any base case bigger than 70 would have worked.) Thus the part with the brace will be nonpositive whenever $c \ge 20a$.

[3] The master method is a special case of a general technique known as *literature search*, which is of fundamental importance in not just computer science but also physics, biology, and even sociology.

Putting these two together, we find that the base case and the inductive step will both be satisfied as long as c is greater than both $a + \frac{b}{70}$ and $20a$. So if we set $c = \max\left\{20a, \frac{b}{70}\right\} + 1$, then we have $T(n) \leq cn$ for all $n \geq 140$. Thus $T(n)$ is $O(n)$.

6.046J / 18.410J Design and Analysis of Algorithms
Spring 2012