# CS 341: Algorithms

Charles Shen

Fall 2016, University of Waterloo

Notes written from Jeffrey Shallit's lectures.

Feel free to email feedback to me at ccshen902@gmail.com.

# Contents

**1 Selection in deterministic linear time** **1**

**2 Lower bounds** **4**

**3 Adversary Strategy** **5**

3.1 Adversary Arguments in General . . . . . 5
3.2 Finding Both the Maximum and Minimum of a List of n Numbers 6
3.3 Remarks on Constructing an Adversary Strategy . . . . . 9
3.4 Finding the Second Largest . . . . . 10
3.5 A lower bound for finding the median . . . . . 10

# 1 Selection in deterministic linear time

← October 25, 2016

In the selection problem, the goal is to determine the $i$-th smallest element from an unsorted list of $n$ elements in linear time.
"Deterministic" here means that no random numbers are used.
For simplicity, suppose that all elements are distinct, but this is not crucial as a small modification will make it work any list.

We want to cleverly choose a pivot to partition the list for selection.
So we have the "medians-of-five" algorithm by Blum, Floyd, Pratt, Rivest, and Tarjan (1973).

Idea:
Split the list into $\lfloor \frac{n}{5} \rfloor$ groups of 5 elements, and one additional group of at most 4 elements.
Then sort each group of 5 elements, and find the median of each group of 5. This can be done in constant time, so sorting each group of 5 costs $O(n)$.
Note that that median is the 3rd (center) element in the group of 5. This gives a list of $\lfloor \frac{n}{5} \rfloor$ medians.
Now recursively determine the median of these $\lfloor \frac{n}{5} \rfloor$ medians, which is the element we'll partition around.

We arrive at the algorithm:

**function** SELECT($A, i$)
  $n \leftarrow |A|$
  **if** $n < 60$ **then**
    SORT($A$)
    **return** $i$-th smallest element
  **else**
    $m \leftarrow \lfloor \frac{n}{5} \rfloor$
    divide A up to $m$ groups of 5 elements, with at most one remaining group of $\leq 4$ elements
    sort each of the $m$ groups in ascending order
    $M \leftarrow$ array of medians of each group
    $x \leftarrow$ SELECT($M$)$\lceil m/2 \rceil$ ▷ median of all the medians
    $k \leftarrow$ X-PARTITION($A, x$) ▷ partition array A into elements $\leq$ x and elements $>$ x; returns number of elements on "low side" of the partition
    **if** $i = k$ **then**
      **return** $x$
    **else if** $i < k$ **then**

```
            return SELECT(A[1..k − 1], i)
        else
            return SELECT(A[k + 1..n], i − k)
        end if
    end if
end function
```

Let's obtain a lower bound on the number of elements $> x$, the median of medians.
Here's a picture for $n = 37$:

```
smallest   S    S    S    S    *    *    *    *

           S    S    S    S    *    *    *    *

           S    S    S    m    L    L    L

           *    *    *    L    L    L    L

largest    *    *    *    L    L    L    L

m = median of medians
```

There are $\left\lfloor \frac{n}{5} \right\rfloor$ total columns in which 5 elements appear.
Of these, at least $\frac{1}{2}$ (more precisely, $\lceil \lfloor n/5 \rfloor / 2 \rceil$) contain an L. All of these columns, except the one where $x$ appears, contributes 3 to the count of L's; the one where $x$ appears contributes 2.

The conclusion is that $\geq 3 \lceil \lfloor n/5 \rfloor / 2 \rceil - 1$ elements are L's, that is, greater than $x$. Hence, at most $\leq n - (3 \lceil \lfloor n/5 \rfloor / 2 \rceil - 1)$ elements are $\leq x$.

Then, $n - (3 \lceil \lfloor n/5 \rfloor / 2 \rceil - 1) \leq 7n/10 + 3$ for all $n \geq 1$. To prove this, see that

$$\lfloor n/5 \rfloor \geq n/5 - 1$$

so

$$ceilfloorn/5/2 \geq n/10 - 1/2$$

then

$$3 \lceil \lfloor n/5 \rfloor / 2 \rceil - 1 \geq 3n/10 - 5/2$$

hence

$$n - (3 \lceil \lfloor n/5 \rfloor / 2 \rceil - 1) \leq 7n/10 + 3$$

We can also claim that at most $(7n/10 + 3)$ elements are $\geq x$ using the same proof above.
So whether we recurse in the smaller elements or larger elements, we can use this bound.

It follows that the time $T(n)$ to select from a list of n elements satisfies the following inequality:

$$T(n) \leq T(\lfloor n/5 \rfloor) + T(r) + dn$$

where $r \leq 7n/10 + 3$, and the $dn$ term soaks up the time needed to do the sorting of 5-element groups, and the partitioning.

Suppose that this recurrence obeys $T(n) \leq cn$ for some $c$,

$$\begin{aligned} T(n) &\leq T(\lfloor n/5 \rfloor) + T(r) + dn \\ &\leq cn/5 + cr + dn \\ &\leq cn/5 + c(7n/10 + 3) + dn \\ &\leq 9cn/10 + 3c + dn \end{aligned}$$

and we want this to be less than $cn$.

Now, $9cn/10 + 3c + dn \leq cn \iff (c/10 - d)n \geq 3c$
Now $n$ had better be bigger than 30 here, or else we are in a heap o' trouble. So let's assume $n \geq 60$. Then $(c/10 - d)n \geq (c/10 - d)60$, and if this is to be $\geq 3c$, then we must have $6c - 60d \geq 3c$, and so $c \geq 20d$.
If $n \geq 60$, and $c \geq 20d$, then the induction step indeed works.

To complete the proof, it remains to see that $T(n) \leq cn$ for $n = 1, 2, 3, \ldots, 59$. To do this, simply choose

$$c = max(20d, T(1), T(2)/2, ..., T(59)/59)$$

That looks like cheating, but it works.
Now the basis (namely $1 \leq n \leq 59$) works out just fine, and so does the induction step.
The presence of these large constants (such as $c = 20d$) suggests that this algorithm is of more theoretical than practical interest.
However, with a bit more work, the number of comparisons can be reduced quite a lot, down to $(3 + o(1))n = 3n + o(n)$.

# 2 Lower bounds

Can we prove that an algorithm is best possible? Sometimes.

We argued that any comparison-based algorithm for sorting must use $\Omega(n \lg n)$ time. And in reference to the convex hull algorithm, we argued that we couldn't do better than $\Omega(n \lg n)$, because we could use convex hull as a subroutine to create a sorting algorithm; if we could solve convex hull in $o(n \lg n)$ time, then we would end up being able to sort in $o(n \lg n)$ time, a contradiction.

By "lower bounds", we mean a lower bound on the complexity of a *problem*, not an algorithm. Basically we need to prove that *no algorithm*, no matter how complicated or clever, can do better than our bound.
We don't need each algorithm to be "slow" as $f(n)$ on the *same* input, but we do need each algorithm to be "slow" as $f(n)$ on *some* input.
Also, a lower bound doesn't rule out the possibility that some algorithm might be very fast on *some inputs* (but not all).
What do we mean by complexity of a problem?
We say that a function $f$ is a lower bound on the complexity of a problem $P$ if, for *every algorithm* $A$ to solve $P$, and every positive integer $n$, there exists some input $I$ of size $n$ such that $A$ uses at least $f(n)$ steps on input $I$.

The easiest kind of lower bound is the "information-theoretic bound".
Idea: every algorithm that uses *only* yes/no questions to decide among $k$ possible alternatives, *must* ask at least $(\lg k)$ questions in the worst case.

*Proof.* Use an "adversary argument".
The adversary doesn't decide on one of the alternatives ahead of time, but rather it answers the algorithm's questions by always choosing the alternative that maximizes the size of the solution space.
Since the questions are yes/no, the solution space goes down in size by a factor of at most two with each question.
The algorithm cannot answer correctly with certainty unless the solution space is of size 1.
Thus at least $(\lg k)$ questions are needed. □

**Theorem 2.0.1.** *Any comparison-based sorting algorithm must use at least* $\lg(n!) = \Omega(nlogn)$ *questions in the worst-case.*

*Proof.* Use the information-theoretic bound, and observe that the space of possible solutions includes the $n!$ different ways the input could be ordered. □

# 3 Adversary Strategy

## 3.1 Adversary Arguments in General

More generally, we can think of a lower bound proof as a game between the algorithm and an "adversary".
The algorithm is "asking questions" (for example, comparing two elements of an array), while the adversary is answering them (providing the results of the comparison).

The adversary should be thought of as a very powerful, clever being that is answering in such a way as to make your algorithm run as slowly as possible. The adversary cannot "read the algorithm's mind", but it can be prepared for anything the algorithm might do.
Finally, the adversary is not allowed to "cheat"; that is, the adversary cannot answer questions inconsistently. The adversary does not not need to have a particular input in mind at all times when it answers the questions, but when the algorithm is completed, there must be at least one input that matches the answers the adversary gave (otherwise it would have cheated).

The algorithm is trying to run as quickly as possible.
The adversary is trying (through its cleverness) to *force* the algorithm to run slowly.
This interaction proves that the lower-bound obtained by this type of argument applies to *any possible* algorithm from the class under consideration.

**Theorem 3.1.1.** *Every comparison-based algorithm for determining the minimum of a set of $n$ elements must use at least $(\frac{n}{2})$ comparisons.*

*Proof.* Every element must participate in at least one comparison; if not, the not compared element can be chosen (by an adversary) to be the minimum.
Each comparison compares 2 elements.
Hence, at least $(\frac{n}{2})$ comparisons must be made. □

**Theorem 3.1.2.** *Every comparison-based algorithm for determining the minimum of a set of $n$ elements must use at least $(n-1)$ comparisons.*

*Proof.* To say that a given element, $x$, is the minimum, implies that (1) every other element has won at least one comparison with another element (not necessarily $x$). (By "$x$ wins a comparison with $y$" we mean $x > y$.)
Each comparison produces at most one winner.

Hence at least $(n-1)$ comparisons must be used.

To convert this to an adversary strategy, do the following: for each element, record whether it has won a comparison ($W$) or not ($N$).
Initially all elements are labelled with $N$.
When we compare an $N$ to an $N$, select one arbitrarily to be the $W$.
If no values assigned, choose them to make this so.
If values are assigned, decrease the loser (if necessary) to make this so.
When we compare $W$ to $N$, always make the $N$ the loser and set or adjust values to make sure this is the case. (You might need to decrease the loser's value to ensure this.)
When we compare $W$ to $W$, use the values already assigned to decide who is the winner.

Now each comparison increases the total number of items labelled $W$ by at most 1.
In order to "know" the minimum, the algorithm must label $(n-1)$ items $W$, so this proves at least $(n-1)$ comparisons are required.
The values the adversary assigned are the inputs that forced that algorithm to do the $(n-1)$ comparisons. □

## 3.2 Finding Both the Maximum and Minimum of a List of n Numbers

It is possible to compute both the maximum and minimum of a list of n numbers, using $(\frac{3}{2}n-2)$ comparisons if n is even, and $(\frac{3}{2}n-\frac{3}{2})$ comparisons if n is odd.
It can be proven that *no comparison-based method* can correctly determine both the max and min using fewer comparisons in the worst case.

We do this by constructing an adversary argument.

In order for the algorithm to correctly decide that $x$ is the minimum and $y$ is the maximum, it must know that

1. every element other than $x$ has won at least one comparison (i.e. $x > y$ if $x$ wins a comparison with $y$),

2. every element other than $y$ has lost at least one comparison

Calling a win $W$ and a loss $L$, the algorithm must assign $(n-1)$ $W$'s and $(n-1)$ $L$'s. That is, the algorithm must determine $(2n-2)$ "units of information" to always give the correct answer.

We now construct an adversary strategy that will force the algorithm to learn its $(2n-2)$ "units of information" as slowly as possible.
The adversary labels each element of the input as $N$, $W$, $L$, or $WL$. These labels may change over time.
$N$ signifies that the element has never been compared to any other by the algorithm.
$W$ signifies the element has won at least one comparison.
$L$ signifies the element has lost at least one comparison.
$WL$ signifies the element has won at least one and lost at least one comparison.

← October 27, 2016

So the the adversary uses the following table.

| labels when comparing elements $(x, y)$ | the adversary's response | the new label | unit of information given to the alg. |
|---|---|---|---|
| $(N, N)$ | $x > y$ | $(W, L)$ | 2 |
| $(W, N)$ or $(WL, N)$ | $x > y$ | $(W, L)$ or $(WL, L)$ | 1 |
| $(L, N)$ | $x < y$ | $(L, W)$ | 1 |
| $(W, W)$ | $x > y$ | $(W, WL)$ | 1 |
| $(L, L)$ | $x > y$ | $(WL, L)$ | 1 |
| $(W, L)$ or $(WL, L)$ or $(W, WL)$ | $x > y$ | no change | 0 |
| $(WL, WL)$ | consistent with assigned values | no change | 0 |

The adversary also tentatively assigns values to the elements, which may change over time.
However, they can only change in a fashion *consistent* with previous answers. That is, an element labelled $L$ may only *decrease* in value (since it lost all previous comparisons, if it is decreased, it will still lose all of them), and an element labelled $W$ may only increase in value.
An element labelled $WL$ cannot change.

**Example.** Consider the following sequence of possible questions asked by the algorithm and the adversary's responses:

| Algorithm compares | Adversary's responses and worksheet x1 | x2 | x3 | x4 | x5 | x6 | |
|---|---|---|---|---|---|---|---|
| (x1, x2) | W-20 | L-10 | N | N | N | N | |
| (x4, x5) | W-20 | L-10 | N | W-30 | L-15 | N | |
| (x1, x4) | W-40 | L-10 | N | WL-30 | L-15 | N | (*) |
| (x3, x6) | W-40 | L-10 | W-11 | WL-30 | L-15 | L-2 | |
| (x2, x5) | W-40 | WL-10 | W-11 | WL-30 | L-7 | L-2 | |
| (x3, x1) | WL-40 | WL-10 | W-50 | WL-30 | L-7 | L-2 | |
| (x2, x4) | WL-40 | WL-10 | W-50 | WL-30 | L-7 | L-2 | (**) |
| (x5, x6) | WL-40 | WL-10 | W-50 | WL-30 | WL-7 | L-2 | |

At this point the algorithm knows that $x_3$ is the maximum, since it is the only element that has never lost a comparison, and $x_6$ is the minimum, since it is the only element that has never won a comparison.
Note that in step (*), the adversary was forced to reassign the value he had previously assigned to $x_1$, since $x_1$ had to win the comparison against $x_4 = 30$. This is permitted, since $x_1$ had won every previous comparison and so increasing its value ensures consistency with previous answers.
Also note that step (**) is superfluous, as the algorithm didn't learn any new information.

**Theorem 3.2.1.** *Every* comparison-based *method for determining both the maximum and minimum of a set of $n$ numbers must use at least $(\frac{3}{2}n - 2)$ comparisons (if $n$ even), or $(\frac{3}{2}n - \frac{3}{2})$ comparisons (if $n$ odd), in the worst case.*

*Proof.* Suppose that $n$ is even.
As shown before, the algorithm must learn $(2n - 2)$ units of information.
The most it can learn in one comparison is 2 units; when both elements have never participated before in a comparison, labelled by $(N, N)$. This can happen at most $(\frac{n}{2})$ times.
To learn the remaining $(n - 2)$ units of info., the algorithm must ask $(n - 2)$ questions.
The total number of questions is therefore at least $\frac{n}{2} + n - 2 = \frac{3}{2}n - 2$.
The same kind of argument works for $n$ odd. □

So we have a lower bound for the problem of finding both the maximum and minimum of a set of $n$ numbers.

So the algorithm is as followed:
If $n$ is even, compare $x[1]$ with $x[2]$ (determining both the maximum and minimum with 1 comparison), $x[3]$ with $x[4]$, etc. We get $(\frac{n}{2})$ maxima and $(\frac{n}{2})$ minima.
To find the maximum, use the usual method on the $(\frac{n}{2})$ maxima, which uses $(\frac{n}{2} - 1)$ comparisons, and the same for the minima.
The total cost is $\frac{n}{2} + \frac{n}{2} - 1 + \frac{n}{2} - 1 = \frac{3}{2}n - 2$. A similar method works when $n$ is odd.

Notice that each comparison adds at most one $W$, and we need to get $(n-1)$ $W$'s before we know the remaining element must be the minimum.
So we need at least $(n-1)$ comparisons.
If the algorithm declares a minimum before getting $(n-1)$ $W$'s, then there are at least two elements without $W$, so if the algorithm declares one of the minimum, the adversary can truthfully produce the other as the minimum.

## 3.3 Remarks on Constructing an Adversary Strategy

Lower bound arguments are some of the most difficult and deepest areas in theoretical computer science.

**Tips for producing an adversary strategy**:

Recall that the lower bound argument can be viewed as a game between you (playing the role of an algorithm) and an adversary.
You want the algorithm to run quickly (for example, by making a small number of queries to the input).
The adversary wants the algorithm to run slowly.
The adversary "wins" (and the lower bound of $f(n)$ is proved) if he/she can force the algorithm to execute $f(n)$ steps.

A proof behaves something like: for all sequences of queries into the input made by the algorithm, there exists a sequence of replies by the adversary, such that the algorithm is forced to execute f(n) steps. So your adversary argument must apply to *every* possible "move" the algorithm could make.

In constructing its replies, the adversary can do as much or as little bookkeeping as necessary.
There is no requirement that the adversary's computation be efficient or run in any particular time bound.

## 3.4 Finding the Second Largest

Problem: given a list of $n$ elements, find the 2nd largest.

The naive method (finding the largest, remove it, and then find the largest again) takes $2n - 3$ comparisons.
Instead, we can do a "tournament" method.
Have the elements in pairs, the larger one (winner) advances; then the 2nd largest (runner-up) is among those defeated by the winner! This is because any other element must have played two other elements superior to it.

This algorithm uses $(n-1)$ comparisons to determine the largest, plus $\lceil \lg n \rceil - 1$ comparisons to determine the runner-up (2nd largest).
The total is $(n + \lceil \lg n \rceil - 2)$ comparisons.

We can prove that the algorithm just given is *optimal*, in the sense that no comparison-based algorithm uses fewer than $(n + \lceil \lg n \rceil - 2)$ comparisons in the worst case.

*Note*: see Lecture 14 for full proof! (Click here.)

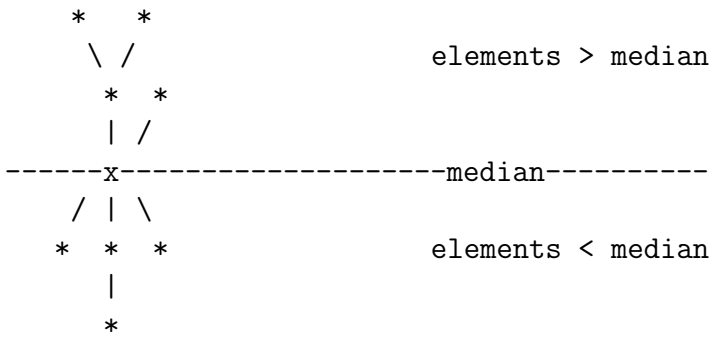
## 3.5 A lower bound for finding the median

The best algorithm known (in terms of number of comparisons used) uses $3n + o(n)$ comparisons in the worst case (see Section 1).
The best *lower bound* known for the median-finding problem is $2n + o(n)$ comparisons.

We will prove a weaker lower bound: $\frac{3}{2}n + O(1)$, where $n$ odd for convenience.

First, note that any comparison-based algorithm for finding the median must use at least $(n-1)$ comparisons. That's because the algorithm *must* establish the relationship of each element to the median—either by direct comparison, or by comparison with another element whose relationship to the median is known.

The algorithm *must* eventually gain enough knowledge equivalent to the following tree:




Note that if there were an element that didn't participate in a comparison, we couldn't know the median.
For an adversary could change the value of that element from, say, less than the median to greater than the median, changing the median to some other element.
Similarly, if there is some element whose relationship to the median is not known (because, for example, it is greater than some element that is less than the median), we could simply move it from one side of the line above to the other, without changing comparisons. This would also change the median.

Now there are n elements in the tree above, so there are $(n-1)$ edges, and hence $(n-1)$ comparisons.
We now improve this to $\frac{3}{2}n + O(1)$.
First, we call a comparison involving x crucial, say x against y, if it is the first comparison where

1. $x < y$ and $y \leq$ median OR if
2. $x > y$ and $y \geq$ median

So the algorithm must perform at least $(n-1)$ crucial comparisons in order to determine the median.

← November 1, 2016