This little paper describes the selection problem, the now-classic algorithm for solving it, and why that algorithm uses the number 5 instead of some other number. The algorithm closely resembles quicksort, and is often used as part of quicksort to choose the pivot value.

## 1 Problem

We are given a list of $n$ distinct numbers, and a number $k$ such that $0 \leq k < n$. We would like to find the $k$th-smallest number in the list — that is, the number that would be at index $k$, if the list were sorted in increasing order. This is called the *selection problem*.

The special case $k = 0$ corresponds to finding the minimum of the numbers; the special case $k = n - 1$ corresponds to finding the maximum. There are obvious $\mathcal{O}(n)$ algorithms for these special cases. The special case $k = \frac{n}{2}$ corresponds to finding the median; I don't see any obvious $\mathcal{O}(n)$ algorithm for this.

The selection problem can be solved by simply sorting the list and then picking out the $k$th element. Assuming a $\mathcal{O}(n \log n)$ sorting algorithm, this is $\mathcal{O}(n \log n)$. The algorithm presented below solves the selection problem in time $\mathcal{O}(n)$.

## 2 Algorithm

In this section we describe the algorithm $\mathit{selection}(L, k)$, which takes as input a list $L$ of length $n$ and a number $k$ such that $0 \leq k < n$. The function is recursive. The base case of the recursion occurs when the length $n$ is small — say, $n = 5$. Then we just compute the $k$th-smallest number manually. The details are not important. If $L$ is too long for the base case, then we proceed as follows.

Divide $L$ into (approximately) $\frac{n}{5}$ sublists of length 5 each. For each of these $\frac{n}{5}$ lists, compute the median manually. Put these $\frac{n}{5}$ medians into a new list $M$. Recursively call this algorithm to compute the median of $M$; that is, call $\mathit{selection}\left(M, \frac{n}{10}\right)$. This median of medians is our *pivot* value.

Scan through the original list of $n$ numbers once, forming two new lists as you go: a list $A$ of all numbers less than the pivot, and a list $B$ of all numbers greater than the pivot. If $k < \mathit{len}(A)$, then the answer we're looking for must be in $A$; recursively call $\mathit{selection}(A, k)$ and return the result. If $k = \mathit{len}(A)$, then the answer we're looking for is the pivot; return it. If $k > \mathit{len}(A)$, then the answer we're looking for must be in $B$; recursively call $\mathit{selection}(B, k - \mathit{len}(A) - 1)$ and return that result.

## 3 Efficiency

The key to the algorithm is that the pivot is pretty close to the median of the list. Thus the lists $A$ and $B$ are both significantly smaller than the original list $L$, and the size of the problem is greatly reduced, no matter which one we recurse on. Here are the details.

The pivot is greater than half of the $\frac{n}{5}$ medians. Think about one of these medians, that is less than the pivot. It was greater than 2 of the other numbers in its original sublist of 5 numbers. Thus the pivot is greater than at least 3 numbers from that sublist (the median and 2 others). The same holds for each of the $\frac{n}{10}$ medians that the pivot is greater than. Thus the pivot is greater than at least $\frac{3n}{10}$ of the original $n$ numbers in the list. By a symmetric argument, the pivot is less than at least $\frac{3n}{10}$ of the original numbers in the list. In other words, the pivot is greater than at least 30% of the original numbers and less than at least 30% of the original numbers; it's somewhere in the middle 40% of the sorted version of the list. Consequently, the size of $A$ is at most $\frac{7n}{10}$, and the size of $B$ is at most $\frac{7n}{10}$.

Let $T(n)$ be the running time of *selection* on a list of length $n$. Constructing the list $M$ of $\frac{n}{5}$ medians takes time proportional to $n$; call it $cn$, where $c$ is some constant. Finding the median of that list takes time $T\left(\frac{n}{5}\right)$. The recursive call on either $A$ or $B$ takes time at most $T\left(\frac{7n}{10}\right)$. Thus

$$T(n) \leq cn + T\left(\frac{n}{5}\right) + T\left(\frac{7n}{10}\right).$$

Now envision the execution of *selection* as a binary tree, where the nodes are calls to *selection*. Each call to *selection* makes two recursive calls — that's why each tree node has two children — until we reach the base-case calls, which show up in the tree as leaves. Consider the size of the list that is being selected on at each node of the tree. The root node represents a *selection* call on a list of length $n$. The left child of the root represents a call on a list of length $\frac{1}{5}n$. Its left child is on a list of length $\frac{1}{5}\left(\frac{1}{5}n\right)$ and its right child is on a list of length $\frac{7}{10}\left(\frac{1}{5}n\right)$. We can work out the list lengths for the right child of the root, and its children, and all of the other nodes in the tree similarly. Here's what we get:

$$n$$

$$\frac{1}{5}n,\quad \frac{7}{10}n$$

$$\frac{1}{5}\frac{1}{5}n,\quad \frac{7}{10}\frac{1}{5}n,\quad \frac{1}{5}\frac{7}{10}n,\quad \frac{7}{10}\frac{7}{10}n$$

$$\frac{1}{5}\frac{1}{5}\frac{1}{5}n,\quad \frac{7}{10}\frac{1}{5}\frac{1}{5}n,\quad \frac{1}{5}\frac{7}{10}\frac{1}{5}n,\quad \frac{7}{10}\frac{7}{10}\frac{1}{5}n,\quad \frac{1}{5}\frac{1}{5}\frac{7}{10}n,\quad \frac{7}{10}\frac{1}{5}\frac{7}{10}n,\quad \frac{1}{5}\frac{7}{10}\frac{7}{10}n,\quad \frac{7}{10}\frac{7}{10}\frac{7}{10}n$$

...

The tree is not really infinite, but let's pretend for a moment that it is; this can only increase the running time estimate.

Notice that at each node the total time taken is $c$ times the list length, plus the total time in the subtrees. In other words, each node is itself responsible for $c$-times-list-length work, not including work counted in other nodes. So if we simply multiply the tree above by $c$, then we get a tree of all of the running times at the nodes:

$$cn$$

$$c\frac{1}{5}n,\quad c\frac{7}{10}n$$

$$c\frac{1}{5}\frac{1}{5}n,\quad c\frac{7}{10}\frac{1}{5}n,\quad c\frac{1}{5}\frac{7}{10}n,\quad c\frac{7}{10}\frac{7}{10}n$$

$$c\frac{1}{5}\frac{1}{5}\frac{1}{5}n,\quad c\frac{7}{10}\frac{1}{5}\frac{1}{5}n,\quad c\frac{1}{5}\frac{7}{10}\frac{1}{5}n,\quad c\frac{7}{10}\frac{7}{10}\frac{1}{5}n,\quad c\frac{1}{5}\frac{1}{5}\frac{7}{10}n,\quad c\frac{7}{10}\frac{1}{5}\frac{7}{10}n,\quad c\frac{1}{5}\frac{7}{10}\frac{7}{10}n,\quad c\frac{7}{10}\frac{7}{10}\frac{7}{10}n$$

$$\ldots$$

The total time taken by the initial call to $selection(L, k)$ is the sum of all of these terms in the whole tree. How can we sum up such a thing? Level by level. For example, look at the grandchildren of the root. The sum of their times is

$$\left(\frac{1}{5}\frac{1}{5} + \frac{7}{10}\frac{1}{5} + \frac{1}{5}\frac{7}{10} + \frac{7}{10}\frac{7}{10}\right) cn = \left(\frac{1}{5} + \frac{7}{10}\right)^2 cn.$$

Look at the next level down; the sum of the times is

$$\left(\frac{1}{5}\frac{1}{5}\frac{1}{5} + \frac{7}{10}\frac{1}{5}\frac{1}{5} + \frac{1}{5}\frac{7}{10}\frac{1}{5} + \frac{7}{10}\frac{7}{10}\frac{1}{5} + \frac{1}{5}\frac{1}{5}\frac{7}{10} + \frac{7}{10}\frac{1}{5}\frac{7}{10} + \frac{1}{5}\frac{7}{10}\frac{7}{10} + \frac{7}{10}\frac{7}{10}\frac{7}{10}\right) cn = \left(\frac{1}{5} + \frac{7}{10}\right)^3 cn.$$

This is a bit miraculous: the total running time at level $\ell$ is simply $\left(\frac{1}{5} + \frac{7}{10}\right)^\ell cn$! (You can check it at levels 0 and 1; if you've taken CS 202, then you can prove it for all levels using induction.) Therefore the total running time of the initial call to $selection(L, k)$ is

$$\sum_{\ell=0}^{\infty} \left(\frac{1}{5} + \frac{7}{10}\right)^\ell cn = cn \sum_{\ell=0}^{\infty} \left(\frac{9}{10}\right)^\ell.$$

The series on the right is a geometric series with common ratio $x = \frac{9}{10}$; its sum is $\frac{1}{1-x} = 10$. Therefore the total running time is $10cn$, and the selection algorithm is $\mathcal{O}(n)$.

## 4 5?!

It turns out that the number 5 is important. Decreasing 5 to 3 causes the running time to explode. Increasing 5 to 7, 9, etc. leads to $\mathcal{O}(n)$ behavior, but not quite as fast as the $\mathcal{O}(n)$ produced by 5 itself.

Consider a selection algorithm that uses sublists of size $2k + 1$. (The case described thus far corresponds to $k = 2$.) We divide $L$ into $\frac{n}{2k+1}$ sublists of size $2k + 1$. We compute the medians of the sublists in time $cn$, and then the median of those medians in time $T\left(\frac{n}{2k+1}\right)$. This median of medians is the pivot. The pivot is greater than $\frac{n}{4k+2}$ medians, each of which is greater than $k$ members of its sublist; thus the pivot is greater than at least $\frac{k+1}{4k+2}n$ of the original list items. By a symmetric argument, the pivot is less than at least $\frac{k+1}{4k+2}n$ of the list items. Thus each of the lists $A$ and $B$ have size at most

$$n - \frac{k+1}{4k+2}n = \frac{3k+1}{4k+2}n.$$

The running time of the selection algorithm on a list of size $n$ is therefore

$$T(n) \leq cn + T\left(\frac{1}{2k+1}n\right) + T\left(\frac{3k+1}{4k+2}n\right).$$

Following the same binary tree argument as above, we compute the running time to be

$$\sum_{\ell=0}^{\infty}\left(\frac{1}{2k+1}+\frac{3k+1}{4k+2}\right)^{\ell} cn = cn\sum_{\ell=0}^{\infty}\left(\frac{3k+3}{4k+2}\right)^{\ell}.$$

We obtain a geometric series with common ratio

$$x = \frac{3k+3}{4k+2}.$$

When $k = 2$ we have $x = \frac{9}{10}$ as before; good.

When $k = 1$ we have $x = 1$; the expression

$$cn\sum_{\ell=0}^{\infty} 1^{\ell} = 1 + 1 + 1 + \cdots$$

diverges — it is infinite. Does this mean that the running time is infinite? No; it just means that this particular way of analyzing the running time fails. It fails because we overestimated the height of the call tree as infinite. In reality, the height of the call tree is finite and the running time is finite.

So how tall is the call tree? Notice that at each node, the size $\frac{1}{2k+1}n$ of the left child's selection problem is less than the size $\frac{3k+1}{4k+2}n$ of the right child's problem. This means that the left recursion will reach the base case after fewer recursive calls; the left subtree will be shorter than the right subtree. In the entire call tree, the shortest path from the root to a leaf is the path that goes left at each node. How long is this path? Well, how many times can you multiply the problem size $n$ by the fraction $\frac{1}{2k+1}$ before you get down to the base case? About $\log_{2k+1} n$ times; that's the length of this path. Similarly, the longest path from the root to a leaf is the path that goes right at each node, and its length is about $\log_{\frac{4k+2}{3k+1}} n$. Therefore the actual height $h$ of the call tree satisfies

$$\log_{2k+1} n \leq h \leq \log_{\frac{4k+2}{3k+1}} n.$$

When $k = 1$, the call tree is of height at least $\log_{2k+1} n = \log_3 n$. Therefore the total running time may be as bad as

$$cn \sum_{\ell=0}^{\log_3 n} 1^\ell = cn(\log_3 n + 1),$$

which is $\mathcal{O}(n \log n)$.

When $k \geq 3$ the geometric series converges and the selection algorithm works in time $\mathcal{O}(n)$. In fact, the constants in the $\mathcal{O}(n)$ even make it appear to be faster than the $k = 2$ version. However, my understanding is that the $k = 2$ running time is less than the running time for any $k \geq 3$. I haven't carried out this analysis in detail, but one reason may be the height of the call tree, which is $\log_{\frac{4k+2}{3k+1}} n$. As $k$ increases, $\frac{4k+2}{3k+1}$ decreases toward $\frac{4}{3}$, the logarithm increases, and hence the height of the call tree increases. This suggests that we want the smallest $k$ that delivers $\mathcal{O}(n)$ performance; that's $k = 2$.

## 5 Quicksort

The quicksort algorithm on a list $L$ of $n$ distinct numbers can be summarized as follows. Choose one of the numbers to be the *pivot*; partition the numbers into those less than the pivot and those greater than the pivot; quicksort each of these two lists; combine the results. Good pivots, which are close to the median value in the list, lead to $\mathcal{O}(n \log n)$ running time; bad pivots, which are far from the median, lead to $\mathcal{O}\left(n^2\right)$ time.

One can always choose the pivot to be the median by making a call $\textit{selection}(L, n/2)$. The selection takes time $\mathcal{O}(n)$, but the partitioning step of quicksort is already $\mathcal{O}(n)$, so using selection does not increase the running time; it produces a quicksort algorithm with worst-case running time $\mathcal{O}(n \log n)$.

In our course we now have three $\mathcal{O}(n \log n)$ sorting algorithms: merge sort, quicksort with selection of pivot, and heap sort. Which is actually best? It comes down to the constants hidden in that $\mathcal{O}(n \log n)$. Merge sort is undesirable because it uses extra storage, which takes extra time to use. Quicksort with selection is undesirable because the selection process takes a lot of extra time. That leaves heap sort, which has no obvious drawbacks. However, my understanding is that most computer languages, libraries, applications, etc. use quicksort with some method pivot-choosing other than selection, such as random choice. This supposedly beats heap sort for typical workloads.