Selection Algorithms

Selection algorithms are those that aim at returning the kth smallest element from a set of numbers. This element is referred to as the kth **Order Statistic**, or we often say the element of **rank k**. For example, in the set

$$\{3, 15, 10, 4, 6, 7, 1\}$$

the element of rank 3 is the element 4 since it's the 3rd smallest in the set.

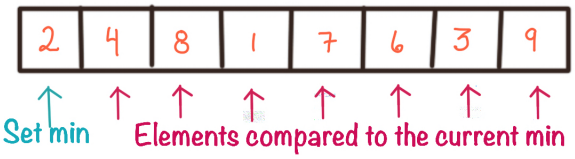
The most commonly used order statistic is the **median**: the value in the middle position of the sorted order of elements. Interestingly, you might assume that the only way to identify the median, or any other order statistic, is by first sorting the values, taking time $O(n \log n)$. As we shall see this lecture - we can actually do much better.

1 The selection problem: a randomized solution

We aim to solve the **selection problem**:

- **Input:** A set of n distinct numbers
- **Output:** The element of rank k.

Let's consider first a specific case of the above problem. Suppose that we looking for the *minimum* of a set of n elements. This is equivalent to selecting the *first* order statistic, $k = 1$. How many comparisons are necessary? The very trivial minimum-finding (and similarly maximum) algorithm uses $n - 1$ comparisons. It works by initializing the minimum to the first element, and comparing all other values to the current minimum, updating the min when necessary. It turns out that this is the *best* we can do: there is no way to find the minimum element of a list of n in *less* than $n - 1$ comparisons. So the minimum (and maximum) finding algorithms that you have probably been using for quite some time are **optimal**, and have a runtime of $O(n)$.



What if we were looking for the element of rank k? Can we also do this in $O(n)$ time? We shall see in this lecture, that yes indeed, it is possible to find the kth order statistic in $O(n)$ time.

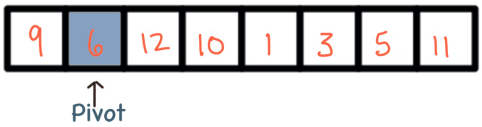
As a warm up, we look at a rather simple divide-and-conquer selection algorithm that uses a technique similar to Quicksort. This algorithm is called **Randomized-Select**, and although it is not *always* fast, it *expects* to run at $\Theta(n)$. Thus Randomized-Select is able to find the k-th order statistic in about the same time as we usually find the minimum or the maximum.

1.1 Randomized-Select

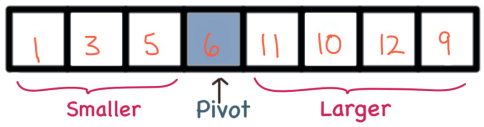
The magic behind Randomized-Select is that it quickly *partitions* the elements, making it faster to identify the kth largest value. We look first at how to carry out this partitioning:

Step 1: Random Partition:

- Select a random number from the array, called p for "*pivot*".



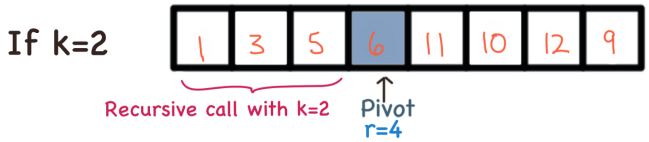
- Partition the elements as those that are *smaller* and *larger* than the pivot, keeping track of how many elements are smaller than the pivot. Return the **rank**, r of the pivot. In the example below, the partition step returns $r = 4$ since after the partitioning, the pivot is in position 4 in the array.



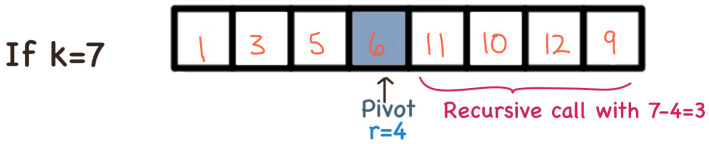
Runtime of partition step:
This partitioning procedure takes time $\Theta(n)$, since it examines each element once in order to decide if it is smaller or larger than the pivot.

Step 2: The recursive call:
Now let's look at how this helps us find the element of rank k.

- Suppose (we were lucky) and $r = k$. Then we found the kth order statistic! We are done and can simply return p. For example, suppose we were looking for the element of rank $k = 4$. Then we would return element 6, which is the pivot.
- Suppose $k < r$. Then the item of rank k is to the left of the pivot. In the example below, $r = 4$ and suppose we are looking for $k = 2$. In this case, we make a recursive call to randomized-select on the left subarray.



- Suppose $k > r$. Then the item of rank k is to the right of the pivot. In the example below, $r = 4$ and suppose we are looking for $k = 7$. We should look for the item of rank $7 - 4 = 3$ in the right subarray. Therefore we make a recursive call to randomized-select on the right-subarray using rank $k - r$.

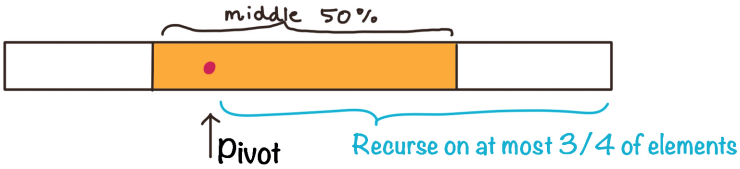


Worse-case run-time:
We could be very unlucky, and each time randomly select the maximum element as our pivot. Each partition operation would only reduce a problem of size n to one of size $n - 1$. The recursive equation for this runtime is $T(n) = T(n - 1) + c$, which we have seen in our section on recursion has a solution of $\Theta(n^2)$.

Expected run-time

Since the outcome of the pivot-selection step of this algorithm is *random*, then the runtime is actually *random*: sometimes it will be lucky and be fast, and other times, unlucky and slow. The actual runtime is a **random variable**. We are interested in the **expected** runtime of the algorithm - in simple terms, this refers to what we *expect* that runtime to be, over all possible inputs.

Consider momentarily the elements as if they were sorted. Notice that a "*good*" selection for the pivot is when the pivot is selected from the *middle* section of the array: the 50% portion of elements in the middle of all values. When our pivot is selected from this yellow section, the recursive call will be carried out on *at most* $\frac{3}{4}$n of the array:



What is the chance that we select a pivot from the middle section in yellow? Since it represents 1/2 the array, this chance is exactly 1/2. Suppose we only consider "good" selections - those that fall in this yellow section. How many times do we expect to select a pivot *until* we make a good selection from the middle section? If we have a 1/2 chance of an event happening, how many times do we need to try until we get it right? The answer is 2, and in fact it is a type of *geometric random variable*, but even if that is not known to you, the concept is quite intuitive. If you have a 1/6 chance of rolling a 3 on a die, then you expect to about every 6 rolls to result in a 3.

Now let's see how to put this all together. Let $T(n)$ be the runtime of this algorithm. After a good pivot is selected, the array size reduces from n to at most $\frac{3}{4}n$. After m such times, the array size is at most $n\left(\frac{3}{4}\right)^m$. Since the runtime of the partition algorithm on an array of size n is $\Theta(n)$, the runtime of the partition algorithm on the array of size $n\left(\frac{3}{4}\right)^m$ is at most:

$$cn\left(\frac{3}{4}\right)^m$$

for some constant c and large enough n. Now we need to sum over *all* the recursive calls, where every *second* time (on average) the array size shrinks to 3/4ths of its size . In the summation below, we simply add up the above numbers, for the first call when $m = 0$, and we multiply by 2 since only every second call will be "good".

$$\sum_{m=0}^{t} 2cn\left(\frac{3}{4}\right)^m$$

The sum terminates after t steps, which is whenever we find the element of rank k. We can upper bound this sum as usual:

$$\begin{aligned}\sum_{m=0}^{t} 2cn\left(\frac{3}{4}\right)^m &= 2cn\sum_{m=0}^{t}\left(\frac{3}{4}\right)^m \\ &\leq 2cn(4)\end{aligned}$$

Again, the last line above uses our usual trick that is for a sum of the form $\sum c^n$, described in our class worksheet. Therefore, the *expected* runtime of Randomized-Select is $O(n)$.

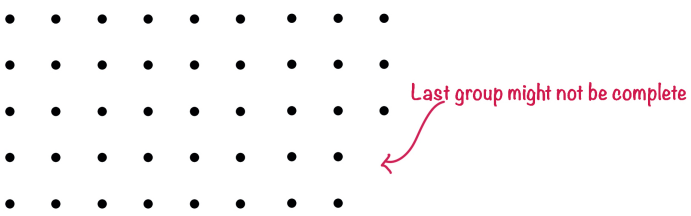
In summary, the worst-case runtime of this algorithm is $O(n^2)$, however, it is *expected* to run much faster: in time $O(n)$.

2 Select: Finding the kth order statistic in linear time

In this section we look at an algorithm that finds the kth order statistic in $O(n)$ worst-case time. This is an improvement over the previous algorithm, that has a *worst-case* of $O(n^2)$. The **Select** algorithm uses a similar partitioning idea. However, it carries out a partition that *guarantees* a good split - so the recursive call is on a substantially smaller array. Suppose we have as input an array of n distinct elements, and let $T(n)$ be the running time of this algorithm.

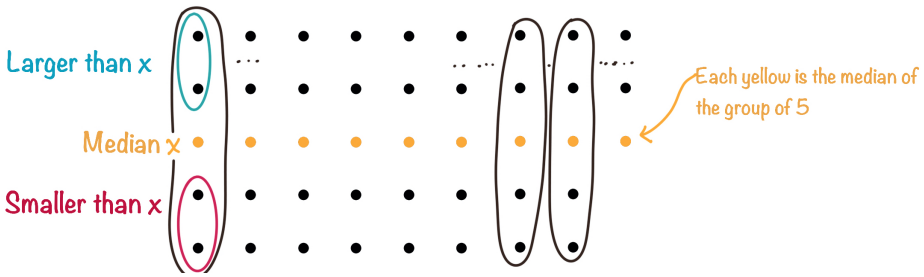
2.1 The Steps!

1. **Divide the n elements into $\lceil n/5 \rceil$ groups of elements of size 5.**



Time: $\boxed{\Theta(n)}$

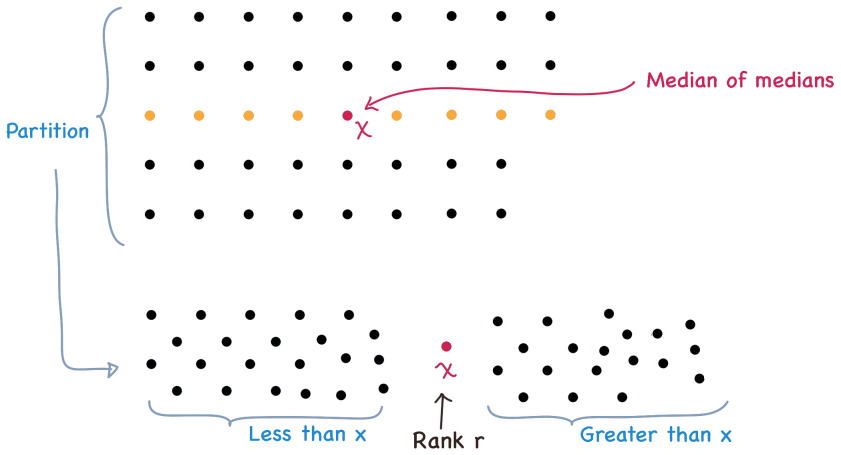
2. **Sort each group and find the median of each group.**



Time:. We can run insertion-sort on each group, then pick the median. Since each group is only size 5, running insertion-sort is still a constant, say c. Since there are $\lceil n/5 \rceil$ groups, the total run-time is $c \cdot \lceil n/5 \rceil$. This step is $\boxed{\Theta(n)}$

3. **Find the median of all the medians, called x.** This is a recursive call, which runs on all the yellow dots in the above picture. **Time:** $\boxed{T(n/5)}$

4. **Go through all elements and compare them to x, partitioning them into those that are smaller and those that are bigger.** As you do this, determine the **rank**, r, of x.

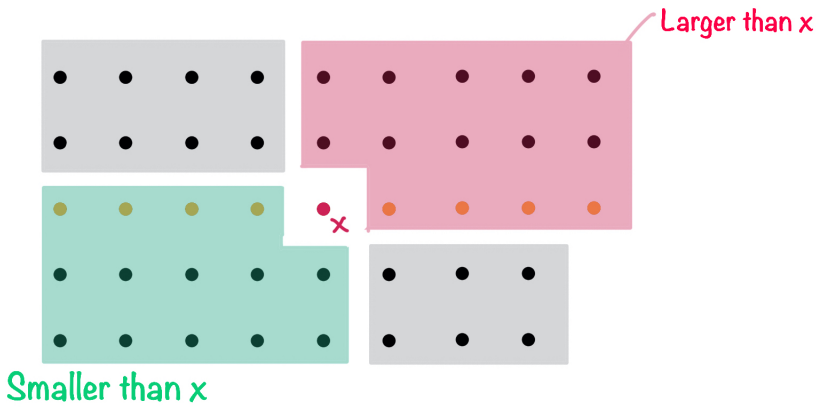


Time: $\boxed{\Theta(n)}$

5. **Compare r to k.** Proceed like in Randomized-Select.

   - If $r = k$, then x is actually the kth order statistic. Return x.
   - If $r < k$ then make a recursive call to all the elements that are *bigger* than x, and update the new rank to $k - r$
   - If $r > k$, then make a recursive call to all the elements that are *smaller* than x.

   Time: This depends on how many elements are involved in the recursive call. In the figure below, the red section consists of elements that are *definitely* larger than x, and those in green are those that are *definitely* smaller than x. The rest of the elements (grey) could be larger or smaller than x.



   Now suppose that $r < k$. The algorithm would recurse on *all* elements larger than x. That would be *all* the red elements, and *some* of the grey elements. We don't know how many that is, but in the worst case, we could assume it would be all red and *all* the grey elements. How many is that? In the green section, there are at least 3 elements per column, and at least $\lceil \frac{1}{2} \cdot \lceil \frac{n}{5} \rceil \rceil \geq \frac{n}{10}$ columns, for a total of at least $\frac{3}{10}n$ elements. To make things simpler, note that $\frac{3}{10}n \geq \frac{1}{4}n$. This means that the red and grey elements make up at most $\frac{3}{4}n$ elements. Therefore, the recursive call would be applied to at most $\frac{3}{4}n$ elements. Same thing if $r > k$. Thus the run time of this step is at most : $\boxed{T(\frac{3}{4}n)}$

2.2 The run-time analysis:

From above, we can see that the Select algorithm makes two recursive calls and the rest of the operations take time $\Theta(n)$. Thus the recurrence for the run time is:

$$T(n) = T(\frac{n}{5}) + T(\frac{3}{4}n) + \Theta(n)$$

As usual, we replace the $\Theta(n)$ with the assumption that this term is $\leq cn$ for large enough n. Thus

$$T(n) \leq T(\frac{n}{5}) + T(\frac{3}{4}n) + cn$$

We can solve this recurrence using substitution. Let's guess that the runtime is $O(n)$. Then the goal is to show that $T(n) \leq dn$.

- **Assume** We assume the 'goal' is true for the smaller values of n:

$$T(\frac{n}{5}) \leq d \cdot \frac{n}{5}$$

$$T(\frac{3}{4}n) \leq d \cdot \frac{3}{4}n$$

- **Substitution step:** . Substitute this into the recurrence:

$$\begin{aligned} T(n) &= T(\frac{n}{5}) + T(\frac{3}{4}n) + cn \\ &\leq d \cdot \frac{n}{5} + d \cdot \frac{3}{4}n + cn \\ &= \frac{19}{20}dn + cn \\ &= dn - (\frac{1}{20}dn - cn) \\ &\leq dn \end{aligned}$$

  where the last line is true as long as $\frac{1}{20}dn \geq cn$.

Therefore we have shown by induction that $T(n) \leq dn$ for n large enough, and therefore the **Select** algorithm runs in time $O(n)$.