Deterministic Select in O of n

1 Interlude: Deterministic O(n) Select?

We can select the *kth* largest element from a list of n elements in $O(n)$ average-case (or with random pivot selection, expected-case) time using the QuickSelect algorithm.

However, the **worst-case** time for QuickSelect is still $O(n^2)$. Practically speaking, with good pivot selection or (if you're in danger of being attacked) solid randomized pivot selection, QuickSelect is awesome and the last word on this problem.

But.. is it *possible* to find the *kth* largest element from a list of n elements in $O(n)$ time in the **worst** case?

1.1 A Concrete Example

Let's get a concrete example to work with.

```
In [1]: import random

        n = 50

        # Generate a random permutation of [1, ..., n]
        permutation = [i+1 for i in range(n)]
        random.shuffle(permutation)

        # Now we should display this.
        # It's too busy to print one element per line. I'll print 5 per line.
        align_spec = str(len(str(n)))
        for i in range(len(permutation)):
            format_str = "{:>" + align_spec + "} "  # meaning right-aligned in enough space fo
            print(format_str.format(permutation[i]), end='')
            if i % 5 == 4:
                print()
```

```
22 24 30 21  7
26 25 31 32 28
 9 49 34 43 42
19 27 50 23 36
 5 33  1 16 18
14 10 38 12 13
45 11 17 40 15
```

```
 3  4 35 47  2
 8 29 48  6 37
20 41 46 44 39
```

1.2 How Much "Room" We Have to Spare

We want our function to run in $O(n)$ time. If we look at a recursive case of a recurrence like this (where the base case takes constant time):

$$T(n) = T(c \cdot n) + n$$

$T(n)$ will be in $O(n)$ if $c < 1$. Why? We get that "stick" shaped tree we had in QuickSelect. Since $c < 1$, the sum of the work at each level will converge: $\sum_{i=0}^{\infty} c^i n = n\frac{1}{1-c}$. Whatever specific value $\frac{1}{1-c}$ has, it's certainly a constant.

That means all we have to do is cut off *some* constant fraction of the work at each level. We don't need to divide in half. Can we do that? What if we just cut off the upper-left corner of the numbers above?

```
In [2]: # Print the upper-left corner in color:
        import math

        purple = "\x1b[35m"
        black = "\x1b[30m"

        def purple_pretty_print(array):
            for i in range(len(array)):
                pformat_str = purple + "{:>" + align_spec + "} "  # in purple, right-aligned i
                bformat_str = black + "{:>" + align_spec + "} "  # right-aligned in enough spa

                if (i // 5 < (n // 5) // 2 and  # top half of the rows
                    i % 5 <= 2):                # left half of its row
                    print(pformat_str.format(array[i]), end='')
                else:
                    print(bformat_str.format(array[i]), end='')
                if i % 5 == 4:
                    print()

        purple_pretty_print(permutation)
```

```
22 24 30 21  726 25 31 32 28 9 49 34 43 4219 27 50 23 36 5 33  1 16 1814 10 38 12 1345 11 17 4
```

1.3 Diving In!

Well, we probably can't cut out THOSE particular elements, but maybe we could cut out at least that much "mass" from the array? How close to the median would we need to get to manage that? It's our familiar 1/4 (roughly speaking).

Let's try to find the 1/4 we want to cut.

```
In [3]: # If we sort each line, we can make the smaller elements purple.
        result = []
        for i in range(len(permutation) // 5):
            # sort the elements i*5 .. i*5+4
            result.extend(sorted(permutation[i*5:(i+1)*5]))

        purple_pretty_print(result)

        # But..
```

```
 7 21 22 24 3025 26 28 31 32 9 34 42 43 4919 23 27 36 50 1  5 16 18 3310 12 13 14 3811 15 17 4
```

```
In [4]: # Those aren't necessarily the elements we're actually looking for.
        # How about if we then sort our lines of 5 but THEIR middle elements?

        blocks_of_5 = []
        for i in range(len(result) // 5):
            blocks_of_5.append(result[i*5:(i+1)*5])
        sorted_blocks = sorted(blocks_of_5, key=lambda block: block[2]) # sort the blocks by t

        result = []
        for block in sorted_blocks:
            result.extend(block)
        purple_pretty_print(result)

        # Now is the upper left the smallest 1/4 that we're looking for?
```

```
 2  3  4 35 4710 12 13 14 38 1  5 16 18 3311 15 17 40 45 7 21 22 24 3019 23 27 36 5025 26 28 3
```

1.4 What We Have Done So Far

Well, we sorted each line of the array (constant time per line, since each line contains a constant number of elements, and so linear time overall) and then sorted the lines by their middle elements. But, all that rearranging was really just for us to see.

All we really *needed* was to divide the array into blocks of 5, find the median of each block (constant time per block and linear overall), and then find the median of all these "median-of-5" elements. We really cannot afford to sort these rows by their middle elements like we did, but we don't have to. Hold on to how much it costs to find this median of $\frac{1}{5}$ of the elements.

Now, we pick *that* median of medians as our pivot. It's guaranteed to have a split no worse than 3/4-1/4. How much did we "spend" to get here? We "spent" $\frac{3}{4} + \frac{1}{5}$ of the "mass" available for our recursive calls. ($\frac{1}{5}$ to find the median of medians and $\frac{3}{4}$ in the worst-case to recurse after we've used the pivot to partition the array.) We want that to be some fixed fraction less than 1, and we did it. Our total is $\frac{19}{20}$.

At this point, we have a pivot guaranteed to be "good enough" in order to proceed with the rest of the QuickSelect algorithm.

1.4.1 Why 5?

Why blocks of 5, by the way?

Well, we probably want an odd number, since the median is then really in the middle. (Consider using blocks of 4. If we get unlucky in how we pick the median element, we can end up with just as much in the portion we still have to consider as with blocks of 5. So, even numbers aren't useful to us here.)

Obviously, blocks of 1 won't work. We'll end up asking for the median of the whole array, which is tantamount to the job we're already doing!

Blocks of 3 turn out not to work either. To see why, try computing how much of the "mass" available for recursive calls we end up spending!

So, blocks of 5 are the first size that works. We can use blocks of 7, 9, 11, etc. as well, as long as the block size is a constant. (Otherwise, we need to figure out how to find the median of that non-constant block size!)

```
In [5]: pivot = result[len(result) // 5 // 2 * 5 + 2]

        Lesser = [x for x in result if x < pivot]
        Greater = [x for x in result if x > pivot]

        Lesser, pivot, Greater
```

```
Out[5]: ([2,
          3,
          4,
          10,
          12,
          13,
          14,
          1,
          5,
          16,
          18,
          11,
          15,
          17,
          7,
          21,
          22,
          24,
          19,
          23,
          25,
          26,
          6,
          8,
          20,
          9],
         27,
         [35,
          47,
38,
33,
40,
45,
30,
36,
50,
28,
31,
32,
29,
37,
48,
39,
41,
44,
46,
34,
42,
43,
49])
```

1.5 What's Left?

Let's make this into an actual, usable function. We're not going to try *at all* to make the constant factors small, just to get a correct, linear-time algorithm for selection: DeterministicSelect.

```
In [6]: def deterministic_select(A, i):
            """
            Given a list of numbers A and a number 1 <= i <= len(A), return the i'th largest e
            """
            # Base Case: When we have fewer than five elements, just find the i'th largest dir
            if len(A) < 5:
                sorted_A = sorted(A)
                return sorted_A[len(A) - i]

            # Note: Python's documentation stipulates that a slice like a[i:j] where j > len(a
            # That's handy for us here.

            # Divide into blocks of five and sort the blocks in preparation for finding their 
            # Does sorting here take n lg n time? NO. We sort a bunch of blocks EACH OF LENGTH
            # constant time to sort 5 numbers! So, overall, this operation takes linear time.
            blocks = []
            for b in range((len(A)-1) // 5 + 1):  # That is, the ceiling of len(A)/5
                # sort the elements b*5 .. b*5+4
                blocks.append(sorted(A[b*5:(b+1)*5]))

            # Find the median of the medians
```

```
        medians = [block[len(block)//2] for block in blocks]  # len(block)//2 rather than 
        median_of_medians = deterministic_select(medians, len(medians) // 2 + 1)  # produc

        # Partition out the smaller/larger elements.
        lesser = [v for v in A if v < median_of_medians]
        greater = [v for v in A if v > median_of_medians]
        moms = [v for v in A if v == median_of_medians]  # yes, this could be done oh-so-m

        # Depending on the sizes of lesser, moms (median of medians),
        # greater, figure out whether we are:
        #  1) Done (when the ith largest element is in moms)
        #  2) Recursing to the left (from our perspective, into greater)
        #  3) Recursing to the right (from our perspective, into lesser)
        if len(greater) < i <= len(greater) + len(moms):
            return median_of_medians
        elif len(greater) >= i:
            # Recurse to the "left" (larger elements)
            return deterministic_select(greater, i)
        else:
            # Recurse to the "right" (smaller elements)
            # We've cut out exactly len(greater) + len(moms) larger elements from consider
            # So, we now want the (i - (len(greater) + len(moms)))'th largest element.
            return deterministic_select(lesser, i - len(greater) - len(moms))
```

In [7]:
```
# Test thoroughly on a small array

import random
size = 22
permutation = [(i+1) for i in range(size)]
random.shuffle(permutation)
[deterministic_select(permutation, v+1) for v in range(size)]
```

Out[7]:
```
[22, 21, 20, 19, 18, 17, 16, 15, 14, 13, 12, 11, 10, 9, 8, 7, 6, 5, 4, 3, 2, 1]
```

In [8]:
```
import time
# Take a flyer on a big array :)


n = 1000000

# Generate a random permutation of [1, ..., n]
permutation = [i+1 for i in range(n)]
random.shuffle(permutation)

start_time = time.time()
result = deterministic_select(permutation, n // 2 + 1)
print("Deterministic Select takes: %s seconds" % (time.time() - start_time))
result
```

```
Deterministic Select takes: 2.89159893989563 seconds
```

```
Out[8]: 500000
```

```
In [9]: import heapq

        # Of course, we didn't worry about constant factors, and DSelect has some awful consta
        # the start anyway. When we compare against an O(n lg n) solution with generally excel
        # clear!

        def heap_select(A, i):
            """
            Given a list of numbers A and a number 1 <= i <= len(A), return the i'th largest e
            """
            h = [-x for x in A]  # negative b/c we want a max-heap
            heapq.heapify(h)
            while i > 1:
                heapq.heappop(h)
                i -= 1
            return -heapq.heappop(h)
```

```
In [10]: size = 22
         permutation = [(i+1) for i in range(size)]
         random.shuffle(permutation)
         [heap_select(permutation, v+1) for v in range(size)]
```

```
Out[10]: [22, 21, 20, 19, 18, 17, 16, 15, 14, 13, 12, 11, 10, 9, 8, 7, 6, 5, 4, 3, 2, 1]
```

```
In [11]: import time

         n = 1000000

         # Generate a random permutation of [1, ..., n]
         permutation = [i+1 for i in range(n)]
         random.shuffle(permutation)

         start_time = time.time()
         result = heap_select(permutation, n // 2 + 1)
         print("Heap Select takes: %s seconds" % (time.time() - start_time))
         result
```

```
Heap Select takes: 0.7666587829589844 seconds
```

```
Out[11]: 500000
```