array_demos_class_version

July 19, 2019

[1]:

```python
# Up here: some helper functions for the demo

import random
import math

def print_blocks_of_5(array):
    # given an array, print to the screen with 5 elements per line
    n = len(array)
    align_spec = str(len(str(n)))
    for i in range(len(array)):
        format_str = "{:>" + align_spec + "} " # meaning right-aligned in␣
↪enough space for print(format_str.format(permutation[i]), end='')
        print(format_str.format(array[i]), end='')
        if i % 5 == 4:
            print()

def print_sorted_blocks_of_5(array):
    # given an array, print to the screen with 5 elements per line, where each␣
↪line of 5 is sorted
    blocks_of_5 = []
    for i in range(len(array) // 5):
        blocks_of_5.extend(sorted(array[i*5:(i+1)*5]))
    print_blocks_of_5(blocks_of_5)

def print_blocks_sorted_by_median(array):
    # given an array, print to the screen with 5 elements per line, where each␣
↪line of 5 is sorted
    # AND the lines are ordered by their middle element (median)
    # returns the median-of-medians
    blocks_of_5 = []
    for i in range(len(array) // 5):
        blocks_of_5.append(sorted(array[i*5:(i+1)*5]))
        sorted_blocks = sorted(blocks_of_5, key=lambda block: block[2]) # sort␣
↪the blocks by the middle element of each block
    result = []
    for block in sorted_blocks:
```

```
        result.extend(block)
    print_blocks_of_5(result)
    medianBlock = sorted_blocks[math.ceil(len(sorted_blocks)/2)-1]
    return medianBlock[math.ceil(len(medianBlock)/2)-1]
```

We're going to try to find a pivot for QuickSelect that:

- Can be selected in no more than linear time
- Is guaranteed to have *some fraction* of the array in both Lesser and Greater

Below: let's try printing out a random array of size 55, broken into blocks of 5:

[2]:
```
n = 55

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

print_blocks_of_5(permutation)
```

```
50 39 51 40 43
37 17 29  1 19
 9  2 25 16 26
24 44 28  7 23
18 30 55 22 33
53  5 42 21 45
 6 47  4 34 54
11  8 13 46  3
20 15 41 12 10
31 52 14 27 35
36 32 49 38 48
```

Now, let's sort each of the blocks of 5:

[3]:
```
print_sorted_blocks_of_5(permutation)
```

```
39 40 43 50 51
 1 17 19 29 37
 2  9 16 25 26
 7 23 24 28 44
18 22 30 33 55
 5 21 42 45 53
 4  6 34 47 54
 3  8 11 13 46
10 12 15 20 41
14 27 31 35 52
32 36 38 48 49
```

0.1 Clicker Question!

I will edit this Markdown cell during lecture to ask a clicker question about what we just did…

Did I spend more than linear time to sort each of these blocks of 5?

A. Yes B. No

…

…

…

Now, let's try *arranging the sorted blocks by their median values*. (FYI, this takes more than linear time and we wouldn't do this in an algorithm; we're just doing it here for demonstrating purposes.)

[4]:
```python
MoM = print_blocks_sorted_by_median(permutation)

print()
print("Median of medians is", MoM)
```

```
 3  8 11 13 46
10 12 15 20 41
 2  9 16 25 26
 1 17 19 29 37
 7 23 24 28 44
18 22 30 33 55
14 27 31 35 52
 4  6 34 47 54
32 36 38 48 49
 5 21 42 45 53
39 40 43 50 51

Median of medians is 30
```

Look at the *median* of the median values printed above. Is there some portion of the array that *must* be smaller than the median-of-medians? Is there some portion that *must* be larger?

0.2 Another Clicker Question!

If we choose the *median of medians* as our pivot for QuickSelect, what is the *worst-case* (i.e., largest possible) value for the size of the array in our recursive call?

A. 3n/4 B. n/2 C. n-1 D. n/4

[ ]:
```

```