In [2]:

```
# setup
from IPython.core.display import display,HTML
display(HTML('<style>.prompt{width: 0px; min-width: 0px; visibility: collapse}</style>'))
display(HTML(open('rise.css').read()))

# imports
import numpy as np
import matplotlib.pyplot as plt
import seaborn as sns
%matplotlib inline
sns.set(style="whitegrid", font_scale=1.5, rc={'figure.figsize':(12, 6)})
```

CMPS 2200

Introduction to Algorithms

Selection

Today's agenda:

- Randomized Linear Work Selection

Given an unsorted list a and an integer k ($0 \leq k < |a|$), the **order statistics** (or **selection**) problem asks us to return the kth smallest element from a. We also refer to the kth smallest element as the element of *rank* k.

Example: Let $a = \langle 2, 5, 4, 1, 3, -1, 99 \rangle$. For $k = 0$, the "0th smallest" element is the minimum element in a, or -1. For $k = n - 1$, it is the maximum, or 99. For $k = 3$, we return 3.

Before we come up with a randomized algorithm, we can make a couple of simple observations.

First, any algorithm for this problem requires $\Omega(n)$ work. Why?

Second, we can reduce this problem to sorting: we sort a and return the kth element of this sorted list. This requires $O(n \log n)$ time.

Can we do any better? Sorting seems like overkill since we don't really need to rearrange all the elements, or even return a list.

A useful observation is that the kth smallest element in a *partitions* a into a set of $k - 1$ smaller elements, and a set of $n - k - 1$ larger elements.

Example: Suppose $a = \langle 2, 5, 4, 1, 3, -1, 99 \rangle$ and $k = 3$. So 3 is larger than $\langle 2, 1, -1 \rangle$ and smaller than $\langle 5, 4, 99 \rangle$.

Notice that *for any* element x in the list, we can look at each element in the list to compute the rank of x. This can be done in $O(n)$ work and $O(\log n)$ span.

Example: Suppose $a = \langle 2, 5, 4, 1, 3, -1, 99 \rangle$ and $k = 3$. So $a[0] = 2$ is larger than 2 elements ($\langle 1, -1 \rangle$) and smaller than 5 elements ($\langle 5, 4, 3, 99 \rangle$).

We can see from this example that once we've identified 3 smaller elements ($\langle 1, -1\rangle$ and 2), the element of rank $k = 3$ in a must be in $\langle 5, 4, 3, 99\rangle$. Moreover it's rank is $k - 3 = 0$ in this list.

This is a little like binary search, but with the partition step helping establish some order.

This observation motivates the following recursive algorithm.

$$
\begin{array}{l}
simple\_select\ a\ k = \\
\textbf{let} \\
\quad p = a[0] \\
\quad \ell = \langle\, x \in a \mid x < p \,\rangle \\
\quad r = \langle\, x \in a \mid x > p \,\rangle \\
\textbf{in} \\
\quad \textbf{if}\ (k < |\ell|)\ \textbf{then}\ simple\_select\ \ell\ k \\
\quad \textbf{else if}\ (k < |a| - |r|)\ \textbf{then}\ p \\
\quad \textbf{else}\ simple\_select\ r\ (k - (|a| - |r|)) \\
\textbf{end}
\end{array}
$$

We just have one recursive call so no parallelism there. However, we can use filter to partition in parallel. This has $O(|a|)$ work $O(\log |a|)$ span.

What is the total work over all recursions? We know that the work in each recursive call is the $\max\{W(|\ l\ |), W(|\ r\ |)\} + O(n)$.

Consider the case where a is a sorted list. Then in every call, $\ell = \emptyset$, and $|r\ | = n - 1$. So we would we have $W(n) = W(n-1) + n = O(n^2)$. This is worse than just sorting the list!

Moreover the span is $S(n) = S(n-1) + \lg n \in O(n \lg n)$ - far worse than sorting!

In [3]:
```
def simple_select(a, k):
    p = a[0]
    print('\na=', a, 'k=', k, 'p=', p)
    l = list(filter(lambda x: x < p, a))  # O(|a|) work, O(log|a|) span
    r = list(filter(lambda x: x > p, a))  # O(|a|) work, O(log|a|) span
    print('l=', l, 'r=', r)
    if k < len(l):
        print('...recursing with l=%s and k=%d' % (str(l), k))
        return simple_select(l, k)
    elif k < len(a) - len(r):
        print('...returning p=%d' % p)
        return p
    else:
        print('...recursing with r=%s and k=%d' % (str(r), k - (len(a) -  len(r))))
        return simple_select(r, k - (len(a) -  len(r)))

# -1, 1, 2, 3, 4, 5, 99
# k=3 -> 3
# k=0 -> -1
# k=6 -> 99
simple_select([2,5,4,1,3,-1,99], 3)
```

```
a= [2, 5, 4, 1, 3, -1, 99] k= 3 p= 2
l= [1, -1] r= [5, 4, 3, 99]
...recursing with r=[5, 4, 3, 99] and k=0

a= [5, 4, 3, 99] k= 0 p= 5
l= [4, 3] r= [99]
```

```
...recursing with l=[4, 3] and k=0

a= [4, 3] k= 0 p= 4
l= [3] r= []
...recursing with l=[3] and k=0

a= [3] k= 0 p= 3
l= [] r= []
...returning p=3
3
```

Out[3]:

In [4]:

```
# worst case: find the max of a sorted list
simple_select([-1,1,2,3,4,5,99], 6)
```

```
a= [-1, 1, 2, 3, 4, 5, 99] k= 6 p= -1
l= [] r= [1, 2, 3, 4, 5, 99]
...recursing with r=[1, 2, 3, 4, 5, 99] and k=5

a= [1, 2, 3, 4, 5, 99] k= 5 p= 1
l= [] r= [2, 3, 4, 5, 99]
...recursing with r=[2, 3, 4, 5, 99] and k=4

a= [2, 3, 4, 5, 99] k= 4 p= 2
l= [] r= [3, 4, 5, 99]
...recursing with r=[3, 4, 5, 99] and k=3

a= [3, 4, 5, 99] k= 3 p= 3
l= [] r= [4, 5, 99]
...recursing with r=[4, 5, 99] and k=2

a= [4, 5, 99] k= 2 p= 4
l= [] r= [5, 99]
...recursing with r=[5, 99] and k=1

a= [5, 99] k= 1 p= 5
l= [] r= [99]
...recursing with r=[99] and k=0

a= [99] k= 0 p= 99
l= [] r= []
...returning p=99
99
```

Out[4]:

The problem is that we just don't know anything about the element we're using for the partition. How do we avoid this worst case?

Pick a random element, or **pivot**, for partitioning!

$$
\begin{array}{l}
\mathit{select}\ a\ k = \\
\quad \textbf{let} \\
\qquad p = \text{pick a uniformly random element from } a \\
\qquad \ell = \langle\, x \in a \mid x < p \,\rangle \\
\qquad r = \langle\, x \in a \mid x > p \,\rangle \\
\quad \textbf{in} \\
\qquad \textbf{if } (k < |\ell|) \textbf{ then } \mathit{select}\ \ell\ k \\
\qquad \textbf{else if } (k < |a| - |r|) \textbf{ then } p \\
\qquad \textbf{else } \mathit{select}\ r\ (k - (|a| - |r|)) \\
\quad \textbf{end}
\end{array}
$$

If we get a sorted list as input, what is the probability of the worst-case?

The size of the l and r will depend on the random choice. Thus the recurrences describing the work and span depend on each random choice and we need to find their expected asymptotic work/span.

In [5]:
```
import random
# random.seed(42)  # for repeatability

def select(a, k):
    p = random.choice(a)
    print('\na=', a, 'k=', k, 'p=', p)
    l = list(filter(lambda x: x < p, a))  # O(|a|) work, O(log|a|) span
    r = list(filter(lambda x: x > p, a))  # O(|a|) work, O(log|a|) span
    print('l=', l, 'r=', r)
    if k < len(l):
        print('...recursing with l=%s and k=%d' % (str(l), k))
        return select(l, k)
    elif k < len(a) - len(r):
        print('...returning p=%d' % p)
        return p
    else:
        print('...recursing with r=%s and k=%d' % (str(r), k - (len(a) -  len(r))))
        return select(r, k - (len(a) -  len(r)))

select([2,5,4,1,3,-1,99], 3)
```

```
a= [2, 5, 4, 1, 3, -1, 99] k= 3 p= -1
l= [] r= [2, 5, 4, 1, 3, 99]
...recursing with r=[2, 5, 4, 1, 3, 99] and k=2

a= [2, 5, 4, 1, 3, 99] k= 2 p= 5
l= [2, 4, 1, 3] r= [99]
...recursing with l=[2, 4, 1, 3] and k=2

a= [2, 4, 1, 3] k= 2 p= 3
l= [2, 1] r= [4]
...returning p=3
3
```

Out[5]:

Let's get some intution for what's happening. We saw that the work of our algorithm depends on $\max\{W(|\ l\ |), W(|\ r\ |)\}$ in each recursive call. While there is only a $1/n$ probability of choosing a balanced split, any constant fraction reduction in the size of the larger list yields good performance.

So, suppose we knew that $\max\{W(|\ l\ |), W(|\ r\ |)\} \leq W(3n/4)$. This would be a good enough split since the overall work would be $W(n) = W(3n/4) + n = O(n)$. (root dominated)

What we want to know is: **What's the probability that the input to the recursive call has size $\leq \frac{3}{4}n$?**

We can examine where p might land in the sorted version of a, to understand the probability of a good split.

If the sampled pivot lies in the green region, then the size of the array passed to the recursive call is at most $3n/4$.

The probability of sampling a point in the green region is $1/2$.

We can see that $\mathbf{P}[\max\{W(|\ l\ |), W(|\ r\ |)\} \leq W(3n/4)] = 1/2$.

If we think of each choice of pivot as a coin flip ("good" vs. "bad") then the expected number of pivot choices to reduce the input to $3n/4$ is 2.

In other words, every two recursions yields the desired reduction in list size, and so in expectation we will do linear work.

What if we're unlucky?

We could keep sampling pivots outside of the green area. What is the probability if we do so i times in a row?

$\frac{1}{2} * \frac{1}{2} * \frac{1}{2} * \ldots = \frac{1}{2^i}$

E.g., for $i = 10$, probability of getting no good pivots is $\approx 0.1\%$.

Thus, probability of getting at least one good pivot for 10 splits is $\approx 99.9\%$

Let's analyze the performance more closely. Let $X(n)$ be the fractional size of the larger side of the split, for an input of size n. So

$$X(n) = \frac{\max\{|l|, |r|\}}{n}$$

e.g., $n = 6$

i	len(l)	len(r)	X(i)
0	0	5	5/6
1	1	4	4/6
2	2	3	3/6
3	3	2	3/6
4	4	1	4/6
5	5	0	5/6

Then the work and span recurrences are:

$$W(n) \leq W(X(n) \cdot n) + O(n)$$
$$S(n) \leq S(X(n) \cdot n) + O(\lg n)$$

First, we'll bound $\mathbf{E}[X(n)]$. As we discussed above, if $|l| = i$, then $|r| = n - i - 1$. Using the fact that the probability of the pivot being any particular i is $1/n$, we have:

$$
\begin{aligned}
\mathbf{E}\left[X(n)\right] &= \sum_{i=0}^{n-1} P[X(i)] \cdot X(i) \\
&= \frac{1}{n} \sum_{i=0}^{n-1} \frac{\max\{i, n-i-1\}}{n} \\
&\leq \frac{1}{n} \sum_{j=n/2}^{n-1} \frac{2}{n} \cdot j \\
&\leq \frac{2}{n^2} \sum_{j=n/2}^{n-1} j \\
&\leq \frac{3}{4}
\end{aligned}
$$

Last line uses $\sum_{i=x}^{y} i = \frac{1}{2}(x+y)(y-x+1)$:

$= \frac{1}{2}(n/2 + (n-1))((n-1) - (n/2) + 1)$

$= \frac{1}{2}(3n/2 - 1)(n/2)$

$= \frac{1}{2}(3n^2/4 - n/2)$

$= (3n^2/8 - n/4)$

$= (3n^2 - 2n)/8$

multiply by $2/n^2$:

$= \frac{3n^2-2n}{4n^2} = \frac{3n^2}{4n^2} - \frac{1}{2n} \leq \frac{3}{4}$

It might seem tempting to say that we are done. However, we could get "unlucky" in a series of recursions even though $\mathbf{E}[X(n)] \leq 3/4$. We will show the following.

Theorem. At the d^{th} level of recursion, the size of the input is $(3/4)^d n$ in expectation.

Proof: We can prove this by induction.

The base case $d = 0$ holds trivially.

For the inductive step, we make the inductive hypothesis that our theorem holds for $d \geq 0$ and will show that it holds after the $d + 1^{st}$ recursive call.

For the d^{th} recursive call, let Y_d be a random variable for the instance size and let Z_d be the rank of the pivot. For any value of y and z, let $f(y, z)$ be the fraction of the input reduced by the choice of the pivot at position z for an input of size y. The expected input size in the $(d+1)^{st}$ call is:

$$
\begin{aligned}
\mathbf{E}[Y_{d+1}] &= \sum_{y,z} y \cdot f(y,z)\mathbf{P}_{Y_d,Z_d}(y,z) \\
&= \sum_{y}\sum_{z} yf(y,z)\mathbf{P}_{Y_d}(y)\mathbf{P}_{Z_d|Y_d}(z \mid y) && \text{by definition } p(a,b) = p(b)p(a|b) \\
&= \sum_{y} y\mathbf{P}_{Y_d}(y)\sum_{z} f(y,z)\mathbf{P}_{Z_d|Y_d}(z \mid y) && \text{grouping terms} \\
&= \sum_{y} y\mathbf{P}_{Y_d}(y)\mathbf{E}\left[X(y)\right] && \text{definition of } X(i) \text{ and expectation} \\
&\leq \frac{3}{4}\sum_{y} y\mathbf{P}_{Y_d}(y) && \text{by our bound above} \\
&\leq \frac{3}{4}\mathbf{E}\left[Y_d\right]. && \text{definition of expectation}
\end{aligned}
$$

Thus $\mathbf{E}[Y_{d+1}] \leq \frac{3}{4}\mathbf{E}\left[Y_d\right]$

This proves the theorem since we can repeatedly apply the bound.

Finally, we need to compute the **expected** work.

(Think of using the tree method, but using the **expected** size of n at each level.)

$$
\begin{aligned}
\mathbf{E}[W(n)] &\leq \sum_{i=0}^{n} \mathbf{E}[Y_i] && \text{since there is linear work at each iteration} \\
&\leq \sum_{i=0}^{n} \left(\frac{3}{4}\right)^i n && \text{by theorem above} \\
&\leq n\sum_{i=0}^{n} \left(\frac{3}{4}\right)^i \\
&\leq 4n && \text{by } \sum_{i=0}^{\infty} \alpha^i < \frac{1}{1-\alpha} \text{ for } \alpha < 1 \\
&\in O(n)
\end{aligned}
$$

For the span we can also use the theorem to show that at each level the span is $O(\log n)$. By showing that the number of levels is $O(\log n)$ with high probability, we can establish that the span is $O(\log^2 n)$ with high probability.

In [ ]: