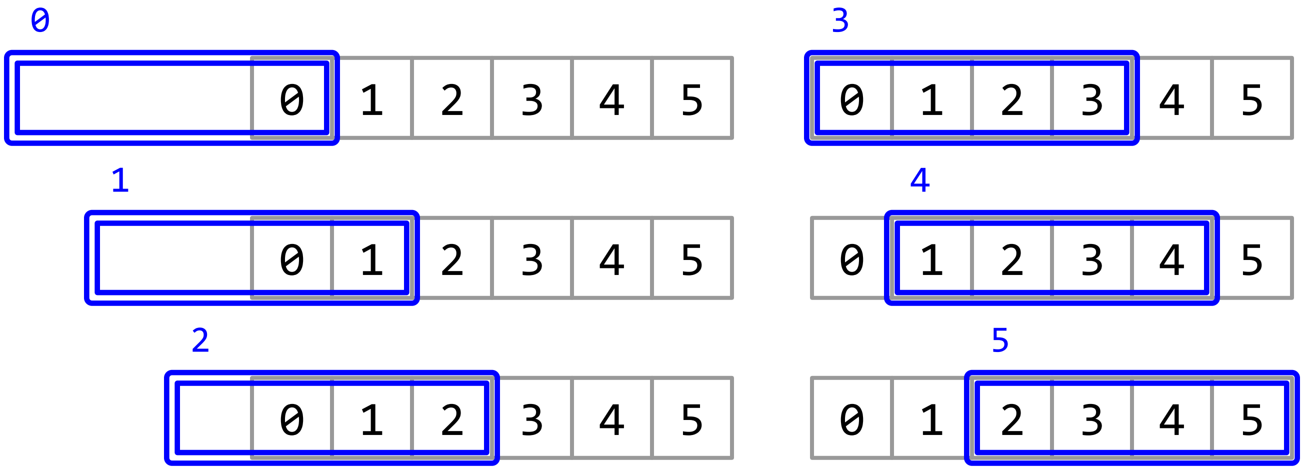
The Secret Life of Rolling Pandas

jaime.frio@gmail.com

So what exactly is this rolling thingy?

- An object that can **efficiently** compute aggregations on rolling windows.
- Available as [`Series.rolling()`]() and [`DataFrame.rolling()`]().
- Close cousin of [`Series.expanding()`]() and [`DataFrame.expanding()`]().



Let’s see it at work!

But I already know about NumPy’s stride tricks!

```
>>> import numpy as np
>>> from numpy.lib.stride_tricks import as_strided

>>> a = np.arange(6)
>>> win_a = as_strided(a, shape=(len(a) - 4 + 1,),
                       strides=a.strides * 2)
>>> win_a
array([[0, 1, 2, 3],
       [1, 2, 3, 4],
       [2, 3, 4, 5]])
>>> win_a.sum(axis=-1)
array([ 6, 10, 14])
```

Big O

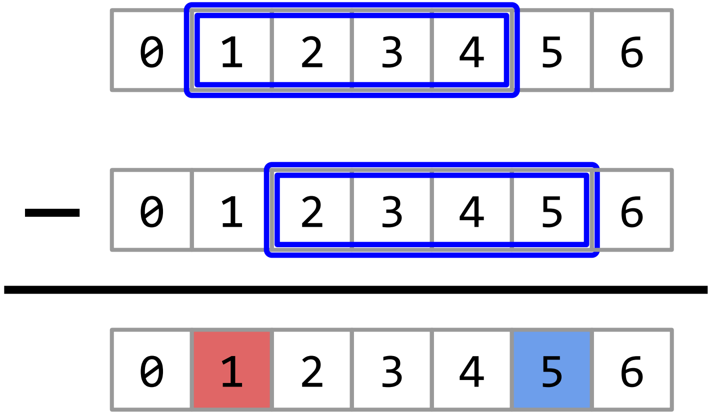
So how much work will our clever numpy magic have to do?

- On a n item array with an m item window, $(n - m + 1) \times (m - 1)$ additions.
- Or as a computer scientist would put it, $O(n\ m)$
- That can be a lot if m is not really small!
- Which poses the infamous question:

Can we do better?

Use this one weird trick!

- Indeed we can do better!
- The happy idea here is reuse, since two consecutive windows are *almost* the same.
- So we only need to remove **one item** and add **another one** to our calculation.
- How hard can that be, huh?



Sums are easy

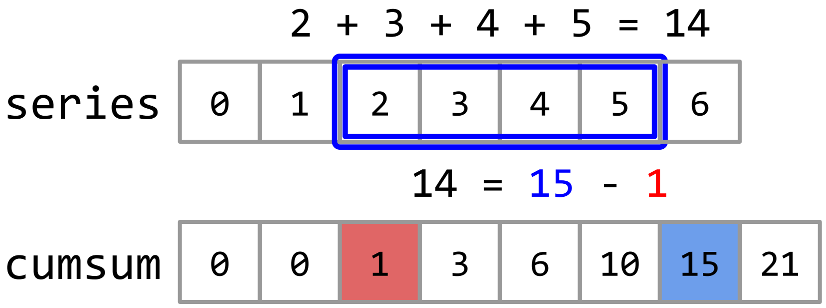
- The idea is very easy to implement for a rolling sum:
- At every step, we have to subtract one item, and add another:

```
rolling_sum[i] = rolling_sum[i - 1] + array[i] - array[i - win]
```

- How much work do we have to do now?
- $2n - m - 1$ additions, or $O(n)$.
- It doesn't get much better than that.

Actually it does get better...

- If we do O(*n*) work to compute, and use O(*n*) memory to store, the values of the accumulated sum of the series...



- ...we can now compute the sum on any window by subtracting two values.
- It really does not get any better than this.

More on sums

- You can use the same trick in higher dimensions with the help of the inclusion-exclusion principle.
- Also known as **summed area tables** or **integral images**.
- Fundamental part of the Viola-Jones face detection algorithm.

1	3	6	4	6
9	3	5	5	3
3	0	1	1	7
4	5	2	8	9

15 = 41 + 1 - 14 - 13

0	0	0	0	0	0
0	1	4	10	14	20
0	10	16	27	36	45
0	13	19	31	41	57
0	17	28	42	60	85

Doing rolling variance the wrong way

- You may remember this formula from your statistics textbook:

$$\mathrm{Var}(X) = \frac{1}{n}\sum_{i=1}^{n}\left(x_i - \frac{1}{n}\sum_{i=1}^{n} x_i\right)^2 = \frac{1}{n}\sum_{i=1}^{n} x_i^2 - \frac{1}{n^2}\left(\sum_{i=1}^{n} x_i\right)^2$$

- So we only need rolling sums on x and x^2, right?

```
>>> n = 100
>>> x = 1e9 + np.random.rand(n)
>>> (x**2).sum() / n - x.sum()**2 / n**2
-640.0
```

- WTF!? Wasn't the variance supposed to *always* be positive?

Welford to the rescue!

- We can update a running calculation of the mean (M_n) and sum of square differences from the mean (S_n), minus the numerical instability, with these formulas:

$$M_n = M_{n-1} + \frac{x_n - M_{n-1}}{n}$$
$$S_n = S_{n-1} + (x_n - M_{n-1})(x_n - M_n)$$

- You can turn those around to remove, rather than add, a value. Interestingly, this last formula was removed from the Wikipedia article on [Algorithms for calculating variance]() as "original research."
- A variant of this algorithm can be used to parallelize variance calculations!

Rolling minimum and maximum

- There is a very clever linear algorithm to compute the rolling minimum.
- The original source seems to be Richard Harter's blog, and has been used (in chronological order) in bottleneck, pandas and scipy.ndimage.
- It uses a deque, or double ended queue, of at most the window size number of items.
- The deque stores increasing candidate minima, wait for next page.
- Because the maximum size of the deque is known, it is stored in a ring buffer.

A visual demonstration

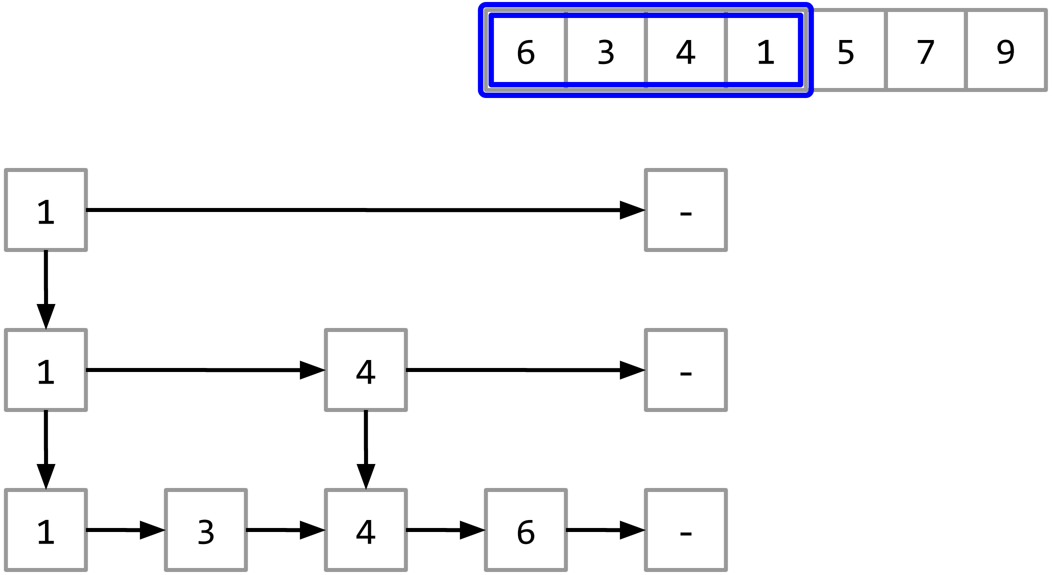
1	3	4	6	5	7	9	6	0	4

1	-	-	-		7	~~3~~	4	5
1	3	-	-		7	9	4	5
1	3	4	-		6	~~9~~	4	5
1	3	4	6		0	~~9~~	4	~~5~~
~~1~~	3	4	5		0	3	4	~~5~~

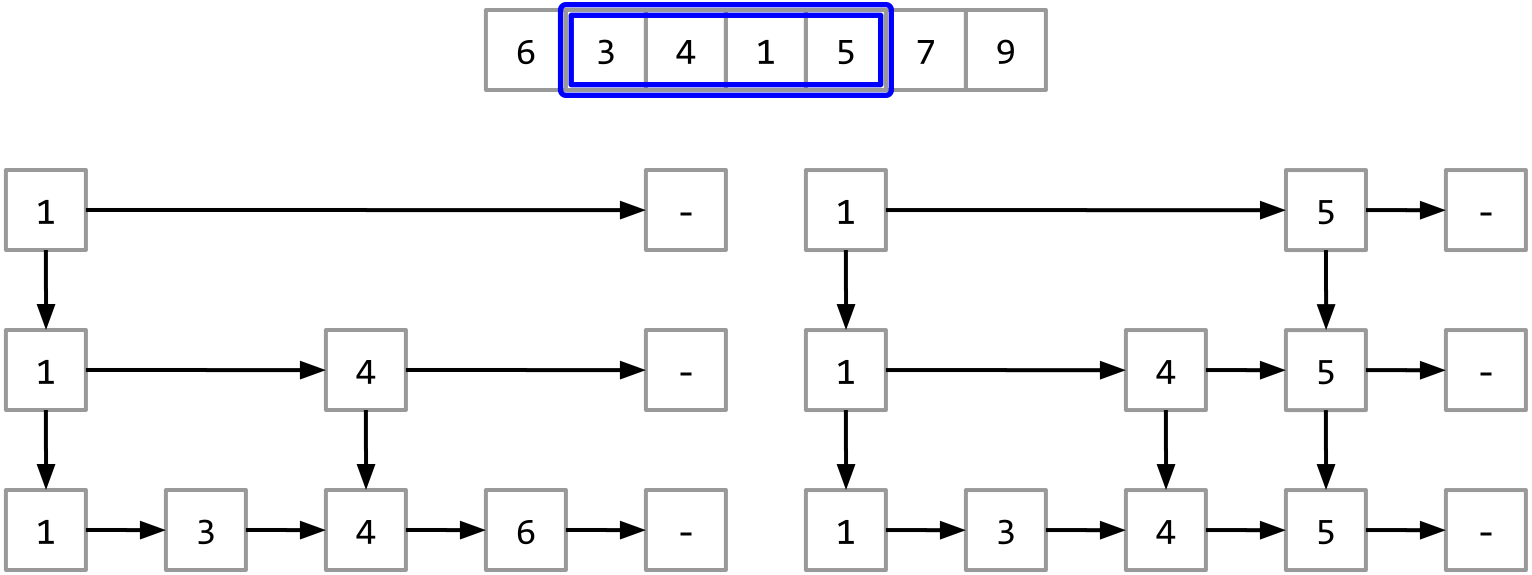
The rolling median

- Sorting based algorithms on a single array are O(*n* log *n*).
- There are clever algorithms to do it in O(*n*).
- Rolling median goes to great lengths, using a (twice) specialized data structure, to get it down to O(*n* log *m*).
- Interesting thing about this approach is that it relies on randomization

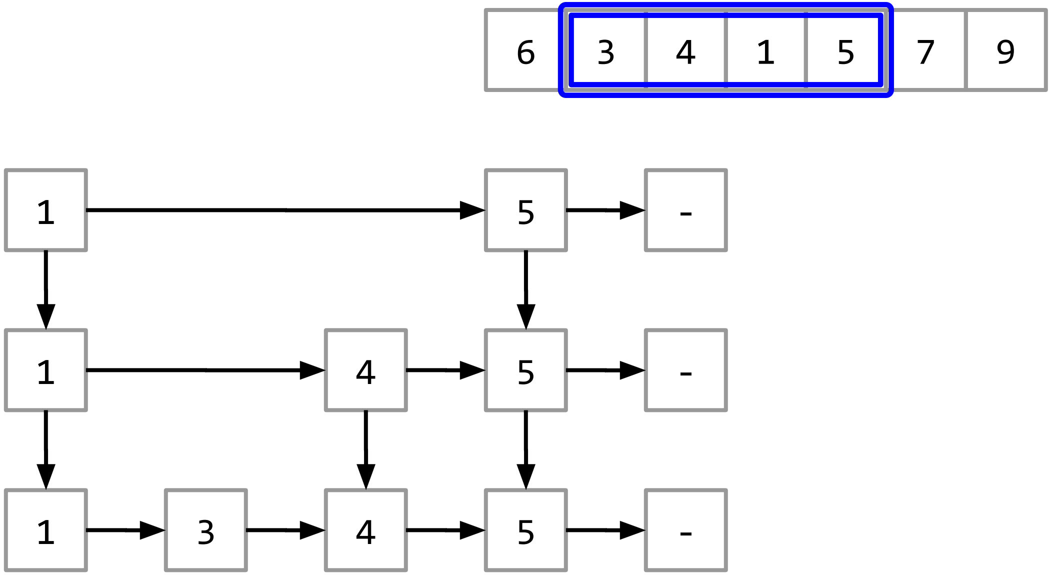
Skip lists in action - i



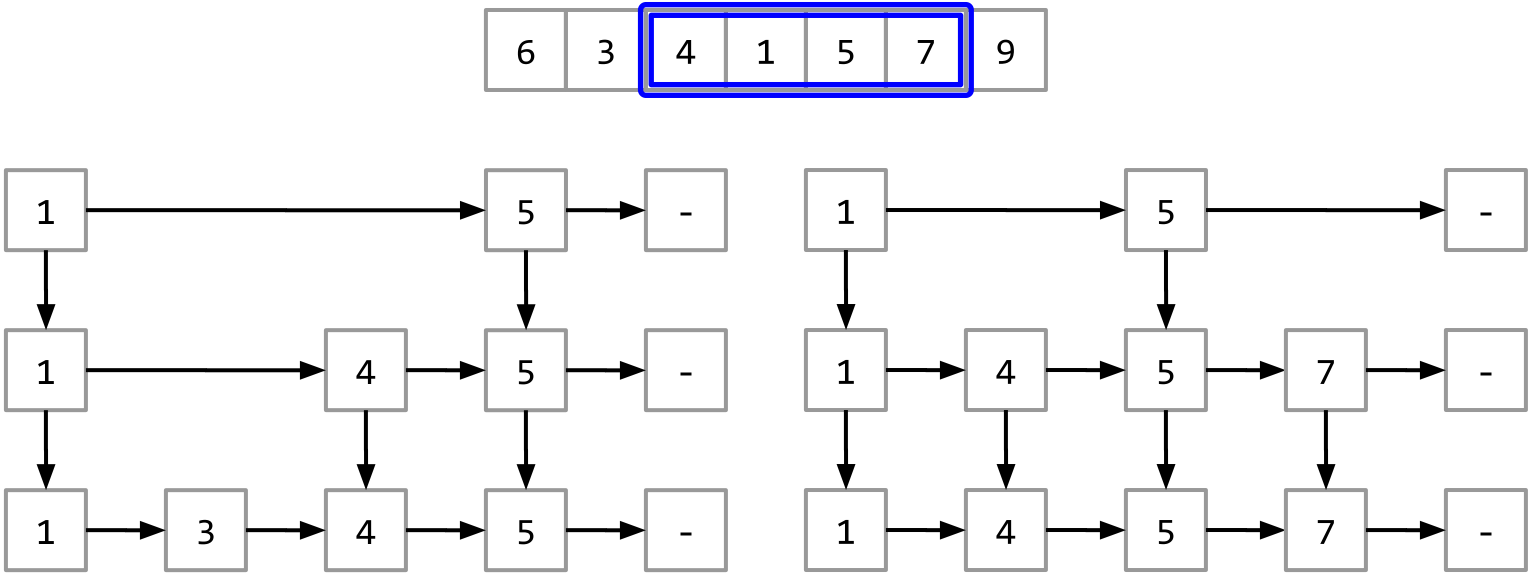
Skip lists in action - ii



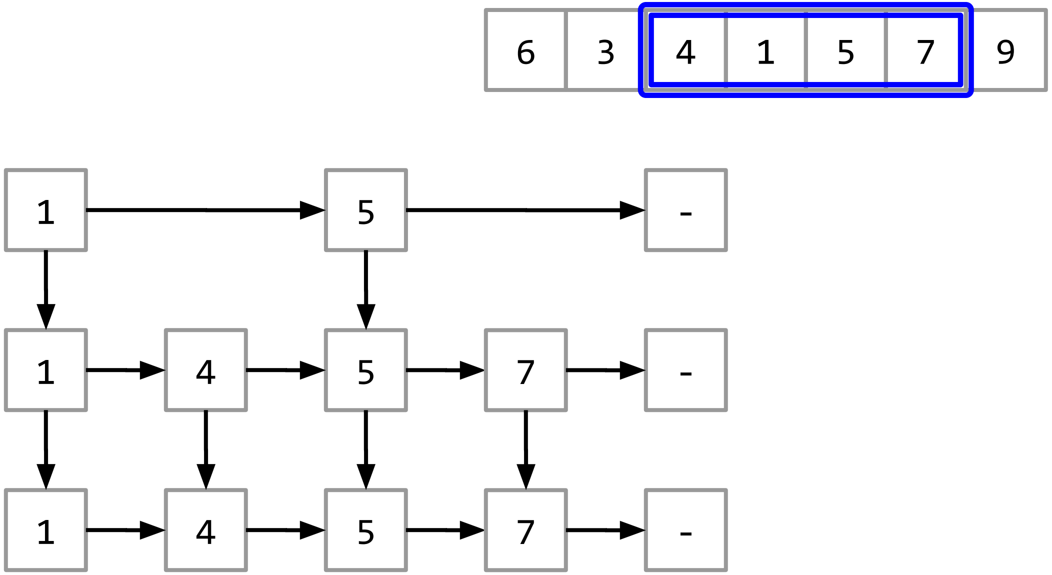
Skip lists in action - iii



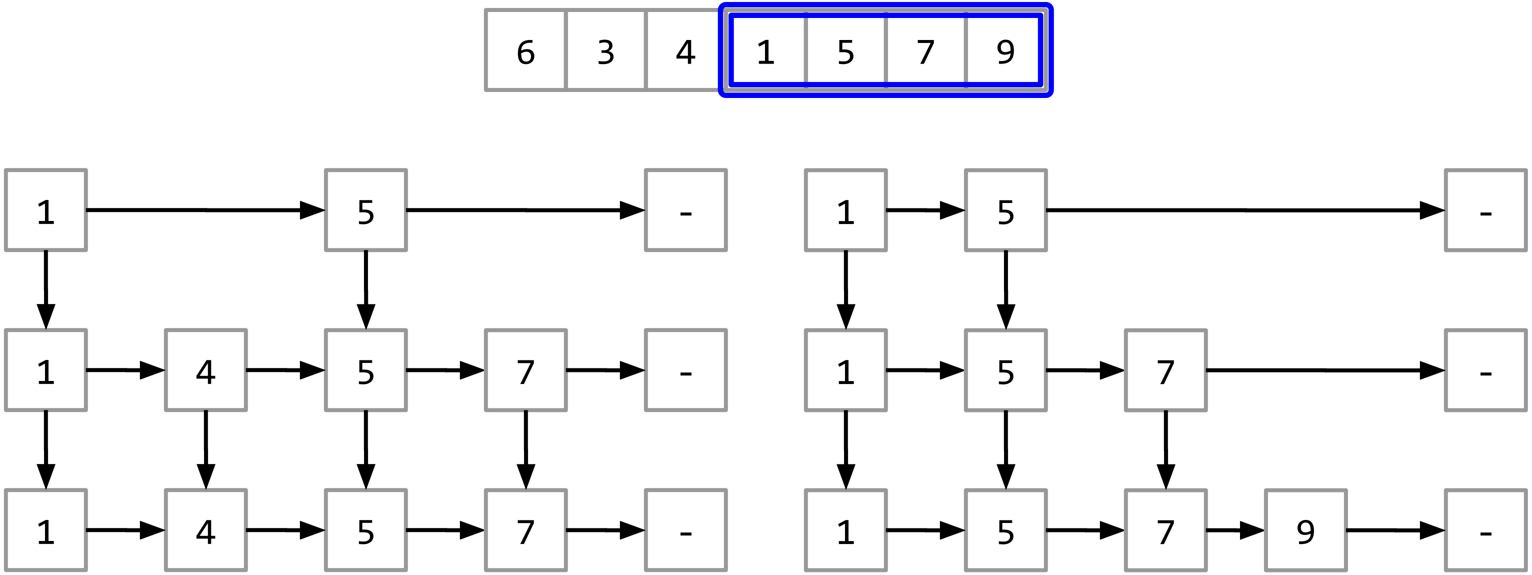
Skip lists in action - iv



Skip lists in action - v



Skip lists in action - vi



Thanks!