Quicksort: Idea

- Recursive like mergesort, except it doesn't just slice the array into two
- The basic idea is to pick a pivot element
  - Any element will do, although we might get better results if we choose more carefully

4	6	1	3	5	9

Start

- We then partition the array into those bigger than the pivot and those smaller than the pivot

$\leq$ $>$

1	4	3	5	6	9

End of iteration

Quicksort: Idea

- Now we recursively apply quicksort to the two partitions

1	4	3	5	6	9

- How do we partition?
  1. Have two pointers, L and R that we initialise to either end of the array, excluding the pivot for now

L R

4	6	1	3	9	5

  2. Increment L until a[L] is bigger than the pivot OR L==R

4	6	1	3	9	5

Bigger than 5

Quicksort: Idea

3. Decrement R until a[R] is less than or equal to the pivot OR R==L

L R

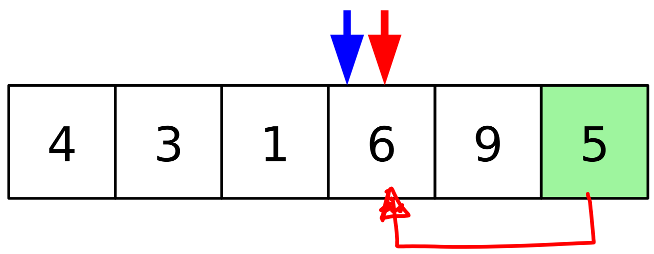
4	6	1	3	9	5

4. If (L!=R) then we can swap a[L] and a[R] in order to make L and R OK

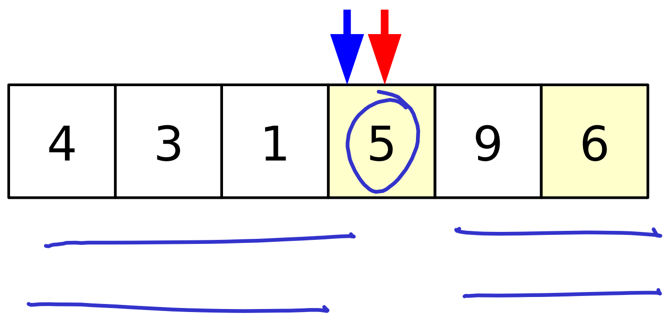
4	3	1	6	9	5

Quicksort: Idea

4. If (L != R) then GOTO 2
   else GOTO 5



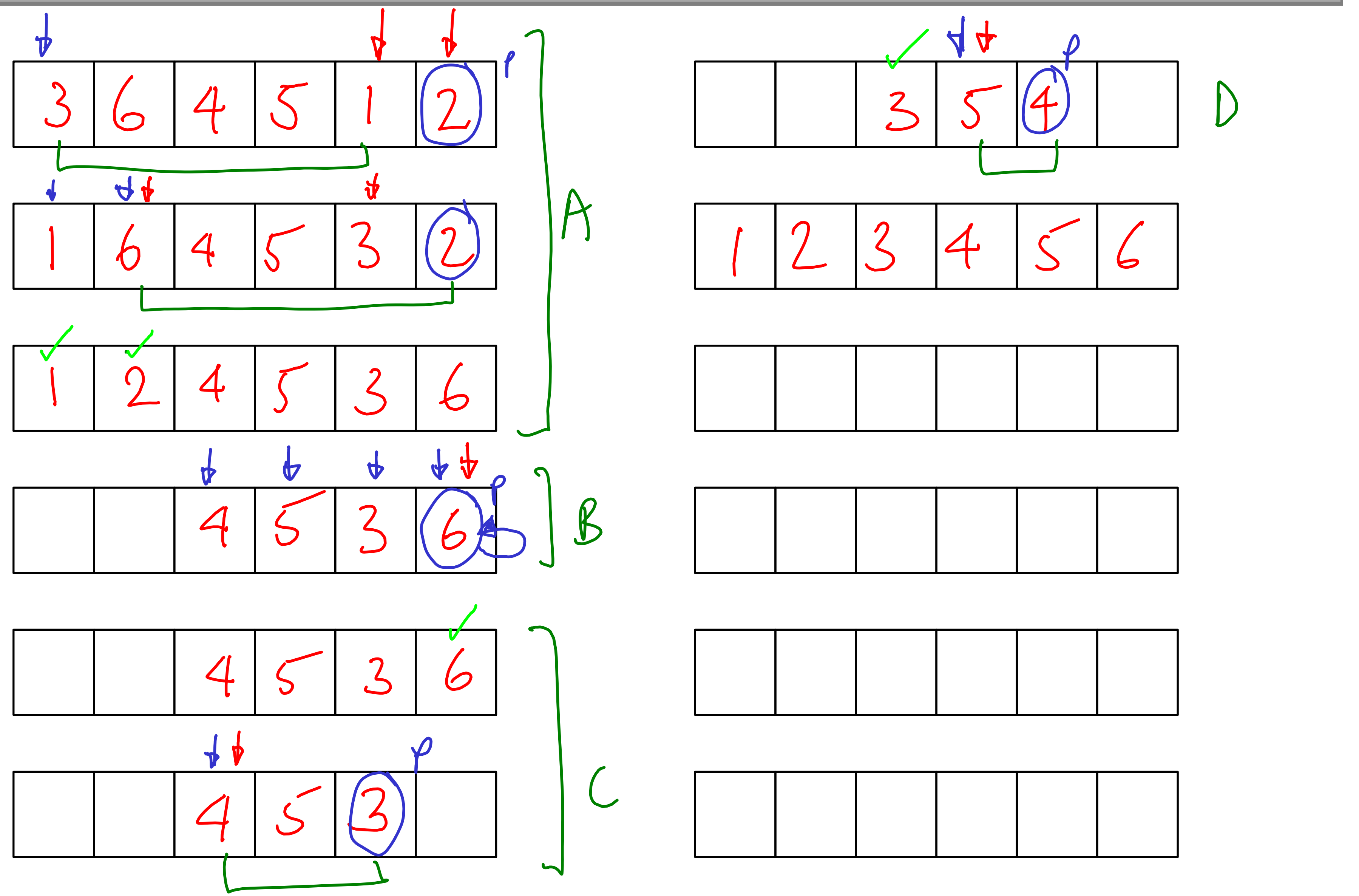
5. Now (L==R) and we swap the pivot with a[L]



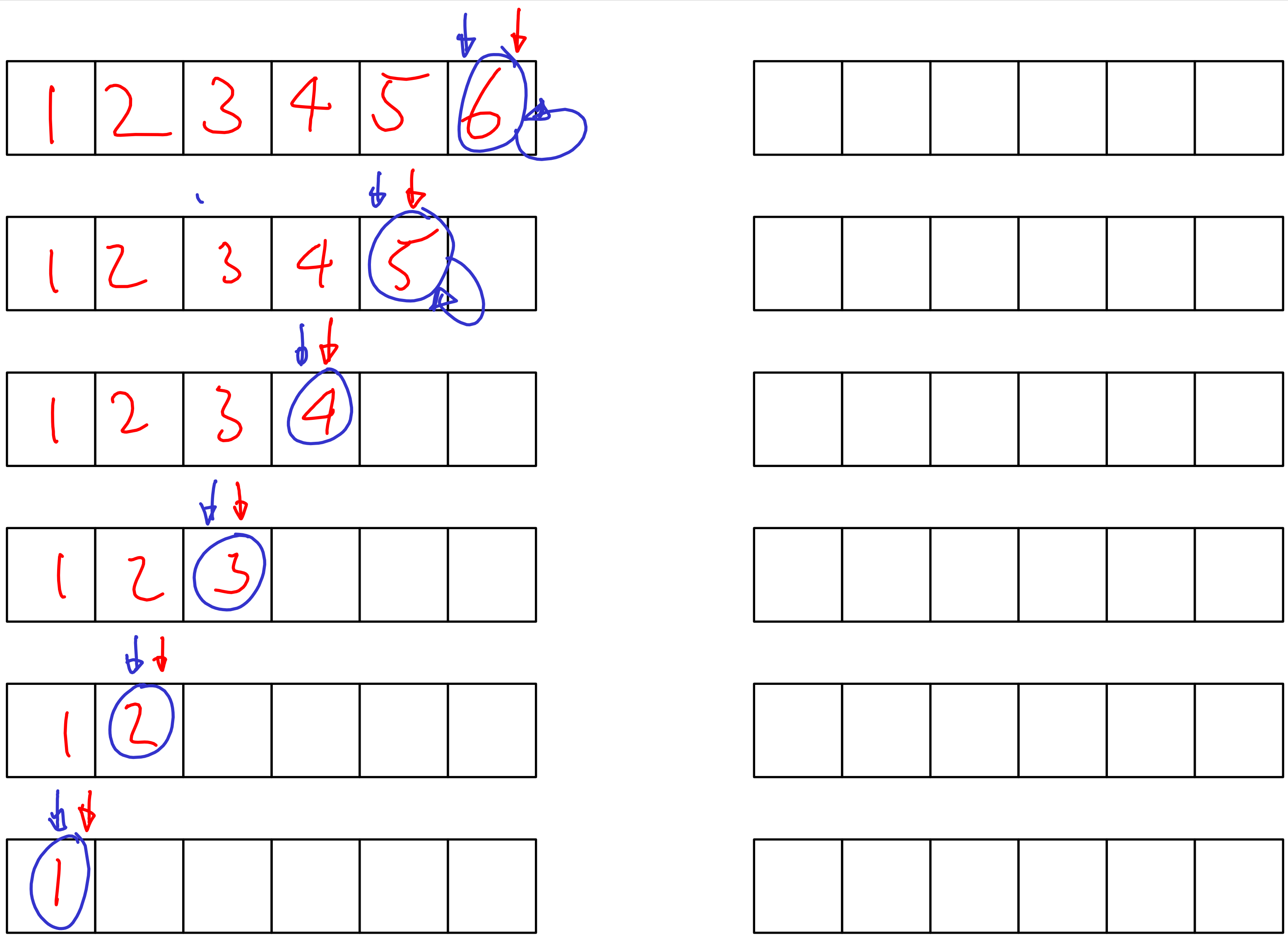
Quicksort: Idea

Beware: Depending on where your pivot is chosen to be, you need to think carefully about what gets swapped where: it's very easy to end up being off by one. You will find lots of subtle variations on the algorithm implementations (all of which work). I recommend you try writing a quicksort that works on any pivot (the one here works just if you choose the last element as the pivot)

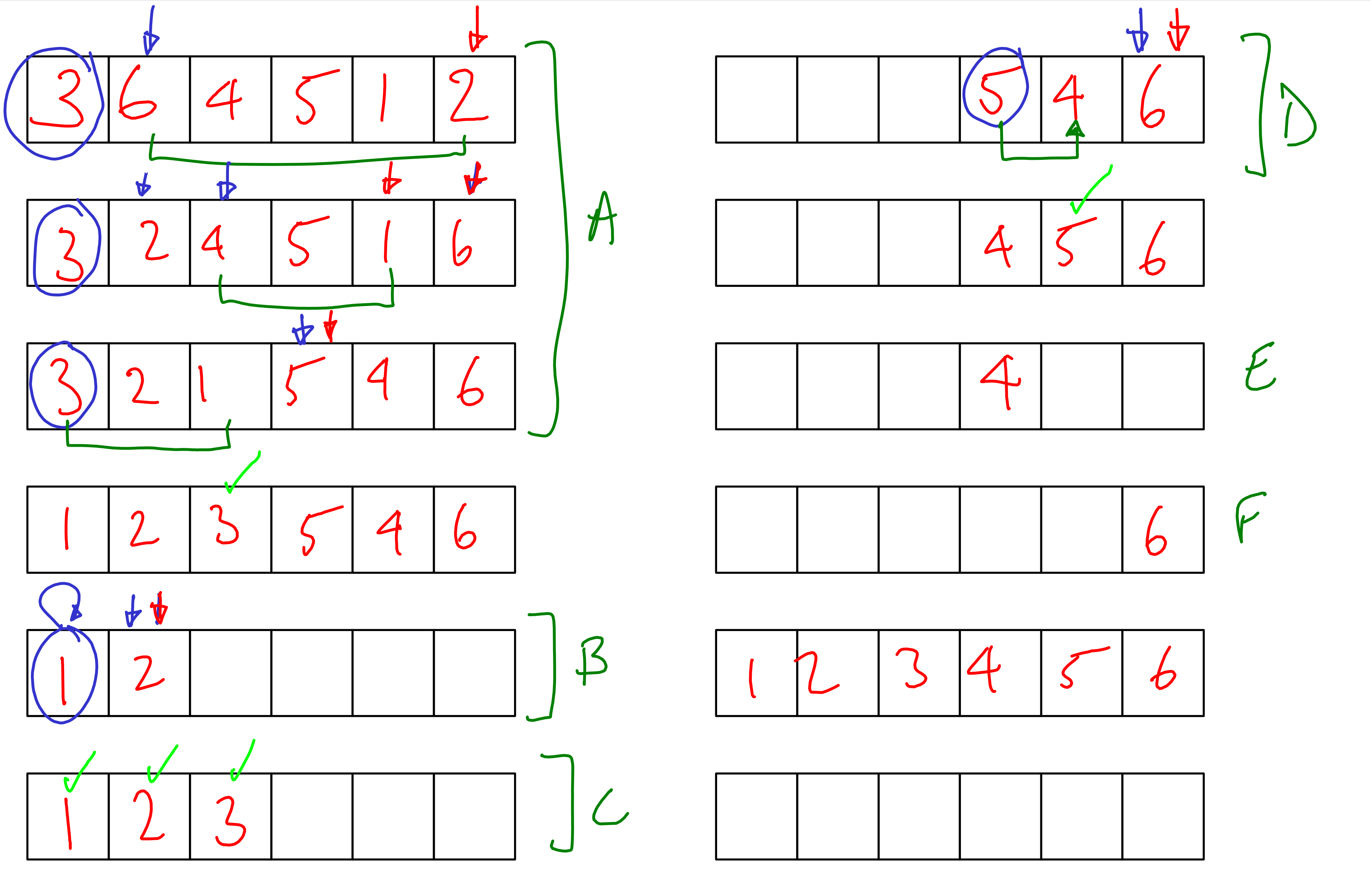
Quicksort: Example (Last element)



Quicksort: Example (sorted array)

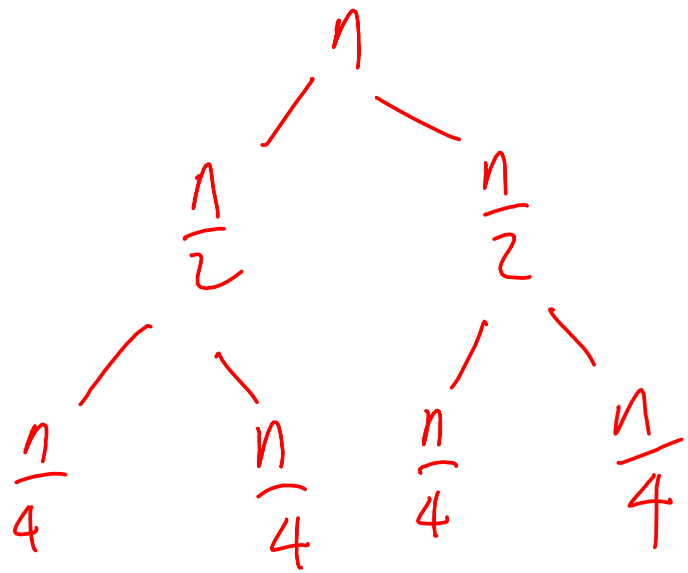


Quicksort: Example (first element)



Quicksort: Analysis (Gulp!)

- Best case
  - Somehow we always choose the pivot that splits the sub-array being sorted into two even chunks
  - Then we have exactly what we had for mergesort
  - $f(n) = 2f(n/2) + kn$
  - That was O(nlogn) :-)



Quicksort: Analysis (Gulp!)

1	2	3	4	5	6

- Worst case
  - Somehow we always choose the biggest (or smallest) element as the pivot every time
  - For a subarray of size k, we will always recurse on a single subarray of size (k-1)
  - f(n) = f(n-1) + kn

$$f(n) = f(n-1) + kn$$
$$= f(n-2) + k(n-1) + kn$$
$$= f(n-m) + k(n-m+1) + k \dots \dots kn$$
$$= f(0) + k + k2 + k3 + \dots kn$$
$$= f(0) + k\sum_{i}^{n} j$$
$$= O(1) + k\frac{n(n+1)}{2}$$

$$\boxed{O(n^2)}$$

Quicksort: Analysis (Gulp!)

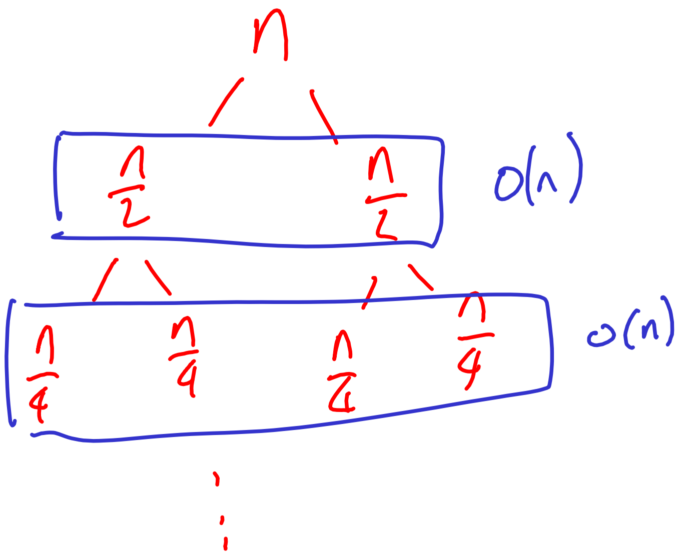
- At best we get O(nlgn) and at worst we get O(n^2) for *performance*
- O(1) for *space* (it is all in-place)
- What about a more general cases??

nlgn ———— n^2

?

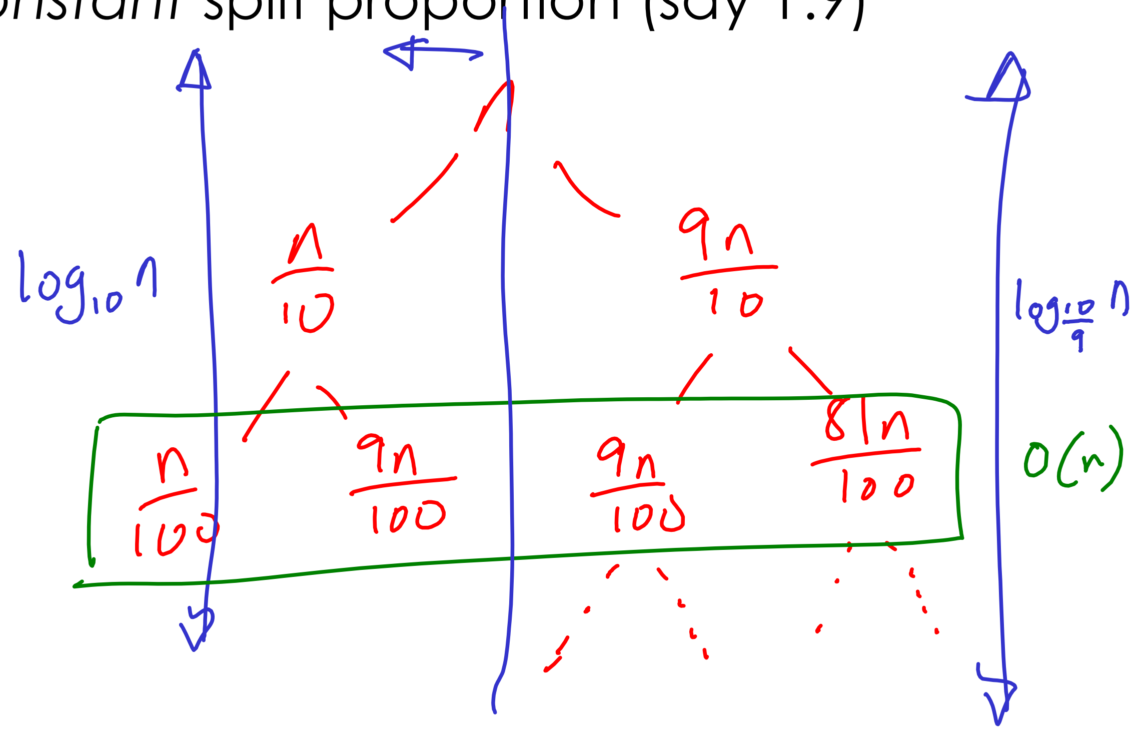
Quicksort: Analysis (Gulp!)

- Recursion tree for *constant* split proportion (say 1:9)



$O(n) \log_2 n$

$O(n \lg n)$



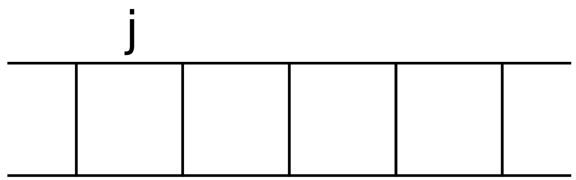
$O(n) \log_{10} n + O(n) \log_{\frac{10}{9}} n$

$= O(n) \lg n + O(n) \lg n$

$= \underline{\underline{O(n \lg n)}}$

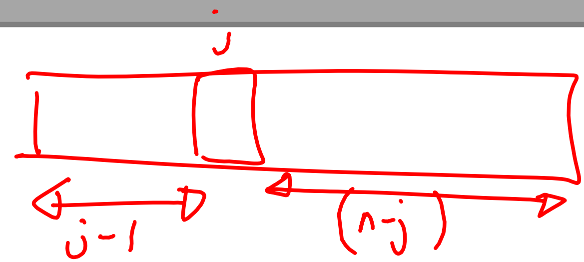
Quicksort: Analysis (Gulp!)

- Average case
- Consider choosing the pivot that is j places along in the *sorted* subarray
  - j=1 would be the same as always taking the first element
  - j=n would be the same as always taking the last element



Quicksort: Analysis (Gulp!)

- Then f(n) = f(n-j) + f(j-1) + kn



- Now let us imagine that we pick a pivot at random each time.
  - The value of j (where the pivot finally ends up) will change, and all values of j will be equally probable
  - So let's take an average

$$\boxed{f(n) = kn + \frac{\sum_{j=1}^{n} f(n-j) + f(j-1)}{n}}$$

Not the easiest thing to solve: see CLRS if you want full detail

Key point: ***O(nlgn)***

Quicksort: Analysis (Gulp!)

- So why is quicksort **the** choice for sorting?
  - It has a poor worst case
  - But the best case is generally *better* than mergesort (smaller constants in O(nlgn)) and the average case is still O(nlgn)
  - *And* quicksort sorts in place Ave O(nlgn)
    Space O(1)

- But you should do what you can to avoid the worst case
  - E.g. randomise your input
  - E.g. randomise your pivot choice

Order Statistics

- Often we don't need a sorted array so much as a partially sorted array
  - E.g. value of the Xth element (think median calculation)
  - E.g. top 30 search results
  - We could sort the entire array and then read off what we want – this would be O(nlgn)
  - Seems like wasted effort...

Median with 'Quickselect'

- The first partitioning leaves us with two subarrays, but we need only recurse on the one that contains the median

3 6 4 2 1 7 5

3 1 4 2 6 7 5

3 1 4 2 5 ~~7 6~~