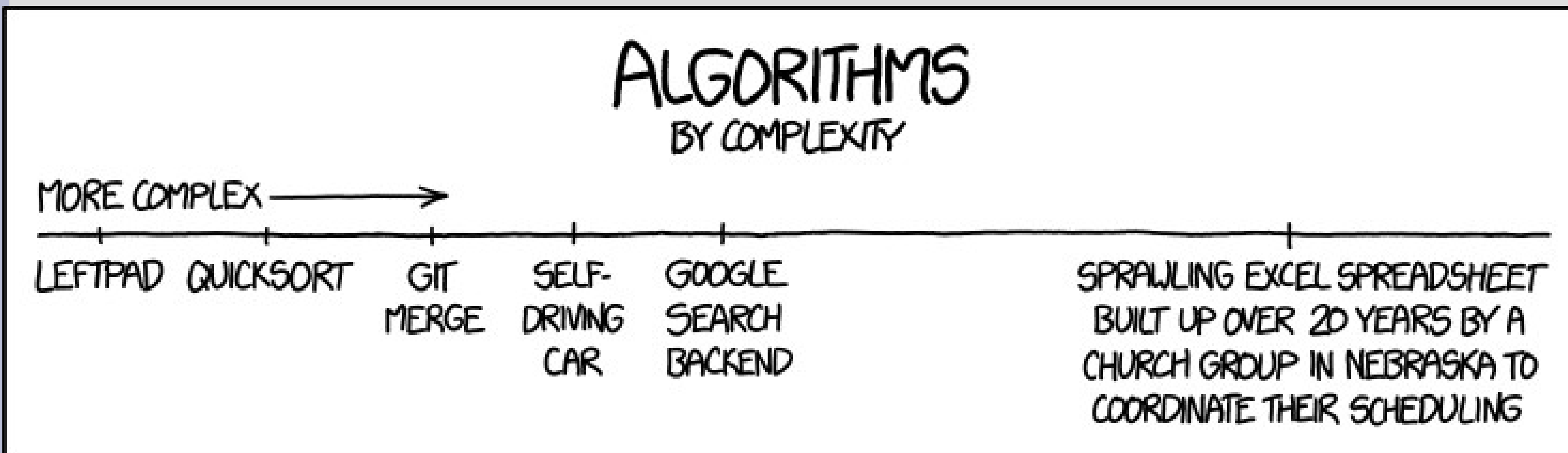
Sorting... more



Divide & conquer

Which works better for multi-cores: insertion sort or merge sort?
Why?

Divide & conquer

Which works better for multi-cores: insertion sort or merge sort? Why?

Merge sort! After the problem is split, each core and individually sort a sub-list and only merging needs to be done synchronized

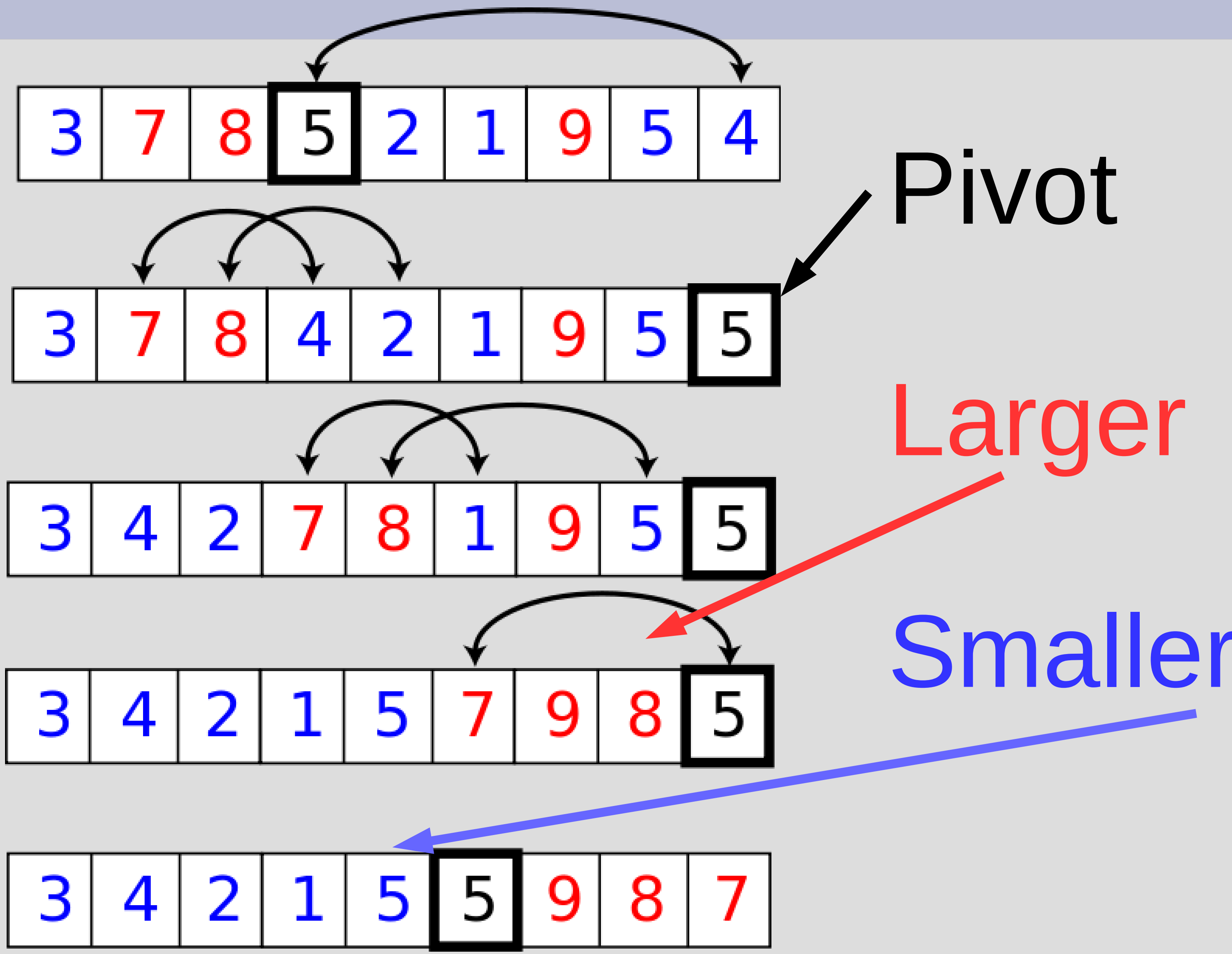
Quicksort

1. Pick a pivot (any element!)

2. Sort the list into 3 parts:
   - Elements smaller than pivot
   - Pivot by itself
   - Elements larger than pivot

3. Recursively sort smaller & larger

Quicksort


3 7 8 5 2 1 9 5 4
Pivot
3 7 8 4 2 1 9 5 5
Larger
3 4 2 7 8 1 9 5 5
Smaller
3 4 2 1 5 7 9 8 5
3 4 2 1 5 5 9 8 7

Quicksort

```
Partition(A, start, end)
x = A[end]
i = start – 1
for j = start to end -1
   if A[j] ≤ x
      i = i + 1
      swap A[i] and A[j]
swap A[i+1] with A[end]
```

Quicksort

Sort: {4, 5, 3, 8, 1, 6, 2}

Quicksort

Sort: {4, 5, 3, 8, 1, 6, 2} – Pivot = 2

{4, 5, 3, 8, 1, 6, 2} – sort 4

{4, 5, 3, 8, 1, 6, 2} – sort 5

{4, 5, 3, 8, 1, 6, 2} – sort 3

{4, 5, 3, 8, 1, 6, 2} – sort 8

{4, 5, 3, 8, 1, 6, 2} – sort 1, swap 4

{1, 5, 3, 8, 4, 6, 2} – sort 6

{1, 5, 3, 8, 4, 6, 2},{1, 2, 5, 3, 8, 4, 6}

Quicksort

For quicksort, you can pick any pivot you want

The algorithm is just easier to write if you pick the last element (or first)

Quicksort

Sort: {4, 5, 3, 8, 1, 6, 2} - Pivot = 3

{4, 5, 2, 8, 1, 6, 3} – swap 2 and 3

{4, 5, 2, 8, 1, 6, 3}

{4, 5, 2, 8, 1, 6, 3}

{2, 5, 4, 8, 1, 6, 3} – swap 2 & 4

{2, 5, 4, 8, 1, 6, 3} (first red ^)

{2, 1, 4, 8, 5, 6, 3} – swap 1 and 5

{2, 1, 4, 8, 5, 6, 3}{2, 1, 3, 8, 5, 6, 4}

Quicksort

Runtime:

Worst case?

Average?

Quicksort

Runtime:

Worst case?

Always pick lowest/highest element, so $O(n^2)$

Average?

Quicksort

Runtime:

Worst case?

Always pick lowest/highest element, so $O(n^2)$

Average?

Sort about half, so same as merge sort on average

Quicksort

Can bound number of checks against pivot:

Let $X_{i,j}$ = event A[i] checked to A[j]

$\text{sum}_{i,j}\ X_{i,j}$ = total number of checks

$E[\text{sum}_{i,j}\ X_{i,j}] = \text{sum}_{i,j}\ E[X_{i,j}]$

$= \text{sum}_{i,j}\ \Pr(A[i] \text{ check } A[j])$

$= \text{sum}_{i,j}\ \Pr(A[i] \text{ or } A[j] \text{ a pivot})$

Quicksort

$= \text{sum}_{i,j}\ \Pr(A[i] \text{ or } A[j] \text{ a pivot})$

$= \text{sum}_{i,j}\ (2 / j-i+1)$ // j-i+1 possibilties

$< \text{sum}_i\ O(\lg n)$

$= O(n \lg n)$

Quicksort

Correctness:

Base: Initially no elements are in the "smaller" or "larger" category

Step (loop): If A[j] < pivot it will be added to "smaller" and "smaller" will claim next spot, otherwise it it stays put and claims a "larger" spot

Termination: Loop on all elements...

Quicksort

Two cases:

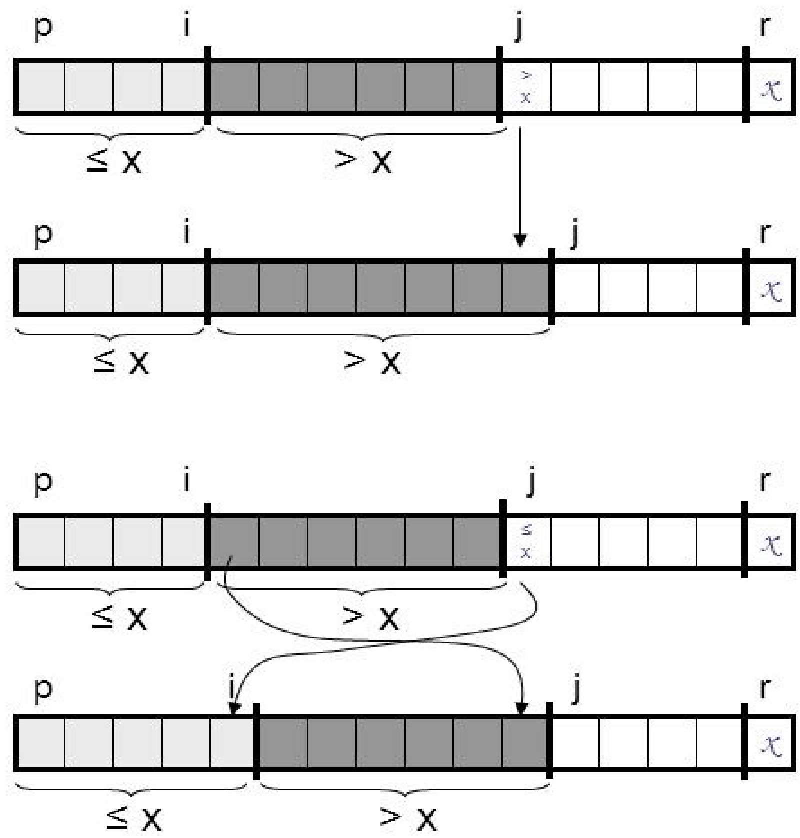
Maintenance of Loop Invariant (4)

If A[j] > pivot:

- only increment j

If A[j] ≤ pivot:

- i is incremented, A[j] and A[i] are swapped and then j is incremented



Quicksort

Which is better for multi core, quicksort or merge sort?

If the average run times are the same, why might you choose quicksort?

Quicksort

Which is better for multi core, quicksort or merge sort?
Neither, quicksort front ends the processing, merge back ends

If the average run times are the same, why might you choose quicksort?

Quicksort

Which is better for multi core, quicksort or merge sort?
Neither, quicksort front ends the processing, merge back ends

If the average run times are the same, why might you choose quicksort?
Uses less space.

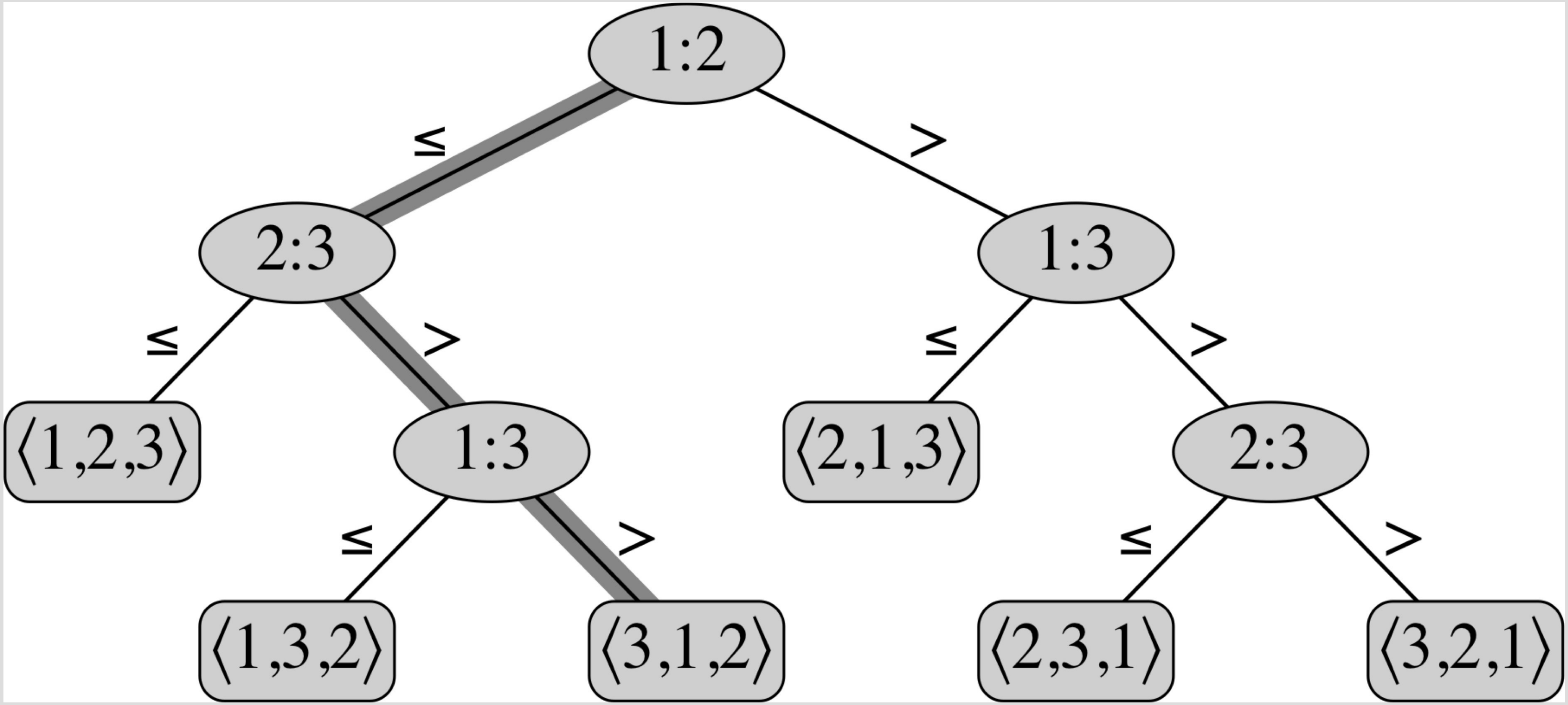
Sorting!

So far we have been looking at comparative sorts (where we only can compute < or >, but have no idea on range of numbers)

The minimum running time for this type of algorithm is $\Theta(n \lg n)$

Comparison sort

All n! permutations must be leaves



Worst case is tree height

Comparison sort

A binary tree (either $<$ or $\geq$) of height h has 2^h leaves:

$2^h \geq n!$

$\lg(2^h) \geq \lg(n!)$ (Stirling's approx)

$h \geq (n \lg n)$

Comparison sort

Today we will make assumptions about the input sequence to get O(n) running time sorts

This is typically accomplished by knowing the range of numbers

Outline

Sorting... again!

- -Comparison sort
- -Count sort
- -Radix sort
- -Bucket sort

Counting sort

1. Store in an array the number of times a number appears
2. Use above to find the last spot available for the number
3. Start from the last element, put it in the last spot (using 2.) decrease last spot array (2.)

Counting sort

A = input, B= output, C = count

```
for j = 1 to A.length
  C[ A[ j ]] = C[ A[ j ]] + 1
for i = 1 to k (range of numbers)
  C[ i ] = C[ i ] + C [ i – 1 ]
for j = A.length to 1
  B[ C[ A[ j ]]] = A[ j ]
  C[ A[ j ]] = C[ A[ j ]] - 1
```

Counting sort

k = 5 (numbers are 2-7)

Sort: {2, 7, 4, 3, 6, 3, 6, 3}

1. Find number of times each number appears

C = {1, 3, 1, 0, 2, 1}

2, 3, 4, 5, 6, 7

Counting sort

Sort: {2, 7, 4, 3, 6, 3, 6, 3}

2. Change C to find last place of each element (first index is 1)

C = {1, 3, 1, 0, 2, 1}

{1, 4, 1, 0, 2, 1}

{1, 4, 5, 0, 2, 1}{1, 4, 5, 5, 7, 1}

{1, 4, 5, 5, 2, 1}{1, 4, 5, 5, 7, 8}

Counting sort

Sort: {2, 7, 4, 3, 6, 3, 6, 3}

3. Go start to last, putting each element into the last spot avail.

C = {1, 4, 5, 5, 7, 8}, last in list = 3

2 3 4 5 6 7

{ , , ,3, , , , }, C =

1 2 3 4 5 6 7 8 {1, 3, 5, 5, 7, 8}

Counting sort

Sort: {2, 7, 4, 3, 6, 3, 6, 3}

3. Go start to last, putting each element into the last spot avail.

C = {1, 4, 5, 5, 7, 8}, last in list = 6

2 3 4 5 6 7

{ , , ,3, , ,6, }, C =

1 2 3 4 5 6 7 8 {1, 3, 5, 5, 6, 8}

Counting sort

Sort: {2, 7, 4, 3, 6, 3, 6, 3}

1 2 3 4 5 6 7 8 2,3,4,5,6,7

{ , , ,3, , ,6, }, C={1,3,5,5,6,8}

{ , ,3,3, , ,6, }, C={1,2,5,5,6,8}

{ , ,3,3, ,6,6, }, C={1,2,5,5,5,8}

{ , 3,3,3, ,6,6, }, C={1,1,5,5,5,8}

{ , 3,3,3,4,6,6, }, C={1,1,4,5,5,8}

{ , 3,3,3,4,6,6,7}, C={1,1,4,5,5,7}

Counting sort

Run time?

Counting sort

Run time?

Loop over C once, A twice

$k + 2n = O(n)$ as k a constant

Counting sort

Sort: {2, 7, 4, 3, 6, 3, 6, 3}

C = {1,3,1,0,2,1} -> {1,4,5,5,7,8}
instead $C[\ i\ ] = \text{sum}_{j<i}\ C[\ j\ ]$

C' = {0, 1, 4, 5, 5, 7}
Add from start of original and increment

Counting sort

Counting sort is stable, which means the last element in the order of repeated numbers is preserved from input to output

(in example, first '3' in original list is first '3' in sorted list)

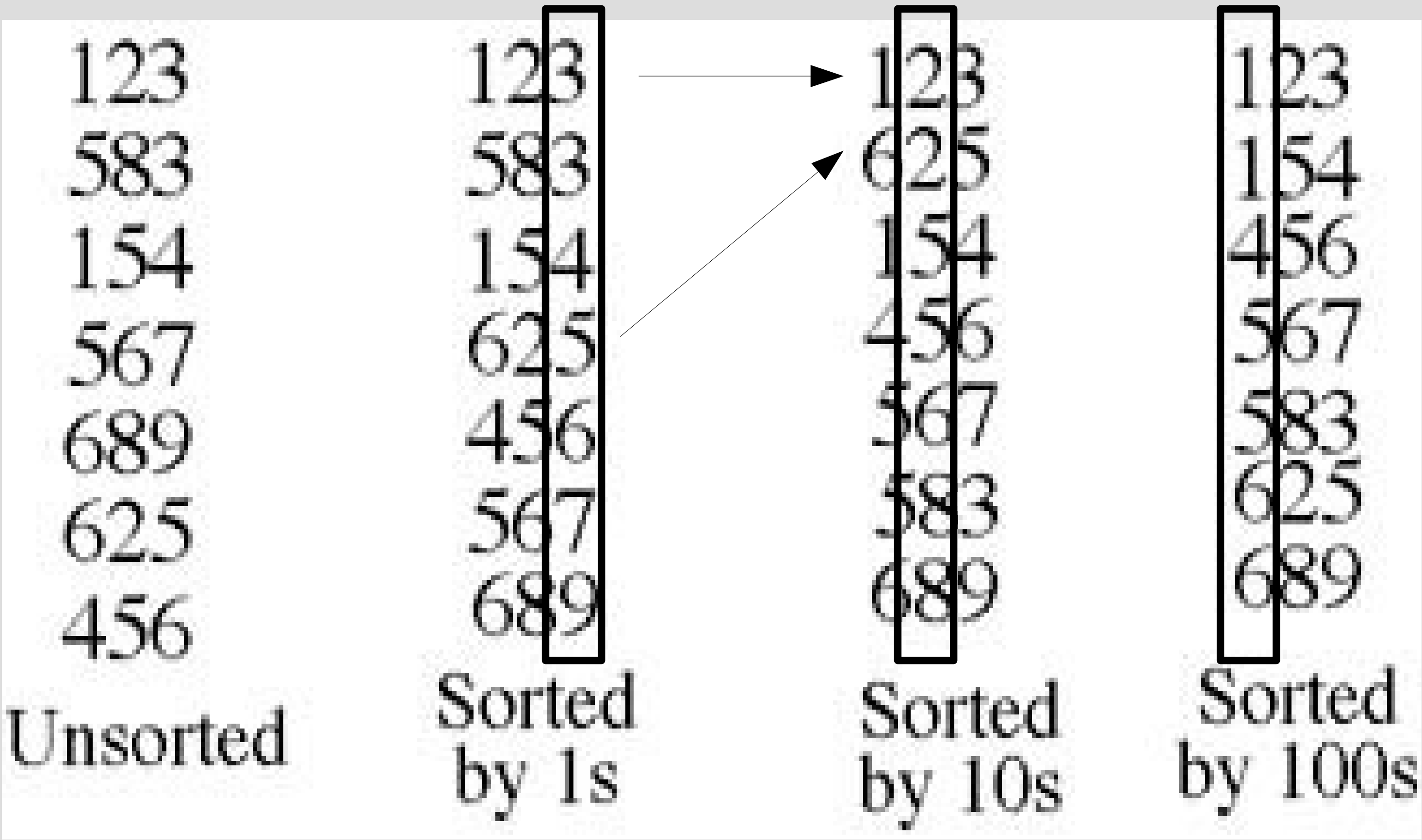
Radix sort

Use a **stable** sort to sort from the least significant digit to most

Psuedo code: (A=input)

```
for i = 1 to d
   stable sort of A on digit i
```

Radix sort



Stable means you can draw lines without crossing for a single digit

Radix sort

Run time?

Radix sort

Run time?

$O((b/r) (n+2^r))$

b-bits total, r bits per 'digit'

$d = b/r$ digits

Each count sort takes $O(n + 2^r)$

runs count sort d times...

$O(d(n+2^r)) = O(b/r (n + 2^r))$

Radix sort

Run time?

if $b < \lg(n)$, $\Theta(n)$

if $b \geq \lg(n)$, $\Theta(n \lg n)$

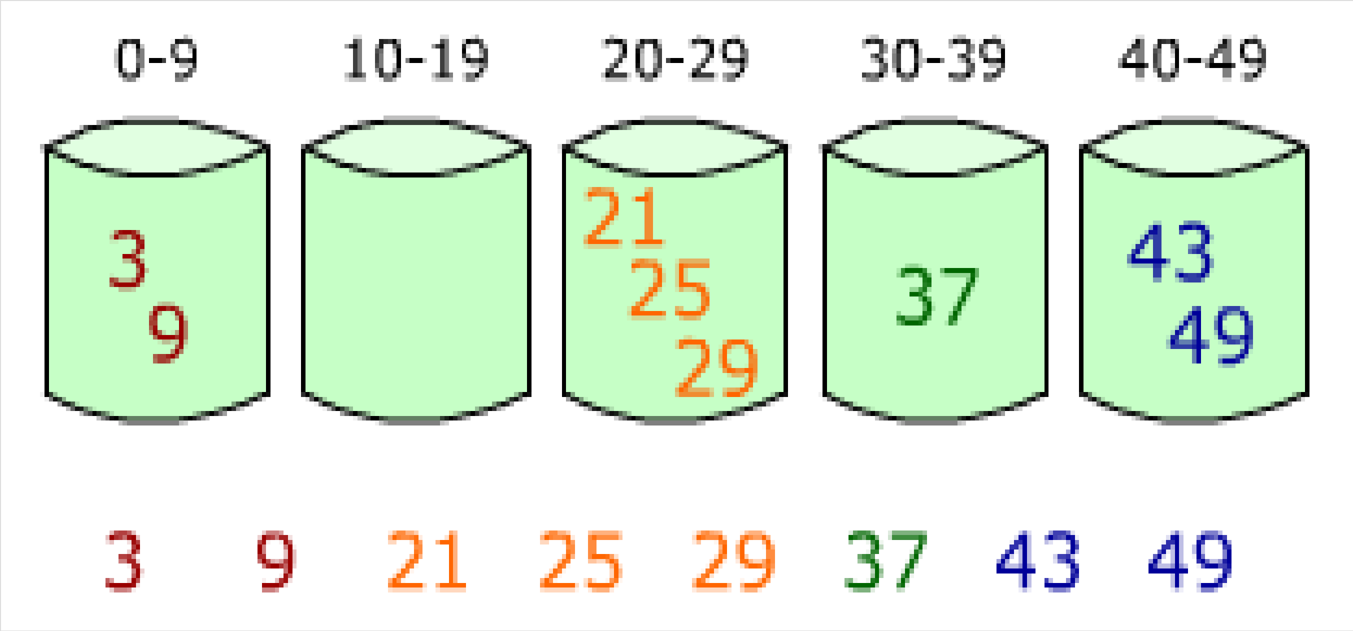
Bucket sort

1. Group similar items into a bucket

2. Sort each bucket individually

3. Merge buckets

Bucket sort



As a human, I recommend this sort if you have large n

Bucket sort

(specific to fractional numbers)

(also assumes n buckets for n numbers)

```
for i = 0 to A.length
  insert A[ i ] into B[ floor(n A[ i ]) ]
for i = 0 to B.length
  sort list B[ i ] with insertion sort
concatenate B[0] to B[1] to B[2]...
```

Bucket sort

Run time?

Bucket sort

Run time?

$\Theta(n)$