Quicksort

Overview

- Invented by Tony Hoare in 1961
- divide-and-conquer approach
- Uses **partitioning** of the data:
  - rearrange array into smaller and larger parts
- Depends on a **pivot** value
- O(n log n) at best, on average
- $O(n^2)$ at worst
  - but worst case can be made very unlikely

Why quicksort?

- Already have mergesort
  - which is O(n log n) in the worst case
- But mergesort
  - needs O(n) extra space
  - copies every value twice
    - once to temp array, then back
- Quicksort copies less data
- More cache-friendly

Partition

- Given array:

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

- 1. Pick a ***pivot*** value (usually first value)

```
(3)  6   1   8   7   5   0   2   9
```

- 2. Rearrange pivot and the other values:

```
 2   1   0  (3)  7   5   6   8   9
 ---------       -----------------
   small               big
```

- Does a bit of sorting:
  - values < pivot: on left side
  - values > pivot: on right side
  - pivot between them, **IN ITS SORTED POSITION**
    - if array was sorted, **3** should have been in fourth position ✓

Do partition again!

- Partition the big values, with pivot 7:
  2 1 0 (3) 5 6 (7) 8 9
  small middle big

- And partition small guys, pivot 2:
  0 1 (2) (3) 5 6 (7) 8 9
  sorted! (got lucky)

- Suggests a recursive algorithm:

Quicksort

- Quicksort: recursive partitioning:

```
Quicksort(array) {
    partition(array);
    quicksort(left side of array);
    quicksort(right side of array);
}
```

More on Partitioning

- Can partition in a ***single*** pass through array
  - O(n) cost
- Destroys order: ("big" values emphasized)

before:	3	**6**	1	**8**	**7**	**5**	0	2	**9**
after:	2	1	0	3	**7**	**5**	**6**	**8**	**9**

6 and 8 moved, relative to other "big" values

  - Hence quicksort is not a ***stable*** sort

Partition, in progress

-

0	1	2	3	4	5	6	7
(3)	1	0	8	9	5	6	2
p	small		big		next	(not yet done)	

- Visit each next value
  - put in small region if a[next] < pivot
  - put in big region if a[next] > pivot

Next value is big?

0	1	2	3	4	5	6	7
(3)	1	0	8	9	5	6	2
p	small		big			next	

- 5 > 3 was big, just extended big region
  - actually did nothing
  - just advanced next

Next value is small?

-

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

p small big next

- 2 < 3 is small, move it to small region, but how?
- Swap with first big value:

(3)	1	0	2	9	5	6	8

p small big

- And extend "small" region

Finishing up

- Pivot is in wrong place

(3)	1	0	2	9	5	6	8
p		small				big	

- Swap it with last small value:

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

- Pivot ends up at index 3, in this case.

Implementing partition

- Need to keep track of:
  1. pivot value
  2. next index
  3. end of small region
  4. end of big region?
     No need (it's just next – 1)

```
int partition(int[] a, int lo, int hi) {

   pivot = a[lo];

   int last_small = lo;

   int next = lo + 1;

   while (next <= hi) {

      if (a[next] < pivot)

         swap(a, next, ++last_small);

      next++;
   }

   swap(a, lo, last_small);

   return last_small;
}
```

← lo and hi give region within the array

← remember the pivot

← small region is empty, initially

← it's small, so

← swap with first big, extend small region

← if big, just extend big region

← put pivot into place

← tell caller where pivot ended up

Partition: Time cost

- Always does n - 1 comparisons
- Does between 1 and n swaps
- Total cost: $O(n)$

Quicksort code

```
quicksort(int[] a) {

   quicksort(a, 0, a.length - 1);

}

quicksort(int[] a, int lo, int hi) {

   if (hi > lo) {

      int j = partition(a, lo, hi);

      quicksort(a, lo, j – 1);

      quicksort(a, j + 1, hi);
   }
}
```

← lo, hi cover whole array

← only sort if more than one value

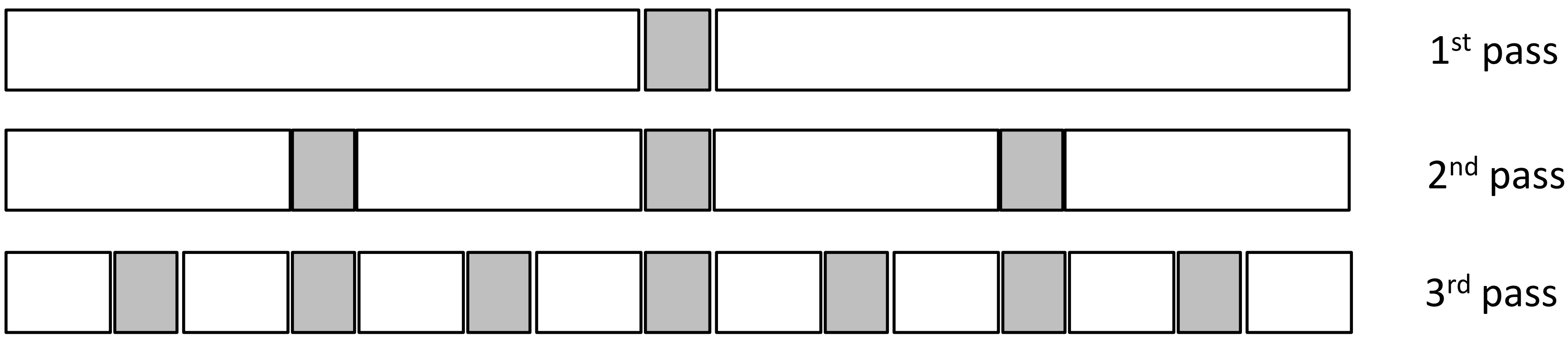
← **j** is where pivot ended up

← sort small region recursively, but leave pivot in place **(j-1)**

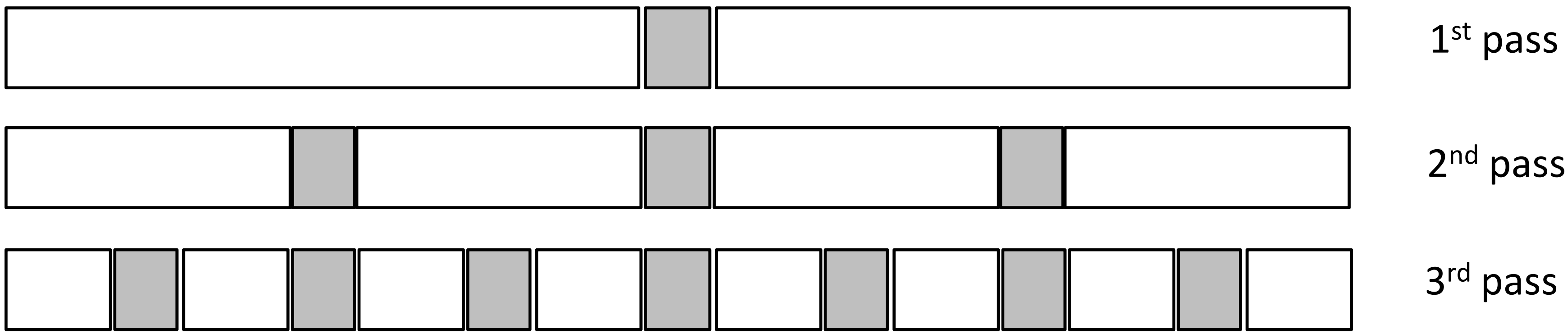
← sort big region recursively, but leave pivot in place **(j+1)**

Quicksort: best case

- Best case: pivot ends exactly in the middle, ***every time***: (pivots in gray)



Quicksort: best case



- How many passes?
  - each pass cuts regions in half
  - last pass has regions with ***one*** value
- So there are $log_2 n$ passes

Quicksort: best case

- Each pass does O(n) comparisons
- And there are $log_2 n$ passes
- So best case is O(n log n) comparisons

- Average case also costs O(n log n)
  - assuming pivot can end up at any position, with equal probability

Quicksort: **WORST** case

- Worst-case when
  - 1. pivot ends up at LEFTMOST (lo) position, or
  - 2. pivot ends up at RIGHTMOST (hi) position
- Then there are n passes, not $log_2 n$ passes
- Total cost: $1 + 2 + 3 + \ldots + (n\text{-}1) = O(n^2)$

Worst-case Input?

- Input array is sorted
  - pivot ends up at leftmost position
- Input array is reverse-sorted
  - pivot ends up at rightmost position

- Q: What’s the real issue?
- A: We’re picking the pivot wrong!

Better Pivot Choice: Randomized

- 1. Choose pivot from random index in array
- 2. Swap with first value
- 3. Then run partition as usual
- Get average-case cost: O(*n log n*)
- Result: you **MIGHT** be very unlucky
  – might **always** pick smallest value,
  – get worst case $O(n^2)$
- But probability of this is *1/n!*
  – which is pretty much zero