23: Quicksort (live)

Lisa Yan and Jerry Cain
November 4, 2020

Quick slide reference

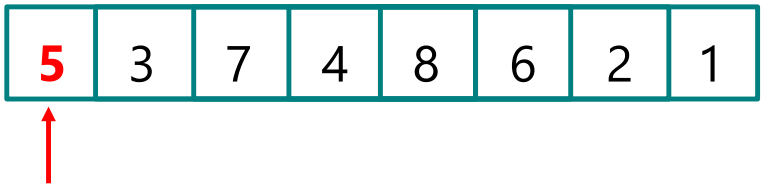
Quicksort

Quicksort

You have been told Quicksort is O(n log n) which is the “average case.”

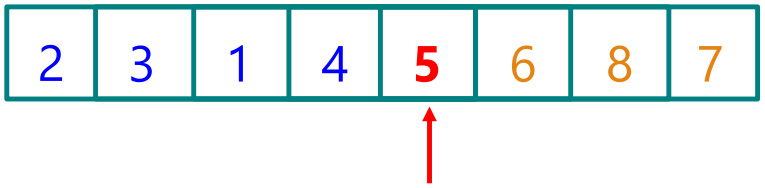
Now we get to prove it! ☺ ☺ ☺ ☺ ☺

Quicksort refresher



1. Select “pivot”

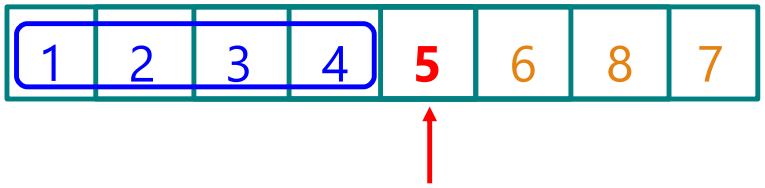
Quicksort refresher



1. Select "pivot"
2. Partition the array

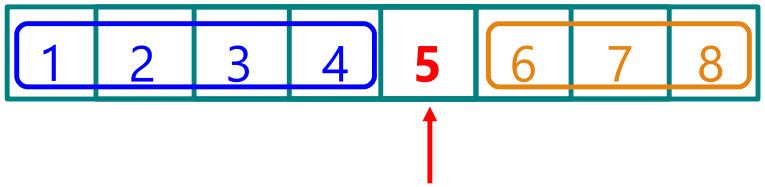
- Everything < pivot on left
- Everything ≥ pivot on right
- Pivot in-between

Quicksort refresher



1. Select “pivot”
2. Partition the array
3. Recursively sort left partition

Quicksort refresher



1. Select “pivot”
2. Partition the array
3. Recursively sort left partition
4. Recursively sort right partition

Quicksort refresher

1. Select “pivot”
2. Partition the array
3. Recursively sort left partition
4. Recursively sort right partition

Everything is sorted!

Quicksort code

```
void quicksort(int arr[], int n)
{
   if (n < 2) return;
   int boundary = partition(arr, n);

   // Sort subarray up to pivot
   quicksort(arr, boundary);
   // Sort subarray after pivot to end
   quicksort(arr + boundary + 1, n – boundary – 1);
}
```

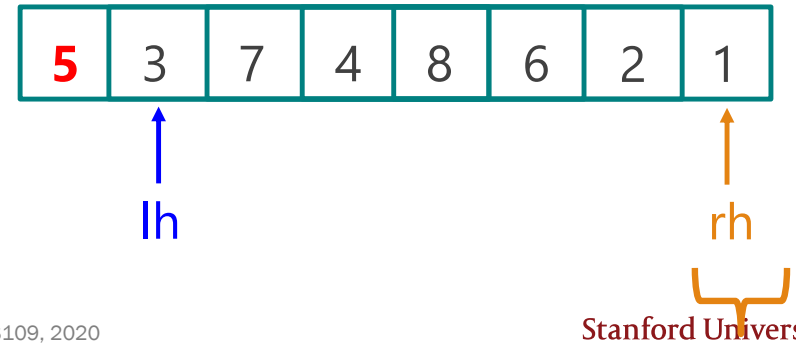
`boundary`:

= index of pivot

= # of elements before pivot

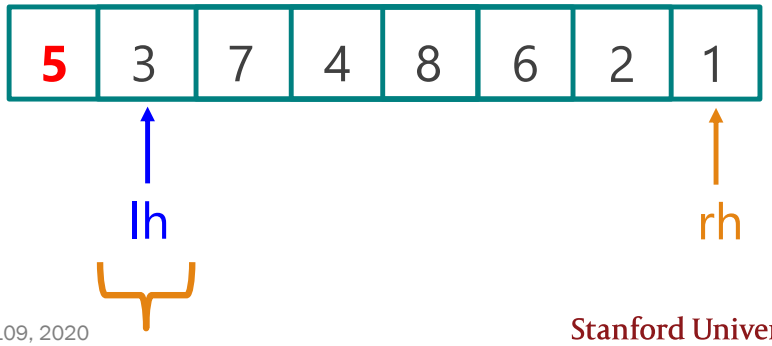
Quicksort's wingman (who does all the work)

```
int partition(int arr[], int n)
{
   int lh = 1, rh = n - 1;

   int pivot = arr[0];
   while (true) {
      while (lh < rh && arr[rh] >= pivot) rh--;
      while (lh < rh && arr[lh] < pivot) lh++;
      if (lh == rh) break;
      swap(arr[lh], arr[rh]);
   }
   if (arr[lh] >= pivot) return 0;
   swap(arr[0], arr[lh]);
   return lh;
}
```



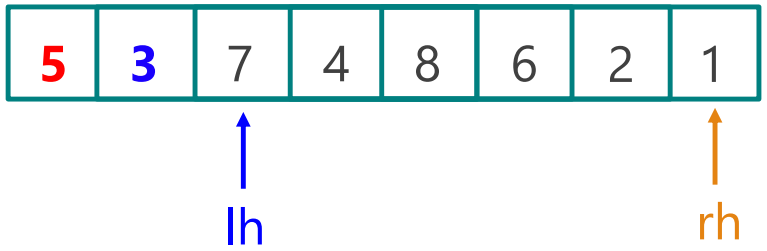
Quicksort's wingman (who does all the work)

```
int partition(int arr[], int n)
{
   int lh = 1, rh = n - 1;

   int pivot = arr[0];
   while (true) {
     while (lh < rh && arr[rh] >= pivot) rh--;
     while (lh < rh && arr[lh] < pivot) lh++;
     if (lh == rh) break;
     swap(arr[lh], arr[rh]);
   }
   if (arr[lh] >= pivot) return 0;
   swap(arr[0], arr[lh]);
   return lh;
}
```



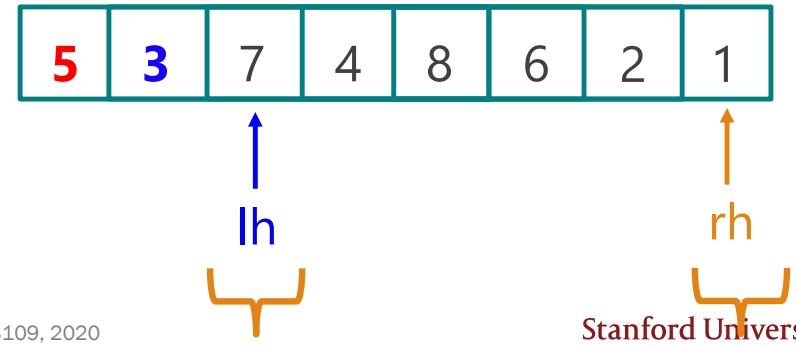
Quicksort's wingman (who does all the work)

```
int partition(int arr[], int n)
{
   int lh = 1, rh = n - 1;

   int pivot = arr[0];
   while (true) {
     while (lh < rh && arr[rh] >= pivot) rh--;
     while (lh < rh && arr[lh] < pivot) lh++;
     if (lh == rh) break;
     swap(arr[lh], arr[rh]);
   }
   if (arr[lh] >= pivot) return 0;
   swap(arr[0], arr[lh]);
   return lh;
}
```



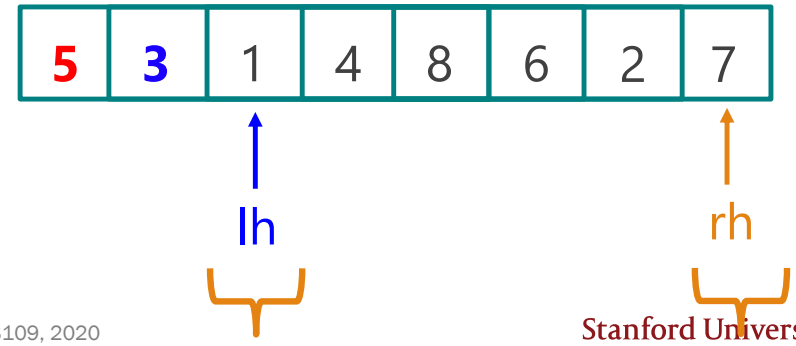
Quicksort's wingman (who does all the work)

```
int partition(int arr[], int n)
{
   int lh = 1, rh = n - 1;

   int pivot = arr[0];
   while (true) {
     while (lh < rh && arr[rh] >= pivot) rh--;
     while (lh < rh && arr[lh] < pivot) lh++;
     if (lh == rh) break;
     swap(arr[lh], arr[rh]);
   }
   if (arr[lh] >= pivot) return 0;
   swap(arr[0], arr[lh]);
   return lh;
}
```



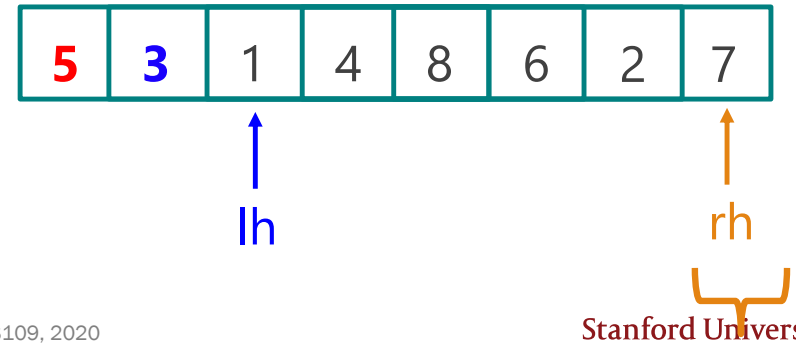
Quicksort's wingman (who does all the work)

```
int partition(int arr[], int n)
{
   int lh = 1, rh = n - 1;

   int pivot = arr[0];
   while (true) {
     while (lh < rh && arr[rh] >= pivot) rh--;
     while (lh < rh && arr[lh] < pivot) lh++;
     if (lh == rh) break;
     swap(arr[lh], arr[rh]);
   }
   if (arr[lh] >= pivot) return 0;
   swap(arr[0], arr[lh]);
   return lh;
}
```



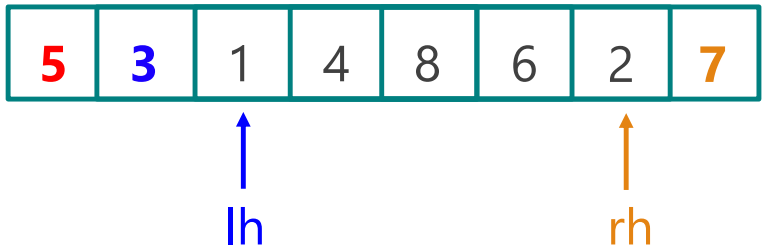
Quicksort's wingman (who does all the work)

```
int partition(int arr[], int n)
{
   int lh = 1, rh = n - 1;

   int pivot = arr[0];
   while (true) {
      while (lh < rh && arr[rh] >= pivot) rh--;
      while (lh < rh && arr[lh] < pivot) lh++;
      if (lh == rh) break;
      swap(arr[lh], arr[rh]);
   }
   if (arr[lh] >= pivot) return 0;
   swap(arr[0], arr[lh]);
   return lh;
}
```



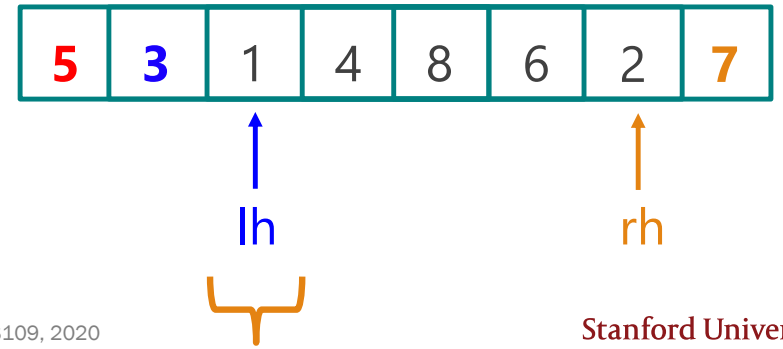
Quicksort's wingman (who does all the work)

```
int partition(int arr[], int n)
{
   int lh = 1, rh = n - 1;

   int pivot = arr[0];
   while (true) {
      while (lh < rh && arr[rh] >= pivot) rh--;
      while (lh < rh && arr[lh] < pivot) lh++;
      if (lh == rh) break;
      swap(arr[lh], arr[rh]);
   }
   if (arr[lh] >= pivot) return 0;
   swap(arr[0], arr[lh]);
   return lh;
}
```



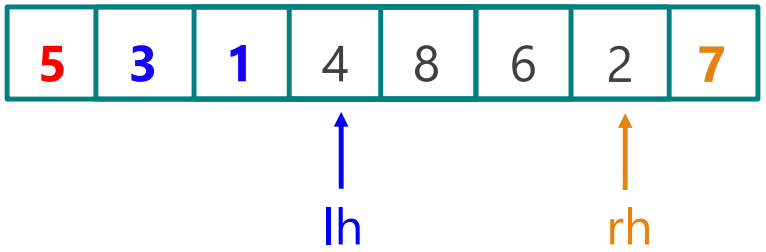
Quicksort's wingman (who does all the work)

```
int partition(int arr[], int n)
{
   int lh = 1, rh = n - 1;

   int pivot = arr[0];
   while (true) {
     while (lh < rh && arr[rh] >= pivot) rh--;
     while (lh < rh && arr[lh] < pivot) lh++;
     if (lh == rh) break;
     swap(arr[lh], arr[rh]);
   }
   if (arr[lh] >= pivot) return 0;
   swap(arr[0], arr[lh]);
   return lh;
}
```



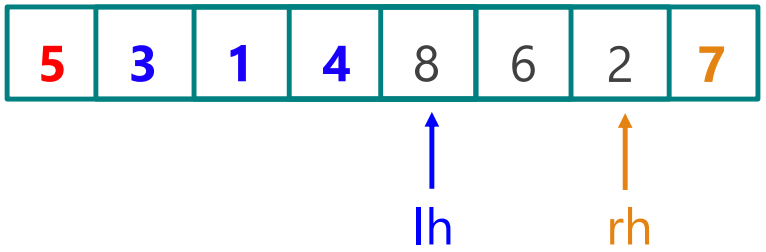
Quicksort's wingman (who does all the work)

```
int partition(int arr[], int n)
{
   int lh = 1, rh = n - 1;

   int pivot = arr[0];
   while (true) {
     while (lh < rh && arr[rh] >= pivot) rh--;
     while (lh < rh && arr[lh] < pivot) lh++;
     if (lh == rh) break;
     swap(arr[lh], arr[rh]);
   }
   if (arr[lh] >= pivot) return 0;
   swap(arr[0], arr[lh]);
   return lh;
}
```



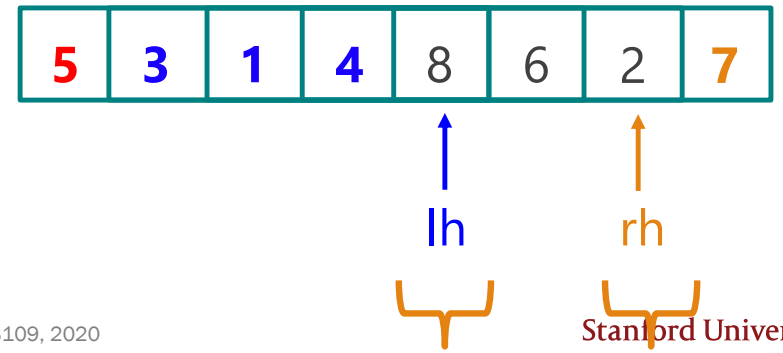
Quicksort's wingman (who does all the work)

```
int partition(int arr[], int n)
{
   int lh = 1, rh = n - 1;

   int pivot = arr[0];
   while (true) {
     while (lh < rh && arr[rh] >= pivot) rh--;
     while (lh < rh && arr[lh] < pivot) lh++;
     if (lh == rh) break;
     swap(arr[lh], arr[rh]);
   }
   if (arr[lh] >= pivot) return 0;
   swap(arr[0], arr[lh]);
   return lh;
}
```



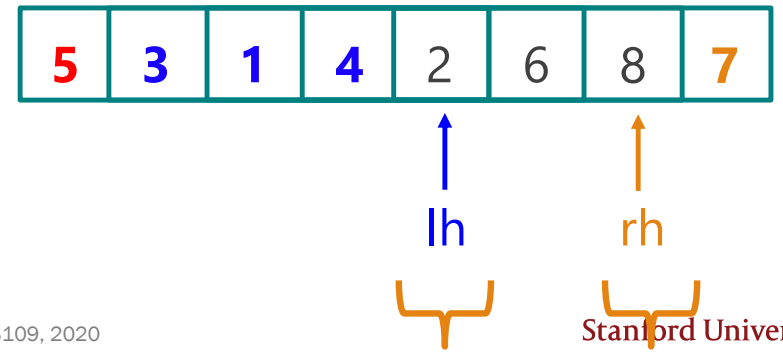
Quicksort's wingman (who does all the work)

```
int partition(int arr[], int n)
{
   int lh = 1, rh = n - 1;

   int pivot = arr[0];
   while (true) {
     while (lh < rh && arr[rh] >= pivot) rh--;
     while (lh < rh && arr[lh] < pivot) lh++;
     if (lh == rh) break;
     swap(arr[lh], arr[rh]);
   }
   if (arr[lh] >= pivot) return 0;
   swap(arr[0], arr[lh]);
   return lh;
}
```



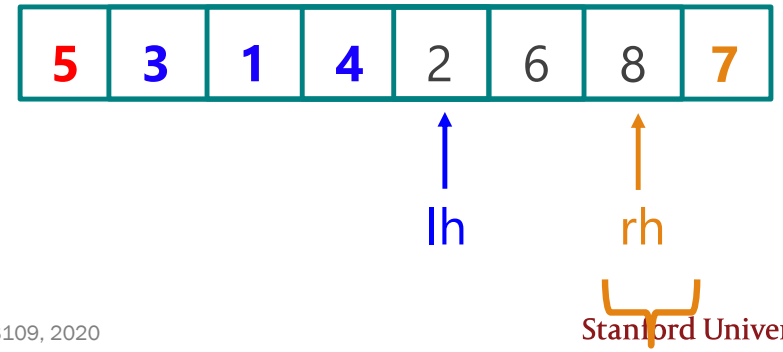
Quicksort's wingman (who does all the work)

```
int partition(int arr[], int n)
{
   int lh = 1, rh = n - 1;

   int pivot = arr[0];
   while (true) {
     while (lh < rh && arr[rh] >= pivot) rh--;
     while (lh < rh && arr[lh] < pivot) lh++;
     if (lh == rh) break;
     swap(arr[lh], arr[rh]);
   }
   if (arr[lh] >= pivot) return 0;
   swap(arr[0], arr[lh]);
   return lh;
}
```



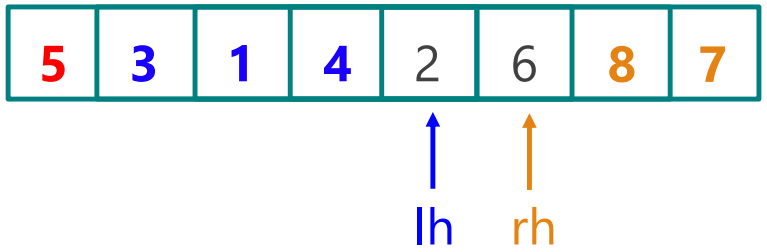
Quicksort's wingman (who does all the work)

```
int partition(int arr[], int n)
{
   int lh = 1, rh = n - 1;

   int pivot = arr[0];
   while (true) {
      while (lh < rh && arr[rh] >= pivot) rh--;
      while (lh < rh && arr[lh] < pivot) lh++;
      if (lh == rh) break;
      swap(arr[lh], arr[rh]);
   }
   if (arr[lh] >= pivot) return 0;
   swap(arr[0], arr[lh]);
   return lh;
}
```



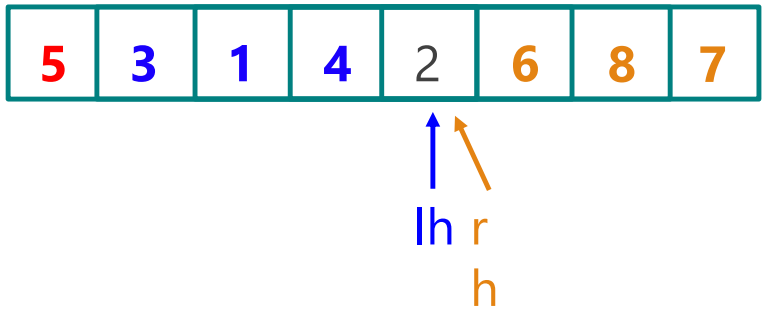
Quicksort's wingman (who does all the work)

```
int partition(int arr[], int n)
{
   int lh = 1, rh = n - 1;

   int pivot = arr[0];
   while (true) {
      while (lh < rh && arr[rh] >= pivot) rh--;
      while (lh < rh && arr[lh] < pivot) lh++;
      if (lh == rh) break;
      swap(arr[lh], arr[rh]);
   }
   if (arr[lh] >= pivot) return 0;
   swap(arr[0], arr[lh]);
   return lh;
}
```



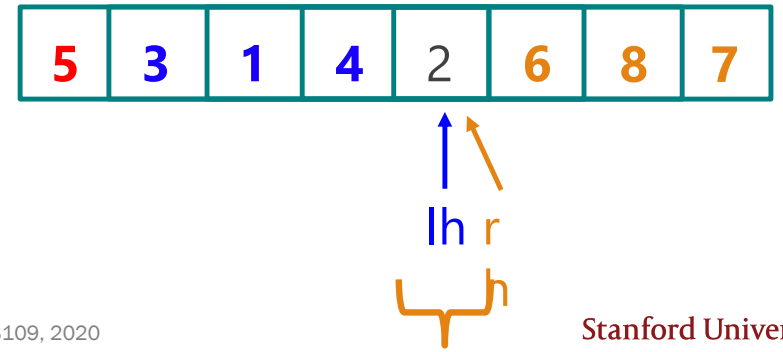
Quicksort's wingman (who does all the work)

```
int partition(int arr[], int n)
{
   int lh = 1, rh = n - 1;

   int pivot = arr[0];
   while (true) {
      while (lh < rh && arr[rh] >= pivot) rh--;
      while (lh < rh && arr[lh] < pivot) lh++;
      if (lh == rh) break;
      swap(arr[lh], arr[rh]);
   }
   if (arr[lh] >= pivot) return 0;
   swap(arr[0], arr[lh]);
   return lh;
}
```



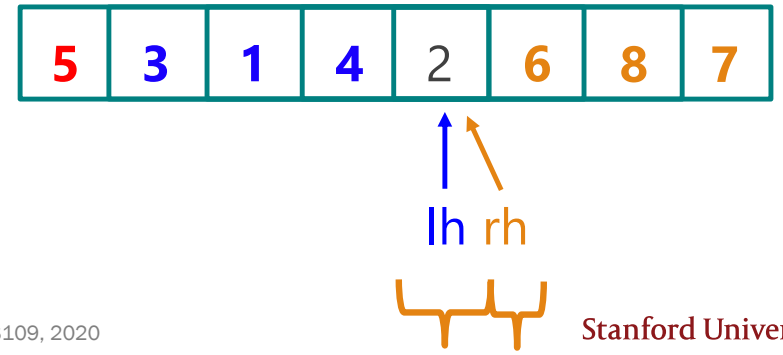
Quicksort's wingman (who does all the work)

```
int partition(int arr[], int n)
{
   int lh = 1, rh = n - 1;

   int pivot = arr[0];
   while (true) {
     while (lh < rh && arr[rh] >= pivot) rh--;
     while (lh < rh && arr[lh] < pivot) lh++;
     if (lh == rh) break;
     swap(arr[lh], arr[rh]);
   }
   if (arr[lh] >= pivot) return 0;
   swap(arr[0], arr[lh]);
   return lh;
}
```



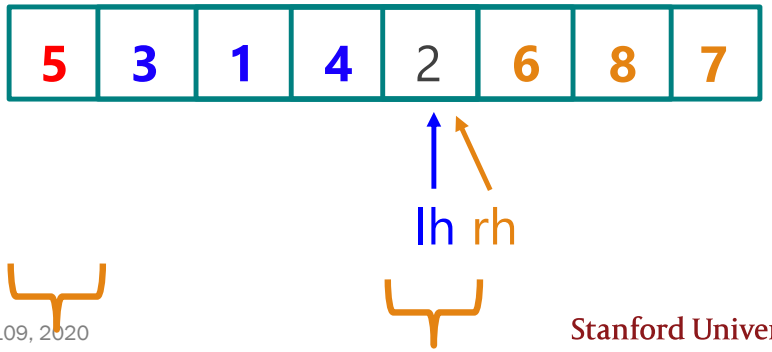
Quicksort's wingman (who does all the work)

```
int partition(int arr[], int n)
{
   int lh = 1, rh = n - 1;

   int pivot = arr[0];
   while (true) {
     while (lh < rh && arr[rh] >= pivot) rh--;
     while (lh < rh && arr[lh] < pivot) lh++;
     if (lh == rh) break;
     swap(arr[lh], arr[rh]);
   }
   if (arr[lh] >= pivot) return 0;
   swap(arr[0], arr[lh]);
   return lh;
}
```



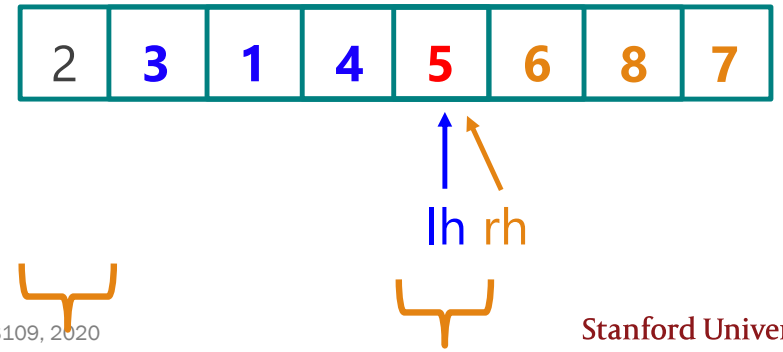
Quicksort's wingman (who does all the work)

```
int partition(int arr[], int n)
{
   int lh = 1, rh = n - 1;

   int pivot = arr[0];
   while (true) {
     while (lh < rh && arr[rh] >= pivot) rh--;
     while (lh < rh && arr[lh] < pivot) lh++;
     if (lh == rh) break;
     swap(arr[lh], arr[rh]);
   }
   if (arr[lh] >= pivot) return 0;
   swap(arr[0], arr[lh]);
   return lh;
}
```



Quicksort's wingman (who does all the work)

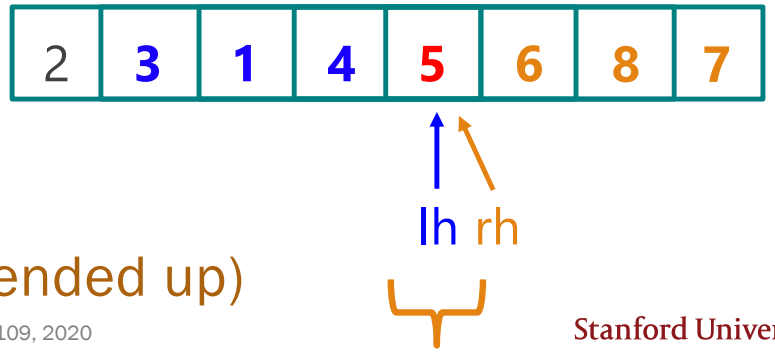
```
int partition(int arr[], int n)
{
   int lh = 1, rh = n - 1;

   int pivot = arr[0];
   while (true) {
     while (lh < rh && arr[rh] >= pivot) rh--;
     while (lh < rh && arr[lh] < pivot) lh++;
     if (lh == rh) break;
     swap(arr[lh], arr[rh]);
   }
   if (arr[lh] >= pivot) return 0;
   swap(arr[0], arr[lh]);
   return lh;
}
```



Quicksort's wingman (who does all the work)

```
int partition(int arr[], int n)
{
   int lh = 1, rh = n - 1;

   int pivot = arr[0];
   while (true) {
     while (lh < rh && arr[rh] >= pivot) rh--;
     while (lh < rh && arr[lh] < pivot) lh++;
     if (lh == rh) break;
     swap(arr[lh], arr[rh]);
   }
   if (arr[lh] >= pivot) return 0;
   swap(arr[0], arr[lh]);
   return lh;
}
```

Complexity of algorithm:
# of comparisons made to pivot



Returns 4 (index where pivot ended up)

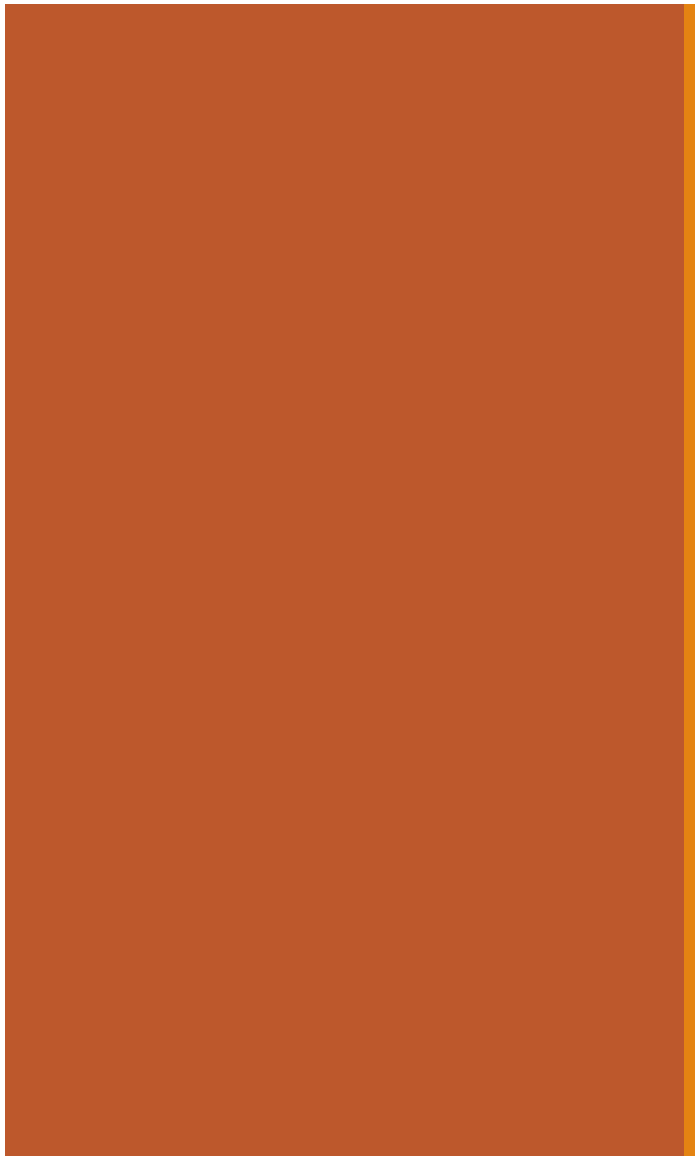
Quicksort complexity

On average, Quicksort is O(n log n), where n = # elements.

Worst case: $O(n^2)$, when the pivot is maximal or minimal on every recursive call.

We can ask two probabilistic questions about runtime:

1. What is P(Quicksort worst case runtime)?
2. What is E[Quicksort runtime]?



LIVE

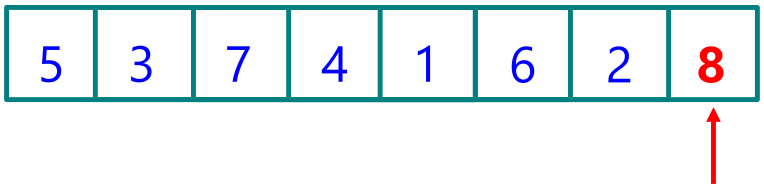
Quicksort: Worst-case runtime

Worst case Quicksort

1. What is P(Quicksort worst case runtime)?

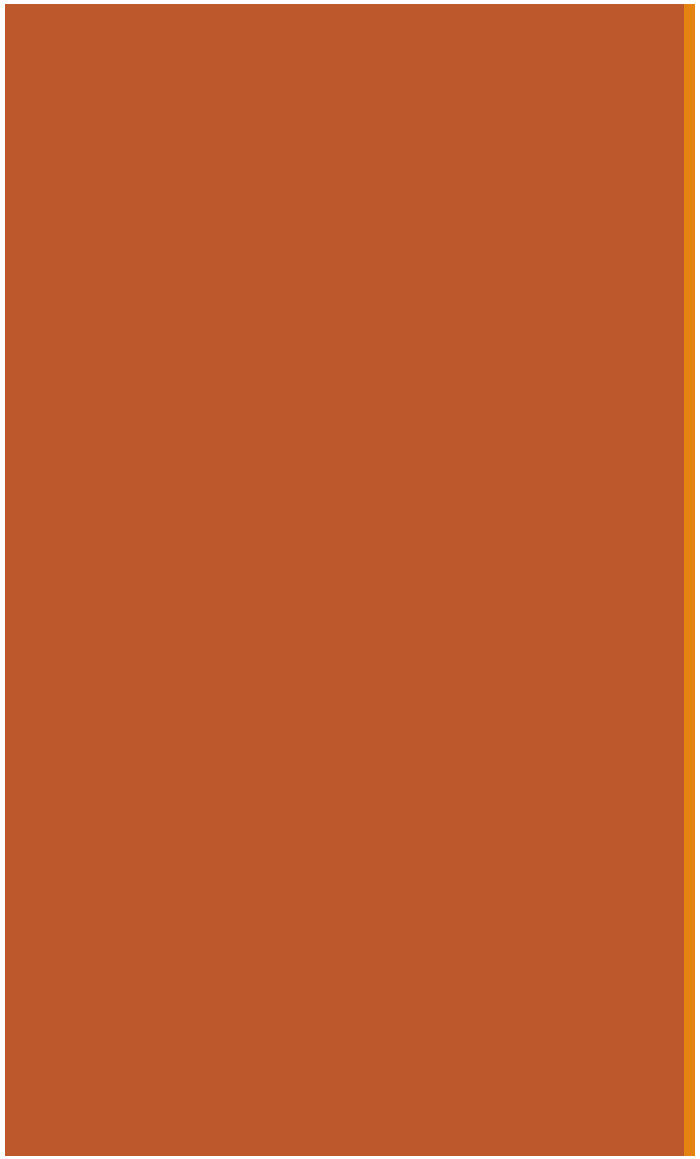
Solution:

- On each recursive call:
  pivot = max/min element, so we are left with n – 1 elements for next recursive call
- 2 possible "bad" pivots (max/min) per call



$$P(\text{worst case}) = \frac{2}{n} \cdot \frac{2}{n-1} \cdots \frac{2}{2} = \frac{2^{n-1}}{n!}$$

Similar for BSTs (pset #1):
As n → ∞, P(worst case) → 0



LIVE

Quicksort: Average runtime

Average Quicksort

2. What is E[Quicksort runtime]?

Define: X = # comparisons to pivot
Want to Find: $E[X]$

(dependent comparisons…use indicator variables!)

Define: $Y_1, Y_2, \ldots, Y_n$ elements in sorted order

$I_{a,b} = 1$ if Y_a, Y_b ever compared (where $Y_a < Y_b$)

Then,

$$E[X] = E\left[\sum_{a=1}^{n-1}\sum_{b=a+1}^{n} I_{ab}\right] = \sum_{a=1}^{n-1}\sum_{b=a+1}^{n} E[I_{ab}] = \sum_{a=1}^{n-1}\sum_{b=a+1}^{n} P\left(Y_a \text{ and } Y_b \text{ ever compared}\right)$$

(unique pairs)

Average Quicksort

2. What is E[Quicksort runtime]?

Define: X = # comparisons to pivot

$Y_1, Y_2, \ldots, Y_n$ elements in sorted order

$I_{a,b} = 1$ if Y_a, Y_b ever compared (where $Y_a < Y_b$)

Then, $$E[X] = \sum_{a=1}^{n-1} \sum_{b=a+1}^{n} P\left(Y_a \text{ and } Y_b \text{ ever compared}\right)$$

$$P\left(Y_a \text{ and } Y_b \text{ ever compared}\right): = \frac{2}{b-a+1}$$

- If pivot < Y_a or > Y_b, not directly compared (but could be in future recursive call)
- Care only about calls where pivot in {Y_a, Y_{a+1}, Y_{a+2}, …, Y_b}

→ either Y_a or Y_b must be pivot if Y_a,Y_b, are to be compared to each other

Average Quicksort

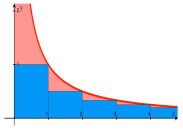
2. What is E[Quicksort runtime]?

Define: X = # comparisons to pivot

$Y_1, Y_2, \ldots, Y_n$ elements in sorted order

$I_{a,b} = 1$ if Y_a, Y_b ever compared (where $Y_a < Y_b$)

$$E[X] = \sum_{a=1}^{n-1} \sum_{b=a+1}^{n} P\left(Y_a \text{ and } Y_b \text{ ever compared}\right) = \sum_{a=1}^{n-1} \sum_{b=a+1}^{n} \frac{2}{b-a+1}$$



$$\sum_{b=a+1}^{n} \frac{2}{b-a+1} \approx \int_{b=a+1}^{n} \frac{2}{b-a+1} db$$

(from here on it's rollercoaster math)

$$= [2\ln(b-a+1)]\Big|_{b=a+1}^{n} = 2\ln(n-a+1) - 2\ln 2 \approx 2\ln(n-a+1)$$

(when *n* is large)

Average Quicksort

2. What is E[Quicksort runtime]?

Define: X = # comparisons to pivot

$Y_1, Y_2, \ldots, Y_n$ elements in sorted order

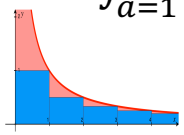
$I_{a,b} = 1$ if Y_a, Y_b ever compared (where $Y_a < Y_b$)

$$E[X] = \sum_{a=1}^{n-1} \sum_{b=a+1}^{n} P\left(Y_a \text{ and } Y_b \text{ ever compared}\right) = \sum_{a=1}^{n-1} \sum_{b=a+1}^{n} \frac{2}{b-a+1} \approx \sum_{a=1}^{n-1} 2\ln(n-a+1)$$

(still more rollercoaster math)

$$\approx \int_{a=1}^{n-1} 2\ln(n-a+1)\,da = 2\int_{a=1}^{n-1} \ln(n-a+1)\,da = -2\int_{y=n}^{2} \ln(y)\,dy$$

(u-substitution: Let y = n – a+1)



$$= -2\,[y\ln(y) - y]\Big|_n^2$$

(Integration by parts: $\int \ln(x)\,dx = x\ln(x) - x$)

$$= -2[(2\ln(2) - 2) - (n\ln(n) - n)] \approx 2n\ln(n) - 2n$$

$$= O(n \log n)$$

Summary of this time

Quicksort:

- While recursive, can be solved as an expectation of a sum of indicator random variables.
- When dealing with a sum of non-trivial indicator probabilities, $\sum_{x=k}^{n} \frac{1}{ax+b} \approx \int_{x=k}^{n} \frac{1}{ax+b} dx$

(QuickSort is beyond the scope of your HW, but it is useful to understand it)