Quicksort

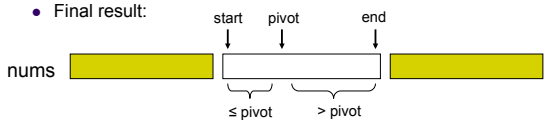
David Kauchak
cs201
Spring 2014

```
public static int partition(double[] nums, int start, int end){
  int lessThanIndex = start-1;

  for( int i = start; i < end; i++ ){
    if( nums[i] <= nums[end] ){
      lessThanIndex++;
      swap(nums, lessThanIndex, i);
    }
  }

  swap(nums, lessThanIndex+1, end);

  return lessThanIndex+1;
}
```

what does this method do?

```
public static int partition(double[] nums, int start, int end){
  int lessThanIndex = start-1;
  for( int i = start; i < end; i++ ){
    if( nums[i] <= nums[end] ){
      lessThanIndex++;
      swap(nums, lessThanIndex, i);
    }
  }
  swap(nums, lessThanIndex+1, end);
  return lessThanIndex+1;
}
```

- nums[end] is called the ***pivot***
- Partitions the elements nums[start…end-1] in to two sets, those ≤ pivot and those > pivot
- Operates in place
- Final result:



… 5 7 1 2 8 4 3 6 …

start end

```
public static int partition(double[] nums, int start, int end){
  int lessThanIndex = start-1;
  for( int i = start; i < end; i++ ){
    if( nums[i] <= nums[end] ){
      lessThanIndex++;
      swap(nums, lessThanIndex, i);
    }
  }
  swap(nums, lessThanIndex+1, end);
  return lessThanIndex+1;
}
```

lessThanIndex

… 5 7 1 2 8 4 3 6 …

start end

```
public static int partition(double[] nums, int start, int end){
    int lessThanIndex = start-1;
    for( int i = start; i < end; i++ ){
        if( nums[i] <= nums[end] ){
            lessThanIndex++;
            swap(nums, lessThanIndex, i);
        }
    }
    swap(nums, lessThanIndex+1, end);
    return lessThanIndex+1;
}
```

lessThanIndex i

… 5 7 1 2 8 4 3 6 …

start end

```
public static int partition(double[] nums, int start, int end){
    int lessThanIndex = start-1;
    for( int i = start; i < end; i++ ){
        if( nums[i] <= nums[end] ){
            lessThanIndex++;
            swap(nums, lessThanIndex, i);
        }
    }
    swap(nums, lessThanIndex+1, end);
    return lessThanIndex+1;
}
```

lessThanIndex i

… 5 7 1 2 8 4 3 6 …

start end

```
public static int partition(double[] nums, int start, int end){
    int lessThanIndex = start-1;
    for( int i = start; i < end; i++ ){
        if( nums[i] <= nums[end] ){
            lessThanIndex++;
            swap(nums, lessThanIndex, i);
        }
    }
    swap(nums, lessThanIndex+1, end);
    return lessThanIndex+1;
}
```

lessThanIndex i

… 5 7 1 2 8 4 3 6 …

start end

```
public static int partition(double[] nums, int start, int end){
    int lessThanIndex = start-1;
    for( int i = start; i < end; i++ ){
        if( nums[i] <= nums[end] ){
            lessThanIndex++;
            swap(nums, lessThanIndex, i);
        }
    }
    swap(nums, lessThanIndex+1, end);
    return lessThanIndex+1;
}
```

lessThanIndex i

… 5 7 1 2 8 4 3 6 …

start **end**

```
public static int partition(double[] nums, int start, int end){
   int lessThanIndex = start-1;
   for( int i = start; i < end; i++ ){
      if( nums[i] <= nums[end] ){
         lessThanIndex++;
         swap(nums, lessThanIndex, i);
      }
   }
   swap(nums, lessThanIndex+1, end);
   return lessThanIndex+1;
}
```

lessThanIndex i

… 5 7 1 2 8 4 3 6 …

start **end**

```
public static int partition(double[] nums, int start, int end){
   int lessThanIndex = start-1;
   for( int i = start; i < end; i++ ){
      if( nums[i] <= nums[end] ){
         lessThanIndex++;
         swap(nums, lessThanIndex, i);
      }
   }
   swap(nums, lessThanIndex+1, end);
   return lessThanIndex+1;
}
```

lessThanIndex i

… 5 7 1 2 8 4 3 6 …

start **end**

```
public static int partition(double[] nums, int start, int end){
   int lessThanIndex = start-1;
   for( int i = start; i < end; i++ ){
      if( nums[i] <= nums[end] ){
         lessThanIndex++;
         swap(nums, lessThanIndex, i);
      }
   }
   swap(nums, lessThanIndex+1, end);
   return lessThanIndex+1;
}
```

lessThanIndex i

… 5 7 1 2 8 4 3 6 …

start **end**

```
public static int partition(double[] nums, int start, int end){
   int lessThanIndex = start-1;
   for( int i = start; i < end; i++ ){
      if( nums[i] <= nums[end] ){
         lessThanIndex++;
         swap(nums, lessThanIndex, i);
      }
   }
   swap(nums, lessThanIndex+1, end);
   return lessThanIndex+1;
}
```

lessThanIndex ↓ i ↓

… 5 1 7 2 8 4 3 6 …

start ↑ end ↑

```
public static int partition(double[] nums, int start, int end){
  int lessThanIndex = start-1;
  for( int i = start; i < end; i++ ){
    if( nums[i] <= nums[end] ){
      lessThanIndex++;
      swap(nums, lessThanIndex, i);
    }
  }
  swap(nums, lessThanIndex+1, end);
  return lessThanIndex+1;
}
```

lessThanIndex ↓ i ↓

… 5 1 7 2 8 4 3 6 …

start ↑ end ↑

```
public static int partition(double[] nums, int start, int end){
  int lessThanIndex = start-1;
  for( int i = start; i < end; i++ ){
    if( nums[i] <= nums[end] ){
      lessThanIndex++;
      swap(nums, lessThanIndex, i);
    }
  }
  swap(nums, lessThanIndex+1, end);
  return lessThanIndex+1;
}
```

lessThanIndex ↓ i ↓

… 5 1 7 2 8 4 3 6 …

start ↑ end ↑

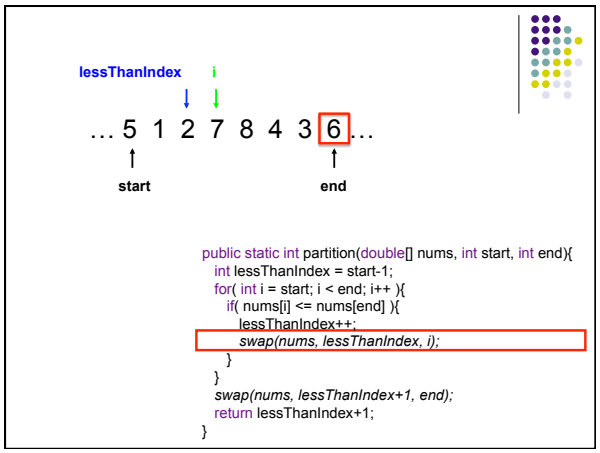
```
public static int partition(double[] nums, int start, int end){
  int lessThanIndex = start-1;
  for( int i = start; i < end; i++ ){
    if( nums[i] <= nums[end] ){
      lessThanIndex++;
      swap(nums, lessThanIndex, i);
    }
  }
  swap(nums, lessThanIndex+1, end);
  return lessThanIndex+1;
}
```

lessThanIndex ↓ i ↓

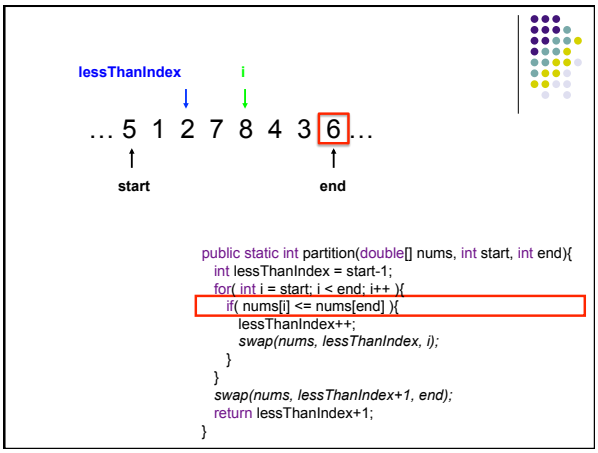
… 5 1 2 7 8 4 3 6 …

start ↑ end ↑

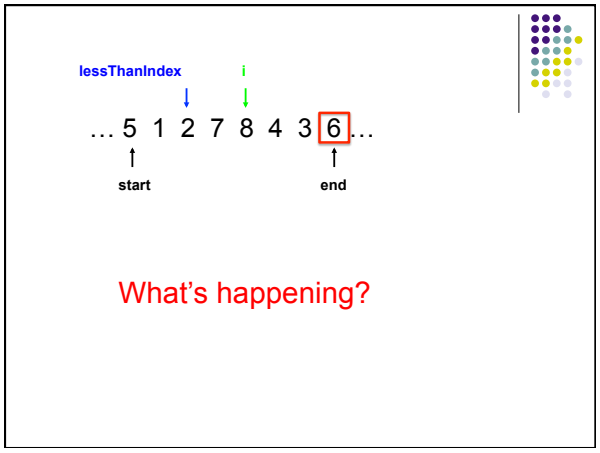
```
public static int partition(double[] nums, int start, int end){
  int lessThanIndex = start-1;
  for( int i = start; i < end; i++ ){
    if( nums[i] <= nums[end] ){
      lessThanIndex++;
      swap(nums, lessThanIndex, i);
    }
  }
  swap(nums, lessThanIndex+1, end);
  return lessThanIndex+1;
}
```

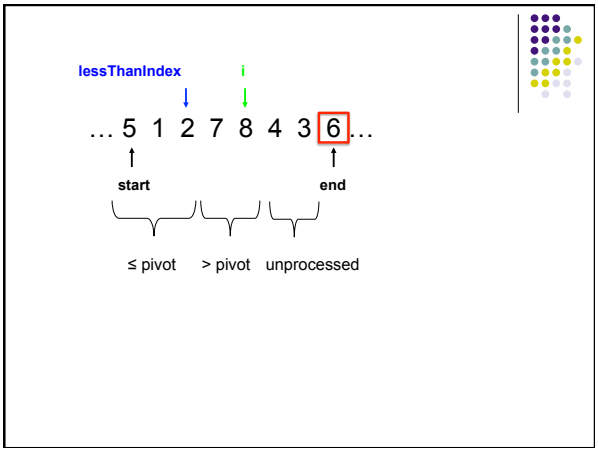


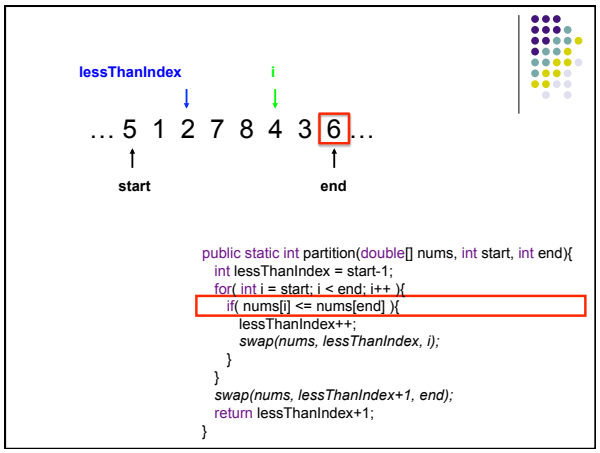

```
public static int partition(double[] nums, int start, int end){
  int lessThanIndex = start-1;
  for( int i = start; i < end; i++ ){
    if( nums[i] <= nums[end] ){
      lessThanIndex++;
      swap(nums, lessThanIndex, i);
    }
  }
  swap(nums, lessThanIndex+1, end);
  return lessThanIndex+1;
}
```



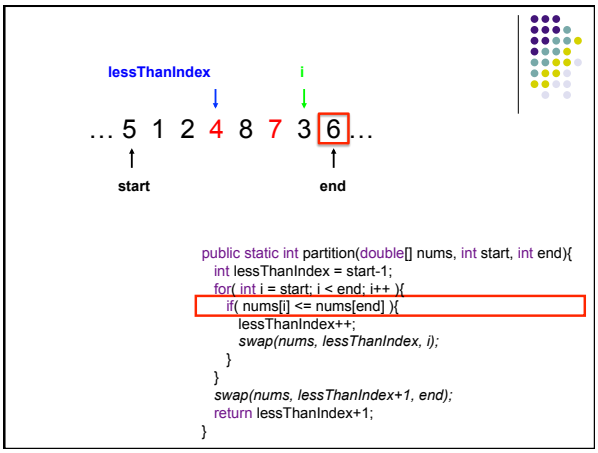

```
public static int partition(double[] nums, int start, int end){
  int lessThanIndex = start-1;
  for( int i = start; i < end; i++ ){
    if( nums[i] <= nums[end] ){
      lessThanIndex++;
      swap(nums, lessThanIndex, i);
    }
  }
  swap(nums, lessThanIndex+1, end);
  return lessThanIndex+1;
}
```



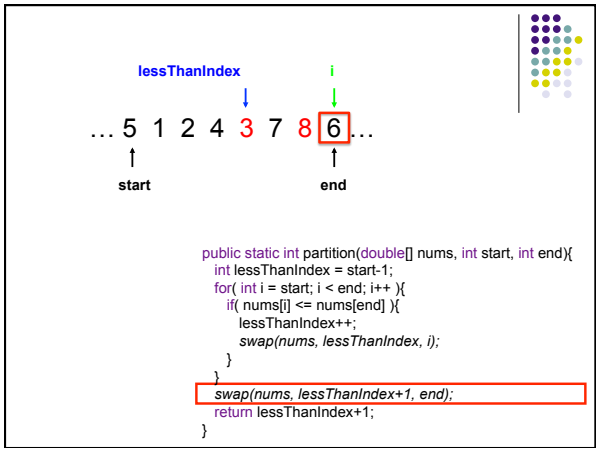




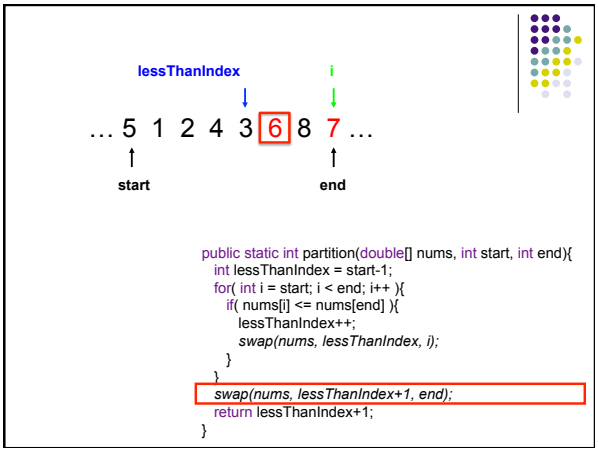

```
public static int partition(double[] nums, int start, int end){
  int lessThanIndex = start-1;
  for( int i = start; i < end; i++ ){
    if( nums[i] <= nums[end] ){
      lessThanIndex++;
      swap(nums, lessThanIndex, i);
    }
  }
  swap(nums, lessThanIndex+1, end);
  return lessThanIndex+1;
}
```




```
public static int partition(double[] nums, int start, int end){
  int lessThanIndex = start-1;
  for( int i = start; i < end; i++ ){
    if( nums[i] <= nums[end] ){
      lessThanIndex++;
      swap(nums, lessThanIndex, i);
    }
  }
  swap(nums, lessThanIndex+1, end);
  return lessThanIndex+1;
}
```




```
public static int partition(double[] nums, int start, int end){
  int lessThanIndex = start-1;
  for( int i = start; i < end; i++ ){
    if( nums[i] <= nums[end] ){
      lessThanIndex++;
      swap(nums, lessThanIndex, i);
    }
  }
  swap(nums, lessThanIndex+1, end);
  return lessThanIndex+1;
}
```




```
public static int partition(double[] nums, int start, int end){
  int lessThanIndex = start-1;
  for( int i = start; i < end; i++ ){
    if( nums[i] <= nums[end] ){
      lessThanIndex++;
      swap(nums, lessThanIndex, i);
    }
  }
  swap(nums, lessThanIndex+1, end);
  return lessThanIndex+1;
}
```

Partition running time?

O(n)

```
public static int partition(double[] nums, int start, int end){
  int lessThanIndex = start-1;
  for( int i = start; i < end; i++ ){
    if( nums[i] <= nums[end] ){
      lessThanIndex++;
      swap(nums, lessThanIndex, i);
    }
  }
  swap(nums, lessThanIndex+1, end);
  return lessThanIndex+1;
}
```

Quicksort

How can we use this method to sort nums?

```
public static int partition(double[] nums, int start, int end){
  int lessThanIndex = start-1;
  for( int i = start; i < end; i++ ){
    if( nums[i] <= nums[end] ){
      lessThanIndex++;
      swap(nums, lessThanIndex, i);
    }
  }
  swap(nums, lessThanIndex+1, end);
  return lessThanIndex+1;
}
```

Quicksort

```
private static void quicksortHelper(double[] nums, int start, int end){
  if( start < end ){
    int partition = partition(nums, start, end);
    quicksortHelper(nums, start, partition-1);
    quicksortHelper(nums, partition+1, end);
  }
}

public static int partition(double[] nums, int start, int end){
  int lessThanIndex = start-1;
  for( int i = start; i < end; i++ ){
    if( nums[i] <= nums[end] ){
      lessThanIndex++;
      swap(nums, lessThanIndex, i);
    }
  }
  swap(nums, lessThanIndex+1, end);
  return lessThanIndex+1;
}
```

8 5 1 3 6 2 7 4

```
private static void quicksortHelper(double[] nums, int start, int end){
  if( start < end ){
    int partition = partition(nums, start, end);
    quicksortHelper(nums, start, partition-1);
    quicksortHelper(nums, partition+1, end);
  }
}
```

8 5 1 3 6 2 7 4

```
private static void quicksortHelper(double[] nums, int start, int end){
   if( start < end ){
      int partition = partition(nums, start, end);
      quicksortHelper(nums, start, partition-1);
      quicksortHelper(nums, partition+1, end);
   }
}
```

1 3 2 4 6 8 7 5

```
private static void quicksortHelper(double[] nums, int start, int end){
   if( start < end ){
      int partition = partition(nums, start, end);
      quicksortHelper(nums, start, partition-1);
      quicksortHelper(nums, partition+1, end);
   }
}
```

1 3 2 4 6 8 7 5

```
private static void quicksortHelper(double[] nums, int start, int end){
   if( start < end ){
      int partition = partition(nums, start, end);
      quicksortHelper(nums, start, partition-1);
      quicksortHelper(nums, partition+1, end);
   }
}
```

1 3 2 4 6 8 7 5

```
private static void quicksortHelper(double[] nums, int start, int end){
   if( start < end ){
      int partition = partition(nums, start, end);
      quicksortHelper(nums, start, partition-1);
      quicksortHelper(nums, partition+1, end);
   }
}
```

1 2 3 4 6 8 7 5

```java
private static void quicksortHelper(double[] nums, int start, int end){
  if( start < end ){
    int partition = partition(nums, start, end);
    quicksortHelper(nums, start, partition-1);
    quicksortHelper(nums, partition+1, end);
  }
}
```

1 2 3 4 6 8 7 5

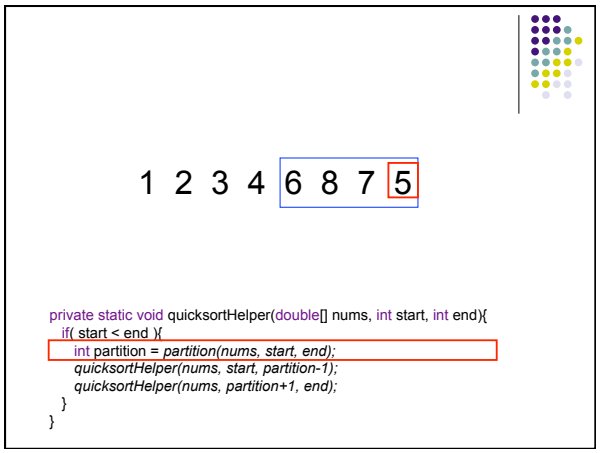
```java
private static void quicksortHelper(double[] nums, int start, int end){
  if( start < end ){
    int partition = partition(nums, start, end);
    quicksortHelper(nums, start, partition-1);
    quicksortHelper(nums, partition+1, end);
  }
}
```

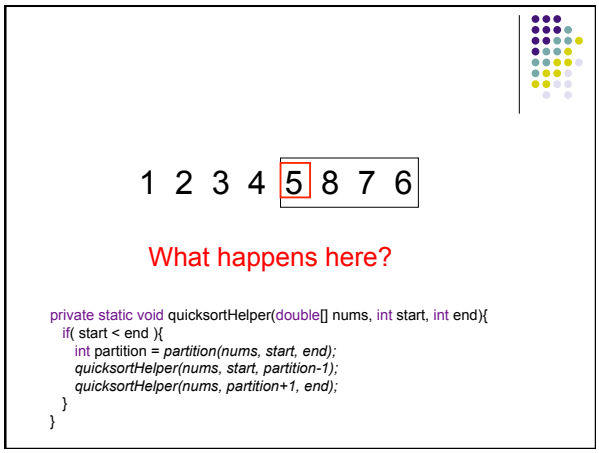
1 2 3 4 6 8 7 5

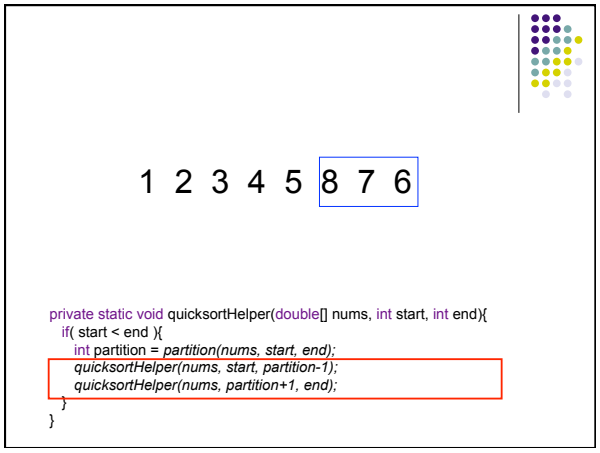
```java
private static void quicksortHelper(double[] nums, int start, int end){
  if( start < end ){
    int partition = partition(nums, start, end);
    quicksortHelper(nums, start, partition-1);
    quicksortHelper(nums, partition+1, end);
  }
}
```

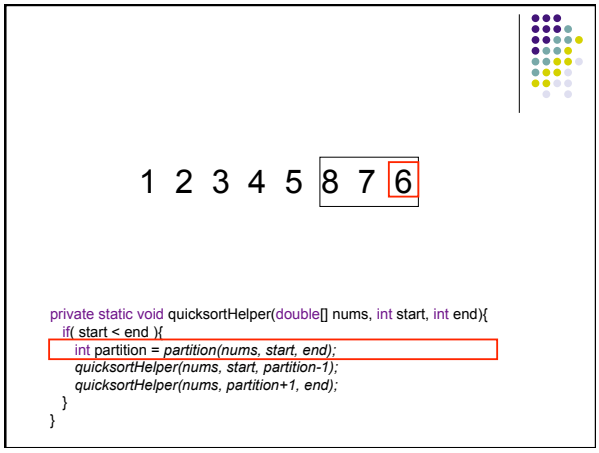
1 2 3 4 6 8 7 5

```java
private static void quicksortHelper(double[] nums, int start, int end){
  if( start < end ){
    int partition = partition(nums, start, end);
    quicksortHelper(nums, start, partition-1);
    quicksortHelper(nums, partition+1, end);
  }
}
```


1 2 3 4 6 8 7 5
private static void quicksortHelper(double[] nums, int start, int end){
if(start < end){
int partition = partition(nums, start, end);
quicksortHelper(nums, start, partition-1);
quicksortHelper(nums, partition+1, end);
}
}


1 2 3 4 5 8 7 6
What happens here?
private static void quicksortHelper(double[] nums, int start, int end){
if(start < end){
int partition = partition(nums, start, end);
quicksortHelper(nums, start, partition-1);
quicksortHelper(nums, partition+1, end);
}
}


1 2 3 4 5 8 7 6
private static void quicksortHelper(double[] nums, int start, int end){
if(start < end){
int partition = partition(nums, start, end);
quicksortHelper(nums, start, partition-1);
quicksortHelper(nums, partition+1, end);
}
}


1 2 3 4 5 8 7 6
private static void quicksortHelper(double[] nums, int start, int end){
if(start < end){
int partition = partition(nums, start, end);
quicksortHelper(nums, start, partition-1);
quicksortHelper(nums, partition+1, end);
}
}

1 2 3 4 5 6 7 8

```
private static void quicksortHelper(double[] nums, int start, int end){
  if( start < end ){
    int partition = partition(nums, start, end);
    quicksortHelper(nums, start, partition-1);
    quicksortHelper(nums, partition+1, end);
  }
}
```

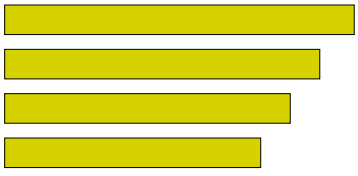
1 2 3 4 5 6 7 8

```
private static void quicksortHelper(double[] nums, int start, int end){
  if( start < end ){
    int partition = partition(nums, start, end);
    quicksortHelper(nums, start, partition-1);
    quicksortHelper(nums, partition+1, end);
  }
}
```

Running time of Quicksort?

Worst case?

Each call to Partition splits the array into an empty array and n-1 array



Quicksort: Worst case running time

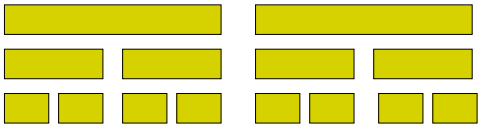
$n-1 + n-2 + n-3 + \ldots + 1 = O(n^2)$

When does this happen?

- sorted
- reverse sorted
- near sorted/reverse sorted

Quicksort best case?

Each call to Partition splits the array into two equal parts



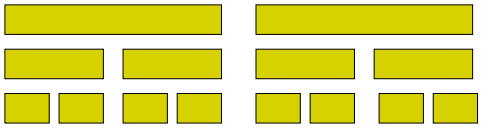
…

How much work is done at each "level", i.e. running time of a level?

O(n)

Quicksort best case?

Each call to Partition splits the array into two equal parts



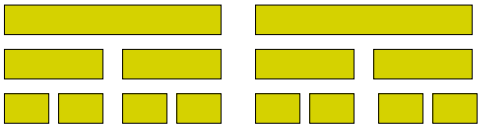
…

How many levels are there?

Similar to binary search, each call to Partition will throw away half the data until we're down to one element: $\log_2 n$ levels

Quicksort best case?

Each call to Partition splits the array into two equal parts



…

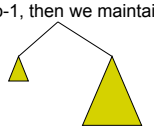
Overall runtime?

O(n log n)

Quicksort Average case?

Two intuitions

- As long as the Partition procedure always splits the array into some constant ratio between the left and the right, say L-to-R, e.g. 9-to-1, then we maintain O(n log n)



- As long as we only have a constant number of "bad" partitions intermixed with a "good partition" then we maintain O(n log n)

How can we avoid the worst case?

Inject randomness into the data

```
private static void randomizedPartition(double[] nums, int start, int end){
   int i = random(start, end);
   swap(nums, i, end);
   return partition = partition(nums, start, end);
}
```

Randomized quicksort is average case O(n log n)

What is the wost case running time of randomized Quicksort?

$O(n^2)$

We could still get very unlucky and pick “bad” partitions at every step