

Quicksort

David Kauchak
cs062
Spring 2011



```
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);

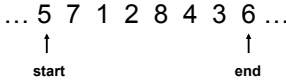
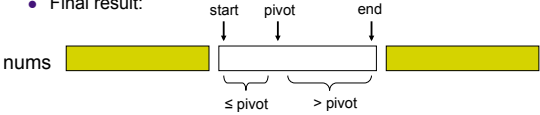
    return lessThanIndex+1;
}
```

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 `A[start...end-1]` into two sets, those \leq pivot and those $>$ pivot
- Operates in place
- Final result:



```
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;
}
```



Diagram showing the initial state of a partitioning algorithm. An array of numbers is shown: ... 5 7 1 2 8 4 3 6 The 'start' pointer is at index 5 (value 5) and the 'end' pointer is at index 3 (value 3). A blue arrow labeled 'lessThanIndex' points to the start position.

```

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;
}

```

Diagram showing the first iteration of the partitioning algorithm. The 'lessThanIndex' pointer has moved to index 6 (value 6), and the element at index 6 (value 6) is highlighted in red. The 'start' pointer remains at index 5 and the 'end' pointer remains at index 3.

```

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;
}

```

Diagram showing the second iteration of the partitioning algorithm. The 'lessThanIndex' pointer has moved to index 7 (value 7), and the element at index 7 (value 7) is highlighted in red. The 'start' pointer remains at index 5 and the 'end' pointer remains at index 3.

```

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;
}

```

Diagram showing the final state of the partitioning algorithm. The 'lessThanIndex' pointer has moved to index 8 (value 8), and the element at index 8 (value 8) is highlighted in red. The 'start' pointer remains at index 5 and the 'end' pointer remains at index 3.

```

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;
}

```

... 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;
}
    
```

... 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;
}
    
```

... 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;
}
    
```

... 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;
}
    
```

```

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;
}
    
```

What's happening?

≤ pivot > pivot unprocessed

```

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

```
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 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

```
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

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

- 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 merge sort: $\log_2 n$ 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 worst case running time of randomized Quicksort?



$O(n^2)$

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