what does this method do?

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

```
nums: start   pivot   end

≤ pivot   > pivot
```

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;
}
```java
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?
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;
}

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;
}

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);
    }
}
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);
    }
}

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);
    }
}

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);
    }
}

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);
    }
}
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);
    }
}
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);
    }
}
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

```java
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.