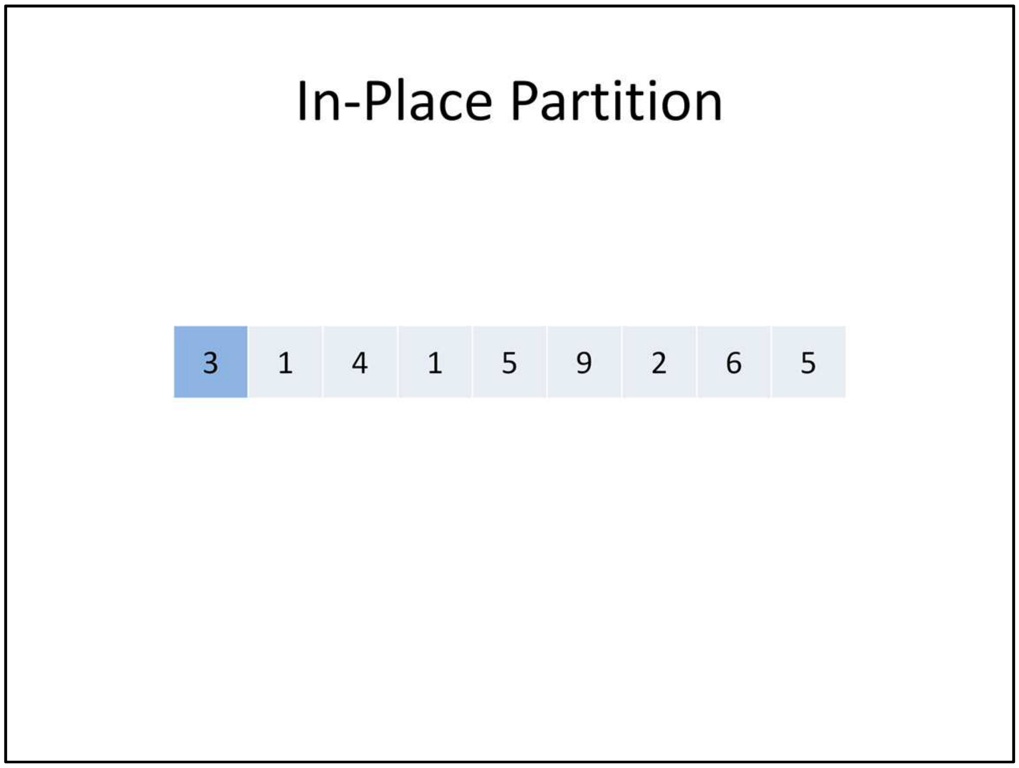
CS161L: Implementation of Algorithms

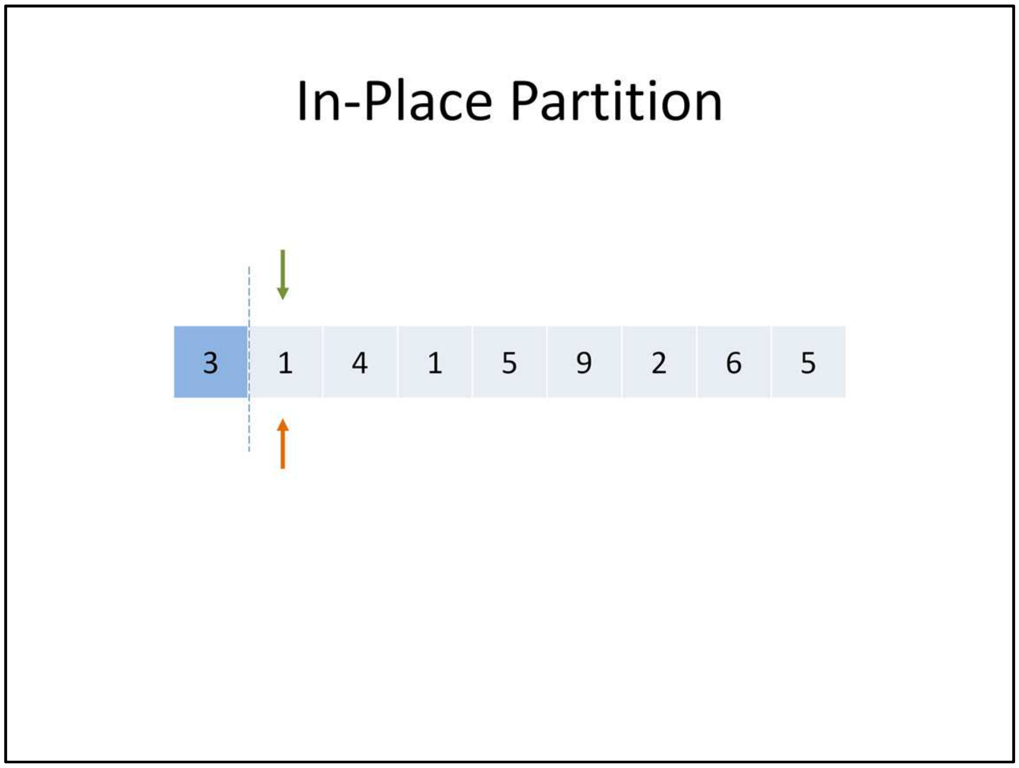
Friday, April 11, 2014



In-Place Partition

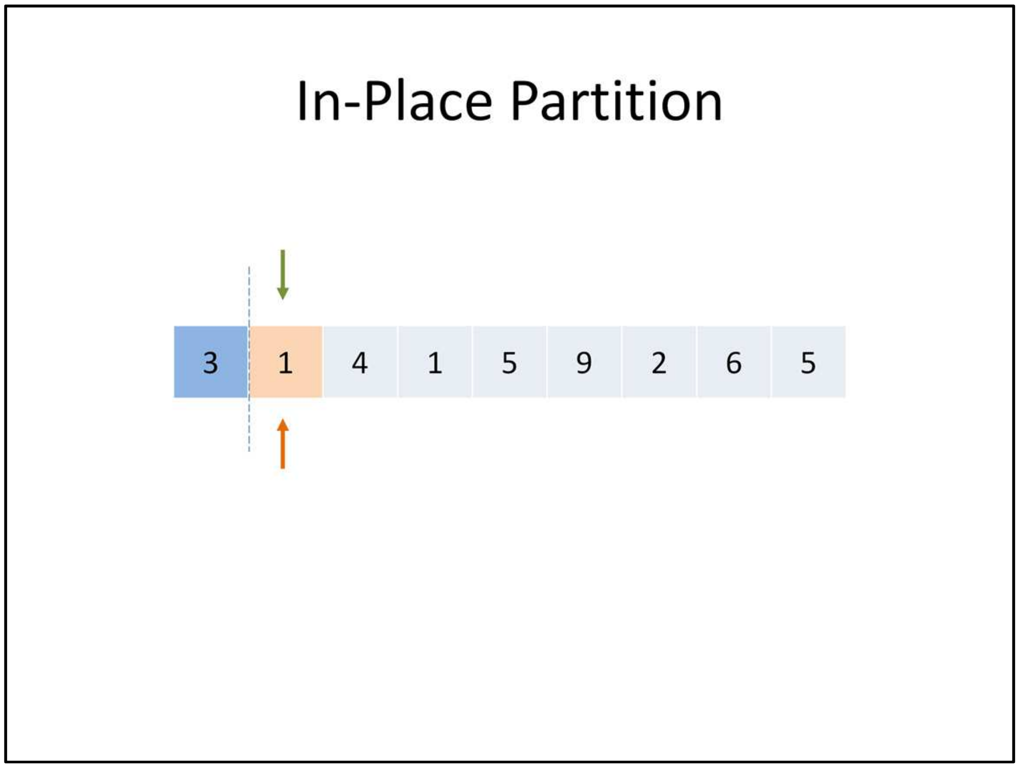
3	1	4	1	5	9	2	6	5

The first problem we're going to tackle is quicksort. Just as all of the complexity of mergesort is in the merge step, not in the recursion, all of the complexity of quicksort is found in the partition step, so let's go over how to perform an in-place partition. If you feel comfortable with the method presented in lecture on Wednesday, feel free to use that method; otherwise, here's another way that turns out to require much less code. Here, let's suppose that we want to partition using the first element, 3, as our pivot.



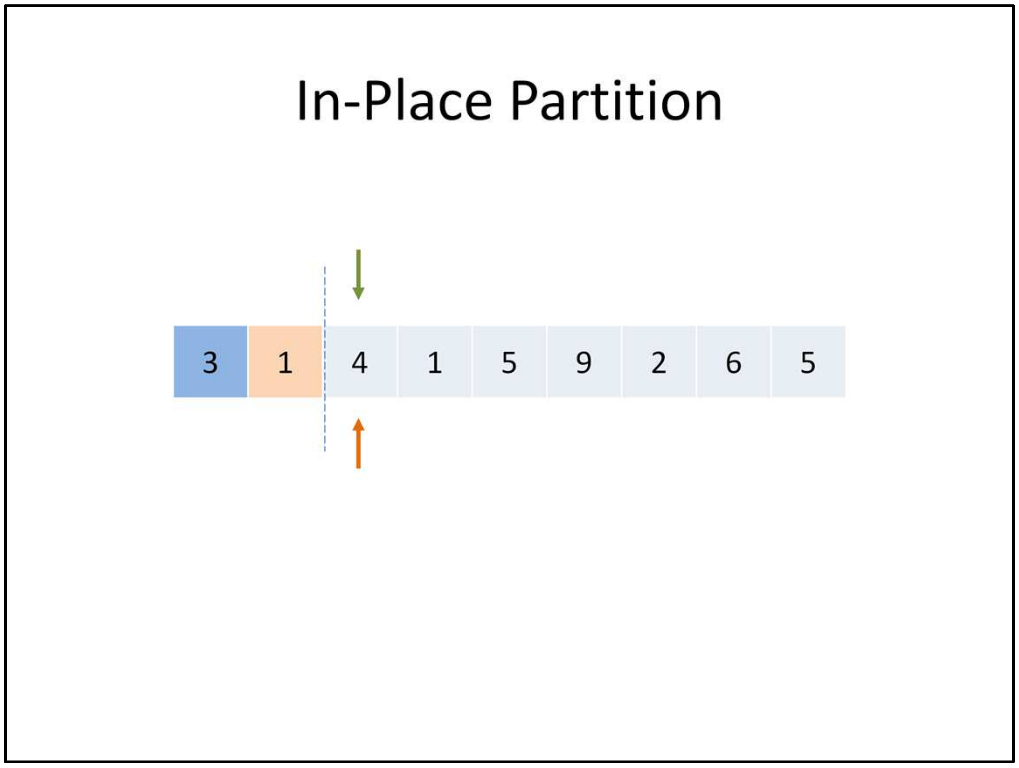
In-Place Partition

What we're going to do is we're going to build up an orange region of all the numbers smaller than the pivot, and a green region of all the numbers larger than the pivot, much like in the method presented in class. However, instead of building one from the left and one from the right, we're going to put them right next to each other. How does this work? We maintain two pointers, an orange pointer and a green pointer. The orange pointer will track how many numbers are smaller than our pivot so far, and the green pointer will track how many numbers we've considered so far.



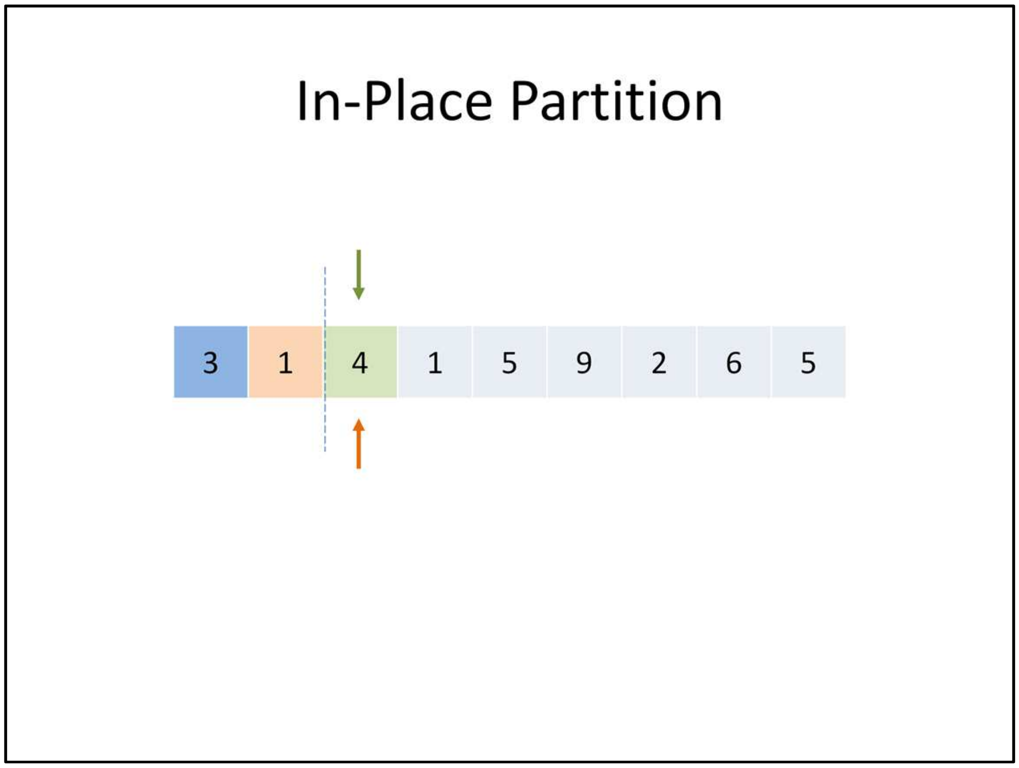
In-Place Partition

Let's look at the first number. It's smaller than the pivot, so what we're going to do is we're going to swap the numbers at the two pointers (which happens to be the same number, so nothing happens),



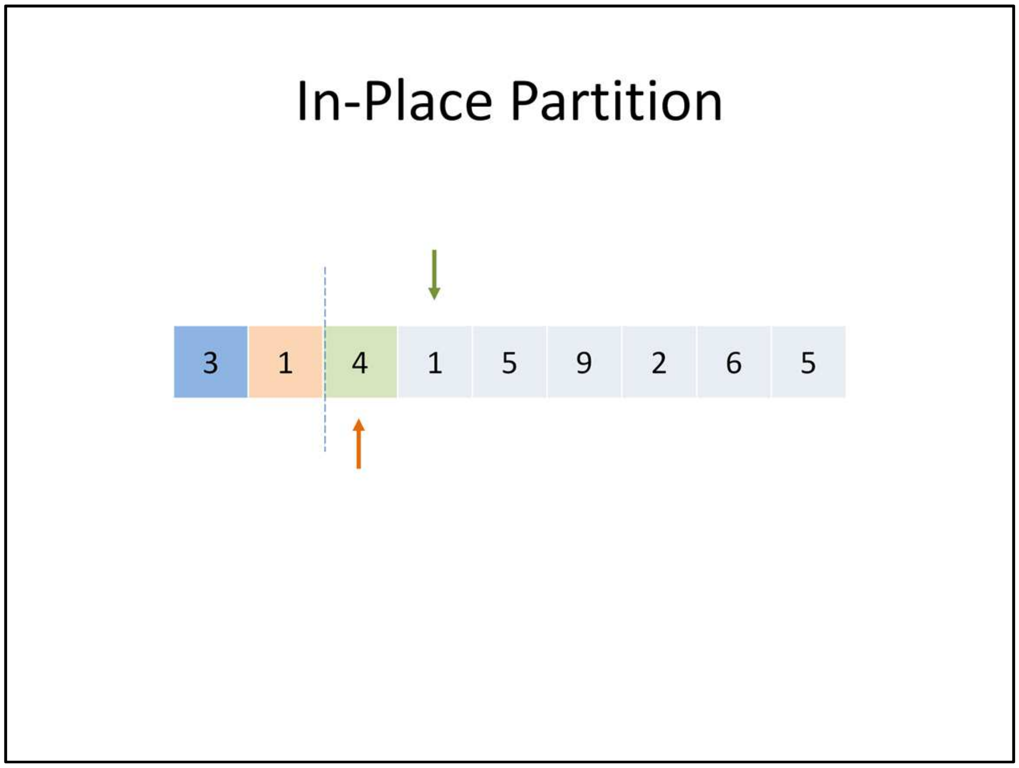
In-Place Partition

and then we increment both our orange pointer and our green pointer, since we got a new number that belongs to the orange region. Now we look at the next number,



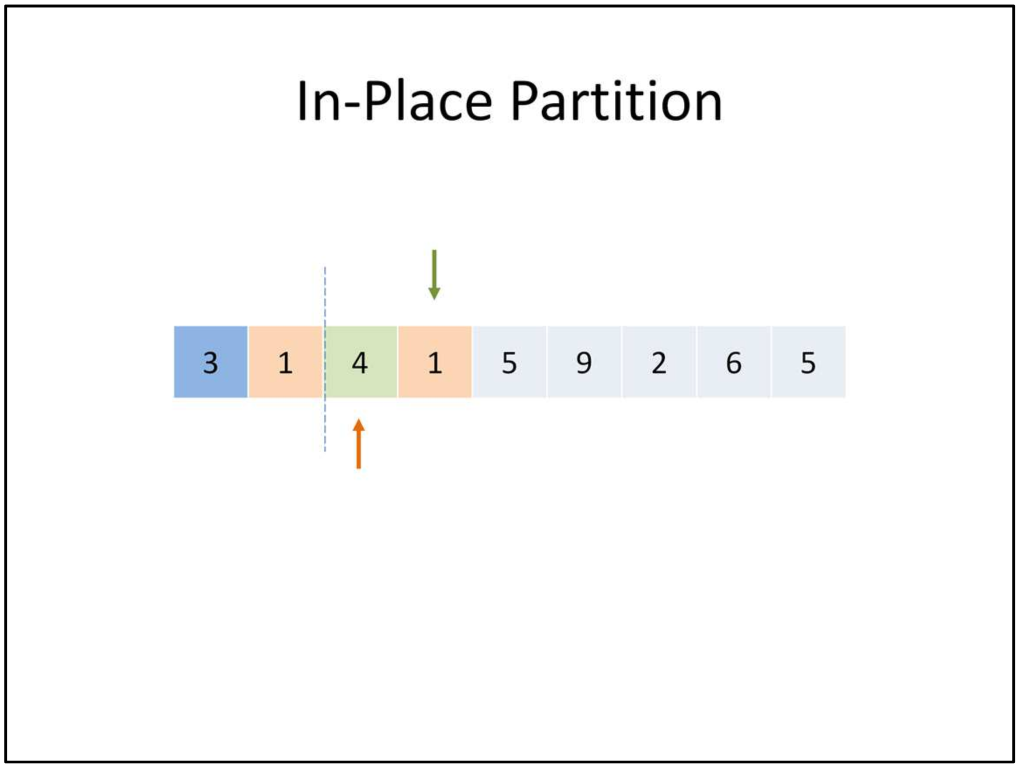
In-Place Partition

which is larger than our pivot, so it belongs to the green region, so no swapping has to occur;



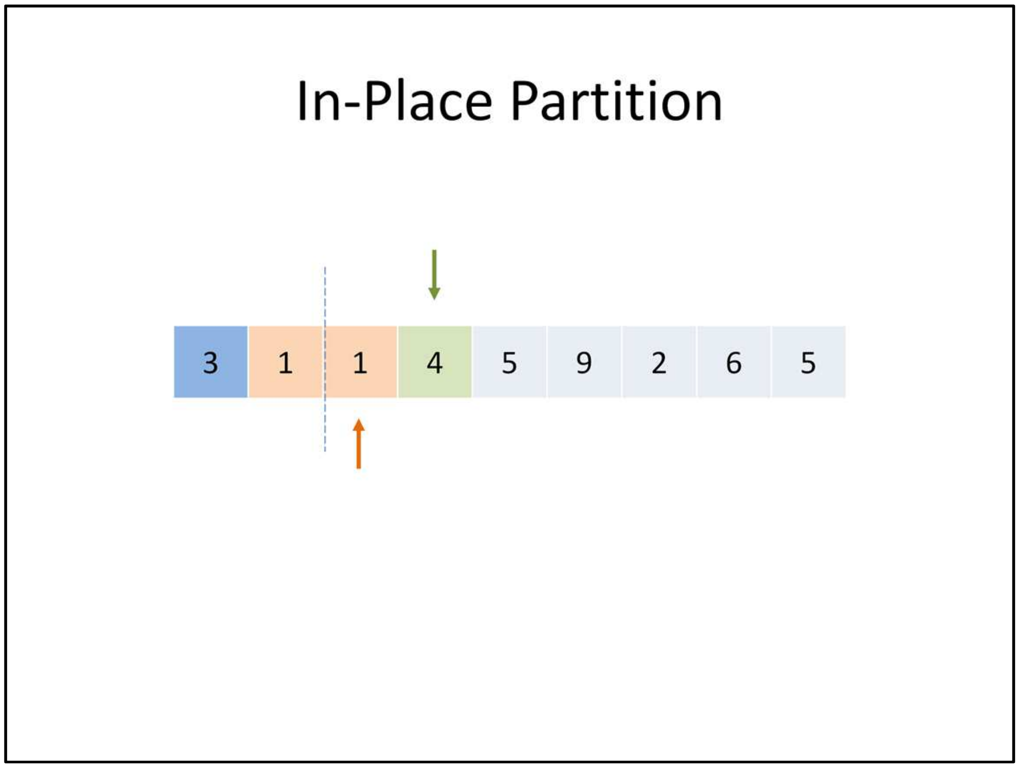
In-Place Partition

we just need to increment the green pointer.



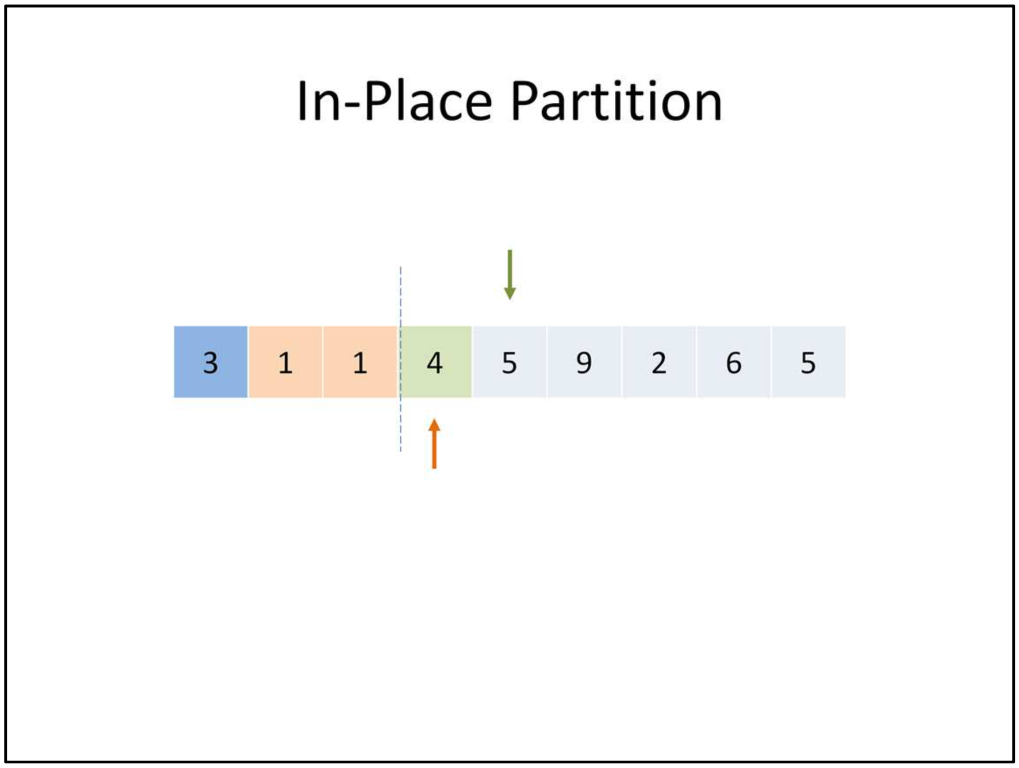
In-Place Partition

The next number is smaller, so it belongs to the orange region. In order to put it in the right place,

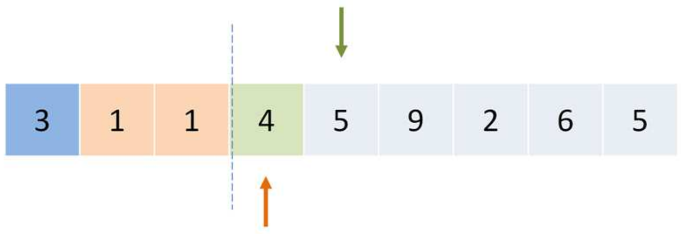


In-Place Partition

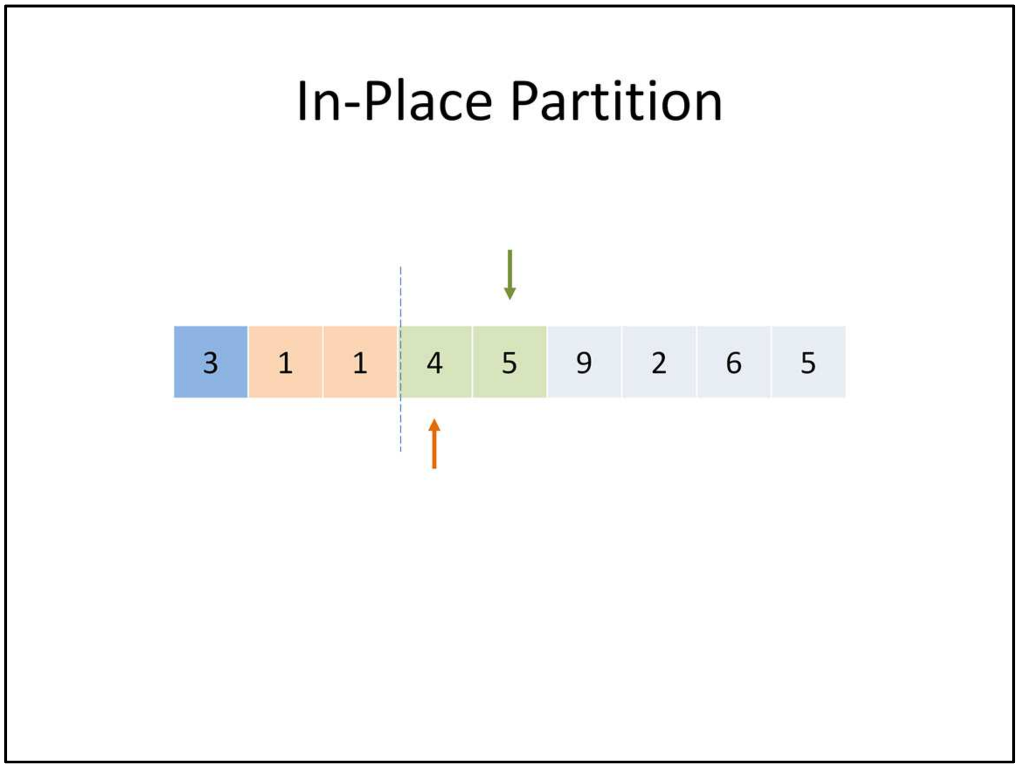
we swap the numbers that the two pointers are pointing to,



In-Place Partition

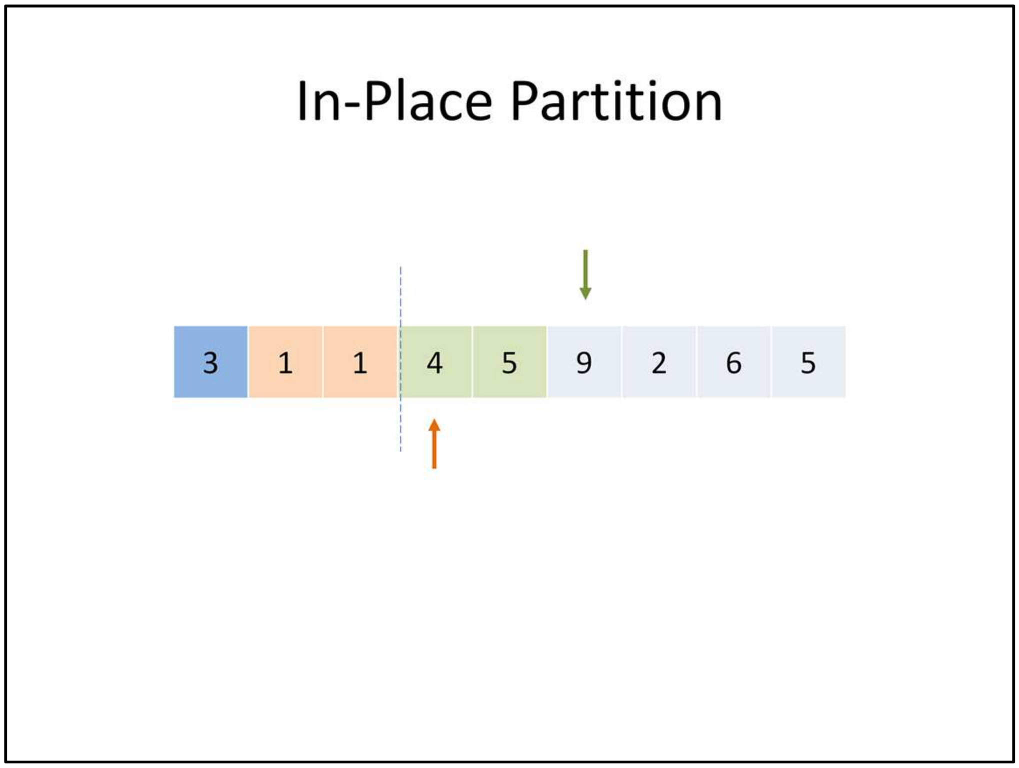


and then we increment both pointers.

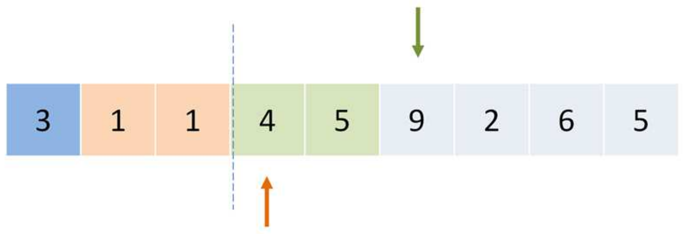


In-Place Partition

We repeat this process until we've gone through the whole array.

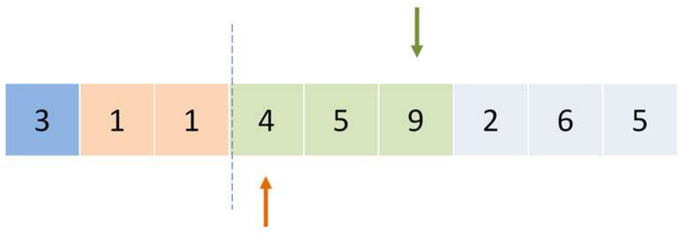


In-Place Partition

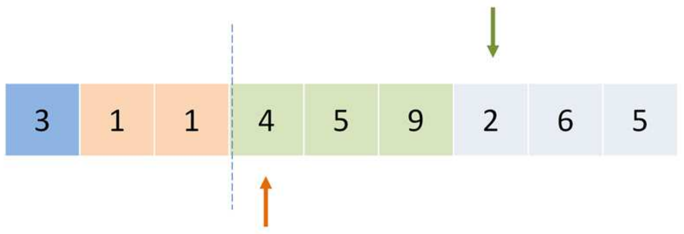


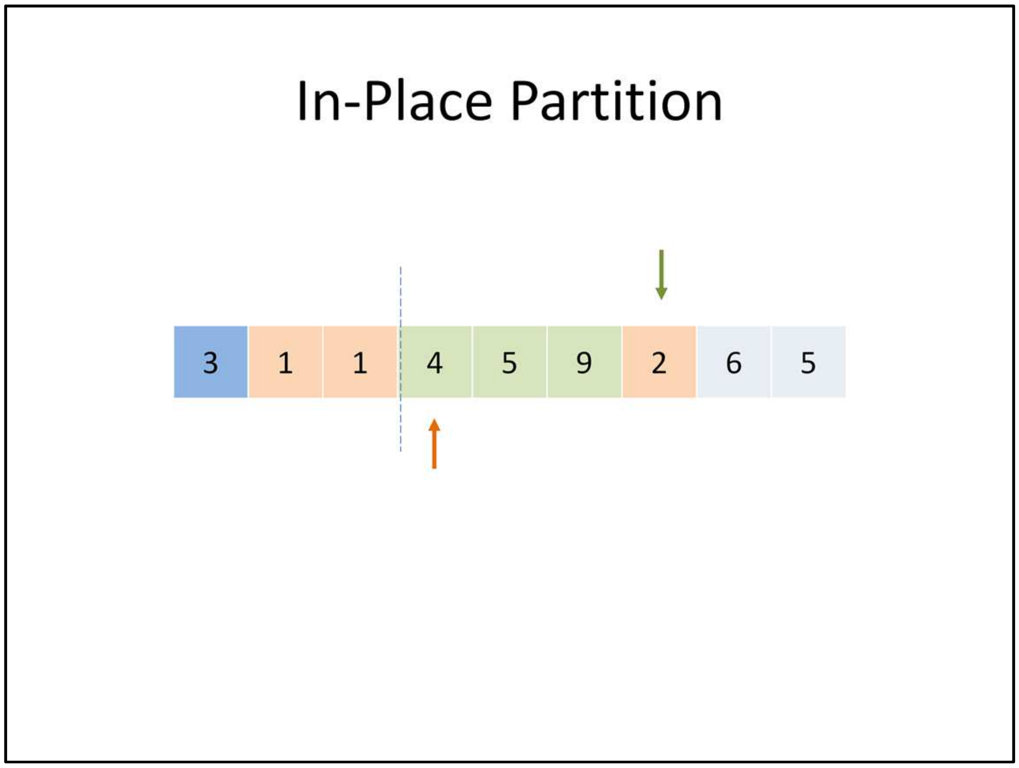
Every time we see a number bigger than our pivot, all we have to do is increment the green pointer.

In-Place Partition



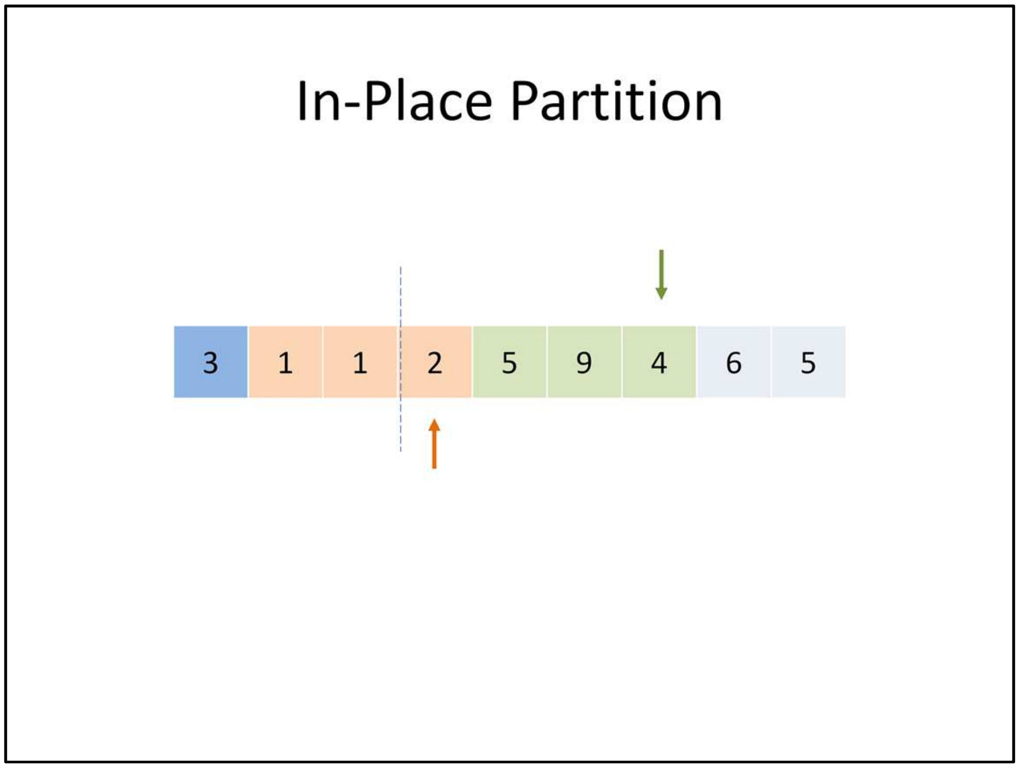
In-Place Partition



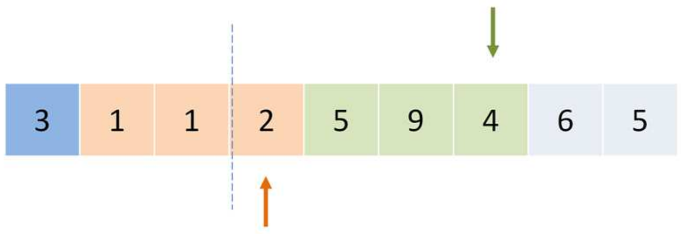


In-Place Partition

And every time we see a smaller number,

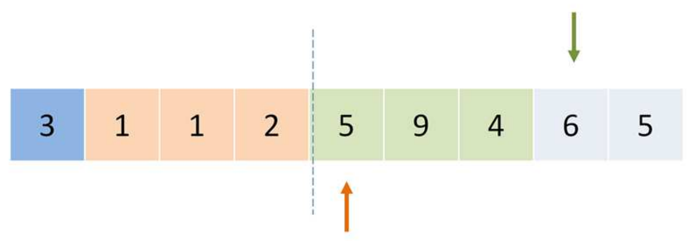


In-Place Partition



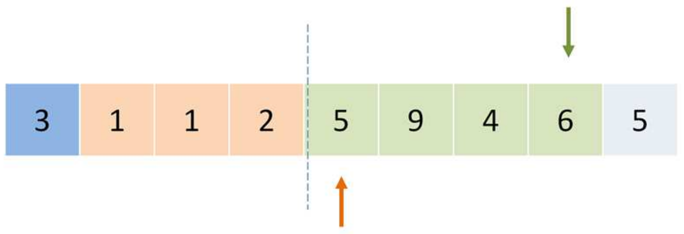
we swap the numbers at both pointers,

In-Place Partition

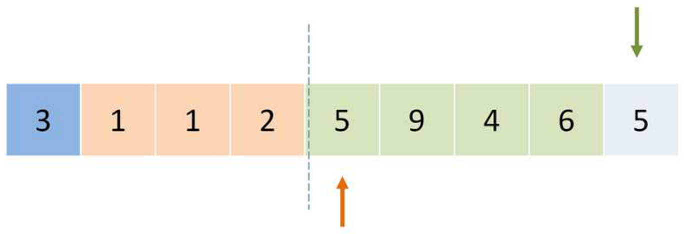


and then increment both pointers.

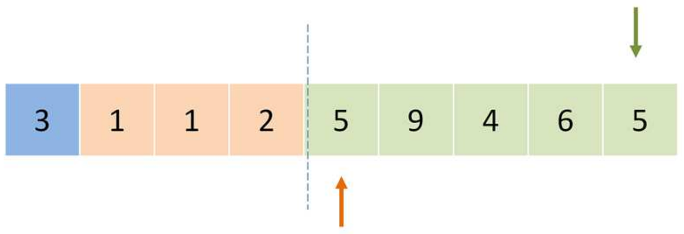
In-Place Partition

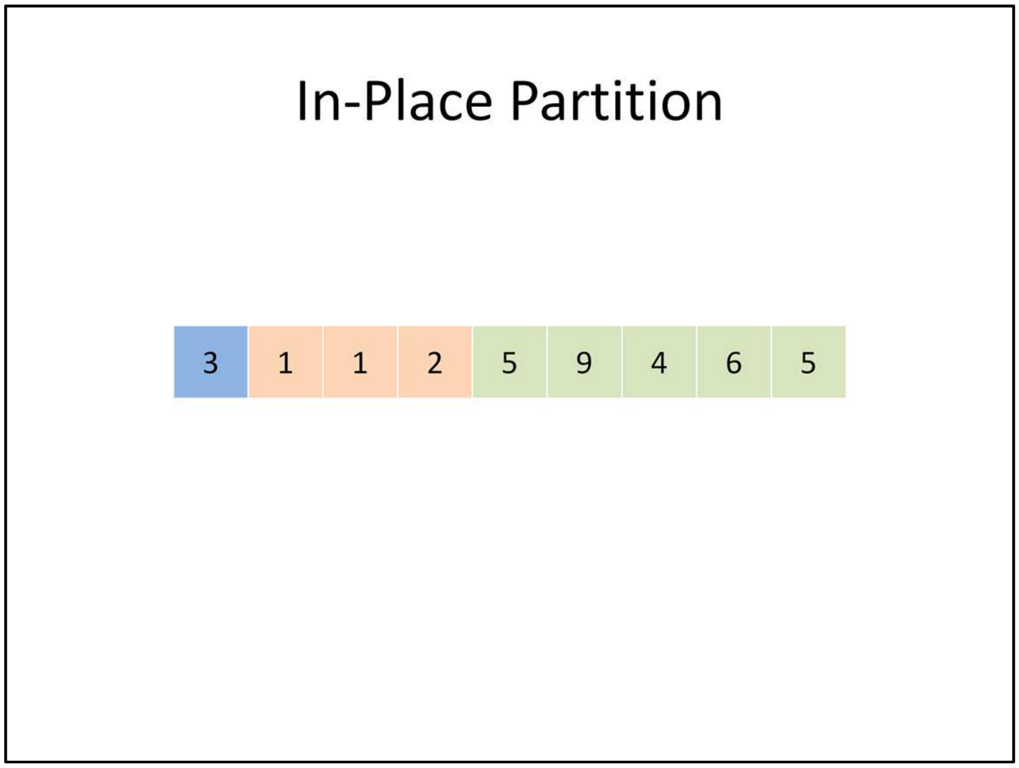


In-Place Partition



In-Place Partition

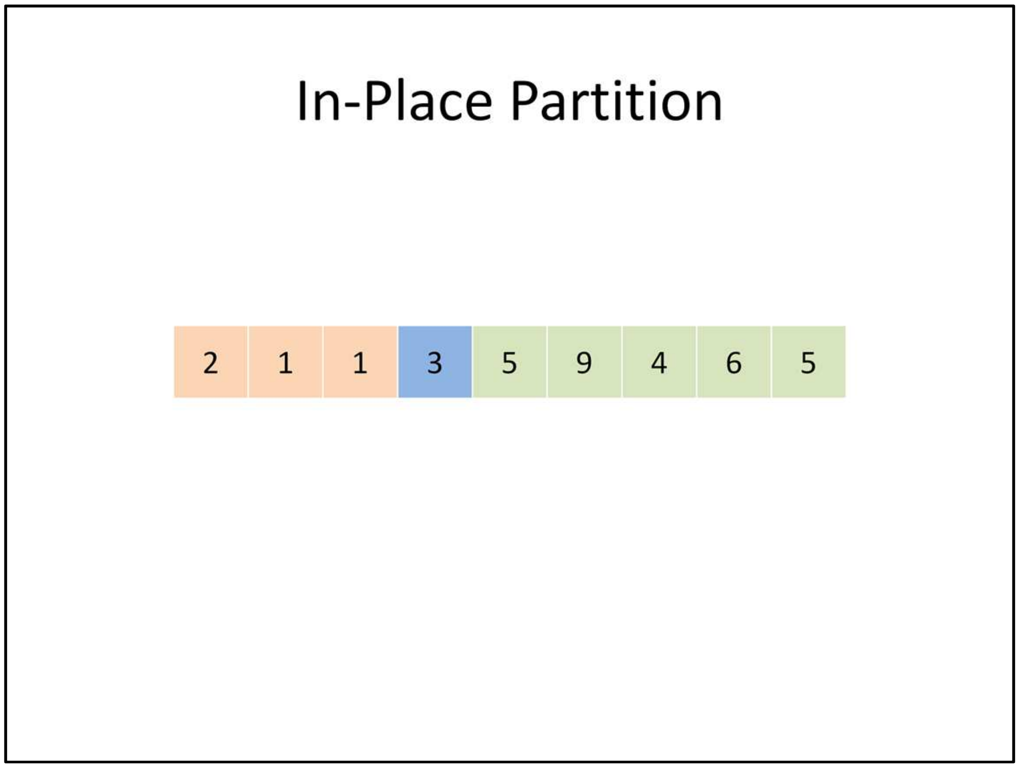




In-Place Partition

3	1	1	2	5	9	4	6	5

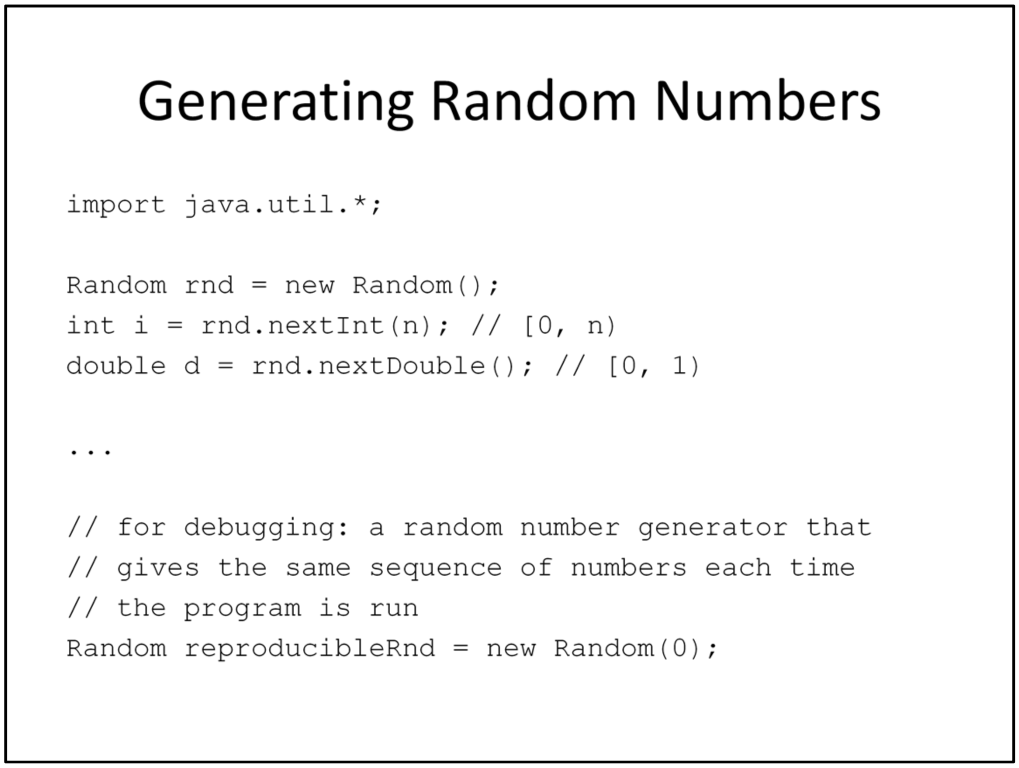
Once we're done using that algorithm, our array will look something like this. The first element will be our pivot, followed by a block of elements that are all NO larger than our pivot, followed by all the elements that ARE larger than our pivot. This result is just like what happened with the in-place partition discussed in lecture. Then, just like in lecture, all we need to do to put our pivot in the right place is to swap it with the last element of the orange block,



In-Place Partition

2	1	1	3	5	9	4	6	5

like so. Now, what if we want to choose a pivot besides the first element in the array? Well, before we run this partitioning algorithm, we can simply swap the pivot we want with the first element in the array.



Generating Random Numbers

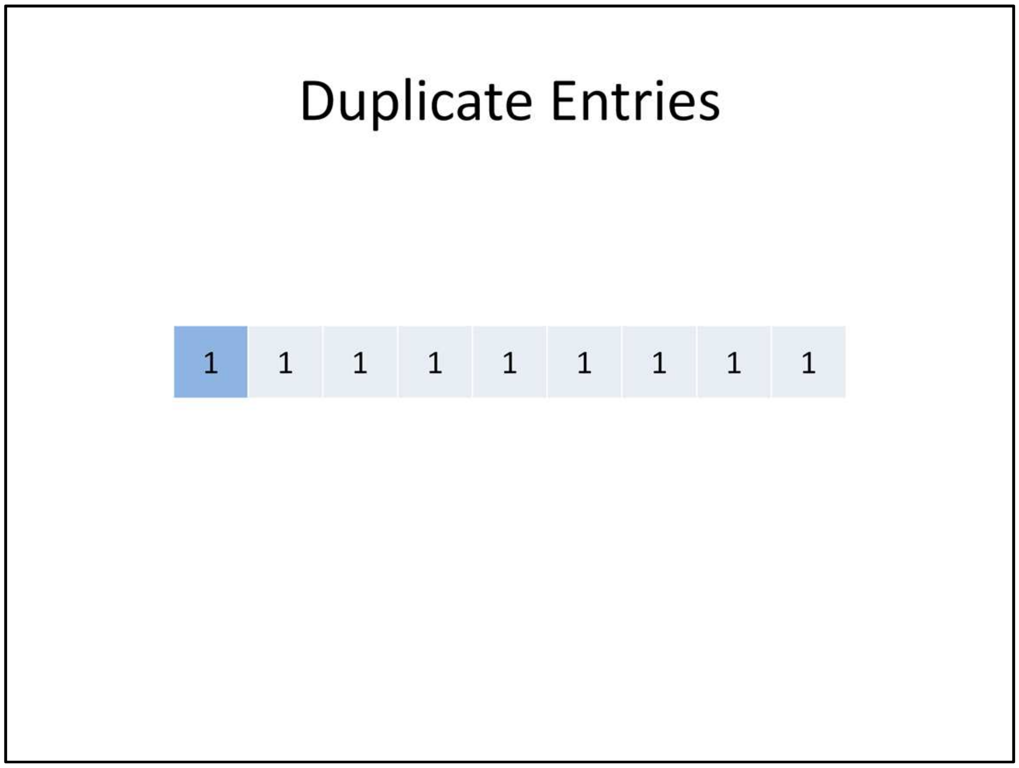
```
import java.util.*;

Random rnd = new Random();
int i = rnd.nextInt(n); // [0, n)
double d = rnd.nextDouble(); // [0, 1)

...

// for debugging: a random number generator that
// gives the same sequence of numbers each time
// the program is run
Random reproducibleRnd = new Random(0);
```

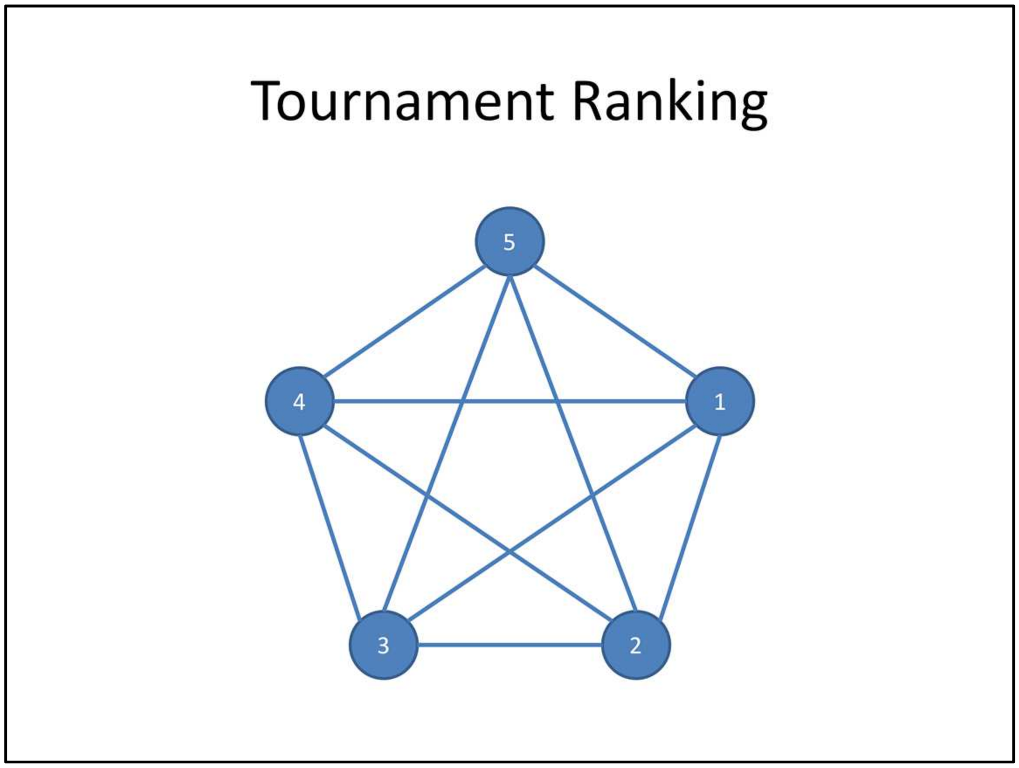
Now, you're going to be choosing your pivots at random. I've included this slide as a quick reference for how to generate a random integer between 0 and n, as well as how to generate a random floating point number between 0 and 1. Obviously you'll use the former for this problem, but the latter might come in handy in the future. For debugging purposes, you might want to fix your random number generator so it gives you the same sequence of pseudorandom numbers each time; once the code behaves as you expect, then you can remove the fixed seed.



Duplicate Entries

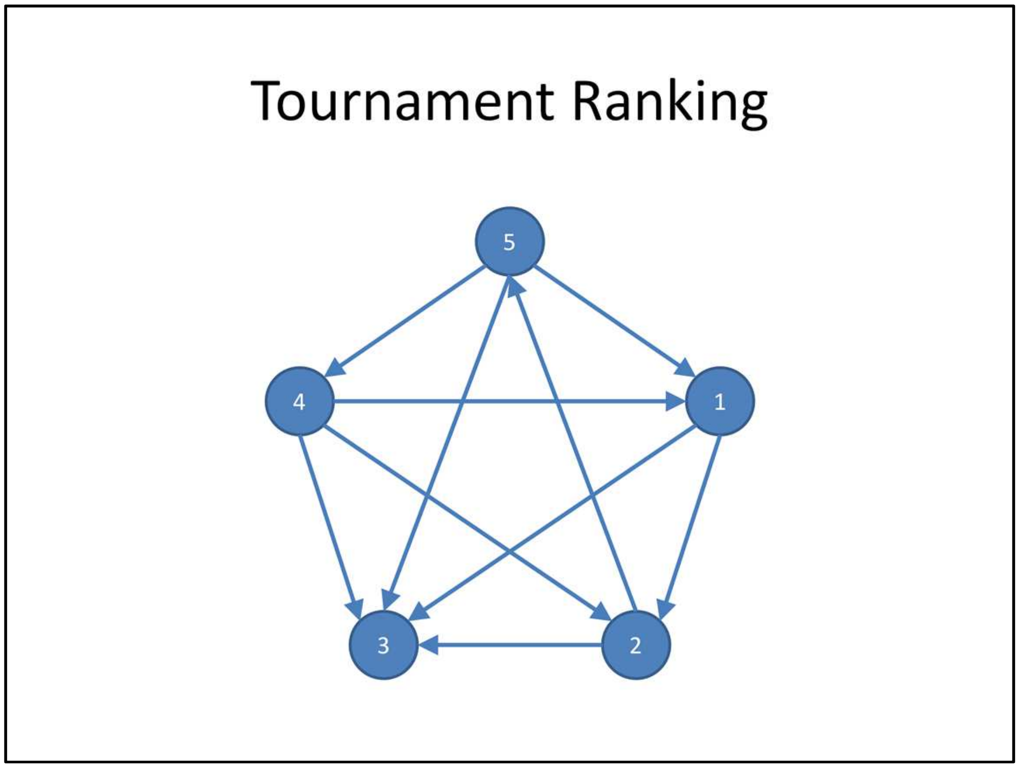
1	1	1	1	1	1	1	1	1

Now, the test data I've provided you with for Quicksort includes a test case that looks something like this. This case is actually kind of annoying for Quicksort to handle, because no matter what pivot you choose, you'll always end up with a bad split. That is, unless you modify the predicate you use in your partition subroutine. What you want to do in this case is whenever you run into entries that are identical to your pivot, you want to send roughly half of those entries to the left and the other half to the right. I'll let you figure out how to do this on your own. One trick that you might be tempted to try is to make a coin flip every time you come across an entry that matches your pivot. Keep in mind, though, that generating random numbers is expensive, so see if you can come up with something less computationally intensive.

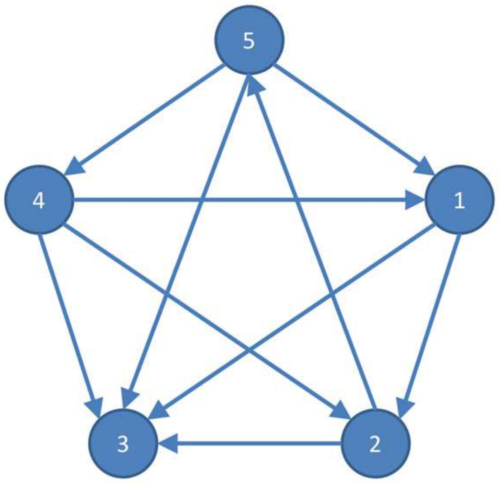


Tournament Ranking

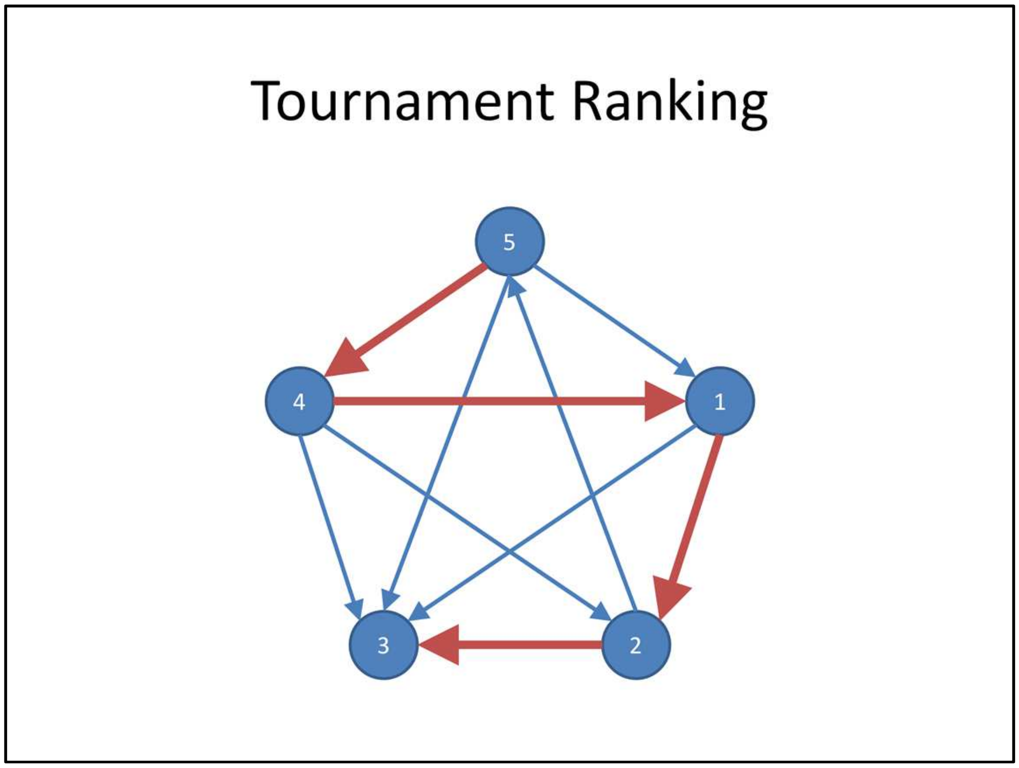
Now, on to the required problem. This problem has a bit of a story to it, so let's go over the story. You're running a round robin tournament, which is a tournament where everybody plays one game against everybody else.



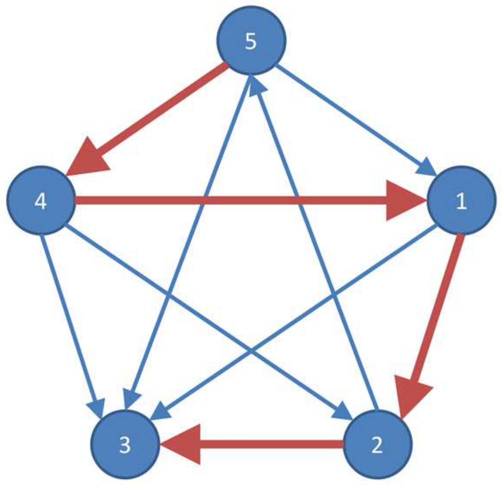
Tournament Ranking



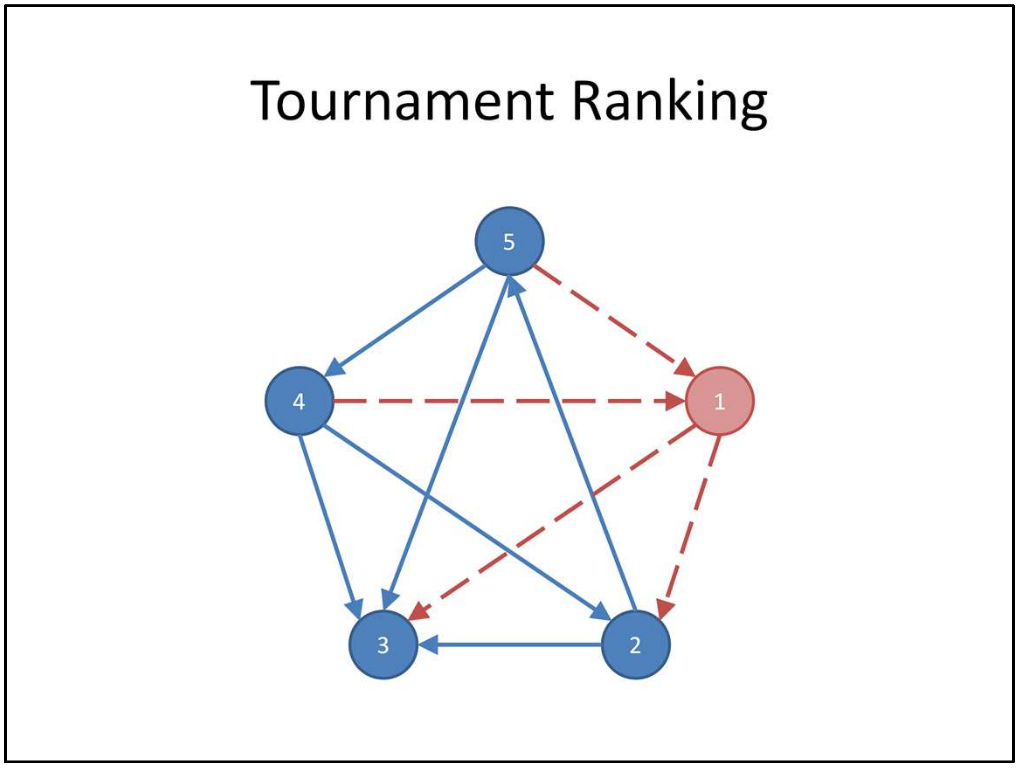
Each game has one winner and one loser, which we'll represent with an arrow pointing from the winner to the loser.



Tournament Ranking

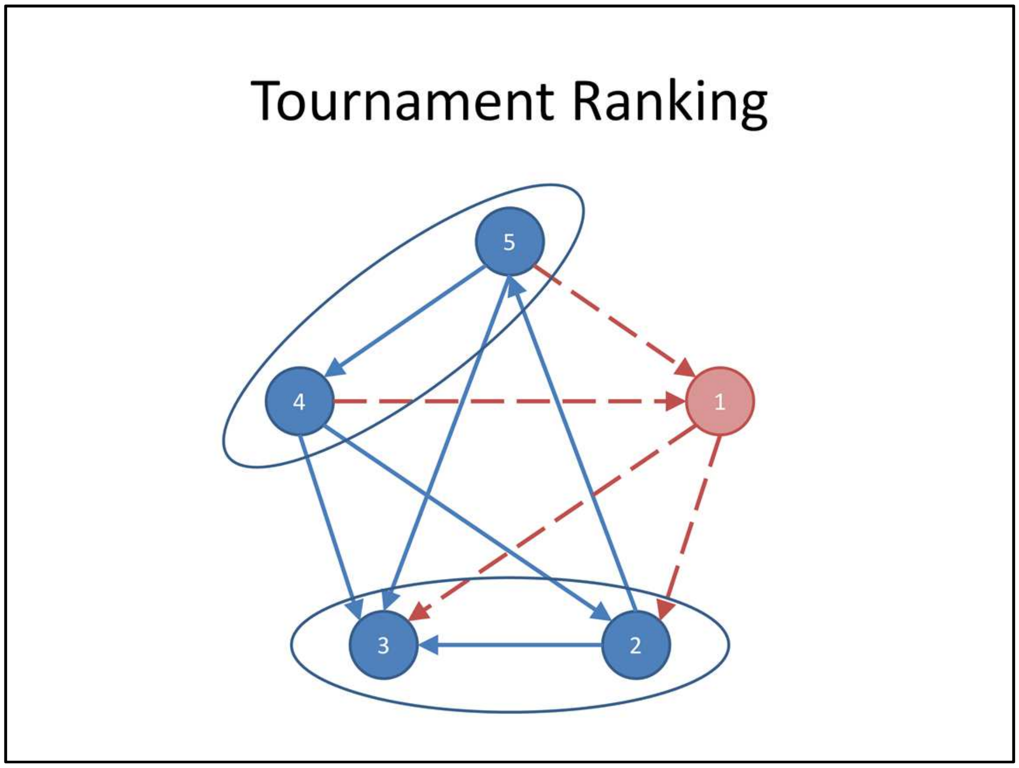


What we need to do in this problem is to find some ranking of all the players in the tournament such that the first place player beat the second place player, who beat the third place player, and so on and so forth.



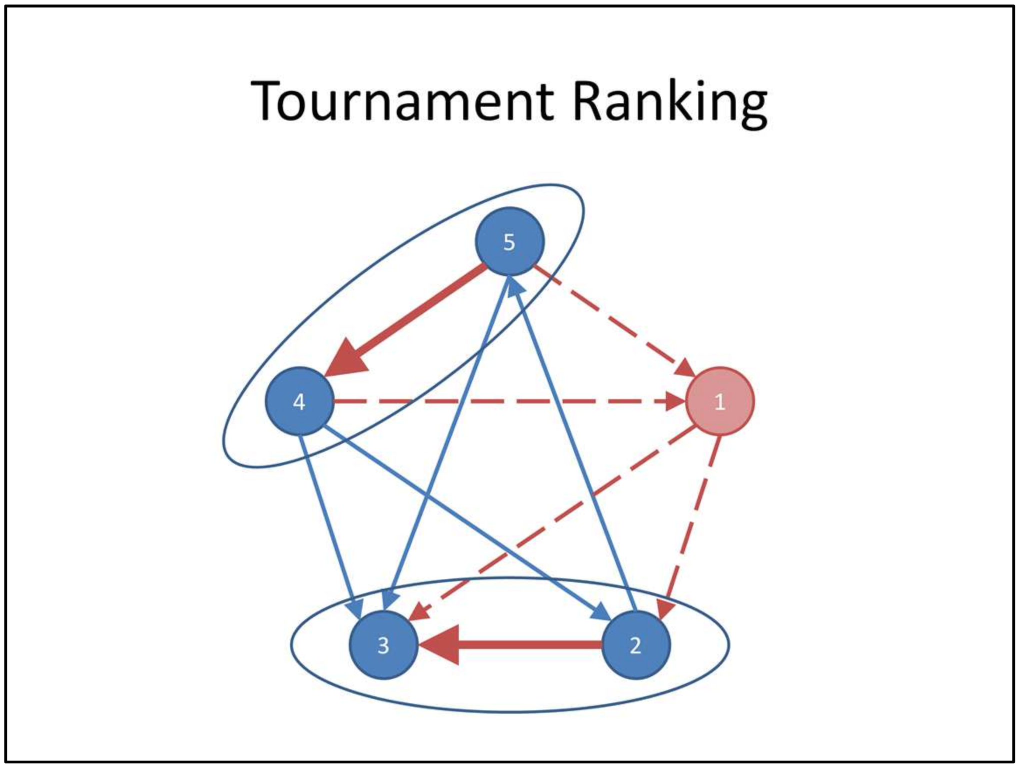
Tournament Ranking

How would we do that? How do we even know that such a ranking can exist? Well, let's focus on a single player, Player 1 in this case. Let's look at all the games she played. For each other player, she either won against or lost to that player.



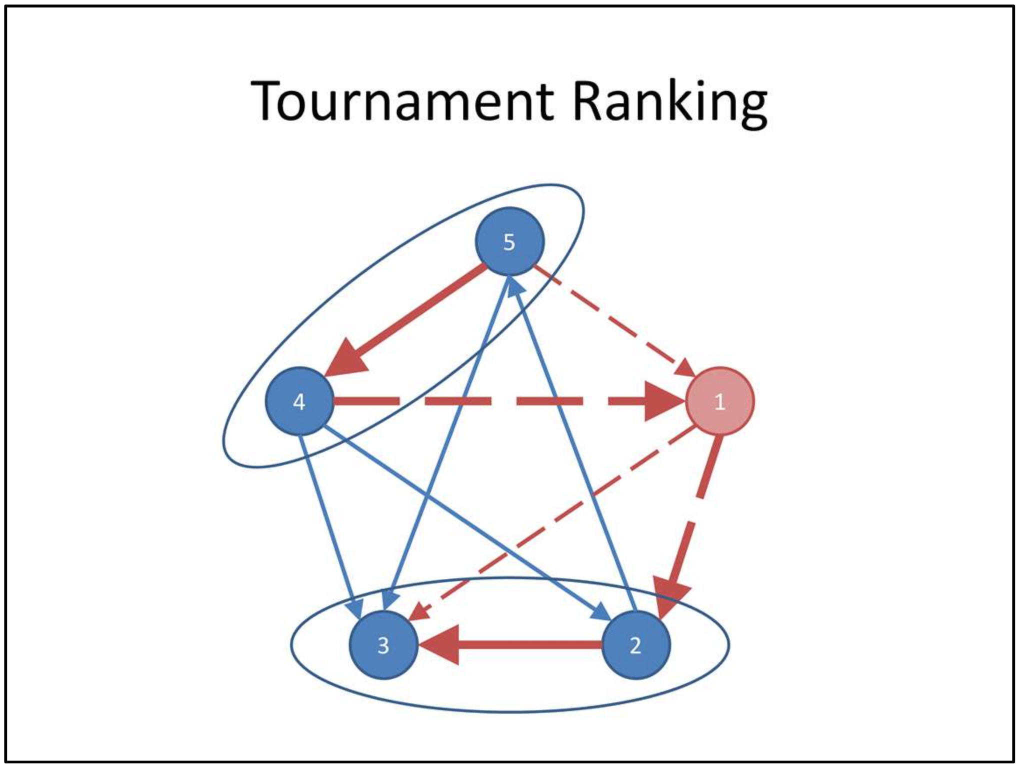
Tournament Ranking

So let's partition the other players into 2 sets: the set of players she won against, and the set of players she lost to.

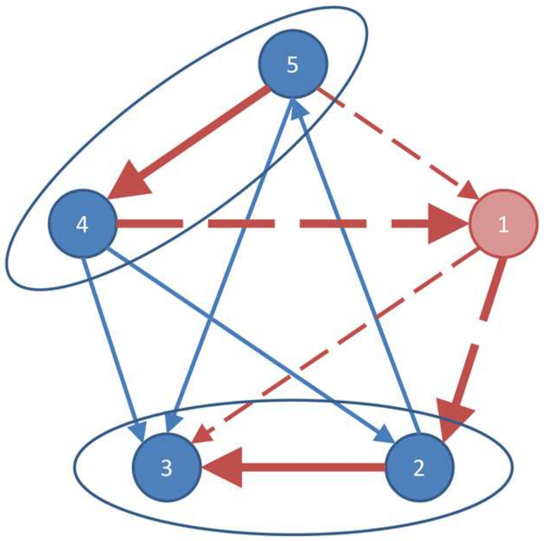


Tournament Ranking

Then let's recursively find rankings for the players within those two sets,



Tournament Ranking



and place Player 1 in her proper spot in this ranking. If you want to prove that this process always works, you can do so with induction. This apparently has been covered in an offering of 103 before, so we won't go into the details. Notice that this process is virtually identical to quicksort: you pick a pivot player, partition the remaining players into the players before and after, recursively rank those two sets, and place the pivot player in the correct spot. The only thing that actually changes is how you determine which side to put each player on in the partition. If you coded quicksort up properly, you'll have to change very little of your code to solve this problem. But that's not the only thing that this problem has in common with quicksort. In lecture, you were presented with various proofs that randomized quicksort runs in expected n lg n time, all of which are far more involved than simply coding up randomized quicksort. Just like quicksort, this algorithm is far easier to code up than it is to prove runtime bounds for. We won't go into what those bounds are and why, though, since it might come up in 161 proper.