CS125 : Introduction to Computer Science

Lecture Notes #38 and #39
Quicksort

©2005, 2003, 2002, 2000 Jason Zych

Lectures 38 and 39 : Quicksort

Quicksort is the best sorting algorithm known which is still *comparison-based*. A comparison-based sorting algorithm sorts using only comparisons on the elements. Other sorting algorithms which take advantage of other specific properties of your data (for example, if you are sorting integers you can use your data as array indices and count how many times each integer occurs) can be faster, but those algorithms are topics for a more advanced course than this. Quicksort relies only on being able to say whether a given element of a given type is less than, equal to, or greater than another element of the same type.

As with the previous two sorting algorithms, Quicksort will operate on subarrays, which means we will be passing the lower and upper array bounds to our recursive function in addition to the array itself.

The algorithm works as follows. Given an array, choose a *pivot* value. *Partition* the array so that you have on one side, all the elements less than the pivot, and on the other side, all the elements greater than the pivot, and in between them the pivot itself (which would by definition be in its correct final location). Then, recursively sort the two sides.

For example, if we had the array:

```
A[i]    2   6   8   1   7   4   3   5
-------------------------------------
  i     0   1   2   3   4   5   6   7
```

and let's suppose we chose 5 as our pivot value. (We will discuss in just a bit how we would go about choosing a pivot. For now, never mind that and assume we end up with 5 as the pivot.) With 5 selected as the pivot, what we want to is arrange the array – i.e. *partition* it – so that all values less than 5 are to the left of 5, and all values greater than 5 are to the right of 5.

```
A[i]    (elements 1-  |   | (elements 6-8
          4 take up   |   |  take up cells
         cells 0-3)   | 5 |  5-7)
-------------------------------------
  i     0   1   2   3   4   5   6   7
```

Above, the values 1-4, which are all less than 5, are to the left of 5. The values 6-8, which are all greater than 5, are to the right of five. This will force the pivot value, 5, into its correct location (i.e. when the array is fully sorted, value 5 will be in cell 4, but that is also where it *already is* right now).

If you had the array

```
A[i]    93  11  71  6   39  67  41  88  23  54
-----------------------------------------------
  i     0   1   2   3   4   5   6   7   8   9
```

and chose 39 as the pivot then the properly partitioned array would be:

```
     (elements 6, |    |  (elements 41, 54, 67,
A[i]  11, 23 here)| 39 |   71, 88 and 93 here)
-----------------------------------------------
  i     0   1   2   3   4   5   6   7   8   9
```

Again, everything less than the pivot ends up to the left of the pivot, everything greater than the pivot ends up to the right of the pivot, and the pivot ends up in its correct location (in the fully sorted array, 39 goes in cell 3, and that is where it is already). (As you can see, though we are calling the two sides "halves", the are not always exactly the same size. More on that later.)

Once you have partitioned the array, then you simply call Quicksort recursively on the two halves. If your array runs from lower bound `lo` to upper bound `hi`, and your pivot ends up at index `m`, then the left half consists of the cells `lo...m-1` and the right half consists of the cells `m+1...hi`. You don't need to include the pivot cell in either of the recursive calls because it is already placed correctly.

And, since we can assume the recursive call works correctly, we get the following:

```
Array after partitioning:

  A[i]    (elements 1-  |   | (elements 6-8
            4 take up   |   |  take up cells
           cells 0-3)   | 5 |  5-7)
  -------------------------------------
    i     0   1   2   3   4   5   6   7

Array after recursively sorting left half:

  A[i]                  |   | (elements 6-8
                        |   |  take up cells
          1   2   3   4 | 5 |  5-7)
  -------------------------------------
    i     0   1   2   3   4   5   6   7

Array after recursively sorting right half:

  A[i]                  |   |
                        |   |
          1   2   3   4 | 5 | 6   7   8
  -------------------------------------
    i     0   1   2   3   4   5   6   7
```

and after that second recursive call returns, we are done!

So, there are three things we still don't really know, and we are going to look at those next:

1. How is the pivot selected?

2. Once the pivot is selected, how exactly does the array get partitioned into "less than pivot" and "greater than pivot" sections? It's all well and good to say "all these values end up in this section", but that is a little vague. Precisely what algorithm gets used to do this partitioning work?

3. We said we should "recursively sort the halves". When are those halves too small? i.e., what is the base case?

Selecting the pivot

The key to making Quicksort run quickly is to get the two halves to actually be as close to *real* halves as possible. That is, if our pivot ends up at index `m`, we would like `m` to be the middle index of the array, and barring that, we would like `m` to be as close to the middle index of the array as possible.

The danger of allowing otherwise can be seen if we go back to our first example array, but pick the leftmost element instead of the rightmost element for the pivot.

```
A[i]    2   6   8   1   7   4   3   5
-------------------------------------
  i     0   1   2   3   4   5   6   7

        2 will be the pivot
```

In this case, after we partition, we will have all values less than 2 to the left of 2, and all values greater than 2 to the right of 2:

```
A[i]    1   2   (everything > 2 here)
-------------------------------------
  i     0   1   2   3   4   5   6   7

        2 (at index 1) is the pivot
```

As you can see, the left subarray has only one element, and the right subarray is only two cells less than the overall array. In fact, the *worst* situation is when we have the minimum or maximum as the pivot. In those situations, one of the two "halves" won't exist at all, and the other will be only one cell shorter than the overall array. Sorting two arrays half the size of the original array actually turns out to take less time than it takes to sort one array only one cell less in length than the original array, so we would prefer to have a pivot location close to the middle so that we can divide our array into two equal-size subarrays rather than one non-existent subarray and one subarray nearly the same size as the overall array.

The problem is that trying to find the exact median element will take far too long for our purposes (take our word for this; the proof of it is beyond the scope of this course). So, we can't go in search of the exact median. Instead, we have to do the best we can to get a decent pivot. But, it turns out this is okay. As long as we get a *decent pivot, most of the time*, we are okay. If sometimes, we get bad pivot, that is not a problem. What we want to avoid is getting a *bad pivot, almost all of the time*.

When we get a bad pivot almost all of the time, that is our worst-case behavior. In that case, Quicksort will take linear time to find the pivot and partition at each step (we'll see why shortly), and yet each step will result in trying to recursively sort an array only one cell less in length – meaning we have `n` nested recursive calls. This will result in quadratic behavior, and indeed, Quicksort runs in quadratic time in the worst case.

However, we generally get *good* pivots *most of the time*, and so we are also concerned with the average case. And in the average case, Quicksort runs in time proportional to $n \log n$ – that is, "linear *times* logarithmic". This is a larger function – and thus a longer (worse) running time – than linear functions, but smaller function – and thus a shorter (faster) running time – than quadratic functions.

So, how can we hit the average case as often as possible? How can we make sure our pivot selection is as good as we can make it?

- As already stated, actually searching for the ideal pivot is not practical, because it will make each recursive call take too much time. We want to partition quickly.

- We could randomly select an index and use that value as the pivot. It would make it unlikely that we would consistently select a pivot value very close to the minimum or very close to the maximum. But (good) random number generation can take a bit of time. It is still constant time, just not as low of a constant as you might think.

- If the array is arranged randomly, we could just grab the first or last element in the array (or subarray). This should be the equivalent of making a random selection of index, while avoiding the random number generation time. But, in the real world (as we mentioned when discussing the running time of InsertionSort) data is often partially sorted, so picking the first or last index is more likely to result in a bad pivot choice than would a purely random selection.

- One other alternative – the one we are going to use, by the way – involves reading three different values, and choosing their median. The three values we read are the ones in the second cell of the subarray, the middle cell of the subarray, and the last cell of the subarray. Even if the data is partially sorted, this is likely to give us a low element, a middle element, and a high element, and at any rate, it makes it even *less* likely that we get a bad pivot, since we'd now have to pick *two* values that were both close-to-the-end on the same side of the array – something which is even less likely than getting one close-to-the-end value. This process is known as *median-of-three pivot selection*, since we are selecting the pivot by taking the median of three different values.

  This method exchanges the random number generation time for three comparisons among the three values at the three cells. But, it tends to get us a better pivot in practice, which makes the algorithm run faster overall. So, this is the more appealing choice. It makes a consistently bad pivot selection unlikely enough that we can consider the worst case to be quite unlikely. And, since the average case – which is much more likely than the worst case – is the fastest known for a comparison-based sorting algorithm (there are other $n \log n$ sorting algorithms, but none with a constant as low as Quicksort's), that makes Quicksort a very appealing choice for many sorting problems.

Example 1:

```
A[i]    2   6   8   1   7   4   3   5
-----------------------------------
  i     0   1   2   3   4   5   6   7
```

In the above array, if we want to perform median-of-three pivot selection, we must (as stated above) look at the second cell, the middle cell, and the last cell. The second cell is `A[1]`, the last cell is `A[7]`, and the middle cell is `A[(0 + 7)/2] == A[3]`. (With arrays of even length, there are technically *two* middle elements, but typically the one on the left gets chosen because typically integer division rounds down (to 3, above) instead of up (to 4, above).)

The values in the second, middle, and last cells are 6, 1, and 5 respectively. And, the median of those values is 5. Therefore 5 would be the pivot value selected for the above array by the median-of-three pivot selection algorithm.

Example 2:

```
A[i]    93  11  71  6   39  67  41  88  23  54
-----------------------------------------------
  i     0   1   2   3   4   5   6   7   8   9
```

In example 2, the second, middle, and last cells are those at indices 1, 4, and 9 respectively. The values at those cells are 11, 39, and 54 respectively. The median of those three values is 39, and thus 39 would be the pivot value selected for the above array by the median-of-three pivot selection algorithm.

Partitioning the array

Once we have chosen our pivot value using median-of-three pivot selection, we then need to partition the array. The problem is, the pivot might be in the last cell, or it might be in the middle cell, or it might be in the second cell. Our partition algorithm is going to want to traverse down the array trying to rearrange values, and for the moment, that pivot is just getting in the way.

So, what we are going to do is get it out of the way, by moving the pivot to the first cell. No matter which of the three cells – second, middle, or last – the pivot was found in, we'll swap it with the value in the first cell and now we can safely traverse from the second cell through the last cell without running into the pivot.

Now, the question is, how can we partition the rest of this array – from the second cell through the last cell – into a "less than pivot portion" and a "greater than pivot" portion?

We will partition the array by inspecting the values in the array one by one. Values that are less than the pivot will go to the left side of the array, and values that are greater than the pivot will go to the right side of the array. However, we want to make sure we don't deal with a value once it has been inspected. For example, once we read a value and we know it belongs on the left side of the array, we would like to put it on the left side of the array and know we'll never run into it again for the rest of the partitioning process.

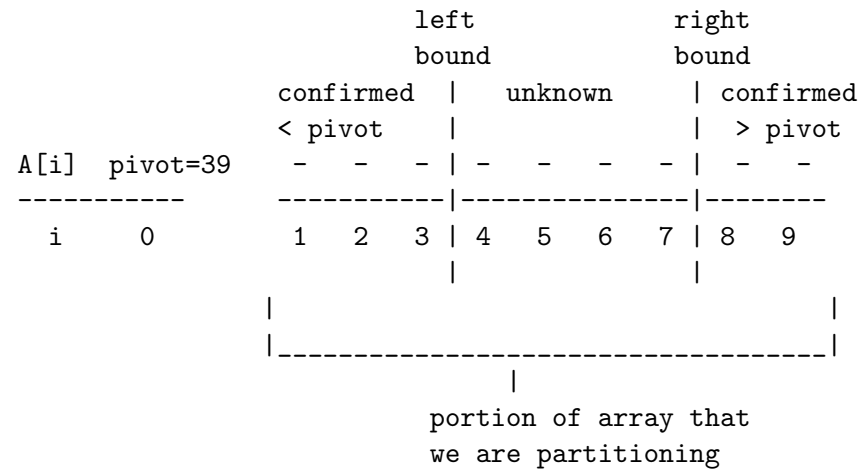
```
  A[i]  p=39  -   26  -   -   -   -   -   -   -
  ------------------------------------------------
    i     0   1   2   3   4   5   6   7   8   9
                  ^
    39 is pivot   |
                  |
We'd like to know we can move 26 to the ``left side''
and not deal with it any longer...which means we need
some knowledge of which cells constitute the ``left side''.
so we know where to move 26 to.
```

Likewise, if a value belongs on the right side of the array, we would like to put it on the right side of the array and know we'll never run into it again for the rest of the partitioning process.

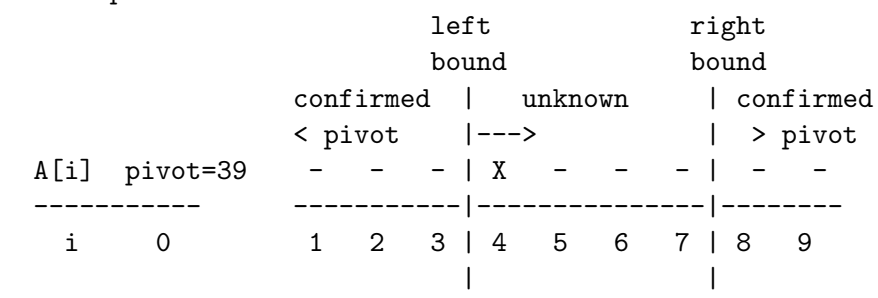
```
  A[i]  p=39  -   86  -   -   -   -   -   -   -
  ------------------------------------------------
    i     0   1   2   3   4   5   6   7   8   9
                  ^
    39 is pivot   |
                  |
We'd like to know we can move 86 to the ``right side''
and not deal with it any longer...which means we need
some knowledge of which cells constitute the ``right side''
so we know where to move 86 to.
```

The solution is to keep track of which cells constitute the "left side" and the "right side", just as we said we needed to do above. However, there is no need to keep track of all of those cells. Rather, we need only keep track of the rightmost value of the left side (anything to its left is also part of the left side and less than the pivot) and the leftmost value of the right side (anything to its right is also part of the right side and greater than the pivot). In other words, we will have "side

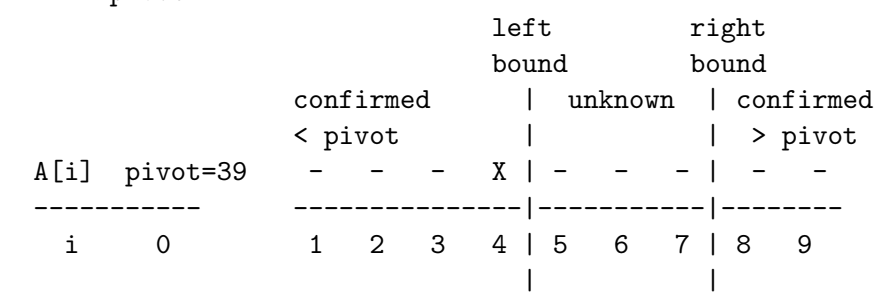
bounds", which will move inwards as we learn about the values in the "unknown" region. So, the next value we check will always be the value in the cell just to the right of the left bound.



As we discover a new value that belongs on the left, we'll move the left bound over to the right one spot to open up a space to the left of the left bound...



...and then put X on the left side of the left bound. Effectively, we just keep X exactly where it is but move the left bound over to the right one location. And now that the left bound has moved to the right one location, the less-than-pivot region is one cell larger and the unknown region is one cell smaller. And we next check cell 5, the cell which is *now* just to the right of the left bound.



Likewise, if we come across a value that *belongs* on the right side of the right bound, we want to *move* that value to the right side of the right bound. But, this means that we need to move the right bound over to the left one location, so that there is an additional space to the right of the right bound where we can store X.

```
If X > pivot
                            left              right
                            bound             bound
                    confirmed  |   unknown      | confirmed
                    < pivot    |            <---|  > pivot
  A[i]  pivot=39     -   -   - | X   -   -   Y  |  -   -
  -----------       -----------|---------------|--------
    i     0          1   2   3 | 4   5   6   7  | 8   9
                               |                |
```

But if we move the right bound over, that moves Y into the right side when we haven't even checked it yet. And then when we move X into that newly-added-to-the-right-side cell (i.e. cell 7), we will write over Y, which is also bad. The solution here is to *swap* X and Y before we move the right bound over one location, and then we will re-check cell 4 because we have moved a new value there – again, the cell we check is always the one just to the right of the left bound. Whenever we *don't* move the left bound, it is because we moved the right bound instead, and that means a new value got swapped into the cell (`A[4]` above) that we just inspected, and so it is okay to inspect that cell a second time (since it contains a new value).

```
If X > pivot
                            left           right
                            bound          bound
                    confirmed  |   unknown  |     confirmed
                    < pivot    |            |      > pivot
  A[i]  pivot=39     -   -   - | Y   -   -  | X    -   -
  -----------       -----------|------------|------------
    i     0          1   2   3 | 4   5   6  | 7   8   9
                               |            |
```

X is where Y was, Y is where X was, and X is on the right side of the right bound.
At this point, an example will help. Our rules, again, will be:

- Check cell `i`, which will always be the cell just to the right of the left bound.
- If the value we encounter at cell `i` is less than the pivot, move the left bound one position to the right (so cell `i` is now to the left of the left bound) and check cell `i+1`.
- If the value we encounter at cell `i` (call it X) is greater than the pivot, swap that value and the value just to the left of the right bound (call it Y). Now, X is just to the left of the right bound, and Y is just to the right of the left bound. Then, move the right bound one position to the left so that X is just to the *right* of the right bound, and then re-check cell `i`, since the swap has placed a new value (Y) there.

Example 1, where 5 was the pivot:

```
Original array:

  A[i]    2   6   8   1   7   4   3   5
  ---------------------------------------
    i     0   1   2   3   4   5   6   7

Swap pivot with first cell to get it out of
the way:

  A[i]    5   6   8   1   7   4   3   2
  ---------------------------------------
    i     0   1   2   3   4   5   6   7

Set up bounds marked with L and R:

            L                           R
            |                           |
  A[i]    5 | 6   8   1   7   4   3   2 |
  ----------|---------------------------|
    i     0 | 1   2   3   4   5   6   7 |
            |                           |

Inspect cell just to right of Left Bound, which
here is A[1]. 6 > pivot, so swap with
value to left of Right Bound and move Right
Bound over one location. Now 6 is on ‘‘right side’’.

            L                       R
            |                       |
  A[i]    5 | 2   8   1   7   4   3 | 6
  ----------|-----------------------|---
    i     0 | 1   2   3   4   5   6 | 7
            |                       |

Inspect cell just to right of Left Bound, which
here is A[1]. 2 < pivot, so here you just
move Left Bound over one location. Now, 2 is
on ‘‘ left side’’.

              L                   R
              |                   |
  A[i]    5   2 | 8   1   7   4   3 | 6
  --------------|-------------------|---
    i     0   1 | 2   3   4   5   6 | 7
              |                   |
```

Inspect cell just to right of Left Bound, which here is A[2]. 8 > pivot, so swap with value to left of Right Bound and move Right Bound over one location. Now 8 is on ‘‘right side’’.

```
               L               R
               |               |
  A[i]    5   2 | 3   1   7   4 | 8   6
  -------------|---------------|-------
    i     0   1 | 2   3   4   5 | 6   7
               |               |
```

Inspect cell just to right of Left Bound, which here is A[2]. 3 < pivot, so here you just move Left Bound over one location. Now, 3 is on ‘‘ left side’’.

```
                   L           R
                   |           |
  A[i]    5   2   3 | 1   7   4 | 8   6
  -----------------|-----------|-------
    i     0   1   2 | 3   4   5 | 6   7
                   |           |
```

Inspect cell just to right of Left Bound, which here is A[3]. 1 < pivot, so here you just move Left Bound over one location. Now, 1 is on ‘‘ left side’’.

```
                       L       R
                       |       |
  A[i]    5   2   3   1 | 7   4 | 8   6
  ---------------------|-------|-------
    i     0   1   2   3 | 4   5 | 6   7
                       |       |
```

```
Inspect cell just to right of Left Bound, which
here is A[4]. 7 > pivot, so swap with
value to left of Right Bound and move Right
Bound over one location. Now 7 is on ‘‘right side’’.
```

```
                      L   R
                      |   |
A[i]    5   2   3   1 | 4 | 7   8   6
----------------------|---|-----------
  i     0   1   2   3 | 4 | 5   6   7
                      |   |
```

As we continue with the partitioning, our “less than” bound will increase, and our “greater than” bound will decrease, thus closing in a smaller and smaller area until we have only one cell left.

So, now we come to the final idea behind our partitioning. What happens when we have only one cell left above? As far as the partitioning goes, there isn't really any problem anymore. Say that last cell is cell `i`. We know that, except for the pivot itself, everything in cells to the left of cell `i` is less than the pivot. And, we also know that everything to the right of cell `i` is greater than the pivot. So, we can picture our value at `i` with one side or the other, as appropriate:

```
Our leftover value is 4:
```

```
                      L   R
                      |   |
A[i]    5   2   3   1 | 4 | 7   8   6
----------------------|---|-----------
  i     0   1   2   3 | 4 | 5   6   7

           |  < pivot     |   > pivot |
           |______________|___________|
```

```
But if it had happened to be 9 instead, we'd
simply view it as being part of the right
side instead of as part of the left side:
```

```
                      L   R
                      |   |
A[i]    5   2   3   1 | 9 | 7   8   6
----------------------|---|-----------
  i     0   1   2   3 | 4 | 5   6   7

           |  < pivot |       > pivot |
           |__________|___ ___________|
```

So, clearly, the problem isn't partitioning this last element. We are fine as far as rearranging the elements goes. The problem is in re-placing the pivot at its correct location. Which of the two situations we have above doesn't matter from the standpoint of "finding sides", since the array is correct in that regard either way. But they are quite different from the standpoint of "placing the pivot".

```
Our leftover value is 4:

                       L   R
                       |   |
  A[i]    5   2   3   1 | 4 | 7   8   6
  ----------------------|---|-----------
    i     0   1   2   3 | 4 | 5   6   7

          |  |  < pivot     |   > pivot |
          |  |______________|___________|
          |
          |_________________^

   pivot goes in between the sides, so really,
   pivot should be at cell 4 and the ``< pivot''
   stuff should be in cells 0-3

But if A[4] had happened to be 9 instead, we'd
simply view it as being part of the right
side instead of as part of the left side:


                       L   R
                       |   |
  A[i]    5   2   3   1 | 9 | 7   8   6
  ----------------------|---|-----------
    i     0   1   2   3 | 4 | 5   6   7

          |  |  < pivot |       > pivot |
          |  |__________|___ ___________|
          |
          |_____________^

   pivot goes in between the sides, so really,
   pivot should be at cell 3 and the ``< pivot''
   stuff should be in cells 0-2.
```

In other words, the *size of the left side* determines where the pivot gets placed, since the pivot gets placed right after it. And, when we are partitioning, and we end up down to one last value, if that value belongs with the left side then it is sitting where the pivot should go, and if it belongs

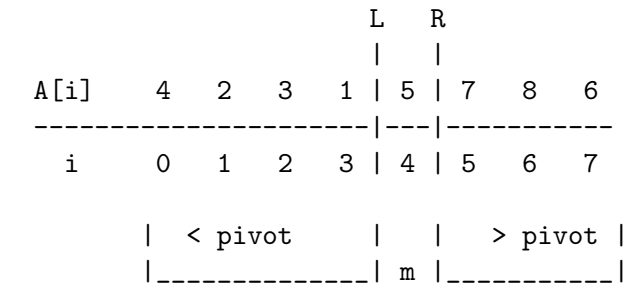
to the right side, then it is the first value in the right side and the pivot should be placed just to the left of it. But in either case, the last cell of the left side is sitting where the pivot should go, whether that last cell is the "last unknown cell" (the first example in the example pairs above), or the cell to its left (the second example in the example pairs above).

Why do things work this way? Why is the last cell of the left side always where the pivot should go? Well, this is because the left side needs to take up `c` cells, but that left side doesn't get to start at 0, since the pivot is in the way. So, instead, it starts at cell 1, and therefore instead of ending up at cell `c-1` (after taking up cells 0 through `c-1`), it ends up at cell `c` (after taking up cells 1 through `c`). Since the `c` values less than the pivot *should* be taking up cells 0 through `c-1`, leaving cell `c` free for the pivot, then since the rightmost value of the left side is taking up cell `c` instead of cell `c-1` due to the pivot forcing the left side to start at cell 1 instead of cell 0, the rightmost cell of the left side will always be sitting where the pivot should be.

What this means, however, is that we can complete our partition by swapping the pivot with the rightmost value of the left side, since the pivot is supposed to go in that rightmost cell of the left side, and the rightmost value of the left side *can* go where the pivot currently is, since it is less than the pivot and thus can end up anywhere on the left side after this partition operation is complete. Once we make that swap, then all the "less than pivot" elements are now to the left of the pivot, as we can see by expanding on our two examples above.

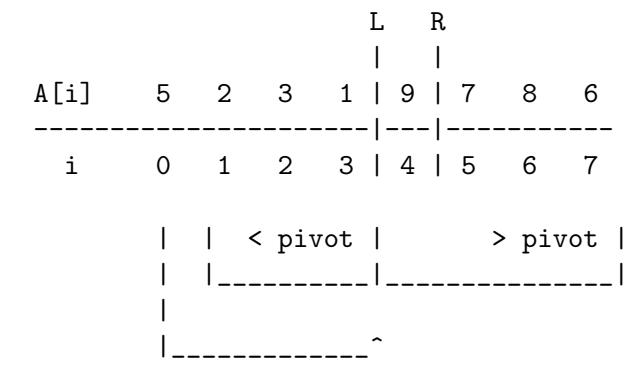
The first example, where the last value to be handled by the partition was less than the pivot:

```
Our leftover value is 4:

                       L   R
                       |   |
  A[i]    5   2   3   1 | 4 | 7   8   6
  ---------------------|---|-----------
    i     0   1   2   3 | 4 | 5   6   7

             |  < pivot     |   > pivot |
             |______________|___________|

   pivot goes in between the sides, so really,
   pivot should be at cell 4 and the ``< pivot''
   stuff should be in cells 0-3

   So swap pivot 5 (A[0] == 5) and our
   one remaining ``unknown'' value (A[4] == 4)
   to fix this problem, and we get the array as shown
   below:
```



And we should return the index m, which will
be the spot that our last ‘‘unknown’’ value was located in.

But if A[4] had happened to be 9 instead, we'd
simply view it as being part of the right
side instead of as part of the left side:



pivot goes in between the sides, so really,
pivot should be at cell 3 and the ‘‘< pivot’’
stuff should be in cells 1-2.

So swap pivot 5 (A[0] == 5) and the cell
just to the left of our last ‘‘unknown’’cell
(A[3] == 1) to fix this problem, and we get
the array as shown below.

```
                      L   R
                      |   |
  A[i]    1   2   3   5 | 9 | 7   8   6
  ----------------------|---|-----------
    i     0   1   2   3 | 4 | 5   6   7

         |  < pivot |   |    > pivot    |
         |__________| m |_______________|
```

And we should return the index m, which will be just to the left of the spot that our last ‘‘unknown’’ value was located in.

Please take note of one last thing – our “wall” or bounds idea is really just a set of recursive calls. Each call attempts to partition an array with bounds `lo` and `hi`, and when the “left bound moves” we are just calling on the subarray with bounds `lo+1` and `hi`, and likewise when the “right bound moves”, we are just calling on the subarray with bounds `lo` and `hi-1`.

Example 2, where 39 was the pivot:

```
  A[i]    93  11  71  6   39  67  41  88  23  54
  ------------------------------------------------
    i     0   1   2   3   4   5   6   7   8   9
```

Swap pivot to side and out of way.
Then Partition the range 1...9

```
            L                                   R
  A[i]    39  11  71  6   93  67  41  88  23  54
  ------------------------------------------------
    i     0   1   2   3   4   5   6   7   8   9
```

Partitioning range 1...9. A[1] is less than pivot so (left bound case) partition range 2...9

```
                L                               R
  A[i]    39  11  71  6   93  67  41  88  23  54
  ------------------------------------------------
    i     0   1   2   3   4   5   6   7   8   9
```

Partitioning range 2...9. A[2] is greater than pivot so (right bound case) swap with A[9] and then partition range 2...8

```
                L                           R
  A[i]    39  11  54  6   93  67  41  88  23  71
  ------------------------------------------------
    i     0   1   2   3   4   5   6   7   8   9
```

```
Partitioning range 2...8. A[2] is greater than
  pivot so (right bound case) swap with A[8] and
  then partition range 2...7
                L                       R
  A[i]    39  11  23  6   93  67  41  88  54  71
  ------------------------------------------------
    i     0   1   2   3   4   5   6   7   8   9

Partitioning range 2...7. A[2] is less than
  pivot so (left bound case) partition range 3...7.
                    L                 R
  A[i]    39  11  23  6   93  67  41  88  54  71
  ------------------------------------------------
    i     0   1   2   3   4   5   6   7   8   9

Partitioning range 3...7. A[3] is less than
  pivot so (left bound case) partition range 4...7.
                      L               R
  A[i]    39  11  23  6   93  67  41  88  54  71
  ------------------------------------------------
    i     0   1   2   3   4   5   6   7   8   9

Partitioning range 4...7. A[4] is greater than
  pivot so (right bound case) swap with A[7] and
  partition range 4...6.
                      L           R
  A[i]    39  11  23  6   88  67  41  93  54  71
  ------------------------------------------------
    i     0   1   2   3   4   5   6   7   8   9

Partitioning range 4...6. A[4] is greater than
  pivot so (right bound case) swap with A[6] and
  partition range 4...5.
                      L       R
  A[i]    39  11  23  6   41  67  88  93  54  71
  ------------------------------------------------
    i     0   1   2   3   4   5   6   7   8   9

Partitioning range 4...5. A[4] is greater than
  pivot so (right bound case) swap with A[5] and
  partition range 4...4.
                      L   R
  A[i]    39  11  23  6   67  41  88  93  54  71
  ------------------------------------------------
    i     0   1   2   3   4   5   6   7   8   9
```

```
Partitioning range 4...4. Low and high bounds are
  equal, so actual partitioning is completed, and it is
  simply a matter of knowing where pivot should go.
  Here, 67 is greater than pivot, so pivot should
  go to its left, into cell 3. Eventually we swap
  A[0] and A[3].
                       L   R
  A[i]    39  11  23  6   67  41  88  93  54  71
  ------------------------------------------------
    i     0   1   2   3   4   5   6   7   8   9

Done!

          < pivot    | m  |  > pivot
  A[i]    6   11  23| 39 |67  41  88  93  54  71
  ------------------|----|----------------------
    i     0   1   2 | 3  |4   5   6   7   8   9
```

Quicksort base cases

There are a number of ways we could handle the base case for Quicksort. The way we will use for this class is one of the most easily understandable, though other methods are likely to be a bit more efficient overall.

Our base case will be the situation where the array size is 0, 1, or 2. In other words, if the array size is 3 or more, then go ahead and perform pivot selection, partitioning, and recursive calls as we have already discussed. But, if the array size is 0, 1, or 2, then we will *not* do those things, but will instead handle those cases with non-recursive code specifically designed for those small-array-size cases.

What do we do in those three cases?

- If the array size is zero, then that means we have passed bounds that don't make sense. This is similar to what happened with `BinarySearch`. There, once `lo` was greater than `hi`, we knew that we had made a recursive call on an "empty" or "non-existent" subarray, and we could simply return without doing anything. The same rule applies here. There *is* no subarray, so just return.

- If the array size is 1, then there is nothing to sort – there is only one element, so it cannot possibly be "out of order". So, you only need to return.

- If the array size is 2, then either the 2-element array is in order, or else we swap the two elements to put it in order. So, we can check to make sure the lower-cell element is less than the higher-cell element, swap the two if it is not, and then return.

In any case where the array size is 3 or more, we will need more than one comparison to learn the correct sorted order, and so those situations will not be part of the base case, but will instead be handled by the recursive case. Essentially, we are saying that our base case is that the array *can* be sorted in one comparison or less, and that the recursive case is that the array *cannot* be sorted in one comparison or less.

Implementing `Quicksort` in Java

First, a few quick preliminaries...

1) We are going to need the `Swap` method again. We first looked at this method when discussing `SelectionSort`, and the `SelectionSort` notes go over the development of `Swap`. But as a reminder of what the code looks like, we'll reproduce it here:

```
public static void Swap(int[] A, int i, int j)
{
   int temp = A[i];
   A[i] = A[j];
   A[j] = temp;
}
```

2) Quicksort needs to recursively call itself on subarrays, just as some of the other recursive algorithms we've looked at have done. So, once again, if you'd prefer that the user just send in the array to be sorted and not worry about having to pass in array bounds to start the recursive calls with, then you can have a `public` Quicksort that accepts only the array, and then that that method can be implemented with the one-line call to the `private` version of Quicksort which *does* take array bounds as parameters and does all the real work of the algorithm.

```
public static void Quicksort(int[] A)
{
   Quicksort(A, 0, A.length - 1);
}
```

On to the Quicksort implementation...

The core of the `Quicksort` method is the recursive structure:

```
// choose pivot and partition the subarray, with pivot at cell m
// sort (lo...m - 1)
// sort (m + 1...hi)
```

Note that we have chosen to describe the pivot selection and partitioning as one procedure. It really is useful to view things this way, and so we are going to view them that way – by having a separate method that takes care of pivot selection and partitioning together. This method will be responsible for returning the index `m` where the pivot eventually ends up, but otherwise, all details of pivot selection and partitioning can be hidden away in that method call, which will make the recursive `Quicksort` method a lot cleaner.

```
private static void Quicksort(int[] A, int lo, int hi)
{
   int m = PartitionWrapper(A, lo, hi);
   Quicksort(A, lo, m - 1);
   Quicksort(A, m + 1, hi);
}
```

We need to pass in our array bounds to `PartitionWrapper`, because the partitioning algorithm as we described it way above needs the actual low and high bounds of the subarray in order to work. But, we don't need to care about the details of `PartitionWrapper` in this `Quicksort` method. All we need is the index `m` where the pivot is located, so we know which subarray constitutes the "left half" and which subarray constitutes the "right half".

Before we move on, we need to add the base case to the recursive case above. We said our base case would be that the subarray size was 2 or less. The cases for subarrays of size 2, 1, and 0 are as follows:

```
size == 2:   hi == lo + 1  (lo is just to left of hi)
size == 1:   hi == lo      (lo and hi are same cell)
                in which case ---> hi < lo + 1
size == 0:   hi == lo - 1  (lo is just to right of hi)
                in which case ---> hi < lo
                in which case ---> hi < lo + 1
```

So, it appears we can say that if `hi <= lo + 1`, then the size of the subarray is 2 or less, and by contrast, if `hi > lo + 1`, then we have 3 or more elements in our subarray.

```
private static void Quicksort(int[] A, int lo, int hi)
{
   if (hi > lo + 1)  // size is 3 or more
   {
      int m = PartitionWrapper(A, lo, hi);
      Quicksort(A, lo, m - 1);
      Quicksort(A, m + 1, hi);
   }
   else // size is 0, 1, or 2
      // handle cases
}
```

The last thing we have to handle here is what to actually *do* for our base cases. For 0 and 1, we don't need to do anything. And, if we have the 2-element case and the lower element is already less than the higer element, we still don't need to do anything. The only time we need to do anything is when we have the two-element case and the lower element is greater than the higher element, in which case we swap them and then we are done.

```
private static void Quicksort(int[] A, int lo, int hi)
{
   if (hi > lo + 1)  // size is 3 or more
   {
      int m = PartitionWrapper(A, lo, hi);
      Quicksort(A, lo, m - 1);
      Quicksort(A, m + 1, hi);
   }
   else // size is 0, 1, or 2
      if ((hi == lo + 1) && (A[lo] > A[hi]))
         Swap(A, lo, hi);
}
```

So, above is the `Quicksort` method. Now we only need to write `PartitionWrapper` and we are done.

Implementing `PartitionWrapper`

We start with the following framework, and will take care of the steps one at a time.

```
private static int PartitionWrapper(int[] A, int lo, int hi)
{
   // 1) Choose pivot.
   // 2) Swap pivot and first element.
   // 3) Partition the remainder of the subarray.
   // 4) Put pivot in its appropriate spot between the halves.
   // 5) Return index where pivot is currently stored.
}
```

1) Choosing the pivot: As stated above, our pivot selection will be done via the median-of-three pivot selection algorithm. We'll pull that selection into a separate method, just to organize things a little bit better.

```
private static int PartitionWrapper(int[] A, int lo, int hi)
{
   int currentPivotLocation =
        MedianOfThree(A, lo+1, hi, (lo+hi)/2 );
   // 2) Swap pivot and first element.
   // 3) Partition the remainder of the subarray.
   // 4) Put pivot in its appropriate spot between the halves.
   // 5) Return index where pivot is currently stored.
}
```

Notice that we are passing the array, and the second index, last index, and middle index to `MedianOfThree`. The method itself won't care which index was the middle one, which was the last one, and which was the second one. What it *will* care about is the values at these indices. Specifically, the method needs to decide, given an array `A` and three indices `i`, `j`, and `k`, what order `A[i]`, `A[j]`, and `A[k]` belong in, and therefore which of the three is the median, and therefore which of `i`, `j`, and `k` is the index of the median.

We have six possibilities for the ordering of the three values at those three indices:

```
A[i] <= A[j] <= A[k]   // return j
A[i] <= A[k] <= A[j]   // return k
A[k] <= A[i] <= A[j]   // return i
A[j] <= A[i] <= A[k]   // return i
A[j] <= A[k] <= A[i]   // return k
A[k] <= A[j] <= A[i]   // return j
```

And so all we really need to do is compare between the three values to determine the median, and then return the appropriate index. I went over this in class, but it is straightforward enough that I'll just put the resultant method here and leave it to you to understand why it is written the way it is.

```
private static int MedianLocation(int[] A, int i, int j, int k)
{
   if (A[i] <= A[j])
      if (A[j] <= A[k])
         return j;
      else if (A[i] <= A[k])
         return k;
      else
         return i;
   else // A[j] < A[i]
      if (A[i] <= A[k])
         return i;
      else if (A[j] <= A[k])
         return k;
      else
         return j;
}
```

2) Swapping the pivot and the first element: With `MedianOfThree` completed, we now need to swap the pivot out of the way, into the first cell in the array, so that the cells from the second index through the last index can be part of the partitioning.

```
private static int PartitionWrapper(int[] A, int lo, int hi)
{
   int currentPivotLocation =
        MedianOfThree(A, lo+1, hi, (lo+hi)/2 );
   Swap(A, lo, currentPivotLocation);
   // 3) Partition the remainder of the subarray.
   // 4) Put pivot in its appropriate spot between the halves.
   // 5) Return index where pivot is currently stored.
}
```

Actually, we will never use `currentPivotLocation` after this swap – from now on, the pivot will either be sitting at `A[lo]` or much later, we will move the pivot to its correct location in the array. So, we really don't care where we found the pivot except that we need the index of that median-of-3 element in order to make the "swap pivot out of the way" work. Because of this, we can write the index directly into the swap by just passing the method call as an argument to swap. But there's no need to store the `currentPivotLocation` in a separate variable because it will never be needed again after this line of code.

```
private static int PartitionWrapper(int[] A, int lo, int hi)
{
   Swap(A, lo, MedianLocation(A, lo+1, hi, (lo+hi)/2));
   // 3) Partition the remainder of the subarray.
   // 4) Put pivot in its appropriate spot between the halves.
   // 5) Return index where pivot is currently stored.
}
```

3) Partitioning the remainder of the array: As we said way above, the partitioning is essentially handled recursively, as "partition this subarray" becomes a matter of "deal witht the leftmost element and then partition a smaller portion of this subarray". So, we could write a recursive version of `Partition` which would only take care of that actual partitioning work, and from this non-recursive `PartitionWrapper`, we would only control the swap of pivot to the leftmost cell (step 2) and then the swap of pivot to it's proper location later on (step 4). That is, `Partition` will be the recursive algorithm that repeats the partitioning step over and over, and `PartitionWrapper` does the setup work that is needed *before* `Parition` runs (such as selecting the pivot) and the clean-up work that is needed *after* `Partition` runs.

All we need returned from that recursive version of `Partition` is the index `m` where the pivot should swap to in the end. Since the pivot is sitting at `A[lo]`, we pass the recursive partitioning algorithm the bounds `lo + 1` and `hi`, which are the bounds of the remainder of this array. We will also pass in the pivot *value*, while `lo + 1` and `hi` are only *indices*.

```
private static int PartitionWrapper(int[] A, int lo, int hi)
{
   Swap(A, lo, MedianLocation(A, lo+1, hi, (lo+hi)/2));
   int m = Partition(A, lo+1, hi, A[lo]);
   // 4) Put pivot in its appropriate spot between the halves.
   // 5) Return index where pivot is currently stored.
}
```

4) Putting pivot in its appropriate place between the halves: Now that you know which index `m` is, well, we know we are suppposed to put the pivot there, which means it is time for another swap (where we swap the pivot value in `A[0]` with the `m` generated by the recursive `Partition` algorithm.

```
private static int PartitionWrapper(int[] A, int lo, int hi)
{
   Swap(A, lo, MedianLocation(A, lo+1, hi, (lo+hi)/2));
   int m = Partition(A, lo+1, hi, A[lo]);
   Swap(A, lo, m);
   // 5) Return index where pivot is currently stored.
}
```

5) Return index where pivot is currently stored: Once the above swap is complete, the pivot is stored at index `m`, so we simply return the index `m` since that is the index that our `Quicksort` method is waiting for in order to perform its own recursive calls.

```
// final version of the PartitionWrapper algorithm
private static int PartitionWrapper(int[] A, int lo, int hi)
{
   Swap(A, lo, MedianLocation(A, lo+1, hi, (lo+hi)/2));
   int m = Partition(A, lo+1, hi, A[lo]);
   Swap(A, lo, m);
   return m;
}
```

So, above, we run median-of-three to get the pivot location and swap the pivot with the far left cell. We then call the recursive `Partition` to partition the rest of the array. It returns the index where the pivot should go, and we swap the pivot into that cell and return that index back to `Quicksort`.

Finally, we need to write the recursive version of `Partition`, which assumes the pivot is already out of the way and just worries about the actual rest of the array.

```
private static int Partition(int[] A, int lo, int hi, int pivot)
```

To do this, we just consider the cases we discussed when going over two examples of partitioning way above: We are partitioning the subarray `lo...hi`. We start by inspecting `A[lo]`;

Recursive case 1: `A[lo]` is less than pivot. In that case, we just left `A[lo]` where it was and recursively partitioned the subarray `lo+1...hi`.

```
if (A[lo] <= pivot)
   return Partition(A, lo+1, hi, pivot);
```

Recursive case 2: `A[lo]` is greater than pivot. In that case, we swapped `A[lo]` with `A[hi]`, and then recursively partitioned the subarray `lo...hi-1` (meaning the right bound moved left one position, and we re-checked `A[lo]` since we had just swapped a new value into that position:

```
if (A[lo] > pivot)
{
   Swap(A, lo, hi);
   return Partition(A, lo, hi-1, pivot);
}
```

Base case: The base case occured when we were down to only one element in our subarray – i.e. when `lo == hi`. In this case, we simply want to return which index the pivot should end up in, and we said it would either be the location of this element in the partition of size 1 (i.e. the index `lo == hi`) or else it would be the cell to the left of that (i.e. with index `lo - 1`). And, the way we made that choice was, we returned the location of the element in the partition of size 1 if that element was less than the pivot and should be swapped into pivot's location on the left. If instead, the element belonged to the right side, the "greater than pivot" side, then we wanted to swap pivot with the element to the left of that instead – in both cases, the rightmost element in the left side.

```
   if (hi == lo)
      if (A[lo] <= pivot)
         return lo;
      else
         return lo-1;
```

Putting it all together...

```
private static int Partition(int[] A, int lo, int hi, int pivot)
{
   if (hi == lo)
      if (A[lo] < pivot)
         return lo;
      else
         return lo-1;
   else if (A[lo] <= pivot)
      return Partition(A, lo+1, hi, pivot);
   else
   {
      Swap(A, lo, hi);
      return Partition(A, lo, hi-1, pivot);
   }
}
```