Data Structures Advanced Sorts Part 2: Quicksort

Phil Tayco

Slide version 1.2

Apr. 5, 2020

Advanced Sorts

Divide and Conquer

- The mergesort algorithm shows that using the divide and conquer approach can lead to improving the sort algorithms from $O(n^2)$ to $O(n \log n)$
- Its challenge is that it requires twice the memory space of the size of the array we are trying to sort
- To combat this, we need to combine a divide and conquer approach with an idea that allows us to not require a temp array
- Without a temp array, we'll need to figure out how improve the sort process using swaps and/or shifts

Advanced Sorts

Mergesort as a model

- The recursive mergesort algorithm contained 3 keys parts:
  - A base case to stop the divide and conquer looping
  - A recursive case that reuses the sorting algorithm on the left and right side of a current look at the array
  - A key function (like the merge) that takes place during the recursive case
- The merge function took place at the end of the recursive case taking advantage of its design to merge two sub arrays into one
- If we take a similar approach, we need a similar key function that optimizes swapping and/or shifting in conjunction with divide and conquer

Advanced Sorts

Left and right

- To use divide and conquer effectively, we need to look at ways to cleverly and recursively split the array
- One way is to put rules around what it means split the array such that the left and right sides of an array have some significant meaning
- One idea there is to define this as the data on the left and right sides are “where they should be”
- This “where they should be” does not necessarily have to mean sorted – the data just has to be in the correct place
- To get there, we first need a reference point

Advanced Sorts

This is pivotal

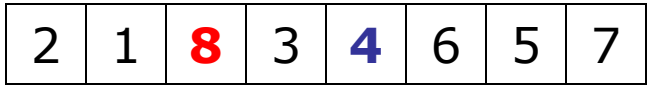
- That reference point can be one data element between the two sides such that the left and right sides of it follow our rules
- We'll call this reference point the "pivot" value and define it as a value somewhere in the array such that:
  - All elements to its left contain values that are less than it (not necessarily sorted)
  - All elements to its right contain values that are greater than it (also not necessarily sorted)
  - Data elements of equal values we'll discuss later

Advanced Sorts

Example “pivot” value in blue correctly positioned:



Example “pivot” value not correctly positioned (left side is incorrect):



Advanced Sorts

Staging the data

- Why is this relationship between sides and a pivot value important?
- It gives us a way to express splitting the array and we can now use this condition to divide an array
- As we did with the mergesort, we can recursively apply these rules to divide the array
- With recursion, we still need to define a base case
- Recall that with the mergesort, the base case was when the array splitting came down to 1 element left, which by definition was also a sorted sub array
- The same can apply such that we want to split the array into sub arrays that meet these pivot rules until we get to a sub array size of 1 (or 0) elements
- So now the question is, how do we create sub arrays that are correctly positioned around a pivot value? How do we even choose the pivot value?

Advanced Sorts

Partitioning

- That process we will call “partitioning” and like the merge function in mergesort, this sorting algorithm will use the partition function in its recursive case
- The idea is then to repeatedly partition the array and its sub parts recursively until there is nothing necessary to partition
- By the time you are done partitioning to the smallest subarrays, the entire array should be sorted
- So how do we partition an array? Here’s the algorithm

Advanced Sorts

Partitioning

- Select the last element in the current part of the array – its value will represent the pivot value for the partition
- Look at the first element in the array and examine elements from left to right until you find a value that is greater than (or equal to) the pivot value – the location you stop at we'll call "left" index
- Now look at the second to last element (the element to the immediate left of the value we're currently calling pivot) - this time go right to left until you find a value less than the pivot value or you've passed the beginning of the array – this stopping point we'll call "right" index
- At this point, we now have 2 index values that will be in 1 of 2 situations:
  - The "left" and "right" index pointers did not cross paths or
  - The "left" and "right" index pointers crossed paths (which also includes left and right being at the same spot)
- Let's look at an example

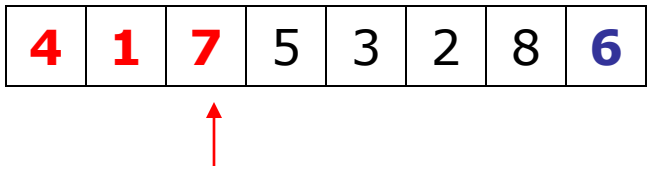
Advanced Sorts

Starting point. Pivot will be the selected value in blue:

4	1	7	5	3	2	8	**6**

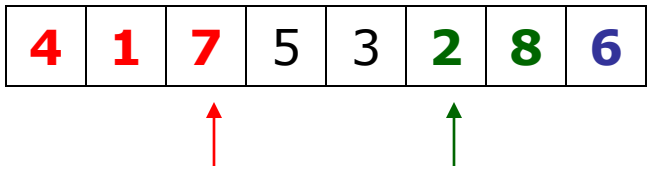
Advanced Sorts

Left pointer starts at 4. It is not greater than or equal to pivot value so we move right until we find a value that does. This ends up being the 3rd element with value 7 (left index = 2):



Advanced Sorts

Right pointer starts at 8. It is not less than 6 so we go left until we find one that does, which is the next element with value 2 (right index = 5):



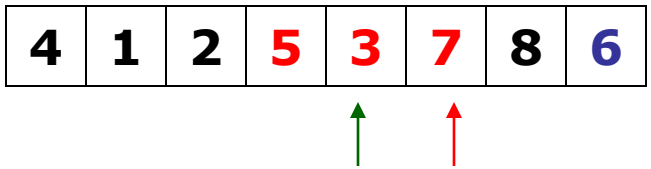
Advanced Sorts

Partitioning

- In this situation, the left and right pointers did not cross paths
- For the partition algorithm, this means we swap the elements at the left and right pointers
- The need to swap occurs because this process finds 2 values that are incorrectly positioned based on our chosen pivot value – the swap puts these elements in the correct side (and not necessarily the correct sorted position)
- After the swap, we repeat the search process of the left and right pointers following the same algorithm
- This repeats until the pointers cross paths

Advanced Sorts

2 and 7 swap. Process repeats this time with the left pointer stopping at 7 (left index = 5) and the right pointer stopping at 3 (right index = 4):



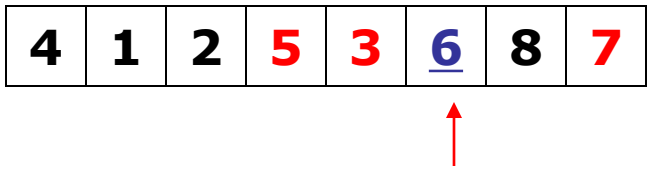
Advanced Sorts

Partitioning

- In this situation, the left and right pointers have crossed paths
- Now, the elements at the left and right pointers do not swap positions with each other
- Instead, the location of the left index pointer becomes, shall we say, “pivotal”
- Notice that the left pointer’s location ends up being the location where the pivot value should go
- Also notice that, by logical rule, the value at the left pointer’s location belongs on the right side
- There is only one value on the right side of where the pivot value should be that is incorrectly positioned – it’s the pivot element itself! Thus, we swap left pointer with pivot

Advanced Sorts

Left and right pointers crossed paths, swap element at left pointer with pivot:



Now notice 2 things have occurred:

- The left and right side values are correctly positioned around the pivot value
- The pivot value is actually in the correct sorted position of the entire array!

Advanced Sorts

Partitioning summarized

1. For a given array (or a given part of an array), make the pivot value be represented by the last value in the array
2. Start left at index 0 and go to the right until you see a value greater than or equal to pivot
3. Then start right at the second to last index and go to the left until you see a value less than pivot
4. Next, ask “did left and right cross paths”
5. If they did not, swap the values at left and right index and go back to step 2, continuing left and right index from where they had stopped
6. If they did cross paths, swap the pivot value with the value at wherever left index stopped

Advanced Sorts

First step done, now divide and conquer

- Now that we know how to partition an array around a pivot value, we can apply this function recursively like we did with the mergeSort algorithm
- In this case, though, we'll make partitioning the first step of the recursive case of the algorithm - with mergeSort's recursive case, we first mergeSorted the left half, then the right half and then we called merge
- After doing the partition, we will then recursively call quicksort on the left and right sides of the pivot value
- That's for the recursive case of the algorithm - what is the base case here?
- With mergeSort, the base case was doing mergeSort on an array size of one element – here, it's if there are 0 or 1 elements on the "side" to sort
- As with merge sort, when we hit the base case, we simply return out of it

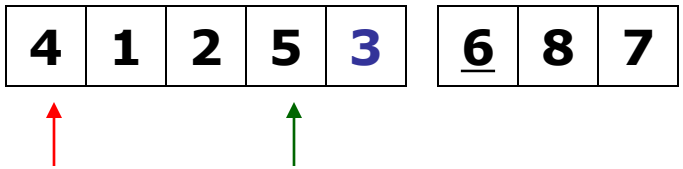
Advanced Sorts

Quicksort

- The algorithm can be stated as follows:
  - If the current array is 0 or 1 elements, return
  - Else
    - Partition the array
    - Quicksort the left side of pivot
    - Quicksort the right side of pivot
- Let’s complete the rest of the sorting of our example with this algorithm

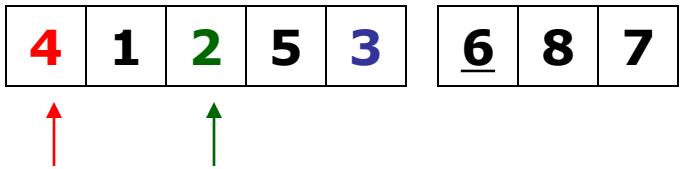
Advanced Sorts

After partitioning the entire array, pivot landed on index 5. Quicksort left side (0..4). It's not the base case so we partition. Pivot value is **3**. Left and right pointers get ready to do their work



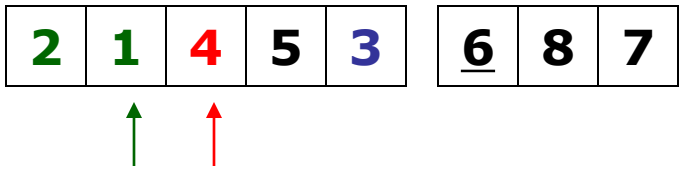
Advanced Sorts

Left stops at 4 (it is greater than 3) and right stops at 2 (5 was greater than 3, but not 2). The pointers do not cross paths so the two elements will swap



Advanced Sorts

After the swap, left and right advance and repeat their process. Left will start at 1 and stop at 4 while right will also start at 1 but stop there



Advanced Sorts

Left and right have crossed paths. Left is in the correct pivot position and we swap it with pivot. 3 is in the correct sorted position and its left and right sides are correctly partitioned



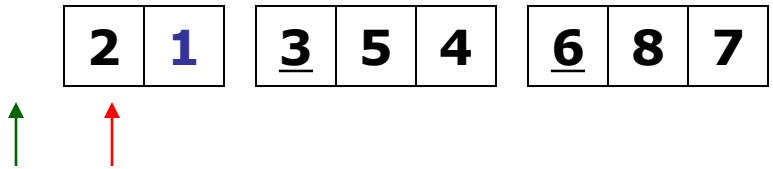
Advanced Sorts

We are still in Quicksort(0..4) and just partitioned it. Now we Quicksort its left and right sides, starting with Quicksort (0..1)



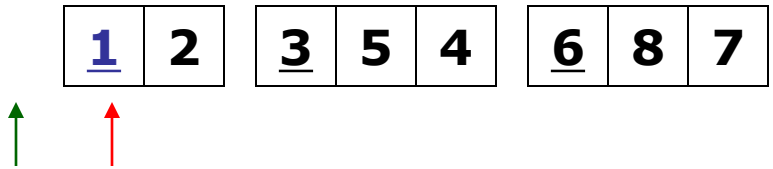
Advanced Sorts

[0..1] has 2 elements so it is not a base case. 1 is the pivot and left will end at index 0 while right will "fall off" the array and be at index -1. More important is that they have crossed paths



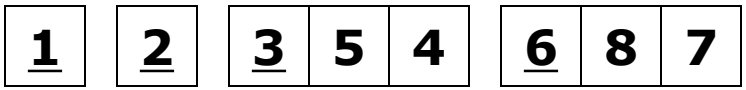
Advanced Sorts

Since left and right crossed, Left and pivot now swap. Partitioning for [0..1] is done but we still must quicksort left and right of "1" (which means pivot's index is [0])



Advanced Sorts

When we Quicksort the left of pivot, the array is empty. This is a base case so that function returns to [0..1]. We then do the right side of [0..1], we will be looking at just [1].



This is also a base case, so we return from there as well

Notice that this results in values 1 and 2 to end up being sorted in the overall array

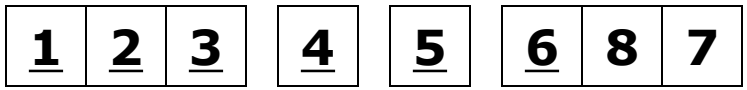
Advanced Sorts

[3..4] will partition similarly to when we did the partition of [0..1]. 4 will be the pivot and left and right will stay at 5. Left swaps with pivot putting 4 in the correct spot



Advanced Sorts

After partitioning, Quicksort of the left and right of [3..4] will be base cases. The 5 stays where it is and we are done with Quicksort of [3..4]



Advanced Sorts

We’re back to the overall array of Quicksort [0..7]! When we left here, we had partitioned around [5] and did Quicksort [0..4]. Now we Quicksort the right side which is [6..7]



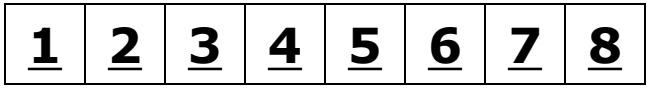
Advanced Sorts

As you may see from previous sub arrays with 2 elements, the partition and Quicksort of [6..7] will result in a swap and completion of base cases



Advanced Sorts

We come back from Quicksort [6..7], and we're all done!



Advanced Sorts

```
public static void quickSort()
{
   recursiveQuickSort(0, numbers.length - 1);
}

private static void recursiveQuickSort(int left, int
   right)
{
   if (right - left <= 0)
        return;
   else
   {
        int pivot = numbers[right];

        int partitionIndex = partition(left, right, pivot);
        recursiveQuickSort(left, partitionIndex - 1);
        recursiveQuickSort(partitionIndex + 1, right);
   }
}
```

Advanced Sorts

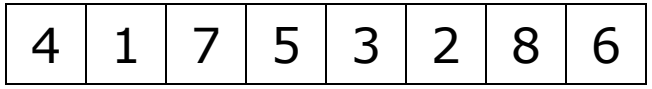
```
private static int partition(int left, int right, int
   pivot)
{
   int leftPtr = left-1;
   int rightPtr = right;

   while(true)
   {
        while(numbers[++leftPtr] < pivot);
        while(rightPtr > 0 && numbers[--rightPtr] > pivot);

        if(leftPtr >= rightPtr)
               break;
        else
               swap(leftPtr, rightPtr);
   }
   swap(leftPtr, right);
   return leftPtr;
}
```

Advanced Sorts

Example of code walkthrough with 8 elements



recQuickSort(0..7)

Advanced Sorts

1st call is not a base case, so we partition and then make the recursive calls. Partition starts with selecting 6

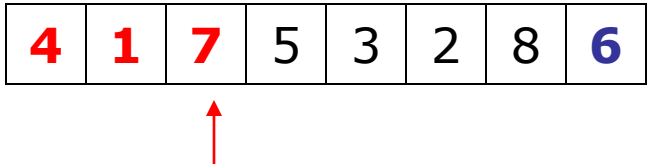
4	1	7	5	3	2	8	**6**

recQuickSort(0..7)

partition(0, 7, 6);

Advanced Sorts

First loop starts left index at 0 and stops when leftPtr reaches index 2 because the value found (7) is greater than pivot



partition(0, 7, 6)

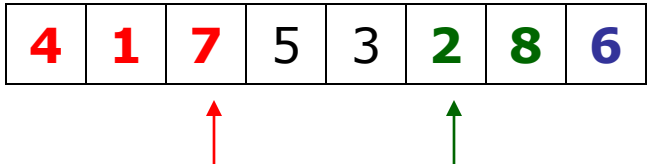
leftPtr = 2

recQuickSort(0..7)

partition(0, 7, 6);

Advanced Sorts

Second loop starts right index at 6 and stops when rightPtr reaches index 5 because the value found (2) is less than pivot



partition(0, 7, 6)

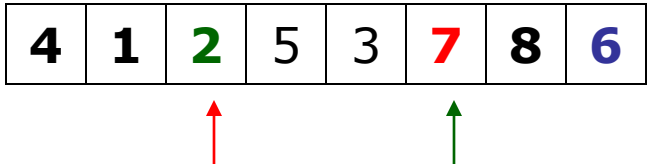
leftPtr = 2, rightPtr = 5

recQuickSort(0..7)

partition(0, 7, 6);

Advanced Sorts

LeftPtr is not greater than or equal to rightPtr (they did not cross paths), so we swap them



partition(0, 7, 6)

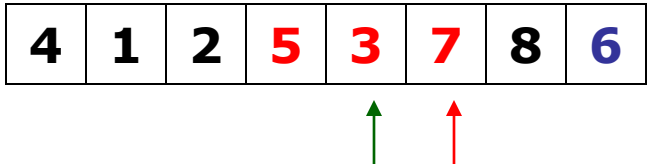
swap (2, 5);

recQuickSort(0..7)

partition(0, 7, 6);

Advanced Sorts

The loops repeat, this time with leftPtr stopping at 5 and rightPtr stopping at 4



partition(0, 7, 6)

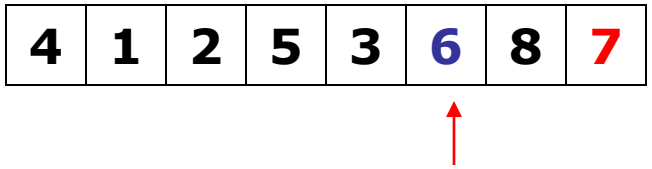
leftPtr = 5, rightPtr = 4

recQuickSort(0..7)

partition(0, 7, 6);

Advanced Sorts

LeftPtr is greater than rightPtr (they have crossed paths), so the loops stop and we swap where leftPtr is with the pivot element



partition(0, 7, 6)

swap (5, 7);

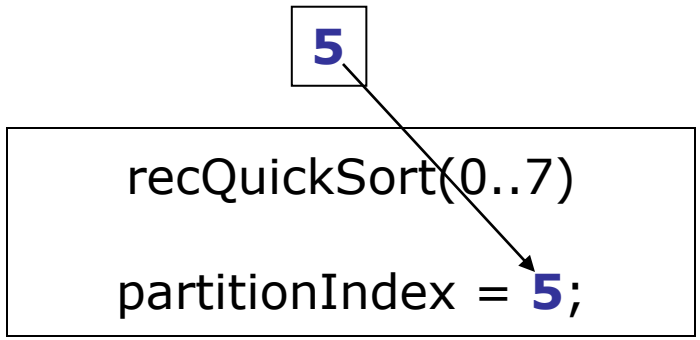
recQuickSort(0..7)

partition(0, 7, 6);

Advanced Sorts

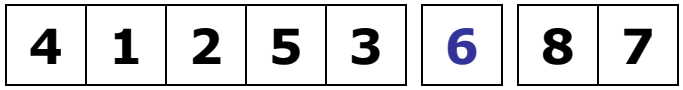
Partition is complete and returns the index of where the pivot element now resides. This will be used by the Quicksort algorithm to recursively call Quicksort on the left and right sides of the array

4	1	2	5	3	6	8	7



Advanced Sorts

With partition complete for [0..7], we now recursively call recQuickSort on the left from [0..4] and then later on the right from [6..7]




```
recQuickSort(0..7)

recQuicksort(0..4);
recQuickSort(6..7);
```

Advanced Sorts

In Quicksort(0..4), the base case is not reached so we partition with pivot value of 3



recQuickSort(0..4)

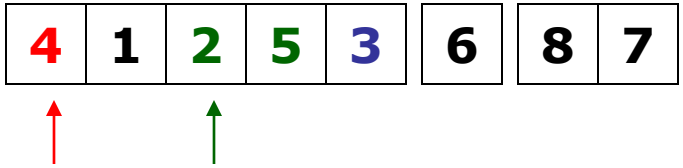
partition(0, 4, 3)

recQuickSort(0..7)

recQuicksort(0..4);
recQuickSort(6..7);

Advanced Sorts

The loops begin with leftPtr starting and stopping at 0 and rightPtr starting at 3 and stopping at 2



partition(0, 4, 3)

leftPtr = 0, rightPtr = 2

recQuickSort(0..4)

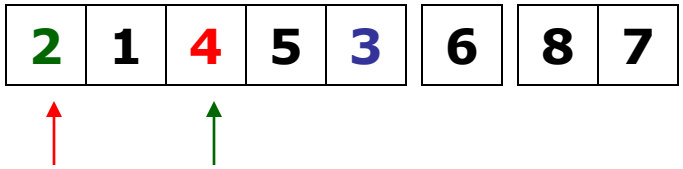
partition(0, 4, 3)

recQuickSort(0..7)

recQuicksort(0..4);
recQuickSort(6..7);

Advanced Sorts

LeftPtr and rightPtr did not cross paths, so they swap



partition(0, 4, 3)

swap(0, 2);

recQuickSort(0..4)

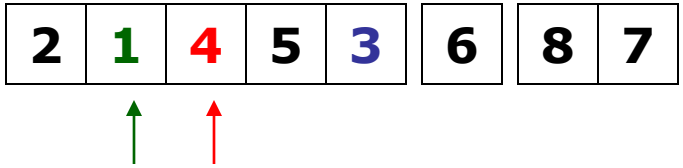
partition(0, 4, 3)

recQuickSort(0..7)

recQuicksort(0..4);
recQuickSort(6..7);

Advanced Sorts

Loops repeat with leftPtr stopping at 2 and rightPtr stopping at 1



partition(0, 4, 3)

leftPtr = 2, rightPtr = 1

recQuickSort(0..4)

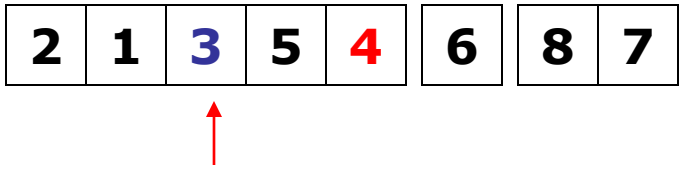
partition(0, 4, 3)

recQuickSort(0..7)

recQuicksort(0..4);
recQuickSort(6..7);

Advanced Sorts

LeftPtr and rightPtr have crossed paths, so the loops stop and leftPtr swaps with pivot



partition(0, 4, 3)

swap(2, 4);

recQuickSort(0..4)

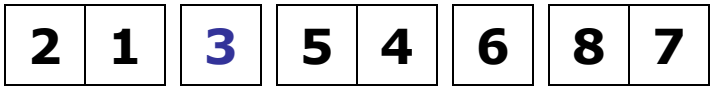
partition(0, 4, 3)

recQuickSort(0..7)

recQuicksort(0..4);
recQuickSort(6..7);

Advanced Sorts

The partition is complete and returns pivot index of 2. This is used to split [0..4] and recursively call Quicksort on the respective sides



recQuickSort(0..4)

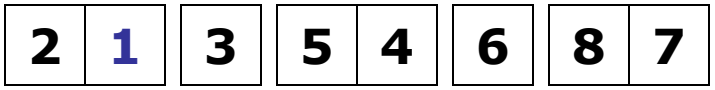
recQuickSort(0..1);
recQuickSort(3..4);

recQuickSort(0..7)

recQuicksort(0..4);
recQuickSort(6..7);

Advanced Sorts

In [0..1], the base case is still not reached.
Partition with pivot value of 1



recQuickSort(0..1)

partition(0, 1, 1);

recQuickSort(0..4)

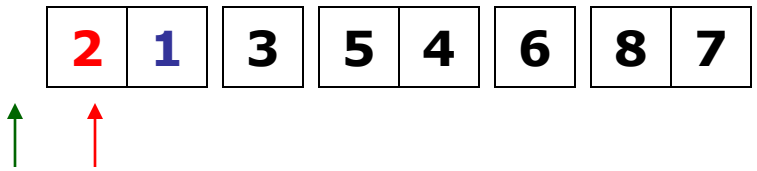
recQuickSort(0..1);
recQuickSort(3..4);

recQuickSort(0..7)

recQuicksort(0..4);
recQuickSort(6..7);

Advanced Sorts

LeftPtr starts and stops at 0 and rightPtr starts at 0 and ends at -1 (rightPtr "fell off" the subarray)



partition(0, 1, 1)

leftPtr = 0, rightPtr = -1

recQuickSort(0..1)

partition(0, 1, 1);

recQuickSort(0..4)

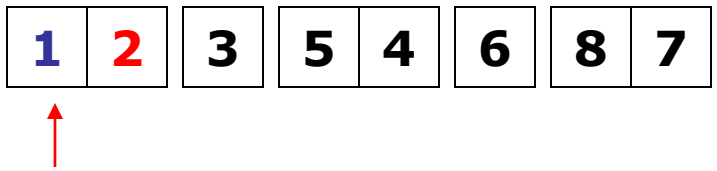
recQuickSort(0..1);
recQuickSort(3..4);

recQuickSort(0..7)

recQuicksort(0..4);
recQuickSort(6..7);

Advanced Sorts

LeftPtr and rightPtr have crossed paths, so we swap with leftPtr and pivot



partition(0, 1, 1)

swap(0, 1);

recQuickSort(0..1)

partition(0, 1, 1);

recQuickSort(0..4)

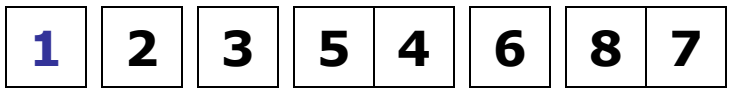
recQuickSort(0..1);
recQuickSort(3..4);

recQuickSort(0..7)

recQuicksort(0..4);
recQuickSort(6..7);

Advanced Sorts

Partition returns to [0..1] with an index of 0. The recursive calls are next, and these will be quick because they are both base cases that will simply return



recQuickSort(0..1)

recQuickSort(0..-1);
recQuickSort(1..1);

recQuickSort(0..4)

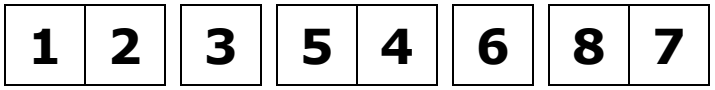
recQuickSort(0..1);
recQuickSort(3..4);

recQuickSort(0..7)

recQuicksort(0..4);
recQuickSort(6..7);

Advanced Sorts

This will conclude recQuickSort(0..1) and return to recQuickSort(0..4). Now the second recQuickSort function is called for [3..4]



recQuickSort(0..4)

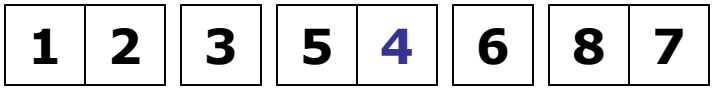
recQuickSort(0..1);
recQuickSort(3..4);

recQuickSort(0..7)

recQuicksort(0..4);
recQuickSort(6..7);

Advanced Sorts

In [3..4], the base case is not reached so we partition with pivot value 4



recQuickSort(3..4)

partition(3, 4, 4);

recQuickSort(0..4)

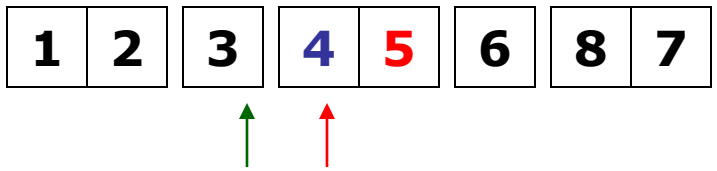
recQuickSort(0..1);
recQuickSort(3..4);

recQuickSort(0..7)

recQuicksort(0..4);
recQuickSort(6..7);

Advanced Sorts

LeftPtr and rightPtr end up crossing paths so we swap leftPtr with pivot and return pivot index of 3



partition(3, 4, 4)

swap(3, 4);

recQuickSort(3..4)

partition(3, 4, 4);

recQuickSort(0..4)

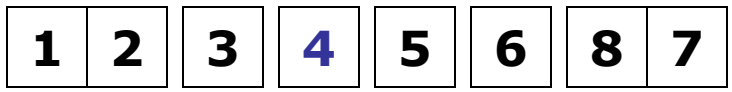
recQuickSort(0..1);
recQuickSort(3..4);

recQuickSort(0..7)

recQuicksort(0..4);
recQuickSort(6..7);

Advanced Sorts

The next 2 recursive calls in [3..4] will be base cases that simply return. When they come back, we can return from recQuickSort(3..4)



recQuickSort(3..4)

recQuickSort(3..2);
recQuickSort(4..4);

recQuickSort(0..4)

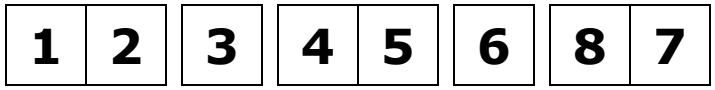
recQuickSort(0..1);
recQuickSort(3..4);

recQuickSort(0..7)

recQuicksort(0..4);
recQuickSort(6..7);

Advanced Sorts

Returning to recQuickSort(0..4) this function will now finish and return to recQuickSort(0..7)



recQuickSort(0..4)

recQuickSort(0..1);
recQuickSort(3..4);

recQuickSort(0..7)

recQuicksort(0..4);
recQuickSort(6..7);

Advanced Sorts

Returning to recQuickSort(0..4) this function will now finish and return to recQuickSort(0..7)



recQuickSort(0..7)

recQuicksort(0..4);
recQuickSort(6..7);

Advanced Sorts

Returning to recQuickSort(0..7), we now make the second recursive function call there for [6..7]. This will result with not being a base case so we'll once again partition with this one having pivot value 7



recQuickSort(6..7)

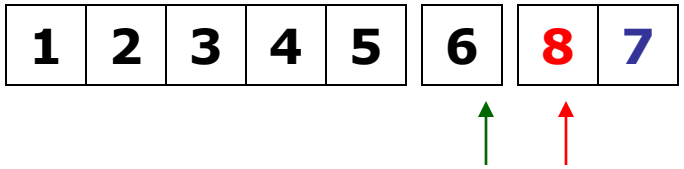
partition(6, 7, 7);

recQuickSort(0..7)

recQuicksort(0..4);
recQuickSort(6..7);

Advanced Sorts

We've seen this pattern before. LeftPtr and rightPtr cross paths at index 6



partition(6, 7, 7)

leftPtr = 6, rightPtr = 5

recQuickSort(6..7)

partition(6, 7, 7);

recQuickSort(0..7)

recQuicksort(0..4);
recQuickSort(6..7);

Advanced Sorts

LeftPtr swaps with pivot and returns pivot index of 6 to [6..7]. This now leads to 2 recursive calls that are both base cases



recQuickSort(6..7)

recQuicksort(6..5);
recQuickSort(7..7);

recQuickSort(0..7)

recQuicksort(0..4);
recQuickSort(6..7);

Advanced Sorts

The base case recursive calls return and also concludes recQuickSort(6..7). We now return to [0..7]



recQuickSort(0..7)

recQuicksort(0..4);
recQuickSort(6..7);

Advanced Sorts

This completes [0..7] and the array is sorted!

1	2	3	4	5	6	7	8

Advanced Sorts

Analysis

- The Quicksort is the same process as the Mergesort except that instead of doing the recursive calls first and then do the merge, we partition first and then do the recursive calls
- Both algorithms use a divide and conquer approach implying that the performance will also be O(n log n)
- This is great news because this algorithm does not use a temp array and thus, does not require twice the memory space to run!
- The worst case situation, though, is not particularly good for Quicksort. Can you spot what it is?

Advanced Sorts

Mergesort vs. Quicksort

- One advantage Mergesort has over Quicksort is that it is much more consistent
- This is due to the fact that Mergesort always has an exact middle point and the recursive function calls are on left and right sides that are close to the same in size
- Quicksort's left and right sides depend on where the pivot value ends up
- This implies that there are good and bad situations with Quicksort
- Good is when a pivot value ends up near the middle of the array and helps with the overall performance
- Bad is when the pivot value ends up near one of the ends of array – this happens when the data is nearly sorted or reverse sorted and leads to performance near $O(n^2)$
  - Which algorithm is better suited for near sorted situations?

Advanced Sorts

Efficiency

- With more good situations than not, though, the performance of Quicksort is O(n log n) because it uses a divide and conquer recursive approach like MergeSort
- The biggest key difference is the unnecessary need for a temp array with Quicksort, so already there is a strong advantage with Quicksort over Mergesort
- However, Quicksort can degrade to $O(n^2)$ if the array is nearly sorted – in such situations, InsertSort works great
- Given random distribution of data though, QuickSort wins despite MergeSort's consistency at O(n log n) because of the memory usage
- All our algorithms can now be summarized on the following table

Advanced Sorts

	Comparisons	Notes
Bubble	$O(n^2)$	Can be O(n) if optimized and data is nearly sorted
Selection	$O(n^2)$	Swaps are O(n)
Insertion	$O(n^2)$ (worst case)	Range is O(n) to (n^2) Excellent for partially sorted lists
Merge	O(n log n)	Requires 2x memory space because of temp array
Quick	O(n log n)	Faster and more space efficient than Merge, but degrades to $O(n^2)$ when the array is nearly sorted

Advanced Sorts

Other ideas

- Other advanced sorts exist:
  - Shell and Radix sort use a similar divide and conquer approach performing at O(n log n) – Quicksort on average still wins
  - Modifications of quicksort to partition or select pivot values differently
  - Use of insertion sort on sub arrays within the quicksort algorithm to take advantage of partitions that result in near sorted situations
- Back track: the sorting algorithms take place with arrays because by sorting the data, the search for an element significantly improves from O(n) to O(log n)
- Arrays are static in memory. Linked lists are dynamic, but even if we sort them, we cannot get O(log n) search because the middle of the list cannot be directly accessed…

- …unless we try another structure…