Lecture Notes on
Quicksort

15-122: Principles of Imperative Computation
Frank Pfenning

Lecture 8
September 18, 2014

1 Introduction

In this lecture we consider two related algorithms for sorting that achieve a much better running time than the selection sort from last lecture: mergesort and quicksort. We developed quicksort and its invariants in detail. As usual, contracts and loop invariants will bridge the gap between the abstract idea of the algorithm and its implementation.

We will revisit many of the computational thinking, algorithm, and programming concepts from the previous lectures. We highlight the following important ones:

Computational Thinking: We revisit the divide-and-conquer technique from the lecture on binary search. We will also see the importance of *randomness* for the first time.

Algorithms and Data Structures: We examine mergesort and quicksort, both of which use divide-and-conquer, but with different overall strategies.

Programming: We have occasionally seen *recursion* in specification functions. In both mergesort and quicksort, it will be a central computational technique.

Both mergesort and quicksort are examples of *divide-and-conquer*. We divide a problem into simpler subproblems that can be solved independently and then combine the solutions. As we have seen for binary search, the ideal *divide* step breaks a problem into two of roughly equal size, because it

means we need to divide only logarithmically many times before we have a basic problem, presumably with an immediate answer. Mergesort achieves this, quicksort not quite, which presents an interesting tradeoff when considering which algorithm to chose for a particular class of applications.

Recall linear search for an element in an array, which has asymptotic complexity of $O(n)$. The divide-and-conquer technique of binary search divides the array in half, determines which half our element would have to be in, and then proceeds with only that subarray. An interesting twist here is that we *divide*, but then we need to *conquer* only a single new subproblem. So if the length of the array is 2^k and we divide it by two on each step, we need at most k iterations. Since there is only a constant number of operations on each iteration, the overall complexity is $O(\log(n))$. As a side remark, if we divided the array into 3 equal sections, the complexity would remain $O(\log(n))$ because $3^k = (2^{log_2(3)})^k = 2^{log_2 3*k}$, so $\log_2(n)$ and $\log_3(n)$ only differ in a constant factor, namely $\log_2(3)$.

2 The Quicksort Algorithm

Quicksort uses the technique of divide-and-conquer in a different manner. We proceed as follows:

1. Pick an arbitrary element of the array (the *pivot*).

2. Divide the array into two segments, those that are smaller and those that are greater, with the pivot in between (the *partition* phase).

3. Recursively sort the segments to the left and right of the pivot.

In quicksort, dividing the problem into subproblems will be linear time, but putting the results back together is immediate. This kind of trade-off is frequent in algorithm design.

Let us analyze the asymptotic complexity of the partitioning phase of the algorithm. Say we have the array

$$3, 1, 4, 4, 7, 2, 8$$

and we pick 3 as our pivot. Then we have to compare each element of this (unsorted!) array to the pivot to obtain a partition where $2, 1$ are to the left and $4, 7, 8, 4$ are to the right of the pivot. We have picked an arbitrary order for the elements in the array segments: all that matters is that all smaller ones are to the left of the pivot and all larger ones are to the right.

Since we have to compare each element to the pivot, but otherwise just collect the elements, it seems that the partition phase of the algorithm should have complexity $O(k)$, where k is the length of the array segment we have to partition.

It should be clear that in the ideal (best) case, the pivot element will be magically the *median* value among the array values. This just means that half the values will end up in the left partition and half the values will end up in the right partition. So we go from the problem of sorting an array of length n to an array of length $n/2$. Repeating this process, we obtain the following picture:
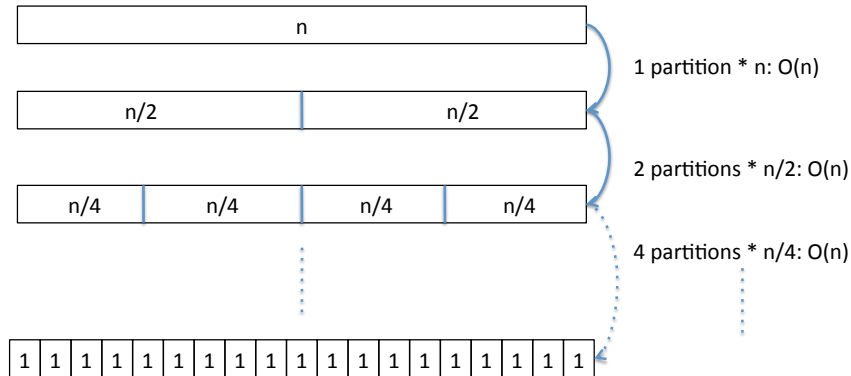
Quicksort, best case: log(n) levels, O(n) per level

At each level the total work is $O(n)$ operations to perform the partition. In the best case there will be $O(\log(n))$ levels, leading us to the $O(n\log(n))$ best-case asymptotic complexity.

How many recursive calls do we have in the worst case, and how long are the array segments? In the worst case, we always pick either the smallest or largest element in the array so that one side of the partition will be empty, and the other has all elements except for the pivot itself. In the example above, the recursive calls might proceed as follows (where we have

surrounded the unsorted part of the array with brackets):

array	pivot
$[3, 1, 4, 4, 8, 2, 7]$	1
$1, [3, 4, 4, 8, 2, 7]$	2
$1, 2, [3, 4, 4, 8, 7]$	3
$1, 2, 3, [4, 4, 8, 8]$	4
$1, 2, 3, 4, [4, 8, 7]$	4
$1, 2, 3, 4, 4, [8, 7]$	7
$1, 2, 3, 4, 4, 7, [8]$	

All other recursive calls are with the empty array segment, since we never have any unsorted elements less than the pivot. We see that in the worst case there are $n - 1$ significant recursive calls for an array of size n. The kth recursive call has to sort a subarray of size $n - k$, which proceeds by partitioning, requiring $O(n - k)$ comparisons.

This means that, overall, for some constant c we have

$$c \sum_{k=0}^{n-1} k = c \frac{n(n-1)}{2} \in O(n^2)$$

comparisons. Here we used the fact that $O(p(n))$ for a polynomial $p(n)$ is always equal to the $O(n^k)$ where k is the leading exponent of the polynomial. This is because the largest exponent of a polynomial will eventually dominate the function, and big-O notation ignores constant coefficients.

So quicksort has quadratic complexity in the worst case. How can we mitigate this? If we could always pick the *median* among the elements in the subarray we are trying to sort, then half the elements would be less and half the elements would be greater. So in this case there would be only $\log(n)$ recursive calls, where at each layer we have to do a total amount of n comparisons, yielding an asymptotic complexity of $O(n\log(n))$.

Unfortunately, it is not so easy to compute the median to obtain the optimal partitioning. It *is* possible to compute the median of n elements in $O(n)$ time, but quicksort is rarely if ever implemented this way in practice. One reason for this is that it turns out that if we pick a *random* element, its expected rank will be close enough to the median that the expected running time of algorithm is still $O(n\log(n))$.

We really should make this selection randomly. With a fixed-pick strategy, there may be simple inputs on which the algorithm takes $O(n^2)$ steps. For example, if we always pick the first element, then if we supply an array

that is already sorted, quicksort will take $O(n^2)$ steps (and similarly if it is "almost" sorted with a few exceptions)! If we pick the pivot randomly each time, the kind of array we get does not matter: the expected running time is always the same, namely $O(n\log(n))$.[1] Proving this, however, is a different matter and beyond the scope of this course. This is an important example on how to exploit randomness to obtain a reliable average case behavior, no matter what the distribution of the input values.

3   The Quicksort Function

We now turn our attention to developing an imperative implementation of quicksort, following our high-level description. We implement quicksort in the function `sort` as an *in-place* sorting function that modifies a given array instead of creating a new one. It therefore returns no value, which is expressed by giving a return type of `void`.

```
void sort(int[] A, int lower, int upper)
//@requires 0 <= lower && lower <= upper && upper <= \length(A);
//@ensures is_sorted(A, lower, upper);
{
  ...
}
```

Quicksort solves the same problem as selection sort, so their contract is the same, but their implementation differs. We sort the segment $A[lower..upper)$ of the array between *lower* (inclusively) and *upper* (exclusively). The precondition in the `@requires` annotation verifies that the bounds are meaningful with respect to A. The postcondition in the `@ensures` clause guarantees that the given segment is sorted when the function returns. It does not express that the output is a permutation of the input, which is required to hold but is not formally expressed in the contract (see Exercise 1).

The quicksort function represents an example of *recursion*: a function (`sort`) calls itself on a smaller argument. When we analyze such a function call it would be a mistake to try to analyze the function that we call recursively. Instead, we reason about it using *contracts*.

1. We have to ascertain that the preconditions of the function we are calling are satisfied.

---

[1]Actually not quite, with the code that we show. Can you find the reason?

2. We are allowed to assume that the postconditions of the function we are calling are satisfied when it returns.

This applies no matter whether the call is recursive, as it is in this example, or not.

Reasoning about recursive functions using their contracts is an excellent illustration of computational thinking, separating the *what* (that is, the contract) from the *how* (that is, the definition of the function). To analyze the recursive call we only care about *what* the function does.

We also need to analyze the *termination* behavior of the function, verifying that the recursive calls are on strictly smaller arguments. What *smaller* means differs for different functions; here the size of the subrange of the array is what decreases. The quantity $upper - lower$ is divided by two for each recursive call and is therefore smaller since it is always greater or equal to 2. If it were less than 2 we would return immediately and not make a recursive call.

For quicksort, we don't have to do anything if we have an array segment with 0 or 1 elements. So we just return if $upper - lower \leq 1$.

```
void sort(int[] A, int lower, int upper)
//@requires 0 <= lower && lower <= upper && upper <= \length(A);
//@ensures is_sorted(A, lower, upper);
{
  if (upper-lower <= 1) return;
  ...
}
```

Next we have to select a pivot element and call a partition function. We tell that function the index of the element that we chose as the pivot. For illustration purposes, we use the middle element as a pivot (to work reasonably well for arrays that are sorted already), but it should really be a random element. We want partitioning to be done *in place*, modifying the array A. Still, partitioning needs to return the index *mid* of the pivot element because we then have to recursively sort the two subsegments to the left and right of the index where the pivot is after partitioning. So we declare:

```
int partition(int[] A, int lower, int pivot_index, int upper)
//@requires 0 <= lower && lower <= pivot_index;
//@requires pivot_index < upper && upper <= \length(A);
//@ensures lower <= \result && \result < upper;
//@ensures ge_seg(A[\result], A, lower, \result);
//@ensures le_seg(A[\result], A, \result+1, upper);
  ;
```

Here we use the auxiliary functions `ge_seg` (for *greater or equal than segment*) and `le_seg` (for *less or equal than segment*), where

- `ge_seg(x, A, lower, mid)` if $x \geq y$ for every y in $A[lower..mid)$.

- `le_seg(x, A, mid+1, upper)` if $x \leq y$ for every y in $A[mid+1..upper)$.

Their definitions can be found in the file arrayutil.c0.

Some details on this specification: we require *pivot_index* to be a valid index in the array range, i.e., $lower \leq pivot\_index < upper$. In particular, we require $lower < upper$ because if they were equal, then the segment could be empty and we cannot possibly pick a pivot element or return its index.

Now we can fill in the remainder of the main sorting function.

```
void sort(int[] A, int lower, int upper)
//@requires 0 <= lower && lower <= upper && upper <= \length(A);
//@ensures is_sorted(A, lower, upper);
{
  if (upper-lower <= 1) return;
  int pivot_index = lower + (upper-lower)/2; /* should be random */

  int mid = partition(A, lower, pivot_index, upper);
  sort(A, lower, mid);
  sort(A, mid+1, upper);
  return;
}
```

It is a simple but instructive exercise to reason about this program, using only the contract for `partition` together with the pre- and postconditions for `sort` (see Exercise 2).

To show that the `sort` function terminates, we have to show the array segment becomes strictly smaller in each recursive call. First, $mid - lower < upper - lower$ since $mid < upper$ by the postcondition for `partition`. Second, $upper - (mid + 1) < upper - lower$ because $lower < mid + 1$, also by the postcondition for `partition`.

4 Partitioning

The trickiest aspect of quicksort is the partitioning step, in particular since we want to perform this operation in place. Let's consider the situation when partition is called:
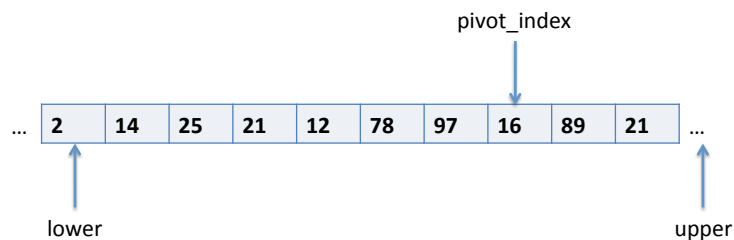

Perhaps the first thing we notice is that we do not know where the pivot will end up in the partitioned array! That's because we don't know how many elements in the segment are smaller and how many are larger than the pivot. In particular, the return value of partition could be different than

the pivot index that we pass in, even if the value that used to be at the pivot index in the array before calling partition will be at the returned index when partition is done.[2] One idea is to make a pass over the segment and count the number of smaller elements, move the pivot into its place, and then scan the remaining elements and put them into their place. Fortunately, this extra pass is not necessary. We start by moving the pivot element out of the way, by swapping it with the leftmost element in the array segment.

pivot = 16

| ... | 16 | 14 | 25 | 21 | 12 | 78 | 97 | 2 | 89 | 21 | ... |

lower   upper
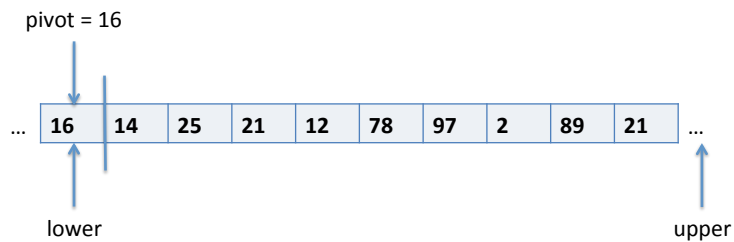
Now the idea is to gradually work towards the middle, accumulating elements less than the pivot on the left and elements greater than the pivot on the right end of the segment (excluding the pivot itself). For this purpose we introduce two indices, *left* and *right*. We start them out as *lower + 1* (to avoid the stashed-away pivot) and *upper*.

pivot = 16

| ... | 16 | 14 | 25 | 21 | 12 | 78 | 97 | 2 | 89 | 21 | ... |

lower   left   right   upper
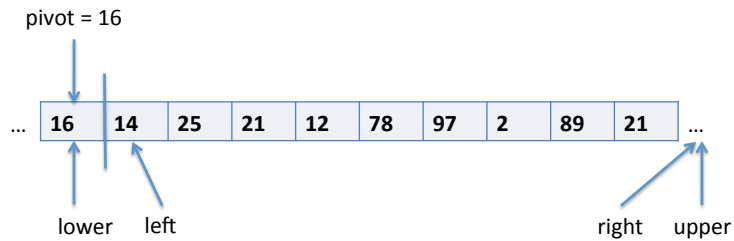
Since $14 < pivot$, we can advance the *left* index: this element is in the proper place.

pivot = 16
≤ pivot

| ... | 16 | 14 | 25 | 21 | 12 | 78 | 97 | 2 | 89 | 21 | ... |

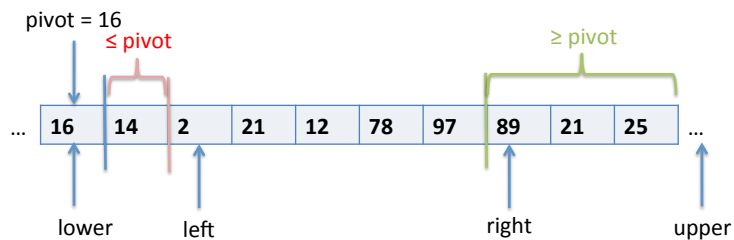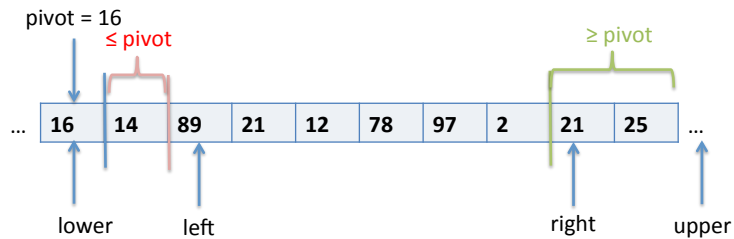lower   left   right   upper
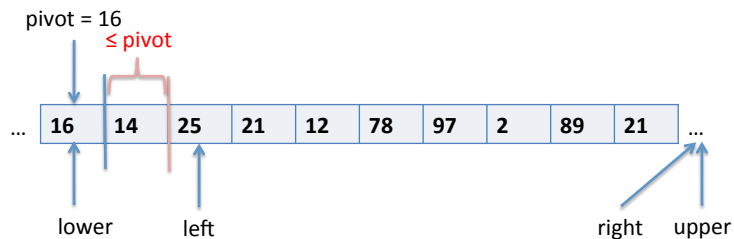
[2]To see why, imagine there are several elements equal to the pivot value.
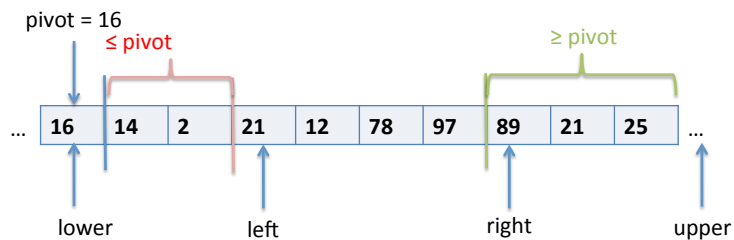
At this point, $25 > pivot$, it needs to go on the right side of the array. If we put it on the *extreme* right end of the array, we can then say that it is in it's proper place. We swap it into $A[right - 1]$ and decrement the $right$ index.

pivot = 16

≤ pivot   ≥ pivot

| ... | 16 | 14 | 21 | 21 | 12 | 78 | 97 | 2 | 89 | 25 | ... |

lower   left   right upper
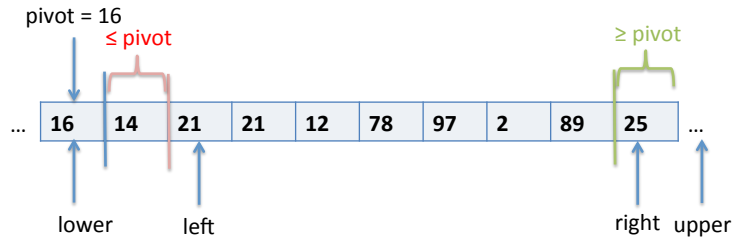
In the next two steps, we proceed by making swaps. First, we decide that the 21 that is currently at *left* can be properly placed to the left of the 25, so we swap it with the element to the left of 25. Then, we have 89 at $A[left]$, and so we can decide this is well-placed to the left of that 21.

pivot = 16

≤ pivot   ≥ pivot

| ... | 16 | 14 | 89 | 21 | 12 | 78 | 97 | 2 | 21 | 25 | ... |

lower   left   right   upper

pivot = 16

≤ pivot   ≥ pivot

| ... | 16 | 14 | 2 | 21 | 12 | 78 | 97 | 89 | 21 | 25 | ... |

lower   left   right   upper

Let's take one more step: $2 < pivot$, so we again just decide that the 2 is fine where it is and increment *left*.

pivot = 16

≤ pivot   ≥ pivot

| ... | 16 | 14 | 2 | 21 | 12 | 78 | 97 | 89 | 21 | 25 | ... |

lower   left   right   upper

At this point we pause to read off the general invariants which will allow us to synthesize the program. We see:

(1) $pivot \geq A[lower + 1..left)$

(2) $pivot \leq A[right..upper)$

(3) $A[lower] = pivot$

We may not be completely sure about the termination condition, but we can play the algorithm through to its end and observe:
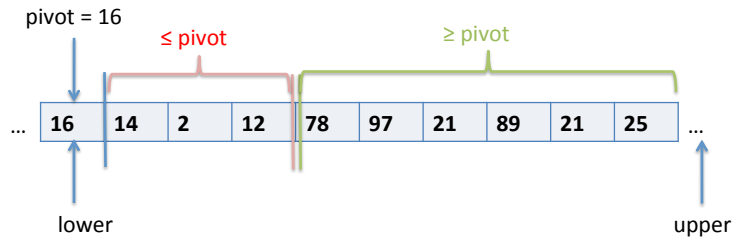
Where do *left* and *right* need to be, according to our invariants? By invariant (1), all elements up to but excluding *left* must be less or equal to *pivot*. To guarantee we are finished, therefore, the *left* must address the element 78 at $lower + 4$. Similarly, invariant (2) states that the pivot must be less or equal to all elements starting from *right* up to but excluding *upper*. Therefore, *right* must also address the element 3 at $lower + 3$.
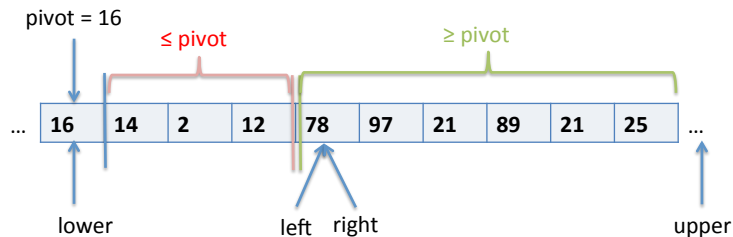
This means after the last iteration, just before we exit the loop, we have $left = right$, and throughout:

(4) $lower + 1 \leq left \leq right \leq upper$

Now comes the last step: since $left = right$, $pivot \geq A[left - 1]$ and we can swap the pivot at *lower* with the element at $left - 1$ to complete the partition operation. We can also see the $left - 1$ should be returned as the new position of the pivot element.

5 Implementing Partitioning

Now that we understand the algorithm and its correctness proof, it remains
to turn these insights into code. We start by swapping the pivot element to
the beginning of the segment.

```
int partition(int[] A, int lower, int pivot_index, int upper)
//@requires 0 <= lower && lower <= pivot_index;
//@requires pivot_index < upper && upper <= \length(A);
//@ensures lower <= \result && \result < upper;
//@ensures ge_seg(A[\result], A, lower, \result);
//@ensures le_seg(A[\result], A, \result+1, upper);
{
  int pivot = A[pivot_index];
  swap(A, lower, pivot_index);

  ...
}
```

At this point we initialize *left* and *right* to *lower + 1* and *upper*, respectively.
We have to make sure that the invariants are satisfied when we enter the
loop for the first time, so let's write these.

```
  int left = lower+1;
  int right = upper;
  while (left < right)
    //@loop_invariant lower+1 <= left && left <= right && right <= upper;
    //@loop_invariant ge_seg(pivot, A, lower+1, left); // Not lower!
    //@loop_invariant le_seg(pivot, A, right, upper);
    {
      ...
    }
```

The crucial observation here is that *lower* $<$ *upper* by the precondition of
the function. Therefore *left* $\leq$ *upper* $=$ *right* when we first enter the loop.
The segments $A[lower + 1..left)$ and $A[right..upper)$ will both be empty, ini-
tially.

The code in the body of the loop just compares the element at index *left* with the pivot and either increments *left*, or swaps the element to *A[right]*.

```
while (left < right)
  //@loop_invariant lower+1 <= left && left <= right && right <= upper;
  //@loop_invariant ge_seg(pivot, A, lower+1, left); // Not lower!
  //@loop_invariant le_seg(pivot, A, right, upper);
  {
    if (A[left] <= pivot) {
      left++;
    } else {
      //@assert A[left] > pivot;
      swap(A, left, right-1);
      right--;
    }
  }
```

Now we just note the observations about the final loop state with an assertion, swap the pivot into place, and return the index *left*. The complete function is on the next page, for reference.

```
int partition(int[] A, int lower, int pivot_index, int upper)
//@requires 0 <= lower && lower <= pivot_index;
//@requires pivot_index < upper && upper <= \length(A);
//@ensures lower <= \result && \result < upper;
//@ensures ge_seg(A[\result], A, lower, \result);
//@ensures le_seg(A[\result], A, \result, upper);
{
  // Hold the pivot element off to the left at "lower"
  int pivot = A[pivot_index];
  swap(A, lower, pivot_index);

  int left = lower+1;
  int right = upper;

  while (left < right)
    //@loop_invariant lower+1 <= left && left <= right && right <= upper;
    //@loop_invariant ge_seg(pivot, A, lower+1, left); // Not lower!
    //@loop_invariant le_seg(pivot, A, right, upper);
    {
      if (A[left] <= pivot) {
        left++;
      } else {
        //@assert A[left] > pivot;
        swap(A, left, right-1);
        right--;
      }
    }
  //@assert left == right;

  swap(A, lower, left-1);
  return left-1;
}
```
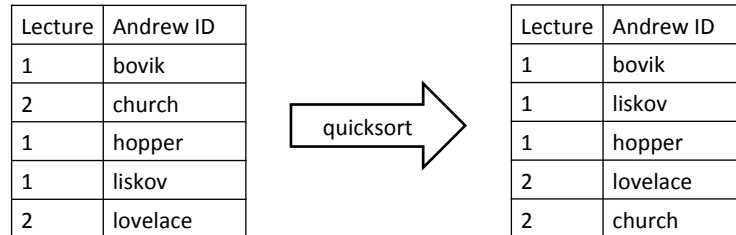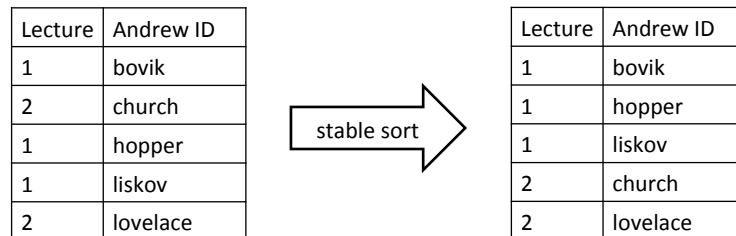
6 Stability

Outside of introductory programming courses, much of the time, we don't sort arrays of integers but arrays of *records*, where the data we are trying to sort (like a lecture number) is associated with another piece of information (like a username). This is also what happens when we tell a spreadsheet to

sort by a particular column: if we take the quicksort algorithm and try to sort the following array by lecture number, we'll get this result:

Lecture	Andrew ID
1	bovik
2	church
1	hopper
1	liskov
2	lovelace

quicksort →

Lecture	Andrew ID
1	bovik
1	liskov
1	hopper
2	lovelace
2	church

The resulting array *is* sorted by lecture number, but whereas `lovelace` appeared after `church` in the unsorted array, they appear in the other order in the sorted array. To put it another way, the first array is sorted by student IDs, we might expect the Andrew IDs *within* Lecture 1 and Lecture 2 to be sorted, but that's not the case.

   If a sort fulfills this expectation – if the relative order in which distinct elements that the sort sees as equivalent, like $(2, \texttt{church})$ and $(2, \texttt{church})$, is preserved by sorting – then the sort is said to be a *stable sort*.

Lecture	Andrew ID
1	bovik
2	church
1	hopper
1	liskov
2	lovelace

stable sort →

Lecture	Andrew ID
1	bovik
1	hopper
1	liskov
2	church
2	lovelace

Selection sort was in-place, stable, but slow. Quicksort was in-place, hopefully fast (with lucky or random pivot selection), but not stable. We'll close out or discussion of sorting with the presentation of a sort that's stable, fast, but not in-place.

7   Mergesort

Let's think more generally about how to apply the divide-and-conquer technique to sorting. How do we divide? In quicksort, the partition function is what divides the problem into two sub-problems, and it's the first thing we do. A characteristic of mergesort is that the *divide* phase of divide-and-conquer is immediate: we only need to calculate the midpoint. On the

other hand, it is (relatively) complicated and expensive (linear in time and temporary space) to combine the results of solving the two independent subproblems with the merging operation.

The simple idea is just to divide a given array in half, sort each half independently. Then we are left with an array where the left half is sorted and the right half is sorted. We then need to *merge* the two halves into a single sorted array. We actually don't really "split" the array into two separate arrays, but we always sort array segments $A[lower..upper)$. We stop when the array segment is of length 0 or 1, because then it must be sorted.

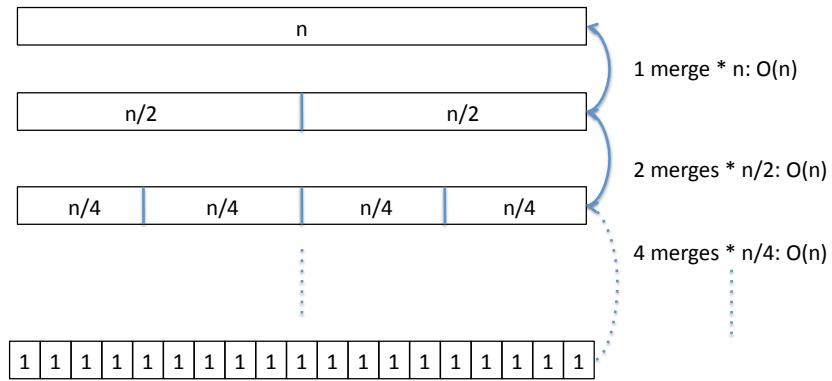A straightforward implementation of this idea would be as follows:

```
void sort (int[] A, int lower, int upper)
//@requires 0 <= lower && lower <= upper && upper <= \length(A);
//@ensures is_sorted(A, lower, upper);
{
  if (upper-lower <= 1) return;
  int mid = lower + (upper-lower)/2;

  sort(A, lower, mid); //@assert is_sorted(A, lower, mid);
  sort(A, mid, upper); //@assert is_sorted(A, mid, upper);
  merge(A, lower, mid, upper);
  return;
}
```

We would still have to write merge, of course, but compare this to the quicksort implementation. They are very similar, but instead of a partition followed by two recursive calls, we have two recursive calls followed by a merge. We use the specification function is_sorted from the last lecture that takes an array segment, defined by its lower and upper bounds.

The simple and efficient way to merge two sorted array segments (so that the result is again sorted) is to create a temporary array, scan each of the segments from left to right, copying the smaller of the two into the temporary array. This is a linear time ($O(n)$) operation, but it also requires a linear amount of temporary space. The merge operation can be seen at mergesort.c0.

Let's consider the asymptotic complexity of mergesort, assuming that the merging operation is $O(n)$.

```
┌─────────────────────────────────────────────┐
│                       n                       │  ⎫
└─────────────────────────────────────────────┘  ⎬  1 merge * n: O(n)
┌──────────────────────┬──────────────────────┐  ⎭
│          n/2          │          n/2          │  ⎫
└──────────────────────┴──────────────────────┘  ⎬  2 merges * n/2: O(n)
┌──────────┬──────────┬──────────┬──────────┐    ⎭
│   n/4    │   n/4    │   n/4    │   n/4    │      ⎫  4 merges * n/4: O(n)
└──────────┴──────────┴──────────┴──────────┘    ⎭

┌─┬─┬─┬─┬─┬─┬─┬─┬─┬─┬─┬─┬─┬─┬─┬─┬─┬─┐
│1│1│1│1│1│1│1│1│1│1│1│1│1│1│1│1│1│1│
└─┴─┴─┴─┴─┴─┴─┴─┴─┴─┴─┴─┴─┴─┴─┴─┴─┴─┘
```

Mergesort, worst case: log(n) levels, O(n) per level

We see that the asymptotic running time will be $O(n\log(n))$, because there are $O(\log(n))$ levels, and on each level we have to perform $O(n)$ operations to merge. The midpoint calculation is deterministic, so this is a worst-case bound.

Exercises

Exercise 1 *In this exercise we explore strengthening the contracts on in-place sorting functions.*

1. *Write a function* is_permutation *which checks that one segment of an array is a permutation of another.*

2. *Extend the specifications of sorting and partitioning to include the permutation property.*

3. *Discuss any specific difficulties or problems that arise. Assess the outcome.*

Exercise 2 *Prove that the precondition for* sort *together with the contract for* partition *implies the postcondition. During this reasoning you may also assume that the contract holds for recursive calls.*

Exercise 3 *Our implementation of partitioning did not pick a random pivot, but took the middle element. Construct an array with seven elements on which our algorithm will exhibit its worst-case behavior, that is, on each step, one of the partitions is empty.*

Exercise 4 *An alternative way of implementing the partition function is to use extra memory for temporary storage. Develop such an implementation of*

```
int partition(int[] A, int lower, int pivot_index, int upper)
//@requires 0 <= lower && lower <= pivot_index;
//@requires pivot_index < upper && upper <= \length(A);
//@ensures lower <= \result && \result < upper;
//@ensures ge_seg(A[\result], A, lower, \result);
//@ensures le_seg(A[\result], A, \result+1, upper);
```