Lecture 7
Quicksort

15-122: Principles of Imperative Computation (Fall 2023)
Frank Pfenning

In this lecture we consider two related algorithms for sorting that achieve a much better running time than the selection sort from an earlier lecture: mergesort and quicksort. We develop 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.

Additional Resources

- Review slides (https://cs.cmu.edu/~15122/handouts/slides/review/07-quicksort.pdf)
- Code for this lecture (https://cs.cmu.edu/~15122/handouts/code/07-quicksort.tgz)

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. Quicksort is *in-place* as it does not require allocating temporary memory, but mergesort is not.

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 trade-off 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 $3^k = (2^{\log_2 3})^k = 2^{\log_2 3 \times k}$, so $\log_2 n$ and $\log_3 n$ only differ in a constant factor, namely $\log_2 3$.

1 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, the elements that are smaller than the pivot and the elements 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

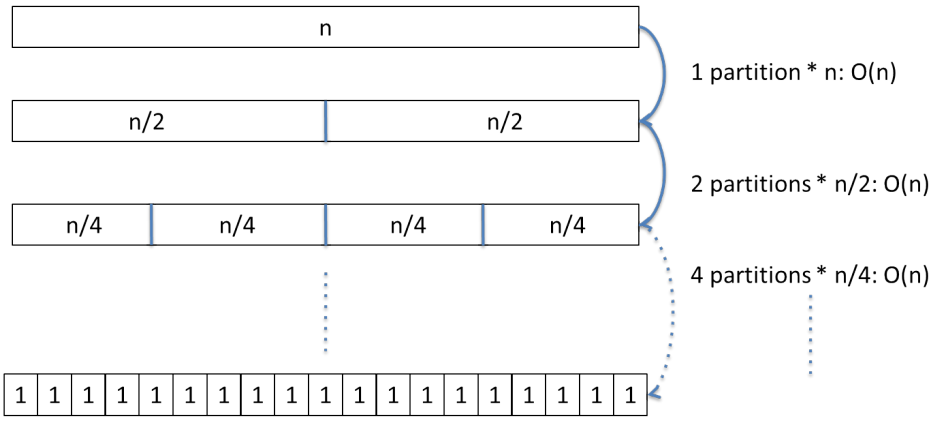
$$3, 1, 4, 4, 7, 2, 8$$

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 k-th 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, the algorithm will run in $O(n \log n)$ time most of the time.

Randomness is very important if we want to claim the algorithm is likely to run in $O(n \log n)$ time. With any fixed-pick strategy, there will be sample 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 $O(n \log n)$. It's still *possible* that we could randomly pick bad pivots, but probabilistically, the chance of this is very, very low. 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.

2 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 a function that modifies a given array instead of creating a new one. It therefore returns no value, which is expressed by writing **`void`** in place of the return type.

```
1 void sort(int[] A, int lo, int hi)
2 //@requires 0 <= lo && lo <= hi && hi <= \length(A);
3 //@ensures is_sorted(A, lo, hi);
4 {
5   ...
6 }
```

Quicksort solves the same problem as selection sort, so their contract is the same, but their implementation differs. We sort the segment $A[lo..hi)$ of the array between lo (inclusively) and hi (exclusively). The precondition in the `@requires` annotation verifies that the bounds are meaningful with respect to A. The post-condition 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.

2. We are allowed to assume that the post-conditions 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 expression $hi - lo$ 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 $hi - lo \leq 1$.

```
1 void sort(int[] A, int lo, int hi)
2 //@requires 0 <= lo && lo <= hi && hi <= \length(A);
3 //@ensures is_sorted(A, lo, hi);
4 {
5   if (hi-lo <= 1) return;
6   ...
7 }
```

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 modify 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:

```
1 int partition(int[] A, int lo, int pi, int hi)
2 //@requires 0 <= lo && lo <= pi;
3 //@requires pi < hi && hi <= \length(A);
4 //@ensures lo <= \result && \result < hi;
5 //@ensures ge_seg(A[\result], A, lo, \result);
6 //@ensures le_seg(A[\result], A, \result+1, hi);
7   ;
```

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, lo, mid)` if $x \geq y$ for every y in $A[lo..mid)$.
- `le_seg(x, A, mid+1, hi)` if $x \leq y$ for every y in $A[mid{+}1..hi)$.

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

Some details on this specification: we require pi to be a valid index in the array range, i.e., $lo \leq pi < hi$. In particular, we require $lo < hi$ 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.

```
1  void sort(int[] A, int lo, int hi)
2  //@requires 0 <= lo && lo <= hi && hi <= \length(A);
3  //@ensures is_sorted(A, lo, hi);
4  {
5    if (hi-lo <= 1) return;
6    int pi = lo + (hi-lo)/2; /* should be random */
7
8    int mid = partition(A, lo, pi, hi);
9    sort(A, lo, mid);
10   sort(A, mid+1, hi);
11   return;
12 }
```

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

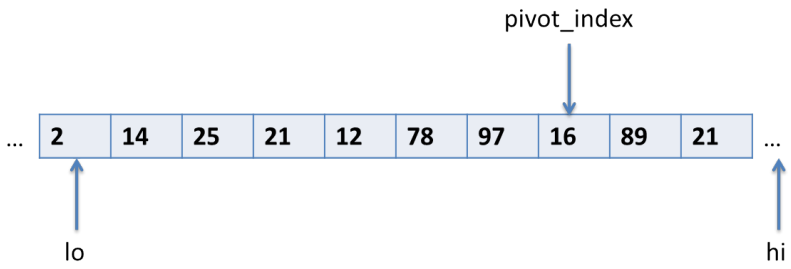
To show that the `sort` function terminates, we have to show the array segment becomes strictly smaller in each recursive call. First, $mid - lo < hi - lo$ since $mid < hi$ by the post-condition for `partition`. Second, $hi - (mid + 1) < hi - lo$ because $lo < mid + 1$, also by the post-condition for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

3 Partitioning

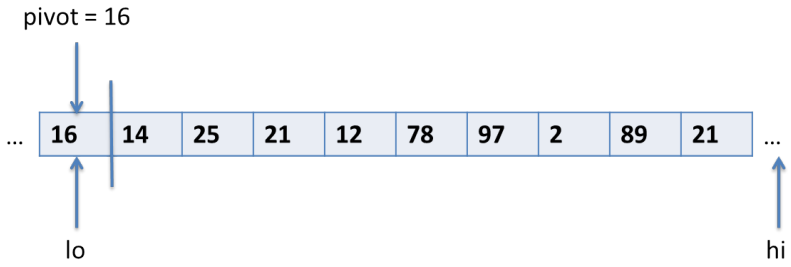
The trickiest aspect of quicksort is the partitioning step. A first idea is to pick the pivot and copy all the other elements of the input array at either side of a temporary array — elements smaller than the pivot towards the left and elements greater than or equal to the pivot towards the right. By the time we are done, one position of the temporary array will be still be open, and that's where we put the pivot. As a final step, we copy the contents of temporary array back into the input array.

This approach requires allocating a new array. Selection sort, by contrast, sorted its input array without the need of allocating a new array. An algorithm that at most allocates constant space is called *in-place*. Selection sort was in-place because it didn't allocate any temporary space. With the above idea for partition, quicksort would not be in-place because each call to partition allocates a new array of the same length of its input.

But with some cleverness, we can write an in-place partition algorithm! A partition algorithm that doesn't allocate new memory. Let's consider the situation when partition is called:

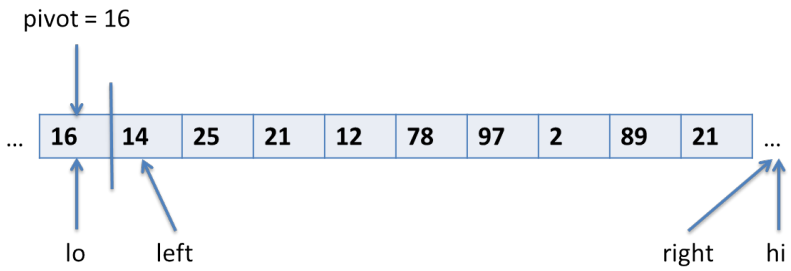


Perhaps the first thing we notice is that we do not know where the pivot will end up in the partitioned array! That's because we don't know how many elements in the segment are smaller and how many are larger than the pivot. In particular, the return value of partition could be different than the pivot 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.[1] 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.

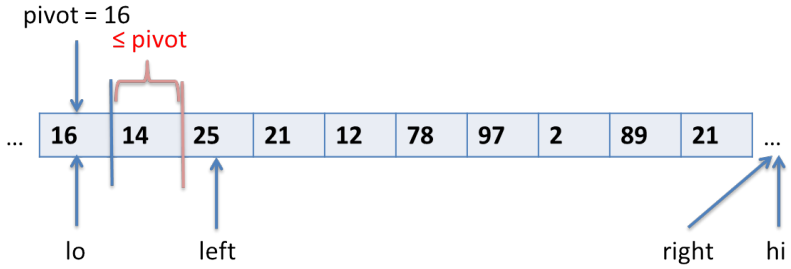


Now the idea is to gradually work towards the middle, accumulating elements less than the pivot on the left and elements greater than the pivot on the right end of the segment (excluding the pivot itself). For this purpose we introduce two indices, $left$ and $right$. We start them out as $lo + 1$ (to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

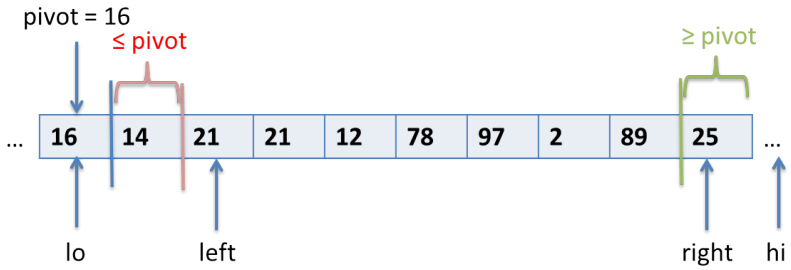
[1] To see why, imagine there are several elements equal to the pivot value.



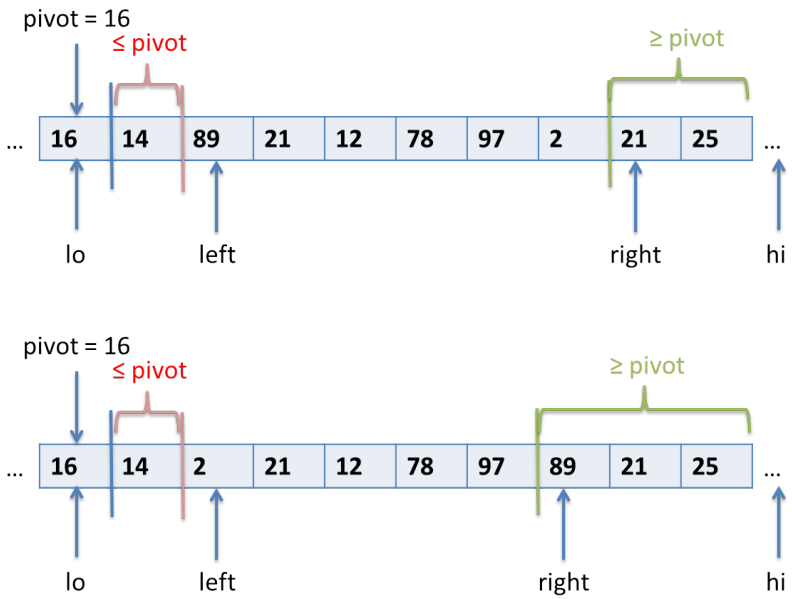
Since $14 < pivot$, we can advance the *left* index: this element is in the proper place.



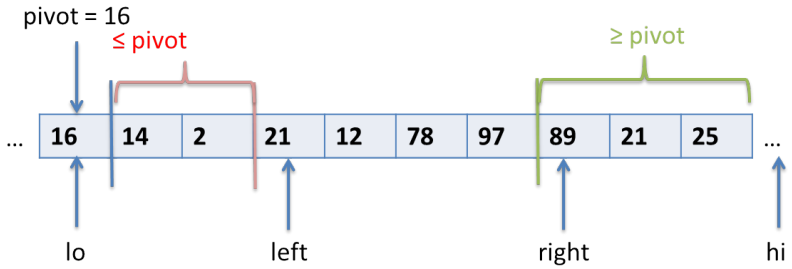
At this point, $25 > pivot$, it needs to go on the right side of the array. If we put it on the *extreme* right end of the array, we can then say that it is in its proper place. We swap it into $A[right - 1]$ and decrement the $right$ index.



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.



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



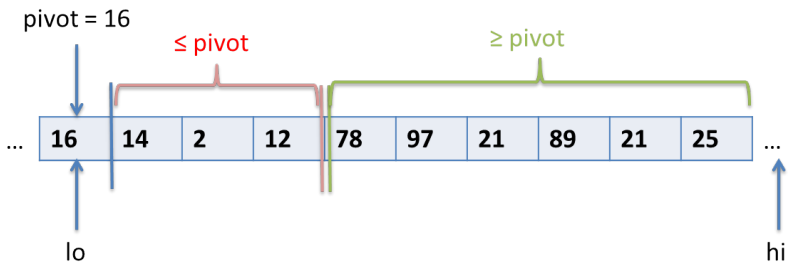
At this point we pause to read off the general invariants which will allow us to synthesize the program. We see:

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

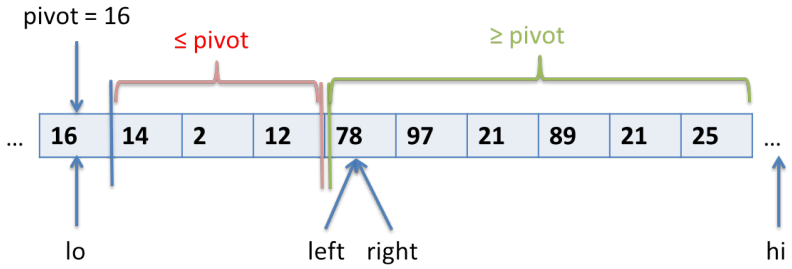
(2) $pivot \leq A[right..hi)$

(3) $A[lo] = pivot$

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 than or equal to *pivot*. To guarantee we are finished, therefore, the *left* index must address the element 78 at $lo + 4$. Similarly, invariant (2) states that the pivot must be less than or equal to all elements starting from *right* up to but excluding *hi*. Therefore, *right* must also address the element 78 at $lo + 4$.



This means after the last iteration, just before we exit the loop, we have $left = right$, and throughout:

(4) $lo + 1 \leq left \leq right \leq hi$

Now comes the last step: since $left = right$, $pivot \geq A[left - 1]$ and we can swap the pivot at lo 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.

4 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.

```
1 int partition(int[] A, int lo, int pi, int hi)
2 //@requires 0 <= lo && lo <= pi && pi < hi && hi <= \length(A);
```

```
3  //@ensures lo <= \result && \result < hi;
4  //@ensures ge_seg(A[\result], A, lo, \result);
5  //@ensures le_seg(A[\result], A, \result+1, hi);
6  {
7    // Hold the pivot element off to the left at "lo"
8    int pivot = A[pi];
9    swap(A, lo, pi);
10   ...
11 }
```

At this point we initialize $left$ and $right$ to $lo + 1$ and hi, 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.

```
12  int left = lo+1;
13  int right = hi;
14
15  while (left < right)
16    //@loop_invariant lo+1 <= left && left <= right && right <= hi;
17    //@loop_invariant ge_seg(pivot, A, lo+1, left); // Not lo!
18    //@loop_invariant le_seg(pivot, A, right, hi);
19    {
20      ...
21    }
```

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

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 $A[right]$.

```
15  while (left < right)
16    //@loop_invariant lo+1 <= left && left <= right && right <= hi;
17    //@loop_invariant ge_seg(pivot, A, lo+1, left); // Not lo!
18    //@loop_invariant le_seg(pivot, A, right, hi);
19    {
20      if (A[left] <= pivot) {
21        left++;
22      } else {
23        //@assert A[left] > pivot;
24        swap(A, left, right-1);
25        right--;
26      }
27    }
```

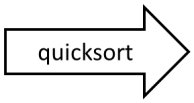
Now we just note the observations about the final loop state with an assertion, swap the pivot into place, and return the index $left - 1$. The complete function is on the next page, for reference.

```
1  int partition(int[] A, int lo, int pi, int hi)
2  //@requires 0 <= lo && lo <= pi;
3  //@requires pi < hi && hi <= \length(A);
4  //@ensures lo <= \result && \result < hi;
5  //@ensures ge_seg(A[\result], A, lo, \result);
6  //@ensures le_seg(A[\result], A, \result, hi);
7  {
8    // Hold the pivot element off to the left at "lo"
9    int pivot = A[pi];
10   swap(A, lo, pi);
11
12   int left = lo+1;
13   int right = hi;
14
15   while (left < right)
16     //@loop_invariant lo+1 <= left && left <= right && right <= hi;
17     //@loop_invariant ge_seg(pivot, A, lo+1, left); // Not lo!
18     //@loop_invariant le_seg(pivot, A, right, hi);
19     {
20       if (A[left] <= pivot) {
21         left++;
22       } else {
23         //@assert A[left] > pivot;
24         swap(A, left, right-1);
25         right--;
26       }
27     }
28   //@assert left == right;
29
30   swap(A, lo, left-1);
31   return left-1;
32 }
```

5 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

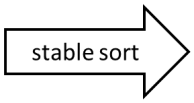


Lecture	Andrew ID
1	bovik
1	liskov
1	hopper
2	lovelace
2	church

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 $(2,$`church`$)$ and $(2,$`lovelace`$)$, 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



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

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

6 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 $A[lo..hi)$. We stop when the array segment is of length 0 or 1, because then it must be sorted.

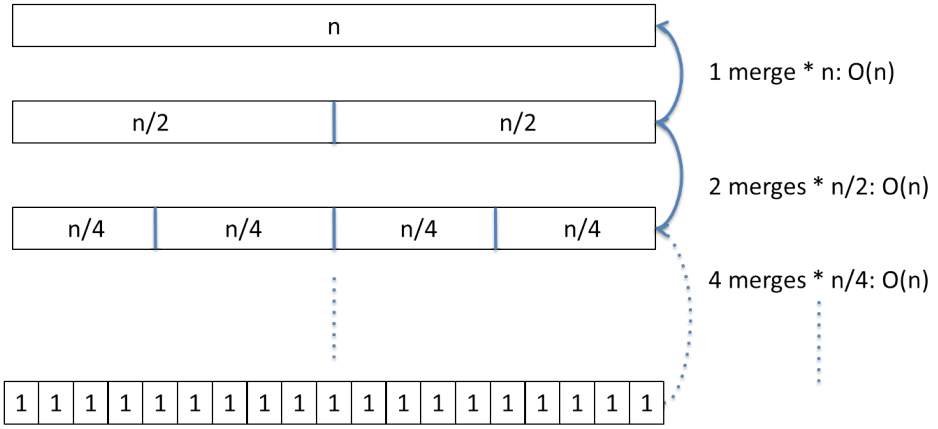
A straightforward implementation of this idea would be as follows:

```
1  void sort (int[] A, int lo, int hi)
2  //@requires 0 <= lo && lo <= hi && hi <= \length(A);
3  //@ensures is_sorted(A, lo, hi);
4  {
5    if (hi-lo <= 1) return;
6    int mid = lo + (hi-lo)/2;
7
8    sort(A, lo, mid); //@assert is_sorted(A, lo, mid);
9    sort(A, mid, hi); //@assert is_sorted(A, mid, hi);
10   merge(A, lo, mid, hi);
11   return;
12 }
```

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. Because of this mergesort is not in-place. The merge operation can be seen in the file `sortlib.c0` included as a part of this lecture's code directory.

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



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.

7 Exercises

Exercise 1. *[sample solution on page 22] In this exercise we explore strengthening the contracts on sorting functions that modify their input array.*

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 (sample solution on page 25)**.** *Prove that the precondition for quicksort's* `sort` *together with the contract for* `partition` *implies the post-condition. During this reasoning you may also assume that the contract holds for recursive calls.*

Exercise 3 (sample solution on page 26)**.** *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 (sample solution on page 26)**.** *An alternative way of implementing the partition function is to use extra memory for temporary storage. Recall the prototype of partition:*

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

Exercise 5 (sample solution on page 29)**.** *Give an example array that shows that selection sort is not stable. A good way to do so is to trace the execution through the iterations of the loop.*

Exercise 6 (sample solution on page 29)**.** *You are given two implementations of quicksort (#1 and #2). They are identical except that one of them always uses the first element of the array segment it is sorted as the pivot, while the other picks a random element of the array segment as the pivot each time. You don't know which implementation is which.*

Based on your knowledge of quicksort, you ask for the timing of both implementations on some sorted and some random arrays. These timings are reported to you in the following two tables:

Num. elements	*#1 runtime (sec)*	*#2 runtime (sec)*
2^{12}	0.09	0.08
2^{14}	0.41	0.43
2^{16}	1.91	1.95
2^{18}	8.75	9.11
2^{20}	37.55	38.20

and

Num. elements	*#1 runtime (sec)*	*#2 runtime (sec)*
2^{12}	0.08	0.42
2^{14}	0.35	3.32
2^{16}	1.73	27.48
2^{18}	7.39	214.39
2^{20}	33.36	1726.54

Which implementation picks the first element of the array segment as the pivot? #1 or #2? Why?

Which set of measurements is for the random array experiment? Why?

Sample Solutions

Solution of exercise 1

The simplest way to check that two array segments are a permutation of one another is to sort both and check that the sorted segments are the same. Doing this directly is unsatisfactory however: sorting the segments would modify them, but we don't want specification functions to modify their input. A way out of this difficulty is to make copies of the array segments, sort these copies and check that the sorted copies are the same. To achieve these goal, we need a few helper functions:

- a function `seg_copy` that copies an array segment into a new array,
- a sorting function — we will use selection sort which we renamed `selsort`, and
- a function `array_equal` which checks that two arrays of the same length have the same element in each position

Here is the code for `seg_copy` and `array_equal` (we saw variants of these functions in past lecture exercises).

```
int[] seg_copy(int[] A, int lo, int hi)
//@requires 0 <= lo && lo <= hi && hi <= \length (A);
{
  int n = hi - lo;
  int[] C = alloc_array(int, n);

  for (int i = 0; i < n; i++)
  //@loop_invariant 0 <= i && i <= n;
  //@loop_invariant lo <= i+lo && i+lo <= hi;
  {
    C[i] = A[lo+i];
  }
  return C;
}

bool array_equal(int[] A, int[] B, int n)
//@requires 0 <= n && n <= \length(A) && n <= \length(B);
{
  for (int i = 0; i < n; i++)
  //@loop_invariant 0 <= i && i <= n;
  {
    if (A[i] != B[i])
      return false;
  }
  return true;
}
```

Then, the code for `is_permutation` is a direct transcription of the approach outlined earlier:

```
bool is_permutation(int[] A, int lo1, int hi1, int[] B, int lo2, int hi2)
//@requires 0 <= lo1 && lo1 <= hi1 && hi1 <= \length (A);
//@requires 0 <= lo2 && lo2 <= hi2 && hi2 <= \length (B);
{
  if (hi1 - lo1 != hi2 - lo2)  // different lengths
    return false;

  int[] A_copy = seg_copy(A, lo1, hi1);
  int[] B_copy = seg_copy(B, lo2, hi2);

  int n = hi1 - lo1;
  //@assert n == hi2 - lo2;

  selsort(A_copy, 0, n);
  selsort(B_copy, 0, n);
  return array_equal(A_copy, B_copy, n);
}
```

Next, we want to use to check that `sort` computes a sorted array that is a permutation of the array it was given in input (as opposed to, say, an array all of whose elements are 0 — which is sorted but unrelated to the input array). Since `sort` updates the input array, we first need to make a copy. Then, just before returning, we check that the sorted array is a permutation of this (copy of the) input array. This is a small hurdle that we overcome using the helper function `seg_copy` once more.

Carrying out the permutation check within an `//@assert` won't work however. This is due to a requirement of contracts called *purity*. A function is *pure* if it does not allocate or modify allocated memory. It is important that contracts be pure otherwise executing a program with and without the `-d` flag may produce different output. The function `is_permutation` is not pure because it extends the allocated memory with two new arrays (via `seg_copy`).

So, we cannot call `is_permutation` within a contract. Second best is to call it using an **`assert`**. Therefore, we can check that the sorted array produced by `sort` is a permutation of the array passed to it as input by bracketing the body of quicksort between the following two lines:

```
int[] C = seg_copy(A, lo, hi);
```

and

```
assert(is_permutation(A, lo, hi, C, 0, hi-lo));
```

We update the code of `partition` similarly.

Summarizing the difficulties we encountered:

- We need to make a copy of the two array segments prior to sorting them to check they are a permutation of each other.
- The sorting function we want to instrument, here quicksort, needs to make a copy of its input array to check that the sorted array is a permutation of the input array.
- Because this approach to checking permutation creates temporary arrays in allocated memory, it is not pure. Therefore, we cannot use it inside an `//@assert` (or any other contract), but must instead call it from an **`assert`** (which is not a contract).

The fact that `is_permutation` is not pure means that it cannot be used in contracts. Therefore it will always be executed when running a program. Can we write a pure version of `is_permutation`? This is possible, and a bonus challenge for this exercise.

Solution of exercise 2 Here's the prototype of `partition` and the code of `sort` for reference:

```
1  int partition(int[] A, int lo, int hi)
2  /*@requires 0 <= lo && lo < hi && hi <= \length(A);  @*/
3  /*@ensures lo <= \result && \result < hi;             @*/
4  /*@ensures ge_seg(A[\result], A, lo, \result);        @*/
5  /*@ensures le_seg(A[\result], A, \result+1, hi);      @*/ ;
6
7  void sort(int[] A, int lo, int hi)
8  //@requires 0 <= lo && lo <= hi && hi <= \length(A);
9  //@ensures is_sorted(A, lo, hi);
10 {
11   if (hi - lo <= 1)
12     return;
13
14   int p = partition(A, lo, hi);
15   sort(A, lo, p);
16   sort(A, p+1, hi);
17 }
```

The function `sort` has two return statements, so we must prove that the postcondition holds in both situations.

Case 1: `sort` returns on line 12

a. `0 <= lo && lo <= hi && hi <= \length(A)` by line 8
b. `hi - lo <= 1` by line 11

c.	`is_sorted(A, lo, hi)`	by (a) and (b) since `A[lo,hi)` has at most one element

Case 2: `sort` returns on line 17

In this case, there are multiple recursive calls to `sort`. We will assume that the postcondition of `sort` is true for these recursive calls (the easy check that their preconditions are met is omitted) and show that the postcondition of `sort` must also be true on line 17.

a.	`A[lo, p) <= A[p]`	by lines 14 and 4
b.	`A[p] <= A[p+1, hi)`	by lines 14 and 5
c.	`is_sorted(A, lo, p)`	by line 15 and A
d.	`is_sorted(A, p+1, hi)`	by line 16 and B
e.	`is_sorted(A, lo, hi)`	by math on (a), (b), (c) and (d)

Solution of exercise 3 Consider the 7-element array `[1, 3, 5, 7, 6, 4, 2]`. Our implementation would do the following, where each step corresponds to one call to `partition`:

1. Pick 7 as the pivot, leaving the arrays `[1, 3, 5, 6, 4, 2]` and `[]` to be sorted.
2. Pick 6 as the pivot, leaving the arrays `[1, 3, 5, 4, 2]` and `[]` to be sorted.
3. Pick 5 as the pivot, leaving the arrays `[1, 3, 4, 2]` and `[]` to be sorted.
4. Pick 4 as the pivot, leaving the arrays `[1, 3, 2]` and `[]` to be sorted.
5. Pick 3 as the pivot, leaving the arrays `[1, 2]` and `[]` to be sorted.
6. Pick 2 as the pivot, leaving the arrays `[1]` and `[]` to be sorted.

Solution of exercise 4 This version of `partition` starts just in the code seen in this chapter, but picking a pivot index `pi` (set arbitrarily to the midpoint of the segment) and, for convenience, swapping its value (the pivot itself) with the leftmost element of the array. Then it diverges.

We create a temporary array `TMP` of length `hi-lo` (just enough to hold all the elements in `A[lo,hi)`) and initialize two indices, `left` and `right`, to `0` and `hi-lo`, respectively. We copy the elements of `A[lo,hi)` that are smaller than or equal to the pivot to the left, and the elements that are larger

than the pivot to the right. At each step, `TMP[left,right)` is the portion of the temporary array we have not filled yet. Once we have processed in this way all the elements of `A[lo,hi)` (except the pivot at `A[lo]`), the temporary array will have exactly one position unfilled, which were we copy the pivot. At this point, all were are left to do is to copy the contents of `TMP` back into `A[lo,hi)`. The resulting code is as follows:

```
1  int partition(int[] A, int lo, int hi)
2  //@requires 0 <= lo && lo < hi && hi <= \length(A);
3  //@ensures lo <= \result && \result < hi;
4  //@ensures ge_seg(A[\result], A, lo, \result);
5  //@ensures le_seg(A[\result], A, \result+1, hi);
6  {
7    int pi = lo + (hi - lo)/2;   // pivot index
8    swap(A, lo, pi);             // move pivot to index 0
9    int pivot = A[lo];           // pivot
10
11   int[] TMP = alloc_array(int, hi-lo);
12   int left = 0;
13   int right = hi-lo;
14
15   for (int i = lo+1; i < hi; i++)
16   //@loop_invariant lo <= i && i <= hi;
17   //@loop_invariant left + (hi - lo - right) == i - (lo+1);
18   //@loop_invariant ge_seg(pivot, TMP, 0, left);
19   //@loop_invariant lt_seg(pivot, TMP, right, hi-lo);
20   {
21     if (A[i] <= pivot) {
22       TMP[left] = A[i];
23       left++;
24     }
25     else {
26       right--;
27       TMP[right] = A[i];
28     }
29   }
30   //@assert left == right - 1;
31   TMP[left] = pivot;
32   //@assert ge_seg(TMP[left], TMP, 0, left);
33   //@assert lt_seg(TMP[left], TMP, right, hi-lo);
34
35   for (int i = 0; i < hi-lo; i++)
36   //@loop_invariant 0 <= i && i <= hi - lo;
37   //@loop_invariant eq_segs(TMP, 0, i, A, lo, lo+i);
38   {
39     A[lo+i] = TMP[i];
40   }
41
42   return lo+left;  // pivot position in A
43 }
```

This function relies on many array indices. The loop invariants pinpoint how these indices are related and what is in the array segments they delimit. The assertions describes this relation at the end of the first loop as well as consequences on the array segments.

In the loop invariants of the second loop , we use the specification function `eq_segs(A, loA, hiA, B, loB, hiB)` to check that two array segments of the same length are equal position by position. If you haven't done so in an earlier exercise, you may want to write it for yourself.

Solution of exercise 5 Recall the code for selection sort:

```
void selsort(int[] A, int lo, int hi)
//@requires 0 <= lo && lo <= hi && hi <= \length(A);
//@ensures is_sorted(A, lo, hi);
{
  for (int i = lo; i < hi; i++)
  //@loop_invariant lo <= i && i <= hi;
  //@loop_invariant is_sorted(A, lo, i);
  //@loop_invariant le_segs(A, lo, i, A, i, hi);
  {
    int m = find_min(A, i, hi);
    swap(A, i, m);
  }
}
```

Consider the integer array $A = [2_a, 2_b, 1]$ which contains two occurrences of 2, which we distinguish using the subscripts $_a$ and $_b$. Let's call `selsort(A,3)` and observe what happens at each iteration.

Iteration 1 (`i == 0`):
`m == 2`, so that $A[m] = 1$.
After the swap, the array A is $[1, 2_b, 2_a]$

Iteration 2 (`i == 1`):
`m == 1`, so that $A[m] = 2_b$
After the swap, the array A is $[1, 2_b, 2_a]$

Iteration 3 (`i == 2`):
`m == 2`, so that $A[m] = 2_a$
After the swap, the array A is $[1, 2_b, 2_a]$

Note that the two occurrences of 2 appear in opposite order compared to the original array A.

Solution of exercise 6 We know that the average time complexity of quicksort is $O(n \log n)$ on an n-element array, which is what we can expect when

called on an array filled with randomly chosen values. The worst-time complexity is instead $O(n^2)$, which we get for example when systematically choosing the first element of the array segment as the pivot and calling quicksort on an array that is already sorted.

Observe that in each of these tables, the size of the input grows 4-fold at each step, from size n to size $4n$ say. For each of these complexity bounds, assume that the time measurement when the input has size n has value k. Let's see what we should expect in each case when the input has size $4n$:

$O(n^2)$**:** Based on the definition of big-O, we know that $k = cn^2$ for some constant c — for an input of size n. The expected timing for an input of size $4n$ is $c(4n)^2 = 16cn^2 = 16k$. Therefore, for an algorithm whose running time is in $O(n^2)$, we expect the measurement to be 16 times larger going from one row of the tables to the next.

$O(n \log n)$**:** In this case, we have that $k = cn \log n$ for some c. Thus, the expected timing for an input of size $4n$ is $c(4n)\log(4n) = 4cn \log n + 4cn \log 2 \leq 8cn \log n = 8k$. Therefore, for an algorithm whose running time is in $O(n \log n)$, we expect the measurement to between 4 and 8 times larger going from one row of the table to the next.

Let's see if the measurement tables support these expectations.

- The measurements in each row of the first table are very similar, which suggests that implementations #1 and #2 have the same effective complexity in this scenario. This can only be the case if the input arrays contain randomly arranged elements — if the elements were sorted, implementations #1 and #2 would take very different times.

  Looking at how the measurements change relative to the size of the input, we notice that in each column the timings grow by a factor of about 4 going from one row to the next. This is fully in line with our expectations for an $O(n \log n)$ complexity.

- The situation is very different in the second table, which necessarily must report on the timing of sorted arrays. Notice that the values in the left column (for implementation #1) are very similar to what we saw in the first table. This indicates that sorting the input array does not influence the timing of implementation #1. Thus, implementation #1 must be the one that chooses the pivot at random at each call to `partition`, yielding an $O(n \log n)$ complexity also in the sorted case.

  To confirm this, let's look at the second column, for implementation #2. The numbers are quite different there! Going from one row to the next, the measurements grow consistently by a factor of about 8

(not quite 16, but significantly more than in the left column or in the first table). This suggests a very different asymptotic behavior which, given our choices we conclude must correspond to $O(n^2)$. Implementation #2 picks the first element of the array as the pivot.

It may be surprising that the measurements in the right column of the second table grow by a factor of about 8 for an $O(n^2)$ complexity. We expected a factor of 16. Recall that big-O notation only highlights the largest contributors to the cost. In particular, lower terms get abstracted away. This may explain this discrepancy.

Summarizing, implementation #1 picked the pivot at random while implementation #2 always used the first element in the array segment. The first experiment involved arrays whose elements were randomly ordered, while the second experiment used ordered arrays.