Lecture Notes on Quicksort

15-122: Principles of Imperative Computation
Frank Pfenning

Lecture 8
February 3, 2011

1 Introduction

In this lecture we revisit the general description of quicksort from last lecture[1] and develop an imperative implementation of it in C0. As usual, contracts and loop invariants will bridge the gap between the abstract idea of the algorithm and its implementation.

2 The Quicksort Algorithm

Quicksort again uses the technique of divide-and-conquer. We proceed as follows:

1. Pick an arbitrary element of the array (the *pivot*).
2. Divide the array into two subarrays, those that are smaller and those that are greater (the *partition* phase).
3. Recursively sort the subarrays.
4. Put the pivot in the middle, between the two sorted subarrays to obtain the final sorted array.

In mergesort, it was easy to divide the input (we just picked the midpoint), but it was expensive to merge the results of the sorting the left and right subarrays. In quicksort, dividing the problem into subproblems could be

[1] omitted from the lecture notes there

computationally expensive (as we analyze partitioning below), 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 such as

$$lt = \{2, 1\}, pivot = 3, geq = \{4, 7, 8, 4\}$$

We have picked an arbitrary order for the elements in the subarrays; 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.

How many recursive calls do we have in the worst case, and how long are the subarrays? 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:

call	$pivot$
$qsort(\{3, 1, 4, 4, 7, 2, 8\})$	1
$qsort(\{3, 4, 4, 7, 2, 8\})$	2
$qsort(\{3, 4, 4, 7, 8\})$	3
$qsort(\{4, 4, 7, 8\})$	4
$qsort(\{4, 7, 8\})$	4
$qsort(\{7, 8\})$	7
$qsort(\{8\})$	

All other recursive calls are with the empty array segment, since we never have any 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 k, which proceeds by partitioning, requiring $O(k)$ comparisons.

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

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

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 always picked 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 turns out that if we pick a *random* element, it will be on average 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 $O(n * \log(n))$. This is an important use of randomness to obtain a reliable average case behavior.

3 The qsort Function

We now turn our attention to developing an imperative implementation of quicksort, following our high-level description. We implement quicksort in the function `qsort` 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 qsort(int[] A, int lower, int upper)
//@requires 0 <= lower && lower <= upper && upper <= \length(A);
//@ensures is_sorted(A, lower, upper);
{
```

```
  ...
}
```

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).

Before we start the body of the function, we should consider how to terminate the recursion. 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 qsort(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 call a partition function. We want partitioning to be done *in place*, modifying the array A. Still, partitioning needs to return the index i of the pivot element because we then have to recursively sort the two subsegments to the left and right of the where the pivot is after partitioning. So we declare:

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

Here we use the auxiliary functions `gt` (for *greater than*) and `leq` (for *less or equal*), where

- `gt(x, A, lower, i)` if $x > y$ for every y in $A[lower..i)$.
- `leq(x, A, i+1, upper)` if $x \leq y$ for very y in $A[i+1..upper)$.

Their definitions can be found in the qsort.c0 file on the course web pages.

Some details on this specification: 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. We ensure that $result < upper$ so that the index of the pivot is a legal index in the segment $A[lower..upper)$.

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

```
void qsort(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 i = partition(A, lower, upper);
  qsort(A, lower, i);
  qsort(A, i+1, upper);
  return;
}
```

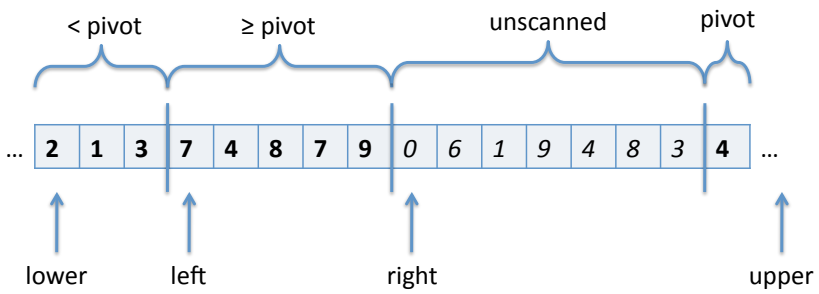
It is a simple but instructive exercise to reason about this program, using only the contract for `partition` together with the preconditions for `qsort` (see Exercise 2).

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

4 Partitioning

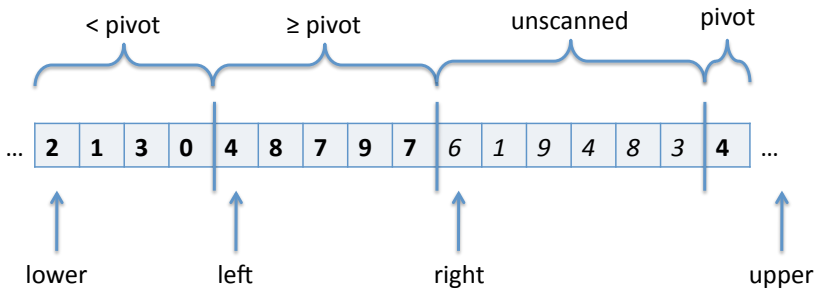
The trickiest aspect of quicksort is the partitioning step, in particular since we want to perform this operation in place. Once we have determined the pivot element, we want to divide the array segment into four different

subsegments as illustrated in this diagram.



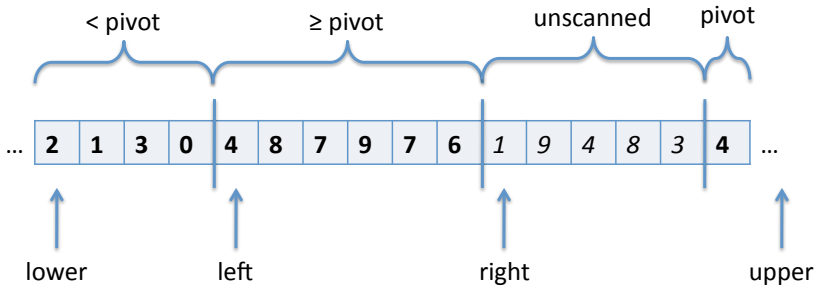
We fix $lower$ and $upper$ as they are when `partition` is called. The segment $A[lower..left)$ contains elements known to be less than the pivot, the segment $A[left..right)$ contains elements greater or equal to the pivot, and the element at $A[upper-1]$ is the pivot itself. The segment from $A[right..upper-1)$ has not yet been scanned, so we don't know yet how these elements compare to the pivot.

We proceed by comparing $A[right]$ with the pivot. In this particular example, we see that $A[right] < pivot$. In this case we swap the element with the element at $A[left]$ and advance both $left$ and $right$, resulting in the following situation:

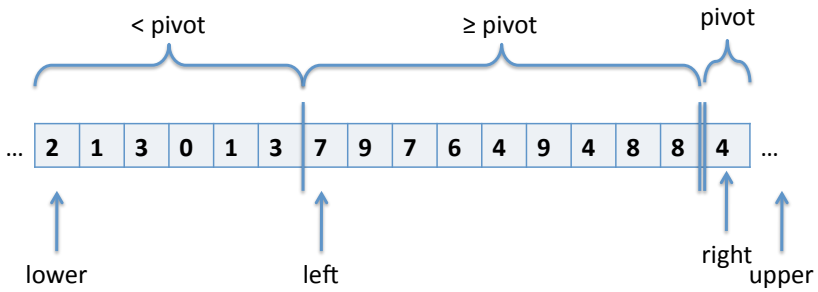


The other possibility is that $A[right] \geq pivot$. In that case we can just advance the $right$ index by one and maintain the invariants without swapping

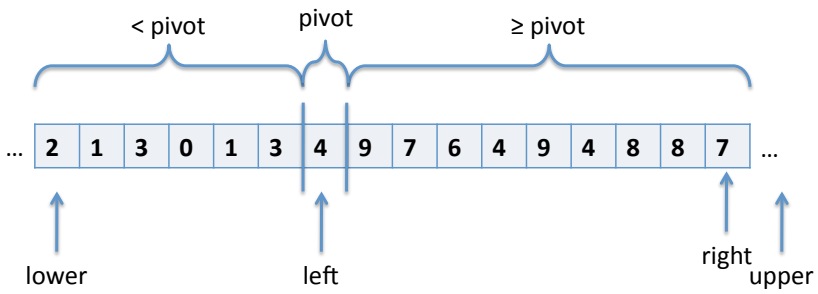
any elements. The resulting situation would be the following.



When $right$ reaches $upper - 1$, the situation will look as follows:



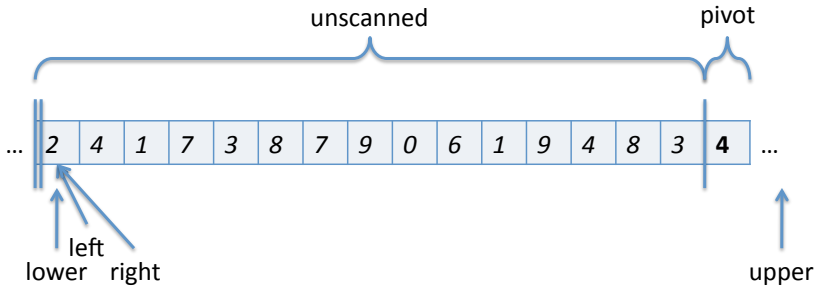
We can now just swap the pivot with $A[left]$, which is known to be greater or equal to the pivot.



The resulting array segment has been partitioned, and we return $left$ as the index of the pivot element.

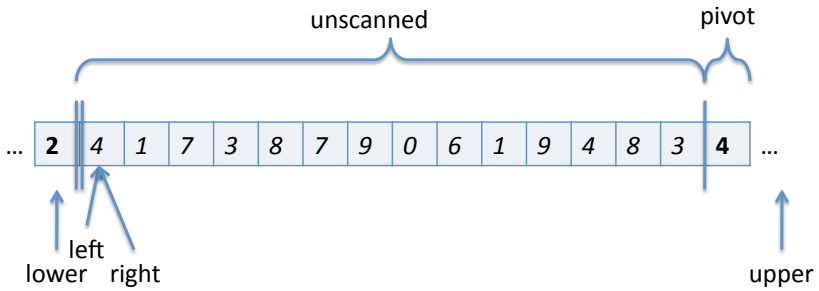
Throughout this process, we have only ever swapped two elements of the array. This guarantees that the array segment after partitioning is a permutation of the segment before.

However, we did not consider how to start this algorithm. We begin by picking a random element as the pivot and then swapping it with the last element in the segment. We then initialize *left* and *right* to *lower*. We then have the situation



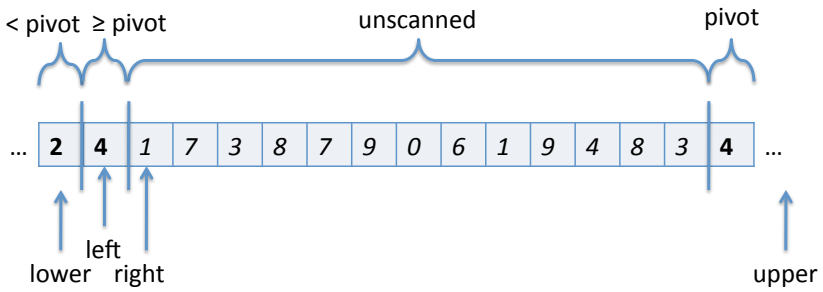
where the two segments with smaller and greater elements than the pivot are still empty.

In this case (where $left = right$), if $A[right] \geq pivot$ then we can increment *right* as before, preserving the invariants for the segments. However, if $A[left] < pivot$, swapping $A[left]$ with $A[right]$ has no effect. Fortunately, incrementing both *left* and *right* preserves the invariant since the element we just checked is indeed less than the pivot.



If *left* and *right* ever separate, we are back to the generic situation we dis-

cussed at the beginning. In this example, this happens in the next step.



If *left* and right always stay the same, all elements in the array segment are strictly less than the pivot, excepting only the pivot itself. In that case, too, swapping $A[left]$ and $A[right]$ has no effect and we return $left = upper - 1$ as the correct index for the pivot after partitioning.

Implementing Partitioning

Now that we understand the algorithm and its correctness proof, it remains to turn these insights into code. We start by computing the index of the pivot and move the pivot to $A[upper - 1]$. To keep the code simple, we take the midpoint of the segment instead of randomly selecting one. This will work well if the array is random, or if it is almost sorted.

```
int partition(int[] A, int lower, int upper)
//@requires 0 <= lower && lower < upper && upper <= \length(A);
//@ensures lower <= \result && \result < upper;
//@ensures gt(A[\result], A, lower, \result);
//@ensures leq(A[\result], A, \result+1, upper);
{
  int pivot_index = lower+(upper-lower)/2;
  int pivot = A[pivot_index];
  swap(A, pivot_index, upper-1);
  ...
}
```

At this point we initialize *left* and *right* to *lower*. We scan the array using the index *right* until it reaches $upper - 1$.

```
int pivot_index = lower+(upper-lower)/2;
int pivot = A[pivot_index];
swap(A, pivot_index, upper-1);
int left = lower;
int right = lower;
while (right < upper-1)
  ...
  {
  }
```

Next, we should turn the observations about the state of the algorithm made in the preceding section into loop invariants. The zeroth one just records the relative position of the indices into the array. The first one states that the pivot is strictly greater than any element in the segment $A[lower..left)$. The second states the the pivot is less or equal any element in the segment $A[left..right)$. The third one expresses that the pivot is stored at $A[upper - 1]$

```
swap(A, pivot_index, upper-1);
int left = lower;
int right = lower;
while (right < upper-1)
  //@loop_invariant lower <= left && left <= right && right < upper;
  //@loop_invariant gt(pivot, A, lower, left);
  //@loop_invariant leq(pivot, A, left, right);
  //@loop_invariant pivot == A[upper-1];
  {
    ...
  }
```

It is easy to verify that the invariants are satisfied initially, given that we also know $lower < upper$ from the function precondition.

In the body of the loop we compare the pivot with $A[right]$ and, in each case, take the appropriate actions described in the previous section.

```
while (right < upper-1)
  //@loop_invariant lower <= left && left <= right && right < upper;
  //@loop_invariant gt(pivot, A, lower, left);
  //@loop_invariant leq(pivot, A, left, right);
  //@loop_invariant pivot == A[upper-1];
  {
    if (pivot <= A[right]) {
      right++;
    } else {
      swap(A, left, right);
      left++;
      right++;
    }
  }
```

Again, it is straightforward to check that the loop invariant is preserved, based on the description in the previous section. It is important to distinguish the special case that $left = right$ when the second invariant (`leq(...)`) is vacuously satisfied.

At the end, we swap $A[left]$ with $A[upper - 1]$ and return $left$ as the index of the pivot in the partitioned arrays. The complete code is on the next page

```
int partition(int[] A, int lower, int upper)
//@requires 0 <= lower && lower < upper && upper <= \length(A);
//@ensures lower <= \result && \result < upper;
//@ensures gt(A[\result], A, lower, \result);
//@ensures leq(A[\result], A, \result+1, upper);
{
  int pivot_index = lower+(upper-lower)/2;
  int pivot = A[pivot_index];
  swap(A, pivot_index, upper-1);
  int left = lower;
  int right = lower;
  while (right < upper-1)
    //@loop_invariant lower <= left && left <= right && right < upper;
    //@loop_invariant gt(pivot, A, lower, left);
    //@loop_invariant leq(pivot, A, left, right);
    //@loop_invariant pivot == A[upper-1];
    {
      if (pivot <= A[right]) {
        right++;
      } else {
        swap(A, left, right);
        left++;
        right++;
      }
    }
  swap(A, left, upper-1);
  return left;
}
```

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* `qsort` *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.*