Lecture Notes on Sorting

15-122: Principles of Imperative Computation
Frank Pfenning

Lecture 7
February 1, 2011

1 Introduction

We have seen in the last lecture that sorted arrays drastically reduce the time to search for an element when compared to unsorted arrays. Asymptotically, it is the difference between $O(n)$ (linear time) and $O(\log(n))$ (logarithmic time), where n is the length of the input array. This suggests that it may be important to establish this invariant, namely sorting a given array. In practice, this is indeed the case: sorting is an important component of many other data structures or algorithms.

There are many different algorithms for sorting: bucket sort, bubble sort, insertion sort, selection sort, heap sort, ... This is testimony to the importance and complexity of the problem, despite its apparent simplicity.

In this lecture we discuss two particularly important sorting algorithms: mergesort and quicksort. Last semester's course instance used mergesort as its main example of efficient sorting algorithms; this time we will use quicksort, giving only a high level overview of mergesort.

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, presumable with an immediately 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 is $O(n)$. 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 Mergesort

Let's see how we can apply the divide-and-conquer technique to sorting. How do we divide?

One simple idea is just to divide a given array in half and 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. This merge operation is a bit more complex so we postpone its detailed discussion to the next section.

To implement the splitting of the array, we pass not only the array, but the subrange we are operating on. We use a specification function `is_sorted(int[] A, int lower, int upper)` to check that the segment $A[lower..upper)$ is sorted.

The other question we have to consider is when can we stop? Obviously, the given range of the array is sorted already if it has size 0 or 1. This leads to the following code:

```
void mergesort (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;
  else {
    int mid = lower + (upper-lower)/2;
    mergesort(A, lower, mid); //@assert is_sorted(A, lower, mid);
    mergesort(A, mid, upper); //@assert is_sorted(A, mid, upper);
    merge(A, lower, mid, upper);
```

```
  }
}
```

This function is intended to *modify* the given segment of the input array in place, rather than returning a new array. That is why the return type is given as `void`, which means it does not actually return a result.

This is an example of *recursion*: a function (`mergesort`) calls itself on a smaller argument. When we analyze such a function we should not try to analyze how the function that we call proceeds 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 postconditions of the function we are calling are satisfied when it returns.

This applies no matter whether the call is recursive, like here, or not. In the `mergesort` code above the precondition is easy to see. We have illustrated the postcondition with two explicit `@assert` annotations.

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.

3 Analysis of Merge

The `merge` function we used above has the following specification. We have two consecutive segments of an array, both non-empty, and both sorted. We guarantee that the union of the two segments is sorted after the merge. We also specify (although this is not expressed in the contract) that the sorted range must be a permutation of the same range when `merge` is called.

```
void merge(int[] A, int lower, int mid, int upper)
//@requires 0 <= lower && lower < mid && mid < upper && upper <= \length(A);
//@requires is_sorted(A, lower, mid) && is_sorted(A, mid, upper);
//@ensures is_sorted(A, lower, upper);
  ;
```

Here is the sketch of a linear-time (that is, $O(upper - lower)$) algorithm for merge. We create a new temporary array of size $upper - lower$. Then we compare the smallest elements of the two ranges we are trying to merge, which must be leftmost in each range. We copy the smaller one of the two to the new array, remove it from consideration, and continue. Eventually, one of the two subranges must become empty. At this point, we just copy the elements from the other, non-empty range without further comparisons.

Finally, we copy the temporary array back to the original array range, where the result is expected.

Each element of the original array is copied, and each of these copies requires at most one preceding comparison, so the asymptotic complexity of the merge is $O(upper - lower)$, the total size of the two subranges to merge. We also need auxiliary space of the same size.

It is this use of auxiliary space which sometimes works against the use of mergesort, even if we can arrange to use the same auxiliary space for all the merge phases.

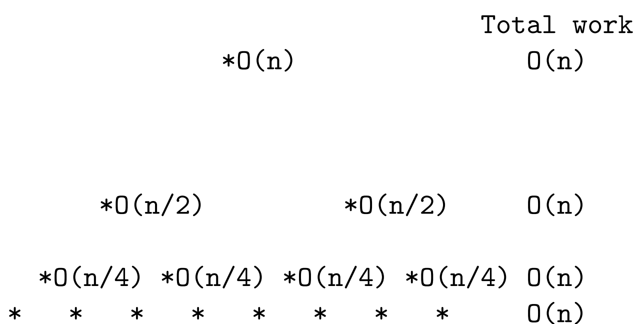
4 Analysis of Mergesort

Before we develop the code for merging, we now analyze the asymptotic complexity of mergesort. As a reminder, here is mergesort:

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

```
}
```

If we call mergesort with an range $(upper - lower)$ of size n, it makes recursive calls of size $n/2$, possibly rounding one or the other down. To avoid special cases and rounding, we just assume that the original range n is a power of 2. Let us draw the tree of recursive calls:



When each pair of recursive calls returns, we have to perform a merge operation which takes $O(m)$, where m is the size of resulting sorted range. For example, at the root node we need $O(n)$ operations to perform the merge. At the next level of recursion, both left and right subrange are of size $n/2$, each requiring $O(n/2)$ operations to merge, which means that the second level in the tree is also $O(n)$. At the next level subranges have size $O(n/4)$, but there are 4 of them, so again processing this layer requires $O(n)$ operations. Overall, there will be $\log(n)$ layers, each layer requiring a total amount of work of $O(n)$, leading to an overall complexity of $O(n * \log(n))$.

It turns out that this is theoretically optimal for certain classes of sorting algorithms, namely those based on comparisons between elements.

5 Programming Merge*

We now turn the algorithmic idea for `merge` into a program, using our method of loop invariants. We start with the following fragment:

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

*Bonus material not covered in lecture. Students are not responsible for this section.

```
  int[] B = alloc_array(int, upper-lower);
  int i = lower; int j = mid; int k = 0;
  ... ?? ...
}
```

We have allocated the auxiliary array B and declared and initialized index variables i (traversing the left range $A[lower..mid)$) and j (traversing the right range $A[mid..upper)$) and k (traversing the target array B). The traversal ranges of these variables become loop invariants. The loop runs as long as i and j are both in their proper range.

```
  int i = lower; int j = mid; int k = 0;
  while (i < mid && j < upper)
    //@loop_invariant lower <= i && i <= mid;
    //@loop_invariant mid <= j && j <= upper;
    //@loop_invariant 0 <= k && k <= upper-lower;
    { ... ?? ... }
```

These invariants are quite weak: they do not express, for example, that we expect the target range $B[0..k)$ to be sorted. We return to refining the invariants in the next section; for now let's concentrate on the code as we wrote it together in lecture.

We copy the smaller of $A[i]$ and $A[j]$ to $B[k]$ and advance the appropriate index i or j, as well as k.

```
  int i = lower; int j = mid; int k = 0;
  while (i < mid && j < upper)
    //@loop_invariant lower <= i && i <= mid;
    //@loop_invariant mid <= j && j <= upper;
    //@loop_invariant 0 <= k && k <= upper-lower;
    {
      if (A[i] <= A[j]) {
        B[k] = A[i]; i++;
      } else {
        B[k] = A[j]; j++;
      }
      k++;
    }
    //@assert i == mid || j == upper;
```

At the end of the loop we know that either $i = mid$ or $j = upper$, because one of the two tests $i < mid$ or $j < upper$ must have failed so we exited the loop.

Now we need to copy the remaining elements in the source array A to B. These are either the elements in $A[i..mid)$ or the elements in $A[j..upper)$. Rather than testing this explicitly, just complete copying both ranges for i and j. One of them will be empty, so one loop body will not be traversed.

```
while (i < mid) { B[k] = A[i]; i++; k++; }
while (j < upper) { B[k] = A[j]; j++; k++; }
```

Finally, we need to copy the temporary array B back into the right range of A.

```
for (k = 0; k < upper-lower; k++) A[lower+k] = B[k];
```

Here is the complete function, missing some invariants.

```
void merge(int[] A, int lower, int mid, int upper)
//@requires 0 <= lower && lower < mid && mid < upper && upper <= \length(A);
//@requires is_sorted(A, lower, mid) && is_sorted(A, mid, upper);
//@ensures is_sorted(A, lower, upper);
{
  int[] B = alloc_array(int, upper-lower);
  int i = lower; int j = mid; int k = 0;
  while (i < mid && j < upper)
    //@loop_invariant lower <= i && i <= mid;
    //@loop_invariant mid <= j && j <= upper;
    //@loop_invariant 0 <= k && k <= upper-lower;
    {
      if (A[i] <= A[j]) {
        B[k] = A[i]; i++;
      } else {
        B[k] = A[j]; j++;
      }
      k++;
    }
  //@assert i == mid || j == upper;
  while (i < mid) { B[k] = A[i]; i++; k++; }
  while (j < upper) { B[k] = A[j]; j++; k++; }
  for (k = 0; k < upper-lower; k++)
    A[lower+k] = B[k];
}
```

6 Strengthening the Invariants

The pre- and post-conditions in the code above look fine, but the loop invariants are very weak. Reasoning through the first loop, the question arises: how do we know that k will remain within the given bounds? The reason is that we always increase either i or j (but never both) and k. Therefore, k cannot change more than the range for i plus the range for j. How can we make this precise as a loop invariant? Think about it before you move on.

We observe that k is always equal to the sum of how far i and j have advanced, that is, $k = (i - lower) + (j - mid)$. Let's replace the weak invariant on k by this stronger one.

```
int i = lower; int j = mid; int k = 0;
while (i < mid && j < upper)
  //@loop_invariant lower <= i && i <= mid;
  //@loop_invariant mid <= j && j <= upper;
  //@loop_invariant k == (i-lower)+(j-mid);
 {
    if (A[i] <= A[j]) {
      B[k] = A[i]; i++;
    } else {
      B[k] = A[j]; j++;
    }
    k++;
  }
```

Now it is easy to see that the invariant on k is preserved, because on each iteration either i and k increase or j and k, keeping the equation in balance. It also follows that the access to $B[k]$ is in bounds, because of the bounds on i and j and loop exit test: i and j start at $lower$ and mid, which means that k starts at 0. And they are bounded above by mid and $upper$, which bounds k by $(mid - lower) + (upper - mid) = upper - lower$.

The same reasoning applies to the two residual loops after the main iteration.

```
int i = lower; int j = mid; int k = 0;
while (i < mid && j < upper)
  //@loop_invariant lower <= i && i <= mid;
  //@loop_invariant mid <= j && j <= upper;
  //@loop_invariant k == (i-lower)+(j-mid);
  {
    if (A[i] <= A[j]) {
      B[k] = A[i]; i++;
    } else {
      B[k] = A[j]; j++;
    }
    k++;
  }
//@assert i == mid || j == upper;
```

```
while (i < mid)
  //@loop_invariant lower <= i && i <= mid;
  //@loop_invariant k == (i-lower)+(j-mid);
  { B[k] = A[i]; i++; k++; }
while (j < upper)
  //@loop_invariant mid <= j && j <= upper;
  //@loop_invariant k == (i-lower)+(j-mid);
  { B[k] = A[j]; j++; k++; }
```

After both loops we know $i = mid$ and $j = upper$, so $k = upper - lower$ and all elements of the target array $A[0..upper - lower)$ have been filled.

The complete code as developed so far can be found in mergesort.c0.

7 Strengthening Loop Invariants Further

Perhaps the code above represents a good intermediate point. It is easy to see that the loop invariants are preserved, and that they guarantee that all array accesses are in bounds.

Still the loop invariants do not mention anything about why the resulting array B is sorted! Why is the range $B[0..k)$ sorted on every iteration? The reason is similar to what we saw in selection sort: the last element $B[k-1]$ is always smaller than both $A[i]$ and $A[j]$. Since we copy the smaller of the two to $B[k]$, this invariant preserved, as is the fact that $B[0..k)$ is sorted. We also need to know that $A[i..mid)$ and $A[j..upper)$ are sorted, but that follows from the fact that the original ranges are sorted. So we add:

```
int i = lower; int j = mid; int k = 0;
while (i < mid && j < upper)
  //@loop_invariant lower <= i && i <= mid;
  //@loop_invariant mid <= j && j <= upper;
  //@loop_invariant k == (i-lower)+(j-mid);
  //@loop_invariant is_sorted(B, 0, k);
  //@loop_invariant is_sorted(A, i, mid) && is_sorted(A, j, upper);
  /*@loop_invariant k == 0 || ((i == mid || B[k-1] <= A[i])
                              && (j == upper || B[k-1] <= A[j])); @*/
 {
    if (A[i] <= A[j]) {
      B[k] = A[i]; i++;
    } else {
      B[k] = A[j]; j++;
```

```
      }
      k++;
    }
```

In the following two loops, the reasons why B remains sorted are similar.

```
  //@assert i == mid || j == upper;
  while (i < mid)
    //@loop_invariant lower <= i && i <= mid;
    //@loop_invariant k == (i-lower)+(j-mid);
    //@loop_invariant is_sorted(B, 0, k) && is_sorted(A, i, mid);
    //@loop_invariant k == 0 || i == mid || B[k-1] <= A[i];
    { B[k] = A[i]; i++; k++; }
  while (j < upper)
    //@loop_invariant mid <= j && j <= upper;
    //@loop_invariant k == (i-lower)+(j-mid);
    //@loop_invariant is_sorted(B, 0, k) && is_sorted(A, j, upper);
    //@loop_invariant k == 0 || j == upper || B[k-1] <= A[j];
    { B[k] = A[j]; j++; k++; }
```

Because $i = mid$ or $j = upper$, only one of these loop will execute and therefore, finally, the array $B[0..upper - lower)$ will be sorted.

We see that the computation is relatively straightforward, but the reason for its correctness is somewhat complicated, if it is made explicit. It is a matter of taste and experience how much of the reasons one makes explicit in the invariants.

```
void merge(int[] A, int lower, int mid, int upper)
//@requires 0 <= lower && lower < mid && mid < upper && upper <= \length(A);
//@requires is_sorted(A, lower, mid) && is_sorted(A, mid, upper);
//@ensures is_sorted(A, lower, upper);
{
  int[] B = alloc_array(int, upper-lower);
  int i = lower; int j = mid; int k = 0;
  while (i < mid && j < upper)
    //@loop_invariant lower <= i && i <= mid;
    //@loop_invariant mid <= j && j <= upper;
    //@loop_invariant k == (i-lower)+(j-mid);
    //@loop_invariant is_sorted(B, 0, k);
    //@loop_invariant is_sorted(A, i, mid) && is_sorted(A, j, upper);
    /*@loop_invariant k == 0 || ((i == mid || B[k-1] <= A[i])
                             && (j == upper || B[k-1] <= A[j])); @*/
    {
      if (A[i] <= A[j]) {
        B[k] = A[i]; i++;
      } else {
        B[k] = A[j]; j++;
      }
      k++;
    }
  //@assert i == mid || j == upper;
  while (i < mid)
    //@loop_invariant lower <= i && i <= mid && k == (i-lower)+(j-mid);
    //@loop_invariant is_sorted(B, 0, k) && is_sorted(A, i, mid);
    //@loop_invariant k == 0 || i == mid || B[k-1] <= A[i];
    { B[k] = A[i]; i++; k++; }
  while (j < upper)
    //@loop_invariant mid <= j && j <= upper && k == (i-lower)+(j-mid);
    //@loop_invariant is_sorted(B, 0, k) && is_sorted(A, j, upper);
    //@loop_invariant k == 0 || j == upper || B[k-1] <= A[j];
    { B[k] = A[j]; j++; k++; }
  //@assert k == upper-lower && is_sorted(B, 0, upper-lower);
  for (k = 0; k < upper-lower; k++)
    //@loop_invariant lower <= lower+k && lower+k <= upper;
    A[lower+k] = B[k];
}
```

Here, we have omitted the precise reasoning on why $A[lower..upper)$ is sorted after the copy operation, because it is evidently a copy of B which is known to be sorted.

It also emerges that it might have been easier, from the specification perspective, to have just one loop instead of three. We leave such a rephrasing as an exercise. The complete code as developed above can be found in mergesort-invs.c0.