Lecture Notes on Sorting

15-122: Principles of Imperative Computation
Frank Pfenning

Lecture 4
September 2, 2010

1 Introduction

Algorithms and data structures can be evaluated along a number of dimensions, using a number of different techniques. We can evaluate them experimentally, for example, determining the average running time over a random set of sample inputs. Or we can evaluate them mathematically, for example, bounding the number of operations a program has to perform in the worst case.

The simplest experimental evaluation is beginning-to-end (often called *end-to-end*): we just run the program with certain parameters and observe the so-called *wall-clock time*. Another technique is *cycle counting*, which you will learn about in the *Computer Systems* class, which provides much more fine-grained information. Another important technique is *profiling*, which determines the relative significance of the function in a larger system for the running time of a program. Besides time, we are often interested in space usage of programs, and we will have more to say about that later in this class.

In this lecture we will look at the mathematical analysis of the running time of a function, trying to distill the most important characteristic that let's us easily compare different algorithms and, to some extent, predict practical performance. We will do this in the context of the searching algorithms from the last lecture and we will also study some sorting algorithms.

Like search, sorting is one of the most fundamental algorithmic problems used in applications. We have already seen that search can be very efficient if our data is sorted, but how do we get it into sorted form? There

are myriads of algorithms, but only a few of real practical importance. We will discuss and analyze a couple of them in this course, a first one called *selection sort* in this lecture.

2 Big-O Notation

In order to compare the space or time use of algorithms, there are two fundamental principles that guide our mathematical analysis.

1. We only care about the behavior of an algorithm *in the long run*, that is, on larger and larger inputs. It is when the inputs are large that differences between algorithms become really pronounced. For example, linear search on a 10-element array will be practically the same as binary search on a 10-element array, but once we have an array of, say, a million entries the difference will be huge.

2. We do not care about *constant factors* in the mathematical analysis. For example, in analyzing the search algorithms we count how often we have to iterate, not exactly how many operations we have to perform on each iteration. In practice, constant factors make a big difference, but they are influenced by so many factors (compiler, runtime system, machine model, available memory, etc.) that at the abstract, mathematical level a precise analysis is neither appropriate nor feasible.

Let's see how these two fundamental principles guide us in the comparison between functions that measure the running time of an algorithm.

Let's say we have functions f and g that measure the number of operations of an algorithm as a function of the size of the input. For example $f(n) = 3 * n$ measures the number of comparisons performed in linear search for an array of size n, and $g(n) = 3 * \log(n)$ measures the number of comparisons performed in binary search for an array of size n.

The simplest form of comparison would be

$g \leq_0 f$ *if for every* $n \geq 0$, $g(n) \leq f(n)$.

However, this violates principle (1) because we compare the values and g and f on all possible inputs n.

We can refine this by saying that *eventually*, g will always be smaller or equal to f. We express "eventually" by requiring that there be a number n_0 such that $g(n) \leq f(n)$ for all n that are greater than n_0.

> $g \leq_1 f$ *if there is some* n_0 *such that for every* $n \geq n_0$ *we have* $g(n) \leq f(n)$.

This now incorporates the first principle (we only care about the function in the long run, on large inputs), but constant factors still matter. For example, according to the last definition we have $3 * n \leq_1 5 * n$ but $5 * n \not\leq_1 3 * n$. But if constants factors don't matter, then the two should be equivalent. We can repair this by allowing the right-hand side to be multiplied by an arbitrary constant.

> $g \leq_2 f$ if there is a constant $c > 0$ and some n_0 such that for every $n \geq n_0$ we have $g(n) \leq c * f(n)$.

This definition is now appropriate.

The less-or-equal symbol $\leq$ is already overloaded with many meanings, so we write instead:

> $g \in O(f)$ if there is a constant $c > 0$ and some n_0 such that for every $n \geq n_0$ we have $g(n) \leq c * f(n)$.

This notation derives from the view of $O(f)$ as a set of functions, namely those that eventually are smaller than a constant times f.[1] Just to be explicit, we also write out the definition of $O(f)$ as a set of functions:

$$O(f) = \{g \mid \text{there are } c > 0 \text{ and } n_0 \text{ s.t. for all } n \geq n_0,\ g(n) \leq c * f(n)\}$$

With this definition we can check that $O(f(n)) = O(c * f(n))$.

When we characterize the running time of a function using big-O notation we refer to it as the *asymptotic complexity* of the function. Here, *asymptotic* refers to the fundamental principles listed above: we only care about the function in the long run, and we ignore constant factors.

The asymptotic time complexity of linear search is $O(n)$, which we also refer to as *linear time*. The asymptotic time complexity of binary search is $O(\log(n))$, which we also refer to as *logarithmic time*. *Constant time* is usually described as $O(1)$, expressing that the running time is independent of the size of the input.

Some brief fundamental facts about big-O. For any polynomial, only the highest power of n matters, because it eventually comes to dominate the function. For example, $O(5{*}n^2{+}3{*}n{+}83) = O(n^2)$. Also $O(\log(n)) \subseteq O(n)$, but $O(n) \not\subseteq O(\log(n))$. Logarithms to different (constant) bases are asymptotically the same: $O(\log_2(n)) = O(\log_b(n))$ because $\log_b(n) = \log_2(n)/\log_2(b)$.

[1] In textbooks and research papers you may sometimes see this written as $g = O(f)$ but that is questionable, comparing a function with a set of functions.

As a side note, it is mathematically correct to say the running time of binary search is $O(n)$, because $\log(n) \in O(n)$. It is, however, a looser characterization than saying that the running time of binary search is $O(\log(n))$, which is also correct. Of course, it would be incorrect to say that the running time is $O(1)$. Generally, when we ask you to characterize the worst-case running time of an algorithm we are asking for the tightest bound in big-O notation.

3 Selection Sort

The idea of selection sort is quite simple. On the first pass we find the minimal element in the whole array and put it into position 0. On the second pass we find the minimal element in the subarray starting at 1 and put it into position 1. In general, on pass $i + 1$ we find the smallest element in the subarray starting at i and put it into position i. The array will be sorted after $n - 1$ passes.

4 Analyzing Selection Sort

Before we start implementing selection sort, let's consider its asymptotic complexity. On the first pass, we have to find the minmal element in an array of size n. This takes $n - 1$ comparisons, each one with the one that is known to be minimal so far. On the next iteration, we take $n - 2$ comparisons to find the minimal elements in the subarray from 1 to $n-1$. After the last pass, we will have made $(n-1)+(n-2)+\cdots+1$ comparisons, which is $n * (n - 1)/2$. This function of n is in $O(n^2)$, so we expect the complexity of selection sort to be quadratic. If we double the size of the input, we expect the running time to be about four times as long.

5 Implementing Selection Sort

Since we have to sort subranges of an array, we generalize the function to take a lower and upper bound, and sort the subarray starting at $lower$ and ending with $upper - 1$. We correspondingly generalize the `is_sorted` function to take a lower and upper bound.

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

```
{ int i;
  for (i = lower; i < upper-1; i++)
    //@loop_invariant lower == upper || (lower <= i && i <= upper-1);
    if (!(A[i] <= A[i+1])) return false;
  return true;
}

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

For the explanations, we introduce a new notation. In analogy with the mathematical notation for intervals, we write $A[i..j]$ for the range $A[i], A[i+1], \ldots, A[j]$. This is a single element if $i = j$. We also write $A[i..n)$ for the range $A[i], A[i+1], \ldots, A[n-1]$, excluding the right end of the interval.

Writing the outer loop is straightforward. For simplicity in the invariants we increment i all the way to $upper$ rather than $upper - 1$, costing us a few irrelevant operations at the end of the outer loop.

```
void selsort(int[] A, int lower, int upper)
//@requires 0 <= lower && lower <= upper && upper <= \length(A);
//@ensures is_sorted(A, lower, upper);
{ int i;
  for (i = lower; i < upper; i++)
    //@loop_invariant lower <= i && i <= upper;
    //@loop_invariant ...??...
    { ... }
}
```

What makes this algorithm work? As usual, if we can express this as an appropriate loop invariant, the body of the loop will almost write itself.

The first observation is that the subarray from $A[lower..i)$ to will be sorted. Then, when we exit the loop, $i = upper$ and the whole range $A[lower..upper)$ will be sorted.

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

```
  for (i = lower; i < upper; i++)
    //@loop_invariant lower <= i && i <= upper;
    //@loop_invariant is_sorted(A, lower, i);
    //@loop_invariant ...??...
    { ... }
}
```

This invariant is not yet strong enough as a loop invariant. Assume we can find the minimal element x in $A[i..upper)$. If we move x to $A[i]$, why are the elements $A[lower..i]$ now sorted? It is because we know from the previous pass that $A[i-1]$ must be less or equal to all elements in $A[i..upper)$. But the invariant does not express this so far.

In order to express this we write an auxiliary function `leq`, intended only for the invariants, where `leq(x, A, lower, upper)` returns `true` if x is less than all elements in $A[lower..upper)$.

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

Now we only have to take into account the boundary case of the first iteration, where $i - 1 = lower - 1$ would be out of the range we want to consider, and possibly even out of bounds (if $lower = 0$).

```
void selsort(int[] A, int lower, int upper)
//@requires 0 <= lower && lower <= upper && upper <= \length(A);
//@ensures is_sorted(A, lower, upper);
{ int i;
  for (i = lower; i < upper; i++)
    //@loop_invariant lower <= i && i <= upper;
    //@loop_invariant is_sorted(A, lower, i);
    //@loop_invariant i == lower || leq(A[i-1], A, i, upper);
    { ... }
}
```

All that remains is to write the body of the outer loop, that is, the inner loop. As our first approximation we pick $A[i]$ as the minimal element.

Throughout the loop, we maintain the variable min as the index of the minimal element in the range $A[i..j)$, where j is the index variable of the inner loop.

```
void selsort(int[] A, int lower, int upper)
//@requires 0 <= lower && lower <= upper && upper <= \length(A);
//@ensures is_sorted(A, lower, upper);
{ int i;
  for (i = lower; i < upper; i++)
    //@loop_invariant lower <= i && i <= upper;
    //@loop_invariant is_sorted(A, lower, i);
    //@loop_invariant i == lower || leq(A[i-1], A, i, upper);
    { int min = i; int j;
      for (j = i+1; j < upper; j++)
        //@loop_invariant i+1 <= j && j <= upper;
        //@loop_invariant leq(A[min], A, i, j);
        if (A[j] < A[min]) min = j;
      //@assert leq(A[min], A, i, upper);
      ...??...
    }
}
```

After the inner loop $A[min]$ is the smallest element in $A[i..upper)$, which follows easily from the invariants. Now we want to move it into its final place, name $A[i]$. Here we remember an important observation about sorting. We would like the array after sorting to be a *permutation* of the array before. A permutation is defined as a sequence of exchanges, and we can obtain any permutation just by swapping elements in the array with each other. So, in order to move $A[min]$ into the correct position at $A[i]$, we just have to swap the elements at $A[i]$ and $A[min]$. In order emphasize this, we write a function to perform this swap, where `swap(A, i, j)` exchanges elements i and j in the array (see below). Then we just call this function to swap $A[i]$ with $A[min]$.

```
void selsort(int[] A, int lower, int upper)
//@requires 0 <= lower && lower <= upper && upper <= \length(A);
//@ensures is_sorted(A, lower, upper);
{ int i;
  for (i = lower; i < upper; i++)
    //@loop_invariant lower <= i && i <= upper;
    //@loop_invariant is_sorted(A, lower, i);
    //@loop_invariant i == lower || leq(A[i-1], A, i, upper);
    { int min = i; int j;
      for (j = i+1; j < upper; j++)
        //@loop_invariant i+1 <= j && j <= upper;
        //@loop_invariant leq(A[min], A, i, j);
        if (A[j] < A[min]) min = j;
      //@assert leq(A[min], A, i, upper);
      // swap A[i] and A[min]
      swap(A, i, min);
    }
}
```

This completes selection sort.

6 Exchanging Elements

Selection sort as we have presented it is an *in-place* algorithm in that it modifies its input to constitute the output. This means it is correct to claim in the postcondition that `is_sorted(A, lower, upper)` even though A is *not* sorted when the function is called.

Our contract does not actually require that the array after the sort is a permutation of the array before the sort. This is difficult to specify for two reasons. First, we would have to retain somewhere a copy of the original array, so that we can compare the array before to the array after. Second, determining if one array is a permutation of another is actually a computationally complex problem, roughly of the same difficulty as sorting the array. Instead, we leave this property implicit, within comments. Fortunately, it is easy to verify here. The only operation that modifies the array at all is with the function `swap`. So we maintain at any point in the function `selsort` that the array A is a permutation of the array at the beginning.

The contract for `swap` is not easy to express. We would like to say `swap(A, i, j)` modifies *only* $A[i]$ and $A[j]$, which remains a comment since

it is difficult to check dynamically. We would also like to say that *after* the swap, $A[i]$ is $A[j]$ from before, and $A[j]$ is $A[i]$ from before. For this we have a primitive expression `\old(e)` that can be used only in postconditions. It evaluates the expression e in the state when the function is first called and stores its value to a temporary variable. It then uses this variable in place of `\old(e)`. With this, we can specify and write `swap`.

```
void swap(int[] A, int i, int j)
//@requires 0 <= i && i < \length(A);
//@requires 0 <= j && j < \length(A);
// modifies A[i], A[j];  // just a comment
//@ensures A[i] == \old(A[j]) && A[j] == \old(A[i]);
{ int tmp = A[i];
  A[i] = A[j];
  A[j] = tmp;
}
```