Lecture Notes on Sorting

15-122: Principles of Imperative Computation
Frank Pfenning

Lecture 7
February 5, 2013

1 Introduction

We begin this lecture by discussing how to compare running times of functions in an abstract, mathematical way. The same underlying mathematics can be used for other purposes, like comparing memory consumption or the amount of parellism permitted by an algorithm. We then use this to take a first look at sorting algorithms, of which there are many. In this lecture it will be selection sort because of its simplicity.

In terms of our learning goals, we will work on:

Computational Thinking: Still trying to understand how order can lead to efficient computation. Worst-case asymptotic complexity of functions.

Algorithms and Data Structures: In-place sorting of arrays in general, and selection sort in particular. Big-O notation.

Programming: More examples of programming with arrays and algorithm invariants.

2 Big-O Notation

Our brief analysis in the last lecture already indicates that linear search should take about n iterations of a loop while binary search take about $\log_2(n)$ iterations, with a constant number of operations in each loop body. This suggests that binary search should more efficient. In the design and

analysis of algorithms we try to make this mathematically precise by deriving so-called *asymptotic complexity measures* for algorithms. There are two fundamental principles that guide our mathematical analysis.

1. We only care about the behavior of an algorithm *on large inputs*, that is, when it takes a long time. 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 can 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* $n \geq n_0$ *it is the case that* $g(n) \leq f(n)$.

This now incorporates the first principle (we only care about the function 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. Usually, we use an analysis of the *worst case* among the inputs of a given size. Trying to do *average case* analysis is much harder, because it depends on the distribution of inputs. Since we often don't know the distribution of inputs it is much less clear whether an average case analysis may apply in a particular use of an algorithm.

The asymptotic worst-case time complexity of linear search is $O(n)$, which we also refer to as *linear time*. The worst-case 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 $O(n) \not\subseteq O(\log(n))$.

[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.

That is the same as to say $O(\log(n)) \subsetneq O(n)$, which means that $O(\log(n))$ is a proper subset of $O(n)$, that is, $O(\log(n))$ is a subset ($O(\log(n)) \subseteq O(n)$), but they are not equal ($O(\log(n)) \neq O(n)$). Logarithms to different (constant) bases are asymptotically the same: $O(\log_2(n)) = O(\log_b(n))$ because $\log_b(n) = \log_2(n)/\log_2(b)$.

As a side note, it is mathematically correct to say the worst-case 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 Sorting Algorithms

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.

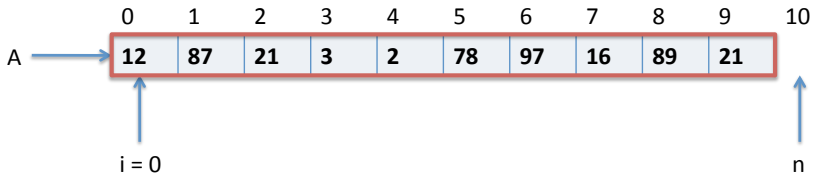
There are many different algorithms for sorting: bucket sort, bubble sort, insertion sort, selection sort, heap sort, etc. This is testimony to the importance and complexity of the problem, despite its apparent simplicity.

In this lecture we discuss selection sort, which is one of the simplest algorithms. In the next lecture we will discuss quicksort. Earlier course instances used mergesort as another example of efficient sorting algorithms.

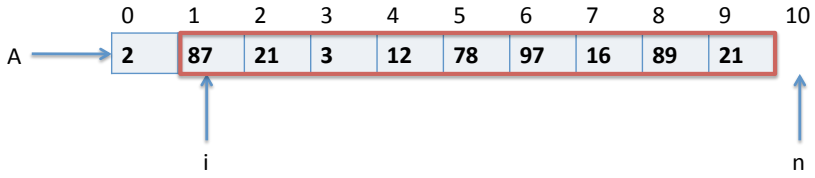
4 Selection Sort

Selection sort is based on the idea that on each iteration we select the *smallest* element of the part of the array that has not yet been sorted and move it to the end of the sorted part at the beginning of the array.

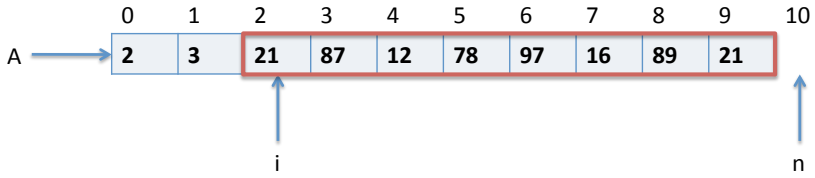
Let's play this through for two steps on an example array. Initially, we consider the whole array (from $i = 0$ to the end). We write this as $A[0..n)$, that is the segment of the array starting at 0 up to n, where n is *excluded*.



We now find the minimal element of the array segment under consideration (2) and move it to the front of the array. What do we do with the element that is there? We move it to the place where 2 was (namely at $A[4]$). In other words, we *swap* the first element with the minimal element. Swapping is a useful operation when we sorting an array *in place* by modifying it, because the result is clearly a permutation of the input. If swapping is our *only* operation we are immediately guaranteed that the result is a permutation of the input.



Now 2 is in the right place, and we find the smallest element in the remaining array segment and move it to the beginning of the segment ($i = 1$).



Let's pause and see if we can write down properties of the variables and array segments that allow us to write the code correctly. First we observe rather straightforwardly that

$$0 \leq i \leq n$$

where $i = n$ after the last iteration and $i = 0$ before the first iteration. Next we observe that the elements to the left of i are already sorted.

$$A[0..i) \quad \text{sorted}$$

These two invariants are not yet sufficient to prove the correctness of selection sort. We also need to know that *all* elements to the left of i are less or equal to *all* element to the right of i. We abbreviate this:

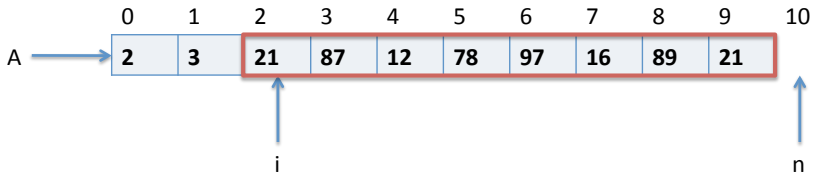
$$A[0..i) \leq A[i..n)$$

saying that every element in the left segment is smaller than or equal to every element in the right segment.

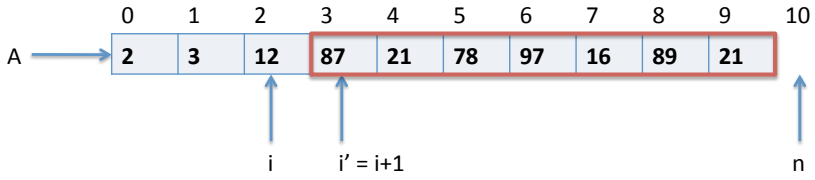
We summarize the invariants

$$0 \leq i \leq n$$
$$A[0..i) \quad \text{sorted}$$
$$A[0..i) \leq A[i..n)$$

Let's reason through *without any code* (for the moment), why these invariants are preserved. Let's look at the picture again.



In the next iteration we pick the minimal element among $A[i..n)$, which would be $12 = A[4]$. We now swap this to $i = 2$ and increment i. We write here $i' = i + 1$ in order to distinguish the old value of i from the new one, as we do in proofs of preservation of the loop invariant.



Since we only step when $i < n$, the bounds on i are preserved.

Why is $A[0..i{+}1)$ sorted? We know by the third invariant that any element in $A[0..i)$ is less than any element in $A[i..n)$ and in particular the one we moved to $A[i{+}1]$.

Why is $A[0..i{+}1) \leq A[i{+}1..n)$? We know from the loop invariant before the iteration that $A[0..i) \leq A[i{+}1..n)$. So it remains to show that $A[i..i{+}1] \leq A[i{+}1..n)$. But that is true since $A[i]$ was a minimal element of $A[i..n)$ which is the same as saying that it is smaller or equal to all the elements in $A[i..n)$ and therefore also $A[i{+}1..n)$ after we swap the old $A[i]$ into its new position.

5 Programming Selection Sort

From the above invariants and description of the algorithm, the correct code is simple to write, including its invariants. The function does not return a value, since it modifies the given array A, so it has declaration:

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

We encourage you to now write the function, using the following auxiliary and contract functions:

1. `is_sorted(A, lower, upper)` which is true if the array segment $A[lower..upper)$ is sorted.
2. `le_seg(x, A, lower, upper)` which is true if $x < A[lower_1..upper_1)$ (which means all x is less than or equal to all elements in the array segment).
3. `le_segs(A, lower1, upper1, lower2, upper2)` which is true if $A[lower_1..upper_1) \leq A[lower_2..upper_2)$ (which means all elements in the first segment are less or equal to the all elements in the second array segment).
4. `swap(A, i, j)` modifies the array A by swapping $A[i]$ with $A[j]$. Of course, if $i = j$, the array remains unchanged.
5. `min_index(A, lower, upper)` which returns the index m of a minimal element in the segment $A[lower..upper)$.

Please write it and then compare it to our version on the next page.

```
void sort(int[] A, int lower, int upper)
//@requires 0 <= lower && lower <= upper && upper <= \length(A);
//@ensures is_sorted(A, lower, upper);
{
  for (int i = lower; i < upper; i++)
    //@loop_invariant lower <= i && i <= upper;
    //@loop_invariant is_sorted(A, lower, i);
    //@loop_invariant le_segs(A, lower, i, i, upper);
    {
      int m = min_index(A, i, upper);
      //@assert le_seg(A[m], A, i, upper);
      swap(A, i, m);
    }
  return;
}
```

At this point, let us verify that the loop invariants are initially satisfied.

- $0 \leq i$ and $i \leq n$ since $i = 0$ and $0 \leq n$ (by precondition (`@requires`)).
- $A[0..i)$ is sorted, since for $i = 0$ the segment $A[0..0)$ is empty (has no elements) since the right bound is exclusive.
- $A[0..i) \leq A[i..n)$ is true since for $i = 0$ the segment $A[0..0)$ has no elements. The other segment, $A[0..n)$, is the whole array.

We should also verify the assertion we added in the loop body. It expresses that $A[m]$ is less or equal to any element in the segment $A[i..n)$, abbreviated mathematically as $A[m] \leq A[i..n)$. This should be implies by the postcondition of the `min_index` function.

How can we prove the postcondition (`@ensures`) of the sorting function? By the loop invariant $0 \leq i \leq n$ and the negation of the loop condition $i \geq n$ we know $i = n$. The second loop invariant then states that $A[0..n)$ is sorted, which is the postcondition.

6 Auxiliary Functions

Besides the specification functions in contracts, we also used two auxiliary functions: `swap` and `min_index`.

Here is the implementation of `swap`.

```
void swap(int[] A, int i, int j)
//@requires 0 <= i && i < \length(A);
//@requires 0 <= j && j < \length(A);
{
  int tmp = A[i];
  A[i] = A[j];
  A[j] = tmp;
  return;
}
```

For `min_index`, we recommend you follow the method used for selection sort: follow the algorithm for a couple of steps on a generic example, write down the invariants in general terms, and then synthesize the simple code and invariants from the result. What we have is below, for completeness.

```
int min_index(int[] A, int lower, int upper)
//@requires 0 <= lower && lower < upper && upper <= \length(A);
//@ensures lower <= \result && \result < upper;
//@ensures le_seg(A[\result], A, lower, upper);
{
  int m = lower;
  int min = A[lower];
  for (int i = lower+1; i < upper; i++)
    //@loop_invariant lower < i && i <= upper;
    //@loop_invariant le_seg(min, A, lower, i);
    //@loop_invariant A[m] == min;
    if (A[i] < min) {
      m = i;
      min = A[i];
    }
  return m;
}
```

7 Asymptotic Complexity Analysis

Previously, we have had to expend some effort to prove that functions actually terminate (like in the function for greatest common divisor). Here we do more: we do counting in order to give a big-O classification of the number of operations. If we have an explicit bound on the number of operations that, of course, implies termination.

The outer loop iterates n times, from $i = 0$ to $i = n - 1$. Actually, we could stop one iteration earlier, but that does not effect the asymptotic complexity, since it only involves a constant number of additional operations.

For each iteration of the outer loop (identified by the value for i), we do a linear search through the array segment to the right of i and then a simple swap. The linear search will take $n - i$ iterations, and cannot be easily improved since the array segment $A[i..n)$ is not (yet) sorted. So the total number of iterations (counting the number of inner iterations for each outer one)

$$n + (n-1) + (n-2) + \cdots + 1 = \frac{n(n+1)}{2}$$

During each of these iterations, we only perform a constant amount of operations (some comparisons, assignments, and increments), so, asymptotically, the running time can be estimated as

$$O(\frac{n(n+1)}{2}) = O(\frac{n^2}{2} + \frac{n}{2}) = O(n^2)$$

The last equation follows since for a polynomial, as we remarked earlier, only the degree matters.

We summarize this by saying that the worst-case running time of selection sort is quadratic. In this algorithm there isn't a significant difference between average case and worst case analysis: the number of iterations is exactly the same, and we only save one or two assignments per iteration in the loop body of the `min_index` function if the array is already sorted.

8 Empirical Validation

If the running time were really $O(n^2)$ and not asymptotically faster, we predict the following: for large inputs, its running time should be essentially cn^2 for some constant c. If we *double* the size of the input to $2n$, then the running time should roughly become $c(2n)^2 = 4(cn^2)$ which means the function should take approximately 4 times as many seconds as before.

We try this with the function `sort_time(n, r)` which generates a random array of size n and then sorts it r times. You can find the C0 code at sort-time.c0. We run this code several times, with different parameters.

```
% cc0 selectsort.c0 sort-time.c0
% time ./a.out -n 1000 -r 100
Timing array of size 1000, 100 times
0
0.700u 0.001s 0:00.70 100.0% 0+0k 0+0io 0pf+0w
% time ./a.out -n 2000 -r 100
Timing array of size 2000, 100 times
0
2.700u 0.001s 0:02.70 100.0% 0+0k 0+0io 0pf+0w
% time ./a.out -n 4000 -r 100
Timing array of size 4000, 100 times
0
10.790u 0.002s 0:10.79 100.0% 0+0k 0+0io 0pf+0w
% time ./a.out -n 8000 -r 100
Timing array of size 8000, 100 times
0
42.796u 0.009s 0:42.80 99.9% 0+0k 0+0io 0pf+0w
%
```

Calculating the ratios of successive running times, we obtain

n	Time	Ratio
1000	0.700	
2000	2.700	3.85
4000	10.790	4.00
8000	42.796	3.97

We see that especially for the larger numbers, the ratio is almost exactly 4 when doubling the size of the input. Our conjecture of quadratic asymptotic running time has been experimentally confirmed.