ECS 122A Design and Analysis of Algorithms **Fall 2025**

Lecture 1: Correctness and Runtime Analysis

Lecturer: Jasper Lee *Scribe: Jasper Lee*

In this writeup, we consider the basic algorithm of selection sort, implemented by iteration and by recursion. We will see how to analyze the correctness and runtime of these implementations.

As a bonus, we will also analyze binary search (the first part of Lecture 2).

1 Selection Sort by Iteration

Here is a straightforward iterative implementation of selection sort.

Algorithm 1 Selection Sort by Iteration

1. Input: Array A of length n, 1-indexed
2. for $i = 1$ to n:
   (a) Find index $j \in [i \,..\, n]$ that maximizes $A[j]$
   (b) Swap $A[i]$ and $A[j]$

1.1 Runtime

We will start with runtime analysis since that's much easier.

In the i^{th} loop iteration, the amount of work is $O(n - i)$. Summing over i from 1 to n gives $O(n(n-1)/2) = O(n^2)$.

We could have even used a looser upper bound, that each loop iteration does $O(n)$ work instead of the tighter bound of $O(n - i)$.

1.2 Correctness

We will use induction to prove the correctness of Algorithm 1. Induction is an extremely useful and important tool for analyzing algorithms and computation—it essentially keeps track of what is changing throughout the computation, allowing us to reason about the process.

To give an inductive proof, we need to *carefully* state the induction hypothesis.

Induction hypothesis At the end of the i^{th} loop iteration, $A[1 \,..\, i]$ contains the smallest i elements (of the original input) in sorted order **and** A is a permutation of the original input.

For loops/iterations, such induction hypothesis is sometimes also called a "loop invariant"—essentially, what remains true at each loop iteration (which is exactly what our induction hypothesis is also capturing).

At this point, you might be wondering, it seems that the first part of the induction hypothesis is the important part, why do we even need the second part (that A is a permutation of the original input)? As with many induction proofs, sometimes, we need to keep track of more information than what we are trying to prove, in order for the induction step to be provable. In the following proof, we will make sure to use *both* parts of the induction hypothesis assumed for the i^{th} iteration, and prove also *both* parts for the $i + 1^{\text{st}}$ iteration.

Let's now give the induction proof.

Base case ($i = 0$, or right before the iteration begins): The induction hypothesis is trivially true.

(It is frequently the case that base cases of inductions are trivial and uninteresting. On the midterm and exam, if you don't write out boring base cases, you will not be penalized. The important thing is to spell out the interesting/core parts of the argument.)

Inductive step: We will assume that the induction hypothesis is true for some $i \geq 0$, and now we need to show that the same holds for $i + 1$.

First, we will show that at the end of the $i + 1^{\text{st}}$ iteration, $A[1..(i + 1)]$ contains the smallest $i + 1$ elements in sorted order. To show this, let A_i be the version of the array at the end of the i^{th} iteration, or equivalently, the beginning of the $i + 1^{\text{st}}$ iteration. By the induction hypothesis, A_i is a permutation of the original input, while $A_i[1 \,..\, i]$ contains the smallest i elements. Thus, the index j found in the $i + 1^{\text{st}}$ iteration must be for the $i + 1^{\text{st}}$ smallest element. The swap step ensures that $A[i + 1]$ does indeed contain the $i + 1^{\text{st}}$ smallest element at the end of the iteration. Furthermore, given that $A[1 \,..\, i]$ is unchanged in the loop iteration, we can conclude that $A[1 \,..\, (i + 1)]$ does indeed contain the smallest $i + 1$ elements in sorted order.

Next, we will show that A remains a permutation of the original input at the end of the $i + 1^{\text{st}}$ iteration. But this is trivial, since the only edit on the array is a swap.

The above two paragraphs complete the induction step.

Wrap up: A lot of the times, after we complete an induction proof, we need to write a little bit more, to use the conclusion of the induction to show what we finally want. In this instance, it's pretty straightforward. The case $i = n$ from the induction hypothesis implies A is sorted at the end of the n^{th} iteration.

1.3 The Purpose and Style of *Pseudo*code

Before we move on to the recursive implementation of selection sort, let's briefly discuss the purpose of pseudocode and what it should look like.

Pseudocode is a means to communicate an algorithm to the reader—the emphasis is on *communication*.

In order to communicate effectively, a lot of the times it's more effective to use English to abstract out some algorithmic details, so that the high level structure of the algorithm becomes more apparent from the text. For example, Algorithm 1 uses a single sentence describing the bigger step of finding the min in a sub-array, instead of writing out the minute details of implementing it. This makes the pseudocode much more readable and communicative.

It requires judgement and balance to decide the degree to which to use English over code. Generally, we want to consider how complex the overall problem and algorithm are. The more complex they are, the more leeway we have to abstract out lower level details (for example, Algorithm 1 is a sorting algorithm, and we chose to abstract out min-finding). On the other hand, if the problem/algorithm is to find the min in an array, then of course we should not just write a single English sentence saying "we will find the min".

The final thing to remember is that, when writing pseudocode, don't just invent a dialect of C or Python and write out a bunch of code. Think carefully what needs to be communicated.

2 Selection Sort by Recursion

Let's now consider a different implementation of selection sort, by recursion.

Algorithm 2 Selection Sort by Recursion

function SORT(A)

1. if $size(A)$ is 1, then return A, else
2. Find $\arg\min_j A[j]$ ($\leftarrow$ This is mathematical notation to precisely state that we want to find the index j that minimizes $A[j]$)
3. $B \leftarrow A$; $\quad B[j] \leftarrow A[1]$;
4. return $A[j]$ :: SORT($B[2\,..\,size(A)]$)

2.1 Runtime

A common way to analyze the runtime of a recursive algorithm is to set up a *recurrence relation* (or recurrence *inequality*) and bound the growth of the algorithm runtime.

Let $T(n)$ be the (worst-case) runtime of SORT in Algorithm 2, when given an array of size n. A call to SORT performs $O(n)$ work, and makes one recursive call to an array with one fewer element. Thus, $T(n)$ can be upper bounded by

$$T(n) \leq c_1 \cdot n + T(n-1)$$

for some constant $c_1 \geq 1$, and without loss of generality we will assume that $T(1) = 1$ by saying that the $n = 1$ case takes "unit" time.

We want to show that SORT runs in $O(n^2)$ time, so it suffices to show that $T(n) \leq c_2 n^2$ for some sufficiently large constant c_2. We will, you guessed it, again show this by induction.

Base case: We know that $T(1) = 1$, so as long as $c_2 \geq 1$, the base case will hold.

Induction step: Assuming $T(n-1) \leq c_2(n-1)^2$ for some $n > 1$, we have

$$\begin{aligned} T(n) &\leq c_1 n + T(n-1) \quad \text{by the recurrence inequality} \\ &\leq c_1 n + c_2(n-1)^2 \quad \text{by the induction hypothesis} \\ &= c_2 n^2 - 2c_2 n + c_1 n + c_2 \end{aligned}$$

If we choose $c_2 \geq c_1$ (and recalling that $n > 1$), then $c_1 n - 2c_2 n + c_2 \leq c_2 - c_2 n \leq 0$, implying that

$$T(n) \leq c_2 n^2$$

Thus, as long as $c_2 \geq \max(c_1, 1)$, then by induction we have shown that $T(n) \leq c_2 n^2$ for all $n \geq 1$, concluding that SORT runs in $O(n^2)$ time.

2.2 Correctness

As with the iterative version of selection sort, we will use induction to prove correctness. However, since the implementation is different, we will need to write a different proof, **with a different induction hypothesis**.

Induction hypothesis For every input array of size n, SORT returns the sorted version

We now proceed with the induction proof.

Base case: Again, trivial for $n = 1$.

Induction step: Assume the induction hypothesis for some $n \geq 1$, now consider giving an array of size $n+1$ as input to SORT. We know that $A[j]$ is the smallest element in A, and $B[2\,..\,(n+1)]$ is some permutation of the n largest elements of A. Applying the induction hypothesis on $B[2\,..\,(n+1)]$, we conclude that $A[j] :: \text{SORT}(B[2\,..\,size(A)])$ is the sorted permutation of A.

Wrap up: There really isn't anything to wrap up. The induction hypothesis is exactly the statement we are trying to prove.

3 What have we learnt?

So what have we learnt from studying two different implementations of selection sort?

- The two implementation uses two different induction variables. How do we choose generally?
  - The key point is to choose an induction variable that *changes* during the computation. Induction as a proof technique keeps track of what is changing throughout the computation, and so the induction variable should also be changing.
  - For loops and iterative algorithms, generally the induction variable is the loop index (even for while loops where there is no explicit loop index in the code or pseudocode).
  - For recursive algorithms, typically the induction variable is input size, although there certainly are exceptions (that we will not go into, at least not in these notes).
- At a higher level, make sure that the proof structure follows and matches code structure. As a corollary, if we write spaghetti code with complicated structure, any analysis will necessarily be complicated and messy as well.

- Precision: notice that in the above examples, we never used any vague words like "correct" in the induction hypothesis or anywhere else in the proofs. We spelled out exactly what "correct" means in each context, to be precise about our claims.

4 Bonus: Binary Search by Iteration

Let's analyze the basic binary search algorithm as a bonus. We will see this in Lecture 2.

Setting: Given sorted array A, find the index of a given element x if x exists in the array.

Algorithm 3 Binary search by iteration

Assume $A[0\,..\,(n-1)]$ is a sorted array

1. $L \leftarrow 0$, $R \leftarrow n-1$
2. while $L \leq R$
   (a) $m \leftarrow L + \lfloor (R-L)/2 \rfloor$
   (b) If
      i. $A[m] < x$ then $L \leftarrow m+1$
      ii. $A[m] > x$ then $R \leftarrow m-1$
      iii. $A[m] = x$ then return m
3. return "FAIL"

4.1 Runtime

$R - L$ is (roughly) halved each time, and so the algorithm runs in $O(\log n)$ time. Formally we can do an induction, but the formality required for these analyses depends on how convincing the informal argument is.

4.2 Correctness

This is an iterative implementation of binary search, so what should we induct on as the induction variable? Even though there is no explicit loop counter, we will still use the loop count.

Loop invariant At the end of loop iteration i, if $A[t] = x$ for some t, then $t \in [L, R]$.

The induction step is pretty straightforward, following the case analysis within binary search. Try it out yourself!