CPSC 365 Proofs Class 2

James Williams

February 16, 2015

1 Background

1.1 Types

For alogrithms specifically, there are three important types of proof or reasoning: termination, correctness, and efficiency / complexity. These are frequently merged into one analysis of the algorithm or method, but require different approaches and degrees of formality, so we will consider each of them separately for clarity.

Termination

Sometimes it is not trivial that an algorithm actually terminates for all possible inputs, and this needs to be formally proved. For iterative algorithms, this is usually trivial. For divide and conquer algorithms, recursive algorithms, random algorithms, dynamic programming, and other more complicated algorithms, this is less trivial.

For direct or deterministic algorithms, if there is a finite run time complexity, or a finite number of operations required for the algorithm to end, then termination is implied. In this case, proof of termination becomes trivial after computing the run time complexity.

For approximate or randomized algorithms, computing the run time complexity might not be trivial, as we would need to prove convergence or a probabilistic bound. However, in this case, proof of termination would be implied by the proof of convergence or the probabilistic bound, and can be thought of as a corollary.

Correctness

The most important proof is that the algorithm does what it says, and always returns the correct output for any permissible input! Formal proof of correctness can be surprisingly long, when done correctly, and sometimes only a sketch of the proof is presented (especially in papers or textbooks, where the details of the proof can be included in the appendices).

A distinction is made between partial correctness, which requires that *if* an output is returned, it is correct, and total correctness, which also requires that the algorithm terminates, or that an output is *always* returned. A formal proof of total correctness is analogous to a formal mathematical proof as described in CSPC 365 Proofs Class 1, and is where the same types, styles, methods, and tools that we discussed will be most useful.

Efficiency / Complexity

There are two main measures of algorithm efficiency, space complexity and run time complexity. Space complexity describes the amount of storage required by the algorithm, either in memory (RAM) or mass storage (HDD), and run time complexity describes the number of operations required for the algorithm to end. These are expressed as a function of the input size n, exactly or asymptotically, such as constant complexity $\mathrm{O}(1)$, logarithmic complexity $\mathrm{O}(\log n)$, linear complexity $\mathrm{O}(n)$, or polynomial complexity $\mathrm{O}(n^k)$ where $k > 1$.

Computing space complexity is typically easier than computing run time complexity, as it only requires counting the memory requirements of all variables or data structures used. Note that this measure of complexity only computes the theoretical memory requirements, and implementation or language can introduce additional memory overhead.

Computing run time complexity requires counting the cost of the operations and routines involved, including any iterative or recursive procedures such as for loops, while loops, or recursive routines (which typically have a fixed depth based on the input size n).

Simple operations such as arithmetics, data handling, and logical control take constant time $\mathrm{O}(1)$. For loops take time proportional to the product of the loop length and the internal time complexity, but while loops also require a bound on the number of iterations necessary for the loop conditional to be satisfied. Similarly, recursive routines take time proportional to the depth of recursion and the internal time complexity, and sometimes require a bound on the depth of recursion as this can be non trivial.

arithmetics	addition, subtraction
	multiplication, division
	remainder
	round, floor, ceiling
data handling	allocation, deallocation
	copy
	read, write
logical control	conditional / unconditional branching
	subroutine call, recursive call
	return

Table 1: Simple operations that take constant time $\mathrm{O}(1)$

sorting	quicksort, heapsort, mergesort
	bubble sort, insertion sort, select sort, bucket sort
	bucket sort, radix sort
searching	DFS, BFS
	shortest path algorithms
data structures	arrays, dynamic arrays, associative arrays
	lists
	binary search trees, heaps, priority queues
	stacks, queues
	matrices, sparse matrices

Table 2: Complex operations that do not take constant time $\mathrm{O}(?)$

Having an awareness of the run time complexity of these simple and complex operations, especially iterative and recursive structures, makes it much easier and faster to compute the run time complexity of algorithms in general.

1.2 Tools

Data Structures

Algorithms depend on data structures, which make it easier to define algorithms and prove correctness concisely and accurately. Arrays make it easy to index a set of elements efficiently, linked lists make it easy to insert and delete elements dynamically, stacks and queues make it easy to access elements in a particular ordering, and binary search trees, heaps, and priority queues make it easy to implement specific sorting, selection, and graph algorithms.

Note also that while multidimensional data structures such as dense matrices or sparse matrices are implemented using arrays, they are still a good distinction to make, as it can be simpler to talk about arrays as one dimensional and matrices as multidimensional.

Also, it is important to understand the run time complexity of data structure operations, such as indexing, searching, inserting, and deleting elements, noting the distinction between *average* and *worst* run time complexity. The Big O Cheat Sheet gives a very good overview of complexities for searching algorithms, sorting algorithms, data structure operations, heap operations, and graph operations.

Algorithms

Several algorithm procedures that come up frequently have standard proofs that can be reused. Logical control, conditional statements, for loops, while loops, and recursion have reasonably standard axioms and approaches, and proving algorithms that can be broken down into these components can be fairly straightforward.

Notably, for loops, while loops, and recursion are typically analysed using inductive reasoning, which is described as using invariants in some computer science terminology or simply proof by induction in mathematical terminology.

There are many good references that go into more detail about the analysis of these algorithm procedures, several of which are provided in the bibliography.

Methods

All of the types, styles, methods, and tools that we discussed in CSPC 365 Proofs Class 1 will help with describing algorithms and proving correctness

- types / methods
  - trivial, direct, indirect types
  - contradiction, contrapositive
  - induction / invariants
- tools
  - notation / terminology
  - diagrams
  - components

and the *plan of action* that we outlined applies equivalently to deriving, describing, and proving algorithms as well.

2 Method

Elaborating on the plan of action from CSPC 365 Proofs Class 1, there is a specific order to deriving, describing, and proving algorithms that can help make it much easier to write and understand. There should be a preliminary section before the sequence of operations that summaries the problem and introduces convenient notation, along with any important data structures, the sequence of operations, proof of termination / complexity, and a formal proof of correctness (where the level of detail depends on what the problem requires, obviously).

Introduction / Background

- rewrite or reduce the problem to a concise but formal statement using notation relevant to your method
  - restate the problem
  - state a more general problem that is actually more straightforward
  - state a simpler problem without loss of generality
- state the input / precondition provided and the output / postcondition required
- introduce convenient notation, symbols, and terminology (along with any important data structures)

Algorithm

- state the initialization (preconditions and initial assignments)
- state the sequence of operations
  - use pseudocode that is easy to read / understand
  - use consistent notation, symbols, and terminology
- state the termination condition / return statement as needed

Termination / Complexity

- prove that the algorithm terminates
- compute space / run time complexity

Correctness

- prove that the algorithm always returns the correct output for any permissible input as required
  - or that the postcondition is respected on all inputs satisfying the precondition, equivalently
- state that the algorithm terminates on all inputs as well

Additionally, make sure that the sequence of operations is labelled / commented so that it can be easily referenced without copying chunks of operations repeatedly. You can label specific chunks of operations by hand, especially when you are writing the algorithm by hand, or you can use an algorithms package that automatically numbers each line and emphasises algorithm procedures and logical control.

3 Summary

3.1 Comments

- writing algorithms
  - outlines can help make the sequence of operations pseudocode easier to read / understand
  - avoid using *clearly*, *trivially*, and *obviously* to cover up that you don't know what you are doing
  - avoid using an overabundance of subscripts and superscripts
- proving correctness
  - always conclude your proof with a clear statement that you solved and proved what was defined
  - note that to prove that an algorithm isn't correct only an example is required
- general comments
  - try to break your algorithm and your proof (works for all inputs and uses correct arguments)
  - implement your algorithm and run some simulations
  - come up with illustrative examples that show worst case / best case efficiency

3.2 Warnings

- writing algorithms
  - check that you have not simplified the problem to a special case or a specific example
  - without loss of generality can be dangerous when you actually lose some generality
  - don't segment your algorithm or your proof of correctness into too many components
- efficiency
  - local and global maxima and minima are **not the same** and the distinction **is** necessary
  - not being able to improve the result by a small change **does not** imply the result is optimal
- general warnings
  - don't write the algorithm and the proof together (although the algorithm should be commented)
  - examples are not sufficient, but can help make your proof easier to read / understand
  - watch out for a logical fallacy ($A \Rightarrow B$ and $A \Leftarrow B$ and $A \Leftrightarrow B$ are not equivalent statements)

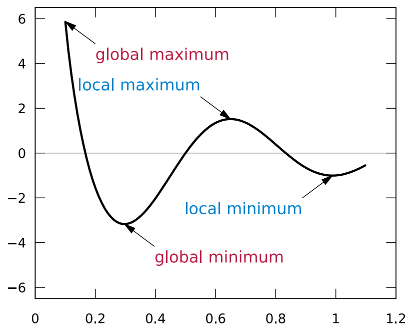

Figure 1: Examples of local and global maxima and minima for $\cos(3\pi x)/x$ where $0.1 \leq x \leq 1.1$

3.3 References / Bibliography

- types of algorithms

wikipedia list of algorithms	bit.ly/1lyR1Ea
types of algorithms	bit.ly/1Bj50ET
algorithms every programmer must know	bit.ly/1ANCn1b

- proving correctness

proving algorithm correctness	bit.ly/17MV22g
correctness proofs	bit.ly/1A2Epde
verification of algorithms correctness	bit.ly/1yRHQ1t

- complexity

wikipedia time complexity	bit.ly/xYyzWn
introduction to time complexity	bit.ly/1AzZYE9
determining time complexity	bit.ly/1CEcXlU
Big O misconceptions	bit.ly/1L8uarj
Big O cheat sheet *“Know Thy Complexities!”*	bit.ly/UiNYBl

Bubble Sort (Bad)

Description of Method

Bubble Sort works by moving the largest remaining element in the unsorted list to the right end. It does this by scanning over the list from left to right, swapping pairs of elements which appear in decreasing order, moving the largest element to the right end.

Method

1. Look at the $n - 1$ pairs, moving from left to right, and swap the pairs which appear in decreasing order, such that the largest element is at the right end.
2. Repeat for the remaining $n - 1$ elements, leaving the largest element unchanged.

Proof of Method

Clearly the worst case is when the list is in reverse order, so if this can be sorted, then all lists can be sorted.

Each step swaps the elements that appear in the wrong order, moving the largest element from left to right until the largest element is at the right end. Eventually there is only one element remaining at the left end, which is the smallest element, so the list is sorted.

Bubble Sort (Not as Bad)

Introduction / Background

Let A be an array of n real numbers to be sorted. We denote the i^{th} element of the array $A[i]$, where $i \in \{1, \ldots, n\}$.

Without loss of generality, we assume that the elements are unique, $A[i] \neq A[j]$ for all $i \neq j$. We can then say that if $A[i] < A[j]$ for every $i < j$, the array is sorted.

Algorithm

Let A be the input array of n real numbers to be sorted.

```
1. for  $i = 1$ to $n$  do
2.    $k \leftarrow n - i + 1$
3.    for  $j = 1$ to $k - 1$  do
4.       if  $A[j] > A[j+1]$  then
5.          swap A[$j$] and $A[j+1]$
6.       end if
7.    end for
8. end for
```

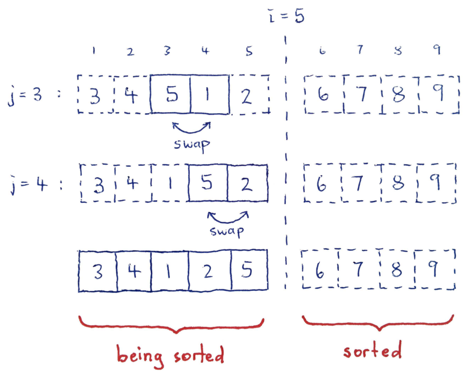

Figure 2: Example of an iteration that swaps pairs of elements to move the largest element to the right end.

Termination / Complexity

The outer loop requires n iterations to end. The assignment of k is O(1). The inner loop requires $k - 1 < n$ iterations to end, and the conditional exchange is O(1). From this, the algorithm performs $O(n^2)$ operations and terminates successfully.

Correctness

Observation 1

First note that after each swap the array A contains the same elements as the input array, though the order may have changed. There might be arbitrary changes to the entries being swapped during the swap, but this is always resolved when the swap is completed.

Lemma 1

The inner loop moves the largest element in $A[1] \dots A[k]$ to $A[k]$ such that $A[i] < A[k]$ for all i such that $i < k$, and the elements in $A[k+1] \dots A[n]$ are not affected.

Proof of Lemma 1

The elements in $A[k+1] \dots A[n]$ are not accessed, and thus are not affected. Hence, only the order of $A[1] \dots A[k]$ is changed. Note that at iteration $j+1$, elements $A[1] \dots A[j]$ have been accessed and possibly affected, and elements $A[j+1] \dots A[k]$ have yet to be affected.

By way of contradiction, assume that the largest element in $A[1] \dots A[k]$ is at some $i < k$ at the inner loop's end such that $A[i] > A[j]$ for all j such that $j > i$. At the inner loop's end, $j = k-1$, and iterations $j = i+1 \dots k-1$ affect only elements $A[i+1] \dots A[n]$. At iteration $j = i$, $A[j] > A[j+1]$ as $A[i] > A[i+1]$, and the conditional swaps $A[j]$ and $A[j+1]$, which is a contradiction, as the largest element in $A[1] \dots A[k]$ is now at $i+1$.

Hence, the largest element in $A[1] \dots A[k]$ is $A[k]$ such that $A[i] < A[k]$ for all i such that $i < k$, and the elements in $A[k+1] \dots A[n]$ are not affected. Note that the above reasoning can be applied inductively to prove directly that the largest element is moved to $A[k]$ through iterations $j = i+1 \dots k-1$. $\square$

Theorem 1

The i^{th} largest element is in $A[k]$ for all $i \in \{1, \dots, n\}$, where $k = n - i + 1$.

Proof of Theorem 1

We will use induction to prove Theorem 1.

By Lemma 1, the 1st largest element is moved to the $A[n]$ at the end of the 1st iteration of the outer loop, and remains there until the outer loop's end. By Observation 1, only the order of the remaining elements in the array can be changed, so this remains the 1st largest element of A.

Suppose that at the start of the i^{th} iteration of the outer loop, the j^{th} largest element is in $A[n-j+1]$ for all j such that $j < i$. By Lemma 1, the i^{th} largest element is moved to $A[n-i+1]$ at the end of the i^{th} iteration, and the elements in $A[n-j+1]$ for all j such that $j < i$ are not affected. Hence, at the start of iteration $i+1$ of the outer loop, the j^{th} largest element is in $A[n-j+1]$ for all j such that $j < i+1$.

By induction, it follows that the i^{th} largest element is in $A[k]$ for all $i \in \{1, \dots, n\}$, where $k = n - i + 1$. $\square$

From Theorem 1, the i^{th} largest element of the output array is in position $n - i + 1$, and for any $i < j$, we have $A[i] < A[j]$, so the output array A is sorted. From Observation 1, the output array A contains the same elements as the input array, and so the sorted input array is returned.