Determining and Proving Algorithm Correctness

AUBURN UNIVERSITY

Hugh Kwon

Adapted from Dr. Debswapna Bhattacharya's slides

Part I: Algorithm Design

1. Specify the problem well
2. Come up with at least one solution strategy
3. Develop corresponding algorithms
    1. Find and understand existing algorithms (by simulating their operation) that use these strategies to solve the problem
    2. If no such algorithms can be found, find and understand existing algorithms that solve a similar problem, then modify them appropriately to solve your problem
    3. Design new algorithms

Part II: Algorithm Analysis

4. Ensure/prove correctness of all algorithms
5. Analyze and compare their performance/efficiency
    4. Theoretically: Using a variety of mathematical tools
    5. Empirically: Code, run and collect performance data
6. Choose the most efficient algorithm to implement

- First, read the algorithm.
- Second, understand its mechanics.
- Third, make an informed hypothesis – that the algorithm is correct/incorrect
- Fourth, prove it!

https://www.youtube.com/watch?v=dB9h11Us87c

If your hypothesis is correct, you will be successful in proving it, and the proof guarantees it.

Otherwise, the process of proving will show you why your hypothesis is wrong.

First, check if the algorithm will always terminate:

- Iterative algorithms:
  - Make sure that each loop will eventually reach the exit condition
    - Easy for "for" loops which have a bounded loop variable
    - For other kinds of loops: test (simulate) each loop on inputs that are boundary cases i.e., inputs for which the loop:
      - » May not execute at all
      - » Executes only once
      - » Executes multiple times then exits

First, check if the algorithm will always terminate:

- Recursive algorithms
  - Make sure that there is at least one base case,
  - that the algorithm returns correct answers for the base cases
  - that each recursive call makes progress towards the base case(s)

Next,

1. Map the legal input sets
2. Determine boundary cases of the input ranges
3. Systematically test (simulate) the algorithm on representative inputs that cover all valid classes of inputs, especially the boundary cases

Problem: Find the k-th largest number in an array of n distinct numbers, where $1 \leq k \leq n$

K-th Largest Number Problem

Algorithm A1(A: Array [1..n] of distinct numbers, k:integer, 1<=k<=n) print A[k]

Correct or incorrect?

Algorithm A2(A: Array [1..n] of distinct numbers, k:integer, 1<=k<=n)

Sort A in ascending order print A[k]

Correct or incorrect?

Algorithm A3(A: Array [1..n] of distinct numbers, k:integer, 1<=k<=n)

Sort A in descending order print A[k]

Correct or incorrect?

If we think that an algorithm is correct/incorrect, how can we convincingly show that?

Correctness proofs demonstrate and guarantee that an algorithm is correct (or incorrect)

Four proof techniques:

1. Proof by Counterexample
2. Proof by Contradiction
3. Proof by Loop Invariants
4. Proof by Induction

- Come up with a problem instance for which the algorithm produces incorrect output
  1. Provide the problem instance
  2. Show the correct answer(s)
  3. Provide the answers generated by the algorithm (or state it does not halt)
  4. Logically justify how/why the algorithm generates incorrect answers or does not halt

- Can be used to show that an algorithm is incorrect

- Cannot be used to show an algorithm is correct! (why?)

Proof by Counterexample

Algorithm A1(A: Array [1..n] of distinct numbers, k:integer, 1<=k<=n) print A[k]
Incorrect!

1. Problem Instance n=4, k=1, A is [1000, 2000, 4000, 3000]
2. The correct answer is 4000
3. The algorithm outputs 1000, which is incorrect
4. Because print A[k] will print A[1]=1000

- Start by **assuming** the opposite of what you want to prove
- Develop the **logical** and **factual** implications of this assumption
- Compare with what the algorithm specification says about its operation
- Show that this leads to a contradiction
- Can be used to show that an algorithm is correct or incorrect

Algorithm A2(A: Array [1..n] of distinct numbers, k:integer, 1<=k<=n)

Sort A in ascending order print A[k]

Incorrect!

1. Suppose the algorithm is correct.
2. That means that for all valid input arrays of n distinct numbers and all integers k, 1<=k<=n, it must print the k- th largest number.
3. Algorithm A2 sorts the array A in the increasing order.
4. So after sorting, A[n-k+1] will contain the k-th largest number.
5. Therefore by our assumption A2 must print out A[n- k+1].

Proof by Contradiction (A2)

4. A2 actually prints A[k].
5. A[n-k+1] = A[k] only when n-k+1 = k or 2k = n+1. I.e. only when n is odd and k = (n+1)/2. For all other values of k, A[n-k+1] ≠ A[k].
6. This means that for all valid n-sized arrays and all integers k, 1<=k<=n, it will not print the k-th largest number.
7. Step 6 of the proof contradicts step 2!
8. So our assumption in step 1 that the algorithm is correct must be wrong, i.e. the algorithm is incorrect.

Algorithm A3(A: Array [1..n] of distinct numbers, k:integer, 1<=k<=n)

Sort A in descending order print A[k]

Correct!

1. Suppose *the* algorithm is incorrect.
2. That means that for at least one valid input array of n distinct numbers and integer k, 1<=k<=n, it will produce an incorrect answer.
3. In other words, for at least one valid input array and integer k where 1<=k<=n, A3 will not print the k-th largest number.

Proof by Contradiction (A3)

4. Algorithm A3 sorts the array A in the decreasing order of incomes.
5. So after sorting, A[1..k] must contain the k largest numbers in A for any k where 1<=k<=n.
6. So for any n and 1<=k<=n, A[k] must contain the k-th largest number after A3 finishes sorting. Then it prints A[k].
7. So for all valid inputs A and k, A3 will print the k-th largest number.
8. Step 7 of the proof contradicts step 3!
9. So our assumption in step 1 must be wrong, i.e. the algorithm has to be correct.

```
Algorithm A4 (A: Array [1..n] of distinct numbers; k: integer 1≤k≤n)
1       for i=2 to k
2           for j=n downto i
3               if A[j]>A[j-1] then
4                   swap A[j] and A[j-1]
5       print A[k]
```

- Is A4 a correct algorithm?

- If you say correct, can you explain in plain English why it will produce the correct answer given any problem instance?

- If you say incorrect, does it produce an incorrect answer for every problem instance? If yes, why? If no, can you characterize the inputs for which it will produce a correct answer and inputs for which it will produce a wrong answer?

Thinking Assignment

- Prove by contradiction that algorithm A4 is incorrect (or correct if that is the conclusion you reached when doing the previous thinking assignment).

```
Boggle(A: n X n array of characters)
1 For i=1…n
2    For j=1…n
3          temp ← make-string(A[i,j])
4          if dict_lookup(temp)=T then print temp
5          For k=(i+1)…n
6                temp ← make-string(A[i,j]…A[k,j])
7                if dict_lookup(temp)=T then print temp
8                temp ← string-reverse(temp)
9                if dict_lookup(temp)=T then print temp
10         For k=(j+1)…n
11               temp ← make-string(A[i,j]…A[i,k])
12               if dict_lookup(temp)=T then print temp
13               temp ← string-reverse(temp)
14               if dict_lookup(temp)=T then print temp
```

What does it mean for this algorithm to be correct?

It must halt.

It must print out all English words that appear in a row or column in either direction.

It must not print out any non-English word.

Boggle Correctness Proof

A. Assume the algorithm is incorrect.

B. All its loops have fixed bounds so algorithm will end when the i-loop ends.

C. We assume dict_lookup works correctly and the algorithm will print only those words that the lookup confirms, so no non-English words will be printed.

D. So the implication of our assumption is that there must be at least one English word that the algorithm fails to print.

E. This implies that there must be at least one string that the algorithm does not check against the dictionary among the $(2n^3 - n^2)$ possible strings row-wise and columnwise in an nXn character array.

F. There are two cases: (1) the string is in a row (2) it is in a column

G. Consider (1). Let this string be in row a: A[a,b]…A[a,c] where $1 \leq a,b,c \leq n$.

Boggle Correctness Proof

H. The part of the algorithm checking row-wise is:

```
1.For i=1…n
2.   For j=1…n
3.          temp ← make-string(A[i,j])
4.          if dict_lookup(temp)=T then print temp
5.          For k=(j+1)…n
6.                  temp ← make-string(A[i,j]…A[i,k])
7.                  if dict_lookup(temp)=T then print temp
8.                  temp ← string-reverse(temp)
9.                  if dict_lookup(temp)=T then print temp
```

I. Lines 1, 2 and 3 ensure that all 1-character strings A[i,j]…A[i,k] for $1 \leq i,j \leq n$, $k=j$ are checked against the dictionary in line 4.

J. Lines 1, 2, 5 and 6 ensure that all strings A[i,j]…A[i,k] are checked against the dictionary in line 7 for $1 \leq i,j \leq n$, $j<k \leq n$.

K. Therefore lines 1-7 of the algorithm check all possible strings A[i,j]…A[i,k] for $1 \leq i,j \leq n$ and $j \leq k \leq n$.

Boggle Correctness Proof

L. So the string A[a,b]…A[a,c] that is not checked must have c<b.

M. But if c<b then string A[a,b]…A[a,c] is the reverse of string A[a,c]…A[a,b] where 1≤a,c ≤ n and c<b ≤ n, so by statement K line 6 will consider string A[a,c]…A[a,b].

N. But then lines 8 & 9 will consider string A[a,b]…A[a,c].

O. So there can be **no string** string A[a,b]…A[a,c] in any row that is not checked.

P. An argument similar to steps G-O can be used to show that there can be no unchecked (against the dictionary) string A[a,b]…A[c,b] in a column either.

Q. O & P together contradict E.

R. Therefore the assumption we started with must be incorrect. I.e., the algorithm is correct!

- A Loop Invariant (LI) is a property that holds true before any execution of the loop.
- Usually this is a property of the data that the loop manipulates.
- Usually the property is stated in terms of the i^{th} execution of the loop.

- For algorithms that do their work by using a main loop, stating and proving a LI that relates to what the algorithm is intended to do is a way of proving the algorithm to be correct.

- You have to first understand the algorithm in order to come up with a LI that relates to the intended behavior of the algorithm!

```
Insertion-sort(A)
1    for j = 2 to n
2            key=A[j]
3            i  = j - 1
4            while i>0 and A[i]>key
5                    A[i+1] =A[i]
6                    i  = i - 1
7            A[i +1] = key
```

Correct or incorrect?

- *Insertion Sort Loop Invariant* :
  - Before the start of the loop's iteration with value j, $2 \leq j \leq n$, of the for loop of lines 1-7, the subarray A[1..j-1] consists of the elements originally in A[1..j-1] but in sorted order.

1. Decide on the appropriate LI
2. Prove that it holds before the loop starts:
   Initialization
3. Prove that if it holds before the i^{th} execution of the loop, then it will hold before the next $(i+1)^{th}$ execution as well: **Maintenance**
4. These two proofs together prove that the LI must hold after the last execution too, and this should show that the algorithm solves the problem correctly: **Termination**

- Before the start of iteration of the loop with j=2, the subarray A[1…1] consists of the elements originally in A[1...1] but in sorted order.
- Trivially true because A[1…1] is a one-element array which is by definition sorted!

1. Suppose that before the start of the loop's iteration with value j, the subarray A[1...j-1] consists of the elements originally in A[1...j-1] but in sorted order.
2. During the j^{th} iteration of the loop, the while loop (lines 4-6) compares A[j] with A[j-1], A[j-2] etc. until (i) either a number $A[k] \leq A[j]$ is found or (ii) it turns out that all numbers A[j-1], A[j-2]…A[2],A[1] are greater than A[j].
3. Each number found to be greater than A[j] is moved one cell to the right, i.e., if A[j-1]>A[j] then A[j-1] is moved to A[j] and so on.
4. If case (i) holds, then the numbers A[k+1],A[k+2],…A[j-1] are moved to cells A[k+2], A[k+3],…A[j] respectively and A[j] is placed (line 8) in A[k+1].
5. Since in this case A[1]…A[k] are not moved, by assumption (1) these are in the sorted order.
6. By the same assumption, A[k+1],A[k+2],…A[j-1] were in sorted order, so now A[k+2], A[k+3],…A[j] are in sorted order.

7. Since A[k]≤A[j] and A[j] is now in A[k+1], A[k]≤A[k+1].
8. Since A[k] is the first number found by the while loop such that A[k]≤A[j], we know that A[k+1] must have been greater than A[j]. So now A[k+1]<A[k+2].
9. 7 & 8 imply that A[k], A[k+1] and A[k+2] are in sorted order.
10. 5, 6 & 9 imply that A[1]…A[j] are in sorted order at the **end of current iteration**.
11. Suppose case (ii) holds instead, i.e., A[j-1], A[j-2]…A[2],A[1] are all greater than A[j].
12. Then the while loop will move A[j-1], A[j-2]…A[2],A[1] to A[j],A[j-1]…A[2] respectively, and place A[j] in A[1].
13. By assumption 1, A[j-1]…A[1] were in sorted order, so now A[j]…A[2] are in sorted order.
14. Since A[1] was greater than A[j], now A[2]>A[1].
15. 13 & 14 imply that A[1]…A[j] are in sorted order at the **end of current iteration**.
16. So in both cases, at the end of the of j^{th} iteration of the loop, A[1]…A[j] will be in sorted order.
17. I.e., before the start of the next iteration with the loop variable having value j+1, the subarray A[1…j] will be in sorted order and so the LI will be true.

1. Initialization showed that the LI will be true before the loop begins (i.e. before the first iteration of the loop with j=2).
2. Maintenance showed that if the LI was true before an iteration, it would still be true before the next iteration.
3. So LI must be true before the second iteration of the loop with j=3, before the third with j=4 and so on.
4. I.e., LI will be true before every iteration of the loop.
5. The loop ends after the iteration with j=n, i.e., before the iteration with j=n+1. The LI must be true at that time.
6. LI: Before the start of the loop's iteration with value j, $2 \leq j \leq n$, of the for loop of lines 1-7, the subarray A[1..j-1] consists of the elements originally in A[1..j-1] but in sorted order.
7. I.e., before the start of the loop's iteration with value j=n+1, the subarray A[1…n] will consist of the elements originally in A[1...n] but in sorted order.
8. Thus, this algorithm sorts the array correctly!

Ch. 2: Section 2.1: p.18-20 - the LI proof of the correctness of Insertion-Sort

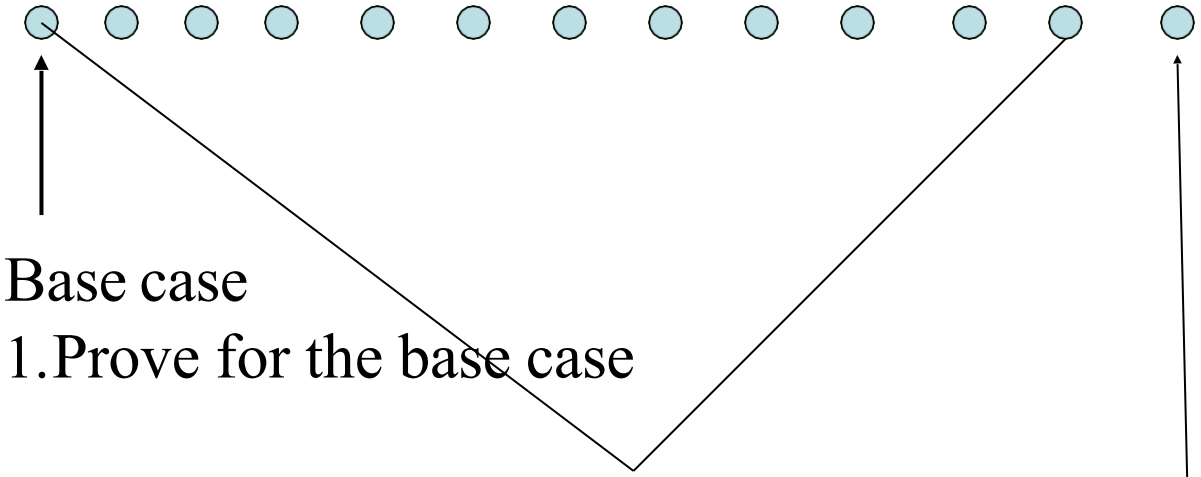
Ch. 2: Section 2.3: p.31-34 - the LI proof of the correctness of the Merge algorithm

Do Problem 2.1-3, p.22

```
Algorithm A5  (A: Array [1..n] of distinct numbers; k: integer 1≤k≤n)
        1       for i=2 to (k+1)
        2               for j=n downto i
        3                       if A[j]>A[j-1] then
        4                               temp=A[j]
        5                               A[j]=A[j-1]
        6                               A[j-1]=temp
        7       print A[k]
```

Develop and state an appropriate LI for this algorithm.

Prove using this LI that this algorithm to solve the k-th largest number problem is correct.



Base case
1.Prove for the base case

2.Inductive hypothesis: Assume true for the first k items

3.Inductive step: Show that it holds for the k+1-th item

Inductive Proofs: An Example

- Fibonacci numbers
  - $F_0 = 1; F_1 = 1; F_n = F_{n-1} + F_{n-2}$
  - 1 1 2 3 5…

Prove F_i $<(5/3)^i$ for $i>=1$

- Two base cases: F_1 and F_2 - why?
- Proving the base cases:
  - $F_1=1<(5/3)^1$ and $F_2=2<(5/3)^2=25/9$
- Inductive Hypothesis:
  - $F_i <(5/3)^i$ for $1<=i<=k$ for some k
- Inductive step
  - Now we need to show that $F_{k+1} <(5/3)^{k+1}$

- We know $F_k < (5/3)^k$ and $F_{k-1} < (5/3)^{k-1}$ from the inductive hypothesis
- We also know that $F_{k+1} = F_k + F_{k-1}$ by definition of the Fibonacci series
- So $F_{k+1} < (5/3)^k + (5/3)^{k-1} = (3/5)^* (5/3)^{k+1} + (9/25)^* (5/3)^{k+1}$

$$= (5/3)^{k+1} * (3/5 + 9/25)$$

$$= (5/3)^{k+1} * (24/25)$$

$$< (5/3)^{k+1} \text{ - done!}$$

- Start by showing that the algorithm produces the correct outputs for one or a few initial inputs (proving the base cases)
- Then make an assumption that the algorithm produces the correct outputs for a finite set of k valid inputs (making the inductive hypothesis)
- Finally (and most importantly) show that the algorithm will produces the correct output for the next (k+1-th if the algorithm reduces the input size by 1 for each recursive call; 2k if the algorithms halves the input size for each recursive call, etc.) input (proving the inductive step)
- Generally used to prove correctness of recursive algorithms

find-max(A:array [i...j] of number) returns number k: number
1 if i=j then return A[i]
2 k= find-max(A:array [i+1...j])
3 if k>A[i] then return k else return A[i]

To prove that find-max correctly finds the max number in an array of n integers:

1. Proving the Base Cases:

   If A is an array of length 1, by definition its element is the max. Step 1 of the algorithm returns the single element if array size is 1, so the algorithm is correct for the base case input.

2. Making the Inductive Hypothesis

Assume when the algorithm is called with an array of size k, k≥1 (1=the base case size), it returns the correct max element.

3. Proving the Inductive Step:

We need to prove that if the input A is an array of size k+1, the algorithm will return the max element in it.

If $k \geq 1$ then $k+1>1$. So the input array contains more than one element, so $j>i$, i.e., the condition checking in Step#1 will fail, and Steps #2 & #3 will be executed.

After Step #2 is executed, the variable k will contain the max element in the subarray $A[i+1 \ldots j]$ of size k (by the Inductive Hypothesis).

Step#3 will choose and return the larger of A[i] or k.

I.e., the algorithm will return the larger of the first element of the array and the max element in the rest of the array when the input is an array of size k+1.

The maximum of k+1 numbers is the same as the maximum of the first number and the maximum of the other k numbers.

So when the input size is k+1, the algorithm will return the correct answer.

Thus, we have shown that the algorithm:

Produces the correct answer for the base case (array of size 1)

AND

Assuming that it produces the correct answer for all arrays of size k, $k \geq 1$, it will produce the correct answer for an input array of size (k+1)

SO

It will produce the correct answer for ALL input arrays of size 1 or more

string-reverse(s:string) returns string

//string-length, last-character, delete-last-character & concatenate are functions. String- length returns the length of its argument, last-character returns the last character of its argument, delete-last-character returns the input string without its last character; concatenate attaches its first argument to the beginning of the second argument and returns the resulting string

```
lc: character
temp: string
1  if string-length(s)≤1 then return s
2  lc=last-character(s)
3  temp=string-reverse(delete-last-character(s))
4  return concatenate(lc,temp)
```

To prove that string-reverse correctly reverses all strings:

1. Proving the Base Cases:

   There are two base cases: s is a string of length 0 or 1.
   In either case, by definition $s^{Reverse} = s$.

   Step 1 of the algorithm returns s if its length is 0 or 1, so the algorithm is correct for base case inputs.

2. Making the Inductive Hypothesis

Assume the algorithm returns the reverse of all strings of length from 0 to k, for some $k \geq 1$ (1 = largest base case input size).

3. Proving the Inductive Step:

We need to prove that if the input s is a string of length k+1 (i.e. with 1^{st}, 2^{nd},..., k^{th} and $(k+1)^{th}$ characters), the algorithm returns its reverse.

If k≥1 then k+1≥2, so the condition checking in Step#1 will fail and Steps #2-#4 will execute.

After Step #2 executes, the variable lc will contain the last – i.e. $(k+1)^{th}$ - character of s.

Step #3: deletes last character of s, then calls string-reverse recursively on s, and assigns the returned value to temp.

After Step #3 is executed, temp will contain a string of length k, containing the first k characters of s in reverse order (by the Inductive Hypothesis).

I.e. temp = "k^{th} $(k-1)^{th}$ ... 1^{st} characters of s".

Correctness Proof (contd.)

Step #4: concatenates (i.e. attaches) lc to the beginning of temp and then returns temp.

So after the concatenation temp will contain a string of length k+1.

I.e. temp = "$(k+1)^{th}$ k^{th} $(k-1)^{th}$ ... 1^{st} characters of s" This is exactly the reverse of the input string s.

Thus, we have shown that the algorithm:

Produces the correct answer for the base cases (empty or 1- character strings)

AND

Assuming that it produces the correct answer for all strings of length k, $k \geq 1$, it will produce the correct answer for a string of length (k+1)

SO

It will produce the correct answer for ALL strings of length 0 or more

How to prove that an algorithm is incorrect: Come up with a counterexample

Or produce a proof by contradiction

How to prove that an algorithm is correct:

Produce a proof by contradiction or a proof by LI How to prove that a recursive algorithm is correct:

Produce an inductive proof

How will you modify this string-reverse algorithm to pull off the first character instead of the last character each time?

1. Come up with your own correct recursive algorithm for reversing a string. Hint - try one of these two strategies: swap the last and first characters and recursively reverse the rest, or, a divide & conquer strategy: split the string into two halves and recursively reverse each half.
2. Write it in pseudocode and verify:
    1. Is it correct?
    2. How does it reverse strings?
    3. What if it gets the empty string as input?
    4. What if it gets a string of length 1?
    5. What if it gets a string of odd length? Even length? Does it matter?
    6. Draw the recursion tree if the input is the string “abcd”
3. Prove that it is correct using an inductive proof

Even More Thinking Assignments

Algorithm efficiency depends on data structures too:

A string can be represented either as an array of characters or as a linked list of characters with only a head pointer or with both a head and a tail pointer.

Which data structure best matches each of the four algorithms corresponding to the four strategies mentioned (pull off the first, or the last, character and recursively reverse the rest; swap the last and first characters and recursively reverse the rest; split the string into two halves and recursively reverse each half)? Why?

Notice how using the "wrong" data structure can decrease the efficiency of an algorithm.

AUBURN UNIVERSITY