Design and Analysis of Algorithms

Correctness of Algorithms

[Loop Invariance, Mathematical Induction]

Dr. Soharab Hossain Shaikh
BML Munjal University

1

What is Correctness?

- Simply, an algorithm is **correct** if for any **valid input** it produces the **result required** by the algorithm's **specification.**

- For example, a sorting function: **sort(int array[])**
  - We specify that for a **valid** integer array as input
  - The sort function will sort the input integer array into **ascending numerical order**
  - The result that is returned is a correctly sorted integer array, for any valid array of integers

2

Testing Not Sufficient

- Testing is a **critical skill** for a developer

- Provides **no guarantee** of correctness
  - Depends on generation of **good test-cases**
  - Generally these follow sensible heuristics
    - Test on correct values, e.g. **sort({1,2,3})**
    - Test on incorrect values, e.g. **sort({'a', 'b', 'c'})**
    - Test on boundaries, e.g. **sort({0, 1, …, MAX_VALUE})**

- However, this is by **no means exhaustive**, and there will be numerous missing tests
- Trade-off in **effort vs. completeness**
- Time constraints, developers have deadlines

3

How to Formalize?

- We can express our algorithms as **mathematical entities** and use various strategies to prove correctness (rather than guess work)

However, this is an expensive process.
- That is, the cost up front is expensive
- The benefits come late (**contradicts time to market**)
- Yet, software estimation is more art than science
  - Money, time, developers. As a rule, take your best guess and multiply it by 3
- But **can be useful for algorithms in particular**

Two approaches considered here:
- – **Loop Invariants**
- – **Mathematical Induction**

4

Proof
by
Loop Invariant

5

Loop Invariance

- In computer science, when designing algorithms/programs, loops are used to carry out repetitive tasks need to be performed. We can prove the correctness of the algorithm/program formally with a ***loop invariant***, where we **state that a desired property is maintained in a loop**.

Such a proof is broken down into the following parts:

- ***Initialization***: It is true before the loop runs for the first time.
- ***Maintenance (Update)***: If it is true before an iteration of a loop, it remains true before the next iteration.
- ***Termination (Conclusion)***: It is true after the last iteration (termination) of the loop.

6

Loop Invariance

- Non-trivial algorithms involve forms of looping
  - Iteration
  - Recursion
- Loop invariants allow **logical assertions** about the **behaviour** of the loop to be declared
  - Can be implicitly **documented** as comments
  - Can be explicitly **coded** as a condition or assertion

- By establishing the correctness of looping behaviour,
- We can move towards correctness of an algorithm
- As well as understand better **the effects of the loop**

7

Sum of all the Elements in an Array

Algo Array-Sum (A, n) // Array A with n elements

```
{
  answer = 0
  For i = 1 to n
    answer = answer + A[i]
  return answer
}
```

How to find loop invariance?

> Think about a specific iteration
> Think about what you want to know at the end

8

Loop Invariance Proof : Array-Sum Algorithm

We prove correctness by a loop invariant proof using the following invariant:
***Loop Invariant:** At the start of the *j*-th iteration of the loop, the variable *answer* should contain the sum of the numbers from the subarray A[1..j-1].

 # Initialization: At the start of the first loop the loop invariant states: 'At the start of the first iteration of the loop, the variable *answer* should contain the sum of the numbers from the subarray A[1..0], which is an empty array. The sum of the numbers in an empty array is 0, and this is what *answer* has been set to.

 # Maintenance: Assume that the loop invariant holds at the start of iteration *j*.
Then it must be that *answer* contains the sum of numbers in subarray A[1..j-1]. In the body of the loop we add A[j] to *answer*. Thus at the start of iteration *j*+1, *answer* will contain the sum of numbers in A[1..j], which is what we needed to prove.

 # Termination: When the **for**-loop terminates *i*=*n*+1.
Now the loop invariant gives: The variable *answer* contains the sum of all numbers in subarray A[1..(n+1)-1]=A[1..n]=A. This is exactly the value that the algorithm should output, and which it then outputs. **Therefore the algorithm is correct.**

9

Maximum Element of an Array

```
Algo Array-Max (A, n)   // Array A with n elements
{
  answer = A[1]
  For i = 2 to n
    {
      if (A[i] > answer) then
        answer = A[i]
    }
  return answer
}
```

10

Loop Invariance Proof : Array-Max Algorithm

***Loop Invariant:** At the start of the iteration with index j of the loop, the variable *answer* should contain the maximum of the numbers from the subarray A[1..j-1].

 # Initialization: At the start of the first loop, we have j=2. Therefore the loop invariant states: 'At the start of the iteration with index j of the loop, the variable *answer* should contain the maximum of the numbers from the subarray A[1..1], which is A[1]. This is what *answer* has been set to.

 # Maintenance: Assume that the loop invariant holds at the start of iteration *j*. Then it must be that *answer* contains the maximum of the numbers in subarray A[1..j-1]. There are two cases:
(1) A[j]>answer. From the loop invariant we get that A[j] is larger than the maximum of the numbers in A[1..j-1]. Thus, A[j] is the maximum of A[1..j]. In this case, the algorithm sets answer to A[j], thus in this case the loop invariant holds again at the beginning of the next loop.
(2) A[j] ≤ answer. That is, the maximum in A[1..j-1] is at least as large as A[j], thus the maximum of A[1..j] is the same as the maximum of A[1..j-1]. The algorithm also doesn't change *answer*, thus in this case the loop invariant holds again at the beginning of the next loop.

 # Termination: When the **for**-loop terminates *i*=*n*+1.
Now the loop invariant gives: The variable *answer* contains the maximum of all numbers in subarray A[1..(n+1)-1]=A[1..n]=A. This is exactly the value that the algorithm should output, and which it then outputs.

Therefore the algorithm is correct.

11

Proof by Mathematical Induction

12

Proof by Induction

Proof by Mathematical Induction done as follows:

- Prove for the **Base case**
- Make an **Assumption that the Hypothesis holds good for some arbitrary *k^{th} step***
- **Inductive case:** based on the Assumption prove that the Hypothesis holds good for the *$(k+1)^{th}$* step

13

Proof by Induction

- What is the sum of the first *n* powers of 2?
- $2^0 + 2^1 + 2^2 + \ldots + 2^n$

- k = 1: $2^0 = 1$
- k = 2: $2^0 + 2^1 = 1 + 2 = 3$
- k = 3: $2^0 + 2^1 + 2^2 = 1 + 2 + 4 = 7$
- k = 4: $2^0 + 2^1 + 2^2 + 2^3 = 1 + 2 + 4 + 8 = 15$

- For general n, the sum is $2^n - 1$

14

Proof by Induction

Theorem: *P(n)* holds for all $n \geq 1$

Proof: By induction on *n*

- Base case: *n*=1. Sum of first 1 power of 2 is 2^0, which equals $1 = 2^1 - 1$.
- Inductive case:
  - Assume the sum of the first *k* powers of 2 is 2^k-1
  - Show the sum of the first (*k*+1) powers of 2 is $2^{k+1}-1$

15

Proof by Induction

- The sum of the first k+1 powers of 2 is

$$\underbrace{2^0 + 2^1 + 2^2 + \ldots + 2^{(k-1)}}_{\text{sum of the first k powers of 2}} + 2^k$$

by inductive hypothesis

$$= 2^k - 1 \qquad + 2^k$$

$$= 2(2^k) - 1 = 2^{k+1} - 1$$

16

Proof: Prim's Algorithm

In Prim's algorithm, we start with a node and grow an MST out of it.

> We start with an arbitrary node ***u*** and include it in ***T***.

> We then find the lowest-weight edge incident on ***u***, and add this to ***T***.

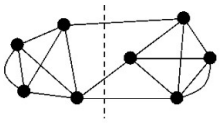
> We then repeat, always adding to ***T*** the minimum-weight edge that has exactly one endpoint in ***T***.

Slightly more formally, Prim's algorithm is the following:

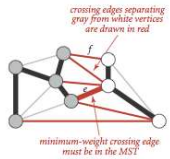
1. Pick some arbitrary start node ***u*** Initialize ***T = {u}***
2. Repeatedly the lowest-weight edge incident to T (the lowest-weight edge that has exactly one vertex in ***T*** and one vertex **not** in ***T***) until ***T*** spans all the nodes.

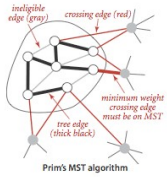
17

Graph Cut and Minimum-weight Edge Across a Cut



In graph theory, a **cut** is a partition of the vertices of a graph into two disjoint subsets. Any cut determines a cut-set, the set of edges that have one endpoint in each subset of the partition. These edges are said to cross the cut.





18

Graph Cut and Minimum-weight Edge Across a Cut

Let (S, S') be an arbitrary cut of a weighted connected graph, and let e be an edge across the cut (one endpoint in S, the other in S') that has the smallest weight of any edge across the cut. Then there is an MST T which contains e.

Let ***T*** be an arbitrary MST. If ***e∈T*** then we are done.

Otherwise, consider adding ***e*** to ***T***. This causes a cycle (as T is a tree and therefore, minimally connected), and there must be at least another edge ***e'*** in this cycle which crosses the cut.

By definition ***w(e)≤w(e'),*** so we can remove ***e'*** and add ***e*** to get an MST which contains ***e***.

Moreover, it is easy to see that any MST must contain some minimum-weight edge across the cut.

19

Proof: Prim's Algorithm

Let's use induction to prove - **Prim's algorithm correctly computes an MST.**

The **induction hypothesis** will be that **after each iteration, the tree *T* is a sub-graph of some minimum spanning tree *M*.**

Base case: This is trivially true at the start, since initially ***T*** is just a single node and no edges.

Inductive step: Now suppose (assumption) that at some point in the algorithm we have ***T*** which is a sub-graph of ***M***, and Prim's algorithm tells us to add the edge ***e***.

So we need to prove that ***T ∪ { e }* is also a sub-tree of some MST**.

If ***e∈M*** then the hypothesis is true, since by induction ***T*** is a sub-tree of ***M*** and ***e∈M*** and thus ***T∪{e}*** is a sub-tree of ***M***.

20

Proof: Prim's Algorithm

Now suppose that $e \notin M$. Consider what happens when we add e to M.

This creates a cycle (since M is a tree and therefore minimally connected). Since e has one endpoint in T and one endpoint not in T (since Prim's algorithm is adding it), there has to be some other edge e' in this cycle that has exactly one endpoint in T.

So, Prim's algorithm could have added e' but instead chose to add e, which means that $w(e) \leq w(e')$.

So, if we add e to M and remove e', we are left with a new tree M' whose total weight is at most the weight of M, and which contains $T \cup \{e\}$.

This maintains the induction, **so proves the theorem**.

One interesting point to note about the above proof is the fact that it must be the case that w(e) = w(e'), since if w(e)<w(e') then M' would have weight less than M, contradicting the assumption that M is an MST. But that fact wasn't necessary for the proof to go through.

21

Loop Invariance and Mathematical Induction

You may have noticed a similarity...

– **Loop Invariants** ideally have:

- **Initialization**
- **Maintenance/Update**
- **Termination/Conclusion**

– **Mathematical Induction** involves:

- **Proof of the Base Case**
- **Assumption**
- **Proof of Inductive step**

– The loop invariant should tell us about the starting point **P(1)**, then the progress of the loop, such that we can create an inductive proof for **P(k)** and for all next steps **P(k+1)**

– We might not always have a convenient mathematical form to prove (e.g. more complex data structures involved), but we can still use invariants and induction to reason and prove algorithm behaviour (for good **design** and **documentation**)

22

End of Lecture

23