Correctness & Time Complexity

Ric Glassey
glassey@kth.se
Outline

• Correctness
  – **Aim: Proving the correctness of algorithms**
  – Loop Invariants
  – Mathematical Induction

• Time Complexity
  – **Aim: Determining the cost of recursive algorithms**
  – Recursion reminder
  – Expressing recursive algorithms as recurrences
  – Applying the Master Theorem
CORRECTNESS
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[] in)`
  – 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
Why not just test?

- Testing is a **critical skill** for a developer
- Provides no guarantees of correctness
  - Depends on generation of **good test-cases**
  - Generally these follow sensible heuristics
    - Test correct values, e.g. `sort({1, 2, 3})`
    - Test incorrect values, e.g. `sort({'a', 'b', 'c'})`
    - Test 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
    - Let’s face it, developers have deadlines
Adopt more formalism

• 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
Invariants

• In English: adjective never changing

• IN CS: An invariant is a statement about the state of some variable(s) that can be relied to hold true during execution

• Two example types:
  – **Loop Invariants**: Invariants hold true before and after each iteration of the loop (e.g. for or while)
  – **Class invariants**: Invariants hold true throughout the life-span of an object instantiated from the class
Loop Invariants

• 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**
/ **
* Returns the sum 1 + 2 + ... + n, where n> = 1st
* /
public long sum (int n) {
  long sum = 0;
  int i = 1;
  while (i <= n) {
    // Invariant: sum = 1 + 2 + ... + (i - 1)
    sum+ = i;
    i +=;
  }
  return sum;
}
Loop Invariants: Good Properties

- **Three characteristics ensure goodness:**
  - **Initialisation**
    - The loop invariant must be true before we execute the loop
    - e.g. sum = 0 and i = 1
  - **Update**
    - If the loop invariant is true in one iteration, it must also be true in the next iteration of the loop
  - **Conclusion**
    - Finally, the loop invariant will tell us something useful about the conclusion of the loop, which should help us understand the behaviour
      - sum = 1 + 2 + ... + n-1 + n
Loop Invariants: In design of max( )

/ **
 * Returns the max value in the vector v.
 * /
public int max (int [] v) {
    int max = ...;
    for (int i = 0; i <v.length; i ++) {
        // Invariant: max = max (v[0], v[1], ..., v[i-1])
        ...
    }
    return max;
}
/ **
* Returns the max value in the vector v, where v.length > 0
*
* throws IllegalArgumentException if v.length = 0
* /
public int max (int [] v) {
    if (v.length == 0)
        throw new IllegalArgumentException ("v.length = 0");

    int max = ...;
    for (int i = 1; i <v.length; i ++) {
        // Invariant: max = max (v[0], v[1], ..., v[i-1])
        ...
    }
    return max;
}
Loop Invariants: In design of max( )

/ **
 * Returns the max value in the vector v, where v.length > 0
 *
 * throws IllegalArgumentException if v.length = 0
 */
public int max (int [] v) {
    if (v.length == 0)
        throw new IllegalArgumentException ("v.length = 0");

    int max = v [0];
    for (int i = 1; i <v.length; i ++) {
        // Invariant: max = max (v [0], v [1], ..., v [i-1])
        if (v [i]> max) {
            max = v [i];
        }
    }
    return max;
}
Afterthought...Class Invariants

• In Java, it is typical to encode some invariants into classes
  – e.g. a Date class only permits hours from $\geq 0$ to $<24$

• The main idea is that class methods ensure that there are no violations of the invariant
  – A form of **defensive programming**; or presuming the worst will probably happen
  – Although there may be temporary changes internally

• Java has no default syntax for invariant
  – Standard expressions and assertions can be used
  – 3rd Party efforts like Java Modelling Language
Mathematical Induction

• Method of mathematical proof
• Attempts to show that a statement is true for all natural numbers
• Consider: $3^n - 1$ is a multiple of 2

<table>
<thead>
<tr>
<th>n</th>
<th>statement</th>
<th>result</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>$3^1 - 1 = 3 - 1$</td>
<td>2</td>
</tr>
<tr>
<td>2</td>
<td>$3^2 - 1 = 9 - 1$</td>
<td>8</td>
</tr>
<tr>
<td>3</td>
<td>$3^3 - 1 = 27 - 1$</td>
<td>26</td>
</tr>
<tr>
<td>...</td>
<td>...</td>
<td>...</td>
</tr>
<tr>
<td>n - 1</td>
<td>$3^{n-1} - 1$</td>
<td>?</td>
</tr>
<tr>
<td>n</td>
<td>$3^n - 1$</td>
<td>?</td>
</tr>
</tbody>
</table>
Mathematical Induction: Domino Effect
Mathematical Induction

• Process
  – **Base Case:** Prove that the statement holds for the first natural number (usually 1 or 0)
    • e.g. for $P(n) = 3^n - 1$
    • base case is $P(1) = 3^1 - 1 = 2$
    • which is clearly true as 2 is a multiple of 2
  – **Induction Step:** prove that, if the statement holds for some natural number $k$, then the statement holds for $k + 1$

  $P(k) = 3^k - 1$ is assumed to be true
  $P(k+1) = 3^{k+1} - 1$
  $= 3 \times 3^k - 1$
  $= 3^k + 3^k + 3^k - 1$
  $= (2 \times 3^k) + (3^k - 1)$
Mathematical Induction: Sum

• We have seen sum as an iterative function with a loop invariant, but it can also be summarised by the proposition for any natural number n: 
  \[ P(n) = \frac{n(n + 1)}{2} \]

Base Case: 
  \[ P(1) = \frac{1(1 + 1)}{2} = 1 \]

Induction Step:
  \[ P(k) = \frac{k(k + 1)}{2} \]
  is assumed to be true

\[ P(k+1) = 1 + 2 + 3 + \ldots + k + (k+1) \]
  \[ = \frac{k(k + 1)}{2} + (k + 1) \]  # substitute P(k)
  \[ = \frac{(k(k + 1) + 2(k + 1))}{2} \]  # common denominator
  \[ = \frac{(k + 1)(k + 2)}{2} \]  # factor by k+1
  \[ = \frac{(k + 1).((k + 1) + 1)}{2} \]  # observe 2 as 1+1

Proving: 
  \[ 1+2+3+\ldots+k + (k+1) = \frac{(k+1).((k+1) + 1)}{2} \]
Mathematical Induction: Sum

- We have seen the iterative, mathematical forms of sum. We also have a recursive form:

```java
// **
// * Returns the sum 1 + 2 + ... + n, where n >= 1
// *
public int sum (int n) {
    if (n == 1) {
        return 1; // base case!
    }
    return n + sum (n-1);
}
```

# sum(0) - not permitted by the specification!
# sum(1) - returns 1
# sum(2) - returns 2 + sum(1) = 3
# sum(3) - returns 3 + sum(2) = 6
# ...
# sum(n) - returns n + sum(n-1) = n(n+1)/2
Mathematical Induction: Sum

Base Case:

\[ P(1) = 1 \text{ (directly written in conditional statement)} \]

Induction Step:

\[ P(k) = k + P(k - 1) \text{ is assumed to be true} \]

We choose \( P(k-1) \) for our next step (why?)
We know that \( P(k - 1) = 1 + 2 + \ldots + (k - 1) \)
So: \( k + (1 + 2 + \ldots + (k - 1)) = 1 + 2 + \ldots + (k - 1) + k \)

• The above illustrates the intuitive connection between induction and recursion
Mathematical Induction & Loop Invariants

• You may have noticed a similarity...
  – Loop Invariants ideally have:
    • Initialisation
    • Update
    • Conclusion
  – Mathematical Induction involves:
    • Base Case
    • Induction Step
    • Proof
  – 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 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)
Readings 1

• Loop invariants ** required **
  – http://www.nada.kth.se/~snilsson/algoritmer/loopinvariant/
  – Example 3 in particular

• Induction & recursion ** required **
  – http://www.nada.kth.se/~snilsson/algoritmer/induktion/

• Overview of mathematical induction and how it relates to Computer Science
  – Gentle introduction that ties together induction and invariants in a comprehensible manner

• Loop invariants abbreviate inductive proofs
TIME COMPLEXITY
Progress and Running Time

• Invariants not only tell us something about the **progress** of an algorithm:
  – e.g. For a sorting algorithm
    • So far, all items are sorted up to some $n$ [progress]

• They can tell us about **running time** or cost
  – e.g. For a sorting algorithm
    • The worst case performance will be $O(n^2)$ [running time]

• Complexity for iterative algorithms is mostly an exercise in counting operations, then stating the complexity (covered previously...i hope)

• How can we estimate **complexity of recursive algorithms**?
• Choose a number between 1-100

• How can we guess?
  – Linear search
    • Worst case is $O(n)$
  – Random search
    • Again, worst case can be $O(n)$
  – Is there a smarter strategy or algorithm?

• Think about how we can divide and conquer
  – Divide: break into 2 or more sub-problems, whose solution is similar to the solution of the main problem
  – Conquer: call recursively until an end point is reached
  – Combine: bring all the results of sub-problems together
Visualising Binary Search

$x > 50$?

$x > 25 \land \land < 50$?
Expressing Recurrences: Binary Search

• It is beneficial to express our recursive function as a recurrence in order to reason about its behaviour and work towards estimating its time complexity.

• Let $T(n)$ be the **number of questions** in binary search on a range of numbers between 1 and $n$, assuming $n$ is a power of 2, then we have:

$$T(n) = \begin{cases} 
T(n/2) + 1 & \text{if } n \geq 2 \\
1 & \text{if } n = 1 
\end{cases}$$

• That is, number of guesses is equal to 1 step (the guess) plus the time to solve remaining $n/2$ items.
Mergesort

- Popular recursive algorithm for sorting a list of items \( (A) \) within an expressed range \( (\text{low}--\text{high}) \)
  - Base case is when \( \text{low}=\text{high} \) we end, or
  - Make two recursive calls on problems of size \( n/2 \)
  - Finally combines sorted lists via an iterative merge

- Can be expressed in pseudo-code as follows:

\[
\text{MergeSort}(A, \text{low}, \text{high})
\]
\[
\text{if } (\text{low} == \text{high})
\]
\[
\text{return}
\]
\[
\text{else}
\]
\[
\text{mid} = (\text{low} + \text{high}) / 2
\]
\[
\text{MergeSort}(A, \text{low}, \text{mid})
\]
\[
\text{MergeSort}(A, \text{mid}+1, \text{high})
\]
\[
\text{Merge the sorted lists from the previous two steps}
\]
Mergesort Visualised

mergesort(A, 0, A.length)

mergesort(A, 0, A.length/2)

merge sorted lists)
Expressing Recurrences: Mergesort

• As with BinarySearch, we can express Mergesort as a mathematical recurrence.
• To reflect the two recursive calls we state $2T$:
  – Indicates the branching factor.
• We still split the problems using $n/2$.
• Finally, $+n$ represents the merge effort:
  – Given a list of size $n$, the worst case is $n$ steps to sort.

$$T(n) = \begin{cases} 
2T(n/2) + n & \text{if } n > 1 \\
1 & \text{if } n = 1 
\end{cases}$$
Recursion Tree for Mergesort

Problem Size

Work

Total work done will be $n(\log_2 n + 1)$
Using the Master Theorem

• If we can express our recursive function as a recurrence in the following form, we can approximate its time complexity using the Master Theorem

\[
T(n) = \begin{cases} 
  aT(n/b) + n^c & \text{if } n > 1 \\
  d & \text{if } n = 1 
\end{cases}
\]

a = number of sub-problems, or branches in the recursion tree
b = fractional size of the sub-problem, within each branch
c = exponent of additional work to be done (e.g. \(n^0\) for 1 work unit, or \(n^2\) work units)
d = some constant
Using the Master Theorem

• Given a recurrence of the form:

\[ T(n) = \begin{cases} 
  aT(n/b) + n^c & \text{if } n > 1 \\
  d & \text{if } n = 1 
\end{cases} \]

• We can produce an estimate of complexity

\[ T(n) = \begin{cases} 
  \Theta(n^c) & \text{if } a < b^c \\
  \Theta(n^c \log n) & \text{if } a = b^c \\
  \Theta(n^{\log b a}) & \text{if } a > b^c 
\end{cases} \]
Master Theorem: Examples

\[ T(n) = 2T(n/2) + n \]  
(mergesort)

\[ a=2, \ b=2, \ c=1 \]

so, \( b^c = 2 = a \), time complexity is \( \Theta(n^1 \log_2 n) \)

\[ T(n) = 8T(n/2) + 1000n^2 \]  
(no idea!)

\[ a=8, \ b=2, \ c=2 \]

so, \( b^c = 4 < a \), time complexity is \( \Theta(n^{\log_2 8}) \) or \( \Theta(n^3) \)

And try yourselves for binary search: \( T(n) = T(n/2) + 1 \)
Readings 2

• Time complexity ** required
  – http://www.nada.kth.se/~snilsson/algoritmer/rekursion/

• Introduction to recursion and recurrences
  – https://math.dartmouth.edu/archive/m19w03/public_html/Section4-2.pdf
  – https://math.dartmouth.edu/archive/m19w03/public_html/Section5-1.pdf

• Master Theorem
  – https://math.dartmouth.edu/archive/m19w03/public_html/Section5-2.pdf
  – Nice and easy video on Master Theorem + Mergesort :-) https://www.youtube.com/watch?v=B_TJcOaiJiU