Correctness of Algorithms

CS:3330

What does an algorithm?

- An algorithm takes inputs and produces output:
  - **Precondition**: restrictions on input data
  - **Postcondition**: what is the result
- Example: Binary Search
  - Input: a:array of integer; x:integer;
  - Output: result:integer;
  - Precondition: a is sorted in ascending order
  - Postcondition: result = -1 if x is not in a[]; otherwise, it is a position of x in a[].

Correct algorithms

- An algorithm is correct if:
  - for any correct input:
    - it stops and
    - it produces correct output.
  - Correct input: satisfies precondition
  - Correct output: satisfies postcondition

Proving correctness

- An algorithm = a list of actions
- **Proving that an algorithm is totally correct**:
  1. Proving that it will terminate (taken care by complexity analysis)
  2. Proving that the list of actions applied to the precondition imply the postcondition
    - This is easy to prove for simple sequential algorithms
    - This can be complicated to prove for repetitive algorithms (containing loops or recursion)

Example – a small algorithm

```java
// swap two elements in a[]
static void swap(int a[], int x, int y) {
    int aux = a[x];
    a[x] = a[y];
    a[y] = aux;
}
```

**Precondition:**
- 0 <= x, y < a.length,
- a[x] == c, a[y] == d

**Postcondition:**
- a[x] == d, a[y] == c

**Proof:** the list of actions applied to the precondition imply the postcondition
1. Precondition:
   - 0 <= x, y < a.length,
   - a[x] == c, a[y] == d
2. “aux = a[x]” => aux == a[x]
3. “a[x] = a[y]” => a[x] == d
4. “a[y] = aux” => a[y] == c
5. a[x] == d, a[y] == c are the postcondition

Mathematical induction - Review

- Let T(n) be a theorem that we want to prove, where n is a natural number.
- Proving that T(n) holds for all natural values of n is done by proving following two conditions:
  1. T(0) is true (base case)
  2. For every n>=0, (T(n) => T(n+1)) is true (inductive case)

**Terminology:** T(n) is Induction Hypothesis
The number 0 can be replaced by c, if we want to prove that T(n) is true for all n >= c.
We may prove (T(n-1) => T(n)) in the inductive case.
Let \( f(n) \) be defined by the following recurrence relation:
\[
\begin{align*}
f(1) &= 1; \\
f(n) &= 2f(n-1) + 1.
\end{align*}
\]

**Theorem T(n):** \( f(n) = 2^n - 1 \)

Prove by induction on \( n \).

- **Base step:** \( n = 1 \).
  - Left: \( f(1) = 1 \);
  - Right: \( 2^1 - 1 = 1 \)

- **Induction hypothesis:** \( f(n-1) = 2^{n-1} - 1 \).

- **Inductive step:** For \( n > 1 \),
  - Left: \( f(n) = 2f(n-1) + 1 = 2(2^{n-1} - 1) + 1 = 2^n - 1 \).
  - Right: \( 2^n - 1 \).

---

**Mathematical induction - review**

- **Strong Induction:** a variant of induction where the inductive step builds up on all the smaller values
- **Proving that T(n) holds for all natural values of n is done by proving following two conditions:**
  1. \( T(c) \) is true (base case)
  2. For every \( n > c \), \( c \leq k < n \) and \( T(k) \Rightarrow T(n) \) is true (inductive case)

---

**Typical Errors in Math Induction**

- **Missing the base case:**
  - Counter Example: Prove \( T(n) \): \( n < n \).
  - Inductive case: We show \( T(n) \Rightarrow T(n+1) \).
  - If \( T(n) \) is true, then \( n+1 < n \), so \( (n+1) + 1 < (n+1) \), i.e., \( T(n+1) \) is true.
  - So \( T(n) \Rightarrow T(n+1) \) is true.

- **Proving \( T(n) \Rightarrow T(n-1) \) in inductive case:**
  - Counter Example: Prove \( T(n) \): \( n < n \).
  - Base case: \( T(0) \) is true because \( 0 < 1 \).
  - Inductive case: If \( T(n) \) is true, then \( n < 1 \).
  - So \( n-1 < 0 \), i.e., \( T(n-1) \) is true.
  - So \( T(n) \Rightarrow T(n-1) \) is true.

- **Using invalid induction hypothesis:**
  - Counter Example: Prove \( T(n) \): \( n \) is even.
  - Base case: \( T(0) \) is true because \( 0 \) is even.
  - Induction hypothesis: \( T(n-2) \) is true.
  - Since \( (n-2) \) is even, so \( (n-2) + 2 = n \) is even.

---

**Correctness of algorithms**

- **Induction can be used for proving the correctness of repetitive algorithms:**
  - Recursive algorithms
    - Direct induction
      - Hypothesis = a recursive call itself; often a case for applying strong induction
  - Iterative algorithms:
    - Loop invariants (complex technique)
      - Induction hypothesis = loop invariant = relationships between the variables during loop-execution

---

**Correctness of Serial Search**

```java
L1: static int recSerS(int[] a, int target, int n) {
  // Pre: n < a.length */
  /* Post: if target is in a[0..n], then the output is position of target in a[0..n]; otherwise it's -1. */
  L2: if (n == -1) return -1; // target not found
  L3: if (a[n] == target) return n; // target found
  L4: return recSerS[a, target, n - 1];
} 
```

**Claim:** If \( \text{Pre} \) is true, \( \text{Post} \) is true after the call \( \text{RecSerS}(a, \text{target}, n) \).

**Proof:** By induction on \( n+1 \) (number of elements in \( a[0..n] \)).

**Base case:** \( n = -1 \)
It means \( a[0..n] \) is empty, so target cannot be in \( a[0..n] \) and \( \text{Post} \) is trivially true.

**Inductive Hypothesis:** \( \text{Post} \) is true when \( n < n \).

**Inductive case:**
L3: \( n \) is returned when \( a[\text{target}] \) is true, so \( \text{Post} \) is true.
L4: if \( a[n] \) is target, \( \text{target is in a[0..n]} \) iff \( \text{target is in a[0..n-1]} \). By induction hypothesis, \( \text{recSerS}(a, \text{target}, n - 1) \) returns the correct answer.
Correctness of Binary Search

L1: static int recBinS(int[] a, int target, int min, int max) {
    /* Pre: a[] is sorted */
    /* Post: if target is in a[min..max], then the returned is position of target in a[]; otherwise it's -1. */
    L2: if (min > max) return -1; // target not found
    L3: int mid = (min + max) / 2;
    L4: if (a[mid] == target) return mid; // target found
    L5: if (a[mid] < target) return recBinS(a, target, mid + 1, max);
    L6: return recBinS(a, target, min, mid - 1);
}

Claim: If Pre is true, Post is true after the call RecBinS(a, target, min, max)
Proof: By induction on the size of max – min + 1.
Base case:
L2: min>max means a[min..max] is empty, so target cannot be in a[min..max] and Post is trivially true.
L3: mid is returned when a[mid] == target, Post is true.
Inductive Hypothesis: Post is true when (max' – min' + 1) < (max – min + 1).

Correctness of Binary Search

L1: static int recBinS(int[] a, int target, int min, int max) {
    /* Pre: a[] is sorted */
    /* Post: if target is in a[min..max], then the returned is position of target in a[]; otherwise it's -1. */
    L2: if (min > max) return -1; // target not found
    L3: int mid = (min + max) / 2;
    L4: if (a[mid] == target) return mid; // target found
    L5: if (a[mid] < target) return recBinS(a, target, mid + 1, max);
    L6: return recBinS(a, target, min, mid - 1);
}

Claim: If Pre is true, Post is true after the call RecBinS(a, target, min, max)
Proof: By induction on max-min+1.
Inductive case:
L5: if (a[mid] < target) is true, because a[] is sorted, “target is in a[min..max]” iff “target is in a[mid+1..max]”. By induction hypothesis, recBinS(a, target, mid + 1, max) returns the correct answer.
L6: We have (a[mid] > target), and it’s analogue to the case of L5.

Prove Interactive Algorithms

Example: Sum of N numbers
Input: a[], an array of N numbers
Pre: a[] has at least N numbers.
Output: integer s
Post: s is the sum of the first N numbers in a[].

static int sum(int a[], int N) {
    int s=0;
    int k=0;
    while (k<N) {
        s = s + a[k];
        k = k + 1;
    }
    return s;
}

Proof: the list of actions applied to the precondition imply the postcondition
BUT: we cannot enumerate all the actions in case of a repetitive algorithm !
We use techniques based on loop invariants.

Loop invariants

• A loop invariant is a theorem (a logical statement) about the loop: if it is satisfied before entering any single iteration of the loop then it is also satisfied after the iteration.

Using loop invariants in proofs

• We must show the following 3 things about a loop invariant:
  1. Initialization: It is true prior to the first iteration of the loop.
  2. Maintenance: If it is true before an iteration of the loop, it remains true before the next iteration.
  3. Termination: When the loop terminates, the invariant gives us a useful property that helps show that the algorithm is correct.
Example: Proving the correctness of the Sum algorithm (1)

Input: a, an array of N numbers
Pre: a[] has at least N numbers.
Output: integer s
Post: s is the sum of the first N numbers in a[].

static int sum(int a[], int N) {
    int s=0;
    int k=0;
    while (k<N) {
        s = s + a[k];
        k = k + 1;
    }
    return s;
}

Example: Proving the correctness of the Sum algorithm (2)

Input: a, an array of N numbers
Pre: a[] has at least N numbers.
Output: integer s
Post: s is the sum of the first N numbers in a[].

static int sum(int a[], int N) {
    int s=0;
    int k=0;
    while (k<N) {
        s = s + a[k];
        k = k + 1;
    }
    return s;
}

Example: Proving the correctness of the Sum algorithm (3)

Input: a, an array of N numbers
Pre: a[] has at least N numbers.
Output: integer s
Post: s is the sum of the first N numbers in a[].

static int sum(int a[], int N) {
    int s=0;
    int k=0;
    while (k<N) {
        s = s + a[k];
        k = k + 1;
    }
    return s;
}

Loop invariants and induction

• Proving the loop invariants is similar to mathematical induction:
  – showing that the invariant holds before the first iteration corresponds to the base case, and
  – showing that the invariant holds from iteration to iteration corresponds to the inductive step.