Notes, 3(b)

ECE 606

Termination

Example: the following algorithm purports to check if an integer i is in the array $A[1, \ldots, n]$ of integers.

```
isInArray(A[1,...,n], i)
1 foreach j from 1 to n do
2     if i = A[j] then return true
3 return false
```

Claim 1. isInArray *meets/possesses the termination property.*

Proof. We assume that: (i) n is a finite positive integer, (ii) each $A[j]$ is a finite integer, and, (iii) i is a finite integer. Then, each comparison of i with $A[j]$ in Line (2) is guaranteed to terminate with the correct true/false result. Now if i is indeed in A, then we will eventually return in Line (2) thus guaranteeing termination. If i is not in A, then we know that the **foreach** loop of Lines (1)–(3) will eventually end, and we will return in Line (3) thus, again, guaranteeing termination. (2) □

An algorithm that does not possess the termination property, but nevertheless may be deemed useful: our randomized algorithm for finding the median, RandMedian — see Lecture 2(a), or the next page.

Correctness

Claim 2. isInArray *above is correct.*

Proof. We need to prove that when the algorithm halts, it does so with correct output. The algorithm isInArray can halt in one of two lines only: (i) Line (2), or, (ii) Line (3).

In Case (i), the true return value is preceded by a check that i is indeed in the array. So for that case, the algorithm is correct.

In Case (ii), the **foreach** loop checks against every possible entry in the array, and only if none is i, do we reach Line (3). Therefore in this case as well, the algorithm is correct. □

```
RandMedian(A[1, ..., n])
1 while true do
2     i ← uniformly random choice from 1, ..., n
3     c ← 0
4     foreach j from 1 to n do
5         if A[j] < A[i] then c ← c + 1
6     if c = (n−1)/2 then return i
```

Claim 3. RandMedian *is correct.*

Proof. The only line in which we return is Line (6). Thus, at the moment we return i, we know that $c = (n-1)/2$ given the **if** condition immediately prior to the **return** statement. But from Lines (4)–(5), we know that c is exactly the number of entries that are smaller than $A[i]$, and we know that m is the median of $A[1, \ldots, n]$, where n is odd and each item in A is distinct if and only if exactly $(n-1)/2$ items are smaller than m. □

The textbook has another example, that of an approximation algorithm for an optimization problem. That algorithm is correct, but only because we adopt an eased objective. The problem is that of identifying a minimum-sized vertex cover of an undirected graph.

Example: binary search

Input: (i) $A[1, \ldots, n]$ of sorted positive integers, (ii) lo, hi with $1 \leq lo \leq hi \leq n$, and, (iii) a positive integer i.

Output: true if $i \in A[lo, \ldots, hi]$, false otherwise.

```
BinSearch(A, lo, hi, i)
1 while lo ≤ hi do
2     mid ← ⌊(lo+hi)/2⌋
3     if A[mid] = i then return true
4     if A[mid] < i then lo ← mid + 1
5     else hi ← mid − 1
6 return false
```

Example: $A = [1, 8, 8, 14, 123, 645, 646, 1023]$, BinSearch$(A, 2, 7, 134)$.

Iteration #	*lo*	*hi*	*mid*	*condition*	*change*
1	2	7	4	$A[4] = 14 < 134$	$lo \leftarrow 5$
2	5	7	6	$A[6] = 645 > 134$	$hi \leftarrow 5$
3	5	5	5	$A[5] = 123 < 134$	$lo \leftarrow 6$
4	$lo = 6 > 5 = hi$, return false in Line (6)				

Claim 4. BinSearch *is guaranteed to terminate.*

We first split two cases: (1) $lo > hi$, and, (2) $lo \leq hi$. In case (1), we are guaranteed to terminate immediately because... Case (2) we can prove by induction on $hi - lo + 1$.

Step: if the **if** condition in Line (3) is true, we terminate in this iteration. Otherwise, we have two cases for lo', hi', which are the new values of lo, hi respectively.

Case (a): $lo' = mid + 1, hi' = hi$

$$\begin{aligned} hi' - lo' &= hi - mid - 1 \\ &= hi - \left\lfloor \frac{lo + hi}{2} \right\rfloor - 1 \\ &< hi - \left(\frac{lo + hi}{2} - 1 \right) - 1 \qquad \because \forall x \in \mathbb{R}, x - 1 < \lfloor x \rfloor \\ &= \frac{hi - lo}{2} \leq hi - lo \end{aligned}$$

Case (b): $lo' = lo, hi' = mid - 1$ – see textbook.

Claim 5. BinSearch *is correct.*

We need to prove two things:

(A) If BinSearch$(A, lo_{\text{in}}, hi_{\text{in}}, i)$ halts with output true, then $i \in A[lo_{\text{in}}, \ldots, hi_{\text{in}}]$, and,

(B) If BinSearch$(A, lo_{\text{in}}, hi_{\text{in}}, i)$ halts with output false, then $i \notin A[lo_{\text{in}}, \ldots, hi_{\text{in}}]$.

For (A), we observe that the only line in which BinSearch returns true is Line (3). And that happens only if $A[mid] = i$. So, it suffices that we prove that $lo_{\text{in}} \leq mid \leq hi_{\text{in}}$ at the moment we return true.

This clarifies why we add the subscript "in" to the input values of lo and hi: because the values of lo and hi can change during a run of the algorithm.

We prove this in the textbook by proving that at any point in the run of the algorithm, the values of lo and hi satisfy:

- $lo \leq mid \leq hi$ at any moment we compute mid in Line (2), and,
- $hi \leq hi_{\text{in}}$ and $lo \geq lo_{\text{in}}$.

Together, we can infer that at the moment we return true in Line (3), $lo_{\text{in}} \leq lo \leq mid \leq hi \leq hi_{\text{in}}$.

We prove (B) by induction on $hi_{\text{in}} - lo_{\text{in}}$.

(Non-)existence of algorithms

Natural question: given a function, is it guaranteed that there exists an algorithm that computes it? Answer: no. Proof: by construction. We propose a function and prove that there can exist no algorithm that computes it.

To start off the proof: we assume that every algorithm can be encoded in our preferred alphabet of symbols, i.e., it can be written down.

Such an encoding can then be provided to our hypothetical computer to run the algorithm.

Consider the following function expressed as an algorithmic problem:

Given as input: (i) an encoding of an algorithm A that takes a string as input, and, (ii) a string x that is intended to be an input to A.

Does there exist an algorithm that:

Outputs true if A when run with input x halts (terminates); Outputs false otherwise, i.e., A goes into an infinite loop when run with input x.

We will prove that no algorithm exists for the above problem.

Proof by contradiction: assume such an algorithm, call it B, exists. Then construct two new algorithms, call them C and D, as follows:

$C(x, y)$
  if $B(x, y) =$ false **then return**
  else go into infinite loop

$D(z)$
  $C(z, z)$

Now suppose we invoke $B(D, D)$. Then:

- $B(D, D) =$ true $\implies$ $D(D)$ halts $\implies$ $C(D, D)$ halts $\implies$ $B(D, D) =$ false.
- $B(D, D) =$ false $\implies$ $D(D)$ does not halt $\implies$ $C(D, D)$ does not halt $\implies$ $B(D, D) =$ true.

Hence the contradiction.

Lesson to learn: need to be careful when devising an algorithm given a problem. One may not even exist.

We will go on to see that there exist problems for which even if an algorithm exists, an efficient is unlikely to exist.