CS 624 - HW 2

Max Weiss

10 February 2020

1. Suppose that x is a binary tree. I'll write $c(x)$ for the set of trees rooted at child nodes of x. Every binary tree belongs to the smallest set X such that (i) each singleton tree belongs to X, and (ii) if $y \in X$ for all $y \in c(x)$, then $x \in X$. So, to show that some predicate P holds of all binary trees, it suffices to show

$$\forall x(Sx \to Px) \wedge \forall x(\forall y(y \in c(x) \to Py) \to Px) \tag{1}$$

where S defines the class of singleton trees.

I'll also write $c^*(x)$ for the smallest set X such that $x \in X$ and $y \in X$ for all $y \in c(x)$. Let's say that a predicate P *perists downward* provided that Py follows from $y \in c(x)$ and Px. Then,

to prove that $\forall y(y \in c^*(x) \to Py)$, it suffices to prove both that Px and that P persists downward.

It follows that

$$\forall x(Px \wedge P \text{ perists downward} \to \forall z(z \in c(y) \to Pz)) \to \forall y(y \in c^*(x) \to Py)). \tag{2}$$

For a lemma, let me argue that the property of being a pre-tree persists downward. So, suppose that x is a pre-tree, and that y, z are the trees rooted at the left and right children of x. Then there are four possibilities: y, z are complete of height n, or they are complete of heights n, $n-1$, or y is complete of height n and z is a pretree of height n, or y is a pretree of height n and z is complete of height $n-1$. In any case, both y and z are pretrees, as required.

Finally, for the main claim to be proven: suppose that P persists downward, that Q is true of all singleton trees, and that H_1, H_2, P, Q are predicates satisfying

$$H_1x \leftrightarrow Px \wedge \forall y(c^*(y) \to Qy) \tag{3}$$

$$H_2x \leftrightarrow Px \wedge Qx \wedge \forall y(y \in c(x) \to H_2y). \tag{4}$$

I'll argue that $H_1x \leftrightarrow H_2x$ for all binary trees x.

$\Leftarrow$. I'll use the induction (1). The claim is clear for singletons. So, assume that H_1x holds. From (3), it follows that H_1x implies Px and also Qx. By (4), it therefore remains to show $\forall y(y \in c(x) \to H_2y)$.

So, assume $y \in c(x)$; I will first argue that H_1y. Since P persists downward, we do have Py. Furthermore, $c^*(y) \subseteq c^*(x)$, which implies that $\forall z(z \in c^*(y) \to Qz)$. So by (3), H_1y does hold. By induction hypothesis, we can thus infer H_2y as required.

$\Rightarrow$. Suppose that H_2x holds; I will argue for H_1x. From (4), it is immediate that Px holds. So by (3), it remains to establish $\forall y(y \in c^*(x) \to Qy)$. Using the induction (2) on $c^*(x)$ it will be easier to prove $\forall y(y \in c^*(x) \to H_2y)$. So, first suppose $y = x$; then H_2y amounts to H_2x, and this we assumed at the outset. On the other hand, given H_2y, the third clause of the (4) immediately implies H_2z for any $z \in c(y)$. This completes the proof that H_2y holds for all $y \in c^*(x)$, as desired.

2 (6.5.8, p166. Suppose we extend the ordering of keys (i.e., the ordering on the natural numbers, if that's what the keys are) so that $\infty > k$ for every key k, where ∞ is some new symbol. Now consider the following algorithm:

```
heap_increase_key(A, i, infinity)
heap_extract_max(A)
```

Since $\infty > A[i]$, the result of line 1 is that A is a heap whose elements are precisely those of A except with ∞ in place of $A[i]$. Since $\infty > k$ for all keys k of nodes in A, therefore the result of line 2 is that A is a heap whose elements are precisely those of A except without $A[i]$. Therefore, the given algorithm implements HEAP-DELETE.

3 (6.5.9, p166). If A, B are heaps, write $A < B$ if $A[1] < B[1]$. In the below, I assume that H is an array of k heaps with n total elements.

```
Z = MaxHeap()
S = [None]*n
for L in H:
    maxheap_insert(Z, L)
for i in 1 .. n+1:
    L = maxheap_extract_max(Z)
    x = L.pop()
    S[i] = x
if L is not empty:
    maxheap_insert(Z, L)
```

This algorithm breaks into two tasks: first, inserting the k heaps into the meta-heap Z, and second, inserting elements of the heaps into the array S.

The first task requires k insertions into a heap of size at most k; hence it is in $O(k \lg k)$. The second task requires n iterations of (at most) the following:

heap-extract the minimum L from Z; pop the last (maximum) x from L and array-insert x into S; and heap-insert L back into Z. Since Z has size at most k, the heap-insertions and extractions are $O(\lg k)$, while the list operations are constant time. Consequently, the second task is $O(n \lg k)$. So, the whole thing is $O(n \lg k)$.

4 (6.1, lecture 3). Let S be an unsorted array of n distinct elements. Consider the following algorithm:

```
build_min_heap(S)
R = []
for _ in 0 .. k:
    R.push(minheap_extract_min(S))
```

The first line of the algorithm converts S into a MinHeap. I'll now argue that the ith iteration of the loop satisfies this invariant:

- S is a MinHeap
- each element of R is less than all elements of S
- R has i elements and S has $n - i$ elements

This is clear for $i = 0$, so suppose it holds for i. Since S is now a heap with $n - i$ elements, the effect of calling MinHeap-Extract-Min is to convert S into a heap with $n - i - 1$ elements and to return the smallest element x of those. So when x is pushed onto R, the result is that R contains $i + 1$ elements, all of which are smaller than all elements of S. This establishes the invariant. It follows that after the prescribed k loop iterations, R contains the k smallest elements of the originally given set, as desired.

As for running time: the call to Build-MinHeap is $O(n)$, while each of the k calls to MinHeap-Extract-Min (and push of the result onto a list) costs $\lg n$. So, the overall running time is $O(n + k \lg n)$.

5 (7.3.2, p180). There is roughly one call to Random per call to Partition. So, the question amounts to how many calls to Partition there are in the worst and best cases. It is tempting simply to count the number of Partition calls in the worst and best cases for the running time of QuickSort. However, the contributions to running time of those calls are weighted by the size of the subarrays they transform. Here, we are considering a notion of "running time" where the contributions are unweighted.

The worst case cannot generate more than n calls to Partition, since each call consumes an element of the input array as its pivot. Conversely, if the random number generator always returns the largest possible array index, then all n calls will be issued. So, the worst case requires n calls to Random.

On the other hand, in any case it is clear that every array entry eventually gets used as a pivot (even if only on a length-one subarray), so each array entry

requires at least one unique call to RANDOM. So, the best case cannot improve on n calls to RANDOM, and any best case is a also worst case. From this, I conclude that the number of calls to RANDOM is $\Theta(n)$.

This reasoning can be verified using the substitution method. Let's write $T^g(n)$ for the running time on n inputs given the random number generator g. And suppose $\sup_g T^g(k) = k = \inf_g T^g(k)$ for all $k < n$. Then

$$\begin{aligned}
\sup_g T^g(n) &= \max_{0 \le q \le n-1} (T(q) + T(n-q-1)) + 1 \\
&= \max_{0 \le q \le n-1} (q + n - q - 1) + 1 \\
&= n \\
&= \min_{0 \le q \le n-1} (q + n - q - 1) + 1 \\
&= \min_{0 \le q \le n-1} (T(q) + T(n-q-1)) + 1 \\
&= \inf_g T^g(n).
\end{aligned}$$

6a (7.4, p188). I will argue by induction on $r - p$ that upon conclusion of TR-QUICKSORT(A, p, r), the array $A[p..r+1]$ is correctly sorted. The claim is trivial for $r - p = 0$. So, suppose that it holds for all $t < r - p$. Let's write $\hat{p}$ for the lower array index p on which TR-QUICKSORT is initially called. I will now argue, by an inner induction on i, that the ith call to TR-QUICKSORT(A, p, r), as generated by TR-QUICKSORT$(A, \hat{p}, r)$, satisfies the following invariant:

(1) $p \ge \hat{p} + i$,

(2) $A[\hat{p}..r+1]$ is partitioned with pivot $p - 1$,

(3) $A[\hat{p}..p]$ is sorted.

From this invariant, it will follow that after at most $r - \hat{p}$ iterations, $A[\hat{p}..r]$ is sorted. The condition is trivial for $i = 0$, so suppose it holds for i; I'll argue that it holds for $i + 1$.

By the inner induction hypothesis, we can assume that where p_0 is the value of p after the ith step, the following hold:

(1) $p_0 \ge \hat{p} + i$,

(2) $A[\hat{p}..r+1]$ is partitioned with pivot $p_0 - 1$,

(3) $A[\hat{p}..p_0]$ is sorted.

Observe that the correctness of PARTITION ensures that after execution of line 3 of the pseudocode, we have that $A[p_0..r+1]$ is partitioned with q as pivot. It also implies that $q - 1 < r$. So, from the outer induction hypothesis, we can infer that TR-QUICKSORT$(A, p_0, q-1)$ is correct. So after line 4, the subarray $A[p_0..q]$ must be sorted. Since q was pivot of a correct partition call on line 3, this sort actually covers $A[p_0..q+1]$. And indeed, by combining this result with

clauses (2) and (3) of the inner induction hypothesis, it follows that after line 4, it is all of the subarray $A[\hat{p}..q+1]$ which must be sorted.

The last line of the pseudocode is the assigment $p = q + 1$. Since $A[\hat{p}..q+1]$ is sorted, this confirms invariant clause (3) for step $i + 1$. Moreover, since q was pivot of a correct partition on $A[p_0..r+1]$, and since $A[\hat{p}..q+1]$ is sorted, therefore $p - 1 = 1$ is a pivot of a correct partition on $A[\hat{p}..r+1]$, confirming invariant (2). Finally, since $q \geq p_0 \geq \hat{p} + i$, it follows that $p = q + 1 \geq \hat{p} + i + 1$, which confirms clause (1).

6b (7.4, p188). Suppose that the array $A[1..n+1]$ is already sorted. This will then require n nested calls to TR-QUICKSORT, and so the maximum stack depth can be as bad as $O(n)$.

6c (7.4, p188). Not sure how to guarantee a logarithmic bound on the stack depth without turning the algorithm into mergesort....

7. Let x be a binary tree, and let d be the depth of x. Let me write $|x|$ for the number of nodes of x. I will use the lemma (1) from question 1 to prove the stronger claim that

$$|x| \leq 2^d - 1. \tag{5}$$

If x is a singleton, then x has one node, and depth 1; hence x has at most $1 = 2^d - 1$ leaves. So, suppose that (5) is satisfied by all trees rooted at children of x. Those trees must have at most depth $d-1$ (this follows from the definition of depth). Furthermore, there are at most two such trees. Thus,

$$\begin{aligned} \sum_{y \in c(x)} |y| \;&\leq\; 2^{d-1} - 1 + 2^{d-1} - 1 \\ &=\; 2^d - 2. \end{aligned} \tag{6}$$

Meanwhile,

$$|x| \;\leq\; 1 + \sum_{y \in c(x)} |y|. \tag{7}$$

The desired result is immediate from (6) and (7).

7 redux. As suggested in the homework, it is also possible to prove the result using "mathematical induction", i.e., to deduce (5) from the induction principle

$$P0 \wedge \forall x(Px \to Px') \to \forall x(Nx \to Px) \tag{8}$$

where x' represents the successor of x.

7a. Hmm.... The induction principle (8) requires the provability of $\forall x(Nx \wedge Px \to Px')$. It is *not* sufficient to have the proability both of $P0$ and of each statement $P0 \wedge \ldots \wedge Pn \to Pn'$.

Suppose that T is a consistent, recursively enumerable set of axioms sufficient to carry out a reasonable amount of arithmetic. And, let $A[x]$ be a formula which expresses (roughly) "x is not the code of a proof that T is consistent." Then it holds both that T proves $A[0]$, and also that T proves $A[0] \wedge \ldots \wedge A[n] \to A[n+1]$ (of course, T proves $A[n]$ outright for all n), but it is not the case that $T \vdash \forall x A$.

7b. I'll use the induction (8) to prove

$$\forall d \in \mathbb{N}\, \forall x \in \mathcal{B}(d = \text{depth}(x) \to |x| \leq 2^d). \tag{9}$$

Here we must use the property Pd defined by

$$d > 0 \to \forall x \in \mathcal{B}(d = \text{depth}(x) \to |x| \leq 2^d - 1). \tag{10}$$

The case $P0$ is vacuous and the case $P1$ is trivial. So suppose d satisfies (10); we need to show that $d+1$ satisfies this as well. To this end, pick $x \in \mathcal{B}$ such that $d+1 = \text{depth}(x)$ with $d > 0$. The children of x must have depths at most d. By (8), we may thus assume each child has at most $2^d - 1$ nodes. The sum of the numbers of nodes of the children is therefore at most $2^{d+1} - 2$, while the number of nodes of x is one more than that sum. Hence the number of nodes of x is at most $2^{d+1} - 1$, as desired.

7c. The sketched line of reasoning would prove at most

$$\forall d \in \mathbb{N}\, \exists x \in \mathcal{B}(d = \text{depth}(x) \wedge |x| \leq 2^d - 1)$$

which does not imply (9).