15-122 : Principles of Imperative Computation

Fall 2012

Assignment 3 – Selected Solutions

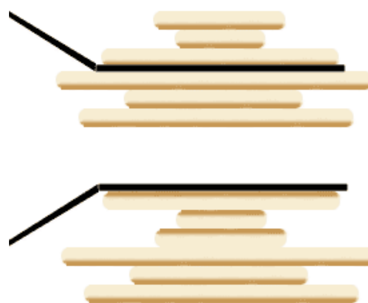
(Theory Part)

The following solutions are provided to you to help you study this semester. They are not to be distributed to others outside of the class nor are they intended to be used by students in future semesters as a substitute for completing one's own homework assignments.

1. **Pancake sort.** In lecture we talked about stacks, a data structure with a last-in, first-out behavior. This question deals with a much more delicious and *entirely unrelated* sort of stack: stacks of pancakes!

   Suppose that you have a stack of pancakes that you would like to sort from smallest to largest such that largest pancake is at the bottom of the stack after sorting. However, the only operation allowed on this data structure is flipping a portion of the stack over. To flip a portion of the stack, you choose a starting pancake and flip the entire portion of stack from that pancake up to the top, including the starting pancake.

   For example, here is what happens when the top 3 pancakes are flipped upside down. The first image shows the original pancake stack before flipping the top 3 pancakes. The second image shows the pancake stack after flipping the top 3 pancakes.



   Here is a possible algorithm for sorting a stack with n pancakes. You can assume that there is some abstract type pancake_stack that represents a stack of pancakes. We index the pancakes with pancake 0 on the top of the stack.

```
struct pancake {
    int size;
    string flavor;
```

```
};

void pancakesort(pancake_stack pancakes, int n)
{
    for (int i = n; i > 1; i--)
    {
        int max = findmax(pancakes, 0, i);
        flip(pancakes, max + 1);
        flip(pancakes, i);
    }
}
```

The abstract type `pancake_stack` implements the following interface:

```
int pancake_stack_size(pancake_stack pancakes)
//@ensures \result >= 0;
;

bool is_sorted(pancake_stack pancakes, int top, int bottom)
/*@requires 0 <= top && top <= bottom
    && bottom <= pancake_stack_size(pancakes);@*/
;

/* findmax returns the index of the largest size pancake that's
 * between indices top and bottom (not including bottom)
 */
int findmax(pancake_stack pancakes, int top, int bottom)
/*@requires 0 <= top && top < bottom
    && bottom <= pancake_stack_size(pancakes);*@/
//@ensures top <= \result && \result < bottom;
;

/* flip flips the substack containing the n top items over.
 * for example, flip(pancakes, 3) is in the picture above.
 */
void flip(pancake_stack pancakes, int n)
//@requires 0 <= n && n <= pancake_stack_size(pancakes);
;

/* le_seg returns true if and only if the size of p is less
 * than or equal to that of every pancake in the stack from
 * lower to upper (excluding upper)
 */
bool le_seg(struct pancake p, pancake_stack pancakes, int lower, int upper)
/*@requires 0 <= lower && lower <= upper
```

```
    && upper <= pancake_stack_size(pancakes);*@/
;

/* ge_seg returns true if and only if the size of p
 * greater than or equal to every pancake in the stack from
 * lower to upper (excluding upper)
 */
bool ge_seg(struct pancake p, pancake_stack pancakes, int lower, int upper)
/*@requires 0 <= lower && lower <= upper
    && upper <= pancake_stack_size(pancakes);@*/
;

/* le_segs returns true if and only if the size of everything in the
 * stack from lower1 to upper1 is less than or equal to the size of
 * everything in the stack from lower2 to upper2
 * (not including upper1 or upper2)
 */
bool le_segs(pancake_stack pancakes, int lower1, int upper1,
    int lower2, int upper2)
/*@requires 0 <= lower1 && lower1 <= upper1
    && upper1 <= pancake_stack_size(pancakes);@*/
/*@requires 0 <= lower2 && lower2 <= upper2
    && upper2 <= pancake_stack_size(pancakes);@*/
;

/* ge_segs returns true if and only if the size of everything in the
 * stack from lower1 to upper1 is greater than or equal to the size of
 * everything in the stack from lower2 to upper2
 * (not including upper1 or upper2)
 */
bool ge_segs(pancake_stack pancakes, int lower1, int upper1,
    int lower2, int upper2)
/*@requires 0 <= lower1 && lower1 <= upper1
    && upper1 <= pancake_stack_size(pancakes);@*/
/*@requires 0 <= lower2 && lower2 <= upper2
    && upper2 <= pancake_stack_size(pancakes);@*/
;
```

(1) (a) What is the tightest upper bound for the worst case asymptotic complexity of your algorithm using big O notation in terms of n where n is the number of pancakes? Explain your answer.

Assume that we have specialized state-of-the-art pancake-flipping hardware, so `flip(pancakes, max)` is a constant time operation ($O(1)$).
Also assume `findmax(pancakes, 0, i)` is a linear time operation ($O(i)$).

Solution:
$O(n^2)$

Explanation:

1. The operation findmax(pancakes,0,i) is a linear time operation, which is applied a total of n times (as the loop completes n iterations).
2. Alternatively, it could be noted that, in the worst case, the largest pancake is at the at the bottom. So, it takes $n - i$ steps to obtain the ith smallest pancake. Each pass of the loop requires two constant time operations ($flip(pancakes, i)$), so the total number of operations can be computed as a summation $\sum_{i=0}^{n} 2i = n(n+1)$.

(3) (b) Insert the missing preconditions, postconditions and loop invariants in the code. You can use only the functions declared above for your contracts.

Solution:

```
void pancakesort(pancake_stack pancakes, int n)

1. //@requires 0 <= n && n <= pancake_stack_size(pancakes);
2. //@ensures is_sorted(pancakes, 0 , n);

{
    for (int i = n; i > 1; i--)

    3. //@loop_invariant (n == 0 && i == 0) || 1 <= i && i <= n;
    4. //@loop_invariant n == 0 || is_sorted(pancakes, i, n);
    5. //@loop_invariant le_segs(pancakes, 0, i, i, n);

    {
        int max = findmax(pancakes, 0, i);
        flip(pancakes, max + 1);
        flip(pancakes, i);
    }
}

1. check lower/upper indices
2. check if sorted at the end
3. basic guard for loop counter
4. pancakes are sorted from bottom up
5. relationship between pancakes in sorted/unsorted positions
```

> Some students did not check for $n = 0$. Thinking in terms of boundary cases is important here.
>
> It's also important to think about loop invariants in terms of how easy they make it to prove partial correctness of the function – is it easy to see how the post-conditions follow from the pre-conditions by simply noting that the loop invariants still hold when the loop guard is negated?

(1) (c) Prove that the loop invariants hold before the first iteration of the loop.

Solution:

1. Basic loop index guard: Observe that $i = n$ initially, so loop invariant trivially holds
   `@loop_invariant (n == 0 && i == 0) || 1 <= i && i <= n`
2. Pancakes sorted from bottom up: $i = n$ initially, so we find that an array slice of length zero (the slice from $n-1$ to n) is trivially sorted or that an input array of length zero is trivially sorted.
   `n == 0 || is_sorted(pancakes, i, n)`
3. Initially, $i = n$, so the invariant `leg_segs(pancakes, 0, n, n, n)` trivially holds true (doesn't consider any pancakes in the range n to n)
   `leg_segs(pancakes, 0, i, i, n)`

(1) (d) Prove that the loop invariants are preserved (if the loop invariants are true before some iteration, they'll also be true after it).

Solution:

1. `@loop invariant (n == 0 && i == 0) || 1 <= i && i <= n`: The first case ($n = 0$) is not applicable, as we would not have entered the body of the loop at all. This follows from the loop guard `i > 1`, which implies that `i = 0` and therefore `n = 0` would not enter the loop body.

   `for (int i = n; i > 1; i--)`.
   The second case is trivially shown $1 < i \&\& i <= n$ initially (we're in a non-termination state on the last iteration), so in the next iteration $i' = i - 1$, so $1 <= i \&\& i <= n$ holds true.
2. `//@loop invariant n == 0 || is_sorted(pancakes, i, n);`. The first case ($n = 0$) is not applicable here because the loop guard `i > 1` combined with the first loop invariant `(n == 0 && i == 0) || 1 <= i && i <= n` implies that `n != 0`

`for (int i = n; i > 1; i--)`. The second case can be argued as follows.

(a) From the second loop invariant and the loop guard, we know that since `n != 0`, we can say that `is_sorted(pancakes, i, n)`

(b) We want to show that at the completion of this iteration we will have `is_sorted(pancakes, i', n)` where $i' = i - 1$.

(c) By the third loop invariant, we know that `le_segs(pancakes, 0, i, i, n)`, meaning we can select any element from the range $[0, i)$ and prepend it to the sequence $[i, n)$ to and the resulting sequence will still be sorted. (It's important to specifically pick the maximal element only to maintain the third loop invariant, but we'll handle that separately).

(d) In the loop body, we first select an element from the region $[0, i)$ with `findmax(pancakes, 0, i)` (remember, it's not important exactly which we select for this LI). Then, the two flips in the loop body `flip(pancakes, max+1)` and `flip(pancakes, i)` place the element first on top and then back into the region $[0, i)$.

(e) At the conclusion of the iteration, the loop counter i will be decremented such that $i' = i - 1$.

(f) Together, these previous three observations imply that `is_sorted(pancakes, i-1, n)` holds true since it will now include element $i - 1$, which we already know satisfied `le_segs(pancakes, 0, i, i, n)`.

3. `//@loop invariant le_segs(pancakes, 0, i, i, n)`. We can argue this case as follows:

(a) We want to show that at the completion of this iteration we will have `le_segs(pancakes, 0, i', i', n)`. where $i' = i - 1$

(b) In the loop body, `int max = findmax(pancakes, 0, i)` selects the largest pancake from the array region $[0, i)$.

(c) We then flip the `max` pancake to the top via `flip(pancakes, max+1)`

(d) We then flip the `max` pancake into the ith position via `flip(pancakes, i)`

(e) By the definition of the maximum, we know that `max` is greater than all other elements in the region $[0, i]$, so we can conclude that `le_segs(pancakes, 0, i-1, i-1, n)` holds.

(1) (e) Prove that the loop invariants combined with the negated loop guard imply the postcondition. *Hint: This part should be straightforward if your loop invariants are*

strong enough.

Solution:
When the loop terminates, we know that the loop guard $i > 1$ must no longer hold true so that its negation is true $i \leq 1$. There are two cases. First, the function was passed $n = 0$, in which case we received an empty stack of pancakes, which is trivially sorted and satisfies `is_sorted(pancakes, 0, n)`.

Otherwise, we received a non-empty stack of pancakes, in which case the second loop invariant guarantees that `is_sorted(pancakes, i, n)`. We also know that $i = 1$ upon loop termination in this case, since we know by the negated loop guard that $i \leq 1$ and the first loop invariant gives us that $i \geq 1$. Together, these satisfy `is_sorted(pancakes, 1, n)`. Finally, following the logic of preservation of the second loop invariant, since pancake 0 is guaranteed to be less than the pancakes in the range $[1, n)$, it follows by the definition of `is_sorted` that `is_sorted(0, n)`.

(1) (f) Prove that the loop terminates.

Solution: Since the loop counter i (initialized to n) decrements by 1 at the conclusion of *every* iteration and is never increased, we will eventually reach $i = 1$, at which point the loop will terminate as specified by the loop guard.