CSE 331 Winter 2023 Homework 2

Directions:

- Due Tuesday January 17, by **11 pm**.
- Turn in your work online using Gradescope. You should turn in a single pdf file. You can have more than one answer per page, but if you can, please try to avoid page breaks in the middle of a question. Insert page breaks between questions as needed. Scanned copies of hand-written documents are fine if they are legible.
- Feel free to rewrite the problems and solutions on a separate sheet – you do not have to turn in these specific pages with the blanks filled in.
- You may use any standard symbols for "and" and "or" (& and |, $\wedge$ and $\vee$, etc.)
- If no precondition is needed for a code sequence, write {true} to denote the trivial precondition.
- As in Homework 1, assume that all numeric values are integers and that integer overflow will never occur. Also, assume that division is integer division like in Java (truncating towards 0).

And be sure:

- When proving code correct, you should write down all the intermediate steps unless directed otherwise. The slides from class often omit steps because we need the "key idea" to fit on a slide in a large-enough font. The reading notes may be a better guide in how detailed to be, even for Problem 1, which refers to an algorithm from class.
- You do not need to prove that your loops or algorithms terminate.

1. (Warmup) In class, we developed an algorithm to find the largest value in a non-empty list of integers `items[0..size-1]`. In our code, the loop had the following invariant:

   `max = largest value in items[0..k-1]` (or `max = max(items[0..k-1])`

   Suppose we had used the following slightly different invariant instead:

   `max = largest in items[0..k]`

   Rework the code and proof in the example to use this new invariant instead of the original one and show that the modified code is correct. Insert or modify assertions and code as needed.

   After solving this problem, write a brief description of how this change to the invariant affected the algorithm and the associated proof. What were the major changes? Did this change make the code easier or harder to write or prove compared to the original version? Why? (You should be able to keep your answers brief and to the point.)

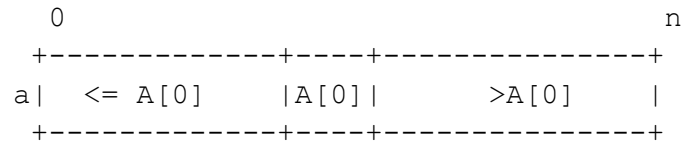
2. Given two non-negative integers x and y, we can calculate the exponential function x^y (x raised to the power y) using repeated multiplications by x a total of y times. The following code is alleged to compute x^y much more quickly (it supposedly takes time proportional to log y).

```
int expt(int x, int y) {
  int z = 1;
  while (y > 0) {
    if (y % 2 == 0) {
      y = y/2;
      x = x*x;
    } else {
      z = z*x;
      y = y-1;
    }
  }
  return z;
}
```

Give a proof that this code is correct. A suitable precondition for the function would be $x=x_{pre}$ && $y=y_{pre}$ && $x_{pre}>=0$ && $y_{pre}>=0$, and an appropriate postcondition would be that the returned value $z=x_{pre}^{ypre}$. (The superscript (exponent) in the postcondition should be y_{pre}, but the word processor won't cooperate and allow a subscripted superscript.) We define 0^0 to be 1 for this problem. You will need to develop (discover!) a loop invariant and pre- and post-conditions for the statements inside the loop, and verify that the code has the necessary properties to ensure that these assertions hold. You are not required to give pre- and post-conditions for each individual assignment statement inside the `if` if these are not needed to clearly show the code is correct, but you should include whatever is needed so that the grader can follow your proof and see that it is valid. You do not need to prove that the algorithm runs in log y time.

Hint: to try to figure out a suitable invariant, you might try tracking the code on some simple examples like `expt(3,3)` and `expt(2,9)`.

3. We are given an integer array `a` containing `n` elements and would like to develop an algorithm to rearrange the array elements as follows and prove that the algorithm is correct. If the original elements of the array are `A[0], A[1], ..., A[n-1]`, we would like to rearrange the array so that all of the elements less than or equal to the original value in `A[0]` are at the beginning of the array, followed by the value originally stored in `A[0]`, followed by all the elements in the original array that are larger than `A[0]`. In pictures, the postcondition we want is the following:



The operation `swap(a[i],a[j])` can be used to interchange any pair of elements in the array, and this is the only operation that can be used to modify the contents of the array. (As a result of this restriction, the array will be a permutation of its original contents, so you do not need to prove this.) Your code should run in linear ($O(n)$) time.

You should develop a suitable loop invariant, then write code to partition the array using that invariant and prove that when the code terminates the postcondition has been established. You do not need to provide assertions for every trivial statement in the code, but there should be sufficient annotations so that the correctness of your code is obvious. You do not need to write a complete method, just the necessary loop and supporting statements and assertions, including any necessary initialization and final statements before and after the loop.

Hints: As you are partitioning the values in the array, you might find it simpler if you are not constantly moving the original value from `A[0]` into new locations. Try different invariants and see which one makes things simplest. Also remember that you can include additional statements before and after the main loop if needed to establish the loop invariant or the postcondition.

Note that the values in the array partitions do not need to be sorted or stored in any particular order as long as all the values `<=A[0]` (if any) appear before the original `A[0]` value in the result and all following values (if any) are `>A[0]`.

4. Give an implementation of the algorithm *selection sort* and a proof of its correctness. The algorithm should sort an array `a` containing `n` integer values. The precondition is that the array `a` contains `n` integer values in some unknown order. The postcondition is that the array holds a permutation of the original values such that `a[0] <= a[1] <= ... <= a[n-1]`. As in the previous problem, you should use the operation `swap(a[i],a[j])` to interchange array elements when needed.

   Selection sort proceeds as follows. First we find the smallest element in the original array and swap it with the original value in `a[0]`. (If `a[0]` is the smallest element in the original array it is fine to swap it with itself to avoid having additional special cases in the code.) Next we find the smallest element in the remaining part of the array beginning at `a[1]` and swap it with `a[1]`. Then we find the smallest element in the remaining part of the array starting at `a[2]` and move it to the front of that part of the array. This search-and-swap operation continues on the successively smaller remaining parts of the array until all elements are sorted.

   As with the previous problem, you should develop suitable invariants for the nested loops needed to perform the sort, then write the code and annotate it with appropriate assertions so that it is clear that your code is correct and that when it terminates the array is sorted.

   Do not introduce additional methods or functions to solve parts of the problem. The program and proof are simpler if you write a straightforward pair of nested loops, with an appropriate invariant for each loop. Also do not introduce any additional arrays or collection data structures to hold the array elements. Rearrange the original array in place by swapping elements in it.