CSE 331 Autumn 2017 Homework 2

1. Fill in the proof of correctness for the following code.

   In addition to filling in each of the blanks below, you need to provide additional argument in the following cases:

   - If your last assertion before the loop is not *identical* to the loop invariant, explain why your last assertion implies the loop invariant.
   - If your first assertion in the body of the loop is not *identical* to the loop invariant, explain why the loop invariant implies your first assertion.
   - If your last assertion in the body of the loop is not *identical* to the loop invariant, explain why your last assertion implies the loop invariant.

```
{{ P: 1 <= n <= A.length and every A[i] >= 1 }}
int v = 0;
{{ ____________________________________ }}
int i = 0;
{{ ____________________________________ }}


{{ Inv: P and v = A[0]! + A[1]! + … + A[i-1]!  }}
while (i != n) {
  {{ ____________________________________ }}
  int w = A[i];
  {{ ____________________________________ }}
  int s = w;
  {{ ____________________________________ }}

  {{ Inv2: Inv and s = A[i] * (A[i]-1) * ... * w }}
  while (w != 1) {
       {{ ____________________________________ }}
       w = w - 1;
       {{ ____________________________________ }}
       s = s * w;
       {{ ____________________________________ }}
  }
  {{ ____________________________________ }}
  i = i + 1;
  {{ ____________________________________ }}
  v = v + s;
  {{ ____________________________________ }}
}

{{ v = A[0]! + A[1]! + … + A[n-1]! }}
```

2. Fill in the proof of correctness for the following method, `merge`. It takes as input two sorted arrays `A` and `B`, of length at least `n`, and an empty array `C`, of the length at least `2n.`

   In addition to filling in each of the blanks below, you need to provide additional argument in the following cases:

   - If your last assertion before the loop is not *identical* to the loop invariant, explain why your last assertion implies the loop invariant.
   - If your first assertion in the body of the loop is not *identical* to the loop invariant, explain why the loop invariant implies your first assertion.
   - If your last assertion in the body of the loop is not *identical* to the loop invariant, explain why your last assertion implies the loop invariant.

   The precondition (P) below includes the assertions that A and B both contain n+1 elements, where the last element in each sorted array is the max integer (`Integer.MAX_VALUE`), a value that is larger than any number in either A or B. This means that, inside the loop body, we will always have i, j <= n and neither A[i] nor B[j] will cause an array out-of-bounds exception at run time. You can **assume** those facts in your proof of correctness (i.e., you do not need to prove them yourself).

   It may help to rewrite the code below on a separate sheet of paper because the assertions that go in the blanks could get fairly long.

```
{{ P: A & B are sorted and 2n <= C.length and n+1 = A.length = B.length and
     A[n] = B[n] = max integer and A[0], ..., A[n-1], B[0], ..., B[n-1] < max integer }}
void merge(int[] A, int[] B, int[] C, int n) {
  int i = 0;
  int j = 0;
  int k = 0;
  {{ ________________________________ }}


  {{ Inv: P and C[0], …, C[k-1] = sorted(A[0], …, A[i-1], B[0], …, B[j-1]) and C[k-1] <= A[i], B[j] }}
  while (k != 2*n) {


    {{ ________________________________________________________________ }}
    if (A[i] < B[j]) {
      {{ ______________________________________________________________ }}
      C[k] = A[i];
      {{ ______________________________________________________________ }}
      i = i + 1;
      {{ ______________________________________________________________ }}
    } else {
      {{ ______________________________________________________________ }}
      C[k] = B[j];
      {{ ______________________________________________________________ }}
      j = j + 1;
      {{ ______________________________________________________________ }}
    }
    {{ ________________________________________________________________ }}
    k = k + 1;
    {{ ________________________________________________________________ }}


  }

  {{ C[0], …, C[2n-1] = sorted(A[0], …, A[n-1], B[0], …, B[n-1]) }}
}
```

3. Below are two different snippets of code. They both take in two arrays A and B and attempt to make each A[i] the larger of A[i] and B[i].

   Each of these snippets is **incorrect**. For each one, indicate whether the proof of correctness fails because (1) the invariant does not hold initially, (2) the invariant does not hold after the loop body is executed, or (3) the invariant does not imply the post condition upon termination of the loop. (No explanation is necessary.)

   The variable n holds the length of A and B. You may assume that n is at least 1.

-
  ```
  int i = n - 1;
  ```

  {{ Inv: A[i] = max(A[i], B[i]), ..., A[n-1] = max(A[n-1], B[n-1]) }}

  ```
  while (i != 0) {
    i = i - 1;
    if (A[i] < B[i])
      A[i] = B[i];
  }
  ```

  {{ A[0] = max(A[0], B[0]), ..., A[n-1] = max(A[n-1], B[n-1]) }}

-
  ```
  int i = n - 1;
  ```

  {{ Inv: A[i+1] = max(A[i+1], B[i+1]), ..., A[n-1] = max(A[n-1], B[n-1]) }}

  ```
  while (i != 0) {
    if (A[i] < B[i])
      A[i] = B[i];
    i = i - 1;
  }
  ```

  {{ A[0] = max(A[0], B[0]), ..., A[n-1] = max(A[n-1], B[n-1]) }}

4. Fill in an implementation of the method `findPairSum` below. It takes as input an array A, of length at least n, and a number x. It's goal is to find a pair of indices (i,j), with 0 <= i, j < n, such that A[i] + A[j] equals x. If such a pair exists, it should return an array [i, j]. Otherwise, it should return null.

   The invariant for the loop is already provided. ***Do not add any additional loops.***

   You **do not need** to *turn in* a complete proof of correctness, but you should complete one since your code will be graded on correctness.

   Hints:

   - It is **only necessary** to consider pairs (i,j) with i <= j since the two pairs (i,j) and (j,i) give the same sum. Your method should be optimized to only consider such pairs.
   - Think about how to change i & j each time through the loop so that the new Inv only adds a claim about *one* more pair. You should only need to check one pair per iteration.

{{ P: 0 <= n <= A.length }}

```
int[] findPairSum(int[] A, int n, int x) {
  int i, j;
```

{{ Inv: P and there is no (i',j') with 0 <= i' < i or (i' = i and j < j' < n) such that A[i'] + A[j'] = x }}

```
  while (______________________) {

  }
```

{{ there is no (i,j), with 0 <= i, j < n, such that A[i] + A[j] = x }}

```
  return null;
}
```

5. Fill in another implementation for `findPairSum`. In this version, the method assumes that the array A is **sorted**. As you will see, this allows a much faster solution.

   The invariant for the loop is already provided. ***Do not add any additional loops.***

   You **do not need** to *turn in* a complete proof of correctness, but you should complete one since your code will be graded on correctness.

   Hint: If A[i] + A[j] < x, then the same is true with i replaced by any i' < i because A is sorted. (A similar fact applies when A[i] + A[j] > x.) For this reason, unlike in problem 4, in a single iteration, you can eliminate many pairs as potential solutions by checking only one pair.

```
{{ P: 0 <= n <= A.length and A is sorted }}
int[] findPairSum(int[] A, int n, int x) {
  int i, j;



  {{ Inv: P and there is no (i',j') with 0 <= i' < i or j < j' < n such that A[i'] + A[j'] = x }}
  while (______________________) {










  }

  {{ there is no (i,j), with 0 <= i, j < n, such that A[i] + A[j] = x }}
  return null;
}
```

6. Fill in an implementation of the method `divRemainder` (on the next page). It takes as input two arrays, `A`, and `B`, each of length at least `n`, and a number `x`. Your method should store in each B[i] the result of A[i] divided by `x`. It should return the number of ***non-zero*** remainders of these divisions, m, with the those remainders stored in A[0], ..., A[m-1].

   For example, if n = 3, A contains [1, 2, 3], and x = 2, then upon completion, B would contain [0, 1, 1], the method would return 2 (not 3, since A[1] % 2 = 0), and A would **begin with** the values [1, 1]. (Note that the postcondition has **no claims** about what is stored in A[2], here. It could be anything.)

   You **do not need** to *turn in* a complete proof of correctness, but you should complete one since your code will be graded on correctness.

   You **do need** to document any loop invariants. Keep in mind that your goal is to communicate to another human being, so feel free to use English prose where a mathematical description would be awkward.

{{ 0 <= n <= A.length, B.length and 0 < x }}

```
int divRemainder(int[] A, int[] B, int n, int x) {

}
```