CSE 331 Summer 2017 Homework 2

Notes:

- You may use any standard symbols for logical operators (e.g., V for "or").
- As in HW1, you may assume that all numbers are integers and integer overflow will never occur.

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.

```
{{ 1 < n <= A.length }}
int v = A[n-1];
{{ _____ }}
int i = n-1;
{{ _____ }}

{{ Inv: v = A[i] * A[i+1] * … * A[n-1] }}
while (i != 0) {
  {{ _____ }}
  i = i - 1;
  {{ _____ }}
  int w = A[i] * v;
  {{ _____}}
  v = w;
  {{ _____ }}
}

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

2. Fill in the proof of correctness for the following method, `insertionSort`. It takes as input an array `A` of length at least `n`.

   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.

   Note: In the inner loop invariant, the expression "A[0], …, A[j-1], A[j+1], …, A[i]" means all of the array A[0], A[1], …, A[i] **except** for the element A[j], which is left out. The invariant says that, if you ignore the number in A[j], the rest of the subarray A[0], A[1], …, A[i] is sorted.

```
{{ 0 < n <= A.length }}
void insertionSort(int[] A, int n) {
  int i = 1;
  {{ _____ }}

  {{ Inv: A[0], A[1], …, A[i-1] is sorted }}
  while (i < n) {
    {{ _____ }}
    int j = i;
    {{ _____ }}

    {{ Inv: A[0], …, A[j-1], A[j+1], …, A[i] is sorted and A[j] < A[j+1], …, A[i] }}
    while (j > 0 and A[j-1] > A[j]) {
      {{ _____ }}
      swap A[j-1], A[j];
      {{ _____ }}
      j = j - 1;
      {{ _____ }}
    }

    {{ _____ }}
    i = i + 1;
    {{ _____ }}
  }

  {{ A[0], A[1], …, A[n-1] is sorted }}
}
```

3. Fill in the proof of correctness for the following method, `reverse`. It takes as input an array `A` of length at least `n`.

   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.

   Below, the expression "A[i]" refers to the *current value* in A[i], whereas $A[i]_1$ (subscript "1") refers to the *original value* in A[i].

```
{{ 0 < n <= A.length }}
void reverse(int[] A, int n) {
  i = -1;
  j = n;
  {{ _____ }}

  {{ Inv: A[0] = A[n-1]₁, …, A[i] = A[n-1-i]₁, A[j] = A[n-1-j]₁, …, A[n-1] = A[0]₁, j = n-1-i }}
  while (i < j) {
    {{ _____ }}
    i = i + 1;
    {{ _____ }}
    j = j - 1;
    {{ _____ }}
    swap A[i], A[j];
    {{ _____ }}
  }

  {{ Inv: A[0] = A[n-1]₁, …, A[n-1] = A[0]₁ }}
}
```

4. Fill in the missing code for the following method, `split`. It takes as input an array `A`, an index `i`, a length `n`, and an integer `x`. The goal is to rearrange the numbers in the part of the array from index `i` to `i+n-1` (i.e., `A[i]`, `A[i+1]`, …, `A[i+n-1]`), so that all the numbers less than or equal to `x` come before all the numbers greater than `x`, and then return the index of the *last number* less than or equal to `x`.

   The loop invariant is already provided. It refers to variables `L` and `R` that you will need to create in your code. In English, the invariant says that all of the numbers in the array from index `i` to `L-1` (i.e., `A[i]`, `A[i+1]`, …, `A[L-1]`) are less than or equal to x and those from index `R` to `n-1` (i.e., `A[R]`, `A[R+1]`, …, `A[n-1]`) are greater than `x`.

   Your code should only modify the array by swapping elements. (This restriction implies that your code will only rearrange the numbers in the array, not change them, so we do not need to prove that the numbers in the array at the end are the same as those at the beginning.) You may use the pseudocode "`swap A[i], A[j]`" to perform a swap.

   You **do not** need to *turn in* a proof of correctness, but you should complete one (using the techniques of problems 1-2) since your code will be graded on correctness.

```
{{ n >= 0 and i >= 0 and i + n <= A.length }}
int split(int[] A, int i, int n, int x) {




  {{ Inv: A[i], …, A[L-1] <= x < A[R], …, A[i+n-1] }}
  while (_____) {









  }

  {{ A[i], …, A[L-1] <= x < A[L], …, A[i+n-1] }}
  return L-1;
}
```

5. Fill in an implementation of the following method, `copyParts`. It takes as input three arrays, `A`, `B`, and `C`, each of length at least `n`, and a number `x`. Your method should copy those elements of `A` that are less than or equal to `x` into `B` and those that are greater than `x` into `C`. The return value should be the number copied into `B`. (The number copied into `C` is then equal to `n` minus the number copied into `B`.)

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

   Important: For any loops in your code, you must provide the loop invariant. When writing the invariant, your goal is to communicate to another human being, so you can use whatever notation you want provided it is clear and precise.

   As in the examples from lecture, you can leave out some parts of the invariant that would be awkward to describe mathematically, e.g., that the arrays `B` and `C` contain only numbers that came from `A`. While that would be necessary for a complete a formal proof of correctness, your goal is to argue correctness to a human being. In this case, it should be easy to see from your code that the only numbers copied into `B` and `C` come from `A`.

```
{{ A.length >= n and B.length >= n and C.length >= n }}
int copyParts(int[] A, int[] B, int[] C, int n, int x) {




}
```

6. Fill in the missing code in the loop body for the following method, `compress`. It takes as input an array `A` of length at least `n`. The goal is to remove all the negative numbers from A, leaving just the non-negative numbers, packed together at the beginning of the array, and to return the number of non-negative elements found.

   Below, the expression "A[i]" refers to the *current value* in A[i], whereas $A[i]_1$ (subscript "1") refers to the *original value* in A[i].

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

```
{{ A.length >= n }}
int compress(int[] A, int n) {




  {{ Inv: A[0], …, A[j-1] stores the non-negatives from A[0]₁, …, A[i-1]₁ }}
  while (_________________________) {









  }

  {{ A[0], …, A[j-1] stores the non-negatives from A[0]₁, …, A[n-1]₁ }}
  return j;
}
```