CSE 331 Spring 2021 HW2

1. Fill in the proof of correctness for the method `strToInt` on the next page. It returns the `int` value corresponding to the decimal number written in the first `n` characters of the array `s`.

   That code references a function `charToInt` that takes a character in the range `‘0’`, `‘1’`, …, `‘9’` to the corresponding `int` in the range `0`, `1`…, `9`. It has the following implementation:

   ```
   // Precondition: ‘0’ <= ch <= ‘9’
   int charToInt(char ch) {
     return ch – ‘0’;
   }
   ```

   Reason in the direction (forward or backward) indicated by the arrows on each line: forward outside the loop and backward inside the loop.

   In addition to filling in each blank below, you must provide additional explanation whenever two assertions appear right next to each other, with no code in between: in those cases, explain why the top statement *implies* the bottom one. You can skip this explanation if the two statements are identical or if the bottom one simply drops facts included the top one. (Each triple that requires additional explanation is marked with a “?“ on the left.)

   Notes on the notation used:

   - A summation over a range like “s[a] + … + s[b]” should be interpreted as 0 if there are no indexes between the lower bound, a, and the upper bound, b, (i.e., if b < a).

   - The assertions make reference to a *mathematical* function “int” that takes a character in the range ‘0’, ‘1’, …, ‘9’ to the corresponding integer value in the range 0, 1, .., 9. (The Java function `charToInt` mentioned above implements this function.)

```
{{ Precondition: 0 < n <= s.length() }}
int strToInt(char[] s, int n) {
↓  int i = 0;
   {{ ____________________________________________ }}
↓  int val = 0;
   {{ ____________________________________________ }}

?

   {{ Inv: val = 10^(i-1) * int(s[0]) + … + 10 * int(s[i-2]) + int(s[i-1]) }}
   while (i != n) {

?

     {{ __________________________________________ }}
↑    int d = charToInt(s[i]);     // in our notation, now d = int(s[i])
     {{ __________________________________________ }}
↑    val = 10 * val + d;
     {{ __________________________________________ }}
↑    i = i + 1;
     {{ __________________________________________ }}
↑  }

↓
   {{ ____________________________________________ }}

?

   {{ Postcondition: val = 10^(n-1) * int(s[0]) + … + 10 * int(s[n-2]) + int(s[n-1]) }}
   return val;
}
```

2. Fill in the missing parts of each implementation of the method `isPalindrome` below. It takes as input a string `s` of length `n`, and it returns true if and only if s is the same read forward and backward.

   In each case, the precondition and postcondition are the same, so each loop is trying to accomplish the same task, but the provided code or the invariant is slightly changed, so the details of how the code will accomplish it should be different.

   In each case, your code must be correct according to the loop invariant that is **provided**. You do not need to turn in your proof of correctness. However, since your code will be graded on its correctness, you should still complete this for yourself.

   Additionally, **you may not** (1) add any additional loops or (2) call any other methods other than `String.charAt`.

   Notes on the notation used:

   - It is okay to exit (by returning) from the middle of the loop.
   - To formally argue correctness, we need a formal definition of what the code is supposed to do. The function rev referenced in the assertions below does that. Informally, rev(s) means the reversal of the string s.
   - Formally, we can define rev recursively (as in CSE 311) as follows:

     rev(“”) = “”

     rev(wa) = a · rev(w) (reversal of wa is a followed by reversal of w)

     (Here, “w” represents any string and “a” any single character. Each string can either be written in the form wa for some w and some a or it is the empty string “”.)
   - The notation “s[i .. j]” means the substring from index i up to and including index j. For example, if s = “abcde”, then s[2 .. 3] = “cd”. If j = i – 1, then this indicates an *empty string*. In symbols, we have s[i .. i-1] = “”.

a) {{ Precondition: s != null and s.length() = n >= 0 }}

```
boolean isPalindrome(String s, int n) {
  int i = ______;

  {{ Inv: s != null and s.length = n and s[0 .. i-1] = rev(s[n-i .. n-1]) }}
  while (______________________) {



    i = i + 1;
  }

  {{ Postcondition: s = rev(s) }}
  return true;
}
```

b) {{ Precondition: s != null and s.length() = n >= 0 }}

```
boolean isPalindrome(String s, int n) {
  int i = ______;

  {{ Inv: s != null and s.length = n and s[0 .. i] = rev(s[n-i-1 .. n-1]) }}
  while (______________________) {



    i = i + 1;
  }

  {{ Postcondition: s = rev(s) }}
  return true;
}
```

c)
```
{{ Precondition: s != null and s.length() = n >= 0 }}
boolean isPalindrome(String s, int n) {
  int i = ______;

  {{ Inv: s != null and s.length = n and s[0 .. i-1] = rev(s[n-i .. n-1]) }}
  while (______________________) {
    i = i + 1;



  }

  {{ Postcondition: s = rev(s) }}
  return true;
}
```

d)
```
{{ Precondition: s != null and s.length() = n >= 0 }}
boolean isPalindrome(String s, int n) {
  int i = ______;

  {{ Inv: s != null and s.length = n and s[0 .. i] = rev(s[n-i-1 .. n-1]) }}
  while (______________________) {
    i = i + 1;



  }

  {{ Postcondition: s = rev(s) }}
  return true;
}
```

3. Fill in the proof of correctness for the method `reverseWords` on the next page. It takes as input an array `s`, of length at least `n`, containing words separated by spaces. When the method completes, it will have reversed each of those words *in place*. For example, if s contains “ab cde gh”, then the result would be “ba edc hg” since the reversal of “ab” is “ba”, the reversal of “cde” is “edc”, and the reversal of “gh” is “hg”. (Note that the order of the individual words in the sentence are not changed.)

   The code references a function `reverse` that takes an array of characters `s` and two indices `i` and `j` and reverses the characters in the sub-array s[i .. j], while leaving the rest of the array unchanged (as usual, the subscript “1” indicates the values stored in the array *initially*):

   ```
   // Precondition: s != null and 0 <= i <= j < n <= s.length()
   // Postcondition: s = s[0 .. i-1]₁ + rev(s[i .. j]₁) + s[j+1 .. n-1]₁
   void reverse(char[] s, int i, int j, int n)
   ```

   As in problem 1, you should reason in the direction (forward or backward) indicated by the arrows on each line, and you must provide additional explanation whenever two assertions appear right next to each other, with no code in between. Some assertions have been **provided for you**. When such an assertion is adjacent to your own, you should explain why they form a valid Hoare triple. (The areas that require additional explanation are marked ?.)

   Notes on the notation used:

   - The expression w_1 + “ “ + … + “ “ + w_j + “ “ is a string containing those j words each <u>followed by</u> a space. If j = 0, this is “”. If j = 1, it could be “ab “, for example. If j = 2, it could be “ab cde “, for example. *This string will have exactly j spaces.*
   - The expression w_1 + “ “ + w_2 + … + “ “ + w_j is a string containing those j words <u>separated</u> by spaces. If j = 0, this is also “”. If j = 1, it could be “ab”, for example, and if j = 2, it could be “ab cde”. *If j > 0, this string will have j-1 spaces, and it will have no spaces also in the case j = 0.*

{{ Precondition: $0 \le n \le$ s.length and s[0 .. n-1] = w_1 + " " + w_2 + .. + " " + w_k
    for some non-empty words w_1, w_2, .., w_k not containing any spaces }}

```
void reverseWords(char[] s, int n) {
↓  int j = 0;
   {{ ____________________________________________________ }}
↓  int i = 0;
   {{ ____________________________________________________ }}

?

   {{ Inv: Precondition and s[0 .. i-1] = rev(w_1) + " " + rev(w_2) + … + " " + rev(w_j) and 0 ≤ i ≤ n }}
   while (i != n) {
↓
     {{ ____________________________________________________ }}
↓    if (j > 0) {
       {{ ____________________________________________________ }}

?

       {{ ____________________________________________________ }}
↑      i = i + 1;
       {{ ____________________________________________________ }}
     } else {
       {{ ____________________________________________________ }}

?

       {{ ____________________________________________________ }}
↑    }

     {{ Precondition and s[0 .. i-1] = rev(w_1) + " " + … + rev(w_j) + " " and 0 ≤ i ≤ n }}
```

(code continues on the next page…)

```
↓      int t = i;
       {{ ______________________________________________________ }}

?

       {{ Inv: Precondition and s[0 .. i-1] = rev(w_1) + " " + … + rev(w_j) + " " and 0 ≤ i ≤ t ≤ n and
               s[i], …, s[t-1] != " " }}
       while (t != n && s[t] != ' ') {
↓
         {{ ______________________________________________________ }}

?

         {{ ______________________________________________________
            ______________________________________________________ }}
↑        t = t + 1;
         {{ ______________________________________________________
            ______________________________________________________ }}
       }
↓
       {{ ______________________________________________________
         ______________________________________________________ }}

?

       {{ Precondition and s[0 .. i-1] = rev(w_1) + " " + … + rev(w_j) + " " and 0 ≤ i ≤ t ≤ n and
          s[i .. t-1] = w_{j+1} }}
       reverse(s, i, t-1, n);

?

       {{ ______________________________________________________ }}
↑      i = t;
       {{ ______________________________________________________ }}
↑      j = j + 1;
       {{ ______________________________________________________ }}
↑    }
```

```
  // after the outer loop…
↓
  {{ ____________________________________________________________ }}
```

Note: Feel free to use the fact the fact that s[0 .. i-1] contains the same number of words as it did initially. That fact was left out of the invariant to simplify the problem, but you should be able to see that it is true.

```
?



  {{ Postcondition: s[0 .. n-1] = rev(w1) + " " + rev(w2) + … + " " + rev(wk) }}
}
```

Bonus Questions (0 points)

a. What is the running time of this method as a function of n?

b. This method, which reverses the individual words in a string, operates “in place”, meaning that it does not use any additional arrays of data structures.

   Suppose you were asked to reverse the *order of the words* in the array but to leave the individual words as they are. How could you do that in place and in linear time?

4. Fill in the missing parts of the method `removeDups` on the next page. It takes as input two arrays of integers, `A` and `B`, both of length at least `n`. `A` is also promised to be sorted.

   The goal of the method is to copy the first `n` elements of `A` *except* those that are duplicates into the array B. For example, if the input is A = [1, 1, 2, 3, 3] and n = 5, then the method will copy [1, 2, 3] (the elements of A with duplicates removed) into B and return 3 as its length.

   In this case, the loop invariant is provided. Your code must be correct **with this invariant**.

   You do not need to turn in your proof of correctness. However, since your code will be graded on its correctness, you should still complete this for yourself.

   Additionally, **you may not** (1) add any additional loops or (2) call any other methods.

   Notes on the notation used:

   - The notation “set(A)” means the mathematical set containing the elements in A.

```
{{ Precondition: 0 < n <= A.length, B.length and A[0] <= A[1] <= … <= A[n-1] }}
int removeDups(int[] A, int[] B, int n) {




  {{ Inv: Precondition and B[0] < B[1] < … < B[k-1] and set(B[0 .. k-1]) = set(A[0 .. i-1]) }}
  while (______________________) {




  }




  {{ B[0] < B[1] < … < B[k-1] and set(B[0 .. k-1]) = set(A[0 .. n-1]) }}
  return k;
}
```

5. Fill in the missing parts of the implementation of the method `intersectionSize` on the next page. It takes as input arrays `A` and `B` of lengths at least `n` and `m`, respectively, both of which are promised to be sorted, and returns the number of distinct integers found in both lists.

   Your code must be correct according to the loop invariant that is **provided**. You do not need to turn in your proof of correctness. However, since your code will be graded on its correctness, you should still complete this for yourself.

   Additionally, **you may not** (1) add any additional loops or (2) call any other methods.

   Notes on the notation used:

   - In the invariant, assume that A[-1] = B[-1] = -infinity and A[n] = B[m] = +infinity. This allows us to skip having to write the exceptional cases like "j = n" separately. (Of course, that the code must handle the exceptional cases correctly!)
   - If X and Y are lists, the expression |set(X) $\cap$ set(Y)| means the size of the intersection of two sets containing the numbers in those two lists.

     This is just the number of *distinct* numbers in common between the two lists. E.g., if X = [1, 2, 2, 3] and Y = [2, 2, 3], then the value of the expression would be 2 since they have the numbers 2 and 3 in common. (It does not matter that both lists have two copies of the number 2. It still only counts as one more number in common.)

   - You can interpret max of an empty list of numbers to be minus infinity (so that it is smaller than any finite number).

```
{{ Precondition: A and B are sorted arrays of length at least n > 0 and m > 0, respectively  }}
int intersectionSize(int[] A, int n, int[] B, int m) {
  int s = 0;
  int j = ______;
  int k = ______;

  {{ Inv: Precondition and s = |set(A[0 .. j-1]) ∩ set(B[0 .. k-1])| and
        max(A[0], …, A[j-1], B[0], …, B[k-1]) <= min(A[j], B[k]) }}
  while (j != n && k != m) {
    if (j-1 >= 0 && A[j] == A[j-1]) {



    } else if (k-1 >= 0 && B[k] == B[k-1]) {



    } else if (j-1 >= 0 && B[k] == A[j-1]) {



    } else if (k-1 >= 0 && A[j] == B[k-1]) {



    } else if (A[j] < B[k]) {



    } else if (A[j] > B[k]) {



    } else {



    }
  }
```

(code continues on the next page…)

```
   // after the outer loop…
↓
  {{ Precondition and s = |set(A[0 .. j-1]) ∩ set(B[0 .. k-1])| and
    max(A[0], …, A[j-1], B[0], …, B[k-1]) <= min(A[j], B[k]) and (j = n or k = m) }}





  {{ Postcondition: s = |set(A[0 .. n-1]) ∩ set(B[0 .. m-1])| }}
  return s;
}
```

Bonus Questions (0 points)

a. Two of the 7 cases in the body of the loop above are actually impossible. Which ones?

b. The code on the previous page would operate correctly with *no code after the loop*, even though it would *not be correct* in that case with the invariant given. Why would it still work properly in that case?