CSE 331 Autumn 2020 Homework 2

1. Fill in the proof of correctness for the method `evalPoly` on the next page. It takes the coefficients of a polynomial in the array `A`. For example, the polynomial $3 + 2x + 4x^2$ would be represented with A = [3, 2, 4]. It takes a value for x in the second parameter, `v`. The goal of the method is to return the value of the polynomial at x = v.

   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.

   Reason in the direction (forward or backward) indicated by the arrows on each line. Within the body of the loop, you will reason forward from the top and backward from the bottom, with the two meeting in the middle as indicated. Where they meet, explain why the top assertion implies the bottom one.

   Notes on the notation used:

   - Use “n” to denote the length of A.
   - The precondition, which is assumed true at the start of the method, will remain true throughout (since `A.length` cannot be changed). To save space, we will not bother to write this additional fact (0 < A.length) on every line. (We will do the same for the preconditions in each of the other problems in this assignment as well.)

```
{{ Precondition: 0 < n = A.length }}
float evalPoly(float[] A, float v) {
↓  int i = A.length - 1;
   {{ ______________________________________________________________ }}
↓  int j = 0;
   {{ ______________________________________________________________ }}
↓  float val = A[i];
   {{ ______________________________________________________________ }}


   {{ Inv: val = A[i] + A[i+1] v + … + A[n-1] v^j and i + j = n – 1 }}
   while (j != A.length - 1) {
     {{ ______________________________________________________________ }}
↓    j = j + 1;
     {{ ______________________________________________________________ }}
↓    i = i - 1;
     {{ ______________________________________________________________ }}


     {{ ______________________________________________________________ }}
↑    val = val * v + A[i];
     {{ ______________________________________________________________ }}
↑  }

↓
   {{ ______________________________________________________________ }}


   {{ Postcondition: val = A[0] + A[1] v + … + A[n-1] v^(n-1) }}
   return val;
}
```

2. Fill in the missing parts of each implementation of the method `removeChar` below. It takes as input a string `s` and a character `c`, and it returns `s` with all the `c`'s removed.

   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 statements outside of the loop, (2) add any additional loops, or (3) call any methods other than `String.charAt`, `String.length`, and `StringBuilder.append`.

   Notes on the notation used:

   - To formally argue correctness, we need a formal definition of what the code is supposed to do. The function del_c referenced in the assertions below does that. Informally, $\text{del}_c(s)$ means the string s but with all c's removed.
   - Formally, we can define del_c recursively (as in CSE 311) as follows:

     $\text{del}_c(\text{“”}) = \text{“”}$

     $\text{del}_c(wa) = \text{del}_c(w)$ if $a = c$
     $\text{del}_c(wa) = \text{del}_c(w)\ a$ otherwise

     (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 }}

```
String removeChar(String s, char c) {
  StringBuilder t = new StringBuilder();
  int i =

  {{ Inv: t = del_c(s[0 .. i-1]) }}
  while (                     ) {



    i = i + 1;
  }

  {{ Postcondition: t = del_c(s) }}
  return t.toString();
}
```

b) {{ Precondition: s != null }}

```
String removeChar(String s, char c) {
  StringBuilder t = new StringBuilder();
  int i =

  {{ Inv: t = del_c(s[0 .. i]) }}
  while (                      ) {



    i = i + 1;
  }

  {{ Postcondition: t = del_c(s) }}
  return t.toString();
}
```

c) {{ Precondition: s != null }}

```
String removeChar(String s, char c) {
  StringBuilder t = new StringBuilder();
  int i =

  {{ Inv: t = del_c(s[0 .. i]) }}
  while (                    ) {
    i = i + 1;



  }

  {{ Postcondition: t = del_c(s) }}
  return t.toString();
}
```

d) {{ Precondition: s != null }}

```
String removeChar(String s, char c) {
  StringBuilder t = new StringBuilder();
  int i =

  {{ Inv: t = del_c(s[0 .. i-1]) }}
  while (                    ) {
    i = i + 1;



  }

  {{ Postcondition: t = del_c(s) }}
  return t.toString();
}
```

3. Fill in the proof of correctness for the method `isSubArray` on the next page. It takes as input two arrays `A` and `B`. The goal is to determine whether `B` appears somewhere within `A`, i.e., whether A = X + B + Y for some sequences X and Y.

   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.

   Notes on the notation used:

   - Use "n" to denote the length of `A` and "m" to denote the length of `B`.
   - The notation "A[i .. j]" means the subarray from index i up to and including index j. For example, if A = [0, 2, 4, 6, 8, 10], then A[2 .. 4] = [4, 6, 8]. If j = i – 1, then this indicates an *empty array*. In symbols, we have A[i .. i-1] = [].

   Hints for the "else" case of the outer loop:

   - It may be easier to prove the contrapositive. I.e., instead of showing that the top assertion implies the bottom one, show that the bottom one being false implies that the top one is also false.
   - If B is a subarray of A[0 .. i], then B must match some length-m subarray. We can consider separately the case that the match ends at index i and the case that the match ends before index i. (We called this "proof by cases" in 311.)

```
{{ Precondition: 0 < n = A.length and 0 < m = B.length }}
boolean isSubArray(int[] A, int[] B) {
↓  int i = B.length - 1;
   {{ _______________________________________________________________ }}



   {{ Inv: B is not a subarray of A[0 .. i-1] }}
   while (i != A.length) {
↓
     {{ _______________________________________________________________ }}
↓    int k = 0;
     {{ _______________________________________________________________ }}



     // Try to match B to the length-m subarray ending at index i. The latter
     // is A[i-m+1 .. i], so we need B[k] = A[i-m+1+k] for k = 0 .. m-1.
     // (This says B[0] = A[i-m+1] when k=0 and B[m-1] = A[i] when k = m-1.)
     {{ Inv: B is not a subarray of A[0 .. i-1] and B[0 .. k-1] = A[i-m+1 .. i-m+k] }}
     while (k != B.length && B[k] == A[i − B.length + k + 1]) {
↓
       {{ _______________________________________________________________
          _______________________________________________________________ }}



       {{ _______________________________________________________________ }}
↑      k = k + 1;
       {{ _______________________________________________________________ }}
     }
↓
```

(code continues on the next page…)

```
↓
     {{ ____________________________________________________________________
        ____________________________________________________________________ }}

     if (k == B.length) {
↓
       {{ __________________________________________________________________
          __________________________________________________________________ }}



       {{ Postcondition: B = A[i-m+1 .. i] }}
       return true;
     } else {
↓
       {{ __________________________________________________________________
          __________________________________________________________________ }}



(see hints)




       {{ __________________________________________________________________ }}
↑
     }

     {{ ____________________________________________________________________ }}
↑    i = i + 1;
     {{ ____________________________________________________________________ }}
↑  }

↓
   {{ ______________________________________________________________________ }}


   {{ Postcondition: B is not a subarray of A }}
   return false;
}
```

4. Fill in the missing parts of the implementation of the method `movingAverage` on the next page. It takes as input an array `A` and an integer `k`, along with an array `B` into which the output will be written. The goal is to write into B the average of each length-k subarray of A. I.e., B[0] = (A[0] + … + A[k-1]) / k and B[1] = (A[1] + … + A[k]) / k and so on.

   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:

   - Use "n" to denote the length of A. (The length of B will hence be n – k + 1.)
   - Use "avg" to denote the function that averages the values in an array. In other words, "avg A[i .. j]" means (A[i] + A[i+1] + … + A[j]) / (j – i + 1).

```
{{ Precondition: 0 < k <= n = A.length and n – k + 1 = B.length }}
void movingAverage(float[] A, int k, float[] B) {
  float s = 0;
  int j =

  {{ Inv: s = A[0] + … + A[j-1] }}
  while (                              ) {


    j = j + 1;
  }

  int i = 0;




  {{ Inv: B[0] = avg A[0 .. k-1], B[1] = avg A[1 .. k], …, B[i] = avg A[i .. i+k-1] and
        s = A[i] + … + A[i+k-1] }}
  while (                              ) {




    i = i + 1;
  }

  {{ Postcondition: B[0] = avg A[0 .. k-1], B[1] = avg A[1 .. k], …, B[n-k] = avg A[n-k .. n-1] }}
}
```

5. Fill in the missing parts of the method `runLengthDecode` on the next page. It takes as input an array of characters `A` and an array of integers `B`. The goal is to return a string that has B[0] copies of character A[0] followed by B[1] copies of character A[1] and so on.

   In this case, the invariant for the outer loop is provided, but you must fill in the invariant for the inner loop.

   Your code must be correct according to the loop invariants in your answer. 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** call any method other than `StringBuilder.append`. You can add additional **while** loops, but the invariant for each loop must also be given.

   Notes on the notation used:

   - Use “n” to denote the length of A.
   - The invariant and postcondition use the notation “a * b”, when a is a character and b is a non-negative integer, to mean the string that has b copies of the character a. E.g., ‘w’ * 3 would be “www”.

```
{{ Precondition: 0 < n = A.length = B.length }}
String runLengthCode(char[] A, int[] B) {
  StringBuilder s = new StringBuilder();
  int i =



  {{ Inv: s = A[0] * B[0] + A[1] * B[1] + … + A[i-1] * B[i-1] }}
  while (                          ) {














  }



  {{ s = A[0] * B[0] + A[1] * B[1] + … + A[n-1] * B[n-1] }}
  return s.toString();
}
```