Background

In this problem, we will verify the correctness of operations that work with finite sets of real numbers stored in a sorted array. We will also include the number -∞ as the first element in the array and +∞ as the last element in the array. This removes some special cases that would otherwise need to be handled in the code.

In summary, the array S represents a set in this manner iff it satisfies the constraint that

$$-\infty = S[0] < S[1] < S[2] < \ldots < S[n] < S[n+1] = +\infty$$

where n is the number of elements in the set (which is 2 less than the length of S due to the extra elements at the front and back). This array represents the set {S[1], S[2], .., S[n]}.

In other words, the constraint above is the *representation invariant*, and for an array S satisfying this invariant, its *abstract value* is {S[1], S[2], .., S[n]}.

We will represent real numbers using Java's `float` type. In addition to being able to represent a wide range of numbers (e.g., -10^{100}, $+10^{100}$, and $1/10^{100}$), a `float` can also represent -∞ and +∞. Although we will not need it in the code below, there is also a special `float` value called NaN ("not a number"), which is produced for invalid operations. For example, the calculation 1.0 / 0.0 produces +∞, while the calculation 0.0 / 0.0 produces NaN.

In these problems, we will not require you to *show* every line of reasoning, only the most important ones. However, you will likely want to fill in the important assertions by working through the code line-by-line as before.

Hints: The following facts will be useful in your explanations:

- The assertion "a < min(b, c)" is equivalent to (another way of writing) "a < b and a < c".
- The assertion "max(a, b) < c" is equivalent to "a < c and b < c".
- Together, "max(a, b) < min(c, d)" is equivalent to "a < c and a < d and b < c and b < d".
- These equivalences also hold with "<" replaced by "≤".

Understanding the Specification

To prove correctness, we need a specification. What does the specification of the `union` method say? Write a short English interpretation for the precondition and postcondition below.

1. Pre (be sure to describe the given variables, e.g. parameters)

2. Post

Understanding the Invariant

What does the invariant in the `union` method claim? Write a short English interpretation for **each fact** in the loop invariant below. You are welcome to include examples to provide intuition.

3. Pre (same as above, no explanation needed)

4. $-\infty = U[0] < U[1] < \ldots < U[k-1] < \min(S[i], T[j])$

5. $\{U[1], U[2], \ldots, U[k-1]\} = \{S[1], S[2], \ldots, S[i-1]\} \cup \{T[1], T[2], \ldots, T[j-1]\}$

Verifying Correctness of Union

Fill in the indicated assertions by reasoning in the direction indicated by the arrows. We will address the places where "`?`"s appear in late questions. We will use `n` and `m` to refer to the number of points in sets S and T, respectively.

{{ Pre: $-\infty = S[0] < S[1] < \ldots < S[n] < S[n+1] = \infty$ and $-\infty = T[0] < T[1] < \ldots < T[m] < T[m+1] = \infty$ and
U is an array containing at least n+m+2 elements }}

```
↓ U[0] = Float.NEGATIVE_INFINITY;
↓ int i = 1, j = 1, k = 1;
```

{{ Pre and _____ }}

```
  ? answer 1 on a separate page

 {{ Inv: Pre and -∞ = U[0] < U[1] < … < U[k-1] < min(S[i], T[j]) and
              {U[1], U[2], …, U[k-1]} = {S[1], S[2], ..., S[i-1]} ∪ {T[1], T[2], …, T[j-1]} }}
 while (S[i] < Float.POSITIVE_INFINITY || T[j] < Float.POSITIVE_INFINITY) {
↓   if (S[i] < T[j]) {
      {{ Inv and _____________________________________________________________ }}
      U[k] = S[i];   // ? answer 2 on the page 4
      {{ _____________________________________________________________
          _____________________________________________________________ }}
↑     i = i + 1;
↑     k = k + 1;
↓   } else if (S[i] > T[j]) {
      {{ Inv and _____________________________________________________________ }}
       U[k] = T[j];   // ? answer 3 on the page 4
      {{ _____________________________________________________________
          _____________________________________________________________ }}
↑      j = j + 1;
↑      k = k + 1;
↓   } else {
      {{ Inv and _____________________________________________________________ }}
       U[k] = S[i];   // ? answer 4 on the page 4
      {{ _____________________________________________________________
          _____________________________________________________________ }}

      // continued on the next page…
↑      i = i + 1;
↑      j = j + 1;
↑      k = k + 1;
    }
  }

↓ U[k] = Float.POSITIVE_INFINITY;
 {{ Inv and _____________________________________________________________ }}

 ? answer 5 on the next page

 {{ Post:  -∞ = U[0] < U[1] < … < U[k] = ∞ and
         {U[1], U[2], …, U[k-1]} = {S[1], S[2], ..., S[n]} ∪ {T[1], T[2], …, T[m]} }}
```

Explanations

Explain why the invariant holds initially (answer 1) and why the postcondition holds (answer 5). Keep your explanations concise. You should not need more than 1-4 sentences for each part.

1.

5.

Comparing Assertions

The three "?"s in the middle of the loop (answers 2-4) appear next to an assignment and between two assertions. In each case, compare the two assertions and explain which *additional facts*, not listed in the top assertion, are needed to establish that the bottom assertion holds. In other words, **list each fact** in the bottom assertion that is not stated exactly in the top assertion. In the next question, you will justify why these facts hold.

2.

3.

4.

More Explanations

For each case, explain why the additional facts in the bottom assertion hold. You should argue why **each fact** that you wrote in “Comparing Assertions” follows from the top assertion. You will likely need to use the hints provided in the background section on the first page.

Try to keep your arguments concise – you probably will not need all the space provided.

2.

3.

4.

Code Review

For which of the five cases we considered earlier (before the loop, after the loop, and the three parts of the if/else if/else statement), do you think the author should have included comments? Briefly justify your answers. It may help to consider which cases you found most confusing.

Nice Work So Far!

You have successfully checked the correctness of a complex algorithm! While arrays are familiar data structures, the constraints added to them with this representation are tricky. If you were able to work through this example, I am confident that you can check the correctness of other similarly complex algorithms.

As you continue programming, you will find that this example is **typical** of what it is like to check the correctness of complex algorithms. They typically have invariants with many facts to keep track of and loop bodies with several cases to consider. We check their correctness exactly as you did above: by reasoning forward and backward into each case, comparing the facts known before and needed after to identify the additional facts that need to hold, and then figuring out why the known facts in that specific case ensure that they do always hold.