CS 284: Homework 3
Due: Tuesday, March 8th, 2022 at 11:59pm

1 Assignment Policies

Don't forget the honor pledge!

Collaboration Policy. Homework will be done individually: each student must hand in their own answers. It is acceptable for students to collaborate in understanding the material but not in solving the problems or programming. Use of the Internet is allowed, but should not include searching for existing solutions.

Under absolutely no circumstances code can be exchanged between students. Excerpts of code presented in class can be used.

Assignments from previous offerings of the course must not be re-used. Violations will be penalized appropriately.

2 Assignment

This short assignment is meant to give you practice with recursion, which we will be using to define more interesting data structures after the midterm.

3 Submission instructions

Submit a single file named `Recursion.zip` through Canvas that has the following structure:

```
src/
  cs284/
    Permutations.java
    BinarySearch.java
```

No report is required. Your grade will be determined as follows:

- You will get 0 if your code does not compile.
- The code must implement the following UML diagram precisely. Helper functions are of course allowed.

- You implementation of `Permutations.allPermutations` and `BinarySearch.binarySearch` must use recursion in an essential way to receive more than 50% credit. (`Permutations.allInsertions` can be implemented however you like.)
- Your code style and readability count for "style points".

4 Permutations of a list

For this section, all of your functions should go on a class `Permutations`. We'd like to you define a static method `allPermutations` that takes a list and returns a list of lists, containing every possible permutation of the input. You should not alter the input list in any way (i.e., do not call any destructive methods, like `remove`). For example, the following code:

```
public static void main(String[] args) {
   List<String> l = new LinkedList<>();
   l.add("a");
   l.add("b");
   l.add("c");

   for (var perm : allPermutations(l)) {
     StringBuffer sb = new StringBuffer();

     for (var s : perm) {
       sb.append(s);
     }

     System.out.println(sb);
   }
}
```

Produces the following output:

```
abc
bac
bca
acb
cab
cba
```

Your code should produce output in the same order. To get more than 50% credit, you must use recursion to implement `allPermutations`. It's okay to have a loop in there, so long as recursion is used meaningfully. (It's very hard to write this code without recursion, anyway.) We recommend the following recipe:

- Use `List.get(int)` to select the first element.
- Use `List.sublist(int,int)` to select the sublist of all remaining elements. (NB this method is very cheap, as the sublist won't allocate a whole new list, but instead offer a 'view' of the original list.)
- For each permutation of the sublist (here you may use a loop), use the `allInsertions(E, List<E>)` helper method to get the list of lists that puts the first element in each possible position.

- Return the accumulated, inserted permutations.

To implement `allInsertions`, you'll want to create copies of the input list, putting the inserted element in each possible position. This code is probably easiest to write with a `for` loop, but recursion works great, too. Either implementation is okay.

Here's an example of how `allInsertions` works. The following code:

```
public static void main(String[] args) {
  List<String> l = new LinkedList<>();
  l.add("a");
  l.add("b");
  l.add("c");

  for (var lIns : allInsertions("!", l)) {
    System.out.println(lIns);
  }
}
```

Should produce the following output:

```
[!, a, b, c]
[a, !, b, c]
[a, b, !, c]
[a, b, c, !]
```

Our tester will exercise both of your functions, so be certain to make them `public static`. You may define other helpers as necessary (but we didn't need any). Be sure to test *both* functions!

4.1 Performance

The following questions are *not* part of the homework, but they are stimulating to think about. If given a list `l` of length n, how many lists will `allPermutations(l)` produce? What is the big-O complexity class of `allInsertions`? What about `allPermutations`?

5 Binary search

Binary search is a fast algorithm for finding an element in a *sorted* list. Here's the intuition: suppose we're looking for the number 10 in a list of sorted numbers. We glance at the list and see the number 50. If the list is sorted ascending, we know that if 10 is in the list, it's to the left of 50—we can ignore everything to the right, since it's greater than or equal to 50 which is greater than 10.

The algorithm is called binary search, because we cut the elements we're looking at—the range of the list we're considering—in half every time. Here's the idea to find a needle/target element `v` in a list of `n` elements:

1. Start out with a lower bound of `lo = 0` and an upper bound of `hi = n` (exclusive). These define our current range.

2. Look at the middle element of our range (i.e., `mid = lo + ((hi-lo) / 2)`), call it `x`.

(a) If `v` is equal to `x`, return the index `mid`.

(b) If `v` is less than `x`, recursively search from `lo` to `mid` (exclusive).

(c) If `v` is greater than `x`, recursively search from `mid+1` to `hi` (exclusive).

3. If `lo >= hi`, the element is not in the list.

4. Go back to step (2).

It's also possible to structure this function so that the upper index is inclusive, rather than exclusive. Doing so will slightly change the termination condition.

We ask you to implement a method `BinarySearch.binarySearch(E, E[])`. You'll notice that method has no `lo` or `hi` arguments. Your method should call a `private static` helper method to do the binary search. To receive more than 50% credit, your helper method must do its work using recursion, not iteration. That is, there should be no loops of any kind in your `BinarySearch` class.

5.1 Performance

Binary search is a more complicated algorithm than linear search—it's very easy to get it wrong! But it comes with a huge advantage: it is much more performant. If your list is sorted, binary search is very much superior to linear search.

A deep analysis requires some more advanced tools (induction, and some careful math). But we can sketch it out here, again using "number of comparisons" as our measure.

In the best case, our element is exactly in the middle. We found it on our first go, making only one comparison.

In the worst case, our element isn't in the list. How many elements will be consider? Each time we run the loop, our range halves. How many times can we run the loop before our range shrinks to nothing (i.e, `hi <= lo`) and we have to give up? The number of times you can halve a number corresponds to its logarithm base 2 a/k/a its binary logarithm, written $\log_2(n)$.

Logarithmic performance is fantastic. Why? Well, suppose we have 1000 elements. Linear search will step through the list from start to finish, performing 1000 comparisons before giving up. We'll perform $\log_2(1000)$ comparisons. If you know that $2^{10} = 1024$, you can guess how many comparisons, but we can also just compute:

$$\begin{aligned} 1000/2 &= 500 \\ 500/2 &= 250 \\ 250/2 &= 125 \\ 125/2 &\simeq 62 \\ 62/2 &= 31 \\ 31/2 &\simeq 15 \\ 15/2 &\simeq 7 \\ 7/2 &\simeq 3 \\ 3/2 &\simeq 1 \\ 1/2 &\simeq 0 \end{aligned}$$

We can expect 10 comparisons before giving up. That's a huge improvement over linear search's 1000 comparisons! On 2000 elements, we'd have 11 comparisons, while linear search would do... 2000! Double yikes!

Our average case resembles the worst case—given uniform distribution, we can expect to do no more than $\log_2(n)$ comparisons.

6 Technical details

6.1 UML

The class `Permutations` should include the following operations:

Permutations
public static <E> List< List<E> > allInsertions(E elt, List<E> l); public static <E> List< List<E> > allPermutations(List<E> l);

The class `BinarySearch` should include the following operations:

BinarySearch
public static < E extends Comparable<? super E> > int binarySearch(E elt, E[] a);

As always, helper functions are allowed.

6.2 Fancy parametric signatures

Both `Permutations` and `BinarySearch` defined special parametric methods. So far, we've only put parameters on classes, which are 'global' for the whole class. Here, our type parameters are per-method.

The syntax is `modifiers <T,...> ret-type name(arg-type arg-name, ...)`. The type parameters `T` (etc.) is in scope for both `ret-type` and for each `arg-type`.

There's a new syntax for `BinarySearch`. For the `binarySearch` static method, we have a *bounded* type parameter `E`. Here, we given an upper bound on `E`, saying `E` must be a subclass of `Comparable<? super E>`. Since `Comparable` is an interface, 'subclass of' really means `implements`. That is, we can only call `binarySearch` on arrays that hold `E`s that implement `Comparable<? super E>`.

So... what does `Comparable<? super E>` mean? Here, we give an *upper bound* on the parameter of `Comparable`. That is, each `E` must be able to compared to *some* type `T` such that `T` is a super type of `E`. We could have said `Comparable<E>` to say that `E` must be comparable directly to itself. But that ends up being too specific. Suppose the type `E` is integers; we might say that `E` is comparable not just to other integers, but rationals or reals, too. If we required `Comparable<E>`, then being comparable to a super type wouldn't work.

To sum up: all of these bounds on *binarySearch* combine to say that we only work with types `E` that can be compared to *themselves* (and maybe other things, too).