Task 1: Last Index (2 points)

For this task, save your work in `Last.java`

Consider the binary search algorithm from class. A moment's thought reveals that if the key we're looking for appears multiple times, the algorithm, as written, may return the index of any of them.

In this subtask, you'll adapt the binary search algorithm so that it always returns the index of the last occurrence of the search key. More precisely, let a be a sorted list of elements (i.e., $a[0] \le a[1] \le \cdots \le a[n-1]$, where n is the length of a). You'll implement a function

```
public static Integer binarySearchLast(int[] a, int k)
```

that returns the value of $\max\{i \mid a[i] = k\}$; or otherwise, returns `null` if $k \notin a$. For example:

- `binarySearchLast`$(\{1,2,2,2,4,5\},2)$ returns 3.
- `binarySearchLast`$(\{1,2,2,2,4,5\},0)$ returns `null`.
- `binarySearchLast`$(\{1,2,2,2,4,5\},5)$ returns 5.

Whatever the input may be, your implementation must take at most $O(\log n)$ time, $n = \texttt{len}(a)$, to receive full credit.

Hints: Your code must run in $O(\log n)$ even in the case when there is a long streak (say of length about n) of the key we're searching for. It won't be fast enough to use standard binary search and keep moving right from that point by one until we reach the last key.

Task 2: Zombies (2 points)

For this task, save your work in `Zombies.java`

In a remote village known as Salaya, zombies and humans have lived happily together for many decades. In fact, no one can quite tell zombies and humans apart. However, when these "people" line up in a single row, all sorts of trouble ensue, including this weird phenomenon: human beings will line themselves up from tall to short, but zombines act erratically.

In particular, if `line` is an array of heights of the population of this village, we would expect that `line`$[i] \ge$ `line`$[j]$ for $i \le j$. But this simply isn't true in many cases especially with zombies around.

Hence, one nobleman—or is he a zombie?—came to you for help: he wants to know how many pairs of his people violate this social norm.

Your Task: Write a function `public static int countBad(int[] hs)` that takes an array of n numbers and returns the number of pairs $0 \le i < j < n$ such that $\texttt{hs}[i] < \texttt{hs}[j]$ (i.e., the number of pairs that violate the social norm).

For example (abusing Java's array notation):

- `countBad({35, 22, 10}) == 0`
- `countBad({3,1,4,2}) == 3`
- `countBad({5,4,11,7}) == 4`
- `countBad({1, 7, 22, 13, 25, 4, 10, 34, 16, 28, 19, 31}) == 49`

Performance Expectations: We expect your code to run in at most $O(n \log n)$ time, where n is the length of the input array. Your program should give the same answer as the following *inefficient* code:

```
public class NaiveZombies {
  public static int countBad(int[] hs) {
    int badPairs=0;
    for (int i=0;i<hs.length;i++)
      for (int j=i+1;j<hs.length;j++)
        if (hs[i] < hs[j]) badPairs++;

    return badPairs;
  }
}
```

(Hint: Write a `merge`*-like algorithm that computes two things: (1) the combined sorted list and (2) the number of out-of-wack pairs. When you're done, it should look almost identical to the merge sort algorithm except it computes this additional thing.)*

Task 3: Quick Sort Recurrence (4 points)

Consider the recurrence

$$f(n) = n + 1 + \frac{2}{n}\big(f(n-1) + f(n-2) + \cdots + f(1)\big), \quad \text{where } f(0) = 0.$$

This recurrence represents the "average" running time of the randomized quick sort algorithm[1]. For now, we won't discuss how one can come up with such a recurrence. In this exercise, we're interested in solving this recurrence for a closed form. You'll do this in a few steps:

(i) Consider $f(n)$ and $f(n-1)$, where $n \ge 2$. We have

$$f(n) = (n+1) + \frac{2}{n}\big(f(n-1) + f(n-2) + \cdots + f(1)\big)$$
$$f(n-1) = n + \frac{2}{n-1}\big(f(n-2) + f(n-3) + \cdots + f(1)\big)$$

[1] Technically, for those who took Discrete Math already, this is the expected running time.

By multiplying the first equation by n and the second equation by $(n-1)$, we have

$$n \cdot f(n) = (n+1)n + 2\big(f(n-1) + f(n-2) + \cdots + f(1)\big) \tag{1}$$
$$(n-1)f(n-1) = n(n-1) + 2\big(f(n-2) + f(n-3) + \cdots + f(1)\big) \tag{2}$$

Subtracting equation (2) from equation (1), we'll get

$$n \cdot f(n) - (n-1)f(n-1) = 2n + 2f(n-1)$$

In other words,

$$n \cdot f(n) = 2n + (n+1)f(n-1) \tag{3}$$

Your task in this step is to understand the derivation we just made. Other than that, no actions are required on your part.

(ii) Let $g(n) = \frac{f(n)}{n+1}$. Can you write what you have in terms of the function g? (*Hint:* divide equation (3) by $n(n+1)$.)

(iii) Your task in this step is to find a closed form for g. The recurrence g that you have isn't listed in the table. But you can easily solve for a closed form. Before you begin, let us show you how to solve a related recurrence: $h(n) = h(n-1) + \frac{1}{n}$, with $h(0) = 0$

We start out by unraveling $h(n)$. By the definition of $h(n)$,

$$\begin{aligned} h(n) &= h(n-1) + \frac{1}{n} && \text{expand the recurrence one more time} \\ &= h(n-2) + \frac{1}{n} + \frac{1}{n-1} && \text{expand the recurrence one more time} \\ &= h(n-3) + \frac{1}{n} + \frac{1}{n-1} + \frac{1}{n-2} \end{aligned}$$

It is pretty clear that if we keep on expanding the recurrence, we'll get

$$h(n) = \frac{1}{n} + \frac{1}{n-1} + \frac{1}{n-2} + \cdots + \frac{1}{3} + \frac{1}{2} + 1.$$

In Math, this has a name: the n-th *Harmonic number*, denoted by H_n, is given by $H_n = \frac{1}{n} + \frac{1}{n-1} + \frac{1}{n-2} + \cdots + \frac{1}{3} + \frac{1}{2} + 1$, so in terms of Harmonic number $h(n) = H_n$.

(iv) Now that you can express $g(n)$ in terms of some expression involving Harmonic numbers, you can proceed to derive a closed form for $f(n)$. Finally, use the following fact to conclude that $f(n)$ is $O(n \ln n)$:

Fact: $H_n \leq 1 + \ln(n)$, where ln denotes the natural logarithm.