# Homework 4

Data Structures & Algorithms (Term III/2018–19)

*built on 2019/05/24 at 07:59:38* ***due:*** *thu may 30 @ 11:59pm*

This assignment contains both a coding portion and a written portion. The aim of this assignment is to provide you with more experience solving algorithmic problems in Java and reasoning about code. This assignment requires a starter package, which can be downloaded from the course website.

### READ THIS BEFORE YOU BEGIN:

- All your work will be handed in as a single zip file. Call this file `a4.zip`. You'll upload this to Canvas before the assignment is due.
- For the written part, you *must* typeset your answers and hand it in as a PDF file called `hw4.pdf`, which will go inside your zip file. No other format will be accepted. To typeset your homework, apart from Microsoft Word, there are LibreOffice and LaTeX, which we recommend. Note that a scan of your handwritten solution will not be accepted.
- Be sure to **disclose your collaborators in the PDF file.**
- For each task, save your work in a file as described in the task description.
- A script will process and grade your submission before any human being looks at it. *Do not use different function/file names.* The script is not as forgiving as we are.
- Use the Internet to help you learn: It's OK to look up syntax or how a function/class/method is used. It's **NOT** OK to look up how to solve a problem or answers to a problem. The goal here is to learn, not to just hand in an answer. If you wish to do that, just give us a link to the solution!
- You are encouraged to work with other students. However, **you must write up the solutions separately on your own and in your own words**. This also means you must not look at or copy someone else's code.
- Finally, the course staff is here to help. We'll steer you toward solutions. Catch us in real-life or online on Canvas discussion.

**Collaboration Policy:** To facilitate cooperative learning, you are permitted to discuss homework questions with other students provided that the following whiteboard policy is respected. A discussion may take place at the whiteboard (or using scrap paper, etc.), but no one is allowed to take notes or record the discussion or what is written on the board. *The fact that you can recreate the solution from memory is taken as proof that you actually understood it, and you may actually be interviewed about your answers.*

## Exercise 1: Last Index (2 points)

*For this task, save your code 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 (abusing notation):

- `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.

## Exercise 2: Stuttering Substring (6 points)

*For this task, save your code in* `Stutter.java`

We saw the stuttering-substring problem in class. Let's review that.

**Substring:** If $A$ and $B$ are two strings, we say that $A$ is a *substring* of $B$ if it is possible to find the letters of $A$ inside of $B$ in the same order that they appear originally in $A$, potentially skipping some letters of $B$. More formally, we say that $A$ is a substring of $B$ if there are indicies $0 \le i_0 < i_1 < i_2 < \cdots < i_{n-1} < m$ into $B$ such that

$$B[i_k] = A[k], \qquad \forall\, k = 0, 1, 2, \ldots, n-1.$$

As examples: "cat" is a substring "excavate", but "vet" isn't a substring of "excavate"

**Stuttering:** Let $A = a_0 a_1 a_2 \ldots a_{n-1}$. We define the concept of *stuttering* as follows: $A^{(k)}$ expands the string $A$ by repeating each $a_i$ consecutively $k$ times. We define $A^{(0)}$ to generate the empty string, which is a substring of any $B$.

For example, if $A$ = "hello", then $A^{(2)}$ = "hheelllloo". And if $A$ = "hi", then $A^{(5)}$ = "hhhhhiiiii".

Given strings $A$ and $B$, the *stuttering-substring problem* is to find the largest $k \ge 0$ such that $A^{(k)}$ is a substring of $B$. This problem is well-defined because for sure $A^{(0)}$ is a substring of $B$. But what is the largest such $k$?

**Subtask I:** Implement a function `public static boolean isSubstr(String a, String b)` that takes two strings $a$ and $b$ as input, and returns a Boolean indicating whether $a$ is a substring of $b$. To receive full credit, it must run in $O(\texttt{len}(a) + \texttt{len}(b))$ time.

**Subtask II:** You will analyze why your function meets in running time bound. (*Hint:* Argue that each letter is "touched" at most once. Look at `merge` in merge sort for inspiration.)

**Subtask III:** In this quick subtask, you'll write a function `public static String stutter(String A, int k)` that takes a string $A$ and a number $k \geq 0$ and produces $A^{(k)}$. For $A$ of length $n$, we expect this function to run within $O(nk)$ time.

**Subtask IV:** Using the pieces we have built so far (from a previous task as well), you'll implement (perhaps by reusing the logic from your previous task) a function `public static int maxStutter(String a, String b)` that takes as input two *nonempty* strings $a$ and $b$ and returns a number that is the answer to the stuttering-substring problem for the pair of strings.

**Subtask V:** Analyze the running time of your implementation. We expect your code to run in at most $O((m+n)\log(\frac{m}{n}))$ time, where $m = \texttt{len}(B)$ and $n = \texttt{len}(A)$.

(*Hint:* To analyze the running time, write a recurrence to determine the number of "probes." How should we define a probe? If the number of probes is $O(P(n))$ and each probe costs at most $O(f(n))$, then the total running time is at most $O(P(n)f(n))$. )

## Exercise 3: Zombies (2 points)

*For this task, save your code 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`$[i] \geq$ `line`$[j]$ for $i \leq j$. 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 $0 \leq i < j < n$ such that `hs`$[i] >$ `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.)*

**How to return two values?** There are many ways one can return multiple values. For this task, we're providing for you a class `Pair<S, T>` with the following features: The class represents an ordered-pair $(s, t)$, where $s$ has type $S$ and $t$ has type $T$.

- To create an ordered pair, you can write `new Pair<S, T>(s_value, t_value)`, where $S$ and $T$ are the appropriate types. For example, `new Pair<Integer List<Integer>>(25, sortedList)` results in a pair with 25 and `sortedList`.
- If $p$ is a pair, then `p.first` is the left value in the ordered pair, and `p.second` is the right value in the ordered pair.

As an example, the following code makes a pair of an integer and a string—and prints the ordered pair (a useful thing for debugging):

```
Pair<Integer, String> p = new Pair<>(73001, "Hello World");
System.out.println(p);
Integer theIntPart = p.first;
String theStrPart = p.second;
System.out.println(theStrPart);
System.out.println(theIntPart);
```

## Exercise 4: Quick Sort Recurrence (4 points)

Consider the recurrence

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

This recurrence represents the "average" running time of the randomized quick sort algorithm[1]. 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 $n \geq 2$. We have

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

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

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

(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 the closed form for $h(n)$ is $h(n) = H_n$.

(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.

## Exercise 5: HackerRank Problems (6 + 4 extra-credit points)

*For this task, save your code in* `hackerrank.txt`

There are *five* problems in this set; only three problems are required—if you solve all four, you'll get some extra credits. You **must** write your solutions in Java (1.8). You will hand them in electronically on the HackerRank website as part of the contest.

**Important:** You will write down your Hacker ID username in a file called `hackerrank.txt`, which you will submit as part of the assignment. This will be used to match you with your submission on HackerRank.

You can find your problems at

`https://www.hackerrank.com/muic-data-structures-t-318-assignment-4`