Homework 2

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

built on 2019/05/02 at 16:19:04 ***due:*** *thu may 9 @ 11:59pm*

This assignment contains both a coding portion and a written portion. The aim of this assignment is to get you thinking about algorithmic problems in Java, gain more experience analyzing running times, and practice new concepts that we covered in class this week.

READ THIS BEFORE YOU BEGIN:

- All your work will be handed in as a single zip file. Call this file `a2.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 `hw2.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: Poisoned Wine (2 points)

You own n bottles of wine, exactly one of which has been poisoned. You don't know which bottle, however. What you know is, if someone drinks just a tiny amount of wine from the poisoned bottle, s/he will start laughing uncontrollably after 30 days. In fact, the poison is so potent that even the faintest drop, diluted over and over, will still cause the symptom.

Design a scheme that determines exactly which one bottle was poisoned. You are allowed 31 days and can expend $O(\log n)$ testers (people). *Explain why your scheme meets the $O(\log n)$-tester requirement.*

(*Hint:* Numbers between 1 and n (inclusive) can be represented using $\lceil \log_2(n+1) \rceil$ bits.)

(*More Hint:* Think hard about the first hint before being spoiled by this hint. If the bottles are labeled 1 through n, how can we find out the i-th bit of the label of the poisoned bottle?)

Exercise 2: More Running Time Analysis (6 points)

For the most part, we have focused almost exclusively on worst-case running time. In this problem, we are going to pay closer attention to these Java methods and consider their worst-case and best-case behaviors. The *best-case* behavior is the running time on the input that yields the fastest running time. The *worst-case* behavior is the running time on the input that yields the slowest running time.

(1) Determine the *best-case* running time and the *worst-case* running time of `method1` in terms of Θ.

```java
static void method1(int[] array) {
    int n = array.length;
    for (int index=0;index<n-1;index++) {
        int marker = helperMethod1(array, index, n - 1);
        swap(array, marker, index);
    }
}
static void swap(int[] array, int i, int j) {
    int temp=array[i];
    array[i]=array[j];
    array[j]=temp;
}
static int helperMethod1(int[] array, int first, int last) {
    int max = array[first];
    int indexOfMax = first;

    for (int i=last;i>first;i--) {
        if (array[i] > max) {
            max = array[i];
            indexOfMax = i;
        }
    }
    return indexOfMax;
}
```

(2) Determine the *best-case* running time and the *worst-case* running time of `method2` in terms of Θ.

```java
static boolean method2(int[] array, int key) {
    int n = array.length;
    for (int index=0;index<n;index++) {
        if (array[index] == key) return true;
    }
    return false;
}
```

(3) Determine the *best-case* running time and the *worst-case* running time of `method3` in terms of Θ.

```java
static double method3(int[] array) {
    int n = array.length;
    double sum = 0;

    for (int pass=100; pass >= 4; pass--) {
        for (int index=0;index < 2*n;index++) {
            for (int count=4*n;count>0;count/=2)
                sum += 1.0*array[index/2]/count;
        }
    }
    return sum;
}
```

Exercise 3: Recursive Code (6 points)

For each of the following Java functions:

(1) Describe how you will measure the problem size in terms the input parameters. For example, the input size is measured by the variable n, or the input size is measured by the length of array a.
(2) Write a recurrence relation representing its running time. Show how you obtain the recurrence.
(3) Indicate what your recurrence solves to (by looking up the recurrence in our table).

```java
// assume xs.length is a power of 2
int halvingSum(int[] xs) {
    if (xs.length == 1) return xs[0];
    else {
        int[] ys = new int[xs.length/2];
        for (int i=0;i<ys.length;i++)
            ys[i] = xs[2*i]+xs[2*i+1];
        return halvingSum(ys);
    }
}

int anotherSum(int[] xs) {
     if (xs.length == 1) return xs[0];
     else {
         int[] ys = Arrays.copyOfRange(xs, 1, xs.length);
         return xs[0]+anotherSum(ys);
     }
}

int[] prefixSum(int[] xs) {
    if (xs.length == 1) return xs;
    else {
        int n = xs.length;
        int[] left = Arrays.copyOfRange(xs, 0, n/2);
        left = prefixSum(left);
        int[] right = Arrays.copyOfRange(xs, n/2, n);
        right = prefixSum(right);

        int[] ps = new int[xs.length];
        int halfSum = left[left.length-1];
        for (int i=0;i<n/2;i++) { ps[i] = left[i]; }
        for (int i=n/2;i<n;i++) { ps[i] = right[i - n/2] + halfSum; }
        return ps;
    }
}
```

Exercise 4: HackerRank Problems (4 points)

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

You'll begin by creating an account on `hackerrank.com` if you don't have one already, so you can solve this set of problems. There are 2 problems in this set. You **must** write your solutions in Java (1.8). You will hand them in electronically on the HackerRank website.

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-2`

Exercise 5: Palindromic and Recursive (2 points)

For this task, save your code in `RPal.java` *(adapted from a problem in ACM Regionals, Greater NY 2008)*

This problem will give you more practice in writing recursive programs, in the context of solving a wacky problem.

Let $N > 0$ be an integer. We say that a list X of positive integers is a *partition* of N if the elements of X add up to exactly N. For example, each of $[1,2,4]$ and $[2,3,2]$ is a partition of 7.

As you might know already, a list is *palindromic* if it reads the same forward and backward. Of the above example partitions, $[1,2,4]$ is not palindromic, but $[2,3,2]$ is palindromic. What's more, we know that if X is palindromic, then the *first half* (precisely the first `len(X)/2` numbers) is the reverse of the last half (precisely the last `len(X)/2` numbers).

In this task, we're interested in partitions that are palindromic recursively. A partition is *recursively palindromic* if it is palindromic itself and its first half is recursively palindromic or empty. For example, there are 6 recursively palindromic partitions of 7:

```
[7], [1,5,1], [2,3,2], [1,1,3,1,1], [3,1,3], [1,1,1,1,1,1,1]
```

Your task: Implement a function `public static int countRPal(int N)` that takes as input a number N (an integer between 1 and 100, inclusive) and returns a list of all recursively palindromic partitions of N. We will only test your function with N between 1 and 100 (inclusive). As an example, calling `countRPal(7)` should return 6.

(Hint: There are 9,042 *partitions that are recursively palindromic for* $N = 99$.*)*

Performance Expectations: We expect your code to be reasonably fast. For the largest N (i.e., $N = 99$), your program should not take more than 2 seconds.

Problem 6: Sudoku (0 points)

For this task, save your code in `Sudoku.java`

Sudoku is a famous combinatorial placement puzzle. The game is played by filling a 9×9 grid with digits (i.e., the numbers $1, 2, 3, \ldots, 9$) to meet certain objectives. Aside from the standard notions of rows and columns, Sudoku has the notion of boxes. As highlighted in the figure on the right, there are *nine* 3×3 boxes that make up the grid.

5	3	4	6	7	8	9	1	2
6	7	2	1	9	5	3	4	8
1	9	8	3	4	2	5	6	7
8	5	9	7	6	1	4	2	3
4	2	6	8	5	3	7	9	1
7	1	3	9	2	4	8	5	6
9	6	1	5	3	7	2	8	4
2	8	7	4	1	9	6	3	5
3	4	5	2	8	6	1	7	9

A Sudoku grid is considered solved if all of the following are true:

(1) Every cell is filled with a number;
(2) Every row contains all of the digits from 1 through 9;
(3) Every column, likewise, contains all of the digits from 1 through 9; and
(4) Each of the nine 3×3 boxes that compose the grid also contains all of the digits from 1 through 9.

Otherwise, the grid is *not* solved. As an example, the sample grid is solved.

You are to implement a function `public static boolean sudokuSolved(int[][] grid)` that takes as input a 2d array representing a Sudoku grid and returns a Boolean indicating whether this grid has been solved. Specifically, the input `grid` is an array of length 9, where each subarray is an array of `int` of length 9. For instance, the example grid above is represented in Java as follows. On this grid, `sudokuSolved` should return `true`.

```java
int[][] ex = {{5,3,4,6,7,8,9,1,2},
              {6,7,2,1,9,5,3,4,8},
              {1,9,8,3,4,2,5,6,7},
              {8,5,9,7,6,1,4,2,3},
              {4,2,6,8,5,3,7,9,1},
              {7,1,3,9,2,4,8,5,6},
              {9,6,1,5,3,7,2,8,4},
              {2,8,7,4,1,9,6,3,5},
              {3,4,5,2,8,6,1,7,9}}
```

We expect your function to return within 1 second. Efficient code solves this task instantaneously. *(Hint: What do the* `HashSet` *class and the* `Set` *interface do?)*

Problem 7: Big O Involving Logs (0 points)

(1) Prove that $\log^2 n$ is $O(n^2)$.
(2) For any $\alpha, \beta \geq 2$, prove that $\log_\alpha n = O(\log_\beta n)$.
(3) Prove that $f(n) = 1\lg 1 + 2\lg 2 + 3\lg 3 + n\lg n$ is $\Theta(n \log n)$. For ease, you may wish to assume that

$$\lim_{n\to\infty} \frac{f(n)}{n \log n} \text{ does exist}$$

and use the following theorem:

Theorem 7.1 (Sandwich Theorem). *Let α, f, and β be $\mathbb{Z}_+ \to \mathbb{Z}_+$ functions. If there exists a constant $c_0 > 0$ such that for all $n \geq c_0$, $\alpha(n) \leq f(n) \leq \beta(n)$, then*

$$\lim_{n\to\infty} \alpha(n) \leq \lim_{n\to\infty} f(n) \leq \lim_{n\to\infty} \beta(n)$$