CSC263 – Problem Set 1

Remember to write your **full name** and **student number** prominently on your submission. To avoid suspicions of plagiarism: at the beginning of your submission, **clearly state any resources (people, print, electronic) outside of your group, the course notes, and the course staff, that you consulted**.

Remember that you are required to submit your problem sets as both LaTeX `.tex` source files and `.pdf` files. There is a 10% penalty on the assignment for failing to submit both the `.tex` and `.pdf`.

Due Jan 27, 2020, 22:00; required files: ps1.pdf, ps1.tex, pretend_sorted_array.py

Answer each question completely, always justifying your claims and reasoning. Your solution will be graded not only on correctness, but also on clarity. Answers that are technically correct that are hard to understand will not receive full marks. Mark values for each question are contained in the [square brackets].

You may work in groups of up to THREE to complete these questions.

Please see the course information sheet for the late submission policy.

1. [**10 points**] Recall this code from lecture.

```
1 Search42(L):
2   z = L.head
3   while z != None and z.key != 42:
4      z = z.next
5   return z
```

Suppose that the input list L has n (≥ 25) elements (excluding the `None` element). The list is constructed randomly as follows:

The first number in the list is picked uniformly randomly from the set of all even numbers between 2 and $2n$ (both inclusive), i.e., $\{2, 4, 6, 8, 10, \ldots, 2n\}$, the second number in the list is picked uniformly randomly from the set of all even numbers between 4 and $2n$ inclusive, i.e. $\{4, 6, 8, 10, \ldots, 2n\}$, etc. Generally, the k^{th} element in the list is picked uniformly randomly from $\{2k, 2(k+1), \ldots, 2n\}$. All choices are made independently of each other.

Let us analyze the number of times line 3 is executed. Answer the following questions in **exact form** and **not** in asymptotic notation. Show your work!

(a) (2 points) What is the **smallest number of times** line 3 is executed? What is the probability that line 3 is executed the smallest possible number of times? Justify your answer carefully: show your work and explain your calculation.

(b) (3 points) What is the **largest number of times** line 3 is executed? What is the probability that line 3 is executed the largest possible number of times? Justify your answer carefully: show your work and explain your calculation.

(c) (4 points) What is the **expected number of times** line 3 is executed? Justify your answer carefully: show your work and explain your calculation.

(d) (1 point) For what value of n is the expected number of times line 3 is executed between 13 and 17 (both inclusive)? You are allowed to use math tools such as a calculator or WolframAlpha to obtain your answer.

2. **(total 10 points)** Consider the following code for checking if the input array `A` is a max-heap:

```
  // Assume arrays are zero-indexed
  // and A.size() returns number of elements in A
CheckMaxHeap(A):
  for i from 0 to floor(A.size()/2)-1:
    if CheckMaxHeapHelper(A, i) == False
      return False
  return True

```

```
CheckMaxHeapHelper(A, i):
  if 2i+1 < A.size():
    if A[2i+1] > A[i]:
      return False
  if 2i+2 < A.size():
    if A[2i+2] > A[i]:
      return False
  return True
```

(a) (2 points total) Consider an array `A` with three elements (suppose `A` is 0-indexed). Suppose the elements of `A` are filled with a random permutation of `{1, 2, 3}`.

   i. (1 point) What is the probability that `A` is a max-heap? Briefly explain your answer.

   ii. (1 point) If we run `CheckMaxHeap(A)`, what is the expected number of times `CheckMaxHeapHelper` is called? Justify your answer carefully: show your work and explain your calculation.

(b) (8 points) Now consider an array `A` with five elements (suppose `A` is 0-indexed). Suppose the elements of `A` are filled with a random permutation of `{1, 2, 3, 4, 5}`.

   i. (2 points) What is the probability that `CheckMaxHeapHelper(A, 0)` returns `True`? Justify your answer carefully: show your work and explain your calculation.

   ii. (4 points) What is the probability that `CheckMaxHeap(A)` returns `True`? Justify your answer carefully: show your work and explain your calculation. Enumeration is not a justification.

   iii. (2 points) If we run `CheckMaxHeap(A)`, what is the expected number of times `CheckMaxHeapHelper` is called? Justify your answer carefully: show your work and explain your calculation.

Programming Question

The best way to learn a data structure or an algorithm is to code it up. In each problem set, we will have a programming exercise for which you will be asked to write some code and submit it. You may also be asked to include a write-up about your code in the PDF/TEXfile that you submit. Make sure to **maintain your academic integrity** carefully, and protect your own work. The code you submit will be checked for plagiarism. It is much better to take the hit on a lower mark than risking much worse consequences by committing an academic offence.

3. **(12 points)** Consider the following *slow* data structure that maintains a sorted array along with a *pointer* index, and permits the following commands as follows:

   - The `initialize` command is a string of the form `initialize x1 x2 ... xk`, where `x1`, `x2`, ... `xk` are integers (not necessarily in a sorted order). This command means that the array must be initialized with the collection $\{\texttt{x1}, \texttt{x2}, \ldots, \texttt{xk}\}$ in **sorted order**. Note that the command `initialize` are all separated by a single space. There will be exactly one `initialize` command that will appears as the first command in the list of commands.

     At initialization, consider that the *pointer* is at the location of the entry `x1` in the sorted array.

     e.g. `initialize 15 4 7 12` would initialize the sorted array as `A = [4, 7, 12, 15]`, and will initialize the *pointer* index to 3 since `A[3] = 15` (assume that the array is o-indexed).

   - An `insert` command is a string of the form `insert x`, where `x` is an integer. (Note the space between `insert` and `x`.) This command adds `x` to the array while maintaining a sorted order.

     e.g. executing `insert 8` on the above example implies that we must update our array to be `A = [4, 7, 8, 12, 15]`. The *pointer* remains as 3, and is *not updated*.

   - `move_pointer_left` command moves the pointer left. If the pointer is already at the first element in the array, the pointer stays at the first element.

     e.g. after executing `move_pointer_left` on the above example, the pointer would be updated to 2.

   - Similarly, `move_pointer_right` command moves the pointer right. If the pointer is already at the last non-empty location in the array, the pointer stays at its location.

     e.g. after executing `move_pointer_right` on the above example, the pointer would be updated to 3.

- *Output:* At the end of executing each command, the data structure prints the element at the current location of the pointer.

  e.g. After executing the commands in the above examples in order, the outputs would be `15 12 8 12`.

Note that the above data structure is *slow*.

In this question, you will implement a new ADT that we shall call **PretendSortedArray**. Our ADT emulates the behavior of the above data structure but in a **much faster way**.

The function `pretend_sorted_array` takes a list of commands described above that operate on the current collection of data. Your task is to process the commands in order and return the list of outputs. There are four kinds of commands: `initialize` commands, `insert` commands, `move_pointer_left` commands and `move_pointer_right` commands.

Your goal is to implement

- the `initialize` command in worst-case $O(n)$ time, where n is the number of elements in the initial collection,
- and the `insert`, `move_pointer_left`, and `move_pointer_right` commands each in worst-case $O(\log n)$ time, where n is the number of elements currently in the collection.

The list returned by `pretend_sorted_array` consists of the results, in order, from each of the commands in the list. For example, here is a sample call of `pretend_sorted_array`:

```
pretend_sorted_array(
  [ 'initialize 4 15 7 12',
    'move_pointer_left',
    'move_pointer_right',
    'insert 8',
    'insert 6',
    'move_pointer_right',
    'move_pointer_right',
    'move_pointer_right',
  ])
```

This corresponds to the following states of the sorted array and pointer:

- We initialize the array with the sorted collection of number, and initialize the pointer to the location of 4 in the array.

  `A = [4, 7, 12, 15]` and `pointer = 0`. Outputs 4
- We try to move the pointer left. Since it's already at the first location, we don't update it.

  `A = [4, 7, 12, 15]` and `pointer = 0`. Outputs 4
- We move the pointer to the right.

  `A = [4, 7, 12, 15]` and `pointer = 1`. Outputs 7
- We insert 8 into the sorted array.

  `A = [4, 7, 8, 12, 15]` and `pointer = 1`. Outputs 7
- We insert 6 into the sorted array.

  `A = [4, 6, 7, 8, 12, 15]` and `pointer = 1`. Outputs 6
- We move the pointer right.

  `A = [4, 6, 7, 8, 12, 15]` and `pointer = 2`. Outputs 7
- We move the pointer right.

  `A = [4, 6, 7, 8, 12, 15]` and `pointer = 3`. Outputs 8
- We move the pointer right.

  `A = [4, 6, 7, 8, 12, 15]` and `pointer = 4`. Outputs 12

`pretend_sorted_array` returns `[4, 4, 7, 7, 6, 7, 8, 12]`.

Requirements:

- Your code must be written in Python 3, and the filename must be `pretend_sorted_array.py`.
- We will grade only the `pretend_sorted_array` function; please do not change its signature in the starter code. include as many helper functions as you wish.

Write-up: in your `ps1.pdf/ps1.tex` files, briefly and informally argue why your code is correct, and has the desired runtime.