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 January 28, 2019, 22:00; required files: ps1sol.pdf, ps1sol.tex, moving_min.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.

Authors: Junwen Shen(1004299190), Mengning Yang(1002437552), Jianhao Tian(1001354465)

1. **[4]** 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
```

Rather than supposing that each key in the list is an integer chosen uniformly at random from 1 to 100, let's instead suppose that the list length n is at least 42 and that the list keys are a random permutation of $1, 2, 3, \ldots, n$.

Under these new assumptions, what is the expected number of times that line 3 is executed?

Give your answer in **exact form**, i.e., **not** in asymptotic notations. Show your work!

Solutions:
The expected number: $E[t_n] = \sum_{t=1}^{n+1} t * Pr(t_n = t)$ and $n \geq 42$,
since $Pr(t_n = t) = \begin{cases} (1-\frac{1}{n})^{t-1} * \frac{1}{n}, & 1 \leq t \leq n \\ (1-\frac{1}{n}), & t = n+1 \end{cases}$,
$E[t_n] = \sum_{t=1}^{n} t * Pr(t) + (n+1)(1-\frac{1}{n})^n$
$= \frac{1}{n} * \sum_{t=1}^{n} t(1-\frac{1}{n})^{t-1} + (n+1)(1-\frac{1}{n})^n$
Let $S = \sum_{t=1}^{n} t(1-\frac{1}{n})^{t-1} \cdots\cdots 1$
then $(1-\frac{1}{n}) * S = \sum_{t=1}^{n} t * (1-\frac{1}{n}))^t \cdots\cdots 2$
and subtract equation 1 by equation 2,
$\frac{1}{n} * S = \sum_{t=1}^{n} t(1-\frac{1}{n})^{t-1} - \sum_{t=1}^{n} t(1-\frac{1}{n})^t$
$= \sum_{t=0}^{n-1} -n(1-\frac{1}{n})^n$
$= \frac{1-(1-\frac{1}{n})^n}{1-(1-\frac{1}{n})} - n(1-\frac{1}{n})^n$
$= n - 2n(1-\frac{1}{n})^n$
Therefore, the expected number $E[t_n] = n - 2n(1-\frac{1}{n})^n + (n+1)(1-\frac{1}{n})^n = n + (1-n)(1-\frac{1}{n})^n$

2. **[12]** Consider the following algorithm that describes the procedure of a casino game called "`Survive263`". The index of the array A starts at 0. Let n denote the length of A.

```
1    Survive263(A):
2        '''
3        Pre: A is a list of integers, len(A) > 263, and it is generated
4             according to the distribution specified below.
5        '''
6        winnings = -5.00   # the player pays 5 dollars for each play
```

```
7       for i from n-1 downto 0:
8           winnings = winnings + 0.01  # winning 1 cent
9           if A[i] == 263:
10              print("Boom! Game Over.")
11              return winnings
12      print("You survived!")
13      return winnings
```

The input array A is generated in the following specific way: for $A[0]$ we pick an integer from $\{0, 1\}$ uniformly at random; for $A[1]$ we pick an integer from $\{0, 1, 2\}$ uniformly at random; for $A[2]$ we pick an integer from $\{0, 1, 2, 3\}$ uniformly at random, etc. That is, for $A[i]$ we pick an integer from $\{0, \ldots, i + 1\}$ uniformly at random. All choices are independent from each other. Now, let's analyse the player's expected winnings from the game by answering the following questions. All your answers should be in **exact form**, i.e., **not** in asymptotic notations.

(a) Consider the case where the player **loses the most** (i.e., minimum winnings), what is the return value of `Survive263` in this case? What is the probability that this case occurs? Justify your answer carefully: show your work and explain your calculation.
Answer:
The return value where the player loses the most is -4.99. This happens at the first iteration of the loop. The probability that this case occurs is $\frac{1}{n+2}$, because we know that n > 263 and i $\geq$ 262, for A[i] we pick an integer from $\{0, \ldots, i + 1\}$ uniformly at random, so there are n+2 integers to be picked from.

(b) Consider the case where the player **wins the most** (i.e., maximum winnings), what is the return value of `Survive263` in this case? What is the probability that this case occurs? Justify your answer carefully: show your work and explain your calculation.
Answer:
The return value where the player wins the most is -5.00+0.01n. This happens when the loop runs n times and has successfully survived the game.
The probability that this case occurs:
Case 1: When $i \geq 262$
We already know from (a), P(A[i]=263)=$\frac{1}{n+2}$
So, $P(A[i] \neq 263) = 1 - (\frac{1}{n+2}) = 1 - \frac{1}{n+2} = \frac{n+1}{n+2}$
What we want is to find the probability that very iteration of the loop $P(A[i] \neq 263)$, and because every event is independent and identically distributed, we multiply the probability of each iteration.
$\prod_{i=262}^{n-1} \frac{n+1}{n+2} = \frac{263}{n+1}$

Case 2: When $i < 262$
$P(A[i] \neq 263) = 1$, because 263 is impossible to show up again, once the player has passed this point, the game is not going to be over until the loop terminates. What we want is to find the probability that very iteration of the loop $P(A[i] \neq 263)$, and because every event is independent and identically distributed, we multiply the probability of each iteration.
$\prod_{i=0}^{i=261} P(A[i] \neq 263) = 1$
Finally, multiply $\frac{263}{n+1}$ and $1 = \frac{263}{n+1}$.

(c) Now consider the **average case**, what is the **expected value** of the winnings of a player (i.e., the expected return value of `Survive263`) according to the input distribution specified above? Justify your answer carefully: show your work and explain your calculation.
Answer:
The expected value of winning is E(-5.00+0.01m) = -5.00+0.01E(m) where m is the number of loop iterations.
The goal is to find E(m), the average number of times the loop runs.
Suppose the loop runs n times, there are two cases:
1) the loop runs n times and survives 263
2) the loop runs n-1 times and survives 263, but the loop meets 263 on the nth iteration and the game ends
In order to end the average number of times the loop runs, we have the equation E(m) = $\sum_{k=1}^{n} k * P(k)$ where k is the kth iteration of the loop.

When k = 1, i = n-1. When k = 2, i = n-2 ... So i = n - k.
When i = n-1, P(only one iteration of the loop will run) = $\frac{1}{n-1+2}$ by the results we got from part (a)
When i = n-2, P(two iterations of the loop will run) = $\frac{1}{n-1+2} * \frac{1}{n-2+2}$
(the first iteration of the loop does not meet 263, the second iteration of the loop meets 263)
When i = n-3, P(three iterations of the loop will run) = $\frac{n}{n-1+2} * \frac{n-1}{n-2+2} * \frac{1}{n-3+2}$
(the first and second iteration of the loop does not meet 263, the third iteration of the loop meets 263)
and so on...
After simplification, they all equals to $\frac{1}{n+1}$ when n-k = i ≥ 262 and $k \leq n - 262$. So we can conclude that if the loop runs k times and $k \leq n - 262$, the probability is $\frac{1}{n+1}$.
Another case is that after $(n - 261)^{th}$ iterations, if the game is still not over yet, then the loop will never meet 263 and the loop will run until i = 0. So the probability of the loop terminates when k is between n-261 to n-1 is 0.
And finally, the loop runs n-1 times and survives 263, but the loop meets 263 on the nth iteration and the game ends, the probability that the loop meets 263 at the last iteration is $\frac{263}{n+1}$, which we got from part (2).
So we come up with the following equation:
$E(m) = \sum_{k=1}^{k=n-262} k * \frac{1}{n+1} + \sum_{k=n-262}^{n-1} k * 0 + nP(n)$
$E(m) = \sum_{k=1}^{k=n-262} k * \frac{1}{n+1} + \sum_{k=n-262}^{n-1} k * 0 + n\frac{263}{n+1}$
$E(m) = \frac{1}{n+1}\frac{(1+n-262)(n-262)}{2} + 0 + \frac{263n}{n+1}$
$E(m) = \frac{n^2+3n+68382}{2(n+1)}$
Therefore, $E(winnings) = -5.00 + 0.01\frac{n^2+3n+68382}{2(n+1)}$

(d) Suppose that you are the owner of the casino and that you want to determine a length of the input list A so that the expected winnings of a player is between -1.01 and -0.99 dollars (so that the casino is expected to make about 1 dollar from each play). What value could be picked for the length of A? You are allowed to use math tools such as a calculator or WolframAlpha to get your answer.
Answer:
$E(winnings) = -1.01 = -5.00 + 0.01\frac{n^2+3n+68382}{2(n+1)}$
$401 = \frac{n^2+3n+68382}{2(n+1)}$, n = 96.15 or 702.85. Because len(A) has to be greater than 263, so the length of the input list is 702.85.
$E(winnings) = -0.99 = -5.00 + 0.01\frac{n^2+3n+68382}{2(n+1)}$
$399 = \frac{n^2+3n+68382}{2(n+1)}$, n = 96.79 or 698.2. Because len(A) has to be greater than 263, so the length of the input list is 698.2.
Therefore, the length of the input list would be between 699 and 702.

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]** In this question, you will solve the **Moving Minimum Problem**. The function `solve_moving_min` takes a list of commands that operate on the current collection of data; your task is to process the commands in order and return the required list of results. There are two kinds of commands: `insert` commands and `get_min` commands.

   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 collection.

   A `get_min` command is simply the string `get_min`. The first `get_min` command results in the smallest element currently in the collection; the next `get_min` command results in the second-smallest element currently in the collection; and so on. That is, the `j`th `get_min` command results in the `j`th-smallest element in the collection at the time of the command. You can assume that the collection has at least `j` elements at the time of the `j`th `get_min` command.

   Your goal is to implement `insert` and `get_min` each in $O(\lg n)$ time, where n is the number of elements currently in the collection. The list returned by `solve_moving_min` consists of the results, in order, from each `get_min` command.

Let's go through an example. Here is a sample call of `solve_moving_min`:

```
solve_moving_min(
  ['insert 10',
   'get_min',
   'insert 5',
   'insert 2',
   'insert 50',
   'get_min',
   'get_min',
   'insert -5'
  ])
```

This corresponds to the following steps:

- The collection begins empty, with no elements.
- We insert 10. The collection contains just the integer 10.
- We then have our first `get_min` command. The result is the smallest element currently in the collection, which is 10.
- We insert 5. The collection now contains 10 and 5.
- We insert 2. The collection now contains 10, 5, and 2.
- We insert 50. The collection now contains 10, 5, 2, and 50.
- Now we have our second `get_min` command. The result is the second-smallest element currently in the collection, which is 5.
- Now we have our third `get_min` command. The result is the third-smallest element currently in the collection, which is 10.
- We insert -5. The collection now contains 10, 5, 2, 50, and -5.

`solve_moving_min` returns `[10, 5, 10]` (the three values produced by the `get_min` commands).

Requirements:

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

Write-up: in your `ps1sol.pdf/ps1sol.tex` files, briefly and informally argue why your code is correct, and has the desired runtime.
Solution:

```
1  def insert(sorted_list, number):
2    '''
3    Pre: sorted_list is a sorted list of numbers
4    Post: return a sorted list with number in it
5    '''
6    if len(sorted_list) == 0:
7      return [number]
8    if len(sorted_list) == 1:
9      if sorted_list[0] > number:
10       return [number, sorted_list[0]]
11     elif sorted_list[0] < number:
12       return [sorted_list[0], number]
13     else:
14       return [sorted_list[0], number]
15   else:
```

```
    mid = len(sorted_list) // 2
    if number > sorted_list[:mid][-1]:
        return sorted_list[:mid] + insert(sorted_list[mid:], number)
    else:
      return insert(sorted_list[:mid], number) + sorted_list[mid:]
```

```
def get_min(sorted_list, count):
  '''
  Pre: sorted_list is a sorted list
  Post: return the count-th number in sorted_list
  '''
  return sorted_list[count]
```

```
def solve_moving_min(commands):
  '''
  Pre: commands is a list of commands
  Post: return list of get_min results
  '''
  sorted_list = []
  final_list = []
  count = 0
  for command in commands:
    if command == "get_min":
      final_list.append(get_min(sorted_list, count))
      count += 1
    else:
      number = int(command.split()[1])
      sorted_list = insert(sorted_list, number)
  return final_list
```

Correctness:
For function `insert`, when length of list is 0 or 1, the return of insert will satisfies post-condition. for recursive path, if number is bigger than the last element of the first half sorted list, then insert number into the second half of sorted list, and the function's return will satisfies the post-condition. If number is smaller than the last element of the first half sorted list, then insert number into the first half of sorted list, and the function's return will also satisfies the post-condition.

For `get_min` the function returns the count-th samlles element in sorted list, which satisfies the post-condition.

For function `solve_moving_min`, the function creats a sorted list as the insert command keep append numbers into the collection, and get min well return the count-th smallest element in the sorted list where count is the times that insert has been called.

Run time:
For function `insert` , the function uses constant time for the base case, in the recursive path, the function divide the input into half size, so the run time for insert is $T(n) = T(n/2) + c$, by master theorem, the upper-bond run time for insert is $O(logn)$.

For function `get_min`, the function simply returns the count-th element in the list, so the upper-bond run time for this function should be constant.

Q2 From Dan, "My strategy is to use two heaps: a min-heap (containing large elements) and a max-heap (containing small elements).
Keep the max-heap at size n, where n is the number of get_min commands that have been processed so far.
This way, the root of the min-heap is the value to return for the next get_min command."

insert进min heap，如果call了get_min(),就把那个值放进max_heap。再下一次insert进来的值，先和max_heap的root比较谁大，如果比max heap的root大，就append进min heap，bubble up, 如果比max heap的root小，就append进max heap(他肯定是前n小的element)，，下一次get_min()也不会叫到他。并把max heap现在的root重新放回min heap，因为他又成了下一个get_min()的candidate (因为之前来了个比他小的) max heap里永远只存前n小的值，min heap里永远只存从第n+1小的值开始。max heap里的值是封存着不动的，因为你下一次call get_min()的值只会是min heap的root。