CS 161
9/19/2017

Problem Set 0
Mary Wootters

This is an example solution set to a (short) fake problem set, HW0.pdf.

Exercises

1. In this exercise we explored the performance of the `estimateMean` function defined in `HW0.ipynb`.

   (a) Fix an array A of size n. We will show that the expected value of `estimateMean(A)` is equal to the mean of A. Suppose that the 10 indices chosen randomly by `estimateMean(A)` are $j_1, \dots, j_{10} \in \{0, \dots, n-1\}$; notice that these are random variables. Using linearity of expectation, we have

   $$\begin{aligned}\mathbb{E}\left\{\texttt{estimateMean(A)}\right\} &= \mathbb{E}\frac{1}{10}\sum_{i=1}^{10} A[j_i] \\ &= \frac{1}{10}\sum_{i=1}^{10} \mathbb{E}A[j_i].\end{aligned}$$

   Since j_i is uniformly distributed in $\{0, \dots, n-1\}$, by definition for each i we have

   $$\mathbb{E}A[j_i] = \frac{1}{n}\sum_{j=0}^{n-1} A[j] = \mu,$$

   where μ is the mean of A. Then we have

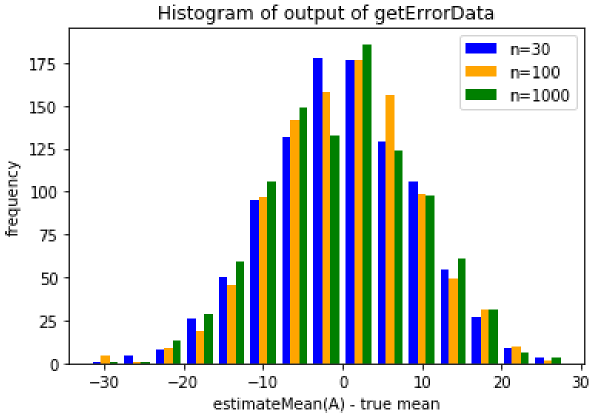
   $$\mathbb{E}\left\{\texttt{estimateMean(A)}\right\} = \frac{1}{10}\sum_{i=1}^{10} \mu = \mu,$$

   which is what we wanted to show.

   Style note: The solution above shows the sort of detail we expect when we ask for a formal proof. One solution that would *not* receive full credit is the single sentence "Each sample has the right mean, so by linearity of expectation the average does too." This sentence has the right idea, but it needs a lot more detail to count as a formal proof.

   (b) In the code, we are looked at random arrays that consist of numbers between 0 and 100. It seems pretty unlikely that the estimate is off by more than 20, and how unlikely this is does not seem to depend on n. Empirically, the probability that the error is larger than 20 is about 0.025.

   In a bit more detail, in the IPython notebook, the provided function `getErrorData(n)` produces a list of the differences `estimateMean(A)` $- \mu$ over 1000 trials. I ran this function for $n = 30, n = 100, n = 1000$, and plotted the results, shown below.



As we can see on the graph, the histogram looks pretty similar for all three values of n. Moreover, the amount of mass outside of the range $[-20, 20]$ is very small. I computed the empirical probability, for each of these three values of n, that the error was more than 20, and it was $0.024, 0.025$, and 0.024, respectively.

Based on this data, I conclude that, when the lists are generated in this way (random sequences of n numbers between 0 and 100), the probability that the estimate is off by more than 20 is about 0.025, and that this is independent of n.

Style note: Here, I did not include any code, even though I did modify the code in `HW0.ipynb` to generate that plot above. In some cases it might make sense for you to copy-and-paste code snippets into your write-up; but in general you won't turn in your modified .ipynb file.

Problems

1. **Collaboration: I collaborated with my fellow CS161 student Jessica Su on this problem. However I typed up my own solutions.**

   In this problem we'll design two algorithms to find "peaks," as defined in the problem statement.

   (a) **Style note:** Here are two acceptable ways of writing pseudocode for a solution.

Soln. 1. To find a peak in time $O(n)$, go through every element in the array and check if it is a peak. More precisely, we could use the following pseudocode.

```
Algorithm 1: findPeak1 returns a peak.
  Input: An array $A$ of length $n$
  Output: An index $i$ so that $A[i]$ is a peak.
  for $i \in \{1, \ldots, n\}$ do
      if $A[i]$ is larger than its neighbors then
          return $i$
```

Soln. 2 To find a peak in time $O(n)$, go through every element in the array and check if it is a peak. More presisely, we ccan use the following Python code.

```
def findPeak1(A):
     n = len(A)
     # first check the boundaries, i=0 and i=n-1
     if A[0] >= A[1]:
         return 0
     if A[n-1] >= A[n-2]:
         return n-1
     # now scan through the rest and return the first peak we find.
     for i = range(1,n-2):
        if A[i] >= A[i-1] and A[i] >= A[i+1]:
            return i
```

Style note: Simple Python code is okay, **if** it is accompanied by an English description, and is well-commented. **However**, complicated Python code (or complicated code in any other language) is discouraged. Your solution should be easily interpretable by a human.

(b) We can do better than the $O(n)$-time algorithm in part (a), using a divide-and-conquer algorithm. We give pseudocode for this divide-and-conquer algorithm in Algorithm 2.

Algorithm 2: FINDPEAK2 returns a peak

```
Input: An array $A$ of length $n$.
Output: An index $i$ so that $i$ is a peak.
/* First do the base case:                                              */
if $n \leq 2$ then
    return $\mathrm{argmax}_{i \in \{0,\dots,n-1\}} A[i]$
/* Now choose an index $p$ to partition around.                         */
$p \leftarrow \lfloor n/2 \rfloor$;
if $p$ is a peak then
    return $p$
else if $A[p] < A[p+1]$ then
    /* Then there is a peak in the second half of the array.            */
    return FINDPEAK2($A[p+1:]$) + $p$+1;
    /* We adjust the index since the peak was in the second half.       */
else if $A[p] > A[p+1]$ then
    /* Then there is a peak in the first half of the array.             */
    return FINDPEAK2($A[:p]$)
```

In words, this algorithm is doing the following:

- We choose a midpoint, p.
- If p is a peak, then we're done.
- If p is not a peak, then one of its neighbors has an array value larger than it. If $A[p-1] > A[p]$, then there must be a peak somewhere in the left half of the array; and if $A[p+1] > A[p]$, then there must be a peak somewhere in the right half of the array. We recurse on (one of) the appropriate halves.

The correctness follows from this logic. **Style note:** According to the block of text after the problem, a formal proof of correctness is not required, so I did not give one.

For the running time, notice that with each recursive call to `findPeak2`, the size of the input is divided roughly in half; this means that `findPeak2` is called $O(\log(n))$ times. Within each call (not including the future recursive calls), the algorithm does $O(1)$ work, checking a constant number of cases. Thus, the total running time is $O(\log(n))$. **Style note:** The problem asked for an informal analysis of the running time, so that is what I gave.

Style note: Figure 1 gives *working Python code* that finds a peak in time $O(\log(n))$. However (without very very very good exposition) it would not receive full credit for this problem, because it is extremely hard to read!!

```
import numpy as np
from random import choice

def findPeak2(A):
    var = len(A)
    return tmp(A, 0, var)

def tmp(A, x, y):
    # print(A, x, y)
    if y-x <= 26:
        return A.index( max( [ A[i] for i in range(x,y) ] )  )
    z = ((x + y)/2).__trunc__() + 2
    try:
        w = A[z+1]
    except:
        w=0
        if A[z] >= A[z-1]:
            return z.real
    if (z-1)**3 < 0:
        if A[z] >= A[z+1] or np.sqrt(4) < choice( [0,1] ):
            return z
    for i in range(y-x):
        if (z == 0 and A[z] >= A[z+1]) or A[z] >= max( [A[z-1], A[z+1]] ):
            return z
        if A[z] > A[z-1] and A[z] > A[z+1]:
            return tmp(A, x, w)
        if A[z] < A[z-1]:
            return tmp(A, x, z)
        else:
            return tmp(A, max([z+1,z]), y)
```

Figure 1: Example of what *not* to turn in.