CSEP 521, Winter 2021: Homework 1

Krishan Subudhi (ksubudhi@uw.edu) - 2040900

January 12, 2021

Problem 1

Given,
probability of heads = p
probability of tails = q,
$q = 1 - p$ and $p \neq q$

Q: How can you use this coin to generate an unbiased coin flip

A: We need a trick to turn the biased coin flip into an unbiased one. This is not possible with just one coin flip. Hence we need to come up with events with multiple coin flips which will give an unbiased result.

If we flip the coin twice, there are 4 possible outcomes.

$$HH, TT, HT, TH$$

Out of these 4 outcomes

$$P(HT) = P(TH) = pq$$
$$P(HT|HT \text{ or } TH) = P(TH|HT \text{ or } TH) = 1/2$$

This will result in an unbiased coin flip.

So if X is a random variable representing number of coin flips to generate an unbiased coin flip, then

$$\begin{aligned} E[X] =& P(HT \text{ or } TH) * 2 \\ &+ P(HH) * (2 + E[X]) \\ &+ P(TT) * (2 + E[X]) \end{aligned} \quad [1.1]$$

1.1 states that if outcome of two consecutive coin flips are not same, then the value of X = 2 else calculate X by repeating the same process again.

Hence

$$\begin{aligned} E[X] &= (2 * pq) * 2 + p^2(2 + E) + q^2(2 + E) \\ &= 4p(1 - p) + p^2(2 + E) + (1 - p)^2(2 + E) \\ &= 2 + 2p^2E - 2pE + E \end{aligned} \quad [1.2]$$

$$\implies E = \frac{1}{p(1 - p)}$$

$\frac{1}{p(1-p)}$ is the expected number of coin flips to generate an unbiased coin flip

Problem 2

```
public static int[] Permutation(int n, Random rand) {
    int[] arr = IdentityPermutation(n);
    for (int i = 1; i < n; i++) {
        int j = rand.Next(0, n);    // Random number in range 0..n-1
        int temp = arr[i];          // Swap arr[i], arr[j]
        arr[i] = arr[j];
        arr[j] = temp;
    }
    return arr;
}
```

Q: Show that this algorithm does not generate perfectly uniform random permutations.

A: The inner loop , which runs n-1 times, swaps the element at index i with an element randomly chosen from all n indexes.

Hence, number of possible ways the algorithm generates a permutation = n^{n-1}

As per definition, n items can be arranged among themselves in $n!$ number of ways.

If the above algorithm had produced uniform random permutations, the algorithm should give equal probability to all the possible permutations.

i.e.

$$n^{n-1} = k(n!) \quad \text{, for } n \in \mathbb{Z}+ \quad \text{, where } k = \text{ positive integer}$$

i.e. $n!$ must be a factor of n^{n-1} for the algorithm to give equal probability to all outcomes.

However, this is not true for all values of n.

$$n^{n-1} = n * n * \cdots * n \quad , n - 1 \text{ times}$$
$$n! = 1 * 2 * \cdots * (n-1) * n$$

Since n is not divisible by $n - 1$, for all $n > 2$ where $n \in \mathbb{Z}+$

$$n^{n-1} \neq k(n!) \quad \text{, for all } n \in \mathbb{Z}+$$

This means the algorithm does not generate all permutations with same probability for all values of n. Some of the permutations will have higher probability than $\frac{1}{n!}$ and some will have lower probability than $\frac{1}{n!}$

Hence this algorithm does not generate perfectly uniform random permutations.

Problem 3

Q: a) Give a scheme to generate random numbers in the range 1, . . . , 6 that is as efficient as possible (in terms of the random bits). What is the expected number of bits you use?

A: To generate a random number from $1, ..., 6$ we will need at least 3 random bits. On $000, 001, 010, 011, 100, 101$ we will return $1, 2, 3, 4, 5, 6$ respectively and on $110, 111$ we start over.

$$
\begin{aligned}
E &= \frac{6}{8} * 3 + \frac{2}{8}(E + 3) \\
\implies 4E &= 12 + E \\
\implies E &= 4
\end{aligned}
$$

Hence, 4 bits will be used on an average to generate random bits from $1, ..., 6$ efficiently.

Q: b) Give a scheme to generate random numbers in the range 1, . . . , 9 that is as efficient as possible (in terms of the random bits). What is the expected number of bits you use?

A: To generate a random number from $1, ..., 9$ we will need at least 4 random bits.

4 random bits can generate 16 different outcomes with uniform probability - out of which 9 can be used to represent numbers from $1, ..., 9$. For other 7 outcomes the experiment needs to start over.

Hence,

$$
\begin{aligned}
E &= \frac{9}{16} * 4 + \frac{7}{16}(E + 4) \\
\implies 16E &= 36 + 7E + 28 \\
\implies E &= \frac{64}{9} = 7\frac{1}{9}
\end{aligned}
$$

We will use $7\frac{1}{9}$ expected number of bits.

Problem 4

Partitions are the number of times a new pivot is selected. In other-words, partitions are the number of recursive operation done to find the median.

expected partitions = total partitions/iterations
expected comparisons =total comparisons/iterations/n
Number of iterations per each row = 100

Table 1: Results of quick select

n	algorithm	expected partitions	expected comparisons
10	Quick select random pivot	3.90	1.95 n
	Quick select Median of 3	3.25	1.61 n
	Quick select Median of 5	3.08	1.48 n
100	Quick select random pivot	7.99	3.13 n
	Quick select Median of 3	6.57	2.48 n
	Quick select Median of 5	6.21	2.26 n
1000	Quick select random pivot	13.16	3.39 n
	Quick select Median of 3	10.75	2.70 n
	Quick select Median of 5	10.13	2.45 n
10000	Quick select random pivot	18.02	3.50 n
	Quick select Median of 3	14.98	2.78 n
	Quick select Median of 5	13.64	2.58 n
100000	Quick select random pivot	24.0	3.42 n
	Quick select Median of 3	17.7	2.68 n
	Quick select Median of 5	17.3	2.49 n
1000000	Quick select random pivot	27.4	3.18 n
	Quick select Median of 3	23.7	2.75 n
	Quick select Median of 5	21.0	2.33 n
10000000	Quick select random pivot	33.6	3.54 n
	Quick select Median of 3	27.1	2.88 n
	Quick select Median of 5	24.9	2.48 n

Note: I have used python's inbuilt sorting algorithm to select the pivot for median of 3 and median of 5. Since 3 and 5 are very small values, this table ignores the comparisons to select pivot and only considers comparisons while partitioning the list around the pivot.

4.1 Summary

- The expected number of comparison is a constant multiple of the number of elements in the list. This mean quick select with random pivot has $O(n)$ expected run time.
- Algorithms where the pivot is selected randomly has higher expected comparisons compared to the algorithms where pivot is calculated from median of 3 or 5. This is because the pivot has higher chance of being towards the middle in the later algorithms resulting in better average case.

- Median of 5 outperforms median of 3 as expected. This is also because of a better expected pivot.

4.2 Source code

```
import random
import pandas as pd
import collections
class QuickSelect:

    def kth_largest(self, data :list, k: int):
        self.data = data
        self.comparison_count = 0
        self.partition_count = 0
        assert k<=len(self.data) and k>0, 'K out of range'
        element =  self._kth_largest(k, 0, len(data)-1)
        return element

    def _kth_largest(self, k: int, start : int, end : int):
        """
        Given a list, find it's kth largest element using Quick select algorithm
        """
        # 1. select a random pivot
        pivotIndex = self.get_pivot_index(start, end)
        pivotIndex = self.partition(pivotIndex, start, end)
        if end-pivotIndex >=k:
            return self._kth_largest(k, pivotIndex+1, end)

        elif end-pivotIndex+1 == k:
            return self.data[pivotIndex]
        else:
            return self._kth_largest(k-1-(end-pivotIndex), start , pivotIndex-1)

    def partition(self, pivotIndex, start, end):
        self.partition_count += 1
        pivotValue = self.data[pivotIndex]
        # Move pivot to right
        self.swap(end , pivotIndex)
        pivotIndex = end
        leftIndex = start
        # Loop till end-1.
        for i in range(start, end):
            self.comparison_count += 1
            if data[i] < pivotValue:
                self.swap(leftIndex, i)
                leftIndex += 1
        self.swap(leftIndex, pivotIndex)
        return leftIndex

    def swap(self, i , j):
        t = self.data[j]
        self.data[j] = self.data[i]
        self.data[i] = t

    def get_pivot_index(self,start, end):
        randomIndex = random.randint(start,end)
```

```
        return randomIndex

    def __str__(self):
        return 'Quick select random pivot'

class QuickSelectMedian(QuickSelect):
    def __init__(self,median_of = 3):
        self.median_of = median_of
    def get_pivot_index(self, start, end):
        randomIndex = [random.randint(start,end) for _ in range(self.median_of)]
        # Use inbuilt sort
        randomIndex = sorted(randomIndex, key = lambda index: self.data[index])
        median = randomIndex[self.median_of//2]
        return median

    def __str__(self):
        return f'Quick select Median of {self.median_of}'

if __name__ == '__main__':
    import sys
    algos = [QuickSelect(), QuickSelectMedian(3), QuickSelectMedian(5)]
    iterations  =10
    results = []
    for n in [100000]:
        for q in algos:
            random.seed(47)
            comparisons = 0
            partitions = 0
            for _ in range(iterations):
                data = [i for i in range(1,n+1)]
                k = len(data)//2+1 #median
                element = q.kth_largest(data, k)
                comparisons += q.comparison_count
                partitions += q.partition_count

            results.append(pd.Series({
                'n':n,
                'algorithm':str(q),
                'expected partitions':partitions/iterations,
                'expected comparisons' :f'{comparisons/iterations/n:.2f} n'
            }))
    df = pd.DataFrame(results).set_index(['n','algorithm'])
    print(df)
    print(df.to_latex(multirow = True, multicolumn_format = 'c'))
```