CSEP 521, Winter 2021: Homework 2

Krishan Subudhi (ksubudhi@uw.edu) - 2040900

January 21, 2021

Problem 1

Number of Office workers = n
Total desks = n

Management assigns each worker to a random desk, without the requirement that only one person is assigned to a desk.

$$\begin{aligned} w_i &= \texttt{worker}_i \\ d_j &= \texttt{desk}_j \\ P(w_i, d_j) &= \frac{1}{n} \quad \text{... probability that } w_i \text{ is assigned to } d_j \end{aligned}$$

a) *What is the expected number of people getting their original desk back?* let X_i be a random variable such that

$$X_i = \begin{cases} 1, & \text{person } i \text{ gets his/her desk back} \\ 0, & \text{otherwise} \end{cases}$$

$$\begin{aligned} & E(\text{number of people getting their original desk back}) \\ &= E(\sum_{i=1}^{n} X_i) \\ &= \sum_{i=1}^{n} E(X_i) \quad \text{... linearity of expectation} \\ &= n * E(X_i) \\ &= n * (1 * P(w_i, d_i) + 0) \\ &= n * \frac{1}{n} \\ &= 1 \end{aligned}$$

Expected number of people getting their original desk back = 1

b) *What is the probability of having zero people assigned to a particular desk?*

for a particular desk d_j,

$$
\begin{aligned}
&P(\text{zero people assigned to } d_j) \\
&= \prod_{i=1}^{n} P(w_i \text{ not assigned to } d_j) \\
&= \prod_{i=1}^{n} 1 - P(w_i, d_j) \\
&= \prod_{i=1}^{n} \left(1 - \frac{1}{n}\right) \\
&= \left(\frac{n-1}{n}\right)^n
\end{aligned}
$$

c) *What is the probability of having exactly one person assigned to a particular desk?*

for a particular desk d_j,

$$
\begin{aligned}
&P(\text{exactly one person assigned } d_j) \\
&= \sum_{i=1}^{n} P(\text{only } w_i \text{ is assigned to } d_j) \qquad \textit{... events for each i are mutually exclusive} \\
&= \sum_{i=1}^{n} P(w_i, d_j) \prod_{k=1, k \neq j}^{n} (1 - P(w_i, d_j)) \\
&= \sum_{i=1}^{n} \frac{1}{n} * \left(\frac{n-1}{n}\right)^{n-1} \\
&= n * \frac{1}{n} * \left(\frac{n-1}{n}\right)^{n-1} \\
&= \left(\frac{n-1}{n}\right)^{n-1}
\end{aligned}
$$

d) What happens to the probabilities from parts b and c and n gets large?

part b

$$
\begin{aligned}
P &= \lim_{n \to \infty} \left(\frac{n-1}{n}\right)^n \\
&= \lim_{n \to \infty} \left(1 - \frac{1}{n}\right)^n \\
&= \frac{1}{e}
\end{aligned}
$$

part c

$$
\begin{aligned}
&\lim_{n\to\infty}\left(\frac{n-1}{n}\right)^{n-1} \\
&= \lim_{n\to\infty}\left(1-\frac{1}{n}\right)^{n-1} \\
&= \lim_{n\to\infty}\left(1-\frac{1}{n}\right)^{n} * \lim_{n\to\infty}\left(1-\frac{1}{n}\right)^{-1} \\
\\
&= \frac{1}{e} * 1 \\
&= \frac{1}{e}
\end{aligned}
$$

e) *Q. What is the probability that no two people are assigned to the same desk?*

Since there are only n desks for n people, each person gets one unique desk.

Number of ways everyone can get a unique desk = $n!$

Total number of ways everyone can get a desk = n^n

$$P = \frac{n!}{n^n}$$

Problem 2

Show that if the pivot is always the median, then the selection algorithm makes (about) 2n comparisons. find the maximum element.

For this problem, I am selecting the median during each pivot selection step. Using that median as pivot, quick select is applied to find the max element.

3 algorithms were used for median selection.

1. select the middle index (only works for identity permutation)
2. Python inbuilt sort (mix of merge and selection sort) and then select the middle element.
3. find median using quick select with random pivot

Table 1: Results of quick select to find max

n	algorithm	exp partitions	exp comparisons	exp runtime
1000	Quick select random pivot	7.2	2.17 n	0.0029 s
	Quick select middle index	10.0	1.98 n	0.0060 s
	Quick select Median from sorting	10.0	1.98 n	0.0100 s
	Quick select Median from quick select	9.0	1.98 n	0.0125 s
10000	Quick select random pivot	9.9	2.15 n	0.0165 s
	Quick select middle index	14.0	2.00 n	0.0158 s
	Quick select Median from sorting	14.0	2.00 n	0.0184 s
	Quick select Median from quick select	13.0	2.00 n	0.0586 s
100000	Quick select random pivot	11.9	2.63 n	0.2300 s
	Quick select middle index	17.0	2.00 n	0.1939 s
	Quick select Median from sorting	17.0	2.00 n	0.2748 s
	Quick select Median from quick select	16.0	2.00 n	0.8080 s
1000000	Quick select random pivot	13.7	1.83 n	1.7610 s
	Quick select middle index	20.0	2.00 n	0.9028 s
	Quick select Median from sorting	20.0	2.00 n	1.1352 s
	Quick select Median from quick select	19.0	2.00 n	3.8964 s

if the pivot is always the median (last 3 rows for each n), then the selection algorithm makes (about) $2n$ comparisons.

Note: The median selection algorithm was not the focus of the assignment. Hence comparisons done to find the median are not included in the result. But the run time includes time taken to select median at each stage. python inbuilt quick sort is faster because it's implemented in C. So even though it's $n(logn)$ run time at each stage, it's faster than my quick select algorithm which is written in pure python.

The algorithm uses identity permutation as the data but also tested for non identity permutations

```
import random
import pandas as pd
import collections
```

```python
class QuickSelect:
    def __init__(self):
        self.comparison_count = 0
        self.partition_count = 0

    def kth_largest(self, data :list, k: int):
        '''
        Find k th largest element in data. K should be between 1 and length of data (
                                                inclusive)
        '''
        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)
        # print('Before partition start, end, pivotIndex ',  start, end, pivotIndex)
        pivotIndex = self.partition(pivotIndex, start, end)
        # print(f'After partition , data = {data}, pivotIndex = {pivotIndex}')

        if end-pivotIndex >=k:
            return self._kth_largest(k, pivotIndex+1, end)

        elif end-pivotIndex+1 == k:
            return pivotIndex, 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)
```

```python
        # print('pivot index, value =',randomIndex,self.data[randomIndex])
        return randomIndex

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

class QuickSelectMedianSort(QuickSelect):
    def get_pivot_index(self, start, end):
        randomIndex = range(start, end+1)
        # Use inbuilt sort
        randomIndex = sorted(randomIndex, key = lambda index: self.data[index])
        median = randomIndex[(end-start)//2]
        # print(f'{start}:{end} - pivot index, value =',median,self.data[median])
        return median

    def __str__(self):
        return f'Quick select Median from sorting'

class QuickSelectMedianQS(QuickSelect):
    def get_pivot_index(self, start, end):
        q= QuickSelect()
        q.data = self.data
        median, medianval = q._kth_largest( (end-start)//2+1, start, end) # this
                                                    manipulates the input though
        # print(f'{start}:{end} - comparisons = {q.comparison_count} , pivot index,
                                                    value =',median,self.data[median])
        return median

    def __str__(self):
        return f'Quick select Median from quick select'

class QuickSelectMiddleIndex(QuickSelect):
    # Only works with identity permutation
    def get_pivot_index(self, start, end):
        median = start + (end-start)//2
        # print(f'{start}:{end} - pivot index, value =',median,self.data[median])
        return median

    def __str__(self):
        return f'Quick select middle index'

import time
if __name__ == '__main__':
    import sys
    algos = [QuickSelect(), QuickSelectMiddleIndex(), QuickSelectMedianSort(),
                                                QuickSelectMedianQS()]
    iterations  =10
    results = []
    for n in [1_000, 10_000, 100_000,1_000_000]:
        data = [i for i in range(1,n+1)]
        for q in algos:
            print(q)
            random.seed(48)
            comparisons = 0
            partitions = 0
            start = time.time()
            for _ in range(iterations):
                # data = [random.randint(0,n) for i in range(1,n+1)]
```

```
            k = 1 #max
            element = q.kth_largest(data, k)
            print(f'q: max index =  {element[0]}, max value = {element[1]}')
            comparisons += q.comparison_count
            partitions += q.partition_count
        end = time.time()
        results.append(pd.Series({
            'n':n,
            'algorithm':str(q),
            'expected partitions':partitions/iterations,
            'expected comparisons' :f'{comparisons/iterations/n:.2f} n',
            'expected runtime' : f'{(end-start)/iterations:.4f} s'
        }))
    df = pd.DataFrame(results).set_index(['n','algorithm'])
    print(df)
    print(df.to_latex(multirow = True, multicolumn_format = 'c'))
```

Problem 3

The undirected minimum S -T cut problem is
given an undirected graph $G = (V, E)$ with distinguished vertices s and t,
find a partition of the vertices (S, T) (where $S \cup T = V$ and $S \cap T = \emptyset$)
with $s \in S$ and $t \in T$ which minimizes the number of edges between S and T.

Q. Show that Karger's algorithm does not work for the S -T mincut problem

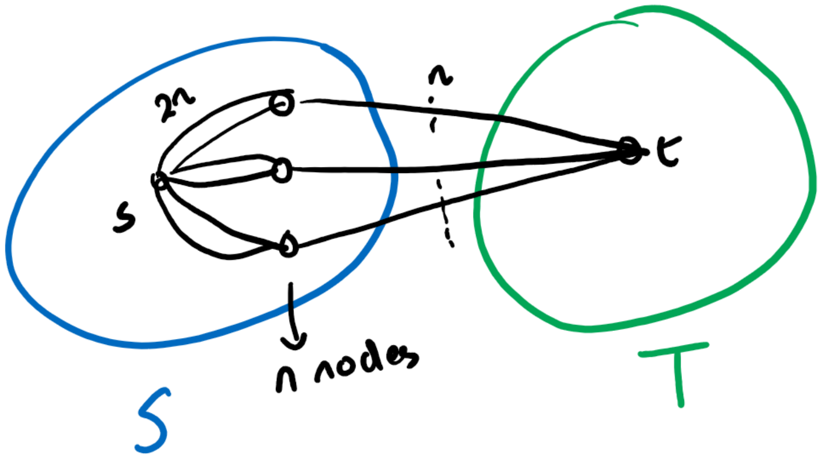

Figure 1: ST cut special case

Let's consider this family of graphs similar to 1 where there are two anchor nodes s and t. There are n other nodes which have edges with both s and t node. Each node has two edges with s and only one edge with t.

Total number of vertices = $n + 2$
Total number of edges = $n + 2n = 3n$
The mincut should pass through all the n edges between intermediate nodes and t
If Krager's algorithm is applied to this family of graphs,

$$P(success) = P(\text{not having any mincut edges contracted by the random contraction process})$$

$$P(\text{mincut edge is contracted in the first step}) = \frac{n}{3n}$$

For simplicity let's start the steps from 0.

At each step, 2 non-mincut edges need be contracted for success.

$$P(\text{ mincut edge is not contracted at step } i \mid \text{no mincut edge was contracted from step 0 to } i-1) = \frac{2n-2i}{3n-2i}$$

$$
\begin{aligned}
P(success) &= \prod_{i=0}^{n-1} \frac{2n-2i}{3n-2i} \quad \text{... since i starts from 0, it will end at n-1} \\
&\leq \prod_{i=n/2}^{n-1} \frac{2n-2i}{3n-2i} \quad \text{... since each term in the product was less than 1} \\
&\leq \prod_{i=n/2}^{n-1} \frac{2n-2*n/2}{3n-2*n/2} \quad \text{... since } \frac{2n-2*n/2}{3n-2*n/2} \geq \frac{2n-2i}{3n-2i} \text{ for } i \geq n/2 \\
&\leq \left(\frac{n}{2n}\right)^{n-1-n/2} \\
&\leq \left(\frac{1}{2}\right)^{\frac{n}{2}-1} \\
&\leq 2\left(\frac{1}{\sqrt{2}}\right)^{n}
\end{aligned}
$$

Since $\frac{1}{\sqrt{2}} < 1$, $P(success)$ will be exponentially small wrt n, Krager's algorithm will fail here as we will need exponentially many such runs for this algorithm to succeed with high probability.

Problem 4

There are n types of coupons.

P(getting coupon i out of n) = $\frac{1}{n}$

Each time you get a coupon, you are given a coupon of a random type (with equal probability of receiving each type of coupon).

Let X_k^n = number of coupons needed to get a new coupon when k coupons remain.
Theoritical value =

$$X_k^n = \frac{n}{k} \tag{4.1}$$

Let C_n = how many coupons you receive before you have completed the set.
Theoritical value =

$$C_n = n * H_n = n \ln n + 0.57n \tag{4.2}$$

4.1 Results

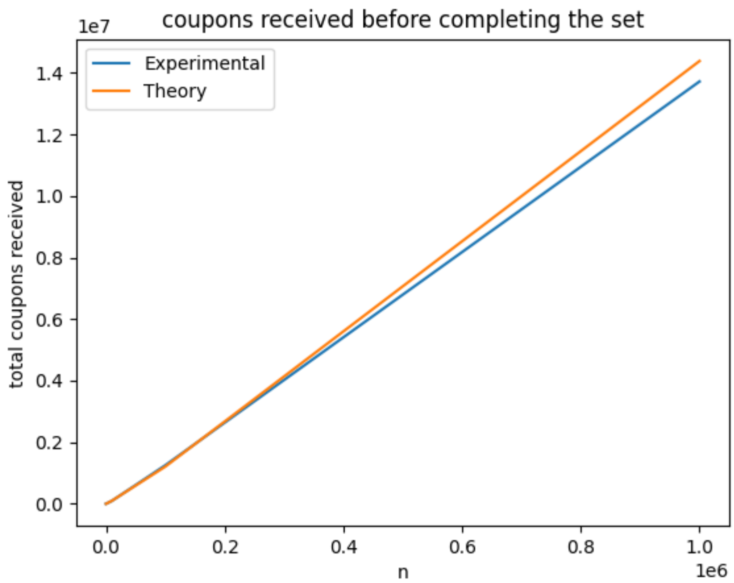

Figure 2: $E(C_n)$

For figure 2 , in 5 different iterations, C_n is calculated by counting total coupons received before completing the set. The range of n's used are [100, 1000, 10000, 100000, 1000000]. The experimental results match with the theoretical value (equation 4.2).

To plot figure 3, The range of n's used are [10000, 100000]. In 5 different iterations, X_k^n is calculated by counting number of coupons collected before collecting a unique coupon while k unique coupons were remaining. The experimental results match with the theoretical value (equation 4.1) which is $\frac{n}{k}$.

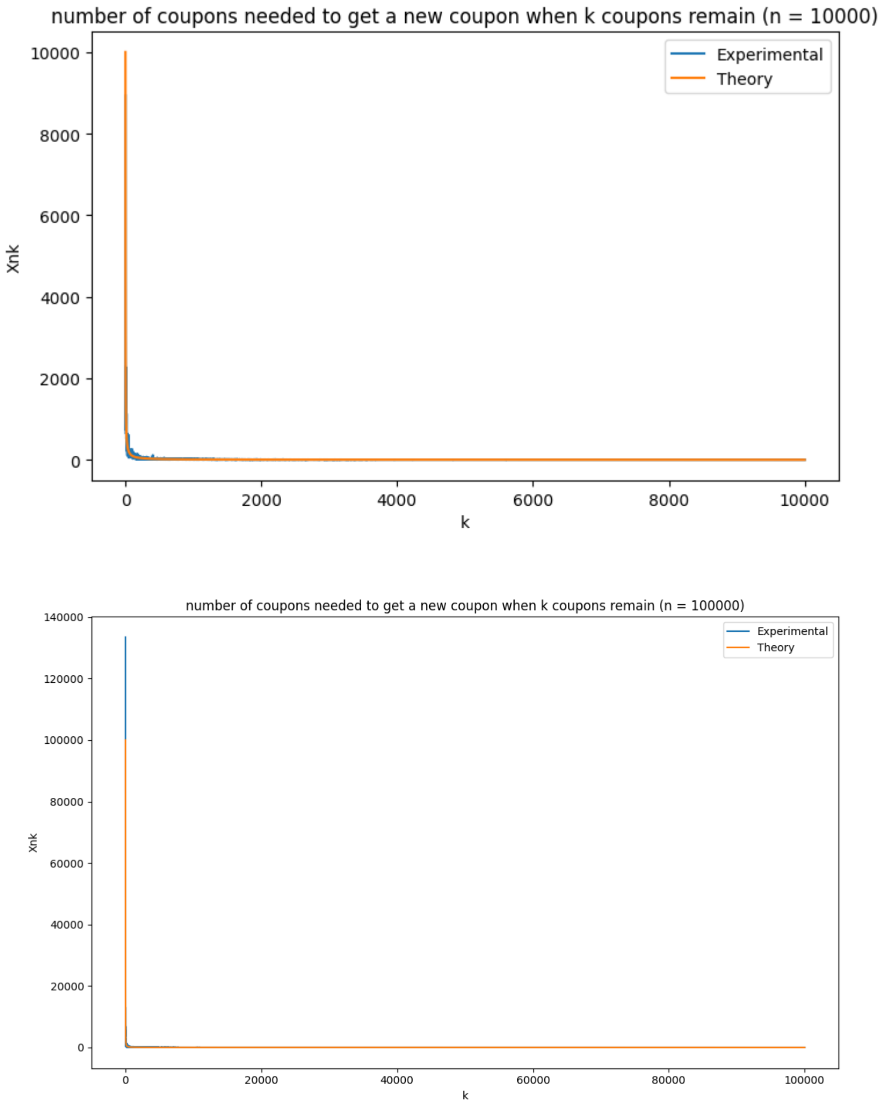

Figure 3: $E(X_k^n)$ for different n

4.2 Source code

```
from collections import defaultdict
import random, math
import numpy as np
import pandas as pd
import matplotlib.pyplot as plt
class CouponCollector():
    def __init__(self,n):
        self.n = n
        self.k = n # remaining coupons to be collected
        self.Xnk =  {}
        self.Cn = 0
        self.unique_coupons_collected = set()
    def collect_all_coupons(self):
        coupons_collected_this_iteration = 0
```

```
        while self.k>0:
            coupon = random.randint(1,self.n)
            coupons_collected_this_iteration += 1
            if coupon not in self.unique_coupons_collected:
                #new coupon collected
                self.unique_coupons_collected.add(coupon) # add to collected list
                self.Xnk[self.k] = coupons_collected_this_iteration #update random
                                                             variable value
                self.k -= 1 # remaining 1 less
                self.Cn += coupons_collected_this_iteration # total count
                coupons_collected_this_iteration = 0 #reset

    def __str__(self):
        return f'Cn = {self.Cn}, Xnk = {self.Xnk}'
    @staticmethod
    def theoritical_Xnk(n, k ):
        return n/k

    @staticmethod
    def theoritical_cn(n):
        return n * math.log(n) + 0.57 * n

def get_experimental_Cn_Xnk(n, iterations = 5):
    Cns = []
    Xnks = []
    for iteration in range(iterations):
        print('n, iteration', n, iteration)
        collector = CouponCollector(n)
        collector.collect_all_coupons()
        Cns.append(collector.Cn)
        Xnks.append(collector.Xnk)
    #expectation
    Cn = np.array(Cns).mean()
    Xnk = pd.DataFrame(Xnks).mean()
    return Cn, Xnk

def get_theoritical_Cn_Xnk(n):
    Xnk = {k: CouponCollector.theoritical_Xnk(n, k) for k in range( n,0,-1)}
    Cn = CouponCollector.theoritical_cn(n)
    return Cn, pd.Series(Xnk)

# plot Xnk for n = 100,00
n = 10000
c, x = get_experimental_Cn_Xnk(n)
plt.plot(x)
c, x = get_theoritical_Cn_Xnk(n)
plt.plot(x)
plt.legend(['Experimental', 'Theory'])
plt.xlabel('k')
plt.ylabel('Xnk')
plt.title(f'number of coupons needed to get a new coupon when k coupons remain (n = {n})
                                             ')


plt.figure()
#plot Cn
ns = [100, 1_000,100_00, 100_000, 1000_000]
c_exp = [get_experimental_Cn_Xnk(n)[0] for n in ns]
```

```
c_theo = [get_theoritical_Cn_Xnk(n)[0] for n in ns]

plt.plot(ns,c_exp)
plt.plot(ns,c_theo)
plt.legend(['Experimental', 'Theory'])
plt.xlabel('n')
plt.ylabel('total coupons received')
plt.title(f'coupons received before completing the set')


plt.show()
```