Computational Physics

Topic 02 — Computational Problems involving Marko Chains

Lecture 02 — The Collector Problem

Dr Kieran Murphy

Department of Computing and Mathematics,
SETU (Waterford).
(kieran.murphy@setu.ie)

Autumn Semester, 2025/26

RESOURCE OUTLINE LABEL

- Problem statement
- Sample run

The Coupon Collector Problem

A company decided to include a toy in their cereal boxes.
What is the expected number of boxes purchased in order to obtain all of the toys?

Some variations . . .

Cards collected in packs

Trading cards are obtained in packs of a fixed size.

Typically no repetition within a pack.

Unequal probabilities

Not all cards are equally likely.

A lot of effort is put into tuning the probabilities to maximise impact (increase demand) or minimise costs (prizes).

Multiple Collectors

One collector might want multiple collections,

or multiple collectors working together, trading cards, so that all get a full collection.

The Coupon Collector Problem — Specification

- Number of distinct **coupons** (trading cards, coins, etc.) is $n > 0$.
- Assumptions:
  - Coupons are obtained one at a time.
    - Later we will consider **packs** of size k with no repetition within a pack.
  - The number of copies of each coupon is effectively infinite.
    - If the number of copies of each coupon was small enough then the probabilities would change during the experiment based on which coupon have been seen already.
      (so would have a sampling without replacement problem — harder).
    - Note: 'effectively infinite' does not mean the actual number is very big, just that it is big enough.
  - Each coupon is equally likely to be found, i.e., uniform probabilities
    - uniform distribution — easiest but unrealistic for most trading cards/competitions situations.
    - **Zipf–Mandelbrot** distribution — more realistic (**power-law**) distribution.

What is the expected number of coupons collected in order to obtain m complete collections of the coupons?

Aside — History of the coupon collector problem

1708 The problem first appeared in 1708 in *De Mensura Sortis (On the Measurement of Chance)* by A. De Moivre.

- Additional results by various authors including Laplace and Euler in the case of uniform probabilities, i.e. when $p_j = 1/n$ for all j.

1954 H. Von Schelling obtained waiting time to complete a collection for non-uniform probabilities.

1960 D. J. Newman and L. Shepp calculated waiting time for two collections ($m = 2$).

Applications

- Electrical engineering — related to the cache fault problem, also used in electrical fault detection.
- Biology — used to estimate the number of species of animals (see Watterson estimator).

First a simulation . . .

```
count:    0    found:      collected: set()
```

The_Collector_Problem.ipynb In[4]:

```
np.random.seed(42)        # fixed seed during testing

n = 4
space = range(n)          # all possible coupons

collected = set()         # coupons collected to date

count = 0
print (f'count:␣{count:4d}␣\tfound:␣␣␣\tcollected:␣{collected}')
while len(collected)<n:   # collection is incomplete

    found = set(choice(space, 1))          # get next (random) coupon
    collected = collected.union(found)     # sets so duplicates dropped
    count += 1

    print (f'count:␣{count:4d}␣\tfound:␣{found}␣\tcollected:␣{collected}')
```

... wrap code up in a function ...

The_Collector_Problem.ipynb In[5]:

Using optional parameters we can set the seed for reproducible results and displaying debug output.

```
def run_experiment(n, seed=None, debug=False):

    if seed is not None: np.random.seed(seed)

    space = range(n)          # all possible coupons

    collected = set()         # coupons collected to date

    count = 0
    if debug: print (f'count:␣{count:4d}␣\tfound:␣␣␣\tcollected:␣{collected}')
    while len(collected)<n:   # not completed collection yet

        found = set(choice(space,1))            # get next (random) coupon
        collected = collected.union(found)      # using sets so duplicates dropped
        count += 1

        if debug: print (f'count:␣{count:4d}␣\tfound:␣{found}␣\tcollected:␣{collected}')

    return count
```

… and a few sample runs …

The_Collector_Problem.ipynb In[6]:

```
run_experiment(5, seed=105, debug=True)
```

```
count:  0   found:      collected: set()
count:  1   found: {0}  collected: {0}
count:  2   found: {1}  collected: {0, 1}
count:  3   found: {4}  collected: {0, 1, 4}
count:  4   found: {0}  collected: {0, 1, 4}
count:  5   found: {0}  collected: {0, 1, 4}
count:  6   found: {4}  collected: {0, 1, 4}
count:  7   found: {0}  collected: {0, 1, 4}
count:  8   found: {4}  collected: {0, 1, 4}
count:  9   found: {1}  collected: {0, 1, 4}
count: 10   found: {1}  collected: {0, 1, 4}
count: 11   found: {1}  collected: {0, 1, 4}
count: 12   found: {3}  collected: {0, 1, 3, 4}
count: 13   found: {4}  collected: {0, 1, 3, 4}
count: 14   found: {3}  collected: {0, 1, 3, 4}
count: 15   found: {1}  collected: {0, 1, 3, 4}
count: 16   found: {4}  collected: {0, 1, 3, 4}
count: 17   found: {4}  collected: {0, 1, 3, 4}
count: 18   found: {1}  collected: {0, 1, 3, 4}
count: 19   found: {4}  collected: {0, 1, 3, 4}
count: 20   found: {2}  collected: {0, 1, 2, 3, 4}
```

The_Collector_Problem.ipynb In[7]:

```
run_experiment(5, seed=1013, debug=True)
```

```
count:  0   found:      collected: set()
count:  1   found: {0}  collected: {0}
count:  2   found: {4}  collected: {0, 4}
count:  3   found: {2}  collected: {0, 2, 4}
count:  4   found: {1}  collected: {0, 1, 2, 4}
count:  5   found: {0}  collected: {0, 1, 2, 4}
count:  6   found: {0}  collected: {0, 1, 2, 4}
count:  7   found: {3}  collected: {0, 1, 2, 3, 4}
```

The_Collector_Problem.ipynb In[8]:

```
run_experiment(3, seed=2, debug=True)
```

```
count:  0   found:      collected: set()
count:  1   found: {0}  collected: {0}
count:  2   found: {1}  collected: {0, 1}
count:  3   found: {0}  collected: {0, 1}
count:  4   found: {2}  collected: {0, 1, 2}
```

Need to get some idea of variation … so repeat runs …

The_Collector_Problem.ipynb In[9]:

```
import scipy.stats as stats

data = [run_experiment(4) for _ in range(10)]
print(data)
m, se = np.mean(data), stats.sem(data)
print("\n95%% CI for number of coupns = %s +/- %.2f" % (m, 1.96*se) )
```

```
[7, 8, 6, 8, 5, 7, 12, 6, 5, 4]

95% CI for number of coupns = 6.8 +/- 1.40
```

The_Collector_Problem.ipynb In[10]:

```
data = [run_experiment(4) for _ in range(100)]
m, se = np.mean(data), stats.sem(data)
print("\n95%% CI for number of coupns = %s +/- %.2f" % (m, 1.96*se) )
```

```

95% CI for number of coupns = 8.82 +/- 0.77
```

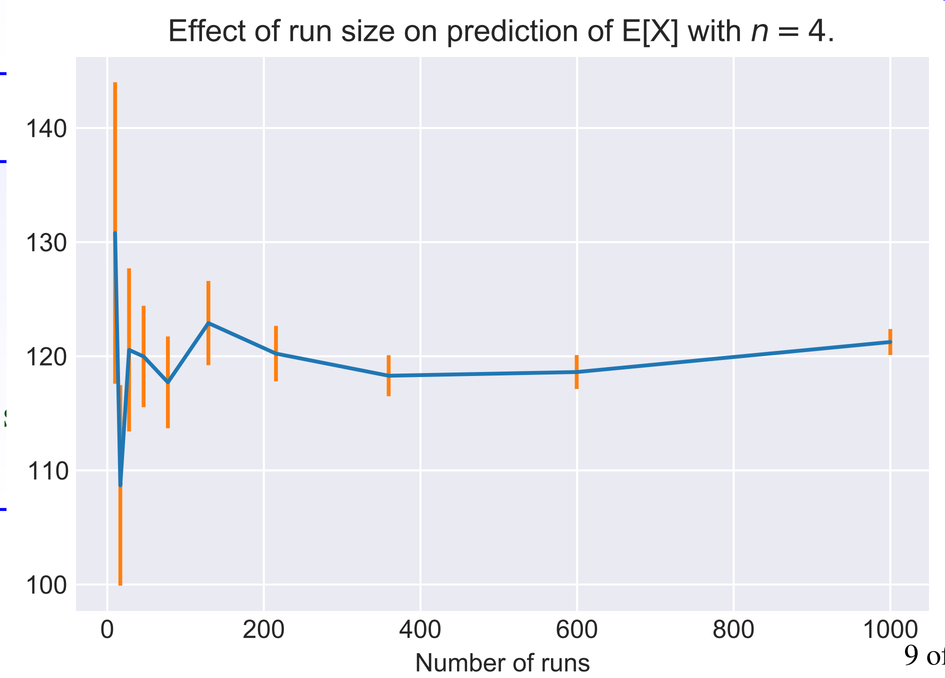
. . . a picture is worth a thousand words . . .

The Collector Problem.ipynb In[13]:

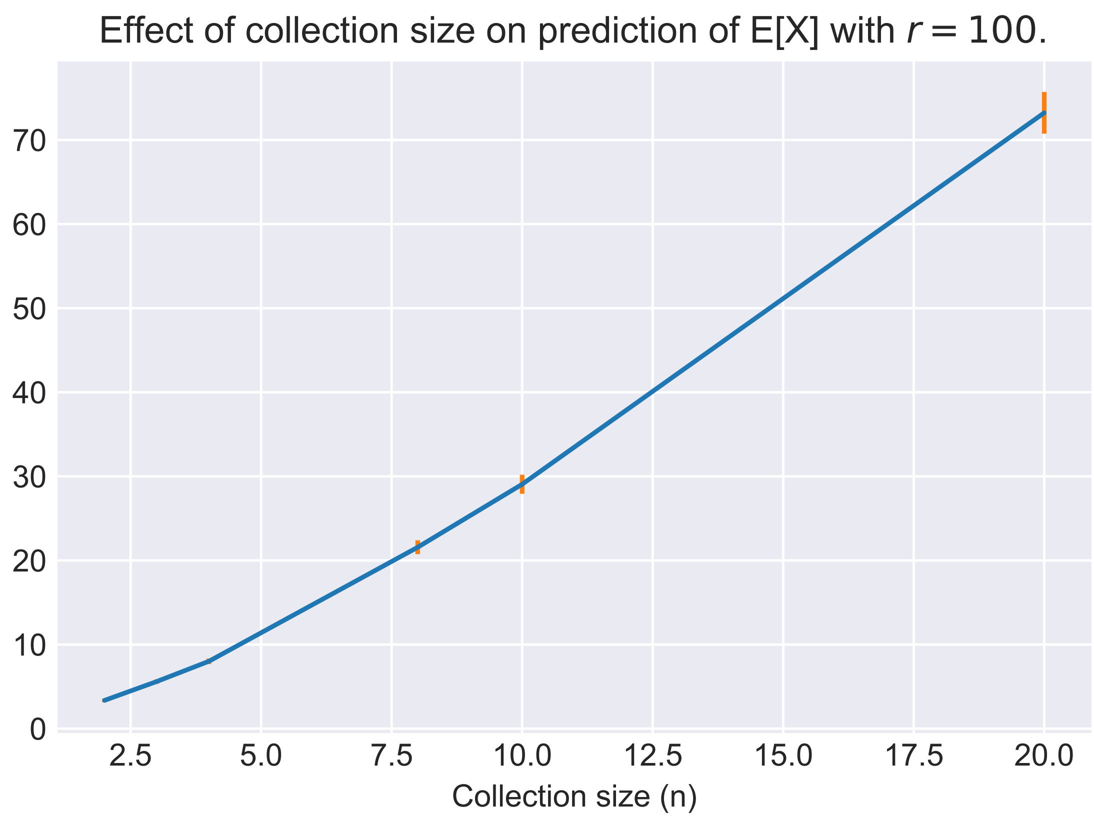
Confidence intervals are shrinking ✔
But only beginning to overlap - prob need larger samples
Best estimate for expected value is about 8.5

```
rValues = np.logspace(1,3,10)
m = []
se = []
for r in rValues:
    data = [run_experiment(30) for _ in range(int(r))]
    m.append(np.mean(data))
    se.append(stats.sem(data))
```

```
plt.plot(rValues, m)
plt.errorbar(rValues, m, se, linestyle='None')
plt.xlabel("Number␣of␣runs")
plt.title("Effect␣of␣run␣size␣on␣prediction␣of␣E
plt.savefig("output/coupons_n_4.pdf", bbox_inches
plt.show()
```



Effect of collection size (n) ...



n	mean	se
2	3.36	0.21
3	5.60	0.27
4	8.01	0.33
8	21.57	0.82
10	29.06	1.13
20	73.23	2.48

- Variance increases with collection size (n), but this is offset by the fact that the estimate for the expected number of coupons needed is increasing faster.

Theoretical approach — via geometric distribution

Let X denote the (random) number of coupons that we need to purchase in order to complete our collection of n coupons.