CS168 Spring Assignment 1

SUNet ID(s): 05794739
Name(s): Luis A. Perez
Collaborators:

By turning in this assignment, I agree by the Stanford honor code and declare that all of this is my own work.

Part 1

(a) Before running the strategies above, we suspect the following will occur.

We expect that Strategy 3 will have a lower maximum value than Strategy 2, which in turn has a lower maximum value than Strategy 1 (this seems rather straight forward, given the fact that Strategy 2 can pick from two buckets and Strategy 3 from three).

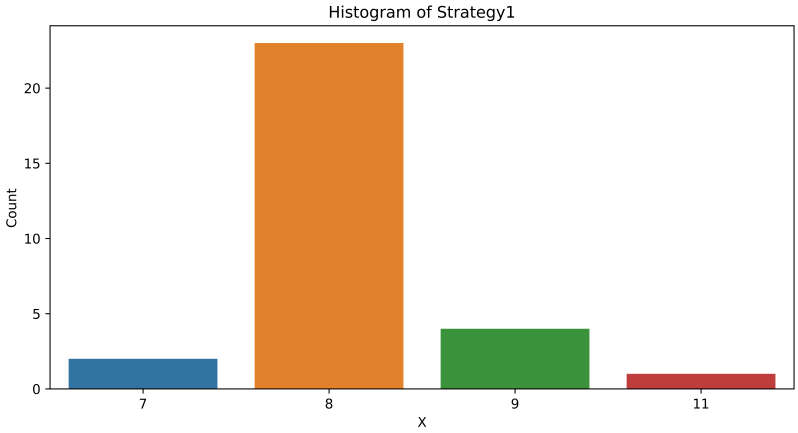
As for Strategy 4, it's quite similar to Strategy 2 except it should be worse (higher return value).

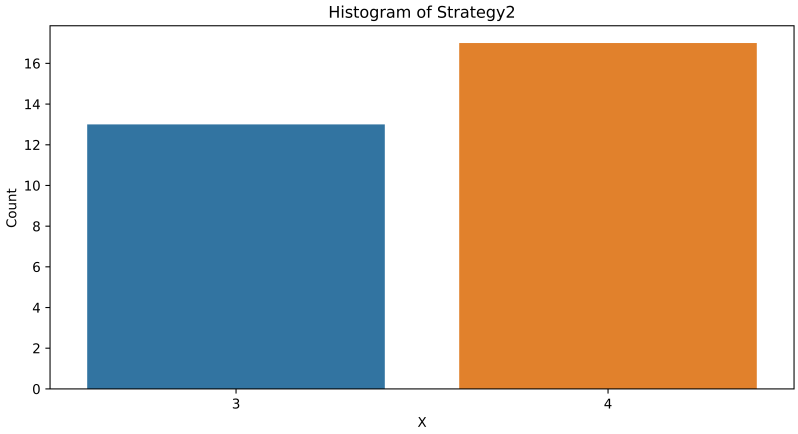
The code for each strategy is below.

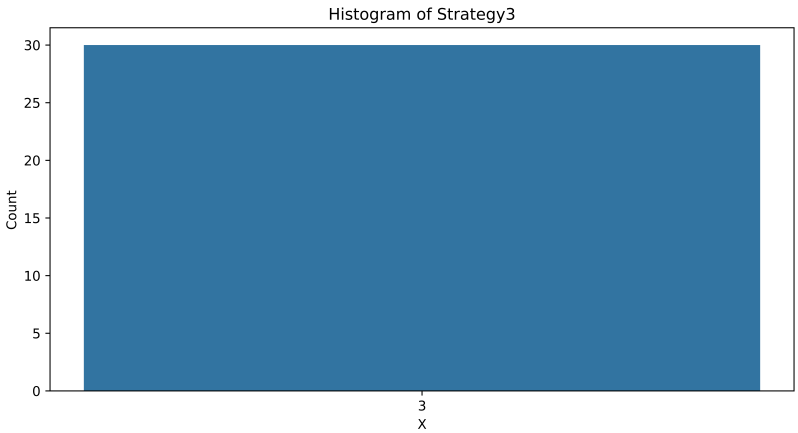
```
def strategy1(N: int) -> int:
  binIdxSamples = np.random.randint(low=0, high=N, size=N)
  bins = collections.Counter(binIdxSamples)
  return max(bins.items(), key=lambda x: x[1])[1]

def strategy2(N: int) -> int:
  firstIdx = np.random.randint(low=0, high=N, size=N)
  secondIdx = np.random.randint(low=0, high=N, size=N)
  coinFlips = np.random.randint(low=0, high=2, size=N)
  bins = collections.Counter()
  for flip, (i,j) in enumerate(zip(firstIdx, secondIdx)):
      if bins[i] < bins[j]:
          bins[i] += 1
      elif bins[j] < bins[i]:
          bins[j] += 1
      else:
          bins[i if coinFlips[flip] else j] += 1
  return max(bins.items(), key=lambda x: x[1])[1]

def strategy3(N: int) -> int:
  firstIdx = np.random.randint(low=0, high=N, size=N)
  secondIdx = np.random.randint(low=0, high=N, size=N)
  thirdIdx = np.random.randint(low=0, high=N, size=N)
```

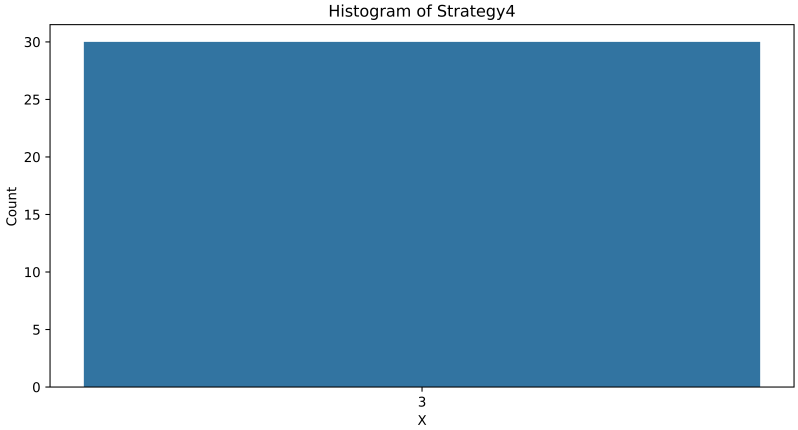
```
  doubleTieFlip = np.random.randint(low=0, high=2, size=N)
  tripleTieFlip = np.random.randint(low=0, high=3, size=N)
  bins = collections.Counter()
  for flip, (i,j,k) in enumerate(zip(firstIdx, secondIdx, thirdIdx)):
      sortedOutcomes = sorted([(bins[i],i), (bins[j], j), (bins[k], k)])
      # Triple tie.
      if bins[i] == bins[j] and bins[j] == bins[k]:
          flipOutcome = tripleTieFlip[flip]
          if flipOutcome == 0:
              bins[i] += 1
          elif flipOutcome == 1:
              bins[j] += 1
          elif flipOutcome == 2:
              bins[j] += 1
          else:
              assert False, "%s is not a valid triple flip outcome" % flipOutcome
      # Min double tie.
      elif sortedOutcomes[0][0] == sortedOutcomes[1][0]:
          outcome = sortedOutcomes[0 if doubleTieFlip[flip] else 1]
          bins[outcome[1]] += 1
      # No tie.
      else:
          outcome = sortedOutcomes[0]
          bins[outcome[1]] += 1
  return max(bins.items(), key=lambda x: x[1])[1]

def strategy4(N: int) -> int:
  assert N % 2 == 0, "We assume for Strategy 4 that N is even!"
  firstIdx = np.random.randint(low=0, high=N // 2, size=N)
  secondIdx = np.random.randint(low=N // 2 + 1, high=N, size=N)
  bins = collections.Counter()
  for (first, second) in zip(firstIdx, secondIdx):
      if bins[first] <= bins[second]:
          bins[first] += 1
      else:
          bins[second] += 1
  return max(bins.items(), key=lambda x: x[1])[1]
```

(b) We now include the histograms for each strategy. The cheapest strategy is Strategy 1, since it requires just one draw from our randomness source (to compute the bucket index). However, this strategy also leads to the highest variance, with at least one bucket containing at least 7 balls in all of our simulation.


Histogram of Strategy1
Count
X
0
5
10
15
20
7
8
9
11


Histogram of Strategy2
Count
X
0
2
4
6
8
10
12
14
16
3
4


Histogram of Strategy3
Count
X
0
5
10
15
20
25
30
3



As we expected, the other strategies reduce the number of collisions, with Strategy 3 performing the best. However, we note that Strategy 3 requires three draws from our uniform distributuon, as well as further randomness in the case of ties (which, if all goes well, we'd expect to be somewhat frequent especially at the beginning when the buckets are primarily empty). From this point of view, it's quite an expensive strategy.

As such, despite the fact that in none of our 30 simulations any bucket contained more than 3 balls, Strategy 3 is too costly to be the best.

In fact, the best is Strategy 4. It quite frequently (28 our of 30 of our simulations) achives the same results as Strategy 3. However, it only costs two hashes, and there's not additional randomness required.

(c) The analogy is straight-forward. We can imagine the processes of placing a ball into a bin as equivalant to that of inserting an element into a hash-table. As such, X in the above strategies represents the length of the longest linked-list in the hashtable. Longer linked lists require more operations to traverse, both when doing a lookup as well as when doing an insertion.

(d) The strategies above do suggest different implementations of hash tables with chaining. Strategy 1 corresponds to the straight-forward implementation where a bucket is picked at random (using a suitable hash function) and a linked list is used to handle collisions. As per our simulations, the worst-case lookup time in this table would consists of one hash and (on average) 8 operations searching to the end of a linked list.

For the other strategies, we assume that each bucket keeps a local count of the number of elements present in the linked list.

These strategies increase the number of hashes (Strategy 2 and 4 require two hashes, Strategy 3 requires three), and tend to cut the insertion time down. For Strategy 2, in the worst case, we insert into a bucket with four elements already present. Similarly for Strategy 3 and 4, we insert into a bucket with 3 elements already present. However,

the lookup times in the worst won't be much improved. Since both Strategy 2 and Strategy 3 rely on randomness to break ties, it means that for Strategy 2 we have to search through two linked lists in the case of tied bucket counts, and for Strategy 3 we have to search through three linked lists in the case of tied bucket count. While we expect these lists to be shorter, this is likely cancelled out somewhat by the increased number of buckets needed to be searched.

As such, Strategy 4 seems to be the best. Since it is fully deterministic, in the event of ties we know the element will have been added to the smaller bucket. Therefore, the worst-case search time for Strategy 4 is in the worst case the length of the longest linked list, which according to our simulations is 3.

Part 2

(a)
```
class CountMinSketch:
    """Implementation of the CountMinSketch datastructure."""
    def __init__(self, trial: int, b=256, l=4) -> None:
        self._id = trial
        # Number of buckets.
        self.b = b
        # Number of hash functions
        self.l = l
        self._table = [[0] * self.b for _ in range(self.l)]

    def _hash(self, x: int) -> List[int]:
        """Hashes x to the correspondg self.l hash functions."""
        inputString = (str(x) + str(self._id - 1)).encode('utf-8')
        inputHash = hashlib.md5(inputString).hexdigest()
        return [int(inputHash[2*i:2*(i+1)], 16) % self.b
                for i in range(self.l)]

    def Increment(self, x: int) -> None:
        """Increments the count of x."""
        for i, h in enumerate(self._hash(x)):
            self._table[i][h] += 1


    def Count(self, x: int) -> int:
        return min([self._table[i][h]
                    for i, h in enumerate(self._hash(x))])
```

(b) There are 21 heavy hitters in the stream with the given frequencies.

As defined, the total number of elements can be computed directly as:

$$\sum_{i=1}^{9} i * 1000 + \sum_{i=1}^{50} i^2 = 87925$$

As such, the elements which appears at least 1% of the time must appear at least 879 times in the stream. Only the elements of the form $9000 + i$ even stand a chance, since each appears i^2 times. However, noting that $29^2 < 879 < 30^2$, the only heavy hitters are $9000 + i$ for $30 \leq i \leq 50$. As such, there are a total of 21 heavy hitters.

(c) For FORWARD Stream, the mean estimate for the frequency of 9050 is 2645.7, while the mean estimate of the number of heavy hitters is 24.6.

For REVERSE Stream, the mean estimate for the frequency of 9050 is 2645.7, while the mean estimate of the number of heavy hitters is 24.6.

For RANDOM Stream, the mean estimate for the frequency of 9050 is 2645.7, while the mean estimate of the number of heavy hitters is 24.6.

As we can see from above, the stream order does not affect our answers at all. This makes complete sense, since for a fixed trial j, all simulations see every element in the stream. Each element x (no matter in what order it is seen) will deterministically increment the same set of l buckets, $h_l(x||j)$. As such, at the end of the stream, the state of the CountMinSketch data-structure is the same, no matter in what order the elements arrived.

(d)
```
class CountMinSketch:
    """Implementation of the CountMinSketch datastructure."""
    def __init__(self, trial: int, b=256, l=4, isConservative: bool=False) -> None:
        self._id = trial
        # Number of buckets.
        self.b = b
        # Number of hash functions
        self.l = l
        self.isConservative = isConservative
        self._table = [[0] * self.b for _ in range(self.l)]

    def _hash(self, x: int) -> List[int]:
        """Hashes x to the correspondg self.l hash functions."""
        inputString = (str(x) + str(self._id - 1)).encode('utf-8')
        inputHash = hashlib.md5(inputString).hexdigest()
        return [int(inputHash[2*i:2*(i+1)], 16) % self.b
                for i in range(self.l)]

    def Increment(self, x: int) -> None:
        """Increments the count of x."""
        hashes = self._hash(x)
        if self.isConservative:
            currentCounts = [self._table[i][h]
                             for i, h in enumerate(hashes)]
            minCount = min(currentCounts)
        else:
            minCount = None
        for i, h in enumerate(hashes):
            if minCount is None or self._table[i][h] == minCount:
                self._table[i][h] += 1

    def Count(self, x: int) -> int:
        return min([self._table[i][h]
                    for i, h in enumerate(self._hash(x))])
```

(e) The count-min sketch will never under-estimate the count of a value, even with conservative updates. Note that we have $\ell = 4$ arrays of counters. For a given element x, every time this element is seen, at least one of the counters is incremented by 1. Furthermore, the incremented counter is the one with the smallest existing count. As such

We can proof more formally through induction. Note that in the base case, each of the l buckets corresponding to an arbitrary element x are an overestimate. That is to say:

$$0 = CMS^0[\ell][h_\ell(x)] \geq f_x^0 = 0, \forall \ell$$

Now, let us assume the above holds after processing the i-th element in the stream.

$$CMS^i[\ell][h_\ell(x)] \geq f_x^i, \forall \ell$$

Then it's clear that even with the conservative update rule, this will hold after processing the $i+1$-th element in the stream. If the $i+1$-th element is not x, this holds trivially. If it is x, then note that we will increment the count for all ℓ such that $CMS^i[\ell][h_\ell(x)] = \min_\ell \{CMS^i[\ell][h_\ell(x)]\}$. This immediately implies:

$$CMS^{i+1}[\ell][h_\ell(x)] \geq f_x^i + 1 \geq f_x^{i+1}, \forall \ell$$

As desired. Therefore our count-min sketch will always over-estimate the true frequency of x throughout the processing of the stream.

(f) **Conservative** For FORWARD Stream, the mean estimate for the frequency of 9050 is 2577.2, while the mean estimate of the number of heavy hitters is 23.4.

Conservative For REVERSE Stream, the mean estimate for the frequency of 9050 is 2500.0, while the mean estimate of the number of heavy hitters is 22.2.

Conservative For RANDOM Stream, the mean estimate for the frequency of 9050 is 2500.0, while the mean estimate of the number of heavy hitters is 22.2.

As we can see from above, the stream order **does** affect the simulations. The FORWARD stream estimates a strictly larger quantity than the REVERSE and RANDOM streams. Why?

This is because during the processing of the forward stream, a significant portion of the data structure will be filled, somewhat uniformly (the elements from 1 to 9003 all occur at most 9 times and at the beginning). As such, by the time we begin adding the more frequently occurring elements (for example 9050 which occurs 2500 times), our conservative update rule is fairly likely to encounger buckets that have collided already with our 9050 buckets.

On the other hand, when processing the stream in reverse order the frequency count for 9050 will be exact (after we finish processing it). As we continue to process the other values, the buckets corresponding to 9050 will be incremented if and only if one

of our elements hashes to the exact same buckets, which basically will never happen. The reason for this is that we only ever update the minimum count bucket to which our element hashed – since the buckets for 9050 are already filled, and 9050 is the *most* frequent element, even in the event that *some* buckets collide, the minimum bucket won't be the one colliding with 9050 since it's count will already be large.

This leads to the following conclusion. Generally speaking, with the conservative update rule, the optimal stream is one where the elements occur in decreasing order of frequency. With this in mind, it makes sense that the RANDOM stream would perform similarly well, since shuffling the elements at random means the likelihood of their appeareance is precisely proportional to their frequence, so in expectation we'll be processing more frequent elements earlier in the stream.

Code

HW1

April 9, 2020

1 CS 168 Spring Assignment [TODO]

SUNet ID(s): 05794739

Name(s): Luis A. Perez

Collaborators: None

By turning in this assignment, I agree by the Stanford honor code and declare that all of this is my own work.

2 Imports

[149]:
```
import collections
import matplotlib.pyplot as plt
import hashlib
import enum
import numpy as np
import pandas as pd
import seaborn as sns
import random

from typing import Callable, List, Iterator, Text, Tuple

# Make figure larger
plt.rcParams['figure.figsize'] = [10, 5]
```

2.1 Part 1

2.1.1 Part 1a : Defining each strategy.

Strategy 1 Select one of the N bins uniformly at random, and place the current ball in it.

[63]:
```
def strategy1(N: int) -> int:
    """Select one of N bins uniformly and place one of the N balls in it.

    Args:
        N: The number of bins and number of balls.
```

```
    Returns:
        The number of balls in the most populated bin.
    """
    binIdxSamples = np.random.randint(low=0, high=N, size=N)
    bins = collections.Counter(binIdxSamples)
    return max(bins.items(), key=lambda x: x[1])[1]
```

Strategy 2 Select two of the N bins uniformly at random (either with or without replacement), and look at how many balls are already in each. If one bin has strictly fewer balls than the other, place the current ball in that bin. If both bins have the same number of balls, pick one of the two at random and place the current ball in it.

[64]:
```
def strategy2(N: int) -> int:
    """Select two bins at random and place ball into bin with fewer. Break ties␣
 ↪at random.

    Args:
        N: The number of bins and number of balls.

    Returns:
        The number of balls in the most populated bin.
    """
    firstIdx = np.random.randint(low=0, high=N, size=N)
    secondIdx = np.random.randint(low=0, high=N, size=N)
    coinFlips = np.random.randint(low=0, high=2, size=N)
    bins = collections.Counter()
    for flip, (i,j) in enumerate(zip(firstIdx, secondIdx)):
        if bins[i] < bins[j]:
            bins[i] += 1
        elif bins[j] < bins[i]:
            bins[j] += 1
        else:
            bins[i if coinFlips[flip] else j] += 1
    return max(bins.items(), key=lambda x: x[1])[1]
```

Strategy 3: Same as Strategy 2 except with 3 bins at random.

[65]:
```
def strategy3(N: int) -> int:
    """Select three bins at random and place ball into bin with fewer. Break␣
 ↪ties at random.

    Args:
        N: The number of bins and number of balls.

    Returns:
        The number of balls in the most populated bin.
    """
```

```
    firstIdx = np.random.randint(low=0, high=N, size=N)
    secondIdx = np.random.randint(low=0, high=N, size=N)
    thirdIdx = np.random.randint(low=0, high=N, size=N)
    doubleTieFlip = np.random.randint(low=0, high=2, size=N)
    tripleTieFlip = np.random.randint(low=0, high=3, size=N)
    bins = collections.Counter()
    for flip, (i,j,k) in enumerate(zip(firstIdx, secondIdx, thirdIdx)):
        sortedOutcomes = sorted([(bins[i],i), (bins[j], j), (bins[k], k)])
        # Triple tie.
        if bins[i] == bins[j] and bins[j] == bins[k]:
            flipOutcome = tripleTieFlip[flip]
            if flipOutcome == 0:
                bins[i] += 1
            elif flipOutcome == 1:
                bins[j] += 1
            elif flipOutcome == 2:
                bins[j] += 1
            else:
                assert False, "%s is not a valid triple flip outcome" %␣
↪flipOutcome
        # Min double tie.
        elif sortedOutcomes[0][0] == sortedOutcomes[1][0]:
            outcome = sortedOutcomes[0 if doubleTieFlip[flip] else 1]
            bins[outcome[1]] += 1
        # No tie.
        else:
            outcome = sortedOutcomes[0]
            bins[outcome[1]] += 1
    return max(bins.items(), key=lambda x: x[1])[1]
```

Strategy 4: Select two bins as follows: the first bin is selected uniformly from the first $\frac{N}{2}$ bins, and the second uniformly from the last $\frac{N}{2}$ bins. (You can assume that N is even.) If one bin has strictly fewer balls than the other, place the current ball in that bin. If both bins have the same number of balls, place the current ball (deterministically) in the first of the two bins.

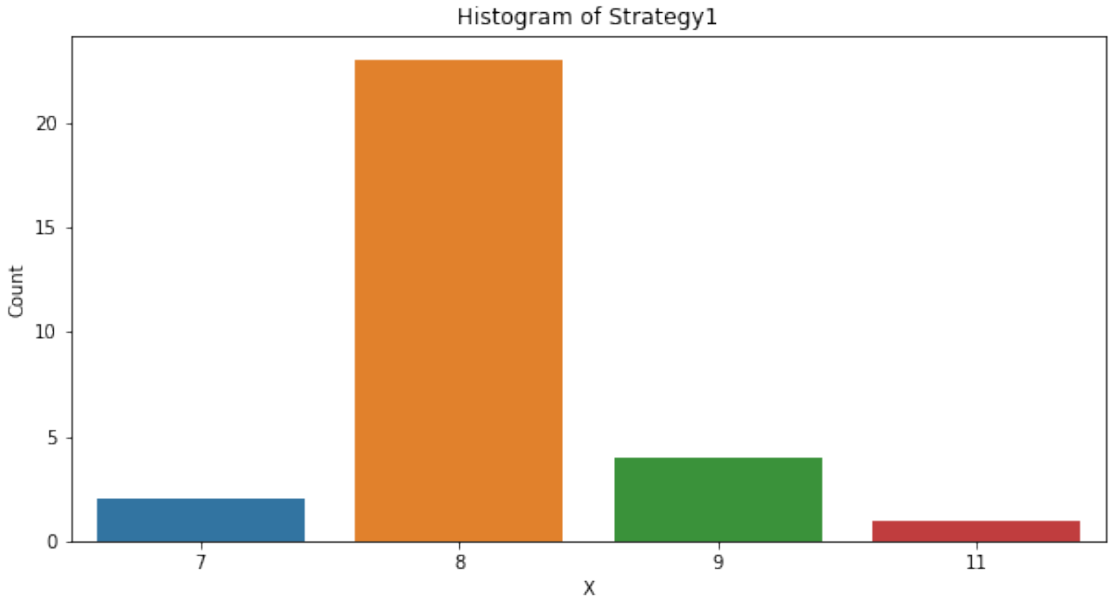
[66]:
```
def strategy4(N: int) -> int:
    """Select bin from first half and second half. Place ball into bin with␣
↪fewer balls, favoring first half.

    Args:
        N: The number of bins and number of balls.

    Returns:
        The number of balls in the most populated bin.
    """
    assert N % 2 == 0, "We assume for Strategy 4 that N is even!"
    firstIdx = np.random.randint(low=0, high=N // 2, size=N)
```

```
    secondIdx = np.random.randint(low=N // 2 + 1, high=N, size=N)
    bins = collections.Counter()
    for (first, second) in zip(firstIdx, secondIdx):
        if bins[first] <= bins[second]:
            bins[first] += 1
        else:
            bins[second] += 1
    return max(bins.items(), key=lambda x: x[1])[1]
```

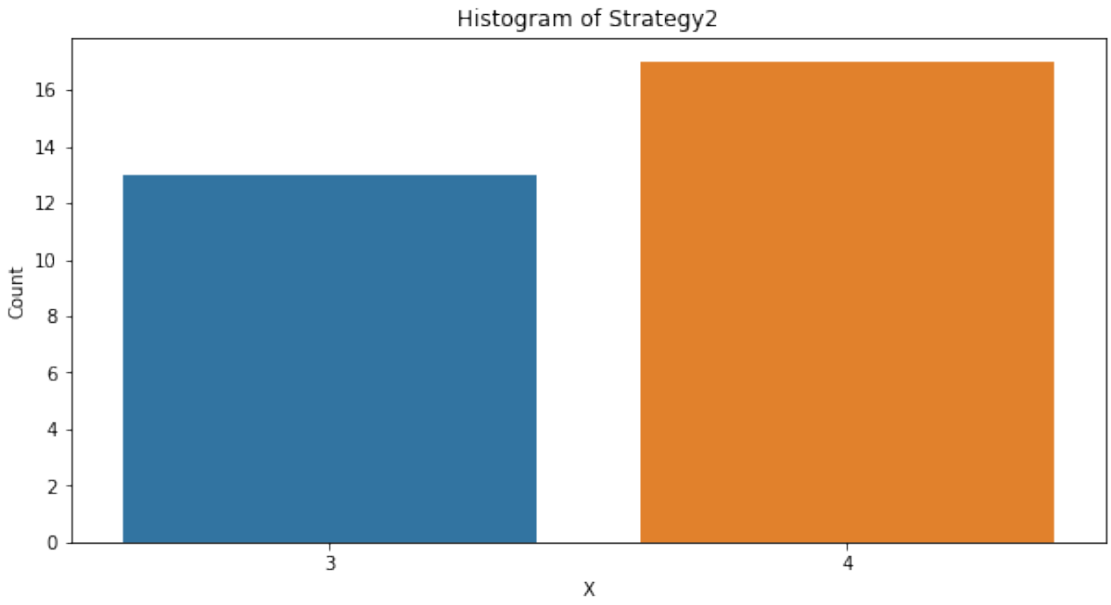
2.1.2 Part 1b: Simulation

[116]:
```
def simulate(strategy_fn: Callable[[int], int], nTrials: int, title: Text) ->␣
↪None:
    """Simulates the given stragy nTrials and plots the histogram.

    Args:
        strategy_fn: The callable defining the strategy.
        nTrials: The number of trials to run.
        title: The plot title for the generated histogram.
    """
    N = 200000
    simulation_results = [strategy_fn(N) for _ in range(nTrials)]
    counts = collections.Counter(simulation_results)
    data = pd.DataFrame(zip(counts.keys(), counts.values()),
                        columns=['value', 'count'])
    sns.barplot(data=data, x='value', y='count')
    plt.xlabel('X')
    plt.ylabel('Count')
    plt.title("Histogram of %s" % title)
    plt.savefig('figures/%s.png' % title, dpi=1200)
    plt.show()
```

[121]:
```
simulate(strategy1, nTrials=30, title="Strategy1")
```

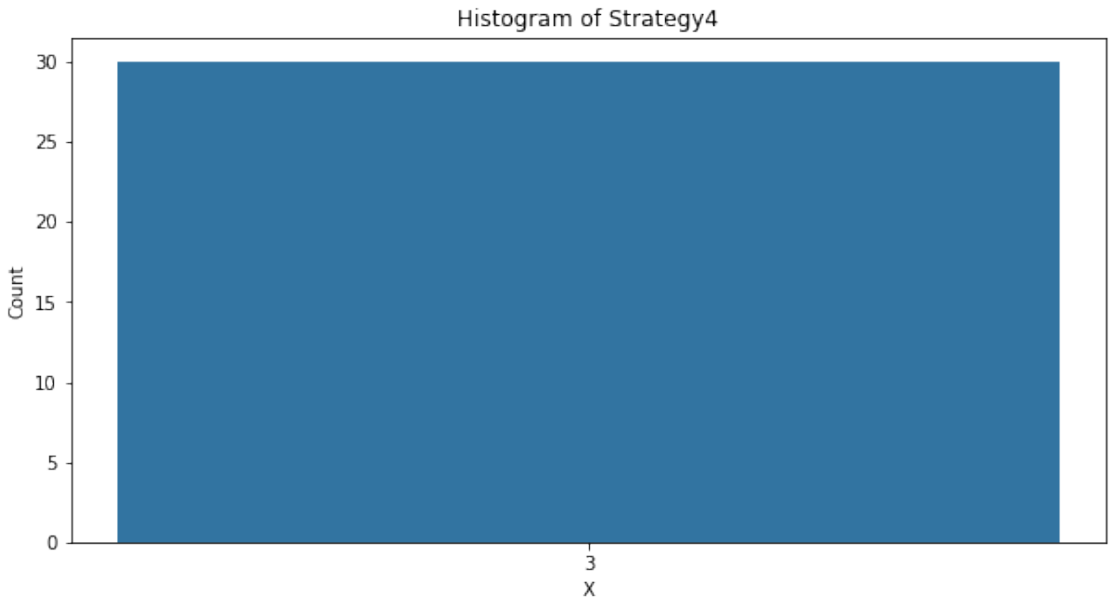


[122]:
```
simulate(strategy2, nTrials=30, title="Strategy2")
```



[123]:
```
simulate(strategy3, nTrials=30, title="Strategy3")
```

[126]:
```
simulate(strategy4, nTrials=30, title="Strategy4")
```



2.2 Part 2

[198]:
```
class CountMinSketch:
    """Implementation of the CountMinSketch datastructure."""
    def __init__(self, trial: int, b=256, l=4, isConservative: bool=False) ->␣
↪None:
        self._id = trial
        # Number of buckets.
        self.b = b
        # Number of hash functions
        self.l = l
        self.isConservative = isConservative
        self._table = [[0] * self.b for _ in range(self.l)]

    def _hash(self, x: int) -> List[int]:
        """Hashes x to the correspondg self.l hash functions."""
        inputString = (str(x) + str(self._id - 1)).encode('utf-8')
        inputHash = hashlib.md5(inputString).hexdigest()
        return [int(inputHash[2*i:2*(i+1)], 16) % self.b
                for i in range(self.l)]

    def Increment(self, x: int) -> None:
        """Increments the count of x."""
        hashes = self._hash(x)
        if self.isConservative:
            currentCounts = [self._table[i][h]
                             for i, h in enumerate(hashes)]
            minCount = min(currentCounts)
        else:
            minCount = None
        for i, h in enumerate(hashes):
            if minCount is None or self._table[i][h] == minCount:
                self._table[i][h] += 1

    def Count(self, x: int) -> int:
        return min([self._table[i][h]
                    for i, h in enumerate(self._hash(x))])
```

[199]:
```
@enum.unique
class StreamType(enum.Enum):
    FORWARD = 1
    REVERSE = 2
    RANDOM = 3
```

[200]:
```
def generate_stream(sType: StreamType) -> Iterator[int]:
    """Iterator for the stream of the given type."""
    if sType == StreamType.FORWARD:
```

```
        i = 0
        while i < 9:
            for el in range(i*1000 + 1, (i+1)*1000 + 1):
                for _ in range(i + 1):
                    yield el
            i += 1
        for el in range(9001, 9051):
            for _ in range((el - 9000)**2):
                yield el
    elif sType == StreamType.REVERSE:
        # Just materialize and reverse it...
        for el in reversed(list(generate_stream(StreamType.FORWARD))):
            yield el
    elif sType == StreamType.RANDOM:
        materialized = list(generate_stream(StreamType.FORWARD))
        random.shuffle(materialized)
        for el in materialized:
            yield el
    else:
        assert False
```

[201]:
```
def simulate_stream(sketch: CountMinSketch, sType: StreamType) -> Tuple[int,␣
↪int]:
    """Simulates the specified stream.

    Returns:
        Estimate of frequency of element 9050.
        Estimate for the number of heavy hitters.
    """
    def is_heavy_hitter(el: int) -> bool:
        return sketch.Count(el) >= 841

    for element in generate_stream(sType):
        sketch.Increment(element)

    estimateOf9050 = sketch.Count(9050)
    heavyHitters = [el for el in range(1, 9051)
                    if is_heavy_hitter(el)]
    return estimateOf9050, len(heavyHitters)
```

[202]:
```
def simulate_trials_for_stream(
    sType: StreamType, isConservative: bool, trials: int=10) -> Tuple[float,␣
↪float]:
    """Simulates multiple trials for the stream type.

    Returns:
        Average estimate of frequency of element 9050.
```

```
        Average estimate for the number of heavy hitters.
    """
    results = [simulate_stream(
        CountMinSketch(trial, isConservative=isConservative), sType)
     for trial in range(1, trials +1)]
    estimatesOf9050, numHeavies = zip(*results)
    return np.mean(estimatesOf9050), np.mean(numHeavies)
```

[207]:
```
def problem2c():
    for sType in StreamType:
        meanFreq9050, meanNumHeavies = simulate_trials_for_stream(
            sType, isConservative=False)
        print("For %s Stream, the mean estimate for the frequency "
              "of 9050 is %s, while the mean estimate of the number of "
              "heavy hitters is %s." % (sType.name, meanFreq9050,␣
 ↪meanNumHeavies))
```

[208]:
```
def problem2f():
    for sType in StreamType:
        meanFreq9050, meanNumHeavies = simulate_trials_for_stream(
            sType, isConservative=True)
        print("[Conservative] For %s Stream, the mean estimate for the␣
 ↪frequency "
              "of 9050 is %s, while the mean estimate of the number of "
              "heavy hitters is %s." % (sType.name, meanFreq9050,␣
 ↪meanNumHeavies))
```

[209]:
```
problem2c()
```

```
For FORWARD Stream, the mean estimate for the frequency of 9050 is 2645.7, while
the mean estimate of the number of heavy hitters is 24.6.
For REVERSE Stream, the mean estimate for the frequency of 9050 is 2645.7, while
the mean estimate of the number of heavy hitters is 24.6.
For RANDOM Stream, the mean estimate for the frequency of 9050 is 2645.7, while
the mean estimate of the number of heavy hitters is 24.6.
```

[210]:
```
problem2f()
```

```
[Conservative] For FORWARD Stream, the mean estimate for the frequency of 9050
is 2577.2, while the mean estimate of the number of heavy hitters is 23.4.
[Conservative] For REVERSE Stream, the mean estimate for the frequency of 9050
is 2500.0, while the mean estimate of the number of heavy hitters is 22.2.
[Conservative] For RANDOM Stream, the mean estimate for the frequency of 9050 is
2500.0, while the mean estimate of the number of heavy hitters is 22.2.
```

[ ]: