hw1

January 14, 2025

1 Problem 1: The power of two choices

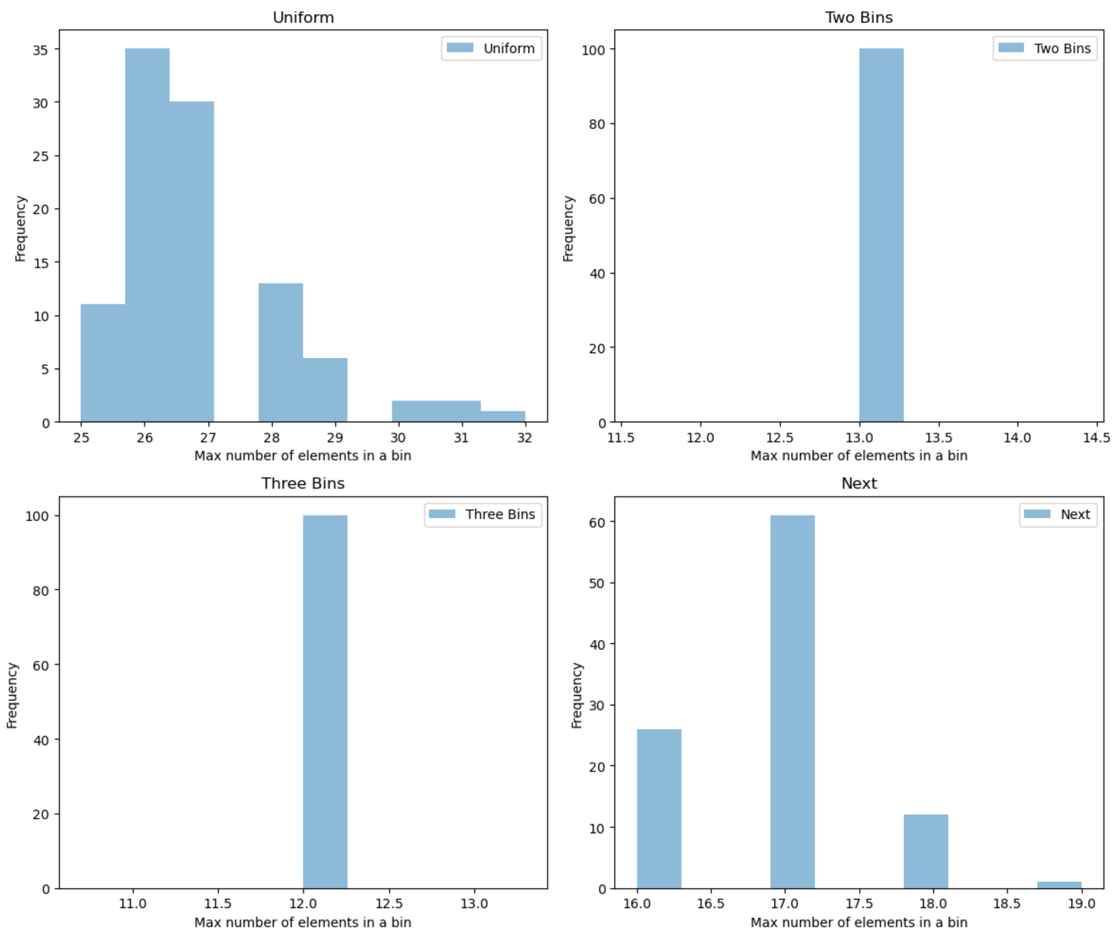
[3]:
```python
import numpy as np
import matplotlib.pyplot as plt
import pandas
from tqdm import tqdm
import torch
import hashlib
from sortedcontainers import SortedDict
from collections import Counter
import pandas as pd
from tabulate import tabulate

DEVICE = torch.device('cuda' if torch.cuda.is_available() else 'cpu')
```

[33]:
```python
import numpy as np

def uniform_sample(m: int, n: int) -> int:
    bins = np.zeros(n)
    for _ in range(m):
        sample = np.random.randint(0, n)
        bins[sample] += 1
    return int(bins.max())

def two_bins_sample(m: int, n: int) -> int:
    bins = np.zeros(n)
    for _ in range(m):
        list1 = np.random.randint(0, n)
        list2 = np.random.randint(0, n)
        if bins[list1] < bins[list2]:
            bins[list1] += 1
        else:
            bins[list2] += 1
    return int(bins.max())

def three_bins_sample(m: int, n: int) -> int:
    bins = np.zeros(n)
    for _ in range(m):
        list1 = np.random.randint(0, n)
        list2 = np.random.randint(0, n)
        list3 = np.random.randint(0, n)
        if bins[list1] <= bins[list2] and bins[list1] <= bins[list3]:
            bins[list1] += 1
        elif bins[list2] <= bins[list1] and bins[list2] <= bins[list3]:
            bins[list2] += 1
        else:
            bins[list3] += 1
    return int(bins.max())

def next_sample(m: int, n: int) -> int:
    bins = np.zeros(n)
    for _ in range(m):
        index = np.random.randint(0, n)
        next_index = (index + 1) % n
        if bins[index] < bins[next_index]:
            bins[index] += 1
        else:
            bins[next_index] += 1
    return int(bins.max())
```

[34]:
```
from tqdm import tqdm  # Import tqdm for progress bars

sample_size = 100
m = 1000000
n = 100000

print("doing uniform: ")
uniform_results = []
for _ in tqdm(range(sample_size)):
    uniform_results.append(uniform_sample(m, n))

print("doing two bins: ")
two_bins_results = []
for _ in tqdm(range(sample_size)):
    two_bins_results.append(two_bins_sample(m, n))

print("doing three bins: ")
three_bins_results = []
for _ in tqdm(range(sample_size)):
    three_bins_results.append(three_bins_sample(m, n))

print("doing next: ")
next_results = []
for _ in tqdm(range(sample_size)):
    next_results.append(next_sample(m, n))
```

```
doing uniform:

100%|      | 100/100 [02:10<00:00,  1.31s/it]

doing two bins:

100%|      | 100/100 [04:20<00:00,  2.61s/it]

doing three bins:

100%|      | 100/100 [06:23<00:00,  3.83s/it]

doing next:

100%|      | 100/100 [02:23<00:00,  1.43s/it]
```

[49]:
```
def plot():
    import numpy as np

    # Combine all plots into one figure
    plt.figure(figsize=(12, 10))

    # Function to determine bins
    def get_bins(data, num_bins=10):
        if len(set(data)) == 1:  # Constant data
            unique_value = data[0]
            # Create centered bins around the constant value
            half_bin_width = (unique_value + 1) / num_bins
            return np.linspace(unique_value - half_bin_width, unique_value +␣
↪half_bin_width, num_bins + 1)
        return num_bins

    # Uniform Results
    plt.subplot(2, 2, 1)
    plt.hist(uniform_results, bins=get_bins(uniform_results), alpha=0.5,␣
↪label='Uniform', align='mid')
    plt.xlabel('Max number of elements in a bin')
    plt.ylabel('Frequency')
    plt.title('Uniform')
    plt.legend()

    # Two Bins Results
    plt.subplot(2, 2, 2)
    plt.hist(two_bins_results, bins=get_bins(two_bins_results), alpha=0.5,␣
↪label='Two Bins', align='mid')
    plt.xlabel('Max number of elements in a bin')
    plt.ylabel('Frequency')
    plt.title('Two Bins')
    plt.legend()
    # Three Bins Results
    plt.subplot(2, 2, 3)
    plt.hist(three_bins_results, bins=get_bins(three_bins_results), alpha=0.5,␣
 ↪label='Three Bins', align='mid')
    plt.xlabel('Max number of elements in a bin')
    plt.ylabel('Frequency')
    plt.title('Three Bins')
    plt.legend()

    # Next Results
    plt.subplot(2, 2, 4)
    plt.hist(next_results, bins=get_bins(next_results), alpha=0.5,␣
 ↪label='Next', align='mid')
    plt.xlabel('Max number of elements in a bin')
    plt.ylabel('Frequency')
    plt.title('Next')
    plt.legend()

    # Adjust layout
    plt.tight_layout()
    plt.show()


plot()
```



2 Problem 2: Count-min sketch

```
[26]: class CountMinSketch:
          def __init__(self, seed: int, n: int, eps: float):
              self.seed = seed
              self.b = 256  # Number of columns
              self.l = 6    # Number of hash functions (rows)
              self.n = n    # Number of elements
              self.eps = eps  # Epsilon for heavy hitters
              self.counts = np.zeros((self.l, self.b), dtype=int)  # 2D array for␣
       ↪counts
              self.heavy_hitters = set()  # Set to track heavy hitters

          def hash(self, key: str) -> np.ndarray:
              """Hash a key using MD5 and return a NumPy array of hash values."""
              hash_val = hashlib.md5((key + str(self.seed)).encode('utf-8')).digest()
```

```
        return np.array([hash_val[i] % self.b for i in range(self.l)],␣
↪dtype=int)

    def inc(self, key: str):
        """Increment the counts for the given key."""
        hashes = self.hash(key)  # Get hash values
        indices = np.arange(self.l)

        # for i,h in enumerate(hashes):
        #     self.counts[i,h] += 1

        # Find the minimum count across hash functions for the given key
        min_count = self.counts[indices, hashes].min()

        for i,h in enumerate(hashes):
            if self.counts[i,h] == min_count:
                self.counts[i,h] += 1

        # Check for heavy hitter
        if min_count >= self.eps * self.n:
            self.heavy_hitters.add(key)

    def count(self, key: str) -> int:
        """Return the estimated count of a key."""
        hashes = self.hash(key)  # Get hash values
        return self.counts[np.arange(self.l), hashes].min()  # Minimum count␣
↪across hash functions
```

[27]:
```
def generate_dataset(n: int):
    dataset = []
    # For i = 1 to n, add i (as a string) i^2 times
    for i in range(1, n + 1):
        dataset.extend([str(i)] * (i ** 2))
    # For i = n+1 to n^2, add i (as a string) once
    dataset.extend(map(str, range(n + 1, n**2 + 1)))
    return dataset

def count_min_sketch_experiment(dataset: list, seed: int):
    count_min_sketch = CountMinSketch(seed, len(dataset), 0.01)
    for key in dataset:
        count_min_sketch.inc(key)
    return count_min_sketch
```

[28]:
```
n = 150

heavy_first_freq100 = np.zeros(10)
heavy_last_freq100 = np.zeros(10)
```

```
random_freq100 = np.zeros(10)

heavy_first_heavyhitters = np.zeros(10)
heavy_last_heavyhitters = np.zeros(10)
random_heavyhitters = np.zeros(10)

for i in tqdm(range(10)):
    dataset = generate_dataset(n)
    heavy_first = sorted(dataset, key=Counter(dataset).get, reverse=True)
    heavy_last = sorted(dataset, key=Counter(dataset).get)
    random_perm = np.random.permutation(dataset)

    count_heavy_first = count_min_sketch_experiment(heavy_first, i)
    count_heavy_last = count_min_sketch_experiment(heavy_last, i)
    count_random = count_min_sketch_experiment(random_perm, i)

    heavy_first_freq100[i] = count_heavy_first.count('100')
    heavy_last_freq100[i] = count_heavy_last.count('100')
    random_freq100[i] = count_random.count('100')

    heavy_first_heavyhitters[i] = len(count_heavy_first.heavy_hitters)
    heavy_last_heavyhitters[i] = len(count_heavy_last.heavy_hitters)
    random_heavyhitters[i] = len(count_random.heavy_hitters)
```

```
100%|      | 10/10 [03:05<00:00, 18.54s/it]
```

[29]:
```
# Calculate averages and prepare the table
data = {
    "Category": ["Heavy First", "Heavy Last", "Random"],
    "Average Frequency of 100": [
        heavy_first_freq100.mean(),
        heavy_last_freq100.mean(),
        random_freq100.mean(),
    ],
    "Average Number of Heavy Hitters": [
        heavy_first_heavyhitters.mean(),
        heavy_last_heavyhitters.mean(),
        random_heavyhitters.mean(),
    ],
}

# Create a DataFrame for the table
df = pd.DataFrame(data)
# Convert to Markdown
markdown_table = tabulate(df, headers="keys", tablefmt="pipe", showindex=False,␣
  ↪floatfmt=".2f")
```

```python
# Save to a Markdown file
with open("table.md", "w") as f:
    f.write(markdown_table)

print("Markdown table saved to 'table.md'")
```

```
Markdown table saved to 'table.md'
```

Unoptimized Table: | Category | Average Frequency of 100 | Average Number of Heavy Hitters | |:————|——————————:|———————————-:| | Heavy First | 10082.00 | 43.30 | | Heavy Last | 10082.00 | 43.00 | | Random | 10082.00 | 43.30 |

Optimized Table: | Category | Average Frequency of 100 | Average Number of Heavy Hitters | |:———————|——————————:|———————————-:| | Heavy First | 10000.00 | 43.20 | | Heavy Last | 10033.80 | 43.00 | | Random | 10000.00 | 43.00 |

CSE 422 HW1

Rohan Mukherjee

January 14, 2025

1. (a) I have written it.

   (b) The pro of the first strategy is that it is the computationally fastest of all of them. It needs to make only one random call and has no if statements in it. However, all the other methods are theoretically better than this method, since they sample a uniform bin, but compare it with at least 1 other bin and puts the ball in the bin with fewer balls.

   The second method and fourth method are similar, except I believe the second method is more expensive. But it also would theoretically perform the best: see the last part. The next method saves by not having to make another random call.

   This third method should just be a generalization of the second method, but with a third random call and now a more if statements since it has to find which bin has the fewest elements. So this is the best method, but it would also be the most expensive to implement or do at large scale.

   Here is a plot:

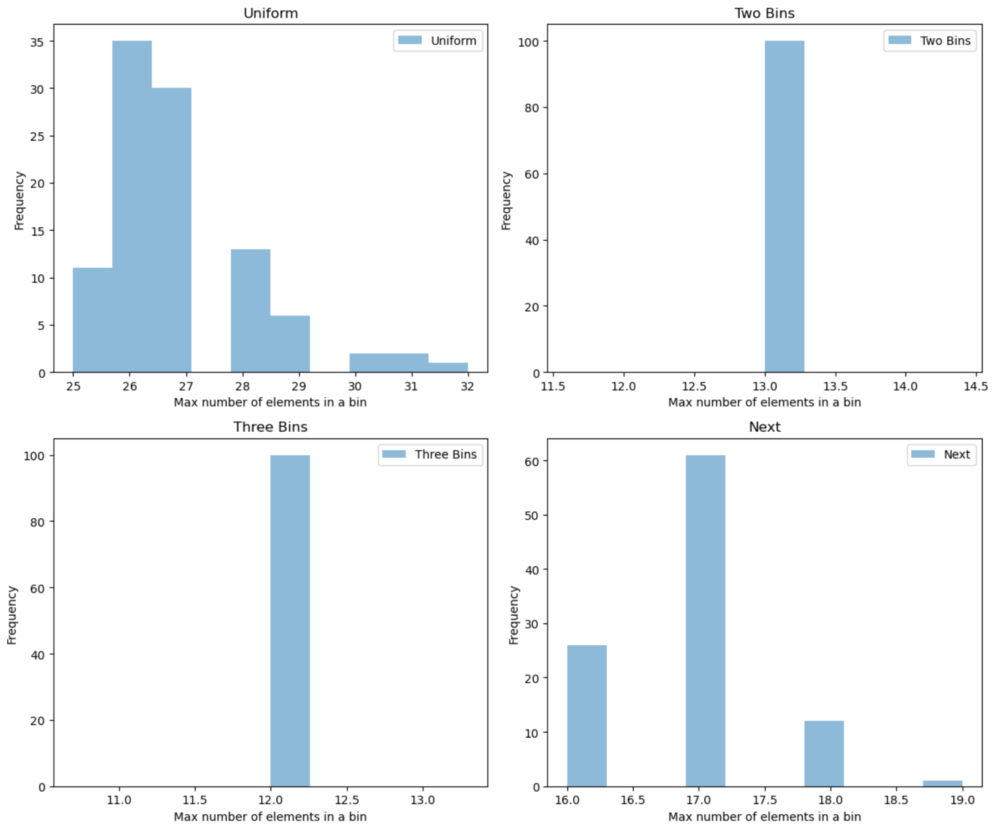


Figure 1: Histogram of the results

(c) To model consistent hashing as a balls-into-bins problem, we can think of the servers as the bins and the keys as the balls. What we are doing right now is hashing the bins and putting them on a circle, and hashing the keys, and putting them on a circle too, and putting all the keys that fall in the clockwise arc connecting two bins into the first bin. Right now, without having servers go out, this is equivalent to just uniformly sampling a bin to put the key in, assuming the hashing function is ”sufficiently random”. Since in expectation all the arcs have the same length and the hash function should hash to a random point on the circle uniformly.

However, the previous homework questions showed us that uniformly sampling is not the best option. From the histogram (and from part e), it seems like the best option is where we uniformly sample 3 bins and then put the ball in the one that has the smallest number of balls. For the 3 bins sampled method, to make this back into the consistent hashing problem, we could have multiple hash functions that hash the key multiple times, which is like putting the key into multiple random arcs, and putting the ball in the arc (server) with fewest balls.

This idea can be extended to the others as well. The 2-bin version is just a simplification of this. The uniform is what we are doing right now, and the next sampling method is just hash the key once, and check the arc it falls in and the arc that is after it clockwise.

(d) I do not think that the first strategy, just uniformly sampling, can be represented in this graph way. This is because there is no consideration of which bin has fewer balls, since it doesn't care about that and just does it randomly. Unless we could somehow use an empty graph with n points that are not connected to each other, but this doesn't have any edges, so I don't think so.

On the other hand, the second strategy can be represnted in this way. Consider the complete graph K_n. Then picking a random edge will just uniformly pick two bins, and we will put it in the one with fewer balls. This is precisely the second strategy.

The third strategy, which compares 3 bins, is not possible, because we would need to consider 3 bins, not two, but there is only 2 vertices for each edge.

The last strategy, which I call the "next" strategy, is easier to represent on a graph. This is just the cycle graph C_n. When we select an edge $(i, i+1)$, it compares i to $i+1$ and puts it in the bin with fewer balls.

(e) I believe that some graphs do better than others based on how "connected" they are. For example, in the worst possible case where we only have 1 edge connecting two vertices, we know that the max load will be huge, since all the load goes on those two vertices. On the other hand, for cyclic graph like the next method does, we know that it can only compare bins that are 1 away from each other. Intuitively, it seems that if we had a high load in a bin, then the nearby bins would also have high load. But in this case we can only look at one other bin once we have drawn the high load bin, which would mean that these bins combined must have high load. The opposite of this strategy is the most connected graph K_n, where once we have picked a bin, we can with equal probability see any other bin, which gives us a lot more options to offload the weight than just the next bin.

(f) From this connectedness idea, I looked up what is the best 3-regular connected graph, and it seems like expander graphs do the job well. So I would pick a 3-regular expander graph.

2. (a) It has been done.

(b) If we wanted to have a heavy hitter when we have more than 1% of the dataset, that is equivalent to $0.01n$ frequencies in the dataset. So we need to find which elements show up more than $0.01n$ times.

(c) This is the table I get:

Category	Average Frequency of 100	Average Number of Heavy Hitters
Heavy First	10082.00	43.30
Heavy Last	10082.00	43.00
Random	10082.00	43.30

Table 1: Unoptimized Results

As you can see, they all the get exact same frequency of 100s, and they have a very similar number of heavy hitters, up to variance of 0.3. There is absolutely no variance in the frequency of 100s, since the hash function is the same for the 3 different datasets. Once you have fixed the hash function, for any permutation of the dataset, you will hash the keys in a different order to the same place, which leads to the same counts.

On the other hand, variance in the heavy hitters can be explained by collisions. Suppose that for some number in the dataset, say x that is more than 1% of the dataset, we had another number that gets hashed to the same place. If this happens before we have counted all the x's, such as in the heavy last, this won't matter because it won't occupy the same space as a heavy hitter until after we have counted all of the low-frequency ones. However, when the order is changed, such as in heavy first/random, we could see a low-frequency collision after x takes more than 1% of the dataset, which would add to the number of heavy hitters.

(d) We prove this statement by induction on the frequency of a fixed element x in the dataset. For the first time, when we hash x and are supposed to put it in the counts matrix, we only update the subset of counters for which the count is minimal. Either all counts are 0, in which case they will all be updated, or some are 0 and others aren't, in which case after updating those that are 0, all the counters corresponding to the hash of x will be greater than or equal to 1.

Now suppose that the we have counted the $n-1$th frequency of x, and all the counters are $\geq n-1$. On the nth encounter of x, either there are some counters that are $n-1$ or not. When there are not, all counters are already bigger than n and we are done. On the other hand, since $n-1$ is the minimum the counter can be, we will update all the counters that are $n-1$ to n. Then in this case all the counters coresponding to x are $\geq n$, and we are again done.

(e) This is wht I get for the conservative updates:

Category	Average Frequency of 100	Average Number of Heavy Hitters
Heavy First	10000.00	43.20
Heavy Last	10033.80	43.00
Random	10000.00	43.00

Table 2: Optimized Results

As before, the average number of heavy hitters is expected to have some variance since low-frequency collisions can happen and will update the set of heavy hitters with fake ones. However, the aaverage frequency of 100 is now different too. This is because the conservative update optimization, which updates only the minimum counters, adds an order dependence to the counts. For example, if we process everything before the number 100, then probably all the counters associated with the hash of 100 will already have some counts, and so when we add the 100^2 100s, the minimum count will be higher than if we had processed the 100s first. On the other hand, if we process all the 100^2 100s first, then all the counters associated with 100 will go straight to 10,000. When low-frequency collisions happen, the 10,000 will ALWAYS be the highest count among them, and hence will not be updated. This explains why the average frequency count for 100 is higher in heavy last, and why it is low in heavy first. For random, you are certainly likely to see lots of the 10,000 100s before seeing any collisions with 100, so the same logic applies but at a lower scale than with heavy last.