CS168, Spring 2024

Mini-Project #1

Due by 11am Thursday, April 11th.

- You can work with up to three partners (groups of at most 4 people). If you work in a group, please submit one assignment via Gradescope (with all group members' names).
- Please submit a single pdf for your group via Gradescope, and add the names of all members of your group in Gradescope. Detailed submission instruction can be found on the course website (`https://web.stanford.edu/class/cs168/`) under "Coursework - Assignments" section.
- Use 12pt or higher font for your writeup.
- Make sure the plots you submit are easy to read at a normal zoom level.
- If you've written code to solve a certain part of a problem, or if the part explicitly asks you to implement an algorithm, you must also include the code in your pdf submission.
- Code marked as "Deliverable" should be pasted into the relevant section, code not marked as deliverables should be included at the very end of the pdf. Keep variable names consistent with those used in the problem statement, and with general conventions. No need to include import statements and other scaffolding, if it is clear from context. Use the `verbatim` environment to paste code in LaTeX.

```
def example():
    print "Your code should be formatted like this."
```

- **Reminder:** No late assignments will be accepted, but we will drop your lowest mini-project grade when calculating your final grade.

Part 1: The Power of Two Choices

Goal: The goal of this part of the assignment is to gain an appreciation for the unreasonable effectiveness of simple randomized load balancing, and measure the benefits of some lightweight optimizations.

Description: We consider random processes of the following type: there are N bins, and we throw N balls into them, one by one. [This is an abstraction of the sort of allocation problem that arises throughout computing—e.g. allocating tasks on servers, routing packets within parallel networks, etc..] We'll compare four different strategies for choosing the bin in which to place each ball.

1. Select one of the N bins uniformly at random, and place the current ball in it.
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.
3. Same as the previous strategy, except choosing three bins at random rather than two.
4. Select two bins as follows: the first bin is selected uniformly from the first $N/2$ bins, and the second uniformly from the last $N/2$ 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.

(a) (5 points) Write code to simulate strategies 1–4. For each strategy, there should be a function that takes the number N of balls and bins as input, simulates a run of the corresponding random process, and outputs the number of balls in the most populated bin (denoted by X below). Before running your code, try to guess how the above schemes will compare to eachother.

(b) (10 points) Let $N = 200,000$ and simulate each of the four strategies 30 times. For each strategy, plot the histogram of the 30 values of X.[1] Discuss the pros and cons of the different strategies, both in terms of computation time and the value of X. [As with many of the mini-projects, there is no single "right answer" to this question. Instead, the idea is to have you think about the processes and your experiments, and draw reasonable conclusions from this analysis.]

(c) (5 points) What does the first of the random processes above say about the standard implementation of hashing N elements into a hash table with N buckets, and resolving collisions via chaining (i.e., one linked list per bucket). Discuss in particular any relationships between X and search times in the hash table.

(d) (5 points) Do the other random processes suggest alternative implementations of hash tables with chaining? **Propose three alternative hash table implementations based on these processes, and in particular describe how insertion and search is done in each.** Briefly discuss the trade-offs between the different hash table implementations that you propose (e.g., in terms of insertion time vs. search time).

(e) (0 points) [Food for thought] Are there other schemes that might get an even lower maximum bucket load while requiring even less computation per ball? If so, how would you design a hash function based on those ideas?

Deliverables: Your code for part (a); your histograms for part (b); your written answers to parts (b), (c) and (d).

Part 2: Conservative Updates in a Count-Min Sketch

Goal: The goal of this part is to understand the count-min sketch (from Lecture #2) via an implementation, and to explore the benefits of a "conservative updates" optimization. (The secondary goal of this part is to give you a sense of the sort of coding that you might need to do during this course :)

Description: You'll use a count-min sketch with 4 independent hash tables, each with 256 counters. You will run 10 independent trials. This lets you measure not only the accuracy of the sketch, but the distribution of the accuracy over multiple datasets with the same frequency distribution. Your sketch should take a "trial" as input, and the hash value of an element x during trial i ($i = 1, 2, \ldots, 10$) for table j ($j = 1, 2, 3, 4$) is calculated as follows:

- Consider the input x as a string, and append $i - 1$ as a string to the end of the string representing x.
- Calculate the MD5 score of the resulting string.[2] Do not implement the MD5 algorithm yourself; most modern programming languages have packages that calculate MD5 scores for you. For example, in Python 3, you can use the `hashlib` library and `hashlib.md5(foo.encode('utf-8')).hexdigest()` to compute the MD5 score of the string foo (returning a hexadecimal string).
- The hash value is the j-th byte of the score.

As an example, to compute the hash value of 100 in the 4th table of the 9th trial, we calculate the MD5 score of the string "1008," which is (in hexadecimal):

15 87 96 5f b4 d4 b5 af e8 42 8a 4a 02 4f eb 0d

[1] For example, in Python, the `numpy` and `matplotlib.pyplot` libraries are useful for creating histograms. See the course Web site for some starter code you can use and modify as necessary.

[2] See `http://en.wikipedia.org/wiki/MD5`.

The 4th byte is 5f in hexadecimal, which is 95 in decimal. In Python, you can parse the hexadecimal string 5f with `int("5f", 16)`.

(a) (5 points) Implement the count-min sketch, as above.

You will be feeding data streams (i.e., sequences of elements) into count-min sketches. Every element of each stream is an integer between 1 and 9050 (inclusive). The frequencies are given by:

- Integers $1000 \cdot (i-1) + 1$ to $1000 \cdot i$, for $1 \leq i \leq 9$, appear i times in the stream. That is, the integers 1 to 1000 appear once in the stream; 1001 to 2000 appear twice; and so on.
- An integer $9000 + i$, for $1 \leq i \leq 50$, appears i^2 times in the stream. For example, the integer 9050 appears 2500 times.

(Each time an integer appears in the stream, it has a count of 1 associated with it.)

(b) (2 points) Call an integer a *heavy hitter* if the number of times it appears is at least 1% of the total number of stream elements. How many heavy hitters are there in a stream with the above frequencies?

Next, you will consider 3 different data streams, each corresponding to the elements above in a different order.

1. Forward: the elements appear in non-decreasing order.
2. Reverse: the elements appear in non-increasing order.
3. Random: the elements appear in a random order.

(c) (6 points) For each of the three data streams, feed it into a count-min sketch (i.e., successively insert its elements), and compute the values of the following quantities, averaged over the 10 trials, for each order of the stream:

   - The sketch's estimate for the frequency of element 9050.
   - The sketch's estimate for the number of heavy hitters (elements with estimated frequency at least 1% of the stream length).

   Record the mean estimate for each of the three orders. Does the order of the stream affect the estimated counts? Explain your answer.

(d) (3 points) Implement the *conservative updates* optimization, as follows. When updating the counters during an insert, instead of incrementing all 4 counters, we only increment the subset of these 4 counters that have the lowest current count (if two or more of them are tied for the minimum current count, then we increment each of these).

(e) (3 points) Explain why, even with conservative updates, the count-min sketch never underestimates the count of a value.

(f) (6 points) Repeat part (c) with conservative updates. Are you surprised by the performance boost? Why or why not?

Deliverables: Your code for parts (a) and (d); your written answers to parts (b), (c), (e), and (f).