Problem Set 4

Both theory and programming questions are due **October 2** at **11:59PM**. Please submit your solutions on Canvas, as two separate files: one image containing your write-up, and one zip containing your programming solutions.

We will provide the solutions to the problem set after the problem set is due, which you will use to find any errors in the proof that you submitted. You will need to submit a critique of your solutions by October 9. Your grade will be based on both your solutions and your critique of the solutions.

Collaborators: None.

Problem 4-1. [35 points] **Hash Functions and Load**

(a) Imagine that an algorithm requires us to hash strings containing English phrases. Knowing that strings are stored as sequences of characters, Alyssa P. Hacker decides to simply use the sum of those character values (modulo the size of her hash table) as the string's hash. Will the performance of her implementation match the expected value shown in lecture?

1. Yes, the sum operation will space strings out nicely by length.
2. Yes, the sum operation will space strings out nicely by the characters they contain.
3. No, because reordering the words in a string will not produce a different hash.
4. No, because the independence condition of the simple uniform hashing assumption is violated.

Answer:

(b) Alyssa decides to implement both collision resolution and dynamic resizing for her hash table. However, she doesn't want to do more work than necessary, so she wonders if she needs both to maintain the correctness and performance she expects. After all, if she has dynamic resizing, she can resize to avoid collisions; and if she has collision resolution, collisions don't cause correctness issues. Which statement about these two properties true?

1. Dynamic resizing alone will preserve both properties.
2. Dynamic resizing alone will preserve correctness, but not performance.
3. Collision resolution alone will preserve performance, but not correctness.
4. Both are necessary to maintain performance and correctness.

Answer:

(c) Suppose that Alyssa decides to implement resizing. If Alyssa is enlarging a table of size m into a table of size m', and the table contains n elements, what is the best time complexity she can achieve?

1. $O(m)$
2. $O(m')$
3. $O(n)$
4. $O(nm')$
5. $O(m + m')$
6. $O(m + n)$
7. $O(m' + n)$

Answer:

(d) In lecture, we discussed doubling the size of our hash table. Ivy H. Crimson begins to implement this approach (that is, she lets $m' = 2m$) but stops when it occurs to her that she might be able to avoid wasting half of the memory the table occupies on empty space by letting $m' = m + k$ instead, where k is some constant. Does this work? If so, why do you think we don't do it? There is a good theoretical reason as well as several additional practical concerns; a complete answer will touch on both points.

Answer:

Problem 4-2. [10 points] **Python Dictionaries**

We're going to get started by checking out a file from Python's Subversion repository at svn.python.org, whose link is below. The Python project operates a web frontend to their version control system, so we'll be able to do this using a browser.

Visit http://svn.python.org/projects/python/trunk/Objects/dictnotes.txt.

These are notes prepared by contributors to the Python project, as they currently exist in the Python source tree. (Cool! Actually, this document is a fascinating read–and you should be able to understand most of it.) Read over the seven use cases defined at the top of this document.

(a) Let's examine the "membership testing" use case. Which statement accurately describes this use case?

1. Many insertions right after creation, and then mostly lookups.
2. Many insertions right after creation, and then only lookups.
3. A workload of evenly-mixed insertions/deletions and lookups.
4. Alternating rounds of insertions/deletions and lookups.

Answer:

(b) Now imagine that you have to pick a hash function, size, collision resolution strategy and so forth (all of the characteristics of a hash table that we've seen so far) in order to make a hash table perfectly suited to this use case alone. Pick the statement that best describes the choices you might make.

1. A large minimum size and a growth rate of 2.
2. A small minimum size and a growth rate of 2.
3. A large minimum size and a growth rate of 4.
4. A small minimum size and a growth rate of 4.

Answer:

Problem 4-3. [55 points] **Matching DNA Sequences**

The code and data used in this problem are in the `dist` directory of the homework download. Take a look at `README.txt` for some instructions.

Ben Bitdiddle has recently moved to the Kendall Square area, which is full of biotechnology companies and their shiny, window-laden office buildings. While mocking their dorky lab coats makes him feel slighly better about himself, he is secretly envious, and so he sets out to earn one of his very own. To pick up the necessary geek cred, he begins experimenting with DNA-matching technologies.

Ben would like to create mutants to do his bidding, and to get started, he'd like to know how closesly related the creatures he's collected are. If two sequences contain mostly the same subsequences in mostly the same places, then they're likely closely related; if they don't, they probably aren't. (This is, of course, a gross oversimplification.)

For our purposes, we'll represent a DNA sample as a sequence of characters. (These characters will all be upper-case. You can look at the Wikipedia page on nucleotides for a list of code characters and their meanings.) These sequences are very long, so comparing subsequences of them quickly is important. We've provided code in `kfasta.py` that reads the .fa files storing this data.

(a) Let's start with `subsequenceHashes`, which returns all length-k subsequences and their hashes (and perhaps other information, if there's anything else you might find useful). You'll want to use the `RollingHash` implementation provided in `dnaseqlib.py`.

Hint: There will be many subsequences; the DNA sequences are tens of millions of nucleotides long. To avoid keeping them all in memory at once, implement your function as a generator. See the Python reference materials available online (or search "Python generators") if you aren't familiar with this construct.

(b) Implement `Multidict` and verify that your work passes the simple sanity tests provided.

`Multidict` should behave just like a Python dictionary, except that it can store multiple values per key. If no values exist for a key, it returns an empty list; otherwise, it

returns the list of associated values. You may (and probably should) use the Python dictionary in your implementation.

(c) Now it's time to implement `getExactSubmatches`. Again, implementing this function as a generator is probably a good idea–you will have many, many matches. Much of the work has already been done by `Multidict` and `subsequenceHashes`; with these completed, the implementation of `getExactSubmatches` does not need to be very complex (my implementation is 9 lines long).

This function should return pairs of offsets into the inputs. A tuple (x, y) being returned indicates that the k-length subsequence starting at index x of the first input matches the one starting at index y of the second input.

A simple sanity test has been provided in `test_dnaseq.py`. `README.txt` describes how to run this test.

In the next part, you will run tests that take around 30 minutes each. If you want to test on shorter runs first, then test using the shorter sequences provided in the `data` folder. Follow instructions in `README.txt` to run said tests.

(d) Run comparisons between the two human samples (maternal and paternal) and between the paternal sample and the chimp sample, and compare the outputs to the expected outputs. These tests will take 15-30 minutes each to run.

Feel free to peak at the image-generation code in `libdnaseq.py` to see how it works. What it's doing is keeping track of how many of our (x, y) matches land in each bin in a two-dimensional grid of bins, each of which corresponds to a pixel in the output image. At the end, it normalizes the match counts in each bin, so the largest number of matches is black and zero matches is white.

What should a perfect match (i.e. comparing a sequence to itself) look like? Hint: if $x = y$, then (x, y) should be a match. Test your results by checking the output when you compare one of the data files to itself.

These tests take a while with $O(1)$ lookup time... Imagine how long they would take if this was implemented with, say, a binary tree with logarithmic lookup time! There are tens of millions (i.e. about 2^{25}) elements in each sequence, so runs would take somewhere on the scale of 12 hours! This is, of course, likely a very bad estimate, as it does not account for any of the constant factor differences between the implementations.

When you're done, submit your writeup and your `dnaseq.py` as two separate files on Canvas (do not zip them up and submit them together).