Problem Set 4

Both theory and programming questions are due **Friday, 14 October** at **11:59PM**. Please submit your solutions on Gradetacular at `http://alg.csail.mit.edu`. Gradetacular will prompt you for your answers to the questions, so you do not need to create a solution template. The text on Gradetacular is the authoritative copy of the problem set.

Remember that for the written response question, your goal is to communicate. Full credit will be given only to a correct solution which is described clearly. Convoluted and obtuse descriptions might receive low marks, even when they are correct. Also, aim for concise solutions, as it will save you time spent on write-ups, and also help you conceptualize the key idea of the problem.

We will provide the solutions to the problem set 10 hours 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 **Thursday, October 20th, 11:59PM**. Your grade will be based on both your solutions and your critique of the solutions.

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.

Solution: No, because reordering the words in a string will not produce a different hash.

(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.

Solution: Both are necessary to maintain performance and correctness. Without collision resolution, no correctness: could have an actual hash collision, and then no amount of resizing will let both be entered into the table. Without dynamic resizing, the load factor will get large, and everything will turn into a linear-time lookup (assuming chaining).

(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. $\Theta(m)$
2. $\Theta(m')$
3. $\Theta(n)$
4. $\Theta(nm')$
5. $\Theta(m + m')$
6. $\Theta(m + n)$
7. $\Theta(m' + n)$

Solution: $\Theta(m' + n)$. It takes $O(m')$ time to create a new hash table (allocating the memory can take constant time, but it then needs to be initialized). It takes $O(m + n)$ time to go through each slot in the old table and copy each item. In total, it comes out to $\Theta(m' + m + n)$, but since $m < m'$, the answer is just $\Theta(m' + n)$.

(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.

Solution: Theoretically, our cost will now be $O(n)$ even after amortization. Loosely speaking, we were able to achieve $O(1)$ amortized cost because we performed an $O(n)$ time operation every $O(n)$ step. Now, however, we're performing this $O(n)$ operation every $O(1)$ steps. Practically, the computer will play more nicely with operations based around doubling (doubling is a fast operation, allocating memory blocks of sizes that are powers of two has plenty of advantages, etc).

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. 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 actual 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 identified 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.

Solution: Many insertions right after creation, and then mostly lookup.

(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 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.

Solution: Membership testing can benefit from highly sparse hash tables, so let's set growth factor to 4. We're going to be inserting a bunch of things up front, so let's start with a larger minimum size.

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

The code and data used in this problem are available on the course website. Please take a peek at the README.txt for some instructions.

Ben Bitdiddle has recently moved into 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 slightly better about himself, he is secretly jealous, 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 closely 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).

Hint: There will likely be many of these matches; 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 for details if you aren't familiar with this important language 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`. Ignore the parameter m for the time being; we'll get to that in the next part. Again, implementing this function as a generator is probably a good idea. (You will probably have many, many matches– think about the combinatorics of the situation briefly.) As a hint, consider that much of the work has already been done by `Multidict` and `subsequenceHashes`; also take a peek at the `RollingHash` implementation we've given you. With these building blocks, your solution probably does not need to be very complex (or more than a few lines).

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

We've provided a simple sanity test; your solution should be correct at this point (that is, `dnaseq.py` will produce the right output) but it'll probably be too slow to be useful. If you like, you can try running it on the first portion of two inputs; we've provided two such prefixes (the short files in the data directory) that might be helpful.

(d) The most significant reason why your solution is presently too slow to be useful is that you are hashing and inserting into your hash table tens of millions of elements, and then performing tens of millions of lookups into that hash table. Implement `intervalSubsequenceHashes`, which returns the same thing as `subsequenceHashes` except that it hashes only one in m subsequences. (A good implementation will not do more work than is necessary.) Modify your implementation of `getExactSubmatches` to honor m only for sequence A. Consider why we still see approximately the same result, and why we can't further improve performance by applying this technique to sequence B as well.

(e) Run comparisons between the two human samples (paternal and maternal) and between the paternal sample and each of the animal samples.

Feel free to take a peek at how the image-generation code works. Conceptually, what it's doing is keeping track of how many of your (x, y) match tuples land in each of a two-dimensional grid of bins, each of which corresponds to a pixel in the output image. At the end, it normalizes the counts so that the highest count observed is totally black and an empty bin is white.

Think for a second about what a perfect match (e.g., comparing a sequence to itself) should look like. Try comparing the two human samples you have (maternal and paternal), one of the humans against the chimp sample, and then against the dog sample. Make sure your results make sense!

We've posted what our reference solution produced for the human-human comparison, the human-chimp comparison, and the human-dog comparison.

Please submit the code that you wrote. (You should only have had to modify dnaseq.py, so that's all you need to submit.)