Problem Set 2 Solutions

Both theory and programming questions are due **Monday, February 28** at **11:59PM**. Solutions should be turned in through the course website. Your solutions to questions that provoke a written response should be in PDF format and typeset using LaTeX. Your solutions to problems that ask you to write code should be valid Python files which run from the command line. A template for writing up solutions in LaTeX is available on the course website. Remember, 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.

Problem 2-1. [20 points] **Hash functions and load**

(a) [3 points] 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? Argue that it will or provide a compelling reason why it will not (and propose a solution).

Solution: The most obvious problem with this scheme is that it ignores order (that is, anagrams will hash to the same values). Another, more subtle, problem is that alphabetic characters have sequential ASCII values. This compounds the first problem–the hash values for words of any particular length will tend to cluster heavily.

(b) [3 points] Imagine that we plan to hash the addresses of small, in-memory data structures. After much optimization (and reading over Alyssa's shoulder), Ben Bitdiddle manages to get this data structure to fit in 32 bytes (one cache line of the computer he's using). He then writes an algorithm that allocates a contiguous block of memory containing many of these data structures and inserts some of their addresses into a hash table whose size is some appropriate power of two.

Will the performance of his implementation match the expected value shown in lecture? Argue that it will or provide a compelling reason why it will not (and propose a solution).

Solution: These memory addresses will be multiples of 32; since the table size is a multiple of 2, this means that only 1/32 of the slots will be used.

(c) [3 points] Ben then decides that he needs to hash unordered sets of plain integers. He gives implementing this a try, but something seems to be wrong. His code is given in `setHash`. Explain what the problem is and provide an implementation that is a valid hash function producing a plain integer. (You may not use any built-in Python hash functions in your solution.)

Solution: The code provided produces different hash values depending on the order in which values are present in the list, but sets are unordered.

(d) [5 points] The ability to quantitatively measure the performance of a hash function will be useful; we'd like to be able to validate our intuition regarding which hash functions are good and which ones are not.

In `hasheval.py`, you'll find a stub called `hashEval`. It takes a hash function, a "hash table size", and a collection of values and produce some statistics about what a hash table of the given size using the given function and storing the given values would look like.

A simple wrapper called `printHashEval` that pretty-prints your results is included. The six statistics that your implementation of `hashEval` should return are listed above the stub; they are (in order) the load factor, the proportion of nonempty slots, the number of values that collide with another value, the mean size of the nonempty slots, the median size of the nonempty slots, and the number of items in the most-filled slot.

You should also implement `randInts`, which generates a uniformly-distributed sequence of random integers that we can use to experiment with `hashEval`.

(e) [3 points] A hash function is provided that is based on Python's own hashing scheme for integers (that is, the hash is the integer itself, modulo table size). Start with a table size of 8 and use `hashEval` to evaluate the provided hash function on a set of 4 values generated with `randInt`. Experiment with different table sizes and ratios of slots to values. Does this strategy seem to work well? What load factor, approximately, can you reliably achieve? Provide a few statistics to support your conclusion.

Solution: We're not providing solutions for the coding parts.

(f) [3 points] As you've just seen in a practical setting, the simple uniform hashing assumption has two parts; it requires that a particular value have an equal chance of being placed in any slot (uniformity) and that the slot a particular value is placed in be unaffected by which slot any other value is placed in (independence). Briefly describe a situation in which each is violated but the other holds.

Solution: To violate uniformity while satisfying independence, we may have a hash function that distributes items evenly into the hash table, but where each item is always fixed to a certain slot.

To violate independence while satisfying uniformity, we may have that every item has an equal chance of being placed into each slot, but that the hash function simply takes a random value and places every item into the same slot once the first hash key is determined. For example, when we are using linear probing as a collision resolution method, each item's slot location is dependent on the previous ones.

Problem 2-2. [25 points] **Collision resolution and dynamic resizing**

In this problem, we'll consider two very serious, real-world challenges that face hash table implementations: collision resolution and dynamic resizing. A simple hash table implementation called `Footable` is provided in `footable.py`. As provided, it cannot dynamically resize itself, and it will raise an exception if a collision occurs.

(The implementation of `Footable` uses a list internally, for which lookup is $O(n)$. We're going to ignore this–that is, treat it as though it were $O(1)$–for the purposes of this problem; dictionaries are so central to the Python way of thinking that it's not otherwise possible to do what we want without using them!)

(a) [2 points] If we have a collision resolution strategy, why do we need to consider dynamically resizing our hash table? It sounds like a lot of work, after all—both for us *and* for our computers!

Solution: Just because we can resolve collisions doesn't mean that they don't affect us; for example, if we are using chaining, we could hypothetically use a hash table with a single bin, but then all of our operations would be $O(n)$ instead of $O(1)$. Dynamic resizing lets us increase the number of bins, thereby reducing the load factor and our expectation of collisions.

(b) [2 points] What about the other way around? If we're going to bother dynamically resizing our hash table, why do we need a collision resolution strategy? Why don't we just resize whenever a collision occurs?

Solution: This would get you very close to being correct, although continual and repeated resizing of the table would be very expensive (and would certainly destroy any theoretical performance expectations). However, keep in mind that it's possible for two values to collide by actually producing the same hash value instead of just mapping to the same slot, and obviously that situation (a true collision) can't be fixed by resizing.

(c) [3 points] One simple method for resolving collisions is linear probing. Implement this strategy in a subclass named `FootableLinear`. Provide in a function named `badLinear` a test case demonstrating a situation in which this strategy is *not* good, and explain what undesirable behavior it produces and why.

Solution: We're not providing solutions for the coding parts.

(d) [3 points] Another simple method for resolving collisions is chaining. Implement this strategy in a sublcass named `FootableChaining`.

Solution: We're not providing solutions for the coding parts.

(e) [3 points] Implement quadratic probing in a subclass called `FootableQuadratic`.

Solution: We're not providing solutions for the coding parts.

(f) [3 points] Implement double hashing in a subclass called `FootableDouble`.

Solution: We're not providing solutions for the coding parts.

(g) [3 points] Compare and contrast, in terms of the problems that they can prevent and the problems that they can cause, the four strategies you've implemented. (Say two things about each method.)

Solution: Linear probing and quadratic probing share the same basic strategy where one would keep looking for an open spot by either a linear or quadratic function. They have a much faster lookup time but you cannot store more elements than the number of slots. Another problem is that deletion can be hard because you cannot simply clear the slot.

Double hashing has the advantage that the gap between each entry is dependent on the data, therefore it is possible to avoid clusters of collisions. But on the other hand the intervals between locations of data can be very big therefore look up time can be longer than that of linear probing in practice (not theoretically).

(h) [3 points] Now think about dynamically resizing your hash table. Provide a description of an algorithm that would turn a `FooTable` of size n into a `FooTable` of size $n + m$. What is the asymptotic time complexity of your algorithm?

Solution: We create the new table and insert each key-value pair into the new table. This requires rehashing the keys.

(i) [3 points] In lecture, we discussed doubling the size of our hash table. Ivy H. Crimson begins to implement this (that is, she lets $m = n$) 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 $m = k$ 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 2-3. [15 points] **Python dictionaries**

As you should know by now (and you're probably in trouble if you don't), Python's dictionaries are hash tables. They're a handy data structure, and you should use them whenever appropriate—but they're also heavily used internally! For this reason, their performance is very important to

Python's overall performance; a correspondingly large amount of work has been done on tuning their implementation.

(a) [1 points] 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.) List the seven different use cases that have been identified in this document.

Solution: They are: passing keywords arguments, class method lookup, instance attribute lookup and global variables, builtins, uniquification, membership testing, and dynamic mappings.

(b) [2 points] Let's examine the "membership testing" use case. Describe in more detail how this scenario will use hash tables, and what the primary concerns of this use case will be (in terms of the hash table's behavior).

Solution: In this use case we will do insertions at the beginning and afterward we will mostly perform lookups. One scenario where we would be using this use case would be string-matching. We would not insert new entries into the hash table after we start looking up for patterns. The main concern for this use case would be the number of collisions. We do not want to have heavy collisions because then our lookup time would not be constant.

(c) [3 points] If you were 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, what would you choose? Why?

Solution: Because we are not limited in memory and we require a lot of lookup, we would want a large dictionary and control the load factor at a small value (say 0.1 or smaller). Also we may want to cache lookups because the access pattern is not random.

(d) [3 points] Some use cases, such as the "global variables" use case, have requirements that conflict with these. Describe the requirements of this second use case and where the conflict emerges. Do you have any ideas about how these concerns might be balanced?

Solution: The global variables use case has a variable dictionary size whereas the membership testing use case rarely changes its size. Also, one writes frequently in the global variable use case. We can balance the two by either make the dictionary large to begin with or by decreasing the load factor threshold.

(e) [3 points] Further into the document are a list of five "tunable parameters"–things that can be changed in the C source code to modify the behavior of the hash table implementation. Briefly describe how each of these five parameters lines up with the "tunables" we've seen on the theory side of things; or, if it doesn't, explain why the item is important practically but not relevant theoretically.

Solution:

1. PyDict-MINSIZE: this is the minimum size of our python dictionary, or we can think this as our initial hash table size.
2. Maximum dictionary load: This is essentially the threshold load we discussed in lecture. Once our hash table exceeds the maximum load, we will dynamically resize.
3. Growth rate: This is the factor by which we resize our hash table by each time.
4. Maximum sparseness: We did not discuss this in lecture. Essentially we are shrinking our hash table to save space. This is useful practically because it saves us memory. We did not discuss this theoretically because it does not affect our running time.
5. Shrinkage rate: this is the factor by which we are shrinking our hash table once it becomes too sparse.

(f) [3 points] How would you adjust each of these tunables to result in the best performance for the "membership testing" use case that you examined previously? Briefly describe and justify your choices (no more than a sentence per tunable).

Solution: We would want to set the minsize to a relatively large number because we do not want to resize very often. We would not want to shrink the table ever so we want to set the maximum sparseness to as small as possible. Also we want to set maximum dictionary load low so we don't have many collisions.

Problem 2-4. [40 points] **Matching DNA sequences**

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 a number of samples from 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 sequence as a string 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. DNA also varies a little bit from individual to individual, so we'll need the ability to find inexact matches.

(a) [1 points] In lecture, we mentioned something called a rolling hash. Describe what it is and what problem it solves. Why is it a better solution to that problem than a normal hash? Why might it be useful to Ben Bitdiddle in his quest for biotech riches?

Solution: A rolling hash is used in the situation where the input is a small moving window of a larger chunk of data. The rolling hash function can efficiently compute the hash value of the next window given the value of the hash function on the previous window. We see that rolling hash is useful because it fully utilizes the fact that every subsequence has many characters in common. So if we implement our rolling hash correctly, we can achieve $O(n)$ running time.

(b) [7 points] Propose and describe a rolling hash function that is appropriate to our situation, and then implement it in the `RollingHash` class in `dnaseq.py`.

Solution: Because our input only consists of four characters (A,C,G,T), we should represent our subsequences as a base four number. So our rolling hash function for a subsequence of a of length $k+1$ should be:

$$h(a[i:i+k]) = a[i] \cdot 4^k + a[i+1] \cdot 4^{k-1} + \cdots + a[i+k]$$

where $a[i]$ is the ASCII value of the i^{th} character of sequence a.

(c) [5 points] Now we'll work towards implementing `getExactSubmatches`, the nucleotide sequence comparison function described above. Let's start with `subsequenceHashes`, which returns all k-length 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; 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.

Solution: We're not providing solutions for the coding parts.

(d) [7 points] Another issue you'll encounter is that of hash collisions. From your work in problem 2, you should know several ways to resolve collisions (although a simple approach like chaining is probably a good place to start). Implement `Multidict` and verify that your work passes the simple sanity tests provided. You *may* use the Python dictionary in your implementation.

Solution: We're not providing solutions for the coding parts.

(e) [5 points] Now it's time to implement `getExactSubmatches`. Ignore the parameter m for the time being; we'll get to that in a moment. 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`, `RollingHash`, and `subsequenceHashes`. Your solution probably does not need to be very complex.

We've provided a simple sanity test (that may stop working once you support m, depending on how you do so; don't worry about this). Your solution should be correct at this point (that is, able to run `compareSequences`) but it'll probably be too slow to be useful. You can try running it on the first portion of two inputs. The `.fa` files are just text files; you can use `head` to siphon a few tens of thousands of lines off into a different file if you'd like to see some results.

Solution: We're not providing solutions for the coding parts.

(f) [5 points] 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. Why should we still see approximately the same result? Why can't we further improve performance by applying this technique to sequence B as well?

Solution: We're not providing solutions for the coding parts.

(g) [5 points] Run comparisons between the two human samples (paternal and maternal) and between the paternal sample and each of the animal samples. What do you notice? Attach and interpret the images that you produce.

As it turns out, all of the samples provided are relatively closely related, but given that we are only currently plotting exact matches, some will not appear quite as correlated as they really are. Which samples are these, and why might you expect these samples to show this result?

Solution: We're not providing solutions for the coding parts.

(h) [5 points] Ben Bitdiddle feels that some of his mutants will have sufficiently different genomes that exact matching might not suffice. Consider a single-mismatch variant of `getExactSubmatches` (that is, one that reports a match even if one base pair is a mismatch). What parts of your implementation would have to change, and what modifications would you have to make to them? What information would have to be hashed? Describe (but do not implement) your algorithm.

Solution: Instead of looking up every subsequence of length k of B in the hash table, we look up every subsequence of length k that has one mismatch in B. For example, consider the subsequence "ACGT", we would look up "ACGT", "CCGT", "GCGT", "TCGT", etc. We see that this algorithm is correct because now we have considered all possible one mismatch subsequences of length k in B.

Runtime Analysis: We see `getExactSubmatches` takes $O(n)$ time to build the hash table. Lookup takes $O(n)$ time to calculate the rolling hashes for each of the $O(n-k)$ lookups (because there are $O(n-k)$ subsequences of length k in B). Now we see for every subsequence in B, we have $4k$ the number of lookups since we have to modify the character at each position 4 times and lookup in the hash table. Therefore the total running time is $O(n) + O(n) + 4k \cdot O(n-k) = O(nk)$.