4-1 Hash functions & load

a. Imagine that an algorithm requires us to hash strings containing English phrases. Knowing that strings are stored as sequences of characters, APH 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 out strings nicely by length

2. . . by the characters they contain

(3.) No, reordering words in a string will not produce a different hash

4. No, because the independence condition of the simple uniform hashing assumption is violated

Not 4, hashing does not depend on where other keys are hashed.

(b). A implements both collision resolution & dynamic resizing for her hash table. However, she doesn't want to ~~use~~ both. Dynamic resizing can avoid collisions, collisions don't cause correctness issues if collision resolution is implemented. Which statement about these 2 properties is true?

1. DR alone will preserve both properties

2. DR will preserve correctness, not performance

3. Collision resolution will preserve performance, not correctness

(4.) Both are needed

DR for performance, collision resolution for performance

c. A implements resizing. ~~[illegible]~~ Enlarging a table of size m into a table of size m', containing n elements. What is the best time complexity she can achieve?

1. $\Theta(m)$ 2. $\Theta(n')$ ~~3.~~ $\Theta(n)$ 4. $\Theta(nm')$ 5. $\Theta(m+n')$ [✓ 6.] $\Theta(m+n)$ 7. $\Theta(m'+n)$

Lecture 09 states resizing has $\Theta(n+m)$ time ~~[illegible] $= \Theta(n)$~~

In this case the size of the final table is m' so plug $m=m'$ into the above formula

d. In lecture, we discussed doubling the size of our hash table. IHC implements $(m'=2m)$ but stops when it occurs 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 don't we do it. Theoretical & practical reasoning required

Theoretically: From Lec 09 Notes

$O(1)$ amortized cost for rebuilding at $m'=2m$: $\Theta(\sum_{i=1}^{m} 2^{n}) = \Theta(m)$

Otherwise, just adding in values would take $O(n)$ amortized cost to find the appropriate location : $\Theta(m+m+k+\dots+2m) = \Theta(m^2)$

Practically:

It is very simple to double size, and do so at powers of 2 instead of some constant

Think about the asymptotic cost over many resizings.

4-2. a. Let's examine the "membership testing" use case. Which statement accurattley describes it?

[✓ 1.] Many insertions right after creation, and then mostly lookups

2. " " .. only ..

3. A workload of evenly-mixed insertions/deletions & lookups.

4. Alternating rounds of insertions/deletions & lookups

It is describes as "Create once then rarely changes" ⇒ many insertions, mostly at creation

"Many calls to has-key" ⇒ many lookups.

b. Now imagine you have to pick a hash function, size, collision resolution strategy, and so forth 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 min size and a growth rate of 2.

2. A small min size & growth rate of 2

[✓ 3.] large min size & . . 4

4. small . . 4

The document recommends a low α ⇒ many slots per key ⇒ large table size ⇒ faster growth rate

Many keys inserted initially ⇒ start w/ a larger table

4-3. 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 creatures 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.

For our purposes, we'll represent a DNA sample as a sequence of characters. These sequences are very long, so comparing subsequences of them quickly is important. We've provided code in kfastsa.py that reads the .fa files storing this data.

a. Let's start with 'subsequenceHashes', which returns all length-k subsequences and their hashes. (implement your function as a generator).

See 'subsequenceHashes' in dnaseq.py

b. Implement 'Multidict' and verify that your work passes the simple sanity tests provided.

See class 'Multidict' in dnaseq.py

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.

See 'getExactSubmatches' in dnaseq.py

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 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. Modify your implementation of 'getExactSubmatches' to honor m only for sequence A.

See 'intervalSubsequenceHashes' and 'getExactSubmatches' in dnaseq.py

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

See 'comparisons' folder in /dist/