CS2040S: Data Structures and Algorithms

Recitation 5

Goals:

- Finish discussions on Merkle Trees
- Explore more uses of hashing as a fingerprint/signature
- Reinforce the concept of space—time trade-off
- Illustrate how hashing can often be used to speed up solutions

Problem 1. Drug Discovery

Note: this problem is carried over from last week, if your recitation class didn't get to it then.

In the bid to find a potential cure for the COVID-19, research labs around the world are racing to study the effects different drugs have on the coronavirus. Viruses respond to drugs via *mutations*. A mutation occurs when there are changes (often small subsequences) along the original genome sequence. Suppose you now work for an international bioinformatics organization which maintains a huge open-access and canonical DNA databank (e.g. GenBank). Your organization is supporting the drug discovery effort by not only releasing the canonical sequence of the virus to labs all over the world, but also delivering it in a special format that would efficiently facilitate the comparisons between different mutations. In this way, all a research lab needs to do is to first download a local copy of the canonical unmutated sequence (in the special format), update it with the mutations from their experimental results, then quickly compare it with the mutated sequence from another lab which ran a different drug experiment. The mutational differences and similarities of the virus in response to different drugs reveal important drug properties to scientists, which in turn help them formulate more effective drugs.

In its raw format (i.e. before special formatting), a virus genome sequence is presented as a list of records where each record contains a subsequence of 60 characters with each character representing a nitrogenous base (e.g. Adenine, Guanine, Cytosine, Thymine). This is illustrated in Table 1 below.

Record#	Subsequence
1	AAAGGTTTAT ACCTTCCCAG GTAACAAACC AACCAACTTT CGATCTCTTG TAGATCTGTT
2	CTCTAAACGA ACTTTAAAAT CTGTGTGGCT GTCACTCGGC TGCATGCTTA GTGCACTCAC
3	GCAGTATAAT TAATAACTAA TTACTGTCGT TGACAGGACA CGAGTAACTC GTCTATCTTC
...	... ...

Table 1: Source: Severe acute respiratory syndrome coronavirus 2 2019-nCoV/Japan/KY/V-029/2020 RNA, complete genome

As the chief of data in the organization, you are tasked to design the special format which would make comparisons between 2 genome sequences efficient. Having been a former student of CS2040S, you know that the "special format" necessarily entails a clever DS that is well-suited for the comparison operation.

Problem 1.a. Design a tree-based DS that can capture a list of n records (i.e a genome sequence) such that when given the tree for another list (i.e a mutated sequence), you can *efficiently* (read: better than $O(n)$ time) determine which are their records that differ from one another.

Problem 2. To Download or Not to Download

Alice has a very large collection of digital photos, consisting of several hundred gigabytes of data. In order to ensure that her photos are stored safely, she prudently maintains backup copies of all her photos on *Mazonia Storage*—a cloud-based server storage service.

Recently, some of Alice's local photos are lost because her hard drive got infected by *Claudius*, a computer virus. Worse, Claudius corrupted all of the filenames (overwriting the metadata in the Master File Table), replacing them with text from Shakespeare's *Hamlet*.

Alice's goal now is to recover her missing photos. It may be helpful to note that since Alice is meticulous in organizing and curating her collection, no photo duplicates exist within it.

Given that Alice has so many photos, it would take a very long time to download all of them from Mazonia Storage's server. The practical approach is therefore to download only the missing photos, i.e., those that were deleted by the virus.

For this problem, we shall denote the following:

- Let n be the original number of photos Alice have *remotely*
- Let $m < n$ be the remaining number of photos Alice have *locally*
- Let $r_1, \ldots, r_i, \ldots, r_n$ be the photos on Alice's *remote* cloud server
- Let $\ell_1, \ldots, \ell_i, \ldots, \ell_m$ be the photos on Alice's *local* computer

Having studied CS2040S, Alice immediately think of using a hash-based signature and proposes the following algorithm:

1. Pick a hash function h that maps a photograph to an integer in the range $\{1, \ldots, n\}$
2. For each photo ℓ_i on Alice's *local* computer, compute its hash value $h(\ell_i)$
3. Save all locally hashed values to a file $H_\ell = \{h(\ell_i) : i \in [1, m]\}$ on Alice's local computer
4. For each photo r_i on the *remote* server:

4.1. Compute its hash value $h(r_i)$

4.2. Download $h(r_i)$ to Alice's local computer

- If $h(r_i)$ is found in H_ℓ, continue the loop
- Else, download the photo r_i

Problem 2.a. Alice claims that this scheme will efficiently restore *all* the missing photos to her computer. Is she right? Explain why or why not.

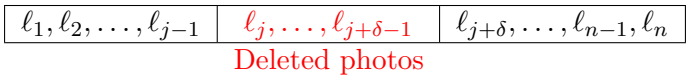
Alice also propose a second scheme:

1. Let k be some integer (which may depend on n and m)
2. Repeat:
   2.1. *Randomly* pick a hash function h that maps a photograph to an integer in the range $\{1, \ldots, k\}$
   2.2. For each photo ℓ_i on Alice's *local* computer, compute its hash value $h(\ell_i)$
   2.3. Save all locally hashed values to a file $H_\ell = \{h(\ell_i) : i \in [1, m]\}$ on Alice's local computer
   2.4. For each photo r_i on the *remote* server, compute its hash value $h(r_i)$
   2.5. Save all remotely hashed values to a file $H_r = \{h(r_i) : i \in [1, n]\}$ on the remote server
   2.6. Download H_r to Alice's local computer
   2.7. If $(|H_r| - |H_\ell|) = (n - m)$,
      2.7.1. Determine the photos r_i whose hash value $h(r_i)$ is in H_r but not in H_ℓ and download them
      2.7.2. Terminate the repeat loop
   2.8. Else, continue the loop to look for a better hash function

Problem 2.b. An interesting criteria for picking the random hash function is stated in Step 2.7.. Why didn't Alice simply chose the condition to be: $|H_r| = n$ `&&` $|H_\ell| = m$? *What* is the invariant in the desired hash function here? *Hint:* Think about H_r and H_ℓ respectively as the hash values *before* and *after* a set of deletion operations.

Problem 2.c. In the second scheme, what is Alice's objective in finding the appropriate hash function? *Why* does the invariant satisfy this objective? Show that when the loop terminates, it means Alice has correctly downloaded all the missing photos.

After some investigation, Alice learnt that the virus had only erased a single consecutive block of photos. Given that her photos were stored on her local hard drive in sequence, the virus simply deleted a contiguous subsequence of photos of length δ. This is illustrated as follows.



Realize the value of δ can be simply calculated as the difference between the number of remote photos and local photos.

Unfortunately, Alice has no idea which photos were deleted. Moreover, Alice's internet connection is very slow, so much so that even transmitting a file containing n hash values will take too long! To make matters worst, Mazonia charges its clients for every bit transmitted to and fro the server. Due to these reasons, Alice is forced to limit the amount of communication with the remote server as much as possible. Fortunately, all is not lost because in this time Alice cleverly devised a *perfect hash function* h which will not produce any collisions on her original set of n photos. She already uploaded this function so it is now available on both the server and her local computer.

Problem 2.d. Given the new findings, design two variants of algorithms to help Alice identify the deleted photos:

Variant 1: Hash values are only computed on individual photos

Variant 2: Hash values are also computed over contiguous subsequences of photos

Meanwhile, your algorithms must satisfy the following efficiency constraints:

Constraint 1: Transmit at most $O(\log n)$ hash values and a constant number of additional integer values to and fro the Mazonia server

Constraint 2: Incur only an additional $O(1)$ space

For now, you may assume that hash value computation with h is $O(1)$. Explain why your algorithms work and give their running time.

Problem 3. Dim-Sum Without Sum is Just Dim

You have an unsorted array A containing n integers where each element represents an *item price* on the menu at *Dim-Sum Dollars*—your favorite dim sum restaurant, and its index representing the *item number* on the menu. Having x dollars on hand in cash and wanting to avoid loose change, you seek to purchase two items such that their price sums up to *exactly* x. Your task is therefore to write a program which will find you *any* such pair of dim sum items on the menu. If no such pair of items exist, the program should notify you of failure.

For example, given the following menu:

\$30	\$8	\$15	\$18	\$23	\$20	\$25	\$1
#1	#2	#3	#4	#5	#6	#7	#8

If your cash on hand is $x = \$33$, then there are two possible item pairs whose values sum up to that amount, namely pairing #2 (\$8) with #7 (\$25) and pairing #3 (\$15) with #4 (\$18).

Problem 3.a. Come up with a solution which finds a valid pair of item *numbers* if it exists. For instance in the given example above, either $(\#2, \#7)$ or $(\#3, \#4)$ will be a valid pair. It must only incur $O(1)$ space. What is its time complexity?

Problem 3.b. Now come up with a solution which finds a valid pair of item *prices* if it exists. For instance in the given example above, either $(\$8, \$25)$ or $(\$15, \$18)$ will be a valid pair. It must only incur $O(\log n)$ extra space. What is its time complexity?

How might you then modify this solution to return a valid pair of item *numbers* and what are the additional overheads your modification entail?

Problem 3.c. Finally, come up with the *fastest* solution which finds a valid pair of item *numbers* if it exists. For instance in the given example above, either $(\#2, \#7)$ or $(\#3, \#4)$ will be a valid pair. Your algorithm may now incur $\Theta(n)$ extra space. What is its time complexity?

Depending on your solution proposed, you might have introduced a subtle caveat. *Hint:* What happens when there are duplicate item prices?

Problem 3.d. How might you minimally modify the previous 3 algorithms such that instead of just finding one valid pair, *all valid pairs* will be returned by the program?

Problem 3.e. Now you want to find three elements a, b, c in the array that sum to x. Can you generalize the previous solutions to solve this variant?

Did you know? This is the classic 3SUM problem! It turns out that there exists many important problems which either reduces to this form or entails a subproblem that does. There is actually a lot of ongoing research trying to understand the best possible solution for 3SUM!