

Goals:

- 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. 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 $r_1, \dots, r_i, \dots, r_n$ be the photos on Alice’s *remote* cloud server
- Let $\ell_1, \dots, \ell_i, \dots, \ell_m$ 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 *any* hash function h that maps a photograph to an integer in the range $[1, n]$
2. For each photo $\ell_i : i \in [1, m]$ on Alice’s *local* computer,
 - 2.1. Compute its hash value $h(\ell_i)$

- 2.2. Save $h(\ell_i)$ to a local file H_ℓ
3. For each photo r_i on the *remote* server:
 - 3.1. Compute its hash value $h(r_i)$
 - 3.2. Download $h(r_i)$ to Alice's local computer
 - If $h(r_i)$ is not found in H_ℓ , download photo r_i
 - Else, continue the loop

Problem 1.a. What are Alice's objectives of using a hash function in this scheme? What is the key to success in achieving those objectives?

no collisions → compare files in $O(1)$ time

Problem 1.b. Is H_ℓ a hash table? → just hash values

no.
Hash table → must have (key, value)

Problem 1.c. Alice claims that this scheme will efficiently restore *all* the missing photos to her computer. Is she right? Explain why or why not.

Depends on the hash functions → if there is no collisions then quite good
→ if not then not good enough → can't decide whether it's the right one

Problem 1.d. What if Alice modified her solution by adding separate chaining to H_ℓ ? What would be stored in each bucket? Would this serve as an effective solution?

construct 2 levels hash functions

Alice also propose a second scheme where she randomly picks a hash function until she finds one that fits her criteria:

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 $[1, k]$
 - 2.2. For each photo $\ell_i : i \in [1, m]$ on Alice's *local* computer
 - 2.2.1. Compute its hash value $h(\ell_i)$

- 2.2.2. Save $h(\ell_i)$ to a local file H_ℓ
- 2.3. For each photo $r_i : i \in [1, n]$ on the *remote* server
 - 2.3.1. Compute its hash value $h(r_i)$
 - 2.3.2. Save $h(r_i)$ to remote file H_r
- 2.4. Download H_r to Alice's local computer
- 2.5. If $(|H_r| - |H_\ell|) = (n - m)$,
 - 2.5.1. Download the photos r_i whose hash value $h(r_i)$ is in H_r but not in H_ℓ
 - 2.5.2. Terminate the repeat loop
- 2.6. Else, continue the loop to look for a better hash function

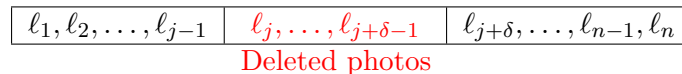
Problem 1.e. An interesting criteria for picking the random hash function is stated in Step 2.5.. Why didn't Alice simply chose the condition to be $|H_r| = n$ && $|H_\ell| = m$? *loses condition \rightarrow easier to achieve* *\rightarrow perfect hash function*

Problem 1.f. If we think about H_r and H_ℓ respectively as the set of hash values *before* and *after* a set of deletion operations, what is the "invariance" in the desired hash function here?

the number of deletion equals to the number of drops

Problem 1.g. In the second scheme, what is Alice's objective of using a hash function? *Hint:* look at the invariance. Why does the criteria $(|H_r| - |H_\ell|) = (n - m)$ 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 1.h  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 times.

Problem 2. Putting the Sum in DimSum

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 .

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).

Your task is therefore to propose algorithms which will find you *any* such winning pairs of dim sum items on the menu.

Problem 2.a. Come up with an algorithm 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 solution must only incur $O(1)$ space. What is its time complexity?

$O(n^2)$

Problem 2.b. Now come up with an algorithm 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. Your solution must only incur $O(\log n)$ extra space. What is its time complexity?

How might you then modify your solution to return a valid pair of item *numbers* instead and what are the additional overheads (if any) due to your modification?

quicksort → two pointers → starting
→ ending

Problem 2.c. Finally, come up with the *fastest* algorithm 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 solution may now incur $\Theta(n)$ extra space. What is its time complexity?

How might you modify your solution to cater for duplicate prices?

Depending on your solution proposed, you might have introduced a subtle caveat. *Hint:* What happens when target sum is an even number?

Problem 2.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 algorithm?

Problem 2.e. What if instead of two, 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!