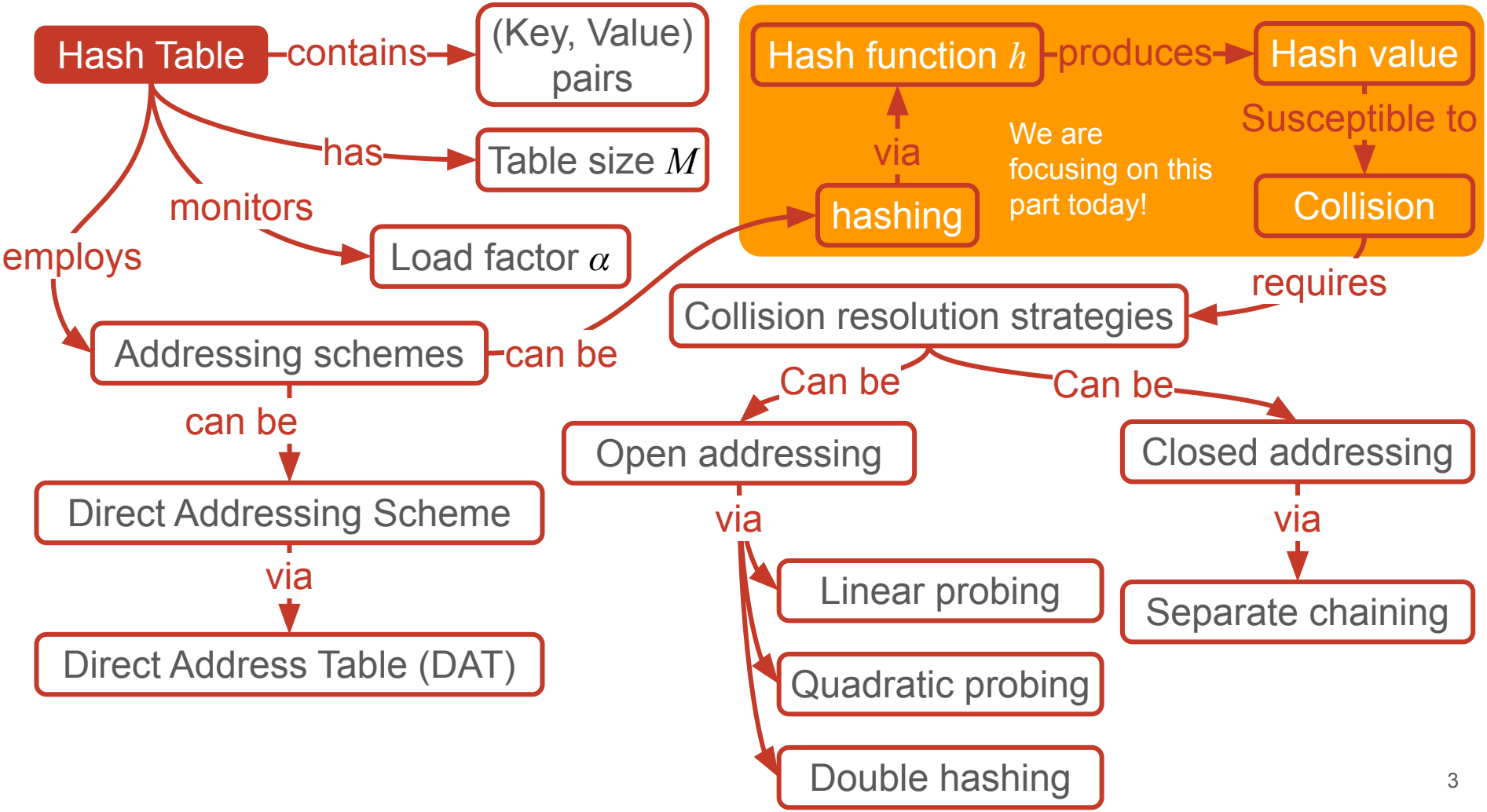
CS2040S

Recitation 6
AY22/23S2

Conceptual re*hash*

Review these slides for a quick recap on hashing.


Hash Table
contains
(Key, Value) pairs
has
Table size M
monitors
Load factor α
employs
Addressing schemes
can be
can be
Direct Addressing Scheme
via
Direct Address Table (DAT)
Hash function h
produces
Hash value
Susceptible to
Collision
via
hashing
We are focusing on this part today!
requires
Collision resolution strategies
Can be
Can be
Open addressing
Closed addressing
via
via
Linear probing
Quadratic probing
Double hashing
Separate chaining

Recitation 6

Recitation Goals

- Understand the merits of hash functions on their own
- Appreciate hashing as a means to fingerprint and summarize data
- Illustrate how hashing can minimize communication complexity
- Explore problem-specific collision resolution strategies

Hashing time

For this entire recitation, you may assume hashing takes $O(1)$ time.

Problem 1

Drug Discovery

- Here we have a genome sequence represented by a list of records
- Each record is a non-overlapping subsequence of the entire genome sequence (e.g. 60 characters)
- Each character in the subsequence represent 1 of 4 nitrogenous bases
- A mutation happens when a few records are modified

Drug Discovery

Genome sequence 1

Record#	Subsequence
1	AAAGGTTTAT ACCTTCCCAG …
2	CTCTAAACGA ACTTTAAAAT …
3	GCAGTATAAT TAATAACTAA …
4	TGCAGGCTGC TTACGGTTTC …
5	CCGGGTGTGA CCGAAAGGTA …

Genome sequence 2

Record#	Subsequence
1	AAAGGTTTAT ACCTTCCCAG …
2	CTCTAAACGA AAAAAAAAAA …
3	GCAGTATAAT TAATAACTAA …
4	TGCAGGCTGC TTACGGTTTC …
5	CCGGGTGTGA CCCCCCCCCC …

Here we have 2 genome sequences with only 5 records each (for sake of illustration). Sequence 2 is a mutation of sequence 1 and we observe that records 2 and 5 are modified.

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.

Drug Discovery

Genome sequence 1

Record#	Subsequence
1	AAAGGTTTAT ACCTTCCCAG …
2	CTCTAAACGA ACTTTAAAAT …
3	GCAGTATAAT TAATAACTAA …
4	TGCAGGCTGC TTACGGTTTC …
5	CCGGGTGTGA CCGAAAGGTA …

Genome sequence 2

Record#	Subsequence
1	AAAGGTTTAT ACCTTCCCAG …
2	CTCTAAACGA AAAAAAAAAA …
3	GCAGTATAAT TAATAACTAA …
4	TGCAGGCTGC TTACGGTTTC …
5	CCGGGTGTGA CCCCCCCCCC …

Can you tell in O(log n) time which records are mutated.

Let’s make the problem simpler

Genome sequence 1

Record#	Subsequence
1	AAAG
2	CTCT

Genome sequence 2

Record#	Subsequence
1	AAAG
2	CTCA

By doing only 1 comparison, can you tell me whether Genome sequence 2 is mutated or not? (You can create new strings and perform comparisons between those strings)

Let's make the problem simpler

Genome sequence 1

Record#	Subsequence
1	AAAG
2	CTCT

Genome sequence 2

Record#	Subsequence
1	AAAG
2	CTCA

Concatenate the strings.

AAAG ⊕ CTCT = AAAGCTCT and AAAG ⊕ CTCA = AAAGCTCA

If two concatenated strings are equal then no mutation happened otherwise at least one record is mutated.

Let's make the problem simpler

Genome sequence 1

Record#	Subsequence
1	AAAGGTTTAT ACCTTCCCAG …
2	CTCTAAACGA ACTTTAAAAT …
3	GCAGTATAAT TAATAACTAA …
4	TGCAGGCTGC TTACGGTTTC …
5	CCGGGTGTGA CCGAAAGGTA …

Genome sequence 2

Record#	Subsequence
1	AAAGGTTTAT ACCTTCCCAG …
2	CTCTAAACGA AAAAAAAAAA …
3	GCAGTATAAT TAATAACTAA …
4	TGCAGGCTGC TTACGGTTTC …
5	CCGGGTGTGA CCCCCCCCCC …

By doing only 1 comparison, can you tell me whether Genome sequence 2 is mutated or not?

Let's make the problem simpler

Genome sequence 1

Record#	Subsequence
1	AAAGGTTTAT ACCTTCCCAG …
2	CTCTAAACGA ACTTTAAAAT …
3	GCAGTATAAT TAATAACTAA …
4	TGCAGGCTGC TTACGGTTTC …
5	CCGGGTGTGA CCGAAAGGTA …

Genome sequence 2

Record#	Subsequence
1	AAAGGTTTAT ACCTTCCCAG …
2	CTCTAAACGA AAAAAAAAAA …
3	GCAGTATAAT TAATAACTAA …
4	TGCAGGCTGC TTACGGTTTC …
5	CCGGGTGTGA CCCCCCCCCC …

Concatenate the strings again and compare.

Drug Discovery

Genome sequence 1

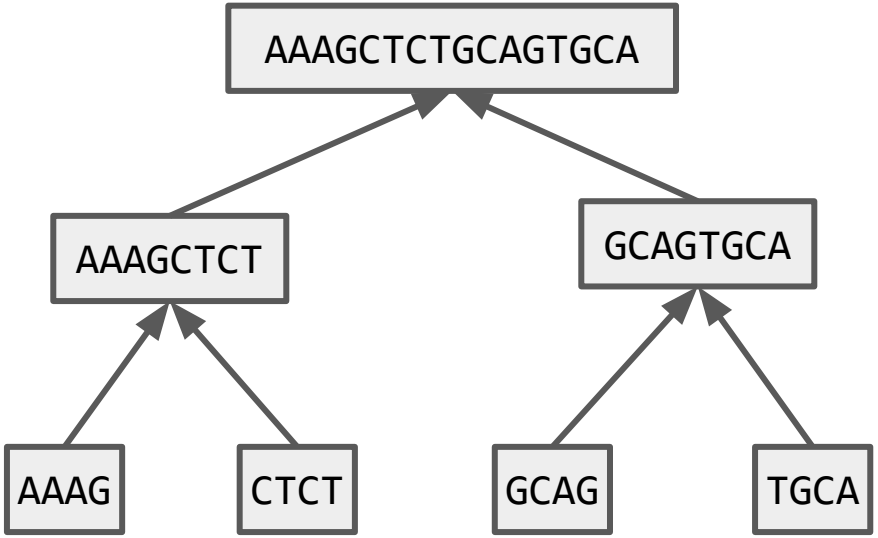
Record#	Subsequence
1	AAAG
2	CTCT
3	GCAG
4	TGCA

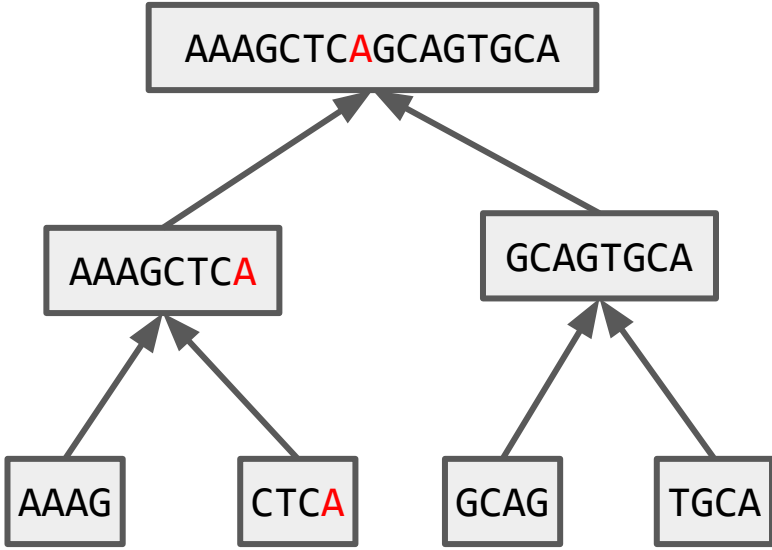
Genome sequence 2

Record#	Subsequence
1	AAAG
2	CTCA
3	GCAG
4	TGCA

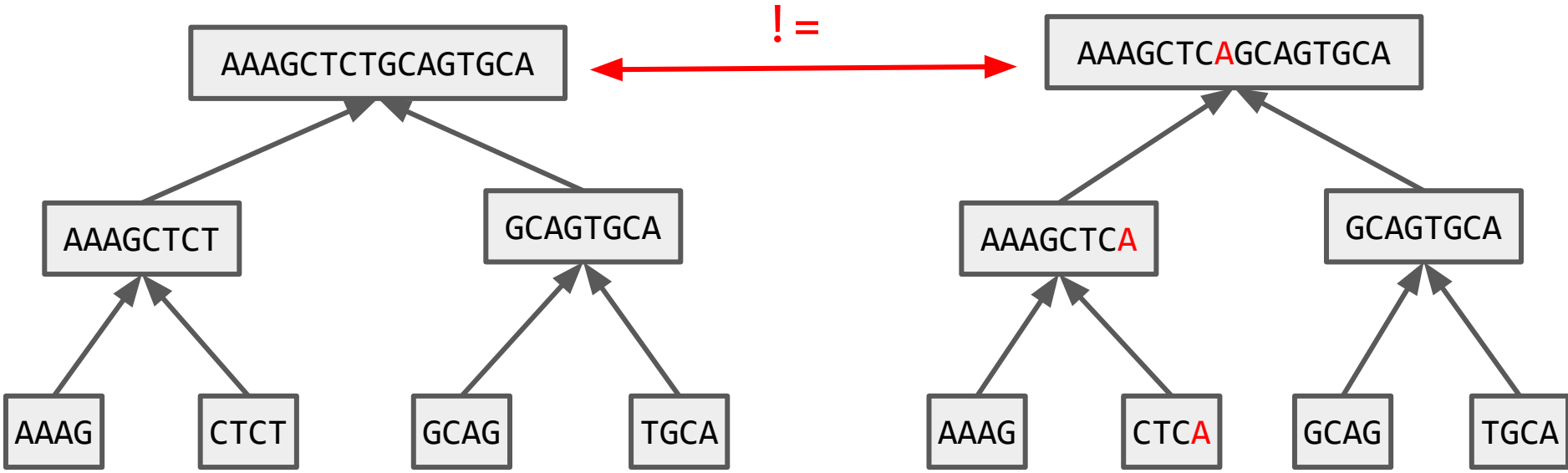
Can you tell in O(log n) time which records are mutated.

Drug Discovery

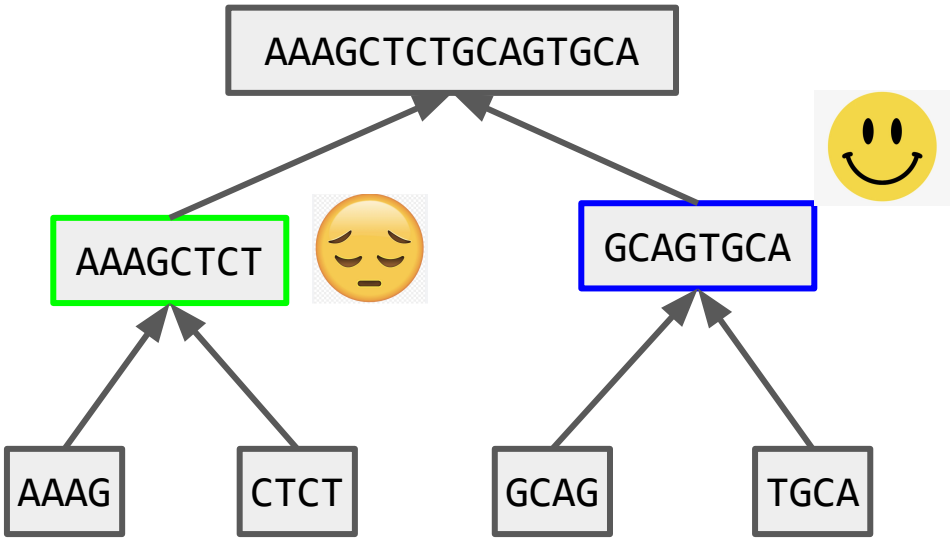


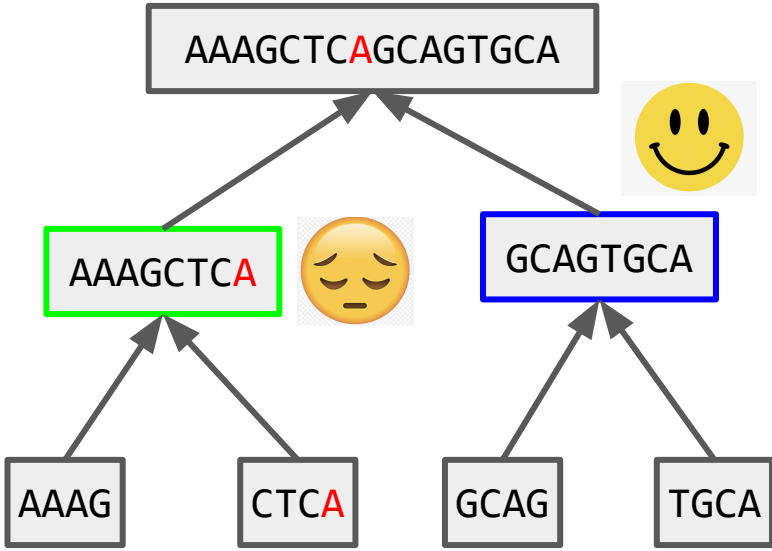


Drug Discovery

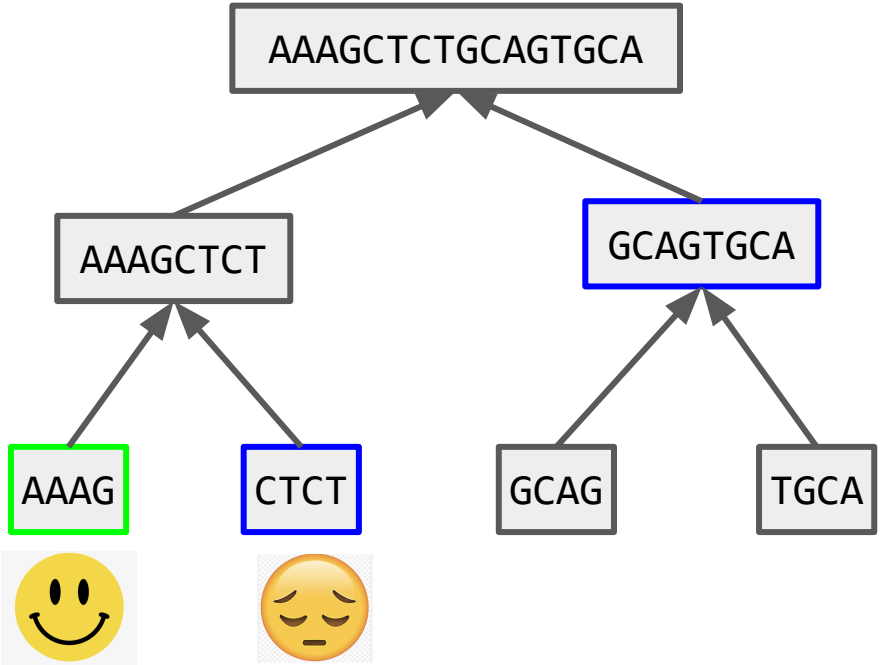


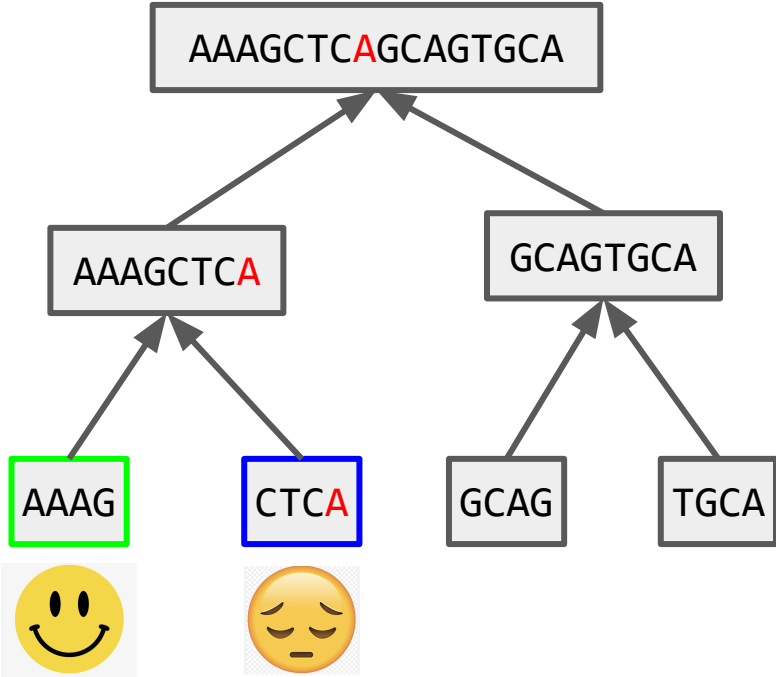
Drug Discovery





Drug Discovery





String comparisons are COSTLY

- But comparison of two strings is COSTLY. So, for fast comparison we hash the records to a number.

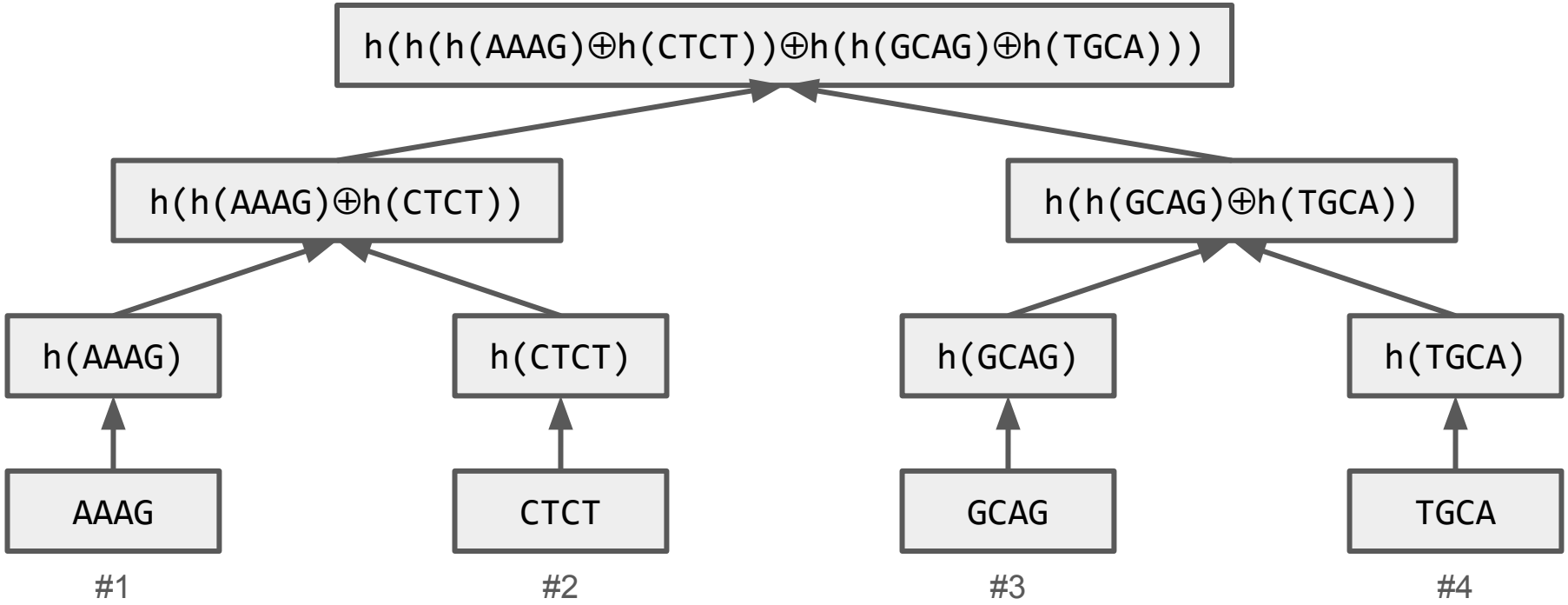
- We must *hash* each record to a unique *number (fingerprint / signature).*

Introducing Hash

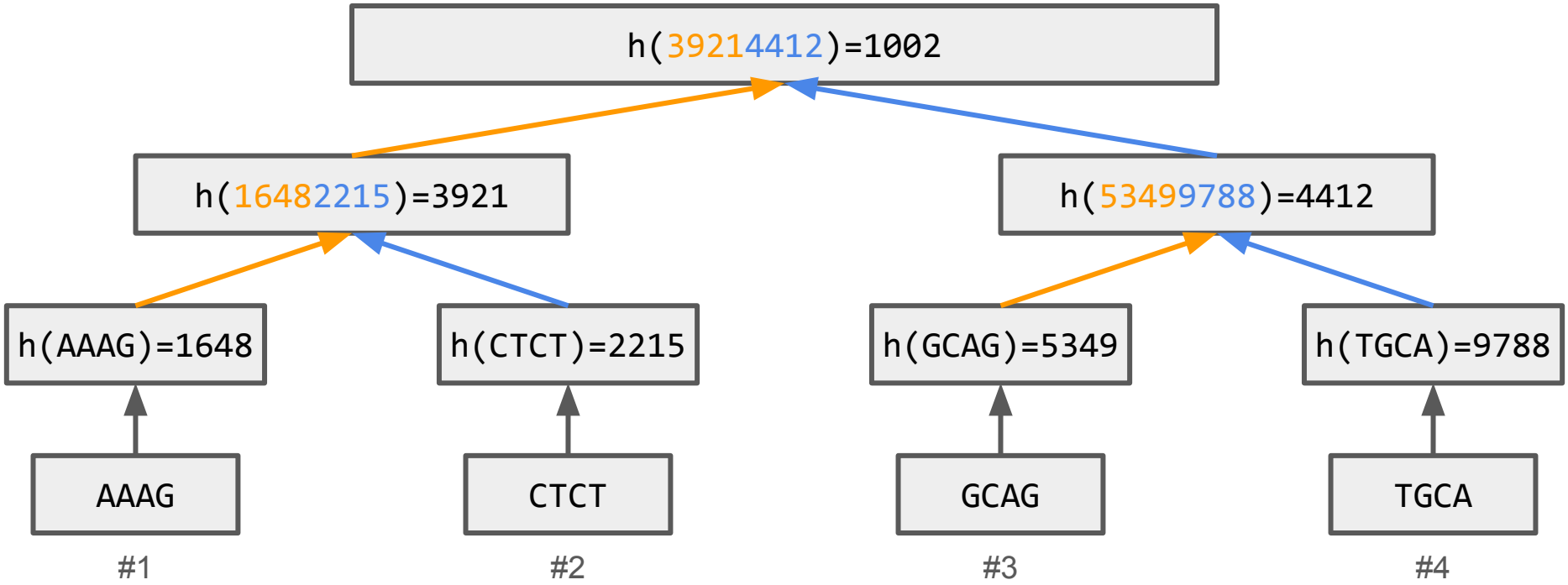
- A *hash function* h is one which turns an input into a numeric signature
- When 2 different inputs are *hashed* to the same output, we call that a *collision*
- A *good* hash function provides high guarantees of uniqueness (i.e. strong signatures), which means low probability of collision
- In fact a 128-bit output hash function with a million hash values will entail a collision probability of as little as 1.469366×10^{-27}

Amending our solution

Where ⊕ means concatenate. So at every level we hash the previous level's hashes! This ensures strong signatures!



Example


h(39214412)=1002
h(16482215)=3921
h(53499788)=4412
h(AAAG)=1648
h(CTCT)=2215
h(GCAG)=5349
h(TGCA)=9788
AAAG
CTCT
GCAG
TGCA
#1
#2
#3
#4

Merkle trees

- What we just showed you is called a Merkle tree
- It marries the hierarchical summarisation property of trees with the fingerprinting property of hashing!
- It is extensively used in distributed/decentralized systems such as peer-to-peer networking and blockchain technologies!
- In practice a really efficient DS for resolving a small number of conflicting/outdated records in a huge list of records at any one time

How Merkle trees are used

Alice

Sends 100$

Bob

Bank (Central Authority) verifies the transaction from Alice to Bob

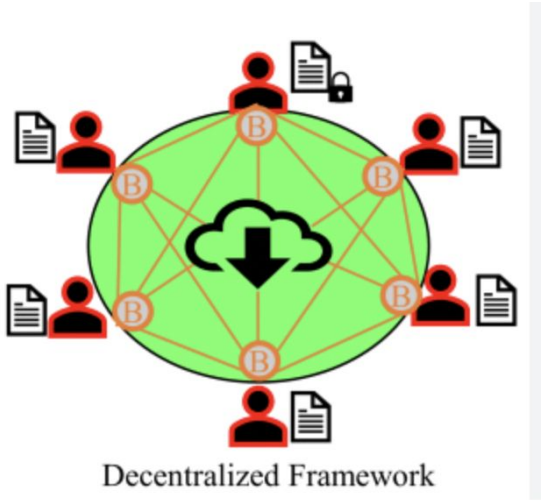
In a centralised system, bank has the records (bank account details, transaction details) of its customers.

How Merkle trees are used

Alice

Sends 2₿

Bob



Everyone in the system verifies the transaction

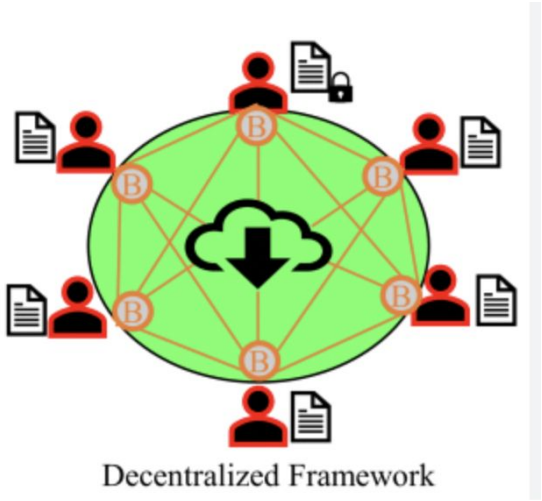
In a decentralised system, every customer has the records of every other customer.

How Merkle trees are used

Sends 2฿

Alice

Bob



Everyone in the system verifies the transaction

For fast verification Merkle Trees are used.

We can detect that a transaction is legal or not only by comparing the hash value stored in the root of the Merkle Tree.

In a decentralised system, every customer has the records of every other customer.

Question 2:

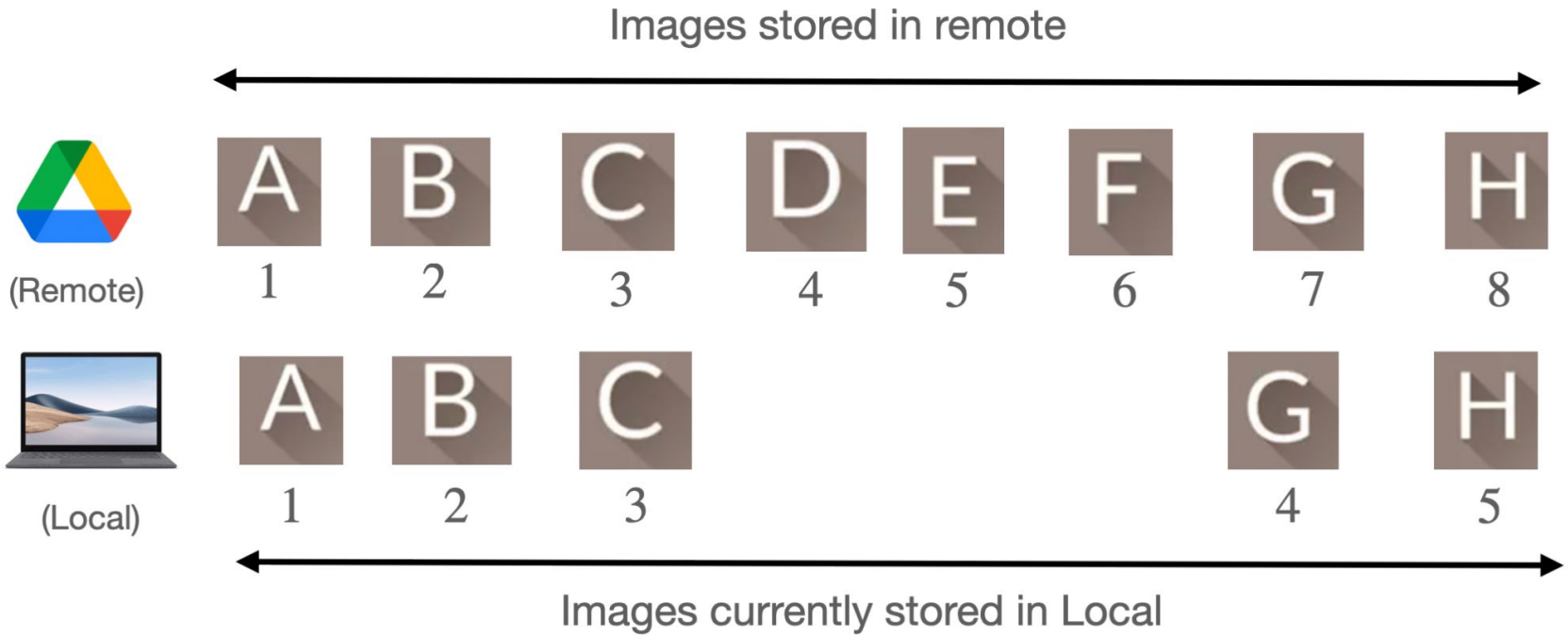
To Download or Not to Download

To Download or Not to Download

- Alice originally have n photos
- They were backed up remotely
  - Remote photos: $r_1, r_2, \ldots, r_n$
- Alice's local computer was infected by a virus which deleted some photos
- Locally only m photos remain
  - Local photos: $\ell_1, \ell_2, \ldots, \ell_m$
- Alice need to limit the number of transmissions to and fro the remote server
- Which are the deleted photos?

Suppose for now

- The virus only deleted a contiguous subsequence of photos
- Alice has a *perfect hash function* h
- Communication constraint: transmit at most $\mathrm{O}(\log n)$ numbers
- Space constraint: $\mathrm{O}(1)$


Images stored in remote
A
B
C
D
E
F
G
H
1
2
3
4
5
6
7
8
(Remote)
A
B
C
G
H
1
2
3
4
5
(Local)
Images currently stored in Local

Guiding question

Now this is clearly a search problem but what should our algorithm be searching for?

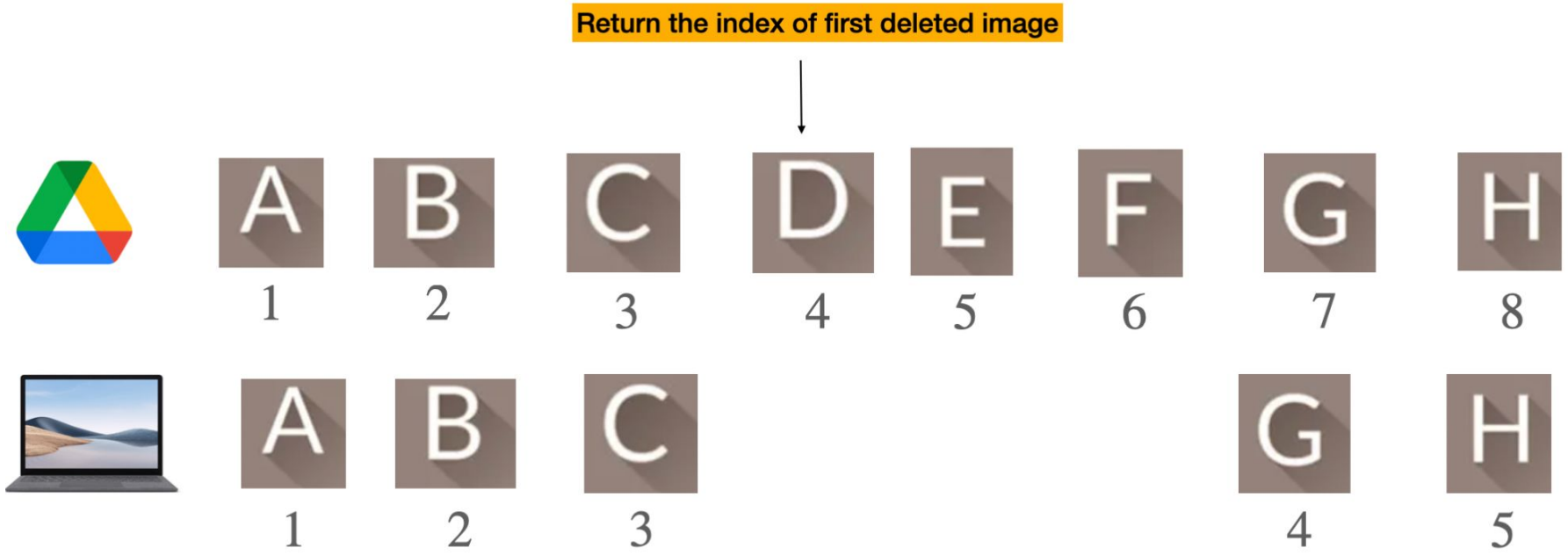
Return the index of first deleted image

A B C D E F G H

1 2 3 4 5 6 7 8

A B C G H

1 2 3 4 5



Thereafter we can recover the missing block from remote (google drive) by downloading next $(\delta - 1)$ images.

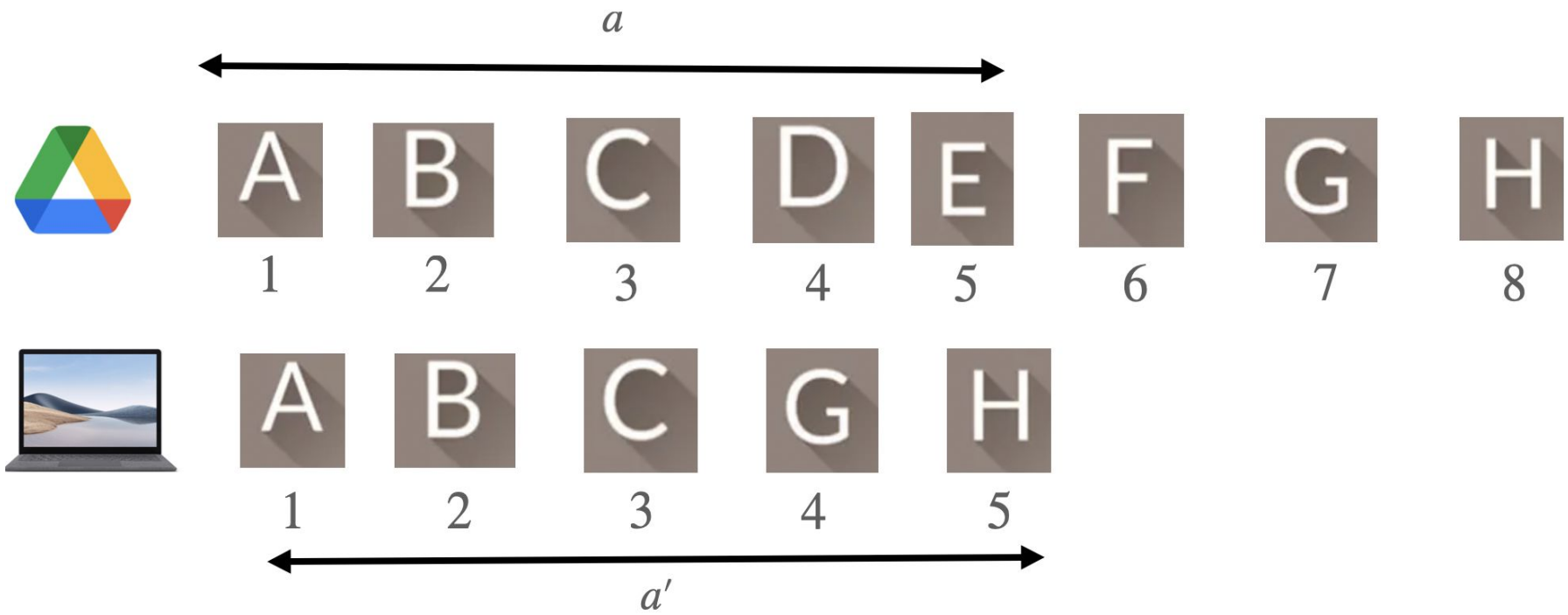
Problem 2.a.

Come up with a solution in which hash values are computed over *contiguous subsequences* of photos.

Explain how and why it works and provide its running time.

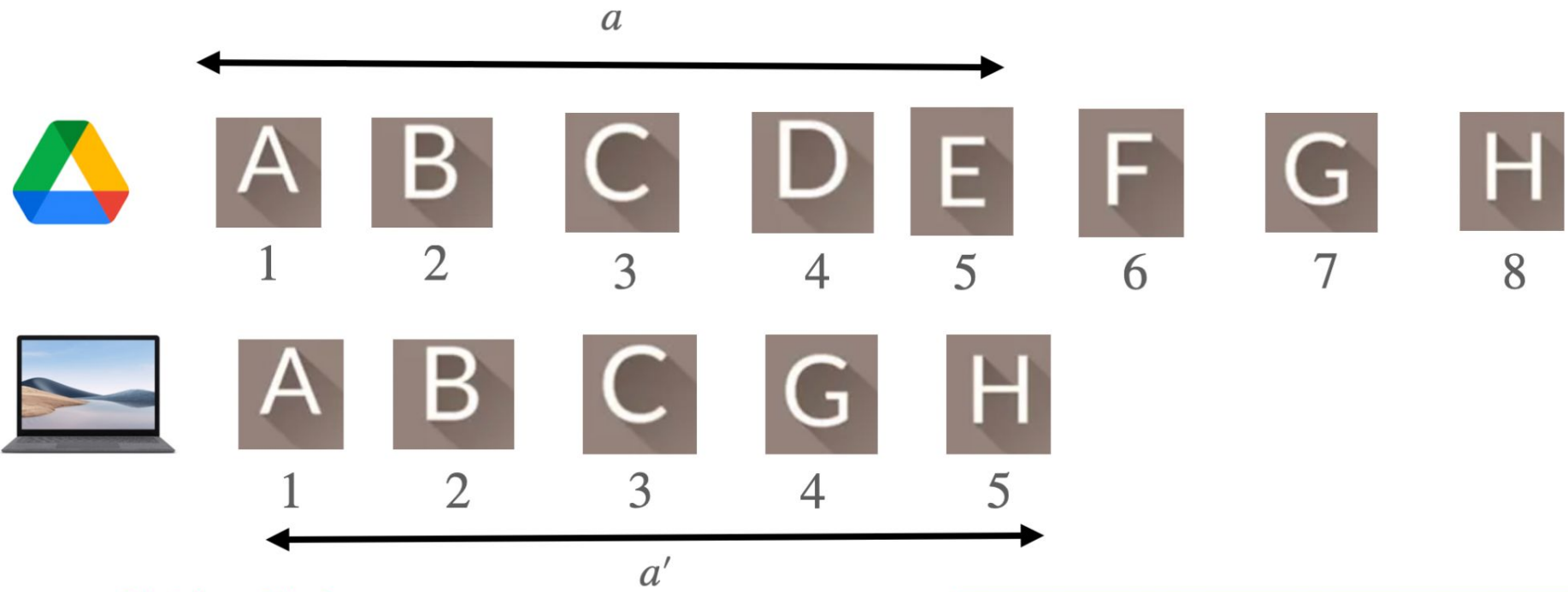
A	B	C	D	E	F	G	H
1	2	3	4	5	6	7	8

A	B	C	G	H
1	2	3	4	5



$h(a) != h(a')$

So we divide the search space in local computer into two halves

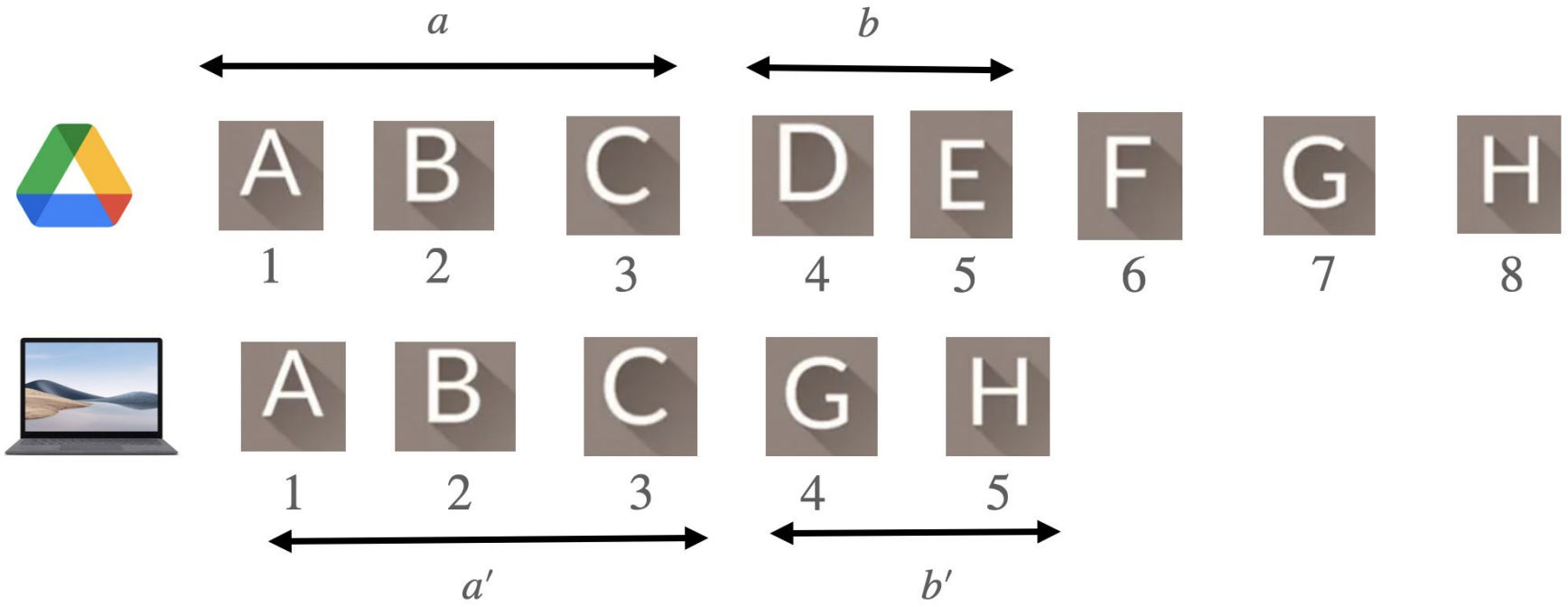


$h(a)! = h(a')$

So we divide the search space in local computer into two halves

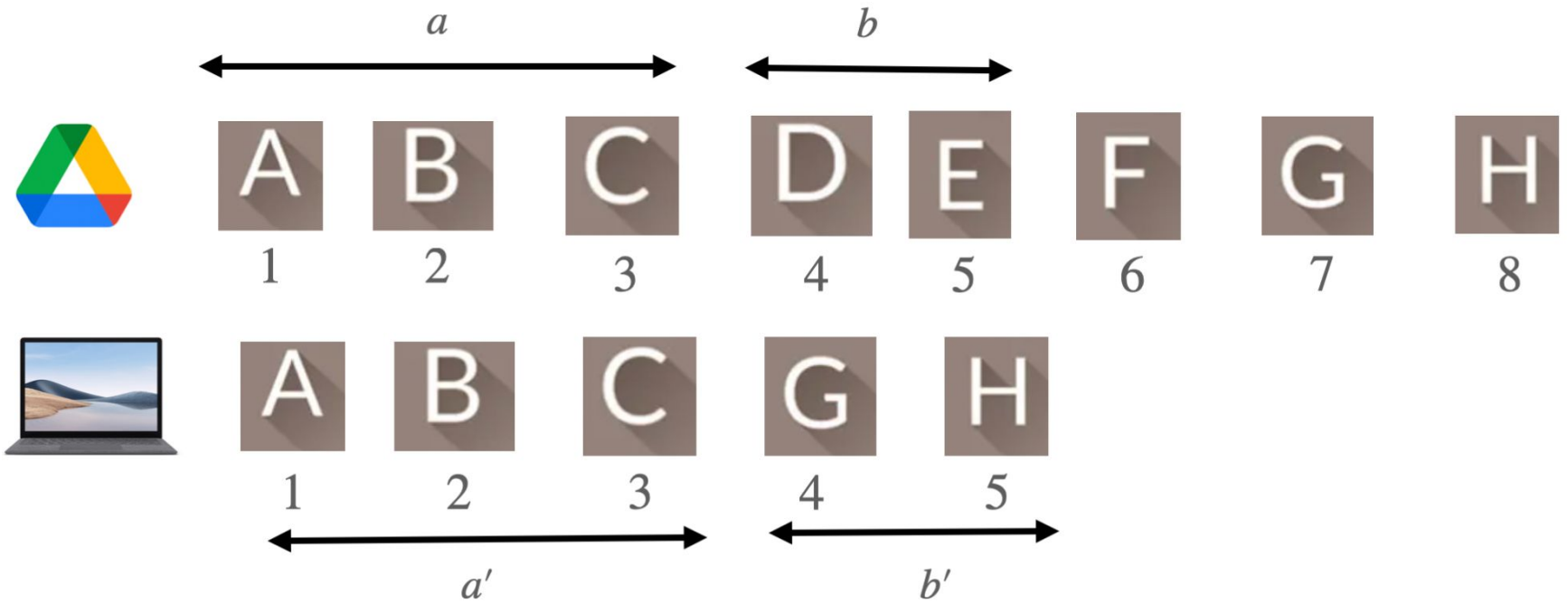
Suppose $h(a) == h(a')$ then deletions must have happened just after the highest index (5 in this example) in the local computer.

Divide the images on the local computer into two halves and compare the hash values with the images present in remote.



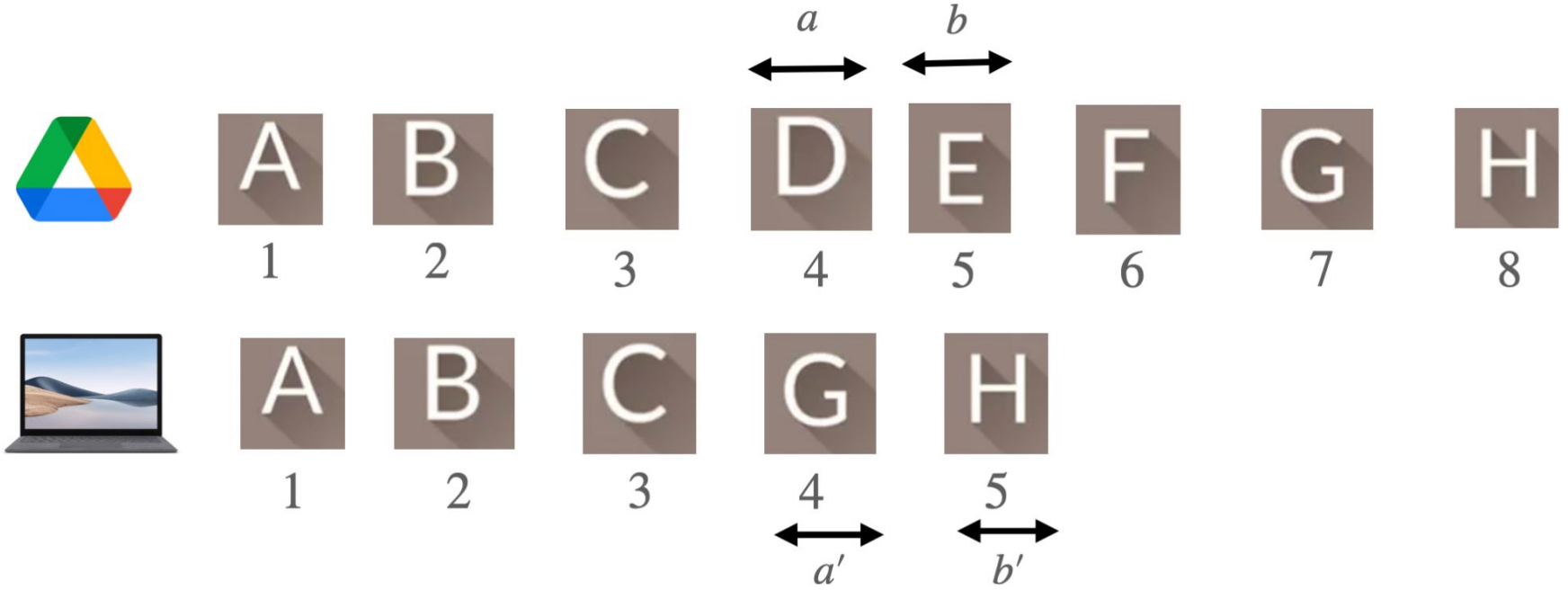
$$h(a) == h(a')$$

Divide the images on the local computer into two halves and compare the hash values with the images present in remote.

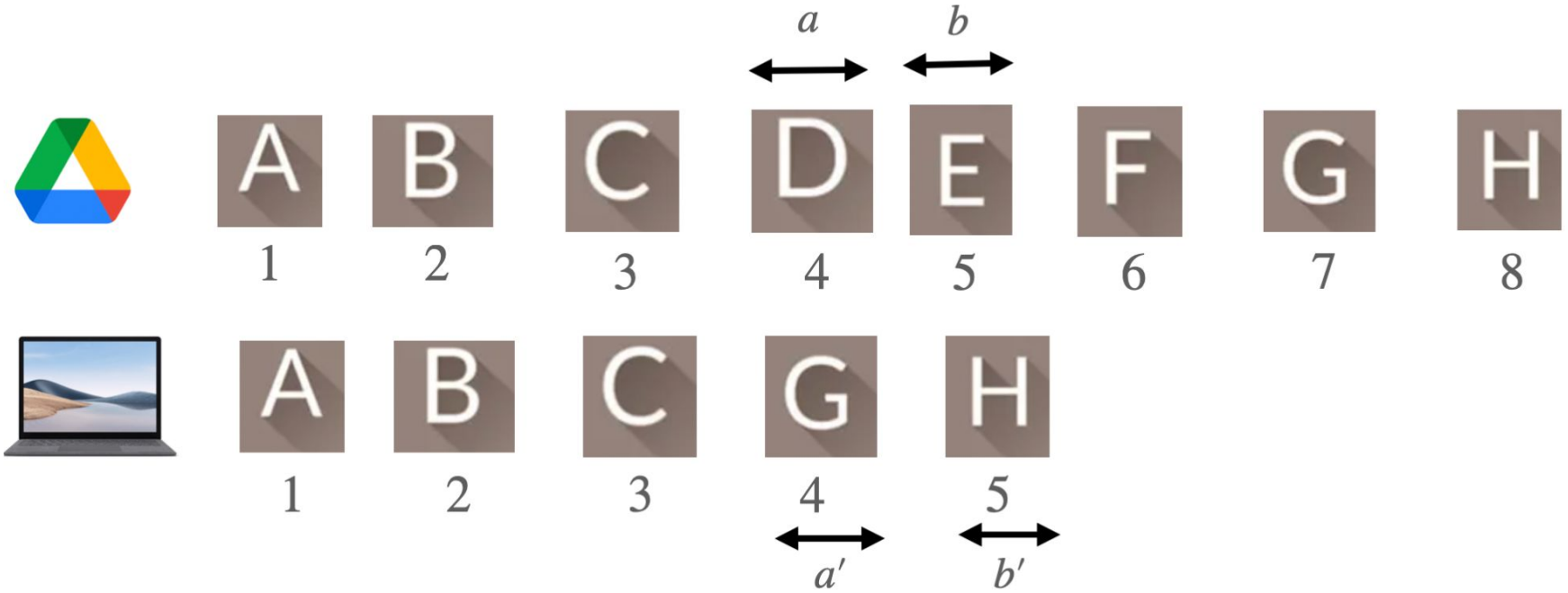


$$h(a) == h(a')$$

So we recurse on the second half

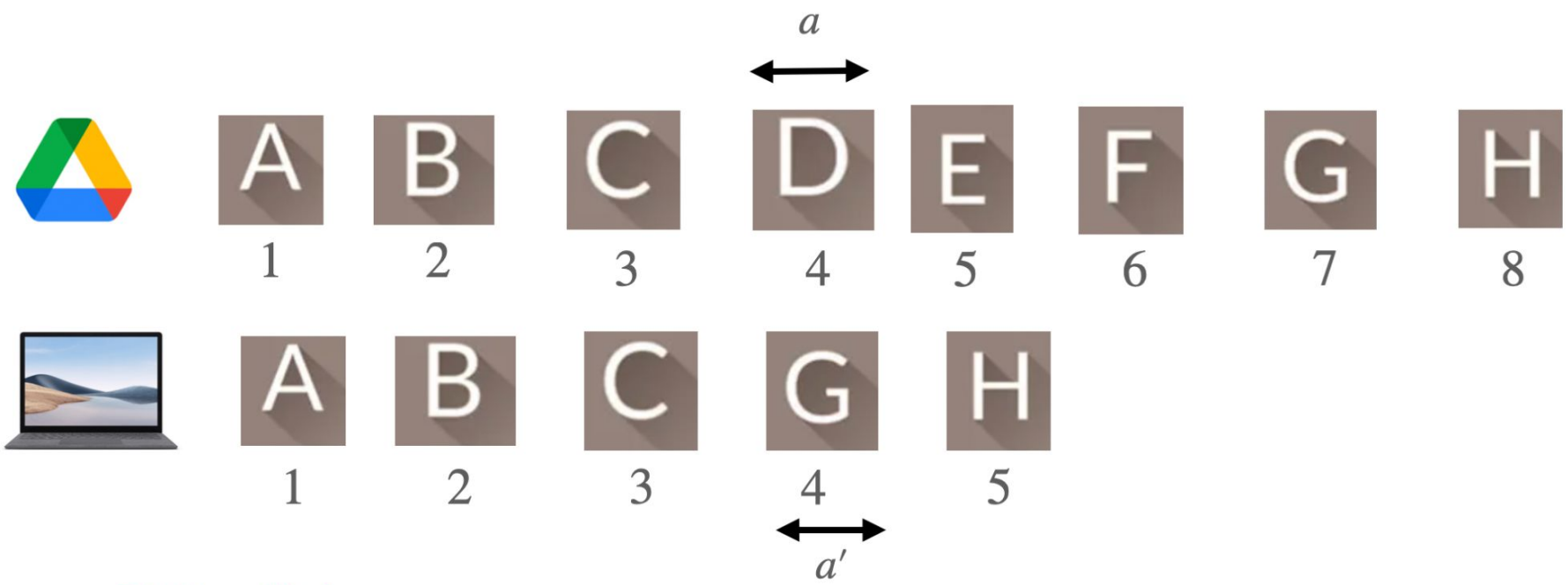


$h(a)! = h(a')$



$h(a)! = h(a')$

In this situation we recurse on the first half



$h(a) != h(a')$

Since, there is only one image now, so we return the index of the image from the remote device.

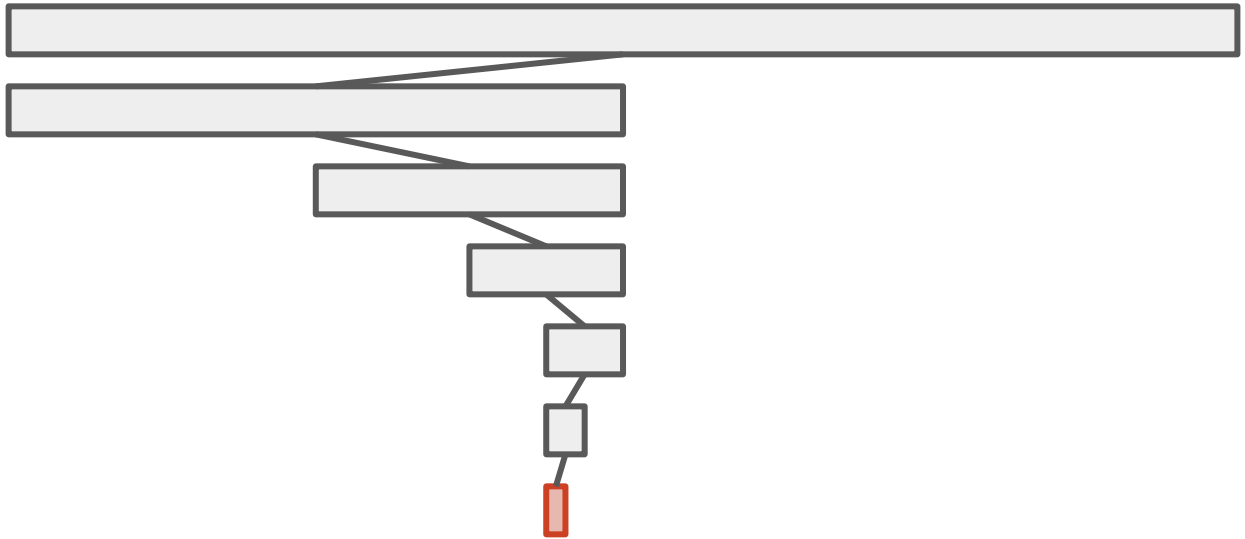
Problem 2.a.—Solution

On demand Merkle Tree:

- Hash the first $n/2$ photos on the client and server and compare
- If the hashes equate, recurse on the second $n/2$ half
- Else, recurse on the first half
- Repeat until you find a missing photo
- Realize that because you will only go left half when there is an inconsistency in the hash values, you will end up with the first deleted photo
- Essentially, you are following a *root-to-leaf* path in the Merkle tree, building the relevant parts of the tree *on the fly* (thus only $\mathrm{O}(1)$ space)

Problem 2.a.—Solution Example

Realise we are effectively doing a root-to-leaf traversal in a Merkle tree without explicitly constructing one?



Problem 2.b.

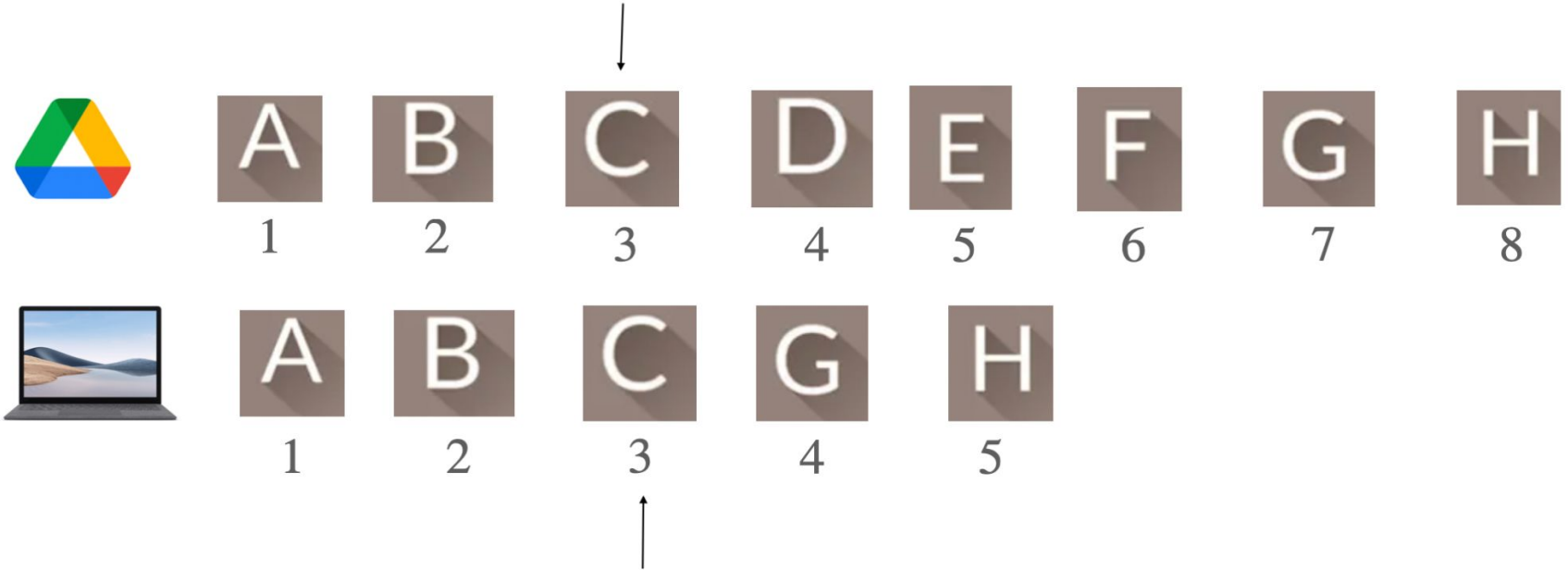
Come up with a solution in which hash values are computed on *individual* photos only.

Explain how and why it works and provide its running time.

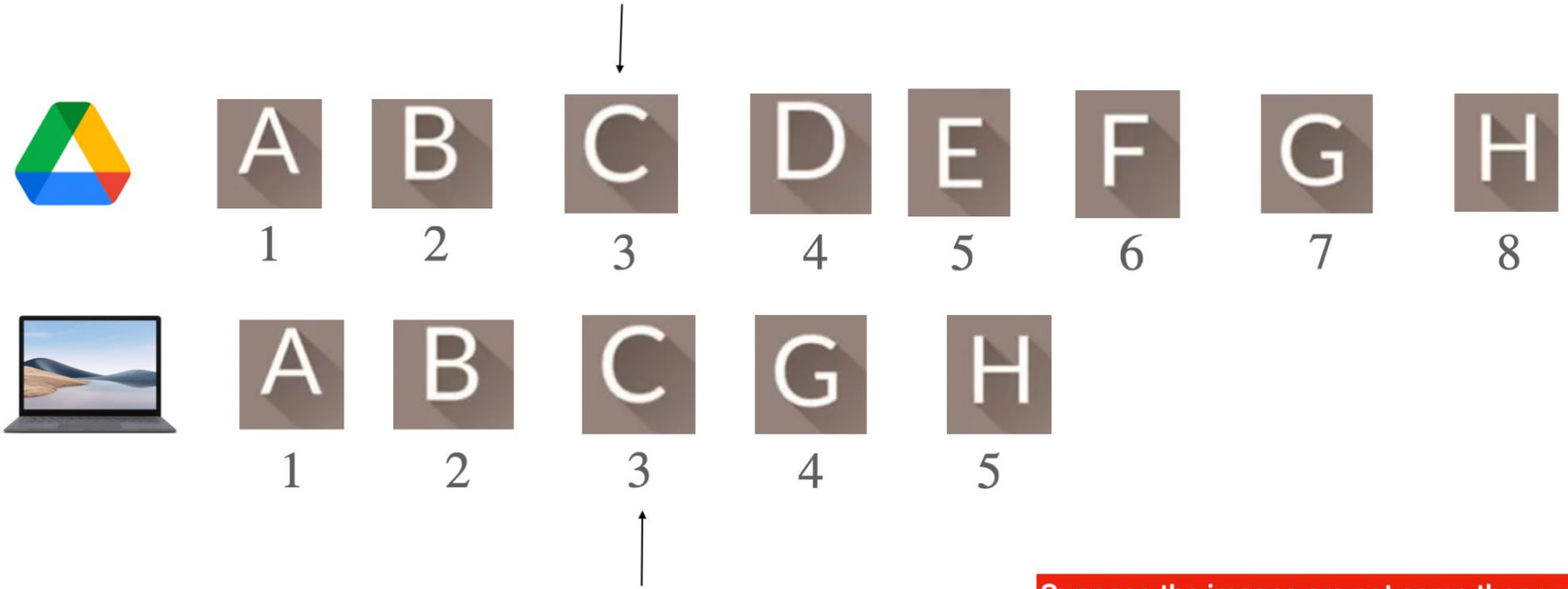
A	B	C	D	E	F	G	H
1	2	3	4	5	6	7	8

A	B	C	G	H
1	2	3	4	5

Since the 3rd element in both places is the image ‘C’ (thus there hash values are equal). So, we recurse on the right half.



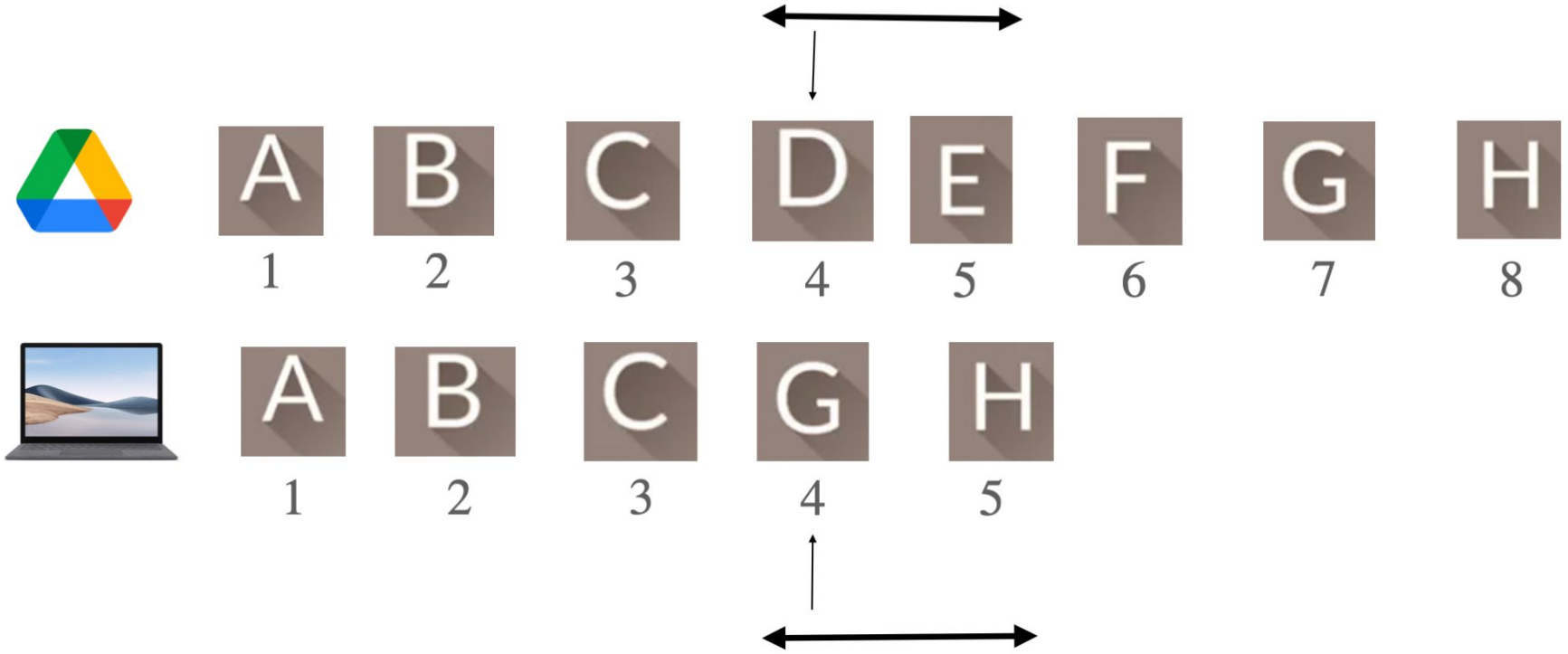
Since the 3rd element in both places is the image ‘C’ (thus there hash values are equal). So, we recurse on the right half.



Suppose the images are not same then we would have recursed on the left half.

Search space, if the images were not same

Since, $h(D) != h(G)$ we will recurse on the left half but there is nothing on left half so we stop here and return the index of image 'D'.



Problem 2.b. — Solution

Observations

- Let j be first index of the photo available remotely but deleted locally
- Let x be any photo index on the local computer
- Realize that if $x < j$, then we are guaranteed remote photo r_x and local photo l_x are the same and hence their hash values will be identical: $h(r_x) = h(l_x)$
- Conversely, realize that if $x \geqslant j$ (assuming of course there is at least one deleted photo), then remote photo r_x and local photo ℓ_x must be different photos since the local indices after j have all shifted down to fill in the gap(s) and so $h(r_x) \neq h(l_x)$ for all $x \geqslant j$

Problem 2.b.－Solution

Key idea:

- Do a binary search for the first deleted photo in the sequence
- Whenever we get an inconsistent hash values, we choose to search on LHS
- When the search converged, we'll end up with the leftmost deleted photo (i.e. first deleted photo in sequence)

Problem 2.b.－Solution

1. Initialize $a = 1$ and $b = m$
2. While $a \neq b$ do:
    1. $x = \lfloor (a + b) / 2 \rfloor$ (i.e. median index)
    2. Compute hash $h(r_x)$ for *remote* photo r_x
        - Else if $h(r_x)$ matches *local* photo hash $h(\ell_x)$, update $a = x + 1$ (i.e. continue searching in the RHS range $[x + 1, b]$)
        - Else, update $b = x - 1$ (i.e. continue searching in the range $[a, x - 1]$
    3. Return a (when $a = b$)
3. The photos to download is therefore $[r_a, r_{a+\delta}]$

Problem 2.c.

What if the deletions done by the virus occurred randomly throughout the photo sequence (i.e. no longer contiguous deletions).

How might you modify your earlier solutions to solve for this?

A	B	C	D	E	F	G	H
1	2	3	4	5	6	7	8

A	B			E		G	H
1	2			3		4	5

If we apply the previous algorithm then it will return the index of image ‘C’

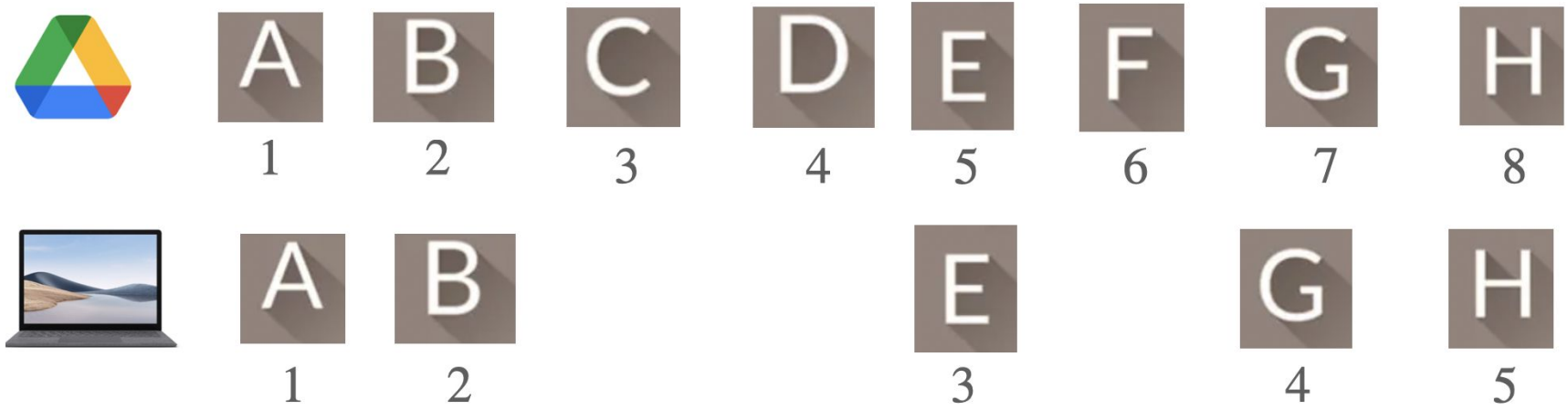
A B C D E F G H

1 2 3 4 5 6 7 8

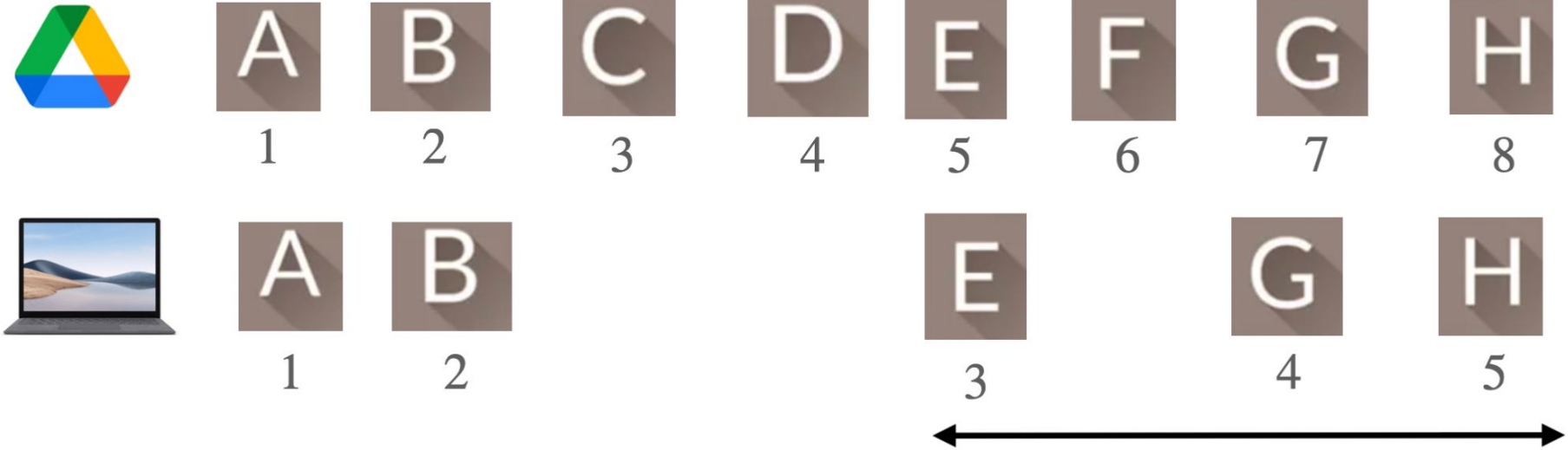
A B E G H

1 2 3 4 5

After that we will do a linear scan until we hit an image which present in our local computer.



After that we reduce our search space and find the next set of deleted images



Problem 2.c.－Solution

Suppose there are δ photos deleted.

Then we just need to run our searching algorithms δ times:

- Each time identify index i that is the *first* (leftmost) deleted photo in the current search space
- Repeat search procedure on the suffix of the sequence after i (i.e. next search space start from $i+1$ item onwards)

Problem 2.c.—Discussion

Each search routine costs $\mathrm{O}(\log n)$ so identifying all δ items cost $\mathrm{O}(\delta \log n)$.

If δ is in the order of n, then this approach is of course inferior to a linear search.

However if $\delta \lll n$, then this is potentially faster.

Suppose for now

- h is no longer guaranteed to be a perfect hash function
- The virus deleted photos randomly
- No longer constrained by $\mathrm{O}(\log n)$ number transfers

Alice’s proposal 1

1. Pick a hash function h that maps a photograph to an integer in the range $1, \ldots, n$
2. For each photo $\ell_i : i \in [1, m]$ on Alice's *local* computer
   i. Compute its hash value $h(\ell_i)$
   ii. Save $h(\ell_i)$ to a local file H_ℓ
3. For each photo r_i on the *remote* server
   i. Compute its hash value $h(r_i)$
   ii. 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

Alice Proposal 1

Alice

Creates two txt files H_r and H_l

$h(A) = 4$
$h(B) = 6$
$h(C) = 8$
$h(D) = 1$
$h(E) = 3$
$h(F) = 5$
$h(G) = 7$
$h(H) = 2$

H_r

$h(A) = 4$
$h(B) = 6$
$h(C) = 8$
$h(D) = 1$
$h(G) = 7$
$h(H) = 2$

H_l

Just download the images whose hash values are not in H_l but present in H_r

Remote)

(Local)

Problem 2.d.

What are Alice’s objectives of using a hash function in this scheme?

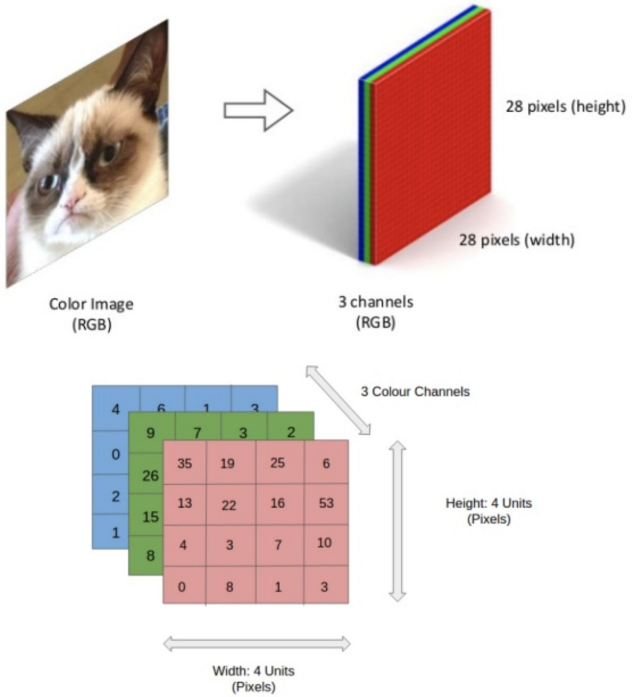
What is the *key* to success in achieving those objectives?

Image file sizes

A colour image is a 3rd order tensor (i.e array of dimensionality 3). For a 1080p resolution colour image, it has 1920px by 1080px. That means comparing 1080p colour images will take

$$1920 \times 1080 \times 3 = 6{,}220{,}800$$

byte comparisons! Even with image compression, the number of comparisons needed will still be very expensive!



Source: Berton Earnshaw

Proposal 1: Objectives

Three objectives of using a hash function:

1. Photo **signature**: Uniquely identify *each and every* photo so as to determine missing local photos
2. Fast **transmission**: To download (main bottleneck) a small hash value as opposed to downloading an entire image
3. Efficient **comparisons**: To *compare* photos in $O(1)$ and avoid image-level comparison which is slow

Objective 1 imposes a hard constraint. The key to success in Alice's strategy is therefore having a *perfect hash function*.

Problem 2.e.

Is H_ℓ a hash table?

$h(A) = 4$

$h(B) = 6$

$h(C) = 8$

$h(D) = 1$

$h(G) = 7$

$h(H) = 2$

H_l

Guiding question

If H_ℓ were a hash table, what would be its keys?

Guiding question

If H_ℓ were a hash table, what would be its keys?

Answer: Since we cannot rely on filenames and metadata in the context of this question, the keys have to somehow refer to the photos themselves!

Realise hash tables *minimally* need to store keys because they are used for comparisons during collision resolution.

Problem 2.e.—Discussion

Therefore H_ℓ cannot be considered as a hash table because it doesn't refer back to the images and has **no collision resolution** strategy in place!

It's simply a *list* or *set* of hash values.

Problem 2.f.

Alice claims that this scheme will efficiently restore all the missing photos to her computer.

Is she right? Explain why or why not.

H_r

$h(A) = 4$

$h(B) = 6$

$h(C) = 8$

$h(D) = 1$

$h(E) = 3$

$h(F) = 5$

$h(G) = 7$

$h(H) = 2$

H_l

$h(A) = 4$

$h(B) = 6$

$h(C) = 8$

$h(D) = 1$

$h(G) = 7$

$h(H) = 2$

Just download the images whose hash values are not in H_l but present in H_r

Problem 2.f.

Alice claims that this scheme will efficiently restore all the missing photos to her computer.

Is she right? Explain why or why not.

H_r

$h(A) = 4$

$h(B) = 6$

$h(C) = 8$

$h(D) = 1$

$h(E) = 4$

$h(F) = 6$

$h(G) = 7$

$h(H) = 2$

$h(A) = 4$

$h(B) = 6$

$h(C) = 8$

$h(D) = 1$

$h(G) = 7$

$h(H) = 2$

H_l

If the collision between the set of deleted images and the set of non-deleted images occur then Alice would have missed some deleted images to download.

Problem 2.f.

Alice claims that this scheme will efficiently restore all the missing photos to her computer.

Is she right? Explain why or why not.

H_r

$h(A) = 4$

$h(B) = 6$

$h(C) = 8$

$h(D) = 1$

$h(E) = 4$

$h(F) = 6$

$h(G) = 7$

$h(H) = 2$

H_l

$h(A) = 4$

$h(B) = 6$

$h(C) = 8$

$h(D) = 1$

$h(G) = 7$

$h(H) = 2$

Observe that, no problem occurs if the collision occurs only within the set of non-deleted images or within the set of deleted images.

Problem 2.f.—Discussion

No Alice is **not right** because there can be collisions! Her solution is contingent on h being a perfect hash function, which there are no guarantees of.

If a deleted photo and a non-deleted photo is hashed to the same value, we would be led to believe that the deleted photo exists locally.

Problem 2.g.

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?

Guiding question

What else need to be added to H_ℓ if we wish to incorporate collision strategies?

Guiding question

What else need to be added to H_ℓ if we wish to incorporate collision strategies?

Answer: We now need to store each photo (or their reference/location) as *keys* in order to differentiate between different photo hashed to the same address.

Problem 2.g.—Discussion

Realize that adding separate chaining to H_ℓ turns it into a table with collision resolution.

This would achieve objective 1 by uniquely identifying each photo.

However, it now fails objective 3 because we can no longer compare photos in $\mathrm{O}(1)$.

Guiding question

When will we fail to compare photos in $O(1)$?

Guiding question

When will we fail to compare photos in $\mathrm{O}(1)$?

Answer: When a remote hash $h(r_i)$ is also found in H_ℓ, we have no choice but to also download remote photo r_i and conduct image matching against every local photo that shares the same hash value $h(r_i)$.

This would happen **very frequently** if only a few photos were deleted!

Problem 2.g.—Discussion

Hash tables in this problem is therefore *impractical*!

It defeats the whole point of using a hash function in the first place!

Problem 2.g.－Observations

So Alice’s proposal 1 isn’t quite right.. But all is not lost! We can make some important observations regarding her remote-local hash value comparing strategy.

There are just 2 scenarios:

1. When a remote signature is found locally
2. When a remote signature is not found locally

Problem 2.g.－Observations

When a remote signature is **found locally**,

It can be a **false positive** due to collisions:

- If a hash value of a photo on the server is found locally, this need not mean that that same photo exist locally
- I.e. another local photo might have the same hash value

Problem 2.g.－Observations

When a remote signature is **not found** locally,

It is always a **true negative**:

- We will never get false negatives
- If a photo on the server has a hash that does not correspond to a hash value locally, then it is really missing locally!

Problem 2.g.—Insight

Therefore true negatives are the only reliable indicators Alice can rely on!

How can she ensure that every deleted photo will be detected via a true negative check?

Alice’s proposal 2

1. Let k be some integer (which may depend on n and m)
2. Repeat:
   i. Randomly pick a hash function h that maps a photograph to an integer in $[1, k]$
   ii. For each photo $\ell_i : i \in [1, m]$ on Alice's local computer
      - Compute its hash value $h(\ell_i)$
      - Save $h(\ell_i)$ to a local file H_ℓ
   iii. For each photo $r_i : i \in [1, n]$ on the remote server
      - Compute its hash value $h(r_i)$
      - Save $h(r_i)$ to a remote file H_r
   iv. Download H_r to Alice's local computer
      - If $|H_r| - |H_\ell| = n - m$,
        1. Download the photos r_i whose hash value $h(r_i)$ is in H_r but not in H_ℓ
        2. Terminate the repeat loop
      - Else, continue the loop to look for a better hash function

Alice Proposal 2

Alice

(1) Picks any number k (let us take $k = 100$ for this example)

(2) Chooses a random hash function which maps each image to a number [1, k]

(3) If $|H_r - H_l| = n - m$ then stop, otherwise again find a random hash function.

n = # images in remote

m = # images in local

(Remote)

(Local)

Alice Proposal 2

Alice

(1) Picks any number k (let us take $k = 100$ for this example)

(2) Chooses a random hash function which maps each image to a number [1, k]

(3) If $|H_r - H_l| = n - m$ then stop, otherwise again find a random hash function.

$h(A) = 40$
$h(B) = 60$
$h(C) = 80$
$h(D) = 10$
$h(E) = 30$
$h(F) = 50$
$h(G) = 70$
$h(H) = 20$

H_r

$h(A) = 40$
$h(B) = 60$
$h(C) = 80$
$h(D) = 10$
$h(G) = 70$
$h(H) = 20$

H_l

Just download the images whose hash values are not in H_l but present in H_r

(Remote)

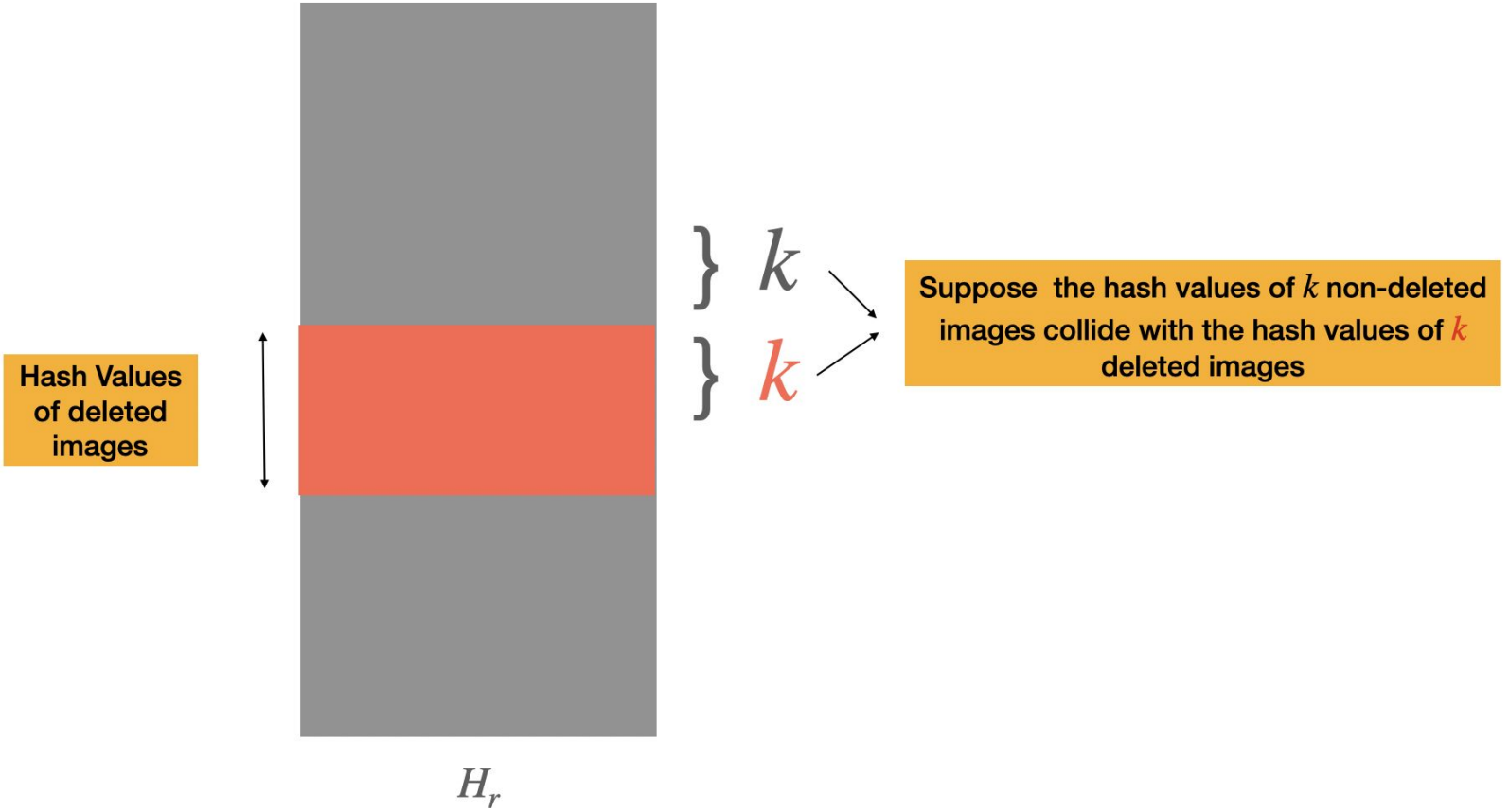
(Local)

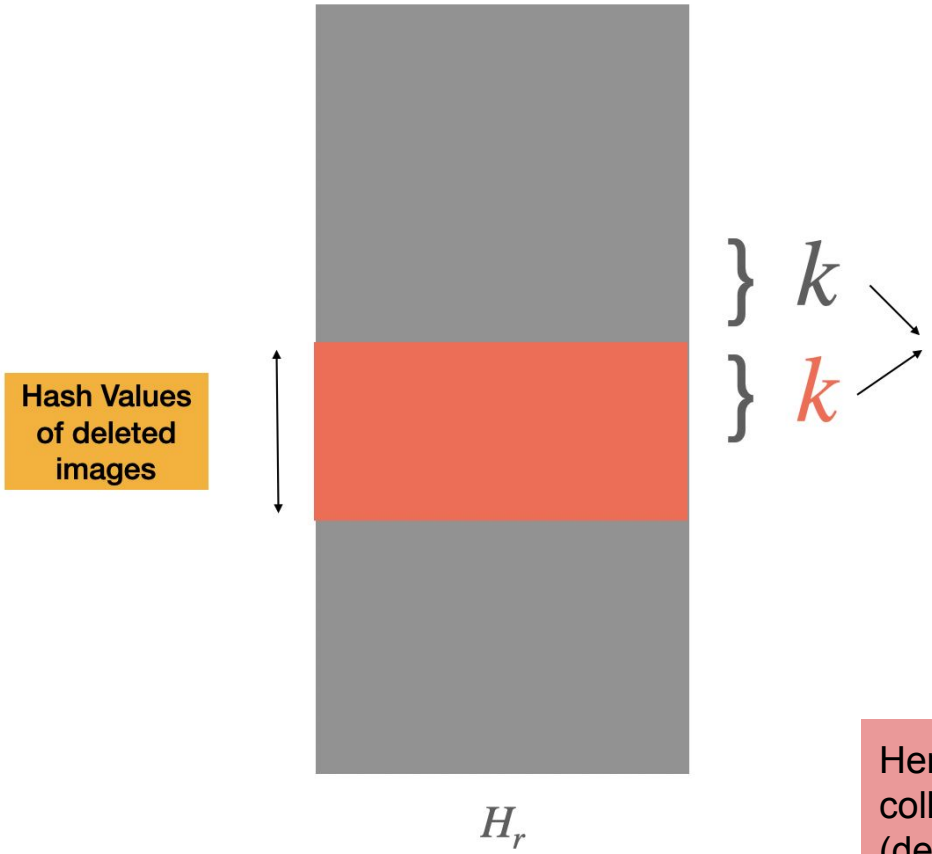
Problem 2.j.

In the second scheme, what is Alice's objective of using a hash function?

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.

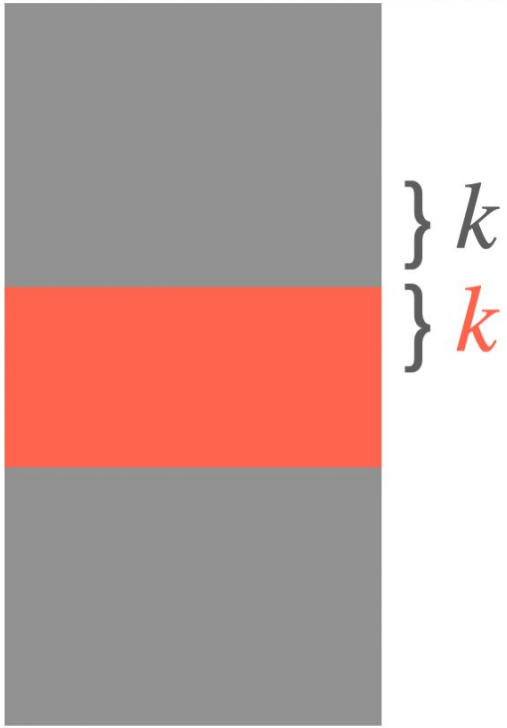

Hash Values of deleted images
k
k
Suppose the hash values of k non-deleted images collide with the hash values of k deleted images
H_r



Suppose the hash values of k non-deleted images collide with the hash values of k deleted images

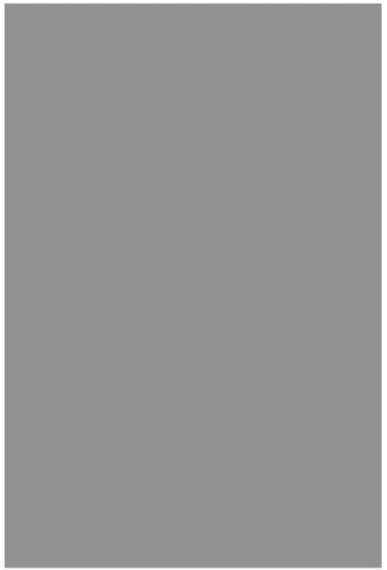
Then the number of distinct elements in H_r is $(n - 2k + 1)$ (where n is the total number of images present remotely.)

Here, for simplicity, I am assuming that collisions also happened within these k (deleted or non-deleted) images and no collisions other than these happened. So, all these 2k hash values are same here.



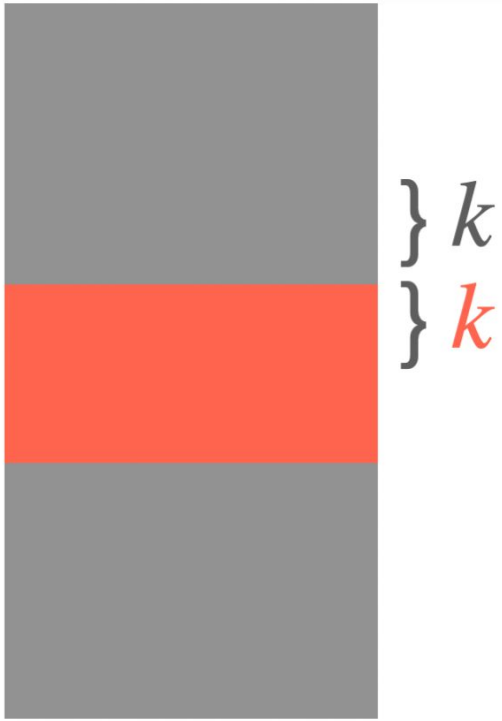
H_r

Then the number of distinct elements in H_r is $(n - 2k + 1)$ (where n is the total number of images present remotely.)



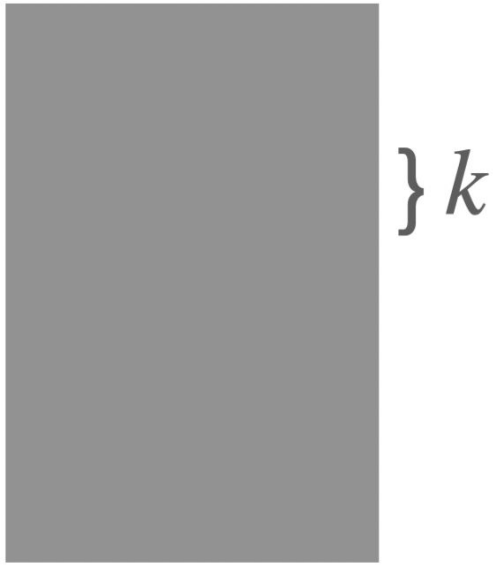
H_l

Then the number of distinct elements in H_l is $(m - k + 1)$ (where m is the total number of images present locally.)



H_r

Then the number of distinct elements in H_r is $(n - 2k + 1)$ (where n is the total number of images present remotely.)



H_l

Then the number of distinct elements in H_l is $(m - k + 1)$ (where m is the total number of images present locally.)

Hence, $|H_r - H_l| = (n - m - k) \neq (n - m)$

$\} k$

$\} k$

- Here, I have assumed that all these k values for non-deleted or deleted images collided within themselves but this might not happen always.
- There can be cases where only a subset of these k values collided within themselves.
- There can be some cases where collisions happened in the remaining part (other than these k deleted or non-deleted images).
- You can try to analyse all these cases by yourself and try to convince yourself that $|H_r - H_l| \neq (n - m)$ for all these cases if there are collisions in between deleted and non-deleted images.

Problem 2.h.

An interesting criteria for picking the random hash function:

$$|H_r| - |H_\ell| = n - m$$

Why didn't Alice simply chose the condition to be :

$$|H_r| = n \text{ and } |H_\ell| = m$$

Guiding question

What does it mean when we have

$$|H_r| = n \text{ and } |H_\ell| = m$$

Guiding question

What does it mean when we have

$$|H_r| = n \text{ and } |H_\ell| = m$$

Answer: A perfect hash function.

Problem 2.h.—Discussion

Therefore ($|H_r| = n$ and $|H_\ell| = m$) is not a good criteria because we will potentially have to iterate very long until we find a perfect hash function by randomly picking one.

But do we even need a perfect hash function?

Problem 2.i.

If we think about $|H_r|$ and $|H_\ell|$ respectively as the hash values before and after a set of deletion operations,

what is the “invariance” in the desired hash function here?

Problem 2.i.—Discussion

The “invariance”:

- After having δ photos deleted locally
- The number of hash values also decrease by δ

Of course this is a one-off invariance which only holds for the deleted photos, it will not hold if we delete more photos later on.