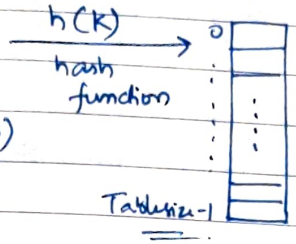
① Hashing Techniques :-

A hash table is an array of some fixed size (usually a prime no).



Hash func :-

1. Simple & fast to compute
2. Avoid collisions.
3. Keys distributed evenly among the Cells.

Collision - When 2 keys map to the same location in hash table.

② 2 Ways to resolve Collisions :-

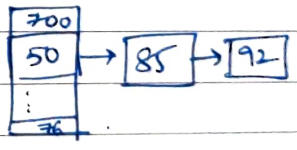
(i) Seperate ~~Hashing~~ Chaining.
(ii) Open Addressing

→ Separate Chaining :- All the keys that map to same hash value are kept in a (list/ bucket).

Note: ('hashing' is technique used to index & retrieve items in a database in a reasonable amount of complexity).

→ Main idea is to make each cell of hash table point to linked list of records that have same hash function value.

ex: $h(x) = x \bmod 7$ if keys = 50, 700, 76, 85, 92 ⇒



disadv :- Cache performance is not good as keys ~~as~~ are stored using linked list.

→ Wastage of space. , Extra space for links (∵ we are storing addresses).

→ If chain becomes long, search space can become O(n) in worst case.

Time Complexity : $O(1+\alpha)$ α = Avg. keys per slot (or) load factor $= n/m$ (n → # keys stored in table, m → # slot in table)

Open Addressing :-

→ All items (keys) are stored in table itself. (size of table ≥ #keys).
→ Hash func. specifies order of slots to probe (try) for a key (for insert / search / delete), not just one slot.

(i) **Linear Probing** :- $h_i(x) = [(Hash(x) + i) \% \text{Hash table size}]$

(try for nxt empty slot)

probe means

i.e If $h_0 = (Hash(x) + 0) \%$ hash table size if it is full then try
$h_1 = (Hash(x) + 1)$ " " try h_2 for h_2
and so on.
(Keep probing until empty slot is empty).

(ii) **Quadratic Probing**: $h_i(x) = [Hash(x) + i^{\wedge}2) \% \text{Hash table size}]$

(iii) **Double Hashing** : $h_i(x) = [Hash_1(x) + i * Hash_2(x)] \% \text{Hash table size}$
(Here diff is 2 diff hash func. are used).

Comparision :-

Linear probing	Quadratic probing	Double hashing
→ Easy to implement	→ Avg cache performance	→ poor cache performance
→ But cache performance	→ Suffers less clustering than linear probing (∵ easier to find empty slot here)	→ No clustering
→ Suffers from clustering		→ Requires more computation time.

Complexity : 'α' → load factor $\alpha = n/m$ → # keys to be inserted in hash table. → # slots

$= O\left(\frac{1}{1-\alpha}\right)$ time.

Cuckoo Hashing :-

→ multiple hash tables (not 1). → (only 1 item per bucket) no chaining.
→ multiple hash functions (1 per hash table).

procedure :-

1st iteration – all items use h_1 & write to t_1
2nd (some fail)
– 2nd iter. – Those that fail use h_2 & t_2.
repeat.

round 1 — hash values.

items	h_1	h_2
A	1	1
B	0	1
C	1	0
D	0	0

use h_1 func. → t_1: D. $h_1=0$; C. $h_1=1$

use h_2 → t_2: $h_2=0$; $h_2=1$

A & C try to go to same bucket

round 2. If C wins. Hy for B, if B wins.

	h_2
A	1
B	1
C	0
D	0

round 2 use table t_2 → B. $h_2=0$; B $h_2=1$ (If B wins)

round 3 use table 1.

	h_1	h_2
A	1	1

t_1: D $h_1=0$; A $h_1=1$ — collide with item in t_1.

t_2: B

So, hash again ⇒ kick out C in t_1 & Continue algorithm.

i.e.

	h_1	h_2
A	1	1
B	0	1
C	1	0
D	0	0

round 4 use table t_2 → C goes here ∵ $h_2=0$ & it's empty.

t_2: C. $h_2=0$; B $h_2=1$

Advs :-

Constant look up time.

Construction is simple & fast.

- Shared mem - really fast.
- Global mem - requires atomics. (write an item in table & kick the other items should be atomic).

2.) Filters And Sketching.

Bloom Filters :- (Space Efficient "probabilistic" data structure).

Note: In order to compare 2 string 'u' can just compare their hash values & if they are same hash value → then its highly likely that 2 strings are same.

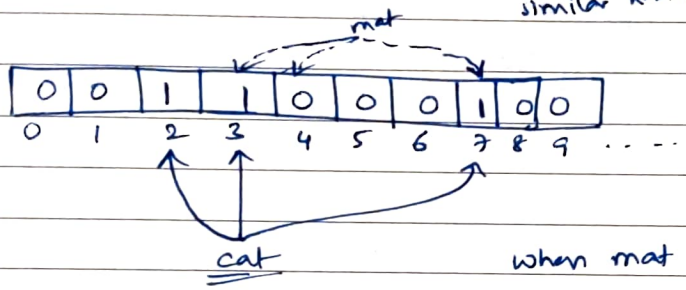
(In general we pass strings to multiple hash funcs. & get their hash values (h_1 h_2 h_3 ...)

(if "ele" present in the set) → yes (false positive are possible).
→ no (if ele is not there but bloom filter says it is there).
[false -ve not possible]

Ex: cat → pass through

	cat	mat
h_1 →	3	4
h_2 →	7	7
h_3 →	2	2

(∵ 2nd char is same) assumption similar hash func gets this values (∴ a, t is same)



when mat is being searched.

we get 1 0 1

3 4 7 bits

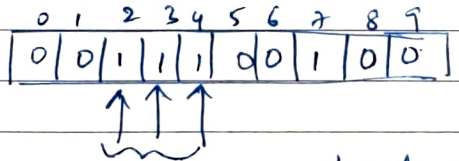
So, set (indices of mat) to 1.

∵ not all are 1. we can clearly say "mat" is not searched.

now if 'mat' is searched → we get 1 1 1. its there.

Bloom filter → has func. has "reset" → if 70% are 1's. (then bloom filter might have less probability) to say yes/no). → then u can reset().

Now if some word like 'fat' → h_1 → 3, h_2 → 4, h_3 → 2



all are 1's → already searched. (false positive) (only drawback).

m → bit array
K hash funcs.
n → strings.

[No false -ves] → i.e (ele is there but bloom filter says is not present) not such case in bloom filter.

Trade off: less mem. → (so, some false +ve) → (but no false -ves).

Using bloom filter reduces caching workload. → ↑es cbaching hit rate.

↳ (check url seen). → (only caching pg on second request).

chrome.

↳ to identify malicious url. → check fusiy bloom filter } → only perform expensive oper → verify → remote Server

(known malicious urls)

if it returns bloom filter probably (yes) / definitely no.

↓

if already known malicious (just discard)

---

(2) Quotient Filter:- approximate membership query data structure.

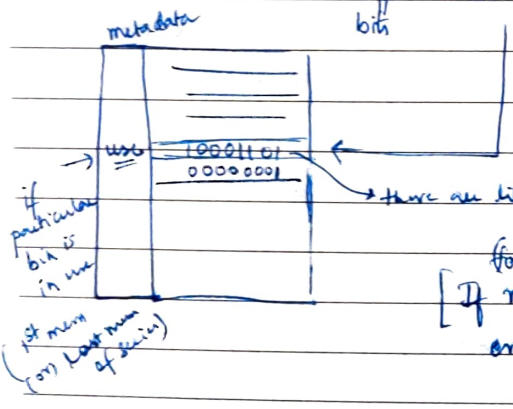
Squeaker: an exact & approx. k-mer counting syst

Counting quotient filter

hash (Kmer) → 10110101 : 10001101

upper bits → location

remainder. (lower bits)



→ these are like finger prints (for new kmer)

→ you can use for querying & also counting (based on val in not location)

[If new hash produces same location → then go to next empty slot.]

④ Locality Sensitive Hashing (LSH):

ex: Similar docs → similar hash-code.
For each doc 'd' → generate k-bit hash code.
insert doc in hash table.

** → If it's collision → possible duplicate (compare docs in same bucket)
(sometimes it might be accident (not same docs))

Can miss near-duplicates
- similar hash-code ≠ same bucket
repeat L times w. diff hash tables (randomized)

	H1	H2	H3
d_a	0100	1011	0110
d_s	1100	1011	0010

↓ coll (collision).

cut 3D space by a plane making some angle with xy, yz, zx.

→ Can generate random hyper planes (for generating hashcodes)
∵ mainly want: (similar hash code for nearby points).
→ Then do comparisons within same bucket → to eliminate false positives.
→ Repeat with diff hyper planes. (some x no. of times). (if a, d are far apart) inactual.

cost: n pts, d-dimensions, k-hyperplanes.
D_k → find bucket where pt lands.
2^k → hashcodes / diff regions (due to intersection of hyperplanes)
$(N/2^k)$ - pts in bucket
$(DN/2^k)$ → cost of comparision (inside a bucket).

(Repeat L times). LSH: $LDK + LDN/2^k \rightarrow d\log N$ if $k = \log N$.

$O(1)$.

Jaccard Similarity:-

$$Sim(C_1, C_2) = |C_1 \cap C_2| / |C_1 \cup C_2|$$

size of intersection
↳ union.

min hashing:-

	permuted rows	C_1	C_2	C_3	C_4	Input matrix
1 → 4	2	1	0	1	0	
3 → 2	4	1	0	0	1	
7 → 1	7	0	1	0	1	
6 → 3	6	0	1	0	1	
2 → 6	1	0	1	0	1	
5 → 7	2	1	0	1	0	
4 → 5	5	1	0	1	0	

hash function
signature matrix

2	1	2	1

→ formed based on 1st permuted rows.
val. based on 1st permuted rows

2	1	4	1

→ based on 2nd permuted rows.

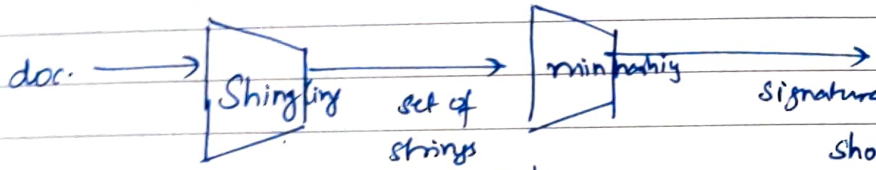
1	2	1	2

→ " 3rd.

dis: If 1 billion rows → no. of pe. → very huge → large disk space (4GB). → accessing permuted rows leads to thrashing.

So, use hash fun. for permuting rows. $h_i(r)$ → i^{th} permutation.

Min hashing



short integer vector that represent the sets & reflect their similarity

elements	S_1	S_2	S_3	S_4 → docs.
a	1	0	0	1
b	0	0	1	0
c	0	1	0	1
d	1	0	1	1
e	0	0	1	0

$S_1 = \{a, d\}$ $S_2 = \{c\}$
$S_3 = \{b, d, e\}$ $S_4 = \{a, c, d\}$.

- a, d are present in doc (S1).

- Charecteristic matrix.

If permuted order is →

b	0	0	1	0
e	0	0	1	0
a	1	0	0	1
d	1	0	1	1
c	0	1	0	1

min hash value.

a	c	b	a.

based on where u see 1st one.

$h(S_1) = h(S_2)$

we see Min hash produces similar result as jacard similarity

rows	S_1	S_2	S_3	S_4	$x+1 \bmod 5$ (h_1)	$3x+1 \bmod 5$ (h_2)
0	1	0	0	1	1	1
1	0	0	1	0	2	4
2	0	1	0	1	3	2
3	1	0	1	1	4	0
4	0	0	1	0	0	3

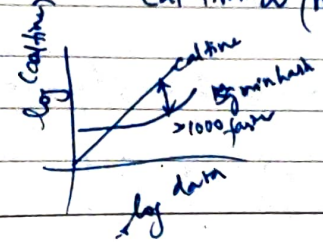
→

	S_1	S_2	S_3	S_4
h_1	1	3	0	1
h_2	0	2	0	0

signature matrix

Weighted min hash.

cal time $\propto$ (no. of ele * sig size) / 4



h_1	h_2
1	1
2	
	2
	0
0	

→ smallest hash value of each ele. reduces no. of dimensionality

⑤ Approximate Nearest Neighbour Search:-

ANN ≃ KNN

In KNN → we find k-closest/nearest data pt for the given pt (physically in space)

Ex: 3 closest word for a given word.

Task: Given a point (x) → Find K closest Neighbours.

But... N points means O(N).

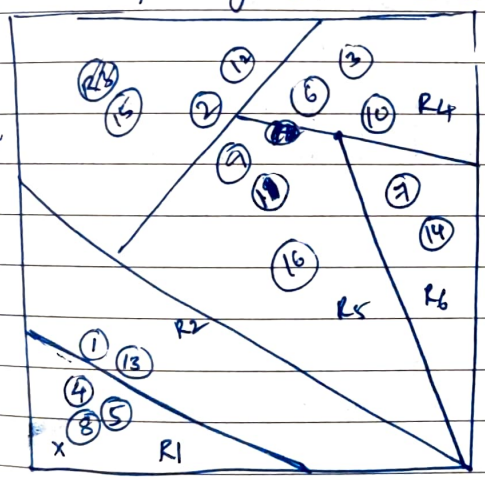
i

0	1	2
d_0	d_1	d_2

→ k closest (if k=3)

→ keep track of distance of pts // (brute force) → (dist b/w given pt & pts in space).

What if we got almost Closest Neighbours:-



(in fraction time).

1. pick any 2 data pts 1, 9 in random.
2. draw a line equi-distant from 2 points.

(1,9)

below (pick any 2 pts) | above (6,12)

(1,4) | 2,12,15 (R3) | (10,7)

4,5,8 ∴ 3 data pts (R1) | 1,13 <3 pts (R2) | (7,11) | 3,6,10 R4

9,11,16 R5 | 7,14 R6.

Squarely Line: below it / above it

(ANNOY)

(Approximate nearest neighbours oh yeah)

designed by smart folks in Spotify

After finding where the new pt lies? (In which region).

→ (if 'x' data pt is given we can just find distance with R1 regions & find the nearest data point)

O(log N).

Note: [Some errors if we take pt near pt ⑨. but we get approx. nearest neighbour.]

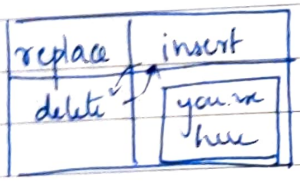
Instead of 1 tree → construct several trees (Construct Forest)
So, result is based on several constructed trees.
& then return the nearest neighbour.

(So, if many tree are constructed → more data structure storage).
but still worth it.

⑥ String Algorithms (Edit distance):-

Levenshtein Distance [edit distance]

Our key



	""	b	e	n	y	a	m
""	0	1	2	3	4	5	6
e	1	1	1	2	3	4	5
p	2	2	2	2	3	4	5
h	3	3	3	3	3	4	5
r	4	4	4	4	4	4	5
e	5	5	5	5	5	5	5
m	6	6	5	5	6	6	5

be → to form → take 2 deletions

① → b, e are diff.

① = min (insert, del, r
2, 2, 1
(insert e after b
delete

(.5 open)

ex:

from "" → need to form subproblem eph

⇒ 3 insertions.

Time complexity : $O(a\,b)$ (a = len(a))
Space " : $O(a\,b)$.

if both e, e are same
its same as $(i-1, j-1)$ subproblem ans

[if diff. letters] $\min[(i-1,j),(i,j-1),(i-1,j-1)]+1$

⑧ Burrows Wheeler Transform (BWT):-

BANANA $ (n)

Improve file compression

pre process our database.

m queries:
AN
NANA
BAN
NA
K size

[So, given a query you can find where the word occurs]

all rotations of sequence:

BANANA$
$BANANA
A$BANAN
NA$BANA
ANA$BAN
NANA$BA
ANANA$B

Sort All rotations alphabatecially

'$' is smallest character assuming

$BANANA
A$BANAN
ANA$BAN
ANANA$B
BANANA$
NA$BANA
NANA$BA

BWT transform is last col. of matrix

ANNB$AA

u can reconstruct the actual content of matrix

sorted

$ B1 A3 N2 A2 N1	A1 ✓
A1	N1
A2	N2
A3	B1
B1	$
N1	A2
N2	A3

follow followed by B1

pattern matching using the BWT :-

	i	First	last	L2F(i)	
top →	0	$1	A1	1	
	1	A1	N1	5 ✓	→ top
	2	A2	N2	6 ✓	→ bottom.
	3	A3	B1	4	
	4	B1	$	0	
	5	N1	A2	2	→ top
bott →	6	N2	A3	3	→ bottom

with top bottom range

Search → NAN/AN

top, bottom would be 6 ⇒ 3

for AN (start here (back words))

A

→ top.
→ bottom.

2,3

try A2, A3

1 → top.
3 → bottom

where AN occured

N

N1 5 → top
N2 6 → bottom
position

For NA

* Suffix Tree / Arrays :-

Suffix → is a substring at the end of a string of characters.

ex: for horses →
e
se
rse
ors
horse.

So, Suffix array is an array which contains all the "sorted suffixes" of a string.

0	camel
1	amel
2	mel
3	el
4	l.

1	amel
0	camel
3	el
4	l
2	mel.

sorted suffixes

→ So, actual 'Suffix array' is array of sorted indices.

→ This provides a compressed representation of sorted suffixes without actually needing to store the suffixes.

→ Suffix array provides space efficient alternative to 'suffix tree' which itself is compressed version of trie.

Note: Suffix array can do everything suffix array can do., with some additional information such as (LCP) array

— longest common prefix. —