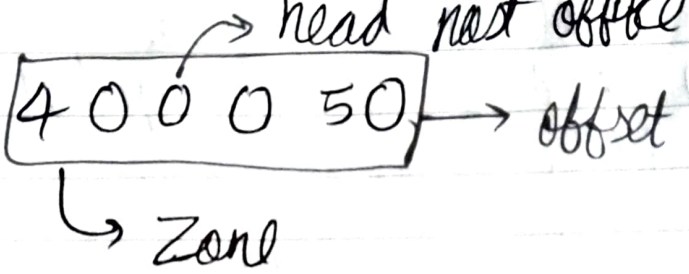
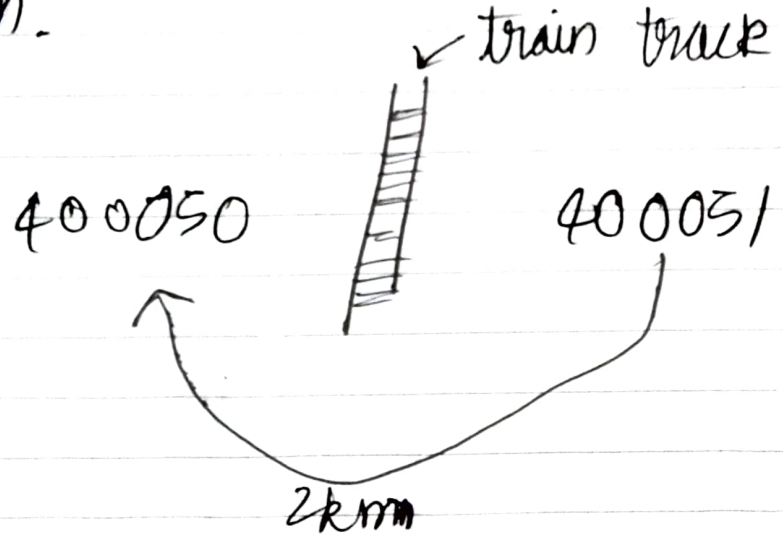
Google Maps Algorithm

Earlier, Pincode/Zipcode sys was used → assign each area code based on that ~~area~~ post office in that area.

eg:-



eg:- customer in 400051, shop in 400050
order comes along another path & not disp. path.



∴ Our sys should satisfy the foll. criteria:

GOAL 1) Measurable dist b/w any 2 pts. ~~for this~~ For this we need
   (i) uniformity in the way nos. are assigned to locs
   (ii) Granularity in our scalable sys. i.e. we shud be able to break areas further into smaller & smaller areas.

GOAL 2) Proximity. eg:- find nearby locations from a user's location

For meeting goals

M-1 ∴ Euclidean dist = $\sqrt{\Sigma(r_1 - r_2)^2}$

no rule for assignment

1-D dist used for proximity

Problem → you gotta check and store dist of all pts in your region

→ as size of region grows, this algo has a really bad TC (O(N)) and so doesn't work very effecently

M-2 :

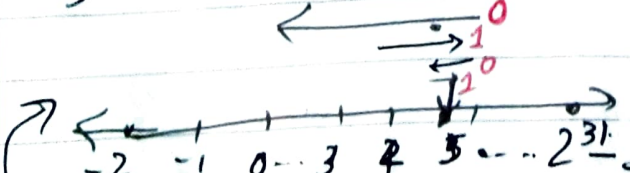
for achievin goal 1 → Find a smart way to assign nos to points in region

for achievin goal 2 → use a data struc., on which, if you do a range query, you get proximity of 2 pts

→ also, this data struc. shud have "scalable granularity" i.e., it should recursively apply its logic when we break down region being looked at into further sub-regions.

32 bits

5 → 0 0 0 0 0 .... 1 0 1



for achievin goal 1 → how pts are represented

say we have a no. = 5.685

Date: M T W T F S S

→ eg:-

for 4th decimal place, we add 1 more bit/ grp of bits i.e. we allocate for more in computer

in comptr we hv 2 data types → 32 bit datatype, 64 bit datatype

↓

each data type represents a range of nos. not the exact no.

i.e $\frac{10}{3}$ is stored on 32-bit as $3.3333333 < \frac{10}{3} < 3.3333333334$

∴ you hv a small "error range" where your actual no. lies in

∴ You can think of this as a way to find a small range of nos.

On no. line when we do a binary search, doin the bits of the no, the more the decimals i.e. more powerful datatype (64 instead of 32) the more accurate your no. is to the actual no. i.e. error range starts gettin smaller & smaller

∴ We can use this sys. to represent pts on a map as well.

64 bits → (32 bits for X-coord, a = error range in number stored here; 32 bits for Y, b = error range in no. stored here)

Error area = ab = 

⇒ the square in which your loca" needs to lie

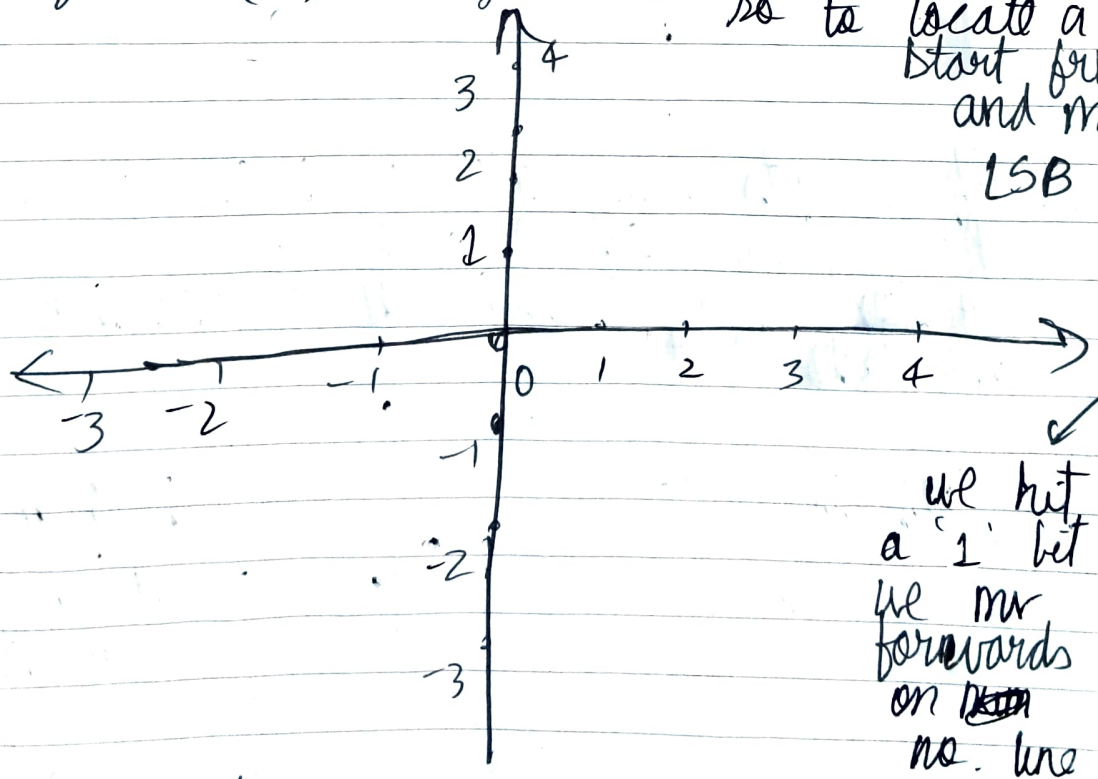
How to search for nos. in a 2-D plane

Now we hv. smartly assigned x, y coords to each pt s.t. if 2 pts are close their leading bits will match

eg:- if 2 pts (x-coords.) are represented as 110000 and 110001, wrt they are quite close by in x-dir

How pt looks like on a 2D plane

eg: → (10,3) $\xrightarrow{\text{same logic}}$ X, Y axes are both no. lines so to locate a pt we start from MSB and move towards LSB s.t.



we hit a '1' bit we mv forwards on no. line →

we hit a '0' bit, we mv backwards on no. line ←

on combining movement in X, Y dir" by reading X, Y coords' bits simult

we get 1 of 4 movements

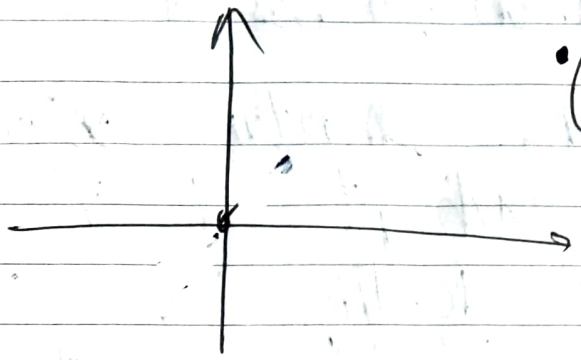
32 bit no → 1st bit → sign → 0 for +ve, 1 for -ve
→ 8 bits → exponent
→ 23 bits → frac"

Date:

eg:- MSB 32 bits → 32 decisions of ↑ ↓ ↑ ↓

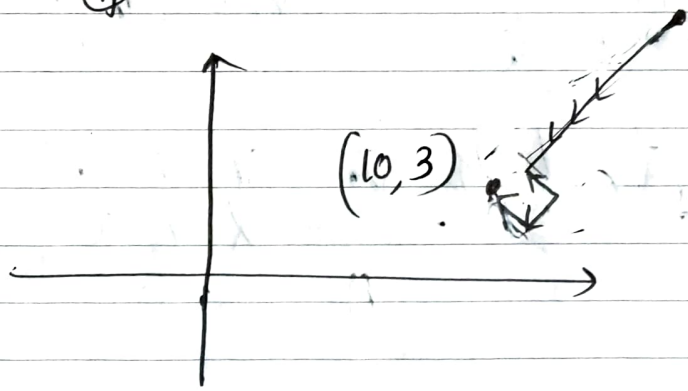
10 → 0 0000 0 0001010
3 → 0 0000 0 000011

moment



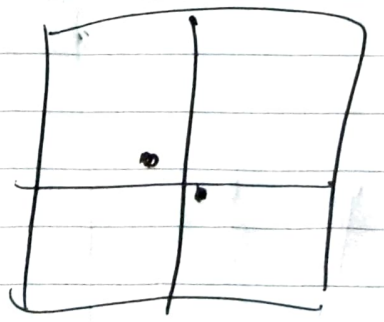
In each such moment you ↓ by lotta nos. and in the next step, you ↓ by exactly 1/2 the value/nos.

∥ after some time



∴ If 2 pts have 1st 13 MSBs as same → then those nos. will be close by in this 2D version → preceding nos determine the quad we lookin at

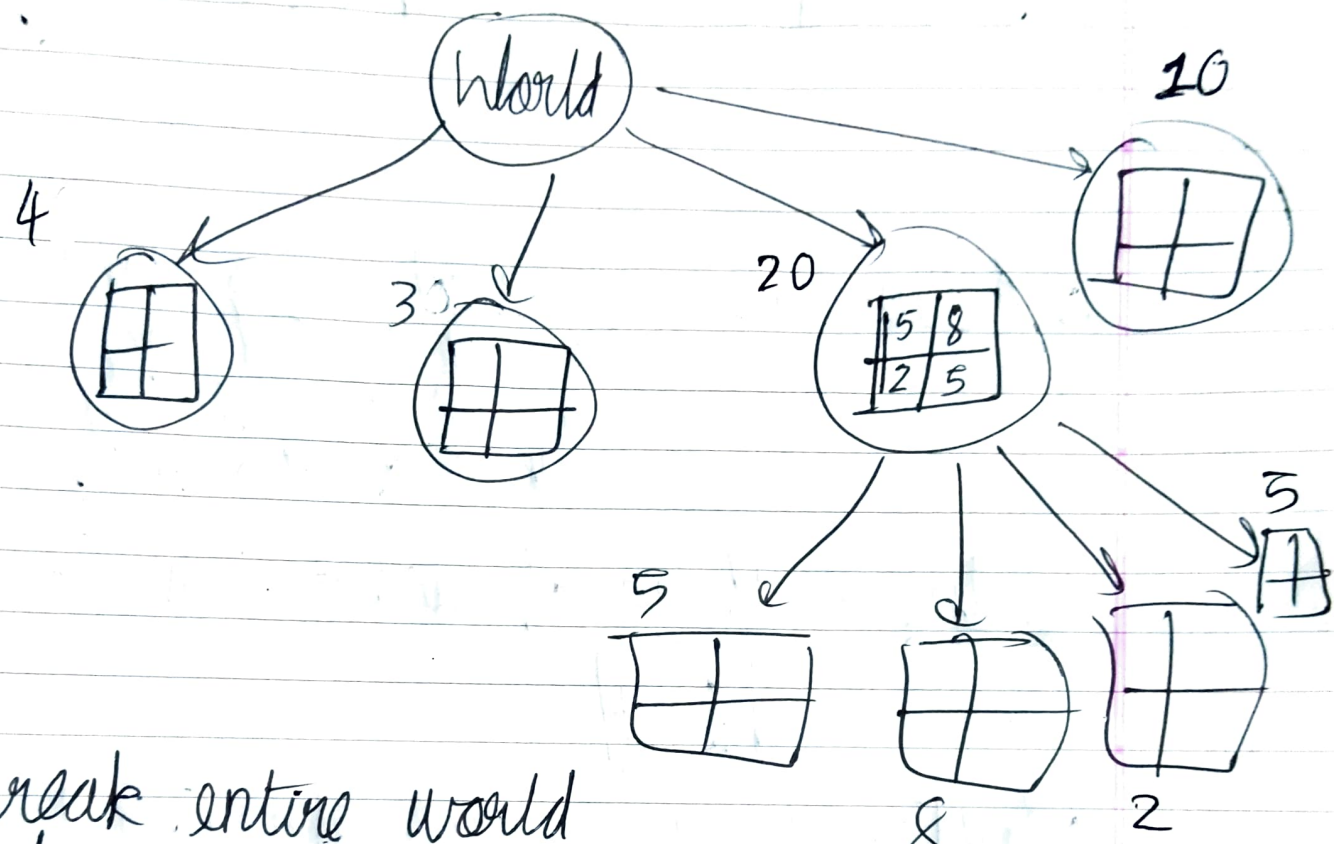
Caveat →



proximity is high but there's a mismatch

Date:

Data Struc. 1 → Quad tree



Break entire world into region → go as deep as appn requires

eg:- Keep breaking down till there's a 1 km radius "in" each loca" ... places nearby

eg 2:- can split even based on no. of loca's in a square. i.e. split if no. of loca's in a sq. is > 17

Problem → tree can be really skewed.

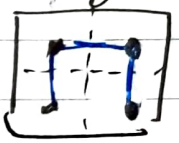
→ 2D plane → we don't hv algos that are really efficient at finding which pts. lie in a range (range based query)

Range query can be done really well in 1-D plane (eg:- interval, segment trees)

$\log N$ T.C. for search

∴ Our aim → Convert 2-D planes' pts to be represented on a 1-D line so we can easily perform "range queries" on pts.

Soln → Usage of Hilbert curve (a fractal)

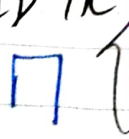


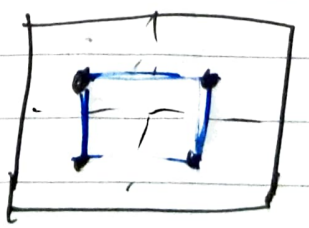
could've also used curves like  , 

but in Hilbert curve, len. of curve is min. so ~~change~~ chance of err. ~~is also~~ durin proximity calc is also min (more proximity info lost)

∴ We can easily take all pts in 2-D plane and represent em all on a 1-D line ~~and the~~ (our Hilbert curve) and

1D-HC { for a single grid





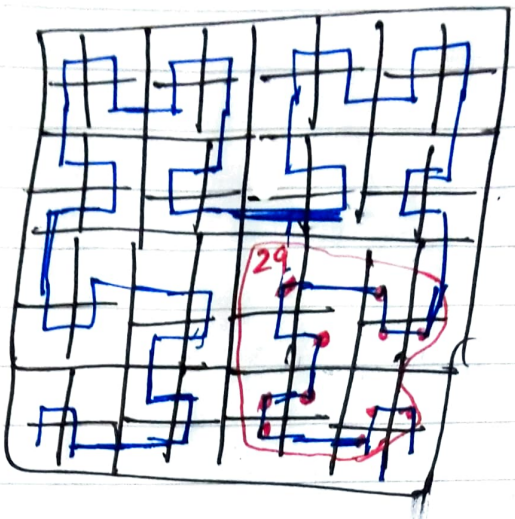
1 Hilbert curve (HC) used

2D-HC { for a slightly complex grid (each time you go a fractal dimension deeper add & invert HCs of $(n-1)$ dims.

4 HCs used
→ 2 regular
2 inverted

∴ Now we go for an infinitely recursive process (of representing 2D plane pts on 1D line) that can go to any arbitrary depth

Eg:- 3



∴ Recursive formula → ① say you go n^{th} level deep

② Use 4 curves of $(n-1)^{th}$ level → 2 regular, 2 inverted

③ Connect these 4 curves

Caveat discussed earlier is also minimized as dim of HC ↑

3 really cool facts abt this kinda soln.

① It is a 2-D space filling recursive curve

② It is scalably granular. (i.e. pos^n of a pt. on a curve doesn't change much when ~~grids that the curve~~ curve dim ↑ i.e. grids are more finely divided

③ Once you set a pt., you can do ± threshold to get all pts in close proximity