Common

PERCENTAGES & RATES

percent change — (next-prev)/prev*100 — denominator is the START value: 80->100=+25%, 100->80=-20% (not symmetric)

percentage points vs percent — 40%->50% = +10 points, +25% relative (10/40) — subtract for points, divide by start for percent

relative change, small base — 1->3 = +200% — tiny denominator, treat as noise not signal

3x vs +300% — 3x = +200%; +300% = 4x

percentages compound, not add — -10% then +10% = 0.99 (net loss), not back to original (1.1*0.9=0.99). General form, multiply the factors: final = start*(1+p1)*(1+p2)*...

reversing a percentage — write the forward equation first: price 80 after 20% off -> old*0.8=80 -> old=100

gross vs net (VAT) — VAT is charged on the net: net*1.2=gross. Recover net: gross/1.2 (never gross*0.8). 1.2 is the factor/multiplier, 20% is the rate

markup vs margin — cost 80, price 100: markup=25% (over cost, 20/80), margin=20% (of price, 20/100). Which to use: setting a price from cost -> markup; reporting profitability -> margin (what investors/accountants use). The trap: a 30% markup is NOT 30% margin — cost 100, price 130, margin is 30/130=23%

chained rates multiply — a pipeline where each stage keeps a fraction of what entered: 1000 people, stage1 keeps 50%->500, stage2 keeps 40% of those->200. Overall 20% (multiply fractions 0.5*0.4)

new value from percent change — decrease: price*(1-percent/100); increase: price*(1+percent/100)

compound growth — (1+r)^n — 5%/month x12 = 1.05^12 = 1.796 (80% rise, not 60%)

rule of 72 — doubling time ~ 72/growthPercent

COMBINATORICS — DEFINITIONS

combinatorics — the study of the number of different ways of combining objects

power set — the set of all possible subsets of a given set, including both the empty set and the set itself. No duplicates (not a multiset)

multiset — a set that allows duplicates: each element has a multiplicity, and order doesn't matter

arrangement — the specific linear or positional placement of elements where order matters

selection — picking elements where only membership, not position, matters

cardinality — the number of elements contained in a set

factorial (n!) — n!=n*(n-1)*...*1; 0!=1; 5!=120, 10!=3,628,800. A factorial function overtakes an exponential. No closed form — must loop/recurse: let result=1; for(i=2;i<=n;i++) result*=i (or n<=1?1:n*factorial(n-1))

COMBINATORICS — COUNTING TECHNIQUES

Rule of Sums: If task A can be done n ways and task B m ways, and they can't both happen, there are n + m ways to do either. Formula: n + m. Use cases: Picking one dessert: 4 cakes or 3 pies -> 7

Rule of Products: If task A can be done n ways and task B m ways, and both must be done, there are n * m ways. Formula: n * m. Use cases: Outfits: 3 shorts x 4 tops -> 12, Menu: 4 mains x 3 drinks -> 12

Permutation: An ordered arrangement of all elements from a set. AKA: full permutation. Formula: n! Use cases: seating people at a table, shuffling a deck of cards, delivery route planning, ordering a playlist. Seating at a round table is a circular permutation, (n-1)!. Straight row is n!.

Variation without repetitions: An ordered arrangement of a subset of k elements from a set of n elements. AKA: k-permutation, partial permutation. Formula: n! / (n-k)!. Use cases: podium — gold/silver/bronze from 10 runners; 302 neurons, directed connections: k-permutations P(n, k), so P(302, 2) = pick 2 from 302, in order = 302!/300! = 302x301.

Variations with repetition: An ordered arrangement of a subset of k elements from a set of n elements with duplications. Formula: n^k. Use cases: fixed-length codes, keyspaces, password combinations, PIN numbers, dice rolls (6^k), binary strings of length k (2^k).

Combination without repetitions (subset): An unordered selection of k elements from a set of n elements. Formula: n! / (k!(n-k)!). Use cases: choosing a team/committee (e.g. 3 of 10 people group), card hands (e.g. 5 of 52 cards)

Combination with repetition (multiset): is a unordered selection of size k from a set of n elements with duplications. AKA: multiset coefficient, stars and bars. Formula: (n+k-1)! / (k!(n-1)!). Use cases: choosing 3 scoops of ice cream from 5 flavors. Stars and bars

Power sets: power set cardinality / total subsets. Formula: (2^n): Use cases: bitmasking, include/exclude decisions

ESTIMATION & ORDERS OF MAGNITUDE

multiply by hand — lead x 10^exp, multiply leads, add exponents: 2e5*3e3=6e8; carry when lead>=10: 4e5*5e3=20e8=2e9

dividing by hand — scale both sides to kill the decimal (4.5/1.5 -> 45/15=3), strip trailing zeros (2400/12 -> 24/12=2, restore to 200), or factor the divisor and divide twice (2400/4=600, 600/3=200)

log2 = how many halvings — binary search over 1000 items ~ ceil(log2(1000))=10 probes. Base 10 version: every multiply by 10 increases log by 1 — log10 of a number is roughly its digit count minus one

node count of a perfect binary tree by height — 2^(h+1)-1 nodes total, 2^h leaves at height h; run backwards: n nodes -> height ~ log2(n)

1+2+...+n — n*(n+1)/2 (n=100 -> 5050); unordered pairs in n items = n*(n-1)/2

fencepost counting — inclusive range day3-day7 = 7-3+1=5 days; half-open [3,7) = 4

growth reflexes — n doubles: linear=2x, nlogn~just over 2x, quadratic=4x, exponential=squared

STATISTICS

mean — the average: sum divided by count. {1,2,3,4,100} has mean 22 — one outlier drags it far from where most data sits, why percentiles exist

average of averages — wrong unless weighted: total/total, never the mean of per-group means. Example: server A 900 reqs @10ms, B 100 reqs @100ms. Wrong: (10+100)/2=55ms. Right: (900*10+100*100)/(900+100)=19ms

median — the middle value of sorted data, or the average of the two middle values. {1,2,3,4,100} has median 3 — unlike the mean, one extreme value barely moves it

mode — the most frequently occurring value. Rarely useful for continuous data (latencies, prices) since ties are common; more useful for categorical data (most common error code, most common browser)

percentile — the value below which a given percentage of the data falls, p50 is the median, p95 means 95% of values are at or below it. Array formula and worked example: see average vs p95 below

variance — the average of the squared differences from the mean — squaring makes it always positive and penalizes big deviations more than small ones

standard deviation — the square root of variance, back in the same units as the data (variance is squared units, rarely reported directly). Ties to z-score below

expected value — the average outcome over many repeats: for each outcome, multiply probability x payoff, sum them. Coin flip: 50%x$10+50%x$0=$5. Engineering: 1% chance of 10hr outage per deploy = 0.01*10=6 min expected downtime per deploy

average vs p95 — latencies 1..100: mean=50.5, p95=95, p99=99 — mean hides the tail

prob. of at least one failure — 1-(1-p)^n — p=1%, n=100 -> 63.4%

probability, one-off — rolling a standard 6-sided die: the chance of rolling a duplicate "4" on your next turn is always 1/6 (approx 16.7%)

tail latency under fanout — per-server p99 becomes the median user experience at scale (1-0.99^100=63%)

collision intuition (birthday) — ~50% collision after ~1.18*sqrt(m) draws from m — 32-bit ids collide by ~77,000 draws

skewed / long-tail distributions — most engineering data (latencies, request sizes) is right-skewed, not a bell curve: a long tail of rare huge values drags the mean above where most data sits. Why percentiles matter more than the mean for latency

correlation vs causation — two things moving together doesn't prove one causes the other, a third factor could drive both. Dashboard trap: "errors rose right after the deploy" could be a coincident traffic spike, not the deploy — check a control group or a deploy-free period before concluding cause

regression to the mean — an extreme reading tends to be followed by a more typical one from randomness alone, even with no real change. Trap: restarting a server after a latency spike gets credited for a recovery that would've happened anyway

sample size / signal vs noise — a rate over a small sample swings wildly and means little (same shape as the relative-change-on-a-small-base row in Percentages & rates, applied to experiments). A/B test with 10 visitors per variant proves nothing; 10,000 per variant might

survivorship bias — analyzing only cases that survived a filter, missing the ones filtered out, skews the conclusion. Latency dashboards from completed-request logs miss timed-out requests (looks faster than reality); satisfaction surveys miss churned users (looks happier than reality)

z-score / anomaly threshold — z=(value-mean)/stddev, how many std devs from the mean. "3-sigma" alerting flags values >3 std devs out as anomalous (~0.3% of readings for a normal distribution). Breaks down for skewed metrics like latency — use percentile thresholds there instead

LOGIC

De Morgan's laws — not(A and B) = (not A) or (not B); not(A or B) = (not A) and (not B). Negating !(isValid && isReady) is !isValid || !isReady, not !isValid && !isReady

necessary vs sufficient — necessary: required but not enough alone (a ticket to board a flight). Sufficient: enough alone but not the only way (US citizenship for a US passport). Can be both, one, or neither

if-then's three relatives — "if A then B" (A->B): converse B->A (NOT equivalent), inverse notA->notB (NOT equivalent), contrapositive notB->notA (ALWAYS equivalent). Only the contrapositive is guaranteed to match the original's truth

negating quantifiers — "all X are Y" negates to "some X are not Y" (not "no X are Y"). "some X are Y" negates to "no X is Y". "all requests succeeded" negates to "at least one failed", not "all failed"

vacuous truth — "if A then B" is true whenever A is false, regardless of B. [].every(x=>x>1000)===true on an empty array; "all users in this empty segment are premium" is technically true, practically meaningless

System Design

POWERS OF 2 → POWERS OF 10 ($2^{10} \approx 10^3$ APPROXIMATION)

$2^{10} \approx 10^3$ — thousand, KB

$2^{20} \approx 10^6$ — million, MB

$2^{30} \approx 10^9$ — billion, GB

$2^{40} \approx 10^{12}$ — trillion, TB

$2^{50} \approx 10^{15}$ — quadrillion, PB

POWERS OF 10 (K/M/B/T/P)

$1 = 10^0$	$1\text{ M} = 10^6$	$100\text{ B} = 10^{11}$
$10 = 10^1$	$10\text{ M} = 10^7$	$1\text{ T} = 10^{12}$
$100 = 10^2$	$100\text{ M} = 10^8$	$10\text{ T} = 10^{13}$
$1\text{ K} = 10^3$	$1\text{ B} = 10^9$	$100\text{ T} = 10^{14}$
$10\text{ K} = 10^4$	$10\text{ B} = 10^{10}$	$1\text{ P} = 10^{15}$
$100\text{ K} = 10^5$		

POWERS OF 2

$2^0 = 1$	$2^4 = 16$	$2^8 = 256$
$2^1 = 2$	$2^5 = 32$	$2^9 = 512$
$2^2 = 4$	$2^6 = 64$	$2^{10} = 1024$
$2^3 = 8$	$2^7 = 128$	

LITTLE'S LAW — L = Λ × W

things in the system = arrival rate × time each one stays

in flight = rate × handling time — 1.6k/s × 8 s = 12.8k

max rate = ceiling / handling time — 120k / 30 s = 4k/s

wait time = backlog / service rate — 200 / 0.08/s = 40 min

same one law, solved for each variable — works for anything with a queue in front: SQS in-flight, thread pools, connection pools, Lambda concurrency

KNOWN SCALE NUMBERS

big social net DAU ≈ 1 B

read:write (read-heavy) ≈ 100:1 → scale READS

32-bit int = 4 B values

64-bit int = 1.8×10^{19} values

URL shortener writes/day ≈ 10M-100M (bit.ly-ish; 1B/day = too high unless told "massive scale")

Twitter/X posts/day ≈ 500M (~5-6k QPS avg, ~10x peak)

generic big consumer app ≈ 100M-1B DAU (pick DAU, derive the rest)

Netflix streaming-hours/day ≈ 100M

Google searches/sec ≈ 100k

Wikipedia total storage ≈ 100 GB

Twitter/X daily active users ≈ 250M

KNOWN SERVER SIZES (AWS) — SO AN ESTIMATE CAN BE CHECKED AGAINST SOMETHING REAL

general-purpose instance ≈ 512 GiB RAM, 128 vCPUs (e.g. M6i.32xlarge)

memory-optimized instance ≈ 4 TB RAM (e.g. X1e.32xlarge), up to 24 TB RAM (e.g. U-24tb1.metal)

local SSD storage on one instance ≈ 60 TB (e.g. i3en.24xlarge)

local HDD storage on one instance ≈ 336 TB (e.g. D3en.12xlarge)

object storage (S3-style) ≈ no practical limit, petabyte-scale is normal

network speed inside one datacenter ≈ 25 Gbps on standard instances, 50-100+ Gbps on high-performance instances

STORAGE SIZING RECIPE

storage = item count × bytes per item × replication factor × retention

typical byte sizes: 1 char ≈ 1 B, URL row ≈ 500 B, tweet ≈ 300 B, UUID ≈ 16 B, timestamp ≈ 8 B, thumbnail ≈ 10 KB, photo ≈ 1 MB, minute of video ≈ 10-50 MB

example: 10M users × 2 KB each × 3 replicas = 60 GB — small enough for one node, and knowing that it's small is the answer. Classic omissions: the replication factor, and indexes (rule of thumb: double the total)

TYPE SIZES (BYTES)

char/ASCII = 1

boolean = 1

int = 4

bigint/long = 8

float = 4

double = 8

timestamp = 8

UUID = 16 raw / 36 string

decimal/numeric ≈ 2 B per ~4 digits + overhead

row overhead ≈ 20–40

LATENCY

L1 cache ref = 1 ns

RAM = 100 ns

SSD = 100 µs

disk = 10 ms

same-DC RTT = 0.5 ms

cross-country RTT = 80 ms

CA <-> Netherlands RTT = 150 ms

memory trick: RAM 100ns -> SSD 100µs -> disk 10ms (each ~100-1000x the last)

read 1 MB from memory ≈ 10 µs

read 1 MB from SSD ≈ 100 µs-1 ms
read 1 MB over 1 Gbps net ≈ 10 ms

SHORT-CODE / KEYSPACE SIZING

62^L via 62~2^6 chain — 62^5~2^30~900M, 62^6~2^36~57B, 62^7~2^42~3.5T, 62^8~2^48~200T
code length — pick smallest 62^L >= demand; demand = new/day x 365 x years
horizon barely moves length — each extra char is x62 capacity; extra years only multiply demand a little — x10 can't cross a x62 gap
headroom — keyspace >= ~10x demand so random codes rarely collide

QPS & SCALING DIRECTION

read QPS from R:W ratio — 100:1 means far more reads than writes -> scale READS

STORAGE & BANDWIDTH

bytes for a row — int=4, bigint/double=8, +20-40B/row overhead (Postgres)
chars vs bytes — Twitter's 280 is code points, not bytes on disk

CAPACITY & UNITS

availability multiplies — 5 services at 99.9% each = 0.999^5 = 99.5%
requests/sec from daily total — total/86400 (86400 = seconds/day, memorize)
MB vs MiB — 1024^2 vs 1e6 (4.9% apart); 1024^3 vs 1e9 (7.4% apart) — gap grows with each step up
bits vs bytes — divide by 8 — a "100 Mbps" link moves 12.5 MB/s
string bytes depend on runtime — UTF-8/disk: per-char (A + CJK = 1+3=4B). UTF-16(JS/Java): flat 2B/char. Python3: whole string widens to widest char
transfer time — size/bandwidth: 50GB over 100Mbps = size*8/bandwidth ~ 67 min
bandwidth needed — requests/sec x payload size (x8 to convert bytes to bits)
availability -> downtime budget — 99.9%=43.2min/mo, 99.99%=4.3min, 99.999%=26s — each extra nine divides by 10
time-unit ladder — 1ms=1000us=1e6ns

TypeScript

OPERATORS & PRECEDENCE

**** right-associative** — 2**3**2 = 512 (groups right); ** binds tighter than *, no parens needed for 100*2**n
% sign follows the dividend — -7 % 3 = -1, 7 % -3 = 1. Non-negative form: ((n%m)+m)%m
safe cyclic wraparound — ((i % n) + n) % n — previous index in a ring buffer
trunc vs floor vs |0 (negatives) — -7/2: trunc=-3, floor=-4, |0=-3. floor keeps %,/ consistent; |0 also clips to 32 bits
prefix vs suffix ++/-- — arr[i++] reads then increments; arr[++i] increments then reads

CLOSED FORMS INSTEAD OF LOOPS

exponential backoff — base*2**attempt; capped: Math.min(base*2**attempt,maxDelay); jitter goes OUTSIDE the cap
why baseattempt is wrong** — units flip between ms/s (100**3 vs 0.1**3) — base is a duration, 2 is the dimensionless growth factor
total time of N retries — geometric series: base*(2**N - 1)
page/chunk count — Math.ceil(total/size), or Math.floor((total+size-1)/size) — never floor(n/size)+1
digit count — Math.floor(Math.log10(n))+1 — breaks at 0 and negatives

DIVIDE & REMAINDER

1D index to 2D grid — row=Math.floor(i/cols), col=i%cols; back: row*cols+col
ms to h:m:s — floor(ms/3600000), floor(ms/60000)%60, floor(ms/1000)%60 — divide for unit, % strips larger units
bucketing a value — Math.floor(value/bucketSize)*bucketSize
distributing a remainder — floor(total/n) each, + (total%n) extras to the first (total%n) shares
hashing into buckets — hash % bucketCount — changing bucketCount moves nearly every key

ROUNDING, CLAMPING, COMPARING

Math.round on halves/negatives — round(2.5)=3, round(-2.5)=-2, round(-0.5)=0 (=== 0, distinct as -0) — halves go toward +infinity
rounding to 2dp for JSON — Math.round(n*100)/100 (toFixed returns a string). (1.005).toFixed(2)='1.00', (2.675).toFixed(2)='2.67'
clamp — Math.min(max, Math.max(min, v))
empty-array seeds — Math.min(...[])=Infinity, Math.max(...[])=-Infinity — seed a reduce with these. Math.max(...arr) throws past ~1e5 elements, use reduce

RANDOMNESS & SAMPLING

random int in [min,max] — min + Math.floor(Math.random()*(max-min+1))
Fisher-Yates shuffle — for(i=len-1;i>0;i--) {j=floor(random()*(i+1)); swap i,j} — sort(()=>random()-0.5) is measurably biased
crypto-safe random — Math.random is not cryptographic — crypto.randomUUID() or crypto.getRandomValues() for tokens/ids
jitter — full: random()*delay; equal: delay/2 + random()*delay/2 — avoids thundering herd
reservoir sampling — unknown-length stream: keep the nth item seen with probability 1/n

INTEGERS, BITS & LIMITS

there's only one number type — JS numbers are all 64-bit IEEE 754 doubles: 1 sign bit, 11 exponent bits, 52 fraction bits, exponent -1022 to +1023
MAX_SAFE_INTEGER — 2**53-1 = 9007199254740991, 16 digits ~9e15. 2**53+1===2**53 (true) — above it only even integers exist
the integer-skipping problem — the gap between representable integers doubles at every power-of-2 threshold: spacing 2 above 2**53, spacing 4 above 2**54, spacing 8 above 2**55. You lose whole integers, not gradual precision
Number.MAX_VALUE — 1.7976931348623157e+308, the largest representable double; past it math overflows silently to Infinity, not a throw. Big and rough (physics, ratios) -> MAX_VALUE is your ceiling; big and exact (ids, counters, cents) -> MAX_SAFE_INTEGER is your ceiling, past it use BigInt
the four extremes — smallest positive: Number.MIN_VALUE=5e-324 (not the same as EPSILON, the smallest gap vs the smallest value). Most negative: -Number.MAX_VALUE=-1.8e308
Infinity / -Infinity — a real value, typeof is 'number'; 1/0=Infinity, -1/0=-Infinity, no exception. Infinity+1===Infinity, but Infinity-Infinity=NaN
large ids over JSON — 18-19 digit ids (snowflake) must be sent as strings — JSON.parse already rounded, client-side fix is too late
bigint — 10n, typeof 'bigint', never mix with number. 2n**64n = 18446744073709551616n. Only fixes integers (10n/3n=3n, truncated, no decimals) — for exact fractions use decimal.js
Decimal (decimal.js) — a number stored in base 10 instead of base 2, arbitrary-precision via a library not a primitive. new Decimal(0.1).plus(0.2).equals(0.3) -> true. Slower, so for money/exact totals, not hot-path math

BITWISE OPERATORS

32-bit truncation — (2**31)|0 = -2147483648, -1>>>0 = 4294967295. Safe midpoint: Math.floor((lo+hi)/2), never (lo+hi)>>1
>> vs >>> — 13>>1=6 (floor div by 2). >> keeps sign (-8>>1=-4), >>> does not (-8>>>1=2147483644)
n & 1 odd test — 1=odd, 0=even — correct for negatives too (-3&1=1)
flags & masks — set f|=FLAG, clear f&=~FLAG, test (f&FLAG)!==0
a^a / a^0 — a^a=0, a^0=a — XOR cancels pairs, finds the one unpaired element
n & (n-1) — clears the lowest set bit; ===0 tests power of two (n>0)

GRID, INTERVALS & DISTANCE

4-neighbour offsets — [[-1,0],[1,0],[0,-1],[0,1]] added to [row,col]; up=[-1,0] since row 0 is the top
interval overlap (half-open) — aStart<bEnd && bStart<aEnd
Manhattan vs Euclidean distance — two ways to measure the distance between two points: Math.abs(dx)+Math.abs(dy) for 4-directional grid moves, Math.hypot(dx,dy) (straight line) otherwise
compare distances w/o sqrt — dx*dx+dy*dy orders identically for ranking, costs less
prefix sums — a is the array, pre[i] is the sum of a's first i elements (pre[0]=0): pre[i+1]=pre[i]+a[i]; sum from lo to hi = pre[hi+1]-pre[lo]
midpoint index — a,b are the two bounds (e.g. binary search's lo/hi): Math.floor((a+b)/2)

HEAPS, UNION-FIND & SEARCH-SPACE PATTERNS

binary heap array indexing — children of i: 2i+1, 2i+2; parent: floor((i-1)/2); build-heap starts at floor(n/2)-1
median of a sorted array — n odd: sorted[floor(n/2)]; n even: (sorted[n/2-1]+sorted[n/2])/2 — this even/odd split is why a running median needs two heaps
union-find complexity — path compression + union by rank -> amortized ~O(1) (inverse-Ackermann). naive chain is O(n)/op
binary exponentiation (fast pow) — x**n by squaring is O(log n): halve n, square base, fold in result when n is odd
binary search on the answer — monotonic yes/no predicate over a value range (not an array) -> binary search the range directly
sum without + or - — sum = a^b (no carry), carry = (a&b)<<1; repeat until carry===0

FLOATS & PRECISION

landmark values — 0.1+0.2 = 0.30000000000000004 (small end, decimal notation); 1e16+1===1e16 = true (large end, exponential notation) — both memorized, not derived
exponent notation — 3e-4 puts the 3 in the 4th decimal place
float tolerance compare — Math.abs(a-b) < 1e-9 (domain-derived), never === after arithmetic. Number.EPSILON ~ 2.2e-16 (2**-52), only the gap at magnitude 1, not universal
float + not associative — (0.1+0.2)+0.3 !== 0.1+(0.2+0.3); never assert exact equality between two recomputed totals, sum small-to-large
money as integer cents — convert in: Math.round(dollars*100) (floor loses a cent: 19.99*100=1998.9999999999998); display: (cents/100).toFixed(2). This is fixed-point (decimal spot implied by the scale factor), vs floating-point where each value stores its own exponent
boundary guard on Number(x) — Number.isFinite(n) is accept-test, Number.isNaN(n) is reject-test. Number('')===0, Number(' ')===0
NaN poisons silently — convert first, then validate with isFinite/isNaN — never regex the raw string first

Vocabulary

numerator — top number of a fraction, the part being divided
denominator — bottom number of a fraction, what you're dividing by (never 0)
dividend — the number being divided (in a / b, a is the dividend)
divisor — the number you're dividing by (in a / b, b is the divisor)
denominator vs divisor — same number, different context. Divisor: division, 10/2, the 2. Denominator: fractions, 3/4, the 4 (bottom)
quotient — the result of a division
remainder — what's left over after integer division (a % b)
factor — generally, one of the things multiplied in a product, not just whole numbers (1.5 is a factor of 6=1.5x4). In number theory specifically it narrows to a whole number that divides another exactly — the sense used elsewhere on this board
to factor — breaking something into the pieces that multiply back to it, a number (12=4x3) or a polynomial (x^2-1=(x-1)(x+1)). Whole-number version used in dividing by hand: 2400/12 becomes 2400/4=600, then 600/3=200
multiple — a number you get by multiplying another by an integer
coefficient — the constant multiplied onto a variable, the 2 in 2x
exponent — the small raised number showing repeated multiplication, the n in x^n
base — the number being raised to a power (x in x^n); also the radix in base-2/base-10/base-62
polynomial — an expression made of variables raised to whole-number powers, multiplied by coefficients, added together: 3x^2+2x-5. Named by its highest exponent (the degree): degree 1=linear, 2=quadratic, 3=cubic
reciprocal — 1 divided by a number, flips numerator and denominator
logarithm — inverse of exponentiation: what power do I raise the base to, to get this number. base 2 = binary log (log2/lg, halvings), base 10 = common log (log10, digit count), base e~2.71828 = natural log (ln, continuous growth)
monotonic — always increasing, or always decreasing, never both
magnitude — the size of a number ignoring sign; order of magnitude is the power of 10 it's closest to (100 and 400 are the same order, 100 and 4000 aren't)
fixed-point — a number stored as a plain integer at an implied, constant decimal position, e.g. money as integer cents (150 = $1.50)
floating-point — a number whose decimal position is stored per-value as an exponent, huge dynamic range but uneven precision (IEEE 754 double, what JS numbers are)
sampling — picking a subset of a population or stream to stand in for the whole, instead of processing everything; reservoir sampling picks one item fairly from a stream you can't buffer
hypotenuse /haɪˈpɒtənjuːs/ — the longest side of a right triangle, opposite the right angle, length sqrt(dx*dx+dy*dy). Math.hypot(dx,dy) computes it directly, avoiding overflow from squaring large values yourself
Fibonacci number — each number is the sum of the two before it: 0,1,1,2,3,5,8,13,21,...; F(n)=F(n-1)+F(n-2). Canonical example for recursion vs memoization: naive is exponential, caching makes it linear
pi (π) — ratio of a circle's circumference to its diameter, ~3.14159, irrational. Shows up on this board as a memorization trick, not geometry: seconds/year ~ pi x 10^7
mantissa — the fraction/significand part of a float, the digits scaled by the exponent; in a 64-bit double this is the 52 fraction bits
single-precision — 32-bit float format: 1 sign, 8 exponent, 23 mantissa bits; C/Java's float
double-precision — 64-bit float format: 1 sign, 11 exponent, 52 mantissa bits; C/Java's double, and the only numeric type JS has. Float is used colloquially as the umbrella term for any float format, which is why JS still says float even though every JS number is a double
multiplier — the factor you multiply by to apply a percent change in one step: 1+p, p is the percent as a decimal. +10% -> multiplier 1.1. -10% -> multiplier 0.9
rate — a ratio comparing two quantities, usually across different units or over time (speed, interest, error rate). Meaningless without its denominator stated — "500 errors" is not a rate, "500 errors per 2M requests" is. Two forms: 20% is the percentage form, 0.2 is the decimal form (what goes into a formula)
arithmetic progression sum — n terms from a, step d: n/2*(2a+(n-1)d), or n*(first+last)/2; 1+2+...+n is the a=1,d=1 case. This is LINEAR growth: each term adds a fixed amount, nth term = a+(n-1)*d
geometric progression sum — n terms from a, ratio r: a*(r^n-1)/(r-1); backoff total (a=base,r=2) and tree node count (a=1,r=2) are both this. This is EXPONENTIAL growth: each term multiplies by r, nth term = a*r^(n-1)