A: Sum of palindromes

Let A be the input number, and let it have n digits (which means that $A \leq 10^n$). Suppose that $n \geq 3$. We can express A as $A_H A_L$ – concatenation of two numbers, where $len(A_H) = \lceil n/2 \rceil$ and $len(A_L) = \lfloor n/2 \rfloor$. We substract 1 from A_H, obtaining $A'_H = A_H - 1$ and then append $len(A_L)$ digits to A'_H such that the result is a palindrome B. Now $A - B \leq 10^{n/2+1}$, so we keep B as one of the desired palindromes, reducing the length of A by a factor of 2. In $\log_2 n + 1 \sim 18$ iterations we reach a 2 digit number, which we deal with as a special case.

B: Bookface

We are given a sequence of non-negative integers x_i and a number d. In one move we can change any x_i by ± 1. We want to change the sequence so that $|x_i - x_j| \geq d$ holds for any $i \neq j$ using the smallest number of moves.

First, let's sort the sequence x_i; from now on we assume $x_1 \leq x_2 \leq \cdots \leq x_n$. We can imagine that the x_i values are points on a line, which we want to move so that no two are closer than d. Note that for any i, the i-th point is to the left of the $(i+1)$-th point, and it is suboptimal for them to "swap", so we can assume the relative order of points will not change, and only enforce the condition that $x_{i+1} - x_i \geq d$ must hold for any i.

Since x_i's must remain non-negative, we cannot shift our points below 0. It is convenient to remove this condition - for example, by adding the values $\{-d, -2d, -3d, \cdots, -(n+1)d\}$ to the input sequence x_i. Then, it will never be optimal to move any of the original values to negative, and we can forget the $x_i \geq 0$ condition from now on.

Now let's define $x'_i = x_i - i \cdot d$; note that x'_i may no longer be a non-decreasing sequence. Moving x_i by ± 1 corresponds to moving x'_i by ± 1. Moreover, $x_{i+1} - x_i \geq d$ translates to $x'_{i+1} \geq x'_i$. While the reduction from this paragraph is not strictly necessary, it allows us to forget the value of d from now on.

To solve the final problem, consider the dynamic programming $f[i][t]$, defined as the minimum total cost to make our sequence non-decreasing if we consider only the first i elements, and the value of the last element has to be t. If we define $g[i][t] = \min_{t' \leq t} f[i][t']$, then $f[i][t] = |t - x_i| + g[i-1][t]$.

Of course we cannot maintain $f[i][t]$ explicitly, as the range for possible values of t would be prohibitively large. However, we can prove by induction that any $f[i]$ is a *bitonic piecewise linear function* with respect to t.

Transforming $f[i]$ to $g[i]$ corresponds to removing everything after the minimum value of $f[i]$ and replacing that with a constant segment. The last linear piece in the function $g[i]$ will have slope 0, so the last piece in the function $f[i]$ will have slope 1.

In order to obtain the optimum solution we don't need to store the values of $f[i]$, only the shape. Then, having restored the optimal sequence x'_i we can compute the cost. We see we only need to store the "breakpoints" of f; i.e. the points where the slope increases by 1. These points can be stored in a multiset.

Finally, the code turns out to be strikingly simple! We process element of the sequence x'_i from left to right. We maintain the multiset of breakpoints S, which corresponds to the current function g. To process an element x, we need to insert x into S twice (since in the point x the slope will change by 2). Now, S represents the new f. To transform it into g, we need to remove the rightmost breakpoint.

The rightmost breakpoint in g at a given moment is the optimum (smallest value of the cost) for a given prefix. If we save these breakpoints in a sequence p_i, then it's easy to use p_i to compute the optimal solution (hint: go from right to left through p_i and greedily fix it so that it's non-decreasing).

C: Space Gophers

Let's consider two tunnels as connected if one can travel directly from one to the other (in other words, if there exists a microcube in one tunnel and another microcube in the other, such that these microcubes touch or coincide). This gives a graph G with tunnels as vertices. In order to see if we can travel between two microcubes c_1 and c_2, we can take any tunnel t_1 that contains c_1, and t_2 that contains c_2, and ask if tunnels t_1 and t_2 are connected in G. Therefore, to solve the problem it is enough to compute the connected components of G.

Unfortunately, there can be $\mathcal{O}(n^2)$ edges in G, so we cannot store G directly. However, we can maintain a disjoint-set-union structure on tunnels, and join only some pairs of tunnels that will span the same set of connected components as the full set of edges in G.

Let's consider which pairs of tunnels are connected in G. There are two cases: either the connected tunnels are parallel or perpendicular. The first case is easy, since for any tunnel there are just four possible candidate tunnels. Let's now focus on the second case.

Assume that we want to consider connected pairs of tunnels a and b, where a is parallel to the x-axis, and b to the y-axis. If we have a function to perform this, we can then call it for the two other pairs of directions.

Denote the z coordinate of a and b as z_a and z_b respectively. Notice that, since a and b are perpendicular, they will be connected if and only if $|z_a - z_b| \leq 1$, the other coordinates of the tunnels don't matter.

Let's now group all tunnels parallel to the x-axis by their z coordinate into groups $X[z]$. Similarly, group all tunnels parallel to the y-axis into groups $Y[z]$.

Consider a group $X[z]$. We would like to join all tunnels in that group with all tunnels in groups $Y[z-1]$, $Y[z]$ and $Y[z+1]$. If these three groups are empty, then there is nothing to be done. Otherwise, we can join all these tunnels into one component. After that, we can reduce groups $X[z]$, $Y[z-1]$, $Y[z]$ and $Y[z+1]$ into just one tunnel each, since all tunnels in each of these groups are in the same component. By doing so we will join two tunnels only $\mathcal{O}(n)$ times.

The complexity is $\mathcal{O}(n \log n)$, since we need to group the tunnels by coordinates.

D: We apologize for any inconvenience

We create a graph with a vertex for each stop and a vertex for each line, and an edge iff a given line serves a given stop. It is easy to see that travelling between two stops requires c changes iff their respective vertices are at distance $2(c+1)$.

We renumber the vertices: first go the "stop" vertices. Then go the "line" vertices for lines which never get suspended. Last go the "line" vertices for the suspended lines, in the reversed order of their disappearing. We run Floyd-Warshall on this graph.

By the basic properties of Floyd-Warshall, the value $FloydWarshall[i][j]$ denotes, after the t-th iteration of the algorithm, the length of the shortest path from i to j which uses only the set $\{1, \ldots, t\}$ as its possible intermediate vertices. Which is precisely what we want, because (in the last s iterations) vertices $\{t+1, \ldots, n+k\}$ are the vertices related to lines which are suspended in this iteration. To obtain the result for the iteration, we simply take a maximum – except for *infinity* value – over the distances between "stop" vertices (note there exist instances where this is not the same as just taking the graph's diameter, i.e. there might be a "line"–"line" pair at a distance strictly larger than the maximum over the "stop"–"stop" pairs).

Complexity: $O((n+k)^3)$.

Remark One might easily observe that the first n iterations of Floyd-Warshall (those when "stop" vertices are added) actually only compute paths of length 2. So, there's really no need to perform them as it is possible to compute the same result with just one pass over the graph (plus a logarithmic factor for sorting each line's stops), which changes the complexity from $O((n+k)^3)$ to $O((n+k)^2 \cdot k)$. This, however, was not necessary to fit within the time limit.

E: Vladiksoft

Let's denote the probability that Intern Vladik generates a path s by $p[s]$. In order to separate a half of the paths that are most likely the product of Intern Vladik's code, it is natural to sort all paths by the probability p, and assign the half with the highest probability to the intern. In fact, it's not hard to prove that this is the *optimal* solution - i.e. it gets the best possible fraction of correct answers (in expectation) for any board.

How to compute $p[s]$? It's somewhat annoying that in some cases the Intern Vladik's algorithm restarts itself, so let's ignore that possibility for a moment, and assume that if the algorithm gets stuck it doesn't return any path. Denote the probability of generating s by this altered algorithm by $p'[s]$. Now, it's easy to see that $p'[s] = \left(\frac{1}{2}\right)^{k[s]}$, where $k[s]$ is the number of places where we can "step off" the path s, i.e. the number of positions on the path where Intern Vladik's code samples the direction. The value $k[s]$ can be simply computed by iterating through s.

How does $p'[s]$ relate to $p[s]$? Let's denote the probability that Intern Vladik's algorithm will get stuck if starting from the upper-left corner as q. The sum of $p'[s]$ will be exactly $1 - q$, while with probability q the algorithm will restart (by the way, restarting the sampling algorithm if it fails to obtain a sample is called *rejection sampling*, and this approach has various interesting applications). The probability mass of q will then be distributed in the same way as if the algorithm was just sampling for the first time. This means that $p[s] = \alpha \cdot p'[s]$, where α is a constant that depends only on q. In other words, to obtain $p[s]$ from $p'[s]$ we just need to rescale these values to sum up to 1.

In the end, it's possible to solve the problem by computing $p[s]$. However, since α is a constant, sort by $p[s]$ is equivalent to sorting by $p'[s]$, which in turn is equivalent to sorting (decreasingly) by k. Therefore, we don't even need to think in terms of probabilities and use floating point values, it's enough to solve by k.

Remark For $m = 100\ 000$ paths, the accuracy obtained by a given algorithm for a fixed board is very stable (usually varies by at most $\pm 0.5\%$). For some boards in the test cases the optimal accuracy is around 76%. Therefore, a suboptimal algorithm is likely to get Wrong Answer.

F: Wizards Unite

Let $t := \sum_{i=1}^{n} t_i$. Let's say that we use silver keys at moment 0 to open first k chests with opening times $t_1, \dots, t_k$. Then the total time to open all the chests equals to $\max\{t_1, t_2, \dots, t_k, t-(t_1+t_2 \dots + t_k)\}$. On the other hand, it's obvious that the minimum possible total time to open all the chests cannot be less than $\max\{t_1, \dots, t_n\}$. So it's easy to see that to minimize the total time we should use silver keys to open k chests with the highest opening times and use the gold key to open other chests.
Complexity: $\mathcal{O}(n \log n)$.

G: Binomial

Theorem 0.1 (Lucas) *For non-negative integers m and n and a prime p, the following congruence relation holds:*

$$\binom{n}{k} \equiv \prod_{i=0}^{m} \binom{n_i}{k_i}$$

where:

$$n = n_0 \cdot p^0 + n_1 \cdot p^1 + \dots n_m \cdot p^m$$

and

$$k = k_0 \cdot p^0 + k_1 \cdot p^1 + \dots k_m \cdot p^m$$

are the base p expansions of m and n respectively.

Using this theorem it's easy to see that $\binom{n}{k}$ is odd iff n is a supermask of k.
Let's assume that $\max\{a_1, \dots, a_n\} < 2^w$ for some w. For each a_i we want to find the number of its submasks in the sequence. It's easy to do in $\mathcal{O}(3^w)$, but it's still too slow. The model solution uses the following idea: let's compute $f(i, mask)$ - the number of such elements in the sequence which are submasks of $mask$ on i first bits and are exactly $mask$ on the rest bits. Then this function can be computed in $\mathcal{O}(2^w \cdot w)$.
Complexity: $\mathcal{O}(MAX \log MAX)$.

Remark It is also possible to solve this problem without the use of Lucas theorem.
It's easy to see that for a non-negative integer n and a prime p, the maximum power of p that divides $n!$ equals

$$\sum_i \left\lfloor \frac{n}{p^i} \right\rfloor$$

Therefore, the maximum power of 2 that divides $\binom{n}{k}$ equals

$$\sum_i \left(\left\lfloor \frac{n}{2^i} \right\rfloor - \left\lfloor \frac{k}{2^i} \right\rfloor - \left\lfloor \frac{n-k}{2^i} \right\rfloor \right)$$

Every term in the above sum is non-negative, so the entire sum is 0 if and only if every term is 0. It's easy to see that this means k and $n-k$ have no bits in common in their binary representations; which is the same as n being a supermask of k.