A. **Alphabet**

The easiest way to think about this problem is, what is the longest subsequence of the input array that we can use to form an alphabet? Then we can calculate how many missing letters we need to add.

Such a subsequence must be strictly increasing, so this is just a slight restatement of the longest increasing subsequence problem.

This can be solved in quadratic time using linear programming; we calculate in order the longest increasing subsequence for every prefix of the input string:

```
for (int i=0; i<n; i++) {
   int best = 1 ;
   for (int j=0; j<i; j++)
      if (s[i] > s[j] && 1 + bestv[j] > best)
         best = 1 + bestv[j] ;
   bestv[i] = best ; // best subsequence of length i
}
```

It is not important for the small input given, but the problem can also be solved in O(n log n) time by keeping track only of the lowest ending value for subsequences of a given length, locating the correct subsequence to extend by binary search:

```
for (char c : s) {
   auto it = lower_bound(ec.begin(), ec.end(), c) ;
   if (it == ec.end())
      ec.push_back(c) ;
   else
      *it = c ;
}
```

Because of the limited range of the input (only 26 different characters) this is actually a linear-time solution in this particular case.

B. **Buggy Robot**

First, to simplify implementation, note that we only need to consider additions (deletions can be transformed to additions).
Consider a graph where nodes are (state, number of commands followed). There are 50^3 nodes in this graph, and 50^3 * 5 edges (1 edge for following the next valid command, 4 edges for inserting an arbitrary command before).

Now, note the shortest path from the (robot square, 0) to some node (exit square, t) for any t is the solution.

The edge weights are 0 or 1, so this can be solved with BFS, though Dijkstras will also pass.

C. **Cameras.**

This problem requires a greedy approach; scan through the consecutive sets of houses of width R, count the number of cameras already there, and if we don't have enough, add as many as needed as far right (towards the next sets of houses) as possible. It is easy to see that adding cameras as far right as possible maximizes their utility for subsequent sets.

To keep the running count, we maintain a trailing and a forward pointer and calculate it incrementally; a quadratic solution will not run in time.

```
int trail = 0 ;
int lead = 0 ;
int sum = 0 ;
while (lead < R) // calculate number of cameras in initial set
   if (hascamera[lead++])
      sum++ ;
int r = 0 ;
while (true) {
   for (int k=lead-1; sum<2; k--) // add cameras for this gap
      if (!hascamera[k]) {
         hascamera[k] = true ;
         r++ ;
      }
   if (lead >= N) // done?
      break ;
   if (hascamera[trail++]) // advance pointers
      sum-- ;
   if (hascamera[lead++])
      sum++ ;
}
```

D. **Contest Strategy**

First, let's consider our expected penalty time, then multiply by n! to get the final answer.
In this case, we'll can do "divisions" as multiplying by the inverse under the mod.

First, suppose we solve the problems in order i_1, i_2, .., i_n (i_k is the index of the k-th problem solved).

Then, the penalty time for this order can be written as
t_(i_1) + (t_(i_1) + t_(i_2)) + ... + (t_(i_1) + ... + t(i_n))

= n t_(i_1) + (n-1) t_(i_2) + ... + 1 t_(i_n)

Let's apply linearity of expectation to this, so we can see that the contribution of a problem to the penalty time only depends on its position.
Namely, if the expected value of the position of the i-th problem is E, this contributes (n-E) * t_i to our penalty.

Let's sort our problems in increasing order of time (if there are ties, break them by index).
It suffices to compute P_i -> expected placement of the i-th problem. This can be computed with dp.

Now, once we fix a problem, we only care about how many problems are easier to solve and harder to solve.
We should keep track of how many easier are unread, how many harder are unread, how many easier are read, how many harder are read, and whether or not we have read the problem we're interested in.
This can be summarized as dp[i][j][k][l] -> have i unread easier problems, have j unread harder problems, have k easier read problems, l = 1 if we have read the problem we're interseted in and 0 otherwise (note that number of harder read can be derived from the above variables). We can write the transition rules based on the problem statement.

Of course, this dp takes O(n^3) time to evaulate per position, which is too slow. Instead, we can note that a lot of the state is common across positions, so let's not reset the dp table in between positions. Thus, this shows that the total runtime is at most O(n^3) over all n positions.

E. Enclosure

The key to this problem is recognizing that the tree to gain control of must
lie on the convex hull of the set of all trees (both controlled and uncontrolled). To prove this, we need to show that the power that we get as a
function of the location of the tree we gain control of is well behaved. In
particular, we will show that the level set of this function (the set of all
points for which the function takes on a specific value k) is always a convex
polygon regardless of the value of k (assuming it is at least as large as our starting power). To do this, we observe that when we add a single point
to our convex hull, this point will get connected to exactly two trees we control. If we look at all such points for those two trees and a fixed k, we
see that we trace out a line segment parallel to the "chord" through those two
trees. Taking all of those line segments for a fixed k and walking along them
in order, we see that the corresponding chords must be monotonic in the angle
they make with a fixed direction, and therefore our line segments must trace
out a convex polygon. Once we have this fact, we can see that for any point
that is not a vertex of the convex hull of all trees, we can always find a
direction to move in that does not reduce our power.

Now it suffices to try all points on the convex hull of all trees. If we were
to naively compute our power resulting from each of these, we would still not

be fast enough, but computing the convex hull of all trees gives us an ordering
of the candidate trees. This ordering is useful because it allows us to reuse
the work of one convex hull calculation in the next. Recall that when we add
a tree, it gets connected to exactly two trees we already control. If we move
clockwise to the next tree on the convex hull of all trees, the next connecting
trees are also clockwise of the previous ones. We may need to skip over some
trees on our original convex hull, but we can reach the next trees just by
moving clockwise, which means if we walk all the way around the outer convex
hull once, we will only walk around the inner convex hull once. This means
that each of these candidates can be tested in amortized constant time.

To maintain the area of each candidate convex hull, we take advantage of the
Shoelace formula and the fact that we can easily update the expression for
all sides that do not include our candidate point.

F. **Illumanation**

The number of possible lamps is sufficiently large that brute force will not run in time. Something more sophisticated is called for.

Let us consider what makes a position impossible. If two lamps are horizontal and on the same row and close enough to interefere, that's one possibility. If two lamps are vertical and in the same column and close enough to interfere, that's the other. As long as none of those occur, we can find a solution.

If we represent lamp i's state with the boolean b_i, and use true for vertical and false for horizontal, this can be expressed as the conjunction of the following disjunctions:

```
For all lamps (i, j) that conflict horizontally
   b_i or b_j
For all lamps (i, j) that conflict vertically
   !b_i or !b_j
```

Since there are only two elements of each disjunction, this is equivalent to **2-SAT**, which can be solved in polynomial time.

One algorithm is to use three values for each lamp: true, false, and unknown; all are initially set to unknown. You iterate through the lamps, skipping known values. When you find a lamp that has an unknown value, you try true (vertical), and recursively chase conflicts. This recursion involves at most a linear number of calls for a given lamp, since the conflict forces the dependent lamp to take on a single value. If any conflict is found, you

"roll back" all values set during this particular exploration, and try false (horizontal). If both fail, there is no solution; if either succeeds, you keep the new lamp values. Since the conflicts are symmetrical, if you've chased all the conflicts going out from that lamp subset, you know there can be no conflicts coming back in to that subset.

For adequate performance, some mechanism to quickly find conflicting lamps is needed. Sorting the lamps in both row-major and column-major order, and providing indexing into those orders, suffice to quickly find conflicts.

G. Maximum Islands

For any cloud touching existing land, it can't hurt to make that cloud water, so we can greedily assign those. So, let's assume there is no clouds adjacent to land.

Now, in a optimal solution, all islands from clouds will consist of a single square. Consider any optimal solution. If it has an island with more than one square, we can just convert one of those to water and still have the same number of islands.

So, we want to choose the maximum number of squares such that no two are adjacent. This sounds a lot like the maximum independent set problem.

This problem is normally NP-hard, but we can notice that the grid is bipartite.

We can also notice that the complement of an independent set is a vertex, so maximizing the size of an independent set is the same as finding the size of a minimum vertex cover.

Lastly, we can use Konig's Theorem (https://en.wikipedia.org/wiki/K%C5%91nig%27s_theorem_(graph_theory)) to equate minimum vertex covers with maximum matchings in a bipartite graph. This can then be solved with flow.

H. Paint

To solve this problem, you construct best solutions for "prefixes" of the fence, and extend them by individual painters one by one. You can consider the painters by end panel left-to-right (ordering painters with the same end panel arbitrarily).

We will store the maximum number of panels painted for our prefixes for simplicity, and subtract this from the total number of panels when printing the result.

Then you iterate through the painters, and you calculate for each painter what the best value would be for that painter's end panel, both if the painter painted and if the painter did not paint.

```
best = 0
for (p1 : painter)
   best max= p1.right - p1.left + 1
   for (p2 : painters left of p1)
      if (p2.right < p1.left)
         best max= p2.best + p1
```

```
   p1.best = best
```

This is quadratic, which is not quite fast enough. You can improve the time easily by, instead of iterating through all painters, find the maximum value strictly less than the left side of the current painter using a HashMap or C++ map. This is the same solution as the previous one, but we only consider the prefix that is closest to this painter's left boundary.

```
best = 0
dmap[0] = 0
for (p1 : painter)
   // find the best solution so far to the *left* of this guy's left
boundary
   int bestprev = dmap.upper_bound(p1.left) ;
   // check if that plus this guy's span is better than what we've seen
   best max= p1.right - p1.left + 1 + dmap[bestprev]
   // add it to the map.
   dmap[p1.lright] = best ;
```

I. Postman

Postman is a simulation problem, but one with a few traps.

The basic approach is to simply deliver as many letters as possible to the furthest houses first; you must make at least that many trips at that distance, so this is optimal. If you instead try to deliver to the closest houses first, you may end up with a suboptimal solution; consider the case

House	Distance	Letters
1	10	500
2	20	1000

with a postman capacity of 1000. If we do the close house first, then the postman will deliver 500 letters to house 1, and then 500 to house 2---but will need to make a second trip to house 2, for a total time of 2 round trips at distance 20 which is 80 time units.

If instead he takes a round trip to house 2 first, he'll deliver all 1000 letters to house 2 in one round trip and finish up with one final trip to house 1 carrying only 500 letters for a total of 60.

The problem is the number of letters and the distances can be large (although the count of houses is small). If you simulate each individual trip, you will run out of time, because the total number of letters could be as high as 10^10. So you want to figure out how many round trips to a given house are required and, with a tiny bit of math, figure out how many letters will be left over to drop off along the route of the last round trip to the next closer house (and possibly additional houses).

The second trap is a common one---the result may not fit in a 32-bit int so you need to use doubles or 64-bit ints.

J. Shopping

First, this problem is equivalent to a cumulative mod over a range (that is, we take v % a[l] % a[l+1] % ... % a[r]).
The second thing to notice is that if a[i] < v, then v % a[i] <= v/2 (to prove this, consider two cases: a[i] <= v/2 and a[i] > v/2).

This shows that any one shopper can buy at most log(v) distinct items.

We can solve this by reducing it to the subproblem: given a range [i,j] and a number v, find the smallest index k where k < v, or report non exists.
This can be solved with a range tree.

Alternatively, we can also solve this using a heap and processing items left to right (see the python solution for the main idea, though it is probably impossible to pass this problem in python).

K. Tournament Wins

You win the final game of a tournament bracket iff you are the highest ranked
player in that bracket. For example, in a tournament with 8 people, you win
your first game iff you are the better player in your 2-player bracket, your
second game iff you are the best player in your 4-player bracket, and your
third game iff you are the best player among all 8 people. If we can find
the probability that you win your ith game using this, then we can use the
linearity of expectation to sum these probabilities up to get our final answer.

So what is the probability that you win your ith game? Well, there are 2^k
players in all, 2^i players in this particular bracket, and r-1 players better
than you (or equivalently, 2^k - r players worse than you). You win in the
event that of the 2^i - 1 other players in your bracket, all of them are chosen
from the 2^k - r players that are worse than you. Since all brackets are equally likely, this happens with probability
(2^k) choose (2^i - 1) / (2^k - r) choose (2^i - 1)
It's dangerous to compute these two values directly and then divide them, since
these are both very big numbers. We have two options for dealing with this:
1) We can work with logarithms of factorials instead of factorials directly.
2) We can do some simplification of the expression by cancelling common terms.

L. Barbells

The easiest way to solve this problem is simple brute force. Try every possible combination of bars and plates on both sides, and stash ones that are balanced into a set, and then order and print the set. Recursion makes this easy. Since we expect most such combinations not to balance, things are fastest if we consider the bars only at the end.

```
void explor(vector<int> &plates, vector<int> &bars, int at,
            int leftweight, int rightweight,
            set<int> seen) {
   if (at == plates.size()) {
      if (leftweight == rightweight)
         for (b : bars)
            seen.insert(b + leftweight + rightweight) ;
      return ;
   }
   explor(plates, bars, at+1, leftweight, rightweight) ;
   explor(plates, bars, at+1, leftweight+plates[at], rightweight) ;
   explor(plates, bars, at+1, leftweight, rightweight+plates[at]) ;
}
```

This runs in time at least O(3^plates), with additional factors depending on the implementation of set and the number of bars but these factors are minimal since matches are rare.

Extra Credit:

A challenging problem is to come up with a set of n plates that has the maximum number of balanced combinations for that n.