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.

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.

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.

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++ ;
  }
```

Equality

This problem is intended to be a simple warm-up, with a very fixed format that eliminates any need to do any parsing. Simply examine the zeroth, fourth, and eighth characters and check the arithmetic.

```
if ((s[0] - '0') + (s[4] - '0') == (s[8] - '0'))
  cout << "YES" << endl ;
else
  cout << "NO" << endl ;
```

A cleaner solution is to parse it using the scanning functionality in most programming languages:

```
int x, y, z ;
char op, equal ;
cin >> x >> op >> y >> equal >> z ;
cout << ((x + y == z) ? "YES" : "NO") << endl ;
```

Gravity

This one was a simulation problem; simply keep checking the board for any occurrence of a pattern where an apple would drop, and iterate as long as any change is seen.

```
while (true) {
  bool changed = false ;
  for (int i=0; i+1<h; i++)
    for (int j=0; j<w; j++)
```

```
        if (b[i][j] == 'o' && b[i+1][j] == '.') {
          swap(b[i][j], b[i+1][j]) ;
          changed = true ;
        }
  if (!changed)
    break ;
}
```

Islands

To find the minimum number of islands for a given set of cloud cover, you want to assume every island is as large as possible (and thus includes as many known L's as possible) without introducing unnecessary islands.

For instance,

```
  LCCL
```

should be a single island, but

```
  WCCW
```

should be zero islands.

The simplest way to do this is to flood-fill from every seen 'I', traversing both 'I' and 'C' values, setting all to 'W'. The number of flood-fills you need to do is the number of islands.

The small data size allows the use of a simple recursive flood-fill.

Mismatched Socks

You can always make mismatched pairs of socks using all of an even number of socks, unless you have a majority of a single color. Simply sort the socks by color and pair 0 with n/2, 1 with n/2+1, etc. Since each color occurs n/2 or fewer times, you will never have a matched pair.

If you have a majority of a single color, you can only make as many mismatched pairs as the total count of socks less the count of socks of that majority color (and every pair will have one of the majority color).

So to solve this problem, you find the count of a most common color (call that m) and then return max(n/2, n-m).

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.

SixSides

If it were not possible for there to be ties, the solution required would be to just iterate over all possible pairs of die values:

```
for (int dv1 : die1)
  for (int dv2 : die2)
    if (dv1 > dv2)
      win1++
    else
      win2++
return win1 / (win1 + win2)
```

and then print the ratio of win1 to the sum of win1 and win2. With ties, though, we need to do a bit of algebra.

Let's split the 36 pairs of die values into three categories:

Player 1 wins	win1
Player 2 wins	win2
They tie	ties

Let x be the final probability that x wins.

$$x = (win1 / 36) + ??$$

The ?? above is the probability of a win that starts with a tie. If they tie, then we simply repeat the procedure. So the ?? in the above function is just

$$(ties / 36) * x$$

So

$$x = (win1 / 36) + (ties / 36) * x$$

$$36 * x = win1 + ties * x$$

$$(36 - ties) * x = win1$$

$$x = win1 / (36 - ties)$$

$$x = win1 / (win1 + win2)$$

so the exact same expression works. The revised code, then, to exclude ties, is simply

```
for (int dv1 : die1)
  for (int dv2 : die2)
    if (dv1 > dv2)
      win1++
    else if (dv2 > dv1)
      win2++
return win1 / (win1 + win2)
```

Three Squares

There are only two arrangements of the rectangles that can result in a square:

ABC
ABC
ABC

ABB
ACC
ACC

(Or rotated/reflected versions of these.) We line up the edges of our rectangles according to thse arrangements and see whether they actually form a square. If they do, great; if they do not, we know we can't possibly make a square.

Zigzag

This problem seems like just another longest subsequence problem, like longest increasing subsequence, longest nondecreasing subsequence, and so on and so forth, one of the standard examples of dynamic programming. And it can be solved that way, acceptably, running in quadratic time; we simply use a pair of arrays:

```
maxLengthEndingUp[pos]
maxLengthEndingDown[pos]
```

We initialize all of the elements to 1 (they all form a length-1 zig-zag subsequence), and then

```
for (v in 2..n-1)
  for (i in 0..v-1)
    if (a[v] > a[i])
      maxLengthEndingUp[v] max= maxLengthEndingDown[i] ;
    else if (a[v] < a[i])
      maxLengthEndingDown[v] max= maxLengthEndingUp[i] ;
```

The maximum value seen in a traversal of these two arrays is the result.

Given the guaranteed small size of the input, this will run in plenty of time, but if we wanted to make it faster, we could use the same sort of O(n log n) improvement possible with other longest subsequence problems (which I won't detail here).

But, in reality, this problem is much simpler. All you need to do is count the times we zig or zag. A wrinkle is values equal to the previous one, but such values can be skipped. Code like this works:

```
r = 1
prevdelta = 0
prevval = a[0]
for v in a:
   if a[i] != prevval:
      delta = a[i] - prevval
      if prevdelta == 0 or delta * prevdelta < 0:
         r += 1
      prevval = a[i]
      prevdelta = delta
```

This is a linear-time, constant-space solution.