Problem A1. Balanced Shuffle (Easy)

The problem statement describes exactly what needs to be done, so we just need to implement it carefully, using a $O(n \log n)$ sorting algorithm from the standard library.

If you're using Python for this problem, the time limit can be a bit tight so you might need to optimize a bit. For example, tuples are much faster than custom classes in Python, so the following passes in 0.5s:

```
s = input()
a = []
b = 0
for i in range(len(s)):
  a.append((b, -i, s[i]))
  if s[i] == '(':
    b += 1
  else:
    b -= 1
a.sort()
print(''.join(x[2] for x in a))
```

Problem A2. Balanced Unshuffle (Medium)

In a balanced parentheses sequence each opening parenthesis corresponds (would form a pair enclosing a subexpression in a mathematical expression) to exactly one closing parenthesis and vice versa. In such a pair the balance before the opening parenthesis is always 1 less than the balance before the closing parenthesis.

After sorting, the parentheses with equal prefix balance go together. Let us consider them in groups of equal prefix balance. The first group, with prefix balance 0, will contain only opening parentheses. The second group, with prefix balance 1, will contain the closing parentheses corresponding to the opening parentheses from the first group, and potentially some more opening parentheses. The third group, with prefix balance 2, will contain the closing parentheses corresponding to the opening parentheses from the second group, and potentially some more opening parentheses, and so on. Moreover, each group except the first one will always start with a closing parenthesis (since we break ties in decreasing order of position).

This observation allows to split the input string into the groups of equal prefix balance: the first group is everything before the first closing parenthesis. The second group is everything after that and before the $k + 1$-th closing parenthesis, where k is the number of opening parentheses in the first group, and so on.

Having done that, we can also construct the original sequence group-by-group. After processing a certain number of groups we will have a string that is a parentheses sequence but not a balanced parentheses sequence yet: some opening parentheses in it will be marked as unmatched (for example, we can use opening square brackets instead of parentheses to denote those). When processing the next group, we put each closing parenthesis and all opening parentheses that follow it after the corresponding unmatched opening parenthesis.

Here is how this process works on the sample testcase:

- After the first group, our string is `[`.
- After the second group, our string is `([[)`.
- After the third group, our string is `(()([))`.
- After the fourth group, our string is `(()(()))` and we have correctly recovered the original sequence.

The straightforward implementation of this algorithm runs in $O(n^2)$, fast enough for the constraints of this subtask.

The transformation described in this problem is called a *sweep map* in the literature, and searching the Internet using that term will lead to a much deeper dive on the subject if desired.

Problem A3. Balanced Unshuffle (Hard)

Solving this problem requires a more careful implementation of the algorithm described in the previous subtask. Since we only ever insert new parentheses into the current string after an unmatched parenthesis, we can use a linked list of characters instead of a string to represent it, and also store a vector of pointers to unmatched parentheses in it. This way each insertion will be done in $O(1)$ for a total running time of $O(n)$.

Another way to view this solution is that we are constructing a rooted tree corresponding to the original parentheses sequence in a breadth-first manner, layer-by-layer, but then need to traverse it in a depth-first manner to print the original sequence. This runs in $O(n)$ if we store the tree using pointers or indices of children.

In fact, we can notice that the layer-by-layer construction is not really necessary: we just create a queue of tree nodes, each opening parenthesis creates a new child of the node at the front of the queue and puts it at the back of the queue, and each closing parenthesis removes the node at the front of the queue from the queue. This way the implementation becomes very short.

Problem B1. Exact Neighbours (Easy)

Place all houses on the diagonal. Notice that for each house i either to the left or to the right, on the diagonal, there will be a house with the desired distance a_i

Problem B2. Exact Neighbours (Medium)

We don't care about satisfying the condition for the first house, since this house will always be at a distance 0 from itself. There are at least two approaches for constructing the desired placement.

Approach 1: Let's sort all a_i in non-increasing order, then we can place houses in the zigzag order the following way: we will start by placing the first house in $(1, 1)$, then we will place each new house in the next column, alternating the relative position to the previous house up to down. Since a_i is sorted in non-increasing order, we will never get out of the bounds of the field and the i-th house in the sorted order will satisfy the condition for the $i - 1$ house in the sorted order.

Approach 2: Place the first house in $(1, 1)$. Then, place each next house in the next column. If $a_i \geq i$, you can place the i-th house in such a way that it will be a_i away from the first one. If $a_i < i$, you can place the house in the same row as the house with index $i - a_i$ and the house with this index will be a_i away.

Problem B3. Exact Neighbours (Hard)

Let's divide all valid arrays a into three cases and solve the problem for each of them.

First case

There exists i such that $a_i = 0$. We can use the solution to Medium.

Now we can assume $1 \leq a_i \leq n$.

Second case

There are two numbers $i \neq j$, such that $a_i = a_j$. We can adapt the algorithm from Medium to this case. For example, consider the first approach to Medium. We can do the same zigzag algorithm but skip the satisfaction of the next house when we have that in the sorted order two values are the same ($b_{i-1} = b_i$ where b is the sorted in non-increasing order array a).

Third case

In the remaining case the set of values will be exactly $\{1, 2, \ldots, n\}$. When $n = 2$, the answer is -1. When $n \geq 3$, we again sort a_i and do the zigzag until we meet $3, 2, 1$. At that point we can do a special construction for $3, 2, 1$ which is not hard to come up with.

Problem C1. Game on Tree (Easy)

We are given a linked list with an initial coin at index $1 \le i \le n$. There are $i-1$ nodes to its left and $n-i$ nodes to its right. Note that after the first move, all the remaining moves are fixed since there will be exactly one inactive neighbor.
If one of $i-1, n-i$ is odd, Ron should move to the corresponding node as there will be an even number left thus guaranteeing a victory for Ron. Otherwise, Hermione is guaranteed to win.

Problem C2. Game on Tree (Medium)

Let's root the tree at node u_1 (the start node). By doing so, we guarantee that if the coin is at node v, the parent of v is already active and thus we can only go down in the tree. This means that each subtree can be seen as its own independent game. So, each node is either a winning or losing position (w.r.t. to the player whose turn is next). We will find recursively whether u_1 corresponds to a winning game for Ron.
If some child of a node v is a losing position, then the current player should move the coin to that child to guarantee a win. If all children are winning positions, the current player will surely lose. So, v is a winning position iff it has a child that is a losing position. Note that the leaves are losing positions.
This is solved in $O(n)$ time.

Problem C3. Game on Tree (Hard)

Repeating the previous solution for each u_i is too slow as it gives a $O(n.t)$ solution.
Instead, we will reuse the computations we did when assigning losing/winning positions. We do so using the re-rooting technique. Let's first compute the positions of the nodes in the tree rooted at 0. We will now find in linear time the position of each node if the tree was rooted at that node.
Let v be the current root (initially $v = 0$) and let w be a child of v. Let T be the initial tree and T' be the tree re-rooted at w. We know the positions of all nodes in T, and we want to find the position of nodes in T'. First note that only the positions of v and w may differ because the subtrees of all the other nodes remain unchanged.
If v has a child other than w which corresponds to a losing position (in T), then v is a winning position in T'. If v has no losing child or has exactly one losing child which is w (in T), then it is a losing position in T'. Similarly, if w has some child that is a losing position (including v in T'), then it is a winning position otherwise it is a losing position in T'. The above checks can be done in constant time by counting the number of losing children before the recursive calls (this count also needs to be maintained before recurring on a subtree). We thus recur on W and repeat. Once the recursive call is over, we backtrack our changes before recurring on a different child of v.

At the beginning of each recursive call, we know whether Ron or Hermione wins if the game started at that node so we can answer all the queries by doing linear-time pre-processing.

Problem D1. Arithmancy (Easy)

Given that we need to output 3 words only, we can manually (trying a few options on paper until we find a solution) construct 3 words such that all 9 spell powers are different.

Problem D2. Arithmancy (Medium)

Now we need to find 30 words, so manually solving on paper is out of question. However, "just trying" still works: what we can do is to keep generating random words until we find 30 that have all 900 corresponding spell powers different. Depending on how you fast is your computation of the spell power you can either do this during the time limit, or precompute locally and then just submit a solution that prints the precomputed words.

To make sure that we have to precompute only once instead of 30 times (for $n = 1$, $n = 2$, ...), we can find 30 words such that the i-th word has length $30 \cdot i$. This way the first n words from this list give a valid answer to the problem for any $n <= 30$.

To compute the spell power, we can either use the naive approach of taking all substrings, sorting them and finding unique ones, some optimization thereof (for example, if the words are randomly generated, the chances that sufficiently long substrings coincide are vanishingly small and can be ignored), or the asymptotically optimal approach using any of the suffix data structures (the suffix array, or the suffix automaton, or the suffix tree).

Problem D3. Arithmancy (Hard)

The previous approach of just generating random words does not even come close to working for $n = 1000$ (in fact, it is hard to push it beyond roughly $n = 50$). Now we need to add our own insights to the process.

The first idea is: in order to answer the queries, we will likely need to know the spell power for all n^2 possible spells. But even if we use a suffix array to compute the spell power, it will be too slow to compute the spell power 10^6 times for spells of length around $6 \cdot 10^4$. Therefore it is better to choose the magic words in such a way that computing the spell power of a concatenation of two such words can be done in $O(1)$.

One way to achieve this is to choose some family of magic words g_i with a regular structure and a single parameter i, and just figure out the function $f(i, j)$ for the spell power for the concatenation of g_i and g_j on paper, hopefully it will be easy to figure out and quick to compute.

Then, what we can do is to go in increasing order of i, and try to take the next word g_i into our set by checking if after adding it to the set, the newly added spells have spell powers that are different from each other and from the existing spell powers. Since we compute the spell power in $O(1)$, the total running time of this process is around $O(n \cdot k)$, where k is the number of g_i we had to check before we managed to add n of them into the set.

The only remaining difficulty, and of course it is actually the main part of the solution, is to choose the family g_i. Here we have two main competing constraints:

- The words have to have simple structure, so that we can compute $f(i, j)$ quickly (in both senses: quickly figure out the formula, and the formula should be simple).
- However, we must have $f(i, j) \neq f(j, i)$. It turns out that this rules out many word families with a very simple structure.

We expect that once you realize the two constraints above, after a small amount of experimentation you will stumble upon a family that works. Here are two families that work that we know about, but I expect that there are many many more:

- $g_i = \mathtt{XOX}^{i-1}$. So $g_1 = \mathtt{XO}$, $g_2 = \mathtt{XOX}$, $g_3 = \mathtt{XOXX}$, and so on. We found this family in the upsolving solution from one of the teams in the onsite round.
- $g_i = \mathtt{XO}^i\mathtt{XO}^i$. So $g_1 = \mathtt{XOXO}$, $g_2 = \mathtt{XOOXOO}$, $g_3 = \mathtt{XOOOXOOO}$ and so on. This family actually leads to a string slightly longer than 30000 for $n = 1000$, but it can be fixed by skipping the short strings that lead to too many collisions later (so we actually use $g_i = \mathtt{XO}^{i+5n}\mathtt{XO}^{i+5n}$). This was the original reference solution.

As you noticed, our solution does not use the fact that the interactor is guaranteed to be random. The reason the problem was set like this is that we're not aware of a way to find collisions quickly enough that we could use to simply check if the n printed magic words yield n^2 distinct spell powers. Therefore to make the problem well-defined and avoid the need for contestants to guess how strong the checker is, we made it weak (just trying 1000 random spells) but well-specified.

Problem E1. Trails (Easy)

Let $t_i := s_i + l_i$. The number of possible paths between cabin i and cabin j is $t_i t_j - l_i l_j$.

Let $\mathbf{v}_{k,i}$ be the number of paths that ends in cabin i after walking for k days. We then have

$$\mathbf{v}_{0,1} = 1, \mathbf{v}_{0,i} = 0 \text{ for } i \neq 1,$$

and

$$\mathbf{v}_{k+1,i} = \sum_{j=1}^{m} (t_i t_j - l_i l_j)\mathbf{v}_{k,j}.$$

Thus, we can compute $(\mathbf{v}_{k+1,1}, \ldots, \mathbf{v}_{k+1,m})$ from $(\mathbf{v}_{k,1}, \ldots, \mathbf{v}_{k,m})$ in $\mathcal{O}(m^2)$ time using this formula. We do this n times to get $(\mathbf{v}_{n,1}, \ldots, \mathbf{v}_{n,m})$. The solution to the problem is now

$$\mathbf{v}_{n,1} + \cdots + \mathbf{v}_{n,m}.$$

The total running complexity is $\mathcal{O}(nm^2)$.

Problem E2. Trails (Medium)

Let $t_i := s_i + l_i$. The number of possible paths between cabin i and cabin j is $t_i t_j - l_i l_j$.

Let $\mathbf{v}_k$ be the vector whose ith entry is the number of paths that ends in cabin i after walking for k days. We then have

$$\mathbf{v}_0 = (1, 0, \ldots, 0),$$

and

$$(\mathbf{v}_{k+1})_i = \sum_{j=1}^{m} (t_i t_j - l_i l_j)(\mathbf{v}_k)_j.$$

Let A be the $(m \times m)$–matrix given by

$$A_{i,j} = t_i t_j - l_i l_j.$$

Then the formula for $\mathbf{v}_{k+1}$ can be rewritten as

$$\mathbf{v}_{k+1} = A\mathbf{v}_k.$$

Hence,

$$\mathbf{v}_n = A^n \mathbf{v}_0.$$

Now, we can compute A^n in $\mathcal{O}(\log(n)m^3)$ time using the following recursive formula.

1. Compute $B := A^{\lfloor n/2 \rfloor}$
2. If n is even: Return B^2
3. Else: Return $B^2 A$

The solution to the problem is now

$$(\mathbf{v}_n)_1 + \cdots + (\mathbf{v}_n)_m.$$

The total running complexity is $\mathcal{O}(\log(n)m^3)$.

Problem E3. Trails (Hard)

Let $t_i := s_i + l_i$. The number of possible paths between cabin i and cabin j is $t_i t_j - l_i l_j$.

Let $\mathbf{v}_k$ be the vector whose ith entry is the number of paths that ends in cabin i after walking for k days. We then have

$$\mathbf{v}_0 = (1, 0, \ldots, 0),$$

and

$$(\mathbf{v}_{k+1})_i = \sum_{j=1}^{m} (t_i t_j - l_i l_j)(\mathbf{v}_k)_j.$$

Let A be the $(m \times m)$–matrix given by

$$A_{i,j} = t_i t_j - l_i l_j.$$

Then the equation for $\mathbf{v}_{k+1}$ can be rewritten as

$$\mathbf{v}_{k+1} = A\mathbf{v}_k.$$

Hence,

$$\mathbf{v}_n = A^n \mathbf{v}_0.$$

Now, observe that A can be written as $A = BC$, where B is the $(m \times 2)$–matrix

$$\begin{pmatrix} t_1 & l_1 \\ t_2 & l_2 \\ \vdots & \vdots \\ t_m & l_m \end{pmatrix},$$

and C is the $(2 \times m)$–matrix

$$\begin{pmatrix} t_1 & t_2 & \cdots & t_m \\ -l_1 & -l_2 & \cdots & -l_m \end{pmatrix}.$$

We now have

$$A^n = B(CB)^{n-1}C.$$

As CB is a (2×2)–matrix, we can compute $(CB)^{n-1}$ in $\mathcal{O}(\log(n))$ time (see the solution in Trails (medium version) for an explanation of the matrix exponentiation algorithm). Finally, computing the expression

$$\mathbf{v}_n = B(CB)^{n-1}C\mathbf{v}_0$$

takes $\mathcal{O}(m + \log(n))$ time (note: we need to multiply the expression from the right side; multiplying from the left would take $\mathcal{O}(m^2 + \log(n))$ time). The solution is $(\mathbf{v}_n)_1 + \cdots + (\mathbf{v}_n)_m$.

Problem F3. Playing Quidditch (Hard)

This subject does not contain theoretical difficulty: it is only needed to simulate the game following the rules described in the statement.

To be able to perform the simulation easily, it is useful to store wisely the current state of the game.

- the position of the goals, either in a grid, a set or a list
- the position of the players, for example a list containing the position of each player
- the position of the balls
- the current score of each team

Then, at each step of the simulation, the current state must be updated following the rules.

Problem G1. Min-Fund Prison (Easy)

The cells and corridors in this subtask form a tree. No matter how we divide the prison into two complexes, there will be at least one existing corridor connecting them, but we must have at most one such corridor, which means that we do not need to build any more corridors.

For each existing corridor, removing it splits the tree into two parts, and those two parts are the only possibility to have two complexes connected only by this corridor. So we need to compute the sizes of the two parts for every corridor, and then pick the corridor that minimizes the sum of squares of the sizes.

In order to compute the sizes of the two parts for each corridor quickly we can root the tree and then use depth-first search that recursively computes the size of each subtree. The running time of this solution is $O(n)$.

Problem G2. Min-Fund Prison (Medium)

The graph is no longer a tree in this problem, but we can try to generalize the solution for the first subtask.

First of all, suppose the graph is connected. Similar to the first subtask, there will always be at least one existing corridor connecting the two complexes, so we do not need to build new corridors. Moreover, there will be exactly one existing corridor connecting the two complexes if and only if said corridor is a bridge in the graph, and the two complexes are the two connected components that appear after removing this bridge. We can modify the depth-first search algorithm that finds bridges in the graph to also compute the component sizes, and therefore we can solve our problem for connected graphs.

Now, what to do if the graph is not connected? There are two cases:

1. At least one connected component will be split between the two complexes. In this case exactly one component must be split in this way, and the split must be done using a bridge in this component similar to the connected graph solution. For every other connected component it must go either completely to the first complex or completely to the second complex. We need to build additional corridors to connect components within the complexes, and the number of additional corridors is equal to the number of connected components in the graph minus one.

2. No connected component will be split between the two complexes. In this case we need to build additional corridors both to connect components within the complexes, but also to connect the complexes to each other. The number of additional corridors is still equal to the number of connected components in the graph minus one.

Since the number of additional corridors is constant, we still need to minimize the sum of squares of the sizes of the complexes. We now need to solve a knapsack problem to find out the possible sizes of the first complex. To account for the fact that at most one component may be split using a bridge, we can add a boolean flag to the knapsack dynamic programming state that tracks whether we have already split a component.

The running time of this solution is $O(n^2)$.

Problem G3. Min-Fund Prison (Hard)

In the hard subtask, the $O(n^2)$ solution is too slow.

One way to get it accepted is to use bitsets to speed it up, as the knapsack transitions can simply be expressed as a bitwise shift + bitwise or.

However, there is also an asymptotically faster approach. First of all, instead of remembering the possible splits using a bridge for each component, we will just remember for each component size, how it can be split by a bridge. Since the sum of component sizes is n, this needs only $O(n)$ memory.

Now, if we have at least four components of the same size x, for the purposes of our knapsack we can replace two of them with a component of size $2x$, and the set of reachable complex sizes will not change. Since we keep at least two components of size x, the set of reachable complex sizes will not change even if we later split one of the components of size x using the bridges.

If we repeat the above procedure until we have at most three components of each size, the total number of components will be $O(\sqrt{n})$. Therefore the knapsack dynamic programming that does not take splitting via a bridge into account will run in $O(n \cdot \sqrt{n})$.

In order to tackle the splitting via a bridge fast, let us first run the dynamic programming that does not allow to split components, but instead of computing a boolean for each complex size that tells if this size can be reached, we will compute the number of ways to reach it, modulo a large prime.

A step of this dynamic programming is reversible, therefore we can then for each component size compute in $O(n)$ which sizes can be reached without using one of the components of this size. Now we need to combine this with the ways to split a component of this size using a bridge, and since the sum of squares of the sizes is smaller whenever the sizes are closer to each other, for each way to split the component using a bridge we need to find the reachable state of the dynamic programming that is the closest to $\frac{n}{2}$ minus the size of the part disconnected by the bridge, which can be done in $O(n)$ for all ways to split using a bridge via the two pointer method.

Therefore the total running time of this solution is $O(n \cdot \sqrt{n})$.