Hints for the first competition day's tasks

Parachute rings

A suboptimal solution covering all but the last subtask considers the following conditions:

- if there is a vertex V of degree >= 4, no other vertex can be critical (because removing V still leaves one or more vertices of degree >= 3); so if there is more than one vertex of degree >= 4, there are no critical vertices;
- if there is a vertex V of degree 3, each critical vertex is either V or one of its neighbors;
- if there is a cycle, all critical vertices lie on the cycle;
- if the graph is linear (a set of disjoint paths), all of its vertices are critical.

These checks can be easily extended to the dynamic case of the last subtask: the only nontrivial check is keeping track of cycle formation, which can be dealt with using suitable data structures (union-find d.s., etc.).

Pebbling odometer

As an illustrative example, we give directly the solution of Subtask 5, where code sharing is employed. Note that finding the minimum is not complicated, removing one pebble per cell. However, all the removed pebbles should be put back to their cells, and this complicates the solution.

Subtask 5: solution generator in Python

```
# For each possible minimum value (except 15),
# look for a cell that holds that many pebbles.
# Various optimizations reduce the number of instructions.

# The first step is looking for zeroes.
print "jump 0_scan_all"

for i in xrange(0,15):

  # This section tries to move on the next row after a single scan. If it hits
  # the border, we're ready to search for the next candidate minimum.
  print "%d_test_next_row:" % i
  print "right"
  print "border %d_scan_all" % (i+1)
  print "move"
  print "right"
  print "%d_test_next_row_l1:" % i
  print "border %d_test_next_row_l1end" % i
  print "move"
  print "jump %d_test_next_row_l1" % i
  print "%d_test_next_row_l1end:" % i
  print "right"

  # Start the evaluation of the next row of the grid.
  print "%d_scan_all:" % i
  print "right"
  print "%d_test_scan_row:" % i
  for j in xrange(i):
    print "get"
  print "pebble %d_test_scan_row_continue" % i
  print "jump end_%d" % i
  print "%d_test_scan_row_continue:" % i
  for j in xrange(i):
    print "put"
  # When it hits the border, try to go to the next row and go back to the
  # first column.
  print "border %d_test_next_row" % i
  print "move"
  print "jump %d_test_scan_row" %i

# When you find the minimum, you can share the code that puts back the pebbles
# in the cell.
```

```
for i in xrange(14,0,-1):
    print "end_%d:" % i
    print "put"
print "end_0:"
# If all the cells have 15 pebbles, any position is ok.
print "15_scan_all:"
```

Crayfish scrivener

A clever way to get an efficient solution consists in representing the evolution of the system through a trie, containing all the contents of the text so far; a point in time is represented by a single pointer to a node in the trie.

Command processing requires O(1) time:

- typing a letter just requires moving down in the trie (creating a new node if necessary)
- undoing K commands requires moving K states back.

For all the subtasks except the final one, after processing all the commands, the final contents can be extracted from the trie into an array and used to answer queries in O(1) time, giving O(N) time and space overall.

Subtask 5 requires a definitely more sophisticated approach to find a point in the text. For this it is sufficient to be able to determine the k-ancestor of the current node: There are a number of standard data structures for this problem that give O(N log N) time overall. For example, every node at depth D can contain a pointer to its 2^k-th ancestor, where k is the position of the rightmost 1 in the binary expansion of D.

Hints for the second competition day's tasks

Ideal city

Simple solutions use Floyd-Warshall algorithm or iterated BFS on the unary-cost edges, and both require O(N) space: time is O(N^3) for Floyd-Warshall, and O(N^2) for the iterated BFS, which requires N times the number O(N) of edges.

A more efficient solution is the following one.

- For every row r, consider the connected groups of cells on row r; each such group becomes a node of a tree, with a weight corresponding to the cardinality of the group. Two nodes of this tree are adjacent iff there are at least two cells in the corresponding groups sharing a common edge. Repeat the same argument for every column c.
- The above description yields two node-weighted trees, one (let us call it TH) corresponding to horizontal node-groups and another (TV) for vertical node-groups.
- Now, a shortest path between any two cells can be decomposed into two shortest paths along TV and TH: the two corresponding integers are called the vertical and horizontal contribution, respectively.
- Let us limit ourselves to the horizontal contributions. The sum of all horizontal contributions can be computed as the sum of w(x)*w(y)*d(x,y) over all possible distinct pairs of distinct nodes x and y in TV: here, w(x) and w(y) are their weight (number of cells) and d(x,y) is their distance in TV.
- The latter summation can be computed in linear time in the number of edges of TV, by observing that it is equivalent to the sum of S(e)*S'(e) over all edges e of TV, where S(e) and S'(e) are the sum of the weights of the two components of the tree obtained after removing the edge e.

Last Supper

Let us first describe how to compute the optimal strategy of Leonardo in O(N log N) time.

- Use an array of size N and scan the requests in C backwards: for each request, it is possible to compute how far in the future the same color will be requested.
- Process the requests in C forward: for each request, we can determine if the color is in the scaffold and which of the colors to remove; the latter can be established in time O(log N) by keeping a priority queue of colors where the priority is determined by the time of the next request.

For encoding the advice, the trivial solution would be to use N log K bits, i.e. log K for every color fault (i.e., every time the requested color is missing from the scaffold): this way we might specify exactly which color (within the K colors available on the scaffold) should be removed.

Here is an alternative encoding that uses only N+K bits and it is optimal in the worst case.

- Divide the colors currently in the scaffold between “active” and “passive”: an “active” color is one that will be requested before it is removed from the scaffold according to the optimal strategy of Leonardo; a “passive” one will not.
- Using N+K bits it is possible to keep track of which colors are active:
  - The initial active colors can be specified using K bits overall.
  - Moreover, with each request we can provide a bit saying if the currently requested color is active.
- Note that removing any passive color is always ok. Of course, if you remove it arbitrarily you will not produce the optimal solution you started from, but an equivalent one with the *same* number of colors removed.

Jousting tournament

Solutions of various ranges of complexity are possible: all of them are essentially based on the trivial idea of trying all possible positions and simulating the tournament.

A trivial way to do that takes time O(N^3). We hereby describe a O(N^2)-time solution.

- The whole tournament can be thought of as a tree whose leaves represent the knights and where all other nodes represent the winner of a round. The structure of the tree is the same regardless of where we put ourselves in the initial rank, only labels of the nodes (i.e., round winners) change.
- The latter tree leads to an O(N^2) solution: the tree construction takes time O(N^2); then for each of the possible N positions, we have to determine how far up our knight can go, so we are left with another O(N^2) checks.

To go down to O(N log N) time we need to optimize the tree construction and the tournament's simulations.

- To speed up the tree construction to O(N log N), we can use a binary range tree to get quickly the knight that is currently in any given challenged position.
- To speed up the second phase, we make two observations.

1. Let us call “white”/”black” the knights that are weaker/stronger than the late knight. Then, when we place the late knight in a certain position, we have to determine how far up we can go without finding a node that has some black leaf below it. For example, if we decide to place the late knight in the leftmost position, we want to find the leftmost node that has the longest path to a leaf and contains only white leaves under it.
2. Every time we place the knight in a given position, we are simply shifting (to the left) every knight to his left.

- Combining these two observations, we don’t need to actually try a position to see how far up we can go: it is sufficient to proceed as described in the first observation but allowing the leftmost descendant to be black, and requiring only the remaining ones to be white.
- Doing it in this way, the second phase is O(N) because of the second observation.