# IDI Open
# Programming Contest
# April 21th, 2012

## Solution sketches

## Problem A

# Angry Grammar Nazi

Problem author: Ruben Spaans

For each line you can do three separate loops over all words, checking for each type of offending word:

- For each word that equals `u` or `ur`, increase the counter.
- For each occurrence of `should` or `would` followed by `of`, increase the counter.
- For each word containing `lol`, increase the counter. Use your favourite language's function for searching for a substring within a string.

Finally, output counter * 10.

Splitting the sentence into individual words can be easily done in Java using `String.split(" ")`.

# Problem B

## Neurotic Network

Problem author: Christian Neverdal Jonassen

Process each node: Add its value multiplied by its outgoing edge weight to the downstream neighbour's value. A node can only be processed if all its upstream neighbours have been processed. There are multiple ways of doing this.

One way is to maintain a queue which at any given time contains all nodes with no unprocessed upstream neighbours, and pick new nodes to process from this list. A node is added to this list when its number of unprocessed upstream neighbours reaches zero. This check can be performed by keeping track the "in-degree" of each node, which is the number of upstream neighbours. When a node is processed, decrease the downstream neighbour's in-degree by 1 and push it to the queue if the in-degree reaches 0.

Another way is to build an adjacency list of neighbours for each node, and do a depth-first search, calculating the final value in a top-down manner. This approach only works if the recursive function has a small stack frame. Otherwise one could risk overflowing the stack.

Approaches needing $O(N^2)$ time (where $N$ is the number of nodes), for instance by searching through all nodes for one with in-degree 0 are not fast enough for $N = 10000$.

The easiest way to find the parity of the output value is to do all intermediate calculations modulo 2,000,000,014, as this doesn't change the parity of the node values. It is also possible to use the original modulus and keep track of the parity for each node and updating it using boolean arithmetic.

# Problem C

## Special Services

Problem author: Eirik Reksten

This problem requires you to efficiently verify whether it is possible to satisfy the demands of a given set of bookings, as this is to be done a whole bunch of times for each test case. To do that, you need to quickly determine whether all demands (across all the current bookings) can be matched to a unique employees. As in so many cases, matching problems can be solved using a max-flow algorithm.

Imagine a bipartite graph consisting of employees on one side, and all possible qualifications on the other. Let there be an edge between each employee and all the qualifications he/she possesses. Now, add an edge from each employee to a sink node, with a flow capacity of one.

To check whether a set of demands could be satisfied, you would then need an edge (capacity one) from a source node to the qualification for each demand of that qualification. If the total flow in this graph is equal to the total number of demands, the whole set of bookings can be accepted. Whenever a new booking is added, you will need to check whether the system consisting of all previously accepted bookings and the new one can be accepted as a whole.

If you build this graph from scratch every time a new booking/cancellation is made, you lose the information already stored in the graph, and thus also a lot of time. We therefore need to handle bookings and cancellations with care.

Adding a booking can be done easily by inserting a new source node into the (accepted) graph, with its own edges to every demanded qualification. Increase flow as far as it goes, and if the increase is equal to the amount of demands made by that booking, it should be accepted and you're finished. If not, do a cancellation of this last booking, as explained next.

When cancelling a booking, you need to remove all flow and edges associated with that particular booking. As the graph is directed and acyclic, this is as simple as first tracing all flow from the booking node to the sink, removing it along the way. Where there is more than one possible path, any will do. Then simply delete the node and all edges entering or leaving it.

The solution will also require appropriate data structures for keeping track of bookings, qualifications and employees, but as that is relatively simple compared to the rest of the solution, this is left as an exercise.

**Problem D**

# Negative People in da House

Problem author: Christian Neverdal Jonassen

Initially, assume that the house is empty. Simulate the process and at each time keep track of the number of people $P$ in da house. If $P$ becomes negative, you know that there must have been at least $-P$ people in the house to begin with. Let $Q$ be the smallest value of $P$ encountered during the simulation. The answer that should be output is $-Q$.

## Problem E

# Ruben Spawns

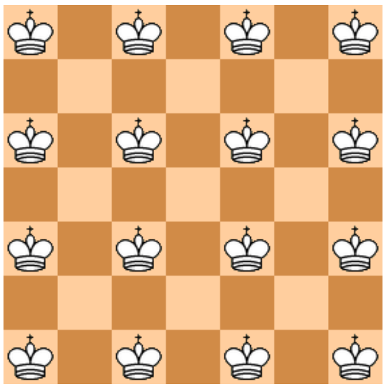
Problem author: Christian Neverdal Jonassen

This problem can be solved by taking the most capable available minion until the total workload is completed. This can be done by sorting the minion work units in reverse order, and find the number of elements you have to sum together before reaching the given workload. If the given workload cannot be reached even by using all minions, output "`no rest for Ruben`".

# Problem F

## Kings on a Chessboard

Problem author: Ruben Spaans



At first this might seem like a backtrack problem similar to the well-known $N$-queens problem. But this approach is too slow here.

In order to solve this we need the following observation: Assume that we have placed kings on the $r$ first rows, and have yet to place kings on the remaining board. Then, only the kings placed on the last of these rows determine where we can and cannot place kings on the next unprocessed row.

This, along with the maximal board dimension size of 15, suggests a dynamic programming approach, where we keep track of where we are allowed to place kings in the next row we are about to process. In addition, we need to keep track of the number of rows we have processed, and the remaining number of kings to place.

The subproblems in the DP can be described as this: In how many ways is it possible to place kings on the first $r$ rows so that we have $j$ kings left to place, and the next unprocessed row has a set of cells $S$ where kings can be placed? The initial state is the one where no rows are processed and we have $k$ kings left to place (where $k$ is given from the input), which can be achieved in one way.

The set $S$ of cells to place the kings can be represented with a bitmask of size 15, where a set bit indicates that the king can't be placed there. We then iterate over all the different subproblems starting from the empty board, and try all ways to place the kings in the current row, and add the number of combinations to later states.

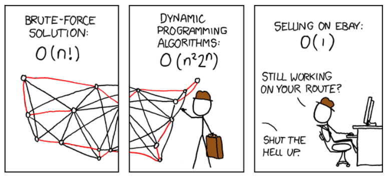
An upper bound for the number of states is the number of bitmasks times the number of rows times the number of kings left to place. It is never possible to place more than 64 kings on a $15 \times 15$ board, so we don't need to include subproblems with 65 or more remaining kings in the state space. Hence, the upper bound on the number of states are $2^{15} \cdot 16 \cdot 65$. Also, not all $2^{15}$ combinations of bitmasks as possible as a consequence of it not being possible to place two kings next to each other. We will not prove it here, but the number of possible bitmasks is $F_{17} = 1597$, the 17th Fibonacci number, which is significantly less than $2^{15}$. However, it was still possible to solve the problem with a DP array with $2^{15} \cdot 16 \cdot 65$ elements, which would require around 136 MB when using 32-bit integers.

Some other optimizations are possible, like storing only two rows simultaneously in memory and precalculating all answers, but none of them were required in order to get accepted.

# Problem G

## Travelling Cellsman

Problem author: Christian Neverdal Jonassen



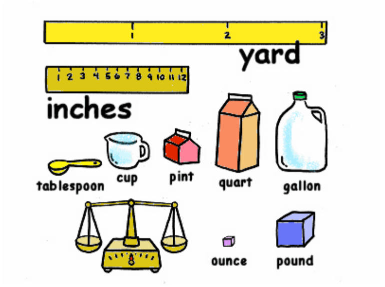
This problem can be broken down into a few cases, all of which can be solved with closed formulas. The first observation is that the starting position "`P`" doesn't matter, since all cells are going to be visited as a part of a closed loop.

- $x = 1$ or $y = 1$: Assume without loss of generality that $x = 1$. The path goes back and forth, so $2(y-1)$ moves are needed. Notice that the answer for the case $x = y = 1$ is 0.
- $x \cdot y$ even: The answer is $xy$.
- $x \cdot y$ odd: At one point one has to visit one cell, and go back to a previously visited cell. Only one extra visit is needed, so the answer is $xy + 1$.

# Problem H

## Dimensions

Problem author: Børge Nordli



The problem describes three operations defined on a *size*, which consists of a *quantity* and some *units*. The main part of the solution is to create a *standard representation* of a size, using one floating-point number for the quantity and an integer list of the powers of each of the 6 SI units. The operations on two sizes in standard representations are as follows:

$X + Y$: The two sizes must have exactly the same dimension powers. The quantity of the sum is $X$`.quantity`$+$ $Y$`.quantity`.

$X - Y$: The two sizes must have exactly the same dimension powers. The quantity of the difference is $X$`.quantity`$-$ $Y$`.quantity`.

$X \cdot Y$: The quantity of the product is $X$`.quantity`$\cdot$ $Y$`.quantity`. The dimensions of the product is the sum of the respective dimensions.

In addition, the power operation is needed:

$X^N$: The quantity of the expression is $(X\texttt{.quantity})^N$. Each dimension power is multiplied by $N$.

When a new unit is defined, you should look for any non-SI units in the input. Their standard representations should be looked up (for instance in a hash table, but a simple array or list should suffice), and using the power and multiplication operations, the new unit should fairly easily be converted to the standard representation as well. When you are asked to convert a size, convert it to the standard representation, just as above. When you are asked to compute the answer to an expression, first split the input by the operator (note that "/" is not an operator in this problem), convert each side to the standard representation and perform the requested operation.

The second part of the solution is to perform the parsing of a size. The syntax definition looks scary, but when you look at it closely, it is fairly friendly. Everything is nicely surrounded by spaces, so `String.split(" ")` should be used first. Then the first part is always the quantity, and the rest are units, possibly with exponents. If a division sign, `/`, is encountered, every subsequent unit power should be negated.

When outputting a size, just make sure that you follow the clarifications of not outputting the unit if its power is 0, not outputting the exponent if its power is 1, and wait with outputting units with negative power until you have written the division sign.

The limits clarifies that a `double` should be sufficient for computing quantities, and that you should not need to take special precautions.

Before submitting, it is wise to test your program with some special cases, like the following:

- `2         // no units`
- `2 / unit  // only units with negative powers`
- `-2 s      // negative quantity`
- `1E-2      // negative mantissa`

# Problem I

## Space Travel

Problem author: Eirik Reksten

The hard part of this problem is calculating the distance between the line segments (and the points and line segments). Once that is done, as well as correctly implemented, this is a simple shortest path problem. The amount of nodes is relatively small, so almost any shortest path algorithm will suffice. Pick your favorite among for instance Bellman-Ford, Dijkstra and Floyd-Warshall.

Calculating the distances requires a little bit of work, however. Lets first solve the simpler case of distance between a line segment $AB$ and a point $P$.

First, we check whether the point is closest to one of the end points on the line. If it is, we can simply calculate the distance between those two points. To find whether this is the case, one can use the dot product of the lines $AB$ and $BP$ to find the angle $\angle ABP$ (and conversely $BA \cdot AP$ for the angle $\angle BAP$. This is due to the fact that

$$AB \cdot BP = |AB||BP|\cos(\theta) \tag{1}$$

where $\theta$ is the angle between $AB$ and $BP$.

If this is not the case, the distance is equal to the distance between $P$ and the infinite line passing through $A$ and $B$. The absolute value of the cross product between $AB$ and $AP$ is equal to the area of the parallellogram with two sides defined by the lines $AB$ and $AP$. Using the fact that this area is also equal to the length of the line $AB$ and its distance to $P$, it is simple math to find the distance.

Now that we have a way to calculate the distance between a line segment and a point, we can do a ternary search along one segment to find the distance between two line segments.

## Problem J

# C.S.I: P15

Problem author: Christian Neverdal Jonassen

The first step is to count the flowers. Iterate through each ground cell, and do a flood fill from the cell to the north if it contains a flower character and it has not been visited in a previous call to the flood fill routine. The flood fill can be implemented using breadth-first or depth-first search, or even a disjoint-set data structure. The flood fill routine checks each of the 8 neighbours for a previously unseen flower character, and in turn processes these and marks them as visited. The number of flowers is identical to the number of times the flood fill routine was successfully launched.

To count the number of birds, scan the entire image after the string `/\/\`. This component is a bird if and only if it has not been visited by the flood fill routine, and each neighbouring cell is air or is outside the grid.