NZPC 2020 Editorial (Problems I-P)

Problem I: Extreme TTT (NZPC 2010)

The basic idea is to iterate over all the possible lines and check if the three cells in each line contain the same symbol.
There are several ways to enumerate all the possible lines. One approach is to observe that in each of the N dimensions, there are 5 possibilities for the values of the coordinates of three cells that form a line:
- increasing (1-2-3)
- decreasing (3-2-1)
- the same (1-1-1, 2-2-2, or 3-3-3)

For example, the cells in the top row of a 2-dimensional grid are *cell[1,1]*, *cell[1,2]*, and *cell[1,3]*. In the first dimension the coordinates are the same (1-1-1) and in the second dimension the coordinates are increasing (1-2-3).
So there are 5^N possible lines to consider, which is tractable for $N \leq 7$.
There is one exception, which is that if the coordinates of the three cells are the same in all N dimensions, then the three cells are in fact the same and don't form a real line.
This approach also double-counts the lines, because each line is considered in both directions; but that is easily fixed by halving the scores.

Representing the board as an N-dimensional array is likely to be awkward because N varies.
One alternative representation is a map (dictionary) where the keys are cell coordinates as arrays of length N and the values are the symbols.
Another possible representation is a flat array. This works best with 0-based coordinates and indexing. For example in three dimensions, *cell[a,b,c]* corresponds to index *9*a + 3*b + c*. More generally, coordinates can be converted to indexes using a `for`-loop that multiplies the coordinates by decreasing powers of 3 and adds them together.

Problem J: Standing Pins (NZPC 2011)

There are several cases where we can be certain in which cell a pin must stand:

1. If one end of a laid-down pin is outside the grid, then the pin must stand on its other end.
2. If one end of a laid-down pin is in a cell that contains a standing pin, then the laid-down pin must stand on its other end.
3. If a cell that does not contain a standing pin contains the end of only one laid-down pin, then that pin must stand in the cell.

These rules can be applied repeatedly to stand some of the pins in their correct cells.
When the rules can no longer be applied, it implies the following negations of their conditions hold:

1. All the laid-down pins have both ends inside the grid.
2. All the laid-down pins have both ends in cells that do not contain a standing pin.
3. Each cell that does not contain a standing pin ("free cell") contains at least two pin ends.

It can be shown that each such "free cell" must contain precisely two pin ends. Proof:
Construct a graph where the nodes are the free cells and the edges are the laid-down pins.
The number of nodes is equal to the number of edges because there must be a one-to-one correspondence between the free cells and laid-down pins.
The degree sum formula states that the sum of the degrees of the nodes is equal to twice the number of edges (which is equal to twice the number of nodes here).
The degrees of the nodes are all at least 2, from the third condition above.
If any of the degrees was greater than 2, then the sum of the degrees would be greater than twice

the number of nodes, contradicting the degree sum formula.
So the degrees must all be precisely 2.

This means that the laid-down pins must form cycles. Choosing where one pin stands will determine where all the other pins in the cycle stand.
If all the pins in the cycle have the same length, the choice doesn't make a difference to the solution.
But if some of the pins in the cycle have different lengths, then the choice will result in distinct solutions so the solution is not unique.

Problem K: Dictionary Attack (NZPC 2013)

This problem requires an exhaustive search, with some pruning to run within the time limit.

The simplest approach is to start from the first letter of the encrypted message and try all the possible decryptions of that letter, then for each option recursively try all the possible decryptions of the remaining letters.
When encountering a letter that has already been decrypted because it occurred earlier in the message, the same decryption must be used.
Otherwise, the possible decryptions are all the letters that haven't been used as decryptions in the current attempt.
When a full word has been decrypted, we can check that the result is in the dictionary; if not then the decryption is invalid.

This approach by itself is too slow. To run within the time limit, one approach for pruning the search is to also check the partially decrypted prefix of the current word; if it's not a prefix of any dictionary word, then the decryption is invalid.
It is possible to efficiently check if a string is a prefix of any dictionary word by building a set with all the prefixes.

Problem L: Frogs (NZPC 2015)

This problem can be solved using dynamic programming or as a shortest path problem.

The dynamic programming approach is to define *DP[i][k]* to be the length of the shortest hop in the best path from the start (lily pad 0) to lily pad *i* using *k* hops.
The recurrence relation is
$DP[i][k] = max\{ min(DP[j][k-1], dist(j, i)) \text{ for } j = 0..P-1 \text{ where } dist(j, i) \le D \}$
Here *dist(j, i)* is the distance between lily pads *j* and *i*, and *min(DP[j][k − 1], dist(j, i))* gives the length of the shortest hop in the path from the start to lily pad *i* using *k* hops where the second-to-last lily pad is *j*.
The base cases are $DP[0][0] = +\infty$ and $DP[i][0] = -\infty$ for $i \neq 0$.
Setting *DP[0][0]* to +∞ marks the empty path to lily pad 0 as valid; the value +∞ will be absorbed in the shortest hop length calculation, since $min(+\infty, dist(0, i)) = dist(0, i)$.
Setting *DP[i][0]* to −∞ for $i \neq 0$ marks those paths as invalid, since −∞ is worse than any possible shortest hop length.
−∞ should also be used as the maximum of the empty set in the calculation of *DP[i][k]* using the recurrence relation if there are no values of *j* such that $dist(j, i) \le D$.
The minimum number of hops to the end (lily pad 1) is the smallest *k* such that $DP[1][k] \neq -\infty$, and the length of the shortest hop in that path is *DP[1][k]*.

Another approach is to use a breadth-first search (BFS) to find the path with the minimum number of hops. Nodes correspond to lily pads, and two nodes are adjacent if the distance between their lily pads is at most *D*.
To determine the best possible shortest hop length, the BFS can be modified to also record the shortest hop on the path to each node. If the BFS finds multiple paths with the minimum number of hops to some node, then the best of the shortest hop lengths should be recorded.

Problem M: Stars (NZPC 2009)

Knowing which stars correspond to two points of a constellation is enough to determine the location, rotation, and scale of the occurrence.
Specifically, the distance between the two stars gives the scale (relative to the distance between the two corresponding points in the constellation); the bearing from the first star to the second star gives the rotation (relative to the bearing of the corresponding points); and the position of either star gives the location of the occurrence.
The location, rotation and scale of the occurrence can be used to compute the required positions for the remaining stars in the occurrence. If stars exist at those positions, then the occurrence is valid.
So for each constellation, one can iterate over all pairs of stars and check if an occurrence of the constellation exists at the location, rotation, and scale determined by having them as the first two stars of the constellation.

To efficiently determine if a star exists at a position, the stars can be stored in a hashtable with their coordinates as the key.
One approach for applying the transformation (translation, rotation, and scaling) is to use trigonometry and floating-point arithmetic. This works, but requires floating-point precision errors to be accounted for.
Another approach is to use vector algebra, which can be done without any floating-point arithmetic:

Let c_1 and c_2 be the positions of the first two points of the constellation and let s_1 and s_2 be the positions of the corresponding stars.
Define two coordinate systems, the *constellation coordinate system* and the *star coordinate system*.
The constellation coordinate system has its origin (O_c) at c_1, the first base vector ($\boldsymbol{A_c}$) is $c_2 - c_1$, and the second base vector ($\boldsymbol{B_c}$) is $\boldsymbol{A_c}$ rotated 90° anticlockwise (the vector *(x, y)* rotated 90° anticlockwise is *(-y, x)*).
Similarly, the star coordinate system has its origin (O_s) at s_1, the first base vector ($\boldsymbol{A_s}$) is $s_2 - s_1$, and the second base vector ($\boldsymbol{B_s}$) is $\boldsymbol{A_s}$ rotated 90° anticlockwise.
A constellation point can be transformed into a star position by resolving it into components in the constellation coordinate system, then applying those components in the star coordinate system.
Specifically, let c be the position of a constellation point.
We can find two real numbers a and b such that $c = O_c + a\,\boldsymbol{A_c} + b\,\boldsymbol{B_c}$.
Then the position of the corresponding star is $s = O_s + a\,\boldsymbol{A_s} + b\,\boldsymbol{B_s}$.
The numbers a and b can be computed by projecting $c - O_c$ onto $\boldsymbol{A_c}$ and onto $\boldsymbol{B_c}$.
To be precise, a must be $((c - O_c) \cdot \boldsymbol{A_c}) / |\boldsymbol{A_c}|^2$ and b must be $((c - O_c) \cdot \boldsymbol{B_c}) / |\boldsymbol{B_c}|^2$, where $|\ldots|$ denotes the magnitude of a vector and $\cdot$ denotes the vector dot product.
To verify that this is the case, observe that the component of $a\,\boldsymbol{A_c}$ in the direction of $\boldsymbol{A_c}$ is $a\,|\boldsymbol{A_c}|$, and that

$a\,|\boldsymbol{A_c}| = (((c - O_c) \cdot \boldsymbol{A_c}) / |\boldsymbol{A_c}|^2)\,|\boldsymbol{A_c}| = ((c - O_c) \cdot \boldsymbol{A_c}) / |\boldsymbol{A_c}|$

The dot product rule states that $\boldsymbol{v} \cdot \boldsymbol{w} = |\boldsymbol{v}|\,|\boldsymbol{w}|\,\cos(\theta)$, where $\boldsymbol{v}$ and $\boldsymbol{w}$ are vectors and θ is the angle between them.
Setting $\boldsymbol{v} = c - O_c$ and $\boldsymbol{w} = \boldsymbol{A_c}$ gives $((c - O_c) \cdot \boldsymbol{A_c}) / |\boldsymbol{A_c}| = |c - O_c|\,\cos(\theta)$, which is the component of $c - O_c$ in the direction of $\boldsymbol{A_c}$ as required.
Substituting a and b into the formula for s gives the required position of the star.
Calculating the magnitude of a vector normally requires taking a square root, but the expressions for a and b use the magnitude squared which is an integer.
As written, floating-point arithmetic is still required for the division, but with a little algebraic manipulation the expressions for the X and Y components of s can be rewritten as rational expressions, with a single division as the very final operation.
That final division operation can then be done using integer arithmetic; if there is a non-zero

remainder then the required star position is not at integer coordinates and the occurrence should be ruled out.

The time complexity of finding all the occurrences of one constellation is $O(N^2S)$. The problem statement does not specify the bound on S, but in the judge data it is at most 20 so this complexity is acceptable.
All that remains is to compute the brightness of each occurrence and remove duplicate occurrences.

Problem N: Congested Networks (NZPC 2010)

The congestion level between two nodes is equal to the maximum flow where each edge has capacity 1.
The maximum flow between a pair of nodes can be computed in $O(n^3)$ using the Ford-Fulkerson algorithm.
The bound on n is small enough that it is possible to run the maximum flow algorithm $O(n^2)$ times on all pairs of nodes, for an overall time complexity of $O(n^5)$.
A simple optimisation is to skip pairs of nodes if either of their degrees is less than or equal to the greatest flow found so far (because the flow between the pair could not possibly be greater). This optimisation was not required to run within the time limit.

Problem O: Jury Compromise (NZPC 2009)

First, consider the problem of finding a subset J with m elements that minimises $|D(J) - P(J)|$, without regard to the tie-breaking rules.
Minimising $|D(J) - P(J)|$ is equivalent to making the signed value $D(J) - P(J)$ as close to zero as possible.
Let u be the maximum possible value of $D(J) - P(J)$, which is $20m$.
Construct a table $DP[0..n][0..m][-u..u]$ where $DP[n'][m'][b]$ is true if there exists a subset J' of $\{1, ..., n'\}$ with m' elements such that $D(J') - P(J') = b$.
The base case is $DP[0][0][0] = true$ (corresponding to the empty set). The rest of the table can be computed using the following recurrence:
$DP[n'][m'][b] = true \text{ if } DP[n' - 1][m'][b] \text{ is true or } DP[n' - 1][m' - 1][b - (d_{n'} - p_{n'})] \text{ is true}$
The two cases correspond to excluding or including person n' in the subset.
The minimum possible value for $|D(J) - P(J)|$ is the smallest non-negative integer b for which $DP[n][m][b]$ is true or $DP[n][m][-b]$ is true.
The output specification asks for the candidates in J to be displayed. To recover the candidates in the subset corresponding to $DP[n][m][b]$, the table can be augmented to also record whether or not person n' was included when obtaining $DP[n'][m'][b]$. This is enough information to recover the full list of candidates in the subset using backtracking.

Now consider the tie-breaking rules, which are to maximise $D(J) + P(J)$ and if there are still ties to choose the jury that comes first in 'pseudo alphabetic' ordering.
When computing $DP[n'][m'][b]$, if it is possible both including and excluding person n', then the choice that maximises $D(J') + P(J')$ should be used to ensure the overall value of $D(J) + P(J)$ is maximised (the sum $D(J') + P(J')$ can be cached in the augmented table for efficiency).
If there are still ties, the subset that comes first in 'pseudo alphabetic' ordering should be chosen — the subsets will be distinct, so adding further people (who must have higher indices) to the jury will not affect the ordering.
The overall time complexity is $O(nm^2u)$ where $u = 2m$, because the size of the table is $O(nmu)$ and computing the value of a cell is $O(1)$ plus $O(m)$ to compare the 'pseudo alphabetic' ordering.
It is possible to reduce the time of the comparison from $O(m)$ to $O(1)$ by observing that in the problem statement, the 'pseudo alphabetic' ordering rule is equivalent to the following rule: "prefer subsets containing person 1 over subsets that don't; if there are still ties, prefer subsets containing person 2 over subsets that don't; etc." The second rule can be implemented efficiently as follows:

reverse the order in which the people are processed, so as to process the highest priority person (1) last, and at each point break ties by preferring to include person *n'* over excluding that person. When choosing the best overall jury, if both *DP[n][m][b]* and *DP[n][m][−b]* are true for the smallest possible *b*, then there are two juries with optimal *|D(J) − P(J)|* and the tie-breaking rules should be used to choose the best of the two.

Problem P: Mobile (NZPC 2015)

Consider an arm. Let *wA* be the total weight of the arm, let *wL* be the total weight of the left end, and let *wR* be the total weight of the right end.
The three variables are linked by the following equations:
wA = wL + wR
*wL*dL = wR*dR* (the balance equation)
If any one of *wA*, *wL*, or *wR* is known, then the other two can be computed using these equations. Specifically:
- If *wL* is known, then *wR = wL*dL / dR* (from the balance equation) and *wA = wL + wR* .
Similarly if *wR* is known.
- If *wA* is known, then solving the equations gives *wL = wA * dR / (dL+dR)* and *wR = wA * dL / (dL+dR)* .
So starting from one known weight, it is possible to compute the weights of all the other arms and weights using a flood-fill algorithm.

This leads to an approach for solving the problem:
Assume weight *m* has weight 1, and compute all the other weights using fractional arithmetic. This gives their weight relative to weight *m*.
Suppose the actual weight of weight *m* is *wM*.
The relative weights will need to be multiplied by *wM* to get their actual weight.
For the results to be integers, *wM* must be a multiple of the denominators of all the relative weights (as simplified fractions).
So *wM* must be a multiple of the lowest common multiple (LCM) of the denominators (Euclid's *Elements*, Book VII, Proposition 35).
Hence the minimum value of *wM* is *w* rounded up to the nearest multiple of the LCM.
Multiplying *wM* by the relative weight of the root arm gives the minimum total weight of the mobile.

There is another approach which does not require fractional arithmetic.
Each arm has a minimum possible weight such that all the weights suspended from it have integer weights.
The weight of each arm must be a multiple of its minimum weight (Euclid's *Elements*, Book VII, Proposition 20).
The minimum weight of an arm can be computed from the minimum weights of its two ends:
Let the minimum weights of two ends of an arm be *mL* and *mR*.
Suppose the actual weights of the two ends are *wL* (a multiple of *mL*) and *wR* (a multiple of *mR*).
Let *T = wL*dL = wR*dR* . *T* must be a multiple of *mL*dL* and of *mR*dR*, so the minimum value of *T* is *LCM(mL*dL, mR*dR)*.
Call this value *mT*. The corresponding value of *wL* is *mT/dL*, and the corresponding value of *wR* is *mT/dL*.
Hence the minimum weight of the arm is *mT/dL + mT/dR* .
The minimum weight of the entire mobile such that all the weights are integers can be computed this way.
The individual weights in this minimum-weight mobile can then be computed. Let *wM* be the weight of weight *m*.
If *wM* ≥ *w*, then we are done. If *wM* < *w*, then all the weights should be multiplied by *ceil(w / wM)*.