NZPC 2023: 30-point problem editorials

Problem I: Packing Cases

If the boxes are sorted into decreasing (or monotonically non-increasing) order, each box fits within its predecessor, except when the predecessor is exactly the same size. The key observation is thus that the number of units required, *n*, equals the count of the most-frequent edge length. This can be found either by iterating through the sorted list of sizes to find the longest run length or by using a dictionary to count occurrences.

You can then create *n* initially-empty lists, and iterate through the sorted box sizes, appending the sizes cyclically to the *n* lists, i.e. size[i] is appended to list[i % n]. This ensures that the list lengths differ by at most 1, which satisfies the criterion of minimising the number of boxes in the largest unit.

This problem, written by Troy Vasiga, Graeme Kemkes, Ian Munro, Gordon V. Cormack, was from a 2007 University of Waterloo contest.

Problem J: Subway

This is a classic Dijkstra shortest-path problem. See https://en.wikipedia.org/wiki/Dijkstra%27s_algorithm

- Construct a graph initially containing just home and school.
- Add a node for every station on every subway line.
- For each subway line
  - Add an edge connecting each station to its neighbours using the time taken for a train to cover the Euclidean distance between the stations as the edge weight.
- For each node
  - For each other node
    - Add an edge with weight being the time taken to walk the Euclidean distance between the nodes
- Use Dijkstra's algorithm to find the "distance" (actually time) from home to all other nodes.
- Print the time from home to school.

Note that you do need to add walking edges between stations on the same subway line, as some of the subway lines loop back on themselves.

If you're coding in C++ or Java you can get away with using the somewhat shorter all-pairs shortest path algorithm (Floyd-Warshall) but this is O(n^3) and isn't quite fast enough in Python, given the 1-second timeout for the problem.

This problem was from a 2001 University of Waterloo contest, author(s) unknown.

Problem K: Car Ferry

This problem can be solved by the somewhat non-obvious strategy of greedily taking full loads from the end of the queue, working backwards. A proof of sorts is as follows:

Let $\{a_i \mid 1 <= i <= m\}$ be the arrival times of the *m* cars.

Define T_k to be the earliest possible return time to the dock if transporting just the first k cars. T_k must be monotonically non-decreasing, as you can't possibly take less time by adding more cars to the queue.

Suppose the last crossing takes r cars. Then the earliest it can leave is $max(T_{m-r}, a_m)$,

Thus $T_m = max(T_{m-r}, a_m) + 2t$.

This is minimised by minimising T_{m-r}, which, since T is monotonically non-decreasing, is achieved by maximising r, i.e. by taking a full load of n cars on the last trip.

The same logic applies for minimising T_{m-n} and so on, recursing backwards down the queue.

T_m can be computed recursively using the recurrence:

$$\begin{aligned} T_i &= a_i + 2t && \text{if } i <= n \\ &= max(T_{i-n}, a_i) + 2t && \text{otherwise} \end{aligned}$$

Note that the the required answer is $T_m – t$, since we need the time of offloading the last cars. The number of trips is just $ceil(m / n)$.

Alternatively you can loop through the queue of cars from the start, taking $mod(m, n)$ cars on the first load (unless this is zero) and n on all subsequent loads.

This problem was from a 2003 University of Waterloo contest, author(s) unknown.

Problem L: Molecules 1

The first step is to compute a grid of unresolved valences, where H, O, N, C are 1, 2, 3 and 4 respectively. For example, the first sample input would be represented as

```
1 2 1 0
3 4 2 1
2 2 0 0
```

We *solve* a particular cell of valence *i* by setting it to zero and decrementing a subset of size *i* of its non-zero neighbours, if there is such a subset. If not, the cell cannot be solved.

The grid can be solved row-wise, top to bottom left to right, solving each cell in turn and recursing with a depth-first search to solve the rest of the grid. With this top-down left-to-right process, all cells above and to the left of the current cell are zero, so:

- If the residual valence of the current cell is greater than 2, the grid has no solution.
- If the valence of the current cell is 2, the only possible solution is to decrement both the cells to the right and below if these cells exist and are non zero. Then recurse on the new grid.
- If the valence is 1 we first try to decrement the cell to the right and recurse. If that fails we try to decrement the cell below and recurse. If that also fails, the grid has no solution.
- A cell of zero is trivially solved and we move to the next cell.

Pseudocode, where the *next* function moves to the next cell in top-to-bottom left to right order, and where (row, col) is the current cell, is:

```
function is_valid(grid, (row, col)):
     if (row, col) is bottom right cell:
         return grid[row, col] == 0
     if grid[row, col] > 2:
         return False
     if grid[row, col] == 0:
         return is_valid(grid, next(row, col))

     new_grid = clone(grid)
     new_grid[row, col] = 0
     if grid[row, col] == 2:
         if neighbours below and to right exist and are non-zero:
             new_grid[row + 1, col] = new_grid[row, col + 1] = 0
             return is_valid(new_grid, next(row, col))
     if grid[row, col] == 1:
         if neighbour to right exists and is non-zero:
             new_grid[row, col + 1] = 0
             if is_valid(new_grid, next(row, col)):
                 return True
         if neighbour below exists and is non-zero:
             new_grid[row + 1, col] = 0
             return is_valid(new_grid, next(row, col))
     return False
```

A backtracking approach could be used to avoid cloning the grid repeatedly but with such a small grid isn't necessary.

Although this might appear to be O(2^n) where n is the number of cells, it actually runs in only a few tens of msecs in Python on 5 x 5 grids. The reason is that the recursion call-tree branches (i.e. has two children) only when a cell has a residual valence of 1. It's not possible to encounter a valence of 1 at all levels, since we are zeroing either the cell to the right or the cell below at each step. The measured performance seems to be approximately O($2^{n/2}$).

There is of course a much more efficient algorithm: see problem M, which has a 20 x 20 grid.

This problem was from a 2002 NZ Programming Contest, author(s) unknown.