NZPC 2024: 30-point and 100-point problem editorials

version 1

Problem I: Watersheds

This problem was from the Google Code Jam 2009 Qualification Round (authors: Andrew Gove, Xiaomin Chen, Igor Naverniouk).

For an editorial, see the analysis from that contest.

It should be noted that the algorithm proposed in that analysis has a worst-case time complexity of $O(N^4)$ for an $N \times N$ grid. Although it passes the test data (provided by Code Jam), it is possible to construct tests that would cause it to exceed the time limit.

A simple fix to the algorithm is to keep track of all cells visited while searching for sinks: any search that enters an already-visited cell can be terminated immediately, since the corresponding basin has already been determined. With this change, the algorithm becomes $O(N^2)$.

Problem J: Snaggle

Snaggle is derived from a problem written by Gordon V. Cormack for use in a University of Waterloo contest, 2007.

It is an expression evaluation exercise, probably involving recursion, although you can also use a stack-based evaluator.

A tokeniser that breaks the input string into its lexical elements is a good start. A regular expression to do this, using a function like Python's `re.findall` is:

```
TOKEN_RE = r'[01]\.[0-9]+|-?[0-9]+|\(|\)'
```

Given a token stream, a recursive-descent parser/evaluator is of the following form:

```
function evaluate(token_stream)
    token = next(token_stream)
    if token is an integer
        return int(token)
    else
        assert token is '('
        f = float(next(token_stream))
        e1 = evaluate(token_stream)
        e2 = evaluate(token_stream)
        assert next(token_stream) == ')'
        return f * (e1 + e2) + (1 - f) * (e1 - e2)
```

Problem K: Microspikes

This is a problem from NZPC 2012. It is a simulation problem that requires two stages.

- Stage 1: compute the times and magnitudes of all changes in power usage.
- Stage 2: process the sequence of changes in power usage in time order, keeping track of the current total power usage and looking for spikes that satisfy the definition.

Stage 1 requires a dictionary/map recording the time at which each appliance last changed its power level and a data structure to store the power-changes as a function of time. An array of power-change indexed by time t for t in the range 0 to T inclusive is the natural choice for the latter, given that T is at most 100,000.

Stage 1

```
Set all appliance times to 0
Set power_changes[t] = 0 for all t in range 0 to T inclusive
for each power change line (appliance, delta_t, delta_p):
    t = appliance_time[appliance] + delta_t
    appliance_time[appliance] = t
    power_changes[t] += delta_p
```

Stage 2

```
power_use = 0
spike_count = 0
for t in 0 .. T inclusive:
    power_use += power_changes[t]
    if power_use just crossed threshold:
        Record possible spike start time
    else if power_use just dropped below threshold:
        if duration <= maximum_duration:
            spike_count += 1
print spike_count
```

An alternative way to record power changes as a function of time is a dictionary/map from time to power change. The dictionary must be sorted by time prior to stage 2. This is a more-general solution, which can handle arbitrarily large values of T with events that are sparsely distributed in time. It introduces an extra factor of *log N* for sorting, but sorting time is dwarfed by I/O time in this problem. Use of an array of discrete times is essentially just a special case that uses a bin sort. The two alternatives are sometimes referred to as "event-based simulation" and "time-based simulation".

Problem L: Crossing the Road

This problem was from Google Code Jam 2009 Round 1A (authors: Mohamed Eldawy and Bartholomew Furrow).

For an editorial, see the analysis from that contest.

Problem M: Music Festivals

This problem was from the Serbian High-School Informatics Olympiad national competition 2022/2023 (author: Pavle Martinovic).

It is a dynamic programming problem that requires some additional insight to make the solution fast enough.

Consider the following subproblem: Can you travel from $(1, 1)$ to (i, j) and visit exactly c festivals along the way? ($1 \le i \le N$; $1 \le j \le M$; $0 \le c \le K$).

If there is no festival at (i, j), then the answer is "yes" if and only if:

- $i > 1$ and there is a path from $(1, 1)$ to $(i-1, j)$ that visits exactly c festivals; or
- $j > 1$ and there is a path from $(1, 1)$ to $(i, j-1)$ that visits exactly c festivals.

If there is a festival at (i, j), then it's almost the same except with $c-1$ instead of c.

The base case is $(i=1, j=1)$, which is possible with $c=0$ if there is no festival at $(1, 1)$, and with $c=1$ if there is a festival at $(1, 1)$.

This leads to an $O(NMK)$ solution; but that's too slow.

We can improve this to *O(NM)*. Consider the path from *(1, 1)* to *(i, j)* for a fixed *(i, j)*. If there is a path that visits exactly c_1 festivals, and there is another path that visits exactly c_2 festivals, then for each integer *c* in the range c_1 to c_2 there must exist a path that visits exactly *c* festivals (in other words, the values of *c* that are possible are consecutive).

Proof:
Consider the paths that visit c_1 and c_2 festivals. We can gradually deform one path to the other, such that at each step we have a valid path and the number of visited festivals changes by at most 1.
In detail: Represent a path as a sequence of R's and D's (right / down). All paths to *(i, j)* must have the same number of R's and D's. The deform operation is to replace an adjacent RD with DR, or vice versa. This produces a valid path which is almost the same but goes through a different cell. The change to the number of festivals visited is -1, 0, or +1 depending on whether the old cell and the new cell contain festivals.
We are guaranteed to be able to deform one path to the other because any permutation of a sequence can be obtained using a series of swaps. And along the way we must have produced (at least) one valid path for each *c* in the range c_1 to c_2.

So for each *(i, j)*, instead of having to keep track individually of the values of *c* for which a path exists, we just need to keep track of the range values (minimum and maximum). To compute the range for *(i, j)*, we merge the ranges for *(i-1, j)* and *(i, j-1)* — meaning we take the minimum of the two minimums, and the maximum of the two maximums. And then, if there is a festival at *(i, j)*, we just need to add 1 to both the minimum and the maximum.

The output is 1 iff *K* is within the range for *(N, M)*.

Problem N: Hotter Colder

This problem was from the Waterloo Programming Contest, 27 January, 2001.

This is a computational geometry problem that ends up being half-plane intersection. The input is small, so there is no need for a fast algorithm.

The first part is to work out how to transform the information in the input into the geometric region where the hidden object might be. If the announcement is "Same", it means the object is equidistant from the current position and the previous position; so it must lie on their perpendicular bisector. This is an infinitely thin line, so its area is 0; hence the output will be `0.00` from that point onward. If the announcement is "Hotter", it means the object is on the same side of the perpendicular bisector as the current position (a region known as a half-plane), and similarly for "Colder". So after a series of Hotter/Colder announcements, the area to output is the area of the intersection of all the half-planes corresponding to the input lines processed so far (intersected with the square representing the room).

The perpendicular bisector can be computed easily (compute the midpoint, and rotate the vector between the two points by 90° to get the direction of the bisector).

There are a number of ways to compute the intersection of the half-planes. For an *O(N log(N))* algorithm, see cp-algorithms.

A simpler $O(N^3)$ algorithm is to compute the pairwise intersections of all the perpendicular bisectors and divide the room into strips at the *x* coordinates of the intersections. This divides the area where the object could be into trapeziums. To calculate the area of each trapezium, compute the *y* values of the perpendicular bisectors at the two *x* values, and then the side lengths are the difference between the *y* values of the perpendicular bisectors that bound their half-plane from above and the *y* values of the perpendicular bisectors that bound their half-plane from below.

Special handling is required for perpendicular bisectors that are parallel to the *y* axis.

Problem O: A Digging Problem

This problem was from Google Code Jam 2009 Round 2 (author: Mohamed Eldawy).

For an editorial, see the analysis from that contest.

Problem P: OR Game

This problem was from the Serbian High-School Informatics Olympiad 2022/2023 contest (author: Aleksa Milisavljevic).

This is a dynamic programming problem. It requires some observations about the nature of the game to get started, and some more insights to get an algorithm which is fast enough.

The first observation is that each operation reduces the size of the array by *K-1*, so the the final array's length will be *1 + (N-1) % (K-1)*. (It is tempting to write *N % (K-1)*, but this simpler formula produces the wrong result when *N* is a multiple of *K-1*.)

The second observation is that, because bitwise OR is associative, each element in the final array is the bitwise OR of some subarray of the original array with length *1 + (K-1)*i* for some integer *i ≥ 0*, and these subarrays are an exact cover of the original array. This can be shown by induction, or by working backwards from the final array: each element of the final array is initially a subarray of length 1, and "expanding" 1 element to *K* elements adds *K-1* elements to the subarray. So the length of each subarray must be of the form *1 + [some multiple of (K-1)]*, and the subarrays are an exact cover (they don't overlap and there are no gaps).

So solving the problem is a matter of partitioning the original array into *1 + (N-1) % (K-1)* subarrays, such that each subarray has a length of the form *1 + (K-1)*i* for some *i ≥ 0*, and such that the sum of the bitwise OR of the subarrays is minimised.

This leads to an $O(N^2/K)$ dynamic programming solution (which is too slow):

Let *DP[m]* be the best score for reducing the first *m* elements to *1 + (m-1) % (K-1)* elements (*1 ≤ m ≤ N*), computed as follows:

- If we are reducing to 1 element, then there is only one option —
  DP[m] = bitwise_or(A[0:m])
  (the notation *A[i:j]* denotes the subarray of *A* from index *i* up to but not including *j*)
- Otherwise, we try all possible lengths for the last subarray:
  DP[m] = min*{ DP[m-w]* + *bitwise_or(A[m-w:m])* for *w* from *1* to *m-1* in steps of *K-1 }*

The time complexity of this approach is *O(N × N/(K-1))*, because for each value of *m* there are about *N/(K-1)* values of *w* that need to be tested. (The bitwise OR of arbitrary ranges can be computed in *O(1)* time, see the very end.)

We can make this algorithm faster by observing that as *w* increases (for a fixed *m*), the value of *bitwise_or(A[m-w:m])* can take on at most 32 distinct values, because once a bit becomes 1 it must stay 1.

So instead of checking each value of *w*, we can group the values of *w* into at most 32 ranges such that in each range the value of *bitwise_or(A[m-w:m])* is the same, and for each range we want the minimum value of *DP[m-w]*.

We can efficiently compute the groups of *w* (and the corresponding minimum values of *DP[m-w]*) for some *m* based on the groups for *m-(K-1)*: the groups will be mostly the same, except that there will be a new group for *w=1*, and some groups might get merged after accounting for the effect of *A[m-(K-1):m]* on the bitwise OR..

All that remains is to efficiently compute *bitwise_or(A[m-(K-1):m])*. There are a number of ways to do this efficiently:

- A sparse table can be used to compute them in *O(1)*, with *O(N*log(K))* precomputation.
- The bit at position *p* of *bitwise_or(A[i:j])* is 1 iff there exists an element in *A[i:j]* that has a 1 at position *p*. We can check that in *O(1)* by precomputing a cumulative sum array for each bit position. (Aside: This corresponds to the "rank" operation of a rank/select data structure.)

The overall time complexity will be *O(N*log(K) + N*32)* or *O(N*32)* depending on the implementation choice.