UAPC 2025

Solution Sketches

Division 2 Exclusive Problems

Testing LEDs - Noah Gergel

D2 Solves: 33

Fastest Solve:
nevgoonies 0:07:12

Loop over the input and keep track of the smallest time value where the LED is off.

It'll also help to have a flag denoting whether you saw the LED ever being off or not.

Search Wizard - Noah Gergel

D2 Solves: 34

Fastest Solve:
Linkedin Lists 0:08:53

Initialize a global count to 0. For every word x in S, loop over the characters in x and record the number of times W appears as a substring.

That is, for the indices [0, len(x) - len(W)], check if x.substr(i, len(W)) == W, and increase the count if so.

Problems Common to Both Divisions

Vandalism - Ian DeHaan

D1 Solves: 21

D2 Solves: 37

Fastest Solve:
vvv Colorado School of Mines vvv 0:00:44

Loop through the letters in "UAPC" and output the ones not present in the input.

Sort of Sort - Grayden Price

D1 Solves: 20

D2 Solves: 32

Fastest Solve:
Calgary School of Mines 0:02:56

By definition if a_i is larger than everything before it, it will be in the sort of sorted list. This leads to the following algorithm:

- Loop through the list.
  - For each a_i loop over all a_j before it, if a_i is greater than or equal to each a_j then it will be in the sort of sorted list.

This should time out, to speed up the solution realize that knowing only the largest a_j is sufficient for checking a_i. This leads to the intended algorithm:

- Loop through the list while maintaining the largest a_j.
  - If a_i is greater than or equal to the largest a_j it will be in the sort of sorted list.

# of in the cold food of out hot eat the food - Ian DeHaan

D1 Solves: 15

D2 Solves: 17

Fastest Solve: Calgary School of Mines 0:13:05

The food needs H*T total energy to finish cooking.

Suppose we cook the food for X minutes.

If X <= H, the amount of energy the food gets is X^2/2.

If X > H, the amount of energy the food gets is H^2/2 + (X-H)*H.

Solve for X to find that if H^2/2 >= H*T, the answer is sqrt(2*H*T). Otherwise, the answer is T + H/2.

Island Exploration - Zac Friggstad

D1 Solves: 17

D2 Solves: 22

Fastest Solve:
vvv Colorado School of Mines vvv 0:08:23

Classic graph search problem. If you have never seen one before, here the basic approach.

- Maintain a set of land cells you know you can reach. Initially, you only know about the cell with **S**.
- When you first find a new cell that can be reached, also add it to a different set of cells you are to “expand” (i.e. want to check the 4 adjacent cells).

Main loop: while there are still cells to expand:

- Remove a cell from the set of cells to expand. Examine its neighbours.
- Each land neighbour that has not yet been reached should be added to both the set of cells you know you can reach and the set of cells you want to expand.

Tracing Laser Pointers
Joseph Meleshko

D1 Solves: 18

D2 Solves: 21

Fastest Solve:
or3 0:16:43

For each laser pointer, you can determine if and where it will intersect the x-axis in a couple of ways. Here is one:

- If the slope is not 0, solve for the x-intercept for the infinite line and see if this is less than or greater than the laser pointer's x-position to determine if the laser itself crosses the x-axis.
- Sort all lasers that hit the x-axis by the x-intercept. You need to remember the laser names as well. In python, you can sort the list of (x-intercept, laser-name) tuples and print out the laser names in this order.
- Use a fast sorting algorithm (i.e. not bubble sort, insertion sort, etc.). The built-in sorting algorithms for all supported programming languages are fast enough.

One Nail / One Hole
Parsa Zarezadeh

D1 Solves: 2

D2 Solves: 0

Fastest Solve: or3 2:36:58

The solution that comes to mind!

- Reduce the problem to a one-dimensional case by considering a straight line instead of a plane. We have n intervals of equal length, and our goal is to select points on the line such that each interval contains exactly one point.
- To solve this subproblem, we can use a greedy algorithm. Select the rightmost starting point among all intervals, place a point there, and remove all intervals that contain this point. Repeat this process until no intervals remain.
- Now, to solve the main problem. For each paper, determine its projection on both the x-axis and y-axis, which will form intervals. Solve the subproblem separately for the x-axis and the y-axis. As a result, each paper will have exactly one point on the x-axis and one on the y-axis, which together determine the coordinates of the nail for that paper. The running time of the solution will be $O(n \log n)$.

A solution that feels out of this world!

- Consider all coordinates *(x, y)* where *x - 1* is divisible by *L* and *y - 1* is divisible by *W*. It can be easily proven that each paper will have exactly one such point on it. Given the problem's constraint that all numbers are even, we can see that this point will not be on the paper's corners and will be unique for each paper.
- To solve the main problem, for each paper, find the unique point that satisfies the above condition in *O(1)* time (ensuring that no point is chosen twice). This results in an overall running time of *O(n)*.

Can you come up with a new and creative solution that runs in $O(n^2)$? Let us know!

Strange Light Switches
Noah Gergel

D1 Solves: 7

D2 Solves: 0

Fastest Solve:
vvv Colorado School of Mines vvv 1:02:34

It'll help reframe the problem as trying to zero runs of 1s, as every run can be handled independently of each other, with one exception. For odd-length runs, you can take the approach of zero-ing every other 1 first, and then zeroing the rest:

```
0111110 -> 0101010 -> 0000000
```

For even-length runs, you can do the same except when you get down to a run of length 2:

```
01111110 -> 01010110 -> 00000110
```

To deal with two 1s, you first extend it by 1 and then treat it like a run of length 3:

```
00110 -> 01110 -> 01010 -> 00000
```

This might not always be possible, if the run is initially just two 1s it needs two 0s before it to work.

Strange Light Switches
Noah Gergel

If an input is solvable though, it suffices to solve it greedily with zeroing odd-length runs first and then the even ones, possibly needing to circle back for the runs of length 2. From this, we also know that the only impossible cases are rotations of the pattern 011011... or just 111 itself as after the first step it is again some rotation of 011.

Coin Exchange - Answer

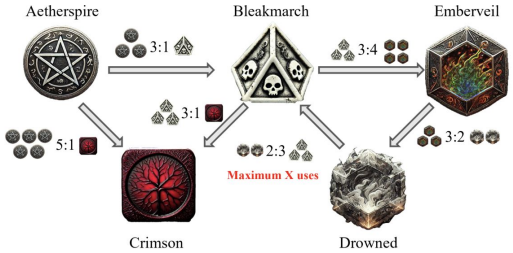
D1 Solves: 5

D2 Solves: 1

Fastest Solve: Alberta Amber 1:18:18

There are three paths to earn C:

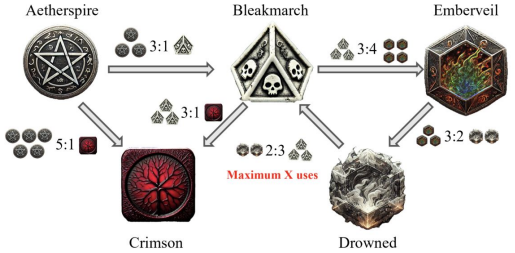
PATH 1. A->C;
PATH 2. A->B->C
PATH 3. A->B->E->D->B->C



Because A can be used directly or indirectly to obtain C, let's start by assuming A equals zero and start from converting B->C. Consider how to get the most C starting from initial B, D, E.

So, first convert all initial D and E into B, then see how many times you can carry out that conversion path. In PATH3, there's a "produce-and-use" strategy in which any leftover D is pointless. Therefore, we can simplify 3E ➔ 2D and 2D ➔ 3B into 3E ➔ 3B. When B ≥ 3, you can perform the cycle 3B ➔ 4E ➔ 3B. The number of such cycles depends on E and X. Every time you have 3 B and spend one X, you can keep your total B constant while gaining 1 E. In other words, for every 3 B, four uses of X will net you one additional B.

Coin Exchange - Answer



Going from A ➔ B ➔ C requires 9 A for 1 C, whereas A ➔ C directly needs only 5 A—so unless you still have some X available, advancing A through B is a net loss. If X is still left, compare whether it's more profitable to convert A into B (start another cycle of PATH 2) or just follow PATH 1 to produce C. Finally, handle any remaining conversions from B to C.

Another solution by Zac: first guesses how many times to convert A➔B—up to a maximum of 25—and converts any remaining A directly to C; then, under A=0, it maximizes E➔D and D➔B conversions, performs the 9B➔12E➔8D➔12B loop whenever B≥9 depending on X, and finally compares fully converting B➔C at once versus converting it once and then recursively solving, choosing the better option.

Nearest Nice Numbers
Noah Weninger

D1 Solves: 10

D2 Solves: 8

Fastest Solve: Calgary School of Mines 0:14:25

For an input value x_i, let N_i be such that N_i/D <= x_i < (N_i+1)/D.

Claim: in the optimal solution x_i will be rounded to one of these two fractions. Otherwise, say (for example) x_i was rounded to a value A/D with A < N_i. Then some other x_j must have been rounded up to >= (N_j+1)/D. But then rounding x_i to (A+1)/D and x_j to N_J/D instead is a better solution (easy to check).

So: initially set all x_i to N_i/D. Let K := D - sum_i N_i be the number of these values that should actually be rounded up to (N_i+1)/D. Using sorting, pick the K values that would result in the least increase (perhaps even a decrease) in the final answer.

Division 1 Exclusive Problems

Generator Dream - Jacob Skitsko

D1 Solves: 5

Fastest Solve: Calgary School of Mines 0:53:05

Recall $x_i = 2^{i-1}x \bmod p$, and $b_i = x_i \bmod 2$.

For some i let's consider x_i, and note $x_{i+1} = 2x_i \bmod p$. Notice x_{i+1} is $2x_i$ if $0 \le x_i < p/2$, and x_{i+1} is $2x_i - p$ if $p/2 \le x_i < p$.

Furthermore, b_i will be 0 exactly in the first case, and 1 exactly in the second case, so long as p is odd (the $p=2$ case is special but simple).

So the bits b_i tell us a range of possible values for each x_i, and by getting allowable ranges for each i from 1 to $\log p$ we should be able to narrow down the possible values for x.

Now the idea is to do a binary search for the value x, based on the fact that each bit b_i will cut the possible range of values of x in half.

We can consider another approach that's not binary search.

Again, notice x_{i+1} is $2x_i$ if $0 \le x_i < p/2$, and x_{i+1} is $2x_i - p$ if $p/2 \le x_i < p$.

Furthermore, b_i will be *0* exactly in the first case, and *1* exactly in the second case.

In this approach we'll progressively learn the *i* least significant bits (LSBs) of each x_i. Note the first LSB is handed to you by b_1, *and* we know all of the bits of *p*. Suppose we've found out the *i* LSBs of some x_i are $a_i a_{i-1} \ldots a_1$.

Then if $b_i = 0$ we know $2x_i = x_{i+1}$, and so the *i+1* LSBs of x_{i+1} are $a_i a_{i-1} \ldots a_1 0$. If $b_i = 1$, then we know $2x_i - p = x_{i+1}$, and we can again calculate the *i+1 LSBs* of x_{i+1} using $2x_i - p$ and a bit mask.

After iteratively calculating the LSBs for $i = 1, \ldots, \log p$ we will know all $\log p$ bits of $x_{\log p}$. We can then backtrack to calculate each of $x_{(\log p) - 1}, x_{(\log p) - 2}, \ldots, x_1$ and the secret *x*.

Underspecified Ultrametrics
Zac Friggstad

D1 Solves: 2

Fastest Solve: or3 3:03:33

Thought process

1) Observe distances d() are ultrametric if and only if for any triple of points u,v,w the maximum value of d(u,v),d(u,w),d(v,w) appears at least twice (i.e. the triangle has at least 2 longest edges).

2) Extend this: distances d() form an ultrametric if and only if for every pair u,v there is not a u ➔ v path using only edges with distance < d(u,v). [easy proof]

3) Finally, if only some distances are given then they can be extended to an ultrametric if and only for every ***given*** distance d(u,v), there is no u ➔ v path of ***given*** edges with strictly smaller distance.

Proof [harder direction]: Set the missing edge distances d(x,y) = minimum edge cost D such that there is an x ➔ y path using edges with cost <= D, or just the max given distance if x, y are not connected.

Algorithm:

Group the given edges by their value, i.e. (u,v) and (w,x) are in the same group if d(u,v) = d(w,x).

- Build a graph over the points, initially with no edges.
- For each group in increasing order of distance:
  - For each edge (u,v) in the group, check that u,v are not in the same connected component (otherwise, the answer is **NO**).
  - After this check, add all edges of the group to the graph.

Using a union/find data structure, this can be implemented to run in O(V log V + E log E) where V = # points and E = # given distances.

Note, this is very similar to Kruskal's MST algorithm except we are batch-processing edges based on their value before adding them.

Separating Enemies - Ian DeHaan

D1 Solves: 3

Fastest Solve:
Calgary School of Mines
1:43:39

Use dynamic programming.

dp(i) = minimum cost to destroy road between i'th and (i+1)'th house AND separate all enemy pairs (s, t) with s, t <= i.

Of all enemy pairs (s, t) with s, t <= i, let (s', t') be the one with maximum s'.

We must destroy some road between s' and t', and doing so will destroy all intervals ending before i and after s'.

dp(i) = c_i + min_{s' <= j < t'} dp(j)

Of all enemy pairs (s, t) with s, t <= i, let (s', t') be the one with maximum s'.

We must destroy some road between s' and t', and doing so will destroy all intervals ending before i and after s'.

dp(i) = c_i + min_{s' <= j < t'} dp(j)

We can find (s', t') by maintaining the latest starting point ending before our position.

We can query min_{s' <= j < t'} dp(j) in log(n) time with a segment tree.

Final Runtime: O(n log n)