UAPC 2026

Solution Sketches

High School Exclusive Problems

HS Solves: 13

Fastest Solve:
// TODO Team Name - 0:00:47

Ready, Set, Go! - Zac Friggstad

Simply output the difference of the two numbers.

Just a Bit - Zac Friggstad

HS Solves: 13

Fastest Solve:
// TODO Team Name - 0:02:47

Iterate through all characters, counting how many 0s and 1s you see. For example:

```
s = input()
zeros = ones = 0
for c in s:
        if c == ‘0’: zeros += 1
        else: ones += 1
print(zeros, ones)
```

Shared Problems

D1 Solves: 11
D2 Solves: 21
HS Solves: 10

Fastest Solve:
Water Black - 0:03:21

Router - Zac Friggstad

Maintain a minimum x, maximum x, minimum y, and maximum y seen so far. Start at (x,y) = (0,0) and update as per the instructions.

Final width is 40 + (max_x - min_x)

Final height is 40 + (max_y - min_y)

Source to Sink - Grayden Price

D2 Solves: 23
HS Solves: 12

Fastest Solve:
Aukland School of Mines - 0:11:48

We can drain A water at a time. Fully draining A water takes 4 valve operations:
- Open valve let water in the pipe
- Close valve
- Open other valve to drain water from the pipe
- Close that valve

Since the problem asks us minimize the valve operations to fully drain the water into the reservoir, we don't have to do the final valve close in the last cycle.

So the answer is 4*ceil(w*h/a) - 1

Spell it Out - Noah Gergel

D1 Solves: 11
D2 Solves: 28
HS Solves: 12

Fastest Solve:
make no mistakes - 0:07:57

This is is mostly about paying careful attention to detail.
My solution has these arrays:

- a = [“zero”, “one”, “two”, ..., “nine”, “ten”, “eleven”, ..., “nineteen”]
- b = [“”, “”, “twenty”, “thirty”, ..., “ninety”]

This limits the number of cases I have to write:

- if n < 20: print(a[n])
- elif n%10 != 0: print(b[n//10] + “-” + a[n%10])
- else: print(b[n//10])

Periodic Encryption - Grayden Price

D1 Solves: 10
D2 Solves: 22
HS Solves: 9

Fastest Solve:
Water Black - 0:05:35

Small enough input that you can just do it directly. Be careful about wraparound logic.

D1 Solves: 6
D2 Solves: 18
HS Solved: 6

Fastest Solve:
Water Black - 0:19:30

Daily Play Lists - Answer

Small enough that you can just do it directly. I used vectors (Python lists) and just pushed to the back of them and erased certain entries. If L denotes the maximum length of a playlist (so L = 100 in this question) then each move takes O(L) time to process and each play takes O(L*C) time to process where C is the number of distinct songs we are interested in.

A harder version of this problem would not have playlist bounds nor bounds on the number of distinct songs. We could use segment trees to support each move in O(log n) time where n is the total number of occurrences of each song and use unordered maps (counting the frequency of each song) to support each play operation in O(C) time.

Swamp Paths - Noah Gergel

D2 Solves: 9
HS Solves: 1

Fastest Solve:
// TODO Team Name - 1:54:15

For each path that is queried, if the path status changed (ADD or REMOVE) in an earlier step we know if the path is safe or not.

But if a queried path has not had its status changed yet, we need to look ahead to see if its status changed some time in the future. If so, we know its current state (e.g. if the first change in the future is to ADD it, we know it was not safe during an earlier query).

The only “unsure” responses are if the status of a queried path never changed at any point in the log.

To avoid looking ahead each time you see query for a path whose status has not yet been logged, you can first compute the first time each path’s status changed and look ahead to that each time a path is queried before its status has changed.

D2 Solves: 15

Fastest Solve:
Daffodills - 0:47:41

Circular Walk
Grayden Price

Output YES if gcd(n,k) divides b, otherwise NO.

Why?

- Reaching 0 from b means for some integer S (# of steps we take in some direction) we have b + k*S is a multiple of n, i.e. b + k*S = d*N for an integer d. Clearly gcd(n,k) divides both k*S and d*N so it must divide b.

- Conversely, from number theory we know there are integers A,B such that A*n + B*k = gcd(n,k). So if gcd(n,k) divides b, then multiplying A,B by b/gcd(n,k) gives integers A', B' with A'*n + B'*k = b, as required. Of course, you could just guess that the gcd() works too.

Input is small enough that if you did not know the fast way to compute gcds (via Euclid's algorithm), you could still compute it by factoring n and k (in O(sqrt n) time) and group their common primes together.

Icarus - Noah Gegel

D1 Solves: 6
D2 Solves: 11
HS Solves: 1

Fastest Solve:
buy YES on UAlberta at https://manifold.markets/ - 0:33:36

There are quite a few approaches that could work

2 “exact” solutions

- Consider the line perpendicular to Icarus’ path that passes through the sun. This is the point on the path that gets closest to the centre of the sun. Check if this point is too close.

- Alternatively, parameterize the line by a value t so the coordinates of the line can be written as (x(t), y(t)) as t ranges over the reals. If (x’,y’) denotes the sun’s centre, we need to see if the line punctures the circle with centre (x’,y’) and radius D+R. That is, solve for ||(x(t),y(t)) - (x’,y’)||^2 = (D+R)^2. If you expand this formula, you get a quadratic in t. It has a solution (i.e. Icarus gets too close) if and only if the discriminant of the quadratic is >= 0.

Also 2 approximate solutions that are “good enough” given the guarantees that the answer is robust (i.e. would not change if the radius changes a bit)

- Sweep: start with the “leftmost” x-value of the danger zone: sun_x - D - R. Increase x by a small amount: each time compute the y-coordinate of the flight path and check if this is too close to the center. Step when the x-value exceeds sun_x + D + R. Make sure to treat a vertical flight path as a separate case.

- Ternary search. Parameterize the line like in the last slide. If Icarus gets too close to the sun, it will be between t = -1e6 and t = 1e6. Then use a ternary search, which is like a binary search except for when the function “goes down and then goes up” (in this case, the “distance to the sun” function).

  Or if you could figure out if some t is “getting closer” or “moving away” from the sun as t increases, then you could do a normal binary search.sun’s radius changes a bit)

Off or On - Parsa Zarazedah

D1 Solves: 2
D2 Solves: 1

Fastest Solve:
On or Off - 0:46:45

Try spreading all N+M batteries as evenly as possible into N-1 groups. By the pigeonhole principle, two good batteries land in the same group. So if we test all pairs of batteries that lie in a common group then we will eventually find a good pair of batteries.

Why is this optimal? This is Turan’s theorem for the minimum size of a graph avoiding cliques of a certain size but we hoped people would be able to guess this strategy and try it out.

Locomotive Lunacy
Jacob Skitsko

D2 Solves: 5

Fastest Solve:
>_< nyaaaaaaa~ :3 - 2:21:20

If there was no ticket, this would be a standard minimum-cost path problem which can be solved using Dijkstra's algorithm in O(M log N) time.

But there is a ticket. Consider creating two copies of the graph. Then for each directed edge (u,v) of cost c, add a directed edge from the copy of u in the first graph to the copy of v in the second graph having cost floor(c/2).

Essentially, the first graph represents the state of "I am at location v and I still have a coupon" and the second graph represents the state "I am at location v but I no longer have the coupon".

Don't forget to add 0-cost edges from a state in the first graph to its copy in the 2nd (representing not using the ticket, which is important for a corner case)

Conference Seating
Zac Friggstad

D1 Solves: 3
D2 Solves: 1
HS Solves: 0

Fastest Solve:
Water Black - 1:25:04

A greedy algorithm can solve this, though we left the bounds small enough so that you could use dynamic programming (more annoying to code, but easier to see).

Observations leading to greedy:

- There is no vertical line such that some chairs go L->R across the line and others go R->L across the line. Note this is not necessarily true for an arbitrary n x m grid, we use the fact there are 2 rows.
- You can also imagine each chair travels only horizontally until, perhaps, its final vertical step.

Together, this uniquely defines a value for row across each vertical line indicating how many chairs will cross the line in one direction. Add this to our total.

But we are not quite done.

This tells us exactly how many chairs will cross over each spot in their horizontal journey. For each cell that requires a chair, if some chair will pass over that spot then use it. Otherwise, use a chair from the other row in this column (adding 1 to the total).

Running time: O(n)

The problem was also small enough to permit other solutions (i.e. dynamic programming or min-cost max flow). The one I described was likely easiest to code.

Division 1 Exclusive Problems

D1 Solves: 3

Fastest Solve:
Water Black - 0:50:41

Brass Section - Zac Friggstad

This is actually classic Horn SAT.

- If every group has a positive value, then everyone can play loud.
- Otherwise, pick a group that only has a negative value. This musician forced to play quiet. Remove all positive occurrences of this musician from other groups.
- If any group with only positives becomes empty, it is impossible. Otherwise, continue with another group that only has a negative value.

Since this only forces negatives when necessary, it also outputs the (unique) optimal solution when there is a solution.

Running time: linear in the entire input size if you are careful in your implementation

Metro Lines - Zac Friggstad

D1 Solves: 5

Fastest Solve:
Water Black - 0:12:43

Sweep from 0 to n and store all metro lines spanning the current set in a min heap. This way, we can look at the top of the heap to find the cheapest line to take each step.

When we come to the start of a new metro line, add it to the heap. We can't necessarily pop a metro line from the heap when we reach the end of it (at least not with how C++/Python heaps work). Instead, we can be lazy about it: whenever we look at the top of a heap we can check if that train is still active. If not, pop it from the heap.

In C++, you could keep an ordered set of pairs {c[i], i} and remove an entry when you scan past the end of a line (i.e. no "lazy" popping required).

Total running time: O(N log N).

Bridge Removal - Ian DeHaan

D1 Solves: 2

Fastest Solve:
Water Black - 0:26:20

If L denotes the number of leaves, we need at least ceil(L/2) extra edges or else some leaf isn't even touched by a new edge. We claim ceil(L/2) suffices.

- View adding an edge as "covering" all edges on the path between its endpoints (they are no longer bridges). We need to add the fewest edges to cover all tree edges.

- Do a DFS to list the leaves in DFS ordering. Pair up leaf i with leaf (i+floor(n/2)) in this ordering for 0 <= i < floor(n/2). If there is an unpaired leaf, we have n is odd and this is leaf (n+1)/2. Pair it up with leaf 0.

One could do something less elegant to pair up the nodes.

Do a DFS and push the unpaired leaves up. Can show it is possible to pair up most leaves pushed up to an internal node so that a pair comes from different children and exactly one or two unpaired leaves are sent up the tree.

If we start the DFS at a leaf node, we just pair this final leaf with the 1 or 2 leaves pushed up to it.

Still O(n) time. That's less elegant, but it requires less observation to see why it works.

Interesting fact: this problem can be solved in linear time even if G is not a tree (just an arbitrary connected graph). Compute the "leaf 2 edge-connected components" using the same DFS that identifies bridges in an arbitrary graph. Pick one vertex from each component and pair them up using the same strategy.

D1 Solves: 2

Fastest Solve:
Water Black - 1:12:53

Separated Points - Zac Friggstad

Suppose we can do the following quickly: count the number of pairs at distance exactly d (regardless of whether they were sensed or not).

Do this for the input items. Now all we have to do is subtract out the contribution of pairs that are not sensed.

Partition the points into the “gaps” between sensors. Apply the fast counting scheme above to each of these subproblems to count the number of unsensed pairs at each distance d - this finishes the problem.

Finally, to count the # of pairs for each distance build two arrays:

- R[x] = 0 or 1 indicating if there is a point with this x-value
- L[-x] = 0 or 1 indicating if there is a point at value x.

Then we convolve (via FFT, see next slide) these arrays to get an array A[d] = (# pairs of points i,j with x_j - x_i = d). Ignore entries with index d <= 0.

Let V <= 1,000,000 be the maximum x-value. The total running time of all convolutions is O(V log V).

D1 Solves: 2

Fastest Solve:
Water Black - 1:12:53

A different approach (which, admittedly, is the one I thought of even though it is slower than the previous approach):

Suppose (for now) there is only one sensor. Let L[d] be 1 if there is a point distance d to the left of the sensor and R[d] be 1 if there is a point to the right of the sensor.

The answer would be obtained by convolving the arrays L[], R[]. That is, we compute the array A[d] = sum_{0 <= d' <= d} L[d']*R[d-d']. You can use the FFT to do this (I made sure standard FFT was safe enough, no need for NTT).

This array A[] would hold the final solution.

But now let's go back to the case of many sensors!

Divide and conquer

Pick the “median” sensor. Do the convolution trick to get the counts for pairs separated by this sensor. Then recurse on each side of the sensor:

- That is, recurse on only the points lying to the left of the sensor and then only on points lying to the right of the sensor. Combine the solutions found recursively with the one we computed with the convolution in this level of recursion.

If V <= 1,000,000 is the maximum coordinate value, then the total running time of all convolutions across any fixed depth of recursion is O(V log V).

The recursion depth will be O(log M) for a total running time of O(V log V log M).

(divide-and-conquer)^2

D1 Solves: 1

Fastest Solve:
Water Black - 3:11:47

Tournament Rankings
Zac Friggstad

Observation 1: There is a 0-conflict ordering if and only if the tournament (viewed as a directed graph where edges point from winners to losers) is acyclic. An optimal solution can be obtained by reversing the fewest arcs to make the input graph acyclic.

Observation 2: The tournament is acyclic if and only if there is no directed 3-cycle.

Proof: if there is a cycle of length >= 4 then a chord of the cycle can be used to find a smaller cycle.

Recursive Algorithm:

- If there is no cycle, then use the corresponding ordering to update the best solution found so far.
- Otherwise, find a triangle. Recursively try reversing each of its edges. Don’t recurse deeper than K = 8.

The total number of recursive calls is O(3^K).

But we cannot just use O(N^3) time to find a triangle in each call, this is too slow! Even quadratic is too slow.

Here is how to find a triangle in O(N) time with some very simple preprocessing

In addition to the 2d array of games, also maintain an array wins[] that keeps track of how many wins each team has (i.e. just the row sum of the given tournament).

If all wins[] values are distinct, then (bucket) sorting teams by their wins gives an ordering with no conflicts with the games.

Otherwise, if wins[u] == wins[v] then it is easy to prove there is a directed triangle involving u and v. So pick any such u, v and check all other w to find a directed triangle. You can detect wins[u] == wins[v] as you try to bucket sort the teams based on their wins.

When you reverse an edge to make a recursive call, also update wins[] in O(1) time.

Interesting note (not required to solve the problem), we can actually reduce the problem to $O(K^2)$ nodes in $O(N^2 + K*N)$ time and then solve the resulting problem in $O(K^2*3^K)$ time. So we could have done, say, K = 12 and N = 1000.

- Use the outdegree[] trick to find a triangle. If any edge of this triangle lies on > K other triangles then we are forced to reverse it: so do that, reduce the bound K to K-1, and continue.
- Otherwise, record all O(K) triangles that share an edge with this triangle.
- After setting aside all vertices from these O(K) triangles, repeat with the remaining nodes. If after K iterations we still have a triangle, then we have found > K edge-disjoint triangles so it is impossible (stop here).
- Otherwise, we have found $O(K^2)$ nodes such that all triangles use only these nodes. Can show it suffices to focus on reversing edges between these nodes and use the lex-min optimal ordering of these nodes [interleaved with the nodes that were not on any triangle at the end to get the final ordering].