UAPC 2023

Solution Sketches

Thanks!

Problem setters, testers, and judges:

- Ian DeHaan
- Answer
- Noah Gergel
- Joseph Meleshko
- Arseniy Kouzmenkov
- Noah Weninger

Kattis Team: Greg Hamerly, Fredrik Niemelä

UACS - Event site organization and point-of-contact for sponsors.

And Thanks To Our Sponsors!

COMPUTRONIX

CGI

ADVANIS
30 years of innovative research

Division 2 Exclusive Problems

D2 Solves: 43

Fastest Solve:
<p1 /> 0:00:38

Adding Trouble - Noah Gergel

Just do it:

```
A, B, C = map(int, input().split())

if A + B == C:
  print(“correct!”)
else:
  print(“wrong!”)
```

D2 Solves: 37

Fastest Solve:
IDK 0:07:27

International Dates - Noah Gergel

If either A or B is > 12, we know for sure. Otherwise, it could be either.

Need to parse the input a little bit. An easy way in Python is:

```
A, B, C = map(int, input().split("/"))
```

D2 Solves: 20

Fastest Solve:
Pickled Olives
0:37:51

Alien Math - Noah Gergel

Two steps here:

- Identifying the digits (i.e. splitting up the input string)
- Computing the number

For the first, walk along the string until the substring you've seen is a digit string. Remove that digit string and repeat.

Once you have the digit strings, let d0, d1, ..., dk be their values.

The final answer is $d0*B^0 + d1*B^1 + d2*B^2 + d3*B3 + \ldots + d^k*B^k$

The input size was small enough that any approach to get the digit strings should pass. But it can be done in linear time using more advanced data structures (a Trie)

Problems Common to Both Divisions

D1 Solves: 25

D2 Solves: 40

Fastest Solve:
Alberta Amber
0:02:00

Sticky Keys - Noah Gergel

Step through the characters one at a time.

If the character is the first character or is not equal to the previous character, include it in the output. Otherwise, exclude it from the output.

Mean Words - Arseniy Kouzmenkov

D1 Solves: 22

D2 Solves: 29

Fastest Solve:
Revenge of the Bellman Ford 0:05:56

Let M be the maximum word length.

For each 0 <= i < M, compute the averages of the characters at position i of strings with length <= i.

To convert a single character x into ASCII, you can use `ord(x)` in Python. After computing the average `avg` (make sure it is an `int`), use `chr(avg)` to map it back to a string.

If you didn't know these tricks to convert between characters and ASCII, you could have just coded the values yourself (more tedious, but it would work).

D1 Solves: 12

D2 Solves: 8

Fastest Solve: mangos 0:44:45

Mirror Strings - Answer

Considering the height of characters, there will be two types of mirrors, a mirrors string consisting of upper mirror characters (type 1), or a mirror string consisting of lower mirror characters (type 2).

For mirror strings type 1, there are 5 choices to select a character for each digit up to the middle of the string to meet the requirements. And for mirror strings type 2, there are 2 choices to select a character for each digit. Therefore, for a given length N there are $5^C + 2^C$ mirror strings of length N where C = ceil(N/2).

You can just enumerate N from 1 up to R, each time updating your value of 5^C and 2^C whenever C itself increases (keeping it reduced mod $P = 10^9+7$) and adding in $5^C + 2^C$ to your total if N >= L.

CHALLENGE: Do it in O(log R) time (requires understanding modular inverses).

Lefties vs. Righties - Zac Friggstad

D1 Solves: 4

D2 Solves: 8

Fastest Solve: Insert-team-name-here 0:47:02

If any topic is not covered by an expert, it is impossible.

Call a topic “left-only” if only left-leaning experts are experts in that topic, “right-only” if only right-leaning experts are experts in that topic, or “both” otherwise.

Cover all the “left-only” topics and “right-only” topics once. Then while there is a “both” topic that is uncovered, cover it with the expert type that hasn’t been featured as much (either is ok if it is currently balanced). Finally, all topics have been covered. While the interviews so far are not yet balanced between leanings, interview unused experts from the deficient political affiliation (any topic) until things are balanced.

If at any point, there were not enough remaining experts of the required leaning, there is no solution.

Can be implemented to run in O(N + T) time.

Card Counting Club - Arseniy Kouzmenkov

D1 Solves: 8

D2 Solves: 2

Fastest Solve:
Alberta Amber
0:37:25

This is a “just do it” kind of problem - simulate the procedure described in the problem statement. Each player has a container full of ordered values, so pick the “winner”, penalize the “losers”, and keep going.

The real trick to the problem is choosing the right container to enable fast re-insertion of penalized values while maintaining ordering. Remember: inserting into a sorted vector is O(n), need to do better than that to pass the test data.

The intended solution runs in O(n^2*m*log(m))

(Be careful with the Python solution, it is easy to make it TLE by adding too big of a constant factor)

D1 Solves: 2

D2 Solves: 2

Fastest Solve: new HashBrown<Tater>(); 1:07:40

Graduation Table - Ian DeHaan

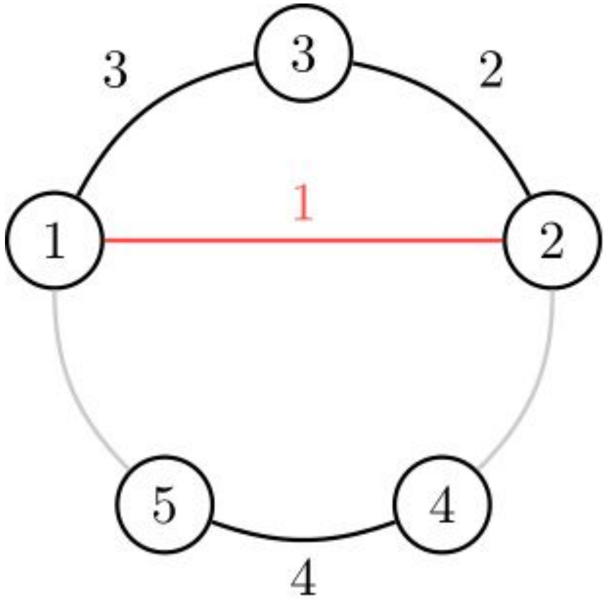
If we think of the bribe requests as edges, we get a graph with maximum degree 2. Graphs with maximum degree 2 are collections of disjoint paths and cycles.

We can choose which requests to fulfill by "gluing" together paths on each component.

If there is exactly one component, then we can fulfill every request.

Otherwise, we will miss a request from each cycle, so choose to miss the least-profitable request from each.

This can be implemented to run in linear time with depth-first-search or breadth-first-search.



Maddison's Square Garden - Joseph Meleshko

D1 Solves: 1

D2 Solves: 1

Fastest Solve: Alberta Amber 3:25:58

For each friend, compute the largest valid garden size. Then, return the minimum.

There are two main approaches for handling each friend:

- Binary search on size of the garden: For each guess of garden size, compute the total time to get from home to work and adjust guess accordingly.
- If you're a fan of geometry, you can just solve for garden length exactly! It can help to notice that the problem can be mirrored without changing the answer. Here's the equation, excepting some edge cases:

$$\ell = 2 \cdot \frac{\left(\cos\left(\operatorname{atan}\left(\frac{y_w - y_h}{x_w - x_h}\right)\right) \cdot \frac{t - \sqrt{(y_w - y_h)^2 + (x_w - x_h)^2}}{\delta - 1}\right) + x_h - \left(\frac{1}{2} \cdot \left(1 - \frac{1}{\left(\frac{y_w - y_h}{x_w - x_h}\right)}\right)\right)}{\left(1 + \frac{1}{\left(\frac{y_w - y_h}{x_w - x_h}\right)}\right)}$$

In either case, the key step is to compute the points where the path between the home and work intersects the garden and then calculate the distance spent in the garden.

Division 1 Exclusive Problems

D1 Solves: 14

Fastest Solve: Alberta Amber 0:16:09

Shopping Bags - Zac Friggstad

Apparently our judge data was not strong enough. A few greedy solutions passed. We will add more data to Open Kattis later to rule them out :)

The intended solution is using dynamic programming. Let f(a,b) be the optimal number of bags to pack “a” type 1 items and “b” type 2 items.

f(0,0) = 0 (no bags required)

otherwise

f(a,b) = 1 + min_{c <= a, d <= b where c*s1 + d*s2 <= T} f(a-c, b-d)
(i.e. try packing (c,d) into one bag for all choices (c,d) and reduce the problem).

But this is too slow: O(n^2) subproblems each iterating over O(n^2) pairs: O(n^4).

Shopping Bags - Zac Friggstad

D1 Solves: 14

Fastest Solve: Alberta Amber 0:16:09

Two optimizations to the recurrence:

1) Since s2 >= T/4, we can constrain the recurrence to only iterate over d <= min(b,4) which reduces the running time to O(n^3).
2) If we already know how many type 2 items to pack in the bag (i.e. for a value d in the recurrence), might as well pack as many type 1 items as possible.

So the recurrence is:
f(a,b) = min_{d <= min(b,4)} f(a-c', b-d)
Where c' is the minimum of a and the number of type1 items that can fit in a bag with capacity T - d*s2.

The running time to process the recurrence is now O(1), so the total running time is O(# states) = O(n^2).

Sneaky Exploration - Ian DeHaan

D1 Solves: 2

Fastest Solve:
I reincarnated as a newbie - Episode 1 1:51:15

This problem asks you to find a hamiltonian path on the complement of a tree.

If the graph is a star on more than one vertex, then it is impossible, as the center vertex has degree 0 in the complement.

Otherwise, we can take the highest degree vertex as the starting point of our path.

This splits the tree into multiple components, one of which has size at least 2.

One can prove that if you keep moving to the highest degree available vertex, you will be able to form a valid path.

D1 Solves: 2

Fastest Solve:
Revenge of the Bellman Ford
3:52:12

Precarious Stacks - Noah Weninger

Can maintain the “skyline” breakpoints using an ordered map.

When a new block drops, use lower_bound() to find the first and last breakpoint. Scan over all breakpoints here to see the max the new box will rest on.

Then also remove these breakpoints, never to be seen again. Add the new breakpoints for the box.

Each box only adds O(1) breakpoints and when a breakpoint is scanned over, it will be removed forever. So the total running time is O(n log n) even though a single box could take O(n) time.

D1 Solves: 2

Fastest Solve:
Revenge of the
Bellman Ford
3:52:12

Precarious Stacks - Noah Weninger

There are at least 2 other approaches that would work:

- Use a lazy segment tree to maintain the maximums over all distinct x-coordinates of the sides of boxes. An update will update a whole range, but we use lazy updates.

- Build a graph over boxes where box (i) has directed edge to box (j) if box (i) can see part of box (j) below it. Straightforward dynamic programming in a DAG solves it. To build the graph, use a sweepline over the start and end edges of the boxes (the sweepline is an ordered set maintaining the “stack” of active boxes).

D1 Solves: 0

Fastest Solve: N/A

Central String - Zac Friggstad

Maintain a guess C for the central string. Initially, let C be the first string in the input. If there is a solution, it differs from C in at most d locations.

Recursively
- If all input strings S have d(S,C) <= D, we are done. Otherwise, let S be any such string.
- Pick any D+1 indices where S disagrees with C. Call these indices L. We know if there is a solution, then at least one index in L must have C_i = S_i.
- So for each i in L, try changing C_i to agree with S_i and recurse (changing C_i back to its original value if the recursion failed).
- Only need to recurse to depth D since the original C differed from a valid solution (if any) in at most D locations as C was an original input string.

Recursion: depth D and branching factor D+1. A careful implementation that tracks how each input string differs from the current string C will have each recursive call take O(N*D) time, so O(N*D*(D+1)^D) time. For D = 6 and N = 50, this is good enough.