New Year's Mash Programming Contest Solutions

CSSS x Competitive Programming Club

Prime Tiling

- **Observation 1:** N * M must be divisible by P. Otherwise, it is impossible since there is no integer number of tiles that fits that area.
- **Observation 2:** If N * M is divisible by P, either N or M (or both) must be divisible by P. This is because P is guaranteed to be prime.
- **Observation 3:** If N or M is divisible by P, it is possible to fill the floor completely.

Therefore, it is possible to retile the floor if and only if N or M is divisible by P

Portal Walk

- **Observation 1:** For any configuration of portals, Bob will not get stuck in an infinite loop. This means that Bob will never reach the same position more than once.
- **Observation 2:** The number of steps it takes Bob to reach the exit is <= distance to exit (X)

Since X <= 1e5, we can just simulate every step that Bob takes. This will run fast enough if we store the portal locations in a map/dictionary.

Stacked Deck

Observation 1: After Bob stacks the deck Bob gets the k highest cards and Alice gets the k lowest cards. This can be easily done by sorting the array of cards.

Observation 2: Bob wins with the **largest** number of points if Bob's highest card is matched with Alice's lowest card, Bobs second highest is matched with Alice's second lowest, etc.

Observation 3: Bob wins with the **smallest** number of points if Bob's highest card is matched with Alice's highest card, Bob's second highest card is matched with Alice's second highest card, etc.

We need to calculate the worst case, which is when Bob wins with the smallest number of points.

Transit Troubles

Observation 1: You can find the time Bob will arrive if he leaves at t=0 in O(N) time.

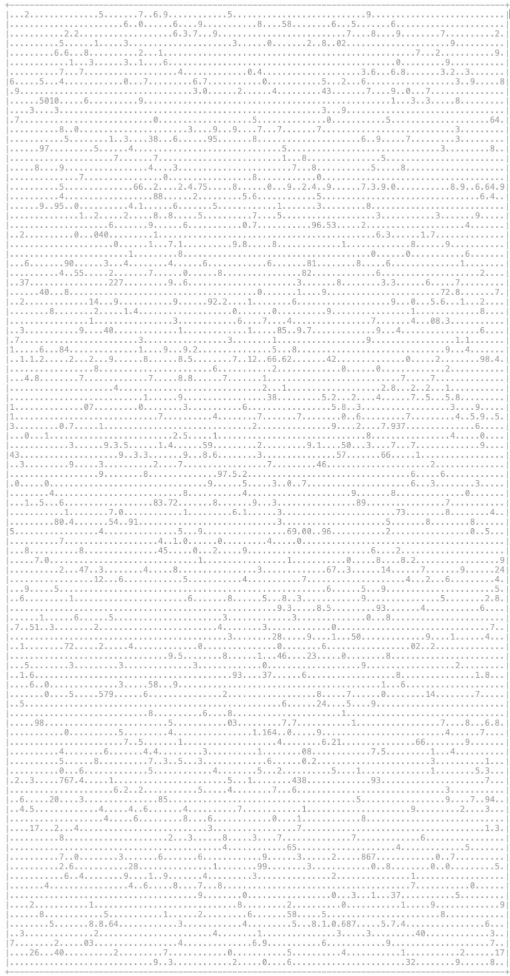
Solution 1: You can binary search on the start time, checking if Bob will arrive at the same time as if he left at t=0. Make sure the upper bound of your search is large enough (1e9 is good). This gives an O(NlogN) solution

Solution 2: Once you find the time Bob would arrive if he leaves at t=0, you can work backwards to find the latest time that Bob can arrive at each of the bus stations. This allows you to find the latest time Bob can leave his house in O(N)

One of
our inputs

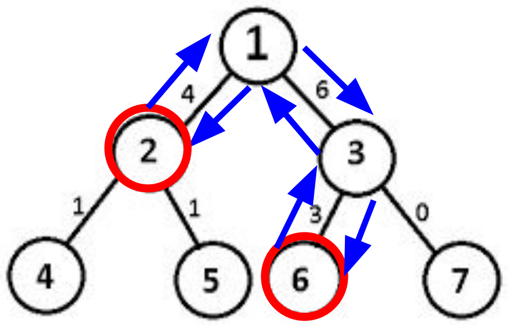
Climbing Wall

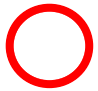
- For every handhold on the wall, calculate and store the minimum sum of difficulties to get there.
- To calculate the minimum sum of difficulties for any given handhold, you only need to look at the handholds below within a Euclidean distance of 5 (< 25 handholds satisfy this constraint).
- Start the calculation from the bottom of the wall, and finish at the top. This solves the problem in O(w * h).



Food Gathering

- This problem is on a tree.
- There's only one path to and from a node and the root.
- Each edge along such path between the root and any food source must be traveled at least twice.
- This is optimal, as you can see!
- Do a DFS from the root, adding twice the weight of any edge with food beyond it.



 = chamber with food

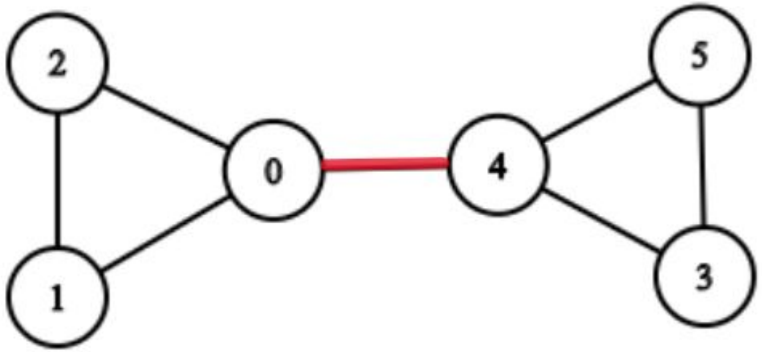
City Planning

Solution idea: It is possible if and only if the graph does not contain any bridges.

Why it works:

- If there is a bridge, then every path from one side of the bridge to the other must cross the bridge. If the bridge is one-way only, then you can't have paths between both sides of the bridge. Thus the answer is **NO.**

Bridges in a graph can be calculated in O(N + M)

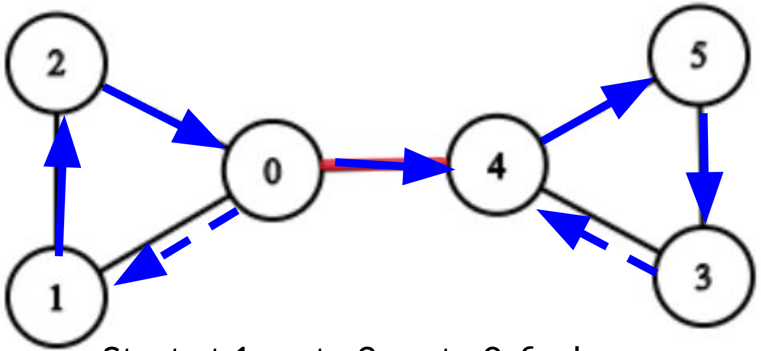


City Planning

- If there **is** a bridge, then when we do a Depth First Search, we find an edge that has no “loop-backs” beyond it.

  Formally: and implementably:

Bridges in a graph can be calculated in O(N + M)



Start at 1, go to 2, go to 0, find a “loop-back” to a vertex before edges 1-2 and 2-0. Thus neither is a bridge.

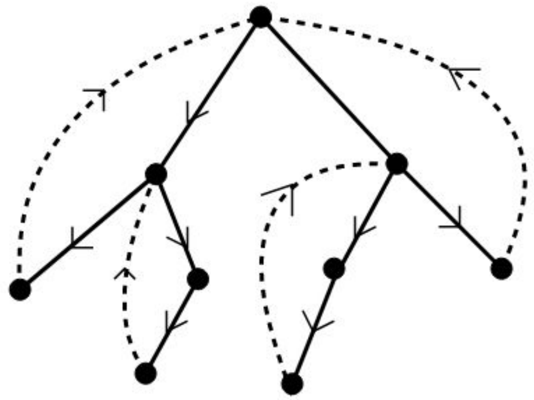
Return to 0, go to 4. We notice that the loop-back from 3 to 4 means the right triangle are not bridges.

But 0-4 has no loop-back!

City Planning

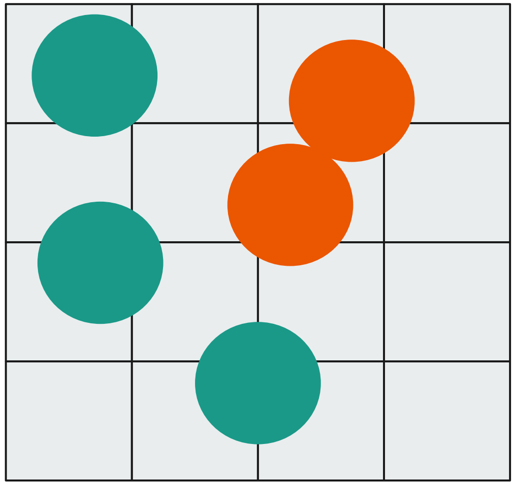
- If there are no bridges, then every edge has a **loop-back**.
- We could orient the forward edges away from the root, and the loop-back (back) edges towards the root. This gives us a directed graph where every node can reach every node (**strongly connected**).

Bridges in a graph can be calculated in O(N + M)


dashed edges are back edges

Bowling Balls

- The naive solution of checking every pair of bowling balls for a collision is O(N^2) - too slow.
  (That's 10 billion checks, a computer does about 1 billion operations per second)
- Use the fact that only one pair of bowling balls are colliding to optimize solution.
- Map the center of every bowling ball to a cell in a grid with cell size ~2 * radius
- For every bowling ball, only check for collisions with balls in the neighbouring cells.
- This gives an O(N) solution, since there can only be a constant number of balls in the neighbouring cells.



(or you could use a closest-pair-of-points algorithm) (<----- That's a useful source of code)
(Here's another algorithm: https://cp-algorithms.com/geometry/nearest_points.html)

Soldier Game

- This is a zero-sum game. This means that every position is either winning or losing.
    - Playing optimally, will the person whose turn it is win from this position?
- A winning position has a move that gives the next player a losing position.
- A losing position has only moves resulting in a winning position for the next player.
- We can solve a game like this by recursively (using memoization) calculating which states are winning or losing.

Soldier Game

- The naïve solution uses a state for each board state. There are 2^32 states, too many to calculate.
- We can improve this: Consider lines of adjacent soldiers (lineups). It doesn’t matter what order they are in, or how much space is between them.
- Let the game state be the sorted list of lengths of soldier “lineups” longer than 1.
- Calculate which of those states wins or loses using recursion and memoization (calculate the answer for a state, then store it so you don’t do it again)
- There are only 1902 such states for a 1x32 board.

Soldier Game (Advanced)

- If you’re feeling jazzy and know some game theory, you can calculate the “Nimber” for each size of lineup.
- A losing state has a Nimber of 0. A state has a Nimber equal to the minimum integer that it can’t reach with a move.
  - If state A can reach states with nimbers 0, 1, 3, then A’s nimber is 2.
- As in the game of Nim, a collection of independent games has a nimber equal to the XOR of all independent games’ nimbers.
- Iteratively calculate the nimber for all lengths of lineups (using the XOR rule), then XOR them all to get the nimber for the whole game. A nonzero nimber means Bob wins.
- https://cp-algorithms.com/game_theory/sprague-grundy-nim.html