Report

1 Cancer Cells

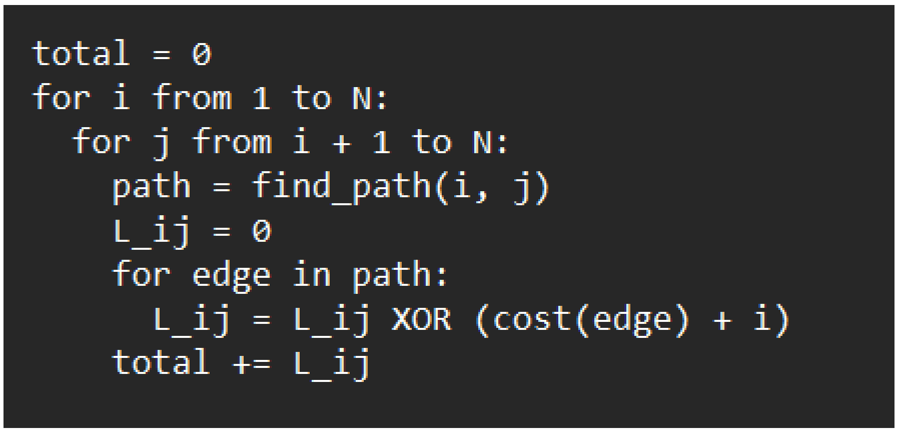
There aren't really many solutions I can think of. In my mind, I might try to get data into an array and then sort with, for example, quicksort/heapsort, and then do work with the biggest value. Or just from recent lessons, using Heaps, especially MaxHeap, is useful there.

Firstly, I integrated a Scanner for fast reading of input and added two more data structures to store cells after division and temporary storage. Also, I stored data in the already-integrated Java data structure, a Priority Queue that always keeps the largest data at its head and behaves like a normal queue. Then, I opened a loop for m times where I take out the biggest data, add q to all elements of cells with using a temporary array, and then divide the largest element and put it back into cells. Then just print out t-th removed cell and t-th final remaining cell. But it was a bit slow, so I integrated Fast Reader and understood that adding q to elements every time is really slow. Thus, I added a new variable q_after_m to calculate just what amount of q I should add in the end. Still, it was slow, so I understood that my own written Priority queue was much faster than the Java Priority queue, and I used code that was in the lecture and integrated into this problem and yeah, it was much faster.

My solution is really good and optimized compared to many other solutions, because I didn't update all the data by q every time, but just saved to add in the end. Also, I wrote Priority Queue myself from the lecture, and it's much faster than Java's Priority Queue.

2 Irminsul's Memory Network

Most easiest way to solve this question is just by brute force, like in this code:



```
total = 0
for i from 1 to N:
  for j from i + 1 to N:
    path = find_path(i, j)
    L_ij = 0
    for edge in path:
      L_ij = L_ij XOR (cost(edge) + i)
    total += L_ij
```

But the Mandate says we should not use that and find a more elegant way to solve it. So some solutions for this problem are just using the tree's common characteristics, which are suggested by the hint, such as Breadth-First Search or Depth-First Search.

Firstly, again for the input reading, I used the same Fast Reader from 1 task. Then, after storing every element in its right place, I built a Tree and enumerated all node pairs where $i < j$. Afterwards, using BFS, I found unique path for each pair and found its cost by finding the direction, which is either upwards or downwards. Knowing about paths and their costs, I used the XOR formula and returned the result/modulo. But this solution was too slow, so I used a different approach. Instead of "compute a full BFS path for every pair", I rewrote to "one BFS per source node". So, I calculated the latency to all other nodes in a single traversal from node i, and finished with the formula of XOR. In the end, time complexity becomes O(N^2).

My solution is better than others because it eliminates all unnecessary repeated work. Instead of recomputing the path between each pair of nodes separately, it performs a single BFS from every source node i, which automatically computes the latency to all other nodes in one traversal.