2020 TJCT Blitz 1 Writeup

Stephen Huan

October 17, 2020

1 A. Teachers

To determine the largest class size, we compute each teacher's class size and take the maximum of them. We maintain a dictionary mapping a teacher id to the number of students they teach. In order to update this dictionary, we iterate over the input, and for each number we see, we add 1 to the corresponding teacher. Finally, we find the maximum value in the dictionary. Since we visit each value in the input a constant number of times, the solution runs in $O(n)$.

2 B. Classwork

We first note that every student must be in a pair. Thus, the largest student is in a pair. We just need to determine who they are paired with, and by conjecture, it seems reasonable they should be paired with the second largest student (since the value of the pair will be the largest student, assigning a smaller time student would be a waste).

Thus, it seems the optimal solution will be greedy, to pair the largest and 2nd largest student, the 3rd largest and 4th largest, and so on. We can implement this fairly easily if we sort the students and add every other number (since the value of a pair will be the larger value in the pair, if the list is sorted, these will occur at odd indexes). We can sort in $O(n \log n)$, which dominates the linear time to add each number at an odd index.

I give a proof of the optimality of the greedy solution as a sidenote.

Lemma 2.1. *Suppose we have two lists a, b of the same length. If we can pair each value of a with a value of b in the form (a_i, b_j) such that $a_i \leq b_j$ for each pair, then a's solution will be less than or equal to b's.*

Proof. Take any solution of b and replace each b_j with its corresponding a_i. Since $a_i \leq b_j$, the maximum of each pair formed must also be less than or equal to the original pair. Thus, the sum of the maximum of each pair is less than or equal to the original sum. □

Theorem 2.2. *The optimal solution will be to pair adjacent values in the sorted list.*

Proof. Proof by induction. For k pairs, the base case is $k = 1$. In that case, there is only one way to make a pair since max is commutative, so forming a solution by sorting is trivially optimal. We now show that if the inductive hypothesis holds for k, it also holds for $k + 1$. We take the largest element, m, and consider which element it should be paired with. Suppose the second largest element is x, and we pick an element that is not the second largest, y. If we pair y with m, that leaves a list with x and the rest of the elements in the list. If we pair x with m, it leaves y and the same remainder. Thus, we can pair up these two lists since each element in the remainder can be paired with itself and y and be paired with x since $y \leq x$. By Lemma 2.1, the list with y must have a solution that is less than or equal to the one with x. Thus, we can pair the largest element with the second largest element and it must be optimal. We formed one pair, so this leaves a list with k pairs. By our inductive hypothesis, we can pair up the remaining elements by sorting. We then add the pair formed by the second largest and largest elements, which maintains the sorted nature of the solution. By induction, the theorem is true for all $k \geq 1$. □

3 C. Robot Project

The simplest solution is to assume we have a certain power level p that we will test whether it works or not. Note that if a smaller power level works, then any power level larger than it will also work (the functionality of the power level is monotonic). Thus, we can binary search on the power level because if a guess works, that serves as an upper bound on the answer, and if a guess fails, then that serves as a lower bound.

We know show how to test a guess efficiently. The observation is that the robot being able to visit at least K rooms is equivalent to finding a connected component of size at least K in the implicit graph, i.e. the graph generated by the power restrictions on the robot. Each (i, j) position in the grid is a node, and there is an edge between two positions if they are adjacent and the absolute value of their power level difference is within k. This graph has N^2 nodes and at most $4N^2$ edges, so we can find its connected components in $O(N^2)$. If the largest possible difference between two elements is P, we do $\log P$ tests from the binary search, so the algorithm overall runs in $O(N^2 \log P)$, which for $P = 10^9$, $\log P \approx 30$, $N = 200$, $N^2 = 4 \cdot 10^4$, will run in time.

3.1 Alternative Solution

If we are generating connected components, we can approach the problem from a union-find perspective instead of a binary search one. The idea is that instead of binary searching to test a particular power level, the number of distinct power levels we need to test is bounded by the number of distinct differences, which is at most $4N^2$. If we again use monotonicity to our advantage and test from lowest to highest, we can successively join components together that are connected by an adjacent power difference less than the one we are currently at, ending when we have a component larger than K.

We start by generating each adjacent difference, both vertical and horizontal. We

add these to a priority queue which we will use to track the smallest power differences to avoid the cost of traversing the adjacent differences, and we will also maintain a set of the values these adjacent differences can take. We start by sorting this set to process each power level from lowest to highest. Suppose we are currently processing a power level p. While our priority queue is non-empty, we see whether the lowest difference in it is less than the current power level p, and if it is, we union the two nodes that generated this difference. Since we union by rank, we can easily check the size of the resulting component, and if it is greater than K, the algorithm terminates and p is reported. If no component is large enough, we take the next largest power level and continue. There are $O(N^2)$ possible values for p, and sorting them costs $O(N^2 \log N^2) = O(N^2 \log N)$. We do at most $4N^2$ adds and pops from our priority queue, and the same bound applies to the number of find and union calls. Thus, the algorithm takes overall $O(N^2 \log N)$. Compare that to the $O(N^2 \log P)$ algorithm derived with binary search. In this case, $P = 10^9$ while $N = 200$, and union-find does run faster. For a large grid with small differences, binary search would run faster (neither algorithm dominates the other).

4 D1. Hallways (easy version)

Because $c_i = 1$, we have an unweighted, undirected graph. In order to minimize the cost, we want to keep the most number of edges (since it costs money to remove an edge). The largest undirected graph without cycles is a tree (since adding an edge to a tree would necessarily add a cycle), so we essentially find all back edges, or edges that violate the tree property. Since the graph is not necessarily connected, we apply the same algorithm to each component. First, we note that if multiple edges connect nodes a, b, then we must remove all except for one because if there are two edges connecting the same nodes, there trivially exists a cycle between the two nodes. Thus, when building the graph, we can assume there is at most one edge between any two nodes, and keep track of the number of edges we remove building the graph. We then run a breadth-first-search (BFS), and when we are expanding on a node by adding its children to the queue, we keep track of the number of children which we have already seen (which is not possible in a tree, since there is only one path between two nodes in a tree). We divide this count by 2, since we count each edge twice (once for each side). BFS runs in $O(N + M)$.

5 D2. Hallways (hard version)

We apply many of the same ideas we used in D1. Instead of constructing arbitrary trees, we must construct the tree of maximum weight (since we don't want to remove expensive edges, we want to keep these). This is a maximum spanning tree, a trivial variant on the minimum spanning tree (MST) problem (take the negative of each weight, and run any standard minimum spanning tree algorithm). Note that the term "spanning" is to be taken liberally, and we will see why. In particular, we will use Kruskal's algorithm for the minimum spanning tree, which greedily builds a MST by adding the minimum weight edges first, skipping adding an edge if it creates a cycle, which can be efficiently tracked

by maintaining the components with union-find. We reverse sort the edges (to put the largest edges first), and proceed as usual. The only special consideration is when we reach a negative edge. If we reach a negative edge, every edge after it must be negative since we process the edges in reverse order. We don't want to add negative edges to the tree we are constructing, since if we remove them we will decrease cost. We also don't have to add negative edges, since we don't need to build a spanning tree (we just need to make sure that there aren't cycles in the resulting graph). Thus, if we see a negative edge, we stop the algorithm there.

Since the resulting solution is identical to Kruskal's except for a single edge case which will cause it to do less iterations, it has the same runtime, where the $O(M \log M)$ cost of sorting the edges dominates the M union and find operations, making our overall algorithm $O(M \log M)$.

Note that any solution to D2 is a valid solution to D1 since D1 is a special case of D2. However, the solution for D1 is faster since it runs in linear time while the solution to D2 requires a sort.