A1 F2022

1. [240 stuff]
A) F in O(G) Yes; G in O(F) No
B) F in O(G) sometimes; G in O(F) sometimes
C) F in O(G) yes; G in O(F) No
D) F in O(G) yes; G in O(F) No

2. [similar to A1 problem]

a) At level k of the recursion tree, there are sqrt(n) * n^1/4 *…*n^1/2k = n^(1-2^-k) nodes with each node having n^(1/2k) work. Hence the total work on level k is n. When k = loglogn, the subproblem size is n^1/2k = 2.

b) The total work done is at most n*k = nloglogn, so we have that T'(n) ∈ $O(n\log\log n)$

c) We prove this statement by induction on n.

Base case: n=3. T(3) = 2 T(2) + dn = 3dn in O(nloglogn)

Inductive step: Assume T(k) in O(kloglogk) for all k < n. We have

$$T(n) = \lceil\sqrt{n}\rceil\, T(\lceil\sqrt{n}\rceil) + dn$$
$$\leq c\lceil\sqrt{n}\rceil^2 \log\log\lceil\sqrt{n}\rceil + dn$$

[idk the math is not mathing]

3. Our algorithm works as follows. We first look at the middle row of the grid and find the minimum element in this row. If it is the first or last row, we return this maximum element. Otherwise, we check the elements immediately to the top and bottom of it. If it is less than or equal to both its top and bottom neighbours, we have found a solution and return it. If it is greater than its top element, we recurse in the upper half of the grid. Otherwise we recurse in the lower half of the grid.

```
function solution(w) {
    middleRow = middle row of w
    i = middleRow.minElement()
    if (n == 1) return middleRow[i]

    if (middleRow[i] <= its top and bottom neighbours)
        return middleCol[i]
    else if (middleRow[i] > top neighbour)
        return solution(top half of w)
    else
        return solution(bottom half of w)
```

}

Runtime: In each iteration, it takes O(n) time to find the minimum element of the middle column. We reduce to one subproblem of size n/2 in each recursive call. Hence the recurrence relation for this algorithm is T(n) = T(n/2) + O(n), which resolves to T(n) in O(n).

Correctness: Our algorithm always returns a solution if w is non-empty, since we either return the maximum of the middle row or recurse into a non-empty subproblem. We prove that the element returned by our algorithm is correct.

Suppose for the sake of contradiction that the returned element (i, j) is not correct. Then it must be greater than one of its neighbours. This element must have been returned in a recursive call where it was the only row in the subproblem, since if it was not the only row then we must have that it is less than or equal to its top, bottom, left, and right neighbours which makes it a correct solution. Hence it is the only row in its recursive call. Its neighbouring row(s) must have been the middle row in some larger recursive call. But since we only recurse in the half where an element is smaller than the middle row's minimum element, we must have that both its neighbours are greater than or equal to it. Hence element (i, j) must be a correct solution.

4. a) If more than half the chips are bad, more than half the results of all pairwise chips will be invalid. This majority of results can be any combination of "good" or "bad", making it impossible to distinguish good chips from bad ones.

b) [Majority element]
We pair up the chips and test each pair, then eliminate all pairs that are not good-good. For each good-good pair, we keep just one chip from the pair. We run this algorithm recursively. When there is one chip left we return it. Since each pair is either completely discarded or only one element is kept, there are at most n/2 elements left.

We show that in each recursive call, we maintain a majority of good chips.

There must be at least one pair where both of the chips are good, more than half the chips are good. For all pairs (p, q), there are 4 possibilities:

p = q, neither are good
p = q, both are good
p != q, neither are good
p != q, one good one bad

Each pair that satisfies one of the latter 3 equations maintains the majority of good chips, since we either have a pair of good chip, remove two bad chips, or remove one good and one bad chip. For each pair that satisfies the first equation, we keep one bad chip. However, every pair of bad chips creates one pair of good chips, so we still maintain majority of good chips in the remaining array.

c) Runtime: The recurrence relation for our algorithm is T(n) = T(n/2) + O(1), which resolves to T(n) in O(n).