1.

a)

i.

Need to both show problem is NP (1), then reduce IND to it (2).

(1) Show 2IND is NP:

Using a guess & check:

Guess the 2 independent sets. Verify in poly time that they satisfy everything: disjoint, independent, correct size etc.

(2) Show poly reduction from IND to 2IND:

Given graph that we want to check IND on: first make entire copy of graph (so we now have 2). Then for every single node in the copy, connect it to every single node in the original. Then run 2IND on this, with same k. This operation is poly.

ii.

Need to both show problem is NP (1), then reduce IND to it (2).

(1) Show IP(D) is NP:

Using a guess & check:

Guess assignment of $$x\_{i}$$s, check expression and constraints are satisfied.

(2) Show poly reduction from IND to IP(D):

Given graph that we want to check IND on, we form a set of values and constraints for IP(D). So the idea behind this it represent each node by an $$x\_{i}$$, so its 1 if it’s contained in the independent set, else 0. Then for the first expression set $$\forall i, c\_{i} = 1$$, and use the same k from IND as in this problem. Also include a first constraint ($$j=1$$) such that $$\forall i, a\_{i} = 1$$ and $$b = k$$, these two ensure the set of nodes is size $$\geq k$$ and $$leq k$$ i.e. $$= k$$.

Now we introduce a lot of constraints to ensure the assignment of nodes is independent. For all nodes which are connected in the graph (e.g. node 3 and node 6) we introduce a constraint s.t. $$a\_{3} = 1$$ and $$a\_{6} = 1$$ (all else 0), and set $$b = 1$$. This constraint means no two nodes adjacent in the graph can both be assigned as 1 for their $$x\_{i}$$.

Now if this expression and set of constraints is satisfiable, then there exists an independent set. This reduction is poly.

b)

i.

see lecture notes

ii.

BIN(D): Given a set of $$n$$ items with sizes $$s\_{1}, \ldots, s\_{n}$$ and bins each with capacity $$c$$, can we accommodate the items with $$k$$ or less bins?

Need to both show problem is NP (1), then reduce PARTITION to it (2).

(1) Show BIN(D) is NP:

Using a guess & check:

Guess the assignment of items to the bins. Check in poly time that this assignment satisfies requirement. This is poly.

(2) Show poly reduction from PARTITION to BIN(D):

For BIN(D) set $$s\_{i}$$ as the non negative integers $$a\_{i}$$. Calculate the sum total of $$s\_{i}$$, and set capacity of the bins as half: $$\frac{ \sum s\_{i} }{2}$$. Set $$k = 2$$. If the minimum number of bins is 2, we have two sets: a partition. (Also note it cannot be 1 bin, since the capacity would be too small to accommodate all the items). Reduction is poly.

iii.

Take any problem in NP, we want to provide an algorithm in P for this.

Since BIN(D) is in NPC (proven in ii), this arbitrary NP problem can be reduced to a BIN(D) problem. So if we can decide this BIN(D) problem in P time we are done.

This can be done with the 3/2 optimal approximation algorithm for BIN. As first see if the optimal solution is 2, just as in (ii), then accept the language, if its 3 or more, no partition can occur so reject. Using the approximation algorithm on this problem will give k times the optimal: < 3/2 \* 2 so < 3. So if the optimal solution is 2, the approximation will give a number less than 3 so it must be 2 - the optimal itself! So In this case, the approximation algorithm gives the optimal so we can base our decision on the decision of approximation.  
Note that solving this special case of k = 2 for BIN in p-time is enough to decide on *any* input for the PARTITION problem, which is NPC.

Since the approximation is in P, we have a machine for the problem in NP. Thus NP = P.

2a) i) p158: For L <= log L’ we have f that takes no more than logspace (O(log n) space) to map a problem in L to a problem in L’ such that x is in L iff f(x) is in L’ for any x.

ii) Take L in Sigma\* arb.

(=>): Assume L is in LOGSPACE. Any problem in LOGSPACE reduces to another problem in LOGSPACE by def LOGSPACE.

For L <= log {a} we define f(x) = a where x is in L, or empty set otherwise. This takes no more than logspace to map. (=> : Assuming x is in L then f(x) is in a; <= : assuming f(x) is in {a} then x must be in L by definition.

So L <= log {a}.

(<=): Assume L <= log {a}; then we have f that maps x in L to a, or empty set otherwise. {a} is in LOGSPACE from reasoning above and LOGSPACE is downwards closed => L is in LOGSPACE.

b) i) RCH is in NL as we guess a path from x to y, where we track only the current node, number of steps and x, y (all require counters => O(log n) space). We travel from x, tracking only the current node and succeed if we reach y in n or fewer steps (can stop after n).

ii) For CYCLE, we guess a path from x to itself in no more than n+1 steps. As before we track x, current node, and num steps with counters => O(log n) space where we stop if we reach x.

c) i) To show:

(1) LRCH is in NP

(2) LRCH is NP-hard

(1): Guess a path from x to y and check we can reach y from x (track curr node, x, y, stop after n steps) => O(log n) space.

(2): Need to show RCH <= LRCH (RCH is NL-complete so LRCH would be NL-hard if this holds)

=> (G, x, y) in RCH iff f(G, x, y) in LRCH

~~Define f as creating partitions of size 1, Ai, containing nodes xi for all i.~~

Let f(G, x, y) return the same graph with the addition of partitions: we start from x and traverse over all edges, incrementing a counter as we do (starting value of 1) and allocating the node we arrive at in the new class (so x is in A1, next node in A2 etc.). If an edge would cause us to move back into a partition then we remove it to satisfy the conditions of a levelled directed graph.

(=>): Assume (G, x, y) in RCH. Then apply f to G and we see we have a path only going between different partitions in G. So f(G, x, y) in LRCH.

(<=): Assume f(G, x, y) in LRCH. Then a path exists from x to y and so (G, x, y) in RCH (note G is unaffected by f except for tracking partitions).

Solution linked from piazza: [[https://people.cs.umass.edu/~barring/cs601sum03/hw/4sol.html]](https://people.cs.umass.edu/~barring/cs601sumy03/hw/4sol.html)

We need a logspace function that, given a directed graph G of n vertices and nodes s and t, will produce an acyclic leveled graph H and nodes s' and t', so that there is a path from s to t in G iff there is a path from s' to t' in H. Here is an easy way to do this. The nodes of H consist of n copies of the nodes of G, each of which constitute a level. For every i and every node x in G, we let "(x,i)" be the name of the copy of node x on level i. We make an edge from every node (x,i) to (x,i+1), and an edge from (x,i) to (y,i+1) whenever (x,y) is an edge in G. We let s' be (s,1) and t' be (t,n).

If there is a path from s to t in G of length d, we can go from (s,1) to (t,d+l) in H by following the copies of each edge in the path in turn. Then we can get to (t,n) by taking a series of (t,i) to (t,i+1) edges, so there is a path from s' to t' in H.

If there is a path from s' to t' in H, the edges in it consist of either (x,i) to (x,i+1) edges or edges mirroring edges in G. If we look at the latter edges in order, they give us a path from s to t in G.

Since we have reduced REACH to REACH-ACYCLIC, and shown that REACH-ACYCLIC is in NL, we have shown that REACH-ACYCLIC is NL-complete.

c) ii) To show:

(1) CYCLE is in NL

(2) CYCLE is NL-hard

(1) As in (b) ii, guess a path from x to itself in no more than n+1 steps, tracking num steps, curr node and x with counters.

(2) To show: RCH <= CYCLE.

(G, x, y) in RCH iff f(G, x, y) in CYCLE

Let f create edges (y, x) for all edges (x, y) in G (counter to iterate over all edges => O(log n^2) = O(log n) space).

(=>): Assume (G, x, y) in RCH.

Apply f to (G, x, y). Then we check if a cycle exists from x to x in no more than n’ +1 steps, n’ = 2n by construction of the new graph G’ using f. As (G, x, y) in RCH we have a path from x to y and so have a path from y to x so f(G, x, y) in CYCLE.

(<=): Assume f(G, x, y) in CYCLE. Then there exists a path from x to x, going through y. So there is a path from x to y in the original graph. So (G, x, y) in RCH.

iii) To show:

1. ACYCLIC is in NL
2. ACYCLIC is NL-hard

(1): Check the complement problem CYCLE to see if a cycle exists and return the opposite result (check all paths one at a time, each taking no more than logspace). If a cycle exists return no else return yes. CYCLE is in NL => ACYCLIC is in co-NL. But co-NL = NL so ACYCLIC is in NL.

(2): **Not sure, please add here.** My idea was to consider the complementary graph where we remove existing (x, y) and add edges (x’, y’) where the edge did not exist before, and work with this somehow. If this sounds totally wrong then replace.

I think a more abstract proof works better here:

Take L in NL arbitrary want to show L <= Acyclic:

L in NL => not-L in co-NL and co-NL = NL so not-L in NL => (not-L <= Cyclic) (Cyclic is NL-complete) => ((L) <= Acyclic), proven for L arbitrary so Acyclic is NL-complete. Where we’ve used (L <= L’ => not-L <= not-L').

3a) i) p204: NCj = PT/WK(log^j n, n^k), NC = union of NCjs for j>=1 = PT/WK(log^k n, n^k) for k > 0.

Question also asks for NC0 which is presumably PT/WK(1, n^k) (slides only define j >= 1)

ii) NC1 is in L is in NL is in NC2 is in NC is in P.

iii) As NC is in P, a P-complete problem in NC would mean for all L in P, L reduces to L’ where L’ is the P-complete problem in NC. L can reduce to L’ (proof on p73) and as both P and NC are downwards-closed we have all L must be in NC.

b) To show: L is in NC1. Can check each digit xi pairwise using the tournament design and add the values of the pair together. Check at the final value if this equals n – 1 (each xi will be either 0 or 1 so this will only be true if there is exactly one 0). This has ceil(log n) steps and therefore takes O(log n) parallel time. Work done is O(n) as we have n digits => L is in PT/WK(log n, n) and so L is in NC1.

Alternative answer, wdyt?:

Firstly, OR every single pair of input characters together (i.e. x\_i \/ x\_j for all i, j). This produces n(n-1) outputs. We can see that if any single one of these ouputs is 0, there must have been a pair of 0s in the input, so the input would not have exactly one 0 as required. So now we just want to reject iff any 0s in this layer.

So next we tournament (binary tree style thing) together all these previous layer outputs with AND gates. If there is even a single 0 for this (caused by a pair of 0s), the output will be 0. Overall, the design outputs 0 iff any pair of 0s in the input, 1 iff at **most one** 0 in the input (including zero 0s :(, see next paragraph).

To fix this issue with at most one 0: we also, in parallel have an AND tournament for all the inputs together, then negate the final result: this will be 0 iff the input is all 1s.

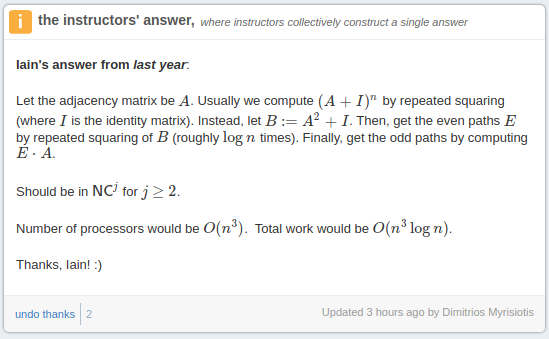
So finally we AND together these two results of the two detectors. This will be 1 iff exactly one 0 in the input.

3rd idea, might be wrong but it seems to work:

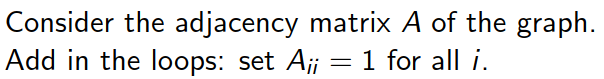
Use XOR gates. Just start traversing and XOR-ing each pair. If we encounter a 0 (and a 1 lets say), the output of this gate will be 1, whereas if we encounter (1,1), we will get a 0. Do this with the outputs of these gates and create a tournament of XOR gates. If in the end we get a 0, it means there are either 0 or >1 number of zeroes, and if the output is 1, we accept it.

Test with diff inputs, arranged as you want: (0,1) (1,1) -> (1,0) -> 1 ; (0,0) (1,1) -> (0,0) -> 0; (1,1) (1,1) -> (0,0) -> 0; (0,1) (1,1) (1,1) (0,1) -> (1,0) (0,1) -> (1,1) -> 0.

c)



Idk about everyone else, but who knew its A + I?! (including that identity). Checkout slide 179 for this subtle but important detail:



**Not sure here.** Idea: construct some circuit with y as the root and navigate backwards using a bitstring to track the current path (enumerating paths with 0, 1 etc.) and track length of current path. If we reach x check if the path is of size n +1 and stop. Not sure what WD or PT would be here though.

My alternative idea but not sure about it:

Use a modified method from pg180: Repeatedly square the adjacency matrix, but using modified arithmetic logic.

First, instead of having 1,0 for edge or no edge, use: 0 no edge, 1 odd number of edges, 2 even number of edges, 3 can be either odd or even number of edges.

Then instead of using sum of multiplications (in boolean, OR of ANDs), replace the AND with a rule to check parity of the joined edges, i.e. 1 x 1 = 2 (odd join odd is even), 2 x 2 = 2 (even join even is even), 1 x 2 = 2 x 1 = 1 (odd joining even is odd), 3 x 1= 3 x 2 = 3 (can be either) etc. Also replace OR with an any-of kind of rule. I.e. if all the ANDs are 1, this sums to 1, if all the ANDs are 2, this sums to 2, if some of both 1 & 2, sum is 3 etc.

Repeatedly squaring in this way ends you with a matrix representing parity of paths, read off x to y.

4.

a)

i. ii. see lecture notes

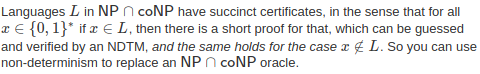
iii. P in RP in NP

iv. tutorial question

v. see lecture notes

b) i.

from piazza:



the same but more thorough:

Take any language in $$NP^{NP \cap co-NP}$$, we can show a machine for this language which is in NP.

We can easily start by constructing a machine which is the base of that exponent language, since that’s in NP, our construction can be.

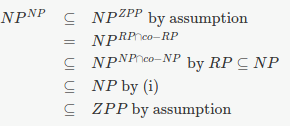
Of course, from time to time, that machine will want to consult an oracle for a language which is in both NP and co-NP ($$CP \cap co-NP$$ - the exponent). For this, take both the NP machine and co-NP for the oracle language, and add these two machines into our construction.

Modify them a bit: run both machines at the same time (by interleaving their steps), so that if one terminates with acceptance, we can kill the other. This means we have a construction which can identify members of the language, and always halts with rejection or acceptance (unlike a machine in NP). And this can be rerun as many times as required.

This pair machine construction is in NP, so together with the first machine, the entire construction is in NP. Thus this language is in NP.

ii.

piazza:



alternative (lengthy, bad):

Take any language in $$NP^{NP}$$, to show this is in ZPP, build a construction as follows:

For the language of the base of the exponent, since the language is in ZPP, use the ZPP machine. For the exponent/oracle, also use a ZPP machine for that language. This exponent machine can be rerun multiple times, and that would still be poly. And since both parts of the construction are in ZPP, they have 0 probability of error, therefore so does the entire construction.

The last thing to check is the probability bound is low enough: for a machine in ZPP, it may consult its RP machine and say x ∉ L (wrong with probability p), then consult the co-RP machine and that says x [∈](https://www.compart.com/en/unicode/U+2208) L (wrong with probability p), so this ZPP machine says ‘not sure’ with error p^2. Likewise the first machine can say ‘not sure’ with prob: p^2. So in total it say ‘not sure’ with prob: 2p^2, and you only need to rerun the entire construction polynomially many times to reduce this within the bound so its good.

[tbh I think a real proof requires a lot more thorough treatment of this bound (what if the oracle is rerun? worst case is actually e^p(n) what to do?) but not sure if that's really required as it would take way long so Ive just done it ‘informally’].

Therefore the construction in poly, has 0 probability or error and the ‘not sure’ answer occurs within bounds.