1.

a)

i. see lecture notes

ii. VAL

iii.

b)

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

(1) Show NIND is NP:

Using a guess & check:

Guess set of nodes of size k. Check: set contains x, set is an independent set. This check is poly.

(2) Show poly reduction from IND to NIND:

Add a single extra node to the graph, connect it to nothing. This can always be contained in an independent set. Output this as NIND problem, with k + 1 as size.

c)

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

(1) Show SIND is NP:

Using a guess & check:

Guess set of nodes of size k. Check set is a strongly independent set. This check is poly.

(2) Show poly reduction from IND to SIND:

Given graph G, replace all edges in G with two edges, and a single node between them connected. Therefore a strongly independent set with 2 edges spacing would have 1 edges spacing in original G. Output new graph, same k.

2.

a)

i. There exists a machine which decides that language. It requires a single counter. Macine reads 0s (incrementing counter each time), until 1. Reads 1s decrementing counter, accepts if counter is now 0.

ii.

Input word of length n. Run f, using at most logspace. Since L subset of P, output of f is at most poly in n (which would not contribute to the space usage).

So input to g is poly in n. Run g, uses log of poly in n (log n ^k) = multiple of log (k log n). So computation of g is also logspace. Add these, and so the composition is also just logspace.

b)

ii.

co-NL = NL

c)

i.

First recall reachability: is there a path from x to y of length k? This is in NL.

Also since co-NL = NL, the following problem is also in NL: is there **no** path from x to y of length k?

Now to solve SHORTRCH, run RCH for k. I.e. is there a path from x to y of length k? Now we need to check this is a minimum: repeatedly (poly times) run the machine for the co problem for 0 upto k-1. This tells us if there is no path from x to y, for any length less than k. If this is the case, the path of length k must be the shortest path.

ii.

Show L = NL implies SHORTRCH in L

By c(i) SHORTRCH is in NL, and we’ve got NL = L so SHORTRCH is in L

Show SHORTRCH in L implies L = NL

This follows from the fact RCH **Cook** reduces to SHORTRCH. Why: Given nodes x, y asking if y reachable from x. Run the oracle for SHORTRCH with the same x, y, and use k as integers from 0 to N (number of nodes). If y reachable from x, it must do this at least within the number of nodes.

Take any problem in NL, since RCH is NL-complete, it can be reduced to an RCH problem. Using the above, we can use our L machine for this. We must rerun the machine multiple times (for a cook reduction), but this does not increase space requirements, thus we’ve solved the NL problem in L. So NL = L.

3

a)

i. see lectures

ii. Coursework question

$$NC\_{i} \subseteq AC\_{i} $$

If the fan-in is bounded at 2, this already satisfies requirement for unbounded fan-in.

$$AC\_{i} \subseteq AC\_{i+1} $$

Idea is, anytime we see a fan-in greater than 2, we do a binary tree style combinment of the inputs. If we have to do this at every level, we multiply levels by $$\log n$$ (height of a binary tree). $$\log^{i} n \times \log n = \log^{i+1} n$$. Should also show WK is still poly.

iii.

First layer of circuit: /\ every adjacent pair together.

Next layers: \/ everything together (done in one layer with an unbounded fan in)

PT total: 2 layers. so $$AC\_{0} $$ (WK is poly)

b)

i.

First layer of circuit: $$\land$$ every pair together (including not adjacent).

Next layers: $$\lor$$ everything together in binary tree style (takes $$\log n$$ PT)

PT total: $$1 + log n$$ layers. so $$NC\_{1}$$ (WK is poly - for the first layer we have n(n-1) pairs)

ii.

We run EVEN on every node, for its connections. Accept only if all nodes are EVEN.

To be a bit more specific, given the graph as an adjacency matrix. We apply a single row (or column, its undirected thus symmetric) to EVEN. Thus 1s represent connections. EVEN checks for the parity. Then we $$\land$$ together all the results, in binary tree style. Althogether PT: $$\log n + \log n$$ so $$ NC\_{1}$$

c)

Consider a word in L1L2, we need to check if there exists a word in L1 and if there exists a word in L2, which concatenated together, made that word.

To be more explicit say L1 is the language or repeated 1s: {1}\*, and L2 the language of repeated 0s: {0}\*. Given the word 11111000, we can say this is in L1L2, since 1111 ∈ {1}\* and 000 ∈ {0}\*. Also see, a machine to check this needs to find out that splitting point - in this case after character 5. (Then check if the first half is in the first language, the second half in the second language). As it won’t know what point that is, when checking a word.

The solution to finding that splitting point is just to check all splitting points. E.g. for a word 11111000, we split at 1, then check both if 1 ∈ L1 & 1111000 ∈ L2, we split at 2 then check both if 11 ∈ L1 & 111000 ∈ L2, we split at 3 then check both 111 ∈ L1 & 11000 ∈ L2 etc for all split points. If any of these split points and two words are valid, we can accept the language.

Of course if we were to check these split points sequentially, this would take too long. Solution: parallelize. For split point 1, we use the (size 1) circuit for L1 (in NCi) on the first character, and **in parallel** the (size n-1) circuit for L2 (in NCi) on the rest of the word. I**n parallel** again, for split point 2, we use the (size 2) circuit for L2 (in NCi), and the (size n-2) circuit for L2 (in NCi) on the rest of the word. Etc. for all split points possible.

Crucially, these circuits are all in parallel - both each pair for each half, as well as for each splitting point, therefore we do not increase the PT at all. It is therefore still in NCi. As for WK, each splitting point adds poly WK, so in total this is still poly + n \* poly which is still poly. Thus NC\_{i}.

4.

a) i. ii.

b) i. ii.

iii. this is a tutorial q

c)

i.

SAT can be expressed as functional problem: find vars s.t. $$\varphi(x\_{1}, x\_{2} ,\ldots )$$ is true.

Run BPP algo on $$\varphi(x\_{1}, x\_{2} ,\ldots )$$. Continue iff (probably) satisfiable.

Run BPP algo on $$\varphi(tt, x\_{2} ,\ldots )$$ and $$\varphi(ff, x\_{2} ,\ldots )$$. Fix $$x\_{1}$$ as whichever is still (probably) satisfiable (or arbitrarily if both are satisfiable). If neither, reject.

Say only $$x\_{1} = tt$$ is (probably) satisfiable. Run BPP algo on $$\varphi(tt, tt,\ldots )$$ and $$\varphi(tt, ff,\ldots )$$. Fix $$x\_{2}$$ as whichever is still (probably) satisfiable (or arbitrarily if both are satisfiable). If neither, reject.

Repeat for all variables

Finally we have a (probably) satisfying assignment. Check this unprobablistically (p-time), if wrong reject. Thus if BPP gave us a wrong answer for any one of those variables fixes, we will now discover it, guaranteeing we cannot find a wrong answer. Of course at any step we could have wrongly thought it is not satisfiable, but this allowed by a RP machine (it just must be less than 1/2 of runs).

Need to check the error probability is within tolerance, which is quite messy but I think ok.

ii.

Assume NP in BPP.

Take any problem in NP. This can deterministically be reduced to SAT, as SAT is NPC. Since SAT is in NP its in BPP. So the result from i. holds (SAT in RP), and so this arbitrary problem in NP is also in RP. So we’ve now got $$NP \subseteq RP$$. Of course $$RP \subseteq NP$$. Thus RP = NP.