1.

1. This is just definitions, see notes.
2. **SPANTREE is** **NP**: guess a spanning tree *M* of *G* and a 1-1 mapping *T -> M* in polynomial time. Then check in polynomial time that the mapping is an isomorphism.

**SPANTREE** **is** **NP-complete**: Assuming that HP is NP-complete, it suffices to show that HP ≤ SPANTREE. Given an input G to HP with n nodes, let T be the graph given by a chain of n nodes: x1 – x2 -- … -- xn. Then, a Hamiltonian path is a spanning tree of G, so G has a Hamiltonian path if and only if T is isomorphic to a spanning tree of G.

1. **LONGRCH is NP**: Given a directed graph G, nodes x and y and a number k, we guess a path starting at x in polynomial time. Then, we can verify in polynomial time that this path ends at y, visits each node only once and has length at least k.

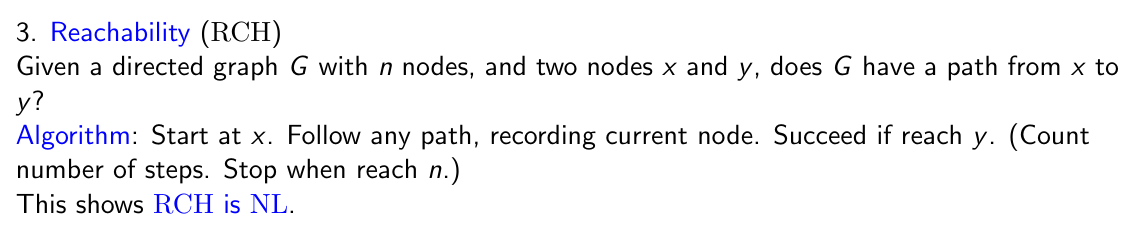
**LONGRCH is NP-complete**: We will show that HP ≤ LONGRCH. Given a graph G, we form G’ by assigning each edge in G a direction, duplicating the edges but with directions reversed, and adding nodes x and y. The node x has an outgoing edge to every node in G, and the node y has an incoming edge from every node in G. Then if G has a Hamiltonian path if and only if G’ has a simple path from x to y of length n + 1, where n is the number of nodes in G. Indeed, if G has a HP x’...y’ (which has length n – 1) we can lift this to a simple path x, x’, …, y’, y in G’, which has length n + 1. Conversely, if G’ has a simple path of length n + 1 from x to y, then by removing the edges from x and to y and forgetting the direction, we get a Hamiltonian path in G.

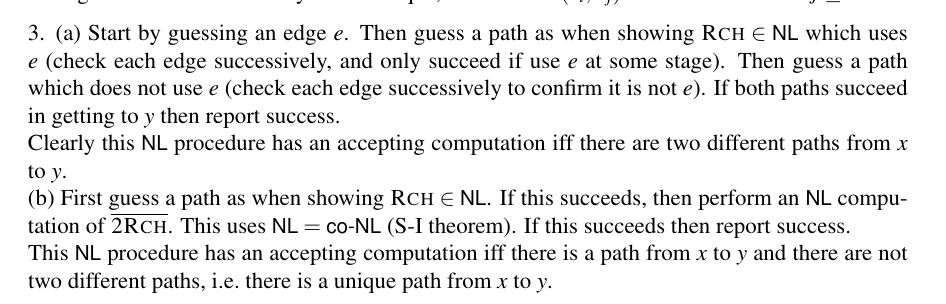
2.

1. 1. Definitions - see notes.

2. Suppose the input has length k = O(log(m) + log(n)). Then we can first check whether one is longer than the other by keeping two counters which take up log(k) space. If, say, m is shorter than n, then we must have m < n, so return m. Now assume that they have the same length. Then we may find the smallest number by comparing bitwise starting with the most significant bit. We need to keep track of where we got to in each number to do this, but this takes log(k) space using a counter.

The rest of this question is covered in lectures and problem sheets.



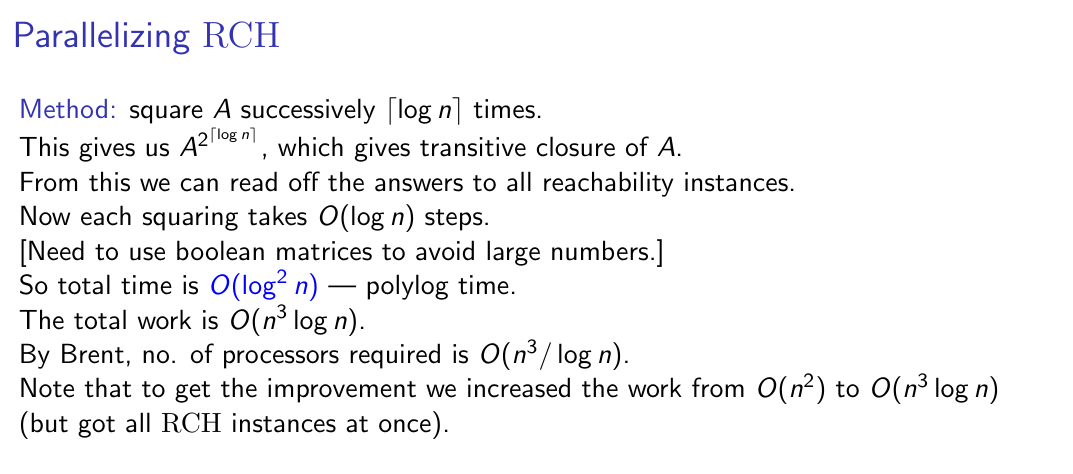


3.

1. Parts 1 – 2 are definitions.

3. If L1 and L2 are both in NC, then we have that L1 is in NC\_j1 and L2 is in NC\_j2, for some j1 and j2. We will show that L1 ∩ L2 is in NC\_j, where j is the maximum of j1 and j2. Fix some n, then we will construct a uniform circuit C\_n. There exist uniform circuits A\_n and B\_n for L1 and L2 respectively; construct C\_n by passing x1, …, xn as inputs to both An and Bn. If the outputs of An and Bn are a and b respectively, then C\_n has an and-gate calculating a and b. C\_n outputs 1 if and only if x1...xn is a word in both L1 and L2. C\_n is uniform since A\_n and B\_n are uniform. The depth of C\_n is then equal to max{depth(A\_n), depth(B\_n)} + 1 = O(log^{max(j1, j2)} n). The size of Cn is still polynomial in n since A\_n and B\_n are polynomial in size.

1. (Sketch) Build a circuit based on the following algorithm. We can assume by appending a string of 1s that the input w has length 2^k. Then write w = xy, where x and y are strings of length 2^{k-1}. Now, w is an element of ZeroOne if and only if: x is an element of ZeroOne and y is a string of 1s OR x is a string of zeroes and y is an element of ZeroOne. For each of x and y, we can then recurse to determine whether they are in ZeroOne. The circuit for this has depth O(log n).
2. This is a simple modification to RCH, covered in slides.



4.

1. 1. FSUBSETSUM asks: given X and k, find a subset of X which sums to k.

2. Suppose X = {x1, …, xn}. We proceed in rounds. Query the oracle to check whether some subset of X sums to k; if not, return ‘no’ and exit.

Set Y = X initially. In each round, choose an element x in Y, and set U = Y \ {x}. Query the oracle to see whether U contains a subset which sums to k. If ‘yes', then set Y = U for the next round. If ‘no’, then do not modify Y and proceed with the next round.

After at most n rounds, Y will contain elements which sum to k. The proof is as follows: suppose the elements at the end are {x\_i1, …, x\_ir}. On the final round, Y must have been either {x\_i1, …, x\_ir, xn} or {x\_i1, …, x\_ir}. In the first case, the oracle returned ‘yes’ on input {x\_i1, …, x\_ir, xn} \ {xn}, so it follows that a subset of {x\_i1, …, x\_ir} sums to k. On the other hand, an element x is kept in the set on a round only if it is an element of all subsets of the current Y which sum to k. From this we see that x\_i1 + … + x\_ir is bounded by k, so it must in fact sum to k.

In the second case, the oracle must have returned ‘no’ on the input {x\_i1, …, x\_ir} \ {x\_ir}. An invariant of the algorithm is that at each stage, Y contains a subset which sums to k. So then {x\_i1, …, x\_ir} must sum to k.

3. FP^NP

1. 1. |APPROX| ≤ k |OPT|

2. |OPT| ≤ k |APPROX|

3. Set all variables to true and count how many clauses are satisfied: call this number n. Now set all variables to false, and count how many clauses are satisfied: call this number m. For each clause, it is satisfied in at least one of the two cases. If less than half the clauses are satisfied in the first case, then more than half the clauses will be satisfied in the second case. Hence this algorithm always satisfies at least half of the clauses. The optimal maximum number of clauses is bounded above by the total number of clauses, so this is a 2-optimal algorithm which is polynomial time (we only need to iterate through each clause twice).