# Question 1

## Part (a)

BTSP is NP: Guess a circuit and check in p-time

BTSP is NP-Hard:

* We can show HC reduces to BTSP
* From our input graph we can create a weighted graph where every node has weight 1
* We then run BTSP with B = 2

## Part (b)

* We can read through the string building two strings A and B
* At each character we make a guess whether the character should be added to A or B
* We then ask M1 (a TM which decides L1) is A is in L1 and vice versa for B and L2
* Iff the input is a merge there will be some accepting computation

## Part (c)

SFL is NP: Guess a path and check in p-time

SFL is NP-Hard

* We can show HP reduces to SFL
* Let G be our input graph with n nodes; we build G’
* The SFL requires the start and finish nodes as input so we add super-nodes x and y connected to each node in G
* We then run SFL on G’ with start / end node x/y and B = n + 1
* Consider G contains a HP:
  + The paths must have start node a and end node b; the path must be simple and be made of n – 1 edges
  + We can add an edge from x to a at the start and from b to y at the end for a simple path size n + 1
* Consider G’ contains a simple path from x to y size >= n + 1
  + There are only n + 2 nodes so the size must be n + 1 exactly
  + We can strip of the initial and final edge to get our HP

# Question 2

## Part (a)

### Part (i)

* We can build a NDTM that decided RCH and uses only logspace as follows:
* Starting at x following a random path keeping track of the current node and the number of nodes visited
* Stop when the current node is y or we have visited more than |nodes(G)| = n nodes
* The counters are both bounded by n so only logspace is used
* We can discard paths over n nodes long because they contain repeated nodes

### Part (ii)

* L1 and L2 have logspace NDTMs M1 and M2 which decide them respectively
* Given input x we can decide if it’s in M1 and M2 by running M1 and then M2 and returning yes only if they both return yes
* We run the TMs in succession so because they are both logspace our new TM is logspace

## Part (b)

For every node, generate all permutations of subsets in O(n^k) (where n is the number of nodes), and check if all nodes are connected to each other, moving onto the next node if there is a valid permutation or rejecting if there is no valid permutation. We need one counter with space O(log n) to store the current node number, and (k-1) counters here to store the node numbers of the subset, which leads to O(klog n) space, which is fine if k is a fix u ed constant.

## Part (c)

### Part(i)

* From part a we have seen that RCH is in NL
* We also know NL = co-NL
* So we can ask are x and y reachable in the original graph and are x and y not-reachable in the graph without e

### Part(ii)

Algorithm:

* First run RCH on the input just to check it is reachable at all. If not reachable return fail.
* Iterate through every edge of the graph, keeping a counter.
  + With an edge, run the machine from c(i) on it, to check if every path uses it.
  + If so, increment the counter. Else don't increment but continue.
* Terminate with success and output the counter.

(Note use of c(i) introduces cases which guess the wrong path and won’t check if an edge is used by every path: in these cases terminate the outer machine with failure also.)

Correctness (1):

(There are n edges used by every path => output n) Since the machine from c(i) will guaranteeably check an edge, and we check every edge, all n edges will be found and it will output n in this case.

(Output n => There are n edges used by every path) If the machine outputs n, we must have had n increments of the counter, since each edge can only contribute +1, we must have n edges do this. Since c(i) works, these n edges must be used by every path.

Termination (2):

There is of course at least one successful termination e.g. a 2 node, 1 edge graph: o-o will return 1

Efficiency:

c(i) is in NL, and we can re-run NL algorithms, by overwriting the memory. We only keep a single counter consistently across re-runs. Therefore NL.

# Question 3

## Part (a)

Definitions

## Part (b)

* Consider we could use xor gates in our circuit
* We have n-1 xor gates, where each takes a pair of adjacent chars as input; the gate will output 1 iff the chars are alternating
* We can then build a tournament of and gates to calculate if every pair is alternating
* Infact we can construct an xor gate with a constant number of not and and gates

## Part (c)

* It’s easy to see how we could build an NC1 for a set value of i
  + Like part a we could look at pairs but we want all the pairs to be the same except for 1
* We can have the input gates feed into n different circuits for every value of i and check that 1 output a 1

## Part (d)

Needs a bit more explanation but I’m bored

* If we knew the size of y we would therefore know the size of x and could check
  + That the middle bit was in L2
  + That the beginning and end are the same
  + That the beginning and end are in L1
* We can test all values for the length of y in parallel, if any return 1 then return 1

# Question 4

## Part (a)

### Part (i)

* We can construct a language L which contains every string; we can build a very simple NDTM which decides this language that just says yes on any input i.e L is NP
* Take L’ in co-NP, clearly insect(L, L’) = L’ so L’ in in DP

### Part (ii)

* Consider that DP = NP
* From part (i) co-NP is a subset of NP

Proof from revision lecture (thanks to Clovis!)

First show co-NP ⊆ NP:

co-NP ⊆ DP = NP so co-NP ⊆ NP as required.

Next show NP ⊆ co-NP:

With co-NP ⊆ NP, take co of both sides:

co-co-NP ⊆ co-NP which shows (since co-co-NP = NP)

NP ⊆ co-NP

So we have both sides and NP = co-NP.

## Part (b)

Bookwork

## Part (c)

### Part (I)

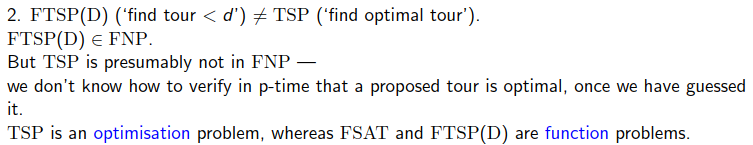
Incrementing k method

### Part (ii)

I don’t think so? I dont’ think you can guess and check that the cover is optimal in p-time.

Whereas FVC(D) (=/= MVC) is probably FNP.

argue just like this (slide 254):



### Part (iii)

First we need to find the minimum size. Using the VC(D), increment a counter for k from 0 up, and find the first k which reports success, providing the minimum size of a vertex cover.

Next we need to test this minimum VC is unique.

*This problem is annoying as usually to show a solution is unique for PNP, we omit something e.g. node, then test if the solution is still possible. Here we have a difference: omitting a node also omits all the edges connected. And if the node on the other side of that edge was in the VC, it would be affected and may not be necessary anymore.*

*Solution: instead of omitting, add.*

Iterate over all the nodes. Keep a counter which will contain the number of vertices which are in any one of the many solutions.

* For each vertex, create a new vertex, and connect these two together: so this new vertex has no other connections. Now run VC(D) for the minimum k we found.
  + If there is still a vertex cover of size k, then there exists a covering which does not include the new vertex (as otherwise it would increase to k+1). So if this covering does **not** include the new vertex, it must include the only vertex it is joined to - the vertex under test (so that the edge joining them is covered).
  + Likewise, If there is no longer a vertex cover of size k, the vertex under test was not in any covering of this minimum size.
  + If the first case happens (still k), increment the counter, as we have found a vertex in any one of the vertex covers.
  + Repeat.
* We now have a total - understand: this total is the number of vertices in **any one** (union) of the possible solutions, since for any vertex it could switch to any one of its solutions to include the vertex under test: thereby maintaining a VC of size k. So if this is larger than k, we have non-uniqueness, so reject. If it is k exactly: success.