**Feel free to criticise and add on comments**

3ai. A graph is planar if and only if it does not contain a subgraph that is homeomorphic to K5 or K3,3.

3aii. Non-planar graph: K5 or K3,3 are trivial examples. No need to make it hard on yourself.

3aiii. As K3,3 is homeomorphic to K5, we can reduce this problem to just considering K5. For a 4-node graph G to be homeomorphic to K5, then a subgraph of G must be derivable from K5 by reducing a pair of arcs to a single arc.

///// Nope, out of ideas.

By Kuratowski’s theorem, a graph is planar iff it does not contain a subgraph homeomorphic to K5 or K3,3. Any subgraph of a 4 node graph must have <= 4 nodes.

3bi. An isomorphism from G1 to G2 means that there exists a pair of bijective functions f: nodes(G1) -> nodes(G2) and g: arcs(G1) -> arcs(G2) such that for all a in arcs(G1) with endpoints n1 and n2, g(a) will have endpoints f(n1) and f(n2).

3bii. Rn has 2n automorphisms.

Explanation:

Consider G1, labelled A, B, C, D, E. Selecting a node n1 to map, we have 5 positions we can map this node to. Fixing n1, we select another node n2 next to it. We can see that we only have 2 options for the position of n2 (if we fix B its original position, we know that C must be either be in the position of node A or node C). If we also fix C to position C as an example, then we know that everything else must go back to its original position, so this is the identity map.

Hence, fixing the positions of two nodes will determine the rest of the nodes, so the number of automorphisms we have is an initial n positions multiplied by the two options we have after to determine the direction of where the nodes go.

3ci. A graph G is connected if for all x,y in nodes(G), there exists a path from x to y.

3cii. Show by induction that a connected graph with n nodes has at least n – 1 arcs.

Base case: Graph with 1 node. Trivial.

Inductive Step:

For our inductive hypothesis, take that a connected graph with n nodes has at least n – 1 arcs to be true.

For a connected graph G of n + 1 nodes, we can break it into cases.

If there exists some node connected by only 1 arc, we can delete it to get a connected graph G’ with n nodes, which must have at least n - 1 arcs. Hence, G must have at least n arcs, so this proves our inductive step in this case.

If not, then the connected graph G with n + 1 nodes only has nodes of degree greater than 1 (if a node has degree 0, then G is not connected, so we know that the degree of each node is at least 2). Hence, the total degree of G is at least 2n, which amounts to n arcs required to fulfill this.

Hence, it is true that for all n, connected graphs with n nodes have at least n – 1 arcs.

3di. A graph G is 2-colourable such that for all n in nodes(G), n can be coloured one of two colours such that no two nodes adjacent to each other are coloured the same colour.

3dii. A 2-colourable graph G can be rearranged into a bipartite graph, with disjoint sets X and Y, such that X contains nodes of 1 colour, and Y contains nodes of the other colour by the 2-colouring.

///// Do we need G to be simple? With loops, this breaks what we’re showing.

By definition a 2 colourable graph cannot have loops since no adjacent nodes can be of the same colour. I think wants the graph to be simple otherwise you have no way of putting an upper bound on the number of arcs? You can just draw an infinite number of parallel arcs between 2 nodes. Assuming it’s simple: the maximum number of arcs are when every node in 1 colour connected to every other node in the other colour. Suppose we have p in 1 and q in 2. Hence, p = n - q and max no. = p \* q. To maximise p \* q it is clear by differentiation that p = q when p \* q is max. This is because we get p \* q = (n - q) \* q = nq - q^2. Differentiate this to get n - 2q since n is a constant. Maximum point when n = 2q hence n = 2p hence p = q = n/2. Hence the max is floor(n/2) \* ceil(n/2) = floor(n^2/4).

///// I didn’t use the triangle inequality which is why I feel like I’m missing something

4ai. IS takes an input list and will sort the elements from left to right within the list, shifting each unsorted element into the necessary position. The left part of the list is composed of sorted elements, whilst right is unsorted, and the algorithm continues until the final element is inserted into its correct position.

4aii. Worst case complexity is **O(n^2)**. This is when the input list is in reverse order. This is because the IS insertion algorithm will require each element to be swapped to the front, requiring the maximum number of comparisons. So we will need 1 + 2 + 3 + … + (n – 1) comparisons. This arithmetic series is equivalent to (n – 1) (n / 2) comparisons we will need. This is of n^2 complexity.

4aiii. [5,4,3,2,1].

We say that 5 is sorted already, requiring 0 comparisons. To sort 4, we compare it to 5, and know that we have to swap it. Our list should now be [4,5,3,2,1], and our sorted sublist is [4,5]. We sort 3, and this requires the 2 comparisons to go at the front. We sort 2, requiring 3 comparisons. Similarly for 1, requiring 4 comparisons. This is 0 + 1 + 2 + 3 + 4 = **10 comparisons in total**.

aiv. Consider our input list L = [5,2,3,4,1].

We can break it into a simpler problem of [2,3,4] and then adding more complex things onto it. Sorting [2,3,4] only requires only 2 comparisons to confirm it is sorted (3 with 2 to insert 3, 4 with 3 to insert 4 into the sorted sublist of [2,3]). Generalised to a size of (n – 2), this is (n – 3) comparisons. If we are to sort [5,2,3,4], this is one additional swap for each of our (n – 2) elements that we have to make as every element has to be checked with 5 initially. The number of comparisons to sort [5,2,3,4] can then be generalised to (n – 2) + (n – 3) = 2n – 5 comparisons. To sort 1, we will need to compare 1 to every element of [5,2,3,4] as we know that 1 has to be swapped to the front. Our IS algorithm still has to do the same old steps, so this is (n – 1) + (n – 2) + (n – 3) = **3n – 6 comparisons** that we have to do in total for any list of this format.

4bi. For a sorted list of size n, our binary search will make 1 comparison with the middle term of the list to check whether our search term is the middle element, left of it, or right of it.

If the search term is our middle element, we can stop performing comparisons.

If we know to search in the left half of the list, we can exclude the middle element and the right half of the list and apply our binary search algorithm to this. Similar argument for the right half. Applying binary search again on either of these halves means we have to make more comparisons, so is our worst case.

Hence, we can define the recurrence relation to be **W(n) = 1 + W(floor(n/2)) from the reasoning, with a base case of W(1) = 1**.

4bii. B(n) = 1 + B(floor(n / 2)) = 2 + B(floor(n / 4)) = 3 + B(floor(n / 8)) = …

From this, if n = 2^k, then B(n) = k + 1 = log2(n) + 1. In the case where n = 2^k + m, e.g n = 9, we can see that the flooring handles our problem of remainders, so our generalised formula for B(n) is floor(log2(n)) + 1 = ceil(log2(n)).

4ci & 4cii.The worst case for binary search is ceil(log2(n)) by (ii) when the item we are to find is at the start or end of the list. Our Insertion Sort algorithm requires us to be shifting the elements into place from the back into our sorted list, so we can immediately see that the worst case for Binary IS would be the case of a list in reverse order. All the elements would need to be moved to the front through shifting along inside the sorted list e.g [2,5,4] -> [2,4,5], but because the Binary Search determines the position we need to move to, we do not need to make any comparisons when shifting.

The list is built up from sorting 1, then 2, then 3, … up to (n – 1) elements first before we insert the nth element, so our recurrence relation is iterative.

Hence, W(1) = 0 and W(n) = ceil(log2(n)) + W(n – 1).

4ciii. From our recurrence relation, we can tell that Binary IS is a O(nlog(n))) problem through expansion, and noticing the arithmetic series. This is shown below.

W(n) = ceil(log2(n)) + W(n - 1) = ceil(log2(n)) + ceil(log2(n - 1)) + ceil(log2(n - 2)) + ceil(log2(n - 3)) + ….

We can bound the ceiling’d log terms to simply ceil(log2(n)). We have (n - 1) of these.

Then W(n) is bounded by <= (n - 1) \* (ceil(log2(n))) = (n - 1) \* (log2(n) + 1) = (n - 1) + (n - 1) (log2(n)).

We can ignore (n - 1), so W(n) is O(nlogn), which is better than O(n^2).