Problem 1 - 2-ary Fixed Point

Let's begin by tackling the easy part: for 2-ary Fixed Point (2FP) to be in NP, it has to have answers that are checkable/verifiable in polynomial time (at least, for those answers which are correct). This is clearly the case: all y depend on at most 2 x, and any series of operations on those at most 2 x can be boiled down to a single one of the 16 truth tables for two boolean variables. Thus, given a certain configuration of the n bits $x_1...x_n$, returning the corresponding configuration of $y_1...y_n$ is n operations, and the comparison of the two lists is another n, taking $O(n)$ time, and as the complexity for returning a positive confirmation to a given solution is polynomial, $2FP$ is in NP. (2FP is also a decision problem and has the nice characteristic of having its wrong answers also confirmable as wrong in linear time - any incorrect set of x booleans can also be computed into the corresponding y in n and rejected as a different series of truth values in n.)

Onto the hard bit. Proving $2FP$ is in NP-hard will require a demonstration of the property that some other known NP-complete problem can be reduced to $2FP$ in polynomial time - this will effectively show that all instances of that other problem correspond to some instances of $2FP$ and thus that in solving $2FP$ we are also solving that other problem.

Let's give 3-SAT a try as our other problem. We know that 3-SAT is NP-hard via our lectures / the Kleinberg-Tardos text, and thus if we can demonstrate that 3-SAT $\leq_P 2FP$ we can show that $2FP$ is also NP-hard. 3-SAT is a decision problem with m clauses (here marked $C_1, C_2, C_3...$ etc.), each of which has 1, 2, or 3 boolean variables in it or their negations, combined using only the operator OR. A 3-SAT problem returns "yes" if there exists a configuration of booleans such that all clauses are true, and "no" if such a configuration does not exist. The algorithm for solving 3-SAT using 2FP is as follows:

1. For each 3-SAT clause $C_1, C_2...C_m$, with variables x_1, x_1 and x_2, or x_1 and x_2 and x_3, construct the following 2FP problem:
   (a) If clause C_n $(1 < n \leq m)$ has 1 variable (x_1), construct a 2FP circuit with inputs x_1 and x_{C_n} and outputs y_1 and y_{C_n}, as well as a single XNOR operator. Attach an edge (x_1, y_1), and edges $(x_1, XNOR)$ and $(x_{C_n}, XNOR)$, adding a NOT onto the edge from x_1 to XNOR if the 3-SAT clause has x_1's negation as its sole term instead of just x_1. Finally, add an edge $(XNOR, y_{C_n})$.
   (b) If clause C_n has 2 variables, x_1 and x_2, construct a 2FP circuit with inputs x_1, x_2, x_{12} and x_{C_n}, and outputs y_1, y_2, y_{12}, and y_{C_n}, as well as an XNOR operator as before and an OR operator. The edges here are $(x_1, y_1), (x_2, y_2)$, and then (x_1, OR) and (x_2, OR), adding NOTs onto the corresponding edge as required by whether or not that term is negated in the 3-SAT clause, and then (OR, y_{12}). No edge needs to be added between x_{12} and y_{12}, as the result dictated by OR to y_{12} is also dictated to x_{12} (else they'd be different and we wouldn't have a fixed point). Lastly, add edges $(x_{12}, XNOR), (x_{C_n}, XNOR)$, and $(XNOR, y_{C_n})$.
   (c) The drill is sort of clear from here, but if clause C_n has 3 variables, x_1, x_2, and x_3, construct a 2FP circuit with inputs x_1, x_2, x_{12}, x_3, x_{123}, and x_{C_n}, outputs $y_1, y_2, y_{12}, y_3, y_{123}$, and y_{C_n}, as well as operators OR_1, OR_2, and $XNOR$. Add edges $(x_1, y_1), (x_2, y_2)$, and edges $(x_1, OR_1), (x_2, OR_1)$, adding NOTs onto the latter two as the clause requires, and (OR_1, y_{12}). Then add (x_3, y_3), as well as (x_{12}, OR_2) and (x_3, OR_2), adding NOTs onto the latter one of those as the negation on x_3 in the clause requires, and then to wrap things up, add $(OR_2, y_{123}), (x_{123}, XNOR), (x_{C_n}, XNOR)$, and $(XNOR, y_{c_n})$.
2. Now, you should have m 2FP problems of input/output cardinality of 2, 4, or 6 each. As the boolean variables which are shared between clauses are standardized (i.e. you can't have x_5 be 0 in one clause and 1 in another at the same time), those variables shared between 2FP problems must also be standardized - the inputs to different problems that use the same variable should have edges from the same input node (e.g. if x_5 is in C_2 and C_7, those problems' inputs on x_5 are edges from the same x_5 node, which must have the same value in both problems). Run the 2FP black box on the resultant m 2FP problems - if there is a configuration of the variables that holds fixed points in each 2FP problem, return "yes", as this configuration will also satisfy all 3-SAT clauses. If any 2FP problem is without a fixed point, return "no".

Okay, to unpack. It may be easier to work backwards, considering that all but the last input and output of each clause work in sort of easier to understand ways (there's either an input-to-output direct connection, or the necessity for an input to maintain a fixed point with an output that has a value dictated to it by an OR makes the corresponding input have to be the same value as the output). Besides, the really interesting bit of the algorithm is the bit about XNOR - in which x_{123} (or x_{12}, or x_1, depending on the size of the clause) and x_{C_n} are pumped into an XNOR which feeds the result into y_{C_n}. x_{123} or whatever the combination of all the previous ORs is for this particular clause is either 0 or 1 - depending on whether or not the given configuration of booleans works for the ORs of the boolean variables or their negations - but x_{C_n} is an unknown element. The algorithm is designed to have the yes or no answer to the 2-ary fixed problem be the answer to the corresponding 3-SAT problem - and if the given configuration of booleans doesn't work for this particular clause, the corresponding 2FP problem with that configuration should not have a fixed point. Hence XNOR, which has the truth table (0, 0) = 1, (0, 1) = 0, (1, 0) = 0, and (1, 1) = 1. If x_{123}'s value is 0, indicating a failure for the given configuration, x_{C_n} must be the opposite of y_{C_n}, according to $y_{C_n} = x_{123}\ XNOR\ x_{C_n}$. If x_{123} is 1, however, x_{C_n} must be the same as y_{C_n}, and there exists a fixed point for that particular problem with that particular configuration. The black box for 2FP, of course, checks all configurations and thus may get multiple instances where a particular clause/problem has a fixed point and multiple where it doesn't - what matters is that if no configuration can create a fixed point for all clauses, the algorithm returns no - and the XNOR at the end makes it do just that.

Let's go back and justify the assumption that x_{123} (or x_{12}, or x_1) represents the actual truth value of the clause for a given configuration - and in the process fulfills all but the last input/output truth requirement. Note that whatever the configuration of the up-to-three variables-as-inputs in the problem is, their matching outputs are forced to be the same: that is to say, $x_1 = y_1$, $x_2 = y_2$, and $x_3 = y_3$, as a result of the edges directly connecting those inputs with those outputs. All that's left to consider, then, is the hybrid inputs/outputs and what they mean for the completeness of the fixed-point problem and what x_{123} or its counterpart represents. Note that y_{12} is the result of the OR of the variables x_1 and x_2, or of one or both of their negations. This means that the value that y_{12} has is 1 if and only if one of the inputs (or their negations, depending on whether the term or the term's negation is in the clause in that particular place) or both of them send a 1 value to the OR operator. If two 0s end up in the OR operator, $y_{12} = 0$. Regardless, to continue with the assignment of fixed points, $y_{12} = x_{12}$, or else the 2FP problem fails immediately - the value thus gets propagated down to x_{12}. This then gets dumped into an OR with x_3 or $NOT x_3$, whatever the case from the clause may be, and the value from that OR ends up in y_{123}, which is propagated down to x_{123} by the same need to maintain input/output sameness. The x_{123} input thus holds the result of the original clause in the 3-SAT problem given a particular configuration of booleans, which is then XNOR'ed with x_{C_n} to make (if $x_{123} = 1$, indicating a satisfied clause) or break (if $x_{123} = 0$, indicating an unsatisfied clause) the fixed point across x_{C_n} and y_{C_n}, as detailed above.

It is thus clear that if there is a successful configuration of booleans for a given 3-SAT clause, the corresponding 2FP problem as constructed above will return "yes", as it will have equivalent lists for its input and output. This fact combined with the application of this algorithm to each clause means that only a configuration that can satisfy all clauses can return all "yes"es for each 2FP problem - and the overall "yes" can only be returned if all problems come back with equivalent input/output lists. That is to say, *a configuration that satisfies a set of 3-SAT clauses will also satisfy the corresponding set of 2FP problems*, meaning a "yes" in 3-SAT is a "yes" in 2FP. On the flipside, any configuration of booleans that cannot satisfy a clause makes the final x_{123} (or x_{12}, or x_1) equivalent to 0, which, through the use of an XNOR gate, disrupts the corresponding 2FP problem, having it return "no". Thus, a "no" answer for a clause (which becomes a "no" for the whole set of clauses) corresponds - always - to a "no" answer for the equivalent 2FP problem, which becomes a "no" for the whole set of 2FP problems.

In reverse, we can see that any "yes" answer for a set of 2FP problems means that there's a "yes" for the corresponding set of 3-SAT clauses - I have shown that a 2FP problem only returns "yes" if its final XNOR gate has equivalent y_{C_n} output and x_{C_n} input, which can only happen if x_{123} - the other XNOR input - is 1, and I have shown that that only occurs if somewhere along the line beforehand, a term forwards a 1 into an OR. This is equivalent to saying that that individual 3-SAT clause has some term which is a 1 - satisfying the clause at large. This 2FP algorithm must have all its problems satisfied to result in "yes", corresponding to the satisfaction of all 3-SAT clauses for a "yes". Anything short of this (i.e. more than 0 $x_{123} = 0$) is a "no", as it leaves a 2FP problem without a fixed point, corresponding to a 3-SAT "no", as one of its clauses had all 0 as the results of its terms. As 3-SAT can be solved by polynomial (indeed, m, one per clause) calls to 2FP after the construction of circuits (also polynomial - a one-variable clause is 4 nodes and edges, a 2-variable clause is 8 nodes and edges, and a 3-variable clause is 12 nodes and edges [each discounting the addition of a NOT operator as a vertex]) in this manner, we can determine that $3 - SAT \leq_p 2FP$ and thus that 2FP is NP-hard. That said, we already showed that solutions can be verified as true solutions in 2FP in polynomial time, so 2FP $\in$ NP. These together prove that 2FP is NP-complete.

Problem 2 - Densest Bipartite Induced Subgraph

Here, too, proving that the densest bipartite induced subgraph problem is in NP is as simple as showing that given a positive solution, checking its positiveness is a polynomial operation. This is pretty clearly the case: given any particular subgraph induced by a subset of the vertices, it's very simple to check to see whether or not the number of edges in the subgraph divided by the number of vertices exceeds some given rational number x - just go through whatever representation the subgraph is in and count. This is also a decision problem, and thus not only the "yes" answer is easily verifiable given a particular subset, but so is the "no" answer - any subgraph only has one measure for density, and this is easily retrievable just by counting edges and dividing by vertices.

Okay, so the DBIS problem is in NP. What about NP-hard? If we can find some algorithm which exploits a polynomial use of DBIS so that all "yes" instances of the other problem result in "yes" being output by the algorithm's use of DBIS (and vice versa).

As for the problem itself, let's use the independent set decision problem. This essentially asks whether or not, given a particular graph, there exists an independent set containing k vertices - $IS(G, k)$ - where an independent set, to refresh, is one in which no edges in the graph as a whole have two vertices in the independent set as endpoints (one or none is fine). The algorithm exploiting a black box solver B for DBIS to solve the IS decision problem given G and k looks like this:

1. Create a new graph G' containing all the vertices and edges in G, but with the addition of $|V|$ new vertices, doubling the number of vertices. Each new vertex is connected by a single edge to *each* member of the original graph G, adding $|V|^2$ edges between the new $|V|$ vertices and the old ones. (e.g. if $|V| = 5$, G' has 10 vertices, and the 5 new ones have 25 edges connecting them to the original vertices between them - 5 per new vertex).
2. Call the DBIS black box on the new graph with the rational x as $\frac{nk}{n+k}$ - $B(G', \frac{nk}{n+k})$ - where n is the number of vertices in the original G and k is the input for the given k in the call to the Independent Set problem (i.e. the size of the desired independent set). $B(G', \frac{nk}{n+k})$ will return a yes or no depending on whether or not $\frac{|(u,v) \in E : u \in S, v \in S|}{|S|} \geq \frac{nk}{n+k}$ - which is also the answer to $IS(G, k)$.

Okay, let's pick apart why. The construction of the new graph automatically points towards a bipartite partition using the new vertices - none of which are attached to each other and all of which are attached to every vertex in the original graph. We want our call to the DBIS black box to confirm whether or not an independent set of size k exists in the original graph - or, really, we want the answer of whether or not some sufficiently dense subgraph exists to also be the answer of whether or not an independent set of sufficient size exists.

The DBIS black box need not actually, in its operation, find the densest bipartite induced subgraph in the augmented graph above - it need only say whether a sufficiently dense subgraph exists. However, I will assume (and then retroactively justify) the assumption that the black box does in fact get the absolute densest in the graph, which is to say, the subset which can still return "yes" on the highest x value possible. In this assumption, the densest bipartite induced subgraph for any graph as described above (with the additional $|V|$ vertices and $|V|^2$ edges and all that) *cannot* be one internal to the original graph G - consider that any simple undirected graph with n vertices may have at most $\frac{n*n-1}{2}$ edges (projecting $n-1$ edges from vertex 1, $n-2$ from vertex 2 (as 1-to-2 is projected already)... and 1 from vertex $n-1$ sums to $\frac{n*n-1}{2}$ starting at 0, well established by a certain apocryphal story about Gauss). This is all well and good until it becomes clear that only two of the vertices in such a "full" graph can be used as a result of the bipartite constraint, and only a single edge between them. In fact, the maximum setup for a dense bipartite original graph is, well, one that's naturally bipartite, in which we can almost fill the graph up with edges, but designate two equal-sized or near-equal sized (for even and odd numbers respectively) sets of vertices, in which we connect all vertices in each set to all vertices in the corresponding set, and not to each other. In this situation, with n vertices in total, even n can establish $(\frac{n}{2})^2$ edges and odd n can establish $floor(n/2) * ceiling(n/2)$ edges. The resultant density of the edges across the bipartite sets divided by their collective vertices is the maximum number doable in-graph.

The new augmented graph paints a different option. n new vertices with n^2 new edges connecting them to the old graph makes a great opportunity to form a bipartite set - the n new vertices aren't connected to each other and so naturally form one half of the partition. The other half should be as many of the vertices in the original G that the DBIS algorithm can get its hands on - each original vertex on that side of the partition brings n edges across with it, after all, and so as long as $n \geq 1$, more vertices in the partition means greater density across it. In the best case where the original graph makes the best attempt at densest bipartite induced subgraph, the resultant density, as we saw above, is $\frac{(\frac{n}{2})^2}{n}$ or $\frac{n}{4}$. Using the augmented graph as described above, this case can pick $n/2$ vertices from one half of the original bipartite graph, and match each of them with the n new vertices added into the augmented graph: $\frac{n*\frac{n}{2}}{n+\frac{n}{2}}$. That's $\frac{n}{3}$, a better ratio than the best the original graph can do. Indeed, in the best case scenario for such a scheme, where the original graph is n nodes totally unconnected to each other, the DBIS black box can attach n nodes to n nodes using n^2 edges, for a total density of n as opposed to the best the original can do (0, as there aren't any edges internal to the original graph), and in the worst case scenario, where the original graph is "full" as described above (n nodes and $\frac{n*n-1}{2}$ edges), the DBIS black box can attach n new nodes to just one original in the graph and still get away with a density of $\frac{n}{n+1}$ as compared to the original graph alone, which can only connect 2 nodes with a single edge for $\frac{1}{2}$ (and of course DBIS cannot be run in any form without at least two nodes connected by an edge). So, in using the additional vertices as one side of the partition, the DBIS black box achieves the best density.

Interestingly, this means that the other side of the partition is the largest available independent set in the original graph. A bipartite graph is formed of two sets, each of which contains vertices all independent of each other. As the DBIS black box uses the added n independent vertices as one side of the partition, the other side (if DBIS gets the absolute densest subgraph) forms the largest independent set in the original graph. From here, we just have the question of proving adequacy, not maximality (like I said, I'd retroactively justify my assumption), to tackle.

The density adequacy argument $\frac{nk}{n+k}$ in the running of DBIS asks the following question - using all of the new n vertices in the augmented graph, and as many (call it m) vertices as possible in the old graph, constituting $n * m$ edges, and knowing that partitions of this form are categorically the densest for any given graph, is it possible to meet or exceed the bound set by $\frac{nk}{n+k}$, indicating the n new vertices in the augmented graph forming nk edges with the k required vertices to form a k-cardinality independent set in the original graph divided by the total number of vertices between the added and original vertices? If k is asking about the maximum-cardinality independent set in the original G or an independent set of a lesser cardinality, the answer to DBIS is yes. A DBIS tuned to find the absolute densest possible subgraph will find n vertices in the added group of vertices, k in the old group (which necessarily form an independent set), and the nk edges between them - and thus the argument $\frac{nk}{n+k}$ into DBIS satisfies the density of an extant subgraph (the largest!) in G', $\frac{nk}{n+k}$, returning "yes". It's clear to see that if k is lesser, the argument into DBIS becomes more permissive (i.e. lesser), as it's fairly simple to show that n, which must be at least 1, has n subtracted from it - the numerator of the argument - if k is decremented, while the denominator $n + k$ just has 1 subtracted if k is decremented - these too should return "yes". On the flipside, should k be a number larger than the largest independent set (we'll call its cardinality l for the moment) in the original graph, DBIS will only be able to find the densest bipartite graph, which has density $\frac{nl}{n+l}$, but the argument will still be $\frac{nk}{n+k}$, and as we showed above, if $k > l$, $\frac{nk}{n+k} > \frac{nl}{n+l}$, returning a "no" for the DBIS call - just like with the known fact that there isn't a k-cardinality independent set.

In reverse, should DBIS return a "yes", it's because a k-cardinality set of vertices grouped into the partition on the old side of the graph, as included in the argument to DBIS, was found (whether or not the algorithm is optimal doesn't particularly matter, as long as it finds such a set) , satisfying the above need for a particular density on the subgraph. The corresponding $IS(G, k)$ is thus also "yes". If DBIS returns a "no" on $\frac{nk}{n+k}$, there isn't a large enough independent set in the original graph to achieve a high enough density, and thus the corresponding independent set problem should also return "no". It doesn't particularly matter whether DBIS *does* use the optimally dense subgraph, just that if the largest independent set has k vertices, DBIS *can* find the corresponding independent set using its method (and it will, as the only one that would be dense enough to accomodate $\frac{nk}{n+k}$ contains that independent set). Should the IS problem ask about less than k vertices, the DBIS algorithm might find the largest set again, or a smaller one - again, it doesn't particularly matter as long as the argument constraint is met. If no k-cardinality independent set exists, DBIS couldn't find a sufficiently dense set on the augmented graph anyway, and its optimality would be tested (but fail, which in this case is the 'correct' answer). DBIS is thus in NP-hard, as it is proven to be at least as hard as another NP-hard problem, IS. DBIS is also $\in NP$: therefore, DBIS is NP-complete.

To top all this off, this transformation is polynomial - from the original graph, add n vertices, then n^2 edges, and then run the polynomial DBIS algorithm a single time. This thus either runs in $O(n^2)$ time, or time equivalent to the DBIS black box's complexity - whatever the case, neither of them exceed the polynomial class.

Problem 3 - Vertex Cover

The problem's solution here is actually fairly simple. To find the minimum vertex cover for a particular graph, given a black box decision machine B which takes a graph and an (assumed to be positive, it'd sort of be hard to have negative vertices) integer k and returns "yes" or "no" depending on whether or not a vertex cover with k vertices is possible on the graph, use the following algorithm:

1. Call $B(G, k)$ on all k from 1 to n, stopping once the black box returns its first "yes" and storing that value k (the number of vertices in the minimum vertex cover).
2. Procedurally step through each vertex x in G, and call the black box on the subgraph induced by deleting that vertex and the edges that have it as one of their endpoints - that is to say, $G'(V - \{x\}, E - \{(u, v)$ where u or v is $x\})$ - and $k - 1$. Calling $B(G', k - 1)$ will check if a vertex cover of the rest of the graph (the subgraph induced by nixing x) is possible using $k - 1$ vertices, indicating that x is an acceptable member of the minimal vertex cover.
   Thus, if $B(G', k - 1)$ returns "yes", assign $G = G', k = k - 1$ and place x into the minimal vertex cover set, so that the edges connecting x to the rest of the graph as well as x itself are no longer in consideration and only the problem of finding a minimum vertex cover for the rest of the graph with one less vertex to spare remains. Move onto the next vertex. If $B(G', k-1)$ returns no, change nothing, and continue on to the next vertex (now knowing that it isn't possible to have a vertex cover for G of size k that includes x, because a vertex cover for G' of size $k - 1$ isn't possible.)

The explanation of the algorithm is more or less all in there - it's fairly clear that trying $B(G, k)$ on each k incrementing each time will return the cardinality of the smallest possible vertex cover, and the only remotely dubious part is the second step, in which possible subgraphs are tried. Here, essentially, each "successful" check against a potentially deleted vertex indicates that the subgraph induced by removing the vertex and its edges could be vertex covered by $k - 1$ vertices; this is true as a virtue of the fact that by repeated application of successful checks and deletions of vertices and edges, the graph will eventually come to a point where $k = 1$ and all that remains is a vertex (or multiple vertices, it doesn't really matter) which is connected to every other vertex. Taken from this point (and clearly, there are graphs where this is true - two vertices connected by an edge have a minimal vertex cover of one of the vertices, all star graphs have a vertex cover of the internal vertex, et cetera) one can see that adding vertices to the graph either increases the cardinality of the minimum vertex cover (by adding a vertex that isn't reachable from the single-vertex vertex cover) or it doesn't (by adding a vertex that is). The part of the algorithm that only removes vertices from the graph when the rest of the graph can accommodate the rest of the size of the minimal vertex cover is essentially doing this in reverse, and will eventually reach the single-vertex cover case. When it doesn't remove a vertex from the graph, it's because the rest of the graph cannot accommodate the rest of the size of the minimal vertex cover (i.e. no configuration of $k - 1$ other vertices in the vertex cover as well as x is actually a vertex cover), as indicated by a "no" response from B.

Finally, this algorithm runs in polynomial time - to figure out the initial value of k it calls B at most n times, and then in going through the vertices it calls B on each vertex once, making again at most n calls. As B is a polynomial operation, and a polynomial of a polynomial is a polynomial (trivial, when said polynomial is $2n$), the algorithm runs in polynomial time on the size of the graph.