##
1. Let k be the number of clauses in a boolean formula in 2CNF form, where each clause contains exactly two literals. One algorithm to decide if such a formula is satisfiable in polynomial time would be a progressive backtracking algorithm. The basic idea is to make progressive choices of the boolean value of each literal, checking all clauses follow certain conditions that would allow us to decide whether the formula is satidfiable or not. The algorithm proceeds as follows:

-Initially, start with no literal assignments.
-Assign a value to a literal in the following manner:
	-First, check if there is a clause where both literals are assigned values and the clause evaluates to False, backtrack to the last assignment choice and flip the assignment. If that choice was already flipped, then declare the formula to be unsatisfiable. [k steps]
	-Second, if there is a clause where one literal is False and the other is unassigned, assign the unassigned literal True. With that assignment, check for the first and second cases again. [k steps]
	-Third, check if there are any clauses where one literal is True and the other is unassigned. This is a new choice point. From this point on, any assigned literals are assumed to be correct. Make a choice for this literal and repeat the First and Second checks. [m steps]
	-If all literals have been assigned, perform a single pass over the clauses to verify the formula is satisfied [k steps]

	Analyzing the time complexity, for each literal choice we make, we can run steps 1 and 2 [2m*2k] times, where m is the number of literals and k is the number of clauses. Case 3 runs at most m times over all literals, with each literal assignment choice resulting in running steps 1 and 2 again. Therefore the time complexity is roughly (4m^2)k which is O(k) with the constant 4m^2.

##
2. We formulate this knapsack problem as a decision problem, in order to decide whether there exists a subset of items where their weight does not exceed W, but their total value is >= V. To show this problem is NP-complete, we show that the Subst-Sum <=p Knapsack.

We first prove that Knapsack is in NP. If we are given a subset, it is trivial to verify that the weights of the subset are <= W and the values of the subset are >= V. This can be done in linear time, therefore, the knapsack problem is in NP.

We now show that Subset-Sum <=p Knapsack. Each element in the given set has a value, v, and a weight, w. The goal is to choose a subset of these items that have a weight <= W and a value >= V. If we modify this problem slightly, and specify that the weights and values of each item are the same (i.e. v_i = w_i), thus requiring that the maximum weight and minimum value are equal (i.e. W=V), then this becomes an instance of the Subset-Sum problem. In this case, if our elements in our subset for subset-sum are [s_i...s_n], our items in the modified knapsack problem are equivalent to this set, where w_i=v_i=s_i. And for the target sum in the subset-sum problem, M, our max weight and min value are equivalent to this sum, where W=V=M. From this formulation, it follows that the sum of the weight of all items i<= W iff the sum of the value of all items >= V iff the sum of the subset of elements in the Subset-Sum problem are equal to M.

With this formulation of the knapsack problem, it is clear to see that the decision problem for Subset-Sum is reduceable in polynomial time to the Knapsack problem. Therefore, the Knapsack problem is NP-complete.

##
3. We construct an NFA N in polynomial time with states O(cn). We build the NFA to mimic the structure of the formula where each row represents a clause, each state represents a literal

in a clause, and each transition represents the literal being assigned to True or False. The structure of the NFA is the following:

We start in the state for the first literal of the first clause. If we read in the value of the first literal as False, we move to the next state in the clause. We read in the next symbol for the next literal assignment. If this assignment is False, we transition to the final state for this clause. If the next symbol is False, we transition to the accept state. The accept state represents the formula not being satisfied. If any of the symbols read for this clause are True, we transition to the first state of the next clause. This process continues for all clauses, with the last clause having no outgoing transitions if an assignment is True. This NFA accepts all non-satisfying assignments of the formula.

If we were to be able to minimize this NFA in polynomial time, then it must be true that P=NP. We have shown that an NFA can be create in polynomial time that accepts all non-satisfying assignments of some 3CNF formula. If we were to be able to minimize this NFA in polynomial time, we would obtain some minimized NFA M that would contain a single state. Since this is an equivalent NFA, if it accepts all binary strings (i.e. this state is an accept state with both transitions looping back to the same state), then we would reject the formula (stating that the formula is non-satidfiable). Otherwise we would accept. This cannot be the case, since we constructed the NFA in poly time, and reduced it in poly time, this would give a poly time algorithm to 3SAT, which would require P=NP.

##
(NOTE: uppercase U is used as the Union symbol)
4.
i. Given a certificate in the form of two sets of vertices (V_1, V_2), it takes poly-time to check if this set forms a BCB subgraph. We first check that |V_1| = |V_2| = K/2. This takes linear time. We then check that:

- Each vertex in V_1 has a connecting edge to every vertex in V_2
- Each vertex in V_1 has NO connecting edges to any vertex in V_1
- Each vertex in V_2 has NO connecting edges to any vertex in V_2

The total time complexity to check these three conditions is roughly (K/2)^2 with is O(K^2) = O(n^2) where n is the number of vertecies in G since K <= n. Since we can verify a certificate in poly-time, BCB is in NP.

ii. We assume we have an undirected graph G = (V,E) for an instance of CLIQUE, with some integer K. Here, we set K = |V|/2. In order to reduce CLIQUE to BCB, we construct G' = (V',E'), with K' defining an instance of BCB. We define V', E', and K' as follows:

-V' = V U E U W, were W is "K_choose_2" - K new variables. (K_choose_2 - K variables is equivalent to K(K-1)/2 - K new elements).
-E' = {{e,w}: e in E, w in W} U {{e,v}: e in E, v in V, v not an endpoint of e}
-K' = K_choose_2 = K(K-1)/2

The two "classes" or "sides" of the bipartite subgraph can be viewed as E' and V' U W. Suppose G has a clique C of size K. We look to find a subset of vertices B, and edges D, where |B|=K and |D|=K(K-1)/2. By our definintion of E', no vertex in B can be an endpoint in D. Therefore, all endpoints must exists in V-B, where |V-B|=K since we choose K=|V|/2. Now we have a set of vertices A = V-B, and connected edges D. With |A|=K vertices and |D|=K(K-1)/2 edges, this subgraph must contain a clique. Therefore, the graph G contains a clique iff G' contains a balanced-complete bipartite subgraph. Since we construct G' in poly-time from an instance of CLIQUE, and CLIQUE is NP-complete, BCB must also be NP-complete.

iii. Assuming that we have an algorithm to decide BCB, we show that the optimization problem BCB-o reduces to BCB. We aim to iteratively build a graph G' = (V',E') from the nodes V in graph G and test whether a BCB exists of size equal to |V'|. To do this, we start by taking a single edge from E' as the starting point for our new graph and add its endpoints from V to V'. We use our decision algorithm (which we will label as BCB-d) to query whether this graph forms a BCB by querying this graph with K=|V|=2. Then, for every edge remaining un-selected from E, add the vertex enpoints to the graph and query BCB-d with K=|V'|. If any vertex pair added to V' maintain a BCB, keep these vertices in V' and continue on with the rest of the

edges in E. We follow this iteration scheme, choosing a new edge at the beggining of each round until we have iterated through all edges, returning the graph vertices V' that is the largest. Since this algorithm makes use of BCB-d, running in poly-time on the order of O(n^2), BCB-o is NP-complete.

iv. Assuming that we have an algorithm for the BCB optimization problem, we show that the search problem BCB-s trivially reduces to BCB-o. To show this reduction, we simply modify a small part of the optimization algorithm. Instead of continuing the iterations through all egdes in E, once a round finds a BCB with |V'| >= K, we return this vertex set as a solution. This is an early-stopping version of the optimization algorithm, but also runs in O(n^2) using BCB-d.