601.432/632 Declarative Methods
Answers to "SAT Basics" discussion problems

```
1.  9 variables.
    Bb = Ms Baker is the baker, etc.

     ~Bb ^ ~Gg ~ ^Mm        # each avoided obvious profession
     ^ ~Gm                  # Mr. Gardner talks to the mayor, so they're !=
    ^ exactlyone(Bb,Bg,Bm) # Ms. Baker has one job
    ^ exactlyone(Gb,Gg,Gm) # Mr. Gardner has one job
    ^ exactlyone(Mb,Mg,Mm) # Ms. Mayer has one job
    ^ exactlyone(Bb,Gb,Mb) # there's one baker
    ^ exactlyone(Bg,Gg,Mg) # there's one gardener
    ^ exactlyone(Bm,Gm,Mm) # there's one mayor

    where exactlyone(X,Y,Z) is short for
    (X ^ ~Y ^ ~Z) v (~X ^ Y ^ ~Z) v (~X ^ ~Y ^ Z)

    or alternatively for
    (X v Y v Z) ^ ~(X ^ Y) ^ ~(X ^ Z) ^ ~(Y ^ Z)

    In case you were wondering,
        Ms. Baker is the mayor,
        Mr. Gardner is the baker,
        Ms. Mayer is the gardener.
    But that's not the point of the problem!

2.  (a) Original: (A ^ B) v (C ^ (D v E v F))  (disjunction of 2 CNF formulas)

        Recall that v distributes over ^:
            X v (Y ^ Z) = (X v Y) ^ (X v Z)

        Here, we have:
            X = (A ^ B)
            Y = C
            Z = (D v E v F)

        Step 1:
            ((A ^ B) v C) ^ ((A ^ B) v (D v E v F))

        We can distributed again and obtain the solution:
            (C v A) ^ (C v B) ^ (A v D v E v F) ^ (B v D v E v F)

    (b) Original: (A ^ B) v (C ^ (D v E v F))  (disjunction of 2 CNF formulas)

        Recall how switching variables work, given:
            X v (Y ^ Z)
        we introduce a switching variable S such that:
            (S -> X) ^ (~S -> (Y ^ Z))

        Here:
            1. (Z -> (A ^ B)) ^ (~Z -> C ^ (D v E v F))          (switching var)
            2. (~Z v (A ^ B)) ^ (Z v (C ^ (D v E v F))           (eliminate -> )
            3. (~Z v A) ^ (~Z v B) ^ (Z v C) ^ (Z v D v E v F)  (distribute v over ^)

    (c) (1 point)   1 solution:
            namely A=true, B=true

        (2 points)  7 solutions:
            total solutions: 2^3 = 8
            1 wrong assignment: A=false, B=false, C=false

        (2 points)  7 solutions:
            there is 1 solution for C, and
            we have already seen 7 solutions for (D v E v F),
        and they combine freely (1 * 7 = 7)

    (3 points)   37 solutions:
        one way to see this:
            1 * 2^4 solutions       where (A ^ B) is true
                                    and (C ^ (D v E v F)) could be anything
            2^2 * 7 solutions       where (A ^ B) could be anything
                                    and (C ^ (D v E v F)) is true

            but this double-counts the following cases:
                1 * 7 solutions     where (A ^ B) is true
                                    and (C ^ (D v E v F)) is true

            so subtracting off the double-counting, we have
                1 * 2^4 + 2^2 * 7 - 1 * 7 = 37

        another way to see this:
            1 * (2^4-7) solutions   where (A ^ B) is true
                                    and (C ^ (D v E v F)) is false
            (2^2-1) * 7 solutions   where (A ^ B) is false
                                    and (C ^ (D v E v F)) is true
            1 * 7 solutions         where (A ^ B) is true
                                    and (C ^ (D v E v F)) is true

            so we have a total of
                1 * (2^4-7) + (2^2-1) * 7 + 1 * 7 = 37
```

(d) Part (b) has more satisfying assignments than part (a) because it has more variables: it also has the variable Z.

The switching variable construction just guarantees that part (b) is SAT if and only if part (a) is SAT. But the number of satisfying assignments does change.

In fact, here is an even more extreme example of how adding variables can lead to more satisfying assignments. Consider the original formula

```
    (A ^ B) v (C ^ (D v E v F))
```

but now regard it as a formula over A, B, C, D, E, F, Z, where Z just doesn't happen to appear at all in the formula and therefore isn't constrained at all. So if there are 37 satisfying assignments to (A,B,C,D,E,F), there are 37*2 satisfying assignments to (A,B,C,D,E,F,Z), since Z can be anything at all.

Or if you don't like the idea of having variables that don't appear in the formula, try this version, which mentions Z but doesn't really constrain it since it allows either value:

```
    ( (A ^ B) v (C ^ (D v E v F)) ) ^ (Z v ~Z)
```

Again, there are 37*2 satisfying assignments.

Note that by adding an unconstrained variable, we are DOUBLING the number of assignments. This doesn't change whether the formula is SAT, since doubling a positive number (SAT) gives a positive number, while doubling zero (UNSAT) gives zero. It is in this sense that the formula specifies the same SAT problem even though the number of assignments changes.

Now, how about the case of a switching variable, which is more constrained than in the extreme example above? Let's look at the formula from part (b), step 1:

```
    (Z -> (A ^ B)) ^ (~Z -> C ^ (D v E v F))  (switching var)
```

In terms of our answer to part (b), we now have

```
    1 * 2^4 solutions   where Z is true
                        and (A ^ B) is true
                        and (C ^ (D v E v F)) could be anything
    2^2 * 7 solutions   where Z is false
                        and (A ^ B) could be anything
                        and (C ^ (D v E v F)) is true

In contrast to part (b), this DOESN'T double-count the 7 assignments
                        where (A ^ B) is true
                        and (C ^ (D v E v F)) is true
because they are now DIFFERENT assignments -- in one case,
Z is true, and in the other case, Z is false.  So we don't
have to subtract anything to correct for double-counting.
The total number of assignments is now
    1 * 2^4 + 2^2 * 7 = 44
i.e., 7 more than before.
```

```
3.  (a) Let's introduce new helper variables called evenABC, evenAB,
        evenA, as well as oddAB, oddA.

        The meaning of these variables: If evenABC is true, then we
        will only allow assignments that satisfy even([A,B,C]).
        Similarly for the other variables.

        Add these 10 constraints:
                   evenABC ^ C -> oddAB        oddABC ^ C -> evenAB
                   evenABC ^ ~C -> evenAB      oddABC ^ ~C -> oddAB
                   evenAB ^ B -> oddA          oddAB ^ B -> evenA
                   evenAB ^ ~B -> evenA        oddAB ^ ~B -> oddA
                   evenA -> ~A                 oddA -> A

        It is clear how to extend this scheme to handle
        evenABCDEFGHIJKLMNOPQRTUVWXYZ, for instance.

        Now add the unit clause evenABC .

        There are reasonable variations on this method.
        Some remarks:

            * It is okay to leave out the two oddABC constraints
            at the upper right, since they will never get used.

            * You could write ~evenAB instead of oddAB, but then
            you would still have to write clauses like
                oddAB ^ B -> evenA
            It's just that you would call them
                ~evenAB ^ B -> evenA

            Don't leave out those clauses!  If you leave them out,
            you can satisfy everything as long as you make C true
            Making C true does force you to make evenAB false
            (since evenABC is required).  But since you left out the
            clauses above, ~evenAB doesn't imply anything about anything:
            it would let you assign A and B however you like,
            including incorrectly making them both true or both false.

            * There are other ways of writing the constraints above:

            Instead of
                evenABC ^ C -> oddAB
                we could write
                evenABC -> ~C v oddAB
                since both are equivalent to the CNF clause
                ~evenABC v ~C v oddAB

                Or taking this a step further, instead of writing the two clauses
                evenABC ^ C -> oddAB
            evenABC ^ ~C -> evenAB
        we could write
            evenABC -> ~C v oddAB
            evenABC -> C v evenAB
        and then combine these into
            evenABC -> (~C v oddAB) ^ (C v evenAB)

        It is also possible to do this, which is similar:
            evenABC -> (~C ^ evenAB) v (C ^ oddAB)

        * As the base cases, instead of writing
            evenA -> ~A     oddA -> A
        it would be prettier to continue the pattern one more
        step and write
            evenA ^ A -> odd0       oddA ^ A -> even0
            evenA ^ ~A -> even0     oddA ^ ~A -> odd0
        and then take the base cases to be the unit clauses
            even0       ~odd0

        This would have the advantage that we could encode
        even([]), which is always true.

(b) O(n) and O(n). This is true in CNF as well.
```