```python
#!/usr/bin/env python
# -*- coding: utf-8 -*-


def make_formula(formula):
    """
    Constructs a formula from a list of list of integers.
    The integers become indices and negations is implemented
    by negativeness.
    """
    def make_name(l):
        return -l if l < 0 else l
    names = [make_name(l) for c in formula for l in c]
    names = sorted(list(set(names))) # removes duplicates the hard-core way :)
    variables = [Variable("x%s" % name) for name in names]
    def make_lit(l):
        name = make_name(l)
        index = names.index(name)
        var = variables[index]
        lit = Literal(var)
        lit.negated = l < 0
        return lit
    def make_clause(c):
        lits = [make_lit(x) for x in c]
        return Clause(lits)
    clauses = [make_clause(c) for c in formula]
    return Formula(variables, clauses)


class Formula(object):
    def __init__(self, variables, clauses):
        self.clauses = clauses
        self.variables = variables

    def __iter__(self):
        return iter(self.variables)

    def __len__(self):
        return len(self.clauses)

    def __getitem__(self, key):
        return self.clauses[key]

    def __repr__(self):
        return "{%s}" % ','.join([str(c) for c in self.clauses])

    def eval(self):
        for clause in self.clauses:
            if not clause.eval(): return False
        return True


class Clause(object):
    def __init__(self, literals):
        self.literals = literals

    def __iter__(self):
        return iter(self.literals)

    def __getitem__(self, key):
        return self.literals[key]

    def __repr__(self):
        return "{%s}" % ','.join(str(l) for l in self.literals)

    def eval(self):
        for lit in self.literals:
            if lit.eval(): return True
        return False


class Literal(object):
    def __init__(self, variable, negated=False):
        self.variable = variable
        self.negated = negated

    def __repr__(self):
        return "%s%s" % ("!" if self.negated else "", self.variable)

    def eval(self):
        return not self.variable.eval() if self.negated else self.variable.eval()


class Variable(object):
    def __init__(self, name, truth_value=None):
        self.name = name
        self.truth_value = truth_value

    def __eq__(self, other):
        return self.name == other.name

    def __repr__(self):
        return "%s" % self.name

    def assign(self, truth_value):
        self.truth_value = bool(truth_value)
        return self.truth_value

    def eval(self):
        return self.truth_value
```

```
################### ALGORITHM B ###################
#!/usr/bin/env python
# -*- coding: utf-8 -*-

from random import uniform


def max_sat(formula):
    tvs = [var.assign(int(uniform(0,2))) for var in formula]
    return sum([c.eval() for c in formula.clauses]), tvs





################### ALGORITHM C ###################
#!/usr/bin/env python
# -*- coding: utf-8 -*-

from algA import max_sat as algA
from algB import max_sat as algB


def max_sat(formula):
    return max(algA(formula), algB(formula))
```

```python
#!/usr/bin/env python
# -*- coding: utf-8 -*-

import sys
from StringIO import StringIO
from random import uniform
from cvxopt import matrix, solvers


def max_sat(formula):
    # solve the lp
    solution = approx_lp(formula)
    rands = [uniform(0,1) for i in range(len(solution))]
    # round randomized
    solution = [1 if r < s else 0 for s,r in zip(solution, rands)]
    [v.assign(solution[i]) for i,v in enumerate(formula)]
    return sum([c.eval() for c in formula.clauses]), solution


def approx_lp(formula):
    m = len(formula) # numer of clauses
    n = len(formula.variables) # total number variables

    # objective:
    # minimize z_1 + z_2 + ... z_m
    # = maximize -z_1 + -z_2 + ... + -z_m
    c = [1.0 for j in range(m)] + [0.0 for i in range(n)]

    # matrix for side conditions:
    # -1 if a literal is a negated variable, 1 otherwise -> see README
    def valueOf(var, clause):
        for lit in clause:
            if lit.variable == var:
                if lit.negated: return 1.0 # var negated in clause
                else: return -1.0 # var non-negated in clause
        return 0.0 # var not in clause
    def elemAt(i,j):
        var = formula.variables[j-m]
        clause = formula[i]
        return valueOf(var, clause)
    def col_part(j):
        if j < m:
            return unit(j, length=m)
        else:
            return [elemAt(i,j) for i in range(m)]
    def unit(j, value=1.0, length=n+m):
        return [value if i == j else 0.0 for i in range(length)] # unit vector j times value
    def column(j):
        return col_part(j) + unit(j, -1.0) + unit(j)
    A = [column(j) for j in range(n+m)]


    # vector for side condition:
    def nneg(clause): # computes the number of negations of a clause
        return sum([1.0 for lit in clause if lit.negated])
    b = [nneg(formula[j]) for j in range(m)] + [0.0 for x in range(m+n)] + [1.0 for x in range(m+n)]

    # solve it
    # opt_debug(A, b, c)
    sol = solvers.lp(matrix(c),matrix(A),matrix(b))
    return [x for x in sol['x']][m:]


def opt_debug(A, b, c):
    print "----------------------- BEGIN OPT DEBUG -----------------------"
    print "Objective function vector c:\n\t%s" % c
    print "Side condition vector b:\n\t%s" % b
    print "Sice condtion matrix A:"
    for row in A:
        print ' '.join([str(x).rjust(4) for x in row])
    print "------------------------ END OPT DEBUG ------------------------"
```