In [4]:

```
import pandas as pd

from fractions import Fraction

class F(fraction):
    '''
    Classe que representa um numero fracionario, extendendo fractions.Fraction, e implentando a lógica o "M grande"
    '''
    def __init__(self,n,m=Fraction(0)):
        self.fraction = Fraction(n)
        self.m = Fraction(m)

    def __repr__(self):
        """repr(self)"""
        return str(float(self.fraction)) if self.m == 0 else str(float(self.fraction)) + ' + (' + str(float(self.m)) + '*M)'


    def __str__(self):
        """str(self)"""
        return str(float(self.fraction)) if self.m == 0 else str(float(self.fraction)) + ' + (' + str(float(self.m)) + '*M)'

    def __eq__(self, f):
        ""a == b""
        if type(f) is not type(self):
            f = F(f)

        return self.fraction.__eq__(f.fraction) and self.m.__eq__(f.m)

    def __add__(self, f):
        ""a + b""
        if type(f) is not type(self):
            f = F(f)

        return F(self.fraction.__add__(f.fraction),self.m.__add__(f.m))

    def __sub__(self, f):
        ""a - b""
        if type(f) is not type(self):
            f = F(f)

        return F(self.fraction.__sub__(f.fraction),self.m.__sub__(f.m))

    def __mul__(self, f):
        ""a * b""
        if type(f) is not type(self):
            f = F(f)


        if f.m == 0:
            return F(self.fraction.__mul__(f.fraction))
        else:
            return F(self.fraction.__mul__(f.fraction),self.m.__mul__(f.m))

    def __div__(self, f):
        ""a / b""
        if type(f) is not type(self):
            f = F(f)

        if f.m == 0:
            return F(self.fraction.__div__(f.fraction))
        else:
            return F(self.fraction.__div__(f.fraction),self.m.__div__(f.m))

    def __lt__(self, f):
        ""a < b""
        if type(f) is not type(self):
            f = F(f)

        if self.m == f.m:
            return self.fraction.__lt__(f.fraction)

        else:
            return self.m.__lt__(f.m)

    def __gt__(self, f):
        ""a > b""
        if type(f) is not type(self):
            f = F(f)

        if self.m == f.m:
            return self.fraction.__gt__(f.fraction)

        else:
            return self.m.__gt__(f.m)

    def __le__(self, f):
        ""a <= b""
        if type(f) is not type(self):
            f = F(f)

        if self.m == f.m:
            return self.fraction.__le__(f.fraction)

        else:
            return self.m.__le__(f.m)

    def __ge__(self, f):
        ""a >= b""
        if type(f) is not type(self):
            f = F(f)

        if self.m == f.m:
            return self.fraction.__ge__(f.fraction)

        else:
            return self.m.__ge__(f.m)

class Tabela(object):
    '''
    Classe que representa e resolve um PPL, usando o metodo do "M grande", quando necessário.
    '''
    def __init__(self, funcaoObjetivo,restricoes=None):

        #preenche as linhas da tabela das restrições
        self.linhasRestricoes = []

        numeroDeVariaveisExtras = len(restricoes) + len([(c,t,tr) for (c,t,tr) in restricoes if t!='<='])

        #preenche a linha da função objetivo
        self.linhaFuncaoObjetivo = [1] + [c*(-1) for c in funcaoObjetivo] + [0]*numeroDeVariaveisExtras + [0]

        for i,(coeficientes,tipo,termo) in enumerate(restricoes):
            colExtras = [0]*numeroDeVariaveisExtras

            if tipo == '<=':
                colExtras[i] = 1

            elif tipo=='=':
                colExtras[i] = 1
                self.linhaFuncaoObjetivo[1 + len(coeficientes) + i] = F(0, F(1))

            elif tipo=='>=':
                colExtras[i] = -1
                colExtras[i+1] = 1
                self.linhaFuncaoObjetivo[1 + len(coeficientes) + i + 1] = F(0, F(1))

            self.linhasRestricoes.append(self._converteParaF([0] + coeficientes + colExtras + [termo]))

    def _converteParaF(self,lista):
        return [F(e) for e in lista]

    def printTabela(self):
        #tabela = [self.linhaFuncaoObjetivo] + self.linhasRestricoes
        df = pd.DataFrame([self.linhaFuncaoObjetivo] + self.linhasRestricoes)
        display(df)
        #print('\n', matrix([[str(f) for f in l] for l in tabela]))

    '''
    Encontra a coluna corresponte a variavel que entra da base e retorna seu indice
    '''
    def _encontraEntra(self):
        menorCoeficiente = min(self.linhaFuncaoObjetivo[1:-1])

        if menorCoeficiente >= 0:
            return None
        else:
            return self.linhaFuncaoObjetivo[0:-1].index(menorCoeficiente)

    '''
    Encontra a linha corresponte a variavel que sai da base e retorna seu indice
    '''
    def _encontraSai(self, colunaDoPivo):
        termos = [r[-1] for r in self.linhasRestricoes]
        coefisVariavelEntra = [r[colunaDoPivo] for r in self.linhasRestricoes]

        razoes = []
        for i,termo in enumerate(termos):
            if coefisVariavelEntra[i] == 0:
                razoes.append(F(1,1)) #Usando o M grande para garantir que essa razao e sempre maior e portando a variavel nao escolhida
                #razoes.append(99999999 * abs(max(termos)))
            else:
                razoes.append(termo/coefisVariavelEntra[i])

        menorRazaoPositiva = min([r for r in razoes if r > 0])

        return razoes.index(menorRazaoPositiva)

    '''
    Faz o pivoteamento, tendo como elemento pivo o elemento pi,pj da matrix de restrições
    '''
    def _pivoteamento(self, pi, pj):
        #elemento pivo
        p = self.linhasRestricoes[pi][pj]
        #divide a linha do pivo pelo elemento pivo
        self.linhasRestricoes[pi] = [x/p for x in self.linhasRestricoes[pi]]

        #para cada elemento da linha correspondente a F.O. multiplica cada elemento da linha pivo
        tempLinha = [self.linhaFuncaoObjetivo[pj]* x for x in self.linhasRestricoes[pi]]
        #subtrai cada elemendo da linha da F.O. pelo falor calculado acima
        self.linhaFuncaoObjetivo = [self.linhaFuncaoObjetivo[i] - tempLinha[i] for i in range(len(tempLinha))]

        #para cara linha corresponte a restricao i repete o mesmo procedimento feito na linha fa F.O.
        for i,restricao in enumerate(self.linhasRestricoes):
            if i != pi:
                tempLinha = [restricao[pj]* x for x in self.linhasRestricoes[pi]]
                self.linhasRestricoes[i] = [restricao[i] - tempLinha[i] for i in range(len(tempLinha))]

    '''
    Testa se a F.O. atual indica que a solução otima foi encontrada
    '''
    def _solucaoOtimaEncontrada(self):
        if min(self.linhaFuncaoObjetivo[1:-1]) >= 0:
            return True
        else:
            return False

    '''
    Executa o metodo simplex
    '''
    def executar(self):
        self.printTabela

        while not self._solucaoOtimaEncontrada():
            c = self._encontraEntra()
            r = self._encontraSai(c)

            self._pivoteamento(r,c)

            print('\nColuna do pivo: %s\nLinha do pivo: %s'%(c+1,r))

            self.printTabela()

    @property
    def variaveisDentroBase(self):
        dentroDaBase = []
        for c in range(1,len(self.linhaFuncaoObjetivo)-1):
            valoresColuna = [l[c] for l in self.linhasRestricoes]

            numDeZeros = len([z for z in valoresColuna if z==F(0)])
            numDeUms = len([u for u in valoresColuna if u==F(1)])

            if numDeUms == 1 and numDeZeros == len(self.linhasRestricoes) - 1 :
                dentroDaBase.append(c)

        return dentroDaBase

    @property
    def variaveisForaBase(self):
        return [i for i in range(1,len(self.linhaFuncaoObjetivo)-1) if i not in self.variaveisDentroBase]

    @property
    def solucaoOtima(self):
        if not self._solucaoOtimaEncontrada():
            self.executar()

        dentro = self.variaveisDentroBase
        fora = self.variaveisForaBase

        solucao = []

        for val in dentro:
            for l in self.linhasRestricoes:
                if l[val] == F(1):
                    solucao.append((val,l[-1]))
                    break

        solucao += [(val,F(0)) for val in fora]

        return [(t[0],float(t[1])) for t in solucao]

    @property
    def valorOtimo(self):
        if not self._solucaoOtimaEncontrada():
            self.executar()

        return self.linhaFuncaoObjetivo[-1]


def getNomeDeVariavel(index):
    return 'X' + str(index)

def toStringComNomes(lista):
    if type(lista[0]) is type(0):
        return [getNomeDeVariavel(i) for i in lista]

    elif type(lista[0]) is type(()):
        return [(getNomeDeVariavel(l[0]),l[1]) for l in lista]

if __name__ == '__main__':
    """
    max z = x + 2y

    s.a.    x +  y <= 4
            -x + y <= 2
            x,y >= 0

    """
    #t = Tabela([1,2],restricoes=[([1, 1],"<=", 4),([0, 1],"<=", 2)])

    """
    max z = 5x + 2y

    s.a.    x      <= 3
                y <= 4
            x + 2y <= 9
            x,y >= 0
    """
    t = Tabela([5,2],restricoes=[([1, 0],"<=", 3),([0, 1],"<=", 4),([1, 2],"<=", 9)])

    print("\nValor de Z Ótimo: %s (%s)" % (float(t.valorOtimo),t.valorOtimo))

    print("\nVariáveis Basicas: %s" % (toStringComNomes(t.variaveisDentroBase)))

    print("\nVariáveis Não Basicas: %s" % (toStringComNomes(t.variaveisForaBase)))

    print("\nSolução Ótima: %s" % (toStringComNomes(t.solucaoOtima)))
```

```
Coluna do pivo: 2
Linha do pivo: 0
```

	0	1	2	3	4	5	6
0	1	0	-2	5	0	0	15
1	0	1	0	1	0	0	3
2	0	0	1	0	1	0	4
3	0	0	2	-1	0	1	6

```
Coluna do pivo: 3
Linha do pivo: 2
```

	0	1	2	3	4	5	6
0	1	0	0	4	0	1	21
1	0	1	0	1	0	0	3
2	0	0	0	1/2	1	-1/2	1
3	0	0	1	-1/2	0	1/2	3

```
Valor de Z Ótimo: 21.0 (21)

Variáveis Basicas: ['X1', 'X2', 'X4']

Variáveis Não Basicas: ['X3', 'X5']

Solução Ótima: [('X1', 3.0), ('X2', 3.0), ('X4', 1.0), ('X3', 0.0), ('X5', 0.0)]
```

In [ ]: