```haskell
import Data.List (delete, sortBy)
import Data.Ord (comparing)
import Data.String (words)

----------------------------- DATA TYPES ---------------------------------------
type Number = Double

-- | An MKP item
--      tuple: (profit, capacities)
type Item = (Number, [Number])

-- | The MKP instance representation
--      tuple: (list of items, capacities)
type MKP = ([Item], [Number])

-- | The MKP solution representation
--      tuple: (selected itens, profit, weights)
type MKPSolution = ([Int], Number, [Number])

-- | Insert the given item on a MKP solution, updating its profit and weights.
addItem :: Item -> Int -> MKPSolution -> MKPSolution
addItem (itemProfit, itemWeights) idx (solIdxs, solProfit, solWeights) = (solIdxs’, solProfit’, solWeight’)
        where
        solIdxs’ = solIdxs ++ [idx]
        solProfit’ = solProfit + itemProfit
        solWeight’ = map (uncurry (+)) $ zip itemWeights solWeights
--------------------------------------------------------------------------------

-------------------------- DOMINATING SETS  ------------------------------------
-- | Answer if the first set dominates the second.
dominates :: MKPSolution -> MKPSolution -> Bool
dominates (_, p1, cs1) (_, p2, cs2) = betterProfit || dominateWeights
        where
        betterProfit = (p1 > p2)
        dominateWeights = or $ map (uncurry (<)) $ (zip cs1 cs2)

-- | Returns all dominating sets of a MKP instance.
domSets :: MKP -> [MKPSolution]
domSets (items, _) = domSets’ 1 items []
        where
        -- recusrively computes dominating sets
        domSets’ _ [] set = set
        domSets’ idx (it:items) sets = domSets’ (idx+1) items newSets
                where
                newSets = [x | x <- merged, and $ map (dominates x) (delete x merged)]
                merged  = sets ++ map (addItem it idx) sets ++ [([idx], fst it, snd it)]
--------------------------------------------------------------------------------


---------------------------------- SOLVING MKP ---------------------------------
-- | Solves the MKP using domating sets generation.
--     Among the feasible sets the most protitable is selected.
solve :: MKP -> MKPSolution
solve mkp = optimum
        where
        getProfit (_, p, _) = p
        dummySet = ([], 0, snd mkp)   -- for filtering
        feasibles = filter (not.(dominates dummySet)) $ domSets mkp
        optimum = head $ reverse $ sortBy (comparing getProfit) feasibles
--------------------------------------------------------------------------------
```