```
-- 1. RIDKE MATICE ------------------------------------------------------------

{-
Ridka matice je reprezentovana jako trojice (m,n,s), kde m je pocet radek,
n je pocet sloupcu a s je seznam trojic (i,j,a_ij) (kde i je cislo radky,
j je cislo sloupce a a_ij je nenulova hodnota) usporadany vzestupne podle
i a uvnitr radek podle j.

Naprogramujte funkce, ktere v teto reprezentaci realizuji

(a) transpozici matic
(b) nasobeni matic
(c) umocnovani matic (dobrovolne)
-}

-- datova struktura

-- (i,j,a_ij)
type RMVals = (Int,Int,Int)

-- (m,n,s)
type RM = (Int,Int,[RMVals])

-- sortovani trojic

sort3 :: [RMVals] -> [RMVals]
sort3 [] = []
sort3 (x:xs) = sort3 [a | a <- xs, a<- xs, a>=x]

-- transpozice

-- transpozice hodnot
transV :: [RMVals] -> [RMVals]
transV [] = []
transV ((a,b,c):xs) = (b,a,c):(transV xs)

-- transpozice matice

trans :: RM -> RM
trans (m,n,s) = (n,m,sort3 (transV s))

-- nasobeni matic


del0 :: [RMVals] -> [RMVals]    -- vynechani nulovych prvku
del0 [] = []
del0 ((a,b,c):xs) | c==0 = del0 xs
                  | otherwise = (a,b,c):(del0 xs)

-- z matice, ktera muze obsahovat vicekrat jeden prvek udela normalni
-- souctem hodnot techto duplicitnich prvku
-- tedy secte hodnoty stejnych prvku za sebou
simpM :: [RMVals] -> [RMVals]
simpM [] = []
simpM [x] = [x]
simpM ((xa,xb,xc):(ya,yb,yc):xys)
       | xa==ya && xb==yb = simpM ((xa,xb,xc+yc):(xys))
       | otherwise = (xa,xb,xc):(simpM ((ya,yb,yc):xys))


-- vynasobi matice s tim, ze format matice muze obsahovat nulove prvky
mulM :: [RMVals] -> [RMVals] -> [RMVals]
mulM xs ys = simpM (sort3 [(xa,yb,xc*yc)
       | (xa,xb,xc) <- xs, (ya,yb,yc) <- ys, xb==ya])  -- samotne nasobeni mul :: RM -> RM
-> RM
mul (m1,n1,s1) (m2,n2,s2) | n1==m2 = (m1,n2,s3)
```

```
                          where s3 = del0 (mulM s1 s2)

-- testovaci matice

t1 :: RM
t1 = (2,3,[(1,3,1),(2,1,2),(2,2,1)])

t2 :: RM
t2 = (3,4,[(1,1,6),(1,2,2),(2,1,1),(3,4,1)])

-- testy testovacich matic

r1 :: RM
r1 = trans t1

r2 :: RM
r2 = trans t2

r3 :: RM
r3 = t1 `mul` t2

-- 2. VYPOUSTENI Z BVS  ------------------------------------------------------

{-
Definujte prirozenou reprezentaci binarniho stromu, v jehoz uzlech je ulozena
informace nejakeho typu (podtridy Ord).

Naprogramujte funkci, ktera z binarniho vyhledavaciho stromu vypusti uzly
patrici do zadaneho intervalu (nejsou-li tam zadne takove, bude to identita).
-}


-- datova struktura

data BVSTree a = Nil | BVS ((BVSTree a),a,(BVSTree a)) deriving (Show, Eq)

-- najde min/max v BVS
findMin :: (Ord a) => BVSTree a -> a
findMin (BVS (left,val,right))
       | left==Nil = val
       | otherwise = findMin left

findMax :: (Ord a) => BVSTree a -> a
findMax (BVS (left,val,right))
       | right==Nil = val
       | otherwise = findMax right

-- vypusti ze stromu vsechny uzly od m do n
remBVS :: (Ord a) => BVSTree a -> a -> a -> BVSTree a
remBVS Nil _ _ = Nil

remBVS (BVS (Nil,val,Nil)) m n
       | m <= val && val <= n = Nil         | otherwise = BVS (Nil,val,Nil)  remBVS (BVS
(left,val,right)) m n =          if m>n then Nil
       else if n a -> [a] -> Bool
notMember _ [] = True
notMember y (x:xs)
       | y==x = False
       | otherwise = notMember y xs

-- resi danou ulohu
solve3 :: [Int] -> Int -> [Int]
solve3 xs k = take k [x | x <- [1..], x `notMember` xs]  -- testy solve3_test1 = solve3
[8,25,4,7,12,2,1,23] 16 solve3_test2 = solve3 [1,2,3,4,5,6,8,9,10] 1 solve3_test3 = solve3
[] 12     -- 4. 1-2 STROMY ----------------------------------------------------------------
{- Haskell: mam datovy typ reprezentujici 1-2 strom:  data T1 a = Nil | N1 a (T1 a) | N2 a
```

```
(T1 a) (T1 a)  Ukolem je napsat funkci fold typu:  b -> (a -> b -> b) -> (a -> b -> b -> b)
-> T1 a -> b

a funkci hodnota, ktera pomoci funkce fold projde zadany strom a vrati seznam
hodnot z konstruktoru N2 v poradi preorder - tj. vsechny hodnoty z vrcholu,
ktere maji dva potomky

Hint:
V tomto pripade byla fold definovana takto:

fold::b->(a->b->b)->(a->b->b->b)->T1 a->b

b nahradi vrcholy konstruovane Nil
na vrchol s konstruktorem N1 a (T1 a) zavola funkci (a->b->b)
a na N2 a (T1 a) (T1 a) zavola funkci (a->b->b->b)
-}


data T1 a = NilX | N1 a (T1 a) | N2 a (T1 a) (T1 a) deriving (Show, Eq)

-- funkce fold (=svinovani)
fold :: b -> (a -> b -> b) -> (a -> b -> b -> b) -> (T1 a) -> b
fold fNil fN1 fN2 NilX = fNil
fold fNil fN1 fN2 (N1 val next) = fN1 val (fold fNil fN1 fN2 next)
fold fNil fN1 fN2 (N2 val left right) = fN2 val (fold fNil fN1 fN2 left) (fold fNil fN1 fN2
right)

-- funkce hodnota pomoci fold
hodnota :: (T1 a) -> [a]
hodnota = fold [] (\ _ xs -> xs) (\ x xs ys -> (x:xs)++ys)


-- pro lepsi pochopeni jeste funkce pro seznam hodnot ve VSECH vrcholech
hodnota_all :: (T1 a) -> [a]
hodnota_all = fold [] (\ x xs -> (x:xs)) (\ x xs ys -> (x:xs)++ys)

-- testovaci strom

t12_t1 :: (T1 Int)
t12_t1 = N1 20
        (
          N2 13
          (
            N2 1
            (
              N2 21
              (
                N1 9 NilX
              )
              (
                N1 11
                (
                  N1 8 NilX
                )
              )
            )
            (
              N1 2
              (
                N1 5 NilX
              )
            )
          )
          (
            N1 4
            (
```

```
              N1 8
              (
                N1 7 NilX
              )
            )
          )
        )

{-
take se da zapsat jako

N1 20 (N2 13 (N2 1 (N2 21 (N1 9 NilX) (N1 11 (N1 8 NilX))) (N1 2 (N1 5 NilX)))
(N1 4 (N1 8 (N1 7 NilX))))
-}

-- testy
t12_test1 = hodnota t12_t1

-- 5. BATOH ------------------------------------------------------------------

{-
Je dan seznam A cislo N. Napiste funkci, ktera zjisti, zda je mozne poscitat
(nektere) prvky seznamu, aby soucet vysel N.
-}

-- implementujeme funkci, ktera tento seznam vrati, dana uloha pak by byla
-- pokud seznam neexistuje, vraci se []
batoh :: [Int] -> Int -> [Int]
batoh _ 0 = []
batoh [] _ = []
batoh (x:xs) n
 | sum newBatoh1 == n = newBatoh1
 | sum newBatoh2 == n = newBatoh2
 | otherwise = []
       where newBatoh1 = x:(batoh xs (n-x))
             newBatoh2 = batoh xs n

-- testy
b1 = batoh [1,58,3,9,2,1,85,2,6,51,8,69,21] 34
b2 = batoh [1,58,3,9,2,1,85,2,6,51,8,69,21] 35
b3 = batoh [1,58,3,9,2,1,85,2,6,51,8,69,21] 60
b4 = batoh [2,4,8,16,32,64,128] 91
b5 = batoh [1,2,4,8,16,32,64,128] 91
b6 = batoh [1,5..] 91
b7 = batoh [1,1..] 654

-- 6. PERMUTACE USPORADANI ---------------------------------------------------

{-
Je dan seznam A - seznam dvojic prvku, urcujici castecne usporadani. Vyrobte
seznam vsech permutaci puvodniho seznamu, ktere vyhovuji castecnemu
usporadani.
-}

-- vrati seznam bez jendoho prvku
without :: (Eq a) => [a] -> a -> [a]
without [] _ = []
without (x:xs) a
 | a==x = xs `without` a
 | otherwise = x:(xs `without` a)


-- not member vraci, zda prvek neni v seznamu
notMember1 :: (Eq a) => a -> [a] -> Bool
notMember1 _ [] = True
notMember1 y (x:xs)
```

```
       | y==x = False
       | otherwise = notMember1 y xs

-- ze seznamu prvku vylouci opakujici se prvky
uniq :: (Eq a) => [a] -> [a]
uniq [] = []
uniq (x:xs)
       | x `notMember1` xs = x:(uniq xs)
       | otherwise = uniq xs

-- vrati seznam prvku ze seznamu dvojic
tuplList :: (Eq a) => [(a,a)] -> [a]
tuplList [] = []
tuplList ((one,two):xs) = uniq (one:two:(tuplList xs))

-- vytvori seznam vsech permutaci prvku
perm :: (Eq a) => [a] -> [[a]]
perm [] = [[]]
perm xs = [one:others | one <- xs, others <- (perm (xs `without` one)) ]  -- vrati suffix
seznamu od zadaneho prvku  upto :: (Eq a) => [a] -> a -> [a]
upto [] _ = []
upto (x:xs) a
       | x==a = [x]
       | otherwise = x:(xs `upto` a)

-- overi, zda permutace splnuje podminky usporadani
validone :: (Eq a) => [a] -> (a,a) -> Bool
validone p (a,b) = b `notMember1` (p `upto` a)

valid :: (Eq a) => [a] -> [(a,a)] -> Bool
valid p [] = True
valid p (cond:conds) = validone p cond && valid p conds

-- vyresi ulohu tak, ze ze seznamu dvojic udela seznam prvku, z nej vytvori
-- vsechny permutace, ktere pak prefiltruje pres podminky usporadani
permus :: (Eq a) => [(a,a)] -> [[a]]
permus a = [p | p <- perm (tuplList a), valid p a]  -- testy perm_test1 = permus [(1,2),
(17,18),(16,17)]

-- 7. PREVOD N-ARNI -> BINARNI ------------------------------------------------

{-
Sestavte funkci realizujici kanonickou reprezentaci obecneho stromu pomoci
binarniho ("levy syn" = prvorozeny syn, "pravy syn" = mladsi bratr).
-}

-- datova struktura

data TreeN a = NodeN a [TreeN a] deriving (Eq, Show)
data TreeB a = NilB | NodeB a (TreeB a) (TreeB a) deriving (Eq, Show)


--
convNBs :: (Eq a) => [TreeN a] -> (TreeB a)
convNBs [] = NilB
convNBs [NodeN a []] = NodeB a NilB NilB
convNBs [NodeN a [x]] = NodeB a NilB (convNB x)
convNBs [NodeN a xs] = NodeB a (convNBs xs) NilB
convNBs ((NodeN a xs):ts) = NodeB a (convNBs xs) (convNBs ts)

--
convNB :: (Eq a) => (TreeN a) -> (TreeB a)
convNB tree = convNBs [tree]

-- testovaci data
```

```
treeN1 :: TreeN Int
treeN1 = NodeN 10
        [
          NodeN 5
          [
            NodeN 2 [],
            NodeN 7 []
          ],
          NodeN 7
          [
            NodeN 10 []
          ],
          NodeN 12 [],
          NodeN 3
          [
            NodeN 18 [],
            NodeN 1 [],
            NodeN 40 []
          ]
        ]

{-
take se da zapsat jako

NodeN 10 [NodeN 5 [NodeN 2 [],NodeN 7 []],NodeN 7 [NodeN 10 []],NodeN 12 [],
NodeN 3 [NodeN 18 [],NodeN 1 [],NodeN 40 []]]
-}

treeN2 :: TreeN Int
treeN2 = NodeN 10
        [
          NodeN 5
          [
            NodeN 4 [],
            NodeN 1 []
          ],
          NodeN 7 [],
          NodeN 2 []
        ]


-- testy

treeNB_t1 = convNB treeN1
treeNB_t2 = convNB treeN2

-- 8. PRUCHOD KANONICKOU REPREZENTACI ----------------------------------------

{-
Naprogramujte funkci, ktera na zaklade kanonicke reprezentace obecneho stromu
pomoci binarniho stromu vyda seznam vznikly pruchodem puvodniho obecneho
stromu do sirky.
-}

-- datova struktura - shodna s predchozi ulohou
{-
data TreeN a = NodeN a [TreeN a] deriving (Eq, Show)
data TreeB a = NilB | NodeB a (TreeB a) (TreeB a) deriving (Eq, Show)
-}

-- vraci value X (hodnotu v uzlu)
getVal :: (TreeB a) -> a
getVal (NodeB a _ _) = a

-- vraci leveho syna
getLeft :: (TreeB a) -> (TreeB a)
```

```
getLeft (NodeB _ left _) = left

-- vraci praveho syna
getRight :: (TreeB a) -> (TreeB a)
getRight (NodeB _ _ right) = right

-- reseni ulohy podle schematu:
-- 1) je-li fronta prazda -> konec, jinak odeber X prvek z fronty
-- 2) jestlize X==NilB jdi na 1) jinak na 3)
-- 3) dej value X na vystup (hodnota v uzlu)
-- 4) vloz left X do fronty a pro X:=right X proved 2)
--    rekurze pro right X je trikova pomoci zarazeni right X na zacatek fronty
--    a zavolani 1) tudiz se vlastne provede 2) pro right X

tKBFSq :: (Eq a) => [TreeB a] -> [a]
tKBFSq [] = []
tKBFSq (x:xs)
       | x==NilB = tKBFSq xs
       | otherwise = [getVal x]++(tKBFSq ([getRight x] ++ xs ++ [getLeft x]))

-- zavola funkci s frontou o jednom prvku
tKBFS :: (Eq a) => (TreeB a) -> [a]
tKBFS tree = tKBFSq [tree]

-- testovaci data
treeBK1 :: TreeB Int
treeBK1 = NodeB 10
         (
           NodeB 5
           (
             NodeB 2
             NilB
             (
               NodeB 7 NilB NilB
             )
           )
           (
             NodeB 7
             (
               NodeB 10 NilB NilB
             )
             (
               NodeB 12
               NilB
               (
                 NodeB 3
                 (
                   NodeB 18
                   NilB
                   (
                     NodeB 1
                     NilB
                     (
                       NodeB 40 NilB NilB
                     )
                   )
                 )
                 NilB
               )
             )
           )
         ) NilB

{-
take se da napsat jako
```

```
NodeB 10 (NodeB 5 (NodeB 2 NilB (NodeB 7 NilB NilB)) (NodeB 7 (NodeB 10 NilB
NilB) (NodeB 12 NilB (NodeB 3 (NodeB 18 NilB (NodeB 1 NilB (NodeB 40 NilB
NilB))) NilB)))) NilB

nebo jako vysledek predchozi ulohy

treeNB_t1
-}

-- testy

tKBFS_t1 = tKBFS treeBK1
tKBFS_t2 = tKBFS treeNB_t1
tKBFS_t3 = tKBFS treeNB_t2

-- 9. OPERACE NAD FUNKCI APL ------------------------------------------------

{-
Definujte funkci apl se ctyrmi parametry:
  S ... seznam prvku nejakeho typu
  f ... unarni funkce aplikovatelna na prvky tohoto typu
  g ... binarni (vlevo asociativni) funkce aplikovatelna na prvky tohoto typu
  p ... pocatecni hodnota

Funkce apl "provede funkci f na vsechny prvky seznamu S, za takto vznikly
seznam pripoji prvek p a spocita vysledek, ktery vznikne tim, ze do vsech
mezer noveho seznamu vlozime funkci g". (To neni navod k programovani,
ale popis funkce.)

Na zaklade funkce apl vytvorte nasledujici funkce:
  a) minimum prvku z neprazdneho seznamu
  b) aritmeticky prumer z prvku neprazdneho seznamu
  c) geometricky prumer z prvku neprazdneho seznamu (n-ta odmocnina ze soucinu
     jeho prvku - n je delka seznamu.)
  d) harmonicky prumer z prvku neprazdneho seznamu (druha odmocnina ze souctu
     druhych mocnin jeho prvku)

Navod: Maji funkce b) az d) neco spolecneho?
-}

{-
zrejme tedy

apl [1,2,3] (*3) (+) 100

udela seznam

[3,6,9]

potom nasklada funkce g:

(((3 + 6) + 9) + 100)

a vysledek tedy bude

118
-}

-- funkce apl je slozeni foldl a map
apl :: [a] -> (a -> a) -> (a -> a -> a) -> a -> a
apl s f g p = foldl g p (map f s)

-- minimum prvku z neprazdneho seznamu
aplMin :: (Ord a) => [a] -> a
aplMin (x:xs) = apl xs (\ x -> x) (min) x
```

```
-- aritmeticky prumer z prvku neprazdneho seznamu
aplAvA :: (Fractional a) => [a] -> a
aplAvA (x:xs) = apl xs (\ y -> y/lxs) (+) (x/lxs)
       where lxs = fromInt (length (x:xs))

-- geometricky prumer z prvku neprazdneho seznamu
aplAvG :: [Double] -> Double
aplAvG (x:xs) = apl xs (\ y -> y**(1/lxs)) (*) (x**(1/lxs))
       where lxs = fromInt (length (x:xs))

-- harmonicky prumer z prvku neprazdneho seznamu
aplAvH :: [Double] -> Double
aplAvH (x:xs) = sqrt (apl xs (\ y -> y**2) (+) (x**2))
       where lxs = fromInt (length (x:xs))

-- testy

apl_t1 = apl [1,2,3] (*3) (+) 100
apl_t2 = aplMin [4,1,3,6,5,-3,2,6,0]
apl_t3 = aplAvA [5,9,1,0,4,5]
apl_t4 = aplAvG [16,9,12]
apl_t5 = aplAvH [3,5,4,10]              -- = 12.24

-- 10. CETNOST SLOV ------------------------------------------------------------

{-
Typ string je definovan takto: type String = [Char]. Krome zapisu
['a','n','n','a'] muzeme ekvivalentne psat i "anna". Naprogramujte funkci,
ktera dostane jako vstup string Doc (ktery pro jednoduchost muze obsahovat
jen mala pismena anglicke abecedy, znak \n a znak mezera) a cislo N a vyrobi
z nej "abecedni index vyskytu slov delky alespon N na radcich" dokumentu Doc,
tj. provede s nim nasledujici operace (budeme je demonstrovat na priklade):

Doc=="jak kul husar\nluk\nstal jak\n\nkul v plote\nuz jsem zase v tom"

N==3

 a) Rozdeli vstupni string doc na posloupnost radek (stringu) Lines
    (radky jsou oddeleny znakem \n)

    ["jak kul husar", "luk", "stal jak", [], "kul v plote",
     "uz jsem zase v tom"]


 b) Radky v seznamu Lines ocisluje - vystupem bude tedy seznam dvojic
    (cisloradky, radka)

    [(1, "jak kul husar"), (2, "luk"), (3, "stal jak"), (4, ""),
     (5, "kul v plote"), (6, "uz jsem zase v tom")]


 c) Rozdeli kazdou radku na slova - vyda seznam dvojic (cisloradky, slovo)

    [(1, "jak"), (1, "kul"), (1, "husar"), (2, "luk"), (3, "stal"),
     (3, "jak"), (5, "kul"), (5, "v"), (5, "plote"), (6, "uz"),
     (6, "jsem"), (6, "zase"), (6, "v"), (6, "tom")]


 d) Usprada tento seznam podle druhe slozky - slova

    [(1, "husar"), (1, "jak"), (3, "jak"), (6, "jsem"), (1, "kul"),
     (5, "kul"), (2, "luk"), (5, "plote"), (3, "stal"), (6, "tom")
     (6, "uz"), (5, "v"), (6, "v"), (6, "zase")]


 e) Prepracuje vstupni seznam na seznam dvojic
```

```
    (Slovo, SeznamCiselRadkuNaKterychSeTotoSlovoVyskytuje)

    [("husar", [1]), ("jak", [1,3]), ("jsem", [6]), ("kul", [1,5]),
     ("luk", [2]), ("plote", [5]), ("stal", [3]), ("tom", [6]),
     ("uz", [6]), ("v", [5,6]), ("zase", [6])]


 f) Vypusti slova kratsi nez vstupni parametr N (v priklade == 3)

    [("husar", [1]), ("jak", [1,3]), ("jsem", [6]), ("kul", [1,5]),
     ("luk", [2]), ("plote", [5]), ("stal", [3]), ("tom", [6]),
     ("zase", [6])]

-}

-- porovnani zacatku retezce s patternem
match :: String -> String -> Bool
match str pat = (take (length pat) str) == pat

-- ze stringu vybere string az do prvniho vyskytu patternu
upToStr :: String -> String -> String
upToStr "" _ = []
upToStr str@(x:xs) pat
       | str `match` pat = []
       | otherwise = x:(xs `upToStr` pat)

-- ze stringu vybere string po prvnim vyskytu patternu
fromStr :: String -> String -> String
fromStr str "" = str
fromStr "" _ = []
fromStr str@(x:xs) pat@(y:ys)
       | str `match` pat = xs `fromStr` ys
       | otherwise = (xs `fromStr` pat)


-- cast a)
makeLines :: String -> String -> [String]
makeLines [] _ = []
makeLines doc delim = [(doc `upToStr` delim)] ++ (makeLines (doc `fromStr` delim) delim)

-- cast b)
numberLines :: [String] -> [(Int,String)]
numberLines lines = numLines 1 lines

-- reseni b) s akumulatorem cisel
numLines :: Int -> [String] -> [(Int,String)]
numLines _ [] = []
numLines n (x:xs) = [(n,x)] ++ numLines (n+1) xs

-- cast c)
splitLines :: [(Int,String)] -> [(Int,String)]
splitLines [] = []
splitLines ((n,str):xs) = map ((,) n) (makeLines str " ") ++ splitLines xs

-- cast d)
sortPairs :: [(Int,String)] -> [(Int,String)]
sortPairs [] = []
sortPairs ((j,s):xs) = (sortPairs [(i,r) | (i,r) <- xs, r<- xs, s<=t])  -- cast e)
joinPairLists :: [(String,[Int])] -> [(String,[Int])]
joinPairLists [] = []
joinPairLists ((s,i):(t,j):xs)
       | s==t = joinPairLists ([(s,i ++ j)] ++ xs)
       | otherwise = [(s,i)] ++ joinPairLists ((t,j):xs)
joinPairLists xs = xs

joinPairs :: [(Int,String)] -> [(String,[Int])]
```

```
joinPairs xs = joinPairLists (map (\ (i,s) -> (s,[i])) xs)

-- cast f)
cutShorts :: [(String,[Int])] -> Int -> [(String,[Int])]
cutShorts [] _ = []
cutShorts ((s,i):xs) n
       | length(s) < n =" cutShorts" otherwise =" [(s,i)]"> Int -> [(String,[Int])]
wordFreq doc n = cutShorts (joinPairs (sortPairs (splitLines (numberLines (makeLines doc
"\n"))))) n

-- testovaci data
doc :: String
doc = "jak kul husar\nluk\nstal jak\n\nkul v plote\nuz jsem zase v tom"

-- testy
wF_t1 = makeLines doc "\n"
wF_t2 = numberLines wF_t1
wF_t3 = splitLines wF_t2
wF_t4 = sortPairs wF_t3
wF_t5 = joinPairs wF_t4
wF_t6 = cutShorts wF_t5 3
wF_t7 = wordFreq doc 3


-- 11. VYPOUSTENI BVS UZLU ------------------------------------------------------

{-
Definujeme prirozenou reprezentaci binarniho stromu, v jehoz uzlech je
ulozena informace nejakeho typu (podtridy Ord).
Naprogramujte funkci, ktera z BVS vypusti uzel se zadanou hodnotou.
-}

-- jedna se o lehci variantu ulohy 2.

-- THE END ---------------------------------------------------------------------
```

Haskell
=========

* Definujte si vhodným způsobem datovou strukturu pro reprezentaci orientovaného grafu. Vytvořte funkci
   (definovanou na všech grafech), která vrátí topologické uspořádání grafu nebo sdělí, že topologicky
   uspořádat nejde.

Řešení by Martin Všetička:

```
    {-
    Popis algoritmu: Topologické uspořádání grafu: Máme orientovaný graf G s N vrcholy a chceme očíslovat vrcholy
    čísly 1 až N tak, aby všechny hrany vedly z vrcholu s větším číslem do vrcholu s menším číslem, tedy aby pro každou
    hranu e = (vi, vj) bylo i > j. Představme si to jako srovnání vrcholů grafu na přímku tak, aby "šipky"
    vedly pouze zprava doleva.

    Cyklus je to jediné, co muže existenci topologického uspořádání zabránit. Libovolný acyklický graf
    lze uspořádat následujícím algoritmem:

    1. Na začátku máme orientovaný graf G a proměnnou p = 1.
    2. Najdeme takový vrchol v, ze kterého nevede žádná hrana (budeme mu říkat stok).
       Pokud v grafu žádný stok není, výpočet končí, protože jsme našli cyklus.
    3. Odebereme z grafu vrchol v a všechny hrany, které do něj vedou.
    4. Přiřadíme vrcholu v číslo p.
```

```
    5. Proměnnou p zvýšíme o 1.
    6. Opakujeme kroky 2 až 5, dokud graf obsahuje alespoÅˆ jeden vrchol.

    zdroj: http://ksp.mff.cuni.cz/tasks/18/cook2.html
    -}


    data Vertex a   = Vertex a [a] deriving Show
    data  (Eq a) => Graph a    = Vertices [Vertex a]

    topSrch :: (Eq a) => (Graph a) -> [a]

    topSrch g = reverse $ topSrch1 g

    topSrch1 :: (Eq a) => (Graph a) -> [a]
    topSrch1 (Vertices [])    = []
    topSrch1 g@(Vertices lst) = (lastOne):(topSrch1 (Vertices lst3))
       where lastOne = fndLast lst
             lst2 = rmVertex lst
             lst3 = remEdges lst2 lastOne

    fndLast :: [Vertex a] -> a
    fndLast []                 = error "fndLast: No last vertex. It's not possible to
topologically sort the graph"
    fndLast ((Vertex a []):xs) = a
    fndLast (x:xs)             = fndLast xs

    rmVertex :: [Vertex a] -> [Vertex a]
    rmVertex []                  = []
    rmVertex ((Vertex a []):xs) = xs
    rmVertex (x:xs)              = x:(rmVertex xs)


    remEdges :: (Eq a) => [Vertex a] -> a -> [Vertex a]
    remEdges [] _ = []
    remEdges ((Vertex nm1 edg1):xs) v = ((Vertex nm1 (filter (/= v) edg1)):(remEdges xs v))


    {-
    Testy:

    topSrch (Vertices [(Vertex 'a' ['b']),(Vertex 'b' ['c']),(Vertex 'c' ['d']), (Vertex 'd'
['a']),])  -- obsahuje cyklus
    topSrch (Vertices [(Vertex 'a' ['b']),(Vertex 'b' ['c']),(Vertex 'c' ['d']), (Vertex 'd'
[]),])  -- neobsahuje cyklus
    -}
Řešení z http://prgs.xf.cz/pis080107/topolog_tried.hs:

    type Graf a = [(a,Int,[a])]
    --(vrchol,pocet predchodcov, zoznam nasledovnikov)

    sort::Graf a->Graf a
    --zotriedenie grafu podla poctu predchodcov
    sort []        = []
    sort ((v,pp,zn):xs)   = (sort [s|s@(a,b,c)<-xs, b<pp])
                     ++ [(v,pp,zn)]
                     ++ (sort [s|s@(a,b,c)<-xs, b>=pp])

    usp::(Eq a)=>Graf a->(Bool, [a])
    --usp graf z => graf topologicky usporiada do zoznamu a tento pripoji k zoznamu z
    --predpokladame, ze g je zotriedeny funkciou sort
    usp []                = (True, [])
    usp (h@(v,pp,zn):xs)  = if pp>0 then (False, [])
                     else let (res,t) = usp (sort (zniz zn xs))
                          in (res,v:t)
```

```
    zniz::(Eq a)=>[a]->Graf a->Graf a
    --zniz nasledovnikom v grafe pocet predchodcov o jedna
    zniz znasl g          = [(v,pp-1,zn)|(v,pp,zn)<-g, znasl `obsahuje` v]
                     ++[s|s@(v,pp,zn)<-g, not (znasl `obsahuje` v)]
                     where
                          obsahuje::(Eq a)=>[a]->a->Bool
                          (x:xs) `obsahuje` v = if x==v then True
                                                else xs `obsahuje` v
                          [] `obsahuje` v           = False

    topol::(Eq a)=>Graf a->[a]
    --vrati topologicke usporiadanie grafu, ak graf obsahuje kruzniu, vrati prazdny zoznam
    topol g               = let (res,t) = usp (sort g)
                     in if res==True then t else []   sort []  = []
       sort ((v,h):xs) = sort [(a,b)|(a,b)<-xs, b<h]
                         ++ [(v,h)]
                         ++ sort [(a,b)|(a,b)<-xs, b>=h]

       nodes::Tree a->[a]
       --vrati seznam vrcholu stromu setridenych podle vzdalenosti od nejblisiho listu,
ktery je pod nim
       nodes t = [v|(v,h)<-sort (ohodnot t)]
       --nodes strom vrati "aeghbdfc"
```

* Definujte typ vhodný k reprezentaci multimnožiny desítkových cifer (každá z cifer se v ní může vyskytovat vícekrát).
   Uvažme čísla, která lze sestavit z cifer takové multimnožiny (nemusíme použít všechny). Naprogramujte dvě funkce:

   a) první, která nalezne k číslu N a multimnožině M N-té takové číslo
   b) druhou "inverzní", která k takovému číslu X spočítá jeho pořadové číslo

Řešení:

   a)

      TODO

   b)

```
      -- TEST:

      {-
        multPor [1,1,2,5,0] 5
      -}

      -- NENI TO TOTALNE FUNKCI A JE TO DOST NEEFEKTIVNI, JE POTREBA VYMYSLET JAK
ZNOVUPOUZIT VYSLEDKY

      -- multPor (multimnozina + porad)
      --  #1 arg: prvky v multimnozine, libovolne usporadane
      --  #2 arg: cislo slozene z prvku z multimnoziny
      -- vrati: poradi cisla vzhledem k moznym cislum vytvorenym z multimnoziny
      multPor :: [Int] -> Int -> Int

      multPor [] x = 0
      multPor m x  =   sum [pomocnyVypocet m2 l | l <- [1 .. (len-1)]]
                     - sum [pomocnyVypocet m3 l | l <- [1 .. (len-1)], (num,_) <- m2, num ==
0]
                     + (spocitej m2 iList len 1)
                     + 1
        where iList          = reverse (int2list x)
              len            = length iList
              m2             = nasobnosti (quicksort m) 0 (-1)
```

```
            m3            = snizCetnost m2 0
                              -- (-1) je specialni hodnota, potrebuji jen nejak fci
zavolat, aby to neovlivnilo vysledek

      -- spocitej
      --  #1 arg: pocet jednotlivych cifer
      --  #2 arg: zadane cislo (list)
      --  #3 arg: delka cisla, aby se nemusela porad pocitat znovu
      --  #4 arg: na kolikate cifre odpredu jsem
      -- popis: ozn. prvni cifru `c'. Uvazujeme, ze na prvni pozici mohou byt cisla <= c a
na ostatnich muze byt cokoli
      --   co zbyva v multimnozine
      -- vraci: pocet poradi zadaneho cisla v ramci dane multimnoziny (razeno od nejmensiho
po nejvetsi)
      spocitej :: [(Int,Int)]  -> [Int] -> Int -> Int -> Int

      spocitej [] _ _ _      =  0
      spocitej _ [] _ _      =  0
      spocitej m (x:xs) l c  =  sum [ pomocnyVypocet (snizCetnost m y) (l-1) | y <-
moznostiPrvniCifra ]
                             + (spocitej m2 xs (l-1) (c+1))
          where
                m2                  = snizCetnost m x
                moznostiPrvniCifra  = filter (pred1 c x) [frst | (frst,sec) <- m, sec > 0]



      -- pomocnyVypocet
      --  #1 arg: pocet jednotlivych cifer; pr. (1,2)  - jednicku mam 2x
      --  #2 arg: odendana cifra
      --  #3 arg: na kolika prvcich
      -- vraci: pocet cisel delky `l' z dane multimnoziny
      -- pozn: pocitaji se i pripady, kdy na zacatku je nula, jelikoz fce se pouziva jen
funkci `spocitej' v pripade,
      --  ze mame pripad cisla [1-9]xxxx, kde na xxxx se pouziva tato funkce
      pomocnyVypocet :: [(Int,Int)] -> Int -> Int
      pomocnyVypocet m 0  = 1
      pomocnyVypocet m l  = (countDiff m 0) * (pomocnyVypocet ([(prvek-1,cet) | (prvek,cet)
<-m ]) (l-1))

      -- countDiff
      --  #1 arg: seznam dvojic prvek cetnost
      --  #2 arg: inkrementovana promenna, vstupni hodnota je nula
      -- vraci: pocet ruznych prvku s cetnosti vetsi nez nula
      countDiff :: [(Int,Int)] -> Int -> Int
      countDiff [] n   = n
      countDiff ((prvek,cet):xs) n
            | cet > 0   =  countDiff xs (n+1)
            | otherwise =  countDiff xs n


      -- snizi cetnost v seznamu u daneho prvku
      snizCetnost :: [(Int,Int)] -> Int -> [(Int,Int)]

      snizCetnost [] _             =  []
      snizCetnost ((prvek',cet):xs) prvek
               | prvek == prvek'   =  if cet - 1 == 0 then
                                                      xs
                                                 else
                                                      (prvek',cet-1):xs
               | otherwise         =  (prvek',cet):(snizCetnost xs prvek)


      -- U prvni cifry nepovolujeme nulu
      pred1 :: Int -> Int -> Int -> Bool
      pred1 c m n
```

```
      | n > 0 && n < m && c == 1  = True
      | n < m && c > 1           = True
      | otherwise                = False


   -- KATEGORIE PREDIKATU, KTERE UZ BUDOU NEKDE PRAVDEPODOBNE NAPROGRAMOVANE

   factorial :: Int -> Int
   factorial x
         | x == 1  = 1
         | x == 0  = 1

   factorial n = n*(factorial (n-1))


   int2list :: Int -> [Int]

   int2list n
      | ndiv10 > 0  = nmod10:(int2list ndiv10)
      | otherwise   = [nmod10]
      where nmod10 = n `mod` 10
            ndiv10 = n `div` 10


   nasobnosti :: [Int] -> Int -> Int -> [(Int, Int)]

   nasobnosti []     n  lastVal                    = [(lastVal,n)]
   nasobnosti (x:xs) n  lastVal
                | lastVal == (-1)   = nasobnosti xs 1 x
                | lastVal /= x      = (lastVal,n):(nasobnosti xs 1 x)
                | lastVal == x      = nasobnosti xs (n+1) x



   quicksort   :: (Ord a)=> [a] -> [a]
   quicksort []      = []
   quicksort []      = []
   quicksort (x:xs)  =  quicksort [k | k <- xs, k <= x]
                        ++ [x] ++
                        quicksort [k | k <- xs, k > x]
```

* Všechna k-ciferná čísla, v jejichž dekadickém zápisu jsou všechny cifry různé, jsme "myšlenkově" seřadili podle velikosti. Napište procedury či funkce, které k zadanému číslu přímo spočtou jeho pořadí a naopak na základě pořadí naleznou příslušné číslo.

Řešení: [zatím pouze další permutace]

```
quicksort :: (Ord a) => [a] -> [a]
quicksort [] = []
quicksort (x:xs) =
   let smallerSorted = quicksort [a | a <- xs, a <= x]
       biggerSorted = quicksort [a | a <- xs, a > x]
   in  smallerSorted ++ [x] ++ biggerSorted


-- 1. param je posledni hodnota, 2. je soucasne minimum z prochazenych prvku
fnd        :: (Ord a) => Int -> a -> [a] -> Int
fnd n z (x:xs)
     |  z < x && xs /= []  =  fnd (n+1) x xs
     |  z > x             =  n
     |  otherwise         =  error "zadna dalsi permutace"        -- error state
```

-- Myslenka je takova, ze otocime seznam s permutaci, hledame prvni cislo X takove, ze je vetsi nez predchozi cislo,

```
    -- najdeme minimum v casti pred X a prohodime toto cislo s X v seznamu a setridim cast
pred X.
    nepe         :: (Ord a,Show a) => Int -> [a] -> [a]
    nepe k lst    = (take (n-2) lst)  ++ [min'] ++ (quicksort (filter (/= min') (take (j+1)
revLst)))
                    where (y:ys) = reverse lst
                          revLst = (y:ys)
                          j      = fnd 1 y ys
                          n      = k - j + 1
                          min'   = (minimum  (filter (> (revLst !! j)) (take j revLst)))
```

Řešení #2 pomocí algoritmu z přednášky:

```
    naslPerm  :: (Ord a) => [a] -> [a]
    naslPerm lst  =  reverse zbytek ++ (z:lst3)

          where lst2              = reverse lst
                (rZac, y, zbytek) = rostZac lst2
                (lst3, z)         = zarad rZac y


    rostZac  :: (Ord a) => [a] -> ([a],a,[a])
    rostZac []       = error "Prazdny seznam neni povolen"
    rostZac (x:y:xs)
          | x > y    = ([x],y,xs)
          | x < y    = ((x:sezn),z,zbytek)
      where (sezn,z,zbytek) = rostZac (y:xs)

    zarad  :: (Ord a) => [a] -> a -> ([a],a)
    zarad (x:xs) y
        | y < x     = (y:xs, x)   -- x je nejmensi vetsi nez y
        | y > x     = (x:sezn, z)
      where (sezn, z)  = zarad xs y
```

* Řídká matice je reprezentována jako trojice (m,n,s), kde m a n jsou rozměry matice a s je seznam
   trojic (i,j,aij) - i,j souřadnice, a_{ij} nenulové číslo na těch souřadnicích - uspořádany vzestupně
   podle i a uvnitř řádek podle j.

   Naprogramujte:
     a) transpozici
     b) násobení 2 matic

Řešení by Mus:

```
   type Matice = (Int, Int, [Souradnice])    -- synonymum
   type Souradnice = (Int, Int, Int)

   {- a) transpozici      -}
   {- ================== -}

   transpozice :: Matice -> Matice
   transpozice (m, n, []) = (n, m, [])
   transpozice (m, n, s) = (n, m, quicksort [(j, i, aij) | (i, j, aij) <- s])
   -- transpozice (m, n, s) = (n, m, sort [(j, i, aij) | (i, j, aij) <- s])

   sort [] = []
   sort (x:xs) = sort [a | a <- xs, a <= x] ++ [x] ++ sort [a | a <- xs, a > x]

   quicksort :: (Ord a) => [a] -> [a]
   quicksort [] = []
   quicksort (x:xs) =
       let smallerSorted = quicksort [a | a <- xs, a <= x]
           biggerSorted = quicksort [a | a <- xs, a > x]
```

```
       in  smallerSorted ++ [x] ++ biggerSorted

   -- Test: transpozice (4,4,[(1,3,3),(2,2,4),(3,4,5)])
   -- Vysledek: (4,4,[(2,2,4),(3,1,3),(4,3,5)])

   {- b) násobení 2 matic -}
   {- ================== -}

   -- Cij = Suma k=1..n : Aik * Bkj

   vynasob :: Matice -> Matice -> Matice
   vynasob (m, n, a) (o, p, b) | n /= o    = error "matice nelze nasobit"
                               | otherwise = (m, p, secti (srovnej [(i, j, aik*bkj) | (i,
k1, aik) <- a, (k2, j, bkj) <- b, k1 == k2 ]))

   -- seradi seznam trojic (a,b,c) tak, ze za sebou jsou trojice se stejnymi a a b
   srovnej :: [Souradnice] -> [Souradnice]
   srovnej [] = []
   srovnej [a] = [a]
   srovnej ((i1, j1, aij1):s) = [(i1, j1, aij1)] ++
           [ (i, j, aij) | (i, j, aij) <- s, i == i1, j == j1 ] ++
           srovnej [ (i, j, aij) | (i, j, aij) <- s, (i /= i1 || j /= j1) ]

   secti :: [Souradnice] -> [Souradnice]
   secti [] = []
   secti [a] = [a]
   secti ((i1, j1, aij1):(i2, j2, aij2):s)
    | i1 == i2 && j1 == j2       =    if aij1 + aij2 /= 0 then secti ((i1, j1, aij1 +
aij2):s)
                                                              else secti s
    | otherwise = [(i1, j1, aij1)] ++ secti ((i2, j2, aij2):s)
```

* Vytvorte funkci, která pro (nekonecný) seznam reálných císel [xi] vstup a (krátký) konecný rostoucí
   seznam intervaly malých integeru [a1,a2,...,an] - predstavte si treba [2,5,10,25] vytvorí nekonecný seznam,
   jehož k-tý prvek je "n-tice" klouzavých prumeru s intervaly [a1,a2,...,an] konce k-tým.

* Jina formulace zadani: Sestavte funkci, ktera ke vstupujici (nekonecne) posloupnosti realnych cisel a   prirozenemu cislu N vyda posloupnosti "klouzavych prumeru s intevalem N". Klouzavy prumer s   intervalem N je prumer poslednich prvku.

Řešení:
   Klouzave prumery - urcite pomocnou fci (nebo lambda), ktera dostane dvakrat seznam (puvodni a puvodni bez prvnich n clenu), soucasnou sumu a dane N a bude nove soucty pocitat pomoci tech predchozich. Co jsem slysel, nelibi se mu, kdyz to clovek pokazde pocita znovu.

* Mocninna rada je reprezentovana (nekonecnou posloupnosti) posloupnosti jejich koeficientu. Vytvorte funkce, ktere spocitaji
   a) Soucet dvou mocninnych rad
   b) Soucin dvou mocninnych rad
   c) K-tou derivaci mocninne rady

Řešení by Martin Všetička:

   a)

```
      soucet  s t         = [a+b | (a,b) <- zip s t]   -- verze 1
      soucet2 (a:s) (b:t) = ((a+b):soucet2 s t)        -- verze 2

      -- pokud nescitame radu, ale polynomy:
      add                 :: (Num a)=>[a]->[a]->[a]
      add [] []           =  []                        -- neni potreba pokud jde o mocninou
radu
      add (x:xs) []       =  x:xs                      -- dtto
```

```
      add [] (y:ys)       =  y:ys                      -- dtto
      add (x:xs) (y:ys)   =  (x+y):(add xs ys)
   b)

      zipWith'            :: (a -> a -> a) -> [a] -> [a] -> [a]
      zipWith' f [] _     = []
      zipWith' f _ []     = []
      zipWith' f (x:xs) (y:ys) = f x y : zipWith' f xs ys

      product                   :: (Num a)=>[a]->[a]->[a]
      product series1 series2  = [sum (zipWith' (*) xs ys ) | n <- [1,2..],
                                  let xs = take n series1, let ys = reverse (take n
series2)]

      Jiny zpusob:

      let (+++) = zipWith (+);
      (f:fs) *** (g:gs) = f*g : (map (*f) gs +++ map (*g) fs +++ (0: fs***gs))  in  [1..]
*** [1..]

            Jiny zpusob:

      nasob a b = nas a b []
      nas (an:as) b rev = [ sum( [ x*y | (x,y) <- zip b (an:rev)] ) ] ++ (nas as b
(an:rev) )

   c)
      -- 1. zpusob
      derivative              :: (Num a,Enum a)=>Integer->[a]->[a]
      derivative  n lst = if n > 0 then
                                derivative (n-1) (zipWith (*) [1,2..] (drop 1 lst))
                             else
                                lst

      -- 2. zpusob
      der (a:s) k = (k*a):der s (k+1)

      derivace   (a:s) =  der s 1                        -- jednoducha derivace (odeberu
konstantu ze zacatku a nasobim v der)
      derivace_k s 0   =  s                              -- 0-ta derivace
      derivace_k s k   =  derivace_k (derivace s) (k-1)  -- k-ta derivace
```

* Reprezentovat BVS a pak funkci na vypousteni vsech vrcholu s hodnotou mezi Min a Max:

Řešení:

```
   data (Ord a, Eq a, Num a) => BVS a   =  Null | Node (BVS a) a (BVS a) deriving Show

   inorder :: (Ord a, Eq a, Num a) => BVS a -> [a]
   inorder Null = []
   inorder (Node left a right) = inorder left ++ [a] ++ inorder right

   min'        :: (Ord a, Eq a, Num a) => BVS a -> a
   min'        =  head . inorder

   max'        :: (Ord a, Eq a, Num a) => BVS a -> a
   max'        =  head . inorder


   {- Pro jeden prvek -}
   deleteX       ::  (Ord a, Eq a, Num a) => a -> (BVS a) -> (BVS a)

   deleteX e Null = Null
   deleteX e (Node Null a Null)
         | e == a     =   Null
         | otherwise  =   Node Null a Null
```

```
   deleteX e (Node left a Null)
       | a == e    = left
       | otherwise = (Node (deleteX e left) a Null)

   deleteX e (Node Null a right)
       | a == e    = right
       | otherwise = (Node Null a (deleteX e right))

   deleteX e (Node left a right)
       | a == e    =  Node  (deleteX (max' left) left) (max' left) right
       | a <  e    =  Node left a (deleteX e right)
       | a >  e    =  Node (deleteX e left) a right

   {- Test: deleteX 3 (Node (Node Null 1 Null) 2 (Node (Node Null 3 Null) 3 Null))  -}

   {- viceprvkova verze: -}

   -- vypusti ze stromu vsechny uzly od m do n
   remBVS :: (Ord a) => BVS a -> a -> a -> BVS a
   remBVS Null _ _ = Null

   remBVS (BVS (Null,val,Null)) m n
           | m <= val && val <= n        =   Null
           | otherwise                   =   BVS (Null,val,Null)

   remBVS (BVS (left,val,right)) m n =
           if m>n then Null
           else if n < val then BVS (remLeft,val,right)
           else if val < m then BVS (left,val,remRight)
           else if not (remLeft==Null)  then BVS (remLeftMax, maxremLeft, remRight)
           else if not (remRight==Null) then BVS (remLeft, minremRight, remRightMin)
           else Null
                   where remLeft     = remBVS left m n
                         remRight    = remBVS right m n
                         maxremLeft  = findMax remLeft
                         minremRight = findMin remRight
                         remLeftMax  = remBVS remLeft maxremLeft maxremLeft
                         remRightMin = remBVS remRight minremRight minremRight
```

* Soucin a podil dlouhych cisel - dlouhe cislo je trojice Znamenko, [Int], Pozice desetinne carky od prvni cifry

* úloha BATOH: Je dán seznam A číslo N. Napište funkci, která zjistí, zda je možné posčítat (některé) prvky seznamu,
   aby součet vyšel N.

Řešení:

```
   -- implementujeme funkci, ktera tento seznam vrati, dana uloha pak by byla
   -- pokud seznam neexistuje, vraci se []
   batoh      :: [Int] -> Int -> [Int]
   batoh _ 0  = []
   batoh [] _ = []
   batoh (x:xs) n
    | sum newBatoh1 == n = newBatoh1
    | sum newBatoh2 == n = newBatoh2
    | otherwise = []
          where newBatoh1 = x:(batoh xs (n-x))
                newBatoh2 = batoh xs n
```

* Definujte prirozenou reprezentaci binárních stromu, v jehož uzlech je uložena informace

nejakého typu (podtrídy Ord).
   Sestavte funkci, která na základe rostoucího (!) seznamu S a císla N vytvorí z prvních N prvku tohoto seznamu dokonale vyvážený binární vyhledávací strom T (pro každý uzel platí, že velikost L a P podstromu se liší nejvíc o 1) a spolu s tímto stromem vrátí i seznam, který zbyl ze seznamu S po postavení stromu T, tj. S bez prvních N clenu.

Řešení

```
    data (Ord a) => BST a    =    Null | Node (BST a) a (BST a)

    middle_element :: (Ord a) => [a] -> a      -- returns middle element from the list
    middle_element x = last y
                  where y = take (((length x) `div` 2) + 1) x


    build_tree :: (Ord a) => [a] -> BST a    -- builds perfectly balanced BST
    build_tree [] = Null
    build_tree (h:[]) = Node Null h Null
    build_tree x = Node leftOne h rightOne
                  where h        = middle_element x
                        leftOne  = build_tree [ y | y <- x, y < h]
                        rightOne = build_tree [ y | y <- x, y > h]


    build :: (Ord a) => [a] -> Int -> (BST a, [a])
    build x 0  = (Null, x)
    build [] _ = error "Nothing in list"
    build s n | (length s) < n = error "List S contains less than N elements"
              | otherwise      = ( build_tree (take n s), drop n s )
```

Řešení

```
    -- Binarni Strom
    data (Ord a) => BTree a = Null | Node (BTree a) a (BTree a) deriving Show

    createBTree :: (Ord a) => [a] -> Int -> ((BTree a), [a])
    createBTree [] _ =  (Null, [])
    createBTree s 0 =  (Null, s)
    createBTree [s] n = ((Node Null s Null), [])

    createBTree s n = ((Node l v r), rest)
                      where
                      n1 = n - 1
                      nr = n1 `div` 2
                      nl = n1 - nr
                      (l, (v:rs)) = createBTree s nl
                      (r, rest) = createBTree rs nr


    -- casova slozitost: O(N)
```

* Definujte přirozenou reprezentaci binárního stromu, v jehož uzlech je uložena informace nějakého
   typu (podtřídy Ord). Naprogramujte funkci, která ze zadaného binárního vyhledávacího stromu vypustí
   všechny uzly, které obsahují hodnotu klíče, na kterém zadaná funkce `krit' vrátí hodnotu True.

Řešení

```
   data Tree a    =   Nil | ND (Tree a) a (Tree a) deriving Show

   instance Eq (Tree a) where        -- na porovnavani stromu s Nilem
```

```
      Nil == Nil = True
      _ == _ = False

   vypust::(a->Bool)->Tree a->Tree a

   vypust _ Nil = Nil
   vypust krit (ND l v p)
     | not (krit v) = (ND lt v pt)
     | lt == Nil    = pt
     | pt == Nil    = lt
     | otherwise    = (ND ltBEZmax maxlt pt)
      where
        lt  =  vypust krit l
        pt  =  vypust krit p
        (ltBEZmax,maxlt) =  bezMAX lt


   bezMAX::  Tree a->(Tree a,a)
   bezMAX (ND l v p)
     | p == Nil  = (l,v)        -- pro toto porovnani jsme definovali vyse "instance ..."
     | otherwise = ((ND (l) (v) (pt)), maxpt)
      where
     (pt,maxpt) = bezMAX p
```

Užitečné algoritmy
==========
* následující permutace:

```
  % vyda nasledujici permutaci v lexikografickem poradi
  naslperm(Perm, NPerm) :-
           reverse(Perm, OtPerm),
           rozloz(OtPerm, RostZac, X, Zbytek),
           zarad(RostZac, X, Y, NRostZac),
           concrev(Zbytek,[Y|NRostZac], NPerm).

  % rozloz(+Perm,-RostZac,-KlesPrvek,-Zbytek)
  rozloz([X,Y|T],[X|T1],A,Z):-X<Y,rozloz([Y|T],T1,A,Z).
  rozloz([X,Y|T],[X],Y,T):-X>Y.

  % zarad(+Rost,+Vloz,-NejmenšíVětšíNežX,-RostPoVýměně)
  zarad([A|Rost],X,Y,[A|NRost]):- A < X, zarad(Rost,X,Y,NRost).
  zarad([A|Rost],X,A,[X|Rost]) :- A > X.

  %  ================================================================
  %  concrev(+L,+T,-S)  S je zřetězení obráceného seznamu L
  %                     se seznamem T
  %      tedy predikát ot(+L,-S):- concrev(L,[],S) otáčí seznam
  %      ?-concrev([a,b,c],[1,2],S).    S=[c,b,a,1,2]
  %  ================================================================
  concrev([],L,L).
  concrev([X|T],L,P):- concrev(T,[X|L],P).

       % pozn. algoritmus selze, pokud dalsi permutace v lex. poradi neexistuje
```

* Rozdílový seznam: [1,2,3 | T1]-T1,   [4,5,6|T2]-T2

  Spojení rozdílových seznamů: conc(A-B, B-C, A-C). V konstantnim case.

Užitečné Haskellovské funkce
==========
* zip, forldl, foldr, sum

* http://prgs.xf.cz/ - nějaké řešené íºložky
* http://www.haskell.org/haskellwiki/H-99:_Ninety-Nine_Haskell_Problems - rešené problémy

Užitečné progr. techniky z learnyouhaskell.com
==========

* case lze pouzit prakticky kdekoli

```
describeList :: [a] -> String
describeList xs = "The list is " ++ case xs of [] -> "empty."
                                               [x] -> "a singleton list."
                                               xs -> "a longer list."
```

* If you want to get an element out of a list by index, use !!. The indices start at 0.

```
ghci> "Steve Buscemi" !! 6
'B'
ghci> [9.4,33.2,96.2,11.2,23.25] !! 1
```

* Haskell umí porovnávat n-tice: (1,2,3) <= (3,4,5) vrátí TRUE.

FAQ
===
* Rozdil mezi "where" a "let" konstrukci je v tom, ze "where" je pouze syntactic sugar, nicmene "let" lze pouzit kdekoli.