```lisp
; vim: ts=2 sw=2 et :
;.                                    __
;.                                   /\ \__
;.     ___ ___      ____     ____    \ \ ,_\
;.    /\ \/\ \    /' _ `\   / __`\    \ \ \/
;.    \ \ \_\ \  /\ \/\ \ /\ \L\ \    \ \ \_
;.     \/`____ \ \ \_\ \_\\ \____/     \ \__\
;.      `/___/> \ \/_/\/_/ \/___/       \/__/
;.         /\___/
;.         \/__/

; __preable '(__settings __macros __globals)
;;;; Ynot
(defpackage :ynot (:use :cl))
(in-package :ynot)

(defun help (lst)
  (terpri)
  (format t "ynot (v1.0) : not-so-supervised multi-objective optimization~%")
  (format t "(c) 2022 Tim Menzies, MIT (2 clause) license~%~%")
  (format t "OPTIONS:~%")
  (loop for (x(s y)) on lst by 'cddr do
    (format t " --~(~10a~) ~a = ~a~%" x s y)))

; Define settings.
(defvar *settings*
  '(enough ("how many numbers to keep    "   512)
    far    ("where to search for far items"  .9)
    file   ("load data from file          "  "../data/auto93.csv")
    help   ("show help                    "  nil)
    min    ("min size of rows             "  .5)
    p      ("distance coeffecient         "  2)
    seed   ("random number seed           "  10019)
    some   ("how many items to sample     "  512)
    todo   ("start up action              "  "nothing")))

; Copyright (c) 2021 Tim Menzies

; This is free and unencumbered software released into the public domain.

; Anyone is free to copy, modify, publish, use, compile, sell, or
; distribute this software, either in source code form or as a compiled
; binary, for any purpose, commercial or non-commercial, and by any
; means.

; In jurisdictions that recognize copyright laws, the author or authors
; of this software dedicate any and all copyright interest in the
; software to the public domain. We make this dedication for the benefit
; of the public at large and to the detriment of our heirs and
; successors. We intend this dedication to be an overt act of
; relinquishment in perpetuity of all present and future rights to this
; software under copyright law.

; THE SOFTWARE IS PROVIDED "AS IS", WITHOUT WARRANTY OF ANY KIND,
; EXPRESS OR IMPLIED, INCLUDING BUT NOT LIMITED TO THE WARRANTIES OF
; MERCHANTABILITY, FITNESS FOR A PARTICULAR PURPOSE AND NONINFRINGEMENT.
; IN NO EVENT SHALL THE AUTHORS BE LIABLE FOR ANY CLAIM, DAMAGES OR
; OTHER LIABILITY, WHETHER IN AN ACTION OF CONTRACT, TORT OR OTHERWISE,
; ARISING FROM, OUT OF OR IN CONNECTION WITH THE SOFTWARE OR THE USE OR
; OTHER DEALINGS IN THE SOFTWARE.

; For more information, please refer to <http://unlicense.org/>

;.    ____ _    ____ ____ ____ _    ____
;.   | ___|    |    ||___]|    ||    [__
;.   |___]|___ |____||___]|    ||___ ___]

;;; Globals

; List for test cases
(defvar *demos* nil)

; Counter for test failures (this number will be the exit status of this code).
(defvar *fails* 0)

; To reset random number generator, reset this variable.
(defvar *seed* 10019)

;.    _ _ ____ ____ ____ ____ ____
;.   |\/||____||    |___/|    |[__
;.   |  ||    ||____|   \|____|___]

;;; Macros.

; Shorthand for accessing settings.
(defmacro ? (x) `(second(getf *settings* ',x)))

; Shorthand for nested struct access.
(defmacro o (s x &rest xs)
  (if xs `(o (slot-value ,s ',x) ,@xs) `(slot-value ,s ',x)))

; Anaphoic if.
(defmacro aif (expr then &optional else)
  `(let (it) (if (setf it ,expr) ,then ,else)))

; Loop over file
(defmacro with-csv ((lst file &optional out) &body body)
  (let ((str (gensym)))
    `(let (,lst)
       (with-open-file (,str ,file)
         (loop while (setf ,lst (read-line ,str nil)) do ,@body)
         ,out))))

; Ensure `a` has a cells `(x . number)` (where number defaults to 0).
(defmacro has (key dictionary)
  `(cdr (or (assoc ,key ,dictionary :test 'equal)
            (car (setf ,dictionary (cons (cons ,key 0) ,dictionary))))))

; Define a demo function (see examples at end of file).
(defmacro defdemo (name params &body body)
  `(progn (pushnew ',name *demos*) (defun ,name ,params  ,@body)))

; Defined struct with a constructor and a pretty print from `thing`.
(defmacro defthing (x &rest slots)
  `(defstruct
     (,x (:constructor ,(intern (format nil "%MAKE-~a" (symbol-name x))))
         (:include thing))
        ,@slots))

;.    ___ ____ ____ _    ____
;.     |  |   ||   ||    [__
;.     |  |___||___||___ ___]

;;; Library

;.     _   _   _  ._   _   _
;.    (_  (_) (/_ |   (_  (/_
;; coerce
; Cull silly white space.
(defun trim (s) (string-trim '(#\Space #\Tab) s))

; String to number (if we can).
(defun asAtom (s &aux (s1 (trim s)))
  (if (equal s1 "?") #\? (let ((x (ignore-errors (read-from-string s1))))
                           (if (numberp x) x s1))))

; String to list of strings
(defun asList (s &optional (sep #\,) (x 0) (y (position sep s :start (1+ x))))
  (cons (subseq s x y) (and y (asList s sep (1+ y)))))

; String to list of atoms
(defun asAtoms(s) (mapcar 'asAtom (asList s)))

;.     _       _  _|_  _   ._ _
;.    _>  \/  _>   |_ (/_  | | |
;.        /
(defun args ()
  #+clisp ext:*args* #+sbcl sb-ext:*posix-argv*)

(defun stop (&optional (status 0))
  #+sbcl (sb-ext:exit :code status) #+:clisp (ext:exit status))

(defun klass-slots (it)
  #+clisp (clos:class-slots (class-of it)) #+sbcl (sb-mop:class-slots (class-of
it)))

(defun klass-slot-definition-name (x)
  #+clisp (clos:slot-definition-name x) #+sbcl (sb-mop:slot-definition-name x))

(defun slots (it)
  (mapcar 'klass-slot-definition-name (klass-slots it)))

;.    ._   _.  ._    _|   _   ._ _
;.    |   (_|  | |  (_|  (_)  | | |
;; Random
; Unlike LISP, it is easy to set the seed of this random number genertor.
(labels ((park-miller () (setf *seed* (mod (* 16807.0d0 *seed*) 2147483647.0d00)
)
                         (/ *seed* 2147483647.0d0)))
  (defun randf (&optional (n 1)) (*   n (- 1.0d0 (park-miller)))) ;XX check this
  (defun randi (&optional (n 1)) (floor (* n (park-miller)))))

; Return sample from normal distribution.
(defun normal (&optional (mu 0) (sd 1))
  (+ mu (* sd (sqrt (* -2 (log (randf)))) (cos (* 2 pi (randf))))))

;.     _   _.  ._ _   ._   |   _
;.    _>  (_|  | | |  |_)  |  (/_
;.                    |
;; Sampe
; Any item
(defun anv (seq)    (elt seq (randi (length seq))))
(defun many (seq n) (let (a) (dotimes (i n a) (push (any seq) a))))

; Return `p`-th item from seq.
(defun per (seq &optional (p .5) &aux (v (coerce seq 'vector)))
  (elt v (floor (* p (length v)))))

; Find sd from a sorted list.
(defun sd (seq &optional (key 'identity))
  (if (<= (length seq) 5) 0
    (/ (- (funcall key (per seq .9)) (funcall key (per seq .1))) 2.56)))

; Return entropy of symbols in an assoc list.
(defun ent (alist &aux (n 0) (e 0))
  (dolist (two alist) (incf n (cdr two)))
  (dolist (two alist e) (let ((p (/ (cdr two) n))) (decf e (* p (log p 2))))))

;.    ._ _   o   _   _
;.    | | |  |  _>  (_
;; misc
; For each setting `x`, look for `-x` on the command line.
(defun update-settings-from-command-line (lst)
  (loop for (slot (help b4)) on lst by 'cddr do
    (setf (second (getf lst slot))
          (aif (member (format nil "--~a" slot) (args) :test 'equalp)
            (cond ((eq b4 t)   nil) ; boolean flags flip the default
                  ((eq b4 nil) t)   ; boolean flags flip the default
                  (t (asAtom (elt it 1))))
            b4))))

;.    ._   ._   _  _|_ _|_       ._   ._  o  ._   _|_
;.    |_)  |   (/_  |_  |_  \/   |_)  |   |  | |   |_
;.    |                     /    |
(defun round2 (number &optional (digits 2))
  (let* ((div (expt 10 digits))
         (tmp (/ (round (* number div)) div)))
    (if (zerop digits) (floor tmp) (float tmp))))

(defun round2s (seq &optional (digits 2))
  (map 'list (lambda (x) (round2 x digits)) seq))

(defun pretty (x  &optional (str t) (pre nil))
 (labels ((kid (x) (pretty x str (cons " " pre))))
  (format t "~%==> ~a ~a~%"  (type-of x) x)
  (print (numberp x))
  (print (thing-p x))
  (print (atom x))
  (print (consp x))
  (print (arrayp x))
  (cond
        ((numberp x) (print 1) (print x) (format str "~(~a~)~a~%" '("---") x))
        ((thing-p x) (print 2) (kid (cons (type-of x)
                                (mapcar (lambda (s) (list s (slot-value x s)))
                                        (slots x)))))
        ((atom   x) (format str "~(~a~)~a~%" pre x))
        ((consp  x) (dolist (y x) (kid y)))
        ((arrayp x) (map nil (lambda (y) (kid y))  x))
)))

;     w
; (defun pretty (lst &optional (str t) pre)
;   (labels ((item (lst pre)
;      (cond ( (consp lst)
;          t (pretty (first lst) str pre)
;                                          (when (rest lst)
;                                            (format str "~%~{~a~}" pre)
```

```lisp
;                                         (item (rest lst) pre)))))
;     (cond ((null lst)  (princ "()" str))
;           ((atom lst)  (princ lst str))
;           ((listp lst)
;            (princ "(" str) (item lst (cons "   " pre)) (princ ")" str)))))
;
(defstruct thing)

(defmethod xprint-object ((self thing) str)
  (pretty  self str))

;.      ._ _   _.  o  ._
;.      | | |  (_|  |  | |

; Handle tests within a test function"
(defun ok (test &optional (msg " "))
  (cond (test (format t "~aPASS ~a~%" #\Tab  msg))
        (t    (incf *fails* )
              (if (? dump)
                (assert test nil msg)
                (format t "~aFAIL ~a~%" #\Tab msg)))))

; Update *options* from command-line. Show help or run demo suite.
; Before demo, reset random number seed (and the settings).
; Return the number of fails to the operating system.
(defun main (&aux defaults)
  (labels ((fun (x) (find-symbol (string-upcase x)))
           (demo (todo) (when (fboundp todo)
                          (format t "~a~%"  todo)
                          (setf *settings* (copy-tree defaults)
                                *seed*     (? seed))
                          (funcall todo))))
    (update-settings-from-command-line *settings*)
    (setf defaults (copy-tree *settings*))
    (cond ((? help)                 (help *settings*))
          ((equalp "all" (? todo)) (dolist (one *demos*) (demo (fun one))))
          (t                        (demo (fun (? todo)))))
    (stop *fails*)))
```

```lisp
;.     ____ _    ____ ____ ____ ____ ____
;.     |    |    |__| [__  [__  |___ [__
;.     |___ |___ |  | ___] ___] |___ ___]

;;; Classes

;.     o  _
;.     |  _>

; The first/last char of a column name defines meta-knowledge for that column.
(defun is (s kind)
  (let
    ((post '((ignore #\X) (klass #\!) (less #\-) (more #\+) (goal #\+ #\- #\!)))
     (pre  '((num #\$))))
    (or (member (char s (1- (length s))) (cdr (assoc kind post)))
        (member (char s 0)               (cdr (assoc kind pre))))))
;.      _       ._ _
;.     _>  \/  | | |
;.         /
;; Sym
(defstruct (sym (:constructor %make-sym)) (n 0) at name all mode (most 0))

(defun make-sym (&optional (at 0) (name ""))
  (%make-sym :at at :name name))

(defmethod add ((self sym) x)
  (with-slots (n all mode most) self
    (unless (eq x #\?)
      (incf n)
      (let ((now (incf (has x all))))
        (if (> now most)
          (setf most now
                mode x)))))
  x)

(defmethod div ((self sym)) (ent (sym-all self)))
(defmethod mid ((self sym)) (sym-mode self))

;.     ._   _    ._ _
;.     | |  |_|  | | |
;; Num
(defstruct (num  (:constructor %make-num ))
  (n 0) at name
  (all (make-array 5 :fill-pointer 0 :adjustable t ))
  (max (? enough))
  ok w (hi -1E32) (lo 1E32))

(defun make-num (&optional (at 0) (name ""))
  (%make-num :at at :name name :w (if (is name 'less) -1 1)))

(defmethod add ((self num) x)
  (with-slots (n lo hi ok all max) self
    (unless (eq x #\?)
      (incf n)
      (setf lo (min x lo)
            hi (max x hi))
      (cond ((< (length all) max)  (setf ok nil) (vector-push-extend x all))
            ((< (randf) (/ max n)) (setf ok nil)
                                   (setf (elt all (randi (length all))) x)))))
  x)

(defmethod holds ((self num))
  (with-slots (ok all) self
    (unless ok (setf all (sort all '<)))
    (setf ok t)
    all))

(defmethod div ((self num)) (sd  (holds self)))
(defmethod mid ((self num)) (per (holds self)))
(defmethod norm ((self num) x)
  (if (equal x #\?)
    x
    (with-slots (lo hi) self
      (if (< (abs (- hi lo)) 1E-9) 0 (/ (- x lo) (- hi lo))))))
;.      _   _   |   _
;.     (_  (_)  |  _>
;; cols
(defstruct (cols (:constructor %make-cols)) all x y names klass)

(defun make-cols (names &aux (at -1) x y klass all)
  (dolist (s names (%make-cols :names names :all (reverse all)
                               :x (reverse x) :y (reverse y) :klass klass))
    (let ((now (funcall (if (is s 'num) 'make-num 'make-sym) (incf at) s)))
      (push now all)
      (when (not (is s 'ignore))
        (if (is s 'goal)  (push  now y) (push now x))
        (if (is s 'klass) (setf klass now))))))

;.      _   _   _
;.     (/_ (_| _>
;.          _|
;; egs

(defun adds (eg data)
  (if (stringp data)
    (with-csv (row data) (add eg (asAtoms row)))
    (map nil (lambda (row) (add eg row)) data))
  eg)

(defstruct (egs (:constructor %make-egs))
  cols (rows (make-array 5 :fill-pointer 0 :adjustable t)))

(defun make-egs (&optional data &aux (self (%make-egs)))
   (if data (adds self data) self))

(defmethod mid ((self egs) &aux (cols (o self cols y))) (mapcar 'mid cols))
(defmethod div ((self egs) &aux (cols (o self cols y))) (mapcar 'div cols))

(defmethod add ((self egs) row)
  (with-slots (rows cols) self
    (if cols
      (vector-push-extend (mapcar 'add (o cols all) row) rows)
      (setf cols (make-cols row)))))

(defmethod size ((self egs)) (length (o self rows)))

(defmethod clone ((self egs) &optional data)
  (adds (make-egs (list (o self cols names))) data))

(defmethod better ((self egs) row1 row2 &aux (s1 0) (s2 0))
  (let ((n (length (o self cols y))))
    (dolist (col (o self cols y)  (< (/ s1 n) (/ s2 n)))
      (let* ((a (norm col (elt row1 (o col at))))
             (b (norm col (elt row2 (o col at)))))
        (decf s1 (exp (/ (* (o col w) (- a b)) n)))
        (decf s2 (exp (/ (* (o col w) (- b a)) n)))))))

(defmethod betters ((self egs) &optional (rows (o self rows)))
  (sort rows (lambda (row1 row2) (better self row1 row2))))
;.     ____ _    _  _ ____ ___ ____ ____
;.     |    |    |  | [__   |  |___ |__/
;.     |___ |___ |__| ___]  |  |___ |  \

;;; Cluster
```

```
(defmethod dist ((self egs) row1 row2)
  (let ((n 0) (d 0) (p (? p)))
    (dolist (col (o self cols x) (expt (/ d n) (/ 1  p)))
      (let ((inc (dist col (elt row1 (o col at))
                           (elt row2 (o col at)))))
        (incf d (expt inc p))
        (incf n)))))

(defmethod dist ((self num) x y)
  (cond ((and (eq x #\?) (eq y #\?)) (return-from dist 1))
        ((eq x #\?) (setf y (norm self y)
                          x (if (< y .5) 1  0)))
        ((eq y #\?) (setf x (norm self x)
                          y (if (< x .5) 1 0)))
        (t          (setf x (norm self x)
                          y (norm self y))))
  (abs (- x y)))

(defmethod dist ((self sym) x y)
  (if (and (eq x #\?) (eq y #\?))
    0
    (if (equal x y) 0 1)))

(defmethod neighbors ((self egs) row1 &optional (rows (o self rows)))
  (labels ((f (row2) (cons (dist self row1 row2) row2)))
    (sort (map 'vector #'f rows) '< :key 'car)))

(defmethod far ((self egs) row &optional (rows (o self rows)))
  (cdr (per (neighbors self row rows) (? far))))
;.                  _
;.      |_   _.  |  |_
;.      | |  (_|  |  |_
;; half
(defstruct (half (:constructor %make-half)) eg lefts rights left right c border)

(defmethod dist2left ((self half) row)
  (with-slots (eg left right c)  self
    (let ((a (dist eg row left))
          (b (dist eg row right)))
      (max 0 (min 1 (/ (+ (* a a) (* c c) (- (* b b)))
                       (+ (* 2 c) 1E-32)))))))

(defmethod dist2lefts ((self half) rows)
  (sort (map 'list (lambda (r) (cons (dist2left self r) r))  rows) '< :key 'car)
)

(defmethod selects ((self half) row)
  (with-slots (lefts rights border) self
    (if (<= (dist2left self row) border) lefts rights)))

(defun  make-half (eg &optional (rows (o eg rows))
                        &aux      (self (%make-half :eg eg)))
  (let (some nleft)
    (with-slots (lefts rights left right c border)  self
      (setf some   (many rows (? some))
            nleft  (floor (* .5 (length rows)))
            left   (far eg (any some) some)
            right  (far eg left       some)
            c      (dist eg left right)
            lefts  (clone eg)
            rights (clone eg))
      (dolist (tmp (dist2lefts self rows) self)
        (add (if (>= (decf nleft) 0) lefts rights)  (cdr tmp))
        (if (zerop nleft)
          (setf border (car tmp)))))))

(defun best-rest (top)
  (let ((stop  (* 2  (expt (size top) (? min))))
        (rests (clone top)))
    (labels
      ((down (goods bads)
             (loop for bad across (o bads rows) do (add rests bad))
             (across goods))
       (across (eg)
               (if (< (size eg) stop)
                 eg
                 (with-slots (left right lefts rights) (make-half top (o eg rows
))
                   (if (better top left right)
                     (down lefts rights)
                     (down rights lefts)))))))
      (values (across top) rests))))

; (defstruct (cluster (:constructor %make-cluster)) egs top (rank 0) lefts right
s)
;
; (defmethod leaf ((self egs)) (not (o self lefts) (o self rights)))
;
; (defun make-cluster (top &optional (egs top))
;   (multiple-value-bind (half top (o egs rows))
;     (lefts rights left right)
;     (let ((self (%make-cluster :egs egs :top top :left left :right right
;                                :c c :border border)))
;       (when (>= (size egs) (* 2 (expt (size top) (? minItems))))
;         (when (<  (size lefts) (size egs))
;           (setf (o self lefts)  (cluster top lefts)
;                 (o self rights) (cluster top rights))))
;       self)))
;
; ; (defmethod show ((self cluster) &optional (pre ""))
;   (let ((front (format t "~a~a" pre (length (o egs rows)))))
;     (if  (leaf (o self egs))
;       (format t "~20a~a" front (mid (o self egs) (o self egs cols y)))
;       (progn
;         (print front)
;         (if (o self lefts)  (show (o lefts)  (format nil "|.. ~a" pre)))
;         (if (o self rights) (show (o rights) (format nil "|.. ~a" pre)))))))
;.    ___  ____ _  _ ____ ____/
;.    |__\ |___ |\/| |___ [__
;.    |__/ |___ |  | |___ ___]

;;; Demos

(defdemo .rand() (print (randf)))

(defdemo .egs()
  (let ((eg (make-egs (? file))))
    (holds (second (o eg cols y)))
    (print (last (o eg cols x)))))

(defdemo .div()
  (let ((eg (make-egs (? file))))
    (print (div eg))))

(defdemo .dist1(&aux (eg (make-egs (? file))))
  (print (sort (loop repeat 64 collect
                 (round2 (dist eg (any (o eg rows)) (any (o eg rows))) 2)) '<)))

(defdemo .dist2(&aux (out t) (eg (make-egs (? file))))
  (loop repeat 64 do
    (let ((a (any (o eg rows)))
          (b (any (o eg rows)))
          (c (any (o eg rows))))
      (setf out (and out (>= (+ (dist eg a b) (dist eg b c)) (dist eg a c))
                         (=     (dist eg a b) (dist eg b a))
                         (zerop (dist eg a a))))))
  (ok out "ands"))

(defdemo .clone(&aux (eg1 (make-egs (? file))))
  (let ((eg2 (clone eg1 (o eg1 rows))))
    (ok (equal (div (first (o eg1 cols y)))
               (div (first (o eg2 cols y)))))))

(defdemo .neighbors (&aux (eg (make-egs (? file))))
  (loop repeat 2 do
    (let* ((x (any (o eg rows)))
           (y (far eg x)))
      (format t "~%     ~a~%" x)
      (print (neighbors eg x (many (o eg rows) 10)))
      (format t "~%     ~a ~a~%"  y (dist eg x y)))))

(defdemo .mid (&aux (eg (make-egs (? file))))
  (format t "~a = ~a~%" (mapcar (lambda (c) (o c name)) (o eg cols y)) (mid eg)))

(defdemo .betters (&aux (eg (make-egs (? file))))
  (let* ((rows (betters eg))
         (n    (length rows)))
    (format t "~a~%" (mapcar (lambda (col) (o col name)) (o eg cols y)))
    (format t "all  ~a~%" (mid eg))
    (format t "best ~a~%" (mid (clone eg (subseq rows 0 32))))
    (format t "rest ~a~%" (mid (clone eg (subseq rows 33))))
    (format t "worst ~a~%" (mid (clone eg (subseq rows (- n 32)))))))

(defdemo .half (&aux (half (make-half (make-egs (? file)))))
  (format t "    ~a~%" (mapcar (lambda (col) (o col name)) (o half eg cols y)))
  (with-slots (eg lefts rights c) half
    (format t "all  ~a ~a~%left ~a ~a~%right ~a ~a~%c   ~a"
            (size eg)      (mid eg)
            (size lefts)   (mid lefts)
            (size rights)  (mid rights) c)))

(main)
```