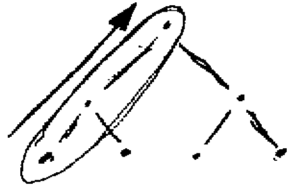
```
;defun diversity (pop)
  (length (remove-duplicates pop :test #'equal)))
```



default to very big

```
(defun evie1 (make-g0 mate fitness (max-pop))
  ;; Count up the populations tree
  pop <= (evie0 make-g0 mate fitness done-on-plateau)
  (while max-pop >
     ~~pop <- (evie0 make-g0 mate fitness done-on-plateau)~~
     pop <- (merge-pop pop
                ~~(evie0 make-g0 mate fitness~~
                   ~~done-on-plateau~~
                (evie1 make-g0 mate fitness
                   pop-count))

  (incf pop-count))
```

```
(defun evie1 (make-g0 mate fitness is-done
              &optional (pop-max 99999))

  (do ((pop-count 1 (1+ pop-count))
       (pop (evie0 make-g0 mate fitness
                   (make-is-done-low-diversity is-done))
            (evie1 make-g0 mate fitness is-done pop-count)))
      ((>= pop-count pop-max) pop)))
```

→ put merge-populations around this expression.

```
(defun merge-populations (p0 p1 fitness)
  (assert (eql (length p0) (length p1)))
  (sort-by-fitness
  (subseq (sort-by-fitness (append p0 p1) fitness)
          0
          (length p0)))

(defun make-diversity-done (&optional (other-done (constantly nil)))
  #'(λ (pop)
      (or (<= (diversity pop) 2)
          (funcall other-done pop))))
```

n

```
(defun mate-numeric (p0 p1)
  "Mate parents which are lists of numbers."
  (mapcar #'crossover-numeric p0 p1))

((defun crossover-numeric (n0 n1)
  (when (> n1 n0)
     (rotatef n0 n1))
  (assert (>= n0 n1))          <- make sure it's not zero
  (let ((r (random n0)))
    (+ (mod n1 r)
       (- n0 (mod n0 r)))))

(deftest test???? ()
  "Test crossover-numeric always returns number
in range & doesn't crash."
  (dotimes (i 10000)
    (let ((x (random 10000))
          (y (random 9999)))
      (assert (<= (crossover-numeric x y)
                  (+ x y))))))

'test????)
```

Margin note (bracketing crossover-numeric): nice but no. range too limited.