```
who= cs472 3 White,Steven
here= /home/swhite24/github/cs472/project3/472.3
total 60
 4 drwxr-xr-x 4 swhite24 swhite24  4096 2012-04-01 17:03 .
 4 drwxr-xr-x 4 swhite24 swhite24  4096 2012-03-12 17:29 ..
 4 -rw-r--r-- 1 swhite24 swhite24  1537 2012-03-31 02:51 de.lisp
 4 -rw-r--r-- 1 swhite24 swhite24   698 2012-03-17 04:03 gade_cf_genes.lisp
 4 -rw-r--r-- 1 swhite24 swhite24  2576 2012-03-17 03:58 gade_genes.txt
12 -rw-r--r-- 1 swhite24 swhite24 10594 2012-03-31 17:48 gade.lisp
 4 -rw-r--r-- 1 swhite24 swhite24   424 2012-03-17 03:56 gade_runtimes.txt
 4 -rw-r--r-- 1 swhite24 swhite24  1616 2012-03-31 03:03 ga.lisp
 4 -rw-r--r-- 1 swhite24 swhite24   659 2012-03-17 03:13 get_genes.lisp
 4 -rw-r--r-- 1 swhite24 swhite24   988 2012-03-17 03:49 get_runtimes.lisp
 4 drwxr-xr-x 2 swhite24 swhite24  4096 2012-04-01 16:57 latex
 4 -rw-r--r-- 1 swhite24 swhite24   232 2012-04-01 17:02 main.lisp
 4 drwxr-xr-x 2 swhite24 swhite24  4096 2012-04-01 17:03 unused


--------------------------------------
running ...

;testing  !RANDS
;testing  !TIME-IT
;testing  !SHELL->OUTPUT
; fail : expected (boot.lisp city.dot city.dot.png go go_bash go_lisp
                   graph-util.lisp known-city.dot known-city.dot.png
                   tricks.lisp wumpus.lisp)
; pass : 2 =  66.7%
; fail : 1 =  33.3%
NIL




=====|  de.lisp |===========================
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;; DE specific ;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;

(defun run_de (&key (c_freq 0.5) (scale_fact 0.7) (gens 100)
               (n 10) (summarize_freq 10) (np 10000)
               (obj_func #'single_obj)
               (compare_func #'get_parent)
               (mem (make-rat_mem :c_freq c_freq
                                  :scale_fact scale_fact)))
  (run_alg #'de_candidate c_freq scale_fact gens n np
           summarize_freq obj_func compare_func mem))

(defun run_de_dec (&key (c_freq 0.5) (scale_fact 0.7) (gens 100)
                   (n 10) (summarize_freq 10)
                   (obj_func #'single_obj)
                   (mem (make-rat_mem :c_freq c_freq
                                      :scale_fact scale_fact)))
  (run_alg #'de_candidate c_freq scale_fact gens n 10
           summarize_freq obj_func #'closest_dec mem))

(defun run_de_obj (&key (c_freq 0.5) (scale_fact 0.7) (gens 100)
               (n 10) (summarize_freq 10)
               (obj_func #'single_obj)
               (mem (make-rat_mem :c_freq c_freq
                                  :scale_fact scale_fact)))
  (run_alg #'de_candidate c_freq scale_fact gens n 10
           summarize_freq obj_func #'closest_obj mem))

(defun de_candidate (mem parent)
  "produces a child gene based on parent and 3 others"
  (labels ((candidate1 (x y z)
             (min 10
                  (max 1 (round (+ x (* (rat_mem-scale_fact mem)
                                        (- y z)))))))
           (cross-over (parent child)
             (if (<= (randf 1.0) (rat_mem-c_freq mem))
                 parent child)))
```



```
    (mapcar #'cross-over (rat-genes parent)
            (mapcar #'candidate1
                    (rat-genes (any_rat mem))
                    (rat-genes (any_rat mem))
                    (rat-genes (any_rat mem))))))




=====|  gade_cf_genes.lisp |===========================
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;; Used to generate average gene with constant number ;;
;; of generations and varying scale factor ;;;;;;;;;;;;

(load "main.lisp")

(let ((pop_size 100000))
  (labels ((header (name size)
             (format t "~11:@<~a~>~8:@<~a~>" name size)))
    (dolist (sf '(0.1 0.3 0.5 0.7 0.9))
      (header 'ga_single sf)
      (run_ga :gens pop_size :n 1 :scale_fact sf)
      (header 'de_single sf)
      (run_de :gens pop_size :n 1 :scale_fact sf)
      (header 'ga_double sf)
      (run_ga :gens pop_size :n 1 :scale_fact sf :obj_func #'two_obj)
      (header 'de_double sf)
      (run_de :gens pop_size :n 1 :scale_fact sf :obj_func #'two_obj))))



=====|  gade_genes.txt |===========================
 GA_SINGLE   1000    6.01     7.21     4.93     6.68     2.59     6.31    12.12
    13.40
 DE_SINGLE   1000   18.96     5.35     2.94     5.53     7.58     5.63     7.55
    17.62
 GA_DOUBLE   1000    2.94     5.64     3.53     7.28     4.93     5.49     2.71
    6.53
 DE_DOUBLE   1000   10.70     4.35     6.36     8.88    11.84     6.14    12.17
    6.00
 GA_SINGLE   5000   54.55     4.77    75.93    16.95     1.48     1.54     2.64
    2.92
 DE_SINGLE   5000   13.75    78.92    34.59    26.51     5.33    33.78    10.01
    34.05
 GA_DOUBLE   5000    8.81     6.81     2.46     1.65     2.16    63.39    13.29
    2.91
 DE_DOUBLE   5000   61.89     8.92     6.91    19.92     6.66    22.96    23.76
    28.37
 GA_SINGLE  10000    2.98    14.45     1.51     1.62     1.96   100.23     7.63
    15.27
 DE_SINGLE  10000  327.53   113.31   411.14   102.55    68.50   223.75   135.91
   413.56
 GA_DOUBLE  10000    1.63    14.98     2.72     2.54     2.09     1.67   173.95
    22.47
 DE_DOUBLE  10000  183.98   219.06   165.08    86.48    59.39   249.06   337.27
   571.37
 GA_SINGLE  25000  257.71    32.27   194.02     1.93     3.63     4.26     1.66
    2.30
 DE_SINGLE  25000  481.62   332.41   507.76   327.48   304.84   526.40   359.08
   520.66
 GA_DOUBLE  25000  278.79    40.45     3.35     1.77     2.68   368.54     5.25
    1.37
 DE_DOUBLE  25000  488.83   449.00   524.23   358.87   265.82   484.74   547.44
   633.82
 GA_SINGLE  50000   71.76     2.23     3.00   215.00     1.82     1.61     2.15
    5.36
 DE_SINGLE  50000  435.55   576.70   446.32   385.15   277.90   523.55   473.81
   623.64
 GA_DOUBLE  50000   58.05     1.68     1.72     2.58     1.40   236.88     2.13
    8.28
 DE_DOUBLE  50000  618.03   392.88   507.87   362.57   335.74   530.75   395.61
   421.83
 GA_SINGLE  75000   82.35     8.11     9.67   235.22     1.88     1.34     5.20
```

```
      3.14
    DE_SINGLE  75000   563.64   524.74   573.30   387.86   320.18   530.18   389.94
     424.19
    GA_DOUBLE  75000   368.72    90.42     3.32     2.32     8.11   204.46     6.79
      2.12
    DE_DOUBLE  75000   571.30   572.15   350.84   391.42   429.05   367.91   418.42
     532.48
    GA_SINGLE  100000   40.48    17.32     1.37   328.92     1.51    10.71     3.67
      5.85
    DE_SINGLE  100000 625.13   493.91   442.01   499.01   314.43   368.95   593.81
     467.59
    GA_DOUBLE  100000   73.54     8.84     4.44   331.15     2.63     1.82     5.63
      5.06
    DE_DOUBLE  100000 348.16   510.24   471.66   469.08   137.23   507.52   566.67
     491.24


=====|  gade.lisp |============================
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;; GA & DE shared constants & functions ;;;;;;;;;

;; size of landscape
(defparameter *width* 100)
(defparameter *height* 30)

;; oasis in landscape
(defparameter *jungle* '(45 10 10 10))

;; constants
(defparameter *plant_energy* 80)
(defparameter *reproduction-energy* 200)

;; counters
(defparameter *dead_rats* 0)
(defparameter *plants_eaten* 0)

;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;; Memory of all rats & plants ;;;;;;;;;;;;;;;;;;
(defstruct rat_mem
  (c_freq 0.5)
  (scale_fact 0.3)
  ;; unique id for future rats
  (current 0)
  ;; generation counter, used for aging rats
  (gen 0)
  (np 50)
  (all_rats (make-hash-table :test #'equal))
  (all_plants (make-hash-table :test #'equal)))

;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;; Struct for rat ;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
(defstruct rat
  (x (randi *width*))
  (y (randi *height*))
  (energy 1000)
  (dir 0)
  ;; unique id of parent in all_rats
  (parent most-positive-fixnum)
  ;; generation when rat was created
  (creation -25)
  (id 0)
  (genes (loop for x from 1 to 8
               collect (1+ (randi 10)))))


;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;; Rat_mem methods ;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
(defmethod update_mem ((mem rat_mem) candidate_func freq
                        obj_func compare_func)
  "update each rat, add plants, and remove dead rats"
  (with-slots (all_plants all_rats gen) mem
    (maphash #'(lambda (key r)
```



```
                 (update_rat r mem candidate_func
                             obj_func compare_func)) all_rats)
    (add_plants mem)
    (kill_rats mem)
    (summarize mem freq)
    (incf gen)))

(defmethod add_rat ((mem rat_mem) new_rat)
  "add rat to list of living rats..if no rat is passed
   a default will be created"
  (with-slots (all_rats current) mem
    (setf (gethash current all_rats)
          (if (null new_rat)
              (make-rat :id current)
              new_rat))
    (incf current)))

(defmethod any_rat ((mem rat_mem))
  "select any currently living rat"
  (with-slots (all_rats current) mem
    (let ((rand (randi current)))
      (or (gethash rand all_rats)
          (any_rat mem)))))

(defmethod random_plant ((mem rat_mem) left top width height)
  "create plant within given constraints"
  (with-slots (all_plants) mem
    (let ((pos (cons (+ left (random width)) (+ top (random height)))))
      (setf (gethash pos all_plants) t))))

(defmethod add_plants ((mem rat_mem))
  "add plant in jungle and somewhere else"
  (apply #'random_plant (cons mem *jungle*))
  (random_plant mem 0 0 *width* *height*))

(defmethod kill_rats ((mem rat_mem))
  "kills off rats with energy <= 0"
  (with-slots (all_rats) mem
    (let ((dead_rats '()))
      (maphash #'(lambda (key r)
                   (if (<= (rat-energy r) 0)
                       (push key dead_rats)))
               all_rats)
      (dolist (key dead_rats)
        (incf *dead_rats*)
        (remhash key all_rats)))))

(defmethod summarize ((mem rat_mem) n)
  "prints summary of population every n generations"
  (with-slots (gen all_rats) mem
    (if (= (mod (rat_mem-gen mem) n) 0)
        (progn (format t "~%gen: ~a living rats: ~a "
                       gen (hash-table-count all_rats))
               (format t "dead rats: ~a plants eaten: ~a~%"
                       *dead_rats* *plants_eaten*)
               (average_gene mem)))))

(defmethod average_gene ((mem rat_mem))
  "print average gene for all living rats"
  (with-slots (all_rats) mem
    ;(format t "Average gene: ")
    (let ((total_gene (make-list 8 :initial-element 0)))
      (maphash #'(lambda (key r)
                   (setf total_gene (mapcar #'+ total_gene (rat-genes r))))
               all_rats)
      (mapc #'(lambda (x) (format t "~9:@<~,2F~>" x))
            (mapcar #'(lambda (x) (/ x (hash-table-count all_rats)))
                    total_gene))
      (format t "~%"))))


;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;; Rat methods ;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
```

```
(defmethod update_rat ((r rat) (mem rat_mem) candidate_func
                       obj_func compare_func)
  "a day in the life of a rat"
  (turn_rat r)
  (move_rat r)
  (eat_rat r mem)
  (reproduce_rat r mem candidate_func)
  (funcall obj_func r mem #'score_rat_energy
           (funcall compare_func r mem)))

(defmethod get_parent ((r rat) (mem rat_mem))
  (rat-parent r))

(defmethod move_rat ((r rat))
  "move rat in direction indicated by dir, then decrement energy"
  (with-slots (dir x y energy) r
    (setf x (mod (+ x (cond ((and (>= dir 2) (< dir 5)) 1)
                            ((or (= dir 1) (= dir 5)) 0)
                            (t -1))
                    *width*) *width*)
          y (mod (+ y (cond ((and (>= dir 0) (< dir 3)) -1)
                            ((and (>= dir 4) (< dir 7)) 1)
                            (t 0))
                    *height*) *height*))
    (decf energy)))

(defmethod turn_rat ((r rat))
  "turn rat based on genes"
  (with-slots (dir genes) r
    (let ((x (randi (apply #'+ genes))))
      (labels ((angle (genes x)
                 (let ((xnu (- x (car genes))))
                   (if (< xnu 0)
                       0
                       (1+ (angle (cdr genes) xnu))))))
        (setf dir
              (mod (+ dir (angle genes x)) 8))))))

(defmethod eat_rat ((r rat) (mem rat_mem))
  "check if rat is near plant, if so consume it and add
   plant_energy to rats curren energy"
  (with-slots (x y energy) r
    (with-slots (all_plants) mem
      (let ((pos (cons x y)))
        (when (gethash pos all_plants)
          (incf *plants_eaten*)
          (incf energy *plant_energy*)
          (remhash pos all_plants))))))

(defmethod reproduce_rat ((r rat) (mem rat_mem) candidate_func)
  "if a rat has enough energy, create a child with genes
   from de_candidate, then check who survives"
  (with-slots (energy id creation) r
    (when (and (>= energy *reproduction-energy*)
               (> (- (rat_mem-gen mem) creation) 25))
      (setf energy (ash energy -1))
      (let ((child (copy-structure r)))
        (setf (rat-genes child) (funcall candidate_func mem r)
              (rat-creation child) (rat_mem-gen mem)
              (rat-parent child) id
              (rat-id child) (rat_mem-current mem))
        (add_rat mem child)))))

(defmethod closest_dec ((r rat) (mem rat_mem))
  "return rat with most similar gene"
  (with-slots (genes) r
    (let ((distance most-positive-fixnum)
          best)
      (maphash #'(lambda (key val)
                   (let ((diff 0)
                         (genes1 (rat-genes val)))
                     (loop for i from 0 to 7 do
```



```
                       (incf diff (abs (- (nth i genes)
                                          (nth i genes1)))))
                   (if (and (< diff distance)
                            (not (equal val r)))
                       (setf distance diff
                             best key))))
               (rat_mem-all_rats mem))
      best)))

(defmethod closest_obj ((r rat) (mem rat_mem))
  "return closest rat in objective space, i.e. the one
   with the most similar energy value"
  (let ((distance most-positive-fixnum)
        best)
    (maphash #'(lambda (key val)
                 (let ((new_dist (abs (- (score_rat_energy r mem)
                                         (score_rat_energy val mem)))))
                   (if (and (< new_dist distance)
                            (not (equal val r)))
                       (setf distance new_dist
                             best key))))
             (rat_mem-all_rats mem))
    best))

(defmethod single_obj ((r rat) (mem rat_mem) obj_func compare_to)
  "check if rat is old enough to be killed, then compare
   its score with parent score.  loser dies."
  (with-slots (energy creation) r
    (with-slots (all_rats gen) mem
      (when (and (= (- gen creation) 25)
                 (gethash compare_to all_rats))
        (let* ((p (gethash compare_to all_rats))
               (p_age (- gen (rat-creation p)))
               (c_score (funcall obj_func r mem))
               (p_score (funcall obj_func p mem)))
          (when (> p_age 25)
            (if (> c_score p_score)
                (and (remhash (rat-id p) all_rats)
                     (incf *dead_rats*))
                ;(setf (rat-energy (gethash compare_to all_rats)) 0))
              (if (< c_score p_score)
                  ;(setf energy 0)
                  (and (remhash (rat-id r) all_rats)
                       (incf *dead_rats*)))))))))

(defmethod two_obj ((r rat) (mem rat_mem) obj_func compare_to)
  "Check if rat is old enough to be killed, then compare
   energy and children score with parent.  If one dominates
   the other, the other dies. If neither dominates, both live."
  (with-slots (creation energy) r
    (with-slots (all_rats gen) mem
      (when (and (= (- gen creation) 25)
                 (gethash compare_to all_rats))
        (let* ((p (gethash compare_to all_rats))
               (p_age (- gen (rat-creation p)))
               (child_score_en (score_rat_energy r mem))
               (child_score_ch (score_rat_children r mem))
               (p_score_en (score_rat_energy p mem))
               (p_score_ch (score_rat_children p mem)))
          (when (>= p_age 25)
            (if (< child_score_en p_score_en)
                (if (<= child_score_ch p_score_ch)
                    (setf energy 0)))
              (if (< child_score_ch p_score_ch)
                  (if (<= child_score_en p_score_en)
                      (setf energy 0)))
              (if (< p_score_en child_score_en)
                  (if (<= p_score_ch child_score_ch)
                      (setf (rat-energy (gethash compare_to all_rats))
                            0)))
              (if (< p_score_ch child_score_ch)
                  (if (<= p_score_en child_score_en)
                      (setf (rat-energy (gethash compare_to all_rats))
```

```
                            0)))))))))

    (defmethod kill_rat ((r rat))
      "true if energy <= 0"
    415 (with-slots (energy) r
          (if (<= energy 0)
              (progn (incf *dead_rats*) t))))

    (defmethod score_rat_genes ((r rat))
    420 "scores rat based on percentage of values in either
         the front or back of gene, max of 1"
        (with-slots (genes) r
          (let ((genes_sum (apply #'+ genes))
                (early_sum (apply #'+ (butlast genes 5)))
    425         (late_sum (apply #'+ (last genes 3))))
            (max (/ early_sum genes_sum)
                 (/ late_sum genes_sum)))))

    (defmethod score_rat_children ((r rat) (mem rat_mem))
    430 "score rat based on the number times a child was
         successfully created, max of 1"
        (with-slots (creation id) r
          (with-slots (all_rats gen) mem
            (let ((child_count 0))
    435       (maphash #'(lambda (key r)
                           (when (= (rat-parent r) id)
                             (incf child_count)))
                       all_rats)
              (/ child_count
    440          (- gen creation))))))

    (defmethod score_rat_energy ((r rat) (mem rat_mem))
      "scores rat based on energy, max of 1"
      (with-slots (energy) r
    445 (/ energy 1000)))


    ;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
    ;; Util Functions ;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
    450
    (defun run_alg (alg c_freq scale_fact gens n init summarize_freq
                    obj_func compare_func mem)
      (when (> n 0)
        (setf *dead_rats* 0)
    455 (setf *plants_eaten* 0)
        ;; initial population - large enough
        (dotimes (i init)
          (add_rat mem nil))
        ;; conduct generations
    460 (dotimes (i gens)
          (update_mem mem alg summarize_freq obj_func compare_func))
        (summarize mem summarize_freq)
        ;(average_gene mem)
        (run_alg alg c_freq scale_fact gens (1- n) init
    465          summarize_freq obj_func compare_func
                 (make-rat_mem :c_freq c_freq
                               :scale_fact scale_fact))))

    (defun print_rats (mem)
    470 (maphash #'(lambda (key r)
                     (print r))
                (rat_mem-all_rats mem)))



    475


    =====|  gade_runtimes.txt |============================
       N     GA_SINGLE  DE_SINGLE  GA_DOUBLE  DE_DOUBLE
      1000     0.136      0.120      0.184      0.168
    480 5000     0.752      0.684      0.980      0.920
     10000     1.548      1.448      1.961      1.904
     25000     3.820      4.228      5.029      5.340
     50000     7.724     10.253      9.885     12.516
```



```
     75000    11.529     18.205     14.997     22.010
485  100000   15.225     28.289     20.206     32.770



    =====|  ga.lisp |============================
    490 ;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
    ;; GA specific ;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;

    (defun run_ga (&key (c_freq 0.5) (scale_fact 0.7) (gens 100)
                   (n 10) (summarize_freq 10) (np 10000)
    495            (obj_func #'single_obj)
                   (compare_func #'get_parent)
                   (mem (make-rat_mem :c_freq c_freq
                                      :scale_fact scale_fact)))
      (run_alg #'ga_candidate c_freq scale_fact gens n np
    500       summarize_freq obj_func compare_func mem))

    (defun run_ga_dec (&key (c_freq 0.5) (scale_fact 0.7) (gens 100)
                       (n 10) (summarize_freq 10)
                       (obj_func #'single_obj)
    505                (compare_func #'get_parent)
                       (mem (make-rat_mem :c_freq c_freq
                                          :scale_fact scale_fact)))
      (run_alg #'ga_candidate c_freq scale_fact gens n 10
               summarize_freq obj_func #'closest_dec mem))
    510
    (defun run_ga_obj (&key (c_freq 0.5) (scale_fact 0.7) (gens 100)
                       (n 10) (summarize_freq 10)
                       (obj_func #'single_obj)
                       (compare_func #'get_parent)
    515                (mem (make-rat_mem :c_freq c_freq
                                          :scale_fact scale_fact)))
      (run_alg #'ga_candidate c_freq scale_fact gens n 10
               summarize_freq obj_func #'closest_obj mem))

    520 (defun ga_candidate (mem parent)
          "produce a child gene with mutation"
          (let* ((which (randi 8))
                 (plus_minus (randi 2))
                 (temp_genes (copy-list (rat-genes parent)))
    525          (gene_to_fiddle (elt temp_genes which)))
            (setf (elt temp_genes which)
                  (min 10 (max 1
                               (if (= plus_minus 0)
                                   (- gene_to_fiddle
    530                               (* gene_to_fiddle
                                          (rat_mem-scale_fact mem)))
                                   (+ gene_to_fiddle
                                      (* gene_to_fiddle
                                         (rat_mem-scale_fact mem)))))))
    535     temp_genes))



    =====|  get_genes.lisp |============================
    540 ;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
    ;; Used to generate average gene for single and ;;;;;;;
    ;; multi-objective runs. ;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;

    (load "main.lisp")
    545
    (labels ((header (name size)
               (format t "~11:@<~a~>~8:@<~a~>" name size)))
        (dolist (pop_size '(1000 5000 10000 25000 50000
                            75000 100000))
    550   (header 'ga_single pop_size)
          (run_ga :gens pop_size :n 1)
          (header 'de_single pop_size)
          (run_de :gens pop_size :n 1)
          (header 'ga_double pop_size)
    555   (run_ga :gens pop_size :n 1 :obj_func #'two_obj)
          (header 'de_double pop_size)
```

```
      (run_de :gens pop_size :n 1 :obj_func #'two_obj)))


=====|  get_runtimes.lisp |===========================
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;; Used to generate runtimes for each algorithm with ;;;
;; a varying number of generations ;;;;;;;;;;;;;;;;;;;;;

(load "main")

(format t "~8:@<~a~>~11:@<~a~>~11:@<~a~>~11:@<~a~>~11:@<~a~>~%"
        'n 'ga_single 'de_single 'ga_double 'de_double)

(let ((start_time (get-internal-run-time))
      (n          1))
  (dolist (pop_size '(1000 5000 10000 25000 50000 75000 100000))
    (format t "~8:@<~a~>" pop_size)
    (format t "~11:@<~,3f~>" (time-it n (run_ga :gens pop_size
                                               :n 1)))
    (format t "~11:@<~,3f~>" (time-it n (run_de :gens pop_size
                                               :n 1)))
    (format t "~11:@<~,3f~>" (time-it n (run_ga :gens pop_size
                                               :n 1
                                               :obj_func #'two_obj)))
    (format t "~11:@<~,3f~>~%" (time-it n (run_de :gens pop_size
                                                 :n 1
                                                 :obj_func #'two_obj))))
  (format t "~%~%Total time for all executions: ~f~%"
          (/ (- (get-internal-run-time) start_time)
             internal-time-units-per-second)))


=====|  latex |===========================



=====|  main.lisp |===========================
(handler-bind ((style-warning #'muffle-warning))
  (mapc 'load '(
                "../tricks.lisp"
                "unused/system.lisp"
                "unused/pick.lisp"
                "gade.lisp"
                "de.lisp"
                "ga.lisp"
                )))

(defun ! () (load "main.lisp"))

(defun main () (tests))



=====|  unused |===========================
```