```
############# Part 3 #############
2014-10-21 13:00:03.560003

Schaffer
--------------------------------------------------
SimulatedAnnealer
Best:  3366.5511
total time: 0.023s      mean time: 0.023s
               |             *, 3366.55, 3366.55, 3366.55, 3366.55, 3366.55



MaxWalkSat
Best:  12540.6658
total time: 0.012s      mean time: 0.012s
               |             *, 12540.67, 12540.67, 12540.67, 12540.67, 12540.67


==================================================
==================================================

Fonseca
--------------------------------------------------
SimulatedAnnealer
Best:  1.0168
total time: 0.033s      mean time: 0.033s
               |             *,   1.02,   1.02,   1.02,   1.02,   1.02


MaxWalkSat
Best:  1.9890
total time: 0.036s      mean time: 0.036s
               |             *,   1.99,   1.99,   1.99,   1.99,   1.99


==================================================
==================================================

Kursawe
--------------------------------------------------
SimulatedAnnealer
Best: -13.3122
total time: 0.027s      mean time: 0.027s
*              |              , -13.31, -13.31, -13.31, -13.31, -13.31


MaxWalkSat
Best: -8.8811
total time: 0.025s      mean time: 0.025s
*              |              ,  -8.88,  -8.88,  -8.88,  -8.88,  -8.88


==================================================
==================================================

ZDT1
--------------------------------------------------
SimulatedAnnealer
Best:  7.0290
total time: 0.060s      mean time: 0.060s
               |             *,   7.03,   7.03,   7.03,   7.03,   7.03


MaxWalkSat
Best:  9.2382
total time: 0.024s      mean time: 0.024s
               |             *,   9.24,   9.24,   9.24,   9.24,   9.24


==================================================
==================================================

ZDT3
--------------------------------------------------
SimulatedAnnealer
Best:  1.1569
total time: 0.061s      mean time: 0.061s
               |             *,   1.16,   1.16,   1.16,   1.16,   1.16


MaxWalkSat
Best:  1.5592
total time: 0.041s      mean time: 0.041s
               |             *,   1.56,   1.56,   1.56,   1.56,   1.56


==================================================
==================================================

Viennet3
--------------------------------------------------
SimulatedAnnealer
Best:  15.9872
total time: 0.025s      mean time: 0.025s
               |             *,  15.99,  15.99,  15.99,  15.99,  15.99

```

```

MaxWalkSat
Best:  18.9430
total time: 0.020s      mean time: 0.020s
               |             *,  18.94,  18.94,  18.94,  18.94,  18.94


==================================================
==================================================
############# Part 5 #############
2014-10-21 13:00:03.950148

Schaffer
--------------------------------------------------
SimulatedAnnealer
Best:  1972297.1176
total time: 0.159s      mean time: 0.005s
* ---------    |              , 6396.10, 95847.89, 294633.22, 1226169.50, 5618096.84


MaxWalkSat
Best:  50210.2488
total time: 0.709s      mean time: 0.024s
 * ----        |              , 1303.73, 10686.09, 18912.00, 53216.74, 113072.48


==================================================
==================================================

Fonseca
--------------------------------------------------
SimulatedAnnealer
Best:  1.0305
total time: 0.220s      mean time: 0.007s
               |           *  ,   1.01,   1.01,   1.02,   1.04,   1.07


MaxWalkSat
Best:  1.5569
total time: 1.392s      mean time: 0.046s
               |--     *      ,   1.11,   1.21,   1.54,   1.98,   1.99


==================================================
==================================================

Kursawe
--------------------------------------------------
SimulatedAnnealer
Best: -12.8642
total time: 0.176s      mean time: 0.006s
 *--           |              , -13.51, -13.38, -13.01, -12.67, -11.52


MaxWalkSat
Best: -9.7534
total time: 0.884s      mean time: 0.029s
   ----  *     |              , -11.52,  -9.84,  -8.89,  -8.87,  -8.83


==================================================
==================================================

ZDT1
--------------------------------------------------
SimulatedAnnealer
Best:  7.4748
total time: 0.399s      mean time: 0.013s
               |        -- *- ,   6.84,   7.37,   7.61,   7.72,   7.95


MaxWalkSat
Best:  9.5124
total time: 0.818s      mean time: 0.027s
               |       - *--  ,   8.57,   9.14,   9.52,   9.85,  10.62


==================================================
==================================================

ZDT3
--------------------------------------------------
SimulatedAnnealer
Best:  1.2086
total time: 0.400s      mean time: 0.013s
               |          -*  ,   1.16,   1.19,   1.20,   1.22,   1.28


MaxWalkSat
Best:  2.4486
total time: 1.051s      mean time: 0.035s
       -   *   |-------       ,   1.38,   1.59,   2.18,   2.92,   4.46


==================================================
==================================================

Viennet3
--------------------------------------------------
SimulatedAnnealer
Best:  16.0983
total time: 0.155s      mean time: 0.005s
               |             *,  15.99,  16.03,  16.08,  16.14,  16.23


MaxWalkSat
Best:  18.5002
total time: 0.691s      mean time: 0.023s
               |     -- *---  ,  16.30,  17.49,  18.21,  19.59,  21.25


==================================================
==================================================
############# Part 6 #############

rank ,         name ,    med   ,  iqr
----------------------------------------------------
   1 ,          SA  ,       9  ,     1 (      --- *--  |              ), 8.76,  9.46,  9.95, 10.44, 11.15
   2 ,          MWS ,      13  ,     2 (               |---  * ---    ),11.97, 12.99, 13.87, 14.58, 15.56
None
```

```
"""## Log Stuff

Adapted from [Dr. Tim Menzies’ logging code](https://github.com/timm/sbse14/blob
/master/log.py).

Logs are places to store records of past events. There are two types of logs:

+ _Num_ : for numbers
+ _Sym_ : for everything else.

Those logs can be queried to find e.g. the highest
and lowest value of the number seen so far. Alternatively,
they can be queried to return values at the same probability
as the current log contents.

### Max Log Size

To avoid logs consuming all memory, logs store at
most _The.cache.keep_ entries (e.g. 128):

+ If more
than that number of entries arrive, then some old
entry (selected at random) will be deleted.
+ The nature of this cache means that some rare
events might be missed. To check for that, running
the code multiple times and, each time, double the
cache size. Stop when doubling the cache size stops
changing the output.

Just as an example of that process, here we are logging 1,000,000 numbers in a l
og with a cache of size 16.
Note that the resulting cache is much smaller than 1,000,000 items. Also, the co
ntents of the cache
come from the entire range one to one million (so our log is not biased to just
the first few samples:

 % python −i log.py
 >>> The.cache.keep = 16
 >>> log = Num()
 >>> for x in xrange(1000000): log += x
 >>> sorted(log._cache)
 [77748, 114712, 122521, 224268,
 289880, 313675, 502464, 625036,
 661881, 663207, 680085, 684674,
 867075, 875594, 922141, 945896]
 >>>

 ### Caching Slow Reports

 Some of the things we want to report from these logs take a little while to cal
culate (e.g. finding the median
    requires a sort of a numeric cache):

+ Such reports should be run and cached so they can be accessed many time withou
t the need
for tedious recalculation.
+ These reports become outdated if new log information arrives so the following
code deletes these reports if ever new data arrives.
+ The protocol for access those reports is to call _log.has().x_ where "x" is a
field
generated by the report.  Log subclasses generate reports using the special _rep
ort()_ method
(see examples, below).

Just as an example of reporting, after the above run (where we logged 1,000,000
numbers), the following reports are available:

>>> log.has().lo
0
>>> log.has().hi
945896
>>> print log.has().median # 50th percentile
662544.0
```

```
>>> print log.has().iqr # (75−25)th percentile
205194

Note that our median is not as expected (it should be around half a million). Wh
y? Well, clearly a cache of size 16 is
too small to track a million numbers. So how many numbers do we need? Well, that
 depends on the distribution being explored
but here’s how the median is effected by cache size for uniform distributions:

>>> for size in [16,32,64,128,256]:
...     The.cache.keep=size
...     log = Num()
...     for x in xrange(1000000): log += x
...     print size, ":" log.has().median
...
16 : 637374.5
32 : 480145.5
64 : 520585.5
128 : 490742.0
256 : 470870.5


Note that we get pretty close to half a million with cache sizes at 32 or above.
 And the lesson: sometimes, a limited
sample can offer a useful approximation to a seemingly complex process.

## Standard Header
"""
from __future__ import division, print_function
import sys, random, math, datetime, time, re
from base import memo
import base
import functools

class Log(object):
    "Keep a random sample of stuff seen so far."

    def __init__(self, inits=None, label=None, max_size=256):
        self._cache            = []
        self._n                = 0
        self._report           = None
        self.label             = label or ’’
        self.max_size          = max_size
        self._valid_statistics = False
        if inits:
            map(self.__iadd__, inits)

    def random_index(self):
        return base.random_index(self._cache)

    def __iadd__(self, x):
        if x is None:
            return x

        if isinstance(x, Log):
            map(self.__iadd__, x._cache)

        self._n += 1
        changed = False

        # if cache has room, add item
        if self.max_size is None or len(self._cache) < self.max_size:
            changed = True
            self._cache.append(x)
        # cache is full: maybe replace an old item
        else:
            # items less likely to be replaced later in the run:
            # leads to uniform sample of entire run
            if random.random() <= self.max_size / self._n:
                changed = True
                self._cache[self.random_index()] = x

        if changed:
            self._invalidate_statistics()
            self._change(x)

        return self

    def __add__(self, x):
        inits = self._cache + x._cache
        return NumberLog(inits=inits, label=’generated via __add__’, max_size=No
ne)

    def any(self):
        return random.choice(self._cache)

    def report(self):
        if self._report is None:
            self._report = self.generate_report()
        return self._report

    def setup(self):
        raise NotImplementedError()

    def contents(self):
        # slow, but most generic copy implementation
        return copy.deepcopy(self._cache)

    def _invalidate_statistics(self):
        ’’’
        default implementation. if _valid_statistics is something other than
        a boolean, reimplement!
        ’’’
        self._valid_statistics = False

    def ish(self, *args, **kwargs):
        raise NotImplementedError()

    def _change(self, x):
        ’’’
        override to add incremental updating functionality
        ’’’
        pass

    def _prepare_data(self):
        s = ’_prepare_data() not implemented for ’ + self.__class__.__name__
        raise NotImplementedError(s)

    @staticmethod
    def log_for(t):
        if t == int or t == float or isinstance(t, (int, float)):
            return NumberLog()
        else:
            return SymbolLog()


def statistic(f):
    ’’’
    decorator for log functions that return statistics about contents.
    if _valid_statistics is False, generate valid stats before calling
    the wrapped function.
    ’’’
    @functools.wraps(f)
    def wrapper(*args, **kwargs):
        self = args[0]
        if not self._valid_statistics:
            self._prepare_data()
        return f(*args, **kwargs)

    return wrapper


"""
### Num

A _Num_ is a _Log_ for numbers.
```

```

+ Tracks _lo_ and _hi_ values.
+ Reports median and the IQR the (75−25)th range.
+ Generates numbers from the log by a three−way interpolation (see _ish()_).


"""
class NumberLog(Log):

    def __init__(self, *args, **kwargs):
        super(NumberLog, self).__init__(*args, **kwargs)
        assert self._n == 0

        # set to values that will be immediately overridden
        self.lo, self.hi = sys.maxint, −sys.maxint

    def _change(self, x):
        # update lo,hi
        self.lo = min(self.lo, x)
        self.hi = max(self.hi, x)

    def _prepare_data(self):
        if not self._valid_statistics:
            self._cache.sort()
        self._valid_statistics = True

    def contents(self):
        return list(self._cache)

    def norm(self,x):
        "normalize the argument with respect to maximum and minimum"
        if self.hi == self.lo:
            raise ValueError(’hi and lo of {} are equal’.format(self.__name__))
        return (x − self.lo) / (self.hi − self.lo)

    def generate_report(self):
        return memo(median=self.median(), iqr=self.iqr(),
            lo=self.lo, hi=self.hi)

    def ish(self,f=0.1):
        """return a num likely to be similar to/representative of
        nums in the distribution"""
        return self.any() + f*(self.any() − self.any())

    @statistic
    def median(self):
        # implementation from http://stackoverflow.com/a/10482734/3408454
        n = len(self._cache)

        if n % 2:
            return self._cache[n // 2]

        return (self._cache[n // 2] + self._cache[n // 2 − 1]) / 2



    def mean(self):
        n = len(self._cache)
        return sum(self._cache) / n

    @statistic
    def iqr(self):
        n = len(self._cache)
        return self._cache[int(n*.75)] − self._cache[int(n*.5)]

    def total(self):
        return sum(self._cache)

    def better(self, log2):
        if not self._cache or not log2._cache: return False
        if self.median() < log2.median(): return True
        if self.iqr() < log2.iqr(): return True
        return False

    @statistic
    def xtile(self, lo=0, hi=0.001,
            width=50,
            chops=[0.1, 0.3, 0.5, 0.7, 0.9],
            marks=["−", " ", " ", "−", " "],
            bar="|", star="*",
            show=" {: >6.2f}"):
        """The function _xtile_ takes a list of (possibly)
        unsorted numbers and presents them as a horizontal
        xtile chart (in ascii format). The default is a
        contracted _quintile_ that shows the
        10,30,50,70,90 breaks in the data (but this can be
        changed− see the optional flags of the function).
        """

        lo = min(lo, self._cache[0])
        hi = max(hi, self._cache[−1])
        if hi == lo:
            hi = hi + .001 # ugh


        pos = lambda p: self._cache[int(len(self._cache) * p)]
        place = lambda x: min(width−1, int(width * float((x − lo))/(hi − lo)))
        pretty = lambda xs: ’,’.join([show.format(x) for x in xs])

        what    = [pos(p)   for p in chops]
        where   = [place(n) for n in  what]

        out     = [’ ’] * width

        for one,two in base.pairs(where):
            for i in range(one, two):
                out[i] = marks[0]
            marks = marks[1:]

        out[int(width / 2)]  = bar
        out[place(pos(0.5))] = star

        return ’’.join(out) +  "," +  pretty(what)

"""

WARNING: the call to _sorted_ in _report()_ makes this code
a candidate for a massive CPU suck (it is always sorting newly arrived data).
So distinguish between _adding_ things to a log in the _last_ era and
using that information in the _next_ era (so the log from the last era
    is staple in the current).

### Sym

A _Sym_ is a _Log_ for non−numerics.

+ Tracks frequency counts for symbols, and the most common symbol (the _mode_);
+ Reports the entropy of the space (a measure of diversity: lower values mean fe
wer rarer symbols);
+ Generated symbols from the log by returning symbols at the same probability of
 the frequency counts (see _ish()_).

"""
class SymbolLog(Log):

    @property
    def valid_statistics(self):
        return self._counts is None

    def _invalidate_statistics(self):
        # ‘_counts is None‘ => invalidation of calculated statistics
        # _mode would be a bad idea: what’s the ’null’ equivalent,
        # when None is a valid index into _counts?
        self._counts = None

    def _prepare_data(self):
```

```
        counts = {}
        mode = None
        mode_count = 0

        for x in self._cache:
            c = counts[x] = counts.get(x, 0) + 1
            if c > mode_count:
                mode = x

        self._counts, self._mode = counts, mode
        return self._counts, self._mode

    @statistic
    def counts(self):
        return self._counts

    @statistic
    def mode(self):
        return self._mode

    @statistic
    def distribution(self):
        return {k: v / len(self._cache) for k, v in self.counts().items()}

    def generate_report(self):
        return memo(
            distribution = self.distribution(),
            entropy      = self.entropy(),
            mode         = self.mode())

    @statistic
    def ish(self):
        tmp = 0
        threshold = random.random()
        for k, v in self.distribution().items():
            tmp += v
            if tmp >= threshold:
                return k
        # this shouldn’t happen, but just in case...
        return random.choice(self._cache)

    @statistic
    def entropy(self,e=0):
        n = len(self._cache)
        for k, v in self.counts().items():
            p = v / n
            # TODO: understand this equation better
            e −= p * math.log(p, 2) if p else 0
        return e
```

```
from independent_variable import IndependentVariable; del independent_variable
from schaffer import Schaffer; del schaffer
from kursawe import Kursawe; del kursawe
from fonseca import Fonseca; del fonseca
from zdt1 import ZDT1; del zdt1
from zdt3 import ZDT3; del zdt3
from viennet3 import Viennet3; del viennet3

del model
```

```
# all adapted from Dr. Tim Menzies’ model code:
# https://github.com/timm/sbse14/blob/master/models.py

class Model(object):
    def __init__(self, independents=None, dependents=None,
        energy_min=None, energy_max=None, enforce_energy_constraints=False):
        if independents is None or dependents is None:
            raise ValueError

        self.xs = independents
        self.ys = dependents
        self.energy_max = energy_max
        self.energy_min = energy_min
        self.enforce_energy_constraints = enforce_energy_constraints

    def normalize(self, x):
        n = x − self.energy_min
        d = self.energy_max − self.energy_min
        try:
            return n / d
        except ZeroDivisionError:
            return 0.5

    def random_input_vector(self):
        return tuple(x() for x in self.xs)

    def __call__(self, v, norm=False):
        energy_vector = tuple(y(v) for y in self.ys)
        energy_total = sum(energy_vector)

        if self.enforce_energy_constraints:
            energy_errmsg =’current energy {} not in range [{}, {}]’.format(
                energy_total, self.energy_min, self.energy_max)

        if self.energy_min is None or self.energy_min > energy_total:
            if self.enforce_energy_constraints:
                raise ValueError(energy_errmsg)
            self.energy_min = energy_total

        if self.energy_max is None or energy_total > self.energy_max:
            if self.enforce_energy_constraints:
                raise ValueError(energy_errmsg)
            self.energy_max = energy_total

        return energy_vector

    def energy(self, energy_vector):
        return sum(energy_vector)
```

```
# all adapted from Dr. Tim Menzies' model code:
# https://github.com/timm/sbse14/blob/master/models.py

import random

class IndependentVariable(object):
    def __init__(self, min=None, max=None, type=float):
        self.min = min
        self.max = max
        self.type = type

    def __call__(self):
        if self.type == float:
            f = random.uniform
        elif self.type == int:
            f = random.randint

        return f(self.min, self.max)
```

```
# all adapted from Dr. Tim Menzies' model code:
# https://github.com/timm/sbse14/blob/master/models.py

from __future__ import division
import math

from model import Model
from independent_variable import IndependentVariable as IV
from witschey.base import memo_sqrt

class Fonseca(Model):
    def __init__(self, ivs=3):
        ivs = tuple(IV(min=-4, max=4) for _ in xrange(ivs - 1))

        def f1(xs):
            e = sum((x - (1 / memo_sqrt(i+1))) ** 2 for i, x in enumerate(xs))
            return 1 - math.exp(-e)

        def f2(xs):
            e = sum((x + (1 / memo_sqrt(i+1))) ** 2 for i, x in enumerate(xs))
            return 1 - math.exp(-e)

        super(Fonseca, self).__init__(independents=ivs, dependents=(f1, f2))
```

```
# all adapted from Dr. Tim Menzies’ model code:
# https://github.com/timm/sbse14/blob/master/models.py

from __future__ import division
import math

from model import Model
from independent_variable import IndependentVariable as IV

class Kursawe(Model):
    def __init__(self, ivs=3, a=0.8, b=3):
        ivs = tuple(IV(min=−5, max=5) for _ in xrange(ivs − 1))
        self.a = a
        self.b = b

        def f1(xs):
            rv = 0
            for i in xrange(len(xs) − 1):
                exponent = (−0.2) * math.sqrt(xs[i] ** 2 + xs[i+1] ** 2)
                rv += −10 * math.exp(exponent)
            return rv

        def f2(xs):
            f = lambda x: (math.fabs(x)**self.a) + (5 * math.sin(x)**self.b)
            return sum(f(x) for x in xs)

        super(Kursawe, self).__init__(independents=ivs, dependents=(f1, f2))
```

```
# all adapted from Dr. Tim Menzies’ model code:
# https://github.com/timm/sbse14/blob/master/models.py

from model import Model
from independent_variable import IndependentVariable as IV

class Schaffer(Model):
    def __init__(self, ivs=1):
        ivs = tuple(IV(min=−10^5, max=10^5) for _ in xrange(ivs))
        # we use def instead of lambdas so the functions keep their __name__s
        def f1(xs):
            return sum(x ** 2 for x in xs)
        def f2(xs):
            return sum((x − 2) ** 2 for x in xs)

        super(Schaffer, self).__init__(
            independents=ivs,dependents=(f1, f2))
```

```
# all adapted from Dr. Tim Menzies’ model code:
# https://github.com/timm/sbse14/blob/master/models.py

from __future__ import division
import math

from model import Model
from independent_variable import IndependentVariable as IV

class Viennet3(Model):

    def __init__(self):

        def f1(xs):
            x_1sq = xs[0] ** 2
            x_2sq = xs[1] ** 2
            a = 0.5 * x_1sq
            b = math.sin(x_1sq + x_2sq)
            return a + x_2sq + b

        def f2(xs):
            x_1 = xs[0]
            x_2 = xs[1]

            a = ((3 * x_1 − 2 * x_2 + 4) ** 2) / 8
            b = ((x_1 + x_2 + 1) ** 2) / 27

            return a + b + 15

        def f3(xs):
            x_1sq = xs[0] ** 2
            x_2sq = xs[1] ** 2

            a = 1 / (x_1sq + x_2sq + 1)
            b = 1.1 * math.exp(−x_1sq − x_2sq)

            return a − b

        ivs = (IV(min=−3, max=3), IV(min=−3, max=3))

        super(Viennet3, self).__init__(
            independents=ivs, dependents=(f1, f2, f3))
```

```
# all adapted from Dr. Tim Menzies’ model code:
# https://github.com/timm/sbse14/blob/master/models.py

from __future__ import division
import math

from model import Model
from independent_variable import IndependentVariable as IV

class ZDT1(Model):
    def __init__(self, ivs=30):

        def g(xs):
            return 1 + 9 * sum(xs[1:]) / (len(xs) − 1)

        def f1(xs):
            return xs[0]

        def f2(xs):
            gxs = g(xs)
            return gxs * (1 − math.sqrt(xs[0] / gxs))

        ivs = tuple(IV(min=0, max=1) for _ in xrange(30))
        super(ZDT1, self).__init__(independents=ivs, dependents=(f1, f2, g))
```

```
# all adapted from Dr. Tim Menzies’ model code:
# https://github.com/timm/sbse14/blob/master/models.py

from __future__ import division

import math

from model import Model
from independent_variable import IndependentVariable as IV
from witschey.base import memo_sqrt


class ZDT3(Model):

    def __init__(self, ivs=30):

        def g(xs):
            return 1 + 9 * sum(xs[1:]) / (len(xs) − 1)

        def f1(xs):
            return xs[0]

        def f2(xs):
            gxs = g(xs)
            a = 1 − memo_sqrt(xs[0] / gxs) − (xs[0] / gxs)
            a *= math.sin(10 * math.pi * xs[0])
            return gxs * a

        ivs = tuple(IV(min=0, max=1) for _ in xrange(30))

        super(ZDT3, self).__init__(independents=ivs, dependents=(f1, f2, g))
```

```
from simulated_annealer import SimulatedAnnealer ; del simulated_annealer
from maxwalksat import MaxWalkSat ; del maxwalksat

del searcher
```

```
from __future__ import division, unicode_literals

from witschey.base import memo, The

from datetime import datetime

class Searcher(object):

    def __new__(cls, *args, **kwargs):
        # construct our object
        future_self = super(Searcher, cls).__new__(cls, *args, **kwargs)

        name = cls.__name__
        # initialize a dict with searcher’s name
        # and the initialization time
        d = dict(searcher=name, initialized=datetime.now())

        # if there are global options for this class or its bases in The
        for k in [name] + [k.__name__ for k in cls.__bases__]:
            if hasattr(The, k):
                # add them to the dict
                d.update(getattr(The, k).__dict__)

        # then, add the kwargs to the constructor call to the dict.
        # NB: this happens after adding options from The, so
        #     call−specific options override the globals
        d.update(kwargs)

        # set our spec with the contents of the dict
        future_self.spec = memo(**d)

        return future_self

    def __init__(self, model, *args, **kw):
        self.model = model

    def run(*args, **kwargs):
        raise NotImplementedError()
```

```
from __future__ import division

import random
import numpy as np
from collections import defaultdict

from searcher import Searcher
from witschey.base import memo, tuple_replace
from witschey.log import NumberLog

class MaxWalkSat(Searcher):

    def __init__(self, model, *args, **kw):
        super(MaxWalkSat, self).__init__(model=model, *args, **kw)

    def local_search_inputs(self, bottom, top, n=10):
        chunk_length = (top − bottom) / n

        for a in np.arange(bottom, top, chunk_length):
            yield random.uniform(a, a + chunk_length)


    def run(self, text_report=True):
        rv = memo(report=’’)

        log_eras = self.spec.log_eras or self.spec.terminate_early
        self.lives = 4

        if log_eras:
            rv.era_logs_by_objective = {f.__name__: defaultdict(NumberLog)
                for f in self.model.ys}
            rv.era_logs_best_energy = defaultdict(NumberLog)

        def report(s):
            if text_report:
                rv.report += s

        self.terminate = False
        def end_era(evals, era_length, log_value):
            report(’\n{: .2}’.format(log_value) + ’ ’)

            self.lives −= 1
            eras = evals // era_length


            for logs in rv.era_logs_by_objective.values():
                if eras not in logs: break
                if len(logs.keys()) < 2: break

                prev_log = logs[logs.keys().index(eras) − 1]
                if logs[eras].better(prev_log): self.lives += 1

            if self.lives <= 0: self.terminate = True


        def log_era(evals, era_length, dependents_outputs):
            era = evals // era_length
            for f, v in dependents_outputs:
                rv.era_logs_by_objective[f.__name__][era] += v
                rv.era_logs_best_energy[era] += rv.best


        init = self.model.random_input_vector()
        solution = init
        state = solution
        current_energy = self.model.energy(self.model(state))
        rv.best = current_energy
        evals = 0

        report(’{: .2}’.format(rv.best) + ’ ’)


        while evals < self.spec.iterations:
            if self.terminate: break

            for j in range(20):
                if evals > self.spec.iterations or self.terminate:
                    break

                dimension = random.randint(0, len(state) − 1)
                if self.spec.p_mutation > random.random():
                    state = tuple_replace(state,
                        dimension, self.model.xs[dimension]())

                    current_energy = self.model.energy(self.model(state))

                    if current_energy < rv.best:
                        solution = state
                        rv.best = current_energy
                        report(’+’)
                    else:
                        report(’.’)

                    evals += 1


                    if evals % self.spec.era_length == 0:
                        end_era(evals, self.spec.era_length, rv.best)

                else:
                    for j in self.local_search_inputs(
                        self.model.xs[dimension].min,
                        self.model.xs[dimension].max
                        ):
                        if self.terminate: break

                        state = tuple_replace(state,
                            dimension, self.model.xs[dimension]())

                        current_energy = self.model(state)

                        if current_energy < rv.best:
                            solution = state
                            rv.best = current_energy
                            report(’|’)
                        else:
                            report(’.’)

                        evals += 1
                        if evals % self.spec.era_length == 0:
                            end_era(evals, self.spec.era_length, rv.best)
                if log_eras:
                    log_era(evals, self.spec.era_length,
                        zip(self.model.ys, self.model(solution)))

        rv.evaluations = evals
        return rv
```

```
from __future__ import division

import random
import math
from collections import defaultdict

from searcher import Searcher
from witschey.base import memo
from witschey.log import NumberLog


class SimulatedAnnealer(Searcher):
    def __init__(self, model, *args, **kw):
        super(SimulatedAnnealer, self).__init__(model=model, *args, **kw)

    def run(self, text_report=True):
        rv = memo(report=’’)
        if self.spec.log_eras:
            rv.era_logs_by_objective = {
                f.__name__: defaultdict(NumberLog)
                for f in self.model.ys
            }
            rv.era_logs_best_energy = defaultdict(NumberLog)
        def report_append(s):
            if text_report:
                rv.report += s

        init = self.model.random_input_vector()
        solution = init
        state = solution
        rv.best = self.model.energy(self.model(solution))

        def p(old, new, temp):
            """
            sets the threshold we compare to to decide whether to jump

            returns e^−((new−old)/temp)
            """
            numerator = new − old

            if not 0 <= numerator <= 1:
                numerator = old − new
            try:
                exponent = numerator / temp
            except ZeroDivisionError:
                return 0
            rv = math.exp(−exponent)
            if rv > 1:
                raise ValueError(’p returning greater than one’,
                    rv, old, new, temp)
            return rv

        report_append(’{: .2}’.format(rv.best) + ’ ’)
        self.lives = 4

        for k in range(self.spec.iterations):
            if self.lives <= 0: break
            neighbor_candidate = self.model.random_input_vector()
            neighbor = tuple(neighbor_candidate[i]
                if random.random() < self.spec.p_mutation else v
                for i, v in enumerate(state))

            rv.best = self.model.energy(self.model(solution))
            neighbor_energy = self.model.energy(self.model(neighbor))
            current_energy  = self.model.energy(self.model(state))


            if neighbor_energy < rv.best:
                solution = neighbor
                rv.best = neighbor_energy
                report_append(’!’)

            if neighbor_energy < current_energy:
                state = neighbor
                report_append(’+’)
            else:
                good_idea = p(
                    self.model.normalize(current_energy),
                    self.model.normalize(neighbor_energy),
                    k / self.spec.iterations)
                if good_idea < random.random():
                    state = neighbor
                    report_append(’?’)

            report_append(’.’)

            if self.spec.log_eras or self.spec.terminate_early:
                era = k // self.spec.era_length
                for f, v in zip(self.model.ys, self.model(solution)):
                    rv.era_logs_best_energy[era] += rv.best
                    rv.era_logs_by_objective[f.__name__][era] += v

            if k % self.spec.era_length == 0 and k != 0:
                report_append(’\n’ + ’{: .2}’.format(rv.best) + ’ ’)

                self.lives −= 1
                eras = k // self.spec.era_length

                for logs in rv.era_logs_by_objective.values():
                    if eras not in logs: break
                    if len(logs.keys()) < 2: break

                    prev_log = logs[logs.keys().index(eras) − 1]
                    if logs[eras].better(prev_log): self.lives += 1

        return rv
```