Intro

This describes an adaptive, stable, natural mergesort, modestly called timsort (hey, I earned it <wink>). It has supernatural performance on many kinds of partially ordered arrays (less than lg(N!) comparisons needed, and as few as N-1), yet as fast as Python's previous highly tuned samplesort hybrid on random arrays.

In a nutshell, the main routine marches over the array once, left to right, alternately identifying the next run, then merging it into the previous runs "intelligently". Everything else is complication for speed, and some hard-won measure of memory efficiency.

Comparison with Python's Samplesort Hybrid

+ timsort can require a temp array containing as many as N//2 pointers, which means as many as 2*N extra bytes on 32-bit boxes. It can be expected to require a temp array this large when sorting random data; on data with significant structure, it may get away without using any extra heap memory. This appears to be the strongest argument against it, but compared to the size of an object, 2 temp bytes worst-case (also expected-case for random data) doesn't scare me much.

  It turns out that Perl is moving to a stable mergesort, and the code for that appears always to require a temp array with room for at least N pointers. (Note that I wouldn't want to do that even if space weren't an issue; I believe its efforts at memory frugality also save timsort significant pointer-copying costs, and allow it to have a smaller working set.)

+ Across about four hours of generating random arrays, and sorting them under both methods, samplesort required about 1.5% more comparisons (the program is at the end of this file).

+ In real life, this may be faster or slower on random arrays than samplesort was, depending on platform quirks. Since it does fewer comparisons on average, it can be expected to do better the more expensive a comparison function is. OTOH, it does more data movement (pointer copying) than samplesort, and that may negate its small comparison advantage (depending on platform quirks) unless comparison is very expensive.

+ On arrays with many kinds of pre-existing order, this blows samplesort out of the water. It's significantly faster than samplesort even on some cases samplesort was special-casing the snot out of. I believe that lists very often do have exploitable partial order in real life, and this is the strongest argument in favor of timsort (indeed, samplesort's special cases for extreme partial order are appreciated by real users, and timsort goes much deeper than those, in particular naturally covering every case where someone has suggested "and it would be cool if list.sort() had a special case for this too ... and for that ...").

+ Here are exact comparison counts across all the tests in sortperf.py, when run with arguments "15 20 1".

```
  Column Key:
      *sort: random data
      \sort: descending data
      /sort: ascending data
      3sort: ascending, then 3 random exchanges
      +sort: ascending, then 10 random at the end
      ~sort: many duplicates
      =sort: all equal
      !sort: worst case scenario

  First the trivial cases, trivial for samplesort because it special-cased
  them, and trivial for timsort because it naturally works on runs.  Within
  an "n" block, the first line gives the # of compares done by samplesort,
  the second line by timsort, and the third line is the percentage by
  which the samplesort count exceeds the timsort count:

      n   \sort   /sort   =sort
-------  ------  ------  ------
  32768   32768   32767   32767  samplesort
          32767   32767   32767  timsort
          0.00%   0.00%   0.00%  (samplesort - timsort) / timsort

  65536   65536   65535   65535
          65535   65535   65535
          0.00%   0.00%   0.00%

 131072  131072  131071  131071
         131071  131071  131071
          0.00%   0.00%   0.00%

 262144  262144  262143  262143
         262143  262143  262143
          0.00%   0.00%   0.00%

 524288  524288  524287  524287
         524287  524287  524287
          0.00%   0.00%   0.00%

1048576 1048576 1048575 1048575
        1048575 1048575 1048575
          0.00%   0.00%   0.00%

  The algorithms are effectively identical in these cases, except that
  timsort does one less compare in \sort.

  Now for the more interesting cases.  lg(n!) is the information-theoretic
  limit for the best any comparison-based sorting algorithm can do on
  average (across all permutations).  When a method gets significantly
  below that, it's either astronomically lucky, or is finding exploitable
  structure in the data.

      n   lg(n!)    *sort    3sort     +sort   %sort    ~sort     !sort
-------  -------   ------  -------  -------   ------  -------  --------
  32768   444255   453096   453614    32908   452871   130491    469141 old
```

```
                  448885    33016    33007    50426   182083     65534 new
                   0.94% 1273.92%   -0.30%  798.09%  -28.33%   615.87% %ch from
new

  65536   954037  972699   981940    65686   973104   260029   1004607
                  962991    65821    65808   101667   364341    131070
                   1.01% 1391.83%   -0.19%  857.15%  -28.63%   666.47%

 131072  2039137 2101881  2091491   131232  2092894   554790   2161379
                 2057533   131410   131361   206193   728871    262142
                   2.16% 1491.58%   -0.10%  915.02%  -23.88%   724.51%

 262144  4340409 4464460  4403233   262314  4445884  1107842   4584560
                 4377402   262437   262459   416347  1457945    524286
                   1.99% 1577.82%   -0.06%  967.83%  -24.01%   774.44%

 524288  9205096 9453356  9408463   524468  9441930  2218577   9692015
                 9278734   524580   524633   837947  2916107   1048574
                  1.88%  1693.52%   -0.03% 1026.79%  -23.92%   824.30%

1048576 19458756 19950272 19838588  1048766 19912134  4430649  20434212
                 19606028  1048958  1048941  1694896  5832445   2097150
                    1.76% 1791.27%   -0.02% 1074.83%  -24.03%   874.38%
```

Discussion of cases:

*sort: There's no structure in random data to exploit, so the theoretical limit is lg(n!). Both methods get close to that, and timsort is hugging it (indeed, in a *marginal* sense, it's a spectacular improvement -- there's only about 1% left before hitting the wall, and timsort knows darned well it's doing compares that won't pay on random data -- but so does the samplesort hybrid). For contrast, Hoare's original random-pivot quicksort does about 39% more compares than the limit, and the median-of-3 variant about 19% more.

3sort, %sort, and !sort: No contest; there's structure in this data, but not of the specific kinds samplesort special-cases. Note that structure in !sort wasn't put there on purpose -- it was crafted as a worst case for a previous quicksort implementation. That timsort nails it came as a surprise to me (although it's obvious in retrospect).

+sort: samplesort special-cases this data, and does a few less compares than timsort. However, timsort runs this case significantly faster on all boxes we have timings for, because timsort is in the business of merging runs efficiently, while samplesort does much more data movement in this (for it) special case.

~sort: samplesort's special cases for large masses of equal elements are extremely effective on ~sort's specific data pattern, and timsort just isn't going to get close to that, despite that it's clearly getting a great deal of benefit out of the duplicates (the # of compares is much less than lg(n!)). ~sort has a perfectly uniform distribution of just 4 distinct values, and as the distribution gets more skewed, samplesort's equal-element gimmicks become less effective, while timsort's adaptive strategies find more to exploit; in a database supplied by Kevin Altis, a

```
  sort on its highly skewed "on which stock exchange does this company's
  stock trade?" field ran over twice as fast under timsort.

  However, despite that timsort does many more comparisons on ~sort, and
  that on several platforms ~sort runs highly significantly slower under
  timsort, on other platforms ~sort runs highly significantly faster under
  timsort.  No other kind of data has shown this wild x-platform behavior,
  and we don't have an explanation for it.  The only thing I can think of
  that could transform what "should be" highly significant slowdowns into
  highly significant speedups on some boxes are catastrophic cache effects
  in samplesort.

  But timsort "should be" slower than samplesort on ~sort, so it's hard
  to count that it isn't on some boxes as a strike against it <wink>.

+ Here's the highwater mark for the number of heap-based temp slots (4
  bytes each on this box) needed by each test, again with arguments
  "15 20 1":

   2**i  *sort \sort /sort  3sort  +sort  %sort  ~sort  =sort  !sort
  32768  16384     0     0   6256      0  10821  12288      0  16383
  65536  32766     0     0  21652      0  31276  24576      0  32767
 131072  65534     0     0  17258      0  58112  49152      0  65535
 262144 131072     0     0  35660      0 123561  98304      0 131071
 524288 262142     0     0  31302      0 212057 196608      0 262143
1048576 524286     0     0 312438      0 484942 393216      0 524287

  Discussion:  The tests that end up doing (close to) perfectly balanced
  merges (*sort, !sort) need all N//2 temp slots (or almost all).  ~sort
  also ends up doing balanced merges, but systematically benefits a lot from
  the preliminary pre-merge searches described under "Merge Memory" later.
  %sort approaches having a balanced merge at the end because the random
  selection of elements to replace is expected to produce an out-of-order
  element near the midpoint.  \sort, /sort, =sort are the trivial one-run
  cases, needing no merging at all.  +sort ends up having one very long run
  and one very short, and so gets all the temp space it needs from the small
  temparray member of the MergeState struct (note that the same would be
  true if the new random elements were prefixed to the sorted list instead,
  but not if they appeared "in the middle").  3sort approaches N//3 temp
  slots twice, but the run lengths that remain after 3 random exchanges
  clearly has very high variance.


A detailed description of timsort follows.

Runs
----
count_run() returns the # of elements in the next run.  A run is either
"ascending", which means non-decreasing:

    a0 <= a1 <= a2 <= ...

or "descending", which means strictly decreasing:

    a0 > a1 > a2 > ...
```

Note that a run is always at least 2 long, unless we start at the array's last element.

The definition of descending is strict, because the main routine reverses a descending run in-place, transforming a descending run into an ascending run. Reversal is done via the obvious fast "swap elements starting at each end, and converge at the middle" method, and that can violate stability if the slice contains any equal elements. Using a strict definition of descending ensures that a descending run contains distinct elements.

If an array is random, it's very unlikely we'll see long runs. If a natural run contains less than minrun elements (see next section), the main loop artificially boosts it to minrun elements, via a stable binary insertion sort applied to the right number of array elements following the short natural run. In a random array, *all* runs are likely to be minrun long as a result. This has two primary good effects:

1. Random data strongly tends then toward perfectly balanced (both runs have the same length) merges, which is the most efficient way to proceed when data is random.

2. Because runs are never very short, the rest of the code doesn't make heroic efforts to shave a few cycles off per-merge overheads. For example, reasonable use of function calls is made, rather than trying to inline everything. Since there are no more than N/minrun runs to begin with, a few "extra" function calls per merge is barely measurable.

Computing minrun

If N < 64, minrun is N. IOW, binary insertion sort is used for the whole array then; it's hard to beat that given the overheads of trying something fancier.

When N is a power of 2, testing on random data showed that minrun values of 16, 32, 64 and 128 worked about equally well. At 256 the data-movement cost in binary insertion sort clearly hurt, and at 8 the increase in the number of function calls clearly hurt. Picking *some* power of 2 is important here, so that the merges end up perfectly balanced (see next section). We pick 32 as a good value in the sweet range; picking a value at the low end allows the adaptive gimmicks more opportunity to exploit shorter natural runs.

Because sortperf.py only tries powers of 2, it took a long time to notice that 32 isn't a good choice for the general case! Consider N=2112:

```
>>> divmod(2112, 32)
(66, 0)
>>>
```

If the data is randomly ordered, we're very likely to end up with 66 runs each of length 32. The first 64 of these trigger a sequence of perfectly balanced merges (see next section), leaving runs of lengths 2048 and 64 to merge at the end. The adaptive gimmicks can do that with fewer than 2048+64

compares, but it's still more compares than necessary, and-- mergesort's
bugaboo relative to samplesort --a lot more data movement (O(N) copies just
to get 64 elements into place).

If we take minrun=33 in this case, then we're very likely to end up with 64
runs each of length 33, and then all merges are perfectly balanced. Better!

What we want to avoid is picking minrun such that in

```
    q, r = divmod(N, minrun)
```

q is a power of 2 and r>0 (then the last merge only gets r elements into
place, and r < minrun is small compared to N), or q a little larger than a
power of 2 regardless of r (then we've got a case similar to "2112", again
leaving too little work for the last merge to do).

Instead we pick a minrun in range(32, 65) such that N/minrun is exactly a
power of 2, or if that isn't possible, is close to, but strictly less than,
a power of 2. This is easier to do than it may sound: take the first 6
bits of N, and add 1 if any of the remaining bits are set. In fact, that
rule covers every case in this section, including small N and exact powers
of 2; merge_compute_minrun() is a deceptively simple function.

The Merge Pattern

In order to exploit regularities in the data, we're merging on natural
run lengths, and they can become wildly unbalanced. That's a Good Thing
for this sort! It means we have to find a way to manage an assortment of
potentially very different run lengths, though.

Stability constrains permissible merging patterns. For example, if we have
3 consecutive runs of lengths

```
    A:10000  B:20000  C:10000
```

we dare not merge A with C first, because if A, B and C happen to contain
a common element, it would get out of order wrt its occurrence(s) in B. The
merging must be done as (A+B)+C or A+(B+C) instead.

So merging is always done on two consecutive runs at a time, and in-place,
although this may require some temp memory (more on that later).

When a run is identified, its base address and length are pushed on a stack
in the MergeState struct. merge_collapse() is then called to see whether it
should merge it with preceding run(s). We would like to delay merging as
long as possible in order to exploit patterns that may come up later, but we
like even more to do merging as soon as possible to exploit that the run just
found is still high in the memory hierarchy. We also can't delay merging
"too long" because it consumes memory to remember the runs that are still
unmerged, and the stack has a fixed size.

What turned out to be a good compromise maintains two invariants on the
stack entries, where A, B and C are the lengths of the three righmost not-yet
merged slices:

1. A > B+C
2. B > C

Note that, by induction, #2 implies the lengths of pending runs form a
decreasing sequence. #1 implies that, reading the lengths right to left,
the pending-run lengths grow at least as fast as the Fibonacci numbers.
Therefore the stack can never grow larger than about log_base_phi(N) entries,
where phi = (1+sqrt(5))/2 ~= 1.618. Thus a small # of stack slots suffice
for very large arrays.

If A <= B+C, the smaller of A and C is merged with B (ties favor C, for the
freshness-in-cache reason), and the new run replaces the A,B or B,C entries;
e.g., if the last 3 entries are

```
    A:30  B:20  C:10
```

then B is merged with C, leaving

```
    A:30  BC:30
```

on the stack. Or if they were

```
    A:500  B:400:  C:1000
```

then A is merged with B, leaving

```
    AB:900  C:1000
```

on the stack.

In both examples, the stack configuration after the merge still violates
invariant #2, and merge_collapse() goes on to continue merging runs until
both invariants are satisfied. As an extreme case, suppose we didn't do the
minrun gimmick, and natural runs were of lengths 128, 64, 32, 16, 8, 4, 2,
and 2. Nothing would get merged until the final 2 was seen, and that would
trigger 7 perfectly balanced merges.

The thrust of these rules when they trigger merging is to balance the run
lengths as closely as possible, while keeping a low bound on the number of
runs we have to remember. This is maximally effective for random data,
where all runs are likely to be of (artificially forced) length minrun, and
then we get a sequence of perfectly balanced merges (with, perhaps, some
oddballs at the end).

OTOH, one reason this sort is so good for partly ordered data has to do
with wildly unbalanced run lengths.

Merge Memory

Merging adjacent runs of lengths A and B in-place is very difficult.
Theoretical constructions are known that can do it, but they're too difficult
and slow for practical use. But if we have temp memory equal to min(A, B),
it's easy.

If A is smaller (function merge_lo), copy A to a temp array, leave B alone,
and then we can do the obvious merge algorithm left to right, from the temp
area and B, starting the stores into where A used to live. There's always a
free area in the original area comprising a number of elements equal to the
number not yet merged from the temp array (trivially true at the start;
proceed by induction). The only tricky bit is that if a comparison raises an
exception, we have to remember to copy the remaining elements back in from
the temp area, lest the array end up with duplicate entries from B. But
that's exactly the same thing we need to do if we reach the end of B first,
so the exit code is pleasantly common to both the normal and error cases.

If B is smaller (function merge_hi, which is merge_lo's "mirror image"),
much the same, except that we need to merge right to left, copying B into a
temp array and starting the stores at the right end of where B used to live.

A refinement: When we're about to merge adjacent runs A and B, we first do
a form of binary search (more on that later) to see where B[0] should end up
in A. Elements in A preceding that point are already in their final
positions, effectively shrinking the size of A. Likewise we also search to
see where A[-1] should end up in B, and elements of B after that point can
also be ignored. This cuts the amount of temp memory needed by the same
amount.

These preliminary searches may not pay off, and can be expected *not* to
repay their cost if the data is random. But they can win huge in all of
time, copying, and memory savings when they do pay, so this is one of the
"per-merge overheads" mentioned above that we're happy to endure because
there is at most one very short run. It's generally true in this algorithm
that we're willing to gamble a little to win a lot, even though the net
expectation is negative for random data.

Merge Algorithms

merge_lo() and merge_hi() are where the bulk of the time is spent. merge_lo
deals with runs where A <= B, and merge_hi where A > B. They don't know
whether the data is clustered or uniform, but a lovely thing about merging
is that many kinds of clustering "reveal themselves" by how many times in a
row the winning merge element comes from the same run. We'll only discuss
merge_lo here; merge_hi is exactly analogous.

Merging begins in the usual, obvious way, comparing the first element of A
to the first of B, and moving B[0] to the merge area if it's less than A[0],
else moving A[0] to the merge area. Call that the "one pair at a time"
mode. The only twist here is keeping track of how many times in a row "the
winner" comes from the same run.

If that count reaches MIN_GALLOP, we switch to "galloping mode". Here
we *search* B for where A[0] belongs, and move over all the B's before
that point in one chunk to the merge area, then move A[0] to the merge
area. Then we search A for where B[0] belongs, and similarly move a
slice of A in one chunk. Then back to searching B for where A[0] belongs,
etc. We stay in galloping mode until both searches find slices to copy
less than MIN_GALLOP elements long, at which point we go back to one-pair-

at-a-time mode.

A refinement: The MergeState struct contains the value of min_gallop that controls when we enter galloping mode, initialized to MIN_GALLOP. merge_lo() and merge_hi() adjust this higher when galloping isn't paying off, and lower when it is.

Galloping

Still without loss of generality, assume A is the shorter run. In galloping mode, we first look for A[0] in B. We do this via "galloping", comparing A[0] in turn to B[0], B[1], B[3], B[7], ..., B[2**j - 1], ..., until finding the k such that B[2**(k-1) - 1] < A[0] <= B[2**k - 1]. This takes at most roughly lg(B) comparisons, and, unlike a straight binary search, favors finding the right spot early in B (more on that later).

After finding such a k, the region of uncertainty is reduced to 2**(k-1) - 1 consecutive elements, and a straight binary search requires exactly k-1 additional comparisons to nail it. Then we copy all the B's up to that point in one chunk, and then copy A[0]. Note that no matter where A[0] belongs in B, the combination of galloping + binary search finds it in no more than about 2*lg(B) comparisons.

If we did a straight binary search, we could find it in no more than ceiling(lg(B+1)) comparisons -- but straight binary search takes that many comparisons no matter where A[0] belongs. Straight binary search thus loses to galloping unless the run is quite long, and we simply can't guess whether it is in advance.

If data is random and runs have the same length, A[0] belongs at B[0] half the time, at B[1] a quarter of the time, and so on: a consecutive winning sub-run in B of length k occurs with probability 1/2**(k+1). So long winning sub-runs are extremely unlikely in random data, and guessing that a winning sub-run is going to be long is a dangerous game.

OTOH, if data is lopsided or lumpy or contains many duplicates, long stretches of winning sub-runs are very likely, and cutting the number of comparisons needed to find one from O(B) to O(log B) is a huge win.

Galloping compromises by getting out fast if there isn't a long winning sub-run, yet finding such very efficiently when they exist.

I first learned about the galloping strategy in a related context; see:

    "Adaptive Set Intersections, Unions, and Differences" (2000)
    Erik D. Demaine, Alejandro López-Ortiz, J. Ian Munro

and its followup(s). An earlier paper called the same strategy "exponential search":

   "Optimistic Sorting and Information Theoretic Complexity"
   Peter McIlroy
   SODA (Fourth Annual ACM-SIAM Symposium on Discrete Algorithms), pp
   467-474, Austin, Texas, 25-27 January 1993.

and it probably dates back to an earlier paper by Bentley and Yao. The
McIlroy paper in particular has good analysis of a mergesort that's
probably strongly related to this one in its galloping strategy.

Galloping with a Broken Leg

So why don't we always gallop? Because it can lose, on two counts:

1. While we're willing to endure small per-merge overheads, per-comparison
   overheads are a different story. Calling Yet Another Function per
   comparison is expensive, and gallop_left() and gallop_right() are
   too long-winded for sane inlining.

2. Galloping can-- alas --require more comparisons than linear one-at-time
   search, depending on the data.

#2 requires details. If A[0] belongs before B[0], galloping requires 1
compare to determine that, same as linear search, except it costs more
to call the gallop function. If A[0] belongs right before B[1], galloping
requires 2 compares, again same as linear search. On the third compare,
galloping checks A[0] against B[3], and if it's <=, requires one more
compare to determine whether A[0] belongs at B[2] or B[3]. That's a total
of 4 compares, but if A[0] does belong at B[2], linear search would have
discovered that in only 3 compares, and that's a huge loss! Really. It's
an increase of 33% in the number of compares needed, and comparisons are
expensive in Python.

index in B where A[0] belongs	# compares linear search needs	# gallop compares	# binary compares	gallop total
0	1	1	0	1
1	2	2	0	2
2	3	3	1	4
3	4	3	1	4
4	5	4	2	6
5	6	4	2	6
6	7	4	2	6
7	8	4	2	6
8	9	5	3	8
9	10	5	3	8
10	11	5	3	8
11	12	5	3	8

...

In general, if A[0] belongs at B[i], linear search requires i+1 comparisons
to determine that, and galloping a total of 2*floor(lg(i))+2 comparisons.
The advantage of galloping is unbounded as i grows, but it doesn't win at
all until i=6. Before then, it loses twice (at i=2 and i=4), and ties
at the other values. At and after i=6, galloping always wins.

We can't guess in advance when it's going to win, though, so we do one pair at a time until the evidence seems strong that galloping may pay. MIN_GALLOP is 7, and that's pretty strong evidence. However, if the data is random, it simply will trigger galloping mode purely by luck every now and again, and it's quite likely to hit one of the losing cases next. On the other hand, in cases like ~sort, galloping always pays, and MIN_GALLOP is larger than it "should be" then. So the MergeState struct keeps a min_gallop variable that merge_lo and merge_hi adjust: the longer we stay in galloping mode, the smaller min_gallop gets, making it easier to transition back to galloping mode (if we ever leave it in the current merge, and at the start of the next merge). But whenever the gallop loop doesn't pay, min_gallop is increased by one, making it harder to transition back to galloping mode (and again both within a merge and across merges). For random data, this all but eliminates the gallop penalty: min_gallop grows large enough that we almost never get into galloping mode. And for cases like ~sort, min_gallop can fall to as low as 1. This seems to work well, but in all it's a minor improvement over using a fixed MIN_GALLOP value.

Galloping Complication

The description above was for merge_lo. merge_hi has to merge "from the other end", and really needs to gallop starting at the last element in a run instead of the first. Galloping from the first still works, but does more comparisons than it should (this is significant -- I timed it both ways). For this reason, the gallop_left() and gallop_right() functions have a "hint" argument, which is the index at which galloping should begin. So galloping can actually start at any index, and proceed at offsets of 1, 3, 7, 15, ... or -1, -3, -7, -15, ... from the starting index.

In the code as I type it's always called with either 0 or n-1 (where n is the # of elements in a run). It's tempting to try to do something fancier, melding galloping with some form of interpolation search; for example, if we're merging a run of length 1 with a run of length 10000, index 5000 is probably a better guess at the final result than either 0 or 9999. But it's unclear how to generalize that intuition usefully, and merging of wildly unbalanced runs already enjoys excellent performance.

~sort is a good example of when balanced runs could benefit from a better hint value: to the extent possible, this would like to use a starting offset equal to the previous value of acount/bcount. Doing so saves about 10% of the compares in ~sort. However, doing so is also a mixed bag, hurting other cases.

Comparing Average # of Compares on Random Arrays

[NOTE: This was done when the new algorithm used about 0.1% more compares
 on random data than does its current incarnation.]

Here list.sort() is samplesort, and list.msort() this sort:

```
"""
import random
```

```
from time import clock as now

def fill(n):
    from random import random
    return [random() for i in xrange(n)]

def mycmp(x, y):
    global ncmp
    ncmp += 1
    return cmp(x, y)

def timeit(values, method):
    global ncmp
    X = values[:]
    bound = getattr(X, method)
    ncmp = 0
    t1 = now()
    bound(mycmp)
    t2 = now()
    return t2-t1, ncmp

format = "%5s  %9.2f  %11d"
f2     = "%5s  %9.2f  %11.2f"

def drive():
    count = sst = sscmp = mst = mscmp = nelts = 0
    while True:
        n = random.randrange(100000)
        nelts += n
        x = fill(n)

        t, c = timeit(x, 'sort')
        sst += t
        sscmp += c

        t, c = timeit(x, 'msort')
        mst += t
        mscmp += c

        count += 1
        if count % 10:
            continue

        print "count", count, "nelts", nelts
        print format % ("sort",  sst, sscmp)
        print format % ("msort", mst, mscmp)
        print f2     % ("", (sst-mst)*1e2/mst, (sscmp-mscmp)*1e2/mscmp)

drive()
"""
```

I ran this on Windows and kept using the computer lightly while it was running. time.clock() is wall-clock time on Windows, with better than microsecond resolution. samplesort started with a 1.52% #-of-comparisons disadvantage, fell quickly to 1.48%, and then fluctuated within that small range. Here's the last chunk of output before I killed the job:

```
count 2630 nelts 130906543
 sort    6110.80   1937887573
msort    6002.78   1909389381
            1.80         1.49
```

We've done nearly 2 billion comparisons apiece at Python speed there, and that's enough <wink>.

For random arrays of size 2 (yes, there are only 2 interesting ones), samplesort has a 50%(!) comparison disadvantage. This is a consequence of samplesort special-casing at most one ascending run at the start, then falling back to the general case if it doesn't find an ascending run immediately. The consequence is that it ends up using two compares to sort [2, 1]. Gratifyingly, timsort doesn't do any special-casing, so had to be taught how to deal with mixtures of ascending and descending runs efficiently in all cases.