Adding Data-Aware Sort Optimizations to CPython

"Comparisons are extraordinarily expensive in Python, precisely because of the maze of test-and-branch code it requires just to figure out which bottom-level comparison function to invoke each time. That's why I spent months of my life (overall) devising a sequence of sorting algorithms for Python that reduced the number of comparisons needed."

— Tim Peters, lead Python developer and author of the Timsort algorithm [5]

Introduction

Sorting is ubiquitous in computer science: it is used as a subroutine in many algorithms, and sort functions are included in the standard library of every modern programming language. It is not surprising, then, that sorting algorithms and implementations are an area of active research. However, nearly all sorting research operates under the assumption that comparisons have a unit cost: all that matters is the number of comparisons made. Python's sort is especially characteristic of this trend: the built-in sort algorithm, Timsort, employs sophisticated optimizations based on assumptions about the distribution of real-world data to try to reduce the number of comparisons necessary to sort a list, taking up nearly half the code in `Objects/listobject.c`.

The unit-cost comparison assumption is reasonable in low-level models of computation, where comparisons only cost a few machine operations, and issues such as cache utilization are much more important than the cost of individual comparisons. However, in dynamic languages like Python, comparisons are extremely expensive, which makes the unit cost assumption tenuous. In this project, it is demonstrated that it is possible to *vastly* increase the speed of Python's sort routine, not by reducing the number of comparisons made, but by reducing the cost of the individual comparisons, by exploiting the homogeneous structure of real-world data.

The dynamic dispatch problem

Python is a dynamic language. This means that functions must check the types of their arguments each time they are called, and execute different implementations for different argument types. The procedure by which the interpreter selects the correct implementation of a polymorphic function given arbitrary argument types is called *dynamic dispatch* [3].

The comparison operator in Python, like every other operator, is polymorphic, and is thus subject to the cost of dynamic dispatch each time it is applied. In general, this is unavoidable, and part of the inherent cost for which one gains the many benefits a dynamic language offers. However, in the common situation where one can show (at runtime) that the objects being compared will have the same type across function applications, having to pay the needless cost of dynamic dispatch is unfortunate, and one wonders whether it can be avoided altogether.

Inline caching

One way in which dynamic language implementations, specifically Just-in-Time (JIT) compilers, have sought to avoid dynamic dispatch is called *inline caching* [2]. The observation is the following: in practice, the input types at any given polymorphic call site will often remain constant across calls. As long as the homogeneity assumption holds, directly calling a cached implementation is then much faster than looking it up every time. This is a gamble: if the assumption fails, time was spent caching for no benefit, yielding a net loss. However, because the assumption nearly always holds, the average-case savings dwarf the overhead. *The success of inline caching demonstrates that by making a homogeneity assumption, and applying that assumption to run-time observations, it is possible avoid a great deal of the overhead of dynamic dispatch.*

The list homogeneity assumption

It is thus clear that to avoid the overhead of dynamic dispatch, we have to make assumptions about the lists we're going to be sorting. We will operate under a weak assumption: that lists are homogeneous with respect to type. This assumption is motivated by two considerations:

- Lists are idiomatically homogeneous data structures. To quote the Python documentation (emphasis added) [1]:
  > *"Lists are mutable, and* ***their elements are usually homogeneous*** *and are accessed by iterating over the list."*
- It is difficult to define comparison in the heterogeneous context. While some types are comparable to each other, like `long`s and `float`s, and of course subtypes of the same type are often comparable, such pairs are rare in practice: it's "like comparing apples and oranges". Truly heterogeneous lists simply don't have well-defined orderings.

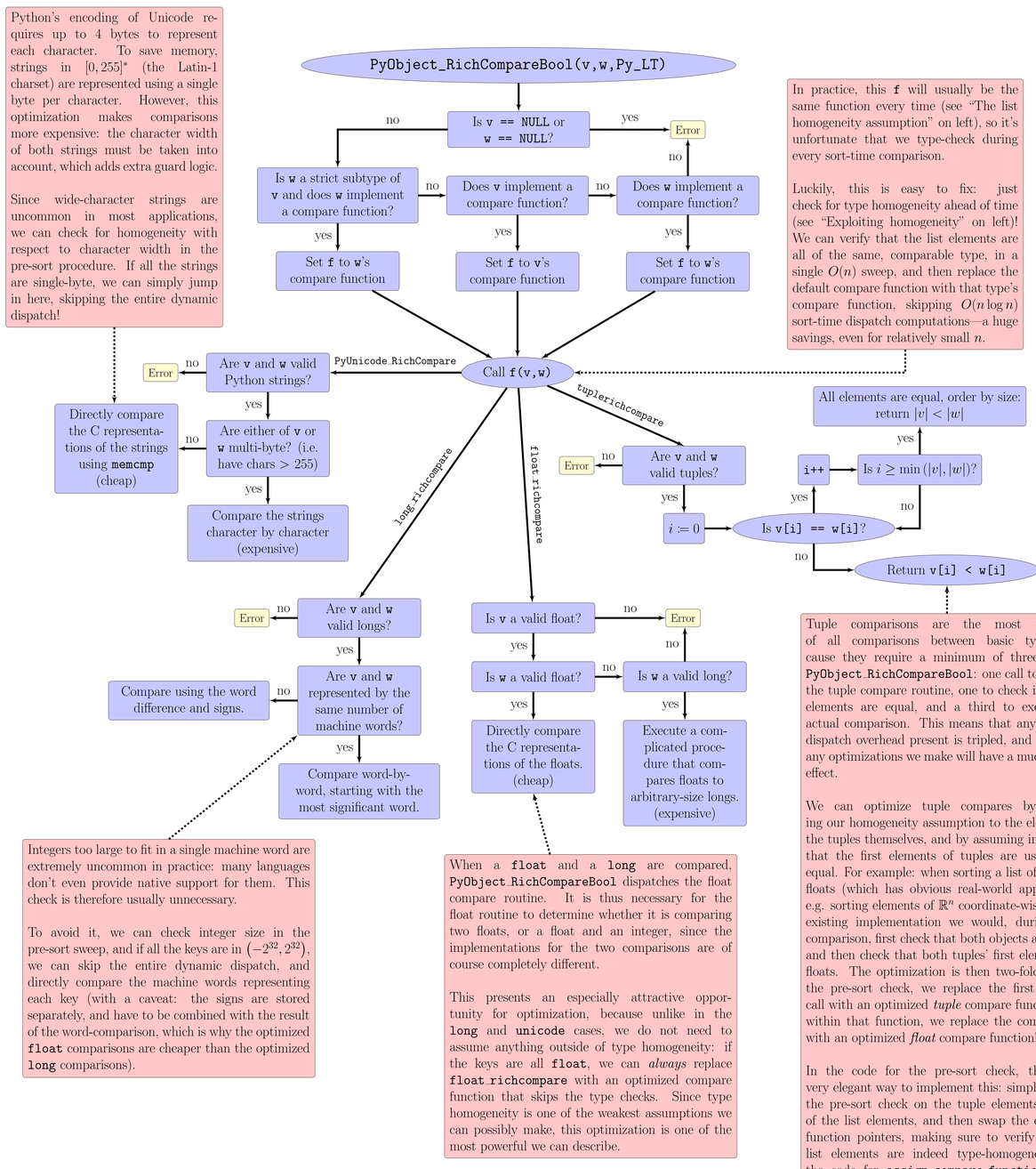
Exploiting homogeneity

The list homogeneity assumption is powerful, because it allows us to justify a pre-sort check: instead of checking types and other properties every time we compare, and then executing the appropriate compare function, we can check *once* for each key, and select the compare function ahead of time! Of course, this is only possible if the keys are homogeneous with respect to whatever properties determine their compare functions, but that is *precisely* our assumption. We can simply replace calls to `PyObject_RichCompareBool()` with calls to a function pointer, which can be set to point to whichever (potentially optimized) compare implementation our pre-sort check indicates is appropriate. This turns $O(n \log n)$ sort-time checks into $O(n)$ pre-sort checks.

There are, of course, many specific properties we could check for, since each built-in type has its own special considerations. For example, strings can have varying character widths, but it's reasonable to assume lists of strings are homogeneous with respect to character width (since most applications use ASCII), so that check can be moved out of the sort loop. The optimizations used in this project are detailed in the red boxes in the flow chart to the right.

Exploring the maze

The flow chart below presents a simplified version of the "maze" of dynamic dispatch logic necessary to compare two Python objects. The blue and yellow boxes correspond to blocks of code in the CPython implementation. The red boxes point out places where type- or other homogeneity found in practice can be exploited to avoid a great deal of that logic during the sort.



Python's encoding of Unicode requires up to 4 bytes to represent each character. To save memory, strings in $[0, 255]^*$ (the Latin-1 charset) are represented using a single byte per character. However, this optimization makes comparisons more expensive: the character width of both strings must be taken into account, which adds extra guard logic.

Since wide-character strings are uncommon in most applications, we can check for homogeneity with respect to character width in the pre-sort procedure. If all the strings are single-byte, we can simply jump in here, skipping the entire dynamic dispatch!

In practice, this `f` will usually be the same function every time (see "The list homogeneity assumption" on left), so it's unfortunate that we type-check during every sort-time comparison.

Luckily, this is easy to fix: just check for type homogeneity ahead of time (see "Exploiting homogeneity" on left)! We can verify that the list elements are all of the same, comparable type, in a single $O(n)$ sweep, and then replace the default compare function with that type's compare function, skipping $O(n \log n)$ sort-time dispatch computations—a huge savings, even for relatively small n.

Integers too large to fit in a single machine word are extremely uncommon in practice: many languages don't even provide native support for them. This check is therefore usually unnecessary.

To avoid it, we can check integer size in the pre-sort sweep, and if all the keys are in $(-2^{32}, 2^{32})$, we can skip the entire dynamic dispatch, and directly compare the machine words representing each key (with a caveat: the signs are stored separately, and have to be combined with the result of the word-comparison, which is why the optimized `float` comparisons are cheaper than the optimized `long` comparisons).

When a `float` and a `long` are compared, `PyObject_RichCompareBool` dispatches the float compare routine. It is thus necessary for the float routine to determine whether it is comparing two floats, or a float and an integer, since the implementations for the two comparisons are of course completely different.

This presents an especially attractive opportunity for optimization, because unlike in the `long` and `unicode` cases, we do not need to assume anything outside of type homogeneity: if the keys are all `float`, we can *always* replace `float_richcompare` with an optimized compare function that skips the type checks. Since type homogeneity is one of the weakest assumptions we can possibly make, this optimization is one of the most powerful we can describe.

Tuple comparisons are the most expensive of all comparisons between basic types, because they require a minimum of three calls to `PyObject_RichCompareBool`: one call to dispatch the tuple compare routine, one to check if the first elements are equal, and a third to execute the actual comparison. This means that any dynamic dispatch overhead present is tripled, and thus that any optimizations we make will have a much greater effect.

We can optimize tuple compares by extending our homogeneity assumption to the elements of the tuples themselves, and by assuming in addition that the first elements of tuples are usually not equal. For example: when sorting a list of tuples of floats (which has obvious real-world applications, e.g. sorting elements of $\mathbb{R}^n$ coordinate-wise), in the existing implementation we would, during every comparison, first check that both objects are tuples, and then check that both tuples' first elements are floats. The optimization is then two-fold: during the pre-sort check, we replace the first compare call with an optimized *tuple* compare function, and within that function, we replace the compare call with an optimized *float* compare function!

In the code for the pre-sort check, there is a very elegant way to implement this: simply execute the pre-sort check on the tuple elements, instead of the list elements, and then swap the computed function pointers, making sure to verify that the list elements are indeed type-homogeneous (see the code for `assign_compare_function` in the supplement).

Benchmarking methodology

Benchmarks were performed according to recommendations in [7] and [4]:

- Build Python with `./configure --enable-optimizations`
- Isolate CPU core with `isolcpus=cpu_list` and `rcu_nocbs=cpu_list`
- Disable frequency scaling with `python -m perf system tune`
- Serialize with `CPUID` and time with `RDTSCP`

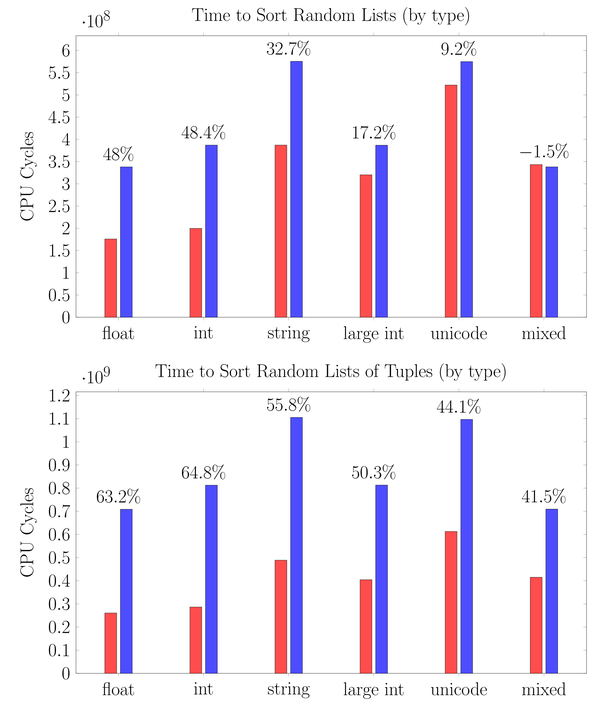
12 different benchmarks were run: sorting random lists of six different types, as well as random lists of tuples of those types. The benchmark lists were constructed using the comprehensions below:

- float: `[random() for _ in range(0,n)]`
- int: `[int((1<<31)*random() - (1<<30)) for _ in range(0,n)]`
- large int: `int_list(n) + [1<<64]`
- string: `[str(random()) for _ in range(0,n)]`
- unicode: `string_list(n) + ["\uffff"]`
- mixed (float + int): `float_list(n) + [0]`
- tuples of `type`: `[(x,) for x in type_list(n)]`

The complete code for the benchmarks can be found in the supplement.

Results

The graphs below give the times it took to run each of the benchmarks on the current CPython codebase (blue) and on the patched codebase (red). Numbers above the bars are percent differences. Confidence intervals could not be shown on the graphs because standard deviation over 1000 iterations was less than 0.3% of the mean for all the benchmarks.



Note that in all the cases commonly encountered in practice (float, int, and string), the improvements are greater than 30%, which compares favorably to numbers commonly reported in the literature, e.g. [6]. Further, note that in the uncommon worst-case (mixed), where the entire pre-sort check is executed without any ability to perform optimization (because the homogeneity assumption is violated), the cost of overhead is less than 1.5%, an order of magnitude smaller than the gain for which we make this near-certain gamble.

Conclusion

In this project, it was shown that by focusing on reducing the cost of the comparison operation, instead of trying to reduce the total number of comparisons, and by making weak assumptions about the distribution of real-world data, it is possible to dramatically reduce the cost of CPython's sort routine, by bypassing nearly all the overhead due to needless dynamic dispatch.

References

[1] *Data Structures.* `docs.python.org/3.3/tutorial/datastructures.html`. Python manual page.

[2] L. Peter Deutsch and Allan M. Schiffman. "Efficient Implementation of the Smalltalk-80 System". In: *Proceedings of the 11th ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages.* POPL '84. New York, NY, USA: ACM, 1984, pp. 297–302.

[3] Scott Milton and Heinz W. Schmidt. *Dynamic Dispatch in Object-Oriented Languages.* Tech. rep. CSIRO – Division of Information Technology, 1994.

[4] Gabriele Paoloni. *How to Benchmark Code Execution Times on Intel IA-32 and IA-64 Instruction Set Architectures.* Tech. rep. Intel, 2010.

[5] Tim Peters. Post on the `python-dev` mailing list. URL: `mail.python.org/pipermail/python-dev/2016-October/146648.html`.

[6] Tim Peters. *An Adaptive, Stable, Natural Mergesort.* Included with the CPython source at `Objects/listsort.txt`. 2002.

[7] Victor Stinner. *My journey to a stable benchmark.* `haypo-notes.readthedocs.io/microbenchmark.html`. 2016.

All graphs and figures by student researcher.