

Lab 11: More on sorting

You may have wondered how the sorting algorithms we've seen in this course compare to Python's built-in `list.sort`. It turns out that they're quite a bit worse on average, but based off similar principles!

Python's `list.sort` uses an algorithm called **timsort**, and yes, it's named after someone named Tim. Tim Peters developed the algorithm in 2002. In this 'lab', we'll be looking at timsort.

Please note that this lab is **not** graded, and is just provided as extra material that you may be interested in. We encourage you to try this lab regardless, as it's excellent practice! If you have questions, feel free to ask on Piazza.

If you get stuck, [we've also provided a "solution" for the lab](https://q.utoronto.ca/courses/204422/pages/lab-11-solution) (<https://q.utoronto.ca/courses/204422/pages/lab-11-solution>). While this does not contain any code, it explains *how* to do the lab.



Learning objectives

By the end of this lab, students will be able to

- Implement a basic version of timsort

This lab is fairly technical, and is a nice programming exercise that requires careful attention to detail—which is good preparation for future courses.

Setup

Download [timsort.py](https://q.utoronto.ca/courses/204422/files/13649016?wrap=1) (<https://q.utoronto.ca/courses/204422/files/13649016?wrap=1>) 
(https://q.utoronto.ca/courses/204422/files/13649016/download?download_frd=1) and [quiz11.py](https://q.utoronto.ca/courses/204422/files/13649013?wrap=1)
(<https://q.utoronto.ca/courses/204422/files/13649013?wrap=1>) 
(https://q.utoronto.ca/courses/204422/files/13649013/download?download_frd=1) into your `labs/lab11` folder.

While the quiz is **not** graded, we've provided it to give you more practice.

Preview: introducing timsort

Fundamentally, timsort is a highly optimized version of mergesort.

Recall the basic idea of mergesort: divide up the list into two pieces, recursively sort each piece, then *merge* the sorted pieces together.

Open `timsort.py` and take a few minutes with your partner to go over the provided `mergesort2` and `_merge` functions. Make sure you understand these before moving on! Note that these functions are a little bit different than the version you just worked on: They do not return a new list, but instead mutate the input list. (However, `_merge` does create a temporary list, as the list concatenation in the last line creates a new list before storing it in `lst[start:end]`.)

Task 1: Finding runs

Standard mergesort is oblivious to the actual data; like selection sort, it behaves the same regardless of whether the data is already sorted or not.

Intuitively, however, if the data is already partially sorted, then mergesort should be able to make use of that: any time it reaches an already sorted piece, it should immediately just halt rather than making new recursive calls.

Note that it takes linear time to detect whether a slice of the list is sorted, which is much more efficient than running mergesort on that slice to make it sorted.

This is the biggest idea in timsort: *look for, and take advantage of, partial sorting in the input data*. In this task, you'll write a function which does the first part: looking for partial sorting.

A **run** in a list is defined as a maximal list slice that's already sorted. For example, the list `[1, 4, 7, 10, 2, 5, 3, -1]` has *four* runs: `[1, 4, 7, 10]`, `[2, 5]`, `[3]`, and `[-1]`. **Duplicates may occur in a run.** Note that the slice `[4, 7, 10]` is *not* a run, because even though it's sorted, it isn't maximal: we could add the initial 1 to it and still have a sorted slice.

In the starter code, implement `find_runs`, which is given a list and outputs an ordered list of tuples, where each tuple represents the "slice indexes" of the runs in the list. For example, if we gave this Python function the list `[1, 4, 7, 10, 2, 5, 3, -1]`, the runs are `[0:4]`, `[4:6]`, `[6:7]`, and `[7:8]`, so the function would return `[(0, 4), (4, 6), (6, 7), (7, 8)]`.

We've provided some hints on how to do this in the starter code in the loop; but keep in mind this can be done by processing each element in the list just once. This already is not easy, so spend some time planning a strategy on paper! If you get stuck, ask questions on Piazza!

Task 2: Merging runs

Now that we have the runs, we can **merge them**, one pair at a time, and this will result in a sorted list! This is done in timsort by treating the list of run indexes returned by `find_runs` as a stack:

```
runs = find_runs(lst)

while <runs> has more than one tuple:
    pop the top two tuples off the stack <runs>
```

```
merge the runs in <lst> that these two tuples describe, producing a new run
push the new run onto <runs>
```

Implement this algorithm in the `timsort` function.

For the sake of speed, rather than using a `Stack` class, use a Python list directly and make the end of the list be the top of the stack. The `pop` and `append` list methods will provide the services you need.

After you complete this task, you should have a fully functional sorting algorithm in `timsort`. Your algorithm is a long way from the fully-optimized timsort, and in fact almost certainly slower than `mergesort` at this point.

Let's keep going!

Task 3: Descending runs

Consider the reverse-sorted list `[5, 4, 3, 2, 1]`.

If we gave this to `find_runs`, we'd get `[(0, 1), (1, 2), (2, 3), (3, 4), (4, 5)]`, which is the longest, and worst, list of runs we could possibly get.

But it seems funny that a reverse-sorted list is a “bad” input, given that it has just as much structure as a sorted list. So here's the next idea: we can detect *descending runs* at the same time as detecting *ascending runs* – in the same loop!

Complete `find_runs2`, which should be the same as `find_runs`, except in addition to finding a ascending runs, it finds descending runs.

It decides whether any run is ascending or descending simply by comparing the first two numbers. If they go up, it's working on an ascending run and if they go down it's working on a descending run. If they are equal, you can decide which way to go.

See the doctests for some examples.

After you're confident in your solution, modify your code so that after a descending run is found, but before it is added to the `runs` variable, you **reverse the run in-place**, so that it becomes an ascending run. Note that this might result in two adjacent runs that should be merged, but that's okay.

Task 4: Minimum run length

It turns out that short runs really aren't that fast to work with, and it's actually more efficient to artificially **create** larger runs by doing some manual sorting.

Modify `find_runs2` into `find_runs3` which does the following:

- In the loop, after it has found an ascending *or descending* run (but before reversing the descending run), if the run has length **less than 64**, sort it and the next few elements after it using insertion sort, to create an ascending run of **length 64**. Then add that run to the stack.

We've provided an insertion sort algorithm for you, but it's up to you to read its docstring to determine how to use it correctly.

Task 5: Optimizing merge

The provided `_merge` algorithm creates a new list that reaches the same size as the original list, and then copies the contents back over to the original.

Let's optimize that a little bit by making the following observation: suppose we want to merge two consecutive runs A and B in the list. We can copy A to a temporary storage, and then just start filling in entries where A started out being (running the standard merge algorithm on B and the copy of A).

This works because the number of “unused” spaces in the list is equal to the number of entries of A still remaining in the temporary storage.

Modify `_merge` into `_merge2` to make use of this observation.

Task 6: Limiting the ‘runs’ stack (harder!)

Even with the optimizations done in Tasks 3 and 4, it's quite possible for the number of runs to grow *linearly* with the size of the list. If we store all of the runs, this can require a large amount of memory if the input is very large.

So rather than finding *all* runs, and then doing *all* of the merges, timsort actually interleaves these two actions to keep the stack small.

Specifically, it performs the following check and actions **each time a run is pushed onto the stack, and the stack has size ≥ 3** :

- Let A, B, and C be the lengths of the three top-most runs on the stack (so C is the length of the rightmost run found so far). Check these two properties:
 1. $B > C$
 2. $A > B + C$
- If these two properties hold, keep going and add a new run to the stack.
- If these properties don't hold, do a merge instead:
 - If $B \leq C$, merge B and C.
 - If $A \leq B + C$, merge the *shorter* of A and C with B.

Continue merging until the properties hold (or until there's only one run on the stack).

Spend your remaining lab time in your lab designing and implementing `timsort2` to use this approach.

It's a really nice exercise in design to think about how you would need to modify your work in the previous parts to make them work here, so we strongly encourage you to talk with your partner and your TA before writing any code at all!

Further Reading

You'll find that even if you completed every part of this lab, your `timsort2` algorithm is still far slower than the built-in `list.sort`.

If you're interested, the full timsort algorithm is covered in-depth by its creator Tim Peters [over here](https://github.com/python/cpython/blob/master/Objects/listsort.txt) [_\(https://github.com/python/cpython/blob/master/Objects/listsort.txt\)_](https://github.com/python/cpython/blob/master/Objects/listsort.txt).