Курс “Алгоритмы на python”

Занятие #6
Сортировки

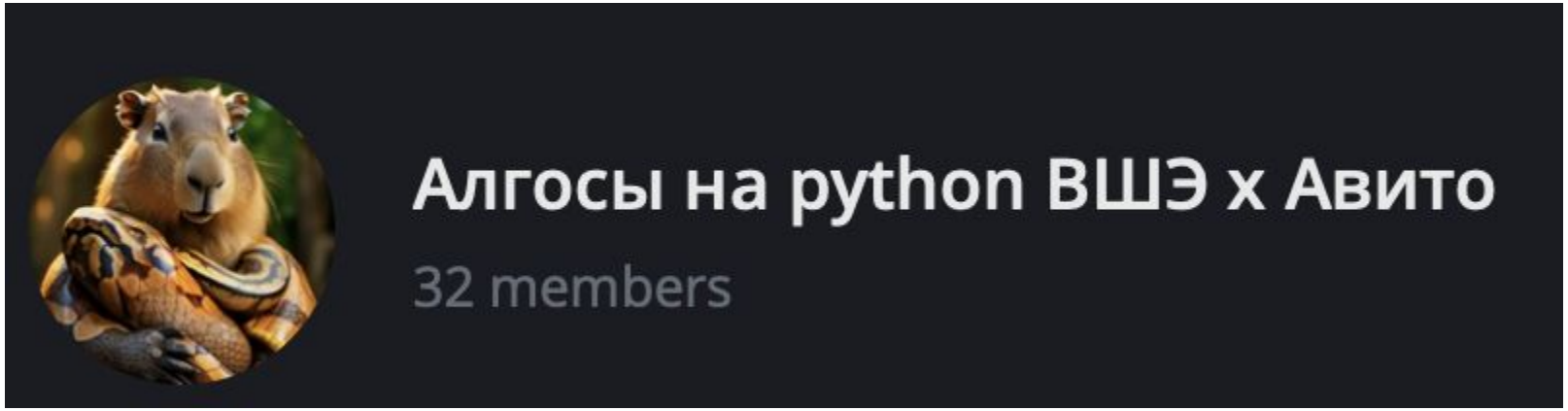
Сентябрь 2025

Avito

Единая точка входа/выхода – степик

- https://stepik.org/course/251189/

Вопросы и обсуждения – чат



Посещаемость

Орг моменты

Дедлайны

2 дз
* без штрафов — 3 октября включительно
* минус балл — 17 октября включительно

3 дз:
* без штрафов 12 октября включительно
* минус балл — 26 октября включительно

4 дз
* без штрафов 20 октября включительно
* минус балл — 3 ноября включительно

5 дз
* без штрафов 29 октября включительно
* минус балл — 12 ноября включительно

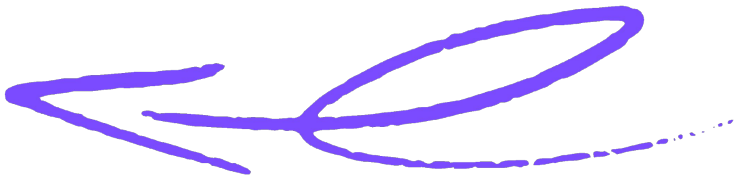
Структура курса «Алгоритмы на питоне»

1 модуль

- Введение в алгоритмы
- Базовые структуры данных
- Хеш-таблицы
- Бинарные деревья поиска
- Рекурсия
- Сортировки
- Кучи

2 модуль

- Графы
- Динамическое программирование
- Алгоритмы в строках
- Алгоритмы в ML и LLM
- Итоговый контест



План занятия

- Часть I. Bubble, insertion, selection
- Часть II. Merge, quick
- Часть III. Timsort
- Часть IV. Powersort

Сортировки

Сортировки

Algorithm	Time Complexity			Space Complexity
	Best	Average	Worst	Worst
Quicksort	Ω(n log(n))	θ(n log(n))	O(n^2)	O(log(n))
Mergesort	Ω(n log(n))	θ(n log(n))	O(n log(n))	O(n)
Timsort	Ω(n)	θ(n log(n))	O(n log(n))	O(n)
Heapsort	Ω(n log(n))	θ(n log(n))	O(n log(n))	O(1)
Bubble Sort	Ω(n)	θ(n^2)	O(n^2)	O(1)
Insertion Sort	Ω(n)	θ(n^2)	O(n^2)	O(1)
Selection Sort	Ω(n^2)	θ(n^2)	O(n^2)	O(1)
Tree Sort	Ω(n log(n))	θ(n log(n))	O(n^2)	O(n)
Shell Sort	Ω(n log(n))	θ(n(log(n))^2)	O(n(log(n))^2)	O(1)
Bucket Sort	Ω(n+k)	θ(n+k)	O(n^2)	O(n)
Radix Sort	Ω(nk)	θ(nk)	O(nk)	O(n+k)
Counting Sort	Ω(n+k)	θ(n+k)	O(n+k)	O(k)
Cubesort	Ω(n)	θ(n log(n))	O(n log(n))	O(n)

Банальные сортировки

bubble sort
selection
insertion

Чуть более интересные сортировки

mergesort
quicksort
timsort

Bubble sort

DESCRIPTION

Bubble Sort is an iterative sorting algorithm that imitates the movement of bubbles in sparkling water. The bubbles represents the elements of the data structure.

The bigger bubbles reach the top faster than smaller bubbles, and this algorithm works in the same way. It iterates through the data structure and for each cycle compares the current element with the next one, swapping them if they are in the wrong order.

It's a simple algorithm to implement, but not much efficient: on average, quadratic sorting algorithms with the same time complexity such as Selection Sort or Insertion Sort perform better.
It has several variants to improve its performances, such as Shaker Sort, Odd Even Sort and Comb Sort.

COMPLEXITY

Average Complexity	$O(n^2)$
Best Case	$O(n)$
Worst Case	$O(n^2)$
Space Complexity	$O(1)$

Insertion sort

DESCRIPTION

Insertion sort is a simple sorting algorithm that builds the final sorted array one item at a time. It's less performant than advanced sorting algorithms, but it can still have some advantages: it's really easy to implement and it's efficient on small data structures almost sorted.

The algorithm divides the data structure in two sublists: a sorted one, and one still to sort. Initially, the sorted sublist is made up of just one element and it gets progressively filled. For every iteration, the algorithm picks an element on the unsorted sublist and inserts it at the right place in the sorted sublist. It's available in several variants such as Gnome Sort.

COMPLEXITY

Average Complexity	$O(n^2)$
Best Case	$O(n)$
Worst Case	$O(n^2)$
Space Complexity	$O(1)$

Selection sort

DESCRIPTION

Selection Sort is an iterative and in-place sorting algorithm that divides the data structure in two sublists: the ordered one, and the unordered one. The algorithm loops for all the elements of the data structure and for every cycle picks the smallest element of the unordered sublist and adds it to the sorted sublist, progressively filling it.

It's a really simple and intuitive algorithm that does not require additional memory, but it's not really efficient on big data structures due to its quadratic time complexity.

This algorithm has been upgraded and enhanced in several variants such as Heap Sort.

COMPLEXITY

Average Complexity	$O(n^2)$
Best Case	$O(n^2)$
Worst Case	$O(n^2)$
Space Complexity	O(1)

Selection sort

	8
	5
	2
	6
	9
	3
	1
	4
	0
	7

Quicksort

DESCRIPTION

Quick Sort is a sorting algorithm based on splitting the data structure in smaller partitions and sort them recursively until the data structure is sorted.

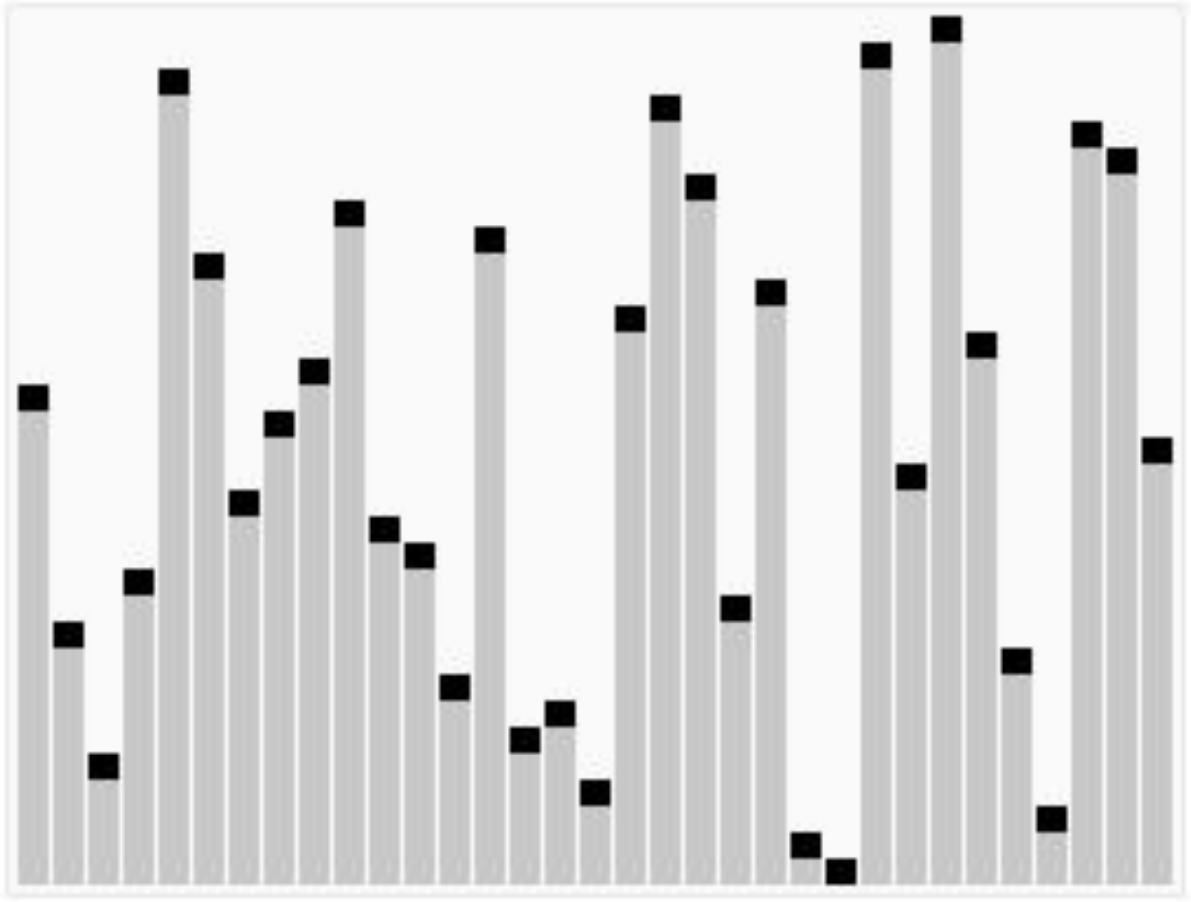
This division in partitions is done based on an element, called pivot: all the elements bigger than the pivot get placed on the right side of the structure, the smaller ones to the left, creating two partitions. Next, this procedure gets applied recursively to the two partitions and so on.

This partition technique based on the pivot is called Divide and conquer. It's a performant strategy also used by other sorting algorithms, such as Merge Sort.

COMPLEXITY

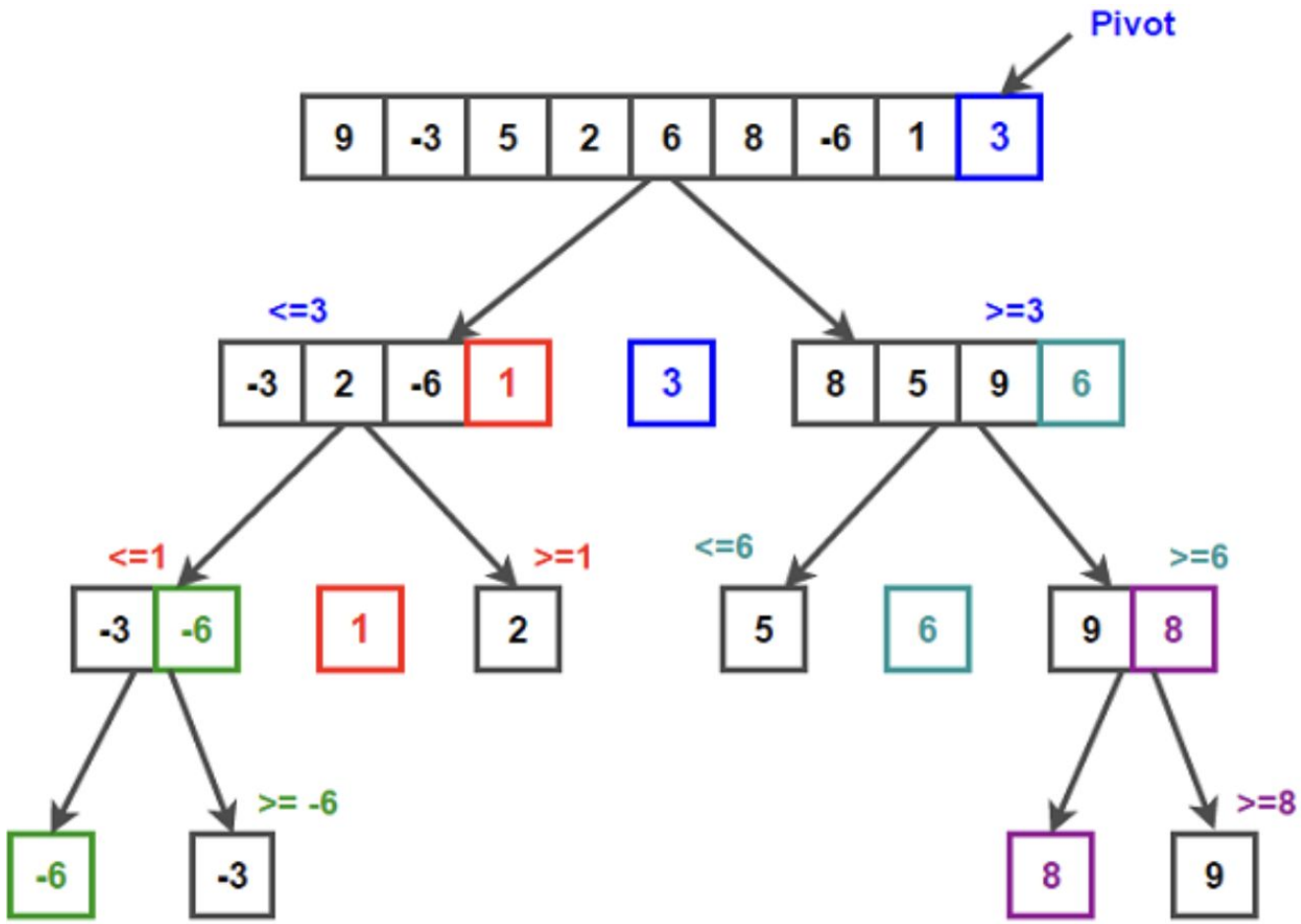
Average Complexity	$O(n \times \log n)$
Best Case	$O(n \times \log n)$
Worst Case	$O(n^2)$
Space Complexity	$O(n)$

Quicksort



1) выбираем pivot
2) другие элементы в массиве перераспределяем так, чтобы элементы меньше опорного оказались до него, а большие или равные — после
3) рекурсивно применяем первые два шага к подмассивам справа и слева от опорного значения.

Quicksort



Mergesort

DESCRIPTION

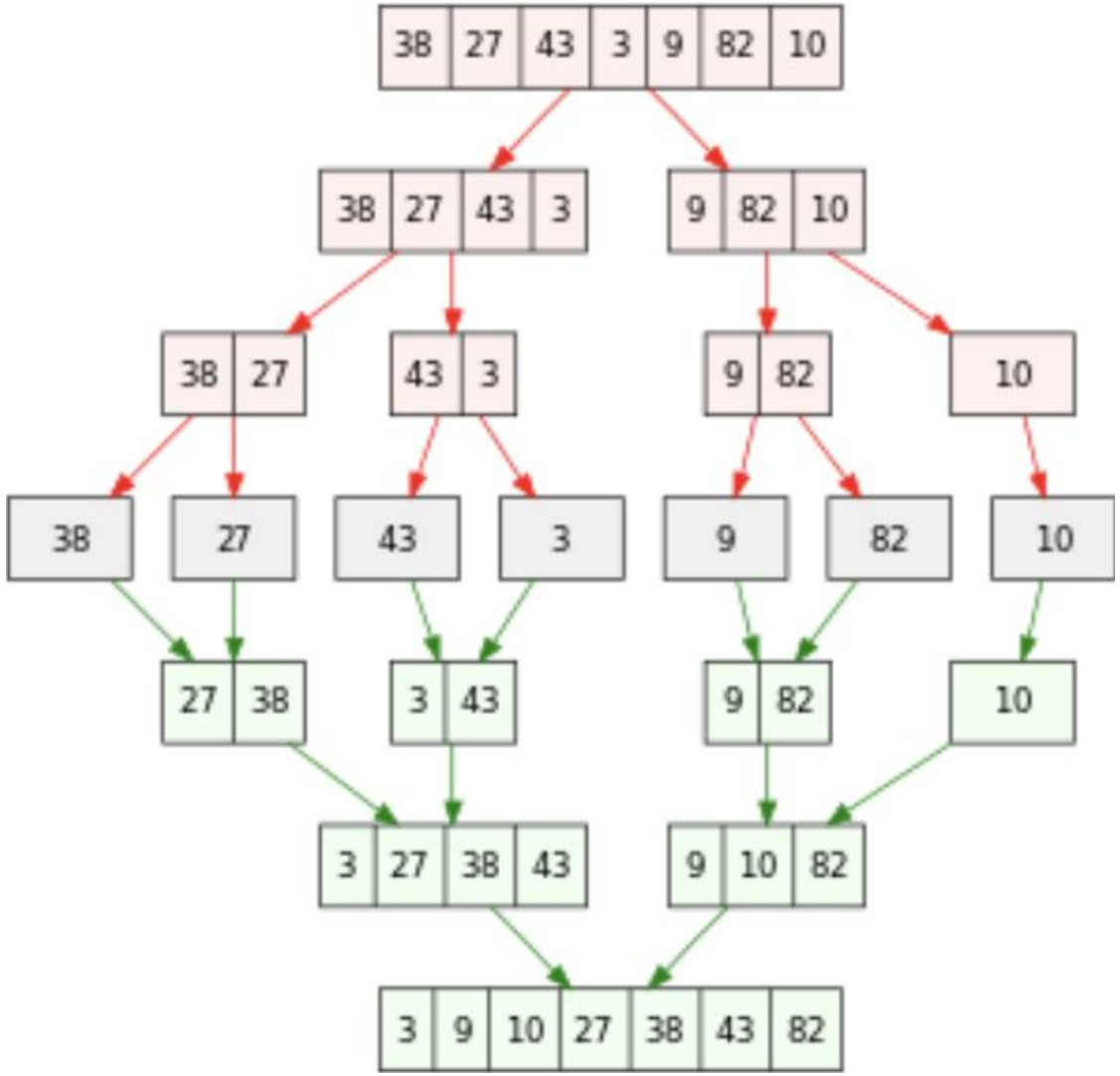
Merge Sort is a sorting algorithm based on the Divide et Impera technique, like Quick Sort. It can be implemented iteratively or recursively, using the Top-Down and Bottom-Up algorithms respectively. We represented the first one.

The algorithm divides the data structure recursively until the subsequences contain only one element. At this point, the subsequences get merged and ordered sequentially. To do so, the algorithm progressively builds the sorted sublist by adding each time the minimum element of the next two unsorted subsequences until there is only one sublist remaining. This will be the sorted data structure.

COMPLEXITY

Average Complexity	O(n × log n)
Best Case	O(n × log n)
Worst Case	O(n × log n)
Space Complexity	O(n)

Mergesort


38 27 43 3 9 82 10
38 27 43 3
9 82 10
38 27
43 3
9 82
10
38
27
43
3
9
82
10
27 38
3 43
9 82
10
3 27 38 43
9 10 82
3 9 10 27 38 43 82

Кто в python?

This wonderful property lets you build complex sorts in a series of sorting steps. For example, to sort the student data by descending *grade* and then ascending *age*, do the *age* sort first and then sort again using *grade*:

```
>>> s = sorted(student_objects, key=attrgetter('age'))     # sort on secondary key
>>> sorted(s, key=attrgetter('grade'), reverse=True)       # now sort on primary key,
[('dave', 'B', 10), ('jane', 'B', 12), ('john', 'A', 15)]
```

This can be abstracted out into a wrapper function that can take a list and tuples of field and order to sort them on multiple passes.

```
>>> def multisort(xs, specs):
...     for key, reverse in reversed(specs):
...         xs.sort(key=attrgetter(key), reverse=reverse)
...     return xs

>>> multisort(list(student_objects), (('grade', True), ('age', False)))
[('dave', 'B', 10), ('jane', 'B', 12), ('john', 'A', 15)]
```

The Timsort algorithm used in Python does multiple sorts efficiently because it can take advantage of any ordering already present in a dataset.

Так-то оно и было придумано для python

Timsort

11 languages

Article Talk Read Edit View history Tools

From Wikipedia, the free encyclopedia

Timsort is a hybrid, stable sorting algorithm, derived from merge sort and insertion sort, designed to perform well on many kinds of real-world data. It was implemented by Tim Peters in 2002 for use in the Python programming language. The algorithm finds subsequences of the data that are already ordered (runs) and uses them to sort the remainder more efficiently. This is done by merging runs until certain criteria are fulfilled. Timsort has been Python's standard sorting algorithm since version 2.3, but starting with 3.11 it uses Powersort instead, a derived algorithm with a more robust merge policy.[5] Timsort is also used to sort arrays of non-primitive type in Java SE 7,[6] on the Android platform,[7] in GNU Octave,[8] on V8,[9] in Swift,[10] and Rust.[11]

The galloping technique derives from Carlsson, Levcopoulos, and O. Petersson's 1990 paper "Sublinear merging and natural merge sort" and Peter McIlroy's 1993 paper "Optimistic Sorting and Information Theoretic Complexity".[12]

Timsort

Class	Sorting algorithm
Data structure	Array
Worst-case performance	$O(n \log n)$[1][2]
Best-case performance	$O(n)$[3]
Average performance	$O(n \log n)$
Worst-case space complexity	$O(n)$
Optimal	No; Powersort and Peeksort are closer to optimal[4]

Tim Peters

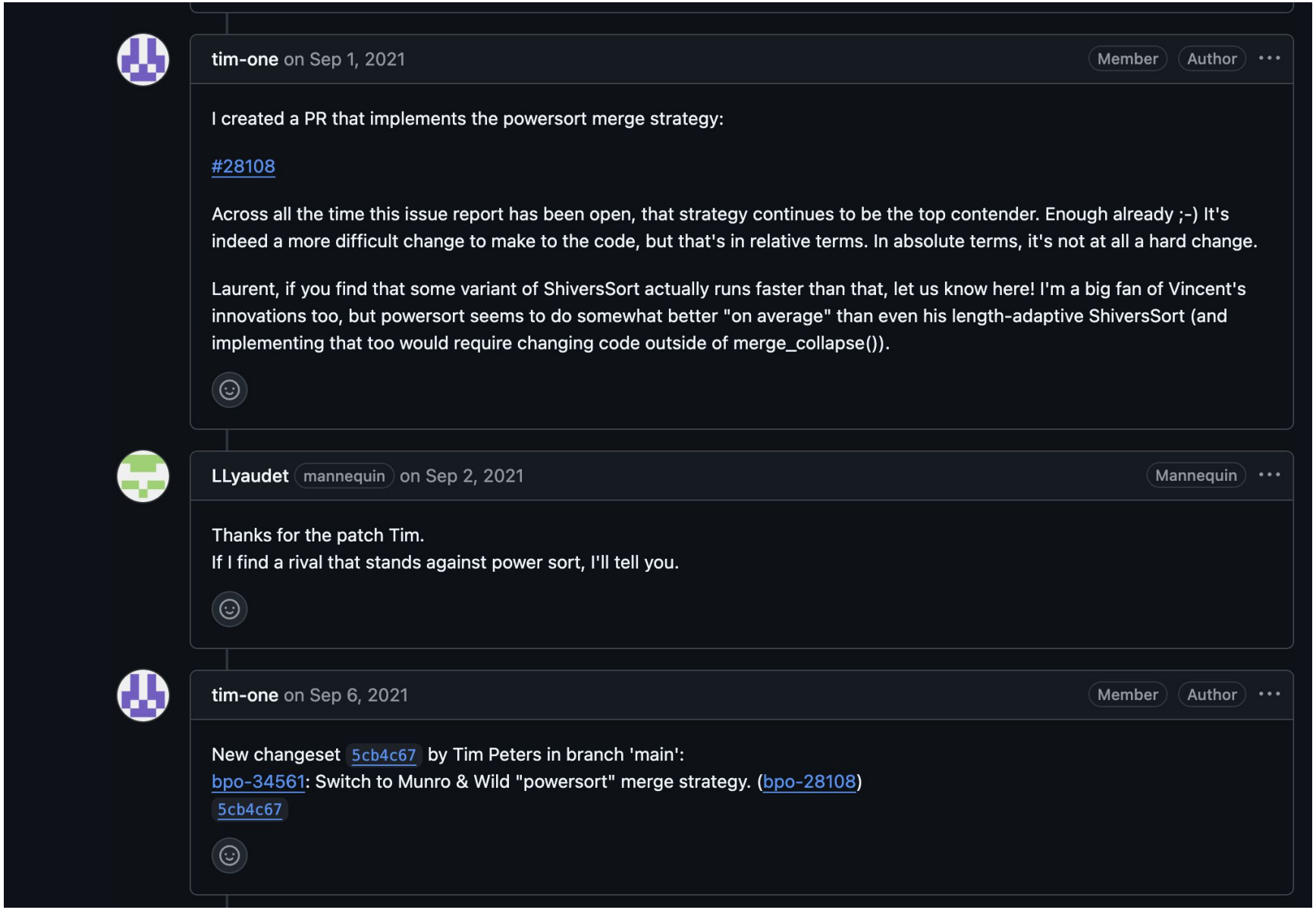
```
Python 3.13.1 (main, Dec  3 2024, 17:59:52) [Clang 16.0.0 (clang-1600.0.26.4)] on darwin
Type "help", "copyright", "credits" or "license" for more information.
>>> import this
The Zen of Python, by Tim Peters

Beautiful is better than ugly.
Explicit is better than implicit.
Simple is better than complex.
Complex is better than complicated.
Flat is better than nested.
Sparse is better than dense.
Readability counts.
Special cases aren't special enough to break the rules.
Although practicality beats purity.
Errors should never pass silently.
Unless explicitly silenced.
In the face of ambiguity, refuse the temptation to guess.
There should be one-- and preferably only one --obvious way to do it.
Although that way may not be obvious at first unless you're Dutch.
Now is better than never.
Although never is often better than *right* now.
If the implementation is hard to explain, it's a bad idea.
If the implementation is easy to explain, it may be a good idea.
Namespaces are one honking great idea -- let's do more of those!
```

Но потом

Liverpool computer scientists improve Python sorting function

А дальше



tim-one on Sep 1, 2021 — Member — Author

I created a PR that implements the powersort merge strategy:

#28108

Across all the time this issue report has been open, that strategy continues to be the top contender. Enough already ;-) It's indeed a more difficult change to make to the code, but that's in relative terms. In absolute terms, it's not at all a hard change.

Laurent, if you find that some variant of ShiversSort actually runs faster than that, let us know here! I'm a big fan of Vincent's innovations too, but powersort seems to do somewhat better "on average" than even his length-adaptive ShiversSort (and implementing that too would require changing code outside of merge_collapse()).

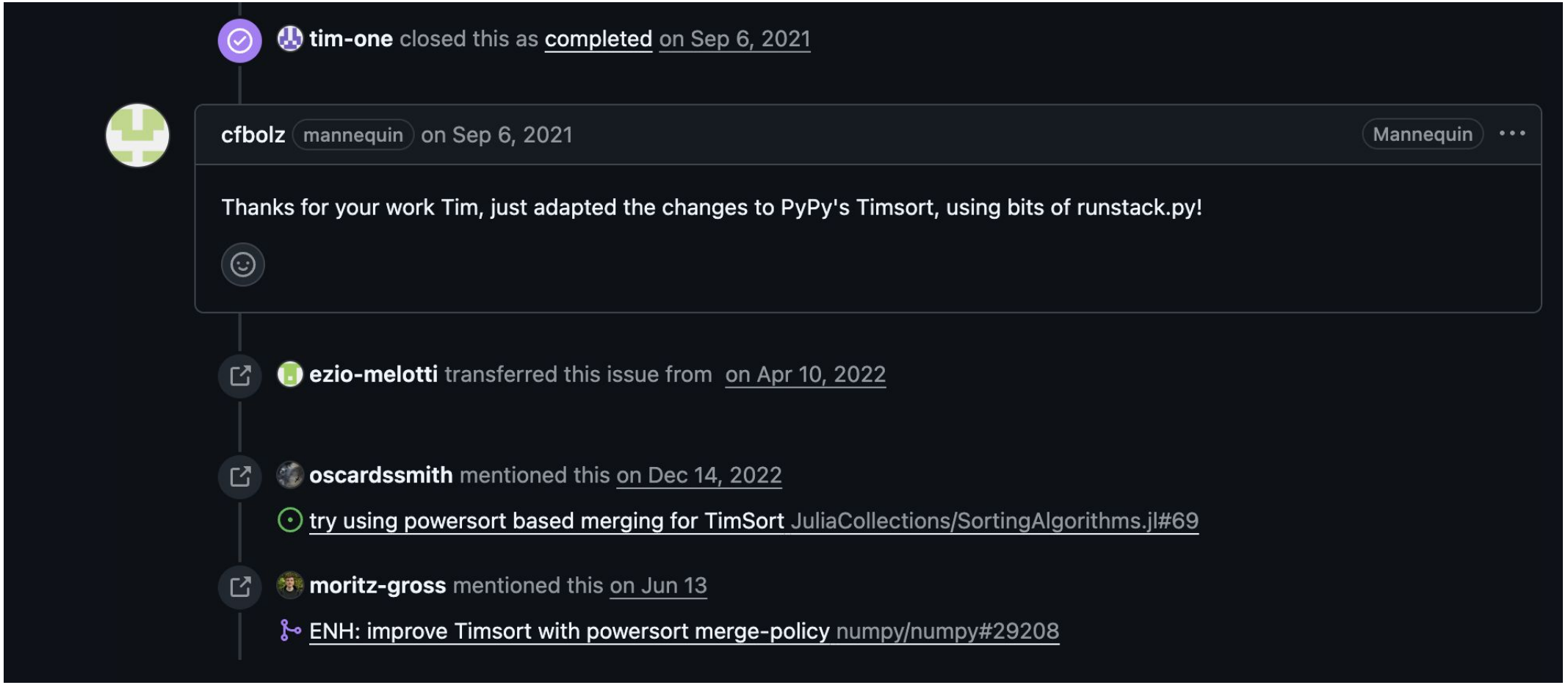
LLyaudet mannequin on Sep 2, 2021 — Mannequin

Thanks for the patch Tim.
If I find a rival that stands against power sort, I'll tell you.

tim-one on Sep 6, 2021 — Member — Author

New changeset 5cb4c67 by Tim Peters in branch 'main':
bpo-34561: Switch to Munro & Wild "powersort" merge strategy. (bpo-28108)
5cb4c67

После чего


tim-one closed this as completed on Sep 6, 2021
cfbolz mannequin on Sep 6, 2021
Mannequin
Thanks for your work Tim, just adapted the changes to PyPy's Timsort, using bits of runstack.py!
ezio-melotti transferred this issue from on Apr 10, 2022
oscardssmith mentioned this on Dec 14, 2022
try using powersort based merging for TimSort JuliaCollections/SortingAlgorithms.jl#69
moritz-gross mentioned this on Jun 13
ENH: improve Timsort with powersort merge-policy numpy/numpy#29208

В итоге

The Shameful Defenestration of Tim

On Tim Peters' recent suspension as a Python core developer

CHRIS MCDONOUGH

AUG 11, 2024

2

Share

When Tim Peters won the 2017 Python Distinguished Service Award, he had already been working on Python for over 20 years. In the early 1990s, he started with the pre-release 0.9.1 version. It did not have classes, and the assignment operator was the same as the comparison operator.

Timsort – гибрид

merge sort + insertion sort

Timsort – гибрид

Основные понятия:

- *minrun*
- *runs, natural runs (*участки массива, которые уже упорядочены*)*
- *merge criteria*

Timsort – гибрид

Определяем minrun (зависит от длины массива, выбирается из диапазона от 32 до 64) – минимальная длина рана, которую мы хотим иметь перед слиянием

Идем по массиву, формируем run (упорядоченный подмассив), если текущий размер меньше minrun, добавляем элемент, сортируя вставкой

Дальше раны складываем в стек

Timsort – гибрид

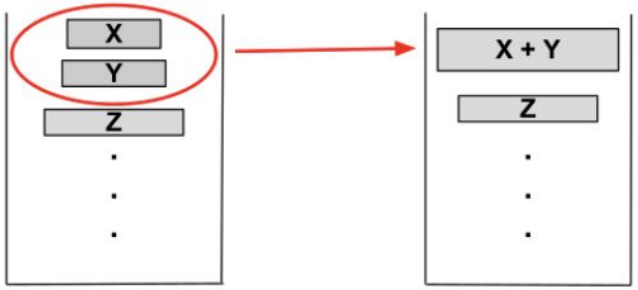
Мы стремимся к сбалансированному слиянию, поэтому есть инварианты или *merge criteria*

In pursuit of balanced merges, Timsort considers three runs on the top of the stack, *X*, *Y*, *Z*, and maintains the invariants:

i. $|Z| > |Y| + |X|$
ii. $|Y| > |X|$[13]

If any of these invariants is violated, *Y* is merged with the smaller of *X* or *Z* and the invariants are checked again. Once the invariants hold, the search for a new run in the data can start.[14] These invariants maintain merges as being approximately balanced while maintaining a compromise between delaying merging for balance, exploiting fresh occurrence of runs in cache memory and making merge decisions relatively simple.

$|Z| \le |Y| + |X|$, then *X* and *Y* are merged and replaced on the stack. In this way, merging is continued until all runs satisfy i. $|Z| > |Y| + |X|$ and ii. $|Y| > |X|$.



The runs are inserted in a stack. If $|Z| \le |Y| + |X|$, then *X* and *Y* are merged and replaced on the stack. In this way, merging is continued until all runs satisfy i. $|Z| > |Y| + |X|$ and ii. $|Y| > |X|$.

Timsort – гибрид

Mergesort – также оптимизирован

Merge space overhead [edit]

The original merge sort implementation is not in-place and it has a space overhead of N (data size). In-place merge sort implementations exist, but have a high time overhead. In order to achieve a middle term, Timsort performs a merge sort with a small time overhead and smaller space overhead than N.

Пример

[5, 7, 3, 1, 2, 8, 9, 6, 4, 10]
Пусть minrun = 4