Algorithms

Sorting (Ch. 7)

Morgan Ericsson

Today

- Sorting
- Simple sorts: Selection, Insert, Bubble, Shell
- Merge
- Quick
- Specialized
  - Radix

Sorting

Preliminaries

- We consider *comparison*-based sorting
  - I.e., `Comparable` and `compareTo` in Java
- To keep it simple, we generally assume `int`
  - But we can sort any type that is comparable
- and arrays (Python lists)
  - But we obviously sort linked structures

Total order

- A total order is a binary relation $\leq$ that satisfies
  - Antisymmetry: if $v \leq w$ and $w \leq v$, then $v = w$
  - Transitivity: if $v \leq w$ and $w \leq x$, then $v \leq x$
  - Totality: either $v \leq w$ or $w \leq v$ or both
- Standard order for, e.g., natural or real numbers

A sorted list

» Total order holds

» So, if `[a,b,c,d]` is sorted, ...

» ... $a \leq b \leq c \leq d$ should hold

Check if sorted

```
def is_sorted(l:list[int]) -> bool:
  for i in range(1, len(l)):
    if l[i - 1] > l[i]:
        return False
  return True
```

Testing it

```
import random

lst = random.sample(range(1, 1_001), k=20)

assert is_sorted(lst) == False
assert is_sorted(sorted(lst)) == True
```

Some sorting terminology

» In-place: the list is sorted in-place, i.e., it does not require any additional storage to sort the list

» Stable: Elements with the same value maintains their relative order

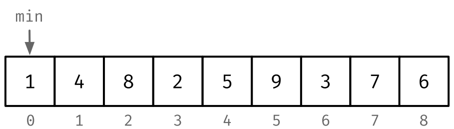
Simple algorithms

Selection sort

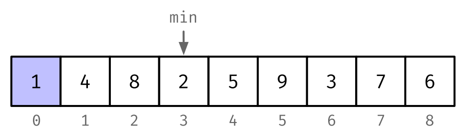
- Simple idea: in iteration *i*, find the index of the smallest remaining entry
- Swap the element at index i and the smallest value

Selection sort

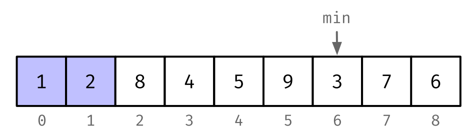
Iteration 0: find the smallest element in [0, 8] and swap with index 0



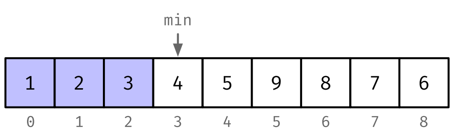
Iteration 1: find the smallest element in [1, 8] and swap with index 1



Iteration 2: find the smallest element in [2, 8] and swap with index 2

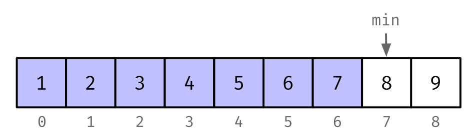


Iteration 3: find the smallest element in [3, 8] and swap with index 3



Iterations 4 to 6

Iteration 7: find the smallest element in [7, 8] and swap with index 7



Implementation

```python
def selection_sort(l:list[int]) -> None:
  n = len(l)
  for i in range(n):
    mn = i
    for j in range(i + 1, n):
        if l[mn] > l[j]:
            mn = j
    l[i], l[mn] = l[mn], l[i]
```

Testing it

```
lst = random.sample(range(1, 1_001), k=20)

assert is_sorted(lst) == False
selection_sort(lst)
assert is_sorted(lst) == True
```

Analysis

- In-place and unstable
  - Consider $[4, 3, 4', 1]$
- $(n - 1) + (n - 2) + \ldots + 1 + 0 \sim n^2 / 2$ compares and n swaps
- Insensitive to input, $O(n^2)$ whether sorted or completely random
- Minimal data movement

Insert sort

- In iteration *i*, swap the value at index i with each larger entry to its left
- So, move the value at index i to the correct place

Insert sort

Iteration 0: do nothing

0	1	2	3	4	5	6	7	8
1	4	8	2	5	9	3	7	6

Iteration 1: move 4 left while the elements are larger. In this case, do nothing

0	1	2	3	4	5	6	7	8
1	4	8	2	5	9	3	7	6

Iteration 2: move 8 left while the elements are larger. In this case, do nothing

0	1	2	3	4	5	6	7	8
1	2	8	4	5	9	3	7	6

Iteration 3: move 4 left while the elements are larger. Swaps 4 and 8.

0	1	2	3	4	5	6	7	8
1	2	8	4	5	9	8	7	6

Iteration 4: move 5 left while the elements are larger. Swaps 5 and 8.

0	1	2	3	4	5	6	7	8
1	2	4	8	5	9	8	7	6

Iteration 5: move 9 left while the elements are larger. In this case, do nothing

0	1	2	3	4	5	6	7	8
1	2	4	5	8	9	3	7	6

Implementation

```python
def insert_sort(l:list[int]) -> None:
  n = len(l)
  for i in range(n):
    for j in range(i, 0, -1):
      if l[j] < l[j-1]:
        l[j], l[j - 1] = l[j - 1], l[j]
      else:
        break
```

Testing it

```python
lst = random.sample(range(1, 1_001), k=20)

assert is_sorted(lst) == False
insert_sort(lst)
assert is_sorted(lst) == True
```

Analysis

- In-place and stable
- Depends on input
  - If sorted, $n - 1$ compares and 0 exchanges
  - If descending order, $\sim 0.5 \cdot n^2$ compares and exchanges
  - Average case, same but 0.25
- Still $O(n^2)$, but runs in linear time if partially sorted

Bubble sort

» Iterate over the list, compare pairs, and swap if left is smaller than right

» Keep iterating until there are no swaps

Bubble sort

Iteration 0
1 4 8 2 5 9 3 7 6
1 4 8 2 5 9 3 7 6
1 4 8 2 5 9 3 7 6
1 4 2 8 5 9 3 7 6
1 4 2 5 8 9 3 7 6
1 4 2 5 8 9 3 7 6
1 4 2 5 8 3 9 7 6
1 4 2 5 8 3 7 9 6
Iteration 1
1 4 2 5 8 3 7 6 9
1 4 2 5 8 3 7 6 9
1 2 4 5 8 3 7 6 9
1 2 4 5 8 3 7 6 9
1 2 4 5 8 3 7 6 9
1 2 4 5 3 8 7 6 9
1 2 4 5 3 7 8 6 9
Iteration 2
1 2 4 5 3 7 6 8 9
1 2 4 5 3 7 6 8 9
1 2 4 5 3 7 6 8 9
1 2 4 5 3 7 6 8 9
1 2 4 3 5 7 6 8 9
1 2 4 3 5 7 6 8 9

Implementation

```python
def bubble_sort(l:list[int]) -> None:
  n = len(l)
  for i in range(n):
    swp = False
    for j in range(n - i - 1):
      if l[j] > l[j + 1]:
        l[j], l[j + 1] = l[j + 1], l[j]
        swp = True
    if not swp:
      break
```

Testing it

```python
lst = random.sample(range(1, 1_001), k=20)

assert is_sorted(lst) == False
bubble_sort(lst)
assert is_sorted(lst) == True
```

Analysis

- In-place and stable
- Similar to insert sort
  - Depends on input, if almost sorted, linear
- So, $O(n^2)$

Shellsort

- Move elements more than one position at a time
- *h*-sorting
- if *h* is 4
  - Check `lst[h] < lst[h + 4]`
- Shellsort
  - *h*-sort the array with decreasing values of *h*
    - 13 sort, 4 sort, 1 sort

Shellsort

- We use insertion sort with stride *h*
- Big increments, small subarray
- Small increments, nearly in order

Shellsort

4-sort

0	1	2	3	4	5	6	7	8	9	10
27	54	30	38	89	29	95	57	80	42	99
27	54	30	38	89	29	95	57	80	42	99
27	29	30	38	89	54	95	57	80	42	99
27	29	30	38	89	54	95	57	80	42	99
27	29	30	38	89	54	95	57	80	42	99
27	29	30	38	80	54	95	57	89	42	99
27	29	30	38	80	42	95	57	89	54	99

1-sort

0	1	2	3	4	5	6	7	8	9	10
27	29	30	38	80	42	95	57	89	54	99
27	29	30	38	80	42	95	57	89	54	99
27	29	30	38	80	42	95	57	89	54	99
27	29	30	38	80	42	95	57	89	54	99
27	29	30	38	80	42	95	57	89	54	99
27	29	30	38	42	80	95	57	89	54	99
27	29	30	38	42	80	95	57	89	54	99
27	29	30	38	42	57	80	95	89	54	99
27	29	30	38	42	57	80	89	95	54	99
27	29	30	38	42	54	57	80	89	95	99

Shellsort

4-sort

0	1	2	3	4	5	6	7	8	9	10
11	10	9	8	**7**	6	5	4	3	2	1
7	**10**	9	8	11	**6**	5	4	3	2	1
7	6	**9**	8	11	10	**5**	4	3	2	1
7	6	5	**8**	11	10	9	**4**	3	2	1
3	**6**	5	4	7	**10**	9	8	11	**2**	1
3	2	**5**	4	7	6	**9**	8	11	10	**1**

1-sort

0	1	2	3	4	5	6	7	8	9	10
3	**2**	1	4	7	6	5	8	11	10	9
2	**3**	**1**	4	7	6	5	8	11	10	9
1	2	*3*	*4*	7	6	5	8	11	10	9
1	2	3	*4*	*7*	6	5	8	11	10	9
1	2	3	4	**7**	**6**	5	8	11	10	9
1	2	3	4	**6**	**7**	**5**	8	11	10	9
1	2	3	4	5	6	*7*	*8*	11	10	9
1	2	3	4	5	6	7	*8*	*11*	10	9
1	2	3	4	5	6	7	8	**11**	**10**	9
1	2	3	4	5	6	7	8	**10**	**11**	**9**

Which sequence of h?

- Any should work, but there are better and worse
- Powers of two is bad (only even until 1)
- $3x - 1$ is ok
  - Performs reasonably well and is easy to compute
- There are better sequences

Implementation

```python
def shellsort(l:list[int]) -> None:
  h, n = 1, len(l)
  while h < n // 3:
    h = 3 * h + 1

  while h >= 1:
    for i in range(h, n):
      j = i
      while j >= h and l[j] < l[j - h]:
        l[j], l[j - h] = l[j - h], l[j]
        j -= h
    h = h // 3
```

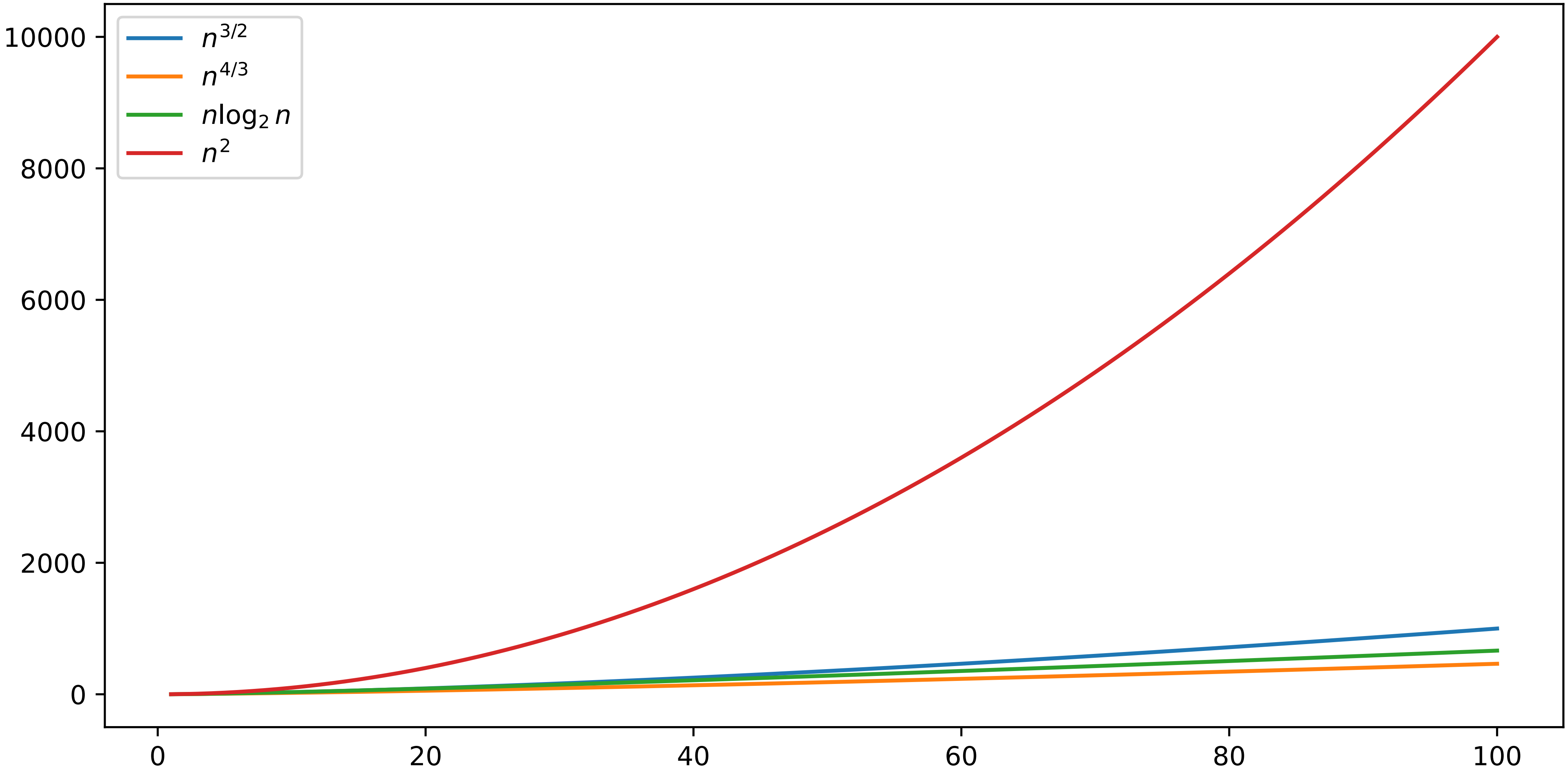
Testing it

```
lst = random.sample(range(1, 1_001), k=20)

assert is_sorted(lst) == False
shellsort(lst)
assert is_sorted(lst) == True
```

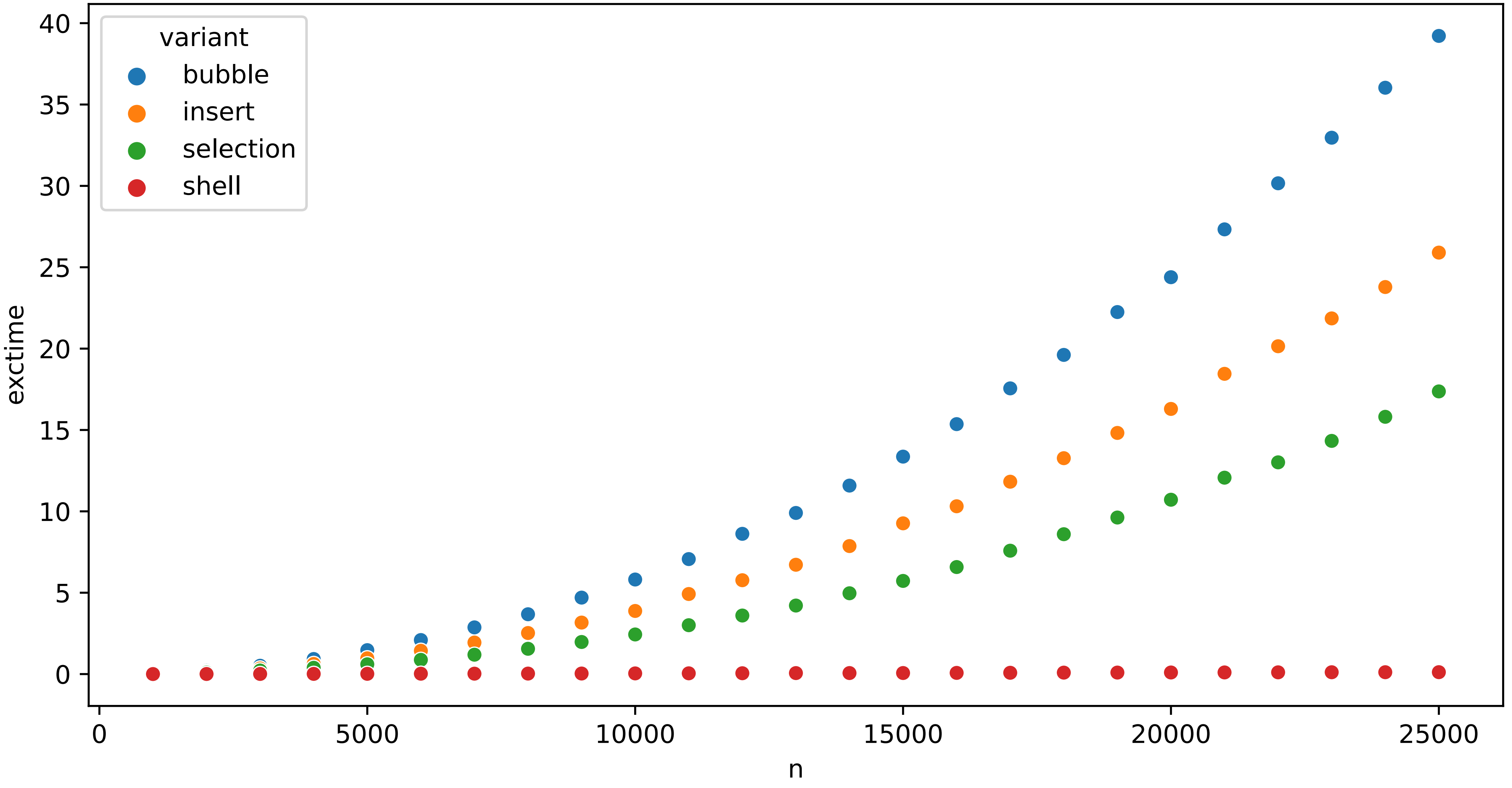
Analysis

- Quite difficult, depends on the sequence
  - And we do not know enough about it
- Bad sequence, $O(n^2)$
- Good sequence, $O(n^{\frac{4}{3}})$
- Ours, $O(n^{\frac{3}{2}})$

What does this mean?



In practice?



variant
bubble
insert
selection
shell
exctime
n
0
5
10
15
20
25
30
35
40
0
5000
10000
15000
20000
25000

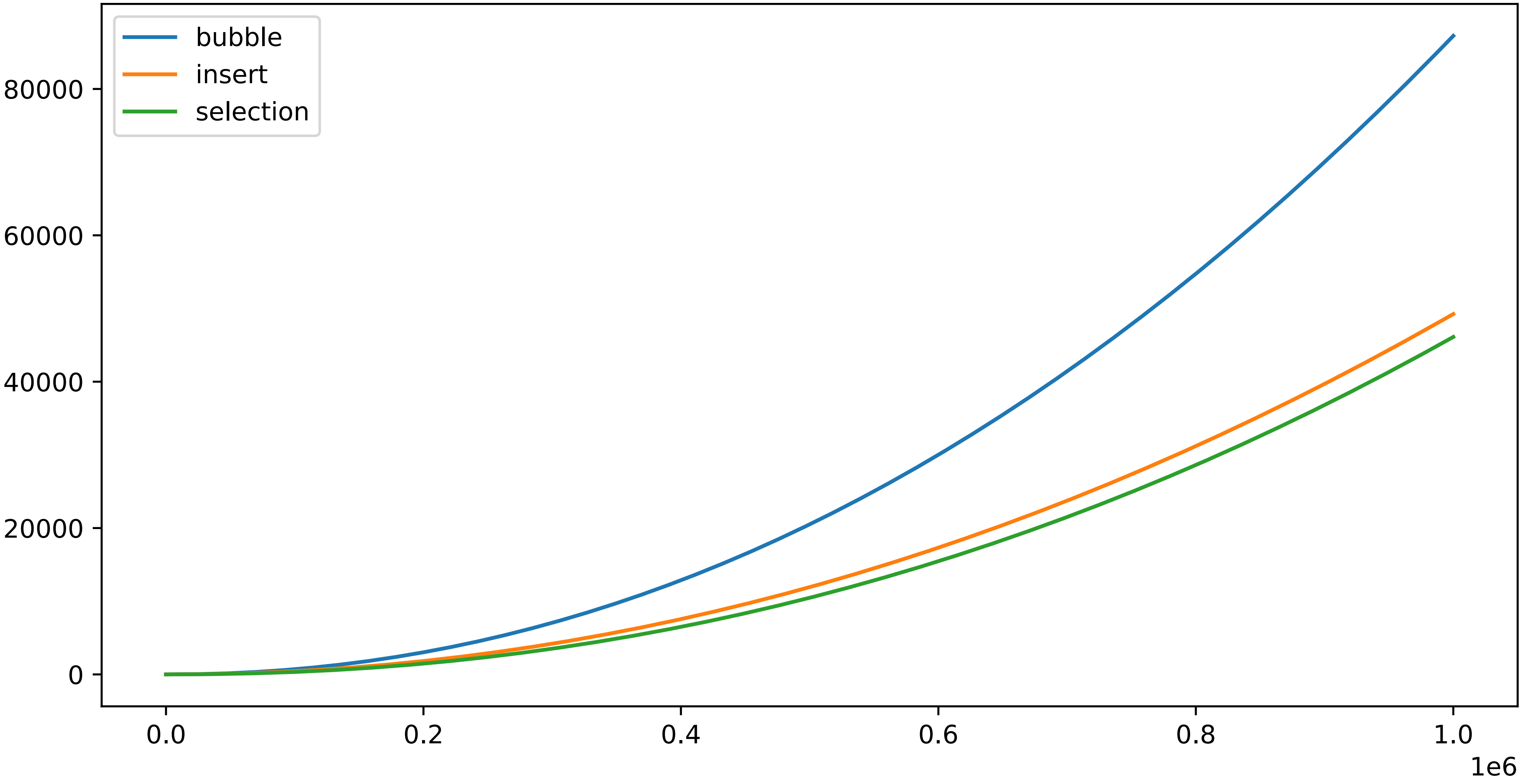
In practice?

$$\text{bubble} : 2.540\text{e} - 08 \cdot \text{x}^{2.089}$$

$$\text{insert} : 2.605\text{e} - 08 \cdot \text{x}^{2.046}$$

$$\text{selection} : 6.772\text{e} - 09 \cdot \text{x}^{2.139}$$

In practice



bubble
insert
selection
80000
60000
40000
20000
0
0.0
0.2
0.4
0.6
0.8
1.0
1e6

Mergesort

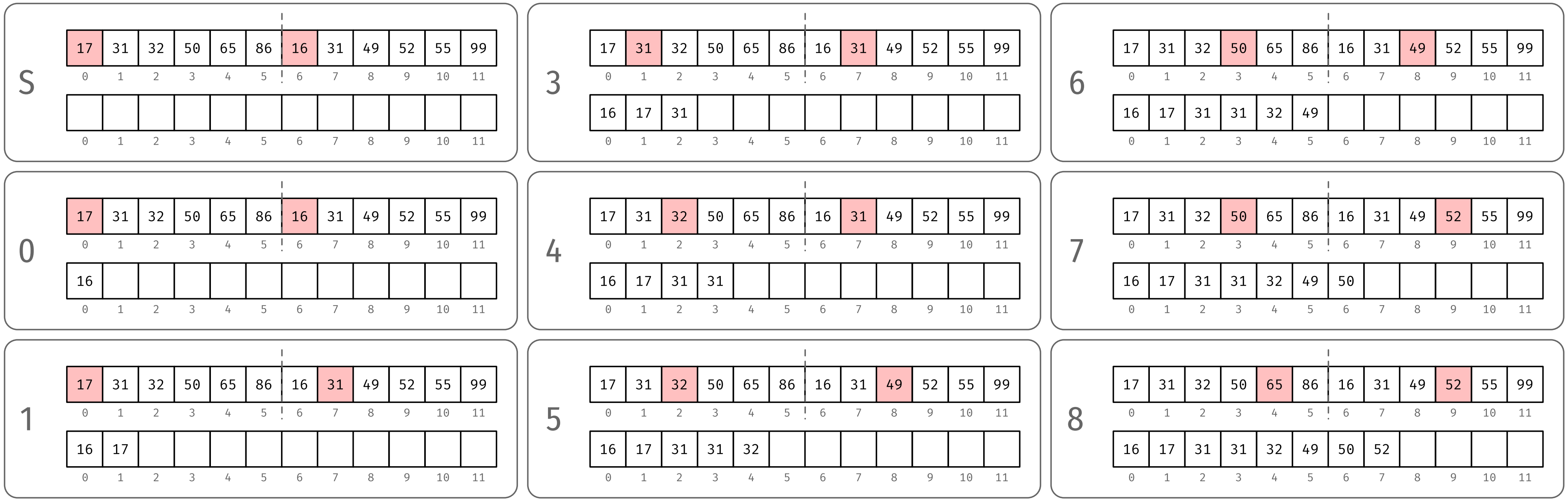
Mergesort

- Simple idea
  - Split the list in half
  - (Merge)Sort both halves (recursively)
  - Merge the two sorted lists
- Divide and conquer

Merge

- We can merge two sorted lists in $O(m + n)$, where m and n are the sizes of the two lists
- Advance pointers in the two lists independently
- Pick the smallest and add to the merged list

Merge



S
17 31 32 50 65 86 16 31 49 52 55 99
0 1 2 3 4 5 6 7 8 9 10 11
0 1 2 3 4 5 6 7 8 9 10 11
0
17 31 32 50 65 86 16 31 49 52 55 99
0 1 2 3 4 5 6 7 8 9 10 11
16
0 1 2 3 4 5 6 7 8 9 10 11
1
17 31 32 50 65 86 16 31 49 52 55 99
0 1 2 3 4 5 6 7 8 9 10 11
16 17
0 1 2 3 4 5 6 7 8 9 10 11
3
17 31 32 50 65 86 16 31 49 52 55 99
0 1 2 3 4 5 6 7 8 9 10 11
16 17 31
0 1 2 3 4 5 6 7 8 9 10 11
4
17 31 32 50 65 86 16 31 49 52 55 99
0 1 2 3 4 5 6 7 8 9 10 11
16 17 31 31
0 1 2 3 4 5 6 7 8 9 10 11
5
17 31 32 50 65 86 16 31 49 52 55 99
0 1 2 3 4 5 6 7 8 9 10 11
16 17 31 31 32
0 1 2 3 4 5 6 7 8 9 10 11
6
17 31 32 50 65 86 16 31 49 52 55 99
0 1 2 3 4 5 6 7 8 9 10 11
16 17 31 31 32 49
0 1 2 3 4 5 6 7 8 9 10 11
7
17 31 32 50 65 86 16 31 49 52 55 99
0 1 2 3 4 5 6 7 8 9 10 11
16 17 31 31 32 49 50
0 1 2 3 4 5 6 7 8 9 10 11
8
17 31 32 50 65 86 16 31 49 52 55 99
0 1 2 3 4 5 6 7 8 9 10 11
16 17 31 31 32 49 50 52
0 1 2 3 4 5 6 7 8 9 10 11

Implementation

```python
class MergeSort:
  def _merge(self, a:list[int], tmp:list[int], \
               lo:int, mid:int, hi:int) -> None:
    for k in range(lo, hi+1):
      tmp[k] = a[k]

    i, j = lo, mid + 1
    for k in range(lo, hi+1):
      if i > mid:
        a[k] = tmp[j]
        j += 1
      elif j > hi:
        a[k] = tmp[i]
        i += 1
      elif tmp[j] < tmp[i]:
        a[k] = tmp[j]
        j += 1
      else:
        a[k] = tmp[i]
        i += 1
```

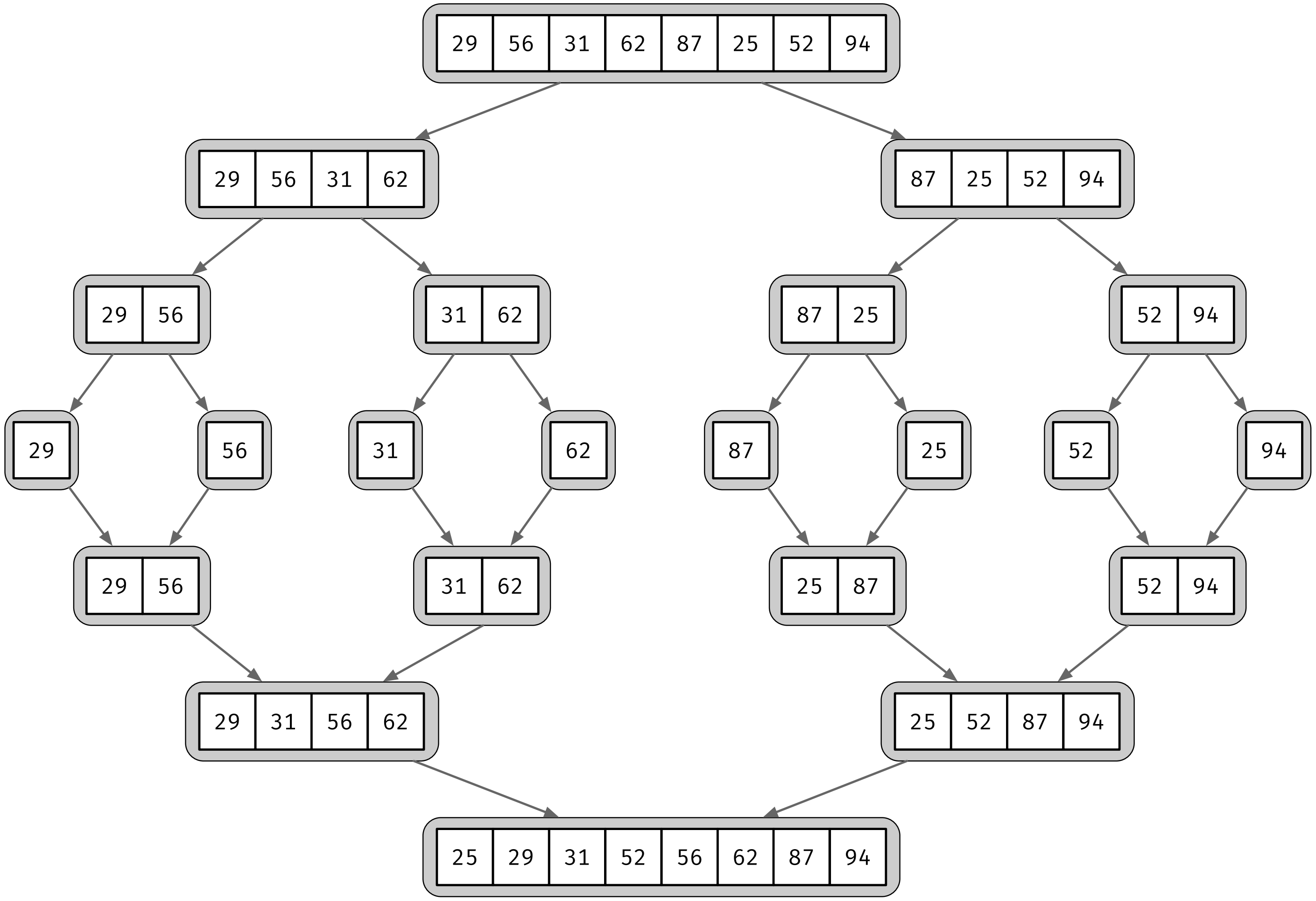
Testing it

```
lst = [17, 31, 32, 50, 65, 86, 16, 31, 49, 52, 55,
tmp = [0] * len(lst)
ms = MergeSort()
ms._merge(lst, tmp, 0, len(lst) // 2 - 1, len(lst)
assert is_sorted(lst) == True
```

Sorting

- When is a random list sorted?
  - When it has 1 (or 0) elements
- Divide lists until they have one element
- Then merge them together in sorted order

Mergesort



29 56 31 62 87 25 52 94
29 56 31 62
87 25 52 94
29 56
31 62
87 25
52 94
29
56
31
62
87
25
52
94
29 56
31 62
25 87
52 94
29 31 56 62
25 52 87 94
25 29 31 52 56 62 87 94

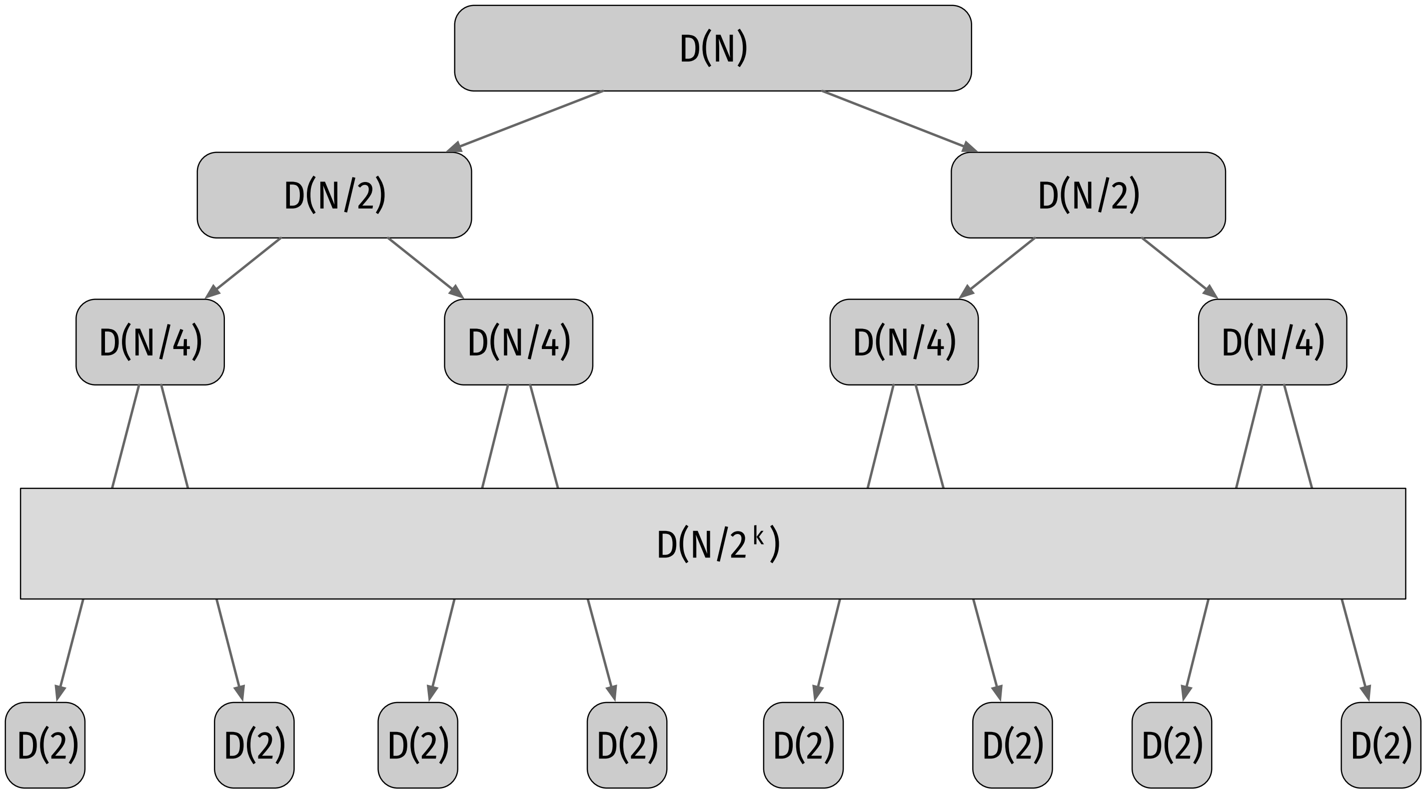
Implementation

```
from fastcore.basics import patch

@patch
def _sort(self:MergeSort, a:list[int], tmp:list[int], \
          lo:int, hi:int) -> None:
  if hi <= lo:
    return

  mid = lo + (hi - lo) // 2
  self._sort(a, tmp, lo, mid)
  self._sort(a, tmp, mid+1, hi)
  self._merge(a, tmp, lo, mid, hi)

@patch
def sort(self:MergeSort, a:list[int]) -> None:
  tmp = [0] * len(a)
  self._sort(a, tmp, 0, len(a) - 1)
```

Testing it

```
lst = [29, 56, 31, 62, 87, 25, 52, 94]
ms = MergeSort()
ms.sort(lst)
assert is_sorted(lst) == True
```

Analysis



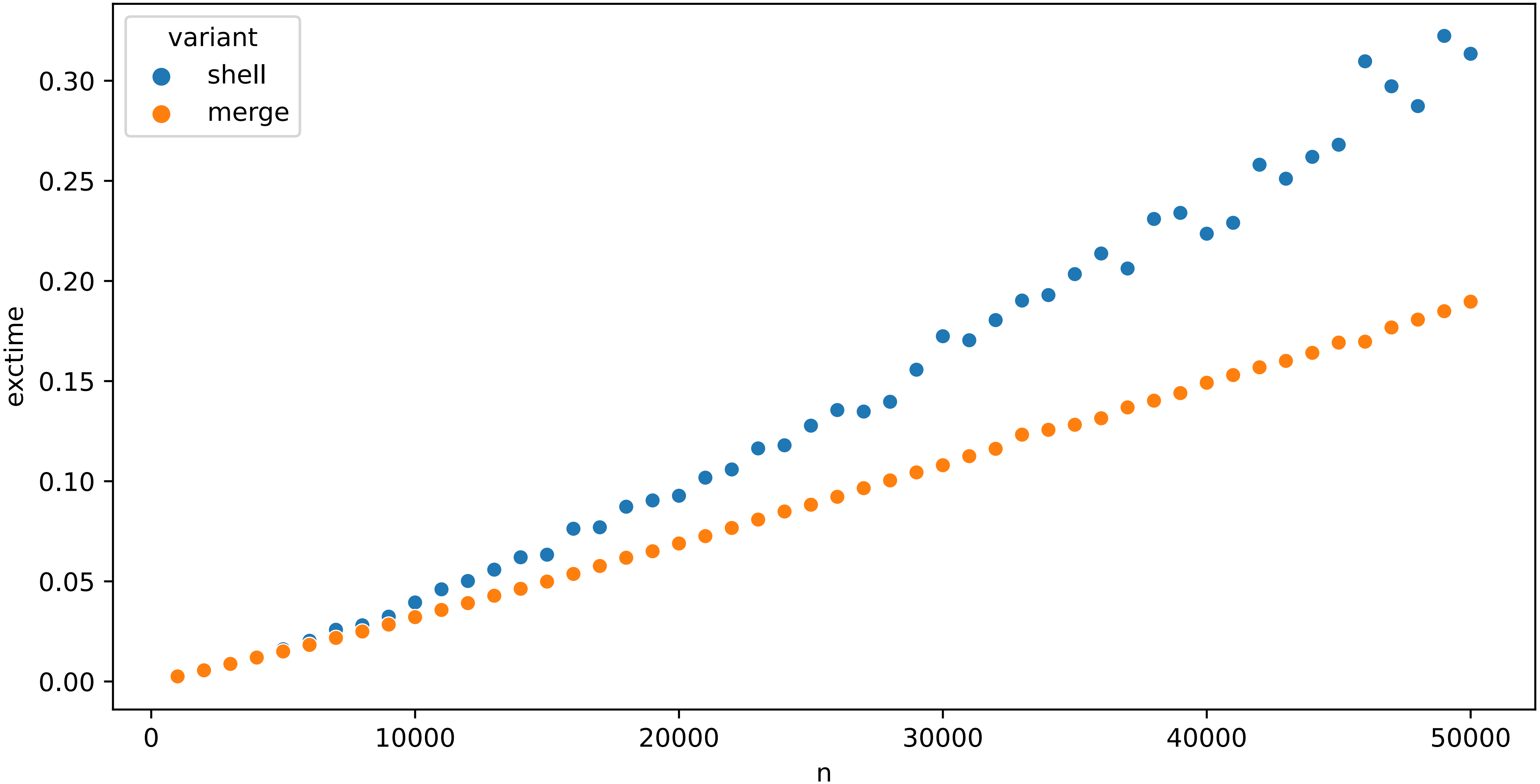
N	=N
2(N/2)	=N
4(N/4)	=N
$2^k(N/2^k)$	=N
(N/2)(2)	=N
	$=N \log_2 N$

(Assuming N is a power of 2)

Analysis

- Not in place, but can be
- Stable
- Almost perfect in terms or comparisons
- $O(n \log n)$

In practice

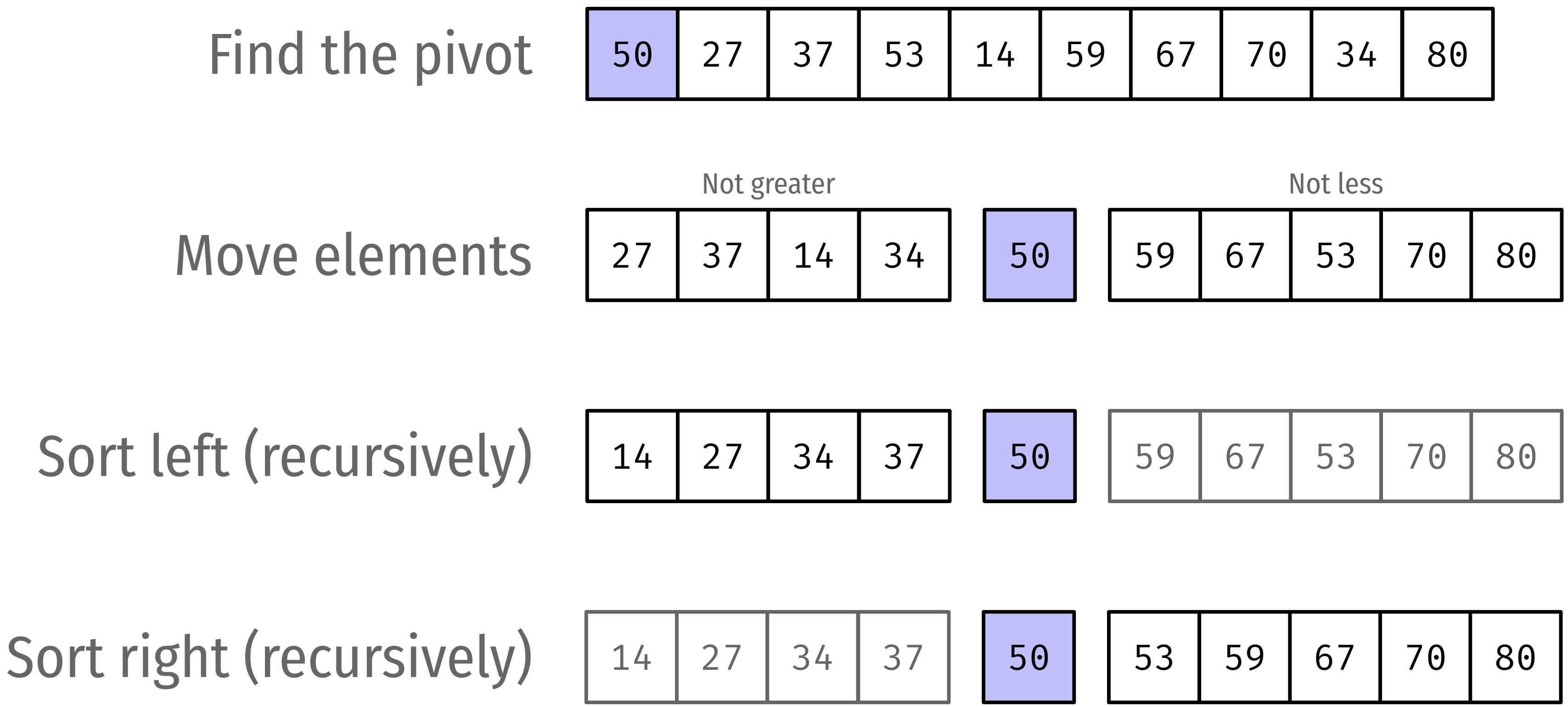

variant
shell
merge
0.30
0.25
0.20
0.15
0.10
0.05
0.00
exctime
0
10000
20000
30000
40000
50000
n

Quicksort

Quicksort

- Divide and conquer, just like Mergesort
- Split the input into two smaller parts
- But split around a *pivot* value and ensure that
  - Values to the left are not greater than ...
  - .. and values to the right not less than the pivot
- Avoids the merge step

Quicksort

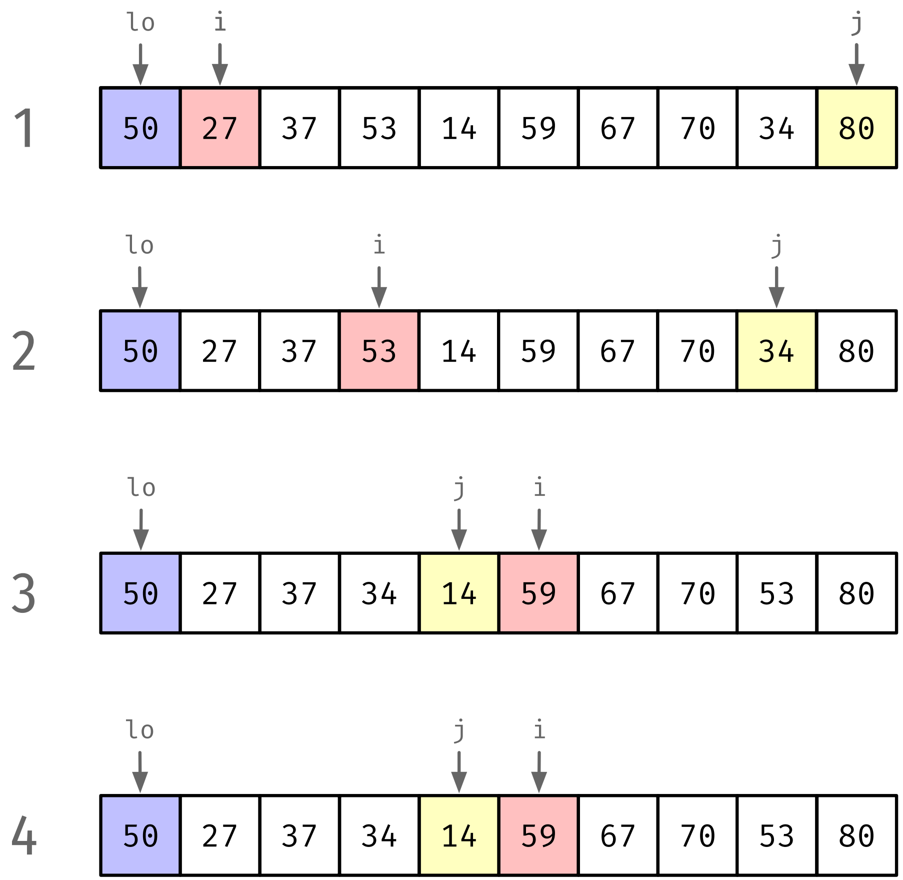


Find the pivot
50 27 37 53 14 59 67 70 34 80
Not greater
Not less
Move elements
27 37 14 34 50 59 67 53 70 80
Sort left (recursively)
14 27 34 37 50 59 67 53 70 80
Sort right (recursively)
14 27 34 37 50 53 59 67 70 80

Implementation

```python
class Quicksort:
  def _partition(self, a:list[int], lo:int, hi:int) -> int:
    i, j = lo, hi + 1

    while True:
      i += 1
      while a[i] < a[lo]:
        if i == hi: break
        i += 1

      j -= 1
      while a[lo] < a[j]:
        if j == lo: break
        j -= 1

      if i >= j: break
      a[i], a[j] = a[j], a[i]

    a[lo], a[j] = a[j], a[lo]
    return j
```

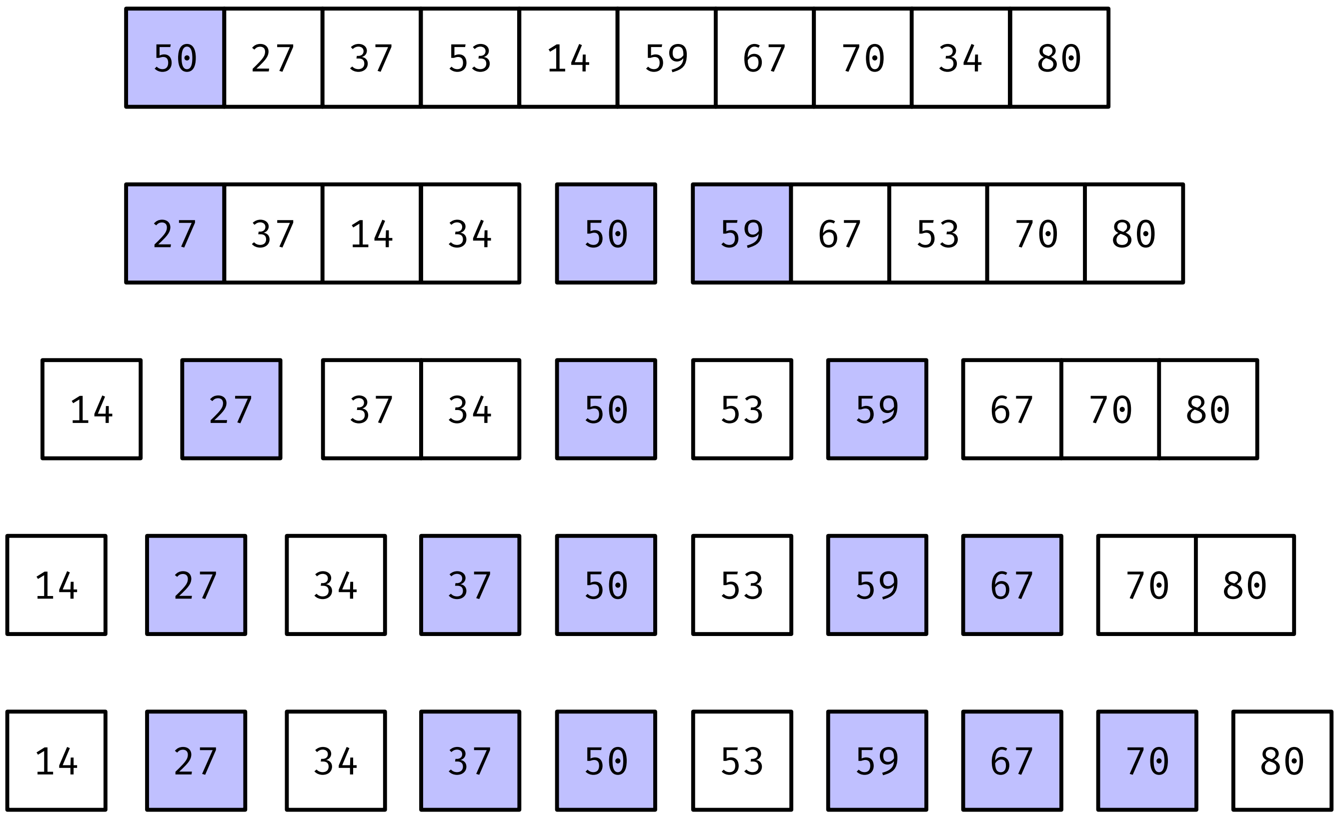
Partition


lo i j
1
50 27 37 53 14 59 67 70 34 80
lo i j
2
50 27 37 53 14 59 67 70 34 80
lo j i
3
50 27 37 34 14 59 67 70 53 80
lo j i
4
50 27 37 34 14 59 67 70 53 80

Implementation

```
@patch
def _sort(self:Quicksort, a:list[int], \
              lo:int, hi:int) -> None:
  if hi <= lo:
    return
  j = self._partition(a, lo, hi)
  self._sort(a, lo, j - 1)
  self._sort(a, j + 1, hi)

@patch
def sort(self:Quicksort, a:list[int]) -> None:
  self._sort(a, 0, len(a) - 1)
```

Partition and sort



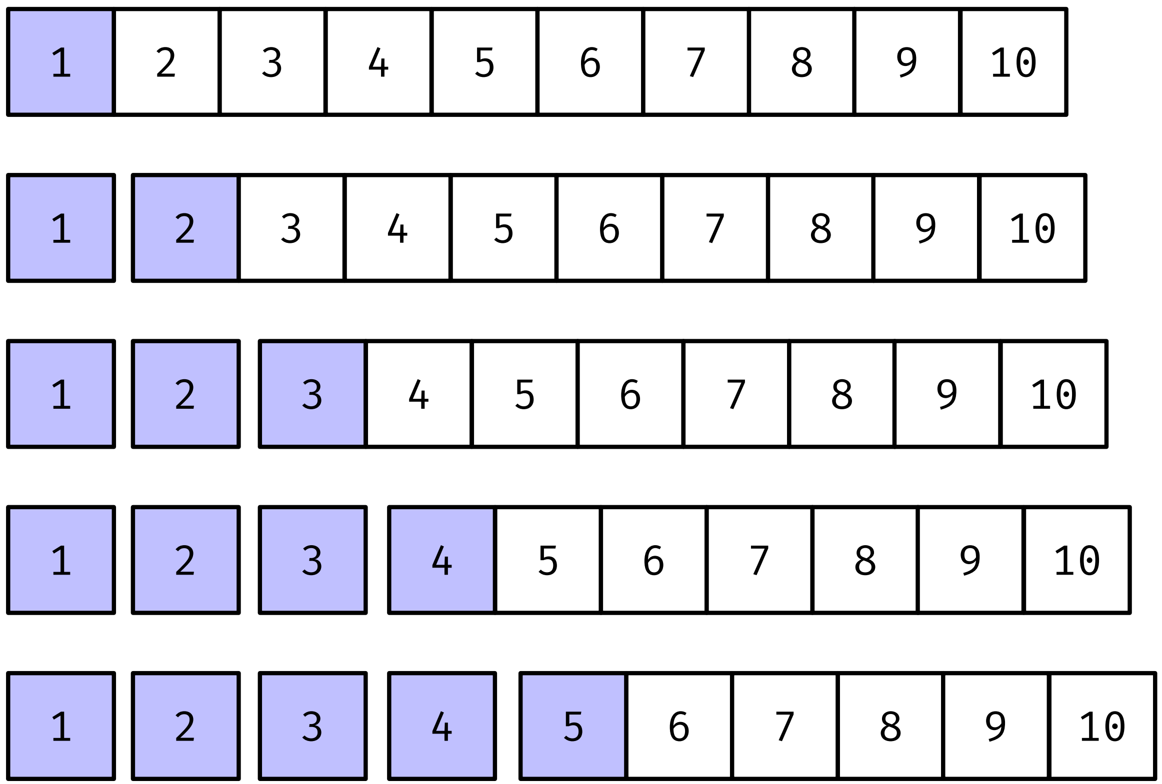
Analysis

» In-place, not stable

» $\sim n \log n$ average case

» $\sim n^2 / 2$ worst case

Worst case?


1 2 3 4 5 6 7 8 9 10
1 2 3 4 5 6 7 8 9 10
1 2 3 4 5 6 7 8 9 10
1 2 3 4 5 6 7 8 9 10
1 2 3 4 5 6 7 8 9 10

Improving the worst case?

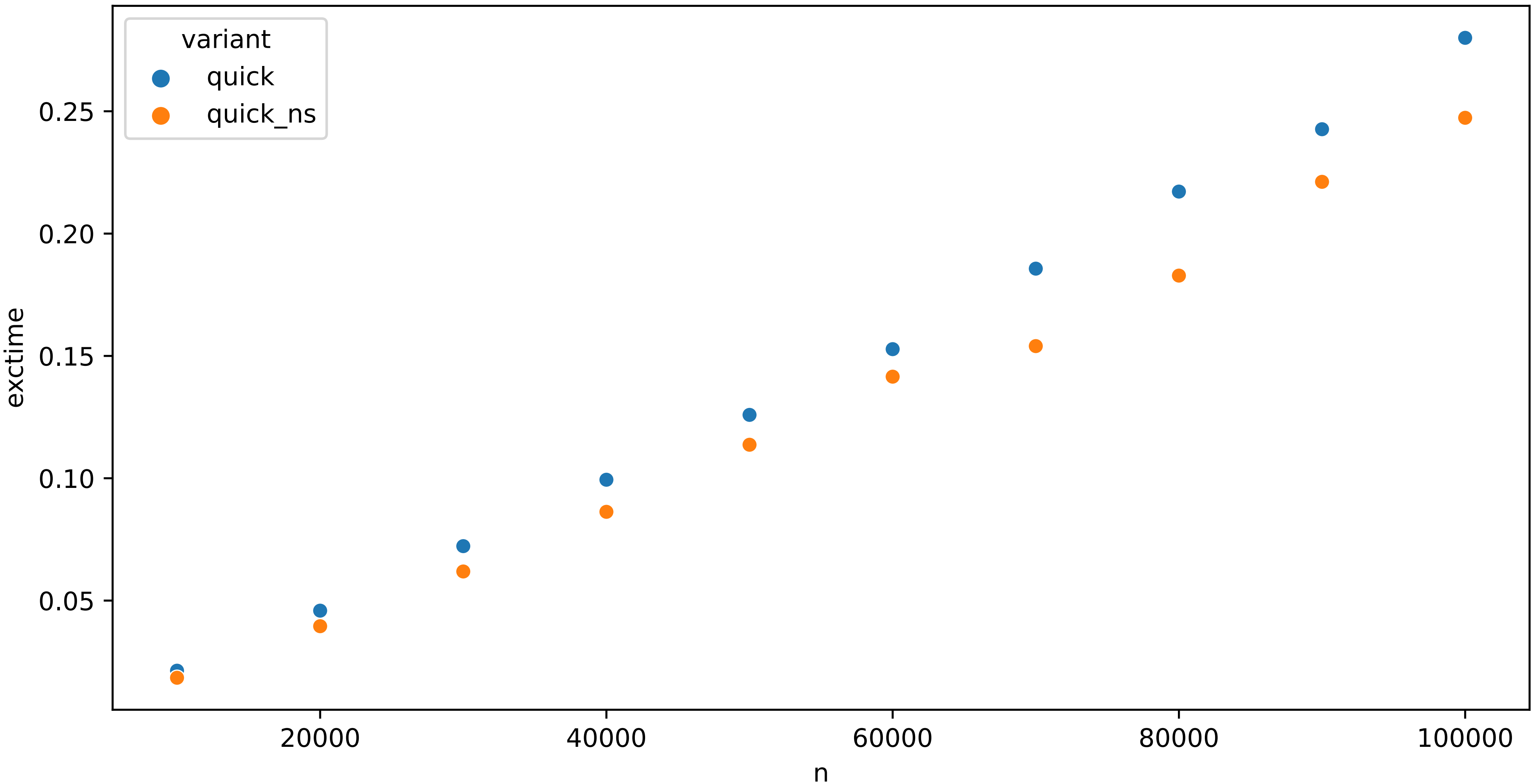
- The worst case is extremely rare
- Ideally, we want the pivot to be the median
  - Too expensive to compute $(O(n))$
- We can shuffle
- Or approximate the median from $[lo, mid, hi]$

Implementation

```python
@patch
def sort(self:Quicksort, a:list[int]) -> None:
  random.shuffle(a)
  self._sort(a, 0, len(a) - 1)
```

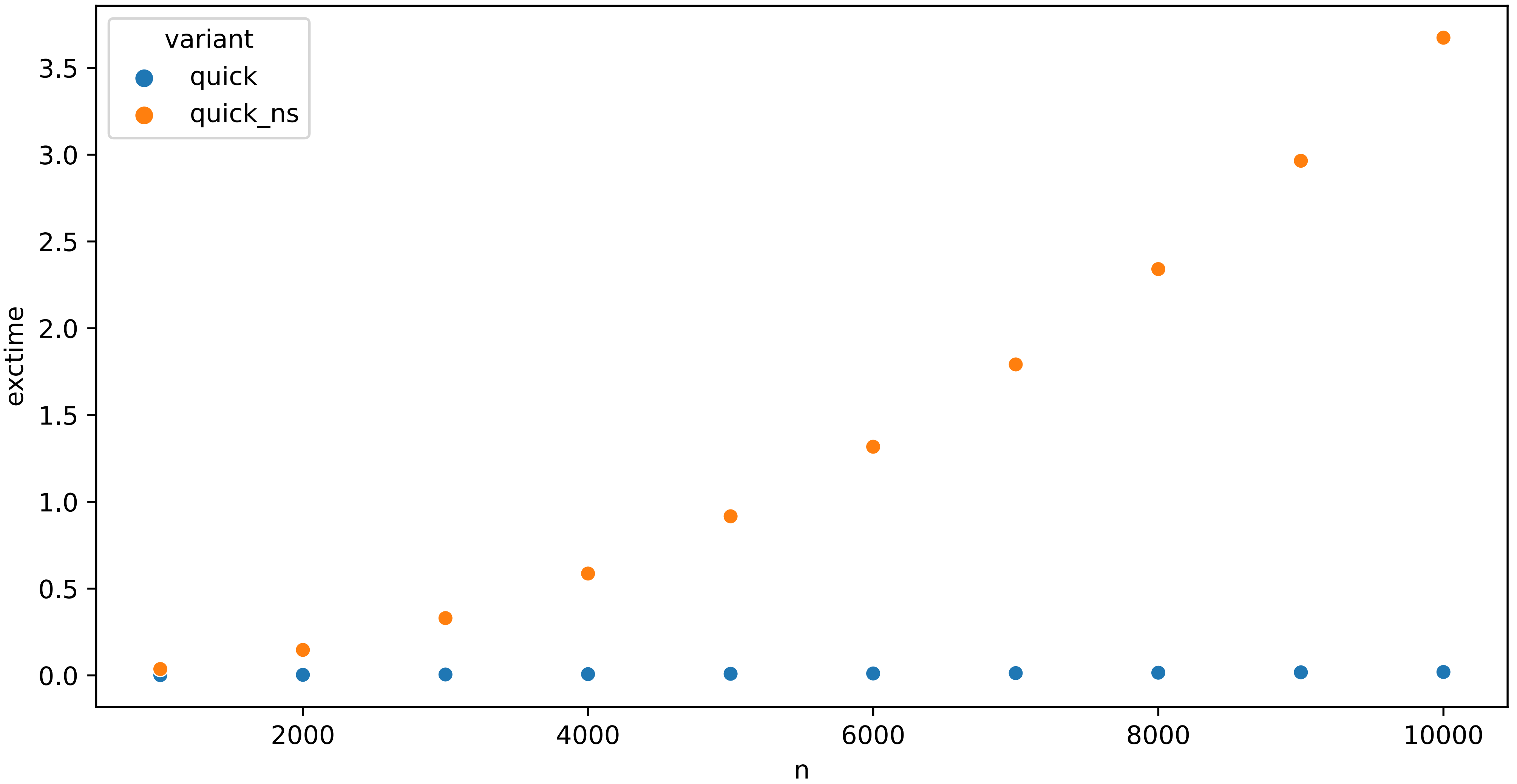
Does it matter in reality?

Random arrays



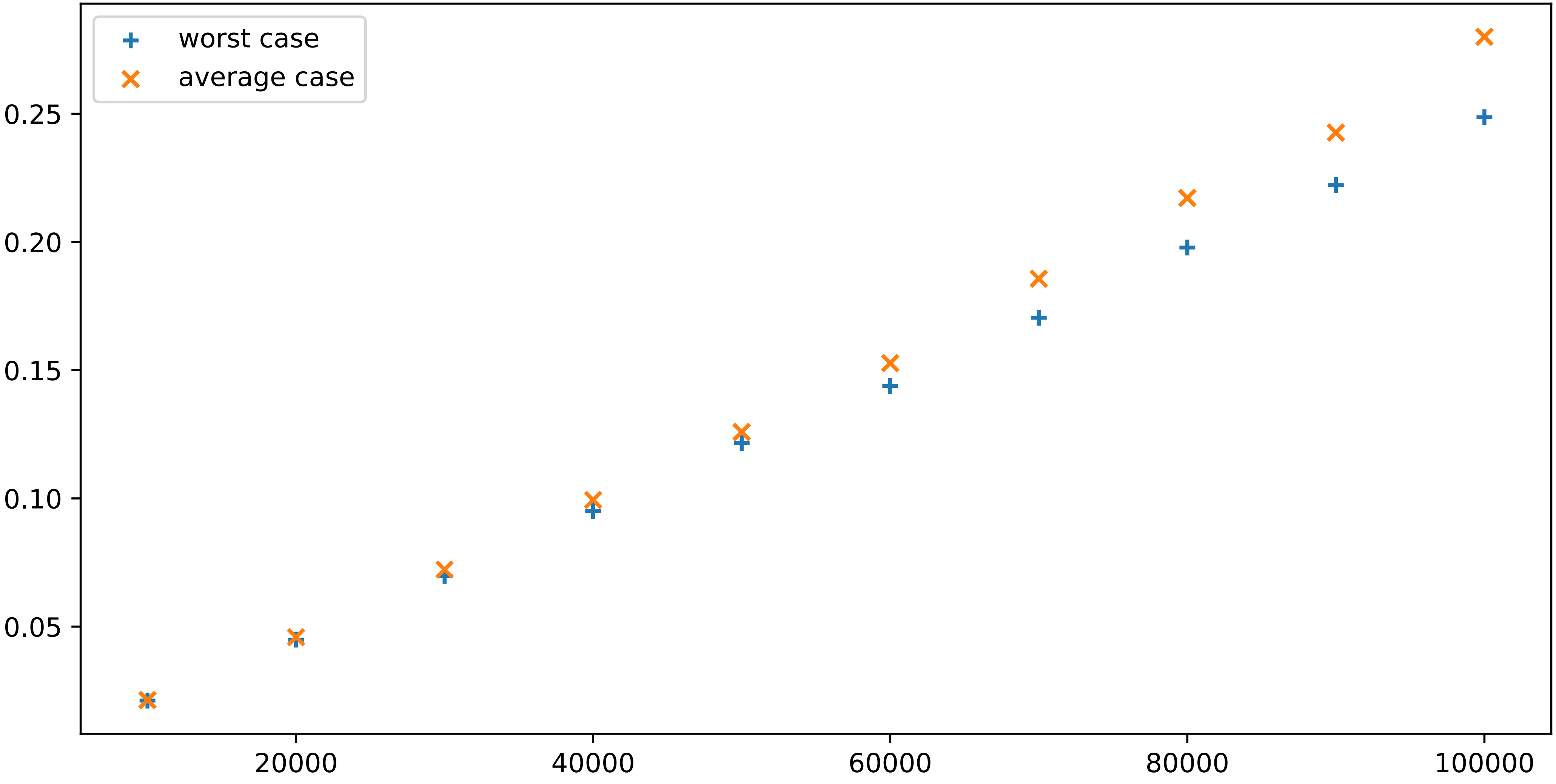
Does it matter in reality?

Worst case

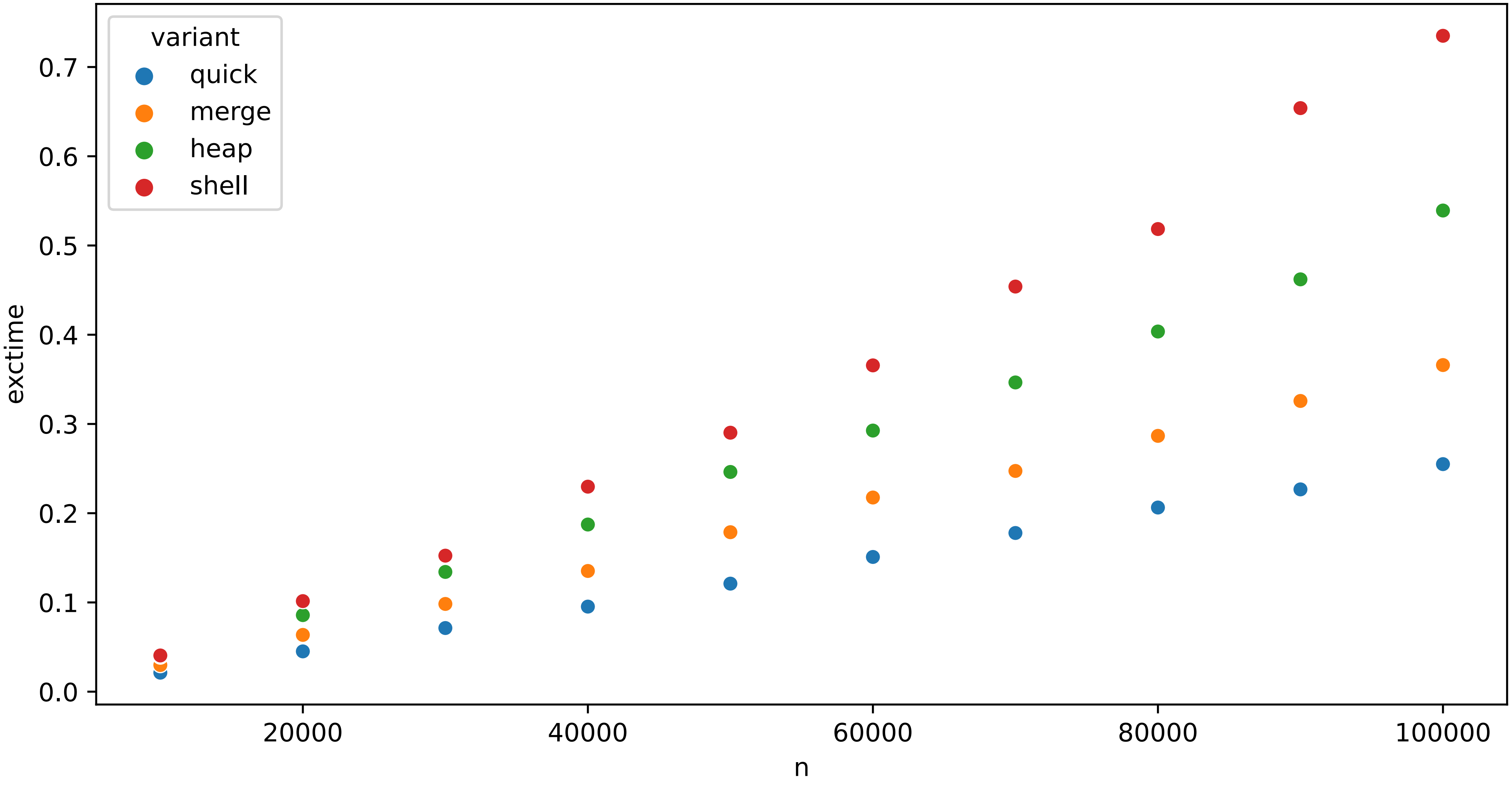


Does it matter in reality?

Worst and average case (shuffle)



Heap vs merge vs quick

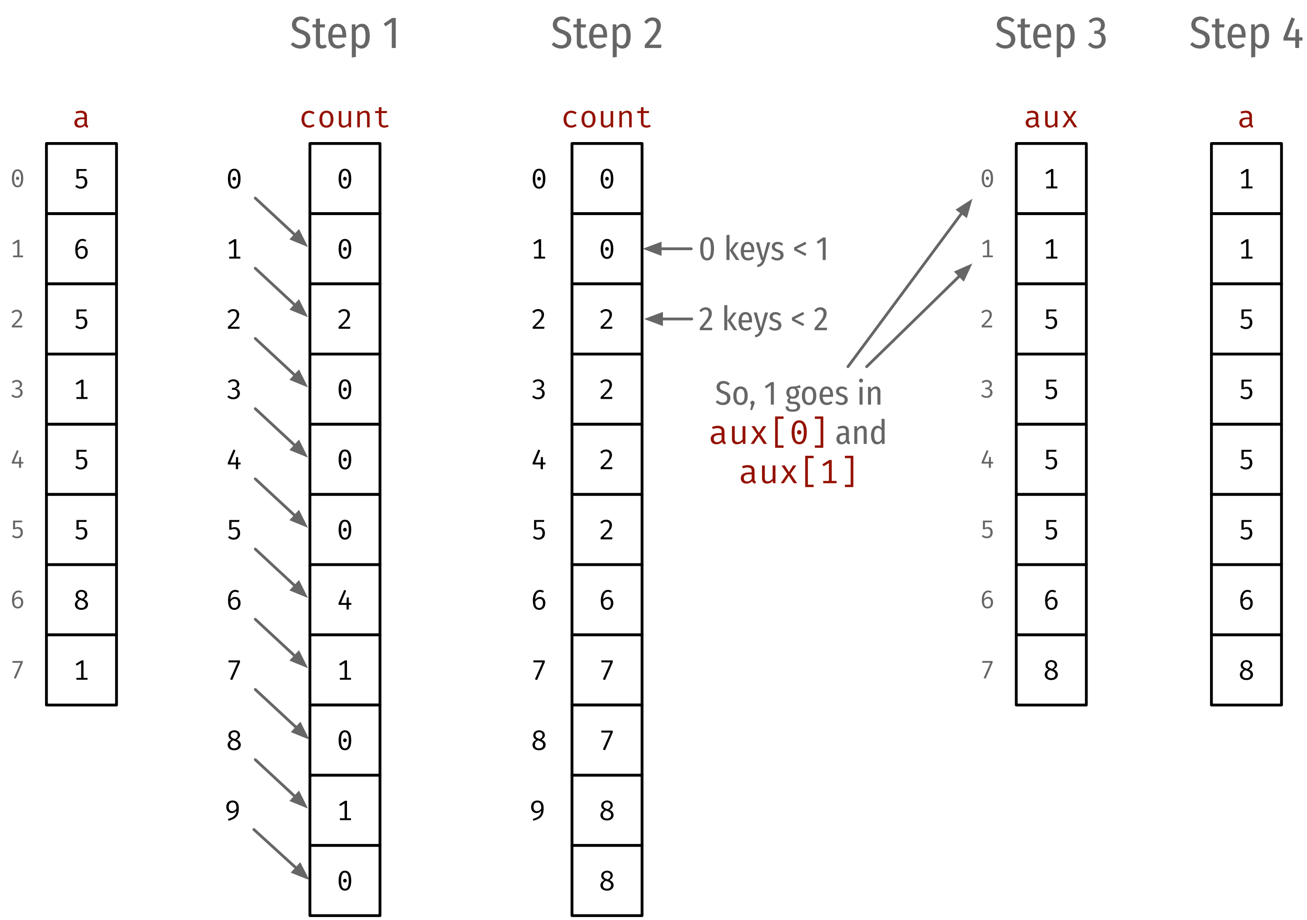


Radix sort

“Counting” sorts

- We know that comparison-based sort is $\Omega(n \log n)$
- We can reduce this if we avoid comparing
- But how can we sort without comparing?
  - We can count…

Illustrating the idea



Implementation

```python
def bucketsort(a:list[int], mx:int) -> None:
  n = len(a)
  cnt, aux = [0] * (mx + 1), [0] * n

  for i in range(n):
    cnt[a[i] + 1] += 1

  for i in range(mx):
    cnt[i+1] += cnt[i]

  for i in range(n):
    aux[cnt[a[i]]] = a[i]
    cnt[a[i]] += 1

  for i in range(n):
    a[i] = aux[i]
```

Testing

```
lst = random.choices(range(0, 10), k=10)
print(lst)
print(sorted(lst))
bucketsort(lst, 10)
print(lst)
```

```
[3, 8, 1, 9, 6, 2, 2, 0, 9, 5]
[0, 1, 2, 2, 3, 5, 6, 8, 9, 9]
[0, 1, 2, 2, 3, 5, 6, 8, 9, 9]
```

Extending to characters/strings

» We can use the same idea to sort a list of strings

» We just to it character per character

» To keep it simple, we assume fixed length strings

» And 8-bit characters

Illustrating the idea

0	e	e	e
1	e	c	e
2	c	a	d
3	f	d	b
4	b	a	c
5	f	f	d
6	b	f	e
7	c	b	f

d = 2

f	d	b
b	a	c
c	a	d
f	f	d
e	e	e
e	c	e
b	f	e
c	b	f

d = 1

b	a	c
c	a	d
c	b	f
e	c	e
f	d	b
e	e	e
f	f	d
b	f	e

d = 0

b	a	c
b	f	e
c	a	d
c	b	f
e	c	e
e	e	e
f	d	b
f	f	d

Implementation

```
def radixsort(a:list[str]) -> None:
  n, W = len(a), len(a[0])
  aux = [0] * n

  for d in range(W-1, -1, -1):
    cnt = [0] * (256 + 1)

    for i in range(n):
      cnt[ord(a[i][d]) + 1] += 1

    for i in range(256):
      cnt[i+1] += cnt[i]

    for i in range(n):
      aux[cnt[ord(a[i][d])]] = a[i]
      cnt[ord(a[i][d])] += 1

    for i in range(n):
      a[i] = aux[i]
```

Testing it

```python
lst = ['eee', 'ece', 'cad', 'fdb', 'bac', \
       'ffd', 'bfe', 'cbf']
radixsort(lst)
assert is_sorted(lst) == True
print(lst)
```

```
['bac', 'bfe', 'cad', 'cbf', 'ece', 'eee', 'fdb', 'ffd']
```

Analysis

- Not in-place, must be stable
- String length · number of strings
  - $O(w \cdot n)$
- Linear for short strings
- Can be effective for sorting, e.g., “personnummer” (strings with 12 digits)

Reading instructions

Reading instructions

- Ch. 7.1 - 7.11