Algorithms and Data Structures

Algorithm Analysis (Ch. 2)

Morgan Ericsson

Today

- Introduction to algorithm analysis
  - Growth
  - Algorithm steps
  - O (big Oh / Omicron) notation
- Analysis

Growth

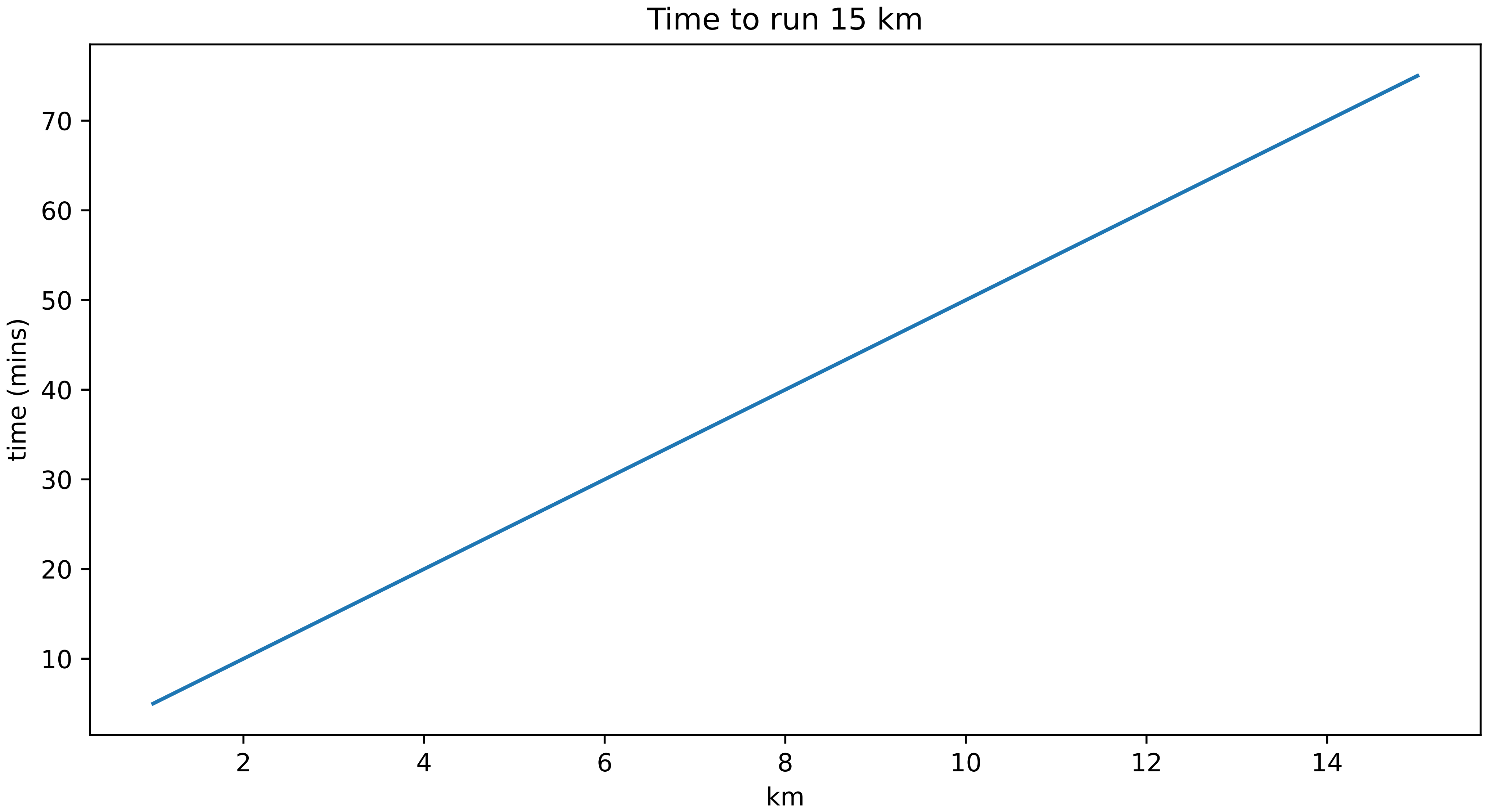
Getting familiar with growth

- Assume you are planning to run 15 km.
- You know you can run a km in at most 5 minutes.
- How long will it take you?
  - Easy, at most $5 \times 15 = 75$ mins.

We can plot it

```
import matplotlib.pyplot as plt
import numpy as np

tpkm = 5
dist = np.arange(1, 16)

plt.plot(dist, dist * tpkm)
plt.title('Time to run 15 km')
plt.xlabel('km')
plt.ylabel('time (mins)')
plt.show()
```

We can plot it



Linear growth

- A dependent quantity (variable) changes at a constant rate with respect to an independent variable
  - In our example, time depends on the independent variable distance
- Linear function or linear model
- Can be expressed as $y = mx + b$
  - In our example, $y = 5x$

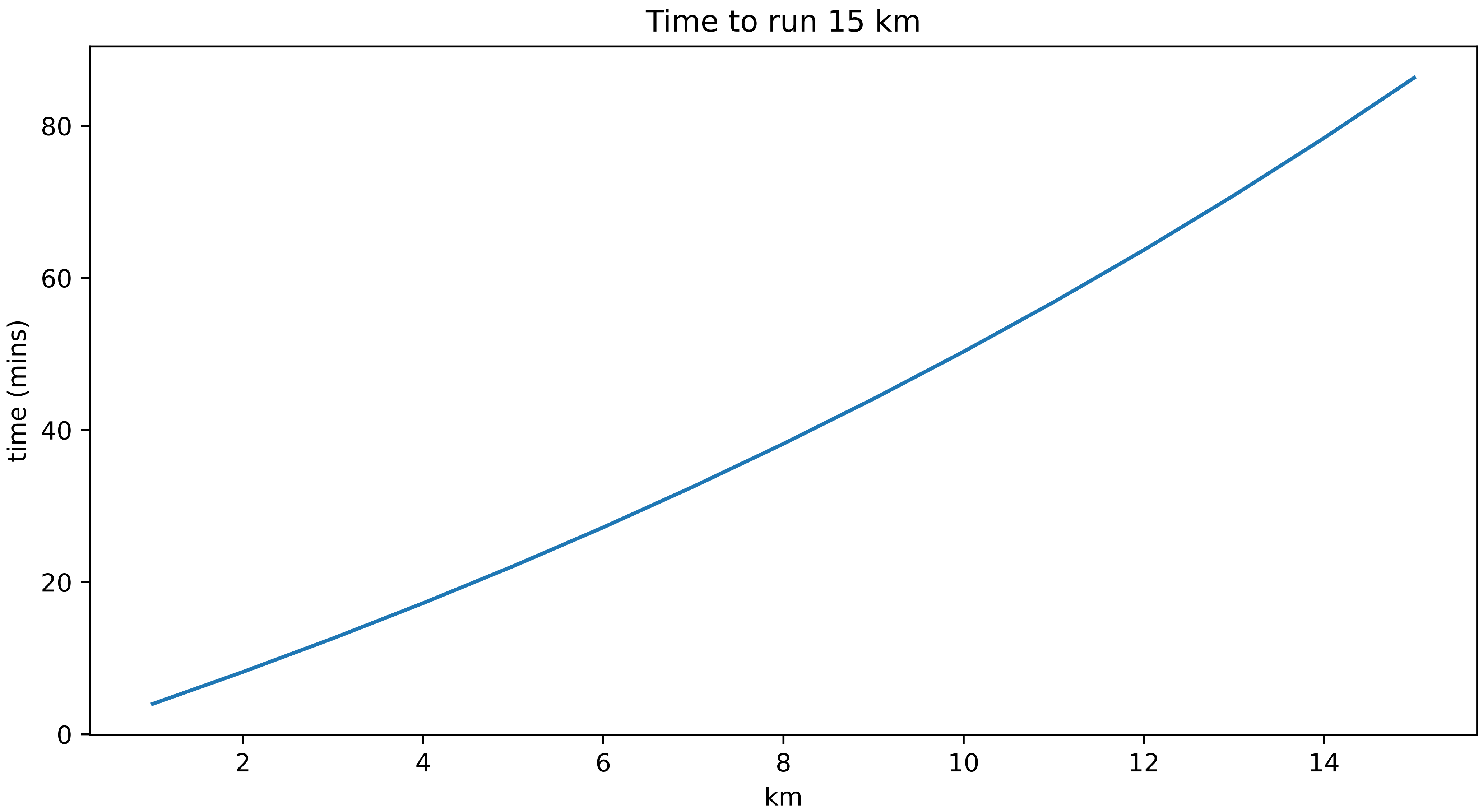
Explaining $y = 5x$

- In $y = mx + b$
  - m is the slope, and
  - b is the y-intercept, e.g., the offset in y when x is 0
- We take two points from our example, e.g., $(0, 0)$ and $(2, 10)$
- $m = \frac{10-0}{2-0} = \frac{10}{2} = 5$
- b is 0, since y is 0 when x is 0.
- So, $y = mx + b = 5x + 0 = 5x$

Going faster?

- Assume that you can run a bit faster, but if you do, you get tired and slow down
  - If you run at a pace of 4 minutes per km,
  - each km takes 5% longer
- So, if the first km takes 4 minutes, the second will take 4 minutes and 12 seconds.

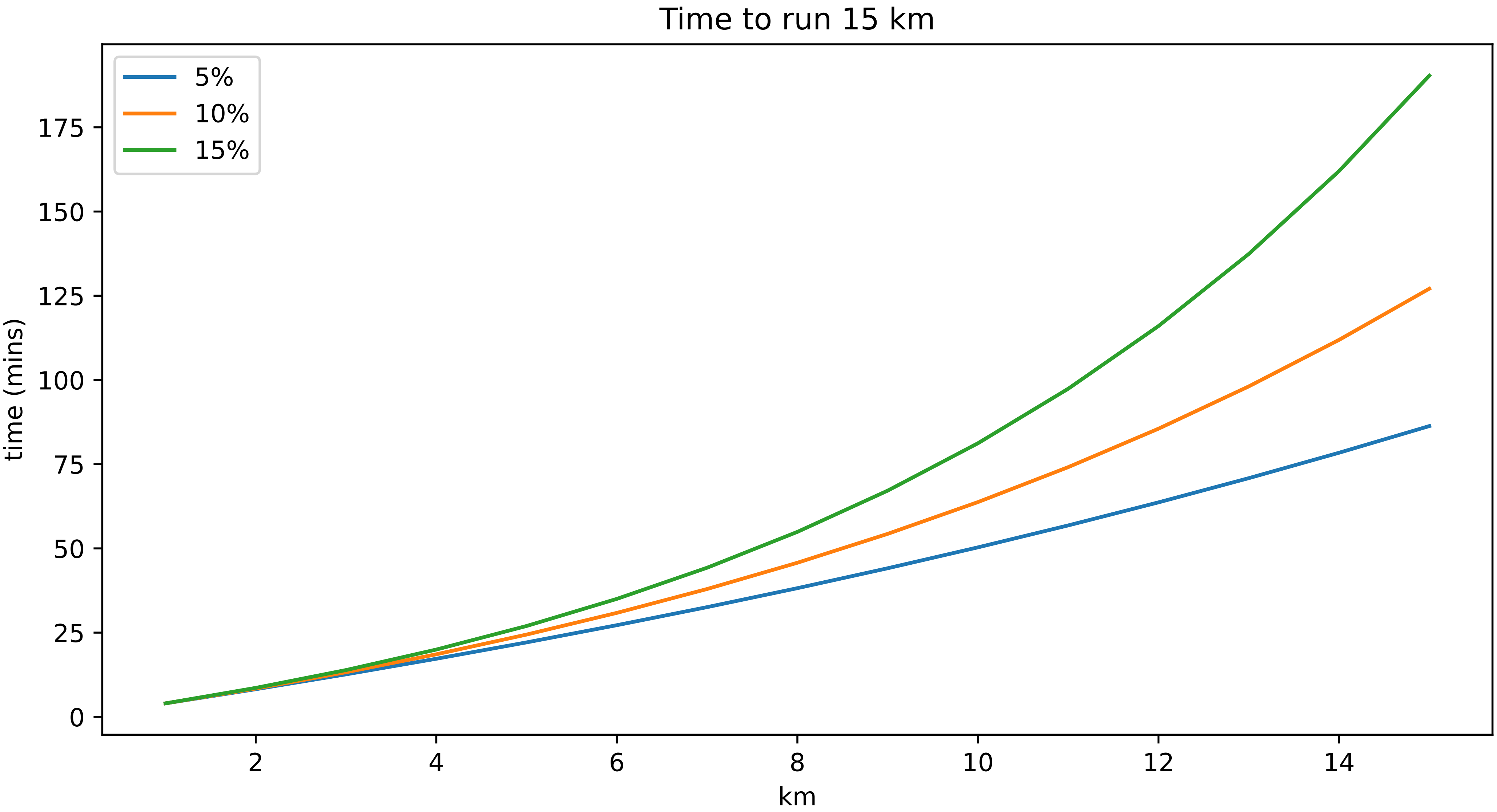
We can plot it



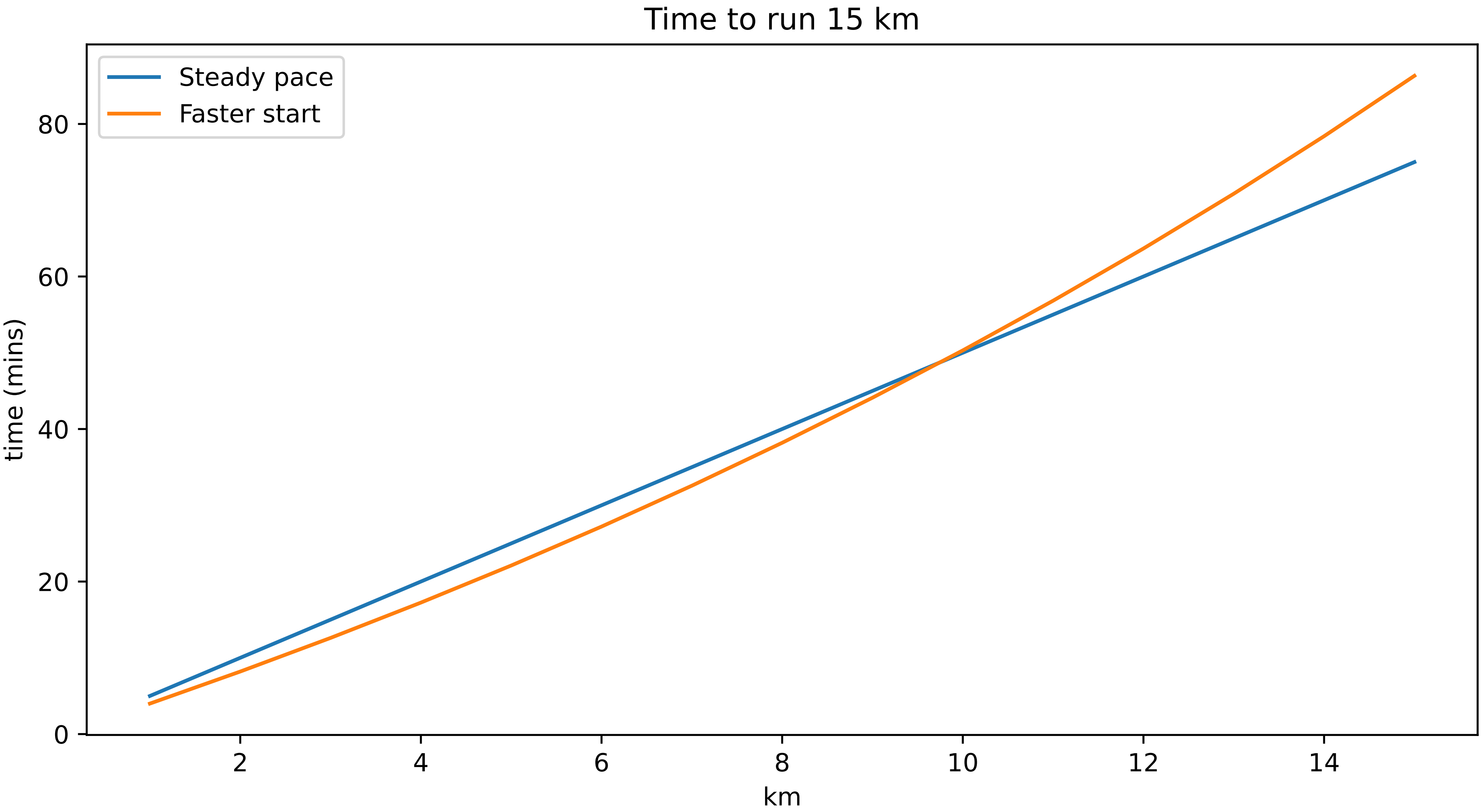
No longer linear

- The time no longer change by a fixed quantity
- The second km took 4 minutes 12 seconds and the last took almost 8 minutes.
- An exponential relationship …
  - $y = \text{speed}_0(\text{increase})^x$
  - $y = 4(1.05)^x$
- … describes the pace
  - $4(1.05)^0 = 4$
  - $4(1.05)^{14} \approx 7.92$

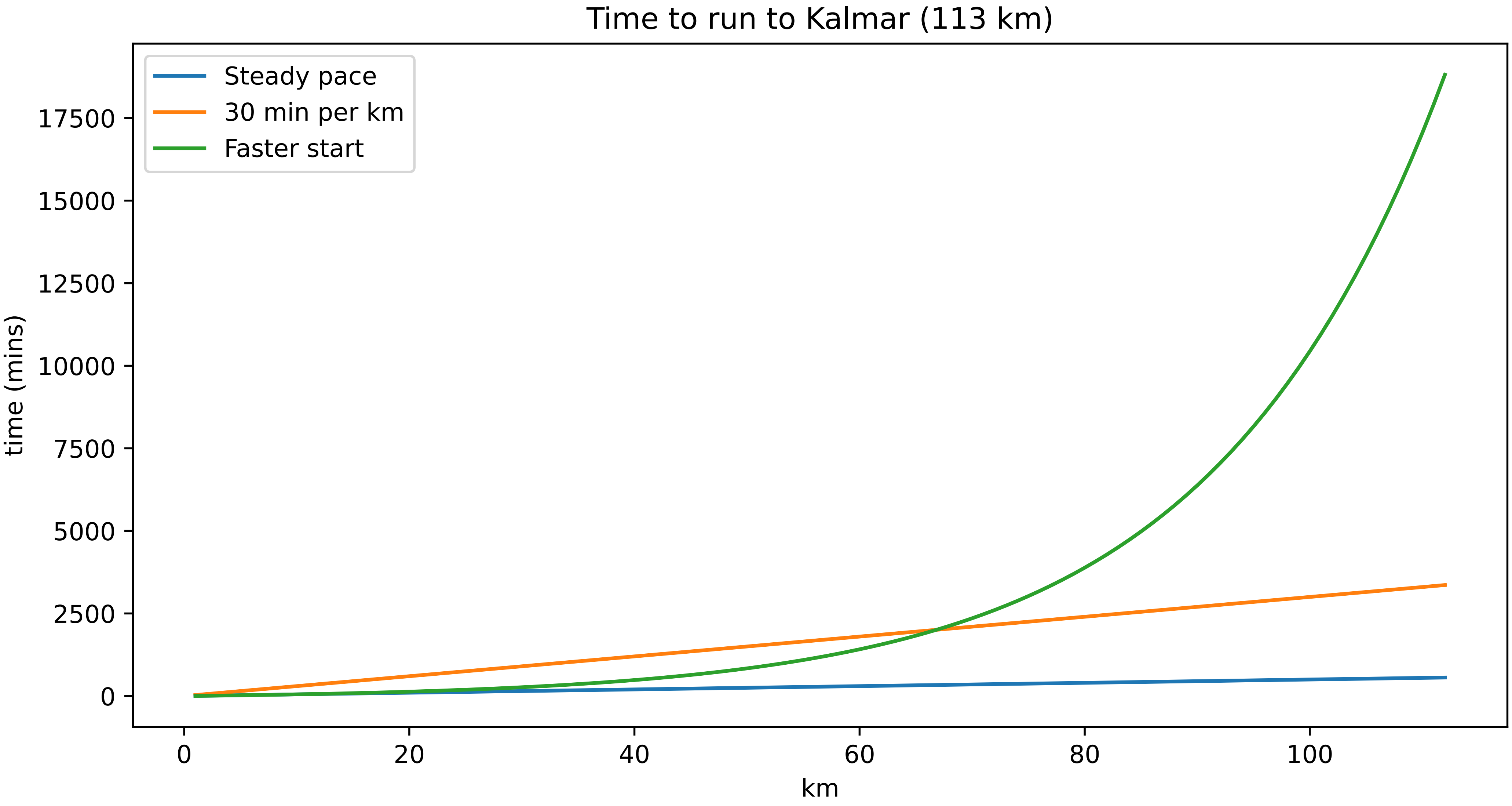
Different slowdown rates?



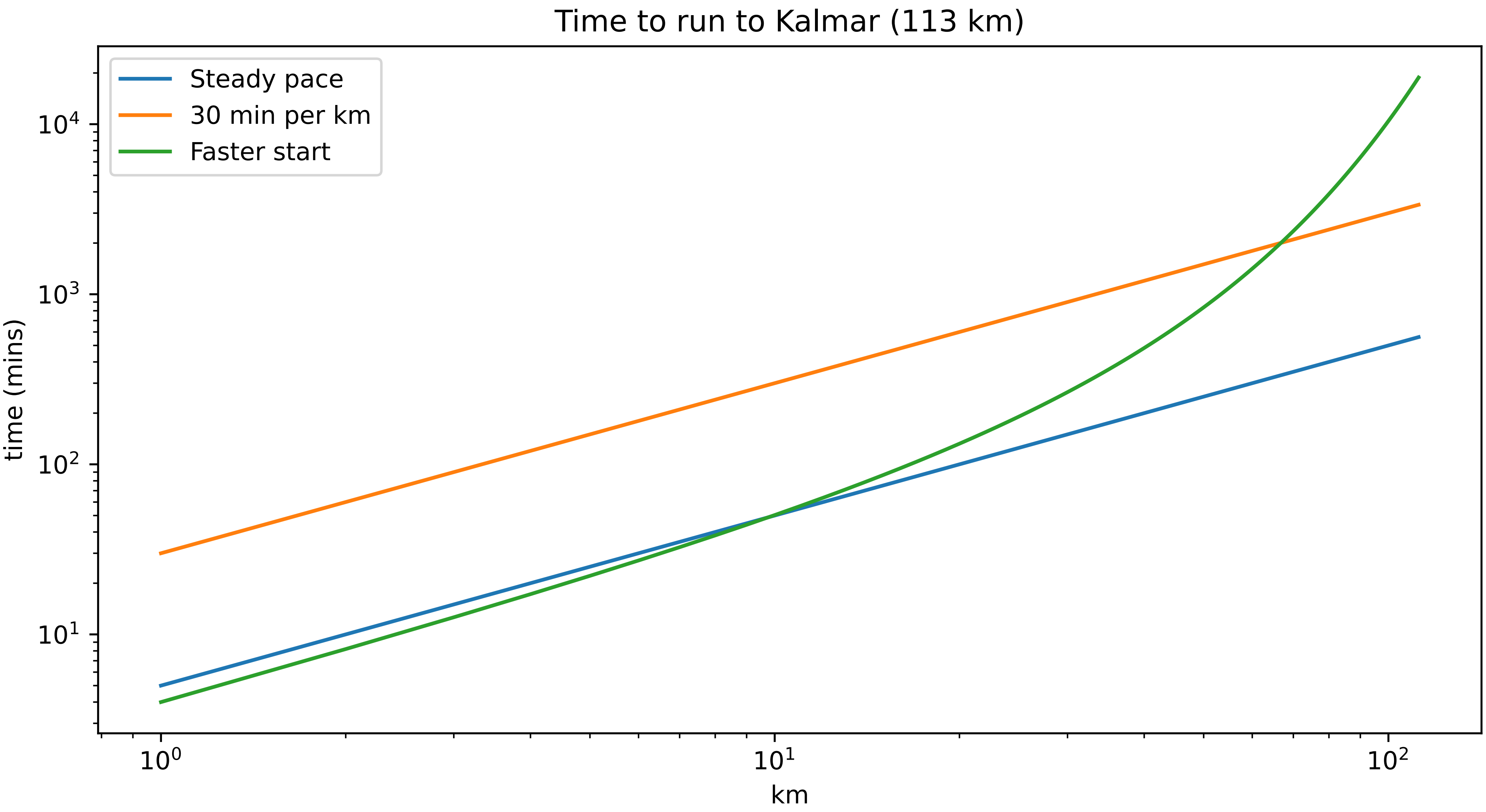
Faster?


Time to run 15 km
Steady pace
Faster start
time (mins)
0
20
40
60
80
2
4
6
8
10
12
14
km

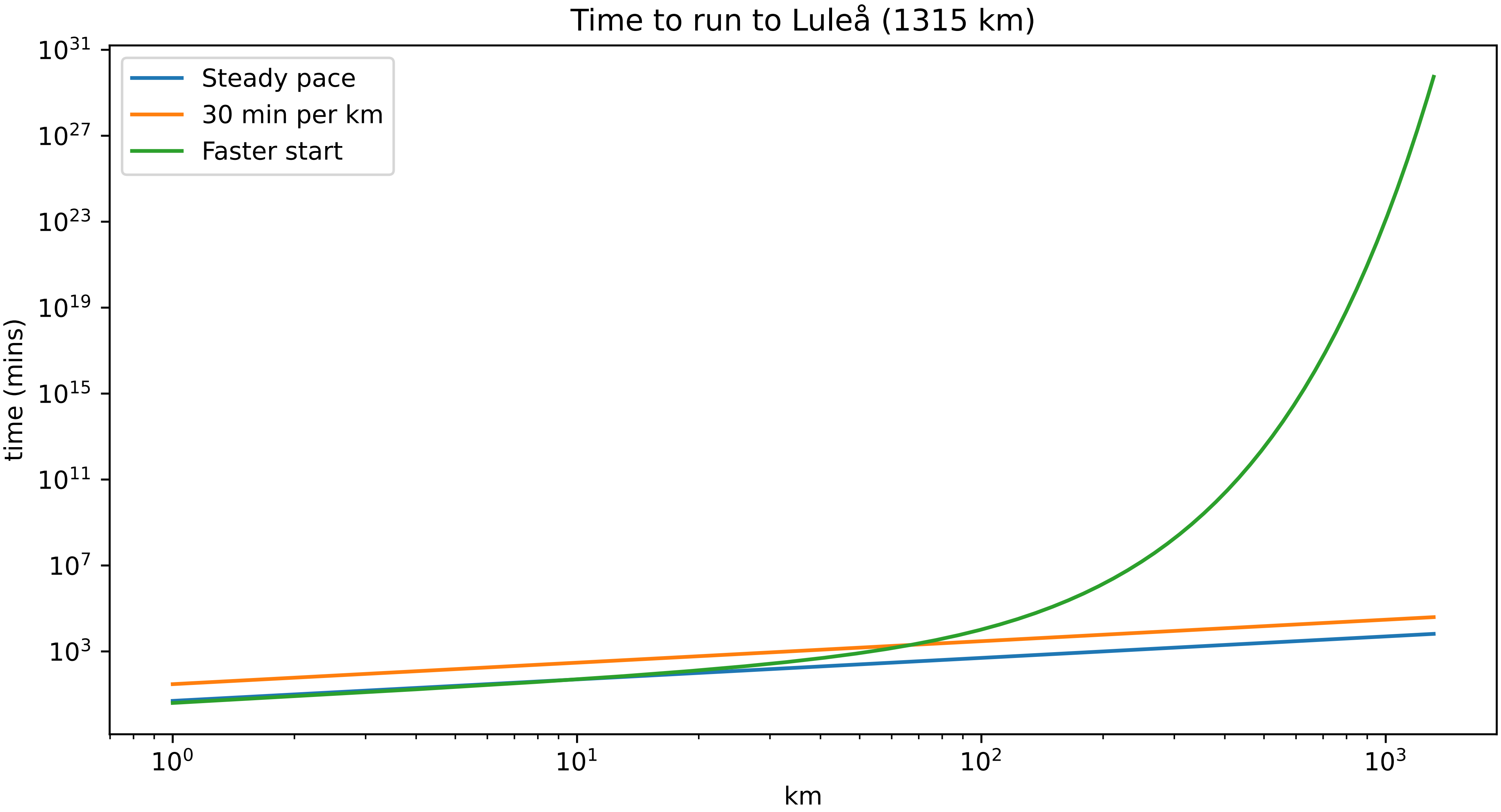
To infinty (and beyond)



To infinty (and beyond)



To infinty (and beyond)



Growth of algorithms

Computing the slope

```
xy0 = (0,0)
xy2 = (2,10)
tx = xy2[0] - xy0[0]
ty = xy2[1] - xy0[1]
m = ty / tx
```

> How many instructions are required to complete the program above?

How many instructions?

- Consider `a = b + c`
  1. Fetch a
  2. Fetch b
  3. Add a and b
  4. Store the result in c
- So, 4 instructions

Computing the slope

```
xy0 = (0,0)           # 2 instructions
xy2 = (2,10)          # 2 instructions
tx = xy2[0] - xy0[0]  # 4 instructions
ty = xy2[1] - xy0[1]  # 4 instructions
m = ty / tx           # 4 instructions
```

Using our simple model, the program has 16 instructions.

What is the growth?

- Assume we want to model like we did in the running example
  - How does the program grow?
- No growth, the program always requires 16 instructions
  - Constant

Refactor to function and check

```python
def slope(c1, c2):
  x0,y0 = c1
  x1,y1 = c2

  return (y1 - y0) / (x1 - x0)

slope((0, 0), (2, 10))
```

```
5.0
```

Timing things in python

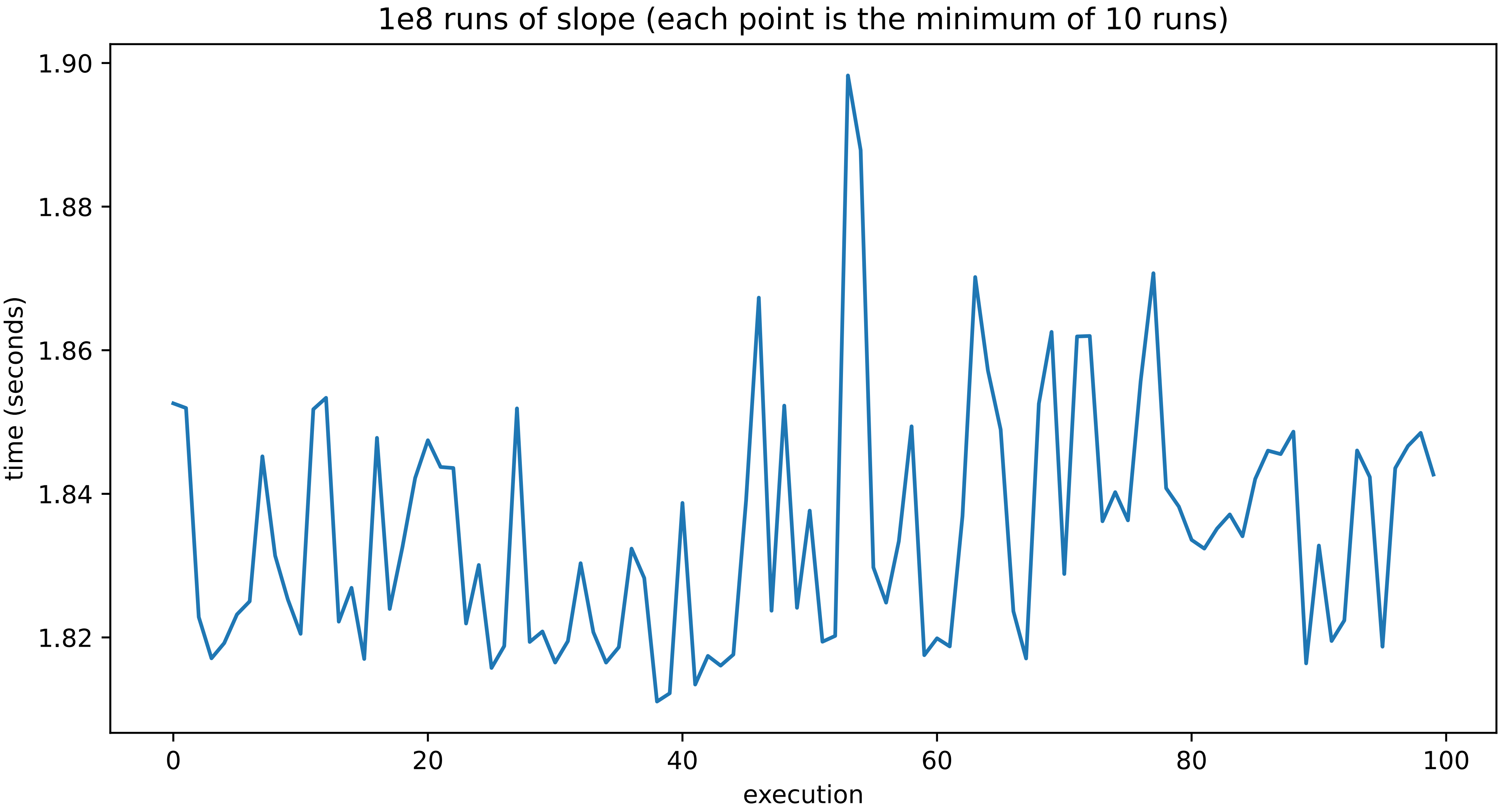
- Use the `timeit` module
- Or the `%timeit` magic command in ipython/jupyter
  - `%timeit slope((0, 0), (2, 10))`

Timing a single run of slope

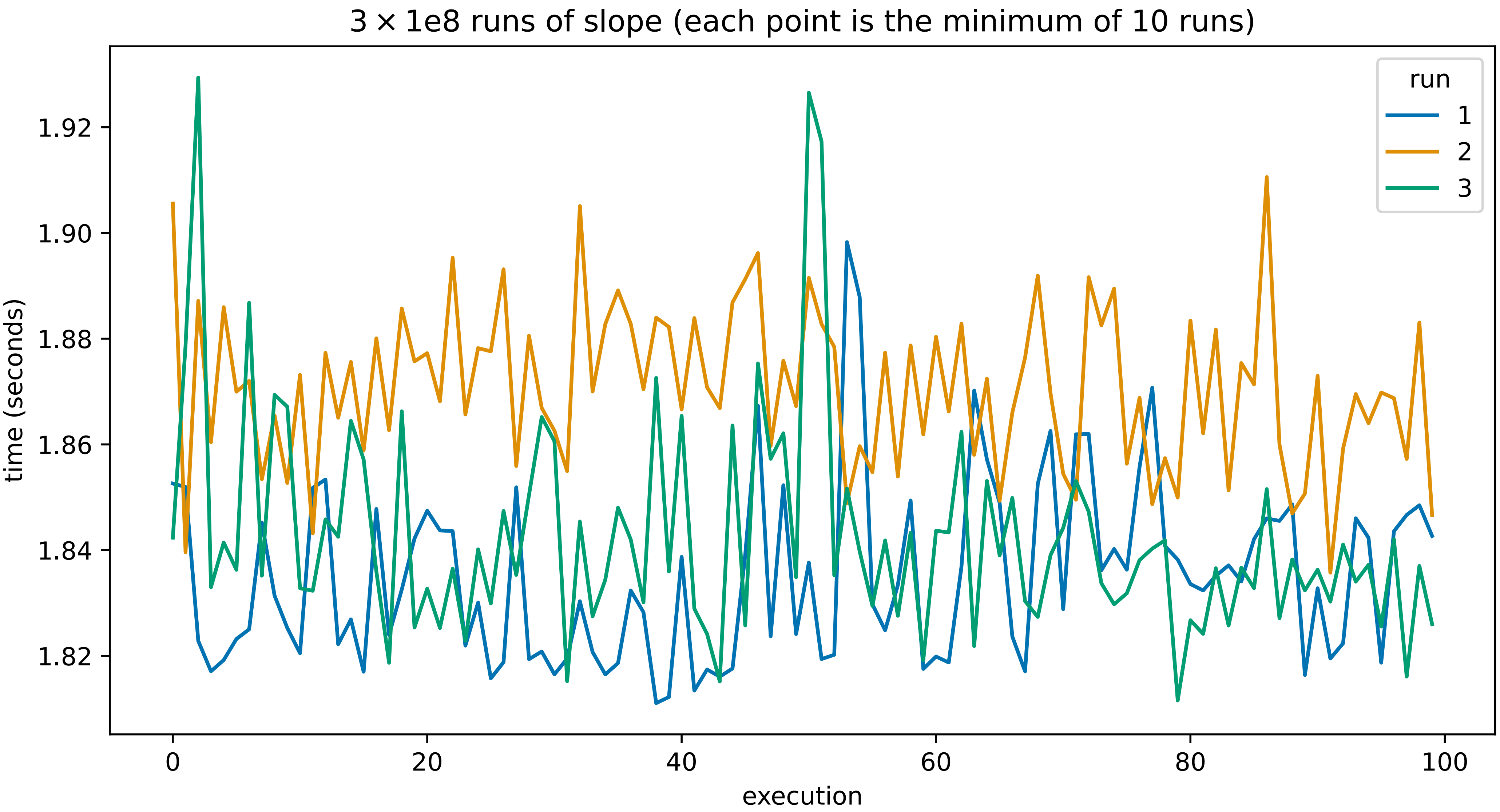
```
%timeit slope((0, 0), (2, 10))
```

```
74 ns ± 1.06 ns per loop (mean ± std. dev. of 7 runs,
10,000,000 loops each)
```

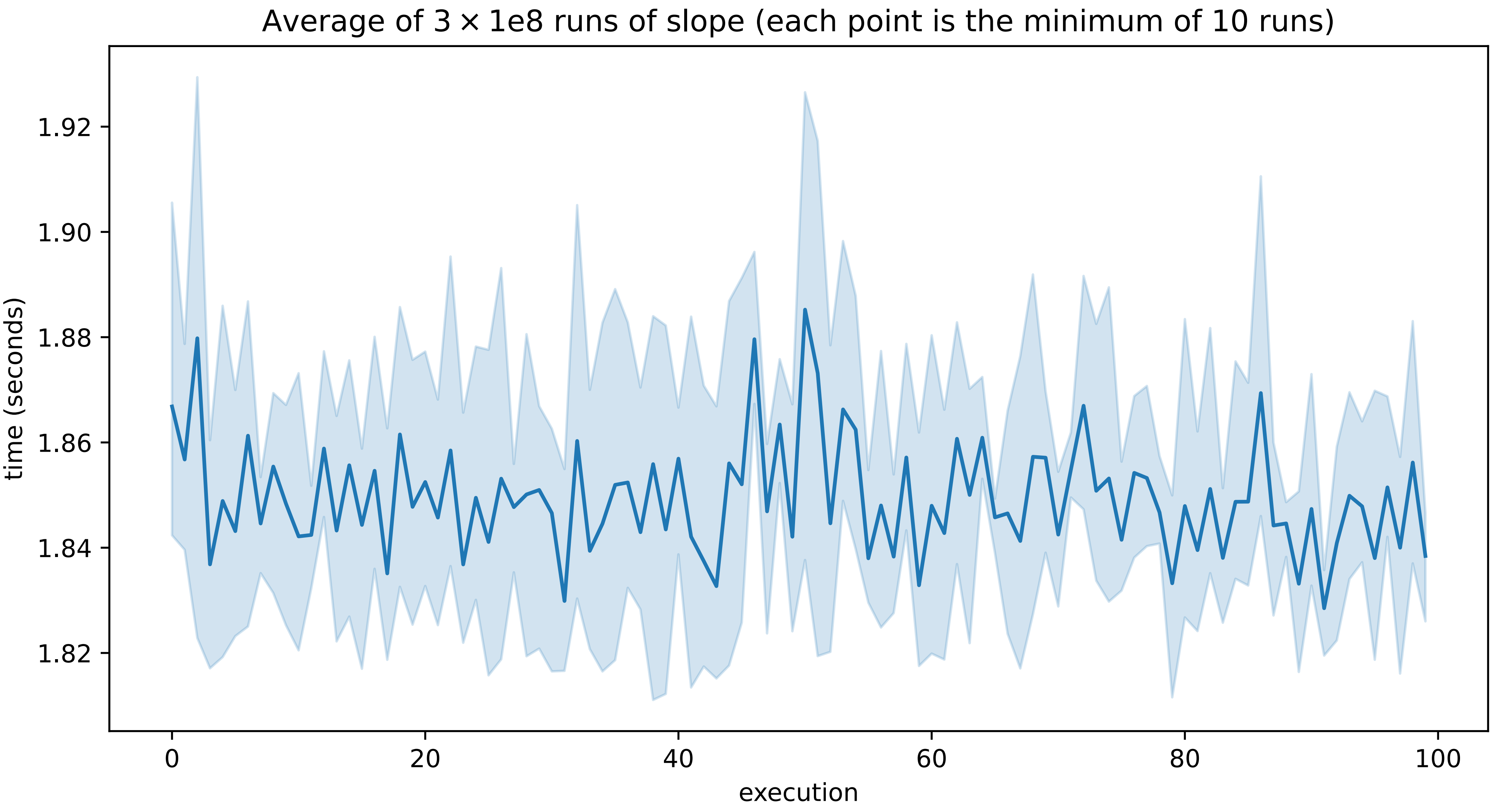
Is slope constant?



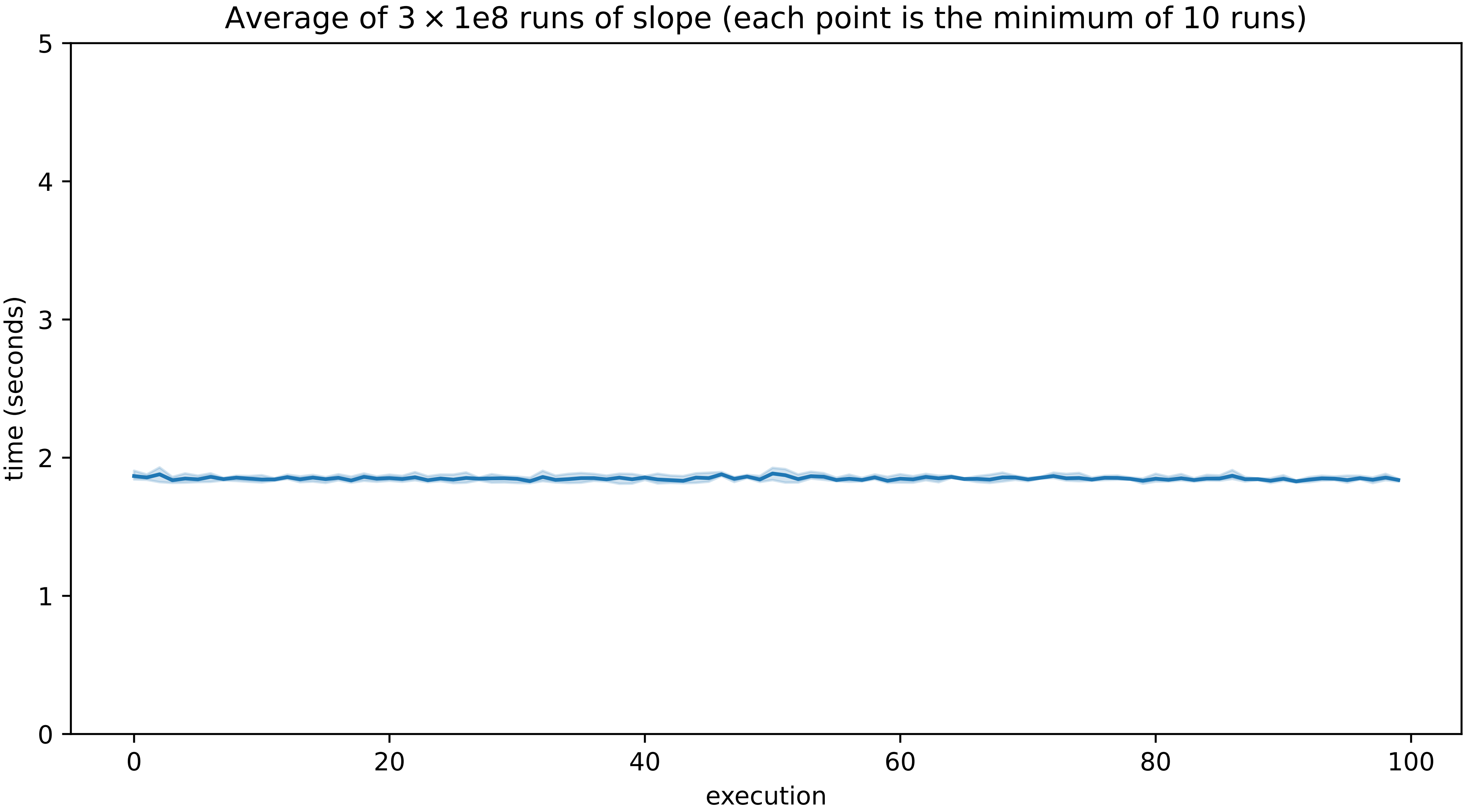
More runs?


3 × 1e8 runs of slope (each point is the minimum of 10 runs)
time (seconds)
1.82
1.84
1.86
1.88
1.90
1.92
execution
0
20
40
60
80
100
run
1
2
3

More runs?


Average of 3 × 1e8 runs of slope (each point is the minimum of 10 runs)
time (seconds)
1.92
1.90
1.88
1.86
1.84
1.82
0
20
40
60
80
100
execution

If we zoom out?



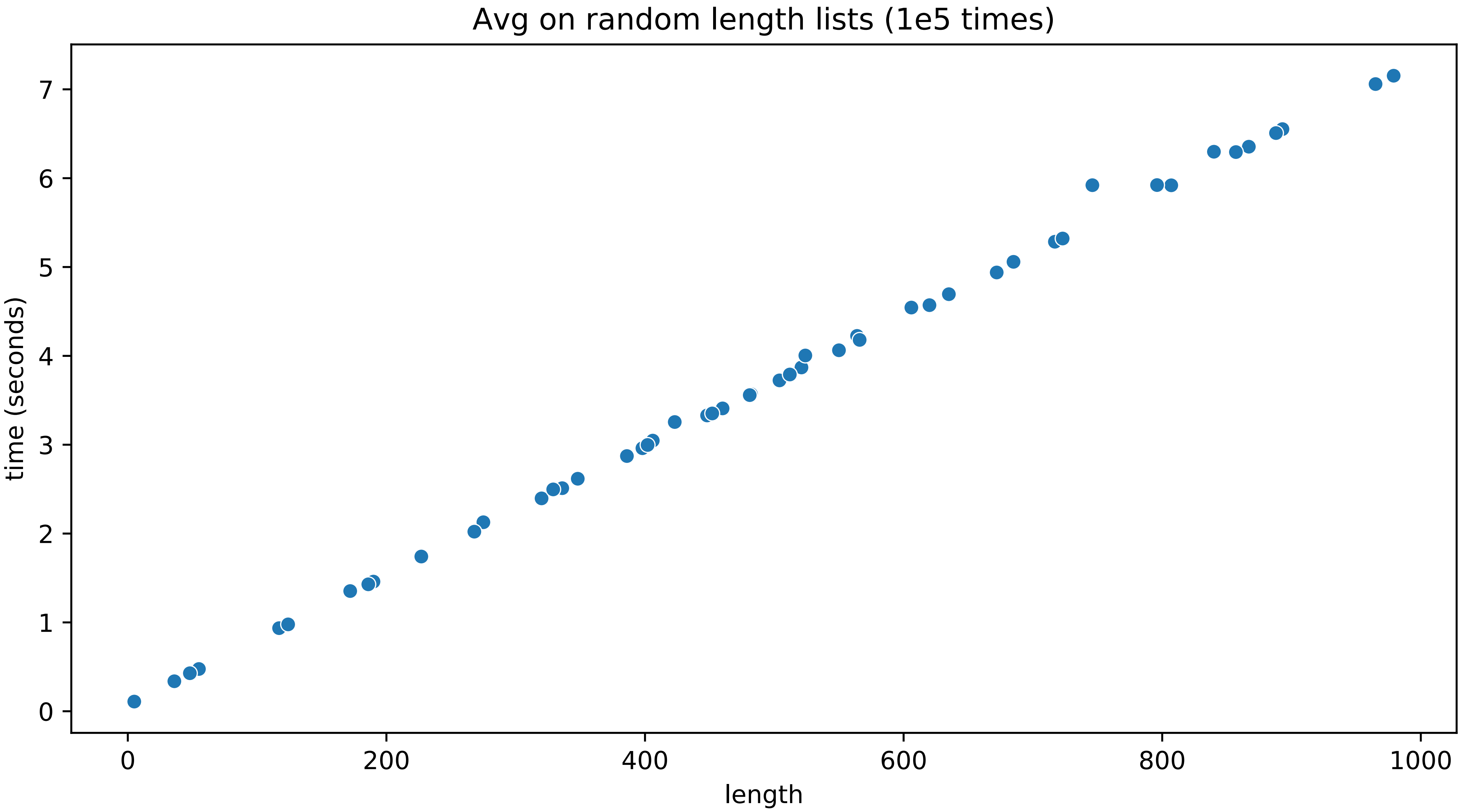
Computing the average

- Assume we want to compute the average of two numbers:
  - `def avg(a:int, b:int) -> float`
- We can deduce that `avg` has a constant runtime
- What about if we want the average of a list of numbers?
  - `def avg(l:list[int]) -> float`
- Also constant? If not, what controls the growth?

Let’s try it

```python
def avg(l:list[int]) -> float:
  s = 0
  for v in l:
    s += v

  return s/len(l)

assert avg([1,2,3,4]) == 2.5
```

How long?


Avg on random length lists (1e5 times)
time (seconds)
0
1
2
3
4
5
6
7
length
0
200
400
600
800
1000

What is the slope?

n	exctime
5	0.10884
36	0.337602
48	0.428082
55	0.475625
117	0.935349
124	0.977824
172	1.35286

$$\frac{0.43 - 0.11}{48 - 5} = \frac{0.32}{43} \approx 0.007$$

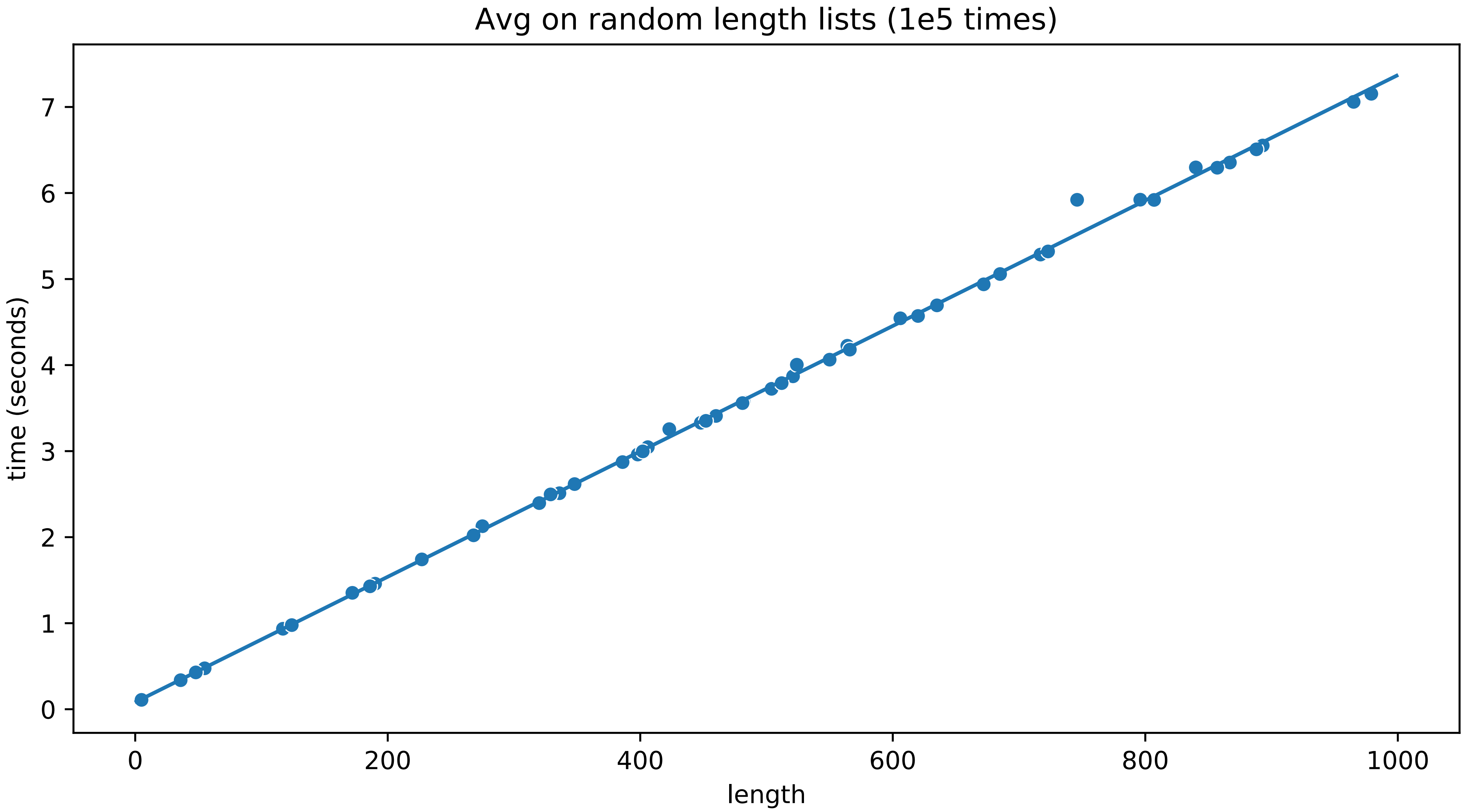
$$\frac{1.35 - 0.48}{172 - 55} = \frac{0.87}{117} \approx 0.007$$

Better tools

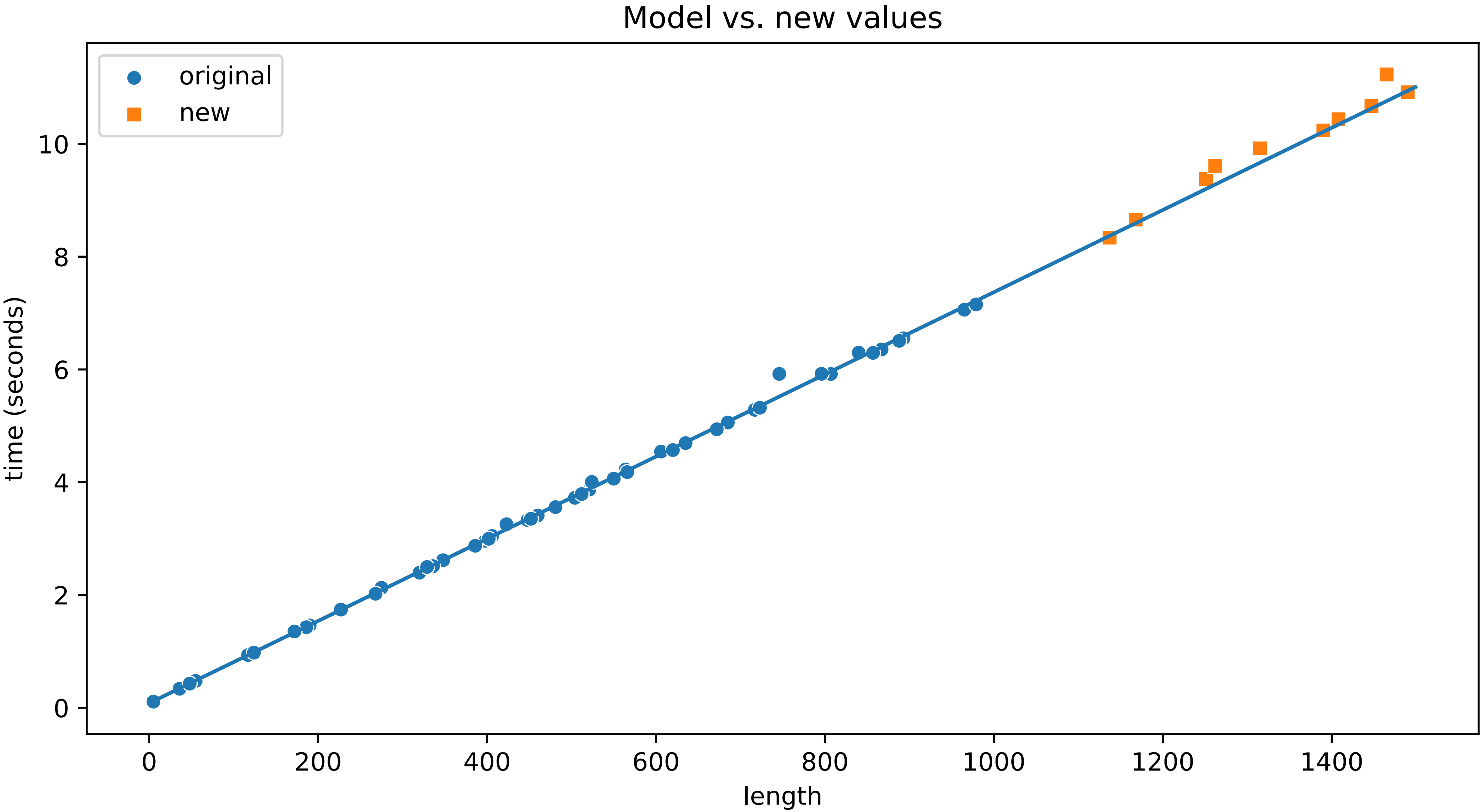
```python
from scipy.optimize import curve_fit

def linmod(x, m, b):
  return m * x + b

[(m, b), _] = curve_fit(linmod, df_avg_t.n,
                                 df_avg_t.exctime)
print(f'Model: {m:.5f} * x + {b:.5f}')
```

```
Model: 0.00729 * x + 0.08164
```

Prediction vs empirical data



Prediction vs empirical data



Prediction vs empirical data

n	exctime	pred	diff
1315	9.921	9.667	0.255
1465	11.232	10.760	0.472
1251	9.377	9.200	0.177
1262	9.612	9.280	0.332
1390	10.239	10.213	0.025

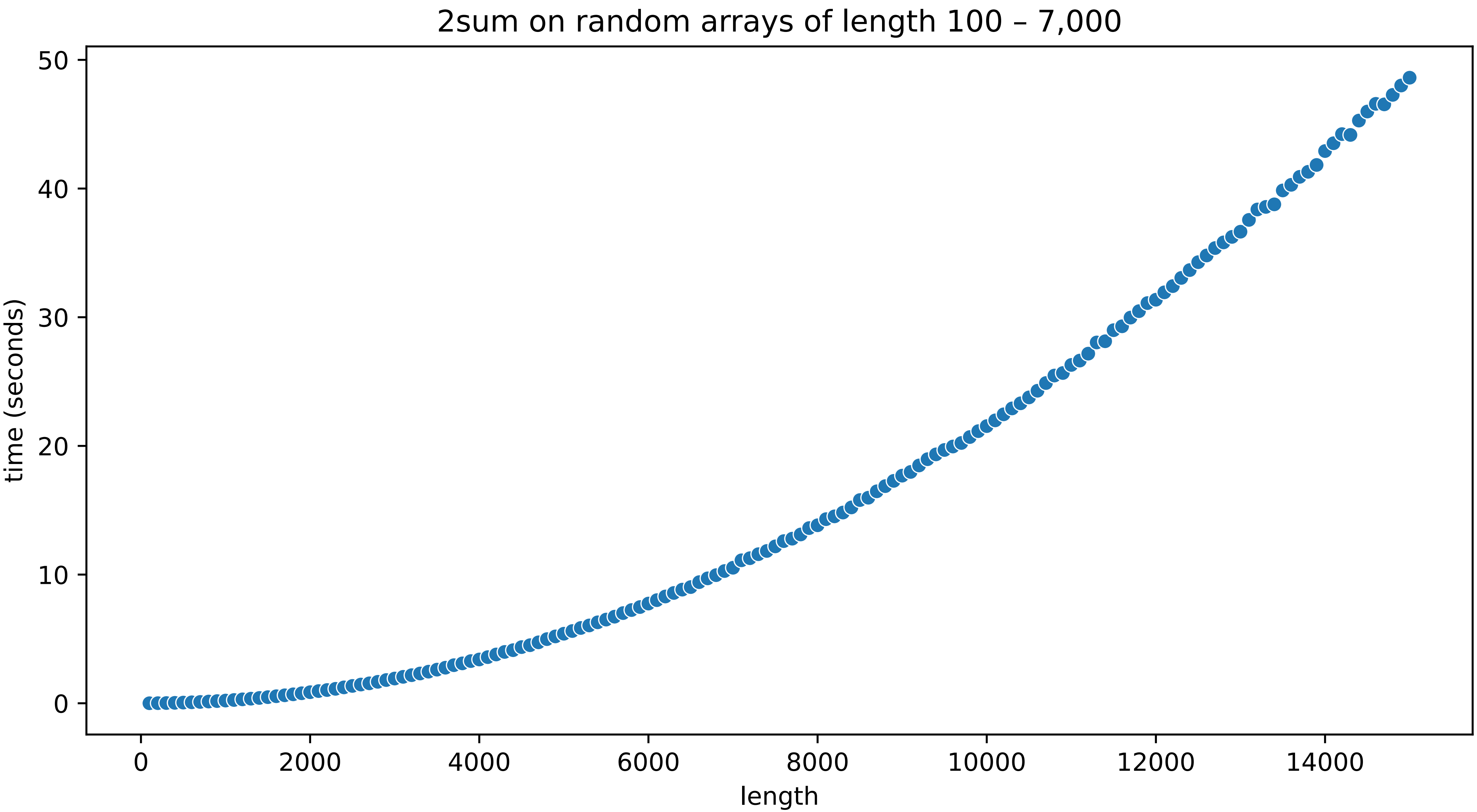
k-sum

- Assume we have a list of n integers
- How many sequences of k numbers in this list sum to 0?
- Given the list `[-2, 0, 2, 1]`:
  - If $k = 2$, `(-2, 2)` sum to 0
  - If $k = 3$, `(-2, 0, 2)` sum to 0

k-sum when k=2 (2sum)

```python
def twosum(l:list[int]) -> list[tuple[int, int]]:
  res = []
  for i, vi in enumerate(l):
    for j, vj in enumerate(l):
      if i == j:
        continue
      if vi + vj == 0:
        res.append((vi, vj))

  return res
```

How does 2sum grow?



How does 2sum grow?

sz	time
100	0.002
200	0.008
400	0.033
800	0.134
1600	0.547
3200	2.182
6400	8.830

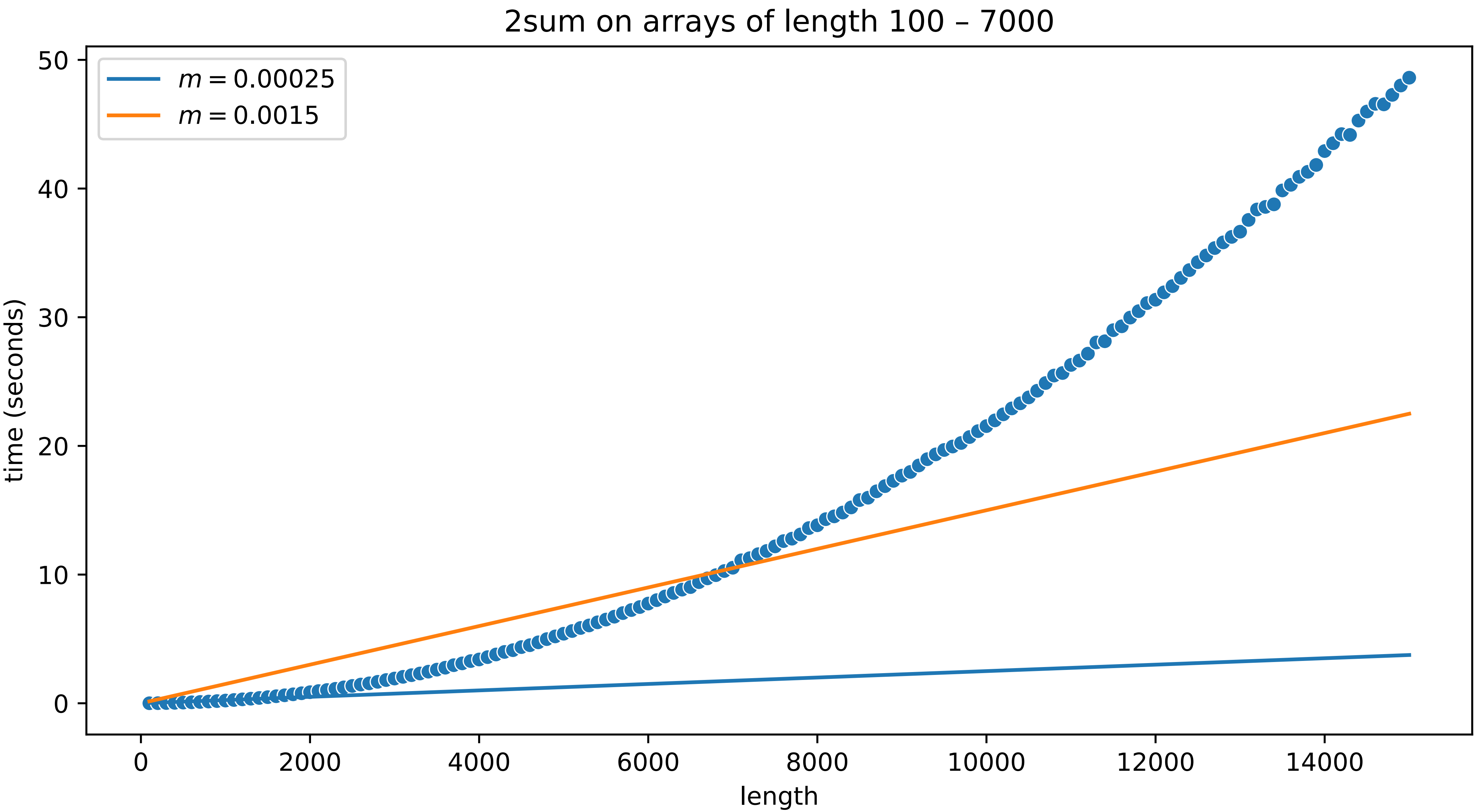
What is the slope?

sz	time
100	0.002
200	0.008
400	0.033
800	0.134
1600	0.547
3200	2.182
6400	8.830

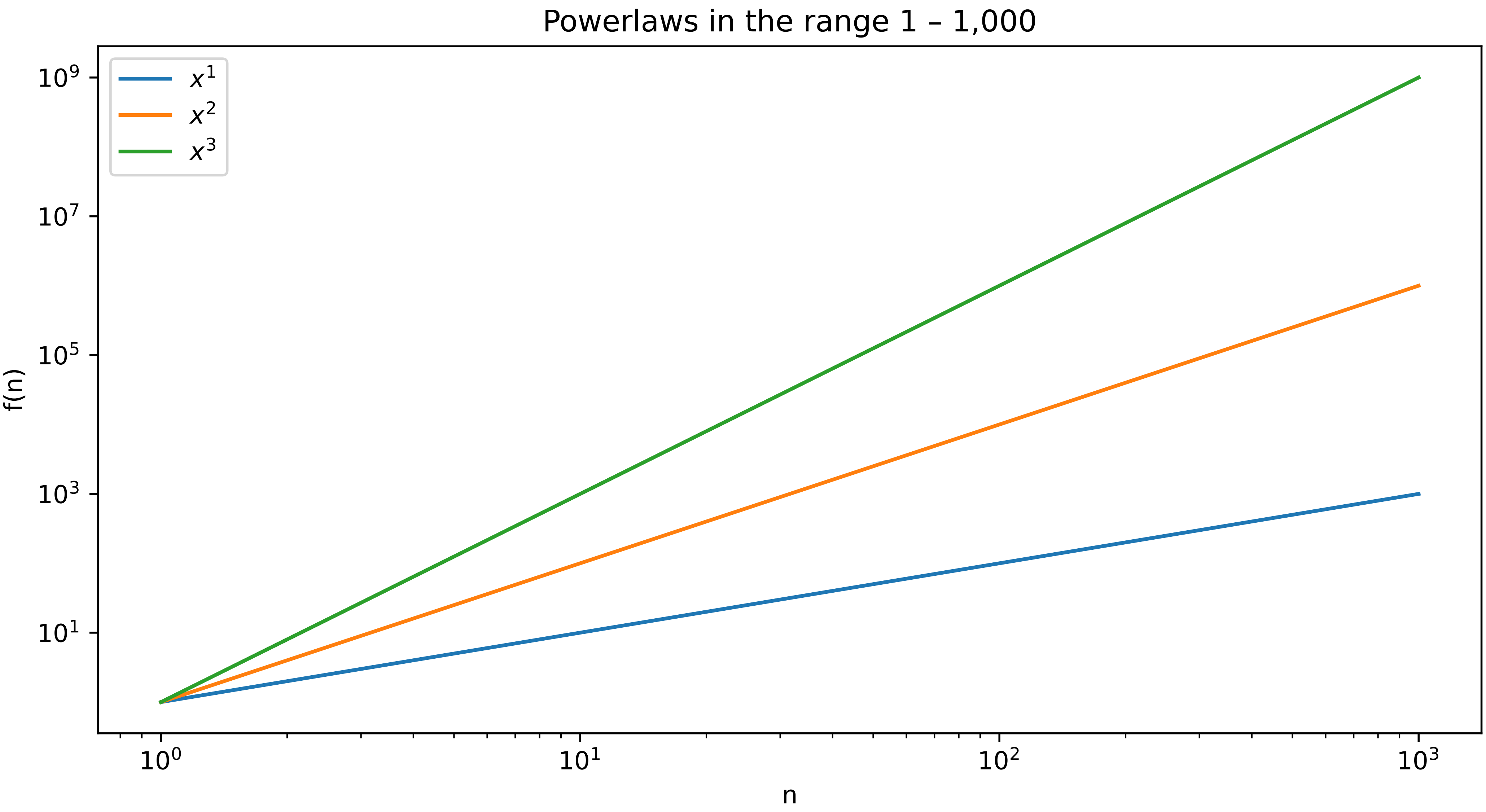
$$\frac{0.134 - 0.033}{800 - 400} \approx 0.00025$$

$$\frac{8.830 - 0.033}{6400 - 400} \approx 0.0015$$

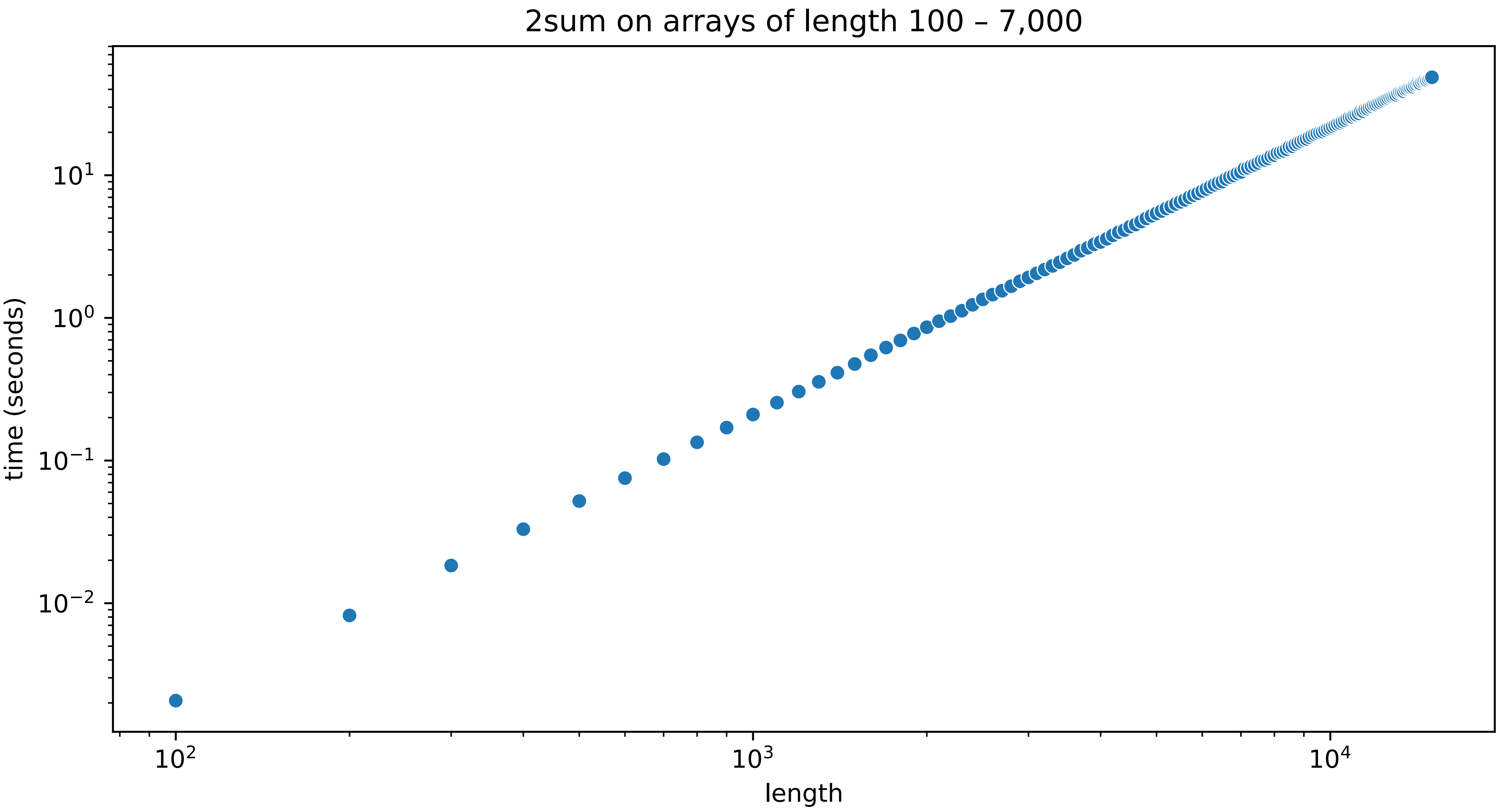
Trying the slopes



Powerlaws and log-log



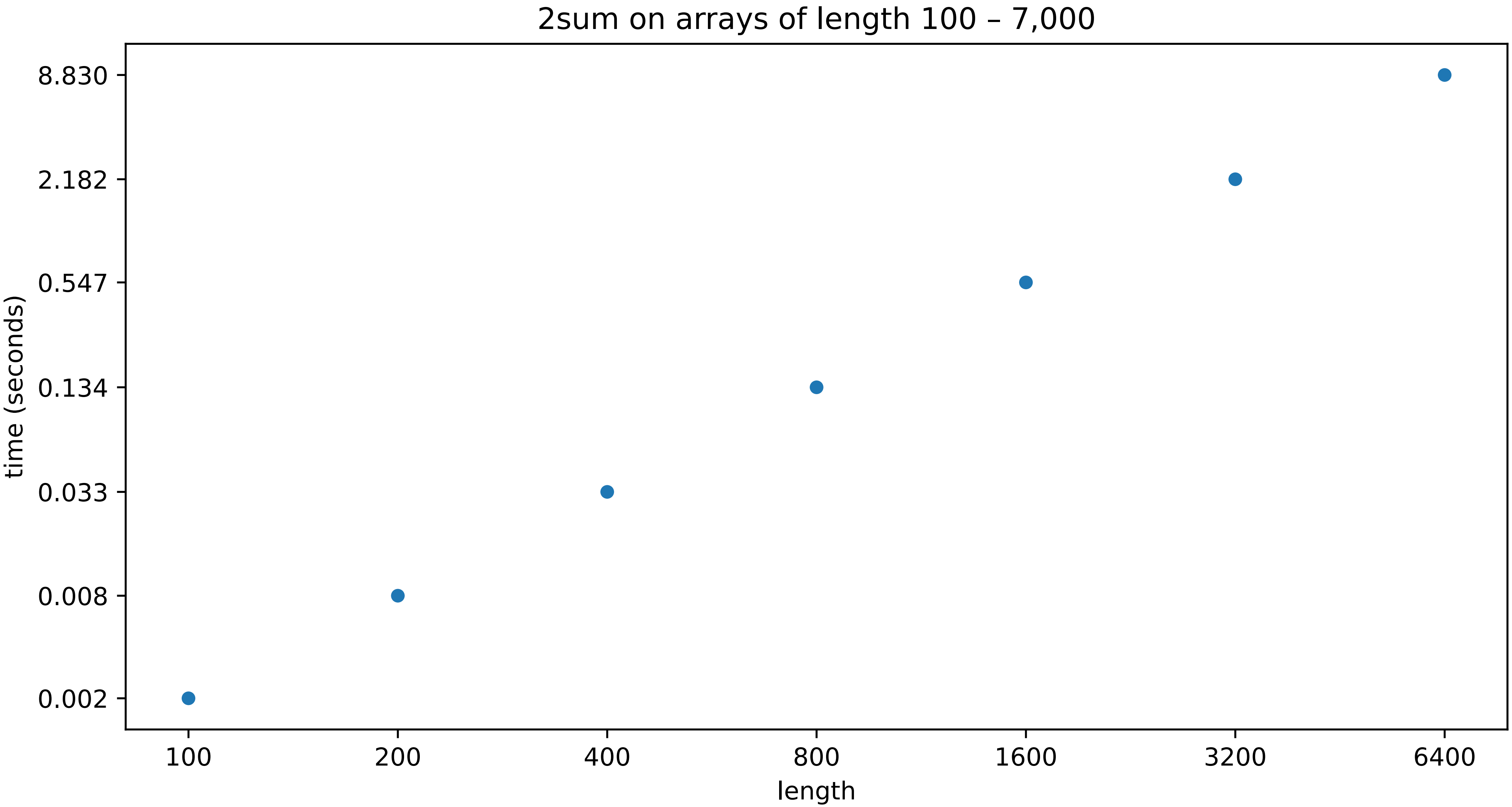
Log-log plot (log10)



Looking at the growth

sz	time	ratio	log2 ratio
100	0.002	0.000	-inf
200	0.008	3.956	1.984
400	0.033	4.022	2.008
800	0.134	4.064	2.023
1600	0.547	4.079	2.028
3200	2.182	3.986	1.995
6400	8.830	4.046	2.017

Log2-log2


2sum on arrays of length 100 – 7,000
time (seconds)
8.830
2.182
0.547
0.134
0.033
0.008
0.002
100
200
400
800
1600
3200
6400
length

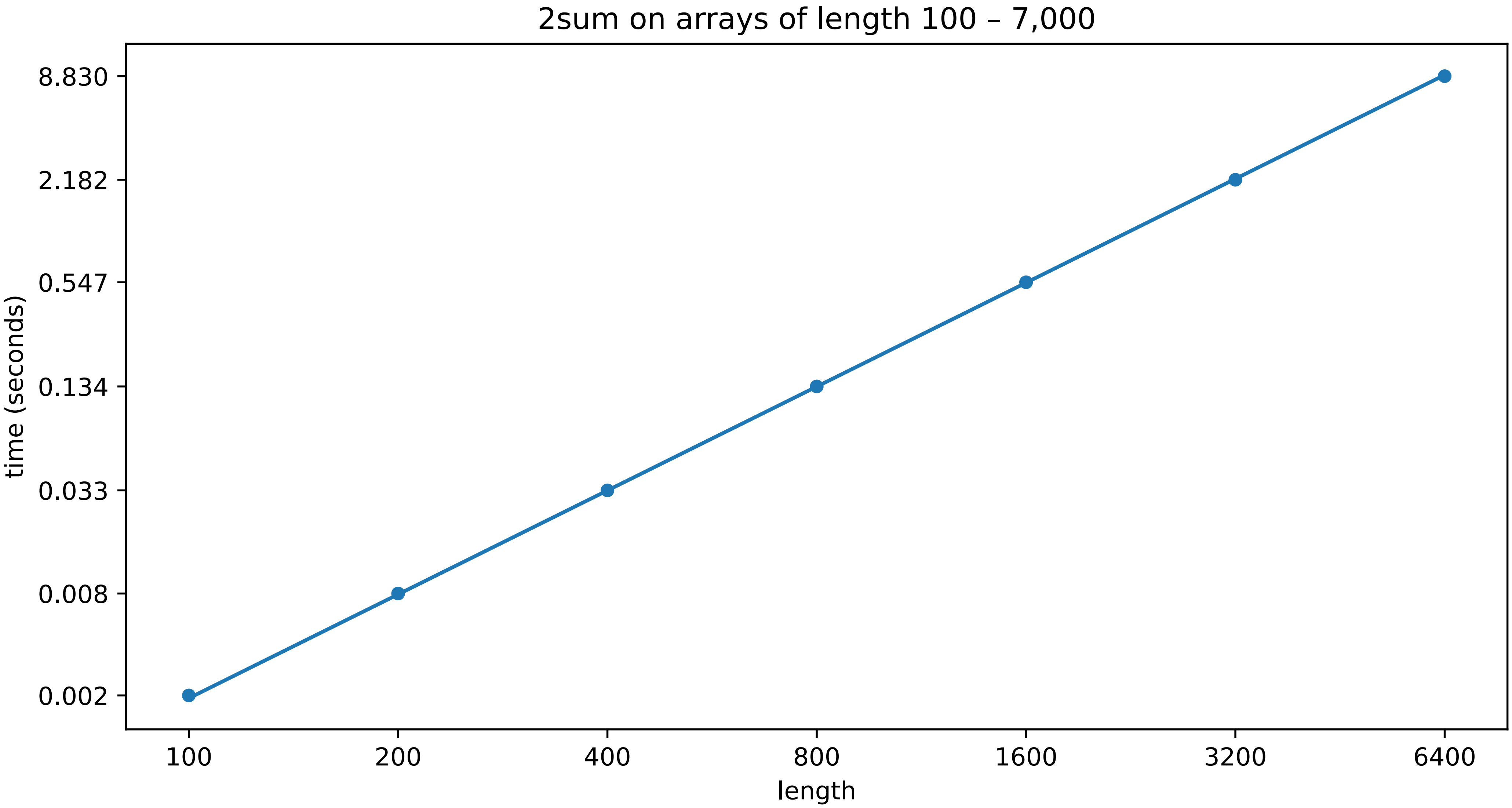
Growth

- We can use the slope and intercept to determine the growth:
  - $\log_2 \text{time}(x) = b \cdot \log_2 x + c$
- We compute the slope (b) as previously, but use the log2:
  - $b = \frac{\log_2 y_1 - \log_2 y_0}{\log_2 x_1 - \log_2 x_0}$
- And we can then determine c:
  - $c = \log_2 \text{time}(x) - b \cdot \log_2 x$

Example from 2sum

- $\frac{\log_2 0.134 - \log_2 0.033}{\log_2 800 - \log_2 400} \approx 2.022$
- $c = \log_2 0.134 - 2.022 \cdot \log_2 800 \approx -22.4$
- so
  - $\log_2 \text{time}(x) = 2.022 \cdot \log_2 x - 22.4$
- or
  - $2.022 \cdot \log_2 3200 - 22.4 \approx 1.14$
  - $\log_2 3200 \approx 1.13$

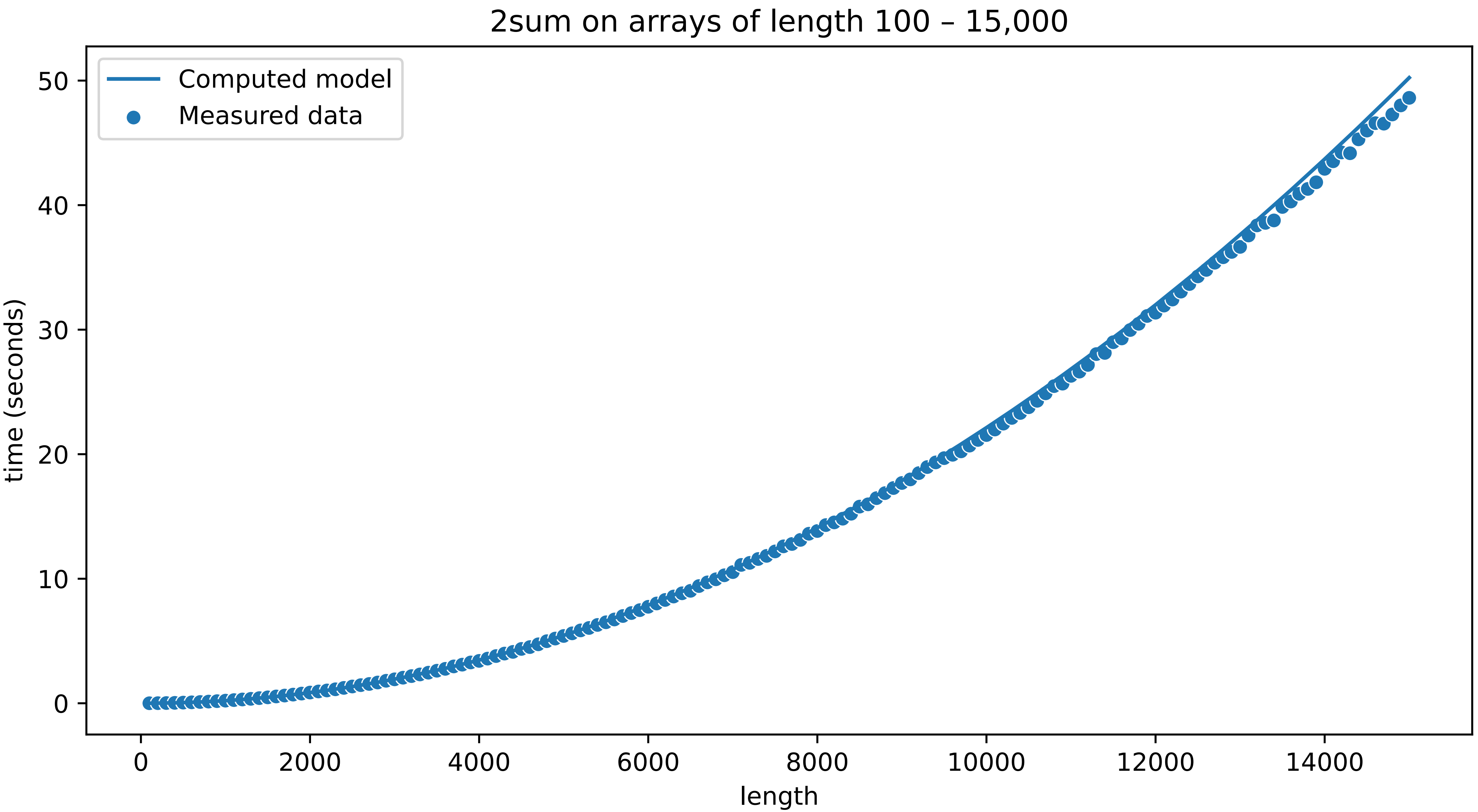
Trying it out



Moving to a powerlaw

- $a \cdot x^b$
- $a = 2^c$
- So, about $2^{-22.4} \cdot x^{2.022}$

Plotting


2sum on arrays of length 100 – 15,000
Computed model
Measured data
time (seconds)
0
10
20
30
40
50
length
0
2000
4000
6000
8000
10000
12000
14000

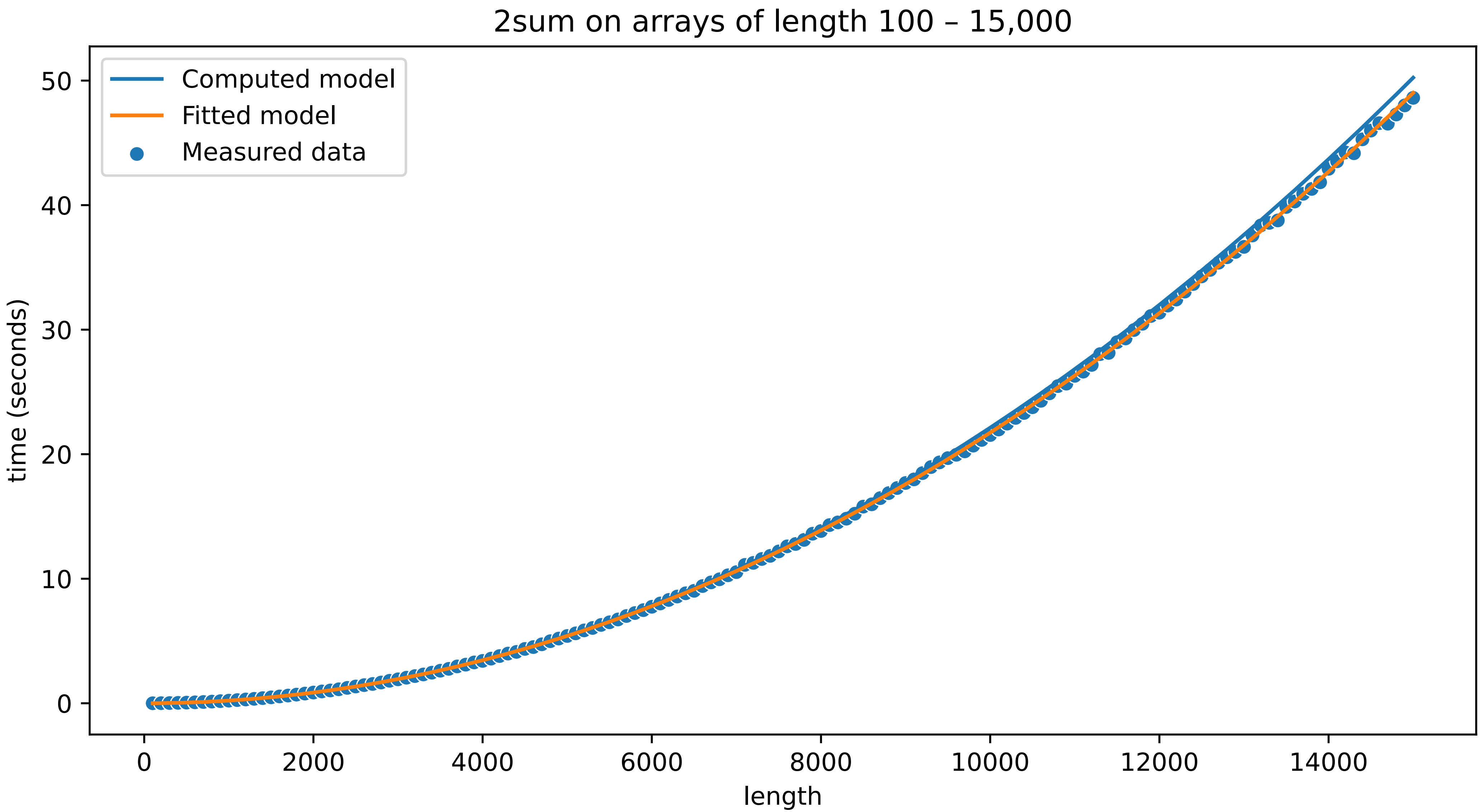
Checking the fit

```
from scipy.optimize import curve_fit

def powmod(x, a, b):
  return a * x ** b

[(a2, b2), _] = curve_fit(powmod, df_2s.n,
                        df_2s.exctime)

# Ignore the following lines
display(Markdown(f'$\\mathrm{{Fitted\ model:}}\\ {a2:.5e} '
display(Markdown(f'$\\mathrm{{Computed\ model:}}\\ {2**-22
```

$$\mathrm{Fitted\ model:}\ 2.04235e-07 \cdot x^{2.00671}$$

$$\mathrm{Computed\ model:}\ 1.80687e-07 \cdot x^{2.022}$$

Plotting


2sum on arrays of length 100 – 15,000
Computed model
Fitted model
Measured data
time (seconds)
0
10
20
30
40
50
0
2000
4000
6000
8000
10000
12000
14000
length

Reasoning about the growth

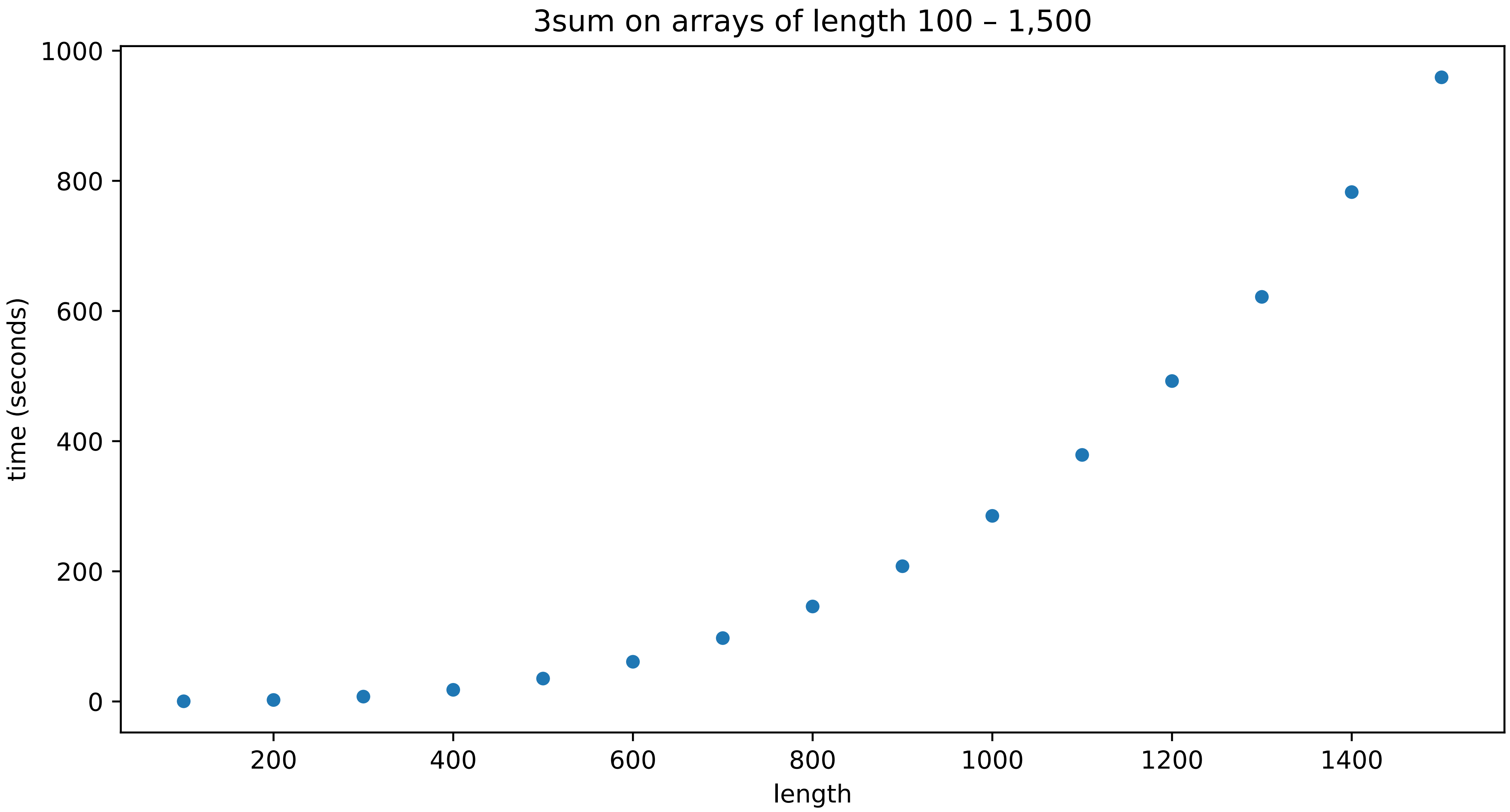
```
 1  def twosum(l:list[int]) -> list[tuple[int, int]]:
 2    res = []
 3    for i, vi in enumerate(l):
 4      for j, vj in enumerate(l):
 5        if i == j:
 6          continue
 7        if vi + vj == 0:
 8          res.append((vi, vj))
 9
10    return res
```

If `len(l) == 100`, then line 5 is executed 100^2 times.

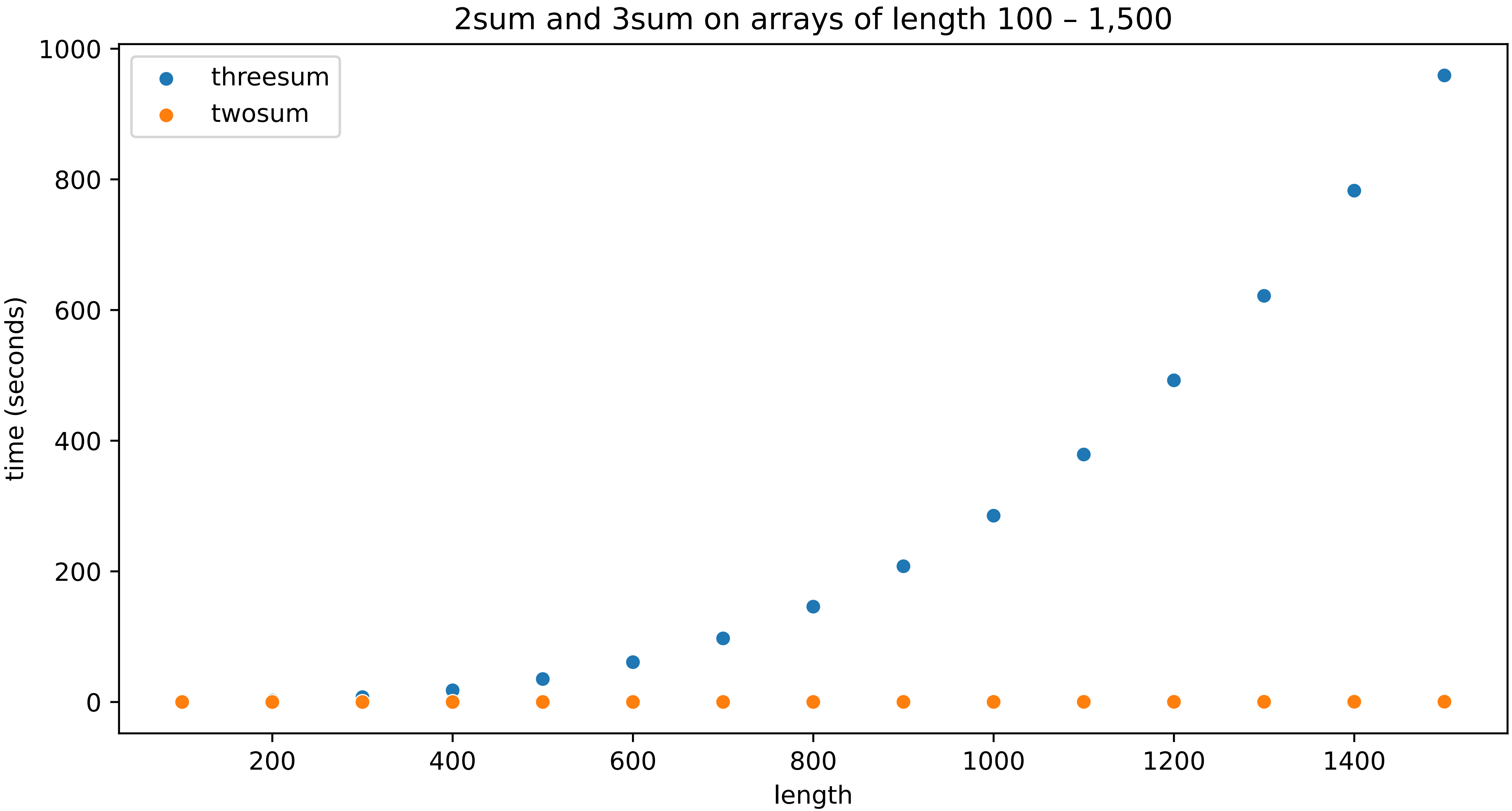
So, what about 3sum?

```python
def threesum(l:list[int]) -> list[tuple[int, int, int]]:
  res = []
  for i, vi in enumerate(l):
    for j, vj in enumerate(l):
      for k, vk in enumerate(l):
        if i == j or i == k or j == k:
          continue
        if vi + vj + vk == 0:
          res.append((vi, vj, vk))

  return res
```

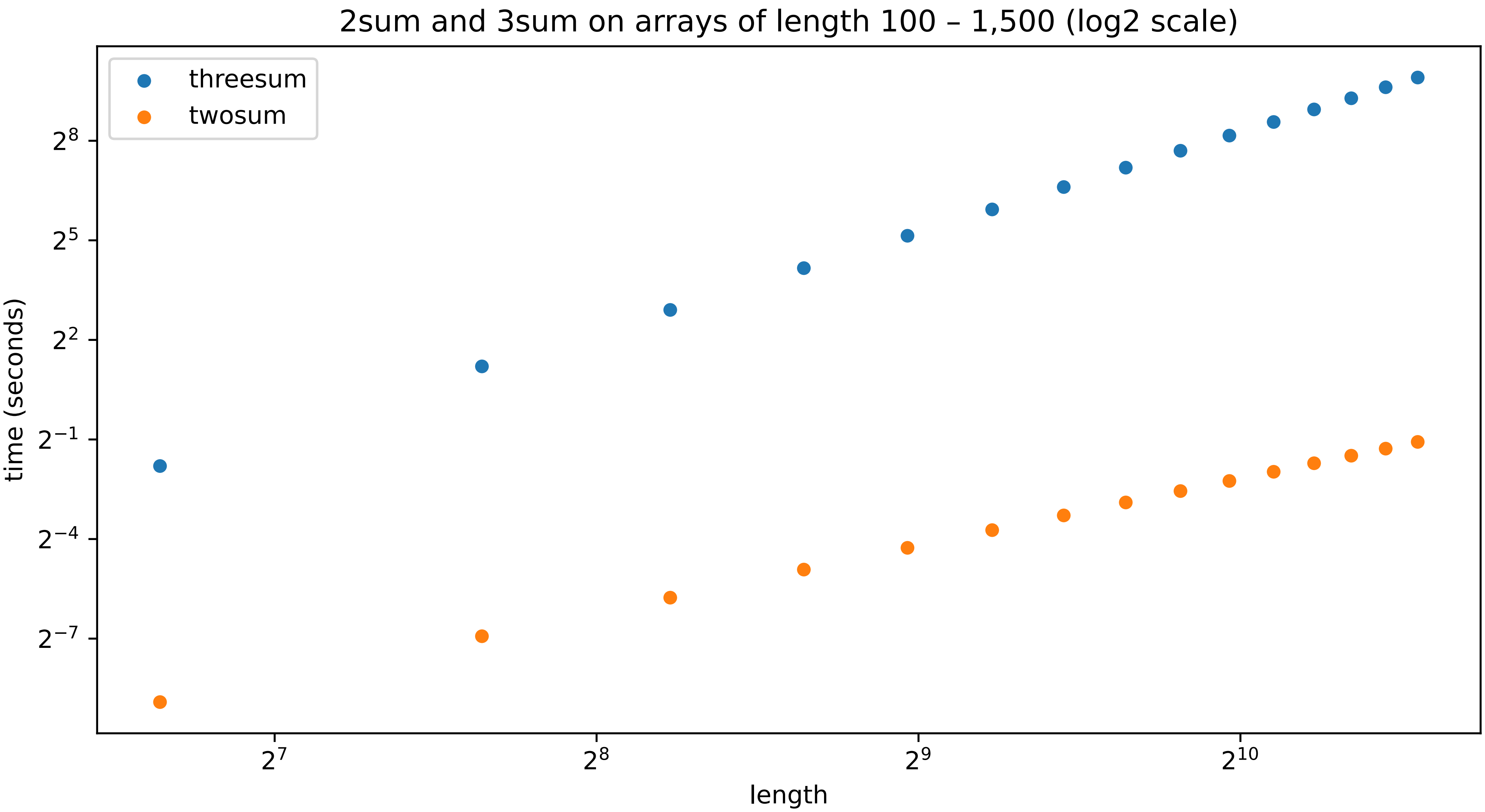
As bad as we think?



As bad as we think?



As bad as we think?



Quick analysis

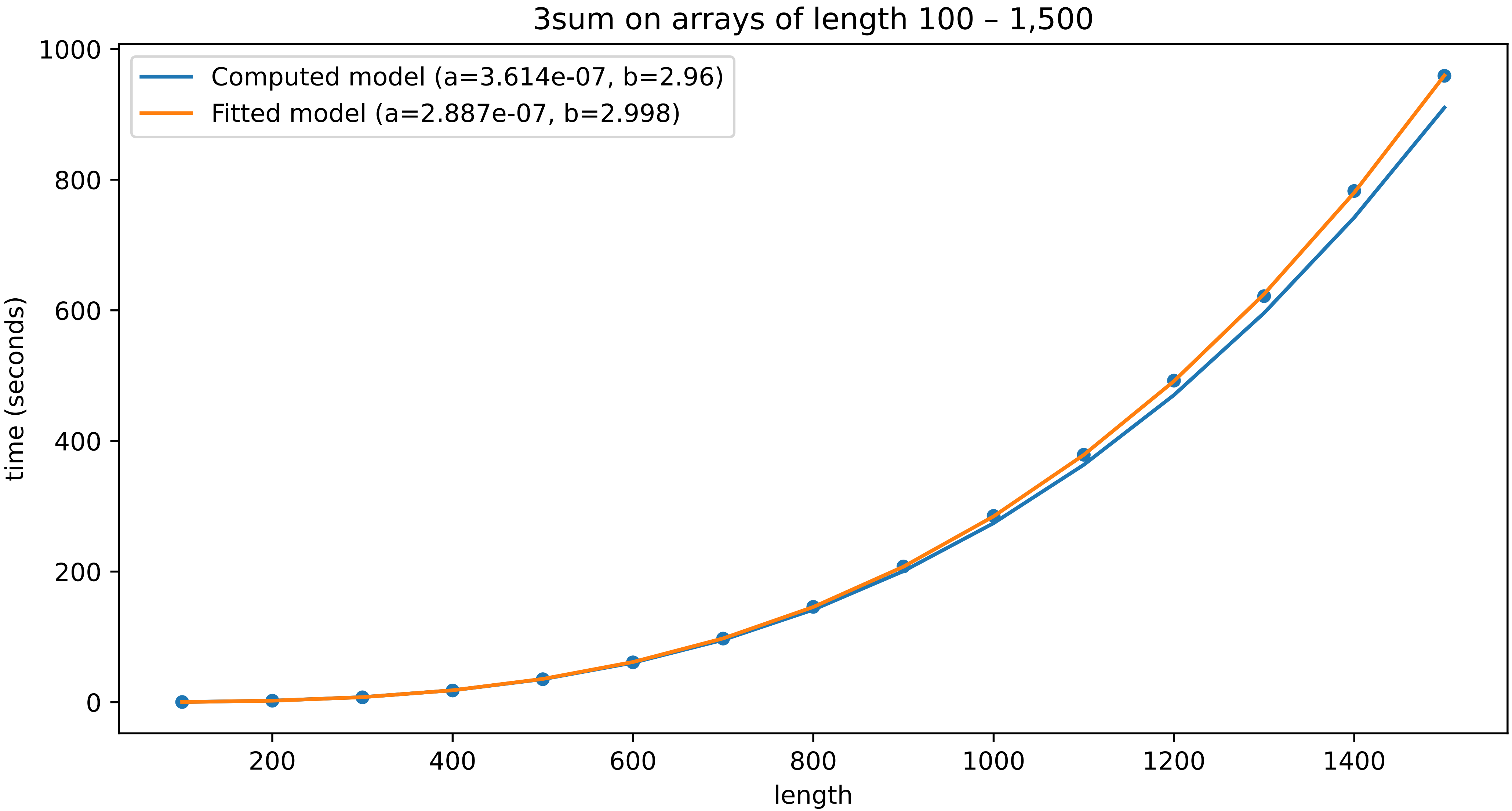
sz	time
100	0.287
200	2.301
400	17.903
800	145.976

$$\frac{\log_2 17.9 - \log_2 2.3}{\log_2 400 - \log_2 200} \approx 2.96$$

$$\log_2 17.9 - 2.96 \cdot \log_2 400$$

$$\approx -21.4$$

Cubic?


3sum on arrays of length 100 – 1,500
Computed model (a=3.614e-07, b=2.96)
Fitted model (a=2.887e-07, b=2.998)
time (seconds)
0
200
400
600
800
1000
length
200
400
600
800
1000
1200
1400

Does it matter?

- We estimate the time for larger arrays using our models:

sz	1sum	2sum	3sum
25000	0.004	137	4416875
50000	0.009	549	35284054
100000	0.018	2206	281865436

> 2,206 seconds is about 40 minutes and 281,865,439 seconds is about 9 years

Upper or lower bound?

Can we do better (or worse)?

- We have looked at a few algorithms to get a feeling for how the runtime increases when the input size increases
- We have used “obvious” algorithms and random input
  - Are there easy improvements to the algorithms
  - Are there special cases in the data
  - (remember union find)
- We will return to these ideas when we discuss algorithm design

Average

```python
def avg(l:list[int]) -> float:
  s = 0
  for v in l:
    s += v

  return s/len(l)
```

- Are there any obvious improvements?
  - Maybe, but as we will see later, they do not matter very much
  - Can *actually* be slower!

1sum

```python
def onesum(l:list[int]) -> list[int]:
  res = []
  for vi in l:
    if vi == 0:
      res.append(vi)
  return res
```

- Tempting to think that we can improve this by sorting
  - Maybe in an implementation, but not generally

2sum

```python
def twosum(l:list[int]) -> list[tuple[int, int]]:
  res = []
  for i, vi in enumerate(l):
    for j, vj in enumerate(l):
      if i == j:
        continue
      if vi + vj == 0:
        res.append((vi, vj))

  return res
```

» Many possible improvments!

Smarter iteration

- We iterate over the whole list every time

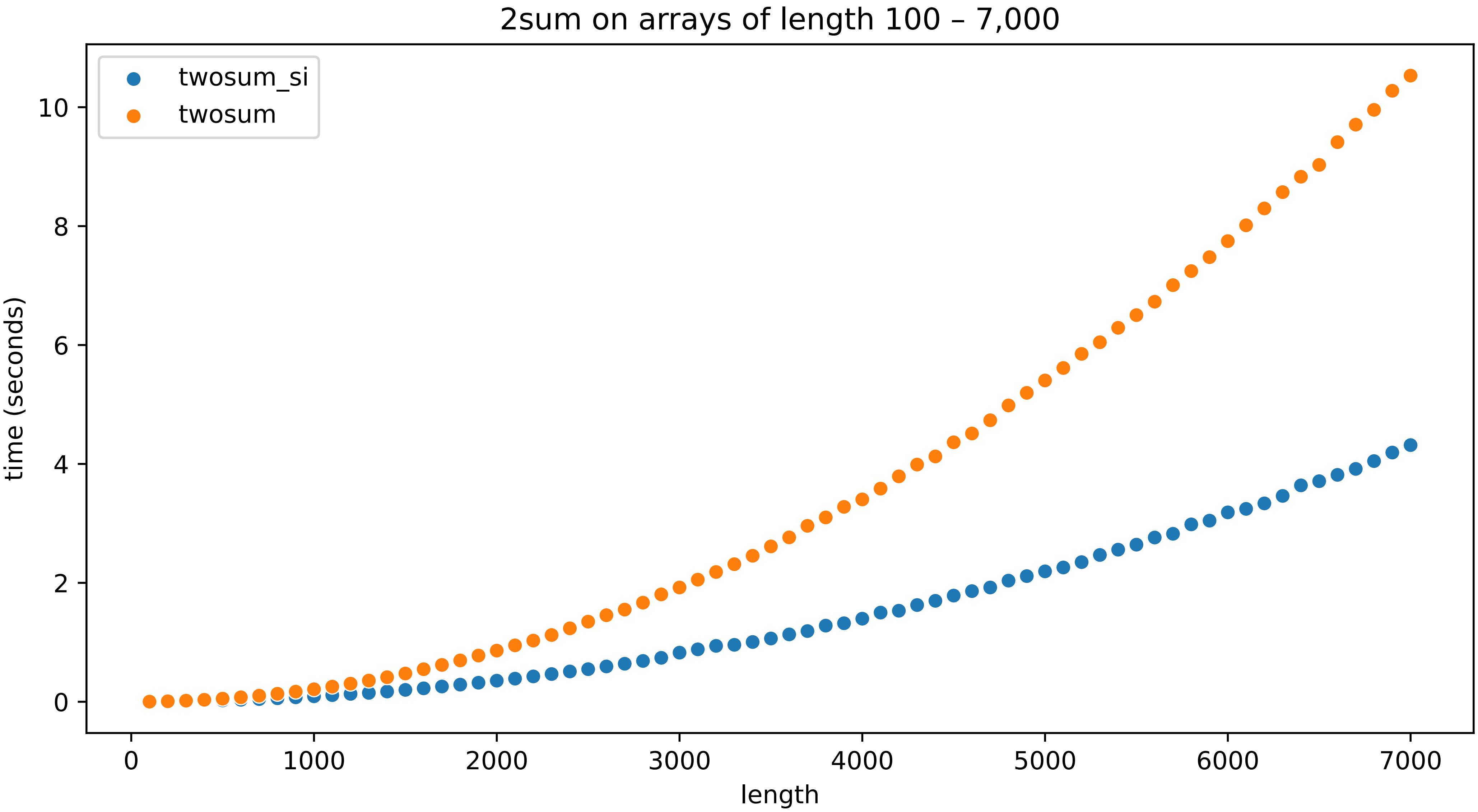
```
for i, vi in enumerate(l):
  for j, vj in enumerate(l):
```

- We have already processed `[0:i]` so `j` can start at `i+1`
- This also solves the problem of having both `(i, j)` and `(j, i)` in the results

Smarter iteration

```python
def twosum_si(l:list[int]) -> list[tuple[int, int]]:
  res = []
  for i, vi in enumerate(l):
    for vj in l[i+1:]:
      if vi + vj == 0:
        res.append((vi, vj))

  return res
```

Does it matter?

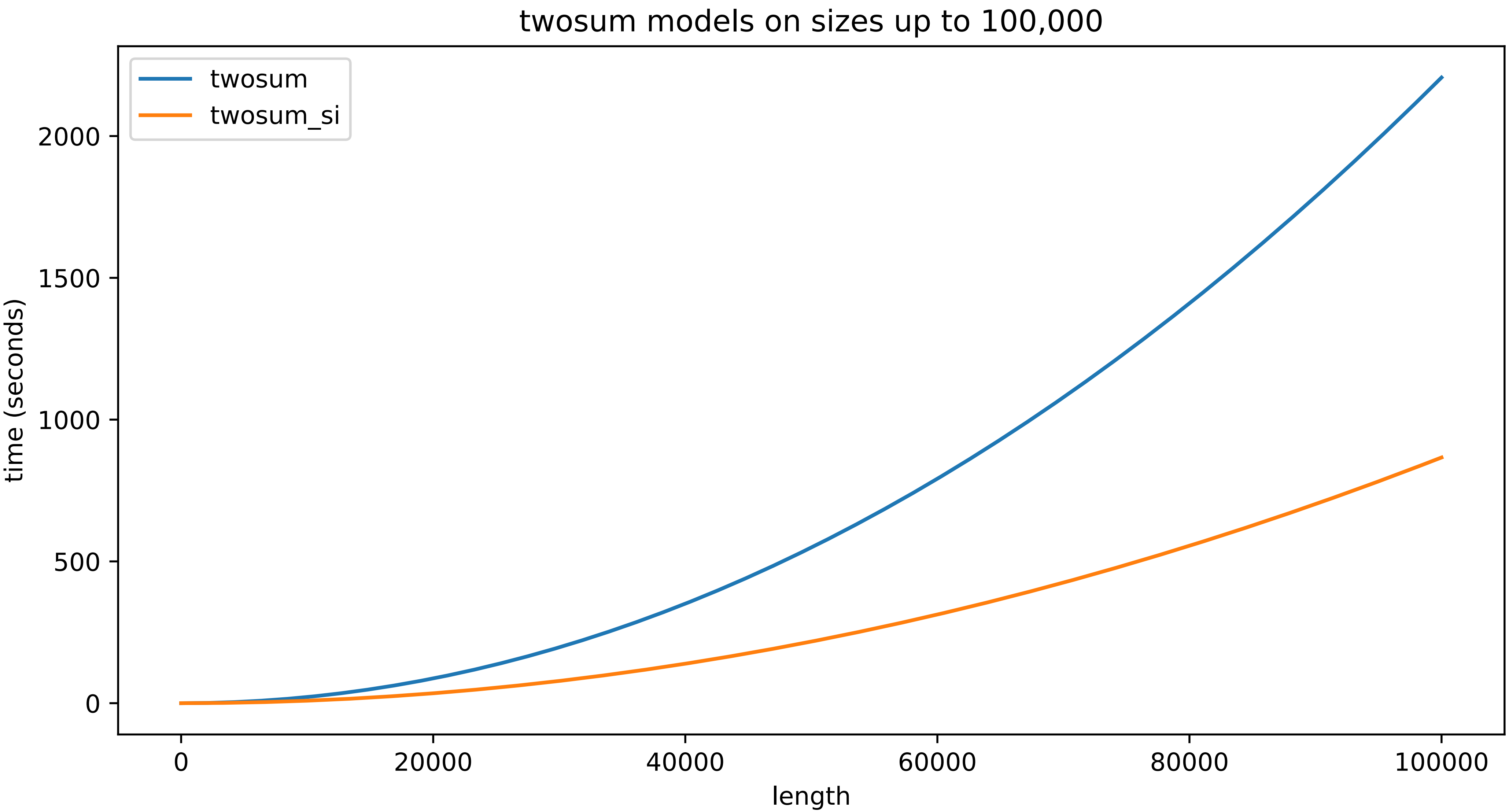


Does it matter?

- » $\text{twosum model} : 2.04235e - 07 \cdot x^{2.00671}$

- » $\text{twosum\_si model} : 9.11327e - 08 \cdot x^{1.99562}$

Does it matter?



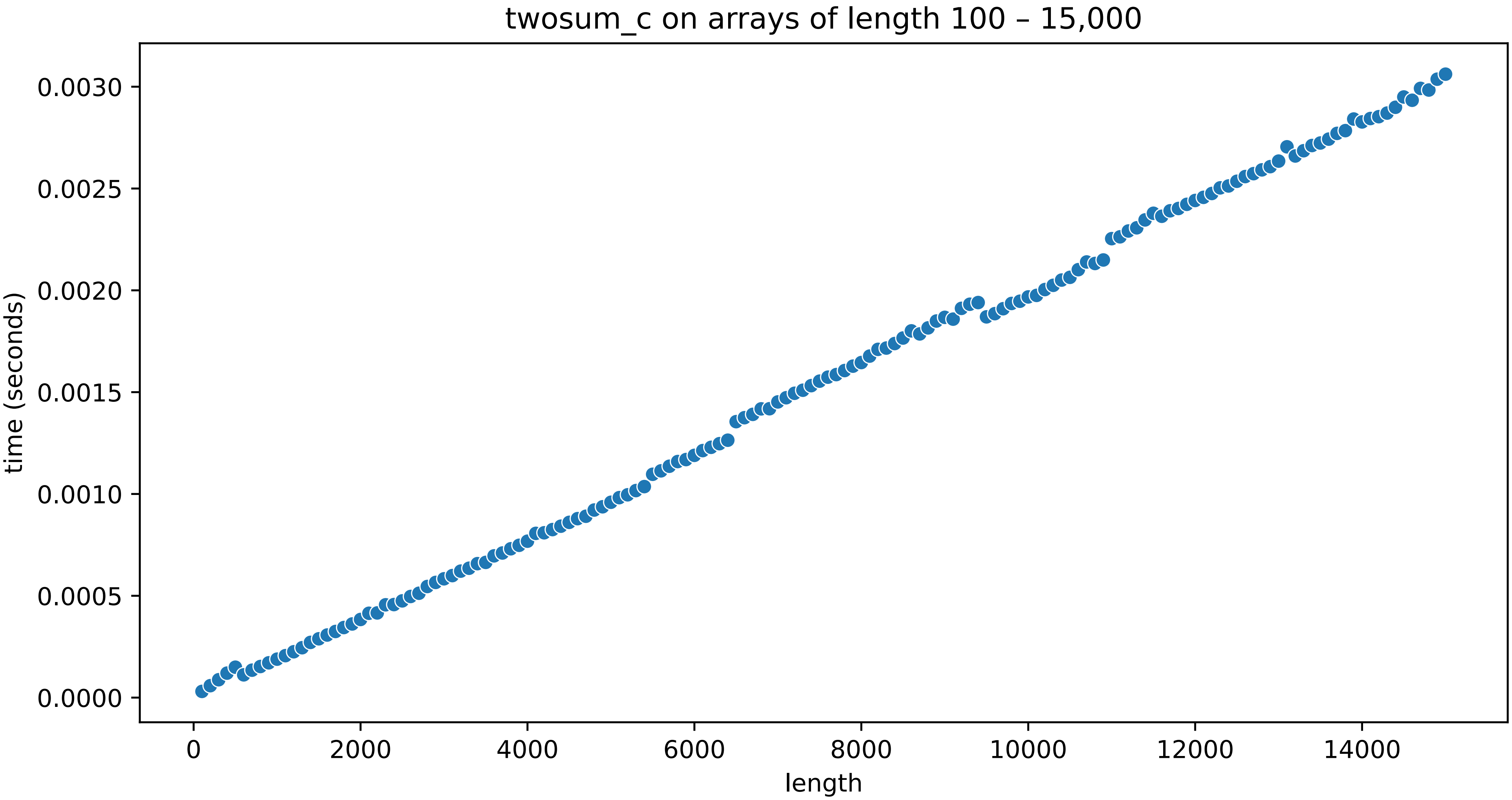
2sum with caching

- Simple idea
  - Store numbers we have already seen
  - If we have seen the current negative number, then it is a match
- Should make it faster, but ...
  - Increases memory usage (cache)

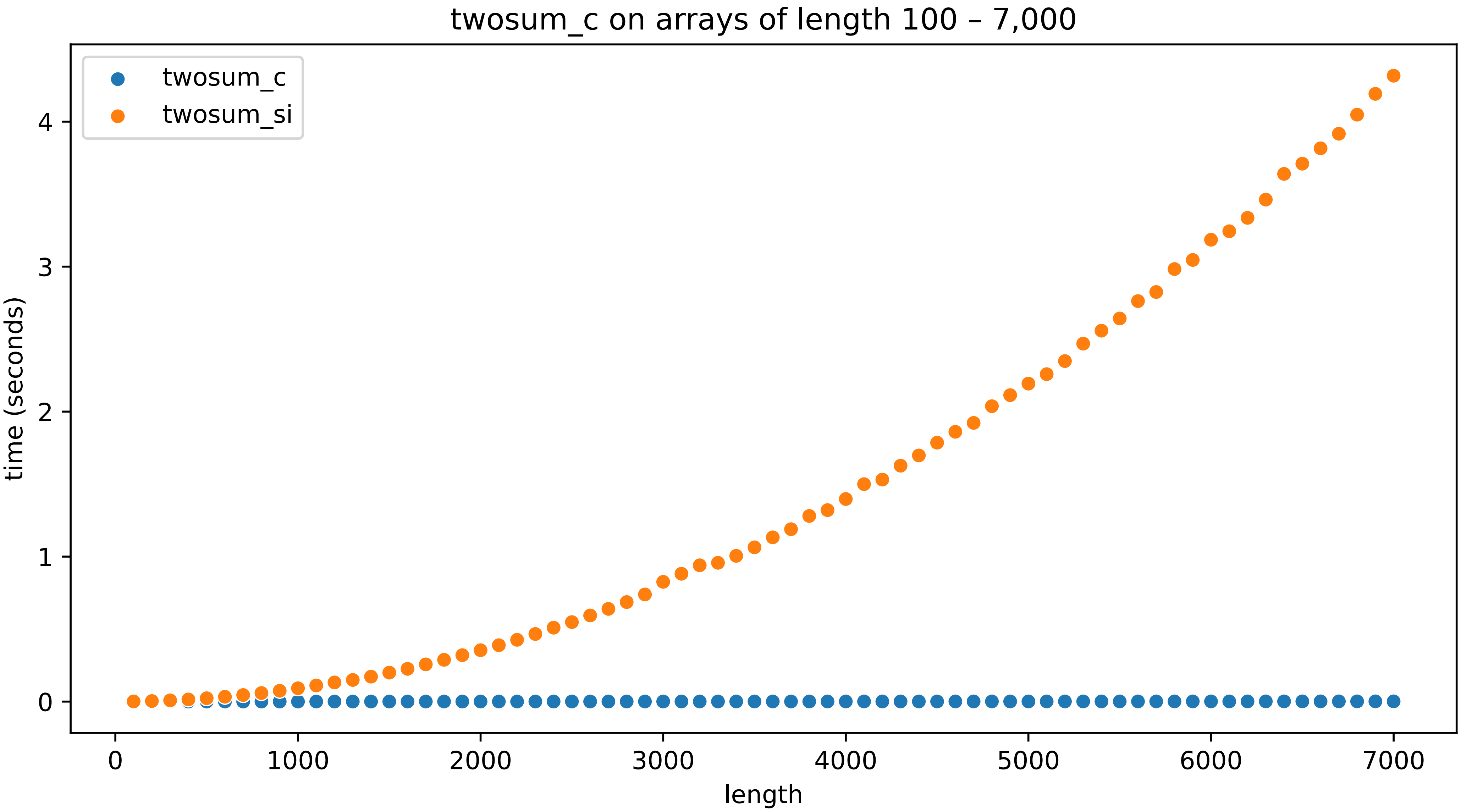
2sum with caching

```python
def twosum_c(l:list[int]) -> list[tuple[int, int]]:
  cache = {}
  res = []

  for i, vi in enumerate(l):
    if -vi in cache:
      res.append((vi, cache[-vi]))
      cache[vi] = vi

  return res
```

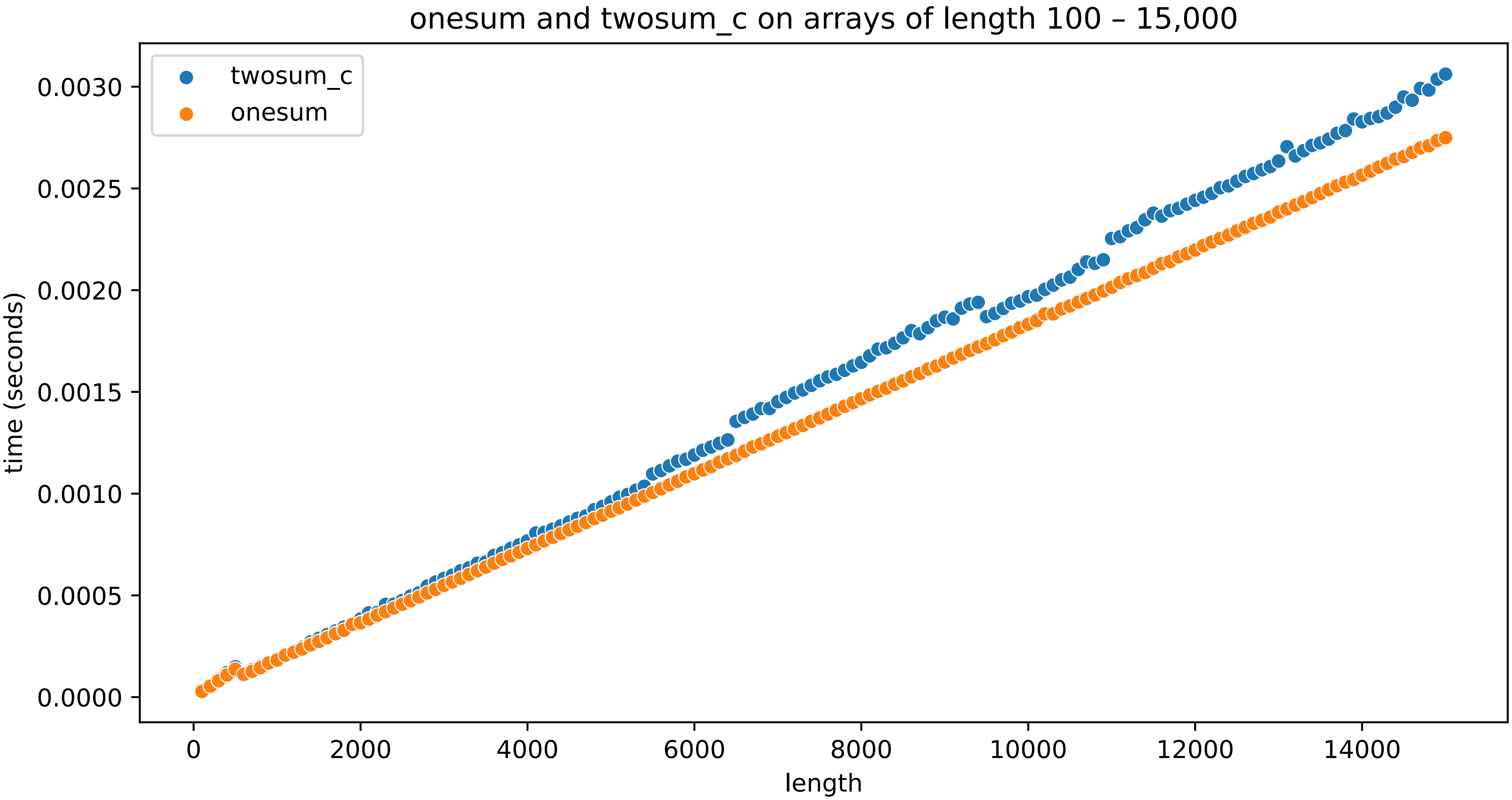
Expected improvement?



Expected improvement?



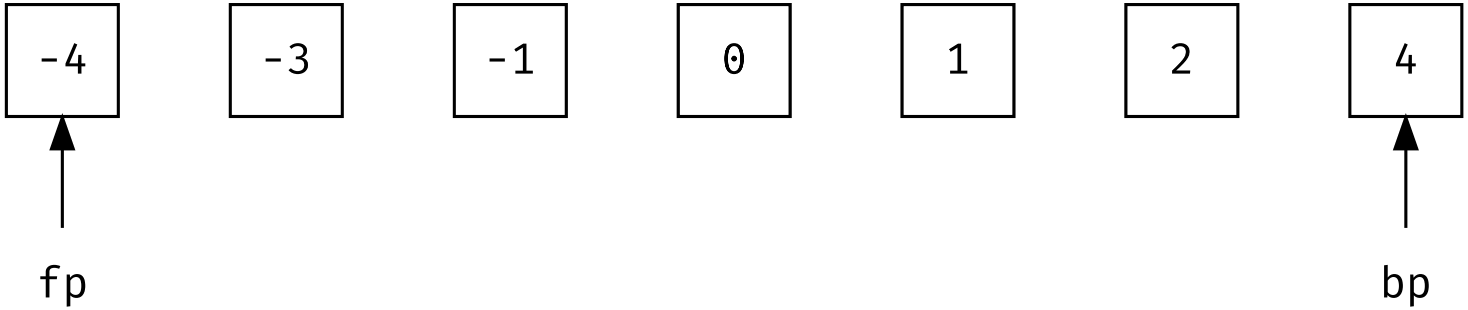
Expected improvement?



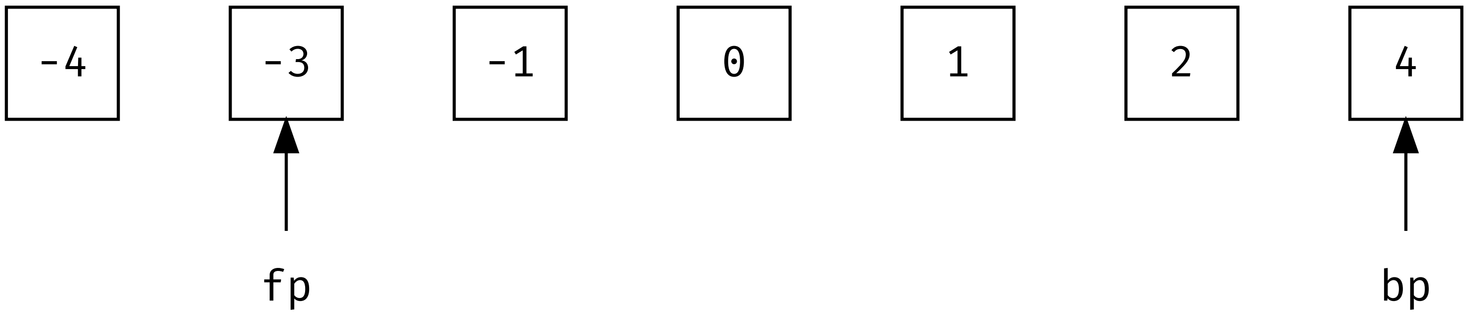
2sum with sorting

- We can also sort the input and add numbers from the two ends of the list
  - If too small, pick a larger number in the front
  - If too larger, pick a smaller number in the end

2sum with sorting

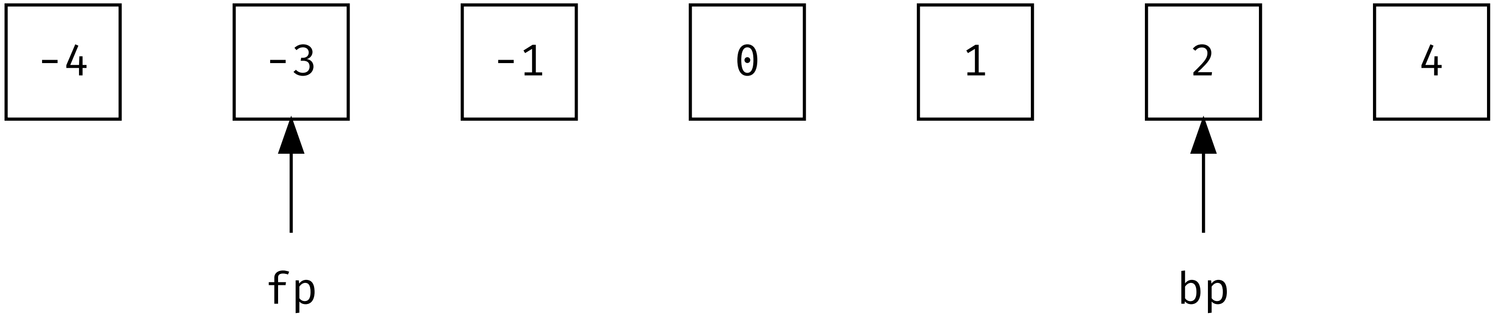


2sum with sorting



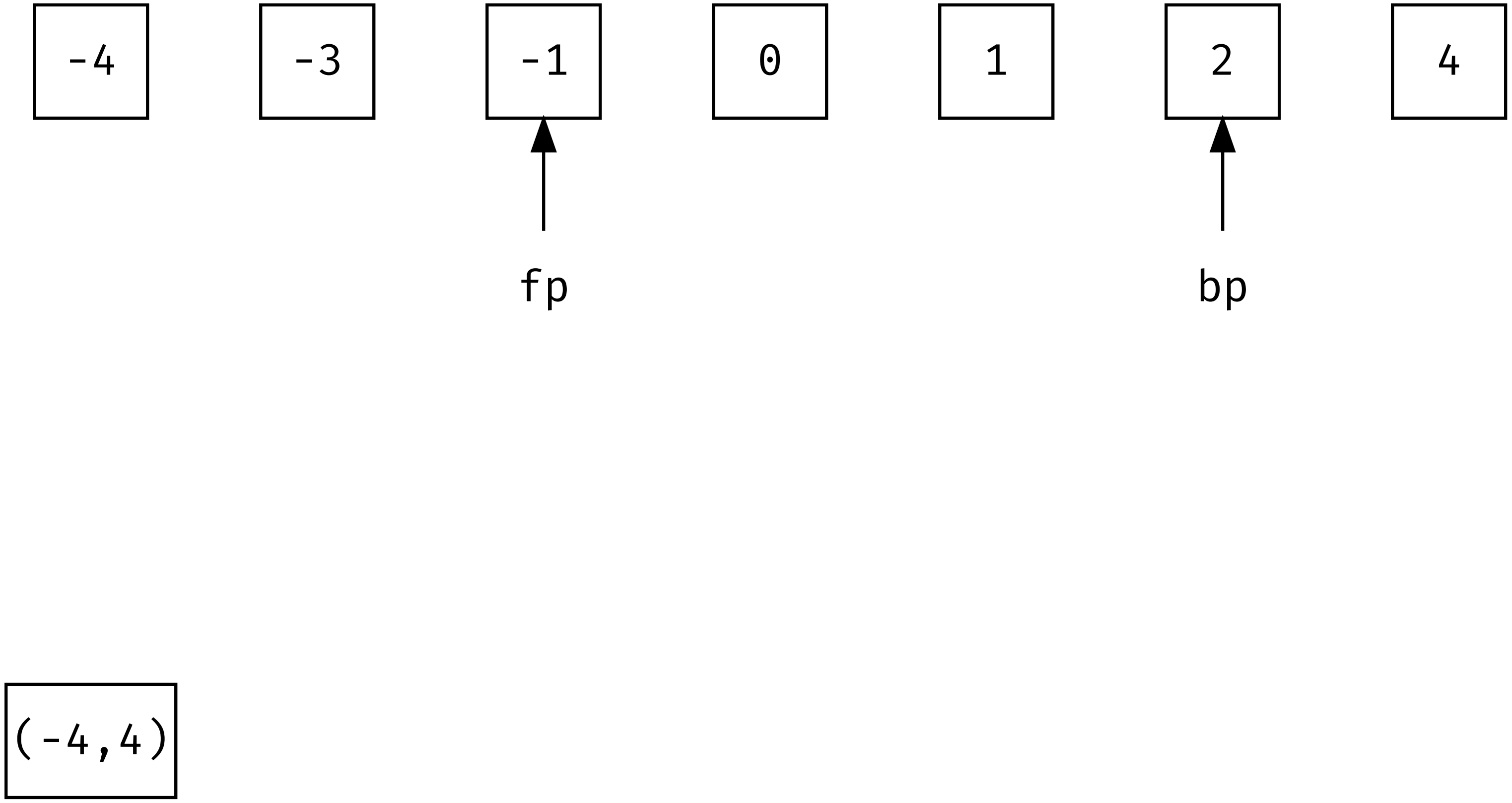
(-4,4)

2sum with sorting

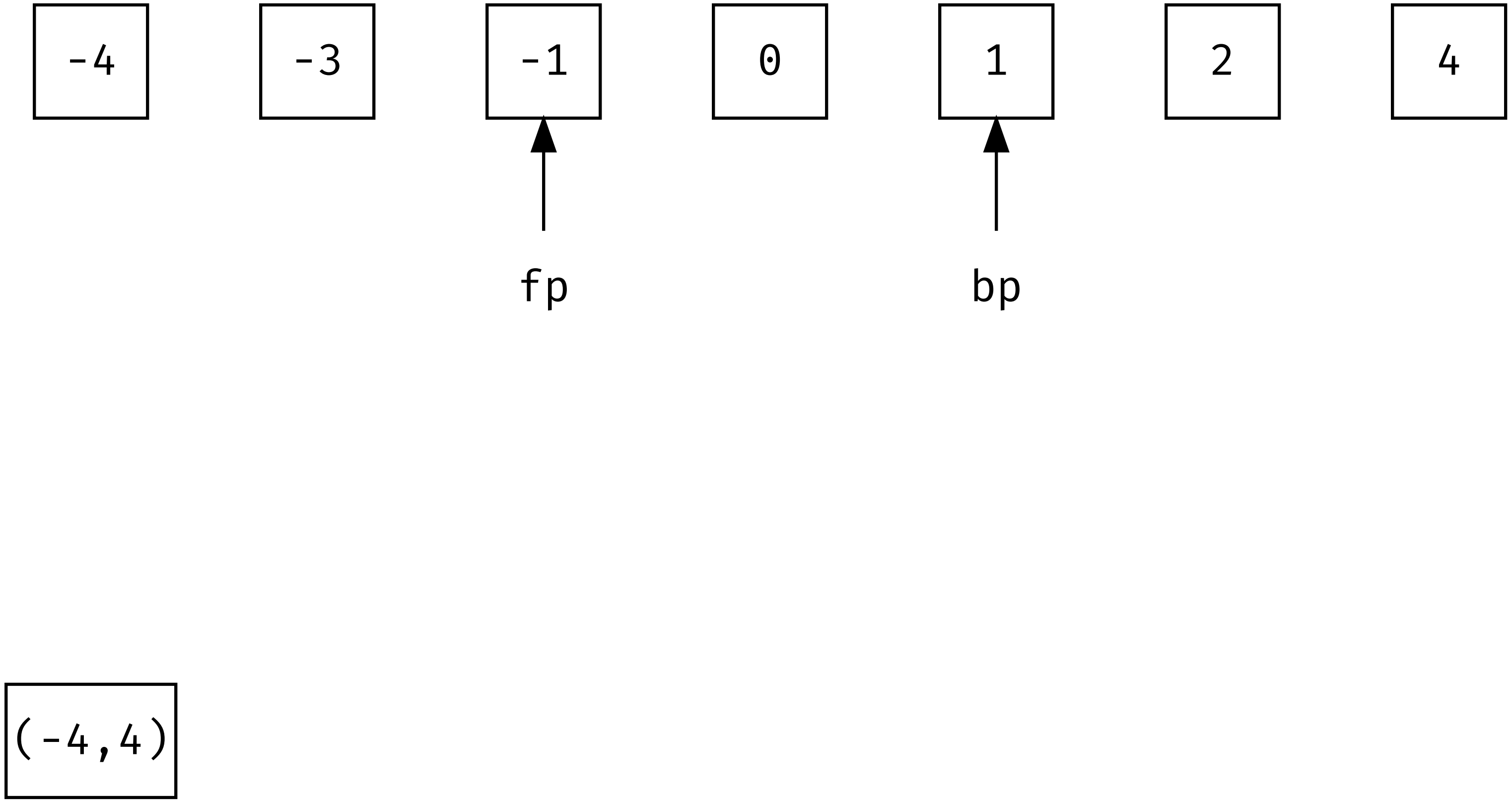


(-4,4)

2sum with sorting

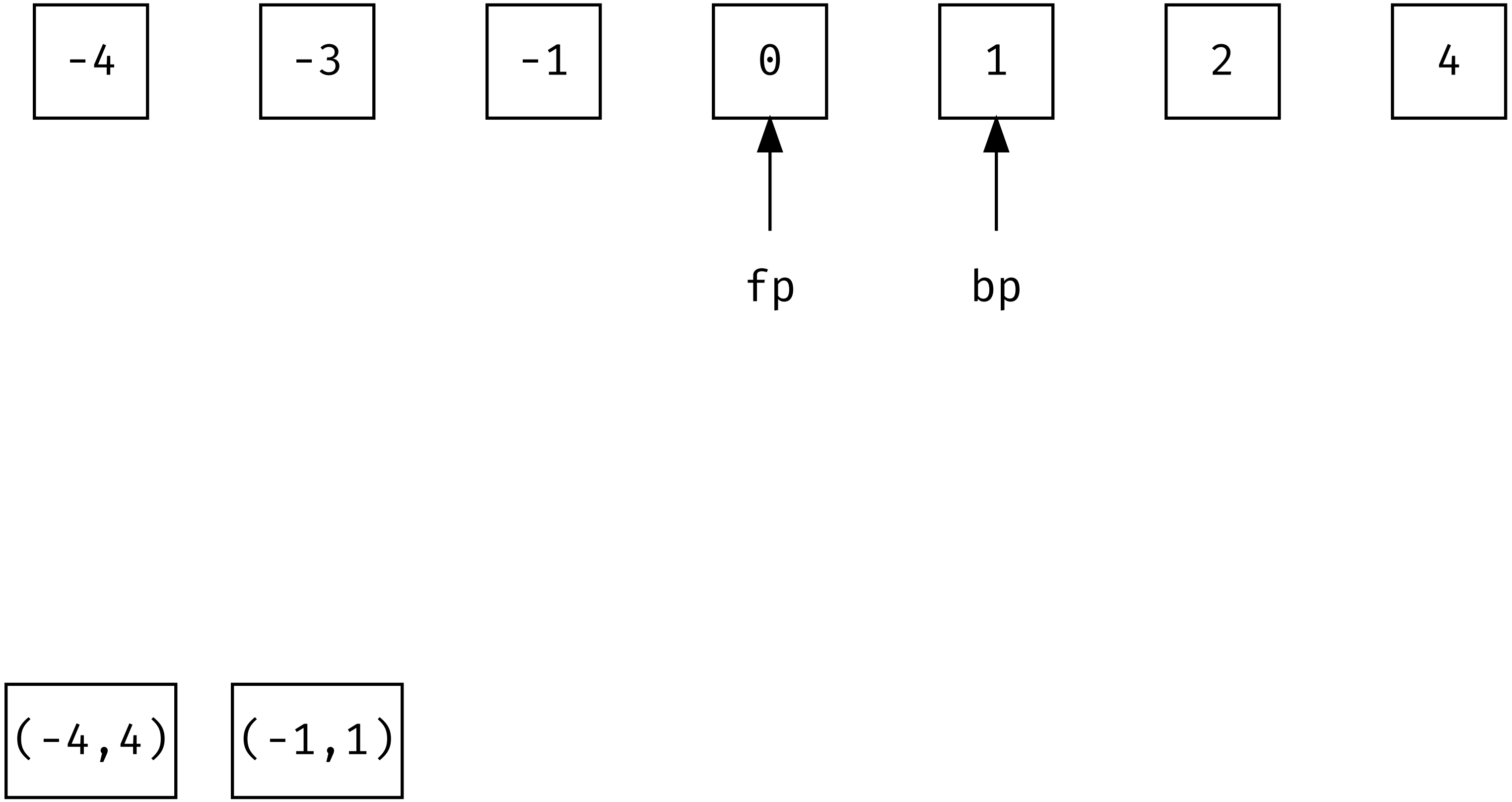


2sum with sorting


-4 -3 -1 0 1 2 4

(-4,4)

2sum with sorting

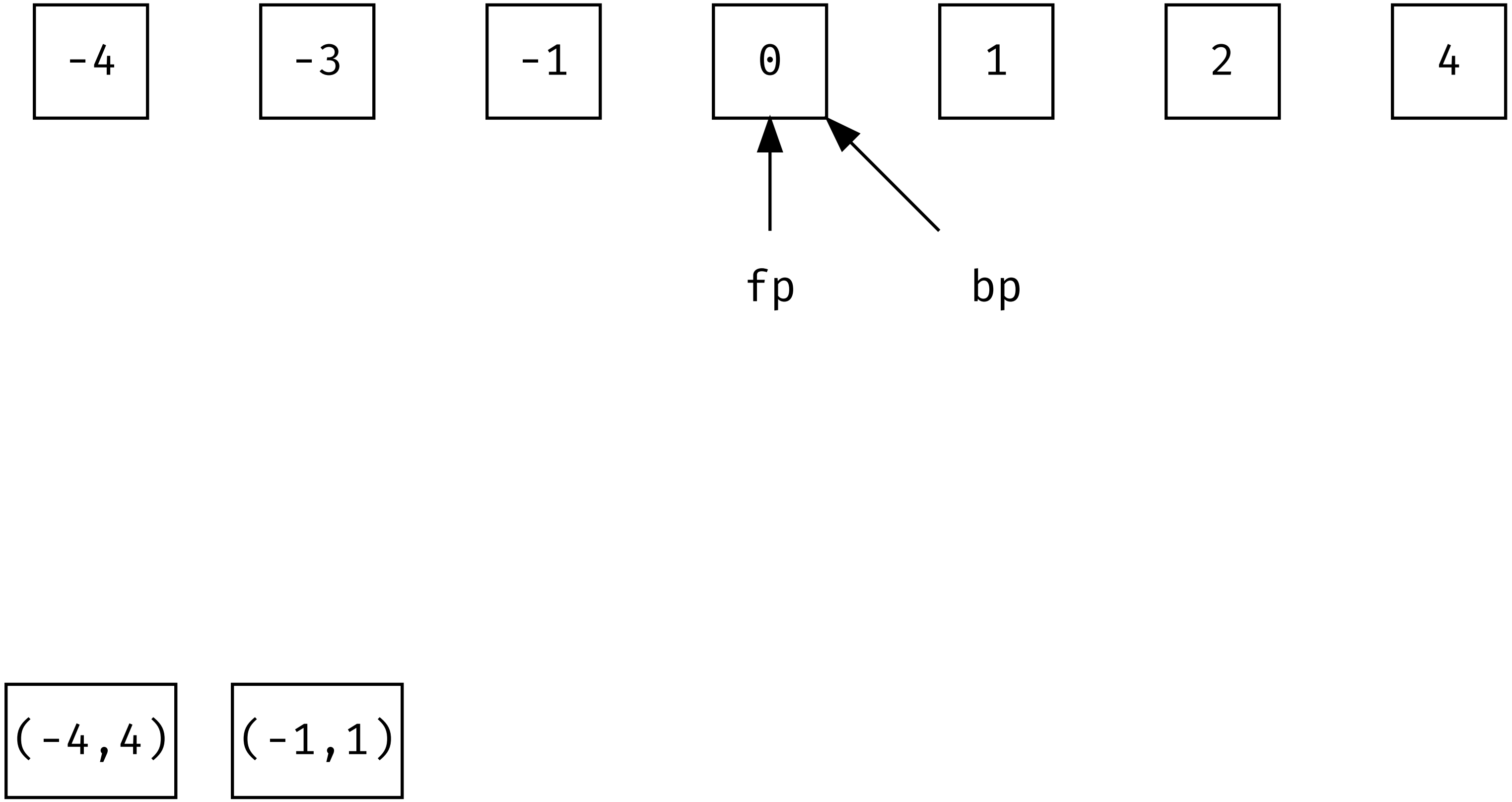


-4 -3 -1 0 1 2 4

fp bp

(-4,4) (-1,1)

2sum with sorting



-4 -3 -1 0 1 2 4

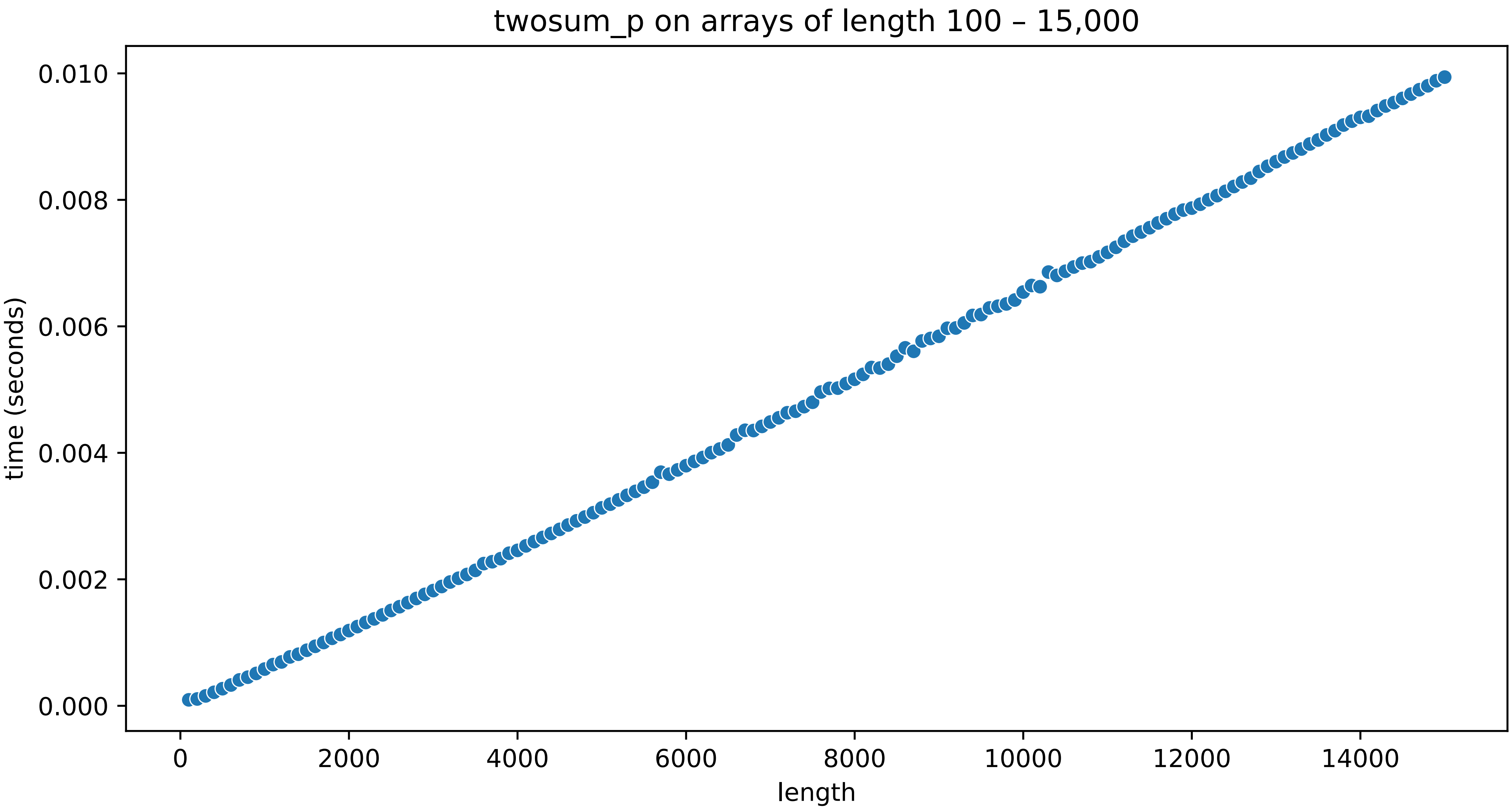
fp bp

(-4,4) (-1,1)

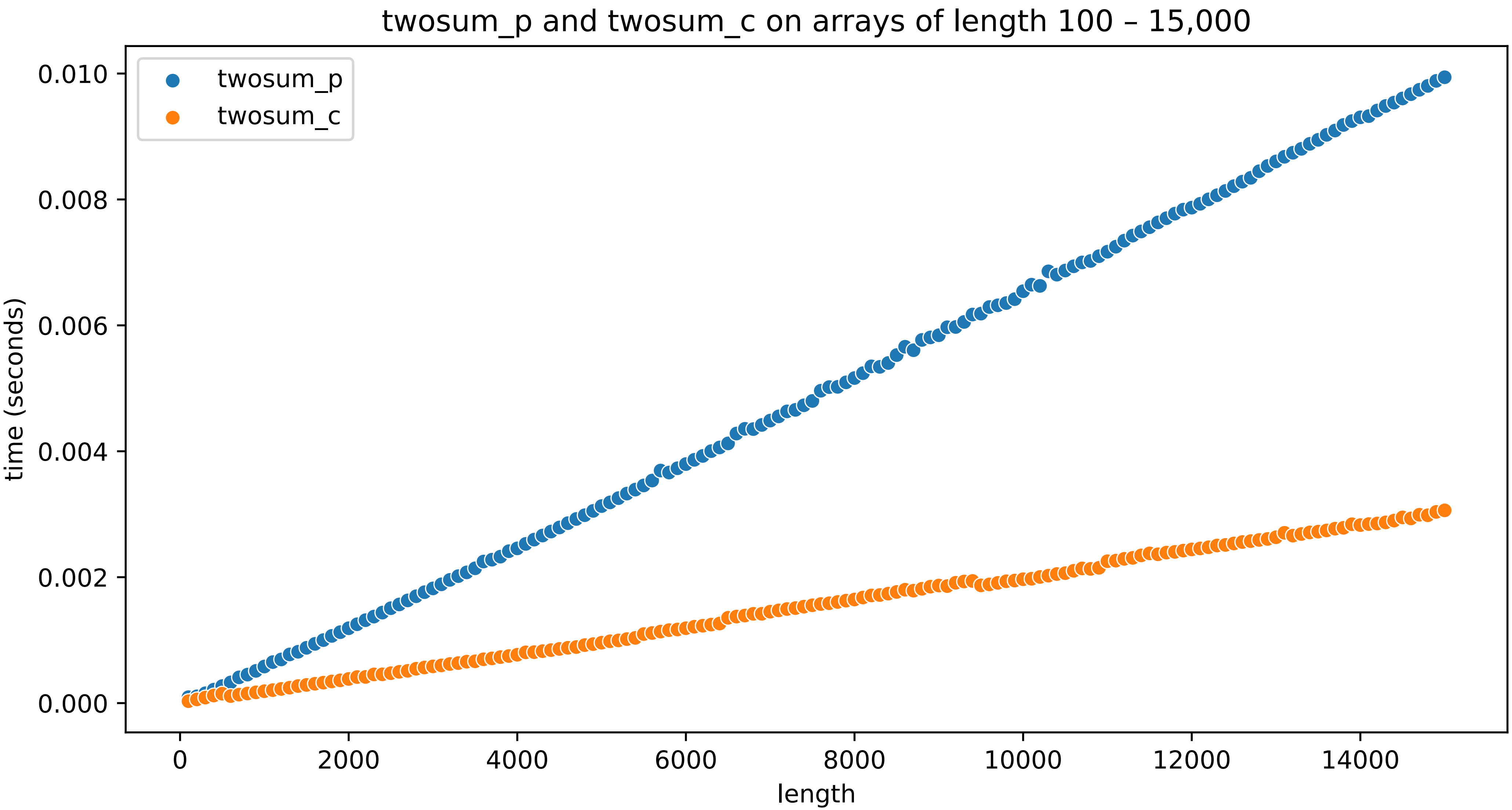
2sum with sorting

```python
def twosum_p(l:list[int]) -> list[tuple[int, int]]:
  res = []
  s = sorted(l)
  fp, bp = 0, len(s) -1
  while fp < bp:
    p = s[fp] + s[bp]
    if p == 0:
      res.append((s[fp], s[bp]))
      fp += 1
    elif p < 0:
      fp += 1
    else:
      bp -= 1
  return res
```

Still linear?



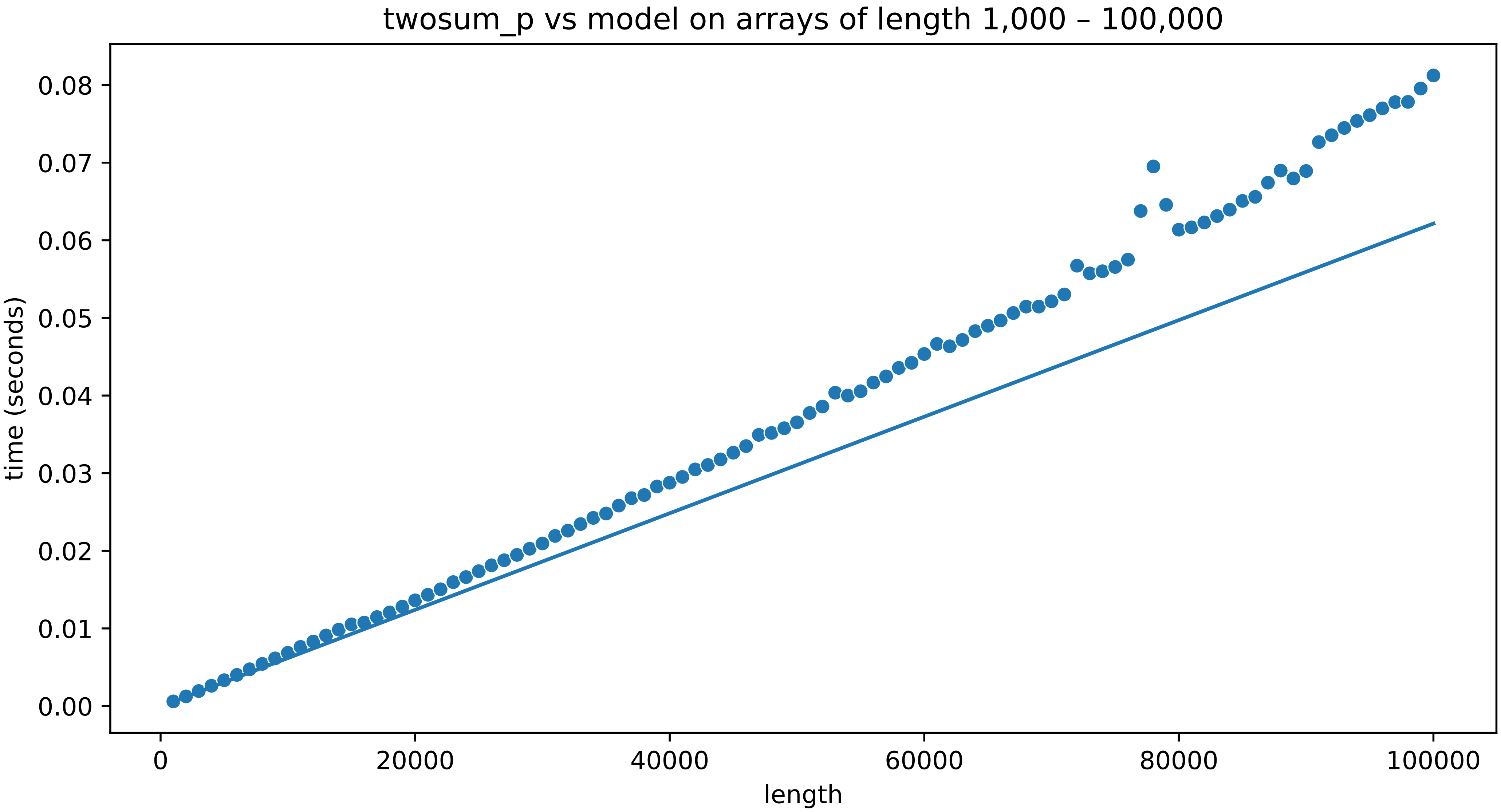
Still linear?



Still linear?

- No, we cannot generally sort in linear time
  - Requires that we compare elements
- However, `sorted()` is fast and hides the cost well

Still linear?



What about 3sum?

- Easy, we can modify 2sum to take a target value instead of 0
  - `def twosum(l:list[int], t:int=0) -> list[tuple[int, int]]`
- We can then iterate over the list and use `twosum` to look for two numbers that sum to the negative
  - `twosum(l[j+1:], -l[j])`
- This will get us to quadratic

What about 3sum?

- We can also use the cache and pointer techniques directly
- Same idea as before
  - Iterate over the outer and use pointers or cache on the "twosum" part

Mathematical models

Remember

```python
xy0 = (0,0)
xy2 = (2,10)
tx = xy2[0] - xy0[0]
ty = xy2[1] - xy0[1]
m = ty / tx
```

We can estimate runtimes

```
%timeit 500 - 32
```

```
4.05 ns ± 0.0167 ns per loop (mean ± std. dev. of 7 runs,
100,000,000 loops each)
```

We can estimate runtimes

- How long should `tx = xy2[0] - xy0[0]` take?
  - Array/tuple access: 26.6ns
  - Subtraction (int): 5.48ns
  - Storing an int: 8.64ns
- So, 26.6 + 26.6 + 5.48 + 8.64 = 67.32
- Takes about 62.8 ns (according to `%timeit`)

We can estimate runtimes

- Can be done, but annoying if we have a large number of possible operations
- More exact the closer we get to “the machine”

Simplifications

Alan Turing, “Rounding-off errors in matrix processes”:

> It is convenient to have a ***measure of the amount of work involved in a computing process***, even though it be a very ***crude*** one. We may count up the number of times that various elementary operations are applied in the whole process and then given them various weights. We might, for instance, count the number of additions, subtractions, multiplications, divisions, recording of numbers, and extractions of figures from tables. In the case of computing with matrices most of the work consists of multiplications and writing down numbers, and ***we shall therefore only attempt to count the number of multiplications and recordings***.

Another 2sum

```
count = 0
i = 0
while i < N:
  j = i+1
  while j < N:
    if a[i] + a[j] == 0:
      count += 1
    j += 1
  i += 1
```

Another 2sum

```
                          # freq
count = 0                 # 1
i = 0                     # 1
while i < N:              # N + 1
  j = i+1                 # N
  while j < N:            # 0.5 * (N+2) * (N+1)
    if a[i] + a[j] == 0:  # N * (N-1) 2 * (0+1+2+...+(N-1))
      count += 1          # 0 to 0.5 * N * (N-1)
    j += 1                # 0.5 * N * (N-1)
  i += 1                  # N
```

What?

- $\sum_{i=1}^{n} i = \frac{n(n+1)}{2}$
- True, but in this case, we are interested in
  - $\sum_{i=0}^{n-1} i = \frac{(n-1)(n-1+1)}{2}$, or
  - $\sum_{i=0}^{n-1} i = \frac{1}{2}n(n-1)$

Another 2sum

```
                         # freq
count = 0                # 1
i = 0                    # 1
while i < N:             # N + 1
  j = i+1                # N
  while j < N:           # 0.5 * (N+2) * (N+1)
    if a[i] + a[j] == 0: # N * (N-1) 2 * (0+1+2+...+(N-1))
      count += 1         # 0 to 0.5 * N * (N-1)
    j += 1               # 0.5 * N * (N-1)
  i += 1                 # N
```

- Declaration and assignment: $N + 2$ each
- Less than compare: $0.5 \cdot (N + 1) \cdot (N + 2)$
- Array accesses: $N \cdot (N - 1)$
- ...

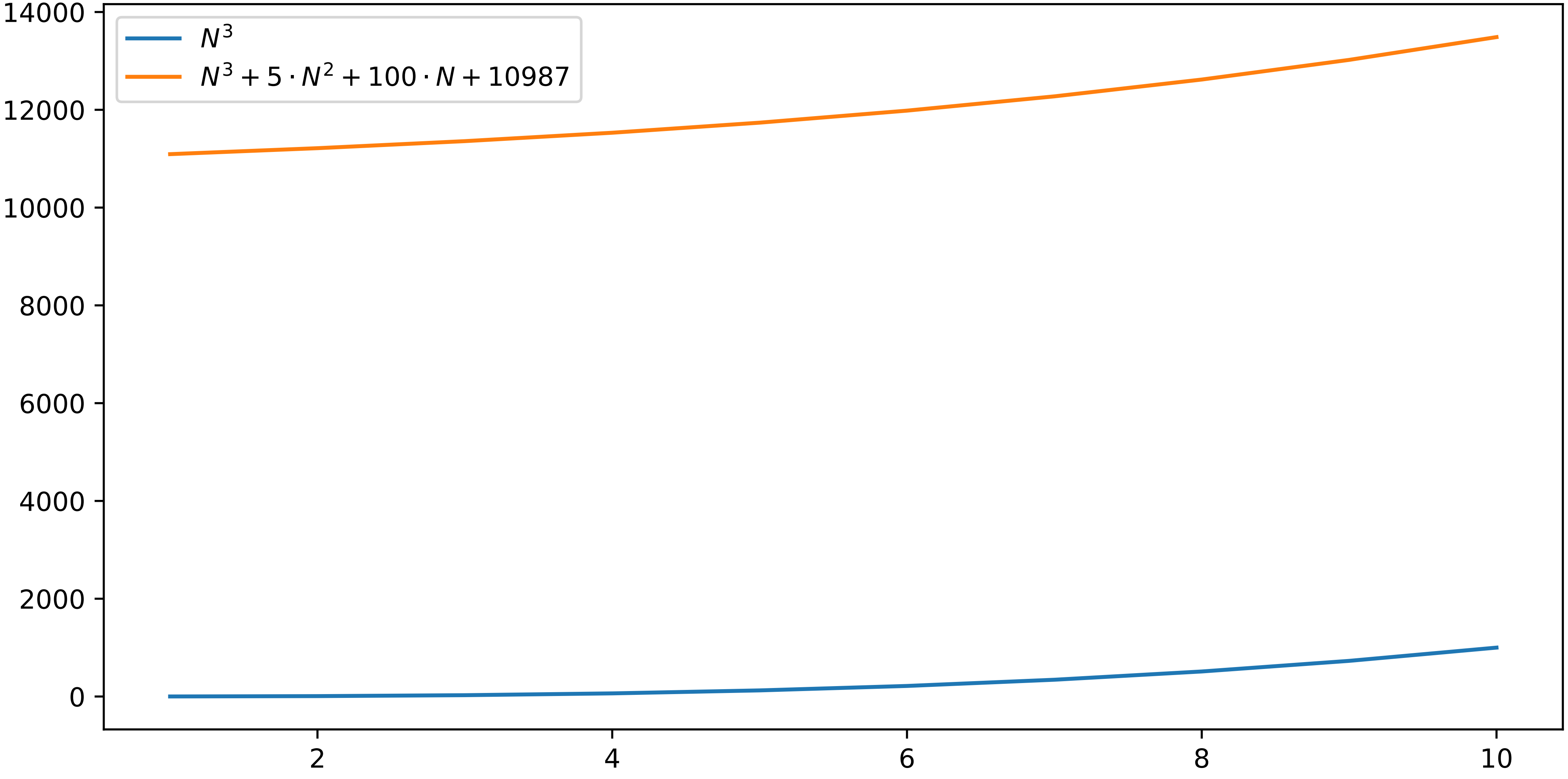
Simplification: decide a cost model

- Use some basic operation as a proxy for running time
  - Too much effort (and guessing) to determine exactly how many times each operation is performed
- We can for example use array accesses $(N \cdot (N - 1))$

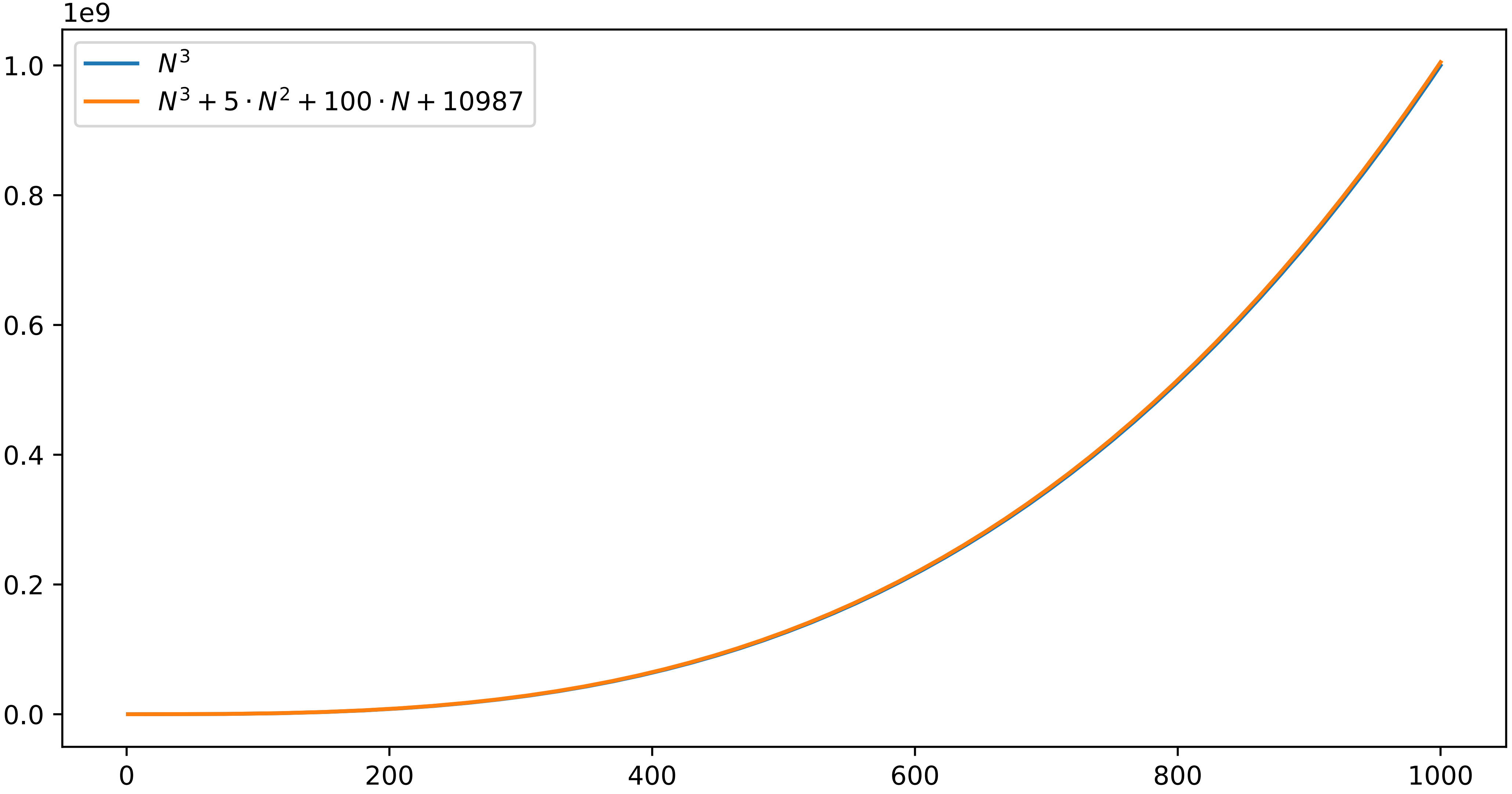
Simplification: tilde notation

- We estimate runtime or memory use as a function of input size N
    - Remember the array length for twosum and threesum, for example
- If we add things together, we will get a number of terms
    - As N grows, the lower order terms are negligible
    - And if N is small, we do not care
- So, $N^3 + 5 \cdot N^2 + 100 \cdot N + 10987 \sim N^3$

Why can we not care?



Why can we not care?



Technically

$$f(N) \sim g(N) \text{ means } \lim_{N \to \infty} \frac{f(N)}{g(N)} = 1$$

Why can we not care?

N	N^3	original/N^3
100	1000000	1.07099
1000	1000000000	1.00511
10000	1000000000000	1.0005
100000	1000000000000000	1.00005
1000000	1000000000000000000	1.00001

So, putting it all together

- We use array access for our cost model to analyze twosum:
    - $N \cdot (N - 1)$ accesses
- We use tilde notation to approximate
    - $N \cdot (N - 1) \sim N^2$
- So, twosum is quadratic with respect to input size

Trying 3sum

Remember (a bit changed, to simplify):

```
1 count = 0
2 for i, vi in enumerate(l):
3   for j, vj in enumerate(l[i+1], start=i+1):
4     for vk in l[j+1:]:
5       if vi + vj + vk == 0:
6         count += 1
```

- How many times is line 5 executed?
  - $\frac{N \cdot (N-1) \cdot (N-2)}{3!} \sim \frac{1}{6} N^3$
- Three array accesses in line 5, so
  - $1/2 \cdot N^3$

Mathematical models

- There are accurate mathematical models available
- In practice:
  - The formulas can be complicated
  - Advanced mathematics might be required
  - So, we leave it for a more advanced course (or experts)
- We use approximate models: $T(N) \sim c \cdot N^3$

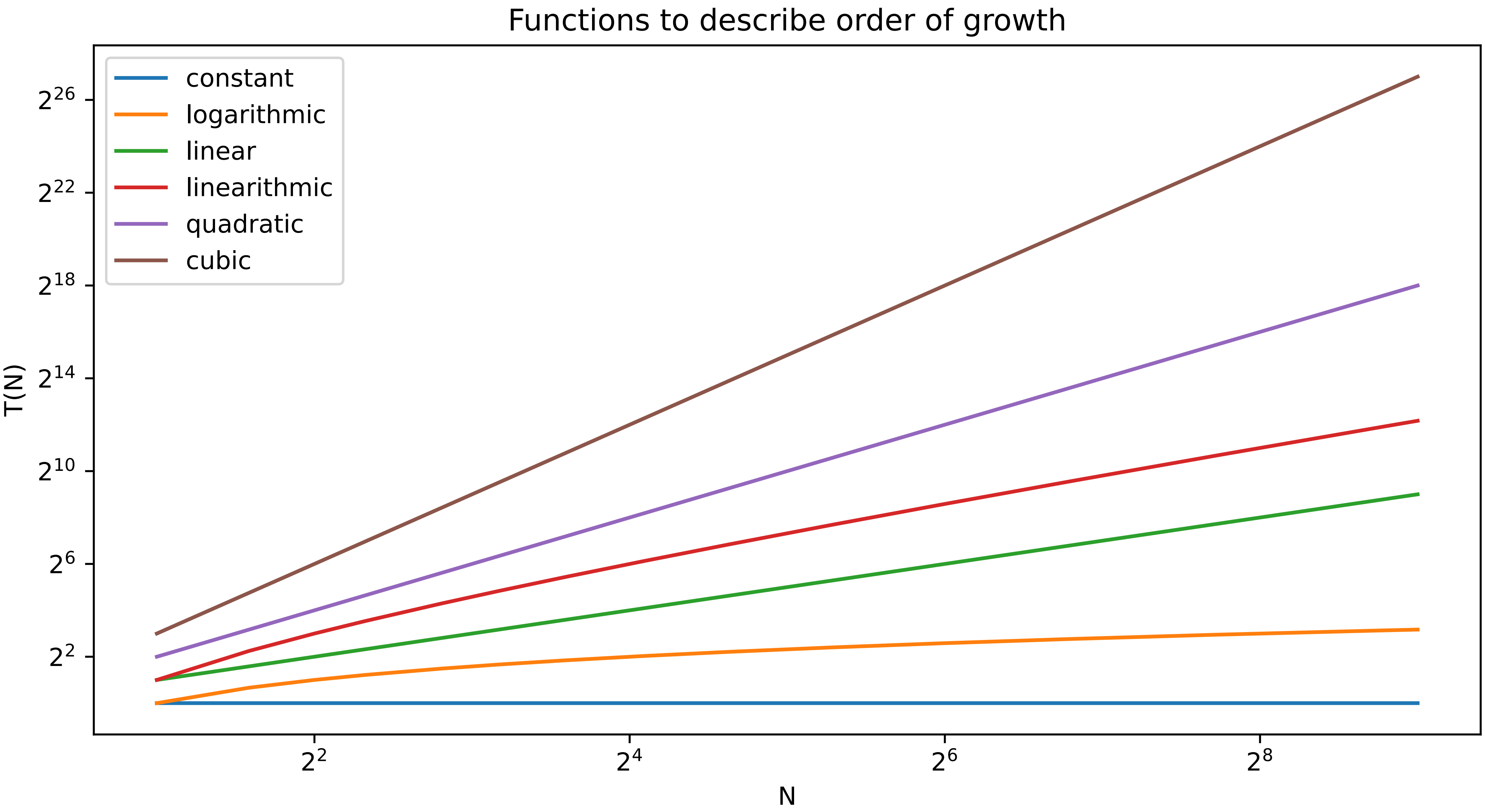
Classifying the order of growth

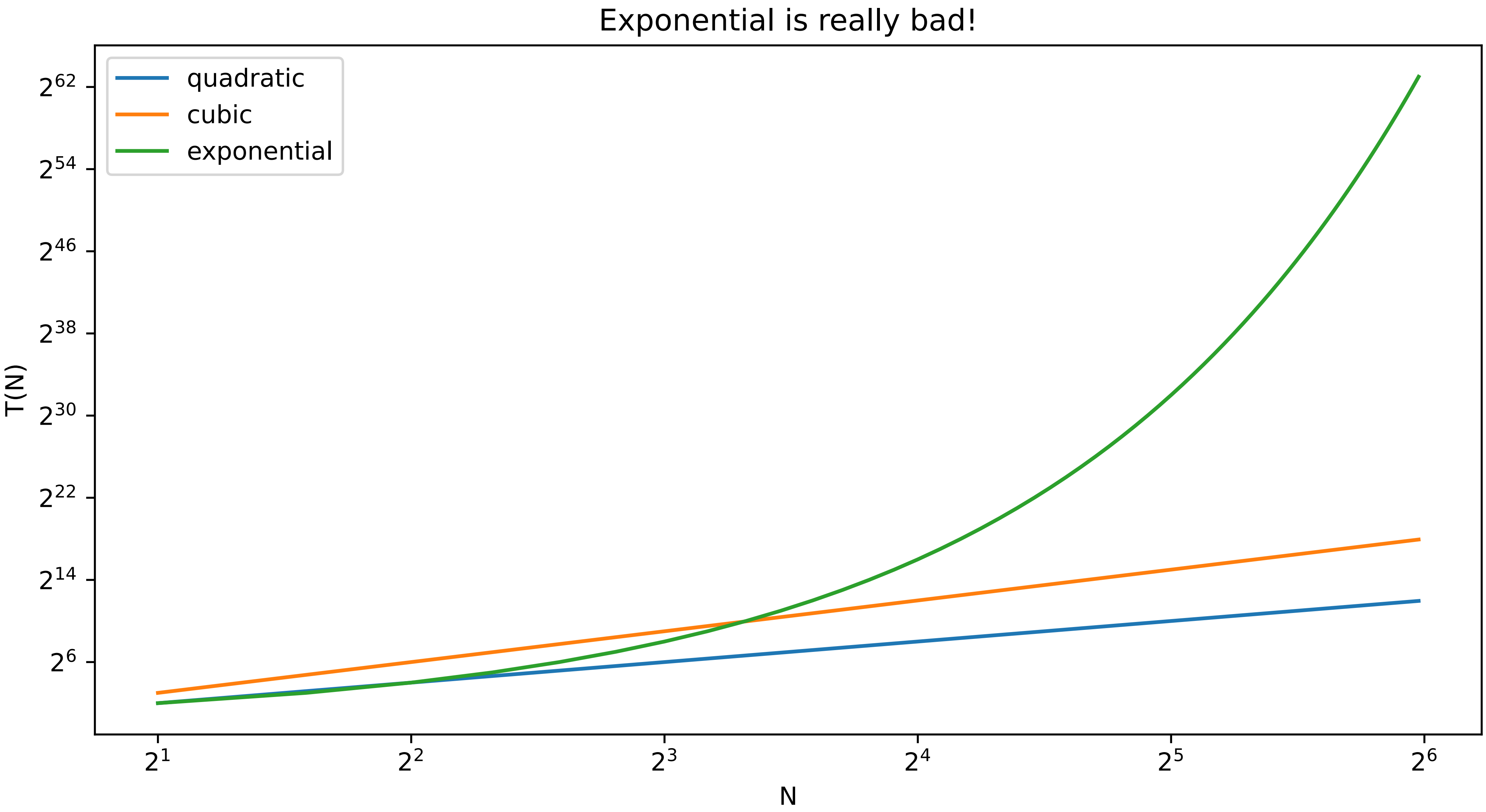
So far

- We have seen the following orders of growth:
  - Constant, 1
  - Linear, N
  - Quadratic, N^2
  - Cubic, N^3
- There are a few more, but a small set of functions is sufficient to describe typical algorithms.

Functions to describe order of growth

order	name	description	T(2N)/T(N)
1	constant	statement	1
$\log N$	logarithmic	divide in half	~ 1
N	linear	loop	2
$N \log N$	linearithmic	divide and conquer	~ 2
N^2	quadratic	double loop	4
N^3	cubic	triple loop	8


Functions to describe order of growth
constant
logarithmic
linear
linearithmic
quadratic
cubic
T(N)
2^{26}
2^{22}
2^{18}
2^{14}
2^{10}
2^{6}
2^{2}
2^{2}
2^{4}
2^{6}
2^{8}
N


Exponential is really bad!
quadratic
cubic
exponential
T(N)
2^{62}
2^{54}
2^{46}
2^{38}
2^{30}
2^{22}
2^{14}
2^{6}
2^{1}
2^{2}
2^{3}
2^{4}
2^{5}
2^{6}
N

Problem solvable in minutes

rate	1970s	1980s	1990s	2000s
1	any	any	any	any
$\log N$	any	any	any	any
N	1e6	1e7	1e8	1e9
$N \log N$	1e5	1e6	1e6	1e8
N^2	100s	1,000s	1,000s	10,000s
N^3	100	100s	1,000	1,000s
2^N	20	20s	20s	30

(Borrowed from Sedgewick)

Theory

3sum

- We concluded that a variant of threesum was $\sim N^3$ using array accesses as the cost model
- What does this mean?
  - Best, worse or average case?
  - What can we expect?

Types of analyses

- Best case, lower bound on cost
    - “Easiest” input
    - Provides a goal for all inputs
- Worst case, upper bound on cost
    - “Most difficult” input
    - Provides a guarantee for all inputs
- Average case, expected cost for random input
    - Provides a wau to predict performance

3sum

- Is there better or worse input for the $\sim N^3$ variant?
  - Not really, we process all input
- So, which type?
  - worst = best = average = $\sim N^3$

Not always the case

```python
def binsearch(l:list[int], x:int) -> int|None:
  low, high = 0, len(l)-1

  while low <= high:
    mid = (low + high) // 2
    if l[mid] == x:
      return mid
    elif l[mid] < x:
      low = mid + 1
    else:
      high = mid - 1

  return None
```

Binary search

- We will save the analysis for later, now we just accept the following
- Best case, constant
- Worse case, $\sim \log N$
- Average case, $\sim \log N$

Commonly used notations

notation	provides	example
Big Theta	asymptotic order of growth	$\Theta(N^2)$
Big Oh	$\Theta(N^2)$ and smaller	$O(N^2)$
Big Omega	$\Theta(N^2)$ and larger	$\Omega(N^2)$

Definitions from the book

- $T(N) = O(f(N))$ if there are positive constants c and n_0 such that $T(N) \leq cf(N)$ when $N \geq n_0$
  - Consider $1000N$ and N^2. There are values for N where $1000 \cdot N$ is larger, but N^2 *grows* faster
  - There is some point, n_0, after which N^2 is always larger than $1000N$
- If $T(N) = 1000N$ and $f(N) = N^2$, $T(N) \leq cf(N)$ when
  - $c = 1$ and $n_0 = 1000$, $c = 100$ and $n_0 = 10$

Definitions from the book

- $T(N) = \Omega(g(N))$ if there are positive constants c and n_0 such that $T(N) \geq cg(N)$ when $N \geq n_0$
- $T(N) = \Theta(h(N))$ if and only if
  - $T(N) = O(h(N))$ and
  - $T(N) = \Omega(h(N))$

Note

- Ω and O are lower and upper bounds
  - $N^2 = O(N^3)$
  - $N^3 = \Omega(N^2)$
- So, $T(N) = O(f(N))$ guarantees that $T(N)$ grows at a rate no faster than $f(N)$
- This implies that $f(N) = \Omega(T(N))$, i.e., that $T(N)$ is a lower bound on $f(N)$

Tighter bounds

- If $f(N) = 2N$, then technically
  - $f(N) = O(N^2)$,
  - $f(N) = O(N^3)$ and so on.
- $f(N) = O(N)$ is the best option

Conventions

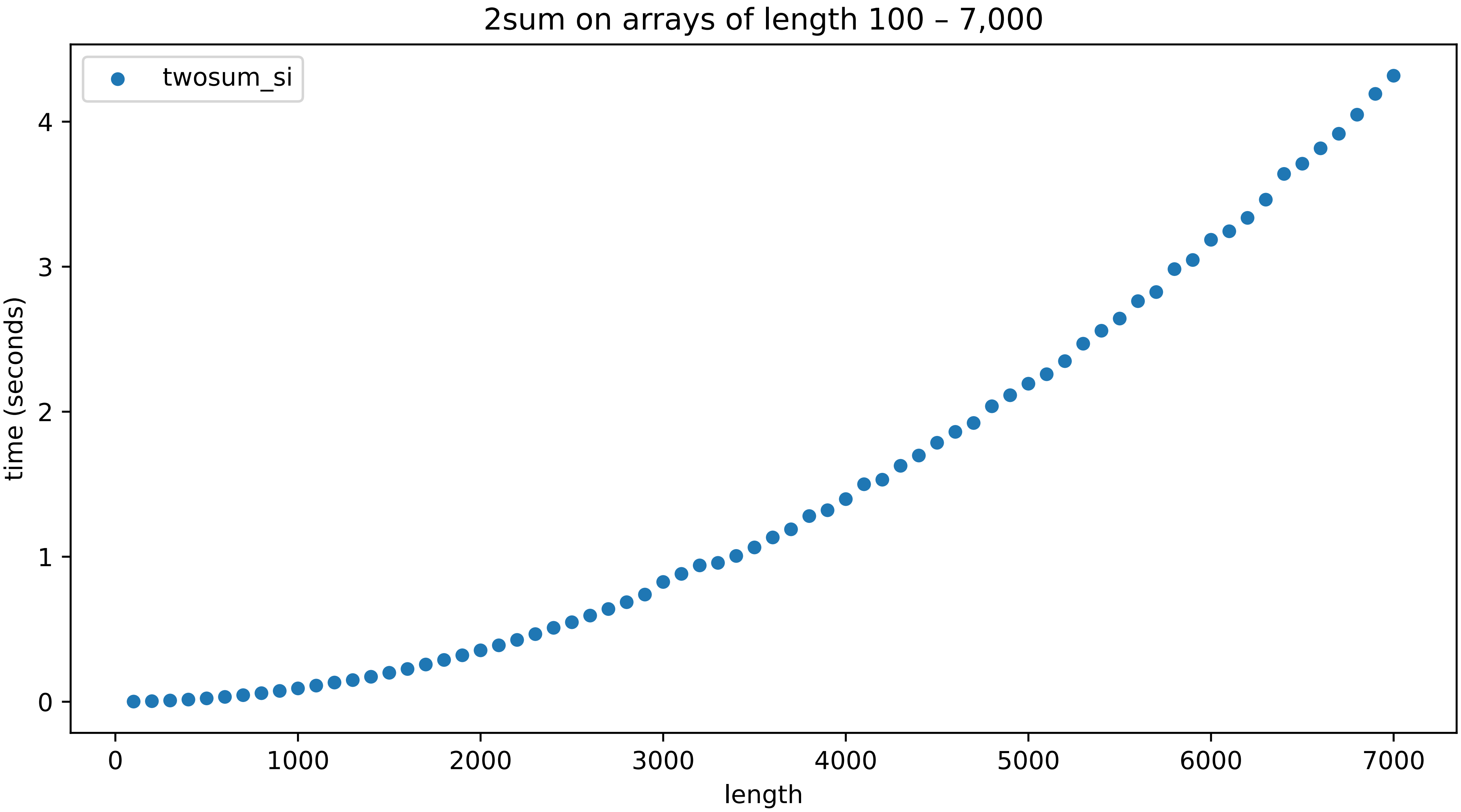
- Do not include lower order terms and constants
  - if $f(N) = 2N$, write $f(N) = O(N)$
  - if $f(N) = N^3 + N + 7$, write $f(N) = O(N^3)$

Remember

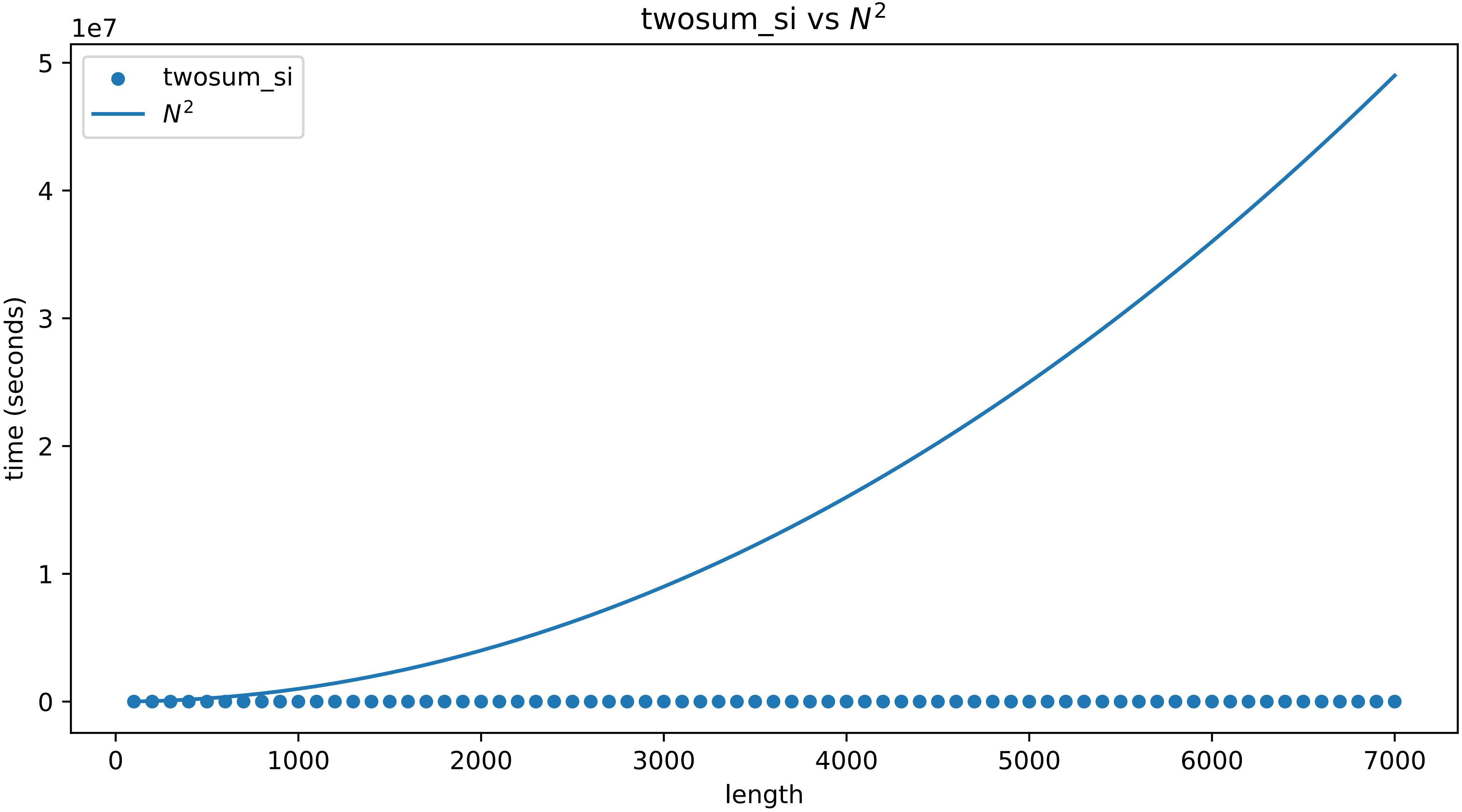
```python
def twosum_si(l:list[int]) -> list[tuple[int, int]]:
  res = []
  for i, vi in enumerate(l):
    for vj in l[i+1:]:
      if vi + vj == 0:
        res.append((vi, vj))

  return res
```

- We claimed that this had an quadratic runtime based on empirical estimation

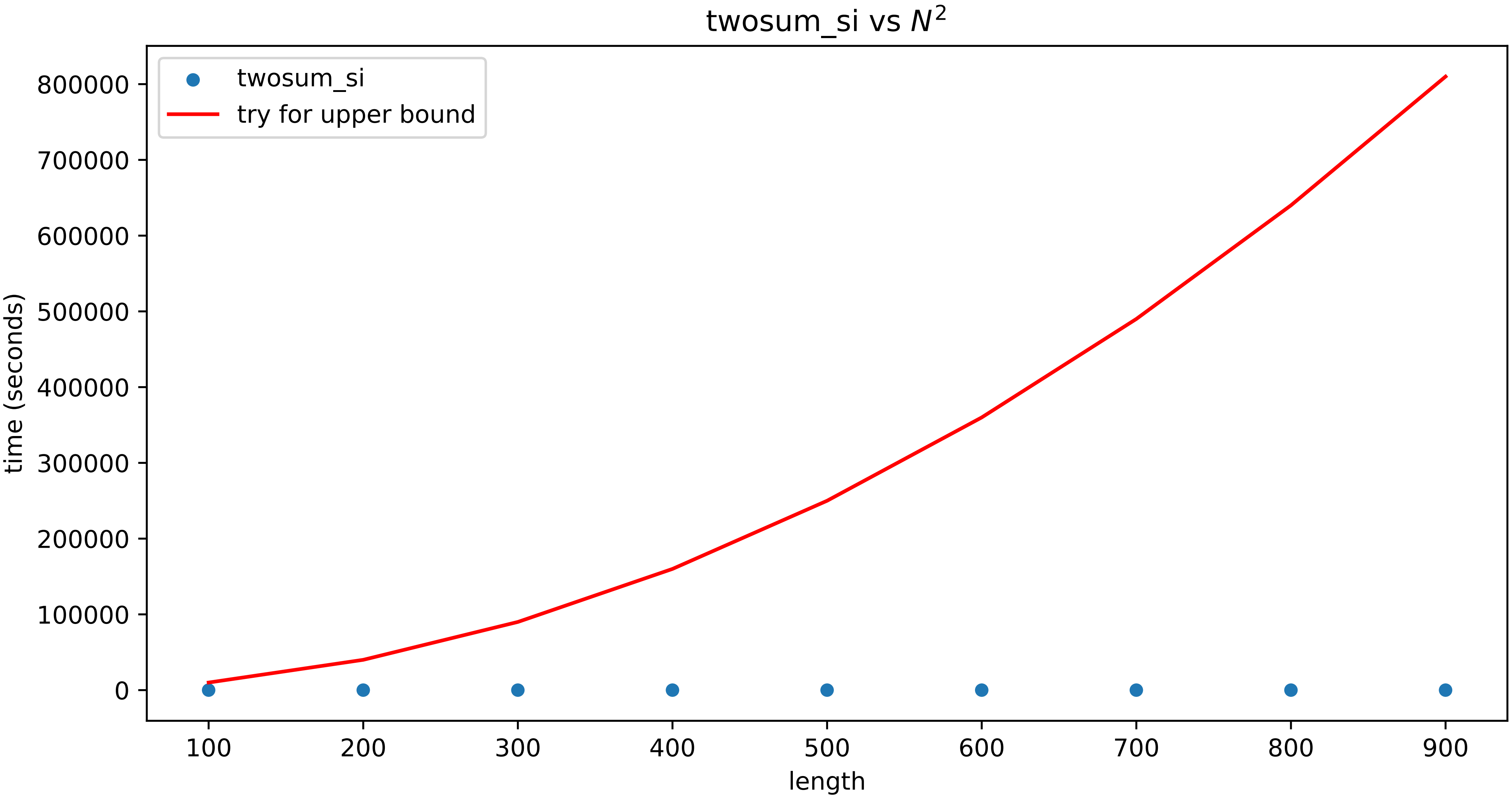
Analysing this version of twosum



Analysing this version of twosum



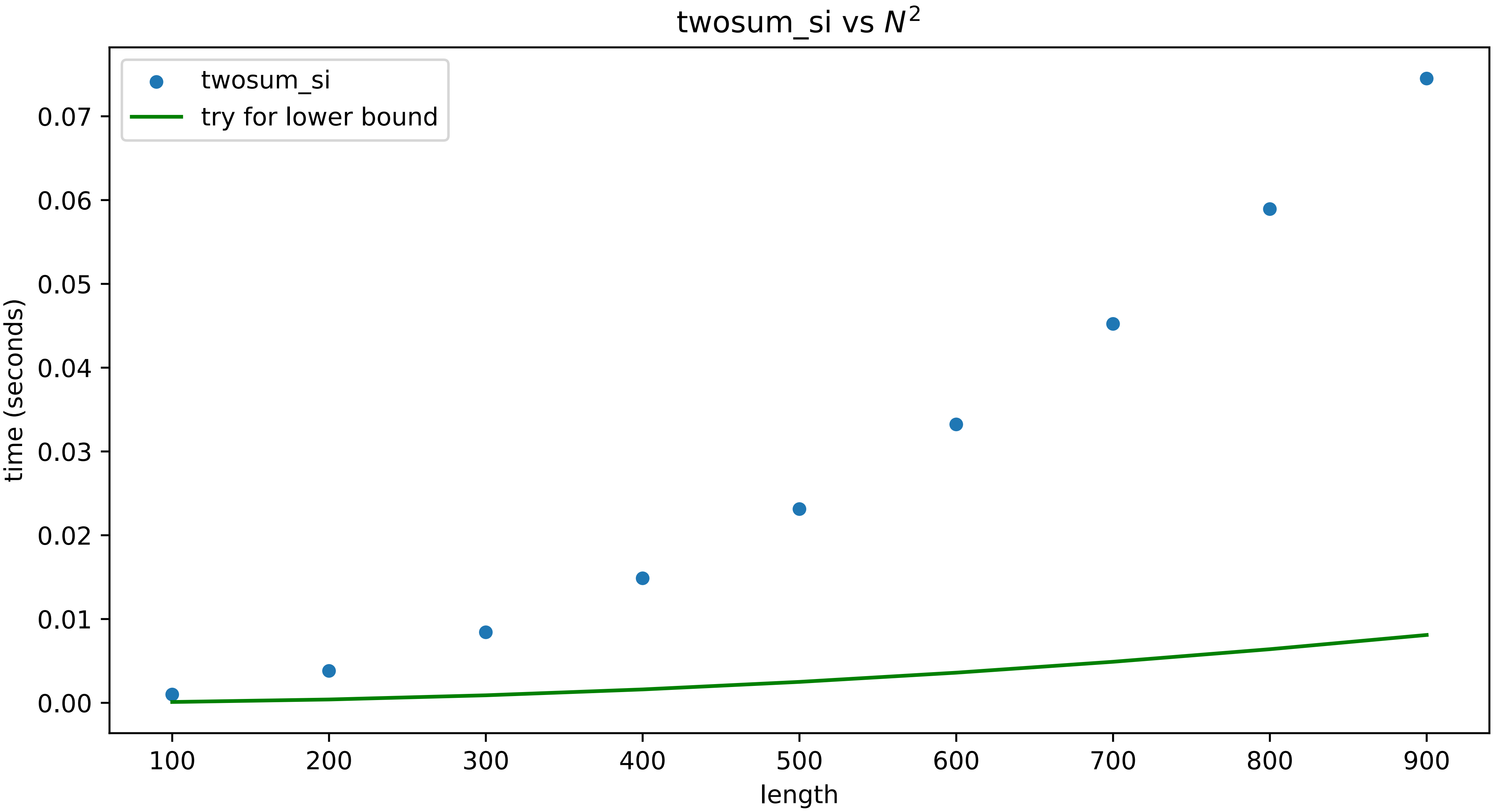
Analysing this version of twosum



Analysing this version of twosum

- Based on visual analysis, we can tell that the growth of N^2 is always larger than twosum_si when $n_0 = 100$ and $c = 1$.
    - We do not have any measurements for smaller arrays, so perhaps we could find a smaller n_0, but does not matter.
- So, twosum_si is $O(N^2)$

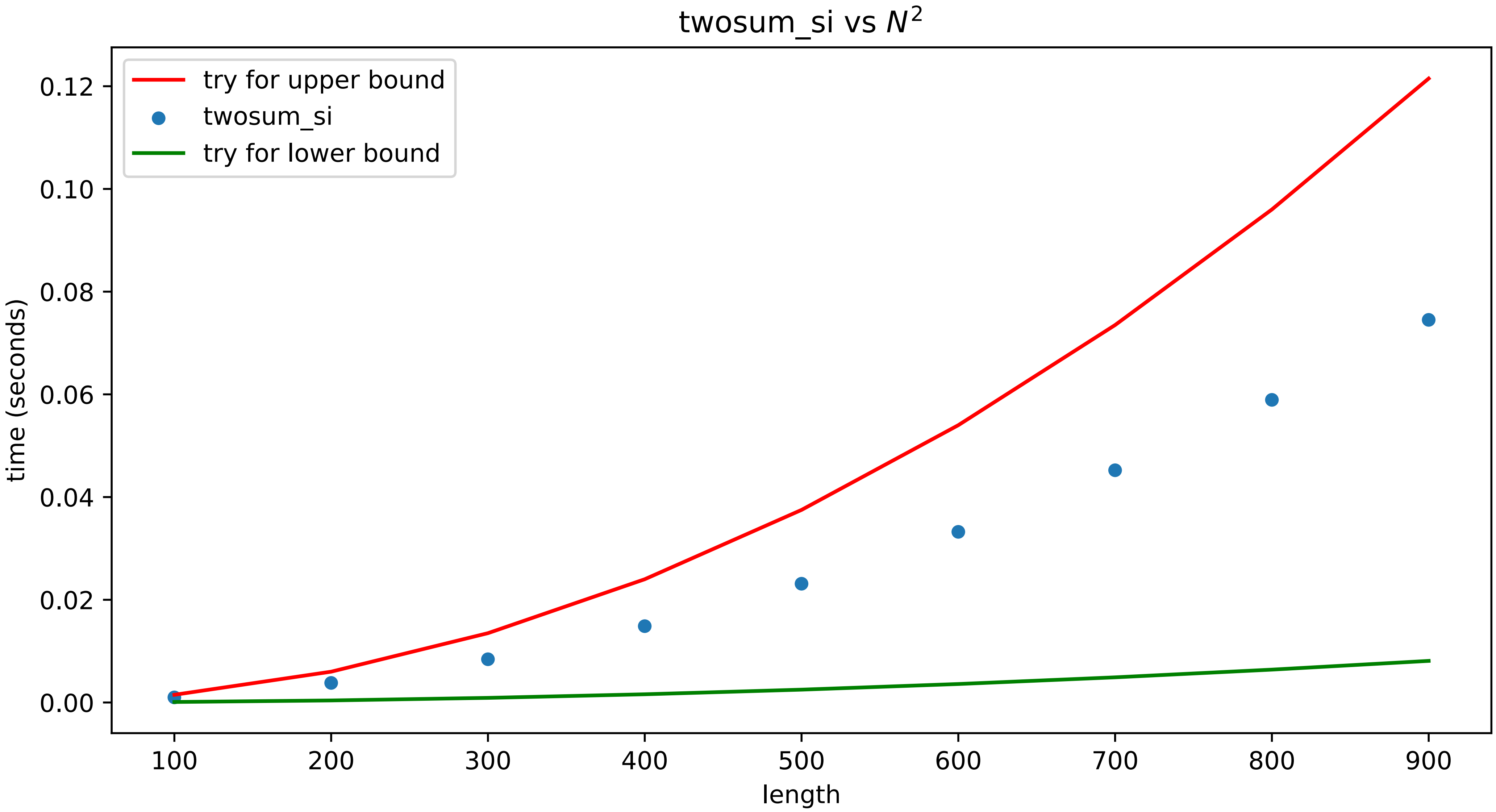
Analysing this version of twosum



Analysing this version of twosum

- Again, based on the visual analysis, we can tell that the growth of cN^2 is always smaller than twosum_si when $n_0 = 100$ and $c = 10^{-8}$
  - We can also check with the values
    - $T(100)$ for twosum_si is 0.000994
    - $T(100)$ for $10^{-8} \cdot N^2$ is 0.0001
- So, twosum_si is $\Omega(N^2)$

Analysing this version of twosum



Analysing this version of twosum

- Since twosum_si is $O(N^2)$ and $\Omega(N^2)$
  - it is also $\Theta(N^2)$
- Does that mean that 2sum is $\Theta(N^2)$
  - No, just the algorithm for 2sum that we analyzed
  - We know that there is a version that is $O(N)$ (based on empirical analysis)
  - We can still say that 2sum is $O(N^2)$

A grain of salt

- We can sometimes empirically show that the analysis is an overestimation
  - We might find a tighter bound
  - Or, average case is much better than the worse case
- In some cases, average case analysis is extremely complex
- The worse case bound is the best analytical result known

Reading instructions

Reading instructions

- Ch. 2