

香港中文大學(深圳)
The Chinese University of Hong Kong, Shenzhen



SCHOOL OF
DATA SCIENCE
數據科學學院

Introduction to Computer Science: Programming Methodology

Lecture 8
Data Structure and Algorithm - Intro

Tongxin Li
School of Data Science

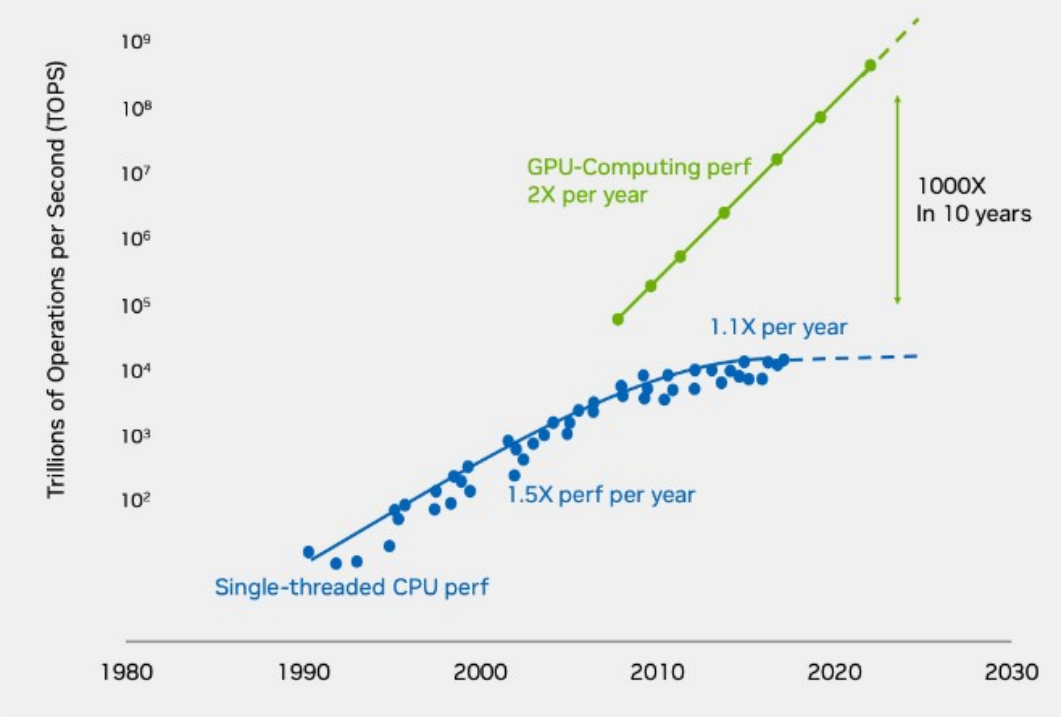
Data Structures and Algorithms

- A data structure organizes data and lets us access it.

- An algorithm is a step-by-step procedure that completes a task in a finite amount of time.

Why Study Algorithms?

Moore’s law describes the historical doubling of transistor counts roughly every one to two years.



Why Study Algorithms?

- Faster hardware improves performance.
- Better algorithms can produce even larger gains.
- Algorithms support every branch of computer science and many technological innovations.

Algorithms Beyond Computing

- Algorithms offer a way to study processes in quantum mechanics, markets, and evolution.
- Designing algorithms develops problem-solving skills.
- The challenge is to find a better way to solve a task.

The Algorithm Designer's Mantra

“Perhaps the most important principle for the good algorithm designer is to refuse to be content”

Aho, Hopcroft, and Ullman
The Design and Analysis of Computer Algorithms, 1974

AI and the Limits of Algorithms

- OpenAI reports that an internal AI model produced a proof claiming the Exponential Time Hypothesis (ETH) is false.
- ETH predicts that 3-SAT requires exponential time in the worst case.

- OpenAI reports verification in Lean. The proof is public for independent examination.

How much faster could a better algorithm be?

[1] OpenAI (26 Oct 2026) [2] Science [3] MLQ

Example: Sorting

Sort prices from lowest to highest.

Input `[30, 10, 40, 20, 50]`

Output `[10, 20, 30, 40, 50]`

n counts the prices: here, n = 5.

How can we sort a million prices efficiently?
We will compare methods, including merge sort.

What Makes a Good Algorithm?

- Running time: how many operations does it need?
- Space usage: how much memory does it need?
- Both depend on the input size.

Measuring Running Time

```
from time import time

startTime = time()

for i in range(1, 20000):
    if i % 10 == 0:
        print(i)

endTime = time()

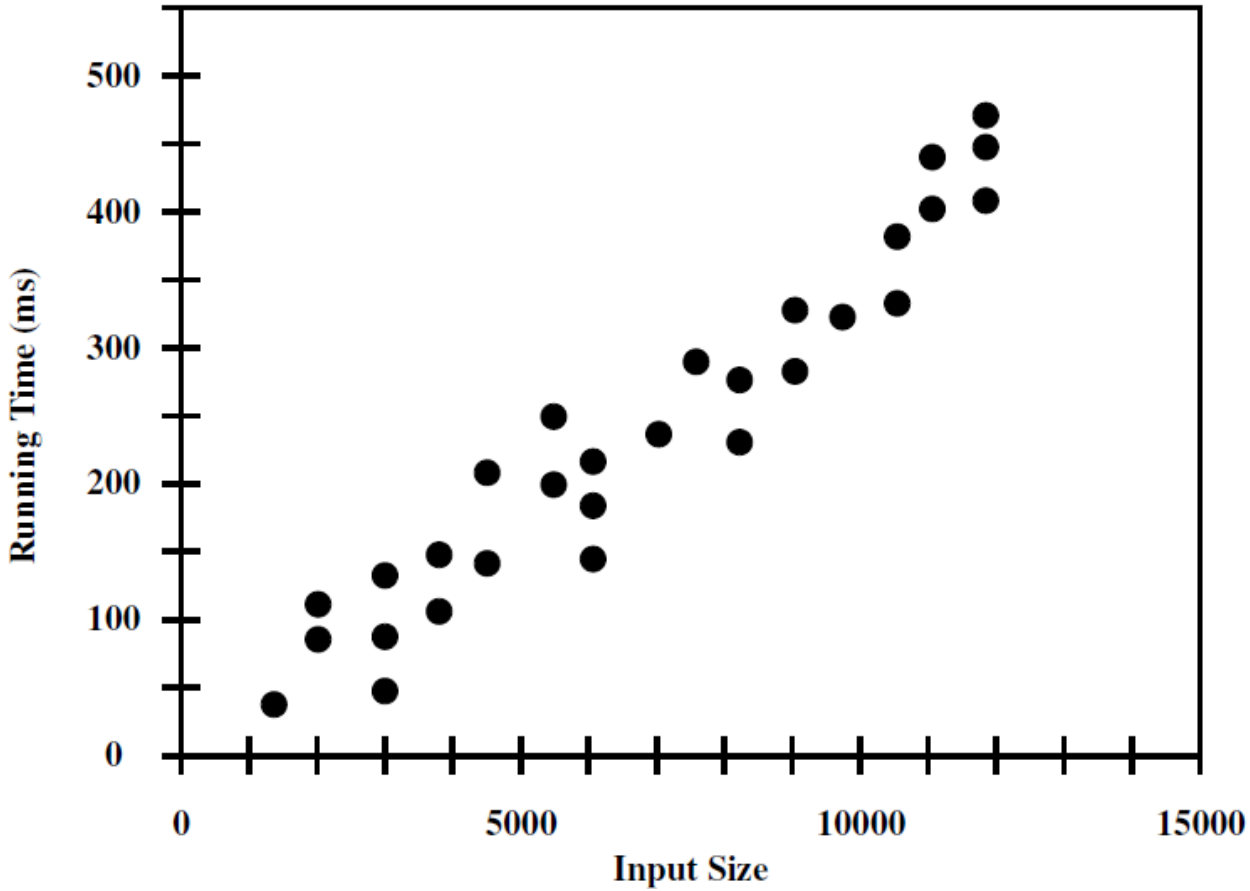
print('The time elapsed is:',
      endTime - startTime, 'seconds')
```

The measured time includes printing the numbers.

Plotting Running Time

Test inputs of different sizes.

Plot the time for each run.



Limits of Experimental Analysis

- A fair comparison needs the same hardware and software environment.
- Experiments cover only the inputs we test.
- We must implement the algorithm before timing it.

Counting Primitive Operations

Count basic operations in the algorithm description.

- Assign a value
- Read a variable
- Do arithmetic
- Compare two values
- Access a list element
- Call a function
- Return from a function

A call's body is counted separately.

Input Size and Running Time

`n = input size`

For sorting, n counts the elements in the list.

Five prices means n = 5, whatever their values or order.

`f(n) = number of primitive operations`

(for the case being analyzed)

We study how the operation count grows with n.

Best, Average, and Worst Cases

Find 50 in a list that contains it.
Check from left to right, stopping when you find 50.

Both lists contain 5 prices: n = 5 in both cases.

Case	Input	Checks
Best	`[50, 10, 20, 30, 40]`	1
Worst	`[10, 20, 30, 40, 50]`	5

Average case: if 50 is equally likely in each position,
(1 + 2 + 3 + 4 + 5) / 5 = 3 checks.

Seven Common Growth Rates

Constant	1
Logarithmic	log n
Linear	n
Linearithmic	n log n
Quadratic	n^2
Cubic	n^3
Exponential	2^n

Asymptotic Analysis

- Focus on how running time grows with input size n.
- Ignore machine-specific details and constant factors.
- Compare algorithms when n becomes large.

Big-Oh Notation

$f(n)$ is $O(g(n))$ if constants $c > 0$ and $n_0 \geq 1$ exist such that

$$f(n) \leq c \cdot g(n) \text{ for all } n \geq n_0$$

Example: $8n + 5 \leq 13n$ for $n \geq 1$,
so $8n + 5$ is $O(n)$.

What Big-Oh Means

- Big-Oh describes an upper bound on growth.

- $f(n)$ eventually stays below a constant multiple of $g(n)$.

- Use it for running time or memory usage.
  Always state what the input size n measures.

Simplifying a Growth Bound

Ignore constant factors and lower-order terms.

$$5n^4 + 3n^3 + 2n^2 + 4n + 1 \qquad O(n^4)$$

$$2^{n+2} = 4 \cdot 2^n \qquad O(2^n)$$

$$2n + 100\log n \qquad O(n)$$

Use the closest useful growth bound.

Comparing Two Algorithms

Two algorithms solve the same problem.

Algorithm A

$T_a(n) = 100n$

Algorithm B

$T_\beta(n) = n^2$

Which algorithm is faster? Does n matter?

Comparing Two Algorithms

Two algorithms solve the same problem.

Algorithm A

$T_a(n) = 100n$

Algorithm B

$T_\beta(n) = n^2$

$n < 100$: B is faster. $n > 100$: A is faster.

A grows more slowly and wins for large n.

Comparing Growth Rates

As n grows, the gap between growth rates widens.

n	$\log_2 n$	n	$n \log_2 n$	n^2	n^3	2^n
8	3	8	24	64	512	256
16	4	16	64	256	4,096	65,536
32	5	32	160	1,024	32,768	4.29×10^9
64	6	64	384	4,096	262,144	1.84×10^{19}
128	7	128	896	16,384	2,097,152	3.40×10^{38}
256	8	256	2,048	65,536	16,777,216	1.16×10^{77}
512	9	512	4,608	262,144	134,217,728	1.34×10^{154}

Polynomial and Exponential Time

Polynomial time

$n, n^2, n^3, \ldots$

Exponential time

$2^n, 3^n, \ldots$

This distinction helps describe tractability.
A polynomial-time algorithm can still be expensive when its constants or degree are large.

Example: Finding the Minimum

```
smallest_so_far = None
print('Before', smallest_so_far)

for num in [9, 39, 21, 98, 4, 5, 100, 65]:
    if smallest_so_far is None:
        smallest_so_far = num
    elif num < smallest_so_far:
        smallest_so_far = num
    print(smallest_so_far, num)

print('After', smallest_so_far)
```

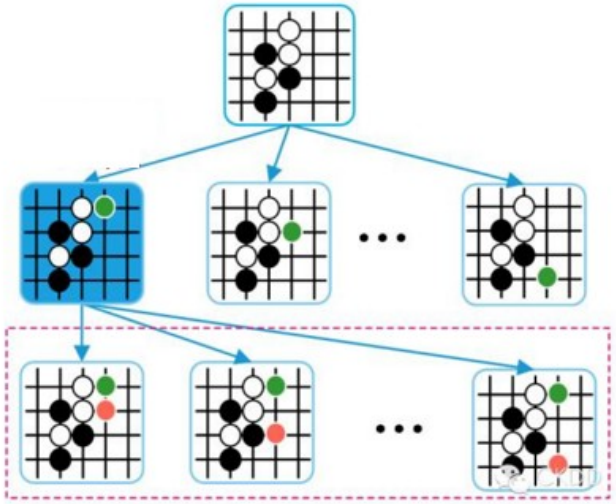
What is the time complexity?

Why AlphaGo Was Remarkable



Why Brute Force Is Difficult

If each position has b possible moves,
searching d moves ahead can involve about b^d paths.



The search grows exponentially with depth.

Learning to Play Minecraft

nature

NEWS | 02 April 2025

AI masters *Minecraft*: DeepMind program finds diamonds without being taught

The Dreamer system reached the milestone by 'imagining' the future impact of possible decisions.

Recursion

- A recursive function calls itself.
- Each call works on a smaller version of the problem.
- A base case gives an answer without another call.

Example: The Factorial Function

For a nonnegative integer n:

$0! = 1$

$n! = n \times (n - 1) \times \cdots \times 2 \times 1 \quad (n \geq 1)$

n! counts the arrangements of n distinct items.

3! = 6: ABC ACB BAC BCA CAB CBA

Base Case and Recursive Case

Base case

Return an answer directly. No recursive call.

Recursive case

Call the function on a smaller problem.
Move closer to the base case.

Factorial: Recursive Definition

$5! = 5 \times 4! = 120$

Base case: $0! = 1$

Recursive case: $n! = n \times (n - 1)!$

How can we write this definition in Python?

Solution: Factorial

```
def facFunc(n):
    if n < 0:
        print('Invalid input.')
        return None
    elif n == 0:
        return 1
    else:
        return n * facFunc(n - 1)


print(facFunc(4))   # 24
print(facFunc(0))   # 1
```

How Python Handles Recursive Calls

Each call has its own frame: parameters, local variables, and the place to resume after a call.

```
facFunc(2)
n = 2
return 2 × ?
```



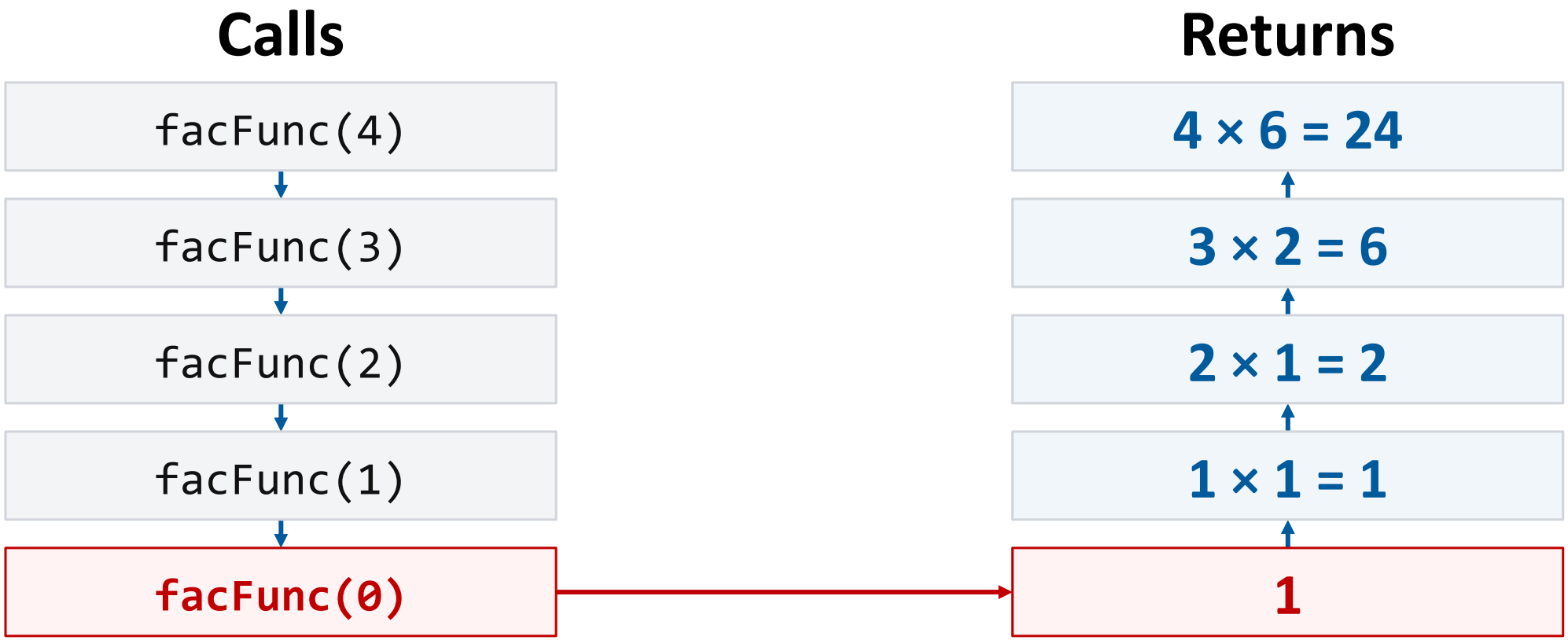
```
facFunc(1)
n = 1
return 1 × ?
```

The caller waits, then resumes with the returned value.

Inception



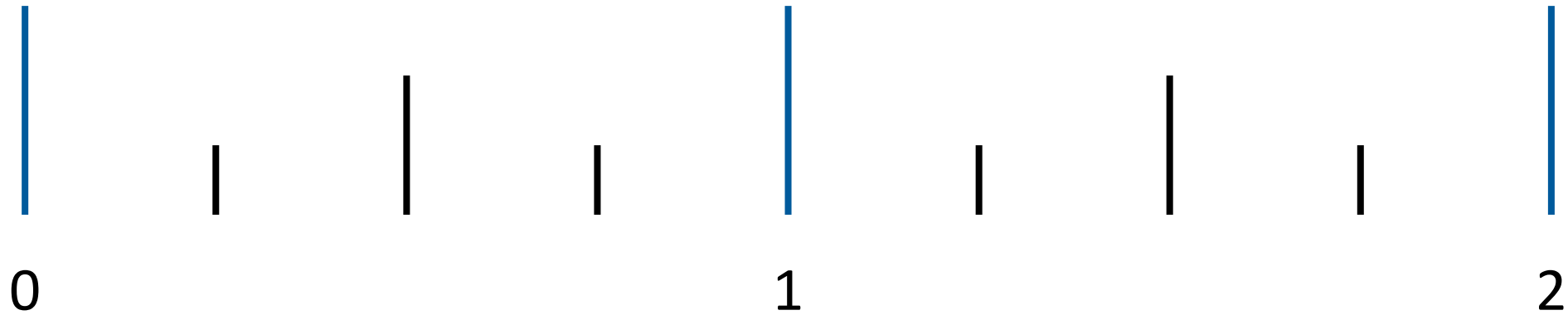
Tracing Factorial



Base case: n == 0 returns 1

Example: Drawing an English Ruler

Whole inches use the longest ticks.
Half inches use shorter ticks.
Quarter inches use still shorter ticks.



The Recursive Ruler Pattern

An interval with central tick length L ≥ 1 contains:

1. Smaller interval
2. Central tick
3. Smaller interval

L – 1 → L → L – 1

At L = 0, draw nothing.

Solution: Drawing an English Ruler

```
def draw_line(length, label=''):
    line = '-' * length
    if label:
        line += ' ' + label
    print(line)

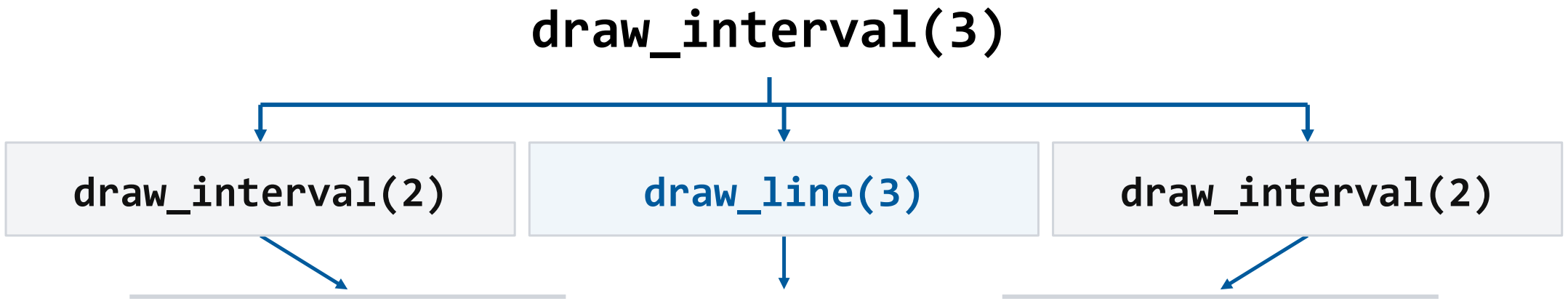
def draw_interval(length):
    if length > 0:
        draw_interval(length - 1)
        draw_line(length)
        draw_interval(length - 1)
```

```
def draw_ruler(inches, major):
    draw_line(major, '0')
    for j in range(1, 1 + inches):
        draw_interval(major - 1)
        draw_line(major, str(j))

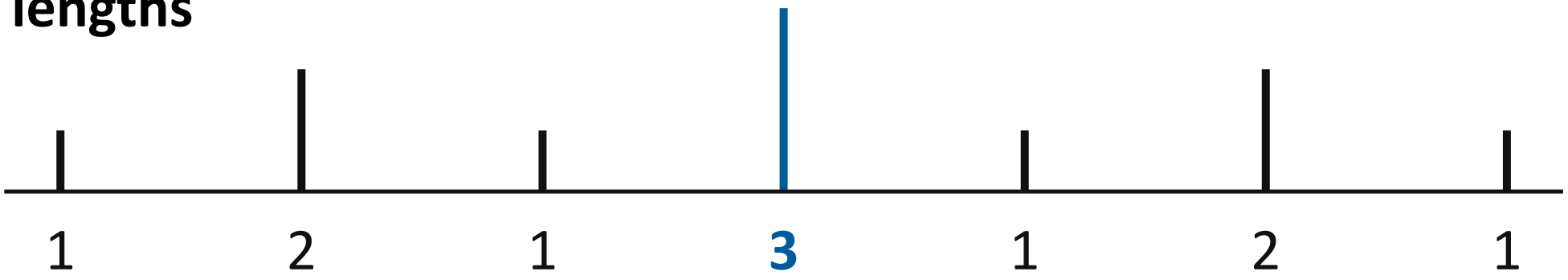

draw_ruler(1, 3)
```

```
--- 0
-
--
-
--- 1
```

Tracing the English Ruler



Tick lengths



Base case: draw_interval(0) draws nothing.

Example: Binary Search

Binary search needs a sorted sequence.

With unsorted data, sequential search checks items one by one until it finds the target or reaches the end.

2	4	5	7	8	9	12	14	17	19	22	25	27	28	33	37

Why Sorted Order Helps

Choose an element at index j.



Sorted order lets us rule out an entire side.

The Search Interval

Candidates are the entries from low to high, inclusive.

Initially: low = 0, high = n − 1

```
mid = (low + high) // 2
```

Compare target with data[mid].

The Three Search Cases

`target == data[mid]`

Found: return True

`target < data[mid]`

Search low … mid – 1

`target > data[mid]`

Search mid + 1 … high

Solution: Binary Search

```
def binarySearch(data, target, low, high):
    if low > high:
        return False
    mid = (low + high) // 2
    if target == data[mid]:
        return True
    if target < data[mid]:
        return binarySearch(
            data, target, low, mid - 1)
    return binarySearch(
        data, target, mid + 1, high)

data = [1, 3, 5, 6, 16, 78, 100, 135, 900]
print(binarySearch(data, 16, 0, len(data) - 1))
```

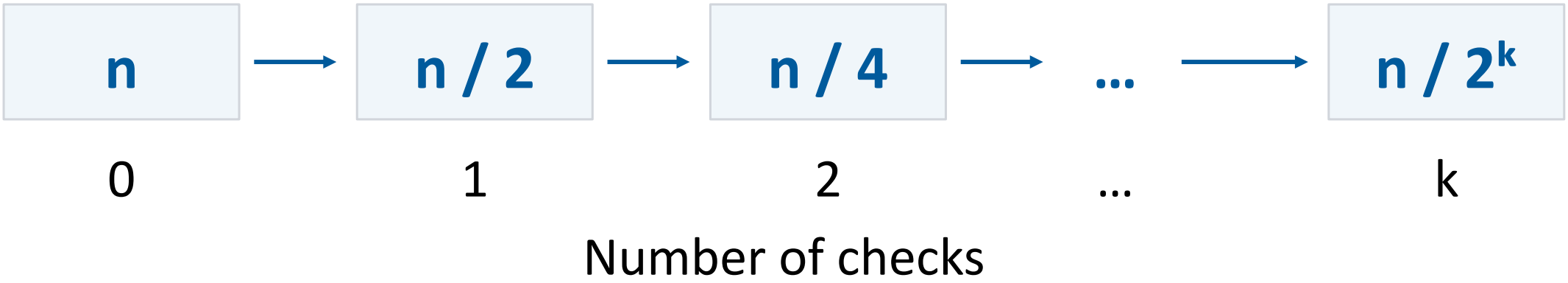
The Cost of Binary Search

For a sorted sequence of n elements,
binary search takes O(log n) time in the worst case.

Why does halving the search interval help?

Why the Cost Is Logarithmic

At most half remain after each check.



After k checks: at most $n / 2^k$ candidates remain.

Choose $k > \log_2 n$: no candidates remain.

Time complexity: O(log n)

Changing the Logarithm Base

For any fixed bases $a > 1$ and $b > 1$:

$$\log_a n = \frac{\log_b n}{\log_b a}$$

The denominator is a fixed constant.

Only a constant factor changes.

Therefore, the complexity is O(log n).