



香港中文大學(深圳)

The Chinese University of Hong Kong, Shenzhen

Introduction to Computer Science: Programming Methodology

Lecture 8 Data Structure and Algorithm - Intro

Prof. Pinjia He
School of Data Science



データ

Data structure and algorithm

- A **data structure** is a systematic way of organizing and accessing data *e.g. a list*
- An **algorithm** is a step-by-step procedure for performing some task in a finite amount of time.

✓ $A \rightarrow B$

$A \rightarrow C \rightarrow B$

$A \rightarrow C \rightarrow D \rightarrow B$

Why study data structure and algorithm?

- Important for **all other branches** of computer science
- Plays a **key role** in modern technological innovation
- **Moore's law** predicts that the density of transistors in integrated circuits would continue to double every 1 to 2 years
- However, in many areas, performance gains due to the improvements in algorithms have **greatly exceeded** even the dramatic performance gains due to increased processor speed

晶体管密度

处理速度

Why study data structure and algorithm?

- Provide novel “lens” on processes outside of computer science and technology, such as quantum mechanics, economic markets, evolution
- Challenging (good for your brain!!) and funny

Example: Integer Multiplication

- **Inputs:** two n -digits number x and y
- **Output:** the product of x and y
- **Primitive operations:** add or multiply 2 single digit numbers

The algorithm designer's mantra

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

拒绝满足

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

How do we define a “good” algorithm?

- The primary analysis of algorithms involves characterizing the running times and space usage of algorithms and data structure operations
- Running time is a natural measure of “goodness,” since time is a precious resource—computer solutions should run as fast as possible
- Space usage is another major issue to consider when we design an algorithm, since we only have limited storage spaces

Measuring the running time experimentally

time()

```
from time import time
```

```
startTime = time()
```

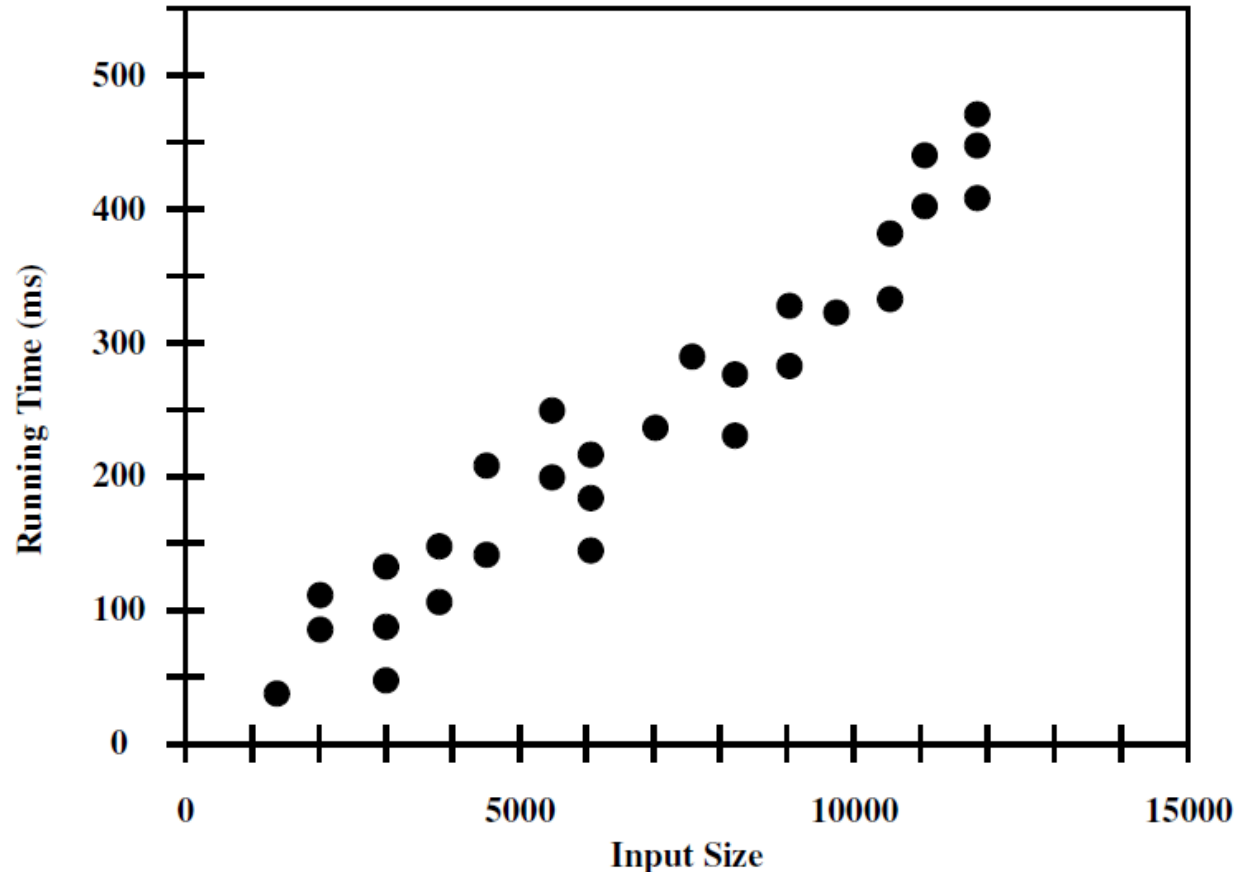
```
for i in range(1, 20000):  
    if i%10 == 0:  
        print(i)
```

```
endTime = time()
```

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


Visualize the running time

- Running time and space usage are dependent on the size of the input
- Perform independent experiments on many different test inputs of various sizes
- Visualize the results by plotting the performance of each run of the algorithm as a point



Challenges of experimental analysis

- Experimental running times of two algorithms are difficult to directly compare unless the experiments are performed in the same hardware and software environments
- Experiments can be done only on a limited set of test inputs; hence, they leave out the running times of inputs not included in the experiment (and these inputs may be important)
- An algorithm must be fully implemented in order to execute it to study its running time experimentally

Principle of algorithm analysis 1: Counting primitive operations

原始操作

- To analyse the running time of an algorithm without performing experiments, we perform an analysis directly on a high-level description of the algorithm

approximation.

- We define a set of primitive operations such as the following:

- ✓ Assigning an identifier to an object
- ✓ Determining the object associated with an identifier
- ✓ Performing an arithmetic operation (for example, adding two numbers)
- ✓ Comparing two numbers
- ✓ Accessing a single element of a Python list by index
- ✓ Calling a function (excluding operations executed within the function)
- ✓ Returning from a function.

标识符

演算

通过索引访问

若为n个循环, $1 * n$.

①

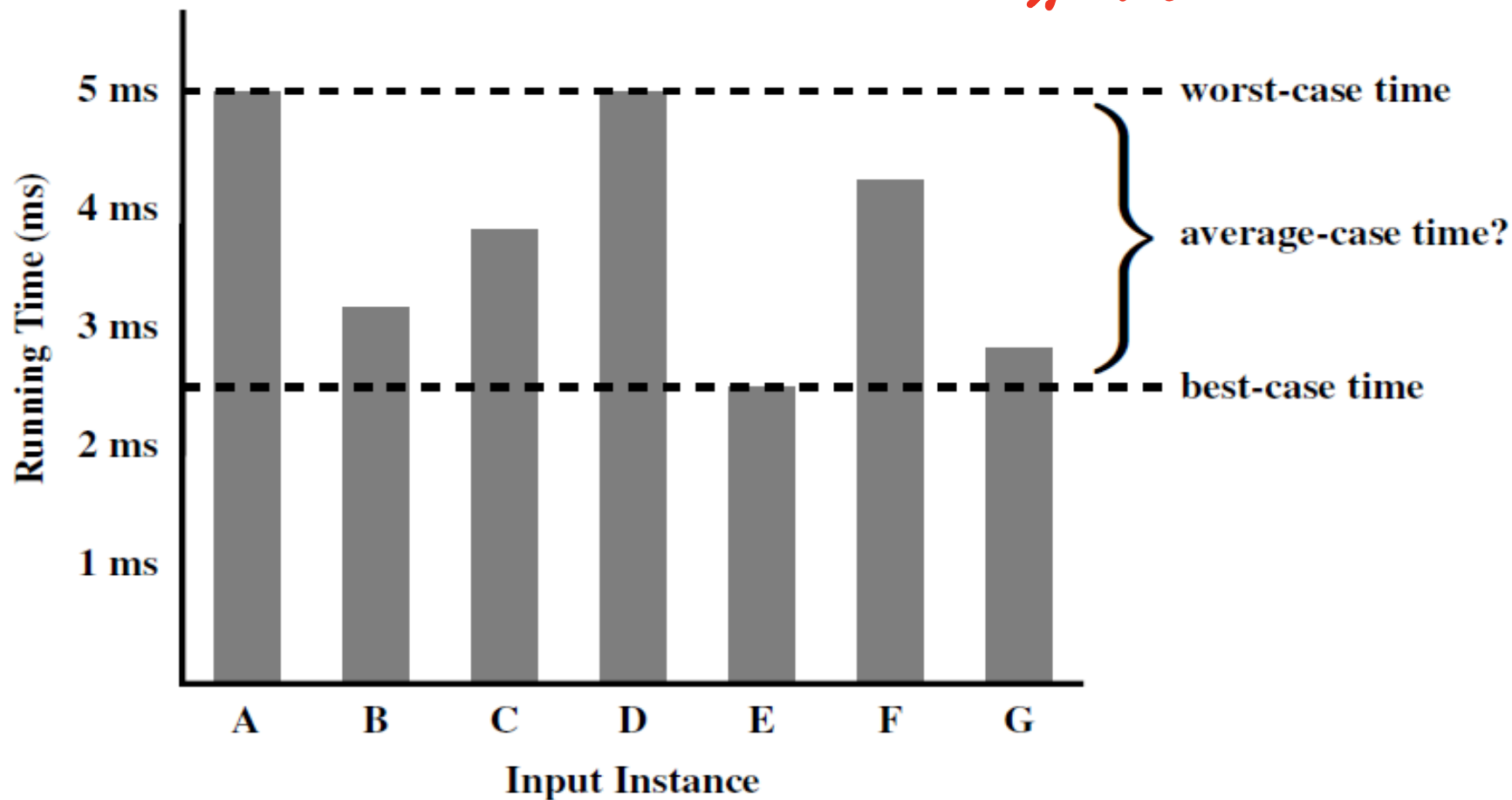
Principle of algorithm analysis 2: Measuring Operations as a Function of Input Size

- To capture the order of growth of an algorithm's running time, we will associate, with each algorithm, a function $f(n)$ that characterizes the number of primitive operations that are performed as a function of the input size n

执行基本操作数量表征为 $f(n)$ 。

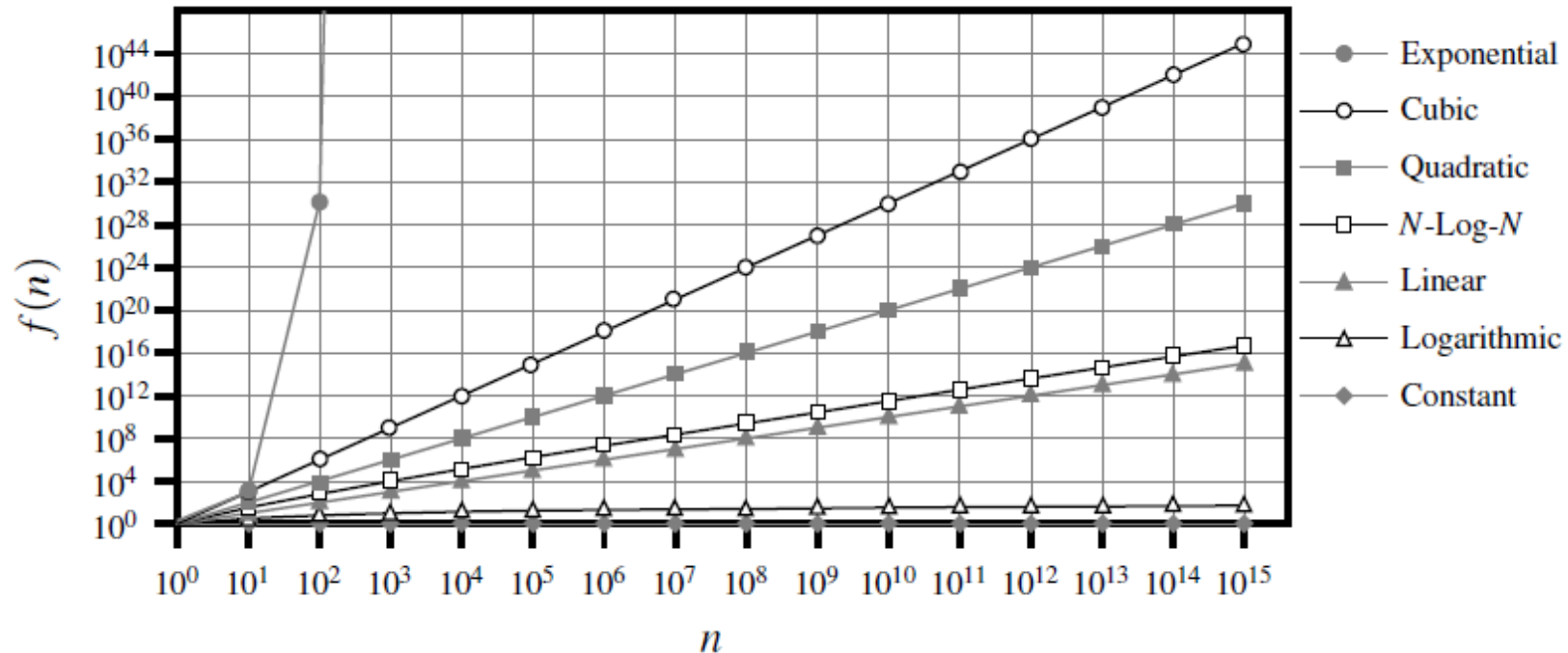
Principle of algorithm analysis 3: Focusing on the Worst-Case Input

(用时最长)



The 7 functions used in algorithm analysis

- We may use the following 7 functions to measure the time complexity of an algorithm: constant, logarithm, linear, $N\text{-log-}N$, quadratic, cubic and other polynomials, exponential



Asymptotic Analysis 渐进分析.

- In algorithm analysis, we focus on the growth rate of the running time as a function of the input size n , taking a "big-picture" approach
- Vocabulary for the analysis and design of algorithms
- "Sweet spot" for high-level reasoning about algorithms
- Coarse enough to suppress unnecessary details, e.g. architecture/language/compiler...
- Sharp enough to make meaningful comparisons between algorithms

The big Oh notation

- Let $f(n)$ and $g(n)$ be functions mapping positive integers to positive real numbers.
- We say that $f(n)$ is $O(g(n))$ if there is a real constant $c > 0$ and an integer constant $n_0 \geq 1$ such that

(for large enough n) $f(n) \leq c g(n)$, for $n \geq n_0$

- This definition is often referred to as the “big-Oh” notation
- **Example:** The function $8n+5$ is $O(n)$.

$$f(n) = \overline{8n+5} \quad g(n) = n.$$

$$8n+5 \leq Cn \rightarrow C \geq 8 + \frac{5}{n}.$$

for $n \geq 1$.

$$C \geq 13.$$

The big Oh notation

- The big-Oh notation allows us to say that a function $f(n)$ is “less than or equal to” another function $g(n)$ up to a constant factor and **in the asymptotic sense** as n grows toward infinity

在 n 趋近无穷的渐近意义上。

- The big-Oh notation is used widely to characterize **running times** and **space bounds** in terms of some parameter n , which varies from problem to problem, but is always defined as a chosen measure of the “**size**” of the problem

Some Properties of the Big-Oh Notation

- The big-Oh notation allows us to ignore constant factors and lower-order terms and focus on the main components of a function that affect its growth
- **Example:** $5n^4 + 3n^3 + 2n^2 + 4n + 1$ is $O(n^4)$
- **Example:** 2^{n+2} is $O(2^n)$
- **Example:** $2n + 100\log n$ is $O(n)$ (常数不计入)
- In general, we should use the big-Oh notation to characterize a function as closely as possible

$$f(n) < C \cdot g(n) = C \cdot n^2$$

Comparative analysis

Question: Suppose two algorithms solving the same problem are available: an algorithm A, which has a running time of $O(n)$, and an algorithm B, which has a running time of $O(n^2)$. Which algorithm is better?

A 渐近优于 B.

Answer: Algorithm A is asymptotically better than algorithm B

(n^2 increases much faster
than n)

linear.

Comparative analysis

- We can use the big-Oh notation to order classes of functions by asymptotic growth rate 渐进增长率.
- Our seven functions are ordered by increasing growth rate in the following sequence

n	$\log n$	n	$n \log n$	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	4,294,967,296
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	1.15×10^{77}
512	9	512	4,608	262,144	134,217,728	1.34×10^{154}

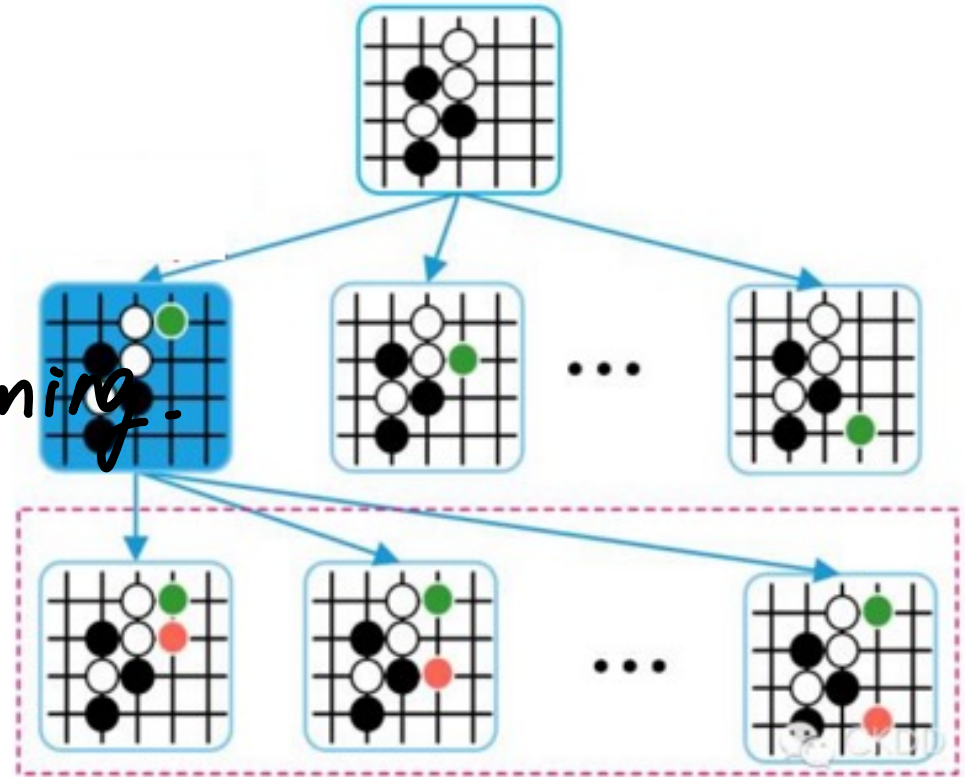
cleverly search for strategies

Why AlphaGo is a remarkable achievement?

- If we use brutal-force to search the best move in Go, the time complexity is at the order of $O(10^n)$

↳ time consuming.

- The search space is even larger than the number of atoms in the universe!!!



易处理

The line of tractability

- To differentiate **efficient** and **inefficient** algorithms, the general line is between **polynomial time algorithms** and **exponential time algorithms**

↓
指数.

↓
多项式时间算法

- The **distinction** between polynomial-time and exponential-time algorithms is considered a **robust measure of tractability**

△ focusing on the worst-case input:

Example: Finding the smallest number in a list

primitive operations:

- Assignment

- Comparing 2 numbers

- Calling a function

(assume that all take 1 unit of time)

n is the size of the list ←

1+1+1+1+4
why not 5: (4n)

★ take the worst case !!

impossible to run all 5 statements

```
3 { /smallest_so_far = None
    / print('Before', smallest_so_far)
    for num in [9, 39, 21, 98, 4, 5, 100, 65]:
        if smallest_so_far == 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 of this algorithm?

total: $4n+4$ is $O(n)$

(loop)

取決於↓的

Recursion 递归

- **Recursion** is a technique by which a **function** makes one or more **calls to itself** during execution
- Recursion provides an elegant and powerful **alternative** for performing **repetitive tasks**
- Recursion is an important technique in the study of data structures and algorithms

Inception



阶乘函数

Example: The factorial function

- The **factorial** of a positive integer n , denoted $n!$, is defined as follows:

$$n! = \begin{cases} 1 & \text{if } n = 0 \\ n \cdot (n-1) \cdot (n-2) \cdots 3 \cdot 2 \cdot 1 & \text{if } n \geq 1. \end{cases}$$

- The factorial function is important because it is known to equal the number of ways in which n distinct items can be arranged into a sequence, that is, the number of permutations of n items

The recursive definition

多个基本情况

➤ 1

- First, a recursive definition contains one or more **base cases**, which are defined **non-recursively** in terms of fixed quantities

多个递归情况

- Second, it also contains one or more **recursive cases**, which are defined by appealing to the definition of the function being defined

调用定义过的函数。

The recursive definition of factorial function

- The factorial function can be naturally defined in a recursive way, for example, $5! = 5 \cdot (4 \cdot 3 \cdot 2 \cdot 1) = 5 \cdot 4!$
- More generally, for a positive integer n , we can define $n!$ to be $n \cdot (n-1)!$
- Therefore, the recursive definition of factorial function is:

$$n! = \begin{cases} 1 & \text{if } n = 0 \\ n \cdot (n-1)! & \text{if } n \geq 1. \end{cases}$$

Solution

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

递归

$n(n-1) \cdots 1.$

How Python implements recursion

递归记录 框架

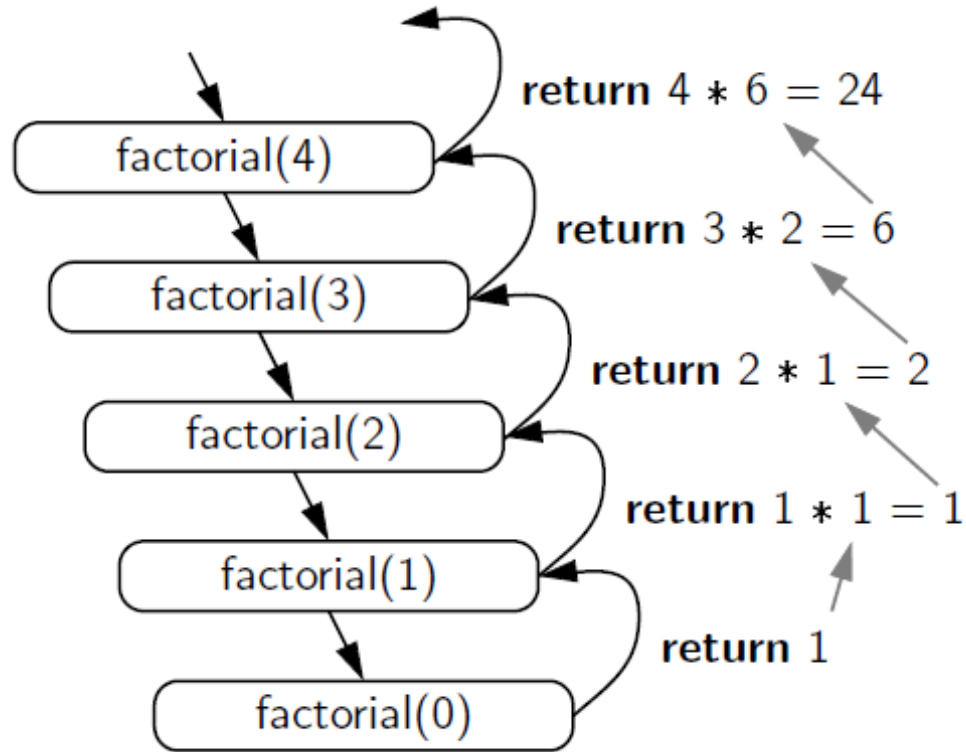
- In Python, each time a function (recursive or otherwise) is called, a structure known as an activation record or frame is created to store information about the progress of that invocation of the function

- This activation record stores the function call's **parameters** and **local variables**

嵌套函数调用

- When the execution of a function leads to a nested function call, the execution of the former call is suspended and its activation record stores the place in the source code at which the **flow of control** **should continue** upon return of the nested call

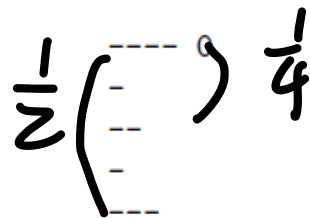
The recursive trace



递归绘制英式标尺

Example: Drawing an English ruler

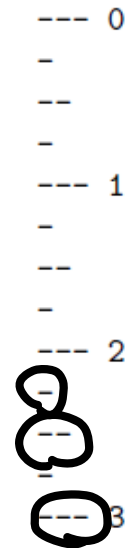
- We denote the length of the tick designating a whole inch as the major tick length.
- Between the marks for whole inches, the ruler contains a series of minor ticks, placed at intervals of $\frac{1}{2}$ inch, $\frac{1}{4}$ inch, and so on.
- As the size of the interval decreases by half, the tick length decreases by one



(a)



(b)



(c)

interval:

$\frac{1}{8}$ inch

$\frac{1}{16}$ inch

$\frac{1}{32}$ inch

刻度线
不一样长!!

Recursive implementation of English ruler

- An interval with a central tick length $L \geq 1$ is composed of:
 - ✓ An interval with a central tick length $L-1$
 - ✓ A single tick of length L 长度为 L 的单独刻度线
 - ✓ An interval with a central tick length $L-1$
一个中央刻度线长度为 $L-1$ 的刻度间隔

Solution

绘制一条刻度线 刻度线长度

刻度标签, 默认为空.

检查
tickLabel
是否存在.

```
def draw_line(tickLen, tickLabel=''):
    line = '-' * tickLen
    if tickLabel:
        line += ' ' + tickLabel
    print(line)
```

代表没有标签的

刻度线 (主刻度, 刻度线会长一些)

```
def draw_interval(centerLen):
```

绘制区间

```
    if centerLen > 0:
```

中央刻度

上方

```
        draw_interval(centerLen-1)
```

```
        draw_line(centerLen)
```

打印中央刻度.

```
        draw_interval(centerLen-1)
```

页尺数 主刻度长度 下方

```
def draw_ruler(numInch, majorLen):
```

```
    draw_line(majorLen, '0')
```

→ 在标尺顶部绘制一条长度为 majorLen 的线并标上 '0'.

```
    for j in range(1, 1+numInch):
```

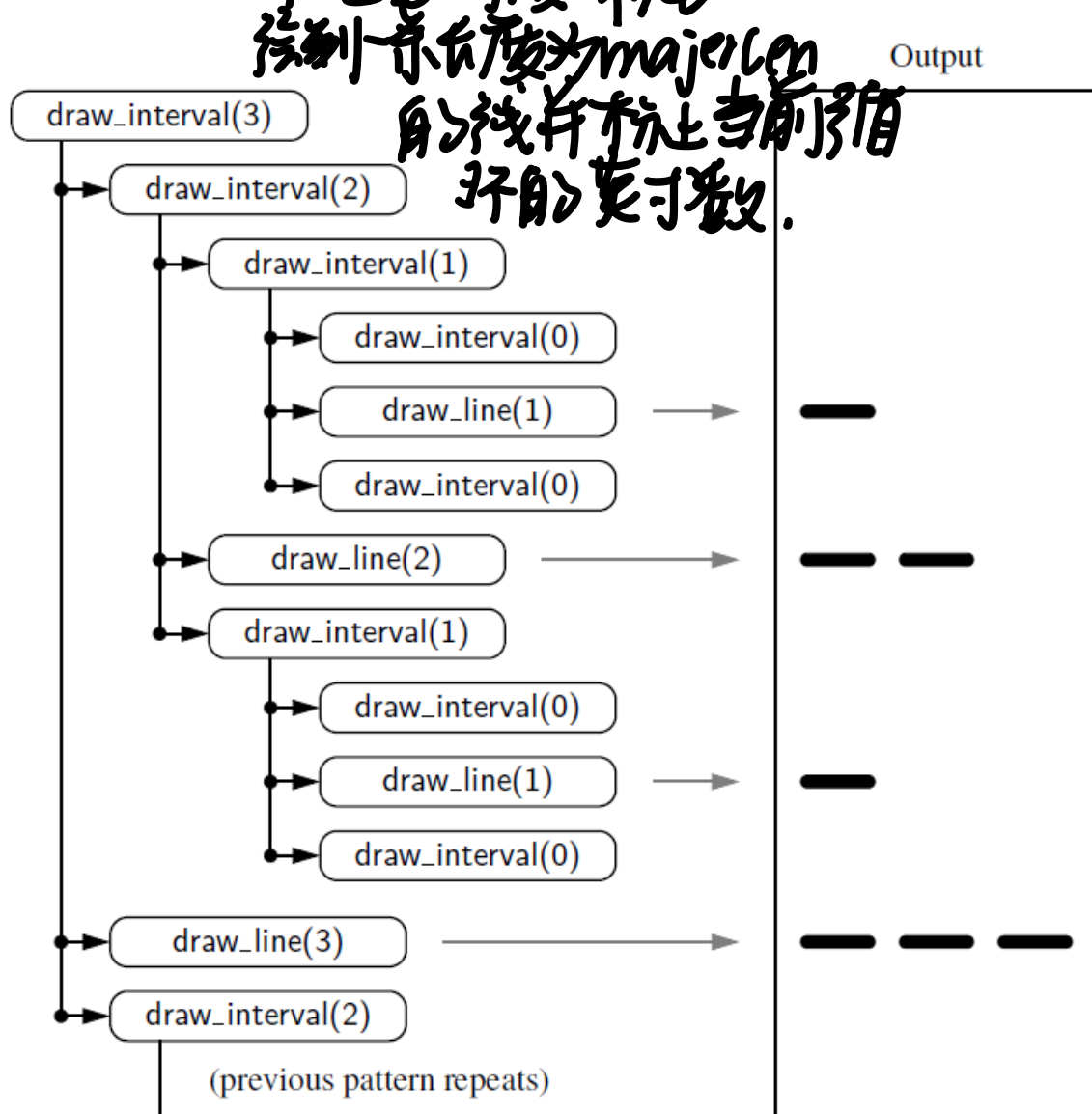
```
        draw_interval(majorLen-1)
```

```
        draw_line(majorLen, str(j))
```

在主要刻度末尾.

调用递归

The recursive trace for English ruler



Example: Binary search

- A classic and very useful recursive algorithm, **binary search**, can be used to **efficiently locate** a target value within a **sorted** sequence of n elements

未排序



- When the sequence is **unsorted**, the standard approach to search for a target value is to use a loop to examine every element, until either finding the target or exhausting the data set; This is known as the **sequential search** algorithm



0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
2	4	5	7	8	9	12	14	17	19	22	25	27	28	33	37

顺序搜索算法

Binary search 二分法

可加索引的

- When the sequence is sorted and indexable, binary search is a much more efficient algorithm
- For any index j , we know that all the values stored at indices $0, \dots, j-1$ are less than or equal to the value at index j , and all the values stored at indices $j+1, \dots, n-1$ are greater than or equal to that at index j

The strategy of binary search

- We call an element of the sequence a candidate if, at the current stage of the search, we cannot rule out that this item matches the target
- The algorithm maintains two parameters, low and $high$, such that all the candidate entries have index at least low and at most $high$
- Initially, $low = 0$ and $high = n-1$. We then compare the target value to the median candidate, that is, the item $data[mid]$ with index
 $mid = \lfloor (low+high)/2 \rfloor$ ★

The strategy of binary search

- If the target equals $\text{data}[\text{mid}]$, then we have found the item we are looking for, and the search terminates successfully



- If $\text{target} < \text{data}[\text{mid}]$, then we recur on the first half of the sequence, that is, on the interval of indices from low to $\text{mid}-1$
- If $\text{target} > \text{data}[\text{mid}]$, then we recur on the second half of the sequence, that is, on the interval of indices from $\text{mid}+1$ to high

(don't need to check every element)

Solution

recursive

```
def binarySearch(data, target, low, high):  
    if low > high:  
        print('Cannot find the target number!')  
        return False  
    else:  
        mid = (low + high) // 2  
        if target == data[mid]:  
            print('The target number is at position', mid)  
            return True  
        elif target < data[mid]:  
            return binarySearch(data, target, low, mid - 1)  
        else:  
            return binarySearch(data, target, mid + 1, high)  
  
def main():  
    data = [1, 3, 5, 6, 16, 78, 100, 135, 900] * sorted.  
    target = 16  
    binarySearch(data, target, 0, len(data) - 1)
```

print, 不是
return, 否则
无输出!
递归

main()

Time complexity of binary search

Sequential search: $O(n) \rightarrow$ linear search.

Proposition: The binary search algorithm runs in $O(\log n)$ time for a sorted sequence with n elements

$$\log_2 n$$

Why ?



\square (0/1 value)

$$2^x = n$$

$$\rightarrow x = \log_2 n.$$

Proof

Justification: To prove this claim, a crucial fact is that with each recursive call the number of candidate entries still to be searched is given by the value

$$\text{high} - \text{low} + 1.$$

Moreover, the number of remaining candidates is reduced by at least one half with each recursive call. Specifically, from the definition of mid, the number of remaining candidates is either

$$(\text{mid} - 1) - \text{low} + 1 = \left\lfloor \frac{\text{low} + \text{high}}{2} \right\rfloor - \text{low} \leq \frac{\text{high} - \text{low} + 1}{2}$$

or

$$\text{high} - (\text{mid} + 1) + 1 = \text{high} - \left\lceil \frac{\text{low} + \text{high}}{2} \right\rceil \leq \frac{\text{high} - \text{low} + 1}{2}.$$

Initially, the number of candidates is n ; after the first call in a binary search, it is at most $n/2$; after the second call, it is at most $n/4$; and so on. In general, after the i^{th} call in a binary search, the number of candidate entries remaining is at most $n/2^i$. In the worst case (an unsuccessful search), the recursive calls stop when there are no more candidate entries. Hence, the maximum number of recursive calls performed, is the smallest integer r such that

$$\frac{n}{2^r} < 1.$$

思路分析
↓

In other words (recalling that we omit a logarithm's base when it is 2), $r > \log n$. Thus, we have

$$r = \lfloor \log n \rfloor + 1,$$

which implies that binary search runs in $O(\log n)$ time. ■

A worst case

n个候选数, 运行j次

原因: 表示快速排序的时间复杂度

平均: 选中间
每-层, 均为 $O(n) \Rightarrow$ 全部过-次.

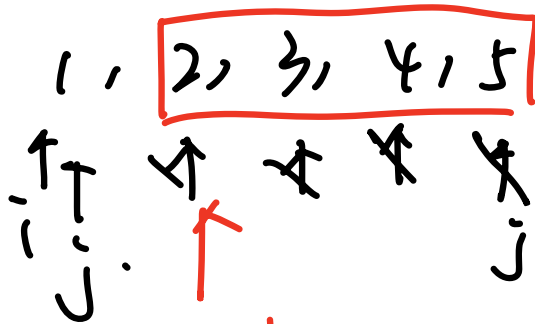
$$O(n) \cdot O(\lg n)$$

$$= O(n \lg n)$$

最坏:
选第一个元素

$$O(n^2)$$

[tips: 递归 ($q, low, i-1$)
($q, j+1, high$)]



从 $j+1$ 开始看.

本质上, $TC =$

$$(n-1) + \dots + 1$$

$$= (n-1+1)(n-1) \cdot \frac{1}{2}$$

$$= \frac{n^2 - n}{2} = O(n^2)$$