

Introduction to Computation

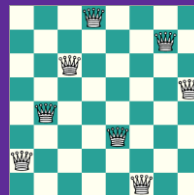
Autumn, 2023

Prof. Fan Cheng

Shanghai Jiao Tong University

chengfan85@gmail.com

<https://github.com/ichengfan/itc>





Outline

- Review of Past Lectures
- Style Guide for Python Code
- Practice problems

Review of Past Lectures



Programming language

- In principle, all the popular PLs have the following elements:
 - Input and output
 - `input()`, `print()`
 - Types and variables
 - `type()`, `id()`, `int()`, `float()`, `str()`, `chr()`, `ord()`
 - Basic expressions: logic, mathematics
 - Conditional expression
 - Code block and indentation
 - Loop expression
 - `while`, `for`, `break`
 - Function
 - Parameters and return values
 - File
 - To be introduced
- Advanced
 - Class and OO
 - Exception
 - Standard library
- When you have become familiar with one PL, you could learn another shortly
 - Don't learn programming languages but [learn how to program](#)
 - You should master several PLs and use them as your primary tools
 - Learn C in 2 two days. 😊

Practice makes perfect

Bug-free code and Debug

- 反复思考程序大的框架，谋定而后动
 - 模块、类、函数
 - 算法、数据结构
- 能正确运行的代码才是好的代码
 - 先实现功能，再优化性能，不要提前优化
 - 提前优化会把系统实现复杂化
- 保证代码结构的清晰和简介
 - 循环深度不超过两轮
 - 单个函数的长度不要过长（25-35行）
 - 多用函数、类、包等机制来隔离代码
 - Zen of Python
- 多试运行
 - 一边编码，一边试运行
 - 3-5行运行一次，看看输出是否正确
- 注意边际输入：:x=[], “”, (,)
- 实现每个功能后都做调试，保证前面不错
- 一个大的功能完成后先反复试运行
- 调试
 - print每个中间的重要数据，保证中间状态正确
 - IDE提供的debug功能，但不建议初学者用这个
 - 初学者的代码可能不超过300行，可以依靠观察能力和print来debug

Common Errors

- `TypeError`
- `NameError`
- `None`
- `SyntaxError`
- `IndentationError`
- `AttributeError`
- `ZeroDivisionError`

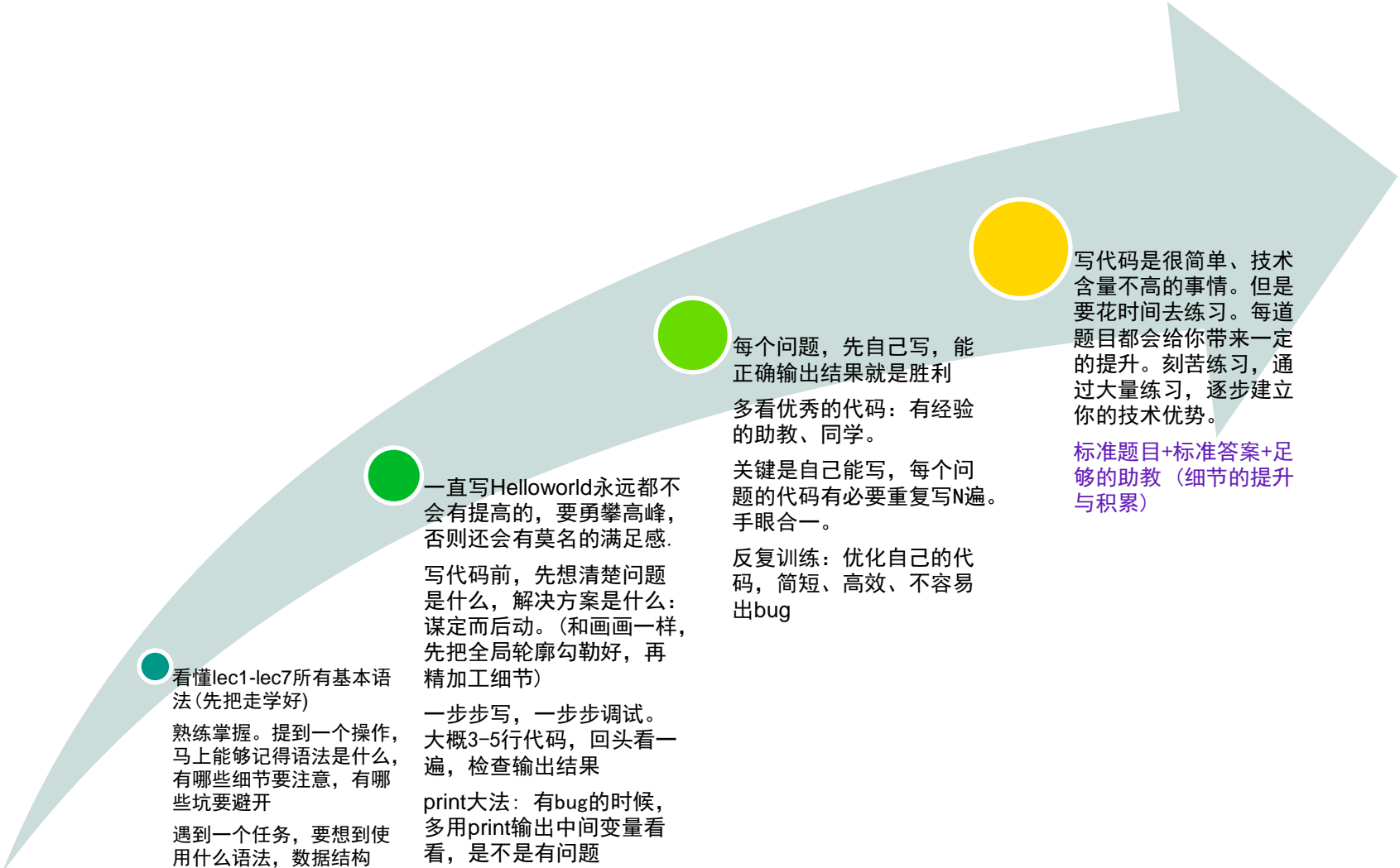
Leetcode练习

- 从简单题目开始
 - 自己写代码，能写出来就可以了
 - AC就是最好的
- 困难的题目
 - 看参考样例
 - 看懂、自己能仿照写
 - 回过头自己多写几遍
 - 不会写的原因是写的少了，见得少了，想的少了：maturity
- 可以先在vscode上写好，vscode可以提供一些辅助功能
- 反复练习：提升速度、减少bug
 - 刚开始总是困难的，怎么写怎么错
 - 当你超越50题的时候就会焕然一新
 - 日积月累，1-2个月可以看到效果
 - 我们不是为了期末考试而设置课程
- 目标：100+100+100

不贪多，每天保证3-5道就足够了

每道题务必做透

刚开始速度慢，一天3题，熟练了，可以远远超过3题
2个月的练习到期末前大概可以做完200道



看懂lec1-lec7所有基本语法(先把走学好)

熟练掌握。提到一个操作，马上能够记得语法是什么，有哪些细节要注意，有哪些坑要避免

遇到一个任务，要想到使用什么语法，数据结构

一直写Helloworld永远都不会有提高的，要勇攀高峰，否则还会有莫名的满足感。

写代码前，先想清楚问题是什么，解决方案是什么：谋定而后动。(和画画一样，先把全局轮廓勾勒好，再精加工细节)

一步步写，一步步调试。大概3-5行代码，回头看一遍，检查输出结果

print大法：有bug的时候，多用print输出中间变量看看，是不是有问题

每个问题，先自己写，能正确输出结果就是胜利

多看优秀的代码：有经验的助教、同学。

关键是自己能写，每个问题的代码有必要重复写N遍。手眼合一。

反复训练：优化自己的代码，简短、高效、不容易出bug

写代码是很简单、技术含量不高的事情。但是要花时间去练习。每道题目都会给你带来一定的提升。刻苦练习，通过大量练习，逐步建立你的技术优势。

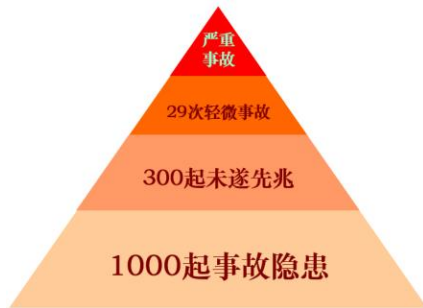
标准题目+标准答案+足够的助教（细节的提升与积累）

Style Guide for Python Code



Python规范

- One of Guido's key insights is that **code is read much more often than it is written**
- import this: **Readability counts**
- **PEP 8 -- Style Guide for Python Code**
 - PEP: Python Enhancement Proposal
 - <https://www.python.org/dev/peps/pep-0008/>
 - This document gives coding conventions for the Python code comprising the standard library in the main Python distribution
- 除了语法上面合格，还要在风格上面保持一致
- 安全生产：不带电操作；电闸的开关为什么要挂上面，而不是下面？
- 海恩法则，是航空界关于飞行安全的法则
- 海恩法则指出：每一起严重事故的背后，必然有29次轻微事故和300起未遂先兆以及1000起事故隐患
- 每一个规范，都是血泪教训



- Introduction
- A Foolish Consistency is the Hobgoblin of Little Minds
- Code Lay-out
 - Indentation
 - Tabs or Spaces?
 - Maximum Line Length
 - Should a Line Break Before or After a Binary Operator?
 - Blank Lines
 - Source File Encoding
 - Imports
 - Module Level Dunder Names
- String Quotes
- Whitespace in Expressions and Statements
 - Pet Peeves
 - Other Recommendations
- When to Use Trailing Commas
- Comments
 - Block Comments
 - Inline Comments

- Documentation Strings
- Naming Conventions
 - Overriding Principle
 - Descriptive: Naming Styles
 - Prescriptive: Naming Conventions
 - Names to Avoid
 - ASCII Compatibility
 - Package and Module Names
 - Class Names
 - Type Variable Names
 - Exception Names
 - Global Variable Names
 - Function and Variable Names
 - Function and Method Arguments
 - Method Names and Instance Variables
 - Constants
 - Designing for Inheritance
 - Public and Internal Interfaces
- Programming Recommendations
 - Function Annotations
 - Variable Annotations
- References
- Copyright

Indentation

- Use 4 spaces per indentation level
- Continuation lines should align wrapped elements either vertically using Python's implicit line joining inside parentheses, brackets and braces, or using a hanging indent. When using a hanging indent the following should be considered; there should be no arguments on the first line and further indentation should be used to clearly distinguish itself as a continuation line

```
# Correct:

# Aligned with opening delimiter.
foo = long_function_name(var_one, var_two,
                          var_three, var_four)

# Add 4 spaces (an extra level of indentation) to distinguish arguments from
the rest.
def long_function_name(
    var_one, var_two, var_three,
    var_four):
    print(var_one)

# Hanging indents should add a level.
foo = long_function_name(
    var_one, var_two,
    var_three, var_four)
```

```
# Wrong:

# Arguments on first line forbidden when not using vertical alignment.
foo = long_function_name(var_one, var_two,
                          var_three, var_four)

# Further indentation required as indentation is not distinguishable.
def long_function_name(
    var_one, var_two, var_three,
    var_four):
    print(var_one)
```

怎么清晰怎么来

Binary Operator

- Donald Knuth explains the traditional rule in his Computers and Typesetting series: "Although formulas within a paragraph always break after binary operations and relations, displayed formulas always break before binary operations"

```
# Wrong:
# operators sit far away from their operands
income = (gross_wages +
          taxable_interest +
          (dividends - qualified_dividends) -
          ira_deduction -
          student_loan_interest)
```

```
# Correct:
# easy to match operators with operands
income = (gross_wages
          + taxable_interest
          + (dividends - qualified_dividends)
          - ira_deduction
          - student_loan_interest)
```

怎么漂亮怎么来

import

- Imports are always put at the top of the file, just after any module comments and docstrings, and before module globals and constants.
- Imports should be grouped in the following order:
 - Standard library imports.
 - Related third party imports.
 - Local application/library specific imports.
- You should put a blank line between each group of imports.
- Absolute imports are recommended, as they are usually more readable and tend to be better behaved
- Wildcard imports (from <module> import *) should be avoided

Correct:

```
import os  
  
import sys
```

Wrong:

```
import sys, os
```

Correct:

```
from subprocess import Popen, PIPE
```

White Space: Pet Peeves (1)

- Avoid extraneous whitespace in the following situations:
 - Immediately inside parentheses, brackets or braces:

Correct:

```
spam(ham[1], {eggs: 2})
```

Wrong:

```
spam( ham[ 1 ], { eggs: 2 } )
```

- Between a trailing comma and a following close parenthesis:

Correct:

```
foo = (0,)
```

Wrong:

```
bar = (0, )
```

White Space: Pet Peeves (2)

- Avoid extraneous whitespace in the following situations:
 - Immediately before a comma, semicolon, or colon:

Correct:

```
if x == 4: print x, y; x, y = y, x
```

Wrong:

```
if x == 4 : print x , y ; x , y = y , x
```

- Immediately before the open parenthesis that starts the argument list of a function call:

Correct:

```
spam(1)
```

Wrong:

```
spam (1)
```


White Space: Pet Peeves (3)

- Avoid extraneous whitespace in the following situations:
 - Immediately before the open parenthesis that starts an indexing or slicing:

Correct:

```
dct['key'] = lst[index]
```

Wrong:

```
dct ['key'] = lst [index]
```

- More than one space around an assignment (or other) operator to align it with another:

Correct:

```
x = 1
```

```
y = 2
```

```
long_variable = 3
```

Wrong:

```
x           = 1
```

```
y           = 2
```

```
long_variable = 3
```

White Space: Recommendations

- Avoid trailing (结尾) whitespace anywhere.
- Always surround these binary operators with a single space on either side: assignment (=), augmented assignment (+=, -= etc.), comparisons (==, <, >, !=, <>, <=, >=, in, not in, is, is not), Booleans (and, or, not).
- If operators with different priorities are used, consider adding whitespace around the operators with the lowest priority(ies). Use your own judgment; however, never use more than one space, and always have the same amount of whitespace on both sides of a binary operator

$x+y*z$

$x + y*z$

- Function annotations should use the normal rules for colons and always have spaces around the -> arrow if present.
- Don't use spaces around the = sign when used to indicate a **keyword argument**, or when used to indicate a default value for an unannotated function parameter
 - When combining an argument annotation with a default value, however, do use spaces around the = sign:
- Compound statements (multiple statements on the same line) are generally discouraged
- While sometimes it's okay to put an if/for/while with a small body on the same line, never do this for multi-clause statements. Also avoid folding such long lines!

White Space: Summary

1. 代码符合英文写作规划
2. 用空格把程序切割成一个个合适的小的单元。每个单元有清晰的意思
3. 风格要统一
4. 不要挤成一坨
5. 清晰最重要

Correct:

```
ham[1:9], ham[1:9:3], ham[:9:3], ham[1::3], ham[1:9:]  
ham[lower:upper], ham[lower:upper:], ham[lower::step]  
ham[lower+offset : upper+offset]  
ham[: upper_fn(x) : step_fn(x)], ham[:: step_fn(x)]  
ham[lower + offset : upper + offset]
```

Wrong:

```
ham[lower + offset:upper + offset]  
ham[1: 9], ham[1 :9], ham[1:9 :3]  
ham[lower : : upper]  
ham[ : upper]
```

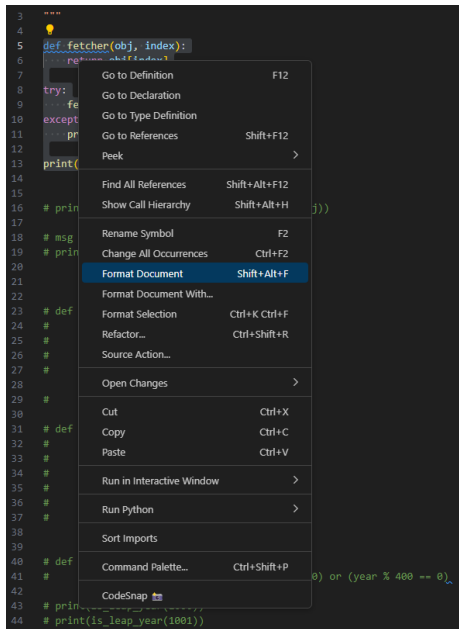
Name Convention

- Python命名: xxx_xxx_xxxxx
- 大小写区分规则: module_name, package_name, ClassName, method_name, ExceptionName, function_name, GLOBAL_CONSTANT_NAME, global_var_name, instance_var_name, function_parameter_name, local_var_name. CLASS_CONSTANT_NAME
- Names to Avoid
 - Never use the characters 'l' (lowercase letter el), 'O' (uppercase letter oh), or 'I' (uppercase letter eye) as single character variable names.
 - In some fonts, these characters are indistinguishable from the numerals one and zero. When tempted to use 'l', use 'L' instead.

<https://www.python.org/dev/peps/pep-0008/>

Useful VS Code Extensions

- autopep8, Black Format
- 解决很多格式问题
 - 空格
 - 布局
- [Survey](#) VS Code useful extensions



Practice problems

无他，唯手熟尔



Leap year

- A year is called leap:
 - It is divisible by 4 exactly
 - If it is divisible by 100, it should be divisible by 400
- Write a function to implement it, try to simplify your code

```
1 def is_leap_year(year):
2     if year % 4 == 0 and year % 100 != 0:
3         return True
4     if year % 400 == 0:
5         return True
6
7     return False
```

```
1 def is_leap_year(year):
2     if year % 4 == 0 and year % 100 != 0:
3         return True
4     elif year % 400 == 0:
5         return True
6     else:
7         return False
```

```
1 def is_leap_year(year):
2     return (year % 4 == 0 and year % 100 != 0) or (year % 400 == 0)
```

代码技巧

```
1 def f(n):
2     if n == 1:
3         return True
4     else:
5         return False
6
7
8 # 等价于
9 def f(n):
10     return n == 1
```

```
1 def f(n):
2     if g(n):
3         return a(n)
4     else:
5         return b(n)
6
7
8 # 等价于
9 def f(n):
10     return a(n) if g(n) else b(n)
```

```
1 # Bad
2 if v == True:
3     f()
4
5 # Good
6 if v:
7     f()
8
9 # Bad
10 if v == False:
11     f()
12
13 # Good
14 if not v:
15     f()
```

- ① 能够连写的地方一律连写
`x = x + 1`
`x += 1`
- ② 函数定义内部不要用print, 一律用return。除非题目让你print。调试可以用print
- ③ 调试的时候用小规模数据, 不要一开始就1,000,000. 可以从1, 2, 3, 4, 5, 6开始
- ④ 问问题记得截屏, 不要手机拍照, 发源代码, 发图
- ⑤ 作业题不要用超出范围的语法, 我们确保可以用讲过的语法解决
- ⑥ 不包含注释, 单个函数的代码长度不要超过40行。超过这个长度, 要么切割成更小的几个几个函数, 要么你的思路有问题, 要重新写
- ⑦ 务必先熟练掌握ppt上面的内容, 做到能自己写出来
- ⑧ 编程是一个精密的数学工程, 不要用蛮力, 不要暴力复制粘贴

Prime Number

- A number n is prime if it only has divisors 1 and itself

```
1 def is_prime_trivial(n):
2     for x in range(2, n):
3         if n%x == 0:
4             return False
5
6     return True
```

```
1 def is_prime_fast(n):
2     for x in range(2, n):
3         if x*x > n:
4             return True
5
6         if n%x == 0:
7             return False
8
9     return True
```

```
True True
True True
False False
True True
False False
True True
True True
False False
```

Happy Number

- (202-Happy Number) Write an algorithm to determine if a number is "happy".
- A happy number is a number defined by the following process: Starting with any positive integer, replace the number by the sum of the squares of its digits, and repeat the process until the number equals 1 (where it will stay), or it loops endlessly in a cycle which does not include 1. Those numbers for which this process ends in 1 are happy numbers.
- Example: Input: 19 Output: true

Explanation:

$$1^2 + 9^2 = 82; 8^2 + 2^2 = 68; 6^2 + 8^2 = 100; 1^2 + 0^2 + 0^2 = 1$$

- Analysis: $n \rightarrow f(n) \rightarrow f(f(n)) \rightarrow f^3(n) \dots \rightarrow 1$ or Loop
 1. Implement a function $f(n)$: the sum of the squares of its digits
 2. How to check whether $1 \in S$? Data structure: list, tuple, dict, set. Dict or set?

Happy Number: solution

```
1 def sum_of_digit_squares(n):
2     total = 0
3     while n > 0:
4         total += (n % 10) ** 2
5         n //= 10
6     return total
7
8
9 for x in (19, 91, 190, 109, 1, 11, 101):
10     print(sum_of_digit_squares(x))
```

```
1 def sum_of_digit_squares(n):
2     return n * n if n < 10 else (n % 10) ** 2 + sum_of_digit_squares(n // 10)
```

```
1 def sum_of_digit_squares(n):
2     return sum([(ord(x)-ord('0'))**2 for x in str(n)])
```

```
1 def happy_number(n):
2     st = {n}
3
4     while n != 1:
5         x = sum_of_digit_squares(n)
6         if x == 1:
7             return True
8
9         if x in st:
10             return False
11
12         st.add(x)
13         n = x
14
15     return True
16
17
18 ans = []
19 for x in range(1001):
20     if happy_number(x):
21         ans.append(x)
22
23 print(
24     len(ans)
25 ) # 143 happy numbers including 921, 923, 931, 932, 937, 940, 946, 964, 970, 973, 989, 998, 1000
```

Max Consecutive Ones

- Given a binary array, find the maximum number of **consecutive 1s** in this array.
- Example 1: Input: [1, 1, 0, 1, 1, 1] Output: 3
- Explanation: The first two digits or the last three digits are consecutive 1s. The maximum number of consecutive 1s is 3.
- Note:
 - The input array will only contain 0 and 1.
 - The length of input array is a positive integer and will not exceed 10,000
- Analysis: how to check **consecutive 1s**.
 - **Start from position i, i++ if the current position is 1**

多测试

```
1 def max_conse_ones(number):
2     ans = 0
3     i = 0
4     while i < len(number):
5         if number[i] == 1:
6             j = i
7
8             while j < len(number) and number[j] == 1:
9                 j += 1
10
11             if ans < j - i:
12                 ans = j - i
13             i = j + 1
14         else:
15             i += 1
16     return ans
```

```
1 print(max_conse_ones([1, 1, 0, 1, 1, 1]))
2 print(max_conse_ones([]))
3 print(max_conse_ones([0]))
4 print(max_conse_ones([1]))
5 print(max_conse_ones([0, 0]))
6 print(max_conse_ones([0, 1]))
7 print(max_conse_ones([1, 0]))
8 print(max_conse_ones([1, 1]))
9 print(max_conse_ones([1, 0, 1, 0, 1]))
```

3
0
0
1
0
1
1
2
1

Longest Continuous Increasing Subsequence

- Given an unsorted array of integers, find the length of longest **continuous increasing subsequence**

- Example 1:

Input: [1,3,5,4,7]

Output: 3

Explanation: The longest continuous increasing subsequence is [1,3,5], its length is 3. Even though [1,3,5,7] is also an increasing subsequence, it's not a continuous one where 5 and 7 are separated by 4

- Example 2:

Input: [2,2,2,2,2]

Output: 1

Explanation: The longest continuous increasing subsequence is [2], its length is 1.

Note: Length of the array will not exceed 10,000

Analysis: very similar to Max Consecutive Ones

```
1 def longest_CIS(number):
2     ans = 0
3     i = 0
4     while i < len(number):
5         j = i+1
6         while j < len(number) and number[j] > number[j-1]:
7             j += 1
8
9         if ans < j-i:
10            ans = j-i
11
12        i = j
13    return ans
```

```
1 print(longest_CIS([1,3,5,4,7]))
2 print(longest_CIS([2,2,2,2,2]))
3 print(longest_CIS([]))
4 print(longest_CIS([1]))
5 print(longest_CIS([1, 1]))
6 print(longest_CIS([1, 2]))
7 print(longest_CIS([2, 1]))
```

3
1
0
1
1
2
1

Valid Anagram

- (242-Valid Anagram) Given two strings s and t , write a function to determine if t is an anagram of s
 - An anagram is a word or phrase formed by rearranging the letters of a different word or phrase, typically using all the original letters exactly once – race: care, part: trap, heart: earth, knee: knee
- Example 1: Input: s = "anagram", t = "nagaram" Output: true
- Example 2: Input: s = "rat", t = "car" Output: false
- Note: You may assume the string contains only lowercase alphabets
- Solution: every character should have the same occurrence in s and t
- Data structure: list, tuple, str, dict or set? Dict Vs. set?

```
1 def build_dt(s):
2     dt = {}
3     for x in s:
4         if x in dt:
5             dt[x] += 1
6         else:
7             dt[x] = 1
8
9     return dt
10
11
12 def is_valid_anagram(s, t):
13     if len(s) != len(t):
14         return False
15
16     dts = build_dt(s)
17     dtt = build_dt(t)
18
19     return dts == dtt
```

```
1 def is_valid_anagram(s, t):
2     if len(s) != len(t):
3         return False
4
5     return sorted(s) == sorted(t)
6
7
8 print(is_valid_anagram("anagram", "nagaram"))
9 print(is_valid_anagram("rat", "cat"))
```