时间复杂度：

P(Polynomial)问题：，，等问题，多项式时间的解法，随着n的增大，时间的增大是n和n的2次方的。

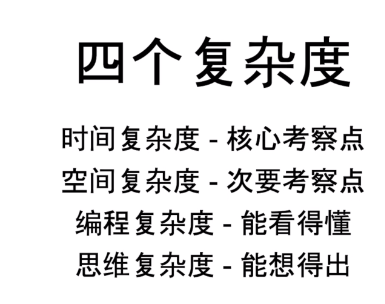
NP(Nondeterministic Polynomial)问题：，，等问题，非多项式时间的解法，随着n的增大，时间的增大是指数级等

时间复杂度分析时只考虑最高项：



通常不考虑常数项和系数：





时间复杂度的计算：

int a = 0;

for (int i = 1; i < n; i++){

for (int j = 1; j < n / i; j++){

a += 1;

}

}

这段代码的时间复杂度为。

与之间的大小：



根据时间复杂度的计算方式，可以得出：

所以：

时间复杂度为的可能算法：

双指针算法，打擂台算法（寻找一个数组的最大值，枚举法的一种），单调栈算法，单调队列的算法等。

双指针算法有3种：

1. 相向双指针
2. 背向双指针
3. 同向双指针

相向双指针算法的分类：

Reverse型：

翻转字符串，判断回文串

Two Sum型：

两数之和，三数之和

Partition型：

快速排序，颜色排序

相向双指针算法：

例1

## 时间复杂度O(n)

class Solution:

"""

@param s: A string

@return: Whether the string is a valid palindrome

"""

def isPalindrome(self, s):

# write your code here

## 即使字符串只有1个标点，则视其为空串，也是回文串

if not s or len(s) <= 1:

return True

left = 0

right = len(s) - 1

## 尽管有两层while循环，但时间复杂度为O(n)

## 内层的两个while循环和外层的一个while循环

## 只会把s中的所有字符都访问1次，不会出现多次的情况

while (left < right):

## 优化代码

while left < right and not self.is\_valid(s[left]):

left += 1

while left < right and not self.is\_valid(s[right]):

right -= 1

continue

if left < right and s[left].lower() != s[right].lower():

return False

left += 1

right -= 1

return True

def is\_valid(self, char):

return char.isdigit() or char.isalpha()