实验报告——关于python题目分类

#### 小组信息：3人

181250093 柳斯宁 1225747052@qq.com 175 分工：代码下载和解压、分析代码难度部分

181250049 黄迪 957822635@qq.com 150 分工：相似度分析样本选取、题目分类思路一部分

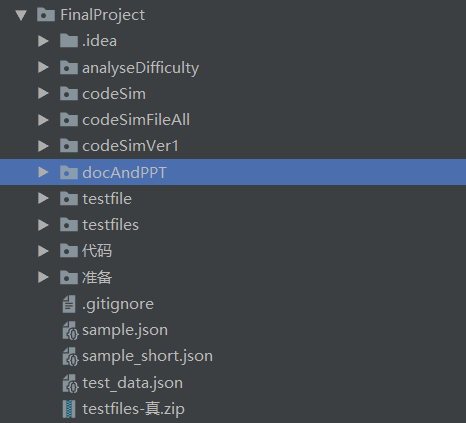
181250016 陈昱霖 1334123884@qq.com 169 分工：题目分类思路二部分、思路补充

#### 研究问题：

题目分类，通过学生提交的代码源码和成绩信息，将不同的题目的代码进行分类，目的在于重建产生我们所感知到的模式的内在模型，连续的学习以使得对同一知识点的理解更加深刻。

#### 代码开源地址：<https://github.com/NJU-SE-LHCL/DataScience>

#### 代码结构



其中

analyseDifficulty 部分是分析以代码难度进行分类

codeSim 、codeSimVer1文件夹是分析代码相似度，codeSimVer1是相似度分析的思路一部分

docAndPPT 用来放PPT

codeSimFileAll 存放代码相似度分析的同学代码

testfile 和testfiles是sample.json下载的代码 代码文件夹是test\_data下载的代码

准备文件夹里是下载和解压同学代码的代码

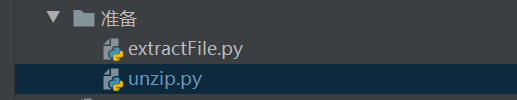
#### 研究方法

数据集：从云端下载的学生代码main.py和标准答案answer.py 以及test\_data.json中学生答题情况

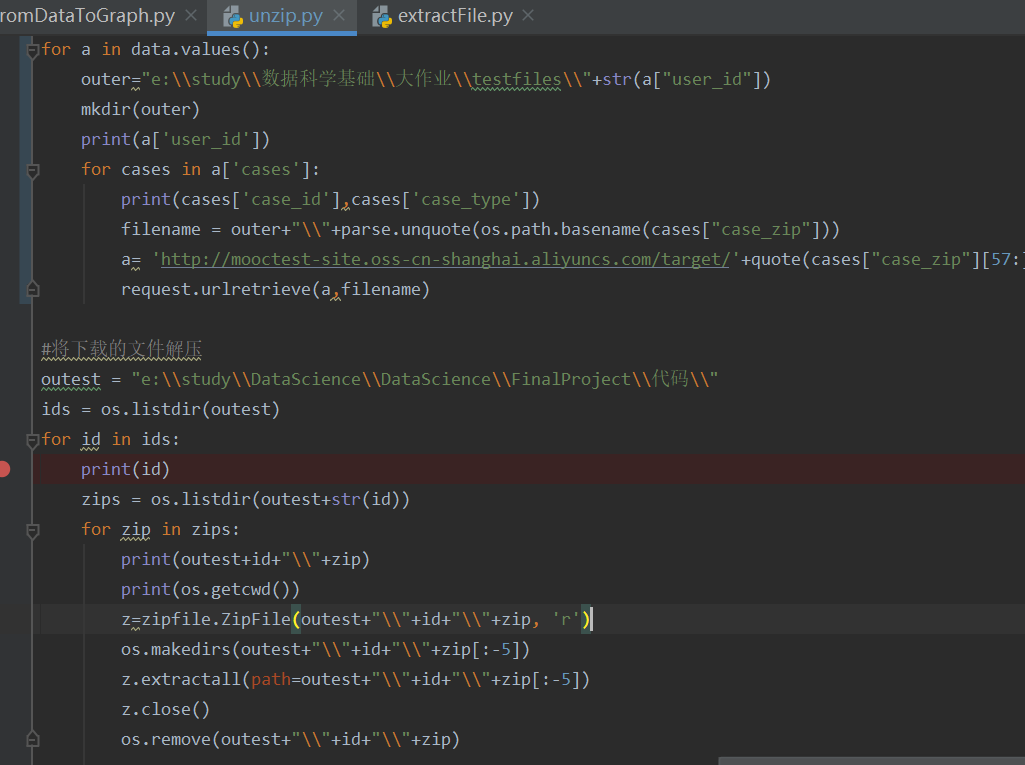
数据分析方法：通过计算数据的数学期望获得变量平均取值的大小，计算偏度即数据的三阶标准化矩计算呈正态分布的数据的整体分布特点

#### 具体案例分析（研究方法的具体说明）

代码下载（准备工作)：



其中unzip.py负责实现代码的下载和解压，extractFile负责将可执行文件复制到可执行位置

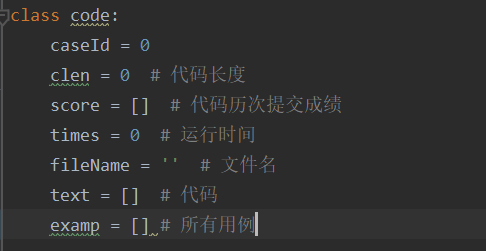


1. 代码按难度分类部分

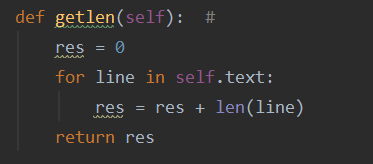
##### analyseCodeToGenerateData.py

读入所有同学的代码进行分析计算产生关于代码长度、提交成绩、运行时间的json数据用于进行分析

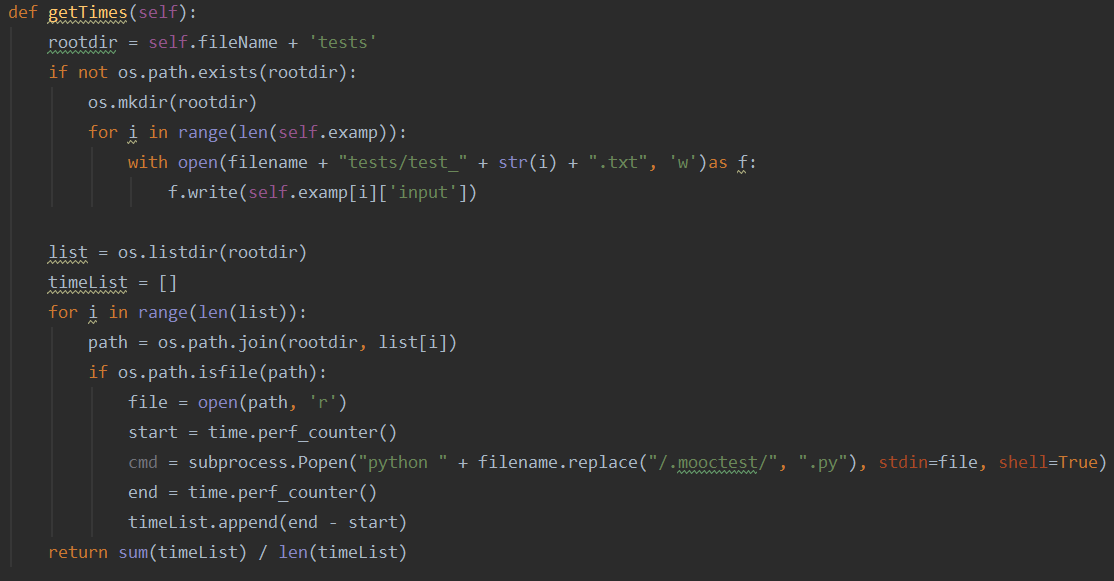
其中定义code类，将所有代码转化为code对象



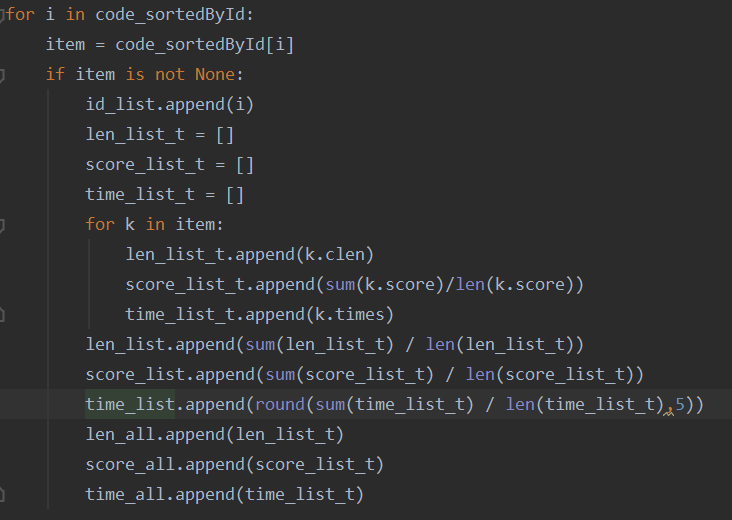
以每行代码的长度总和作为代码长度



通过process.Popen方法运行所有题目并将所有cases作为输入计算每次运行时间求数学期望作为该代码运行时间的平均值



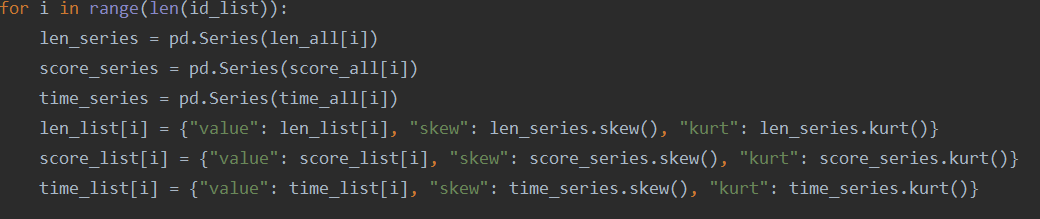
将相同case\_id的代码的长度、成绩、运行时间分别成表，并且分别求数学期望后成表,最后将所有列表生成Json文件



##### fromDataToGraph.py

用于数据可视化，将长度、成绩、运行时间分别生成图像，并且分析其偏度作为难度的影响因素，如果偏度为正证明该因素对难度有正影响，若为负则为负影响，最终通过各个影响因素的加权平均计算某case\_id的难度（取值范围为0~100）

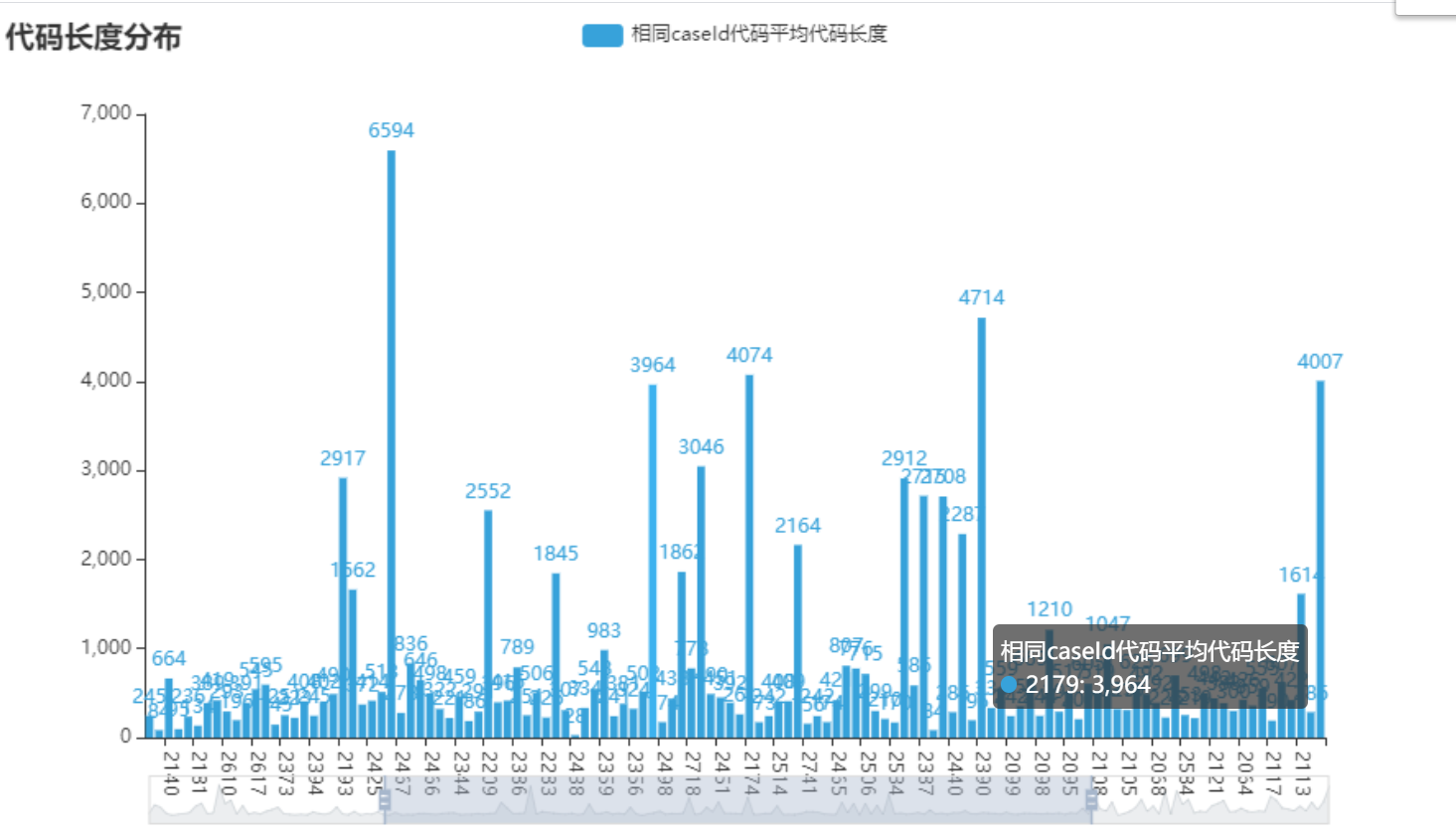
借助pandas库中的skew函数求的数据的偏度



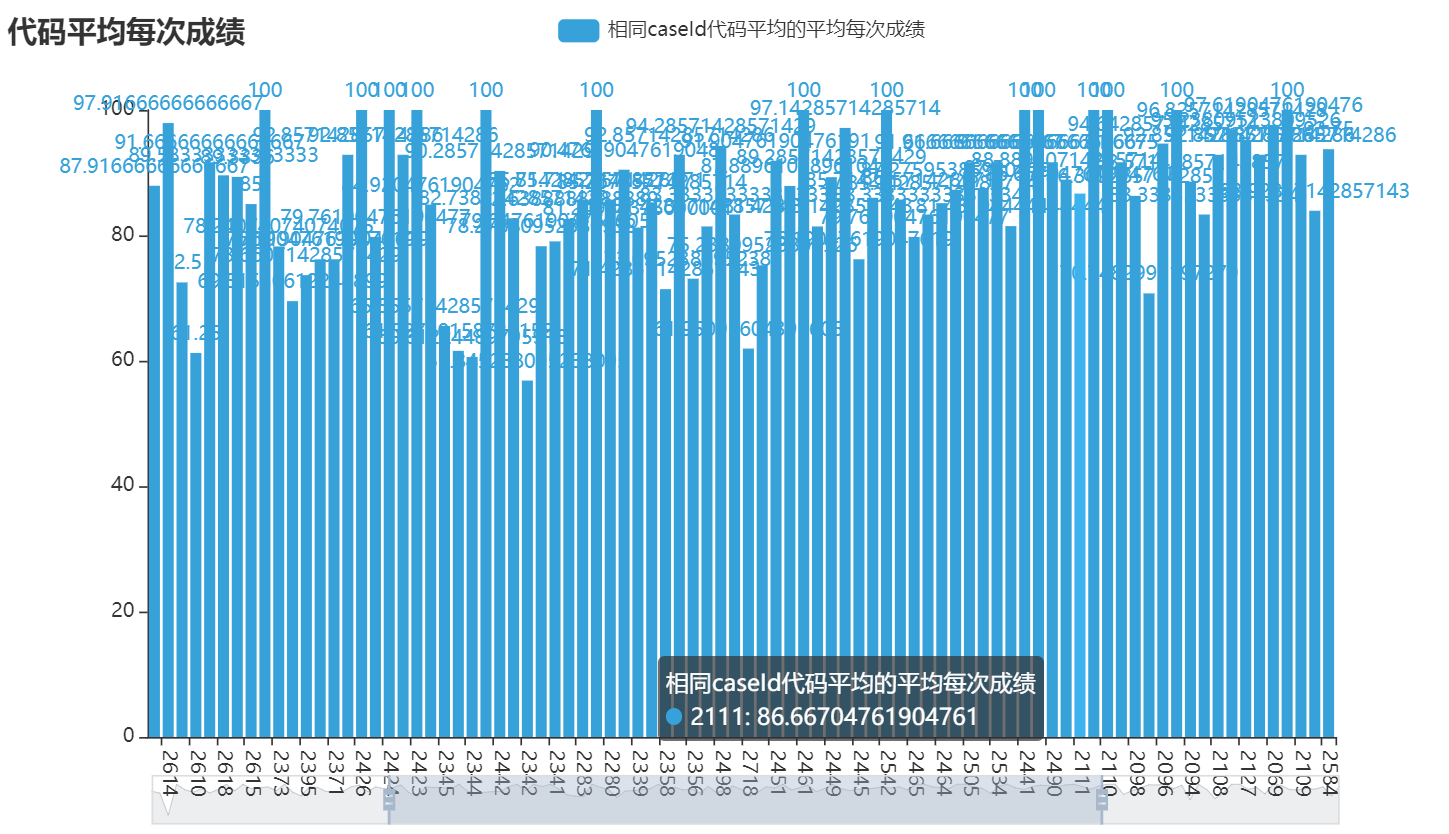
关各个因素对最终的难度影响因为不知道最终的成绩无法利用多元线性回归分析只好单独分析各自数据来进行估计。

最终成果图借助pyecharts API 将list数据转化为html数据图

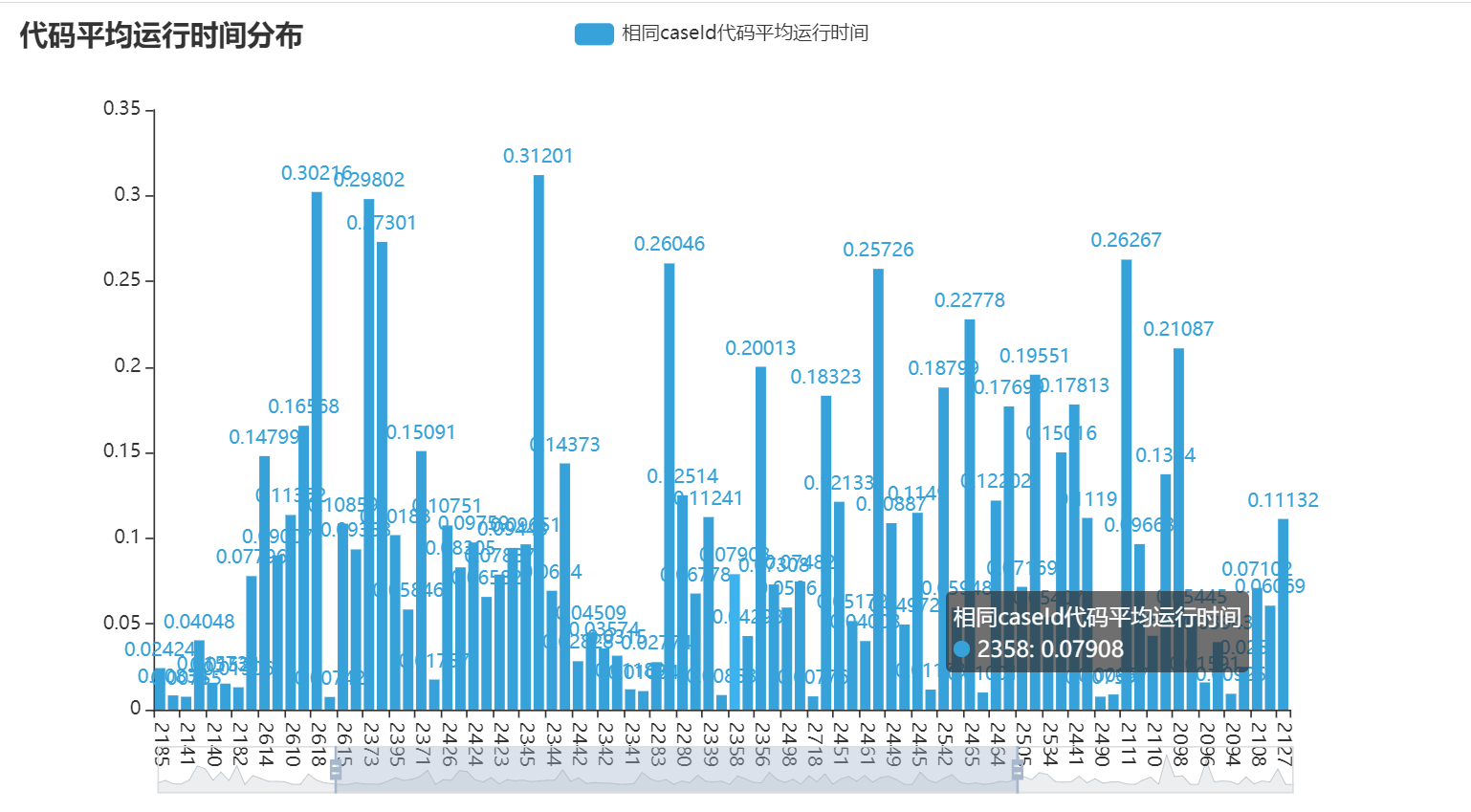




代码长度方面显而易见难度大的题目代码长度差距显著，而相对简单的代码长度差距不明显，因此长度因素的权重可以相对较大

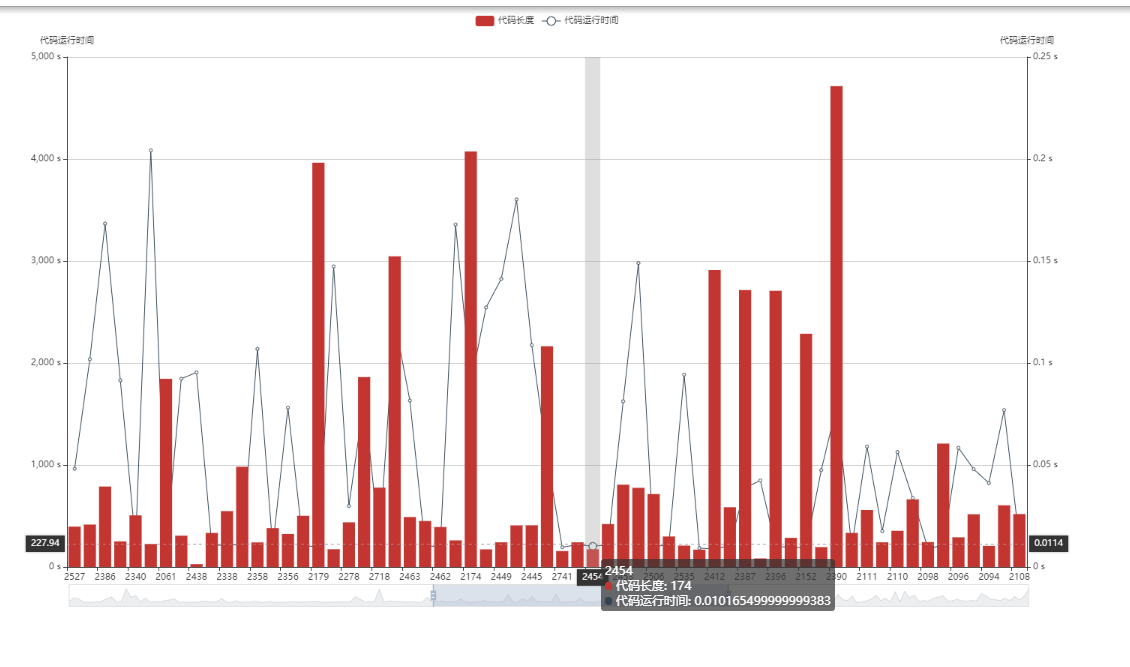


代码平均成绩方面：未达到满分成绩对最终成绩的正影响较大，而满分成绩也并不一定说明题目足够简单，相对来说对最终题目难度影响较小，因此权重适中



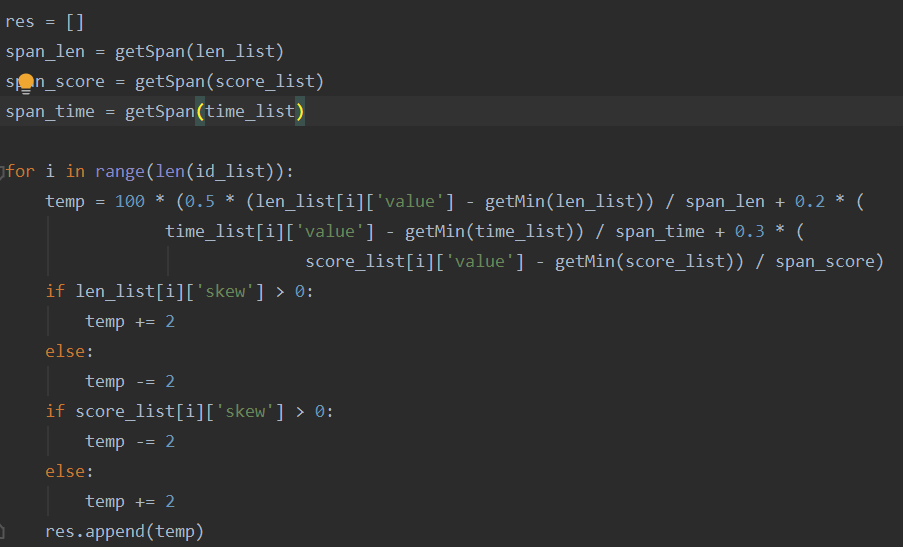
代码运行时间方面：因为代码运行时间与计算机状态有关随机性相对较大，因此权重较低

其中为了说明代码运行时间和代码长度之间并没有直接的线性关系将二者数据进行了比较分析

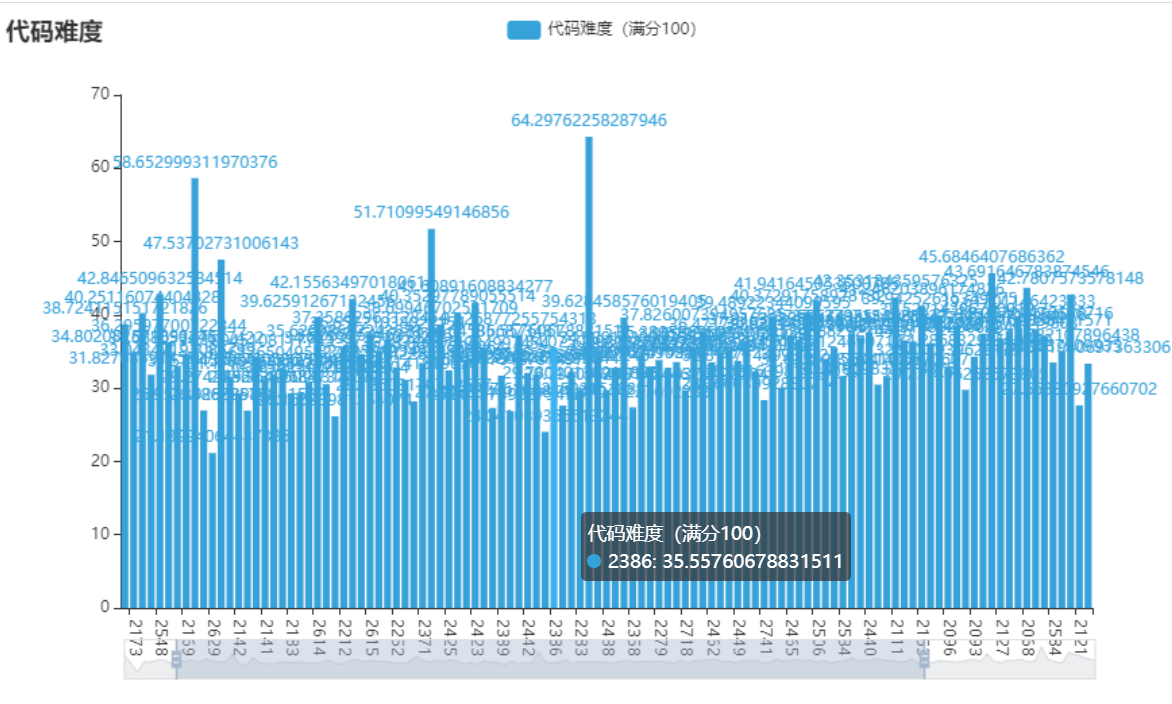


显而易见二者并没有明显的线性关系，说明采用加权平均的分析方法具有可行性

求最终的难度时，将影响因素中最大最小数据差值作为span,各个影响因素数值与该因素数值最小值的查占span的比值乘100以进行数值标准化，再乘以各影响因素的权重得出最终的难度值，其中偏度影响因素按照正负对最终成绩进行正负影响



最终成果如下：



### 2. 通过代码实现分类

### 实现分类的关键之处在于分类的标准，在此部分中，我们设想题目和实现代码之间是有联系的，即我们可以通过观察代码来知道这是一道什么题目，因此我们可以通过比对同学们的实现代码，找出其内在联系来实现分类

#### 1. 第一种思路：在一个同学所写的全部代码中，将每两题之间的相似度分别记下，接着，我们采样更多的同学，最终为每两道题对应的相似度取均值

#### 假设同学A和同学B做 题目1和题目2 两位同学的这两道题目都是满分

#### 然后求同学A 题目1和题目2的代码相似度 求同学B 题目1和题目2的代码相似度 最后取这两个相似度的平均值 我们也可以推广 不止有这两个同学

#### 这样做另外的一个原因是：因不同人代码风格不一样 可能同一道题 两个人的差异确实会比较高我们认为 如果两道题相似 那么同一个人来做 他的代码应该是会相似的

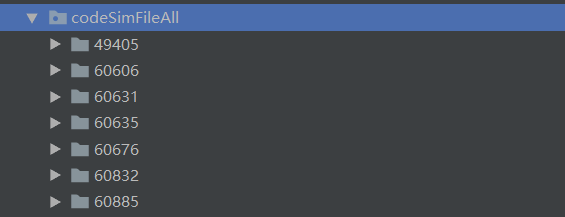
**以下为实现代码解释和研究过程思考**

由于代码总量较大，而且相似度计算运行起来耗时较长，我们最终选择抽取少量学生代码样本的研究方式。

**1. codeSimFileAll文件夹**

1. 采样方面，我从全部学生中筛选出了全部练习题满分的同学，其目的是为了能够保证每两道题都能进行比较一次，以及每次比较相对有效准确
2. 在全部题都满分的同学中，又进一步通过人工筛选，过滤存在面向用例（即通过直接print方法打印答案实现满分）的同学，因为这部分同学会对比较结果产生较大影响

* 下载代码如下:
* 
* 最终随机采样如下：



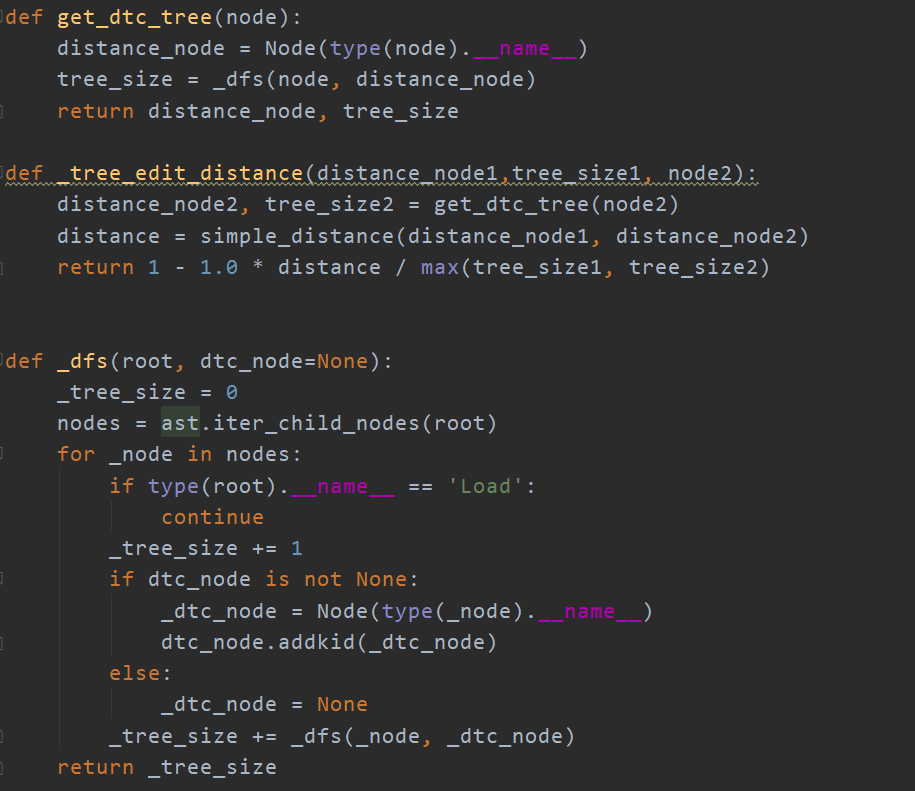
**2. analyse.py**

我们求代码相似度的方式为对两段代码进行抽象语法树解析，然后计算这两棵树的编辑距离，编辑距离是指，只用插入、删除和替换三种操作，最少需要多少步可以把A变成B。例如，从FAME到GATE需要两步（两次替换），从GAME到ACM则需要三步（删除G和E再添加C）。

计算编辑距离时选择引入第三方库zss，解析抽象语法树引入标准库中的ast

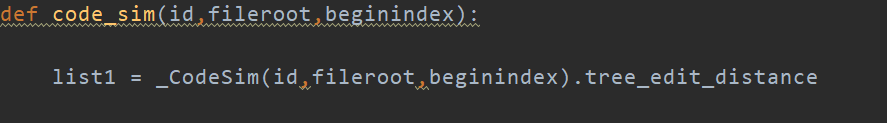
树编辑距离计算公式为1 - 1\*distance/max(tree*size1,tree*size2)，除数确保结果小于1

其中，树的大小通过深度优先遍历来计算，simple\_distance和Node是zss库中定义的计算编辑距离相关类和方法，传入变量为树的根节点



*CodeSim类的code*sim方法

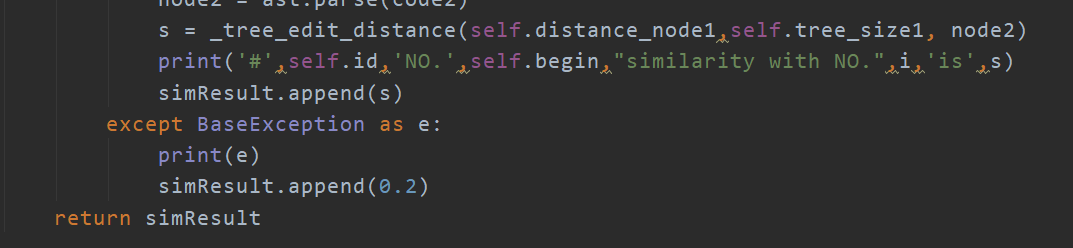
# fileroot 谁的文件夹 beginindex 从哪题开始逐步往后检测



\_CodeSim类

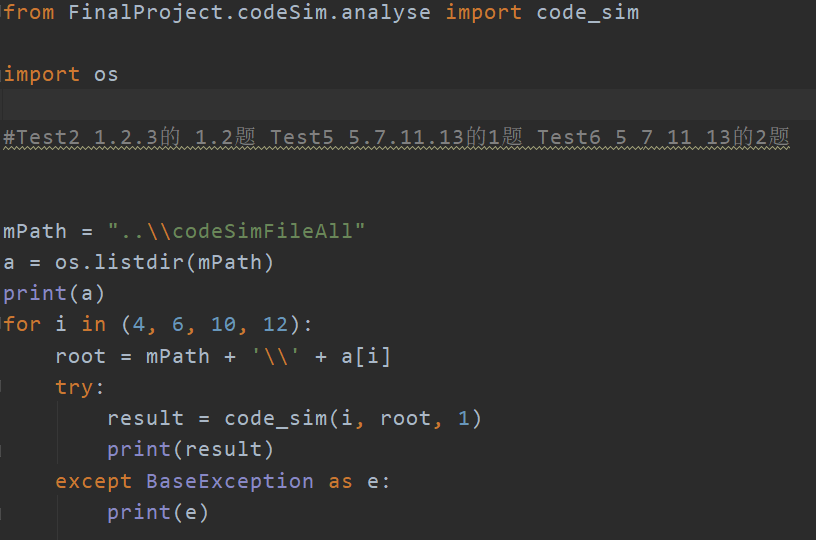
构造方法中为类内保存了比较主体的抽象树和编辑距离参数解析结果，避免了每次调用重新解析耗费时间，fileRoot是某个学生做题记录的根目录,beginIndex是开始往后比较的题目索引





**3. mainEntry.py**

入口函数，可以设置想要测试哪几个同学和他们的哪几题



**4. resultTest**

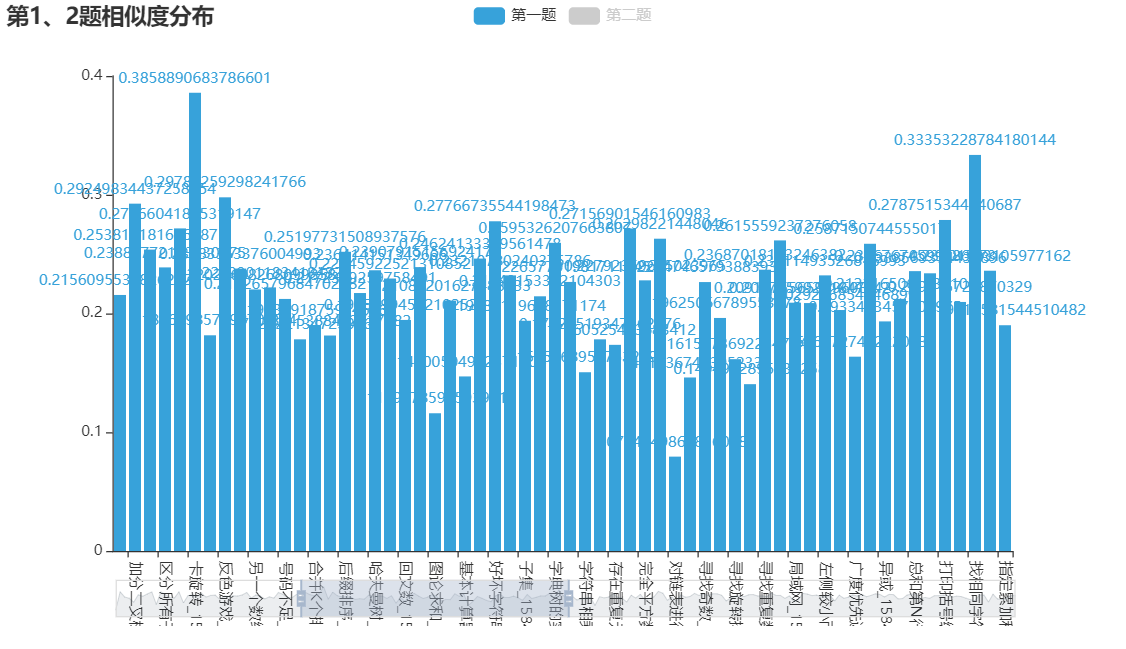
用于对生成的原始数据进行分析，如每题与其他题的相似度分布情况等

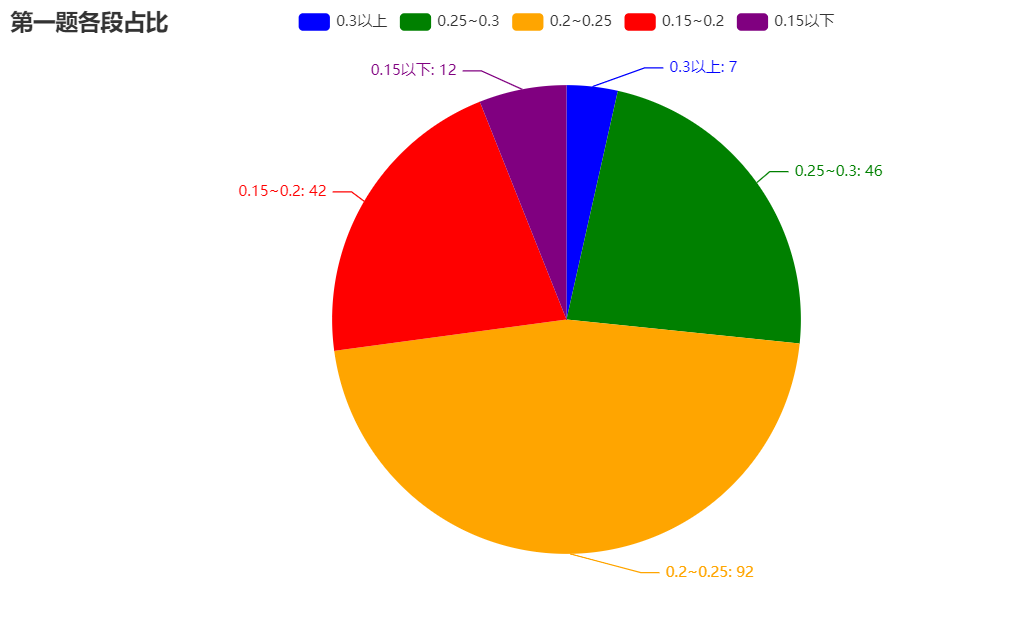
**5. buildGraph & graphTest**

功能上一样，用于生成分析结果的图表，方便更直观的进行分析

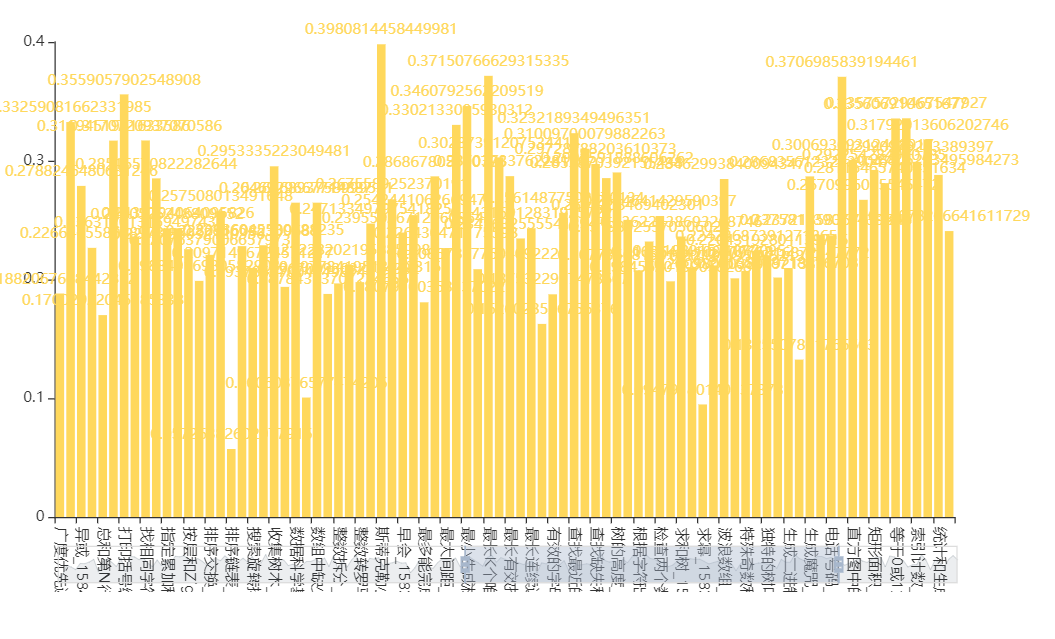
下面以第一第二题（即Array）分析结果举例（html数据图已存在源码文件中）

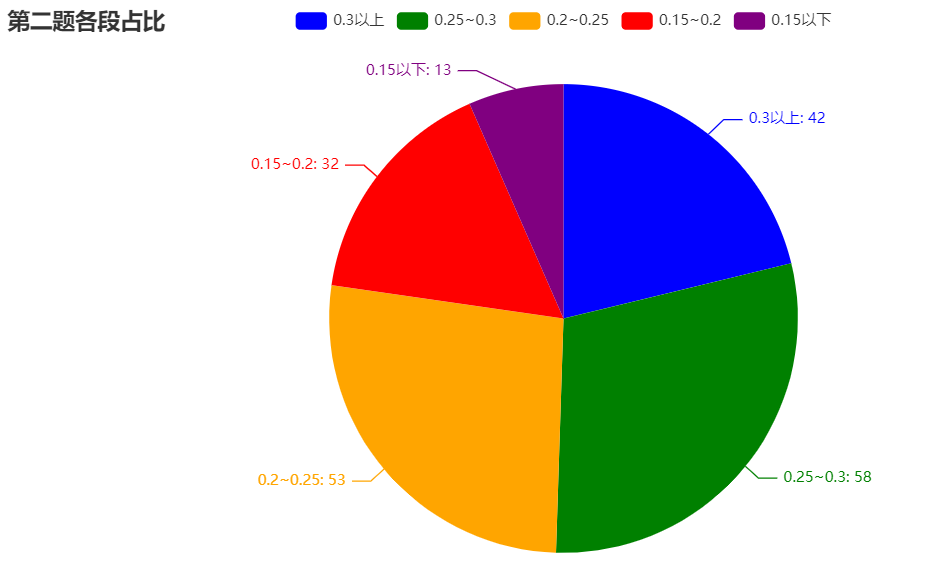
第一题的相似度片段：



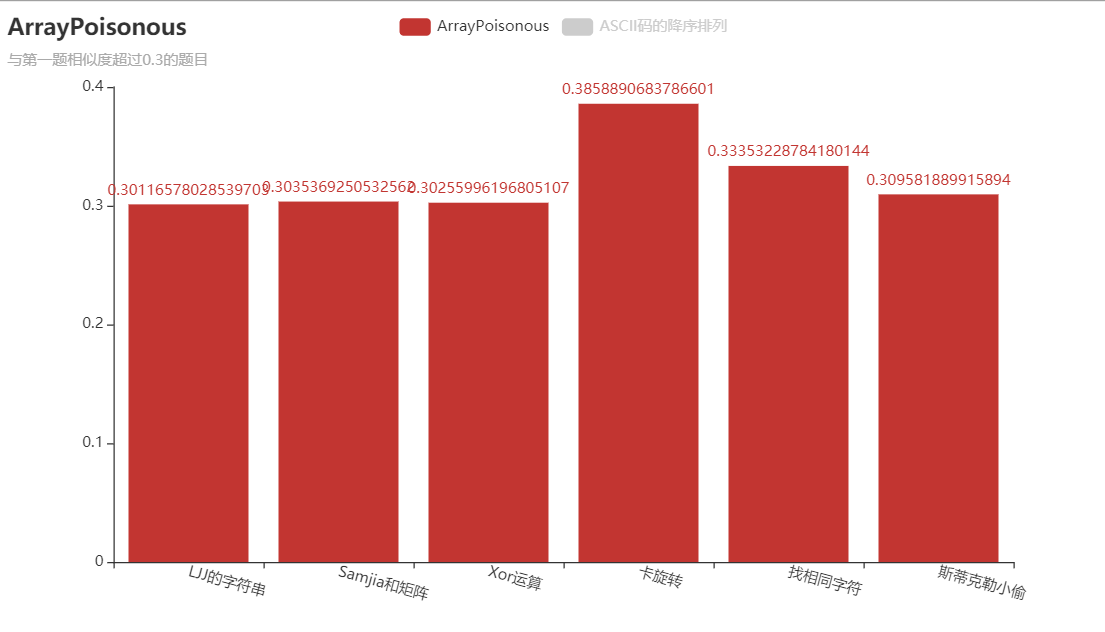


第二题的相似度片段：

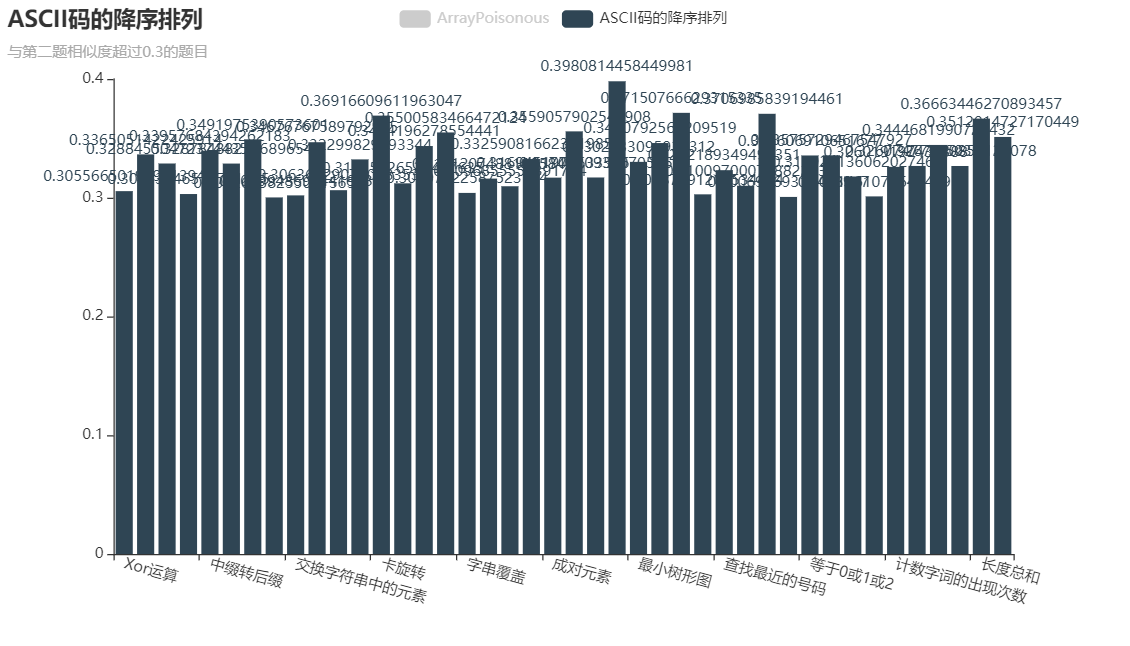




和第一题相似度超过0.3的题目如下：

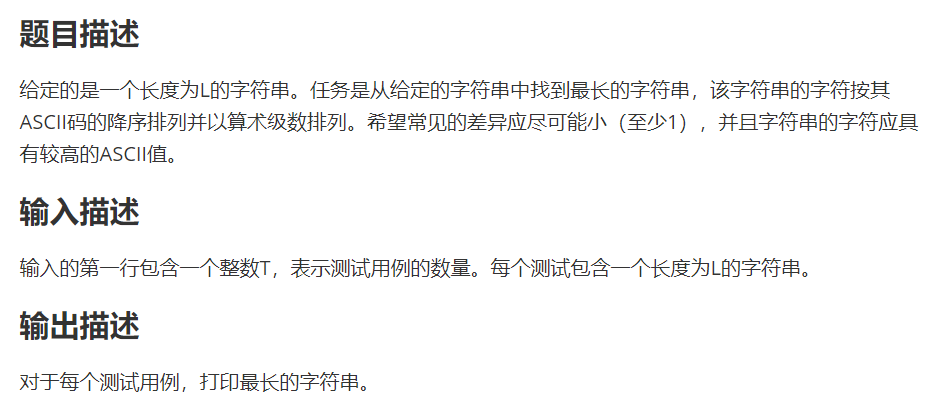


和第二题相似度超过0.3的题目如下：

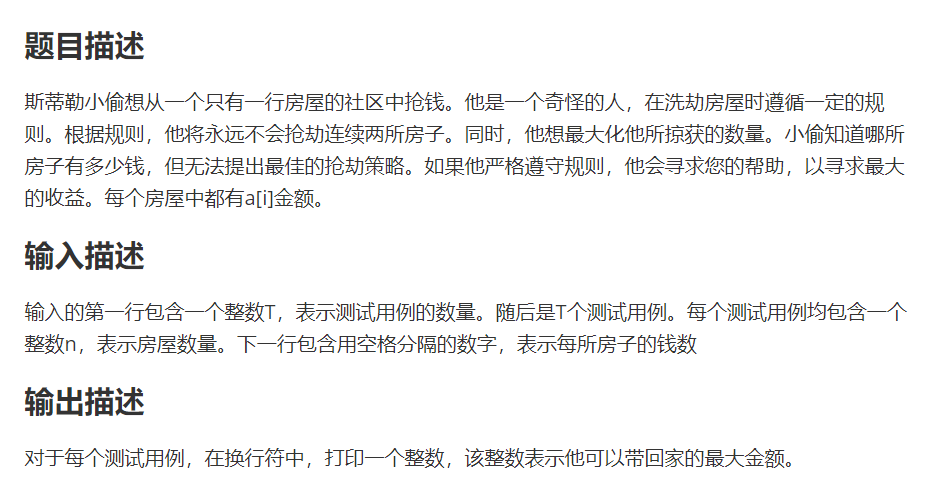


下面以与"ASCII码的降序排列"相似度最高的“斯蒂克勒小偷”为例进行分析：

**"ASCII码的降序排列":**



**“斯蒂克勒小偷”:**



**从解题思路上看，这两道题都需要遍历输入的数组（或是由输入转化生成的数组），另外，这两者都需要进行数组元素求和是否最大值的判断，在所采取的7位同学样本中，经过人工比对，这两道题解答代码所包含的循环层数最大值偏差仅为1.2，即接着两道题，同学们都用了相仿的循环层数，由此可见通过解答代码来进行题目分类适应了结题思路相仿的题目，具有一定的可行性**

**6. 拓展思考**

另外还进行了"相似的两道题是否都和另外的题相似"的思考：

已知“ArrayPoisonous”题目和“ASCII码的降序排列”相似度超过 0.3，经比对，在上述过程中求得的相似度平均值中，第一题和第二题偏差绝对值的平均值为0.04

下面是街区的相似度分布片段图：



可以明显的看出一个“同时高，同时低”的特点，因此可以猜测“相似“是可以传递的，分类时是可以根据"同类的同类"来进行

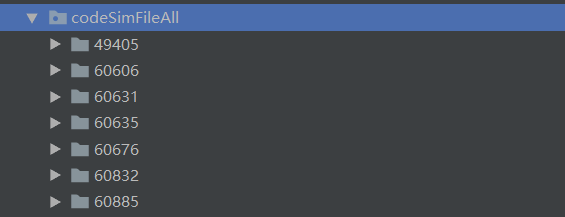
#### 2. 思路二：利用上述相似度工具，我们还可以用来进行横向比较，以达到寻找一题多解的目的

即，针对一道题目，对不同学生的代码进行相似度比较，

相似度越高，则解题方法越相似；相似度越低，则解题方法差异越大。

假设同学A做完ArrayPoisonous这一题时，想要了解这一题目的其他解法，可以通过相似度比较，来找寻与相似度较低的解法。

**数据集：**



在数据集中，针对ArraryPoisonous这一题目又进一步通过人工筛选，过滤存在面向用例（即通过直接print方法打印答案实现满分）的同学，因为这部分同学会对比较结果产生较大影响，最后选取49405、60635、60676、60832四位学生的代码进行比较测试。其中，又以49405为主，分别与其他三位学生的代码进行比较。

在此贴上49405学生的代码。

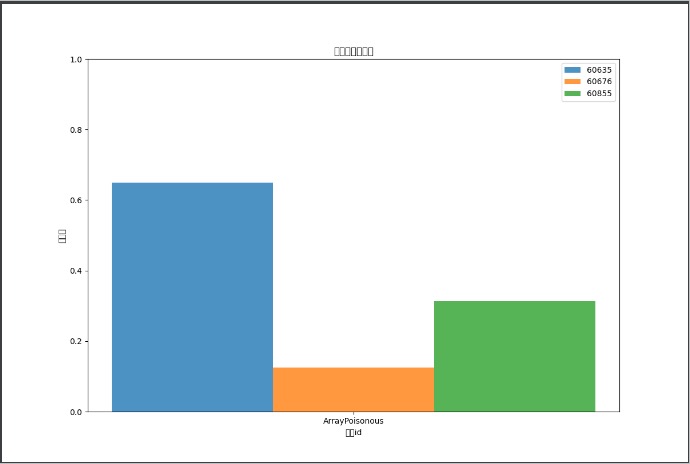
k = int(input())  
print(k + 1)  
ans = [0 for i in range(k + 2)]  
l, r = 0, k + 2  
for i in range(k + 1, 0, -1):  
 if (k - i + 2) % 2 == 1:  
 r -= 1  
 ans[i] = r  
 else:  
 l += 1  
 ans[i] = l  
print(' '.join(map(str, ans[1:])), end=' ')

相似度比较代码依然使用上述工具。

生成图标代码

import matplotlib.pyplot as plt  
import data\_source  
  
mobile\_176xxxx4617 = data\_source.60635  
mobile\_155xxxx9617 = data\_source.60676  
mobile\_173xxxx9636 = data\_source.60855  
exam = data\_source.time  
  
time\_len = len(exam)  
xtick\_index = range(int(exam\_len))  
  
plt.figure(figsize=(12, 8)) # 建立图形  
  
"""  
绘制条形图  
left:长条形中点横坐标  
height:长条形高度  
width:长条形宽度，默认值0.8  
alpha:透明度  
color:颜色  
label:标签，为后面设置legend准备  
"""  
bar1 = plt.bar([i - 0.2 for i in xtick\_index], mobile\_176xxxx4617, width=0.2,  
 alpha=0.8, label='60635') # 第一个图  
  
bar2 = plt.bar([i for i in xtick\_index], mobile\_155xxxx9617, width=0.2,  
 alpha=0.8, label='60676') # 第二个图  
  
bar3 = plt.bar([i + 0.2 for i in xtick\_index], mobile\_173xxxx9636, width=0.2,  
 alpha=0.8, label='60855') # 第三个图  
  
plt.xticks(xtick\_index, exam) # 设置x轴刻度显示值  
plt.ylim(0, 1) # y轴的范围  
plt.title('相似度横向比较') # 标题  
plt.xlabel('题目id') # x轴的标签  
plt.ylabel('相似度') # y轴的标签  
plt.legend() # 设置图例  
  
'''  
get\_height:获取值  
get\_x：获取x轴的位置  
get\_width:获取图形的宽度  
text(x,y,s,fontsize,ha,va)  
 x,y:表示坐标值上的值  
 s:表示说明文字  
 fontsize:表示字体大小  
 ha：垂直显示方式{'centee':'中心', 'right':'右', 'left':'左'}  
 va：水平显示方式{'center':'中心', 'top':'下', 'bottom':'上', 'baseline':'基线'}   
'''  
  
for rect in bar1:  
 height = rect.get\_height() # 获得bar1的高度  
 plt.text(rect.get\_x() + rect.get\_width() / 2, height + 3, str(height), fontsize=6, ha="center", va="bottom")  
for rect in bar2:  
 height = rect.get\_height()  
 plt.text(rect.get\_x() + rect.get\_width() / 2, height + 3, str(height), fontsize=6, ha="center", va="bottom")  
for rect in bar3:  
 height = rect.get\_height()  
 plt.text(rect.get\_x() + rect.get\_width() / 2, height + 3, str(height), fontsize=6, ha="center", va="bottom")  
  
  
def show\_plt():  
 plt.show()

比较结果可见：



从图表清晰可见，49405学生所写的代码与60635学生代码相似度达到了0.6之上，远超0.3。

按照预期，两者代码应当非常相似，查看60635学生代码后：

k = int(input())  
n = k + 1  
print(n)  
l = 0  
r = n + 1  
ans = [0] \* (n + 1)  
for i in range(n, 0, -1):  
 if (n + 1 - i) & 1:  
 r -= 1  
 ans[i] = r  
 else:  
 l += 1  
 ans[i] = l  
print(\*ans[1:], end=' ')

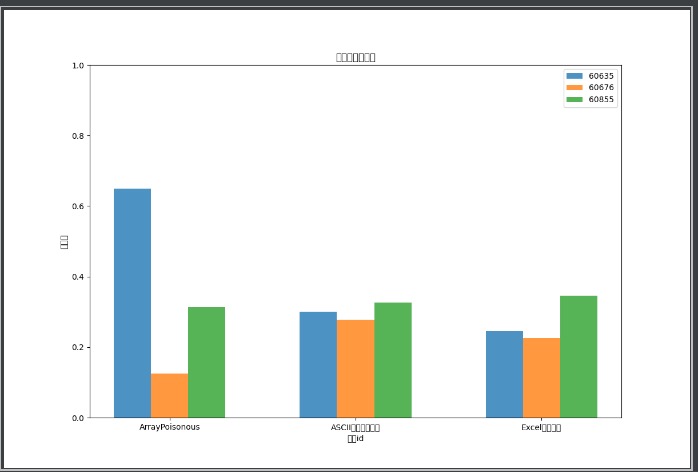
可以判断出，两者解题方法大致相同，符合预期。

图表中，与60676代码相似度较低，推测代码结构及方法相差较大。

查看60676学生代码后，大致符合预期：

from numpy import argsort  
  
  
def suffix\_array(s):  
 if s is None or len(s) == 0:  
 return None  
 suffix = []  
 for i in range(len(s)):  
 suffix.append(s[i:])  
 a = argsort(suffix)  
 for i in range(len(s)):  
 suffix[i] = 1 + a[i]  
 return suffix  
  
  
def rank(s):  
 sa = suffix\_array(s)  
 res = sa.copy()  
 for i in range(len(sa)):  
 res[sa[i]-1] = i+1  
 return res  
  
  
def permutation(arr, begin, end):  
 if begin == end-1:  
 return [arr.copy()]  
 else:  
 res = []  
 i = begin  
 for num in range(begin, end):  
 arr[num], arr[i] = arr[i], arr[num]  
 p = permutation(arr, begin + 1, end)  
 for j in p:  
 res.append(j)  
 arr[num], arr[i] = arr[i], arr[num]  
 return res  
  
  
def advanced\_permutation(arr, length):  
 if length == 1:  
 return arr  
 else:  
 res = []  
 for i in range(len(arr)):  
 ap = advanced\_permutation(arr, length-1)  
 for j in ap:  
 res.append(arr[i] + j)  
 return res  
  
  
def poisonous\_string\_problem(k):  
 elements = []  
 for i in range(k):  
 elements.append(chr(ord('a') + i))  
 n = k + 1  
 res = []  
 while len(res) == 0:  
 temp = permutation([i for i in range(1, n+1)], 0, n)  
 ape = advanced\_permutation(elements, n)  
 ranks = []  
 for i in range(len(ape)):  
 ranks.append(rank(ape[i]))  
 for i in range(len(temp)):  
 if temp[i] not in ranks:  
 res = temp[i]  
 break  
 n += 1  
 return n-1, res  
  
  
k = int(input())  
n = k + 1  
result = []  
for i in range(n):  
 if i % 2 == 0:  
 result.insert(0, n - i//2)  
 else:  
 result.insert(0, i//2 + 1)  
print(n)  
for i in range(n):  
 print(result[i], end=' ')

更多例子：



##### \*\* 思路补充\*\*

此外，我们还设想了一种根据代码构成要素来对代码分类的方法，由于时间因素，我们没有展开，仅当做对分类思路的一种补充。

首先需要对同学代码进行预处理，代码中的一些冗余信息可能会对程序特征的提取有影响，例如：注释、头文件、空格和空行等。所以在代码的特征词提取之前，首先得去除掉程序代码中对特征提取有影响的冗余信息。这个过程不仅可以使程序特征的提取更为精确，还可以使源代码文件大大减小，加快整个分类算法的运行效率。

根据代码构成要素对代码分类，可以利用正则表达式对 Python程序代码进行特征词的提取，根据相似度检测所需要的 Python语言的基本语法特征，制定了一个 Python语法特征词和正则表达式的对应关系表：

|  |  |
| --- | --- |
| 特征词 | 正则表达式 |
| import | r’\bimprot\b’ |
| def | r’\bdef\b’ |
| class | r’\bclass\b’ |
| if | elif | r’\bif\b|\belif\b’ |
| else | r’\belse\b’ |
| for | r’\bfor\b’ |
| while | r’\bwhile\b’ |
| and | r’\band\b’ |
| or | r’\bor\b’ |
| not | r’\bnot\b’ |
| False | r’\bFalse\b’ |
| True | r’\bTrue\b’ |
| None | r’\bNone\b’ |
| is | r’\bis\b’ |
| return | r’\breturn\b’ |
| in | r’\bin\b’ |
| print | r’\bprint\b’ |
| + | - | \* | / | % | r’+|-|\*|/|%’ |
| = | r’\=’ |
| == | r’\==’ |
| > | < | r’>|<’ |
| [ | r’[’ |

在对代码进行特征词提取的过程中，系统根据表中所列的所有语法要素的正则表达式，构建了一个正则特征向量。再利用正则表达式对已预处理过的代码进行特征词比对，提取出当前代码中对应特征词的个数，生成一个与该代码所对应的特征向量。

但是我们不能简单地根据这些特征词在一个 Python程序中出现的次数所构成的特征向量来代表这个 Python程序。 还必须根据每一个特征词在 Python语言中的重要性来对特征向量中的每一个元素进行加权处理，突显出每一个特征词的重要性。

### 对老师说的话