数据结构与算法-Day01

算法概述

- **算法-前序**

```
# 全场动作必须跟我整齐划一，来，我们一起来做一道题
若n1+n2+n3=1000,且n1^2+n2^2=n3^2(n1,n2,n3为自然数),求出所有n1、n2、n3可能的组合

# 思路1:
n1 = 0
n2 = 0
n3 = 0
判断n1+n2+n3是否等于1000,之后变n3=1,n3=2,n3=3,... 然后再变n2
那如果变为 n1+n2+n3=2000 了呢?

# 思路1代码实现
import time

start_time = time.time()
for n1 in range(0,1001):
    for n2 in range(0,1001):
        for n3 in range(0,1001):
            if n1 + n2 + n3 == 1000 and n1**2 + n2**2 == n3**2:
                print('[%d,%d,%d]' % (n1,n2,n3))
end_time = time.time()
print('执行时间:%.2f' % (end_time-start_time))
```

- **算法-概念**

是指解题方案的准确而完整的描述，是一系列解决问题的清晰指令，算法代表着用系统的方法描述解决问题的策略机制。也就是说，能够对一定规范的输入，在有限时间内获得所要求的输出

- **算法五大特性**

```
1) 输入    -- 具有0个或多个输入
2) 输出    -- 至少由1个或者多个输出
3) 有穷性 -- 算法执行的步骤是有限的
4) 确定性 -- 每个计算步骤无二义性
5) 可行性 -- 每个计算步骤能够在有限的时间内完成
```

时间复杂度概述

- **时间复杂度 - 前序**

```
# 各位，一万年太久，只争朝夕，来提升一下上题的效率吧！！！
for n1 in range(0,1001):
    for n2 in range(0,1001):
        n3 = 1000 - n1 - n2
        if n1**2 + n2**2 == n3**2:
            print('[%d,%d,%d]'%(n1,n2,n3))

# 总结与思考
解决同一个问题有多种算法,但是效率有区别,那么如何衡量呢?
1）执行时间反应算法效率 - 绝对靠谱吗?
不是绝对靠谱：因机器配置有高有低,不能冒然绝对去做衡量

2）那如何衡量更靠谱？？？
运算数量 - 执行步骤的数量
```

- **时间复杂度概念**

```
同一个算法，由于机器配置差异,每台机器执行的总时间不同,但是执行基本运算的数量大体相同，所以把算法执行步骤的数量称为时间复杂度

# 总结
时间复杂度：程序执行步骤的数量
```

- **时间复杂度 - 大O表示法前序**

```
#####################################################################
for n1 in range(0,1001):
    for n2 in range(0,1001):
        for n3 in range(0,1001):
            if n1 + n2 + n3 == 1000 and n1**2 + n2**2 == n3**2:
                print('[%d,%d,%d]' % (n1,n2,n3))
#####################################################################

# 计算时间复杂度 - 执行计算步骤的次数
T = 1000 * 1000 * 1000 * 2
T = n * n * n * 2
T(n) = n ** 3 * 2  --> 则时间复杂度为T(n) 及 n**3 * 2


# 渐近函数 - 数学概念
1) 函数1：T(n) = k* g(n) + c ---> k为系数,c为常数
2) 函数2：g(n) = n ** 3
```

特点：在趋向无穷的极限意义下，函数T(n)的增长速度受到函数g(n)的约束，也为函数T(n)与函数g(n)的特征相似，则称 g(n) 是 T(n) 的渐近函数，大O表示法则使用渐近函数来表示，
即：O(g(n))
即：O(n^3)
即：上述时间复杂度为 O(n^3)

总结：什么是时间复杂度?
衡量标准：运算步骤来衡量,n代表解决问题的规模问题,对于同一类问题所花费的步骤有一个统一的表示,这个T(n)为时间复杂度,n**3为它的大O表示法,即O(n^3)

- **时间复杂度表示 - 大O表示法**

1.需要理解
假定计算机执行算法每个基本操作的时间是固定的一个时间单位，则有多少个基本操作就代表会花费多少时间单位。虽然对于不同机器环境切确的时间单位不同，但对于算法进行的基本操作数量在规模数量级上相同，因此可以忽略机器环境影响而客观反映算法的时间效率，用"大O记法"表示

2. 需要记忆
时间复杂度：假设存在函数g，使得算法A处理规模为n的问题所用时间为T(n)=O(g(n))，则称O(g(n))为算法A的渐近时间复杂度，简称时间复杂度，记为T(n)

3. 大O表示法
对算法进行特别具体细致分析虽然好，但实践中实际价值有限。对我们来说算法的时间性质和空间性质最重要的是数量级和趋势，这些是分析算法效率的主要部分。
所以忽略系数，忽略常数，比如5*n^2 和 100*n^2属于一个量级，时间复杂度为O(n^2)

- **时间复杂度分类**

1.分类
1）最优时间复杂度 - 最少需要多少个步骤
2）最坏时间复杂度 - 最多需要多少个步骤
3）平均时间复杂度 - 平均需要多少个步骤
我们平时所说的时间复杂度,指的是最坏时间复杂度

时间复杂度 - 计算规则

- **计算原则**

```
# 1. 基本操作,只有常系数，认为其时间复杂度为O(1)
  顺序 - 基本步骤之间的累加
  print('abc') -> O(1)
  print('abc') -> O(1)
# 2. 循环：时间复杂度按乘法进行计算
# 3. 分支：时间复杂度取最大值(哪个分支执行次数多算哪个)

# 练习:请计算如下代码的时间复杂度
```

```
for n1 in range(0,1001):
    for n2 in range(0,1001):
        n3 = 1000 - n1 - n2
        if n1**2 + n2**2 == n3**2:
            print('[%d,%d,%d]'%(n1,n2,n3))

T(n) = n * n * (1+1+max(1,0))
T(n) = n**2 * 3
T(n) = n**2
T(n) = O(n**2)
```
用大O表示法表示为 O(n^2)

- **常见时间复杂度**

执行次数	时间复杂度	阶
20（20个基本步骤）	O(1)	常数阶
8n+6	O(n)	线性阶
2n^2 + 4n + 2	O(n^2)	平方阶
8logn + 16	O(logn)	对数阶
4n + 3nlogn + 22	O(nlog(n))	nlog阶
2n^3 + 2n^2 + 4	O(n^3)	立方阶
2 ^ n	O(2^n)	指数阶

- **O(1)**

```
【1】O(1)
print('全场动作必须跟我整齐划一')

【2】O(1)
print('左边跟我一起画个龙')
print('在你右边画一道彩虹')
print('走起')
print('左边跟我一起画彩虹')
print('在你右边再画一条龙')
```

- **O(n)**

```
for i in range(n):
  print('在胸口比划一个郭富城')
```

- **O(n^2)**

```
【1】O(n^2)
```

```
for i in range(n):
  for j in range(n):
    print('左边右边摇摇头')

【2】O(n^2)
for i in range(n):
  print('两根手指就像两个窜天猴')
  for j in range(n):
    print('指向闪耀的灯球')

【3】O(n^2)
for i in range(n):
  for j in range(i):
    print('野狼disco')
```

- **O(n^3)**

```
for i in range(n):
    for j in range(n):
        for k in range(n):
            print('走你')
```

- **O(logn)**

```
n = 64
while n > 1:
    print(n)
    n = n // 2

【解释】
2的6次方 等于 64，log264 = 6，所以循环减半的时间复杂度为O(log2n)，即O(logn)
如果是循环减半的过程，时间复杂度为O(logn)或O(log2n)
```

- **O(nlogn)**

```
n = 64
while n > 1:
    for i in range(n):
      print('哪里有彩虹告诉我')
    n = n // 2
```

- **常见时间复杂度排序**

```
O(1)<O(logn)<O(n)<O(nlogn)<O(n2)<O(n2logn)<O(n3)
```

- **练习: 写出如下的时间复杂度**

```
O(5)           --> O(1)
O(2n+1)        --> O(n)
O(n**2+n+1)    --> O(n**2)
O(3n**3+1)     --> O(n**3)
```

数据结构概述

- **数据结构**

```
# 1. 思考
"请利用Python数据类型保存所有学员信息，并通过姓名获取对应学员信息！"
1) 列表存：时间复杂度O(n),因为需要遍历整个列表
2) 字典存：时间复杂度O(1),直接通过key取值

# 2. 描述
在工作中，我们为了解决问题，需要将数据保存下来，然后根据数据存储方式设计算法进行处理，根据数据的存储方式我们使用不同的算法处理，而我们现在需要考虑算法解决问题的效率问题，所以需要考虑数据究竟如何保存，这就是数据结构。在示例中我们引用列表或字典存储学员信息，而列表和字典只是Python帮我们封装好的两种数据结构而已

# 3. 概念
1) 数据是一个抽象的概念，将其进行分类后得到程序设计语言中的基本类型，如：list、tuple、int等。数据元素之间不是独立的，存在特定的关系，这些关系便是结构。数据结构指数据对象中数据元素之间的关系
2)Python提供了很多现成的数据结构类型，如列表、元组、字典等，无须我们自己去定义，而Python没有定义的，就需要我们自己去定义实现这些数据的组织方式，称为Python扩展数据结构，如：栈、队列等

# 4. 为什么学习数据结构
实际上，在真正的项目开发中，大部分时间都是 从数据库取数据 -> 数据操作和结构化 -> 返回给前端，在数据操作过程中需要合理地抽象，组织、处理数据，如果选用了错误的数据结构，就会造成代码运行低效
```

- **数据结构 - 关系**

```
# 1. 逻辑结构
表示数据之间的抽象关系（如邻接关系、从属关系等），按每个元素可能具有的直接前趋数和直接后继数将逻辑结构分为“线性结构”和“非线性结构”两大类
1) 线性结构：线性结构是n个数据元素的有序集合
2) 树形结构：树形结构指的是数据元素之间存在着“一对多”的树形关系的数据结构，是一类重要的非线性数据结构。在树形结构中，树根结点没有前驱结点，其余每个结点有且只有一个前驱结点。叶子结点没有后续结点，其余每个结点的后续节点数可以是一个也可以是多个
3) 图状结构：图是一种比较复杂的数据结构。在图结构中任意两个元素之间都可能有关系，也就是说这是一种多对多的关系
4) 其他结构：除了以上几种常见的逻辑结构外，数据结构中还包含其他的结构，比如集合等。有时根据实际情况抽象的模型不止是简单的某一种，也可能拥有更多的特征

# 2. 存储结构
1) 顺序存储：顺序存储（Sequential Storage）：将数据结构中各元素按照其逻辑顺序存放于存储器一片连续的存储空间中
2) 链式存储：将数据结构中各元素分布到存储器的不同点，用记录下一个结点位置的方式建立它们之间的联系，由此得到的存储结构为链式存储结构
```

- **数据结构+算法总结**

```
1) 数据结构只是静态描述了数据元素之间的关系
2) 高效的程序需要在数据结构的基础上设计和选择算法
3) 程序 = 数据结构 + 算法
4) 算法是为了解决实际问题而设计的，数据结构是算法需要处理的问题载体
```

抽象数据类型

- **概念**

```
# 1. 定义
抽象数据类型是指一个数学模型以及定义在此数学模型上的一组操作，及把数据类型和数据类型上的运算捆在一起进行封装。引入抽象数据类型的目的是把数据类型的表示和数据类型上的运算的实现与这些数据类型和运算在程序中的引用隔开，使他们相互独立

# 2. 描述
把原有的基本数据和这个数据所支持的操作放到一起，形成一个整体

# 3. 最常用的数据运算
1) 插入
2) 删除
3) 修改
4) 查找
5) 排序
```

线性表 - 顺序表

- **顺序表 - 概念**

在程序中，经常需要将一组(通常是同为某个类型)的数据元素作为整体管理和使用，需要创建这种元素组，用变量记录它们，传进传出函数等。一组数据中包含的元素个数可能发生变化（可以增加或删除元素）。对于这种需求，最简单的解决方案是将这样的一组元素看成一个序列，用元素在序列里的位置和顺序，表示实际应用中的某种有意义的信息，或者表示数据之间的某种关系。

这样的一组序列元素的组织形式，我们将其抽象为"线性表",一个线性表是某类元素的一个集合，和记录着元素之间的一种顺序关系。线性表是最基本的数据结构之一

根据线性表的实际存储方式，分为两种实现模型：
1) 顺序表：将元素顺序地存放在一块连续地存储区域里,元素间的顺序关系由它们的存储顺序表示
2) 链表：将元素存放在通过链接构造起来的一系列存储块中

- **顺序表的基本形式**

1) 基本形式：数据元素本身连续存储,每个元素所占存储单元大小固定相同
2) 元素外置：数据元素不连续存储，地址单元连续存储

线性表 - 链表

- **定义**

和顺序表结构不同，链式结构内存不连续的，而是一个个串起来的，这个时候就需要每个链接表的节点保存一个指向下一个节点的指针
顺序表的构建需要预先知道数据大小来申请连续的存储空间，而在进行扩充时又需要进行数据的搬迁，使用起来不灵活，而链表结构可以充分利用计算机的内存空间，实现灵活的内存动态管理

- **示例 - 强化理解**

将线性表L=(a0,a1,……,an-1)中各元素分布在存储器的不同存储块，称为结点，每个结点（尾节点除外）中都持有一个指向下一个节点的引用，这样所得到的存储结构为链表结构

- **单链表 - 代码实现**

```
# 单链表
class Node(object):
    """节点"""
    def __init__(self,elem):
        self.elem = elem
        self.next = None

class SingleLinkList(object):
```

```
    """单链表"""
    def __init__(self,node=None):
        self.head = node

    def is_empty(self):
        """链表是否为空"""
        return self.head == None

    def length(self):
        """链表长度"""
        # current游标,用来移动遍历节点
        current = self.head
        # count记录数量
        count = 0
        while current != None:
            count += 1
            current = current.next
        return count

    def travel(self):
        """遍历整个链表"""
        current = self.head
        while current != None:
            print(current.elem,end=" ")
            current = current.next
        print()

    def add(self,item):
        """链表头部添加元素"""
        node = Node(item)
        node.next = self.head
        self.head = node

    def append(self,item):
        """链表尾部添加元素"""
        node = Node(item)
        if self.is_empty():
            self.head = node
        else:
            current = self.head
            while current.next != None:
                current = current.next
            current.next = node

    def insert(self,position,item):
        """指定位置添加元素
        :param pos 从0开始
        """
        if position <= 0:
```

```
            self.add(item)
        elif position > self.length()-1:
            self.append(item)
        else:
            pre = self.head
            count = 0
            while count < (position-1):
                count += 1
                pre = pre.next
            # 当循环结束后,pre指向pos-1的位置
            node = Node(item)
            node.next = pre.next
            pre.next = node

    def remove(self,item):
        """删除节点
        特殊情况:1.空链表
        特殊情况:2.恰巧删除头节点
        特殊情况:3.只有1个节点
        """
        current = self.head
        pre = None
        while current != None:
            if current.elem == item:
                # 先判断此节点是否为头节点
                if current == self.head:
                    self.head = current.next
                else:
                    pre.next = current.next
                break
            else:
                pre = current
                current = current.next

    def search(self,item):
        """查找节点是否存在"""
        current = self.head
        while current != None:
            if current.elem == item:
                return True
            else:
                current = current.next
        return False

if __name__ == '__main__':
    sll = SingleLinkList()
    print(sll.is_empty())
    print(sll.length())
```

```
    sll.add(1)
    sll.append(2)
    print(sll.is_empty())
    print(sll.length())

    sll.append(3)
    sll.append(4)
    sll.append(5)
    sll.append(6)
    sll.insert(-1,'我是-1')
    sll.insert(2,'我是2')
    sll.travel()
    sll.remove(4)
    sll.travel()
```

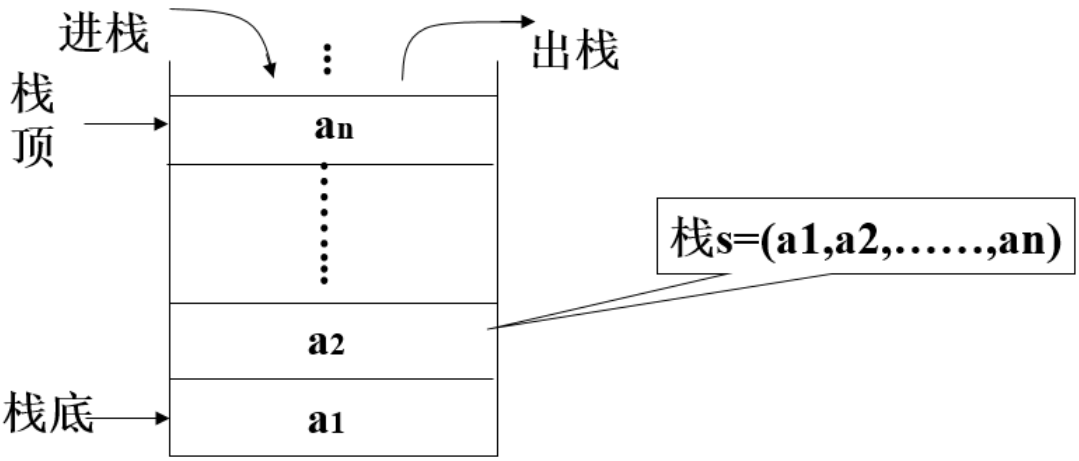
栈（LIFO)

- 定义

> 栈是限制在一端进行插入操作和删除操作的线性表（俗称堆栈），允许进行操作的一端称为"栈顶"，另一固定端称为"栈底"，当栈中没有元素时称为"空栈"

- 特点

> 1）栈只能在一端进行数据操作
> 2）栈模型具有先进后出或者叫做后进先出的规律



- 栈的代码实现

```
# 栈的操作有入栈（压栈），出栈（弹栈），判断栈是否为空等操作
"""
sstack.py  栈模型的顺序存储
重点代码

思路：
```

```
1. 利用列表完成顺序存储,但是列表功能多,不符合栈模型特点
2. 使用类将列表封装,提供符合栈特点的接口方法
"""

# 顺序栈模型
class Stack(object):
    def __init__(self):
        # 开辟一个顺序存储的模型空间
        # 列表的尾部表示栈顶
        self.elems = []

    def is_empty(self):
        """判断栈是否为空"""
        return self.elems == []

    def push(self,val):
        """入栈"""
        self.elems.append(val)

    def pop(self):
        """出栈"""
        if self.is_empty():
            raise StackError("pop from empty stack")
        # 弹出一个值并返回
        return self.elems.pop()

    def top(self):
        """查看栈顶元素"""
        if self.is_empty():
            raise StackError("Stack is empty")
        return self.elems[0]


if __name__ == '__main__':
    st = Stack()
    st.push(1)
    st.push(3)
    st.push(5)
    print(st.top())
    while not st.is_empty():
        print(st.pop())
```

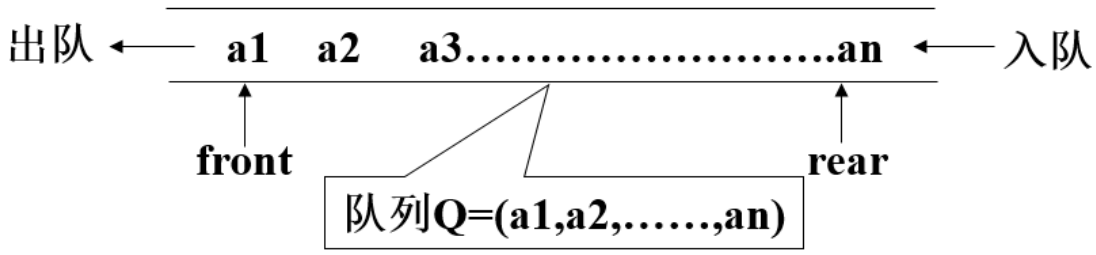
队列（FIFO）

- **定义**

> 队列是限制在两端进行插入操作和删除操作的线性表，允许进行存入操作的一端称为"队尾"，允许进行删除操作的一端称为"队头"

- **特点**

1) 队列只能在队头和队尾进行数据操作
2) 队列模型具有先进先出或者叫做后进后出的规律



- **队列的代码实现**

```
# 队列的操作有入队，出队，判断队列的空满等操作
"""
思路分析：
1. 基于列表完成数据的存储
2. 通过封装功能完成队列的基本行为
3. 无论那边做对头/队尾 都会在操作中有内存移动
"""

# 队列操作
class SQueue:
    def __init__(self):
        self.elems = []

    # 判断队列是否为空
    def is_empty(self):
        return self.elems == []

    # 入队
    def enqueue(self,val):
        self.elems.append(val)

    # 出队
    def dequeue(self):
        if not self._elems:
            raise Exception("Queue is empty")
        return self.elems.pop(0) # 弹出第一个数据


if __name__ == '__main__':
    sq = SQueue()
    sq.enqueue(10)
    sq.enqueue(20)
```

```
    sq.enqueue(30)
    while not sq.is_empty():
        print(sq.dequeue())
```

递归

- **递归定义**

> 递归用一种通俗的话来说就是自己调用自己，但是需要分解它的参数，让它解决一个更小一点的问题，当问题小到一定规模的时候，需要一个递归出口返回

- **递归示例**

```
# 求阶乘
def fact(n):
    if n == 0:
        return 1
    else:
        return n * fact(n-1)
```

- **递归的特点**

> 1) 递归必须包含一个基本的出口，否则就会无限递归，最终导致栈溢出，比如这里就是 `n == 0` 返回 `1`
>
> 2) 递归必须包含一个可以分解的问题，要想求得 `fact(n)`，就需要用 `n * fact(n-1)`
>
> 3) 递归必须必须要向着递归出口靠近，这里每次递归调用都会 `n-1`，向着递归出口 `n == 0` 靠近

- **深入理解 - 递归到底是如何工作的**

```
# 案例：终端输出 1-n
# 1.正常人写法
def f1(n):
    for i in range(1, n + 1):
        print(i)

# 2. 递归版本1
def recu_one(n):
    if n > 0:
        recu_one(n-1)
        print(n)

# 3. 思考如下递归执行过程?
def recu_two(n):
    if n > 0:
        print(n)
```

```
        recu_two(n-1)
```

- **递归原理**

  计算机内部使用调用栈来实现递归，后进先出，每当进入递归函数的时候，系统都会为当前函数开辟内存保存当前变量值等信息，每个调用栈之间的数据互不影响，新调用的函数，入栈的时候会放在栈顶

今日作业

```
# 1. 面试题：小明爬楼梯，一次只能上1级或者2级台阶，一共有n级台阶，一共有多少种方法上台阶？
"""
思路提示
如果有一级台阶，方法有1种
如果有两级台阶，方法有2种
如果台阶数再增加，大于三个台阶以后，可以认为是只有一二级台阶的一个重复实现
"""
```