# 数据结构与算法（Python）

# Why？

我们举一个可能不太恰当的例子：

如果将最终写好运行的程序比作战场，我们码农便是指挥作战的将军，而我们所写的代码便是士兵和武器。

**那么数据结构和算法是什么？答曰：兵法！**

我们可以不看兵法在战场上肉搏，如此，可能会胜利，可能会失败。即使胜利，可能也会付出巨大的代价。我们写程序亦然：没有看过数据结构和算法，有时面对问题可能会没有任何思路，不知如何下手去解决；大部分时间可能解决了问题，可是对程序运行的效率和开销没有意识，性能低下；有时会借助别人开发的利器暂时解决了问题，可是遇到性能瓶颈的时候，又不知该如何进行针对性的优化。

如果我们常看兵法，便可做到胸有成竹，有时会事半功倍！同样，如果我们常看数据结构与算法，我们写程序时也能游刃有余、明察秋毫，遇到问题时亦能入木三分、迎刃而解。

**故，数据结构和算法是一名程序开发人员的必备基本功，不是一朝一夕就能练成绝世高手的。冰冻三尺非一日之寒，需要我们平时不断的主动去学习积累。**

# 1.引入概念

# 引入

先来看一道题:

**如果 a+b+c=1000，且 a^2+b^2=c^2（a,b,c 为自然数），如何求出所有a、b、c可能的组合?**

# 第一次尝试

import time

start\_time = time.time()

# 注意是三重循环

for a in range(0, 1001):

for b in range(0, 1001):

for c in range(0, 1001):

if a\*\*2 + b\*\*2 == c\*\*2 and a+b+c == 1000:

print("a, b, c: %d, %d, %d" % (a, b, c))

end\_time = time.time()

print("elapsed: %f" % (end\_time - start\_time))

print("complete!")

运行结果：

a, b, c: 0, 500, 500

a, b, c: 200, 375, 425

a, b, c: 375, 200, 425

a, b, c: 500, 0, 500

elapsed: 214.583347

complete!

**注意运行的时间:214.583347秒**

# 单向链表

# 单向链表也叫单链表，是链表中最简单的一种形式，它的每个节点包含两个域，一个信息域（元素域）和一个链接域。这个链接指向链表中的下一个节点，而最后一个节点的链接域则指向一个空值。

* 表元素域elem用来存放具体的数据。
* 链接域next用来存放下一个节点的位置（python中的标识）
* 变量p指向链表的头节点（首节点）的位置，从p出发能找到表中的任意节点。

**节点实现**

class SingleNode(object):

"""单链表的结点"""

def \_\_init\_\_(self,item):

# \_item存放数据元素

self.item = item

# \_next是下一个节点的标识

self.next = None

**单链表的操作**

* is\_empty() 链表是否为空
* length() 链表长度
* travel() 遍历整个链表
* add(item) 链表头部添加元素
* append(item) 链表尾部添加元素
* insert(pos, item) 指定位置添加元素
* remove(item) 删除节点
* search(item) 查找节点是否存在

**单链表的实现**

class SingleLinkList(object):

"""单链表"""

def \_\_init\_\_(self):

self.\_head = None

def is\_empty(self):

"""判断链表是否为空"""

return self.\_head == None

def length(self):

"""链表长度"""

# cur初始时指向头节点

cur = self.\_head

count = 0

# 尾节点指向None，当未到达尾部时

while cur != None:

count += 1

# 将cur后移一个节点

cur = cur.next

return count

def travel(self):

"""遍历链表"""

cur = self.\_head

while cur != None:

print cur.item,

cur = cur.next

print ""

**头部添加元素**

def add(self, item):

"""头部添加元素"""

# 先创建一个保存item值的节点

node = SingleNode(item)

# 将新节点的链接域next指向头节点，即\_head指向的位置

node.next = self.\_head

# 将链表的头\_head指向新节点

self.\_head = node

**尾部添加元素**

def append(self, item):

"""尾部添加元素"""

node = SingleNode(item)

# 先判断链表是否为空，若是空链表，则将\_head指向新节点

if self.is\_empty():

self.\_head = node

# 若不为空，则找到尾部，将尾节点的next指向新节点

else:

cur = self.\_head

while cur.next != None:

cur = cur.next

cur.next = node

**指定位置添加元素**

def insert(self, pos, item):

"""指定位置添加元素"""

# 若指定位置pos为第一个元素之前，则执行头部插入

if pos <= 0:

self.add(item)

# 若指定位置超过链表尾部，则执行尾部插入

elif pos > (self.length()-1):

self.append(item)

# 找到指定位置

else:

node = SingleNode(item)

count = 0

# pre用来指向指定位置pos的前一个位置pos-1，初始从头节点开始移动到指定位置

pre = self.\_head

while count < (pos-1):

count += 1

pre = pre.next

# 先将新节点node的next指向插入位置的节点

node.next = pre.next

# 将插入位置的前一个节点的next指向新节点

pre.next = node

**删除节点**

def remove(self,item):

"""删除节点"""

cur = self.\_head

pre = None

while cur != None:

# 找到了指定元素

if cur.item == item:

# 如果第一个就是删除的节点

if not pre:

# 将头指针指向头节点的后一个节点

self.\_head = cur.next

else:

# 将删除位置前一个节点的next指向删除位置的后一个节点

pre.next = cur.next

break

else:

# 继续按链表后移节点

pre = cur

cur = cur.next

**查找节点是否存在**

def search(self,item):

"""链表查找节点是否存在，并返回True或者False"""

cur = self.\_head

while cur != None:

if cur.item == item:

return True

cur = cur.next

return False

**测试**

if \_\_name\_\_ == "\_\_main\_\_":

ll = SingleLinkList()

ll.add(1)

ll.add(2)

ll.append(3)

ll.insert(2, 4)

print "length:",ll.length()

ll.travel()

print ll.search(3)

print ll.search(5)

ll.remove(1)

print "length:",ll.length()

ll.travel()

**链表与顺序表的对比**

链表失去了顺序表随机读取的优点，同时链表由于增加了结点的指针域，空间开销比较大，但对存储空间的使用要相对灵活。

链表与顺序表的各种操作复杂度如下所示：

| **操作** | **链表** | **顺序表** |
| --- | --- | --- |
| 访问元素 | O(n) | O(1) |
| 在头部插入/删除 | O(1) | O(n) |
| 在尾部插入/删除 | O(n) | O(1) |
| 在中间插入/删除 | O(n) | O(n) |

注意虽然表面看起来复杂度都是 O(n)，但是链表和顺序表在插入和删除时进行的是完全不同的操作。链表的主要耗时操作是遍历查找，删除和插入操作本身的复杂度是O(1)。顺序表查找很快，主要耗时的操作是拷贝覆盖。因为除了目标元素在尾部的特殊情况，顺序表进行插入和删除时需要对操作点之后的所有元素进行前后移位操作，只能通过拷贝和覆盖的方法进行。