数据结构与算法

第1节 数据结构与算法基础

1.1 算法的引入

1.1.1 学习目标

1. 了解算法的含义
2. 了解算法的特征

1.1.2 引入(实例)

用纸和笔计算

$$x = \sqrt{10}$$

的值，精确到小数点后四位。(具体例子：计算已知面积为10的正方形之边长)

方法一思路：

1. 第一步：先找到两个数其中一个数A的平方小于10,另一个数B的平方大于10，比如找得A=3, B=4
2. 第二步：计算A和B的平均值C,并计算C的平方 (检查是否达到精度，没有达到精度，则执行第三步，否则终止程序)
3. 第三步: 比较C的平方与10的大小关系，如果大于10,将B的值用C替换；如果C的平方小于10,则将A的值用C替换。
4. 回到第二步。

方法一代码：

```
#coding:utf-8
#方法一
a = 10
x0 = 1.0
x2 = 10.0

for i in range(100):
    x1 = (x2+x0)/2
    print "第", i+1, "步："

    if x1**2 < 10:
        x0 = x1
        print "x0=",x0
        print "x0 * x0 = ", x0**2
        if (x0**2 -a)*(x0**2 -a) <0.00000001:
            break
    else:
        x2 = x1
        print "x2=",x2
        print "x2 * x2 = ", x2**2
```

```
        if (x2**2 -a)*(x2**2 -a) <0.00000001:
            break
```

方法二思路:

1. 第一步: 选择一个试探值

$$x_0 = 3$$

2. 第二步: 计算

$$x_1 = \frac{1}{2}(x_0 + \frac{10}{x_0}) = 3.16666$$

3.第三步: 求x1平方，判断是否达到所要的精度

$$3.16666^2 = 10.02774$$

4. 第四步: 重设试探值

$$x_0 = 3.16666$$

5. 第五步: 重复第二步

方法二代码：

```
#coding:utf-8
# 方法2
a = 10.0
x0 = 3

for i in range(100):
    x1 = (x0+ a/x0)/2
    print "第", i+1, "步："
    print "%f^2=%f" % (x1, x1**2)
    if (x1**2 - a)*(x1**2 - a) < 0.00000001:
        break
    x0 =x1
```

分析：

1. 按方法一计算10的平方根，需14次循环才能达到所要的精度。
2. 按方法二来计算时，需 2 次循环，达到所要的精度。

总结：

1. 解决问题的方法有多种。
2. 为达到同样的目的，不同方法所需要的步骤的数目不一样。
3. 为了减少步骤的数目，选择解决问题方法非常重要。

1.1.3 什么是算法？

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。 简单地说，算法就是解决问题的思路。它独立于编程语言。

注意：

1. 算法是解决问题的思路。
2. 算法与编程语言无关。

1.1.4 算法的特征

1. 有穷性：算法必须能在执行有限个步骤之后终止。
2. 确切性：算法的每一步骤必须有确切的定义。
3. 输入: 一个算法有0个或多个输入，以刻画运算对象的初始情况。
4. 输出: 一个算法有一个或多个输出，给出对输入数据加工后的结果。没有输出的算法毫无意义。
5. 有效性: 算法中执行的每一个步骤都是可以被分解为基本的可执行的操作步，即每个步骤都可以在有限时间内完成。

1.1.5 小结

1. 从上述例子中体会算法的重要性
2. 结合上述思路理解算法的特征

1.2 时间复杂度

1.2.0 学习目标

1. 掌握常见时间复杂度的大小关系
2. 了解基本操作数量的估算

1.2.1 如何衡量一个算法的优劣？

上面两种方法都可以计算10的平方根。为了得到相同精度，它们所需要执行的计算次数是不一样的。

问: 那一个方法更好？

不同计算机执行同一算法所用的时间可能不同，但是如果它们执行同一算法，它们执行的基本操作的数量是一样的。 所以，我们可用程序执行的基本操作的数量来衡量一个算法的优劣。

可以引入时间复杂度的概念。

1.2.2 什么是时间复杂度？(掌握)

算法的时间复杂度是指执行算法所需要的基本操作的数量.

分析： 基本操作包括：

```
算术运算：加减乘除等运算；
逻辑运算：或、且、非等运算；
关系运算：大于、小于、等于、不等于等运算；
数据传输：输入、输出、赋值等运算
```

例：计算上面两种方法计算10的平方根精确到小数点后 4 位所需要的基本操作的数量。

方法一代码：

```
#coding:utf-8
#方法一
a = 10
x0 = 1.0
x2 = 10.0

for i in range(100):
    x1 = (x2+x0)/2
    print "第", i+1, "步："

    if x1**2 < 10:
        x0 = x1
        print "x0=",x0
        print "x0 * x0 = ", x0**2
        if (x0**2 -a)*(x0**2 -a) <0.00000001:
            break
    else:
        x2 = x1
        print "x2=",x2
        print "x2 * x2 = ", x2**2
        if (x2**2 -a)*(x2**2 -a) <0.00000001:
            break
```

赋初值：3次

循环内的操作数量：7次

循环次数：15次

基本操作数量为

$$T = 7 * 15 + 3 = 108$$

方法二：

```
#coding:utf-8
# 方法2
a = 10.0
x0 = 3

for i in range(100):
    x1 = (x0+ a/x0)/2
    print "第", i+1, "步："
    print "%f^2=%f" % (x1, x1**2)
    if (x1**2 - a)*(x1**2 - a) < 0.00000001:
        break
    x0 =x1
```

赋初值：2次

循环内的操作数量：5次

循环次数：2次

基本操作数量为

$$T = 2 * 5 + 2 = 12$$

例2.如果正整数a,b,c满足条件

$$a + b + c = 1000,$$

且

$$a^2 + b^2 = c^2,$$

求a,b,c的所有可能组合。（面试题目）

我们来写出程序，计算时间复杂度。

方法一：

```
#coding:utf-8
#方法一
for a in range(1001):
    for b in range(1001):
        for c in range(1001):
            if a + b + c == 1000 and a**2 + b**2 == c**2:
                print "a,b,c:", a, b, c
```

基本操作的数量： 1000 * 1000 * 1000 *10

方法二：

```
#coding:utf-8
#方法二
for a in range(1001):
    for b in range(1001):
        c = 1000 - a - b
        if a**2 + b**2 == c**2:
            print "a,b,c:", a, b, c
```

基本操作的数量：1000 * 1000 *9

1.2.3 如何表示时间复杂度？

(1) 问题：如上例中，如果遇到的条件是

$$a + b + c = 2000,$$

或者是

$$a + b + c = 3000,$$

或者对任意的整数N,

$$a + b + c = N,$$

该如何计算出基本操作的数量？

```
方法一基本操作数量:

2000: 2000*2000*2000*10
3000: 3000*3000*3000*10
任意正整数N:   N*N*N*10 (N的三次函数)

方法二基本操作数量:

2000: 2000*2000*9
3000: 3000*3000*9
任意正整数N:  N*N*9  (N的二次函数)
```

所以，我们可以把基本操作数量写成一个待解决问题的规模N的一个函数：

$$T = T(N)$$

这就是时间复杂度。

(2)可以计算出算法一和算法二的时间复杂度：

算法一:

$$T = N^3 * 10$$

算法二:

$$T = N^2 * 9$$

(3)数量级的概念

微毫厘分十百千兆吉太

(4) 大O记法

实践中关心的是时间复杂度的数量级. 例2中,方法一的时间复杂度可以记为:

$$T = N^3 * 10 \quad => \quad T = N^3 \quad => \quad T = O(N^3)$$

例2中,方法二的时间复杂度可以记为：

$$T = N^2 * 9 \quad => \quad T = N^2 \quad => \quad T = O(N^2)$$

1.2.4 为什么需要时间复杂度？

```
1. 时间复杂度可用来判断程序执行的效率。
2. 用时间复杂度可衡量算法的优劣，从而有助于我们写出优质代码。
```

1.2.5 常见时间复杂度和大小关系

```
常见的时间复杂度
    常数阶: O(1)
```

```
        T=15
    线性阶: O(N)
        T= 2N +10
    平方阶: O(N^2)
        T= 3N*N + 2N
    对数阶: O(logN)
        T= 2logN + 100
    NlogN阶 : O(NlogN)
        T= NlogN +20
    立方阶: O(N^3)
        T= 6N*N*N + 1
    指数阶: O(2^N)
        T= 2^N +N
常见时间复杂度的大小关系 (掌握)
    O(1) < O(logN) < O(N) < O(NlogN) < O(N^2) < O(N^3) < O(2^N) < O(N!) < O(N^N)
```

1.2.6 小结

```
1. 理解时间复杂度的含义
2. 了解如何评价一个算法的优劣
3. 掌握常见时间复杂度的大小关系
```

1.3 数据结构的引入

1.3.0 学习目标

1. 掌握数据结构的概念
2. 了解数据结构与算法的时间复杂度之间存在关系

1.3.1 引入

用Python保存一个班级学生的信息，包括学生的姓名，年龄，出生地.

1.3.1.1 以什么类型保存信息？

```
#列表嵌套元组
>>>li = [("Zhang",22,"Chengdu"),("Li",20,"Chongqing"),("Qian",20,"Nanjing")]
#列表嵌套字典
>>>li1 = [{"name":"Zhang","age": 22, "hometown":"Chengdu"}, {"name":"Li","age": 20,
"hometown":"Chongqing"},{"name":"Qian","age": 20, "hometown":"Nanjing"}]
#字典嵌套字典
>>>li2 = {"Zhang":{"age":22,"hometown":"Chendu"},"Li":
{"age":22,"hometown":"Chendu"},"Qian":{"age":20,"hometown":"Chendu"}}
```

1.3.1.2 如何查找某学生的信息

计算时间复杂度，比较算法效率

列表嵌套元组：

```
>>> li = [("Zhang",22,"Chengdu"),("Li",20,"Chongqing"),("Qian",20,"Nanjing")]
>>> li[0][0]
'Zhang'
>>> li[2][0]
'Qian'
>>>
```

时间复杂度为

$$T = 3 * 3 = 9$$

列表嵌套字典：

```
>>> li1 = [{"name":"Zhang","age": 22, "hometown":"Chengdu"}, {"name":"Li","age": 20,
"hometown":"Chongqing"},{"name":"Qian","age": 20, "hometown":"Nanjing"}]
>>> li1[0]["name"]
'Zhang'
>>> li1[2]["age"]
20
>>>
```

时间复杂度为

$$T = 3 * 1 = 3$$

字典嵌套字典：

```
>>> li2 = {"Zhang":{"age":22,"hometown":"Chendu"},"Li":
{"age":22,"hometown":"Chendu"},"Qian":{"age":20,"hometown":"Nanjing"}}
>>> li2["Zhang"]["age"]
22
>>> li2["Li"]["hometown"]
'Chendu'
>>>
```

时间复杂度为

$$T = 1 * 1 = 1$$

总结： 算法的时间复杂度与用什么类型保存数据有密切关系。

1.3.2 什么是数据结构？

```
数据结构确定一组数据如何保存.例如上例子中，列表和字典都分别是一种数据结构。

数据结构是对基本数据类型的封装。
基本数据类型有: int, float, str等

数据结构不同，导致算法的时间复杂度不同.
Python数据结构举例：列表，元组，字典，集合.
```

1.3.3 列表中的操作的时间复杂度

现在我们看对于列表的两个不同操作的时间复杂度.

1.3.3.1 构造新列表的三个操作,向列表尾添加元素，列表相加,和插入元素

```
li.append()
li + li2
li.insert()
```

哪一个的效率更高？代码如下：

```
#coding:utf-8
import timeit
from timeit import Timer

def t1():
    li = []
    for i in range(1000):
        li.append(i)

def t2():
    li =[]
    for i in range(1000):
        li = li + [i]
def t3():
    li = []
    for i in range(1000):
        li.insert(0,i)

timer1 =Timer("t1()", "from __main__ import t1")
print (timer1.timeit(100))

timer2 =Timer("t2()", "from __main__ import t2")
print (timer2.timeit(100))

timer3 =Timer("t3()", "from __main__ import t3")
print (timer3.timeit(100))
```

运行结果：

```
0.0151388645
0.2677149772
1.0190680027
```

1.3.3.2 为什么列表上不同的操作的效率差别巨大？

这是由列表的存储方式决定的.

1.3.3.3 列表常见内置操作的时间复杂度

```
index   O(1)  eg: li = [1,2,3];  li[2]
append  O(1)
insert(i)   O(N), N为列表长度
pop()   O(1)
pop(i)  O(N)
contains    O(N)   eg: li=[1,2,3]; 2 in li
iteration    O(N)
get slice [x:y]   O(k),k=y-x
set slice   O(N+k)
del slice   O(N)
sort()      O(NlogN)
reverse     O(N)
```

1.3.3.4 字典常见内置操作的时间复杂度

```
index   O(1)

contains    O(1)  例: dict ={"name":"Zhang","age":23}; "age" in dict
iteration   O(N)
get item   O(1)
set item   O(1)
del item   O(1)
```

总结：

```
1. 列表和字典的内置方法是对基本操作步骤的封装,而非基本操作步骤
2. 列表(字典)的方法各有不同的操作步骤数量，因此具有不同的时间复杂度
3. 不同的方法的效率有差别
4. 列表和字典不属于基本数据类型
```

算法与数据结构有什么关系？

```
算法关注解决问题的思路
数据结构关注待解决问题中的数据该如何保存.
程序=算法 + 数据结构
```