## 1.3 递归简论

当一个函数用它自己定义时，就称为时递归(recursive)的

基准情况(base case)，即此时函数的值可以直接算出而不用求助递归。

Java的递归方法若无基准情况也是毫无意义的。

虽然我们定义一个方法用的是这个方法本身，但是我们并没有用方法本身定义该方法的一个特定实例。

所以，递归并不是循环推理(circular logic)

递归调用将反复进行直到基准情况出现

递归有两个基本法则：

1、基准情况 base case必须总要有些基准的情形，他们不用递归就能求解

2、不断推进 making progress 对于那些要递归求解的情形，递归调用必须总能够朝着一个基准情形推进

3、设计法则，假设所有的递归调用都能运行

4、合成效益法则，在求解一个问题的同一个实例时，切勿在不同的递归调用中作重复性的工作

## 1.4 实现泛型特性构建

1.4.1 使用Object表示泛型

1.4.2 基本类型的包装

1.4.3 使用接口类型表示泛型

1.4.4 数组类型的兼容性

## 1.5 利用Java5泛性实现反省特性成分

1.5.1 简单的泛性类和接口

1.5.2 自动装箱/拆箱

1.5.3 带有限制的通配符

# 算法分析

算法（algotithm）是为求解一个问题所需要遵循的、被清楚指定的简单指定的集合。

这一章，我们将讨论

* 如何估计一个程序所需要的时间
* 如何将一个程序的运行时间从天或年降低到秒甚至更少
* 粗心使用递归的结果
* 将一个数自乘得到其幂，以及计算两个数的最大公因数的非常有效的算法

## 2.1 数学基础

定义2.1 如果存在正常数和使得当N≥时，T(N)≤cf(N),则记为T(N)=O(f(n))。

定义2.2 如果存在正常数c和n0，使得当N≥cg(N)时，T(N)≥cg(N)，则记为T(N) =Ω(g(N))。

定义2.3 T(N) = Ɵ（h（N））当且仅当T(N)=O(h(N))和T(N)=Ω(h(N))。

定义2.4 如果对每一个正常数c，都存在常数n0，使得当N>n0时，T(N)<cp(N),则T(N) = o(p(N))。

有时也可以说，如果T(N)=O(p(N))且T(N)≠Ɵ（p（N））,则T(N) = o(p(N))

这些定义的目的是要在函数间建立一种相对的级别。给定两个函数，通常存在一些点，在这些点上一个函数的值小于另一个函数的值，因此一般宣城，比如f(N)<g(N)，时没有什么意义的。于是我们比较他们的相对增长率(relative rate of growth)。当将相对增长率从应用到算法分析时，我们将会明白为什么它是重要的度量。

如果用传统的不等式来计算增长率，那么

第一个定义是说T(N)的增长率小于或等于f(N)的增长率，

第二个定义T（N）=Ω（g（N））是说T（N）的增长率大于或等于g（N）的增长率。

第三个定义T（N）= Ɵ（h（N））说的是T(N)的增长率等于h(N)的增长率。

最后一个定义T（N）=o（p（N））说的则是T（N）的增长率小于p（N）的增长率。

法则1

如果T1(N) = O(f(N))且T2(N)=O(g(N)),那么

1. T1(N) + T2(N)=O(f(N)+g(N))(直观地和非正式地可以写成max(O(f(N),O(g(N))))
2. T1(N) \* T2(N)=O(f(N)\*g(N))

法则2

如果T(N)是一个k次多项式,则T(N)=Ɵ(Nk).

法则 3

对于任意常数k,logkN=O(N),他告诉我们,对数增长得非常缓慢

有几点要注意:

首先,将常数或者低阶项放进O是非常坏的习惯,不要写成T(N)=O(2N2)或者T(N) = O(N2 + N).

在大O分析中,各种简化都是可能发生的.低阶项一般可以忽略,而常数也可以弃掉.此时要求的精度是非常粗糙的.

第二,我们总能够通过计算极限来确认两个函数的的相对增长率,必要的时候可以使用洛必达法则.该极限可能有四种值:

* 极限是0 这意味着f(n) = o(g(n))
* 极限是c ≠ 0,这意味着f(N) = Ɵ(g(N))
* 极限是∞ 这意味着f(n) = Ω(g(N)),即g(N) = o(f(N))
* 极限摆动,二者无关

|  |  |
| --- | --- |
| 函数 | 名称 |
|  | 常数 |
|  | 对数 |
| log2N | 对数平方 |
| N | 线性的 |
| N |  |
| N2 | 二次的 |
| N3 | 三次的 |
| 2N | 指数的 |

典型的增长率

## 2.2 模型

为了在正式的架构中分析算法,我们需要一个计算模型.

我们的模型基本上是一台标准的计算机,在机器中被顺序地执行.

我们假设模型做任何一件简单的工作都恰好花费一个时间单位

模型机有无限的内存

## 2.3 要分析的问题

主要因素:

1使用的算法

2对该算法的输入

典型的情况是,输入的大小是主要的考虑方面

我们定义两个函数Tavg(N)和Tworst(N),分别为算法对于输入量N所花费的平均时间和运行时间.显然,Tavg≤Tworst(N).

平均情形常常反应典型的行为

最坏情形的性能则代表对任何可能输入的性能的一种保证

程序是算法以一种特殊编程语言的实现,程序设计语言的细节几乎总是不影响大O的答案.

## 2.4 运行时间的计算

为了简化分析,我们将采纳一下的约定:

不存在特定的时间单位

因此,我们抛弃一些前导的常数

并抛弃低阶项

从而要做的就是计算大O的运行时间.

### 2.4.1 一个简单的例子

计算的一个简单程序片段

**public static int** sum(**int** n){  
 **int** partialSum = 0;  
 **for** (**int** i = 0;i <= n;i++){  
 partialSum += i^3;  
 }  
 **return** partialSum;  
}

### 2.4.2 一般法则

* **法则1 for循环**

1个for循环的运行时间至多是该循环内部那些语句的运行时间乘以迭代次数

* **法则2 嵌套的for循环**

从里向外分析这些循环.在一组嵌套循环内部的一句语句总的运行时间为该语句的运行时间诚意改组所有的for循环的大小的乘积

例如:

**public static int** TestON2(**int** n) {  
 **int** k = 0;  
 **for** (**int** i = 0; i < n; i++) {  
 **for** (**int** j = 0; j < n; j++) {  
 k++;  
 }  
 }  
 **return** n;  
}

* **法则3 顺序语句**

将各个语句的运行时间求和即可,这意味着,其中的最大值就是所得的运行时间

**public static int**[] testOrder(**int** n) {  
 **int**[] a = **new int**[n];  
 **for** (**int** i = 0; i < n; i++ ){  
 a[i] = 0;  
 }  
 **for** (**int** i = 0; i< n ;i++) {  
 **for** (**int** j =0;j<n;j++) {  
 a[i] += a[j] +i +j;  
 }  
 }  
 **return** a;  
}

上面的程序片段先是花费O(N),然后是O(N2),因此总量也是O(N2)

* **法则4 if/else语句**

if(condition){

$1

}else{

$2;

}