# 引论

## 1．数学知识、公式

1. 级数







 k≠-1

 (叫做调和数，和为调和和。误差趋近于0.57721566，

这个数被称为欧拉数。）

1. 模运算

如果正整数A与B对N取模，余数相同，记为A≡B(mod N)。  
若A≡B(mod N)，A+C≡B+C(mod N)，并且AD≡BD(mod N)。

1. 证明方法



1. 递归

四条基本法则：

1. 基准情形。定义好不需要递归就可以求出的基准；
2. 不断推进。后续每一次调用需使求解情况朝接近基准情形的方向推进，像查英译

汉的字典一样，查一个我们不会的词的时候又在解释中碰到不认识的词，就要继续查下去，但最后，肯定有一个我们不需要查就认识的单词，此时倒退过来就能明白最初查的词的意思

1. 设计法则。假设所有的递归调用均能运行；
2. 合成效益法则。因为系统每次递归是要通过开一个固定大小的不相同的内存空

间的，所以尽力保证在求解同一问题的同一实例时，切勿在不同的递归调用钟做重复性的工作。

# 算法分析

### 时间复杂度

（1）基本概念

时间复杂度主要用来统计算法执行语句的次数，通过计算时间复杂度，可以判断一个算

法执行的相对时间长短。

一般情况下，算法中基本操作重复执行的次数是问题规模n的某个函数，用T(N)表示，若有某个辅助函数f(N),存在一个正常数c使得f(N)\*c>=T(N)恒成立。记作T(N)=O(f(N)),称O(f(N)) 为算法的渐进时间复杂度，简称时间复杂度。

注：

1.有两个时间复杂度函数，Tworst(N)和Tavg(N),分别是输入为N时，算法的最坏时间复杂度（主要使用）与平均时间复杂度。

2. 时间复杂度是表示一个函数的趋势,并不代表具体值,当n趋于无穷大时,可以忽略低阶项和首项系数

（2）计算

a.基本约定：不存在特定的时间单位，假设计算机做任何一件简单工作即花费一个时间单元(声明不计时间)。

b.一般规则

* 法则1：for循环

对于一个循环，假设循环体的时间复杂度为 O(n)，循环次数为 m，则这个循环的时间复杂度为 O(n×m)。

* 法则2：嵌套for循环

对于多个循环，假设循环体的时间复杂度为 O(n)，各个循环的循环次数分别是a, b, c……，则这个循环的时间复杂度为 O(n×a×b×c……)。分析的时候应该由里向外分析这些循环。

* 法则3：顺序语句

对于顺序执行的语句或者算法，总的时间复杂度等于其中最大的时间复杂度。

* 法则4：IF/ELSE语句

对于条件判断语句，总的时间复杂度等于其中 时间复杂度最大的路径 的时间复杂度。

总结：

时间复杂度分析的基本策略是：从内（或最深层次）向外分析。如果有函数调用，那么这些调用要首先分析。递归也需要仔细分析。

c.运行时间中的对数

如果一个算法用常数时间（O(1)）将问题的大小削减为其一部分（通常为1/2），那么该算法就是O(logN)。另一方面，如果使用常数时间只是把时间减少了一个常数，则为O(N)

注：二分法（logN）,分治法（NlogN）。

### 最大子序列和算法

O(N)复杂度的联机算法：

只对数据进行一次扫描，一旦a[i]被读入并处理，它就不再需要被记忆。因此数组可以被按顺序读入，在主存中不必存储数组的任何部分。具有这种特性的算法叫联机算法(on-line algorithm)，仅需要常量空间并以线性时间运行的联机算法几乎是完美算法！

/\*该算法适用于序列中至少有一个数非负的场合，时间复杂度为线性的O(N)\*/

int MaxSubSequenceSum(int A[], int N)

{

int MaxSum = 0, ThisSum = 0; //ThisSum为当前子序列和

for (int i = 0; i < N; i++)

{

ThisSum += A[i]; //逐个读取，逐个加

if (ThisSum > MaxSum)

{

MaxSum = ThisSum; //当前子序列和如果比最大子序列和大，更新

}

else if (ThisSum < 0)

{

ThisSum = 0; //当前子序列和小于0，即舍弃该子序列，重新开始新的子序列

}

}

return MaxSum;

}