1.4 算法与算法分析

#数据结构

算法

定义 对特定问题求解方法和步骤的一种描述，他是指令的有限序列，其中每个指令表示一个或者多个动作

简而言之：算法就是解决问题的方法和步骤

算法的描述

- 自然语言： 中文 英语
- 流程图： 传统流程图 NS流程图
- 伪代码： 类语言：类C语言
- 程序代码： C语言 Java语言

算法与程序的关系

- 算法是解决问题的一种方法或者一个过程，考虑如何将输入转化为输出
- 一个问题可以有多种算法
- 算法可以扩展到任意一个完整的程序设计语言上
- 也就是说算法不拘泥于特定的程序语言

程序是用某种程序设计语言对算法对具体实现

- 程序 = 数据结构 + 算法
- 数据结构通过算法实现操作
- 算法根据数据结构设计程序

算法的特性

- 有穷性： 一个算法必须总是在执行有穷步后实现，且每一步都在有穷时间内完全。
- 确定性： 算法中的每一条指令都必须有确切的含义，没有二义性，在任何条件下，只有唯一的一条执行路线，即对于同样的输入只能得到相同的输出
- 可行性： 算法是可执行的
- 输入： 有零个或者多个输入
- 输出： 一个算法必须要有一个或者多个输出

算法的设计要求

- 正确性

> 怎么样判定为正确：
> 程序中不包含语法错误
> 程序对于几组输入数据能得出满足要求的结果
> 程序对于精心选择的典型的苛刻且带有刁难性质的几组输入数据能够得出满足要求的结果
> 程序对于一切合法的输入数据都能得出满足要求的结果
> 通常以第三层意义上的正确性作为衡量一个算法是否合格的标准

- 可读性

> 算法主要是为了人的阅读和交流，其次才是为了计算机执行，因此算法应该易于人类理解
> 晦涩难懂的算法容易隐藏较多错误而难以调试

- 健壮性(鲁棒性)

> 指当输入非法数据时，算法恰当做出反应，或者进行相应处理，而不是产生莫名其妙的输出结果
> 处理出错的方法，并不是中断程序运行，而时候应该返回一个表示错误的值，以便在更高的抽象层次进行处理

- 高效性

> 花费尽可能少的时间和空间

如何评价算法

算法分析

首先一个好的算法必须具备健壮性，正确性，可读性，在几个方面都满足的情况下，主要考虑算法的效率，通过算法的效率高低来判断不同算法的优劣程度

算法效率：

- 时间效率： 算法所耗费的时间
- 空间效率：所耗费的空间

注意在有些时候，时间效率和空间效率是矛盾的（即鱼和熊掌不能兼得）

算法时间效率的度量：

算法时间效率可以用依据该算法编制的程序在计算机执行所消耗的时间来度量

一般有两种方法：

1. 事后统计：

- 将算法实现，测量其时间和空间开销
- 缺点： 编写程序实现算法将花费较多的时间和精力，所得到的实验成果依赖于计算机的软硬件等环境因素，掩盖算法本省的优劣

2. 事前分析 （更多的选用）
- 对算法所消耗的资源的一种估算方法
- 事前分析方法：一个算法的运行时间是指一个算法在计算机上运行所耗费的时间大致可以等于计算机执行一种简单的操作（比如赋值，比较，移动）所需要的时间与算法中进行的简单操作次数累计

怎么估算呢？
- 一个算法的运行时间是指一个算法在计算机上运行。
- 公式： 算法运行时间 = 一个简单操作所需要的时间 * 简单操作的次数
- 也就是算法中每条语句的执行时间为之和
- 算法的运行时间 = ∑ 每条语句的执行次数 * 该语句执行一次所需要的时间
- 其中每条语句的执行次数又可以称作语句频度
- 那么公式又可以写为： ∑ 每条语句频度* 该语句执行一次所需要的时间

其中该语句执行一次所需要的时间取决于机器的指令性能，速度和编译的代码质量，是有机器本身的软硬件实现的，他与算法无关！所以我们可以假设每条语句所需的时间均为单位时间，此时对算法的运行时间的讨论就可以转换为讨论该语句中所有语句的执行次数，即频度之和
（就是不考虑每条语句的所执行一次的所需要的时间）

例子：两个 N * N 矩阵相乘的算法可以描述为：

```
for (int i = 1; i <= n; ++i) { //N 次
    for (int j = 1; j <= n; ++j) { // N^2 次
        c[i][j] = 0;                  // N^2 次
        for (int k = 0; k < n; k++) { // N^3 次数
            c[i][j] += a[i][k] * b[k][j]; // N^3 次数
        }
    }
}
```

我们把算法所耗费的时间定义为该算法中每条语句的频度之和。

所以则上述算法所耗费的时间为
将所有语句的执行次数加起来
就是 = 2**N^3 + 2**N^2 + N。
（这是一个关于 N的 函数）

但是这样的算法太麻烦了。

算法时间复杂度的渐进表示法

为了方便比较不同的算法的时间效率，我们仅仅只比较他们的数量集

例如两个不同的算法，时间消耗分别为：
T(n) = 10* n^2
T(n) = 5 * n^3

所以我们只比较算法的数量级

如果有某个辅助函数 F(n),使得当n趋向于无穷大时，t(n)/f(n)的极限值为不等于0的常数，则称f(n)是t(n)的同数量级函数，记作Tn = O(f(n)),称 O(f(n))为算法的渐进时间复杂度（O是数量级符号，order） 简称时间复杂度

笔记作者注（仅供参考）：此处是否可以类比为高等数学中的等价无穷小的概念，变成等价无穷大（？？？）同样也是使用 O符号来表示数量集合，如果比出来是一个C（C不等于0）就是同阶级无穷大

回到矩阵相乘的函数
2**N^3 + 2**N^2 + N
T(n)/n^3 (n趋向于无穷大) 趋向于2 是一个不为0的常数，所以和 n^3 次方是同一个数量级
所以T(n) 可以记作 O(N^3)
这就是我们求解矩阵相乘问题的渐进时间复杂度

由这个算法就可以被称抓大头，就是只要算出算法中频度最高的阶数就是整个算法的渐进时间复杂度

一般情况下，不必计算所有操作的执行次数，而只考虑算法中的基本操作执行的次数，他是问题规模N的某个函数，用T(n)来表示

那么公式可以总结为： 算法中基本语句重复执行的次数是问题规模n的某个函数，算

法的时间量度记作： 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

他表示随着n的增大，算法执行的时间的增长率和f(n)的增长率相同，称为渐进时间复杂度

什么是基本语句：

算法中重复执行次数和算法执行时间成正比的语句

对算法运行时间影响最大的

执行次数最多的

计算时间复杂度的一些简化定理：

1. （抓大头） f(n) = a_m * n ^m + ...(都比n^m阶数小) 就是 T(n) = O(n^m)
   忽略所有低次项幂和最高次项幂的系数，体现出增长率的含义

计算时间复杂度的一些基本办法

1. 找出语句频度最大的那条语句作为一个基本语句
2. 计算基本语句的频度得到问题规模n的某个函数 F(n)
3. 取其数量级用O表示

```
void exam(float x[][1000], int m,int n){
    float sum[100];
    for (int i = 0; i < m; ++i) {
        sum[i] = 0.0;
        for (int j = 0; j < n; ++j) {
            sum[i] += x[i][j]; // 这个是层数最深的语句 两层For循环  N*(N + 1)
        }
    }

    for (int i = 0; i < m; ++i) {
        cout << i << ":" << sum[i] << endl;
    }
}
```

这个程序的时间复杂度是 O(n^2)

例题：分析一下程序段段时间复杂度

```
i = 1;
while(i < n){
    i*=2;
}
```

1. 首先是 i*=2 执行次数最多
2. 执行次数的函数： log_2 n次

- 分析过程 如果执行了一次 i = 2^0 * 2 = 2^1
- 执行了两次 i = 2^1 * 2 = 2^2
- 执行了三次 i = 2^2 * 2 = 2^3
- 执行了X次 i = 2^x-1 * 2 = 2^x
- 设该语句执行次数为X次，有循环条件
- i <= n ; ===> 2^x <= n ===> x <= log2 n

3. 即F(n) <= log2 n 取最大值 F(n) = log 2 n
   所以该程序的时间复杂度为：T(n) = O(log N)

请注意有的时候基本操作的执行测试还跟问题的输入集(dataset)有关

随着数据集不同而不同

1. 例如用顺序查找来查找等于e 的元素，返回其所在的位置
   最好情况：一次，最坏情况：N次
   平均时间复杂度为 O(n)

   最坏时间复杂度：在最坏的情况下
   平均时间复杂度：致所有可能的实例等概论出现的情况下
   最好时间复杂度：在最好的情况下

通常我们会考虑最坏和平均情况，有时候平均很难计算，就会只考虑最坏，因为算法运行时间不会比最坏的时间更长

有些时候，数量级很难求，有时候也可以分别来计算函数的数量级，然后利用大O的加法规则和乘法规则（详见高等数学高阶无穷小的O的计算，一摸一样）

算法时间效率的比较

- 当N取的很大时，指数时间算法和多项式时间算法在所需时间上非常悬殊
  就是指数时间算法所需要的时间 >> 多项式时间算法

渐进空间复杂度

空间复杂度：算法所需存储空间的度量
记作： S(n) = O(f(n))
其中 n 为问题的规模

算法占据的空间：
算法本身需要占据的空间，输入/输出，指令，常数，变量等等
算法所要使用的辅助空间

例子：将一个一位数组a中的n个数逆序存放在原数组中

有两种算法：
1. 使用一个辅助变量t 需要一个辅助空间 S(n) = O(1)
   他是原地工作
2. 使用一个辅助数组，倒序输入 需要n个辅助空间 S(n) = O(n)
   需要一个辅助数组b 问题规模有多大，数组b就要有多大

设计好算法的过程：

抽象数据类型 = 数据的逻辑结构 + 运算功能 （运算的功能描述）