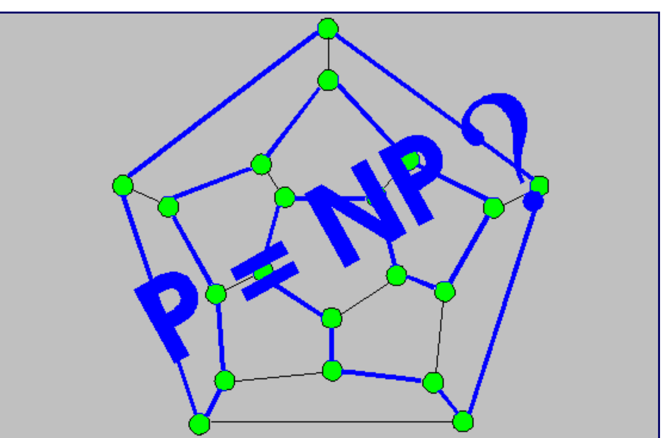
2-算法效率分析基础

陆伟

算法设计与分析

Introduction to the Design and Analysis of Algorithms



September 18, 2022

Lecture Overview

1. 算法效率度量
2. 函数渐进的界
3. 算法复杂性分析基本方法
4. 递归与非递归算法比较
5. 经验与实验分析方法

算法效率的度量

- 算法效率的高低体现在运行该算法所需要耗费资源的多少，对于计算机来讲，最重要的资源是时间和空间，因此，算法效率又可分为**时间效率**和**空间效率**。
- 分别用N，I和A表示要解决问题的规模、算法的输入和算法本身，用C表示复杂性，那么，应该有$C = F(N, I, A)$。如果吧时间复杂性与空间复杂性分开，分别用T和S表示，则$T = F(N, I, A)$，$S = F(N, I, A)$。
- $T = T(N, I)$，$S = S(N, I)$。

算法效率的度量

- 计算机存储容量的发展使得算法空间复杂性已经不再是关注的重点，但时间复杂性仍然十分重要。因此，我们后续也将主要讨论算法的时间复杂性，但是所讨论的方法对于空间复杂性分析也是适用的。
- 根据$T = T(N, I)$的概念，它应该是算法在一台“抽象的计算机”上运行所需要的时间。

为何要抽象？讨论

算法效率的度量

- 设该“抽象的计算机”所提供的元运算有k种，分别记为$O_1,O_2,\ldots,O_k$，又设每执行一次这些元运算所耗费的时间分别为$t_1,t_2,\ldots,t_k$。对于给定算法A，统计其执行过程中用到的元运算O_i的次数，记为e_i，$i=1,2,\ldots,k$。$e_i = e_i(N, I)$。

$$T(N,I)=\sum_{i=1}^{k} t_i e_i(N,I)$$

- 其中，t_i是与N和I无关的常数。

e_i为N和I的函数，I如何表示和确定？

算法效率的度量

- 我们不可能对规模为*N*的每一种合法输入*I*都去统计$e_i(N, I)$，i=1,2,…,k。

$$T_{\max}(N) = \max_{I \in D_N} T(N,I) = \max_{I \in D_N} \sum_{i=1}^{k} t_i e_i(N,I) = \sum_{i=1}^{k} t_i e_i(N,I^*)$$

$$T_{\min}(N) = \min_{I \in D_N} T(N,I) = \min_{I \in D_N} \sum_{i=1}^{k} t_i e_i(N,I) = \sum_{i=1}^{k} t_i e_i(N,\tilde{I})$$

$$T_{\text{avg}}(N) = \sum_{I \in D_N} P(I) T(N,I) = \sum_{I \in D_N} P(I) \sum_{i=1}^{k} t_i e_i(N,I) = \sum_{i=1}^{k} t_i e_i(N,I)$$

- 关于摊销效率

算法效率最终抽象为问题规模的函数，针对同一问题的不同算法，如何进行效率比较？

函数的渐进的界

- 函数的渐进的界
  - 设f和g是定义域为自然数集N上的函数

(1) $f(n)=O(g(n))$

若存在正数c和n_0使得对一切$n \geq n_0$有$0 \leq f(n) \leq cg(n)$

(2) $f(n)=\Omega(g(n))$

若存在正数c和n_0使得对一切$n \geq n_0$有$0 \leq cg(n) \leq f(n)$

(3) $f(n)=o(g(n))$

对任意正数c存在n_0使得对一切$n \geq n_0$有$0 \leq f(n) < cg(n)$

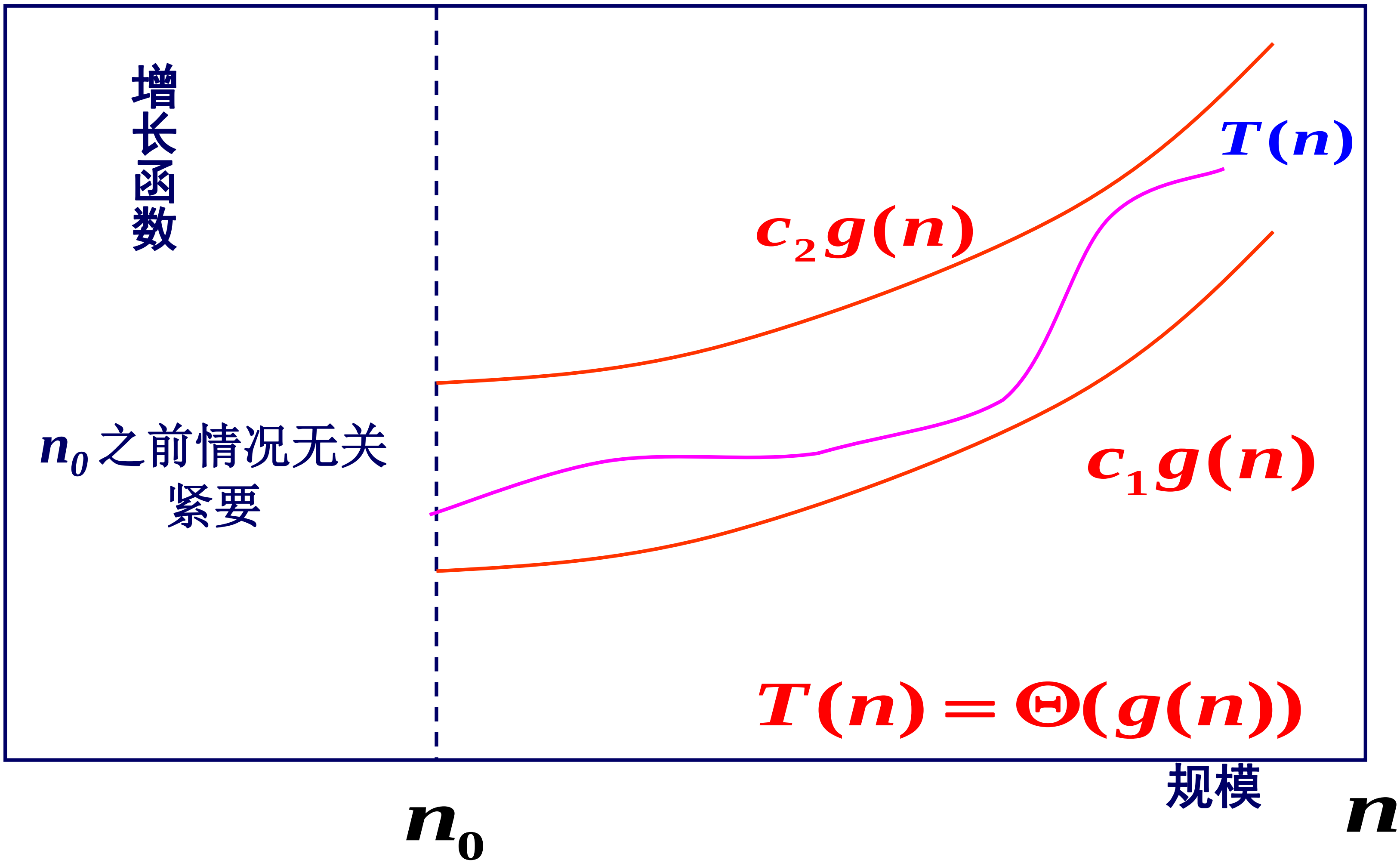
(4) $f(n)=\omega(g(n))$

对任意正数c存在n_0使得对一切$n \geq n_0$有$0 \leq cg(n) < f(n)$

(5) $f(n)=\Theta(g(n)) \Leftrightarrow f(n)=O(g(n))$ 且$f(n)=\Omega(g(n))$

(6) $O(1)$表示常数函数

函数渐进的界



函数的渐进的界

- 函数渐进的界的基本性质（1）
  - 设f和g是定义域为自然数集N上的函数:

（1）若 $\lim_{n\to\infty}\frac{f(n)}{g(n)}=c$ ，c为大于0的常数，那么

$f(n)=\Theta(g(n))$

（2）若 $\lim_{n\to\infty}\frac{f(n)}{g(n)}=0$ ，那么

$f(n)=o(g(n))$

（3）若 $\lim_{n\to\infty}\frac{f(n)}{g(n)}=\infty$ ，那么

$f(n)=\omega(g(n))$

函数的渐进的界

- 函数渐进的界的基本性质（2）
  - 设f , g, h 是定义域为自然数集N上的函数:

（1）如果$f=O(g)$且$g=O(h)$，那么$f=O(h)$.

（2）如果 $f=\Omega(g)$且$g=\Omega(h)$，那么$f=\Omega(h)$.

（3）如果$f=\Theta(g)$和$g=\Theta(h)$，那么$f=\Theta(h)$.

(4) $O(f(n))+O(g(n)) = O(\max\{f(\mathrm{n}),g(n)\})$

(5) $O(f(n))+O(g(n)) = O(f(\mathrm{n})+g(n))$

(6) $O(f(n))*O(g(n)) = O(f(\mathrm{n})*g(n))$

函数的渐进的界

- 函数渐进的界的基本性质（3）
  - 设f , g, h 是定义域为自然数集N上的函数，若对某个其它的函数h, 我们有f =$O(h)$和g=$O(h)$，那么

$$f + g = O(h).$$

  - 假设f 和g是定义域为自然数集合的函数，且满足g=$O(f)$，那么

$$f + g = \Theta(f).$$

函数的渐进的界

- 例:
- 多项式函数$f(n)=a_0+a_1n+a_2n^2+\ldots+a_dn^d$，其中$a_d\neq 0$，证明$f(n)=\Theta(n^d)$。
- 证明 $\log_2 n = o(\sqrt{n})$ 。
- 证明$\log_a n=\Theta(\log_b n)$。
- 对于$b>1$和$\alpha>0$，$\log_b n=o(n^\alpha)$，$n^\alpha=o(b^n)$。
- $n!=o(n^n)$，$n!= \omega(2^n)$，$\log(n!)= \Theta(n\log n)$

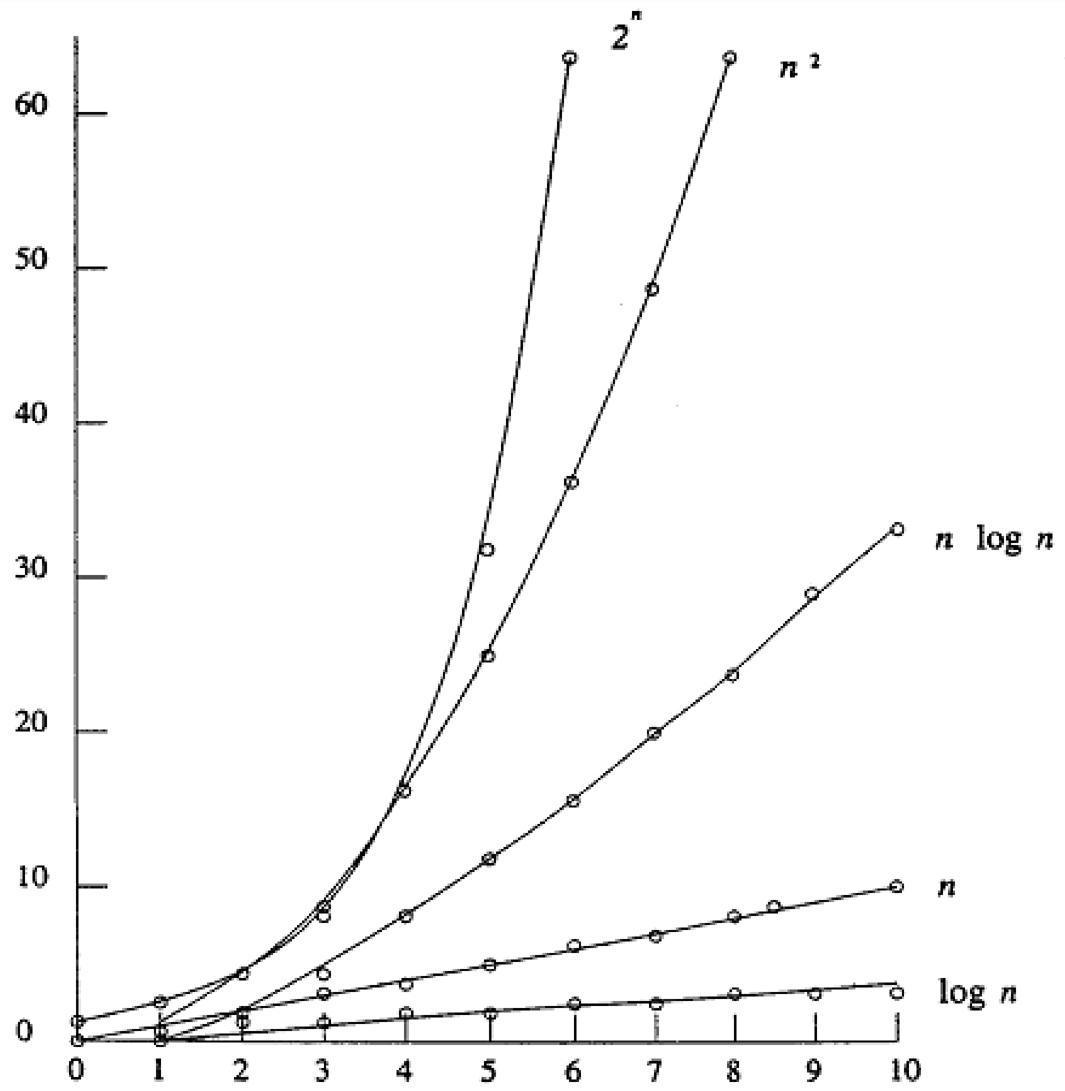
函数渐进的界

算法基本渐进效率类型

类 型	名 称	注 释
1	常量	为数极少的效率最高的算法，难以举出几个例子。通常，当输入规模变得无穷大的时候，算法的执行时间也会趋于无穷大。
logn	对数	算法每次循环都会消去问题规模的一个常数因子。
n	线性	扫描规模为n的列表（如顺序查找）算法。
nlogn	n-logn-n	诸多分治算法，包括合并排序和快速排序平均效率属于此类型。
n^2	二次	一般来说，包含两重嵌套循环的算法。n 阶方阵的某些特定算法。
n^3	三次	一般来说，包含三重嵌套循环的算法。
2^n	指数	求n个元素集合的所有子集的算法。“指数”这个术语常用在更广泛的层面上，不仅包括这种类型，还包括那些增长速度更快的算法如阶乘。
n!	阶乘	求n个元素集合的完全排列的算法。

函数渐进的界

常见渐进效率类型比较



算法复杂性分析基本方法

- 算法复杂性分析基本步骤

（1）确定表示输入规模的参数。

（2）找出算法的基本操作。

（3）检查基本操作的执行次数是否只依赖于输入规模。这决定是否需要考虑最差、平均以及最优情况下的复杂性。

（4）对于非递归算法，建立算法基本操作执行次数的求和表达式；对于递归算法，建立算法基本操作执行次数的递推关系及其初始条件。。

（5）利用求和公式和法则建立一个操作次数的闭合公式，或者求解递推关系式，确定增长的阶。

算法复杂性分析基本方法

- 非递归算法复杂性分析常见求和公式
  - 等差数列 $\sum_{k=1}^{n} a_k$
  - 等比数列 $\sum_{k=0}^{n} aq^k$
  - 调和级数 $\sum_{k=1}^{n} \frac{1}{k}$
  - 对数求和 $\sum_{i=1}^{n} \log i$

算法复杂性分析基本方法

例

- 非递归算法复杂性分析

算法 maxElement(A[0..*n*-1]
//求给定数组中的最大元素
//输入：实数数组A[0..*n*-1]
//输出：A中的最大元素
maxval ←A[0]
for *i* ← 1 to *n*-1 do
    if A[*i*] > *maxval*
        maxval ← A[*i*]
return *maxval*

算法复杂性分析基本方法

例

- 非递归算法复杂性分析

（1）算法输入规模：可以用数组元素个数n度量

（2）基本操作：比较与赋值两种，选择比较

（3）比较操作只与输入规模相关，不用考虑最坏、平均、最好情况

（4）建立基本操作执行次数求和表达式

$$T(n)=\sum_{i=1}^{n-1}1=n-1$$

（5）确定增长的阶 $T(n)=O(n)$

算法复杂性分析基本方法

例

- 非递归算法复杂性分析

算法 uniqueElements(A[0..*n*-1])
//验证给定数组中的元素是否全部唯一
//输入：实数数组A[0..*n*-1]
//输出：如果A中的元素全部唯一，返回“true”，否则，返回“false”

```
for i←0 to n-2 do
    for j←i +1 to n-1 do
        if A[i] = A[j]  return false
return true
```

算法复杂性分析基本方法

例

- 非递归算法复杂性分析

（1）算法输入规模：可以用数组元素个数n度量

（2）基本操作：比较

（3）比较操作执行次数与输入相关，需要考虑最坏、平均、最好情况

（4）建立基本操作执行次数求和表达式

最好：$O(1)$

最坏：

$$\sum_{i=0}^{n-2}\sum_{j=i+1}^{n-1}1=\sum_{i=0}^{n-2}(n-1)-(i+1)+1=i=\sum_{i=0}^{n-2}n-i-1=1+2+\ldots+(n-1)=n(n-1)/2$$

（5）确定增长的阶$O(n^2)$

平均效率如何分析？

算法复杂性分析基本方法

例

```
template<class Type>
void insertion_sort(Type *a, int n)
{
    Type key;                          //   cost    times
    for (int i = 1; i < n; i++){       //   c1      n
        key=a[i];                      //   c2      n-1
        int j=i-1;                     //   c3      n-1
        while( j>=0 && a[j]>key ){     //   c4      sum of t_i
            a[j+1]=a[j];               //   c5      sum of (t_i-1)
            j--;                       //   c6      sum of (t_i-1)
        }
        a[j+1]=key;                    //   c7      n-1
    }
}
```

算法复杂性分析基本方法

$$T(n)=c_1 n+c_2(n-1)+c_3(n-1)+c_4\sum_{i=1}^{n-1}t_i+c_5\sum_{i=1}^{n-1}(t_i-1)+c_6\sum_{i=1}^{n-1}(t_i-1)+c_7(n-1)$$

- 在最好情况下，$t_i=1$, for $1 \le i < n$;

$$T_{\min}(n)=c_1 n+c_2(n-1)+c_3(n-1)+c_4(n-1)+c_7(n-1)$$

$$=(c_1+c_2+c_3+c_4+c_7)n-(c_2+c_3+c_4+c_7)=O(n)$$

- 在最坏情况下，$t_i \le i+1$, for $1 \le i < n$;

$$\sum_{i=1}^{n-1}(i+1)=\frac{n(n+1)}{2}-1 \qquad \sum_{i=1}^{n-1}i=\frac{n(n-1)}{2}$$

$$T_{\max}(n)\le c_1 n+c_2(n-1)+c_3(n-1)+$$

$$c_4\left(\frac{n(n+1)}{2}-1\right)+c_5\left(\frac{n(n-1)}{2}\right)+c_6\left(\frac{n(n-1)}{2}\right)+c_7(n-1)$$

$$=\frac{c_4+c_5+c_6}{2}n^2+\left(c_1+c_2+c_3+\frac{c_4-c_5-c_6}{2}+c_7\right)n-(c_2+c_3+c_4+c_7)$$

$$=O(n^2)$$

在分析算法复杂性时，如果仅考虑复杂性的界，不需要分析算法所有操作，只需选择基本操作即可。

算法复杂性分析基本方法

- 递归算法的复杂性分析

讨论与理解

$$T(n)=\begin{cases} O(1) & n=1 \\ aT(n\text{-}1)+f(n) & n>1 \end{cases}$$

递归算法时间复杂度典型递推方程之一

$$T(n)=a^{n-1}\,T(1)+\sum_{i=2}^{n} a^{n-i} f(i)$$

推导

（1）若取a=2，$f(n)=O(1)$，汉诺塔问题
$T(n)=O(2^n-1)$
（2）若取a=1，$f(n)=n$-1，插入排序最坏情况
$T(n)= O(n^2)$

例

算法复杂性分析基本方法

- 递归算法的复杂性分析

讨论与理解

$$T(n)=\begin{cases}O(1) & n=1\\ aT(\frac{n}{b})+f(n) & n>1\end{cases}$$

递归算法时间复杂度典型递推方程之二

$$T(n)= n^{\log_b a}T(1)+\sum_{j=0}^{\log_b n-1} a^j f(\frac{n}{b^j})$$

推导

（1）若$f(n)=c$

实例

$$T(n)=\begin{cases}O(n^{\log_b a}) & a\neq 1\\ O(\log n) & a=1\end{cases}$$

（2）若$f(n)=cn$

实例

$$T(n)=\begin{cases}O(n) & a<b\\ O(n\log n) & a=b\\ O(n^{\log_b a}) & a>b\end{cases}$$

算法复杂性分析基本方法

- 递归算法的复杂性分析

```
void hanoi(int n, int a, int b, int c) {
    if (n = 1) move(a, c);
    else {
        hanoi(n-1, a, c, b);
        move(a, c);
        hanoi(n-1, b, a, c);
    }
}
```

求解汉诺塔问题算法

例

$$T(n) = \begin{cases} O(1) & n=1 \\ 2T(n-1)+1 & n>1 \end{cases}$$

推导

$$T(n) = 2^n - 1$$

算法复杂性分析基本方法

练习

- 复杂性分析 $T(1)=O(1)$

（1）$T(n)=T(n/2)+O(1)$	二分查找算法递推方程
（2）$T(n)=2T(n/2)+(n-1)$	归并排序算法递推方程
（3）$T(n)=T(n-1)+O(1)$	插入排序最好情况
（4）$T(n)=T(n-1)+(n-1)$	插入排序最坏情况
（5）$T(n)=T(n/3)+T(2n/3)+n$	

算法复杂性分析基本方法

练习

- 复杂性分析

```
for  i←0  to  n-1  do        // 行循环
   for  j←0  to  n-1  do    // 列循环
        M[ i ][ j ]←0           // 初始化
        for  k←0  to  n-1  do  // 用变量 k 表示变化的脚标
            M[ i ][ j ]←M[ i ][ j ] + A[ i ][ k ] * B[ k ][ j ]
return  M
```

方阵乘法

```
int binary(int n)
    count←1
    while  n > 1  do
        count←count + 1
        n←n%2
    return  count
```

十进制数的二进制位数

课堂演示

递归与非递归算法比较

关于递归Recursive

- 阶乘问题

$$n! = \begin{cases} 1 & n = 0 \\ n(n-1)! & n > 0 \end{cases}$$

递归实现

```
int factorial(int n){
    if(n==0) return 1;
    return n*factorial(n-1);
}
```

非递归实现

```
int factorial(int n){
    int fn=1;
    for(int i=2; i<=n; i++)
        fn=fn*i;
    return fn;
}
```

参照代码MyComputer.java（factorial部分），执行并分析效率

课堂演示

递归与非递归算法比较

讨论与理解算法递归与非递归实现对效率影响

- 斐波那契数列

$$F(n)=\begin{cases}0 & n=0\\ 1 & n=1\\ F(n-1)+F(n-2) & n>1\end{cases}$$

递归实现

```
int fibonacci(int n) {
    if (n <= 1) return n;
    return fibonacci(n-1)
            +fibonacci(n-2);
}
```

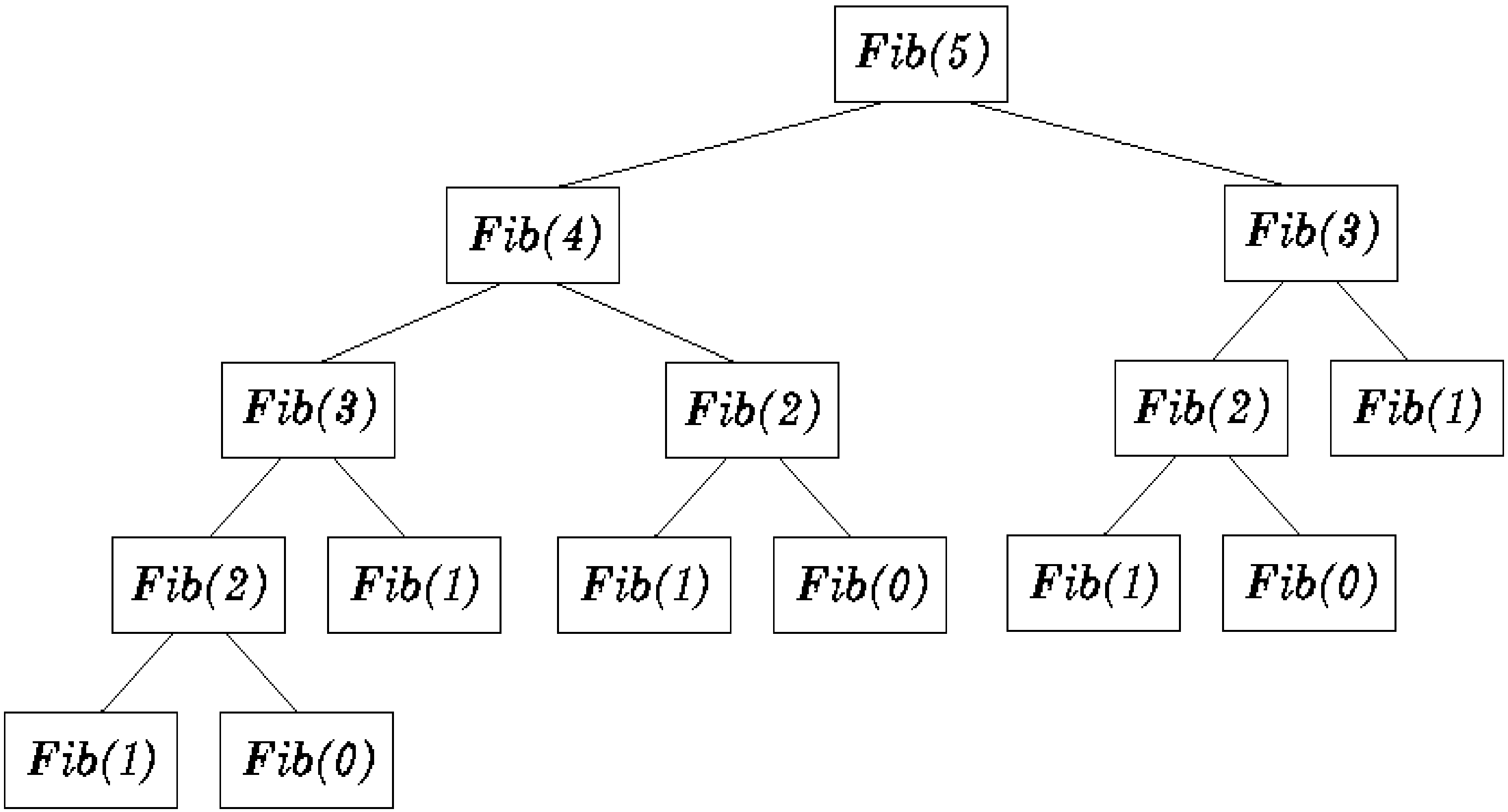
非递归实现

```
int fibonacci(int n){
    int[] a = new int[2];
    a[0] = 1; a[1] = 1;
    for (int i = 2; i < n; i++) {
        a[i%2] = a[(i – 1)%2] + a[(i – 2)%2];
    }
    return a[n %2];
}
```

参照代码MyComputer.java（fibonacci部分），执行并分析效率

递归与非递归算法比较

斐波那契数列递归求解效率低下的重要原因在于大量重复计算



递归与非递归算法比较

- 关于函数的递归与非递归定义问题

n!=n*(n-1)! → n!=n*(n-1)*…*1

F(n)=F(n-1)+F(n-2) → ?

$$F(n)=\frac{1}{\sqrt{5}}\left(\frac{1+\sqrt{5}}{2}\right)^n-\frac{1}{\sqrt{5}}\left(\frac{1-\sqrt{5}}{2}\right)^n=\frac{1}{\sqrt{5}}(\phi^n-\hat{\phi}^n)$$

$$\phi=(1+\sqrt{5})/2\approx 1.61803 \qquad \hat{\phi}=-1/\phi\approx -0.61803$$

递归与非递归算法比较

- Ackerman函数
  - Ackerman函数A(n, m)有两个独立的整型变量m≥0和n≥0，定义如下：

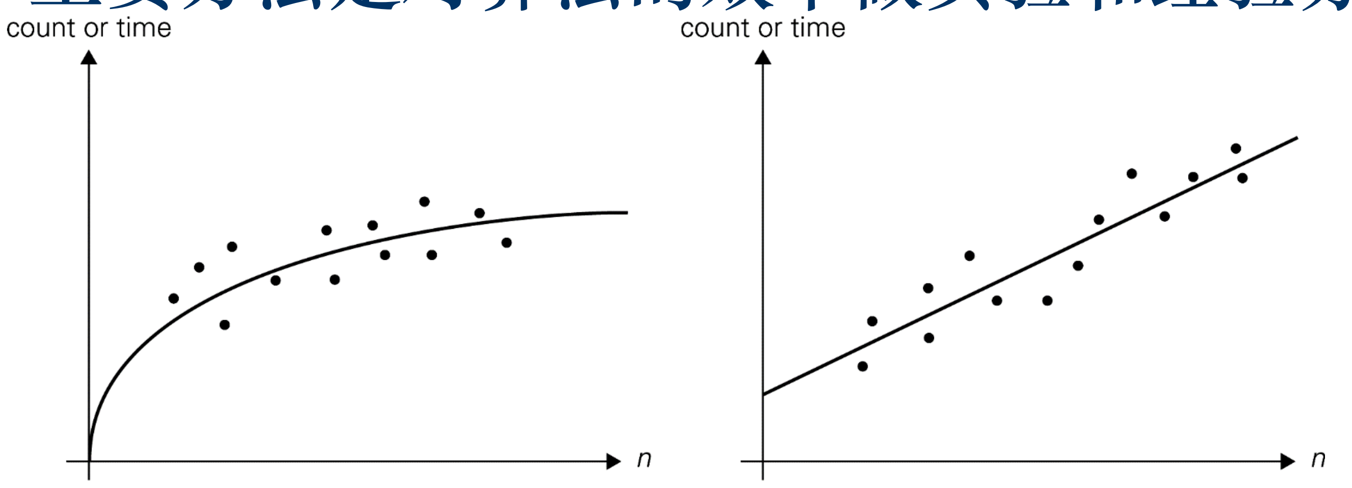
$$
\begin{cases}
A(1,0) = 2 & \\
A(0,m) = 1 & m \ge 0 \\
A(n,0) = n + 2 & n \ge 2 \\
A(n,m) = A(A(n-1,m),m-1) & n,m \ge 1
\end{cases}
$$

当一个函数及它的一个变量是由函数自身定义时，称这个函数是双递归函数。

并非所有递归算法都有非递归定义。

经验与实验分析方法

- 数学远远不是万能的，即使许多貌似简单的算法，有时也很难用数学的精确性和严格性来分析，尤其对一些综合性算法，或是在做平均效率分析的时候。
- 除了可以对算法的效率做数学分析以外，另一种主要方法是对算法的效率做实验和经验分析。



(a) (b)

Discuss