算法定义

- 定义：一个有穷的指令集，这些指令为解决某一特定任务规定了一个运算序列

  算法 + 数据结构 = 程序

- 特性：
  - 输入 有 0 个或多个输入
  - 输出 有一个或多个输出（处理结果）
  - 确定性 每步定义都是确切、无歧义的
  - 有穷性 算法应在执行有穷步后结束
  - 有效性 每一条运算应足够基本，可用计算机指令实现

算法评价

- **正确性**：输入正确的数据，应得出正确的结果。每一个算法，要考虑前置条件和后置条件。
  - 前置条件：算法正确执行需满足的条件。
  - 后置条件：算法执行后应得到的正确结果。
- **可读性**：算法的程序逻辑必须易于阅读和理解，这样才能正确地调试、更细和扩展。
  - 算法代码的结构具有结构化的层次。
  - 算法必须加注释，说明算法的功能、思想、参数使用、程序块的功能等。
  - 算法中各种名字的命名必须有实际含义。

- **健壮性**：算法必须能预见可能的错误，能对意外情况适当地做出反应和处理。
  - 不合理的数据输入。
  - 内存不足。
  - 文件打开失败、文件读写失败等。
- **高效性**：算法应具有良好的时空性能。
  - 时间复杂度度量。
  - 空间复杂度度量。
- 在做算法评价时，需要考虑“问题规模”。所谓问题规模，就是数据输入量或初始数据量。

几种常用算法的类型

- 三种最基本的常用算法：
  - 穷举型：按某种顺序进行逐一枚举和检验，并从中找出那些符合要求的候选解作为问题的解。
  - 递推型：由已知至i-1规模的解，通过递推，获得规模为 i 的解。
  - 递归型：把规模为N的问题分解成一些规模较小的问题，然后从这些小问题的解构造出大问题的解 。

穷举法举例

- 求解“百钱买百鸡”问题：公鸡每只5钱，母鸡每只3钱，小鸡3只1钱。
- 求解思路：设公鸡数为x，母鸡数为y，小鸡数为z，则可以得到下面的整数不定方程组：

$$x + y + z = 100$$

$$5*x + 3*y + z/3 = 100$$

- **利用穷举法可以写出下面的算法程序：**

```
#include <stdio.h>
void main() {
      int x, y, z;
      for ( x = 0; x <= 100; x++ )
         for ( y = 0; y <= 100; y++ )
            for ( z = 0; z <= 100; z++ )
               if ( x+y+z == 100 && 5*x+3*y+z/3 == 100 )
                  printf ("%d%d%d\n", x, y, z);
}
```

- **执行结果**

穷举法的算法分析

- 虽然上述程序很简单，但分析一下可知，此程序为三重循环，循环次数为$101^3 = 1030301$，为10^6量级。如果改为“万钱买万鸡”，循环次数将达10^{12}量级，计算量太大，这正是穷举法的缺点。
- 为此，可考虑对程序做优化。
  - x、y的值确定后，z的值也能确定，去掉最内层循环；
  - 如果用所有的钱去买公鸡，最多可买20只，而用所有的钱去买母鸡，最多可买33只，所以x、y的值范围可限定在20和33；

- 与上面程序等效的程序：

```
#include <stdio.h>
void main() {
    int x, y, z;
    for ( x = 0, x <= 20; x++ )
        for ( y = 0; y <= 33; y++ ) {
            z = 100−x−y;
            if ( 5*x+3*y+z/3 == 100 )
                printf ( "%d%d%d\n", x, y, z );
        }
}
```

- 这个程序的循环次数只有21*34 = 714。

递推法举例

- 编写程序，用递推法计算斐波那契（Fibonacci）数列的第n项。
- 求解思路：斐波那契（Fibonacci）数列为0, 1, l, 2, 3, 5, 8, 13, …，即：

  F(0) = 0，F(1) = 1，

  F(n) = F(n−1)+F(n−2)，　　当n > 1时。

- 用递推法编写的程序为：

```
# include<stdio.h>
int Fib ( int n ) {
    int f0 = 0, f1 = 1, f, i;
    if ( n == 0 || n == 1 ) return n;
    for ( i = 2; i <= n; i++ ) {
        f = f0 + f1;       //由前两步结果推出当前结果
        f0 = f1;           //原前一步当作下一次的前两步
        f1 = f;            //当前结果当作下一次的前一步
    }        //在进行向前传递时，要注意传递的时序
    return f;
}
```

- 主程序调用方式：

```
int n = 7;
printf ( ”Fib(%d)=%d\n”, n, Fib (n) );
```

递归法举例

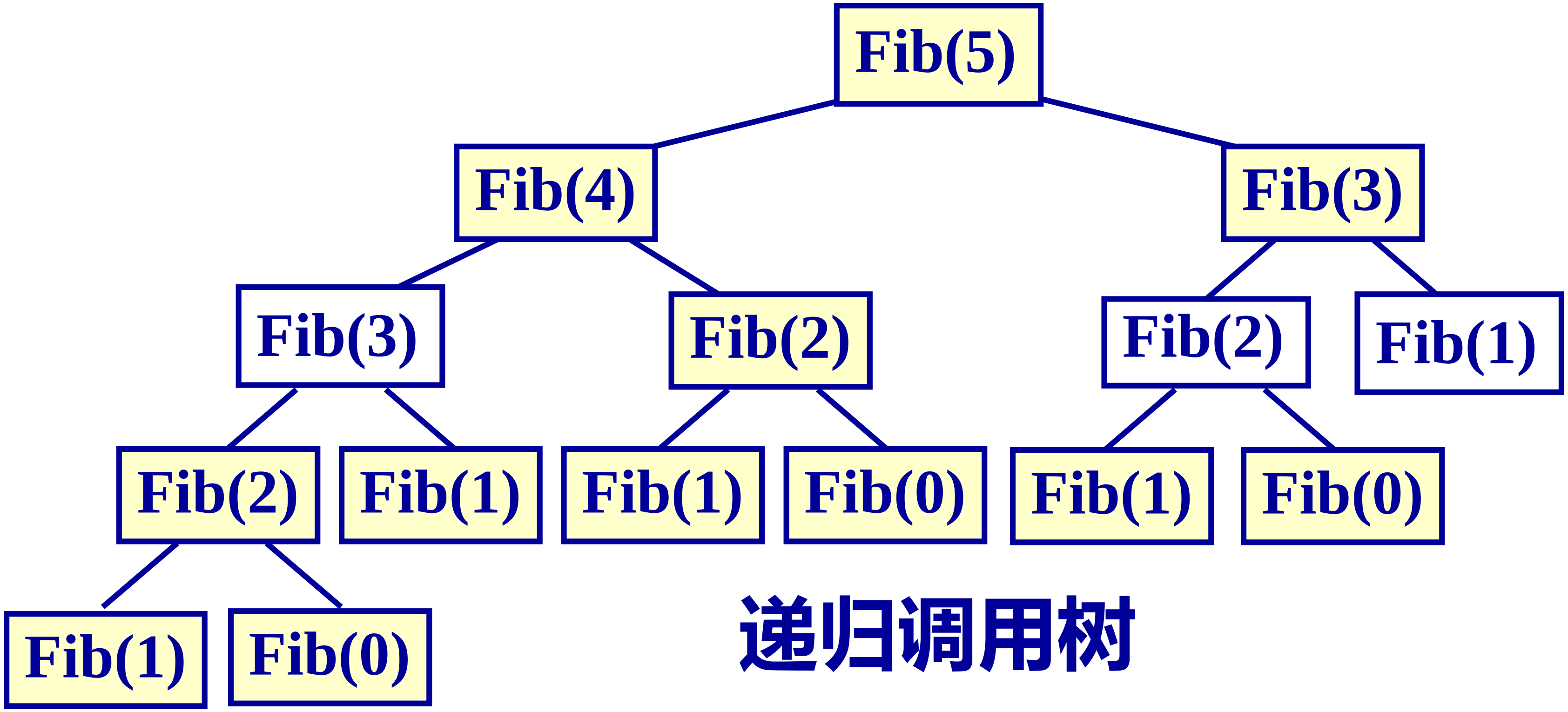
- 所谓递归算法就是在函数过程中直接或间接调用自己。还是求斐波那契数列的问题，相应的递归程序为：

```
int Fib ( int n ) {
    if ( n == 0 || n == 1 ) return n;          //结束项
    else return Fib (n−1) + Fib(n−2);          //递归项
}
```

- 递归算法简单，但不能无限递归。因此，算法中需要设置递归结束条件。

递归法分析

- 递归法的时间效率较低，原因是重复计算太多。
- 例如，计算Fib(5)，总计算次数为2Fib(6)−1 = 15。



递归调用树

更复杂的实例

- 例如，编写一个算法，在一个一维整数数组A[n]中查找满足给定值 x 的元素，找到后函数返回该元素的位置，否则函数返回 –1。

```
int Search ( int A[ ], int x, int n ) {
    int i = 0;
    while ( i < n )                    //通过循环枚举检查
        if ( A[i] == x ) return i;     //满足要求
        else i = i+1;
    return –1;
}
```

- 再看递归法。每次递归自己是为了缩小查找区间，逐步逼近到要查找的元素。在函数参数表中增加本次递归调用时的开始查找位置。

```
int Search ( int A[ ], int x, int n, int start ) {
    if ( start == n ) return –1;
    if ( A[start] == x ) return start;
    else return Search ( A, x, n, start+1 );
}
```

- 递归的主调用语句为loc = Search (A, x, n, 0)。
  如果查找失败或查找成功，函数直接返回结果，否则通过递归，到start以后的区间逐个检查。

性能分析与度量

- 算法就是为了问题求解。算法的效率是衡量是否具有可计算性的关键。
- 性能分析的目的就是要了解算法的效率。
- 性能（Performance），指算法功能实际执行的功效或表现如何。主要从算法执行的时间和空间效率进行分析。分析方式有：
  - 算法的后期测试
  - 算法的事前估计

算法的后期测试

- 在算法中的某些部位插装时间函数 time ( )，测定算法完成某一功能所花费时间。
- 例如，有一个顺序搜索 (Sequenial Search)算法

```
int seqsearch ( int a[ ], int n, int x ) {
    //在a[0],…,a[n-1]中搜索x
    int i = 0;
    while ( i < n && a[i] != x ) i++;
    if ( i == n ) return -1;
    return i;
}
```

- 插装 time( ) 后的计时程序为

```
double start, stop;
time (&start);
int k = seqsearch (a, n, x);
time (&stop);
double runTime = stop - start;
printf ( ”%d%d\n " , n, runTime );
```

- 可以测算。但受限于硬件设备和操作系统、编译器等，测算比较有一定困难。

算法的事前估计

- 空间复杂度度量
  - 存储空间的固定部分
    附加存储空间，常数、简单变量、定长成分（如数组元素、结构成分、对象的数据成员等）变量所占空间
  - 可变部分
    尺寸与问题规模有关的成分变量所占空间、递归栈所用空间、通过 malloc 和 free 命令动态使用空间

- 时间复杂度度量
  - 运行时间 = 算法每条语句执行时间之和。
  - 每条语句执行时间 = 该语句的执行次数 (频度)×语句执行一次所需时间。
  - 语句执行一次所需时间取决于机器的指令性能和速度和编译所产生的代码质量，很难确定。
  - 设每条语句执行一次所需时间为单位时间，则一个算法的运行时间就是该算法中所有语句的频度之和。

- **例 求两个 n 阶方阵的乘积 C = A×B**

```
void MatrixMultiply ( int A[ ][ ], int B[ ][ ],
        int C[ ][ ], int n ) {
    for ( int i = 0; i < n; i++ )              … n+1
        for ( int j = 0; j < n; j++ ) {        … n(n+1)
            C[i][j] = 0;                       … n²
    for ( int k = 0; k < n; k++ )              … n²(n+1)
        C[i][j] = C[i][j] + A[i][k] * B[k][j]; … n³
    }
}
```

$$2n^3 + 3n^2 + 2n + 1$$

渐进时间复杂度 大O表示法

- 算法中所有语句的频度之和是矩阵阶数 n 的函数

$$T(n) = 2n^3 + 2n^2 + 2n + 1$$

- 称 n 是问题的规模。则时间复杂度 T(n) 是问题规模 n 的函数。
- 当 n 趋于无穷大时，称时间复杂度的数量级为算法的渐进时间复杂度

$$T(n) = O(n^3)$$ ——算法的大O表示

- 大O表示表明当$n \rightarrow \infty$时，T(n)的变化趋势。

- **大O表示法的加法规则（针对并列程序段）**

$$\left.\begin{array}{l} T_1(n) = O(\,f\,(n)\,) \\ T_2(m) = O(\,g\,(m)\,) \end{array}\right\}\text{并列}$$

$$T(n, m) = T_1\,(n) + T_2\,(m)$$

$$= O(\max\,(f\,(n), g\,(m)))$$

- **例如，有三段并列程序段**

```
x = 0;  y = 0;
```
$T_1(n) = O(1)$

```
for ( int k = 0; k < n; k ++ )  x ++;
```
$T_2(n) = O(n)$

```
for ( int i = 0; i < n; i++ )
    for ( int j = 0; j < n; j++ )  y ++;
```
$T_3(n) = O(n^2)$

- 整个程序的渐进时间复杂性为

$$T(n) = T_1(n)+T_2(n)+T_3(n) = O(\max(1, n, n^2)) =$$
$$= O(n^2)$$

- 大O表示法的乘法规则（针对嵌套程序段）

$$\left.\begin{array}{l} T_1(n) = O(\,f\,(n)\,) \\ T_2(m) = O(\,g\,(m)\,) \end{array}\right\} \text{嵌套}$$

$$T(n, m) = T_1(n)\times T_2(m) = O(f\,(n)\times g\,(m))$$

- 例如，一个选择排序程序，算法思路为：共循环 n-1 次，循环变量 i 从 1 到 n-1，每次循环从 i 到 n 选择最小者，将其对调到第 i 个元素位置。

```
void SelectSort ( int A[ ], int n ) {
 // n 是表当前长度
    for ( i = 0; i < n−1; i++ ) {
        k = i;
        for ( j = i+1; j < n; j++ )
            if ( A[j] < A[k] )
                k = j;
        if ( k != i ) Swap(A[k], A[i] );
    }   //一趟比较
}
```

SelectSort n-1趟 $T_1(n) = O(n)$

选最小者 A[k] n−i−1次比较

交换 swap(A[k], A[i] 1次比较

- **渐进时间复杂度**

$$O(f(n)*g(n)) = O(n^2)$$

$$\because \sum_{i=0}^{n-2}(n-i-1) = \frac{n(n-1)}{2}$$

算法复杂性的不同数量级的变化

n	$\mathrm{lon}_2 n$	$n\log_2 n$	n^2	n^3	2^n	$n!$
4	2	8	16	64	16	24
8	3	24	64	512	256	80320
10	3.32	33.2	100	1000	1024	3628800
16	4	64	256	4096	65536	$2.1*10^{13}$
32	5	160	1024	32768	$4.3*10^{9}$	$2.6*10^{35}$
128	7	896	16384	2097152	$3.4*10^{38}$	∞
1024	10	10240	1048576	$1.07*10^{9}$	∞	∞
10000	13.29	132877	10^{8}	10^{12}	∞	∞

- $c < \log_2 n < n < n\log_2 n < n^2 < n^3 < 2^n < 3^n < n!$

- 有时, 算法的时间复杂度不仅依赖于问题规模n，还与输入实例的初始排列有关。
- 例如，在数组 A[n] 中查找给定值 k 的算法：

```
int i = n-1;
while ( i >= 0 && A[i] != k )
    i--;
return i;
```

- 算法的语句 i--的频度不仅与 n 有关，还与 A[ ] 中各元素的取值，以及 k 的取值有关。

算法分析例题

例题1 算法的时间复杂度与（ ）有关。

A. 问题规模　　B. 计算机硬件的运行速度

C. 源程序的长度　　D. 编译后执行程序的质量

解答：A。算法的具体执行时间与计算机硬件的运行速度、编译产生的目标程序的质量有关，但这属于事后测量。算法的时间复杂度的度量属于事前估计，与问题的规模有关。

例题2 某算法的时间复杂度是$O(n^2)$，表明该算法（ ）。

A. 问题规模是n^2　　B. 问题规模与n^2成正比

C. 执行时间等于n^2　　D. 执行时间与n^2成正比

解答：D。算法的的时间复杂度是$O(n^2)$，这是设定问题规模为 n 的分析结果，所以A、B 都不对；它也不表明执行时间等于n^2，它只表明算法的执行时间$T(n) \le c \times n^2$（c为比例常数）。

有的算法，如$n \times n$矩阵的转置，时间复杂度为$O(n^2)$，不表明问题规模是n^2。

例题3 有实现同一功能的两个算法A_1和 A_2，其中A_1 的渐进时间复杂度是$T_1(n) = O(2^n)$，A_2 的渐进时间复杂度是$T_2(n) = O(n^2)$。仅就时间复杂度而言，具体分析这两个算法哪个好。

解答：比较算法好坏需比较两个函数2^n和n^2。

当$n = 1$时，$2^1 > 1^2$，算法A_2好于A_1

当$n = 2$时，$2^2 = 2^2$，算法A_1与A_2相当

当$n = 3$时，$2^3 < 3^2$，算法A_1好于A_2

当$n = 4$时，$2^4 > 4^2$，算法A_2好于A_1

当$n > 4$时，$2^n > n^2$，算法A_2好于A_1

当$n\to\infty$时，算法A_2在时间上显然优于A_1。

例题4　设n是描述问题规模的非负整数，下面程序片段的时间复杂度是

```
x = 2;
while ( x < n/2 )
    x = 2*x;
```

A. $O(\log_2 n)$　　B. $O(n)$　　C. $O(n\log_2 n)$　　D. $O(n^2)$

解答：选A。找关键操作，即最内层循环中的执行语句 x = 2*x。因为每次循环x都成倍增长，设 $x = 2^k < n/2$，$2^{k+1} < n$，则$k < \log_2 n-1$，实际while循环内的语句执行了$\log_2 n-2$次。

例题5 求整数n（n≥0）阶乘的算法如下，其时间复杂度是

```
int fact ( int n ) {
    if ( n <= 1 ) return 1;
    return n*fact ( n-1 );
}
```

计算 fact(n−1)

A. $O(\log_2 n)$　B. $O(n)$　C. $O(n\log_2 n)$　D. $O(n^2)$

解答：选B。因为T(1) = 1，T(n) = 1+T(n−1) = 1+(1+T(n−2)) = 2+(1+T(n−3)) = 3+(1+T(n−4)) = … = (n−1)+T(1) = n。

计算 ‘*’