1 Algorithm Analysis

1.1 Definition

- Input：可以存在0输入
- Output：至少一个输出
- Definiteness：clear and unambiguous
- Finiteness：估计算法什么时间能够完成
- Effectiveness

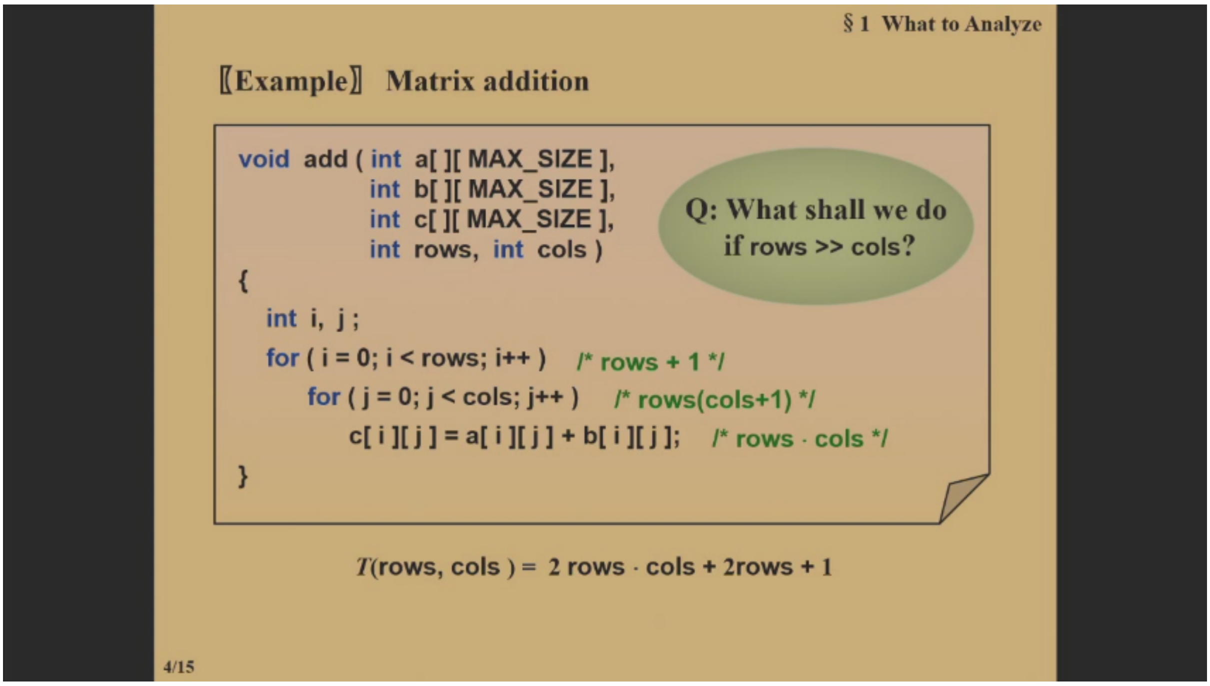
1.2 Difference between program and algorithm

- Algorithm 可以用自然语言描述
- Program 用编程语言

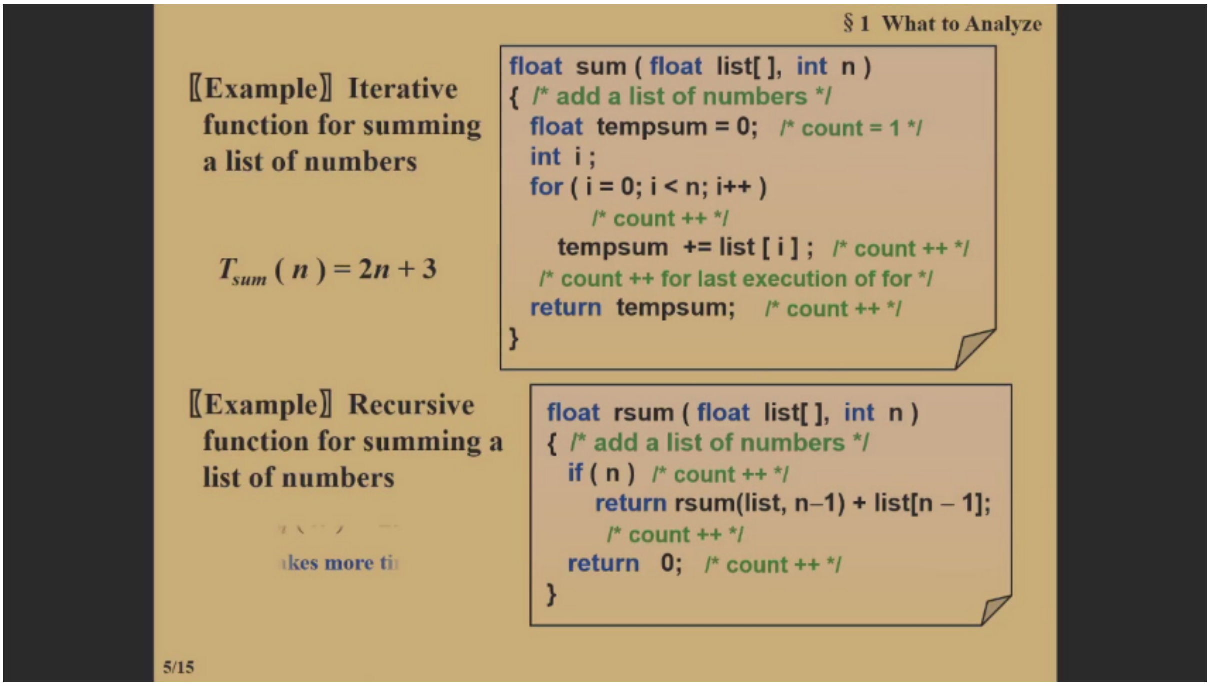
1.3 What to Analyze

1.3.1 时间复杂度

- $T_{arg}(N)$: 平均时间复杂度
- $T_{worst}(N)$: 最坏时间复杂度



- Matrix addition: `for(i)` runs row+1 times, bulbul

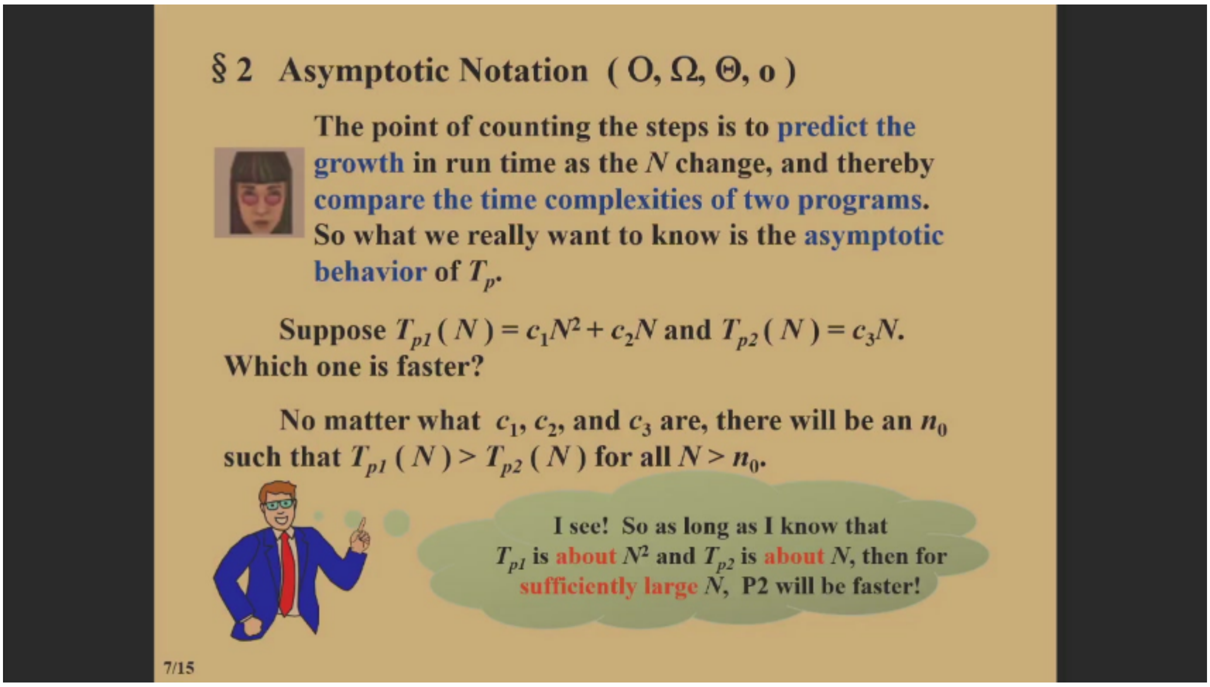


- 写成递归Recursive function后：递归执行n+1次，前n次都是执行if里的语句，每次执行两个语句，最后一次执行了`return 0`
- 所以写成递归总计执行语句$2N+1$次

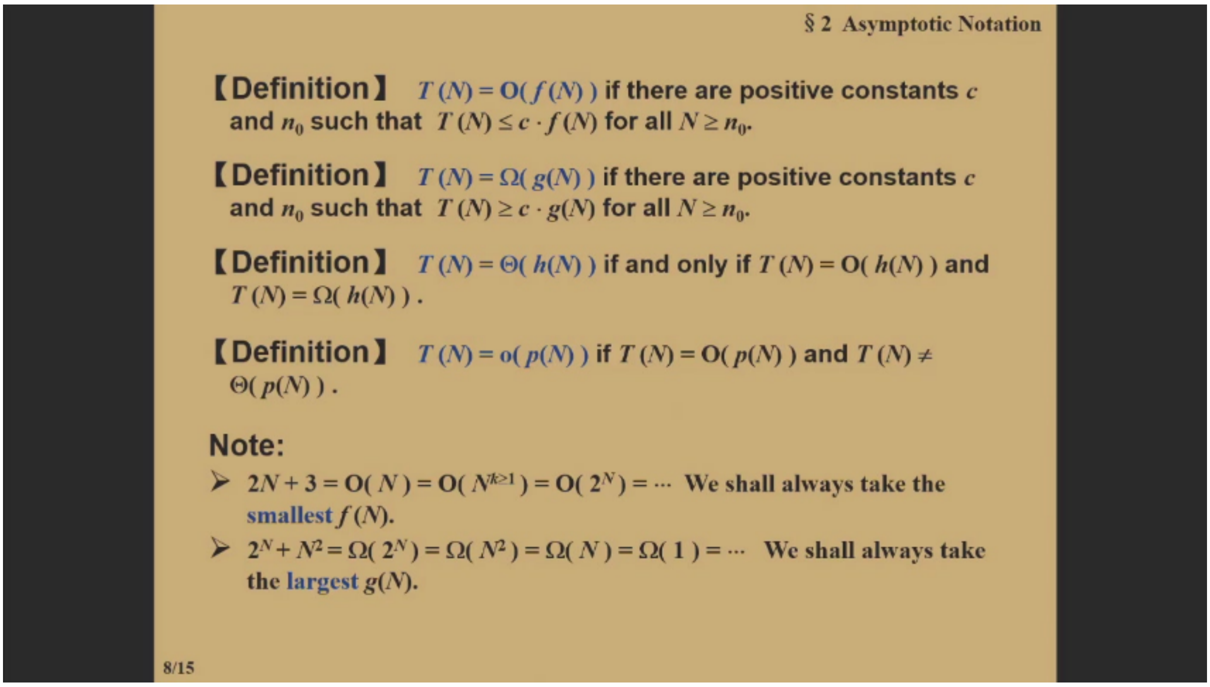
1.4 Asymptotic Notation

1.4.1 Defination

$O \quad \Omega \quad \Theta \ o$

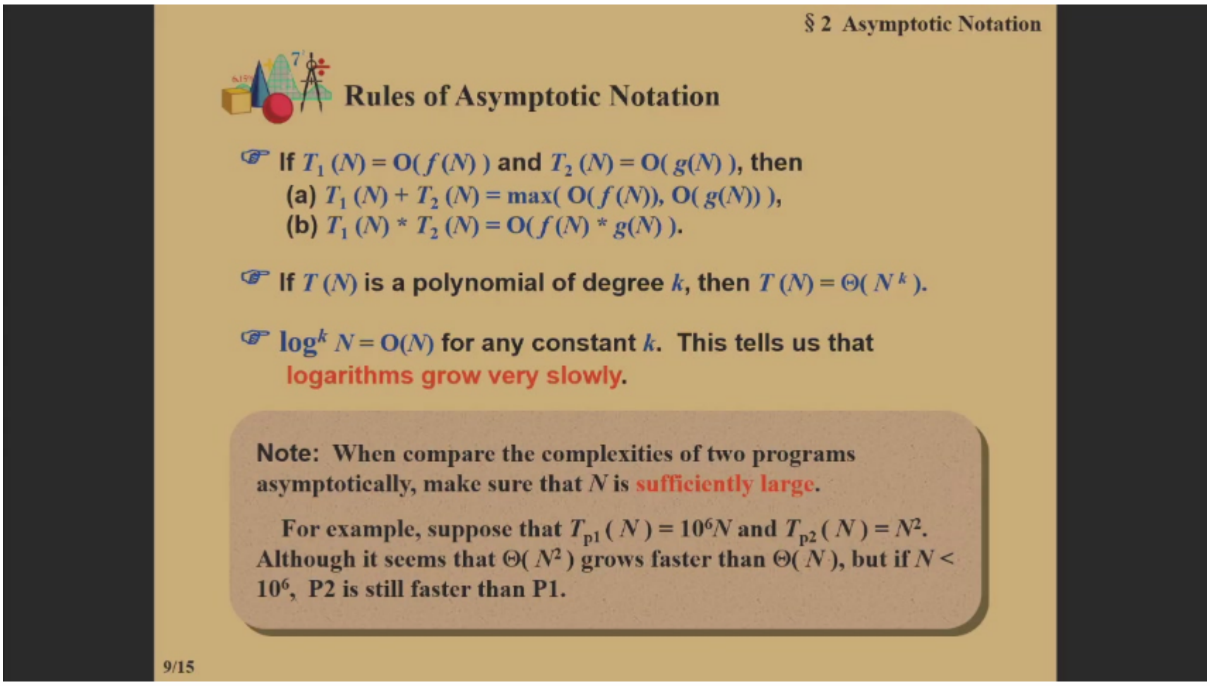


$$\exists N > n_0, st\ T_{p1}(N) > T_{p2}(N) \tag{1}$$



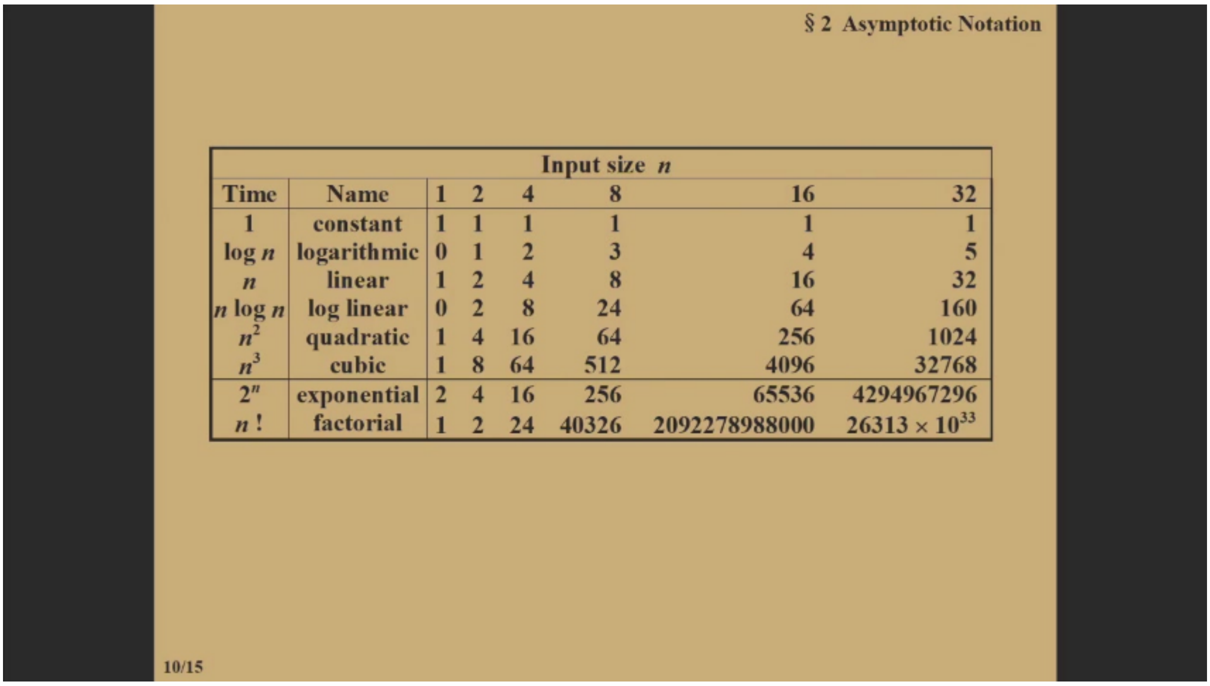
- O ：定义worst case的bound，最差不过这样，Upper bound，取最大的最好。Higher bound。
- Ω ：定义最好的情况下，最好的情况，Lower bound，取最小的最好。Lower Bound。
- $\Theta : O(f(N)) = \Omega(g(N))$ ，任何情况都相同，比如Matrix Addition
- $o : O(f(N)) \rightarrow \Omega(g(N))$ ，趋近，但是永远不相等

1.4.2 Rules



- 两个复杂度相乘，对应于嵌套语句`for(for())`
- 两个复杂度相加，取决于最复杂的那个，对应于程序内的两条语句

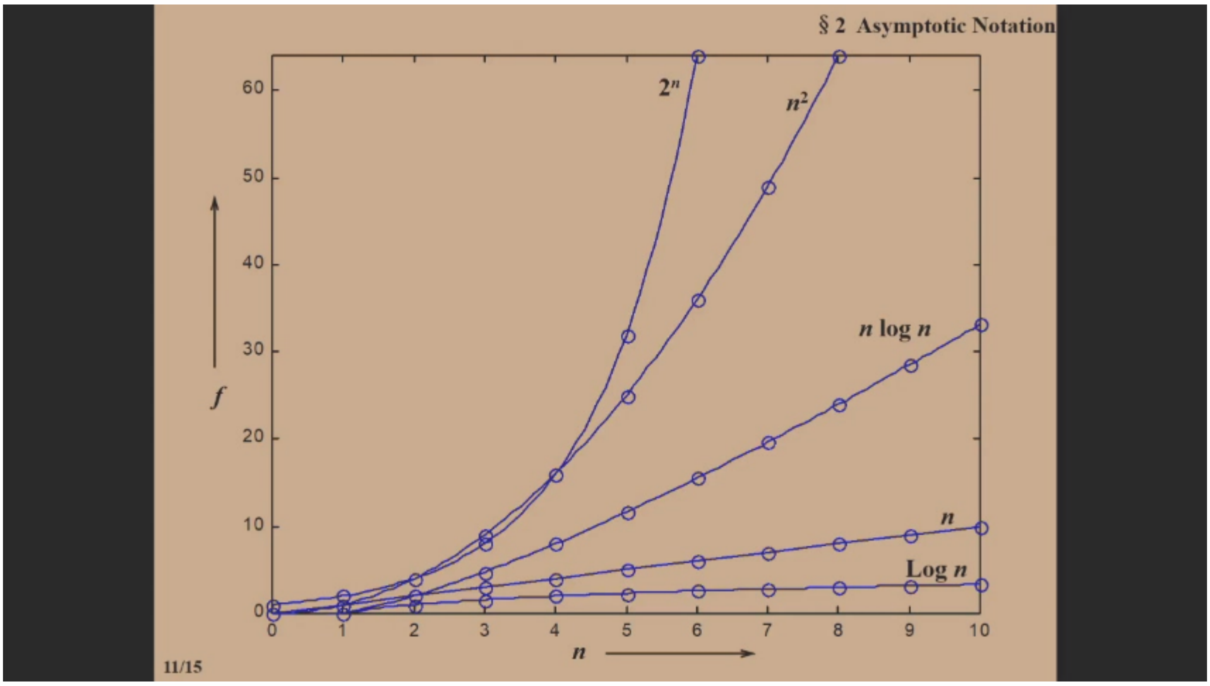
1.4.3 Description


§ 2 Asymptotic Notation

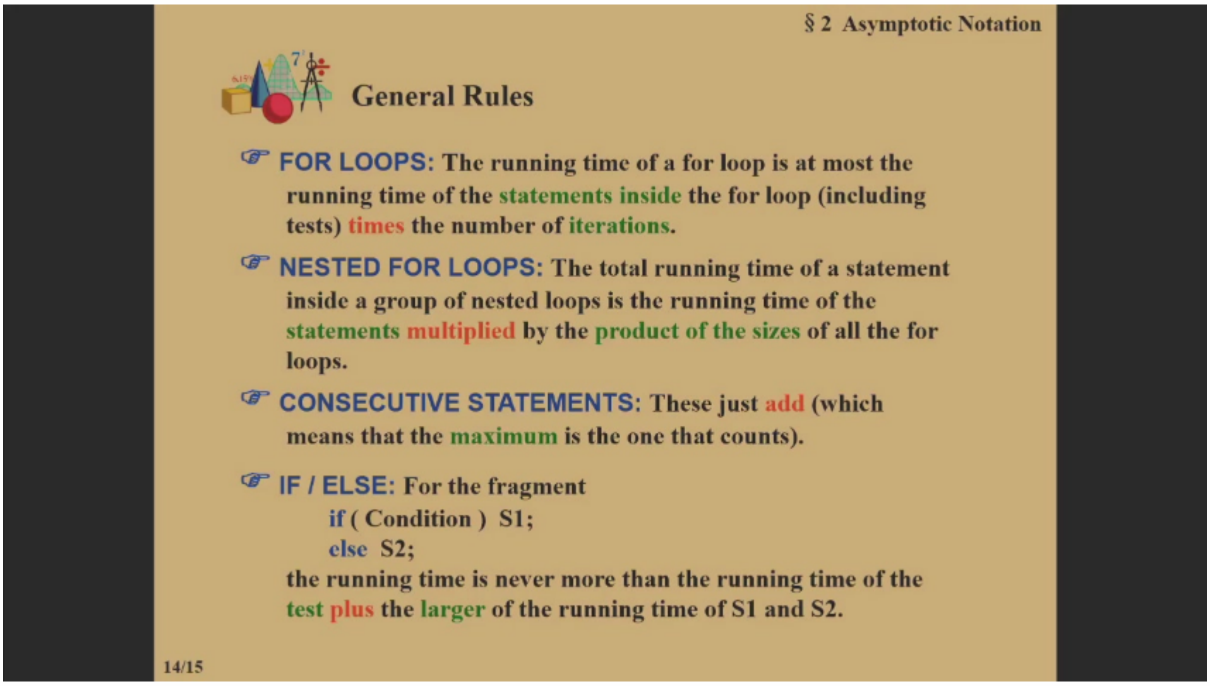
		Input size n					
Time	Name	1	2	4	8	16	32
1	constant	1	1	1	1	1	1
$\log n$	logarithmic	0	1	2	3	4	5
n	linear	1	2	4	8	16	32
$n \log n$	log linear	0	2	8	24	64	160
n^2	quadratic	1	4	16	64	256	1024
n^3	cubic	1	8	64	512	4096	32768
2^n	exponential	2	4	16	256	65536	4294967296
$n!$	factorial	1	2	24	40326	2092278988000	26313×10^{33}

10/15

- 看一下第二列的描述
- Quadratic time：选择排序（rows * columns）
- Cubic time：解方程，分解
- Exponential：下棋…

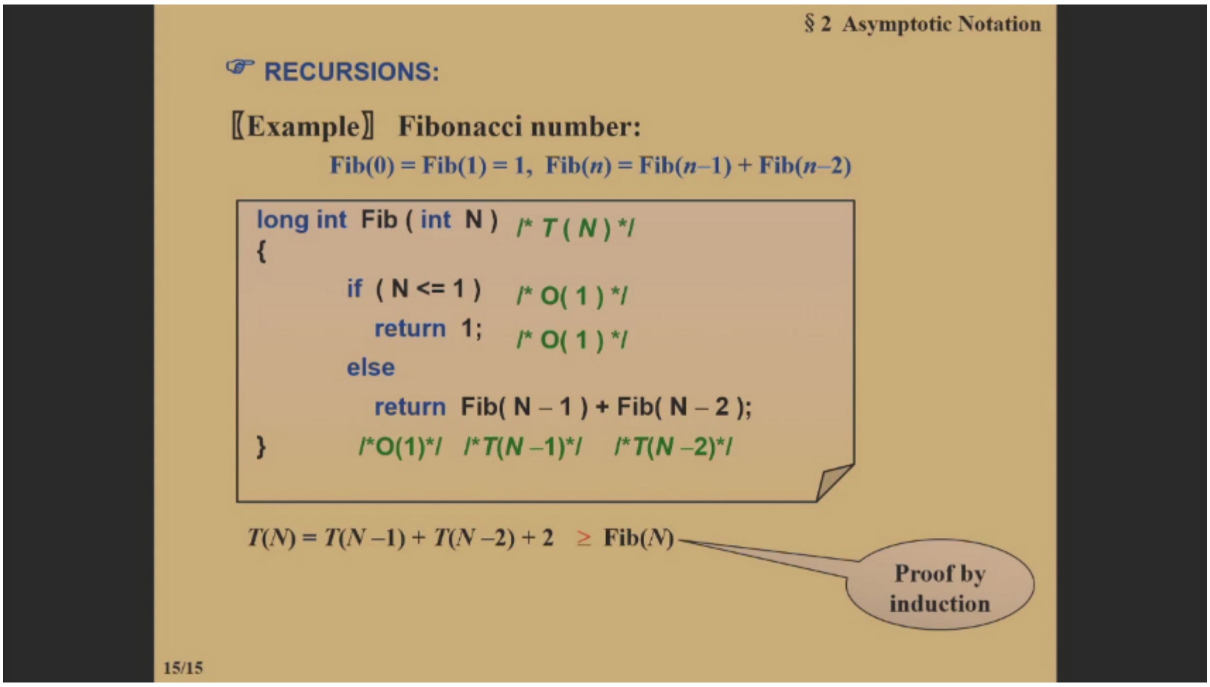


1.4.4 General rules



- FOR LOOPS： 声明*iterations
- NESTED FOR LOOPS 嵌套循环：所有循环声明的乘积
- CONSECUTIVE STATEMENTS：很多声明在一起，取最慢的那个
- IF/ELSE：test * 最慢的语句

1.4.4.1 Recursions



$$\left(\frac{3}{2}\right)^{N} < T\left(N\right) < \left(\frac{5}{3}\right)^{N} \tag{2}$$

2 HOMEWORK

2.1 Nested Loops

```
if ( A > B ){
  for ( i=0; i<N*2; i++ )
    for ( j=N*N; j>i; j-- )
      C += A;
}
else {
  for ( i=0; i<N*N/100; i++ )
    for ( j=N; j>i; j-- )
      for ( k=0; k<N*3; k++)
        C += B;
}
```

- 时间复杂度：O()；分开来算，i怎么变，下面的j怎么变，是否进入下一步循环，再然后考虑k的稳定循环次数

2.2 iteration

- P1: $T(1)=1, T(N)=T(N/3)+1$
- P2: $T(1)=1, T(N)=3T(N/3)+1$
- 类似于高中的数列迭代
- ANS： $O(\log N)$ for P1, $O(N)$ for P2

3 Compare the Algorithm

3.1 Example

- Given (possibly negative) integers A1, A2, …, AN, find the maximum value of $$求最大子序列

3.1.1 Algorithm 1

```
int  MaxSubsequenceSum ( const int A[ ],  int  N )
{
   int  ThisSum,  MaxSum,  i,  j,  k;
/* 1*/  MaxSum = 0;   /* initialize the maximum sum */
/* 2*/  for( i = 0; i < N; i++ )  /* start from A[ i ] */
/* 3*/        for( j = i; j < N; j++ ) {   /* end at A[ j ] */
/* 4*/            ThisSum = 0;
/* 5*/            for( k = i; k <= j; k++ )
/* 6*/                ThisSum += A[ k ];  /* sum from A[ i ] to A[ j ] */
/* 7*/                if ( ThisSum > MaxSum )
/* 8*/                    MaxSum = ThisSum;  /* update max sum */
        }  /* end for-j and for-i */
/* 9*/  return  MaxSum;
}
```

$$T(N) = O(N^3) \tag{3}$$

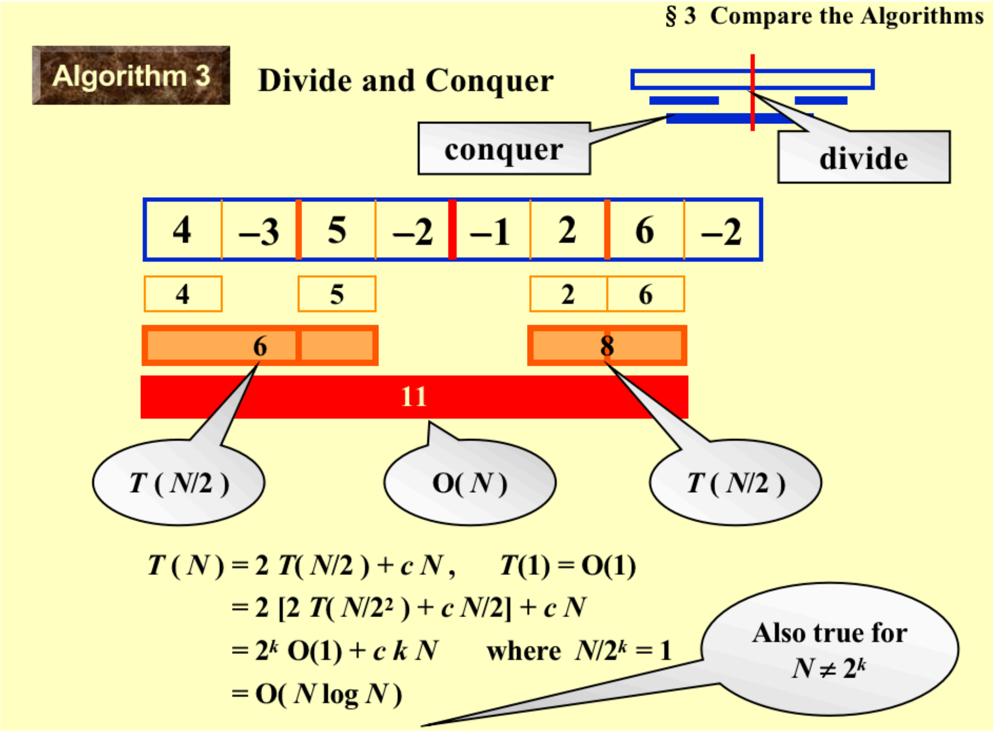
3.1.2 Algorithm 2

```
int  MaxSubsequenceSum ( const int A[ ],  int  N )
{
   int  ThisSum,  MaxSum,  i,  j;
/* 1*/  MaxSum = 0;   /* initialize the maximum sum */
/* 2*/  for( i = 0; i < N; i++ )  {   /* start from A[ i ] */
/* 3*/        ThisSum = 0;
/* 4*/        for( j = i; j < N; j++ ) {   /* end at A[ j ] */
/* 5*/            ThisSum += A[ j ];  /* sum from A[ i ] to A[ j ] */
/* 6*/            if ( ThisSum > MaxSum )
/* 7*/                MaxSum = ThisSum;  /* update max sum */
        }  /* end for-j */
   }  /* end for-i */
/* 8*/  return  MaxSum;
}
```

- 这个算法去掉了一个循环
- 之前的算法把从A[i]开始的，每种长度都要累加一遍，没必要。
- 实际上只要从此开始，不断地加，记录MaxSum即可

$$T(N) = O(N^2) \tag{4}$$

3.1.3 Algorithm 3: Devide and Conquer



- 最后一步证明要自己推一遍
- 出现 $logN$ 是因为有二分情况
- 这个还是再查一下，学一下，没太懂这里（查完了，代码如下）
- $k = logN$

$$T(N) = O(NlogN) \tag{5}$$

```
//求出最大子序列 4 , -3, 5, -2, -1,2, 6, -2
#include <stdio.h>
int max (int a,int b,int c)
{
    int ret;
    if(a > b)
    {
        ret = a;
    }else
    if(a <= b)
    {
        ret = b;
    }
```

```
    if(ret >= c)
    return ret;
    else
    return c;
}
 int Findmaxsum(int box[],int size,int left,int right)      //参数（数组名，数组大小，左边界，右边界）
{
    int mid = (right + left) / 2;
    if(left == right)                                        //分治递归要注意出口条件
    {
        return box[left];
    }
    int leftsum = Findmaxsum(box,size,left,mid );          //求出左半区最大子序列和 ，要有递归信任，不要纠结层层深入，假设该函数是正确的。
    int rightsum = Findmaxsum(box,size,mid + 1,right);       //求出右半区最大子序列和
    int leftbordersum = 0;
    int rightbordersum = 0;
    int i;
    int thissum = 0;
    for(i = mid + 1 ;i <= right;i++)                         //求出含有中间分界点的右半区最大子序列和   （如果最大子序列横跨中间分界点，那么肯定包含中间分界点，）
    {
        thissum += box[i];
        if(rightbordersum < thissum)
        {
            rightbordersum = thissum;
        }
     }
     thissum = 0;
     for(i = mid ;i >= left;i--)                          //求出含有中间分界点的左半区最大子序列和
     {
         thissum += box[i];
         if(leftbordersum < thissum)
         {
             leftbordersum = thissum;
         }
     }
     int midsum = leftbordersum + rightbordersum;              //横跨左右半区最大子序列和
     return max(midsum,leftsum,rightsum);                       //左半区最大子序列和，右半区最大子序列和，跨半区最大子序列和，三者中最大的为所求者
```

```c


}
int main ()
{
    int box[8] = {4,-3,5,-2,-1,2,5,-2};
    int ret = 0;
    ret = Findmaxsum(box,8,0,7);
    printf("%d",ret);
    return 0 ;
 }
```

3.1.4 *Algorithm 4 On-Line Algorithm*

```c
int MaxSubsequenceSum( const int  A[ ],  int  N )
{
  int  ThisSum, MaxSum, j;
/* 1*/  ThisSum = MaxSum = 0;
/* 2*/  for ( j = 0; j < N; j++ ) {
/* 3*/        ThisSum += A[ j ];
/* 4*/        if  ( ThisSum > MaxSum )
/* 5*/            MaxSum = ThisSum;
/* 6*/        else if ( ThisSum < 0 )
/* 7*/            ThisSum = 0;
  }  /* end for-j */
/* 8*/  return MaxSum;
}

```

$$T(N) = O(N) \tag{6}$$

- 如果当前的子序列小于0，那么认为此时的子序列为0，即抛弃之前存的子序列，从这里重新开始找序列
- 如果当前的子序列比我们已存的MaxSum大，那就更新
- 如果不大，就不管，继续加
- 这个算法还挺妙的，关键是适时舍弃一些没用的数列

4 Logarithms in the Running Time

4.1 Example Binary Search

Given: $A[0] \le A[1] \le \ldots\ldots \le A[N-1]$; **X**

Task: Find **X**

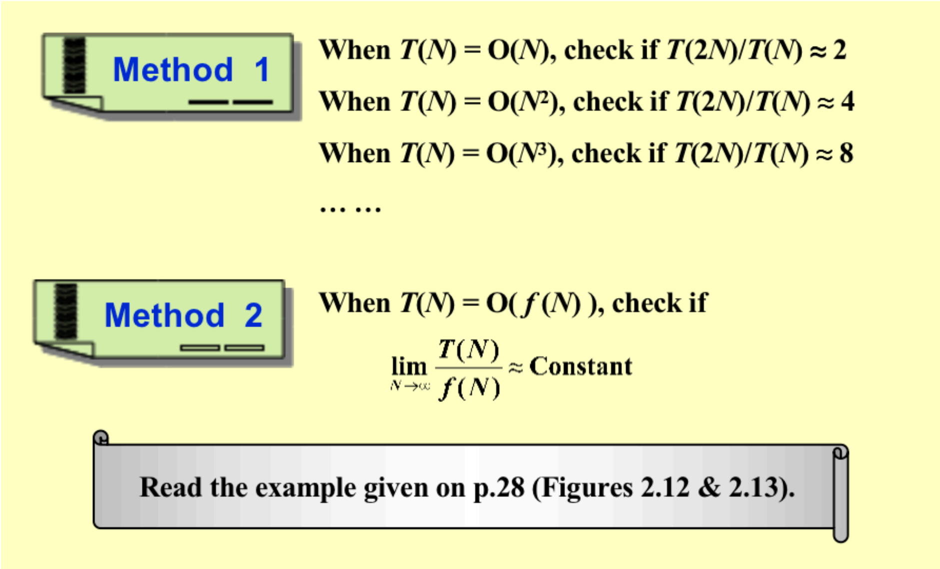
Output: i if **X** $==$ $A[i]$

-1 if **X** is not found

```
int BinarySearch ( const ElementType  A[ ], ElementType  X,  int  N )
{
        int  Low, Mid, High;
/* 1*/  Low = 0;  High = N - 1;
/* 2*/  while ( Low <= High ) {
/* 3*/        Mid = ( Low + High ) / 2;
/* 4*/        if ( A[ Mid ] < X )
/* 5*/            Low = Mid + 1;
              else
/* 6*/            if ( A[ Mid ] > X )
/* 7*/                High = Mid - 1;
                  else
/* 8*/                return  Mid; /* Found */
              }  /* end while */
/* 9*/  return  NotFound; /* NotFound is defined as -1 */
}
```

- 自学recursion的形式

5 Checking your Analysis



6 POINT

- LOOPs
- iteration