

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

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

[[Example]] Matrix addition

```

void add ( int a[ ][ MAX_SIZE ],
           int b[ ][ MAX_SIZE ],
           int c[ ][ MAX_SIZE ],
           int rows, int cols )
{
    int i, j ;
    for ( i = 0; i < rows; i++ ) /* rows + 1 */
        for ( j = 0; j < cols; j++ ) /* rows(cols+1) */
            c[ i ][ j ] = a[ i ][ j ] + b[ i ][ j ]; /* rows · cols */
}

```

Q: What shall we do
if rows >> cols?

$$T(\text{rows}, \text{cols}) = 2 \text{ rows} \cdot \text{cols} + 2\text{rows} + 1$$

4/15

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

[[Example]] Iterative function for summing a list of numbers

$$T_{\text{sum}}(n) = 2n + 3$$

```

float sum ( float list[ ], int n )
{ /* add a list of numbers */
    float tempsum = 0; /* count = 1 */
    int i ;
    for ( i = 0; i < n; i++ )
        /* count ++ */
        tempsum += list [ i ]; /* count ++ */
    /* count ++ for last execution of for */
    return tempsum; /* count ++ */
}

```

[[Example]] Recursive function for summing a list of numbers

...
takes more time

```

float rsum ( float list[ ], int n )
{ /* add a list of numbers */
    if ( n ) /* count ++ */
        return rsum(list, n-1) + list[n - 1];
    /* count ++ */
    return 0; /* count ++ */
}

```

5/15

- 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

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

§ 2 Asymptotic Notation (O, Ω, Θ, o)

The point of counting the steps is to **predict the growth** in run time as the N change, and thereby **compare the time complexities of two programs**.
So what we really want to know is the **asymptotic behavior** of T_p .

Suppose $T_{p1}(N) = c_1N^2 + c_2N$ and $T_{p2}(N) = c_3N$.
Which one is faster?

No matter what c_1, c_2 , and c_3 are, there will be an n_0 such that $T_{p1}(N) > T_{p2}(N)$ for all $N > n_0$.

I see! So as long as I know that T_{p1} is **about** N^2 and T_{p2} is **about** N , then for **sufficiently large** N , P2 will be faster!

7/15

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

§ 2 Asymptotic Notation

【Definition】 $T(N) = O(f(N))$ if there are positive constants c and n_0 such that $T(N) \leq c \cdot f(N)$ for all $N \geq n_0$.

【Definition】 $T(N) = \Omega(g(N))$ if there are positive constants c and n_0 such that $T(N) \geq c \cdot g(N)$ for all $N \geq n_0$.

【Definition】 $T(N) = \Theta(h(N))$ if and only if $T(N) = O(h(N))$ and $T(N) = \Omega(h(N))$.

【Definition】 $T(N) = o(p(N))$ if $T(N) = O(p(N))$ and $T(N) \neq \Theta(p(N))$.

Note:


- $2N + 3 = O(N) = O(N^{k \geq 1}) = O(2^N) = \dots$ We shall always take the **smallest** $f(N)$.
- $2^N + N^2 = \Omega(2^N) = \Omega(N^2) = \Omega(N) = \Omega(1) = \dots$ We shall always take the **largest** $g(N)$.

8/15

- 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。
- Θ : $O(f(N)) = \Omega(g(N))$, 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

§ 2 Asymptotic Notation



Rules of Asymptotic Notation

☞ If $T_1(N) = O(f(N))$ and $T_2(N) = O(g(N))$, then

(a) $T_1(N) + T_2(N) = \max(O(f(N)), O(g(N)))$,

(b) $T_1(N) * T_2(N) = O(f(N) * g(N))$.

☞ If $T(N)$ is a polynomial of degree k , then $T(N) = \Theta(N^k)$.

☞ $\log^k N = O(N)$ for any constant k . This tells us that **logarithms grow very slowly**.

Note: When compare the complexities of two programs asymptotically, make sure that N is **sufficiently large**.

For example, suppose that $T_{p1}(N) = 10^6 N$ and $T_{p2}(N) = N^2$. Although it seems that $\Theta(N^2)$ grows faster than $\Theta(N)$, but if $N < 10^6$, P2 is still faster than P1.

9/15

- 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

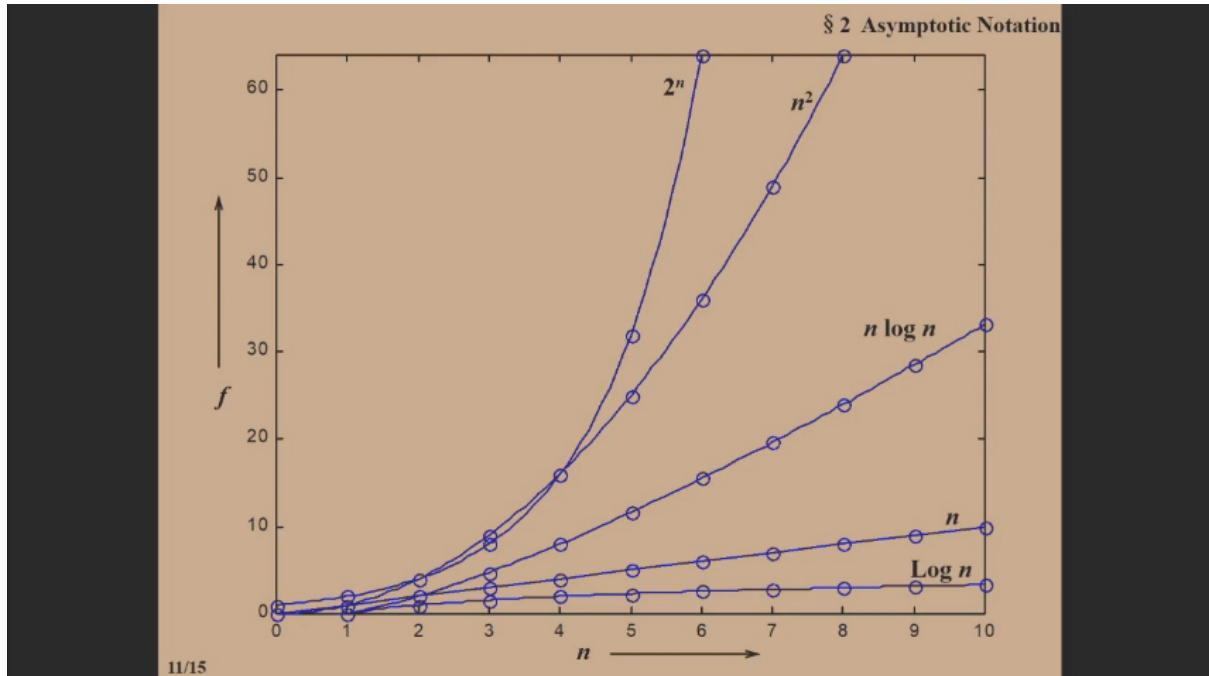
1.4.3 Description

§ 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	40320	2092278988800	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

§ 2 Asymptotic Notation

 **General Rules**

- ☞ **FOR LOOPS:** The running time of a for loop is at most the running time of the **statements inside** the for loop (including tests) **times** the number of **iterations**.
- ☞ **NESTED FOR LOOPS:** The total running time of a statement inside a group of nested loops is the running time of the **statements multiplied** by the **product of the sizes** of all the for loops.
- ☞ **CONSECUTIVE STATEMENTS:** These just **add** (which means that the **maximum** is the one that counts).
- ☞ **IF / ELSE:** For the fragment


```
if ( Condition ) S1;
else S2;
```

 the running time is never more than the running time of the **test plus** the **larger** of the running time of S1 and S2.

14/15

- 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

§ 2 Asymptotic Notation

RECURSIONS:

[[Example]] Fibonacci number:
 $\text{Fib}(0) = \text{Fib}(1) = 1, \text{Fib}(n) = \text{Fib}(n-1) + \text{Fib}(n-2)$

```

long int Fib ( int N ) /* T ( N ) */
{
    if ( N <= 1 ) /* O ( 1 ) */
        return 1; /* O ( 1 ) */
    else
        return Fib( N - 1 ) + Fib( N - 2 );
} /* O(1) */ /* T(N-1) */ /* T(N-2) */

```

$T(N) = T(N-1) + T(N-2) + 2 \geq \text{Fib}(N)$

Proof by induction

15/15

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

2 HOMEWORK

2.1 Nested Loops

```

1  if ( A > B ){
2      for ( i=0; i<N*2; i++ )
3          for ( j=N*N; j>i; j-- )
4              C += A;
5  }
6  else {
7      for ( i=0; i<N*N/100; i++ )
8          for ( j=N; j>i; j-- )
9              for ( k=0; k<N*3; k++)
10                 C += B;
11 }

```

- 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 A_1, A_2, \dots, A_N , find the maximum value of 求最大子序列

3.1.1 Algorithm 1

```

1  int  MaxSubsequenceSum ( const int A[ ], int N )
2  {
3      int  ThisSum, MaxSum, i, j, k;
4      /* 1*/  MaxSum = 0;  /* initialize the maximum sum */
5      /* 2*/  for( i = 0; i < N; i++ )  /* start from A[ i ] */
6      /* 3*/      for( j = i; j < N; j++ ) {  /* end at A[ j ] */
7      /* 4*/          ThisSum = 0;
8      /* 5*/          for( k = i; k <= j; k++ )
9      /* 6*/              ThisSum += A[ k ];  /* sum from A[ i ] to A[ j ] */
10     /* 7*/              if ( ThisSum > MaxSum )
11     /* 8*/                  MaxSum = ThisSum;  /* update max sum */
12     /* 9*/          }  /* end for-j and for-i */
13     /* 10*/  return  MaxSum;
14 }

```

$$T(N) = O(N^3) \quad (3)$$

3.1.2 Algorithm 2

```

1  int  MaxSubsequenceSum ( const int A[ ], int N )
2  {
3      int  ThisSum, MaxSum, i, j;
4      /* 1*/  MaxSum = 0;  /* initialize the maximum sum */
5      /* 2*/  for( i = 0; i < N; i++ ) {  /* start from A[ i ] */
6      /* 3*/          ThisSum = 0;
7      /* 4*/          for( j = i; j < N; j++ ) {  /* end at A[ j ] */
8      /* 5*/              ThisSum += A[ j ];  /* sum from A[ i ] to A[ j ] */
9      /* 6*/              if ( ThisSum > MaxSum )
10     /* 7*/                  MaxSum = ThisSum;  /* update max sum */
11     /* 8*/          }  /* end for-j */
12     /* 9*/  }  /* end for-i */
13     /* 10*/  return  MaxSum;
14 }
15

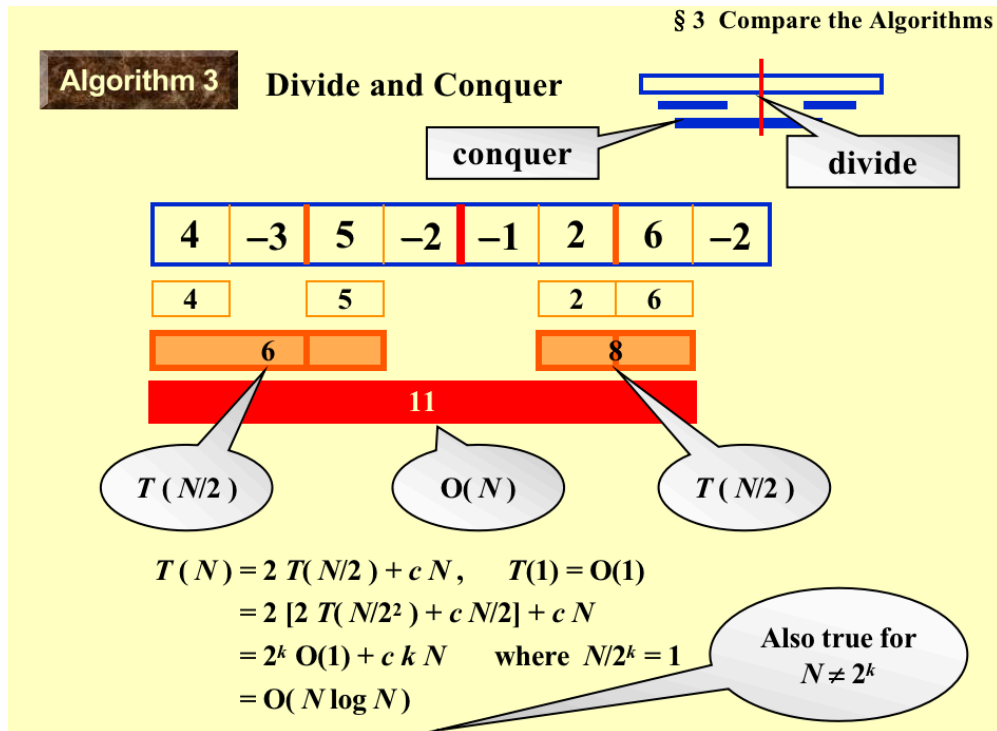
```

- 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

$$T(N) = O(N^2)$$

(4)

3.1.3 Algorithm 3: Divide and Conquer



- 最后一步证明要自己推一遍
- 出现 $\log N$ 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）
- $k = \log N$

$$T(N) = O(N \log N)$$

(5)

```

1 //求出最大子序列 4 , -3, 5, -2, -1, 2, 6, -2
2 #include <stdio.h>
3 int max (int a, int b, int c)
4 {
5     int ret;
6     if(a > b)
7     {
8         ret = a;
9     }else
10    if(a <= b)
11    {
12        ret = b;
13    }

```

```

14     if(ret >= c)
15     return ret;
16     else
17     return c;
18 }
19 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)
20 {
21     int mid = (right + left) / 2;
22     if(left == right)                                     //分治递归
要注意出口条件
23     {
24         return box[left];
25     }
26     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。
27     int rightsum = Findmaxsum(box,size,mid + 1,right);   //求出右半
区最大子序列和
28     int leftbordersum = 0;
29     int rightbordersum = 0;
30     int i;
31     int thissum = 0;
32     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, )
33     {
34         thissum += box[i];
35         if(rightbordersum < thissum)
36         {
37             rightbordersum = thissum;
38         }
39     }
40     thissum = 0;
41     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
42     {
43         thissum += box[i];
44         if(leftbordersum < thissum)
45         {
46             leftbordersum = thissum;
47         }
48     }
49     int midsum = leftbordersum + rightbordersum;          //横跨
左右半区最大子序列和
50     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

```

```

51
52
53 }
54 int main ()
55 {
56     int box[8] = {4,-3,5,-2,-1,2,5,-2};
57     int ret = 0;
58     ret = Findmaxsum(box,8,0,7);
59     printf("%d",ret);
60     return 0 ;
61 }

```

3.1.4 Algorithm 4 On-Line Algorithm

```

1  int MaxSubsequenceSum( const int  A[ ],  int  N )
2  {
3      int  ThisSum, MaxSum, j;
4      /* 1*/  ThisSum = MaxSum = 0;
5      /* 2*/  for ( j = 0; j < N; j++ ) {
6          /* 3*/      ThisSum += A[ j ];
7          /* 4*/      if ( ThisSum > MaxSum )
8              /* 5*/      MaxSum = ThisSum;
9          /* 6*/      else if ( ThisSum < 0 )
10             /* 7*/      ThisSum = 0;
11     } /* end for-j */
12     /* 8*/  return MaxSum;
13 }
14

```

$$T(N) = O(N) \quad (6)$$

- 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: $A[0] \leq A[1] \leq \dots \leq A[N-1]$; X

Task: Find X

Output: i if $X == A[i]$
 -1 if X is not found

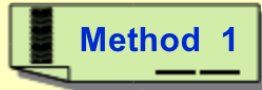
```

1  int BinarySearch ( const ElementType A[], ElementType X, int N )
2  {
3      int Low, Mid, High;
4      /* 1*/ Low = 0; High = N - 1;
5      /* 2*/ while ( Low <= High ) {
6          /* 3*/ Mid = ( Low + High ) / 2;
7          /* 4*/ if ( A[ Mid ] < X )
8              /* 5*/ Low = Mid + 1;
9          else
10             /* 6*/ if ( A[ Mid ] > X )
11                 /* 7*/ High = Mid - 1;
12             else
13                 /* 8*/ return Mid; /* Found */
14         } /* end while */
15     /* 9*/ return NotFound; /* NotFound is defined as -1 */
16 }

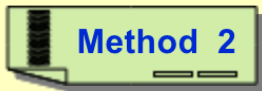
```

- 自学recursion的形式

5 Checking your Analysis



When $T(N) = O(N)$, check if $T(2N)/T(N) \approx 2$
 When $T(N) = O(N^2)$, check if $T(2N)/T(N) \approx 4$
 When $T(N) = O(N^3)$, check if $T(2N)/T(N) \approx 8$



When $T(N) = O(f(N))$, check if

$$\lim_{N \rightarrow \infty} \frac{T(N)}{f(N)} \approx \text{Constant}$$

Read the example given on p.28 (Figures 2.12 & 2.13).

6 POINT

- **LOOPS**
- **iteration**