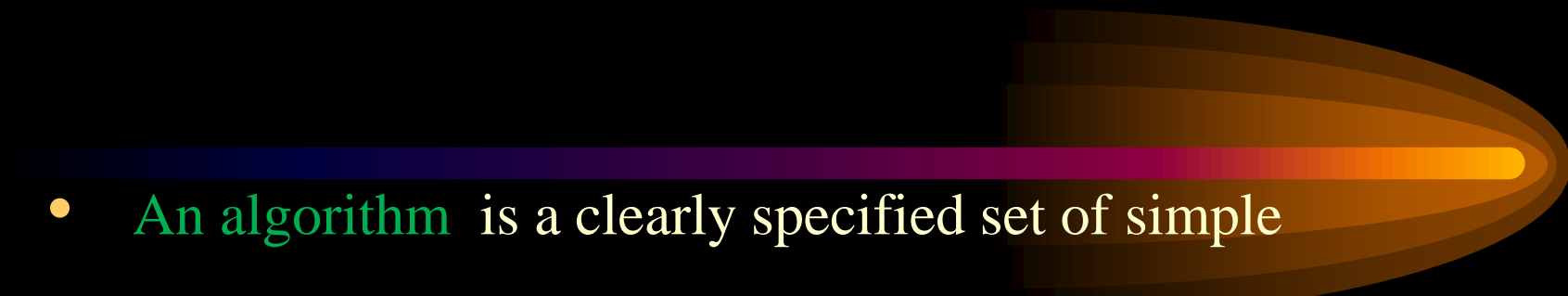




Chapter 2

Algorithm Analysis

- 
- **An algorithm** is a clearly specified set of simple instructions to be followed to solve a problem.
 - **Algorithm analysis** is the amount of computer memory and time needed to run a program (algorithm)

We use two approaches to determine it:

{ performance analysis
performance measurement

Space complexity: the amount of memory a
program needs to run to completion

Time complexity: the amount of time a
program needs to run to completion

2.1 Space Complexity



1) components:

- **instruction space**
- **data space** (space needed for constants, simple variables, component variables)
- **environment stack space** (to save information needed to resume execution of partially completed functions)

2.1 Space Complexity



two parts:

a **fixed part**—include space for instructions, simple variables, fixed-size component variables, constants

a **variable part**—include space for component variables, dynamical allocated space, recursion stack

2.1 Space Complexity

2) example: 精细分析.

例 \ Sequential Search 顺序查找算法

```
public static int SequentialSearch( int [ ] a , int x )
```

```
{ int i;
```

无new, 无需考虑栈空间

```
for(i=0; i<a.length && a[i]!=x; i++) ;
```

```
if(i == a.length) return -1;
```

```
return i;
```

```
}
```

a, x, i 在 $0 \sim a.length$

2.1 Space Complexity

对了 不因搜索新占空间

Total data space:

~~12 bytes~~ : $x, i, a[i], 0, -1, a.length$ 占内存空间的.

~~each of them cost 2 bytes~~ 占6个字节

$S(n)=0$

关于n变化的变化量为0

个
老子. 16位系统.

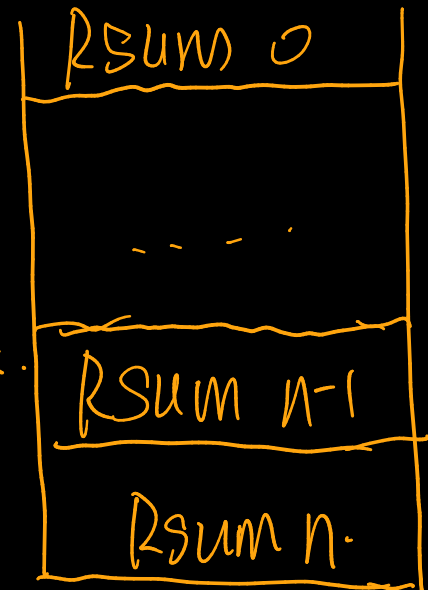
2.1 Space Complexity

例 2

- Recursive code to add $a[0:n-1]$

```
public static float Rsum(float[ ] a, int n)
{ if ( n > 0 )
    return Rsum(a, n-1) + a[n-1];
  return 0;
}
```

$n+1$ 个帧.
每个帧占 3 个字节.



a n 返回地址 ?

2.1 Space Complexity

Recursion stack space:

formal parameters : a (2 byte), n (2 byte) 0.

return address (2 byte)

3个字节.

Depth of recursion: $n+1$

$$S_{\text{Rsum}}(n) = \cancel{6(n+1)}$$

$$3(n+1)$$

随 n 线性变化

2.2 Time complexity

The time taken by a program p is $T(p)$

$$T(p) = \text{compile time} + \text{run time}$$

The compile time does not depend on the instance characteristics

The run time is denoted by t_p (instance characteristics)

1) operation counts

identify one or more key operations and determine the number of times these are performed

2.2 Time Complexity

Example 1

- finding the largest number in $a[0:n-1]$

```
public static int Max( int [ ]a, int n)
{ //locate the largest element in a[0:n-1]
  int pos=0;
  for(int i=1;i<n;i++)
    if(a[pos]<a[i]) pos=i;
  return pos;
}
```

Handwritten annotations:

- 1次 (1 time) above $i=1$
- $(n-1)$ 次 ($(n-1)$ times) above $i < n$
- 1次 (1 time) next to $pos=i$
- 关键操作 (key operation) with an arrow pointing to $a[pos] < a[i]$

compare time : $n-1$

2.2 Time Complexity

Example 2

- selection sort

0	1	2	3	4	5
21	25	49	25*	16	08
08	25	49	25*	16	21
08	16	49	25*	25	21
08	16	21	25*	25	49
08	16	21	25*	25	49
08	16	21	25*	25	49

2.2 Time Complexity

```
public static void SelectionSort( int [ ] a, int n)
```

```
{ //sort the n number in a[0:n-1].
```

```
  for(int size=n; size>1; size--)
```

```
    { int j=Max(a,size);
```

a 从 0 到 size-1 (下标) 中最大值下标.

```
      swap(a[j],a[size-1]);
```

```
    }
```

```
}
```

2.2 Time Complexity

Analysis of selection sort

1)each invocation $\text{Max}(a, \text{size})$ results in $\text{size}-1$ comparisons, so the total number of comparisons is :

$$n-1+n-2+\dots+3+2+1=(n-1)*n/2$$

2)the number of element move is $3(n-1)$

2.2 Time Complexity

Example 3

- **bubble sort**

1 4 5 2 3 6 7 8 9
8 25 32 15 20 38 46 54 67

```
public static void Bubble( int [ ] a , int n)
{ //Bubble largest element in a[0:n-1] to right
  for(int i=0; i<n-1; i++)
    if(a[i]>a[i+1])swap(a[i],a[i+1]);
}
```

8	25	32	15	20	38	46	54	67
8	25	32	15	20	38	46	67	54
8	25	32	15	20	38	67	46	54
8	25	32	15	20	67	38	46	54
8	25	32	15	67	20	38	46	54
8	25	32	67	15	20	38	46	54
8	25	67	32	15	20	38	46	54
8	67	25	32	15	20	38	46	54

2.2 Time Complexity

```
public static void BubbleSort( int [ ] a, int n)
{ //Sort a[0:n-1] using a bubble sort
  for(int i=n ;i>1; i--)
    Bubble(a,i);
}
```


2.2 Time Complexity



Analysis of bubble sort

the number of element comparisons is
 $(n-1)*n/2$, as for selection sort

2.2 Time Complexity

Example 4

- Rank sort 基数排序

r: 0 2 1 4 3

0 1 2 3 4

a: 8 25 16 30 28

```
public static void Rank( int [ ] a, int n, int [ ] r)
```

```
{//Rank the n elements a[0:n-1]
```

```
  for(int i=0;i<n;i++)
```

```
    r[i]=0;
```

```
  for(int i=1;i<n;i++)
```

```
    for(int j=0;j<i;j++)
```

```
      if(a[j]<=a[i]) r[i]++;
```

```
      else r[j]++;
```

```
}
```

记录比它大的元素的个数

2.2 Time Complexity

```
public static void Rearrange( int [ ]a, int n, int[ ] r)
```

```
{//In-place rearrangement into sorted order
```

```
    for(int i=0;i<n;i++)
```

```
        while(r[i]!=i)
```

```
        {   int t=r[i];
```

```
            swap(a[i],a[t]);
```

```
            swap(r[i],r[t]);
```

```
        }
```

```
}
```

每次把 t 位排对.

但 i 位不一定对.

while 一共有 n 次.

2.2 Time Complexity

Analysis of **rank sort**

the number of element comparison is :

$$(n-1)*n/2$$

the number of element swap is $2n$
move.

— 以数据交换 \Leftrightarrow 三次数据移动

2.2 Time Complexity

Best, Worst, and Average Operation Counts

- The average operation count is often quite difficult to determine.
- As a result, we limit our analysis to determining the best and worst counts.

2.2 Time Complexity

Example 5

- **Sequential Search** 顺序查找.

```
public static int SequentialSearch( int [ ] a, int x, int n)
{ int i;
  for(i=0;i<n&&a[i]!=x;i++) ;
  if(i==n)return-1;
  return i;
}
```

2.2 Time Complexity

Analysis of Sequential Search

- For successful searches, the best comparison count is one, the worst is n.
- The average count for a successful search is:

$$(1/n) \sum_{i=1}^n i = \underline{(n+1)/2}$$

前提：所有情况等概率

2.2 Time Complexity

Example 6

- insertion sort

插入排序

0 1 2 3 4 5 6
a: 8 3 2 5 9 1 6
3 8 2 5 9 1 6
2 3 8 5 9 1 6
2 3 5 8 9 1 6
1 2 3 5 8 9 6
1 2 3 5 6 8 9

```
public static void Insert( int [ ] a , int n, int x)
```

```
{//Insert x into the sorted array a[0:n-1]
```

```
    int i;
```

```
    for(i=n-1; i>=0&& x<a[i]; i--)
```

```
        a[i+1]=a[i];
```

```
        a[i+1]=x;
```

```
}
```

从后向前比较.

2.2 Time Complexity

```
public static void InsertionSort( int [ ] a, int n)
{   for(int i=0; i<n; i++)
    {   int t = a[i];
        Insert(a,i,t);
    }
}
```

2.2 Time Complexity

Another version of insertion sort 升序-插入.

```
public static void InsertionSort( int [ ]a, int n)
```

```
{ for(int i=0;i<n;i++)  
  { //insert a[i] into a[0:n-1]  
    int t=a[i];  
    int j;  
    for(j=i-1; j>=0&& t<a[j]; j--)  
      a[j+1]=a[j];  
    a[j+1]=t;  
  }  
}
```

$i = 1$ $l = 0 \dots i$
 $j = 0$
 $a[1] < a[0]$

2.2 Time Complexity

Analysis of **insertion sort**

both version perform the same number of comparisons.

the best case is

n-1 (比较)

(move) number is

$$2*(n-1)$$

the worst case is

$$(n-1)*n/2, \quad \text{即 } 1+2+\dots+n-1$$

move number is

$$(1+2)+(2+2)+\dots+(n-2+2)+(n-1+2) = \\ n*(n-1)/2 + 2*(n-1) = (n^2 + 3n - 4)/2$$



2.2 Time Complexity



2) Step counts

- to account for the time spent in all parts of the program/function.
- we create a global variable **count** to determine the number of steps

2.2 Time Complexity

Counting step

```
public static Comparable Sum( Comparable[ ] a, int n)
{
    Comparable tsum = 0 ;
    count++;
    for (int i = 0 ; i < n ; i++)
    {
        count++;
        tsum += a[i] ;
        count++;
    }
    count++;
    count++;
    return tsum;
}
```

2n+3 step

for 最后一次多余的判断

2.2 Time Complexity

渐近表示法

3). Asymptotic Notation(O , Ω , θ , o):

describes the behavior of the time or space complexity for large instance characteristics.

2.2 Time Complexity

I)Big Oh Notation(O): provide a upper bound for the function f .

Definition:

$f(n)=O(g(n))$ iff positive constant c and n_0 exist such that $f(n)\leq cg(n)$ for all n , $n\geq n_0$

For example:

linear function $f(n)=3n+2$. when $n\geq 2$, $3n+2\leq 3n+n=4n$, so $f(n)=O(n)$;

Quadratic function $O(n^2)$, exponential function $O(2^n)$, constant function $O(c)$

2.2 Time Complexity

Example 1: Selection sort

	Frequency
a[0],a[1],...,a[n-1]	n
for (int i=0 ; i < n-1 ; i++)	n-1
{ int k= i;	(n ² +n-2)/2
for (int j=i+1; j<n ; j++)	<=(n ² -n)/2
if (a[j]<a[k]) k=j;	3(n-1)
int temp=a[i]; a[i]=a[k]; a[k]=temp;	
}	

$$T(n) = n^2 + 5n - 5$$

$$n \rightarrow \infty \quad T(n)/n^2 \rightarrow \text{constants}(1)$$

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

2.2 Time Complexity

Example 2: Binary Search 折半查找

0 1 2 3 4 5 6 7 8

a: -1 0 1 3 4 6 8 10 12 x = 6

```
public static int binarySearch( Comparable [ ] a, Comparable x )
{   int low = 0, high = a.length - 1;
    while( low <= high )
    {   int mid = ( low + high ) / 2;
        if( a[ mid ].compareTo( x ) < 0 )
            low = mid + 1;
        else if( a[mid ].compareTo( x ) > 0 )
            high = mid - 1;
        else return mid;
    }
    return NOT-FOUND;
}
```

最好: 一次 最坏: $\log_2 n$ 平均 $O(\log_2 n)$

2.2 Time Complexity

Example 3: MAXIMUM SUBSEQUENCE SUM PROBLEM 最大子序列和

给定整数 a_1, a_2, \dots, a_n (可能有负数), 求 $\sum_{k=1}^j a_k$ 的最大值。

如果所有整数均为负数, 则最大子序列和为0。

1 2 3 4 5 6
a: -2 11 -4 13 -5 -2

Algorithm 1: merely exhaustively tries all possibilities 穷举

```
public static int maxSubSum1( int [ ] a )  
{ int maxSum = 0;  
  for ( int i = 0; i < a.length; i++ ) 头  
    for ( int j = i; j < a.length; j++ ) 尾  
    { int thisSum = 0;  
      for ( int k = i; k <= j; k++ )  
        thisSum += a[k];  
      if ( thisSum > maxSum )  
        maxSum = thisSum;  
    }  
  return maxSum;  
}
```

时间复杂度

三重循环 $\rightarrow O(N^3)$

↑
要验证并降级

2.2 Time Complexity

analysis

$O(N^3)$



2.2 Time Complexity

Algorithm 2:

```
public static int maxSubSum2( int [ ] a )
```

$O(N^2)$

```
{   int maxSum = 0;
```

```
   for( int i = 0; i < a.length; i++ )
```

```
   {   int thisSum = 0;
```

```
       for ( int j = i; j < a.length; j++ )
```

```
       {   thisSum += a[j];
```

← 简化

```
           if ( thisSum > maxSum )
```

```
               maxSum = thisSum;
```

```
       }
```

```
   }
```

```
   return maxSum;
```

```
}
```

$O(N^2)$

2.2 Time Complexity

Algorithm 3: recursive and relatively complicated $O(N \log N)$

分治法 (divide-and-conquer)

分阶段：把问题分成两个大致相等的子问题，然后递归地对它们求解。

治阶段：将两个子问题的解合并到一起，可能再做些少量的附加工作，最后得到整个问题的解。

example:

4 -3 5 -2 -1 2 6 -2

$a_1 \ a_2 \ a_3 \ a_4 \ \ a_5 \ a_6 \ a_7 \ a_8$

a_1 ---- a_4 的最大子序列和为6, a_1 ---- a_3

a_5 ---- a_8 的最大子序列和为8, a_6 ---- a_7

横跨这两部分且通过中间的最大和为:

$$a_1$$
----- $a_3+a_4 \ + \ a_5+ a_6$ --- $a_7 = 11$

2.2 Time Complexity

复杂度分析

```
private static int maxSumRec( int [ ] a, int left, int right )
```

$O(1)$ {

```
    { if ( left == right )
      if ( a[ left ] > 0 )
        return a[ left ];
      else return 0;
    }
```

递归

```
    int center = ( left + right ) / 2;
    { int maxLeftSum = maxSumRec( a, left, center );
      int maxRightSum = maxSumRec( a, center + 1, right );
      int maxLeftBorderSum = 0, leftBorderSum = 0;
      for ( int i = center; i >= left; i-- )
      { leftBorderSum += a[i];
        if ( leftBorderSum > maxLeftBorderSum )
          maxLeftBorderSum = leftBorderSum;
      }
```

$\frac{N}{2}$

① 左
② 右

③ 横跨

2.2 Time Complexity

N

```

int maxRightBorderSum = 0, rightBorderSum = 0;
for ( int i = center + 1; i <= right; i++ )
{   rightBorderSum += a[ i ] ;
    if ( rightBorderSum > maxRightBorderSum )
        maxRightBorderSum = rightBorderSum;
}
return max3( maxLeftSum, maxRightSum,
             maxLeftBorderSum + maxRightBorderSum );
}
    
```

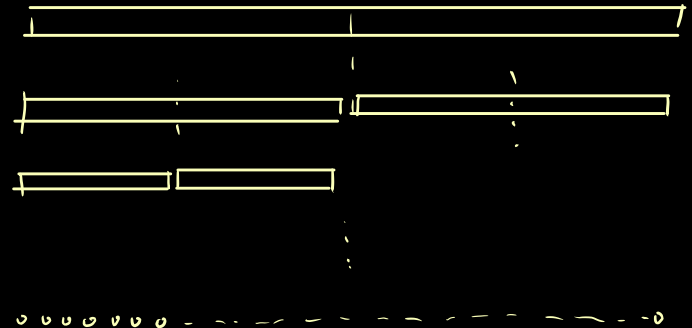
```

public static int maxSubSum3( int [ ] a )
{   return maxSumRec( a, 0, a.length - 1 );
}
    
```

analysis:

$O(N \log N)$

$\log N$



2.2 Time Complexity

Example 4: Euclid's Algorithm

欧几里德算法

计算最大公因数(greatest common divisor)。通过辗转相除法

例如 50, 15 的最大公因数是5

```
public static long gcd( long m, long n )
{ while( n != 0 )
  { long rem = m % n;
    m = n;
    n = rem;
  }
  return m;
}
```

$O(\log N)$

△ 时间复杂度分析

$m_1, m_2, m_3, \dots, m_k, 0$
5 3 2 1 1 0

$m_i = a m_{i+1} + m_{i+2}$

a 取1时 m_i 序列“ \leftarrow ”方向增长最慢。

$\Rightarrow m_i$ 变为斐波那契数列

n 在序列中的下标大概是 $\log N$

2.2 Time Complexity

II).Omega Notation (Ω): is the lower bound
analog of the big Oh notation,permits us to bound
the value f from below.

Definition:

$f(n) = \Omega(g(n))$ iff positive constant c and n_0 exist
such that $f(n) \geq cg(n)$ for all n , $n \geq n_0$

2.2 Time Complexity

III). Theta Notation(θ): is used when the function f can be bounded both from above and below by the same function g .

Definition :

$f(n) = \theta(g(n))$ iff positive constants c_1 and c_2 and a n_0 exist such that $c_1 g(n) \leq f(n) \leq c_2 g(n)$ for all n , $n \geq n_0$

space and time complexity: use Big Oh Notation

(IV) 小 O 表示法
 $O(N^2)$ 平方小于 N^2

Chapter 2

Exercises:

1. Find the complexity of the function used to find the kth smallest integer in an unordered array of integers

```
int selectkth ( int a[], int k, int n){
    int i, j, mini, temp;
    for ( i = 0; i < k; i++){
        mini = i;
        for ( j = i+1; j < n; j++){
            if ( a[j] < a[mini])
                mini = j;
        }
        tmp = a[i];
        a[i] = a[mini];
        a[mini] = tmp;
    }
    return a[k-1];
}
```

Chapter 2

答案为什么不算空间复杂度?

2. Find the computational complexity for the following four loops:

c. for (cnt3=0, i=1; i<=n; i*=2)

for (j=1; j<=n; j++)

cnt3++;

d. for (cnt4=0, i=1; i<=n; i*=2)

for (j=1; j<=i; j++)

cnt4++;

Chapter 2

3. For each of the following two program fragments:

Give an analysis of the running time(Big-Oh will do)

1) `sum = 0;`

`for(i = 0; i < n; i++)`

`for(j = 0; j < i*i; j++)`

`for(k = 0; k < j; k++)`

`sum++;`

2) `sum = 0;`

`for(i = 1; i < n; i++)`

`for(j = 0; j < i*i; j++)`

`if(j % i == 0)`

`for(k = 0; k < j; k++)`

`sum++;`

Chapter 2

4. 设 n 为正整数，分析下列各程序段中加下划线的语句的执行次数。

```
1) for (int i = 1; i <= n; i++)  
    for (int j = 1; j <= n; j++)  
    {   c[i][j] = 0.0;  
        for (int k = 1; k <= n; k++)  
            c[i][j] = c[i][j] + a[i][k] * b[k][j];  
    }
```

```
2) x = 0; y = 0;  
   for (int i = 1; i <= n; i++)  
       for (int j = 1; j <= i; j++)  
           for (int k = 1; k <= j; k++)  
               x = x + y;
```

```
3) int x = 91; int y = 100;  
   while(y > 0)  
   {   if(x > 100) { x -= 10; y--; }  
       else x++;  
   }
```