第2章 算法分析基础（3学时）

主讲：张春元（学院306室）
联系电话：13876004640
课程邮箱：haidasasj@126.com
密码：zhangchunyuan

本章主要内容

- 2.1 算法的时间复杂性分析
- 2.2 算法的空间复杂性分析
- 2.3 最优算法

2.1 算法的时间复杂性分析

❖ 算法的复杂性

* 算法的核心和灵魂是效率，也就是求解速度，与算法的复杂性存在密切关系
* 计算机中最重要的两种资源是时间和空间资源。更确切地说，算法的复杂性是算法运行所需要的计算机资源的量，需要的时间资源的量称为时间复杂性；需要的空间资源的量称为空间复杂性

2.1 算法的时间复杂性分析

❖ 算法时间复杂性分析

* 算法时间复杂性分析是一种事前分析估算的方法，对算法所消耗资源的一种渐进分析方法。
  - 渐进分析：忽略具体机器、编程语言和编译器的影响，只关注在输入规模增大时算法运行时间的增长趋势，从数量级的角度评价算法的效率，即：

$$T=T(n)$$

其中：n为算法输入规模

2.1.1 输入规模与基础语句

❖ 输入规模

* 输入规模：指输入量的多少

❖ 基础语句

* 精确地表示算法的运行时间函数T(n)常常是很困难的，考虑到算法分析的主要目的在于比较求解同一个问题的不同算法效率，为精简客观地反映一个算法的运行时间，用算法中基本语句的执行次数来度量算法的工作量。
* 基本语句：执行次数与整个算法的执行次数成正比的语句，对 算法运行时间贡献最大，是算法中最重要的操作。

2.1.1 输入规模与基础语句

- 例2.1 对如下顺序查找算法，请找出输入规模和基本语句

```
int SeqSearch(int A[], int n, int k){
    for(int i=0; i<n; i++)
        if(A[i]==k) break;
    if(i==n) return 0;
    else return (i+1);
}
```

2.1.1 输入规模与基础语句

❖ 例2.1 对如下顺序查找算法，请找出输入规模和基本语句

```
int SeqSearch(int A[], int n, int k){
    for(int i=0; i<n; i++)
        if(A[i]==k) break;
    if(i==n) return 0;
    else return (i+1);
}
```

2.1.1 输入规模与基础语句

- 例2.2 对如下冒泡排序算法，请找出输入规模和基本语句

```
void SubbleSort(int r[], int n){
    int bound, exchange=n-1;
    while(exchange!=0){
        bound=exchange;  exchange=0;
        for(int j=0; j<bound; j++)
            if( r[j]>r[j+1]){
                int temp=r[j]; r[j]=r[j+1]; r[j+1]=temp;
                exchange=j;
            }
    }
}
```

2.1.1 输入规模与基础语句

- 例2.2 对如下冒泡排序算法，请找出输入规模和基本语句

```
void SubbleSort(int r[], int n){
    int bound, exchange=n-1;
    while(exchange!=0){
        bound=exchange;  exchange=0;
        for(int j=0; j<bound; j++)
            if(  r[j]>r[j+1] ){
                int temp=r[j]; r[j]=r[j+1]; r[j+1]=temp;
                exchange=j;
            }
    }
}
```

2.1.1 输入规模与基础语句

- 例2.3 下列算法实现将两个升序序列合并成一个升序序列，请找出输入规模和基本语句

```
void Union(int A[], int n, int B[], int m, int C[]){
    int i=0,j=0,k=0;
    while(i<n && j<m){
        if(A[i]<=B[j])   C[k++]=A[i++];
        else C[k++]=B[j++];
    }
    while( i<n)  C[k++]=A[i++];
    while( j<m) C[k++]=B[j++];
}
```

2.1.1 输入规模与基础语句

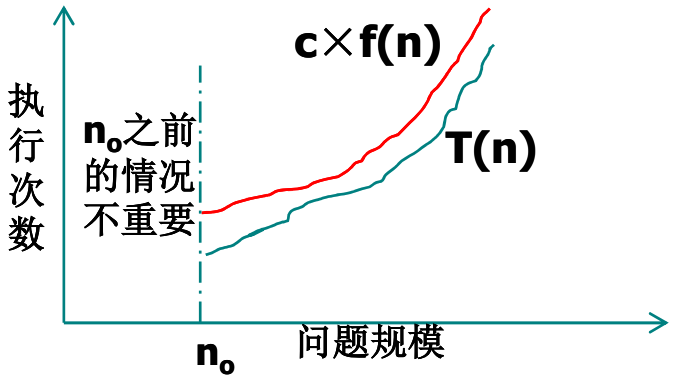
- 例2.3 下列算法实现将两个升序序列合并成一个升序序列，请找出输入规模和基本语句

```
void Union(int A[], int n, int B[], int m, int C[]){
    int i=0,j=0,k=0;
    while(i<n && j<m){
        if(A[i]<=B[j])   C[k++]=A[i++];
        else C[k++]=B[j++];
    }
    while( i<n)  C[k++]=A[i++];
    while( j<m) C[k++]=B[j++];
}
```

2.1.2 算法的渐进分析

❖ 算法的渐进分析

* 算法的渐进分析并不是度量算法的时间量，而是度量算法运行时间的增长趋势。换言之，只考察当输入规模充分大时，算法中基本语句的执行次数在渐近意义下的上界，常用大（读欧）O表示。
* 定义2.1 若存在两个正常数c和n_0，对于任意$n \geq n_0$，都有$T(n) \leq c \times f(n)$，就称函数$T(n) = O(f(n))$。
* 注：大O符号描述增长率的上限，表示T(n)的增长最多像f(n)增长的那样快。



2.1.2 算法的渐进分析

❖ 例2.4 分析例2.3中合并算法的时间复杂性

解：(1) 假设退出第1个循环后i＝n, j=m’，说明序列A处理完毕，第2个循环将不执行，第3个循环执行。算法的时间复杂性为：

$$O(n+m'+m-m')=O(n+m)$$

(2) 假设退出第1个循环后i＝n’, j=m，说明序列B处理完毕，第2个循环将执行，第3个循环不执行。算法的时间复杂性为：

$$O(n'+m+n-n')=O(n+m)$$

综上，算法的时间复杂性为O(n+m)

2.1.3 最好、最坏和平均情况

- 时间复杂性与输入数据的关联性
  - 有些算法的时间代价只依赖于问题的输入规模，而与输入的具体数据无关，如例2.3的合并算法。
  - 但有些算法，即使输入规模相同，如果输入数据不同，其时间代价也不相同。
    - 如果问题规模相同，时间代价与输入数据有关，则需要分析最好情况、最坏情况、平均情况。

2.1.3 最好、最坏和平均情况

❖ 例2.5 分析例2.1中顺序查找的时间复杂性

```
int SeqSearch(int A[], int n, int k){
    for(int i=0; i<n; i++)
        if(A[i]==k) break;
    if(i==n) return 0;
    else return (i+1);
}
```

最好情况：数组中第一元素就是K
最坏情况：数组中最后元素就是K
平均情况：平均比较n/2个元素

2.1.3 最好、最坏和平均情况

❖ 三种情况比较

* 最好情况：对于条件的考虑太乐观了，一般不能作为算法性能的代表；但是当最好情况出现的概率较大时，应该分析最好情况。
* 最坏情况：可以知道算法的运行时间最坏能坏到什么程度，这一点对实时系统尤为重要。
* 平均情况：常用，特别是算法要处理不同的输入时，但它要求已知输入数据是如何分布的，然后求期望，因而最坏情况分析更困难。如果不知具体分布，可以假设是等概率分布。

2.1.4 非递归算法的时间复杂性分析

- 从算法递归调用的角度，算法可分为：递归算法和非递归算法
- 非递归算法时间复杂性分析的一般步骤
  - 1. 决定用哪个（或哪些）参数作为算法问题规模的度量
  - 2. 找出算法中的基本语句
  - 3. 检查基本语句的执行次数是否只依赖于问题规模，如果基本语句的执行次数还依赖于其它一些特性（如数据的初始分布），则需分析三种情况
  - 4. 建立基本语句执行次数的求和表达式
  - 5. 用渐进符号表示这个求和表达式

关键是步骤4、5

2.1.4 非递归算法的时间复杂性分析

❖ 例2.6 分析例2.2中冒泡排序算法的时间复杂性

解：基本语句是比较操作r[j]>r[j+1]，其执行次数与输入序列的分布有关，因而共分三种情况分析：

(1) 最好情况：记录已经是升序排列，算法只执行一次，比较n-1次，时间复杂性为O(n)

(2) 最坏情况：记录是降序排列，第1趟需将降序序列中最大的记录交换到最终位置，所以算法执行n-1趟，第i趟比较n-i次比较，则记录的比较次数为 $\sum_{i=1}^{n-1}(n-i)=\frac{n(n-1)}{2}$，时间复杂性$O(n^2)$

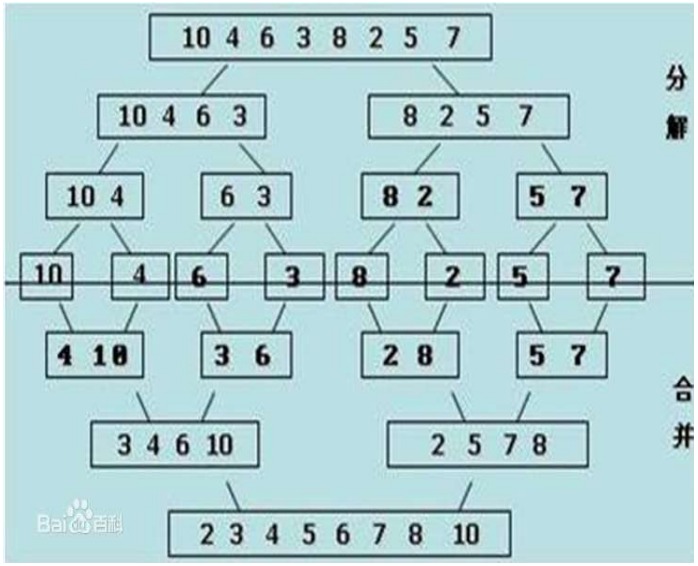
(3) 平均情况：需考虑序列中逆序的个数，过程略，时间复杂性为 $O(n^2)$

2.1.5 递归算法的时间复杂性分析

- 递归算法时间复杂性分析的关键：根据递归过程建立递推关系式，然后求解这个递推关系式。
- 常用分析技术
  - (1) 猜测技术：对递推关系式估计一个上限，然后（用数学归纳法）证明它正确。如果成功，试着收缩上限。如果失败，放松限制再试着证明。直到上限符合要求。
  - (2) 扩展递归技术：将递推关系式中等式右边的项根据递推式进行扩展，扩展后的项被再次扩展，依次下去，会得到一个求和表达式，然后就可以借助求和技术了。

2.1.5 递归算法的时间复杂性分析

- 补例1 使用猜测技术分析二路归并排序算法的时间复杂性。



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

二路归并排序运行时间递推式

2.1.5 递归算法的时间复杂性分析

❖ 解：假定$T(n)\leq n^2$，下面证明这个猜测是正确的，为了方便证明，假定$n=2k$：

(1) k=1时， $T(2)=1\leq 2^2$成立。

(2) 对所有$i\leq n$，假设$T(i)\leq i^2$，则：

$$T(n)=2T(k)+n\leq 2k^2+2k\leq 4k^2=(2k)^2=n^2$$

故得，$T(n)=O(n^2)$成立

2.1.5 递归算法的时间复杂性分析

如果将T(n)猜得更小一些，例如T(n)≤cn，证明失败。即说明上限 应在n和n^2之间。

接下来试试T(n)≤nlogn，类似地：

(1) k=1时，T(2)=1≤ 2log2成立

(2) 对所有i≤n，假设T(i)≤ ilogi ，则：

$$T(n)=2T(k)+2k\le 2k\log k+2k$$

$$=2k(\log k+1)=n\log(n)$$

故得, T(n)=O(nlogn)成立

2.1.5 递归算法的时间复杂性分析

❖ 例2.7 使用扩展递归技术分析下面递推式的时间复杂性

$$T(n)=\begin{cases}7 & n=1\\ 2T(n/2)+5n^2 & n>1\end{cases}$$

解：为了方便推导假定n=2k，则：

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

$$=2(2T(n/4)+5(n/2)^2)+5n^2$$

$$=2(2(2T(n/8)+5(n/4)^2)+5(n/2)^2)+5n^2$$

……

$$=2^kT(1)+2^{k-1}5(\frac{n}{2^{k-1}})^2+\ldots+2\times5(\frac{n}{2})^2+5n^2$$

$$T(n)=7n+5\sum_{i=0}^{k-1}\left(\frac{n}{2^i}\right)^2=7n+5n^2(2-\frac{1}{2^{k-1}})=10n^2-3n\le10n^2=O(n^2)$$

2.1.5 递归算法的时间复杂性分析

❖ 通用分治递推

* 递归算法实际上是一种分而治之的方法，通常满足如下通用分治递推式：

$$T(n)=\begin{cases} c & n=1 \\ aT(n/b)+cn^k & n>1 \end{cases}$$

其中：a,b,c和k都是常数，大小为n的原问题分成若干个大小为n/b的子问题，其中a个子问题需要求解，而cn^k是合并各个子问题的解需要的工作量。

2.1.5 递归算法的时间复杂性分析

* **定理2.1(通用分治递推定理)：**设T(n)是一个非递减函数，且满足通用分治递推式，则有如下结果成立：

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

2.1.5 递归算法的时间复杂性分析

* 证明：采用扩展递归技术对通用分治递推式进行推导，假定$n=b^m$，则：

$$T(n)=aT(\frac{n}{b})+cn^k=a(aT(\frac{n}{b^2})+c(\frac{n}{b})^k)+cn^k$$

……

$$=a^mT(1)+a^{m-1}c(\frac{n}{b^{m-1}})^k+\ldots+ac(\frac{n}{b})^k+cn^k$$

$$=c\sum_{i=0}^{m}a^{m-i}(\frac{n}{b^{m-i}})^k=c\sum_{i=0}^{m}a^{m-i}b^{ik}=ca^m\sum_{i=0}^{m}(\frac{b^k}{a})^i$$

上述求和是一个几何级数，其值依赖于比率$r=b^k/a$。

又因为$a^m=a^{\log_b n}=n^{\log_b a}$，则有以下三种情况：

2.1.5 递归算法的时间复杂性分析

$$(1) r<1: \sum_{i=0}^{m} r^i < \frac{1}{1-r}, \text{由于} a^m = n^{\log_b a}, \text{所以} T(n) = O(n^{\log_b a})$$

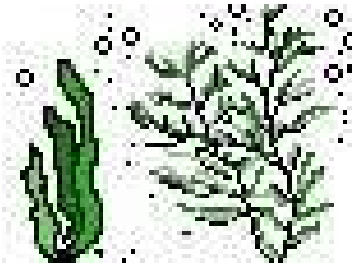
$$(2) r=1: \sum_{i=0}^{m} r^i = m+1 = \log_b n+1, \text{由于} a^m = n^{\log_b a} = n^k, \text{所以} T(n) = O(n^k \log_b n)$$

$$(3) r>1: \sum_{i=0}^{m} r^i = \frac{r^{m+1}-1}{r-1} = O(r^m), \text{ 所以,} T(n) = O(a^m r^m) = O(b^{km}) = O(n^k)$$

等比求和公式

本章主要内容

- 2.1 算法的时间复杂性分析
- 2.2 算法的空间复杂性分析
- 2.3 最优算法



2.2 算法的空间复杂性分析

- 算法在运行过程中所需的存储空间包括：
  - (1) 输入输出数据占用的空间　与待求解问题有关，与算法无关
  - (2) 算法本身占用的空间　大小一般固定
  - (3) 执行算法需要的辅助空间　是决定空间复杂性的主要因素

- 算法的空间复杂性
  - 是指算法在执行过程中需要的辅助空间数量，也就是除算法本身和输入输出数据所占用的空间外，算法临时开辟的存储空间。这个辅助空间也应该是输入规模的函数，即 $S(n)=O(f(n))$

2.2 算法的空间复杂性分析

表示所需空间为常量，并且与n无关

❖ 例2.8 分析例2.2中起泡排序算法的空间复杂性。

解：输入输出都在r[n]中，声明了3个简单变量：exchange、bound和temp。所以空间复杂性为O(1)

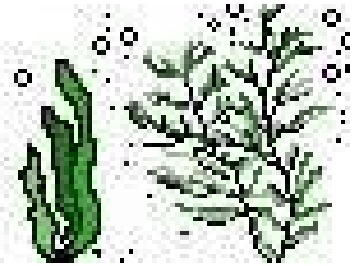
注：如果算法所需的辅助空间相对于问题的输入规模来说是一个常数，我们称此算法为原地（或就地）工作。起泡排序算法属于就地性质。

❖ 例2.9 分析例2.3中合并算法的空间复杂性。

解：合并不能就地进行，需要将合并结果存入另外一个数组中。设序列A的长度为n，序列B的长度为m，则合并后的有序序列的长度为n+m，因此算法的空间复杂性为O(n+m)。

本章主要内容

- 2.1 算法的时间复杂性分析
- 2.2 算法的空间复杂性分析
- 2.3 最优算法



2.3 最优算法

- 如何评价所设计的算法是否最优？
  - 同一个问题可能会存在多种算法，是否有求解该问题的最优算法？
  - 如果能够知道一个问题的计算复杂性下界，也就是求解该问题的任何算法（包括尚未发现的算法）所需的时间下界，就可以较准确地评价各种算法的效率，确定算法还有多少改进余地。

2.3.1 问题的计算复杂性下界

❖ 计算复杂性下界

* 概念：计算复杂性下界是指求解问题所需的最少工作量，通常采用大Ω（读欧米伽）表示。
* 用途：评价所设计的算法是否最优，是否还可以进一步改进？
* 例如，已经证明基于比较的排序算法的时间下界是$\Omega(n\log_2 n)$，意味着不存在时间复杂性小于$O(n\log_2 n)$的基于比较的排序算法。

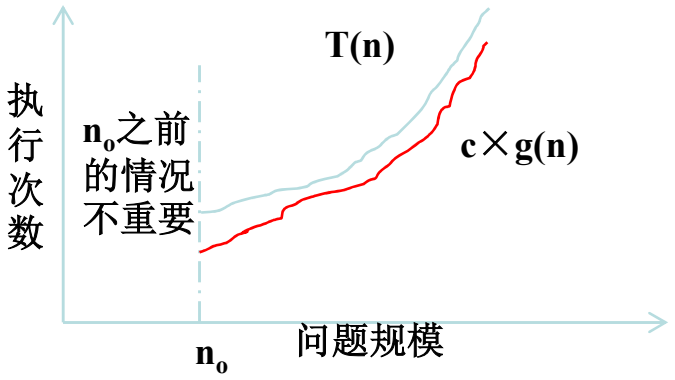
2.3.1 问题的计算复杂性下界

❖ 计算复杂性下界

* 定义2.2 若存在两个正常数c和n_0，对于任意$n \geq n_0$，都有$T(n) \geq c \times g(n)$，就称$T(n) = \Omega(g(n))$。其中：大Ω符号描述增长率的下限，表示T(n)的增长至少像g(n)增长的那样快。

* 如果能找到一个尽可能大的函数g(n)使得求解该问题的所有算法都可以在Ω(g(n))时间内完成，则称g(n)为该问题的计算复杂性下界。

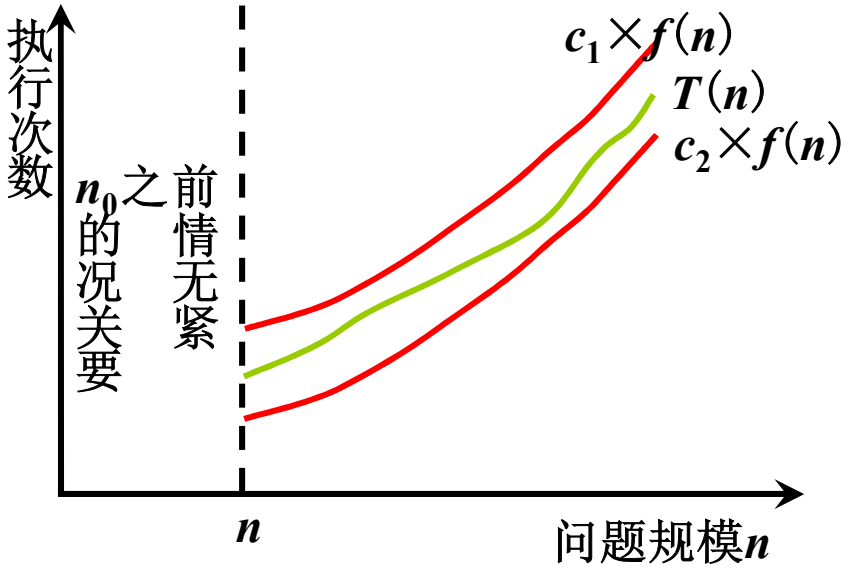
* 如果已经知道一个和下界的效率类型相同的算法，则称该下界是紧密的。



2.3.1 问题的计算复杂性下界

❖ 计算复杂性上下界（准确界）

* **定义2.3** 若存在三个正常数c_1、c_2和n_0，对于任意$n \geq n_0$，都有$c_1 \times f(n) \geq T(n) \geq c_2 \times f(n)$，就称$T(n) = \Theta(f(n))$。其中：Θ符号意味着T(n)与f(n)同阶，用来表示算法的精确阶。



2.3.1 问题的计算复杂性下界

- 例：T(n)=3n-1，上界？下界？上下界？

解：当n≥1时，3n-1≤3n=O(n)

当n≥1时，3n-1≥3n-n=2n=Ω(n)

当n≥1时，3n≥3n-1≥2n，则3n-1=Θ(n)

2.3.1 问题的计算复杂性下界

❖ 最优算法

* 大Ω常与大O配合以证明某问题的一个特定算法是该问题的最优算法，或是该问题中的某算法类中的最优算法。一般情况下，如果能证明某问题的时间下界是Ω(g(n))，那么，对以时间O(g(n))来求解该问题的任何算法（其实就是确定算法的精确阶是Θ(g(n))），都认为是求解该问题的最优算法。

2.3.1 问题的计算复杂性下界

- 例2.10 如下算法实现在一个数组中求最小值元素，证明该算法是最优算法。

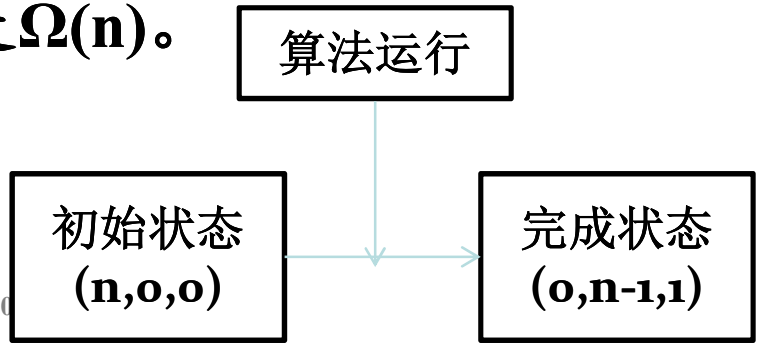
```
int arrayMin(int a[], int n){
    int min=a[0];
    for(int i=1;i<n;i++)
        if(a[i]<min) min=a[i];
    return min;
}
```

2.3.1 问题的计算复杂性下界

- 证明：算法需要进行n-1次比较，时间复杂性是O(n)。下面要证明问题的下界是Ω(n)，即对于任何n个整数，求最小值元素至少需要进行n-1次比较。

先将n个整数分为三个动态的集合：A（待比较数集合），B（非最小数集合），C（最小数集合）。任何一个通过比较求最小值元素的算法都要从(n,0,0)状态开始，最终到达(0,n-1,1)。这个过程实际上是将元素从A向B和C移动，但每次比较，至多能把一个较大元素从集合A移向集合B，因此，任何求最小值算法至少要进行n-1次比较，其时间下界是Ω(n)。

所以算法是最优算法。



2.3.2 平凡下界

- 确定和证明某个问题的计算复杂性下界是很困难的，因为不可能枚举该问题的所有算法。事实中，存在大量问题，它们的下界是不清楚的，大多数已知的下界要么是平凡的，要么是在忽略某些基本运算的意义上，应用某种计算模型（如判定树模型）推出来的
- 平凡下界（Ordinary lower bound）：使用计数方法得出的算法复杂性下界。即对问题的输入中必须要处理的元素进行计数，同时对必须要输出的元素进行计数，得到一个平凡下界。
- 例如，任何生成n个不同元素的所有排列对象的算法必定属于Ω(n!)，因为输出的规模就是n!

2.3.2 平凡下界

- 平凡下界很容易得出，但往往过小而失去意义。
- 例如，TSP问题的平凡下界是$\Omega(n^2)$（输入是n(n-1)/2个城市间的距离），但没有意义，至今没有找到一个多项式时间算法。

2.3.3 判定树模型

- 许多算法的工作方式都是对输入元素进行比较，例如排序和查找算法，故可以用判定树来研究这些算法的时间性能。
- 判定树就是满足如下条件的二叉树：
  - （1）每个内部结点对应形如x≤y的比较，根据关系成立与否，控制转移到左右子树；
  - （2）每个叶子结点表示问题的一个结果。从根到叶子结点所走过的路径代表算法执行的过程，路径的长度即代表算法执行的时间。

2.3.3 判定树模型

❖ 例2.11 用判定树模型求解$\{a_1,a_2,a_3\}$排序的时间下界。

解：根据排序算法中的比较，建立起判定树来描述算法，则判定树的高度就是算法的下界。

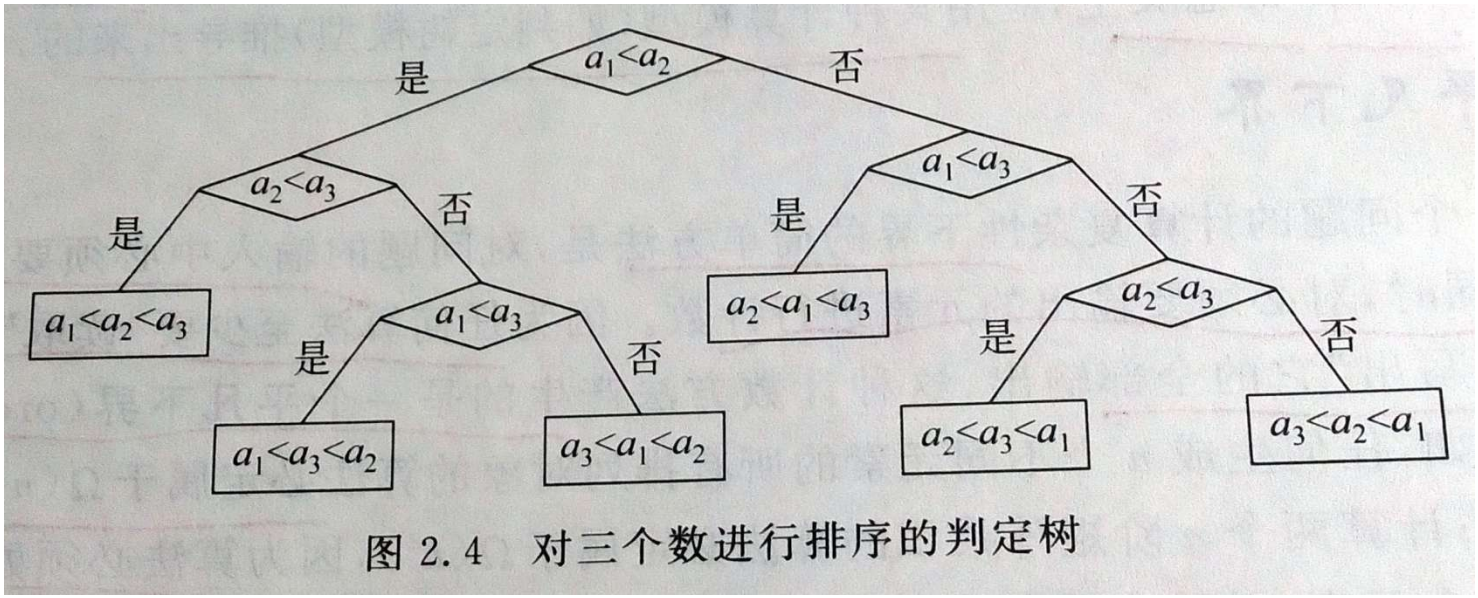


图 2.4 对三个数进行排序的判定树

补充材料：算法的实验分析

- 渐进分析能够在数量级上对算法进行精确度量，但数学不是万能的，许多貌似简单的算法很难用数学的精确性和严格性来分析，尤其在做平均效率分析时。
- 算法的实验分析是一种事后计算的方法，通常需要将算法转换为对应的程序并上机运行。例如可在程序中设置计数器变量来记录基本语句的执行次数。

补充材料：算法的实验分析

冒泡程序
计数分析

```
void SubbleSort(int r[], int n){
    int count1,count2=0;
    int bound,exchange=n-1;
    while(exchange!=0){                    //当上一趟排序有记录交换时
        bound=exchange;exchange=0;
        for(int j=0; j<bound; j++)
            if( ++count1&&r[j]>r[j+1]){
                int temp=r[j]; r[j]=r[j+1]; r[j+1]=temp;
                count2+=3;
                exchange=j;                //记载每一次记录交换的位置
            }
    }
    cout<<"比较次数是"<<count1<<endl;
    cout<<"移动次数是"<<count2<<endl;
}
```

补充材料：算法的实验分析

❖ 算法的实验分析一般步骤

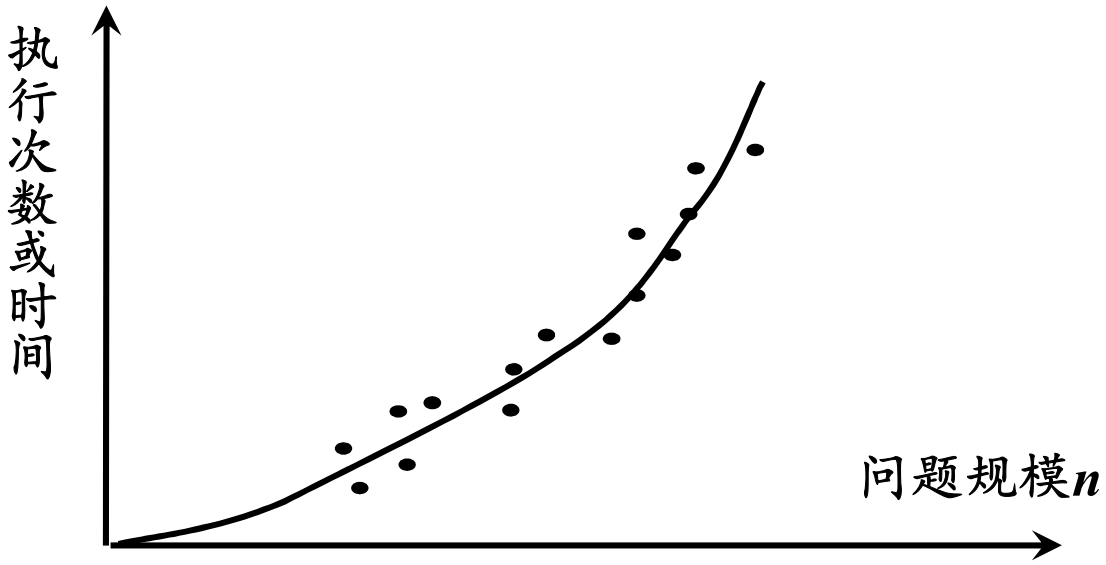
* 1. 明确实验目的（验证正确性？比较效率？）
* 2. 两种度量算法效率的方法：
  - 计数法。插入计数器度量基本语句执行次数。
  - 计时法。通过程序段开始和结束时的系统时间差来度量程序段的运行时间，但要注意分时系统程序运行时间的失真，另外输入和输出的时间不应考虑在内。
* 3. 决定输入样本。经典问题可使用研究人员制定的实例为测试基准，但大多数情况下要自己生成实验数据。
* 4. 对输入样本运行算法对应的程序，记录得到的实验数据，通常用表格或散点图记录实验数据。
* 5. 分析得到的实验数据，得出具体算法效率的有关结论

补充材料：算法的实验分析

- 表格法记录实验数据

1000	2000	3000	4000	5000	6000	7000	8000	9000
11966	24303	39992	53010	67272	78692	91274	113063	129799

- 散点图记录实验数据



补充材料：算法的实验分析

- 渐进分析和实验分析的基本区别
  - 渐进分析不依赖于特定输入，缺点是适用性不强，尤其对算法做平均性能分析时。
  - 实验分析能够适用于任何算法，缺点是其结论依赖于实验中使用的特定输入和特定的计算机系统。
  - 实际应用中，可采用渐进分析和实验分析相结合，首先在理论上描述算法的运行效率，然后针对特定计算机或程序根据实验分析确定其中一些必要的参数。

本章作业

- **P28-29 1-7题**
  - 返校后第一课上交前作业
  - 最好做在5毛钱的本子上

算法改变世界，本章结束！