

算法设计与分析

2023. 9

第1章 算法引论

1.1 算法的概念

- 1.1.1 什么是算法
- 1.1.2 算法和数据结构
- 1.1.3 算法设计的基本步骤

1.2 算法复杂度分析

- 1.2.1 算法时间复杂度分析 ★
- 1.2.2 算法空间复杂度分析

近年来的CSP中涉及算法复杂度的试题

2021csp-j初赛试题

● 单选题

25. 设输入字符串长度为 n , `decode` 函数的时间复杂度为 ()。

- A. $\Theta(\sqrt{n})$ B. $\Theta(n)$ C. $\Theta(n \log n)$ D. $\Theta(n^2)$

31. `init` 函数的时间复杂度为 ()。

- A. $\Theta(n)$ B. $\Theta(n \log n)$ C. $\Theta(n\sqrt{n})$ D. $\Theta(n^2)$

2021-CSP-S(提高组)认证第一轮试题

12. 斐波那契数列的定义为: $F_1=1$, $F_2=1$, $F_n=F_{n-1}+F_{n-2}$ ($n \geq 3$)。现在用如下程序来计算斐波那契数列的第 n 项, 其时间复杂度为 ()。

`F(n):`

`if $n \leq 2$ return 1`

`else return $F(n-1) + F(n-2)$`

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

1.1 算法的概念

1.1.1 什么是算法

算法是求解问题的一系列计算步骤，用来将输入数据转换成输出结果：



如果一个算法对其每一个输入实例，都能输出正确的结果并停止，则称它是正确的。



程序=数据结构+算法

描述数据的
类型、
组织形式

描述对数
据的操作
步骤

著名计算机科学家沃思
(Niklaus Wirth)
(1934 ~)

瑞士学者
Pascal之父
图灵奖获得者

算法具有以下几个重要**特征**：

- **输入性**：必须有0个或多个输入（待处理信息）；
- **输出性**：应有一个或多个输出（已处理信息）；
- **确定性**：组成算法的每条指令是清晰的、无歧义的。
- **可行性**：每一基本操作都可实现，且在常数时间内完成。
- **有限性**：算法中每条指令的执行次数有限，执行每条指令的时间也有限（一个算法无论在什么情况下都应在执行有穷步后结束）。



【例1】 有下列两段描述：

描述1：

```
void exam1 ()  
{   int n;  
    n=2;  
    while (n%2==0)  
    {  
        n=n+2;  
    }  
    printf ("%d\n", n);  
}
```

死循环！

描述2：

```
void exam2 ()  
{   int x, y;  
    y=0;  
    x=5/y;  
    printf ("%d, %d\n", x, y);  
}
```

除0！

这两段描述均不能满足算法的特征，试问它们违反了算法的哪些特征？

解：（1）是一个死循环，违反了算法的有限性特征。（2）出现除零错误，违反了算法的可行性特征。

事实：程序 \neq 算法

【例2】以下算法用于在带头结点的单链表h中查找第一个值为x的结点，找到后返回其逻辑序号（从1计起），否则返回0。分析该算法存在的问题。

```
int findx(LNode *h; int x)
{
    LNode *p = h->next;
    int i = 0;
    while (p->data != x)
    {
        i++;
        p = p->next;
    }
    return i;
}
```

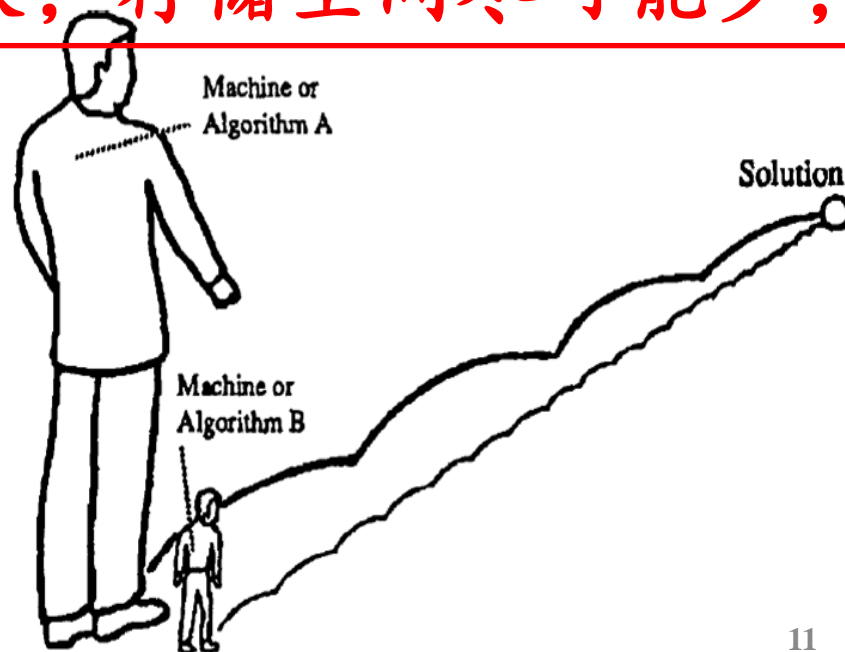
解：当单链表中首结点值为 x 时，该算法返回0，此时应该返回逻辑序号1。另外当单链表中不存在值为 x 的结点时，该算法执行出错，因为 p 为NULL时仍执行 $p=p\rightarrow next$ 。所以该算法不满足**正确性和健壮性**。应改为：

```
int findx(LNode *h;int x)
{
    LNode *p = h->next;    //p初始时指向首结点
    int i = 1;
    while ((p!=NULL) && (p->data!=x))
    {
        i++;
        p = p->next;
    }
    if (p == NULL)    //没找到值为x的结点返回0
        return 0;
    else    //找到值为x的结点返回其逻辑序号i
        return i;
}
```

什么是好的算法：

- 正确性：符合语法、编译通过；能够正确处理各种输入（简单的、大规模的、一般性的、退化的等任意合法输入）
- 健壮性：能辨别不合法的输入并做适当的处理，不至于异常退出（崩溃）
- 可读性：结构化 + 准确的命名 + 注释
- 效率性：速度尽可能快；存储空间尽可能少；

对于任意给定的问题，设计出复杂性尽可能低的算法是设计算法时追求的一个重要目标！



例：人口普查数据排序问题

❖ 考察对全国人口
普查数据的排序
 $n = 10^9 \dots$

普通PC
1GHz
 10^9 flops

天河1A
千万亿次 = 1P
 10^{15} flops

硬件

Bubblesort
 $(10^9)^2$
 10^{18}

10^9 sec
30 yr

10^3 sec
20 min

Mergesort
 $(10^9) \times \log(10^9)$
 30×10^9

30 sec

0.03 ms

算法

不同的算法，在大数据量面前，体现了巨大的效率差距！

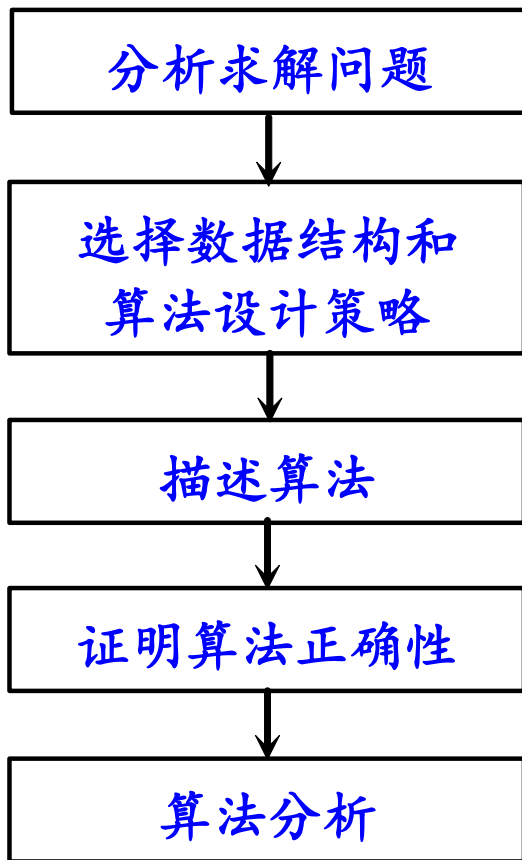
1.1.2 算法和数据结构

算法与数据结构既有联系又有区别。

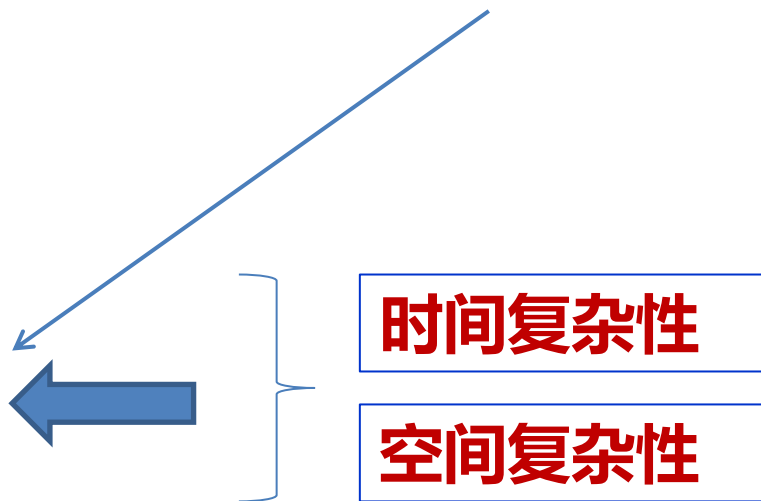
联系：数据结构是算法设计的基础。算法的操作对象是数据结构，在设计算法时，通常要构建适合这种算法的数据结构。数据结构设计主要是选择数据的存储方式，如确定求解问题中的数据采用数组存储还是采用链表存储等。算法设计就是在选定的存储结构上设计一个满足要求的好算法。

区别：数据结构关注的是数据的逻辑结构、存储结构以及基本操作，而算法更多的是关注如何在数据结构的基础上解决实际问题。算法是编程思想，数据结构则是这些思想的逻辑基础。

1.1.3 算法设计的基本步骤



同一问题，通常有多种算法，如何评判其优劣性？



1.2 算法复杂性分析

算法复杂性是算法运行所需要的计算机资源的量，随着问题规模的增长，计算成本如何增长？算法分析是分析算法占用计算机资源的情况。

所以算法分析的两个主要方面是分析算法的时间复杂度 $T(N, I)$ 和空间复杂度 $S(N, I)$ 。(其中 N 是规模， I 是输入)

评价一个算法的代价，主要看执行算法时所需要占用的计算机空间的大小和计算过程需要花费的计算机CPU时间的多少。

算法的分析主要包含时间和空间两个方面。

1.2.1 算法时间复杂度分析

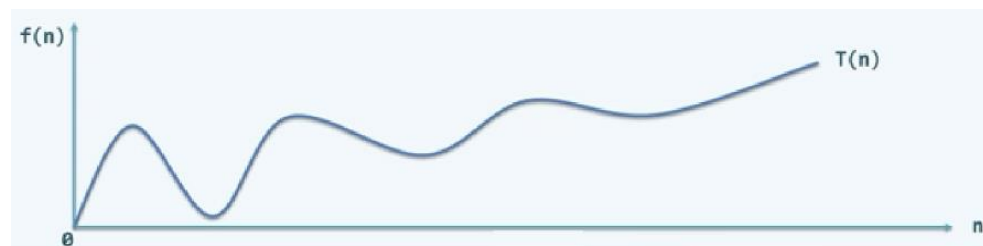
1. 时间复杂度分析概述
2. **算法的最好、最坏和平均情况**
3. 渐进符号 (O 、 Ω 和 Θ)
4. 有关阶的一些性质
5. 非递归算法的时间复杂度分析
6. 递归算法的时间复杂度分析
7. 常用的算法的时间复杂度和空间复杂度

1.2.1 算法时间复杂度分析

1. 时间复杂度分析概述

算法所耗费的时间,除了与所用的计算软、硬件环境有关外, **主要取决于算法中指令重复执行的次数, 即语句的频率相关。**

一个算法中所有语句的运算 (次数) 时间之和构成了该算法的运行时间。



```
void Solve(double a[][MAX],int m,int n,double &s)
{
    int i;
    s = 0;
    if (m != n)
        return 0;
    for (i = 0; i < m; i++)
        s += a[i][i];
    return 1;
}
```

顺序结构

分支结构

循环结构

顺序结构

2. 计算算法复杂度 $T(N)$ 的三种情况：最好、最坏和平均情况

最坏情况下的时间复杂性：

$$T_{\max}(N) = \max_{I \in D_N} T(N, I) = T(N, I^*)$$

最好情况下的时间复杂性：

$$T_{\min}(N) = \min_{I \in D_N} T(N, I) = T(N, \tilde{I})$$

平均情况下的时间复杂性：

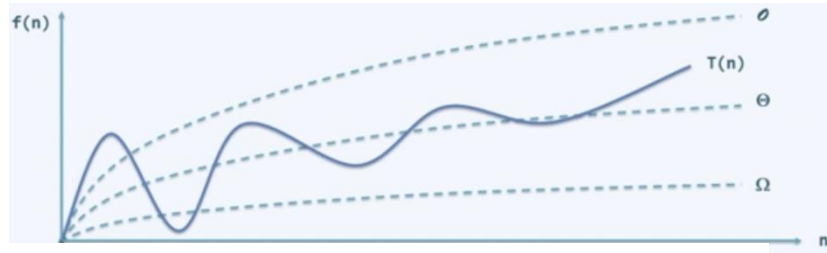
$$T_{\text{avg}}(N) = \sum_{I \in D_N} P(I)T(N, I) \quad \text{指当输入规模为} n \text{时，算法在随机输入情况下的效率；}$$

其中 D_N 是规模为 N 的合法输入的集合； I^* 是 D_N 中使 $T(N, I^*)$ 达到 $T_{\max}(N)$ 的合法输入； \tilde{I} 是中使 $T(N, \tilde{I})$ 达到 $T_{\min}(N)$ 的合法输入；而 $P(I)$ 是在算法的应用中出现输入 I 的概率。

许多算法的运行时间不仅取决于输入的规模，而且取决于特定的输入。

可操作性最好且最有实际价值的是：
最坏情况下的时间复杂性！

算法复杂度具体如何度量：



当输入规模充分大时，只需要考查算法复杂度在渐近意义下的阶，不必关心常数因子：

设 n 为算法中的问题规模，通常用大 O 、大 Ω 和 Θ 等三种渐进符号表示算法的执行时间与 n 之间的一种增长关系。

分析算法时间复杂度的一般步骤：

算法

分析问题规模 n ，找出基本语句，
求出其运行次数 $f(n)$

用 O 、 Ω 和 Θ 表示其阶

3. 渐进符号 (O 、 Ω 和 Θ)

$$\lim_{x \rightarrow \infty} \frac{f(n)}{g(n)} = c$$

定义1 (大 O 符号) :

如果 $\exists C > 0$, 以及自然数 N_0 , 使得当 $N \geq N_0$ 时有: $f(N) \leq Cg(N)$, 则称函数 $f(N)$ 当 N 充分大时上有界, 且 $g(N)$ 是它的一个上界, 记为 $f(N) = O(g(N))$ 。即 $f(N)$ 的阶不高于 $g(N)$ 的阶。

例1: $3n+2=O(n)$, 因为当 $n \geq 2$ 时, $3n+2 \leq 4n$ 。

例2: $10n^2+4n+2=O(n^4)$, 因为当 $n \geq 2$ 时, $10n^2+4n+2 \leq 10n^4$ 。

含义: $f(N)$ 的增长最多象 $g(N)$ 的增长那样快。

大O符号用来描述增长率的上界，表示 $f(n)$ 的增长最多像 $g(n)$ 增长的那样快，也就是说，当输入规模为 n 时，算法消耗时间的最大值。这个上界的阶越低，结果就越有价值，所以，对于 $10n^2+4n+2$ ， $O(n^2)$ 比 $O(n^4)$ 有价值。

一个算法的时间用大O符号表示时，总是采用最有价值的 $g(n)$ 表示，称之为“紧凑上界”或“紧确上界”。

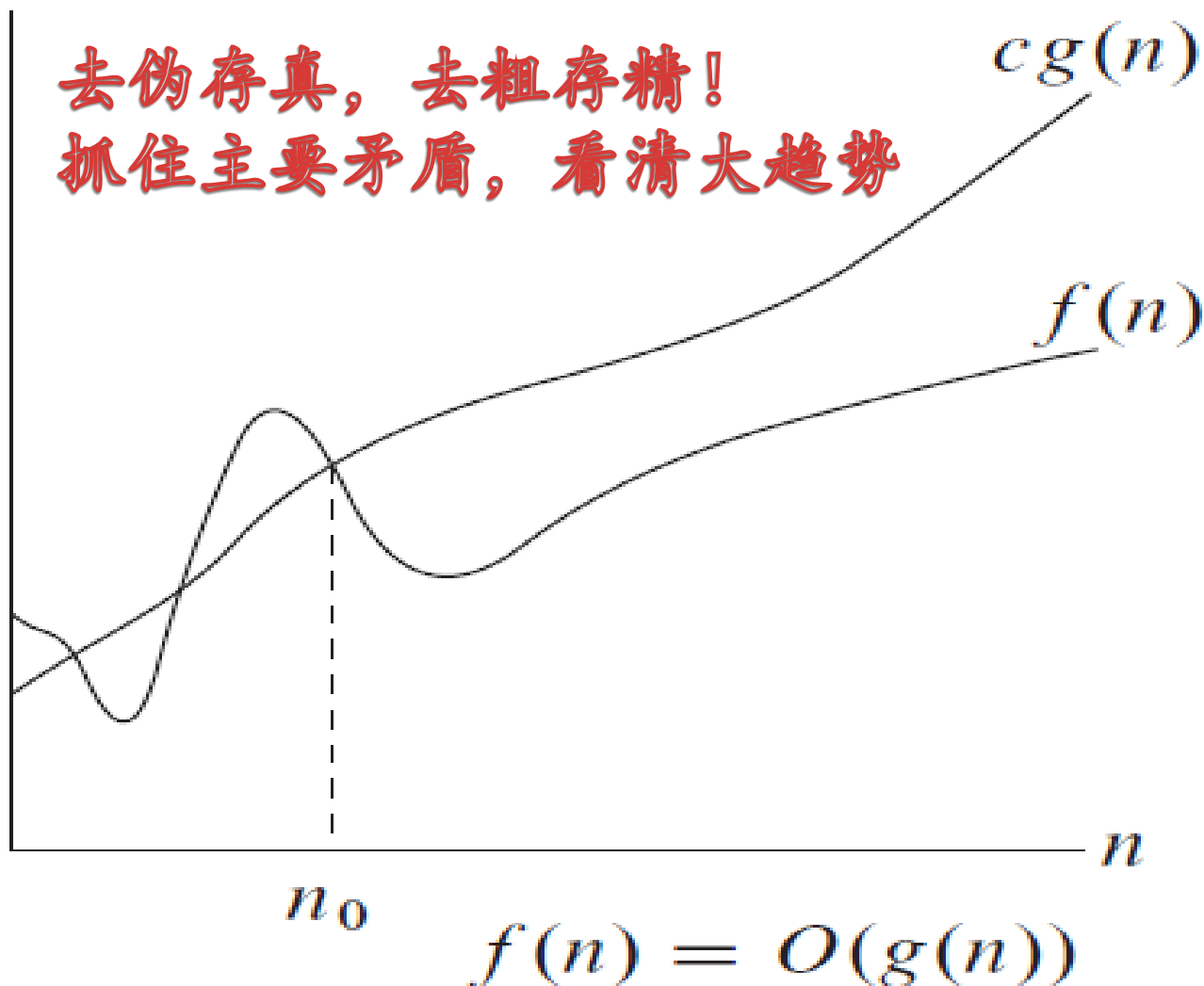
一般地，如果 $f(n)=a_m n^m+a_{m-1}n^{m-1}+\cdots+a_1n+a_0$ ，有 $f(n)=O(n^m)$ 。

*常系数可以忽略： $O(f(n)) = O(c \cdot f(n))$

*低次项可以忽略： $O(n^a + n^b) = O(n^a)$, $a > b > 0$

与 $f(n)$ 相比， $g(n)$ 更为简洁，但依然反映了前者的增长趋势。

与 $f(n)$ 相比， $g(n)$ 更为简洁，但依然反映了前者的增长趋势。



特别地，算法的时间效率是问题规模的函数，记：

$T(n) = O(f(n))$, iff $\exists c > 0$, 当 n 充分大后, 有 $T(n) < cf(n)$, 称 $T(n)$ 算法的渐近时间复杂度 (Asymptotic Time Complexity, 简称时间复杂度), 为 $f(n)$ 。

含义：算法的时间效率是问题规模的函数。随着问题规模 n 的增长，算法执行时间的增长率和 $f(n)$ 的增长率相同。

定义2（大 Ω 符号），

如果存在正的常数 C 和自然数 N_0 ，使得当 $N \geq N_0$ 时有 $f(N) \geq Cg(N)$ ，则称函数 $f(N)$ 当 N 充分大时有下界，且 $g(N)$ 是它的一个**下界**，记为 $f(N) = \Omega(g(N))$ 。即 $f(N)$ 的阶不低于 $g(N)$ 的阶。

如 $3n+2 = \Omega(n)$ ，因为当 $n \geq 1$ 时， $3n+2 \geq 3n$ 。

$10n^2+4n+2 = \Omega(n^2)$ ，因为当 $n \geq 1$ 时， $10n^2+4n+2 \geq n^2$ 。

大 Ω 符号用来描述增长率的下界，表示 $f(n)$ 的增长最少像 $g(n)$ 增长的那样快，也就是说，当输入规模为 n 时，算法消耗时间的最小值。与大 O 符号对称，这个下界的阶越高，结果就越有价值，所以，对于 $10n^2+4n+2$ ， $\Omega(n^2)$ 比 $\Omega(n)$ 有价值。一个算法的时间用大 Ω 符号表示时，总是采用最有价值的 $g(n)$ 表示，称之为“紧凑下界”或“紧确下界”。

一般地，如果 $f(n)=a_m n^m+a_{m-1}n^{m-1}+\dots+a_1n+a_0$ ，有 $f(n)=\Omega(n^m)$ 。

定义3 (大 θ 符号) ,

定义 $f(N) = \theta(g(N))$ iff

$f(N) = O(g(N))$ 且 $f(N) = \Omega(g(N))$ 。此时称 $f(N)$ 与 $g(N)$ 同阶。

如 $3n+2 = \theta(n)$, $10n^2+4n+2 = \theta(n^2)$ 。

一般地, 如果 $f(n) = a_m n^m + a_{m-1} n^{m-1} + \dots + a_1 n + a_0$, 有 $f(n) = \theta(n^m)$ 。

大 θ 符号比大 O 符号和大 Ω 符号都精确, $f(n) = \theta(g(n))$, 当且仅当 $g(n)$ 既是 $f(n)$ 的上界又是 $f(n)$ 的下界。

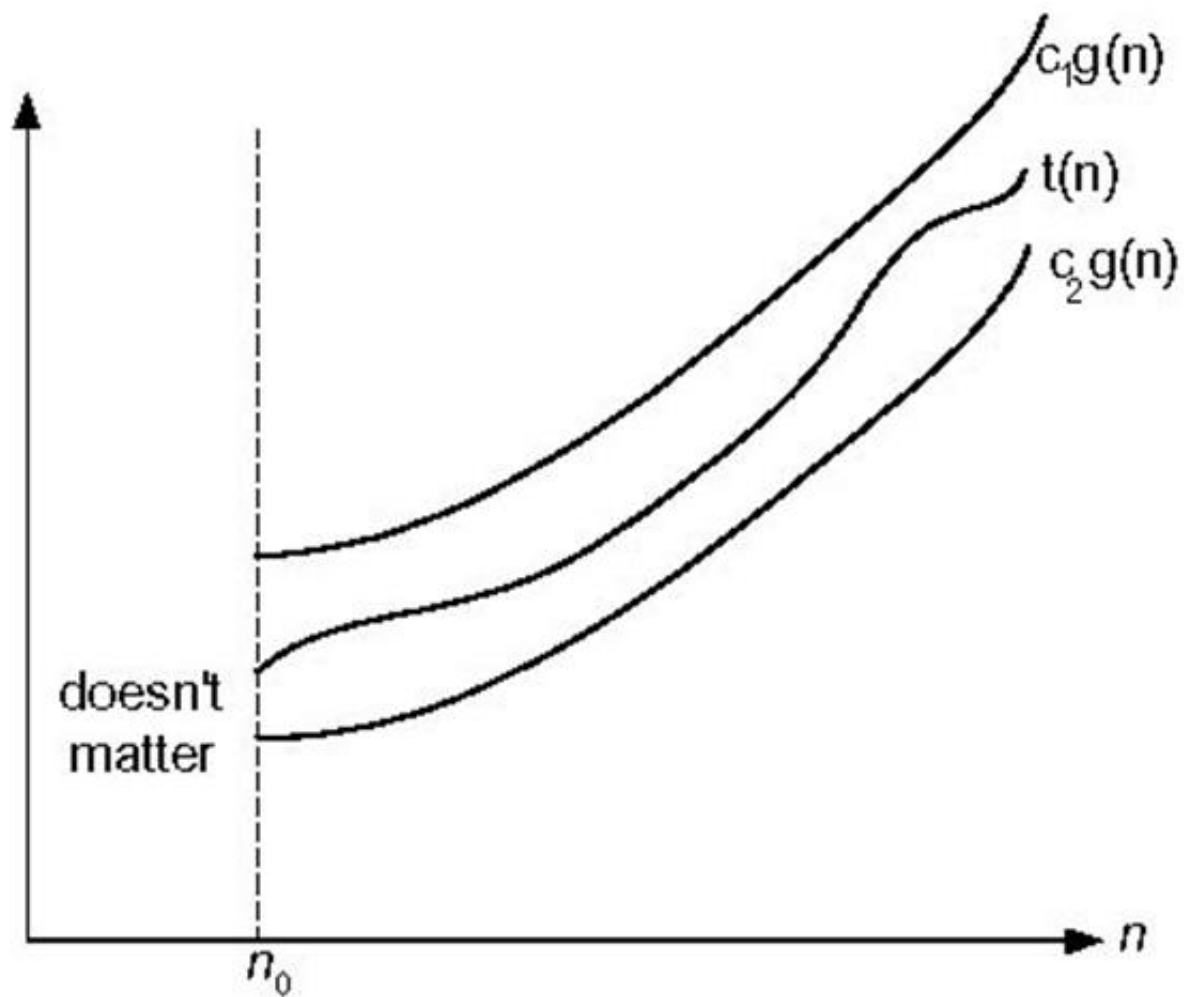


Figure Big-theta notation: $t(n) \in \Theta(g(n))$

$$\lim_{n \rightarrow \infty} \frac{t(n)}{g(n)} = \begin{cases} 0 & \text{表明 } t(n) \text{ 的增长级比 } g(n) \text{ 小} \\ c & \text{表明 } t(n) \text{ 的增长级和 } g(n) \text{ 相同} \\ \infty & \text{表明 } t(n) \text{ 的增长级比 } g(n) \text{ 大} \end{cases}$$

4. 有关阶的一些性质

定理 设 f, g, h 是定义域为自然数集合的函数,

(1) 如果 $f=O(g)$ 且 $g=O(h)$, 那么 $f=O(h)$.

(2) 如果 $f=\Omega(g)$ 且 $g=\Omega(h)$, 那么 $f=\Omega(h)$.

(3) 如果 $f=\Theta(g)$ 和 $g=\Theta(h)$, 那么 $f=\Theta(h)$.

(4) 假设 f 和 g 是定义域为自然数集合的函数, 若对某个其它的函数 h , 有 $f=O(h)$ 和 $g=O(h)$, 那么 $f+g=O(h)$.

运算规则:

(1) $O(f)+O(g)=O(\max(f,g))$;

(2) $O(f)+O(g)=O(f+g)$;



(3) $O(f)O(g)=O(fg)$;

(4) 如果 $g(N)=O(f(N))$, 则 $O(f)+O(g)=O(f)$;

(5) $O(Cf(N))=O(f(N))$, 其中 C 是一个正的常数;

(6) $f=O(f)$.

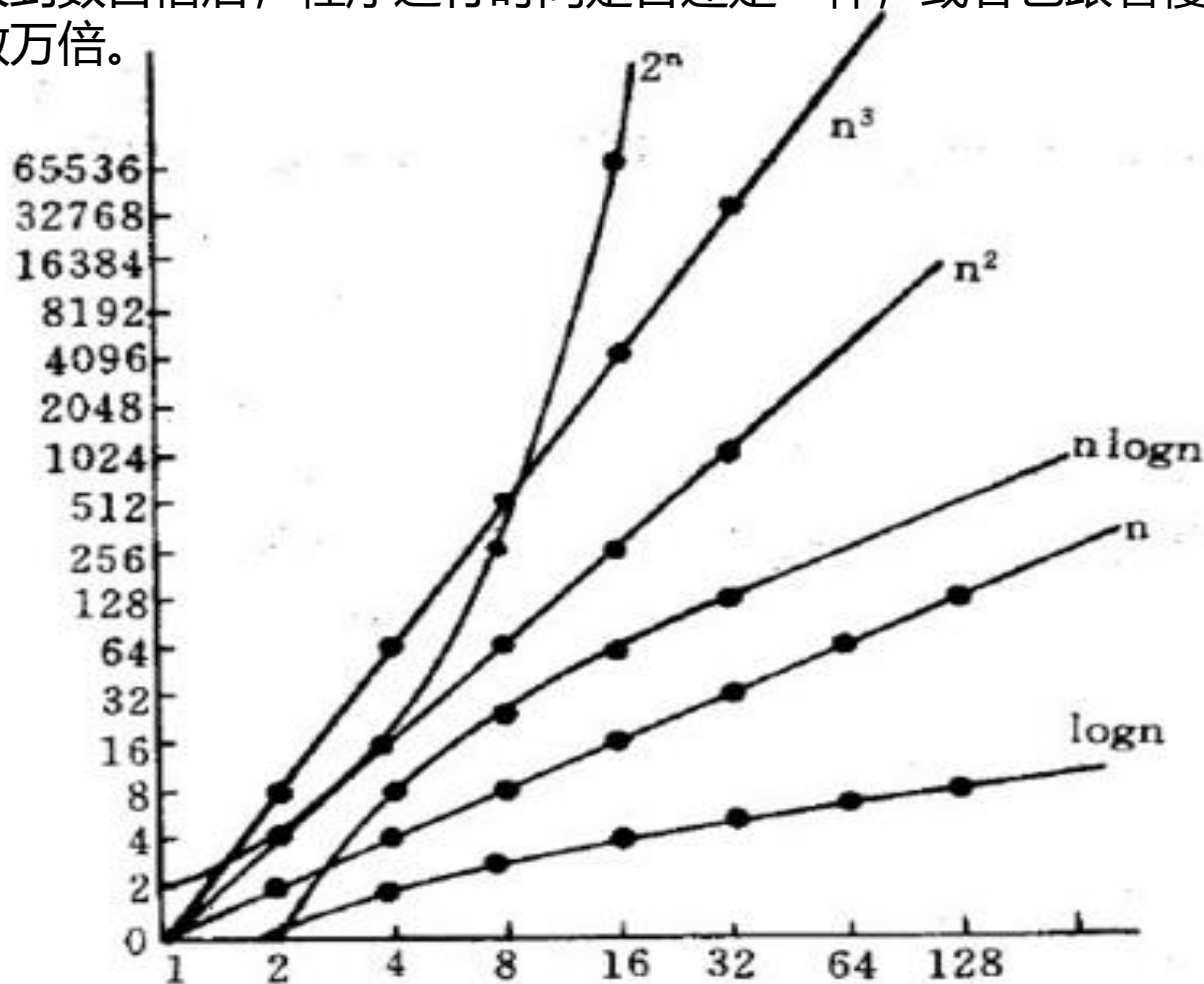
一个算法中**所有语句的频度之和**构成了该算法的运行时间 $T(n)$ 。
算法(**渐进**)时间复杂度,一般均表示为以下几种数量级的形式(n 为问题的规模, c 为一常量):

$O(1)$	称为常数级 constant		(易解问题)
$O(\log n)$	称为对数级 logarithmic		
$O(n)$	称为线性级 linear		
$O(n \log n)$	$n \log n$		
$O(n^c)$	称为多项式级 polynomial		
$O(c^n)$	称为指数级 exponential		(难解问题)
$O(n!)$	称为阶乘级 factorial		

以上为刻画算法复杂度的标准尺寸

典型的计算时间函数的曲线图

注意：时间复杂度并不是表示一个程序解决问题需要花多少时间，而是当问题规模扩大后，程序需要的时间长度增长得有多快。也就是说，对于高速处理数据的计算机来说，处理某一个特定数据的效率不能衡量一个程序的好坏，而应该看当这个数据的规模变大到数百倍后，程序运行时间是否还是一样，或者也跟着慢了数百倍，或者变慢了数万倍。



5. 非递归算法的时间复杂度分析

对于非递归算法，分析其时间复杂度相对比较简单，关键是求出代表算法执行时间的表达式。

通常是算法中基本语句的执行次数，是一个关于问题规模 n 的表达式，然后用渐进符号来表示这个表达式即得到算法的时间复杂度。

O(1) 常数级复杂度 constant

不管数据有多大，程序处理花的时间始终是那么多的，我们说这个程序具有O(1)的时间复杂度，也称常数级复杂度。

```
void swap(int &i, int &j)
{
    int tmp = *i;    //O(1)
    *i = *j;         //O(1)
    *j = tmp;        //O(1)
}
```

- ❁ O(1)表示对应的程序只有一行代码吗？
- ❁ 这类算法的效率最高？
- ❁ 代码行数多就一定复杂吗？
- ❁ 什么样的代码对应于常数执行时间？
 - 是否一定不含循环？
 - 是否规模n必须一定是常量？

【例1】：求下面这段程序的复杂度：

```
double sum=0;  
for(int i=0; i<1000, i++)  
{  
    sum+=i;  
}
```

A. 0

B. 1

C. 1000

D. 不确定

【例2】给出以下算法的时间复杂度。

问题：给定整数子集 S ， $|S| = n$ ($n \geq 3$)

找出元素非极端元素 $a \in S$ ，即： $a \neq \max(S)$ ，且 $a \neq \min(S)$

```
int find_mid(int a[]) // a 必须有大于3个元素
{
    int max_num;
    // 首先将前三个数的最大数冒泡排到第三个位置a[2];
    sort_three(a);
    // 之后比较 a[0] 和 a[1]，取大的那个数；
    max_num = max(a[0], a[1]);
    // 返回该数为非极端元素；
    return max_num;
}
```

解：无论这个子集的数目 n 有多大，上述算法的执行时间都是不变的(该算法段的执行时间是一个与问题规模 n 无关的常数)。

$$T(n) = \text{常数} = O(1)$$

O(n) 线性级复杂度 linear

数据规模变得有多大，花的时间也跟着变得有多长，这个程序的时间复杂度就是O(n)。

```
void Sum(int n)
{
    int s=0;           //O(1)
    for (i=0; i<n; i++) //O(n)
    {
        s = s+i;       //O(1)
    }
}
```

 这类算法的效率比较高

【例3】 给出以下算法的时间复杂度。

```
void func(int n)
{
    int i=1, k=100;
    while (i <= n)
    {
        k++;
        i+=2;
    }
}
```

解： 算法中基本语句是while循环内的语句。设while循环语句执行次数为 $m = (1, 2, 4, \dots, [n/2])$ 。

i 从1开始递增，最后取值为 $1+2m$ ，有：

$i=1+2m \leq n$ ，即 $f(n)=m \leq (n-1)/2 = O(n)$ 。

该算法的时间复杂度为 $O(n)$ 。

$O(\log n)$ 对数级复杂度 logarithmic

复杂度为 $\log n$ 时，这类算法非常有效，复杂度无限接近于常数。

● 常底数是为所谓的

$$\forall a, b > 0, \log_a n = \log_a b \cdot \log_b n = O(\log_b n)$$

● 常数次幂无所谓

$$\forall c > 0, \log n^c = c \cdot \log n = O(\log n)$$

● 这类算法的效率比较高

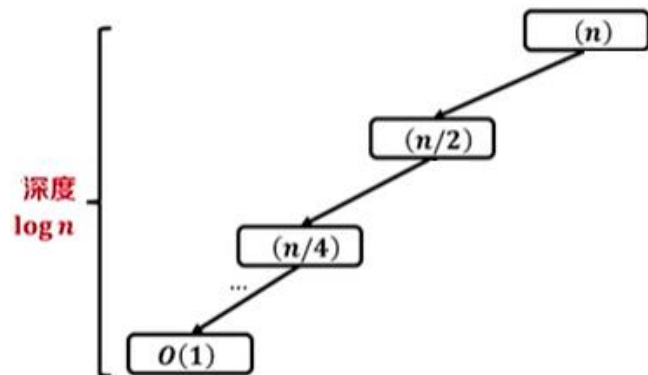
【例4】 分析以下算法的时间复杂度：

```
void fun(int n)
{
    i=1;
    while(i <= n)
    {
        i = i*2;
    }
}
```

解：该算法的基本语句是while循环，有：

设以上循环的次数为 k ，则 $2^k=n$ ，
所以循环的次数为 $\log_2 n$ 。

则该算法的时间复杂度为 $O(\log_2 n)$ 。



【例5】 分析以下算法的时间复杂度：

```
void fun(int n)
{
    s=0;
    while(n>1)
    {
        n = n/2;
        s++
    }
}
```

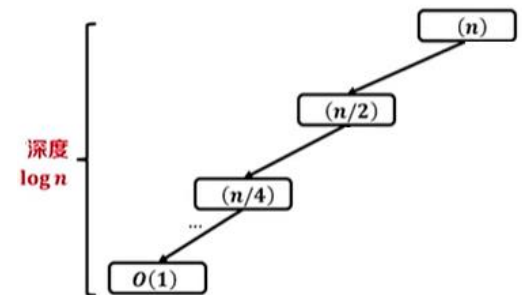
分裂法则，除
以其他常数也
是一样的

解： 该算法的基本语句是while循环，循环结果为：

n 、 $n/2$ 、 $n/2^2$ 、 $n/2^3$ 、... $n/2^m$

$m+1$ 使得 $(n/2^{m+1} < 1)$ ，即： $2^m \leq n < 2^{m+1}$ ，
 $m \leq \log n < m+1$ 。即： $m \approx \log n$

则该算法的时间复杂度为 $O(\log_2 n)$ 。



【例6】 分析以下算法的时间复杂度：

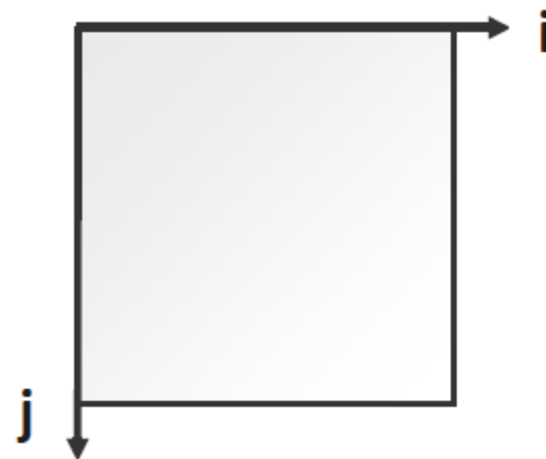
```
for(i=0; i<n; i++)  
{  
    m = n;  
    while(m>1)  
    {  
        m = m/2;  
        s = s+m;  
    }//end while  
}//end for
```

则该算法的时间复杂度为 $O(\mathbf{n}\log_2 n)$ 。

$O(n^2)$ 多项式级复杂度 polynomial

像冒泡排序、插入排序等，数据扩大2倍，时间变慢4倍的，属于 $O(n^2)$ 的复杂度。

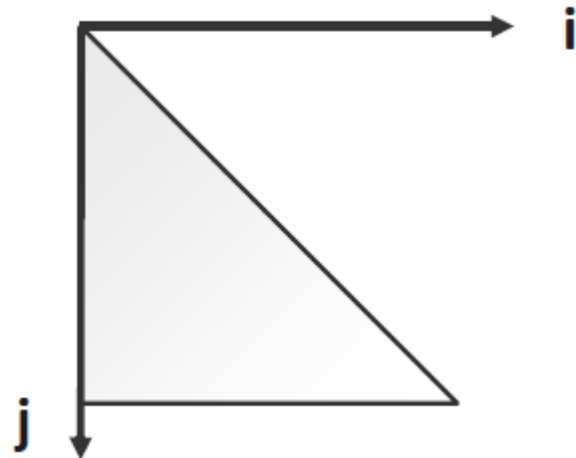
```
void Sum(int n)
{
    int s = 0;
    for (i=0; i<n; i++)
        for (j=0; j<n; j++)
        {
            s = s++;
        }
}
```



🌿 这类算法的效率比较差，但可以忍受。

【例7】 分析以下算法的时间复杂度：

```
void fun(int n)
{
    for(i=0; i<n; i++)
    {
        for(j=i; j<n; j++)
        {
            S++;
        }
    }
}
```



解： 该算法的基本语句是s++，所以有：

$$N = n + (n-1) + (n-2) + \dots + 2 + 1 = n(n+1)/2;$$

则该算法的时间复杂度为

$$T(n) = O(n(n+1)/2) = O(n^2)。$$

$O(2^n)$ 指数级复杂度 exponential

计算成本增长极快，计算机往往不能承受。当我们解决一个问题时，**我们选择的算法通常都需要是多项式级的复杂度**，非多项式级的复杂度需要的时间太多，往往会超时，除非是数据规模非常小。

● 从 $O(n^c)$ 到 $O(2^n)$ ，是从无效算法到有效算法的分水岭

● 很多问题的 $O(2^n)$ 算法往往是显而易见的，然而，设计出 $O(n^c)$ 的算法却极其不易，甚至是找不到的。

【例8】问题描述：S包含n个正整数，且 $\sum S=2m$ ，S是否存在子集T，满足 $\sum T=m$ （给定一个具有n个整数的集合S，是否能把S划分成两个子集S1和S2，使得S1中的整数之和等于S2中的整数之和）

示例：美国选举人制——各州议会选出的选举人团投票，而不是由选民直接投票。

□50个州加一个特区，共538张票。谁先拿到270张票，谁就超过半数获胜。

□若有两位候选人，是否可能恰好各得269票？

问题：是否会有2位候选人恰好各得269票？

解法：逐一枚举S的每一子集，并统计其中元素的总和。

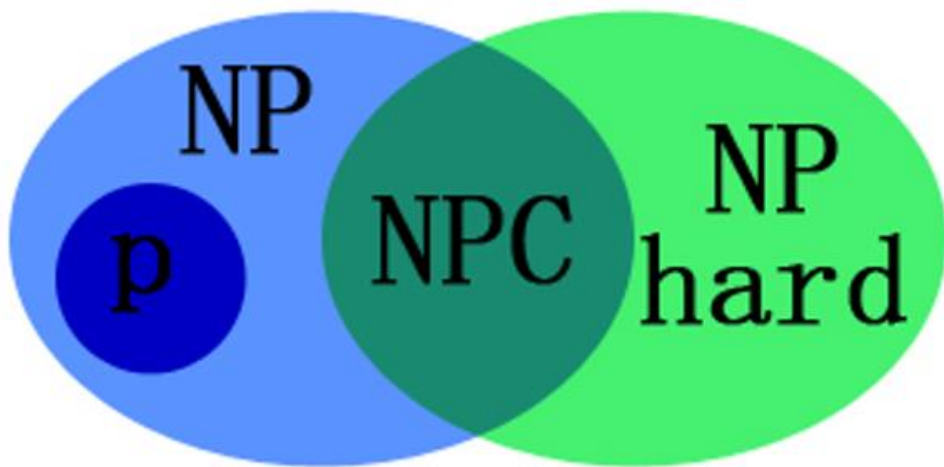
该算法的时间复杂度为 $O(2^{|S|})=O(2^n)$ 。

2-Subset is NP问题：
在目前的计算模型下，
不存在多项式时间

归纳起来，在各种求解问题中，按求解问题算法的时间复杂度可分为三大类：

- 第一类是存在多项式算法的问题（**P类问题，易解问题**）
- 第二类是肯定不存在多项式算法的问题(指数时间复杂性)
- 第三类是尚未找到多项式算法，也不能证明其不存在多项式算法的问题。第三类问题介于第一类和第二类之间。

例如“找出无向图中哈密顿回路”问题



https://blog.csdn.net/qq_36193984/article/details/84954211



6.递归算法的时间复杂度分析

对递归算法时间复杂度的分析，关键是根据递归过程建立递推关系式（**递推方程**），然后求解这个递推关系式，得到一个表示算法执行时间的表达式，最后用渐进符号来表示这个表达式即得到算法的时间复杂度。

还有一种方法是**递归跟踪**，但仅适用于简单的递归模式。

【例9】计算任意n个整数之和：

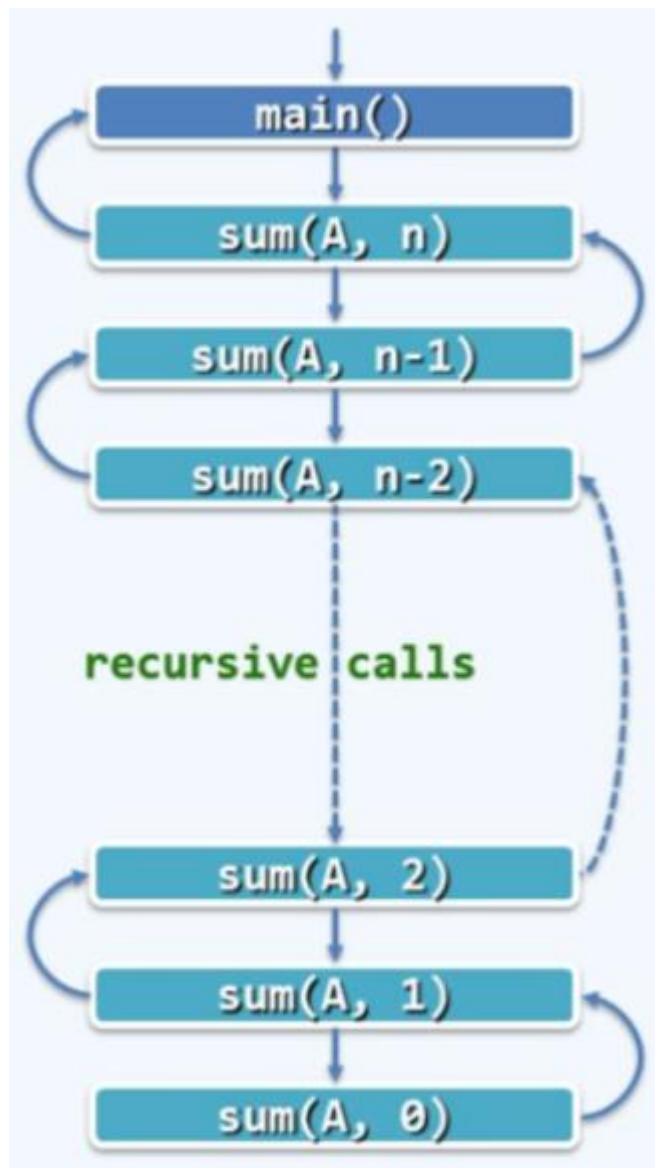
```
int sum(int a[], int n)
{
    return (n<1)? 0 : sum(a, n-1)+a[n-1];
}
```

递归跟踪 (recursion trace) 分析：

检查每一个递归实例，累计所需的时间；

则该算法的时间复杂度为：

$$T(n) = O(1) * (n+1) = O(n)。$$



【例10】 求解梵塔问题的递归算法如下，分析其时间复杂度。

```
void Hanoi(int n, char x, char y, char z)
{
    if (n == 1)
        printf("将盘片%d从%c搬到%c\n", n, x, z);
    else
    {
        Hanoi(n-1, x, z, y);
        printf("将盘片%d从%c搬到%c\n", n, x, z);
        Hanoi(n-1, y, x, z);
    }
}
```

解：设调用 $\text{Hanoi}(n, x, y, z)$ 的执行时间为 $T(n)$ ，由其执行过程得到以下求执行时间的递归关系（递推关系式）：

$$T(n) = O(1) \quad \text{当 } n=1$$

$$T(n) = 2T(n-1) + 1 \quad \text{当 } n > 1$$

则：

$$T(n) = 2[2T(n-2) + 1] + 1 = 2^2T(n-2) + 1 + 2^1$$

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

$$= \dots$$

$$= 2^{n-1}T(1) + 1 + 2^1 + 2^2 + \dots + 2^{n-2}$$

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

7.总结：常见算法的时间复杂度

$O(1)$	常数复杂度	再好不过，但难得如此幸运	对数据结构的基本操作
$O(\log^*n)$		在这个宇宙中，几乎就是常数	
$O(\log n)$	对数复杂度	与常数无限接近，且不难遇到	有序向量的二分查找 堆、词典的查询、插入与删除
$O(n)$	线性复杂度	努力目标，经常遇到	树、图的遍历
$O(n\log^*n)$		几乎几乎几乎接近线性	某些MST算法
$O(n\log\log n)$		几乎接近线性	某些三角剖分算法
$O(n\log n)$		最常出现，但不见得最优	排序、EU、Huffman编码
$O(n^2)$	平方复杂度	所有输入对象两两组合	Dijkstra算法
$O(n^3)$	立方复杂度	不常见	矩阵乘法
$O(n^c)$ ， c 常数	多项式复杂度	P问题 = 存在多项式算法的问题	
$O(2^n)$	指数复杂度	很多问题的平凡算法，再尽可能优化	
...		绝大多数问题，并不存在算法	

*以上总结来自清华大学邓俊辉《数据结构》讲义的总结，供参考。

增长速度的一个链

$$1 < \log \log n < \log n < \sqrt{n} < \sqrt[3]{n} < n \\ < n \log n < n^2 < 2^n < n! < 2^{n^2}$$

**n 趋于无穷大时, ‘<’ 的
前项是后项的高阶无穷小!**

【随堂练】完成以下单选题：

下面函数中渐进时间最小的是_____。

- A. $T_1(n) = n + n \log n$**
- B. $T_2(n) = 2n + n \log n$**
- C. $T_3(n) = n^2 - \log n$**
- D. $T_4(n) = n + 100 \log n$**

1.2.2 算法空间复杂度

空间复杂度是指当问题的规模以某种单位从1增加到n时，解决这个问题的算法在执行时所占用的存储空间也以某种单位由1增加到 $f(n)$ ，则称此算法的**空间复杂度为 $f(n)$** 。

一个算法的存储量包括形参所占空间和临时变量所占空间。在对算法进行存储空间分析时，只考察临时变量所占空间。

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

函数体内分配的变量空间为临时空间，**不计**形参占用的空间，这里的仅计 i 、 $maxi$ 变量的空间，其空间复杂度为 $O(1)$ 。

为什么算法占用的空间只考虑临时空间，而不必考虑形参的空间呢？这是因为形参的空间会在调用该算法的算法中考虑，例如，以下maxfun算法调用图的max算法：

```
void maxfun()  
{   int b[]={1, 2, 3, 4, 5}, n=5;  
    printf("Max=%d\n", max(b, n));  
}
```

maxfun算法中为**b**数组分配了相应的内存空间，其空间复杂度为 $O(n)$ ，如果在max算法中再考虑形参**a**的空间，这样重复计算了占用的空间。实际上，在C/C++语言中，maxfun调用max时，max的形参**a**只是一个指向实参**b**数组的指针，形参**a**只分配一个地址大小的空间，并非另外分配5个整型单元的空间。

复习本章开始时提出的问题

- ✓ 什么是算法？什么是好的算法？
- ✓ 评判一个算法好坏的标准是什么？

课下作业（全课程）

□ 复习、预习

□ 重点练习：

- 计算斐波那契数列的第 n 项和；
- 二分查找；
- 合并排序；
- 快速排序；

递归与分治

- 0-1背包问题；
- 8皇后问题；
- 素数环问题；

回溯（深搜）

- 电子老鼠闯迷宫；
- 8数码难题；

广度搜索

- 最长公共子序列；
- 计算矩阵连乘积；
- 最大子序列和问题；

动态规划

- 活动安排问题；

贪心算法

END