

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。

1.1 算法的概念

1.1.1 什么是算法

算法设计应满足以下要求：

- 正确性
- 可读性
- 健壮性
- 高效率与低存储量需求

1.1 算法的概念

1.1.1 什么是算法

【例】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;
}
```

1.1 算法的概念

1.1.1 什么是算法

解：存在的问题：

- 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。
- 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 $p=p->next$ 。

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

```
int findx(LNode *h,int x)
{
    LNode *p=h->next;
    int i=1;
    while (p!=NULL && p->data!=x)
    {
        i++;
        p=p->next;
    }
    if (p==NULL)
        return 0;
    else
        return i;
}
```

1.1 算法的概念

1.1.1 什么是算法

算法具有以下5个重要特性：

- 有限性
- 确定性
- 可行性
- 输入性
- 输出性

1.1 算法的概念

1.1.1 什么是算法

【例1.2】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);
}
```

描述2：

```
void exam2()
{   int x,y;
    y=0;
    x=5/y;

    printf("%d,%d\n",x,y);
}
```

试问它们违反了算法的哪些特性？

1.1 算法的概念

1.1.1 什么是算法

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。

```
void exam1()
{   int n;
    n=2;
    while (n%2==0)
        n=n+2;

    printf("%d\n",n);
}
```

```
void exam2()
{   int x,y;
    y=0;
    x=5/y;

    printf("%d,%d\n",x,y);
}
```

1.1 算法的概念

1.1.2 算法描述

以设计求 $1+2+\dots+n$ 值的算法为例说明C/C++语言描述算法的一般形式，该算法如下：

返回值

形参列表

```
bool fun(int n, int s)
{
    if (n<0) return false;
    s=0;
    for (int i=1; i<=n; i++)
        s+=i;
    return true;
}
```


1.1 算法的概念

1.1.2 算法描述

通常用函数的返回值表示算法能否正确执行。

有时当算法只有一个返回值或者返回值可以区分算法是否正确执行时，用函数返回来表示算法的执行结果，另外还可以带有形参表示算法的输入输出

1.1 算法的概念

1.1.2 算法描述

在C语言中调用函数时只有从实参到形参的单向值传递，执行函数时若改变了形参而对应的实参不会同步改变。

例如，设计以下主函数调用上面的fun函数：

```
void main()
{
    int a=10,b=0;
    if (fun(a,b)) printf("%d\n",b);
    else printf("参数错误\n");
}
```

```
bool fun(int n,int s)
{
    if (n<0) return false;
    s=0;
    for (int i=1;i<=n;i++)
        s+=i;
    return true;
}
```

执行时发现输出结果为0，因为b对应的形参为s，fun执行后s=55，但s并没有回传给b。

在C语言中可以用传指针方式来实现形参的回传，但增加了函数的复杂性。

1.1 算法的概念

1.1.2 算法描述

为此C++语言中增加了引用型参数的概念，引用型参数名前需加上&，表示这样的形参在执行后将结果回传给对应的实参。上例采用C++语言描述算法如下所示。

引用参数

```
bool fun(int n,int &s)
{
    if (n<0) return false;
    s=0;
    for (int i=1;i<=n;i++)
        s+=i;
    return true;
}
```

当将形参s改为引用类型的参数后，执行时main函数的输出结果就正确了即输出55。

结论：在设计算法时，如果某个形参需要将执行结果回传给实参，需要将该形参设计为引用型参数

1.1 算法的概念

1.1.3 算法和数据结构

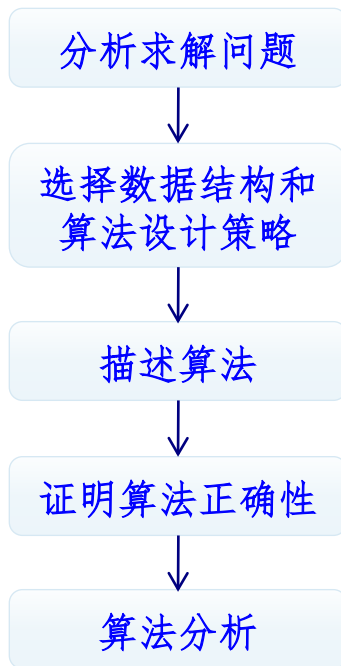
算法与数据结构既有联系又有区别。

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基础。

1.1 算法的概念

1.1.4 算法设计的基本步骤



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和空间复杂度

1.2 算法分析

1.2.1 算法时间复杂度分析

1. 时间复杂度分析概述

一个算法是由控制结构（顺序、分支和循环3种）和原操作（指固有数据类型类型的操作）构成的，算法的运行时间取决于两者的综合效果。

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

顺序结构

分支结构

循环结构

顺序结构

1.2 算法分析

1.2.1 算法时间复杂度分析

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 或 Θ 表示其阶

1.2 算法分析

1.2.1 算法时间复杂度分析

2. 渐进符号 (O 、 Ω 和 Θ)

定义1 (大O符号), $f(n)=O(g(n))$ (读作“ $f(n)$ 是 $g(n)$ 的大O”) 当且仅当存在正常量 c 和 n_0 , 使当 $n \geq n_0$ 时,
 $f(n) \leq cg(n)$, 即 $g(n)$ 为 $f(n)$ 的上界。即 f 的阶不高于 g 的阶。

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

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

- 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**”，

低阶 O

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

1.2 算法分析

1.2.1 算法时间复杂度分析

2. 渐进符号 (O 、 Ω 和 Θ)

定义2 (大 Ω 符号)， $f(n) = \Omega(g(n))$ (读作“ $f(n)$ 是 $g(n)$ 的大 Ω ”) 当且仅当存在正常量 c 和 n_0 ，使当 $n \geq n_0$ 时， $f(n) \geq cg(n)$ ，即 $g(n)$ 为 $f(n)$ 的**下界**。即 f 的阶高于 g 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， $10n^2+4n+2 \geq n^2$ 。

1.2 算法分析

1.2.1 算法时间复杂度分析

2. 渐进符号 (O 、 Ω 和 Θ)

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”，高阶 Ω

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

1.2 算法分析

1.2.1 算法时间复杂度分析

2. 渐进符号 (O 、 Ω 和 Θ)

定义3 (大 Θ 符号)， $f(n) = \Theta(g(n))$ (读作“ $f(n)$ 是 $g(n)$ 的大 Θ ”) 当且仅当存在正常量 c_1 、 c_2 和 n_0 ，使当 $n \geq n_0$ 时，有 $c_1g(n) \leq f(n) \leq c_2g(n)$ ，即 $g(n)$ 与 $f(n)$ 的**同阶**。

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

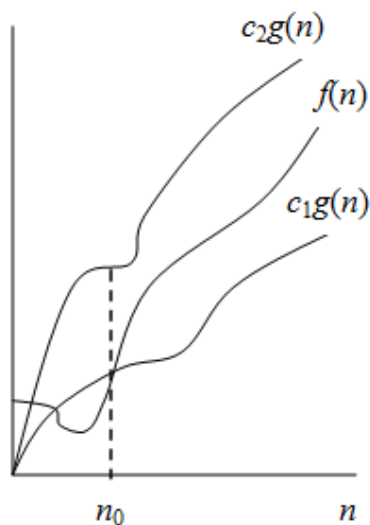
一般地，如果 $f(n) = a_m n^m + a_{m-1} n^{m-1} + \dots + a_1 n + a_0$ ，有 $f(n) = \Theta(n^m)$

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。

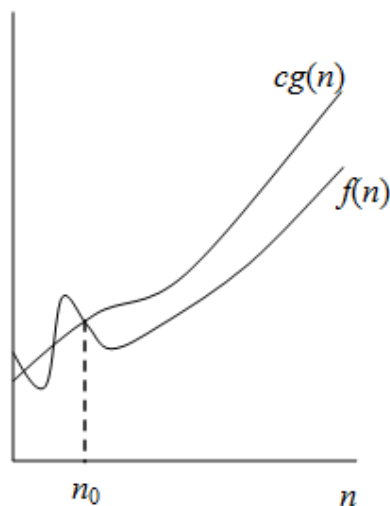
1.2 算法分析

1.2.1 算法时间复杂度分析

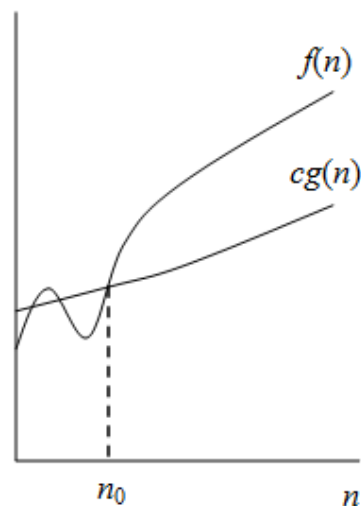
2. 渐进符号 (O 、 Ω 和 Θ)



(a) $f(n) = \Theta(g(n))$



(b) $f(n) = O(g(n))$



(c) $f(n) = \Omega(g(n))$

1.2 算法分析

1.2.1 算法时间复杂度分析

2. 渐进符号 (O 、 Ω 和 Θ)

渐进分析

2、渐进阶分析简化规则

- <1> 若 $f(n)$ 在 $O(g(n))$ 且 $g(n)$ 在 $O(h(n))$ 中, 则 $f(n)$ 在 $O(h(n))$ 中。 (O, Ω, θ 均适用)
- <2> 若 $f(n)$ 在 $O(kg(n))$ 中, ($k > 0$)成立, 则 $f(n)$ 在 $O(g(n))$ 中, 可忽略常数因子
- <3> 若 $f_1(n)$ 在 $O(g_1(n))$ 中, 且 $f_2(n)$ 在 $O(g_2(n))$ 中, 则 $f_1(n) + f_2(n)$ 在 $O(\max(g_1(n), g_2(n)))$ 中
- <4> 若 $f_1(n)$ 在 $O(g_1(n))$ 中, 且 $f_2(n)$ 在 $O(g_2(n))$ 中, 则 $f_1(n)f_2(n)$ 在 $O(g_1(n)g_2(n))$ 中。

1.2 算法分析

1.2 .2 算法复杂性分析方法

阶的证明方法

1>反证法（否定性描述）

例1-8: n^3, n^2 , 证明 n^3 不是 $O(n^2)$

证明: 假设 n^3 是 $O(n^2)$

据低阶 O 定义, $n^3 \leq Cn^2$

但对 $\forall n \geq n_0, n \leq C$ 不成立

$\therefore n^3 \neq O(n^2)$

1.2 算法分析

1.2 .2 算法复杂性分析方法

- 阶的证明方法

2>极限法:

$$\lim_{n \rightarrow \infty} \frac{f(n)}{g(n)} = C \begin{cases} \text{若 } c \neq 0, \text{ 则 } f \text{ 和 } g \text{ 同阶, } f = \theta(g) \\ \text{若 } c = 0, \text{ 则 } f \text{ 和 } g \text{ 不同阶, } f = O(g), \text{ 但 } g \text{ 不是 } \theta(f) \\ \text{若 } c = \infty, \text{ 则 } f = \Omega(g), f \text{ 是 } g \text{ 的高阶} \end{cases}$$

$$\text{若 } \lim_{n \rightarrow \infty} f(n) = \lim_{n \rightarrow \infty} g(n) = \infty, \text{ 则 } \lim_{n \rightarrow \infty} \frac{f(n)}{g(n)} = \lim_{n \rightarrow \infty} \frac{f'(n)}{g'(n)}$$

1.2 算法分析

1.2.2 算法复杂性分析方法

- 阶的证明方法

2>极限法:

例1-8:

$\log n$ 是 $O(n)$,但不是 $\theta(n)$

$$\lim_{n \rightarrow \infty} \frac{\log n}{n} = \lim_{n \rightarrow \infty} \frac{\ln n / \ln 2}{n} = \lim_{n \rightarrow \infty} \frac{1}{n \ln 2} = C = 0$$

$$\therefore C = 0$$

$\therefore \log n = O(n)$ 但不是 $\theta(n)$

【例】分析以下算法的时间复杂度：

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

解：该算法的基本语句是s++，所以有：

$$\begin{aligned} f(n) &= \sum_{i=0}^n \sum_{j=0}^i \sum_{k=0}^{j-1} 1 = \sum_{i=0}^n \sum_{j=0}^i (j-1-0+1) = \sum_{i=0}^n \sum_{j=0}^i j \\ &= \sum_{i=0}^n \frac{i(i+1)}{2} = \frac{1}{2} \left(\sum_{i=0}^n i^2 + \sum_{i=0}^n i \right) = \frac{2n^3 + 6n^2 + 4n}{12} = O(n^3) \end{aligned}$$

则该算法的时间复杂度为 $O(n^3)$ 。

1.2 算法分析

1.2.2 算法复杂性分析方法

3. 算法的最好、最坏和平均情况

定义4 设一个算法的输入规模为 n , D_n 是所有输入的集合, 任一输入 $I \in D_n$, $P(I)$ 是 I 出现的概率, 有 $\sum P(I)=1$, $T(I)$ 是算法在输入 I 下所执行的基本语句次数, 则该算法的平均执行时间为: $A(n) = \sum_{I \in D_n} P(I) * T(I)$

也就是说算法的平均情况是指用各种特定输入下的基本语句执行次数的带权平均值。

1.2 算法分析

1.2.2 算法复杂性分析方法

3. 算法的最好、最坏和平均情况

算法的最好情况为： $G(n) = \min_{I \in D_n} \{T(I)\}$ ，是指算法在所有输入 I 下所执行基本语句的最少次数

算法的最坏情况为： $W(n) = \max_{I \in D_n} \{T(I)\}$ ，是指算法在所有输入 I 下所执行基本语句的最大次数

1.2 算法分析

1.2.2 算法复杂性分析方法

3. 算法的最好、最坏和平均情况

【例】采用顺序查找方法，在长度为 n 的一维实型数组 $a[0..n-1]$ 中查找值为 x 的元素。即从数组的第一个元素开始，逐个与被查值 x 进行比较。找到后返回1，否则返回0，对应的算法如下：

```
int Find(int a[],int n,int x)
{   int i=0;
    while (i<n)
    {   if (a[i]==x) break;
        i++;
    }
    if (i<n) return 1;
    else return 0;
}
```

回答以下问题：

- (1) 分析该算法在等概率情况下成功查找到值为 x 的元素的最好、最坏和平均时间复杂度。
- (2) 假设被查值 x 在数组 a 中的概率是 q ，求算法的时间复杂度。

1.2 算法分析

1.2.2 算法复杂性分析方法

3. 算法的最好、最坏和平均情况

```
int Find(int a[],int n,int x)
{   int i=0;
    while (i<n)
    {   if (a[i]==x) break;
        i++;
    }
    if (i<n) return 1;
    else return 0;
}
```

解：（1）算法的while循环中的if语句是基本语句。 a 数组中有 n 个元素，当第一个元素 $a[0]$ 等于 x ，此时基本语句仅执行一次，此时呈现**最好的情况**，即 $G(n)=O(1)$ 。

当 a 中最后一个元素 $a[n-1]$ 等于 x ，此时基本语句执行 n 次，此时呈现**最坏的情况**，即 $W(n)=O(n)$ 。

1.2 算法分析

1.2.2 算法复杂性分析方法

3. 算法的最好、最坏和平均情况

```
int Find(int a[],int n,int x)
{  int i=0;
  while (i<n)
  {  if (a[i]==x) break;
    i++;
  }
  if (i<n) return 1;
  else return 0;}
```

对于其他情况，假设查找每个元素的概率相同，则 $P(a[i])=1/n$ ($0 \leq i \leq n-1$)，而成功找到 $a[i]$ 元素时基本语句正好执行 $i+1$ 次，所以平均情况下的时间耗费为()：

$$A(n) = \sum_{i=0}^{n-1} \frac{1}{n} (i+1) = \frac{1}{n} \sum_{i=0}^{n-1} (i+1) = \frac{n+1}{2} = O(n)。$$

平均状况下时间耗费的计算有的时候会取：（最好+最坏）/2来估算。

上例： $A(n) = (1+n)/2 = O(n)$

1.2 算法分析

1.2.2 算法复杂性分析方法

3. 算法的最好、最坏和平均情况

(2) 当被查值 x 在数组 a 中的概率为 q (即值为 x 的元素在 a 数组中有 q 个) 时, 算法执行有 $n+1$ 种情况, 即 n 种成功查找和一种不成功查找。

对于成功查找, 假设是等概率情况, 则元素 $a[i]$ ($a[i]$ 值为 x 即值为 x 的元素存在于 a 数组下标为 i 的位置)

那么值为 x 的元素被查找到的概率 $P(a[i]) = q/n$, 成功找到 $a[i]$ 元素时基本语句正好执行 $i+1$ 次。

对于不成功查找, 其概率为 $1-q$, 不成功查找时基本语句正好执行 n 次。

所以:

$$\begin{aligned} A(n) &= \sum_{I \in D_n} P(I) * T(I) \\ &= \sum_{i=0}^{n-1} P(I) * T(I) = \sum_{i=0}^{n-1} \frac{q}{n} (i+1) + (1-q)n = \frac{(n+1)q}{2} + (1-q)n \end{aligned}$$

如果已知需要查找的 x 有一半的机会在数组中, 此时

$q=1/2$, 则 $A(n) = [(n+1)/4] + n/2 \approx 3n/4$

1.2 算法分析

1.2.2 算法复杂性分析方法

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。

1.2 算法分析

1.2.2 算法复杂性分析方法

非递归算法的时间复杂度分析

【例】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循环语句执行的次数为 m ， i 从1开始递增，最后取值为 $1+2m$ ，有：

$$i=1+2m \leq n$$

$$f(n)=m \leq (n-1)/2 = O(n)。$$

该算法的时间复杂度为 $O(n)$ 。

算法复杂性分析示例： 求指定语句的语句频度和时间复杂度

1、给出程序段，计算语句频度和时间复杂度

- 分析一下算法的时间复杂度

```
void func(int n)
{ int i=0,s=0;
  while(s<n)
  {i++;
   s=s+i;}
}
```

求指定语句的语句频度和时间复杂度

```
void func(int n)
{ int i=0,s=0;
  while(s<n)
  {i++;
   s=s+i;}
}
```

- 假设该循环执行了 m 次（从第一次开始），
- 第一次 $i=1,s=1$;
- 第二次 $i=2,s=1+2;....$
- 第 m 次， $i=m,i$ 从0开始递增到 m 为止，有 $s=1+2+3+...+m=m(m+1)/2$,当执行完 m 次时， $s=m(m+1)/2 \geq n$,则该算法的时间复杂度为 $O(\sqrt{n})$

给出程序段，计算
语句频度和时间
复杂度

- 分析一下算法的
时间复杂度

```
void func(int n)
{ int i=0,s=0;
  while(s<n)
  {i++;
   s=s+i;}
}
```



也可以根据递归方程式：

$$\begin{aligned} S_k &= S_{k-1} + k \\ &= S_{k-2} + k-1 + k \\ &= S_{k-3} + k-2 + k-1 + k \\ &= \dots \\ &= S_1 + 2 + 3 + 4 + \dots + k \\ &= k(k+1)/2 > n \\ k &= O(\sqrt{n}) \end{aligned}$$

求指定语句的语句频度和时间复杂度

2、设 n 为非负整数，则下列程序段的时间复杂度是（ ）

```
x=2;  
while(x<n/2)  
x=x*2;
```

算法的基本语句是
 $x=x*2$ ，设其执行了 m 次，
则有 $2^m \leq n/2$ ，即
 $m \leq \log_2 n/2$ ，时间复杂度为 $O(\log_2 n)$

求指定语句的语句频度和时间复杂度

3、设 n 是偶数，计算运行下列程序段后， m 的值并给出该程序段的时间复杂度。

```
int m=0,i,j;
for(i=1;i<=n;i++)
    for(j=2*i;j<=n;j++)
        m++;
```

算法基本操作是 $m++$, 内循环从 $2*i \sim n$ 。因此 $2i \leq n, i \leq n/2$, 所以该语句的频度: $T(n) = \sum_{i=1}^{n/2} \sum_{j=2i}^n 1$

$$= \sum_{i=1}^{n/2} n - 2i + 1 = n * \frac{n}{2} - 2 \sum_{i=1}^{n/2} i + \frac{n}{2} = \frac{n^2}{4}, \text{时间复杂度是 } O(n^2)$$


```
Fun(int n)
{
    for(int i=1;i<=n;i++)
        for (j=1;j<=n;j=j+i)
            printf("*");
}
```

外循环i=1, 内循环n次
i=2, 内循环n/2次
i=3, 内循环n/3次
...
i=n, 内循环1次

求和结果为:
 $n(1+1/2+1/3+1/4+.....1/n)$
 $\approx n \log n$

调和级数

求指定语句的语句频度和时间复杂度

- 分析下列算法的时间复杂度

```
Fun(int n)
```

```
{
```

```
int i=1;
```

```
while(i<n)
```

```
{int j=n;
```

```
while(j>0)
```

- ```
j=j/2; //语句频度为
```

- ```
i=i*2; //外层循环语句频度为 $\log n * \log n$ 
```

```
}
```

```
}时间复杂度为 $O(\log n * \log n)$ 
```

1.2 算法分析

1.2.2 算法复杂性分析方法

递归算法的时间复杂度分析

递归算法是采用一种分而治之的方法，把一个“大问题”分解为若干个相似的“小问题”来求解。

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，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。



【例】有以下递归算法

```
void mergesort(int a[],int
i,int j)
{   int m;
    if (i!=j)
    {   m=(i+j)/2;
        mergesort(a,i,m);
        mergesort(a,m+1,j);
        merge(a,i,j,m);
    }
}
```

其中，mergesort()用于数组 $a[0..n-1]$ （设 $n=2^k$ ，这里的 k 为正整数）的归并排序，调用该算法的方式为：

$\text{mergesort}(a, 0, n-1)$

另外 $\text{merge}(a, i, j, m)$ 用于两个有序子序列 $a[i..j]$ 和 $a[j+1..m]$ 的有序合并，是非递归函数，它的时间复杂度为 $O(n)$ （这里 $n=j-i+1$ ）。分析上述调用的时间复杂度。

解：设调用mergesort($a, 0, n-1$)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）：

$$\begin{array}{ll} T(n)=O(1) & \text{当 } n=1 \\ T(n)=2T(n/2)+O(n) & \text{当 } n>1 \end{array}$$

其中， $O(n)$ 为merge()所需的时间，
设为 cn （ c 为正常量）。因此：

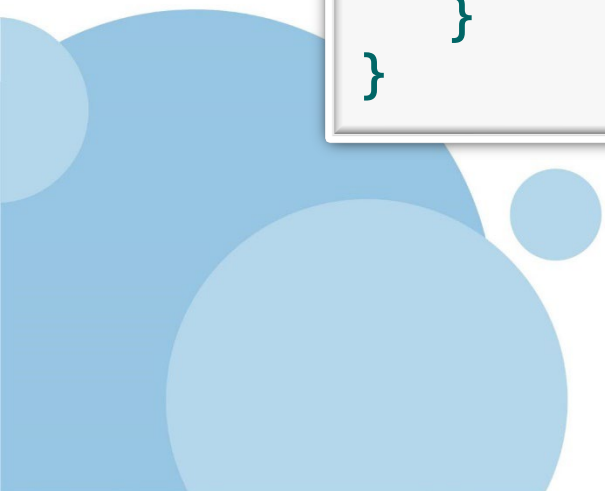
```
void mergesort(int a[],int i,int j)
{
    int m;
    if (i!=j)
    {
        m=(i+j)/2;
        mergesort(a,i,m);
        mergesort(a,m+1,j);
        merge(a,i,j,m);
    }
}
```

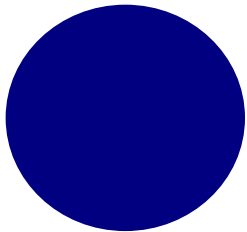
$$\begin{aligned} T(n) &= \\ 2T(n/2)+cn &= 2[2T(n/2^2)+cn/2]+cn=2^2T(n/2^2)+2cn \\ &= 2^3T(n/2^3)+3cn \\ &= \dots \\ &= 2^kT(n/2^k)+kcn \\ &= nO(1)+cn\log_2 n = n+cn\log_2 n && // \text{这里假设} \\ n=2^k, \text{ 则 } k &= \log_2 n \\ &= O(n\log_2 n) \end{aligned}$$




【例】求解汉诺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）：


$$\begin{aligned} T(n) &= O(1) && \text{当 } n=1 \\ T(n) &= 2T(n-1) + 1 && \text{当 } n>1 \end{aligned}$$

```
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);
    }
}
```

$$\begin{aligned} T(n) &= 2[2T(n-2)+1]+1=2^2T(n-2)+1+2^1 \\ &= 2^3T(n-3)+1+2^1+2^2 \\ &= \dots \\ &= 2^{n-1}T(1)+1+2^1+2^2+\dots+2^{n-2} \\ &= 2^n-1 = O(2^n) \end{aligned}$$



1.2 算法分析

1.2.3 算法空间复杂度分析方法

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（辅助变量）所占空间。



1.2 算法分析

1.2.3 算法空间复杂度分析方法

例如，有以下算法，其中临时空间为变量*i*、*maxi*占用的空间。所以，空间复杂度是对一个算法在运行过程中临时占用的存储空间大小的量度，一般也作为问题规模*n*的函数，以数量级形式给出，记作：

$$S(n)=O(g(n))、\Omega(g(n))\text{或}\Theta(g(n))$$

其中渐进符号的含义与时间复杂度中的含义相同。

```
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 $O(1)$ 。

1.2 算法分析

1.2.3 算法空间复杂度分析方法

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。

```
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];  
}
```

1.2 算法分析

1.2.3 算法空间复杂度分析方法

算法空间复杂度的分析方法与前面介绍的时间复杂度分析方法相似。

【例】分析算法的空间复杂度。

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

解：该算法是一个非递归算法，其中只临时分配了*i*、*k*两个变量的空间，它与问题规模*n*无关，所以其空间复杂度均为 $O(1)$ ，即该算法为原时工作算法。

1.2 算法分析

1.2.3 算法空间复杂度分析方法

【例】有如下递归算法，分析调用

`maxelem(a, 0, n-1)`

的空间复杂度。

```
int maxelem(int a[],int i,int j)
{
    int mid=(i+j)/2,max1,max2;
    if (i<j)
    {
        max1=maxelem(a,i,mid);
        max2=maxelem(a,mid+1,j);
        return (max1>max2)?max1:max2;
    }
    else return a[i];
}
```

1.2 算法分析

1.2.3 算法空间复杂度分析方法

解：执行该递归算法需要多次调用自身，每次调用只临时分配3个整型变量的空间 ($O(1)$)。

设调用 `maxelem(a, 0, n-1)` 的空间为 $S(n)$ ，有：

$$\begin{array}{ll} S(n)=O(1) & \text{当 } n=1 \\ S(n)=2S(n/2)+O(1) & \text{当 } n>1 \end{array}$$

```
int maxelem(int a[],int i,int j)
{   int mid=(i+j)/2,max1,max2;
    if (i<j)
    {   max1=maxelem(a,i,mid);
        max2=maxelem(a,mid+1,j);
        return
        (max1>max2)?max1:max2;
    }
    else return a[i];
}
```

$$\begin{aligned} \text{则: } S(n) &= 2S(n/2)+1=2[2S(n/2^2)+1]+1=2^2S(n/2^2)+1+2^1 \\ &= 2^3S(n/2^3)+1+2^1+2^2 \\ &= \dots \\ &= 2^kS(n/2^k)+1+2^1+2^2+\dots+2^{k-1} \quad (\text{设 } n=2^k, \text{ 即 } k=\log_2 n) \\ &= n*1+2^k-1 = 2n-1 = O(n) \end{aligned}$$

1.6 小结

- 小结

本章对算法基本概念、特性及算法设计与分析的基本任务等进行了概述，对算法设计的基本过程和分析算法的准则进行了说明，作为算法复杂度分析的基础知识，回归了算法分析中常用的数学知识并详细介绍了函数阶之间的低阶(O)、高阶(Ω)、同阶(θ)关系定义和分析证明方法。