



中国科学技术大学
University of Science and Technology of China

实用算法设计

主讲：娄文启





- 1. 数据结构，算法**
- 2. 算法的表示**
- 3. 算法的评估**
- 4. 最佳算法选择的决定因素**



- “**数据结构是**数据对象，以及存在于该对象的实例和组成实例的数据元素之间的各种联系。这些联系可以通过定义相关的函数来给出。” —— Sartaj Sahni, 《数据结构、算法与应用》
- “**数据结构是**ADT (抽象数据类型 Abstract Data Type) 的物理实现。” —— Clifford A.Shaffer, 《数据结构与算法分析》
- “**数据结构是**计算机中存储、组织数据的方式。通常情况下，精心选择的数据结构可以带来最优秀率的算法。” —— 中文维基百科

数据结构与数据类型



- 数据类型:
 - 如: 系统定义的int, char等, 用户自定义的struct类型
 - 也可采用typedef将类型名重命名, 以增加代码的可读性。
 -
- int Sqlist[100];
- struct Node{
 int data;
 struct Node *next
};
Typedef struct Node *Link;
Link head;

“顺序表”数据结构的实现:
描述了**100**个整型变量组成的集合,
且隐含着可以利用下标[]来描述两
个整型变量之间的联系.

“单链表”数据结构的实现:
隐含着可以利用指针**next**
来描述两个**struct Node**类
型的变量之间的联系.



例1：如何在书架上摆放图书？

图书的摆放要使得2个相关操作方便实现：

- 操作1：新书怎么插入？
- 操作2：怎么找到某本指定的书？



方法1：随便放

□ 操作1：新书怎么插入？

- 哪里有空放哪里，一步到位！

□ 操作2：怎么找到某本指定的书？

-累死

方法2：按照书名的拼音字母顺序排放

□ 操作1：新书怎么插入？

- 新进一本阿Q正传

□ 操作2：怎么找到某本指定的书？

- 二分查找



方法3：把书架划分成几块区域，某块区域值定摆放某种类别的图书；每种类别内按照书名的拼音字母顺序摆放

□ 操作1：新书怎么插入？

- 先定类别，二分查找确定位置，移出空位

□ 操作2：怎么找到某本指定的书？

- 先定类别，再二分查找
问题：空间如何分配？类别应该分多细？



解决问题方法的效率，跟数据的组织方式有关

- 数据对象在计算机中的组织方式
 - 逻辑结构：线性、树形、图.....
 - 物理存储结构：数组、链表.....
- 数据对象必定与一系列加载其上的操作相关联
- 完成这些操作所用的方法就是算法



例2：计算给定多项式给定点x处的值

$$f(x) = a_0 + a_1x + a_2x^2 + \cdots + a_{n-1}x^{n-1} + a_nx^n$$

方法1：直接计算

```
//n: 多项式系数, a[]: 系数数组, x:给定的点的值
double fun(int n, double a[], double x){
    double p=a[0];
    for(int i=1;i <=n;i++)
        p+=a[i] * myPow(x,i);
    return p;
}
```

myPow函数如何实现



$$f(x) = a_0 + a_1x + a_2x^2 + \cdots + a_{n-1}x^{n-1} + a_nx^n$$

myPow函数：

```
double myPow(double x, int n) {  
    double res=1;  
    if (n<0) { x = 1/x; n = -n; }  
    for(int i=0;i<n;i++)  
        res*=x;  
    return res ;  
}
```

此时多项式计算共需要乘法操作次数

共 $1+2+3+\dots+n$ 次

快速幂：

```
double myPow(double x, int n) {  
    double res=1;  
    if (n<0) { x = 1/x; n = -n; }  
    while (n>0){  
        if(n&1){ res*=x; }  
        x*=x;  
        n>>=1;  
    }  
    return res;  
}
```

复杂度降低



- 对于任何十进制正整数 n ，设其二进制为 " $b_m...b_3b_2b_1$ " (b_i 为二进制某位值, $i \in [1, m]$)，
则有：

- 二进制转十进制： $n = 1b_1 + 2b_2 + 4b_3 + \dots + 2^{m-1}b_m$ (即二进制转十进制公式)；
- 幂的二进制展开： $x^n = x^{1b_1+2b_2+4b_3+\dots+2^{m-1}b_m} = x^{1b_1}x^{2b_2}x^{4b_3}\dots x^{2^{m-1}b_m}$ ；

- 根据以上推导，可把计算 x^n 转化为解决以下两个问题：

- 计算 $x^1, x^2, x^4, \dots, x^{2^{m-1}}$ 的值：循环赋值操作 $x = x^2$ 即可；

- 获取二进制各位 $b_1, b_2, b_3, \dots, b_m$ 的值：循环执行以下操作即可。

- $n \& 1$ (与操作)：判断 n 二进制最右一位是否为 1；
- $n >> 1$ (移位操作)： n 右移一位 (可理解为删除最后一位)。

$$\begin{aligned} n &= 9 \\ &= 1001_b \\ &= 1 \times 1 + 0 \times 2 + 0 \times 4 + 1 \times 8 \\ &\quad (b_1 \times 2^0 + b_2 \times 2^1 + b_3 \times 2^2 + b_4 \times 2^3) \end{aligned}$$

$$x^n = x^9 = x^{1 \times 1} x^{0 \times 2} x^{0 \times 4} x^{1 \times 8}$$

快速幂：

```
double myPow(double x, int n) {
    double res=1;
    if (n<0) { x = 1/x; n = -n; }
    while (n>0){
        if(n&1){ res*=x; }
        x*=x;
        n>>=1;
    }
    return res;
}
```



例2：计算给定多项式给定点x处的值

$$f(x) = a_0 + a_1x + a_2x^2 + \cdots + a_{n-1}x^{n-1} + a_nx^n$$

$$f(x) = a_0 + x(a_1 + x(a_2 + \cdots (a_{n-1} + xa_n) \cdots))$$

方法2：提取公因子

```
double myfun(int n, double a[], double x){  
    double p = a[n];  
    for (int i=n; i>0; i--)  
        p = a[i-1] + x*p;  
    return p;  
}
```

相同数据结构，不同操作方法

关于二进制的小例子



- 问题：对于一个字节（8bit）的无符号整型变量，求其二进制表示中 1 的个数。

解法一

利用整数数据除法的特点。
相除和判断余数的值来分析

以 10100010 为例：

第一次除以 2，商为 1010001，余为 0。

第二次除以 2，商为 101000，余为 1。

```
int Count(BYTE v)
{
    int num = 0;
    while (v)
    {
        if(v % 2 == 1)
        {
            num++;
        }
        v = v / 2;
    }
    return num;
}
```

O(logN) N为数据位宽

关于二进制的小例子



- 问题：对于一个字节（8bit）的无符号整型变量，求其二进制表示中 1 的个数。

解法二

向右移位操作达到相除的目的
判断最后一位是否为1

将八位数字与 00000001 进行与操作，
若为1，则表示当前八位数字的最后一
位为1，否则为0

```
int Count(BYTE v) {  
    int num = 0;  
    while (v) {  
        num += v & 0x01; // 记录当前位是否有1  
        v >>= 1;          // 右移一位  
    }  
    return num;  
}
```

$O(\log N)$

关于二进制的小例子



- 问题：对于一个字节（8bit）的无符号整型变量，求其二进制表示中 1 的个数。

解法三

能否进一步降低复杂度？

使其仅与 “1” 的个数有关。

判断有一个 “1”的情况：01000000

判断方法：通过判断该数是否为 2 的整数次幂

设计操作：与操作（01000000 和 00111111）

```
int Count(int v) {  
    int num = 0;  
    while (v) {  
        v &= (v - 1);  
        num++;  
    }  
    return num;  
}
```

当前数与自身减一的结果进行与操作，消除最低位的 “1”

$O(\log M)$, M 为 1 的个数

关于二进制的小例子



- 问题：对于一个字节（8bit）的无符号整型变量，求其二进制表示中 1 的个数。

解法四

能否进一步加快执行效率

使用分支操作，列举所有情况？

```
int Count(int v){  
    int num = 0;  
    switch (v)  
    {  
        case 0x00:  
            num = 0;  
            break;  
        case 0x01:  
        case 0x02:  
        case 0x04:  
        case 0x40:  
            ...  
        case 0x80:  
            num = 1;  
            break;  
        case 0x03:  
        case 0x0c:  
        case 0x30:  
        case 0xc0:  
            num = 2;  
            break;  
        //...}  
    return num;}
```

O(255)

关于二进制的小例子



- 问题：对于一个字节（8bit）的无符号整型变量，求其二进制表示中 1 的个数。

• 解法五

直接查找表，空间换时间

O(1)的时间复杂度

```
int countTable[256] = { // 预定义的结果表
    0, 1, 1, 2, 1, 2, 2, 3, 1, 2, 2, 3, 3, 3, 3, 4, 1, 2,
    2, 3, 2, 3, 3, 4, 2, 3, 3, 4, 4, 4, 4, 5, 1, 2, 2, 3, 2, 3,
    3, 4, 2, 3, 3, 4, 4, 4, 4, 5, 3, 4, 4, 4, 5, 4, 4, 5, 5, 5, 6, 1,
    2, 2, 3, 2, 3, 3, 4, 2, 3, 3, 4, 4, 4, 4, 5, 3, 4, 4, 5, 4,
    4, 5, 5, 6, 2, 3, 3, 4, 3, 4, 4, 4, 5, 3, 4, 4, 5, 4, 4, 5,
    5, 6, 3, 4, 4, 5, 4, 4, 5, 5, 6, 4, 5, 5, 5, 6, 5, 6, 6, 7,
    1, 2, 2, 3, 2, 3, 3, 4, 2, 3, 3, 4, 4, 4, 4, 4, 5, 1, 2, 2, 3,
    2, 3, 3, 4, 2, 3, 3, 4, 4, 4, 4, 5, 3, 4, 4, 4, 5, 4, 4, 5,
    6, 2, 3, 3, 4, 3, 4, 4, 5, 3, 4, 4, 4, 5, 4, 4, 5, 5, 5, 6, 3,
    4, 4, 5, 4, 4, 5, 5, 6, 4, 5, 5, 5, 6, 5, 6, 6, 7, 2, 3, 3,
    4, 3, 4, 4, 5, 3, 4, 4, 5, 4, 4, 5, 5, 5, 6, 3, 4, 4, 5, 4,
    4, 5, 5, 6, 4, 5, 5, 6, 5, 6, 7, 3, 4, 4, 5, 4, 4, 5, 4, 5,
    5, 6, 4, 5, 5, 6, 5, 6, 7, 4, 5, 5, 5, 6, 5, 6, 6, 7, 5,
    6, 6, 7, 7, 7, 8};}

int Count(BYTE v){
    // check parameter
    return countTable[v];
}
```

什么是算法



- 描述了特定问题的**有限求解步骤**；
 - 指为了解决特定问题，而**操作数据**的方式；
 - 指求解问题的**策略**。
-
- 例子：
 - 经典问题1：计算 $1,2,3,\dots,50$ 的和。
 - 经典问题2：将大象放到冰箱里。
 - 经典问题3：计算 $[m+(m+1)+\dots+(m+n-1)]$ 。

算法的特征



有穷性: 算法在执行有穷步之后能结束;

确定性: 每一条指令有确定的含义;

可行性: 每一操作都可以通过已实现的基本运算执行有限次来实现

输入: 接受一些输入 (有些情况可能没有) ;

输出: 必须有一个或者多个输出;

每一条指令必须:

- 有充分明确的目标, 不可以有歧义
- 计算机能处理的范围之内
- 描述应不依赖于任何一种计算机语言以及具体的实现手段



● 自然语言

- 贴近人类思维，易于理解主旨
- 语言描述繁琐、不严谨，容易歧义



ChatGPT

是的，我了解选择排序和插入排序。它们都是简单的排序算法，通常用于小型数据集或部分有序的数据。

1. 选择排序 (Selection Sort) :

- 选择排序的基本思想是每次从待排序的数据中选择最小（或最大）的元素，然后将其放在已排序序列的末尾（或开头），依次进行，直至整个序列有序。
- 具体步骤：
 - 1.1. 在未排序序列中找到最小元素；
 - 1.2. 将最小元素交换到未排序序列的起始位置；
 - 1.3. 不断重复这个过程，直到整个序列有序。

算法的表示



● 程序语言

- 精确表达，规避歧义
- 不同语言之间存在差异，关注细节

选择排序

```
void select_sort(int *a, int n) {
    for (int i = 0; i < n - 1; i++) {
        int cur_min = a[i];
        int cur_min_pos = i;
        for (int j = i; j < n; j++) {
            if (cur_min > a[j]) {
                cur_min = a[j];
                cur_min_pos = j;
            }
        }
        int temp = a[i];
        a[i] = a[cur_min_pos];
        a[cur_min_pos] = temp;
    }
    return;
}
```

C语言

```
def select_sort(a, n):
    for i in range(0, n - 1):
        cur_min = a[i]
        cur_min_pos = i
        for j in range(i, n):
            if cur_min > a[j]:
                cur_min = a[j]
                cur_min_pos = j
        temp = a[i]
        a[i] = a[cur_min_pos]
        a[cur_min_pos] = temp
```

Python语言



- 伪代码
- 书写约定

兼顾两者优势

Algorithm 1: Selection Sorting

```
1 Input: array A: [a0, a1, ..., an-1]
2 Output: array A': [a'0, a'1, ..., a'n-1], s.t., a'0 ≤ a'1 ≤ ... ≤ a'n-1
3 for i ← 0 to n - 2 do
4     cur_min ← A[i]
5     cur_min_pos ← i
6     for j ← i + 1..do
7         // record the index and current minimum
8         if A[j] < cur_min then
9             cur_min ← A[j]
10            cur_min_pos ← j
11        end
12    end
13    Swap A[i] and A[cur_min_pos]
14 end
```

定义算法的输入输出

循环语块缩进

必要的注释语句

不关注交换细节

选择排序示例



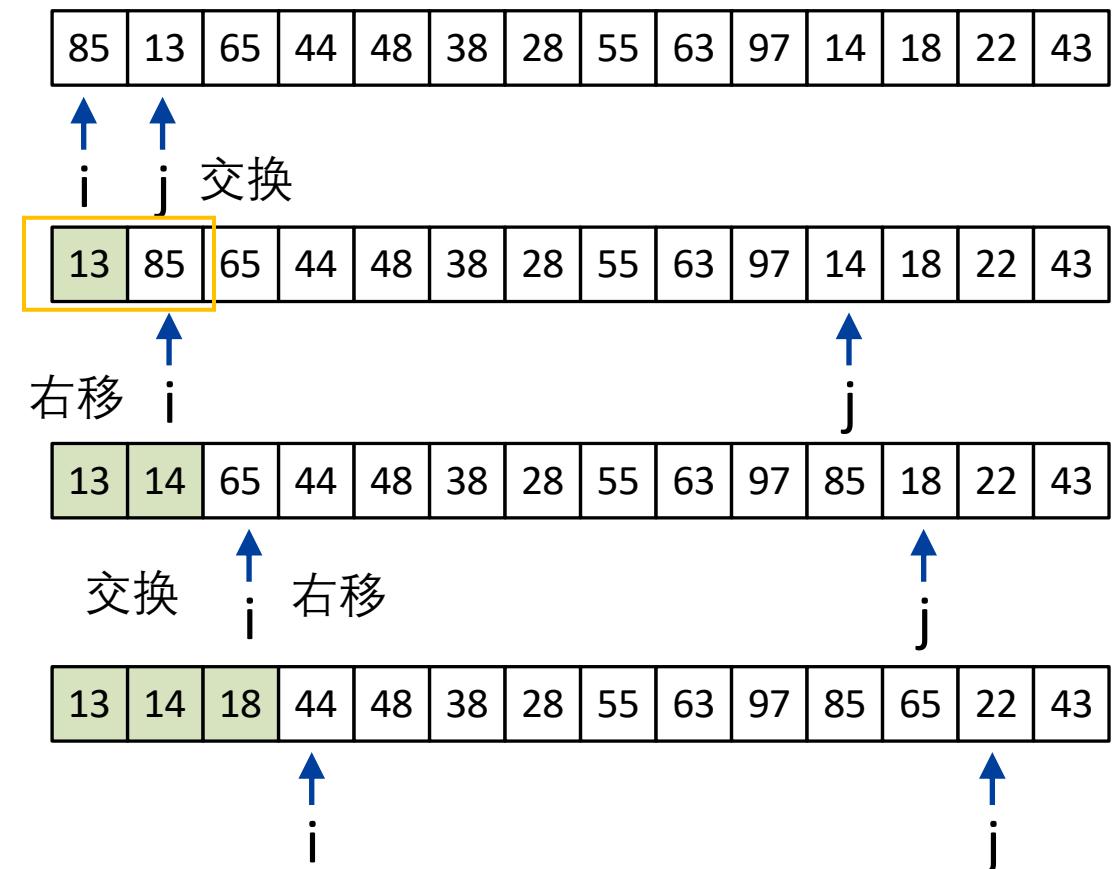
● 伪代码

● 示例解读

选择排序的思想是从未排序的序列中找到最小（或最大）的元素，然后将其放到已排序序列的末尾（或开头）

Algorithm 1: Selection Sorting

```
1 Input: A:  $[a_0, a_1, \dots, a_{n-1}]$ 
2 Output:  $A'': [a'_0, a'_1, \dots, a'_{n-1}]$ , s.t.,  $a'_0 \leq a'_1 \leq \dots \leq a'_{n-1}$ 
3 for  $i \leftarrow 0$  to  $n - 2$  do
4   cur_min  $\leftarrow A[i]$ 
5   cur_min_pos  $\leftarrow i$ 
6   for  $j \leftarrow i + 1$  to  $n - 1$  do
7     // record the index and current minimum
8     if  $A[j] < cur\_min$  then
9       cur_min  $\leftarrow A[j]$ 
10      cur_min_pos  $\leftarrow j$ 
11    end
12  end
13  Swap  $A[i]$  and  $A[cur\_min\_pos]$ 
14 end
```



插入排序示例



● 伪代码

● 示例解读

将未排序的元素逐个插入到已排序的部分，每次插入一个元素，保持已排序部分的有序性

Algorithm 2: Insertion Sort

```
1 Input:  $A: [a_0, a_1, \dots, a_{n-1}]$ 
2 Output:  $A': [a'_0, a'_1, \dots, a'_{n-1}]$ , s.t.,  $a'_0 \leq a'_1 \leq \dots \leq a'_{n-1}$ 
3 for  $i \leftarrow 1$  to  $n - 1$  do
4    $key \leftarrow A[i];$ 
5    $j \leftarrow i - 1;$ 
6   // insert A[i] to the sorted sub-array
7   while  $j \geq 0$  and  $A[j] > key$  do
8      $A[j + 1] \leftarrow A[j];$ 
9      $j \leftarrow j - 1;$ 
10  end
11   $A[j + 1] \leftarrow key;$ 
12 end
```

具体步骤：

1. 从第一个元素开始，有序序列；
2. 取出下一个元素，从后向前扫描有序序列；
3. 若该元素大于新元素，将已排序元素移到下一位；
4. 重复步骤3，直到找到已排序元素小于或等于新元素的位置；
5. 将新元素插入到该位置后；
6. 重复步骤2-5，直至整个序列有序。

算法评估



- 基本要求：
 - 正确性
 - 可读性
 - 健壮性（异常处理机制）
- 更实用的要求：（性能）
 - 高效率（时间复杂度低）
 - 低存储量（内存开销小）

```
if ( argc < 3 || argc > 5 )  
{  
    fprintf ( stderr,  
              "Usage: BUFSIZE infile outfile [insize  
[outsize]]\n" );  
    return ( EXIT_FAILURE );  
}
```

健壮性处理

算法评估



- 时间复杂度: 衡量算法运行得有多快。

1) 如何度量? (估算和实际测量)

- 通常不依赖于计时(实际物理平台), 而依赖于性能方程, 以显示输入的大小/规模与性能的关系。
- 机器的运算速度影响算法的运行时间

机器	运算速度	运行算法	运行时间
天河三号	百亿亿次/秒	插入排序	无法公平比较
个人电脑	十亿次/秒	选择排序	



分析算法的运行时间应独立于机器



算法分析的原则

- 归纳基本操作类型
 - 赋值、比较、运算
- 统一机器性能
 - 假设基本操作代价为1

统一机器性能后，算法运行时间依赖于问题输入规模与示例



算法分析的原则

- ◆ 相同输入规模，但输入实例影响程序执行情况

Algorithm 2: Insertion Sort

```
1 Input:  $A: [a_0, a_1, \dots, a_{n-1}]$ 
2 Output:  $A': [a'_0, a'_1, \dots, a'_{n-1}]$ , s.t.,  $a'_0 \leq a'_1 \leq \dots \leq a'_{n-1}$ 
3 for  $i \leftarrow 1$  to  $n - 1$  do
4    $key \leftarrow A[i];$ 
5    $j \leftarrow i - 1;$ 
6   // insert  $A[i]$  to the sorted sub-array
7   while  $j \geq 0$  and  $A[j] > key$  do
8      $A[j + 1] \leftarrow A[j];$ 
9      $j \leftarrow j - 1;$ 
10  end
11   $A[j + 1] \leftarrow key;$ 
12 end
```

插入排序算法伪代码

循环次数未知

最好?

最坏?

一般?



算法分析的原则

- ◆ 相同输入规模，但输入实例影响程序执行情况

- 插入排序的最好情况：数组升序
 - 比较次数： $1 + 1 + 1 + \dots + 1 = n - 1$

$$\underbrace{\qquad\qquad}_{n-1}$$

4	8	13	14	17	18	21	22	24	28	32	37	40	40	47	48
---	---	----	----	----	----	----	----	----	----	----	----	----	----	----	----

- 插入排序的最坏情况：数组降序

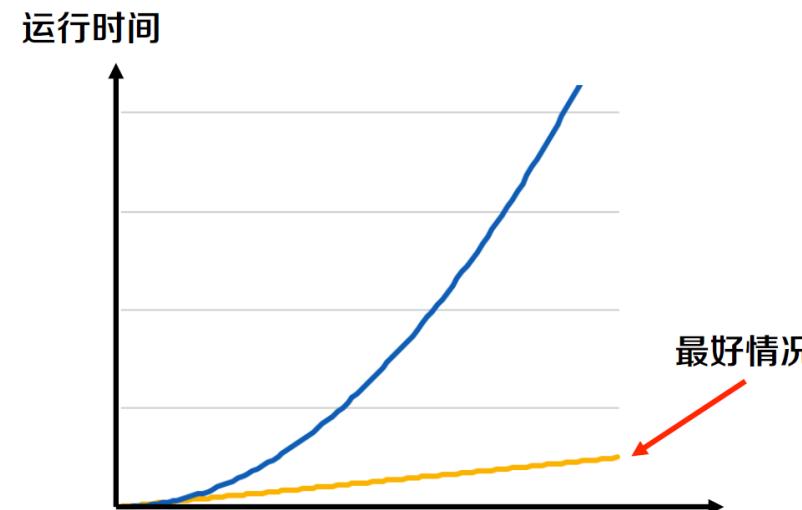
- 比较次数： $1 + 2 + 3 + \dots + (n - 1) = \frac{n(n-1)}{2}$

48	47	40	40	37	32	28	24	22	21	18	17	14	13	8	4
----	----	----	----	----	----	----	----	----	----	----	----	----	----	---	---



算法分析的原则

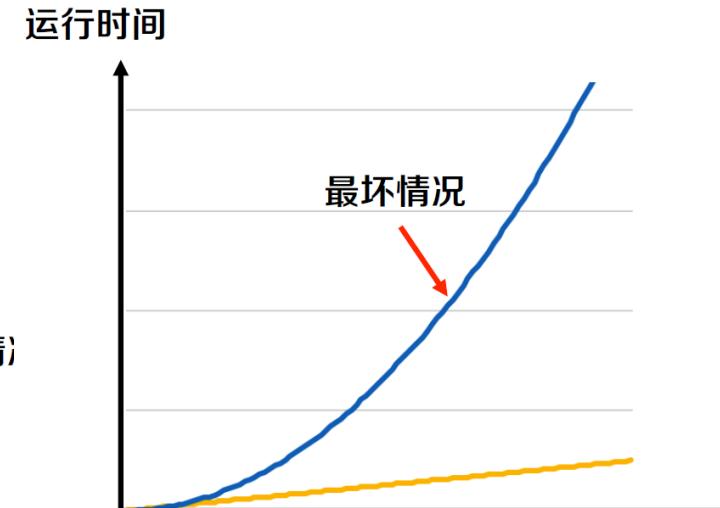
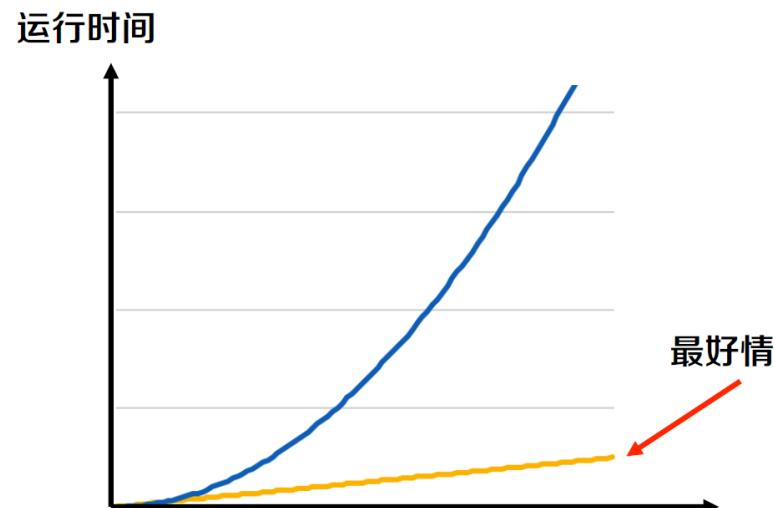
输入情况	情况说明
最好情况	不常出现，不具普遍性





算法分析的原则

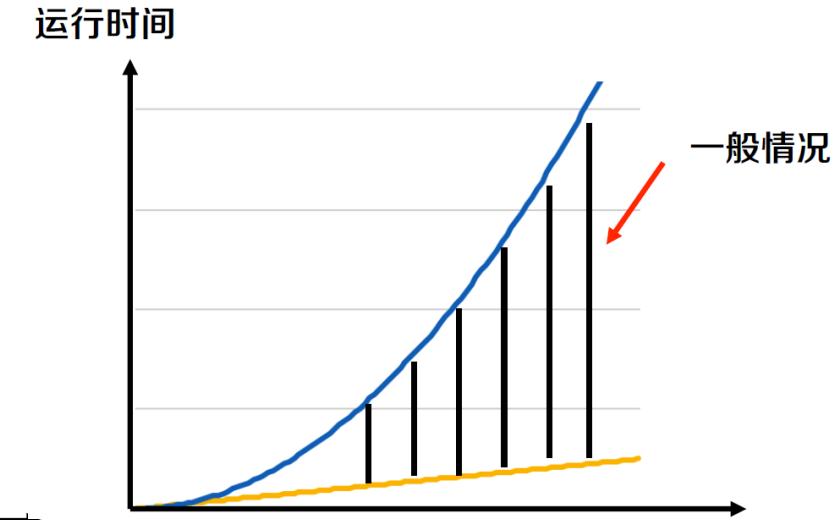
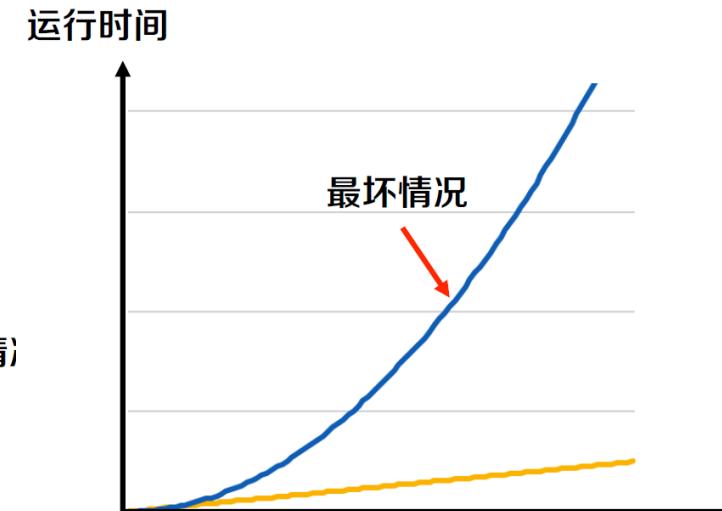
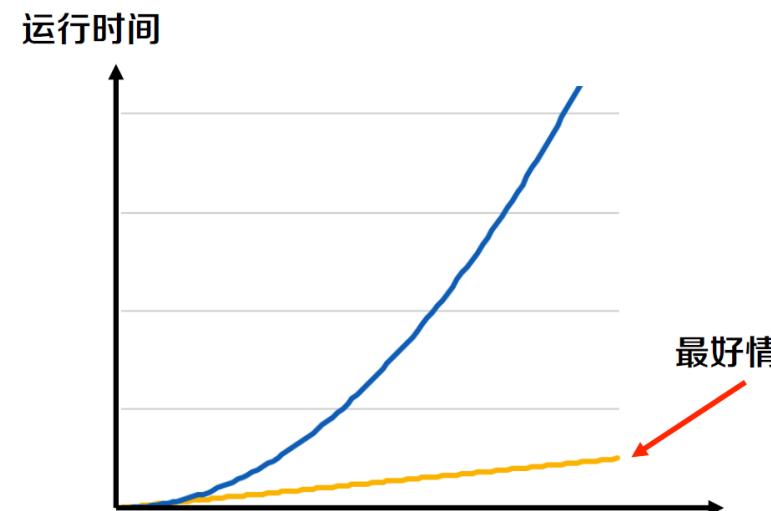
输入情况	情况说明
最好情况	不常出现，不具普遍性
最坏情况	确定上界，更具一般性





算法分析的原则

输入情况	情况说明
最好情况	不常出现，不具普遍性
最坏情况	确定上界，更具一般性
一般情况	情况复杂，分析难度大





最坏情况分析

Algorithm 2: Insertion Sort

```
1 Input:  $A: [a_0, a_1, \dots, a_{n-1}]$ 
2 Output:  $A': [a'_0, a'_1, \dots, a'_{n-1}]$ , s.t.,  $a'_0 \leq a'_1 \leq \dots \leq a'_{n-1}$ 
3 for  $i \leftarrow 1$  to  $n - 1$  do
4    $key \leftarrow A[i];$ 
5    $j \leftarrow i - 1;$ 
6   // insert  $A[i]$  to the sorted sub-array
7   while  $j \geq 0$  and  $A[j] > key$  do
8      $A[j + 1] \leftarrow A[j];$ 
9      $j \leftarrow j - 1;$ 
10  end
11   $A[j + 1] \leftarrow key;$ 
12 end
```

$n-1$ 次

$\sum_{k=1}^{n-1} k$ 次

$n-1$ 次

求和得到最坏复杂度 $O(n^2)$



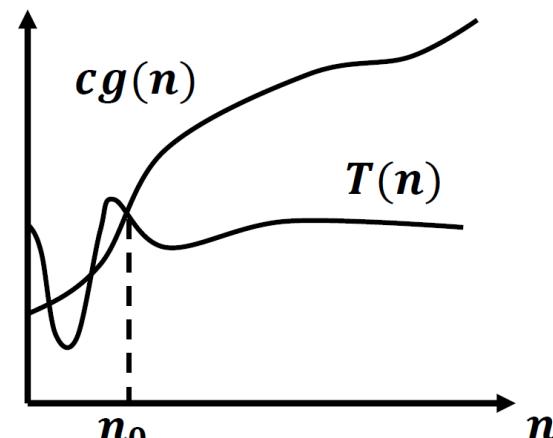
渐近分析：渐近上界

O 记号

定义：

- 对于给定的函数 $g(n)$, $O(g(n))$ 表示以下函数的集合:

$$O(g(n)) = \{T(n) : \exists c, n_0 > 0, \text{使得} \forall n \geq n_0, 0 \leq T(n) \leq cg(n)\}$$



$$T(n) = O(g(n))$$



渐近分析：渐近紧确界

Θ记号

定义：

- 对于给定的函数 $g(n)$, $\Theta(g(n))$ 表示以下函数的集合:

$$\Theta(g(n)) = \{T(n) : \exists c_1, c_2, n_0 > 0, \text{使得} \forall n \geq n_0, c_1 g(n) \leq T(n) \leq c_2 g(n)\}$$

Θ记号示例

- $T(n) = \frac{3}{2}n^2 + \frac{7}{2}n - 4 = ?$
- 令 $n_0 = 2$, 当 $n \geq n_0$ 时, 有
- $\frac{3}{2}n^2 + \frac{7}{2}n - 4 \geq \frac{3}{2}n^2 \geq n^2$
- $\frac{3}{2}n^2 + \frac{7}{2}n - 4 \leq \frac{3}{2}n^2 + \frac{7}{2}n^2 + n^2 = 6n^2$
- 故存在 $c_1 = 1, c_2 = 6, n_0 = 2$, 使得 $\forall n \geq n_0, c_1 n^2 \leq T(n) \leq c_2 n^2$



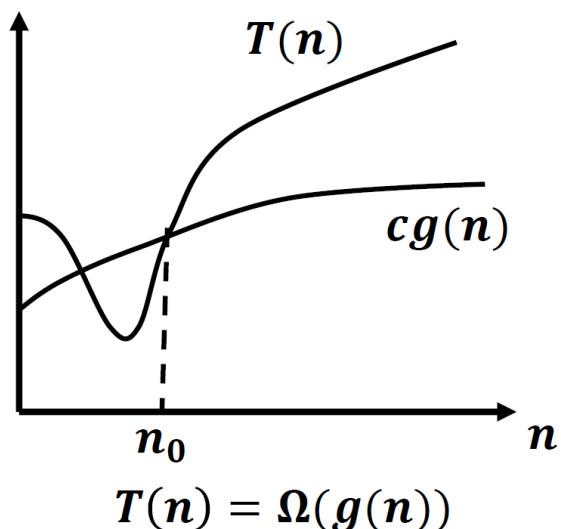
渐近分析：渐近下界

Ω记号

定义：

- 对于给定的函数 $g(n)$, $\Omega(g(n))$ 表示以下函数的集合:

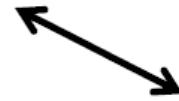
$$\Omega(g(n)) = \{T(n) : \exists c, n_0 > 0, \text{使得} \forall n \geq n_0, 0 \leq cg(n) \leq T(n)\}$$



算法评估



渐近记号	名称	输入情况	情况说明
Θ	渐近精确界	最好情况	不常出现，不具普遍性
O	渐近上界	最坏情况	确定上界，更具一般性
Ω	渐近下界		



算法运行时间称为算法的时间复杂度，通常使用渐近记号 O 表示

算法评估



复杂度的直观体现

函数	1	2	4	8	16	32
1	1	1	1	1	1	1
$\log n$	0	1	2	3	4	5
n	1	2	4	8	16	32
$n \log n$	0	2	8	24	64	160
n^2	1	4	16	64	256	1024
n^3	1	8	64	512	4096	32768
2^n	2	4	16	256	65536	4294967296
$n!$	1	2	24	40326	2092278988000	26313×10^{33}

$$O(1) < O(\lg N) < O(N) < O(N * \lg N) < O(N^2) < O(N^k) < O(2^N)$$

常量阶、对数阶、线性阶、 $N * \lg N$ 阶、平方阶、多项式阶、指数阶 ($\lg N$ 这里指以2为底的对数)

算法评估



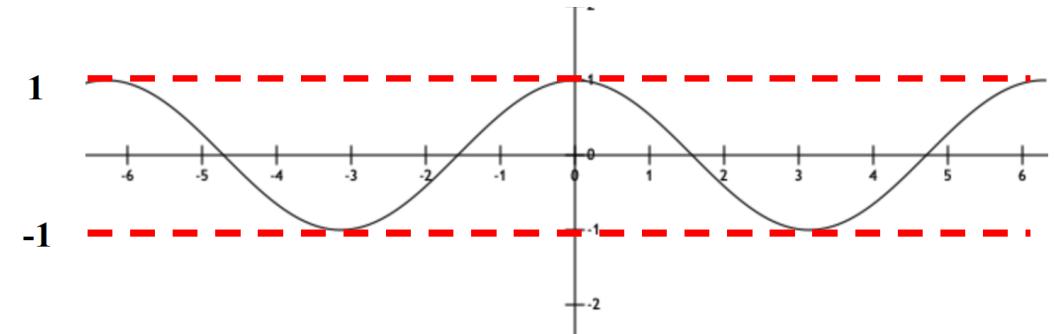
● O 记号示例

- $\cos n = O(1)$
- $\log_7 n = \frac{\log_2 n}{\log_2 7} = O(\log_2 n) = O(\log n)$
- $\sum_{i=1}^n \frac{1}{i}$ (假设 n 是 2 的整数幂)

$$\begin{aligned}&= \frac{1}{1} + \frac{1}{2} + \frac{1}{3} + \frac{1}{4} + \frac{1}{5} + \frac{1}{6} + \frac{1}{7} + \frac{1}{8} + \frac{1}{9} + \frac{1}{10} + \dots + \frac{1}{n} \\&< \frac{1}{1} + \frac{1}{2} + \frac{1}{2} + \frac{1}{4} + \frac{1}{4} + \frac{1}{4} + \frac{1}{4} + \frac{1}{8} + \frac{1}{8} + \dots + \frac{1}{\frac{n}{2}} + \frac{1}{n} \\&= \frac{1}{1} + 2 \cdot \left(\frac{1}{2}\right) + 4 \cdot \left(\frac{1}{4}\right) + 8 \cdot \left(\frac{1}{8}\right) + \dots + \frac{n}{2} \left(\frac{1}{\frac{n}{2}}\right) + \frac{1}{n} = \frac{1}{n} + \sum_{j=0}^{\log n - 1} 1 = \log n + \frac{1}{n} = O(\log n)\end{aligned}$$

$\underbrace{\qquad\qquad\qquad}_{\log n} \qquad\qquad\qquad n = 2^k$

$$2^0 + 2^1 + \dots + 2^{k-1} = 2^k - 1 = n - 1$$



算法评估



● Ω 记号示例

- $\sum_{i=1}^n \frac{1}{i}$ (假设n是2的整数幂)

$$= \frac{1}{1} + \frac{1}{2} + \frac{1}{3} + \frac{1}{4} + \frac{1}{5} + \frac{1}{6} + \frac{1}{7} + \frac{1}{8} + \frac{1}{9} + \frac{1}{10} + \cdots + \frac{1}{n}$$

$$> \frac{1}{1} + \frac{1}{2} + \frac{1}{4} + \frac{1}{4} + \frac{1}{8} + \frac{1}{8} + \frac{1}{8} + \frac{1}{8} + \frac{1}{16} + \frac{1}{16} + \cdots \frac{1}{n}$$

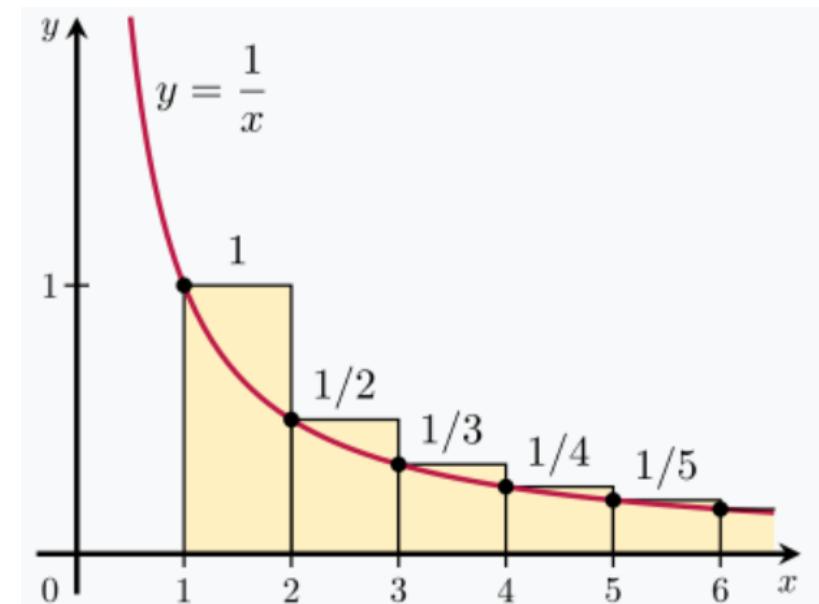
$$= \frac{1}{1} + \frac{1}{2} + 2 \cdot \left(\frac{1}{4}\right) + 4 \cdot \left(\frac{1}{8}\right) + 8 \cdot \left(\frac{1}{16}\right) + \cdots + \frac{n}{2} \left(\frac{1}{n}\right)$$

$\log n$

$$= 1 + \sum_{j=1}^{\log n} \frac{1}{2}$$

$$= 1 + \frac{1}{2} \log n$$

$$= \Omega(\log n)$$



$$\sum_{i=1}^n \frac{1}{i} > \log(n + 1)$$



时间复杂度分析 – 序列求和法

- 等差、等比数列和调和级数求和

等差数列求和公式

$$\sum_{k=1}^n a_k = \frac{n(a_1 + a_n)}{2}$$

等比数列求和公式

$$\sum_{k=0}^n aq^k = \frac{a(1-q^{n+1})}{1-q} \quad \sum_{k=0}^{\infty} a_k q^k = \frac{a}{1-q} \quad (q < 1)$$

调和级数求和公式

$$\sum_{k=1}^n \frac{1}{k} = \log n + O(1)$$

算法评估示例-1



序列求和示例

$$\sum_{t=1}^k t \cdot 2^{t-1} = \sum_{t=1}^k t \cdot (2^t - 2^{t-1})$$

拆项

$$= \sum_{t=1}^k t \cdot 2^t - \sum_{t=1}^k t \cdot 2^{t-1} = \sum_{t=1}^k t \cdot 2^t - \sum_{t=0}^{k-1} (t+1) \cdot 2^t$$

变限

$$= \sum_{t=1}^k t \cdot 2^t - \sum_{t=0}^{k-1} t \cdot 2^t - \sum_{t=0}^{k-1} 2^t$$

抵消

$$= k \cdot 2^k - (2^k - 1) = (k - 1) \cdot 2^k + 1$$

更形象一些? 展开!

算法评估示例-1



算法平均时间复杂度—二分查找为例

算法 0.1 二分查找算法 BinarySearch(T, l, r, x)

Result: 返回 j

输入: 数组 T , 下标从 l 到 r ; 数 x

输出: j

```
1  $l \leftarrow 1; r \leftarrow n;$ 
2 while  $l \leq r$  do
3    $m \leftarrow \lfloor(l + r)/2\rfloor;$ 
4   if  $T[m] == x$  then
5     return  $m$ ; //x 中位元素
6   else if  $T[m] > x$  then
7      $r \leftarrow m - 1$ 
8   else  $l \leftarrow m + 1$ 
9 end
10 return 0;
```

假设 $n = 2^k - 1$, 输入有 $2n + 1$ 种情况:

$x = T[1]$

$x = T[2]$

...

$x = T[n - 1]$

$x = T[n]$

$x < T[1]$

$T[1] < x < T[2]$

...

$T[n - 1] < x < T[n]$

$T[n] < x$

$x \in T$

$x \notin T$

算法评估示例-1



算法平均时间复杂度—二分查找为例

假设二分查找序列长度为 $n = 2^k - 1$, (输入有 $2n + 1$ 种情况):



比较1次: 1个



比较2次: 2个

比较t次: ?个



比较3次: 4个

对于 $t = 1, 2, \dots, k - 1$, 比较 t 次: 2^{t-1} 个

对于 $t = k$ 次的输入, 有 $2^{k-1} + n + 1$ 个

总次数: 对每个输入乘以次数并求和

算法评估示例-1



算法平均时间复杂度—二分查找为例

假设 $n = 2^k - 1$, 各种输入概率相等

$$\begin{aligned} A(n) &= \frac{1}{2n+1} \left[\sum_{t=1}^{k-1} t \cdot 2^{t-1} + k(2^{k-1} + n + 1) \right] \\ &= \frac{1}{2n+1} \left[\sum_{t=1}^k t \cdot 2^{t-1} + k(n+1) \right] = \frac{1}{2n+1} [(k-1)2^{k-1} + 1 + k(n+1)] \\ &= \frac{k2^k - 2^k + 1 + k2^k}{2^{k+1} - 1} \approx k - \frac{1}{2} = \lfloor \log n \rfloor + \frac{1}{2} \end{aligned}$$

算法评估示例-2



时间复杂度分析 -- 估计和式上界的放大法

$$1. \sum_{k=1}^n a_k \leq n a_{max}$$

2. 假设存在常数 $r < 1$, 使得对一切 $k \geq 0$ 有 $\frac{a_{k+1}}{a_k} \leq r$ 成立

放大法的例子

估计 $\sum_{k=1}^n \frac{k}{3^k}$ 的上界.

$$\sum_{k=0}^n a_k \leq \sum_{k=0}^{\infty} a_0 r^k = a_0 \sum_{k=0}^{\infty} r^k = \frac{a_0}{1-r}$$



$$a_1 \leq a_0 r, a_2 \leq a_1 r \leq a_0 r^2, \dots$$

$$\frac{a_{k+1}}{a_k} = \frac{\frac{k+1}{3^{k+1}}}{\frac{k}{3^k}} = \frac{1}{3} \frac{k+1}{k} \leq \frac{2}{3}$$

$$\sum_{k=1}^n \frac{k}{3^k} \leq \sum_{k=1}^{\infty} \frac{1}{3} \left(\frac{2}{3}\right)^{k-1} = \frac{1}{3} \frac{1}{1 - \frac{2}{3}} = 1$$

调和级数的上界求解时也使用了放大法



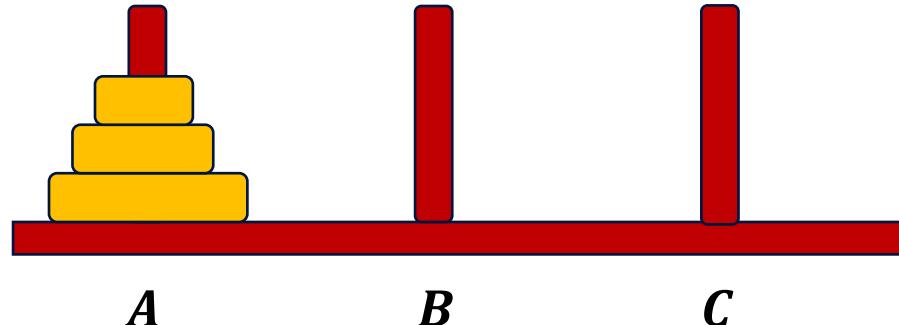
时间复杂度分析 – 迭代法

- 不断用递推方程的右部替换左部
- 每次替换，随着 n 的降低在和式中多出一项
- 直到出现初值停止迭代
- 将初值代入并对和式求和
- 可用数学归纳法验证解的正确性

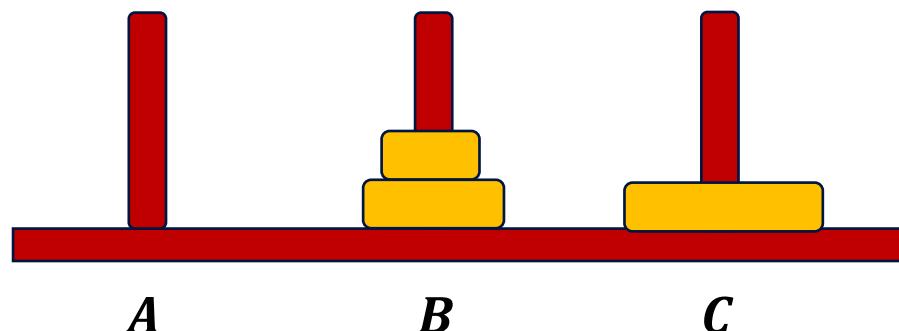
算法评估示例-3



时间复杂度分析 – Hanoi塔算法



n 个盘子从大到小顺序放在 A 柱上，要把它们从 A 移动到 C ，每次移动 1 个盘子，移动时不允许大盘压在小盘上，设计一种移动方法



算法 1.2 Algorithm Hanoi(A, C, n) // n 个盘子从 A 移动到 C

```
1 if  $n = 1$  then
2   | move( $A, C$ ) // 1 个盘子从  $A$  移动到  $C$ 
3 end
4 else
5   | Hanoi( $A, B, n - 1$ );
6   | move( $A, C$ );
7   | Hanoi( $B, C, n - 1$ );
8 end
```

设 n 个盘子的移动次数为 $T(n)$ ，则有：

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

$$T(1) = 1$$

算法评估示例-3



时间复杂度分析 – Hanoi塔算法

Hanoi 塔算法

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

$$T(1) = 1$$

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

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

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

= ...

$$= 2^{n-1}T(1) + 2^{n-2} + 2^{n-3} + \dots + 2 + 1$$

$$= 2^{n-1} + 2^{n-1} - 1$$

$$= 2^n - 1$$

代入初值
求和

假设每秒能移一个盘子，64个盘子需要多少时间？

答案是5000亿年！

如果速度达到千万亿次/秒，则需4个多小时完成

有没有更好的算法？

没有！这是一个难解的问题，不存在多项式时间的算法！



算法平均时间复杂度—二分归并排序算法

算法 1.3 二分归并排序算法 MergeSort(A, p, r)

输入: 数组 A , 下标从 p 到 r

输出: 按递增顺序排列的数组 A

```
1 if  $p < r$  then
2    $q \leftarrow \lfloor (p + r)/2 \rfloor$ ;
3   MergeSort( $A, p, q$ );
4   MergeSort( $A, q + 1, r$ );
5   Merge( $A, p, q, r$ );
6 end
```

换元

假设 $n = 2^k$, 递推方程如下:

$$W(n) = 2W\left(\frac{n}{2}\right) + n - 1$$

$$W(1) = 0$$

换元:

$$W(2^k) = 2W(2^{k-1}) + 2^k - 1$$

$$W(0) = 0$$

这里 $W(\cdot)$ 和之前的 $T(\cdot)$ 只是为了区分, 意义相同



算法平均时间复杂度—二分归并排序算法

迭代求解

$$W(2^k) = 2W(2^{k-1}) + 2^k - 1$$

$$W(0) = 0$$

$$\begin{aligned} W(n) &= 2W(2^{k-1}) + 2^k - 1 \\ &= 2[2W(2^{k-2}) + 2^{k-1} - 1] + 2^k - 1 \\ &= 2^2W(2^{k-2}) + 2^k - 2 + 2^k - 1 \\ &= 2^2[2W(2^{k-3}) + 2^{k-2} - 1] + 2^k - 2 + 2^k - 1 \\ &= 2^3W(2^{k-3}) + \boxed{2^k} + \boxed{-2^2} + \boxed{2^k} + \boxed{-2^1} + \boxed{2^k} + \boxed{-2^0} \end{aligned}$$

$$\begin{aligned} &= \dots \\ &= 2^k W(1) + k2^k - (2^{k-1} + 2^{k-2} + \dots + 2 + 1) \\ &= k2^k - 2^k + 1 \\ &= n \log n - n + 1 \end{aligned}$$

时间复杂度分析 - 差消法

高阶递推方程

- 对于高阶递推方程先要用差消法化简为一阶方程
- 迭代求解

快速排序

- 假设 $A[p..r]$ 的元素彼此不等，以首元素 $A[1]$ 对数组 $A[p..r]$ 划分，使得：
 - 小于 x 的元素放在 $A[p..q - 1]$
 - 大于 x 的元素放在 $A[q + 1..r]$
- 递归对 $A[p..q - 1]$ 和 $A[q + 1..r]$ 排序

工作量：子问题工作量+划分工作量

算法评估示例-4



快速排序-单趟

单向划分 & 将数组中第一个元素作为基准

【例】假设有8个记录，关键字的初始序列为 {45,34,67,95,78,12,26,45}

一 趟 快速 排序 过 程 :

1.	45	34	67	95	78	12	26	45	(A1为空数组)
2.	45	34	67	95	78	12	26	45	(A1中有1元素)
3.	45	34	67	95	78	12	26	45	(A1中有1元素)
4.	45	34	67	95	78	12	26	45	(A1中有1元素)
5.	45	34	67	95	78	12	26	45	(A1中有1元素)
6.	45	34	12	95	78	67	26	45	(A1中有2元素)
7.	45	34	12	26	78	67	95	45	(A1中有3元素)
8.	45	34	12	26	78	67	95	45	(A1中有3元素)
9.	26	34	12	45	78	67	95	45	(将基准放中间)

m: 表示存放小于基准的元素的子数组A1的最末元素的下标。

一趟将数组分为两个子部分。



时间复杂度分析 - 差消法

根据开头元素在排序数列中的大小关系

- 有 n 种可能的输入

x 排好序位置	子问题 1 规模	子问题 2 规模
1	0	$n - 1$
2	1	$n - 2$
3	2	$n - 3$
...
$n - 1$	$n - 2$	1
n	$n - 1$	0

- 对每个输入，划分的比较次数都是 $n - 1$

工作量总和

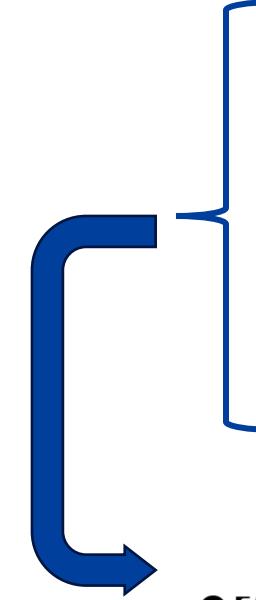
$$T(0) + T(n - 1) + n - 1$$

$$T(1) + T(n - 2) + n - 1$$

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

...

$$T(n - 1) + T(0) + n - 1$$


$$2[T(1) + \dots + T(n - 1)] + n(n - 1)$$



时间复杂度分析 - 差消法

$$2[T(1) + \dots + T(n-1)] + n(n-1)$$

假设首元素排好序在每个位置是等概率的

$$T(n) = \frac{2}{n} \sum_{i=1}^{n-1} T(i) + O(n), \quad n \geq 2$$

$$T(1) = 0$$

直接迭代非常困难和**历史项目数**相关

利用两个方程相减，将右边的项尽可能消去，以达到降阶的目的

$$T(n) = \frac{2}{n} \sum_{i=1}^{n-1} T(i) + cn,$$

$$nT(n) = 2 \sum_{i=1}^{n-1} T(i) + cn^2,$$

$$(n-1)T(n-1) = 2 \sum_{i=1}^{n-2} T(i) + c(n-1)^2$$

算法评估示例-4

时间复杂度分析 - 差消法

$$nT(n) - (n-1)T(n-1) = 2T(n-1) + cn^2 - c(n-1)^2$$

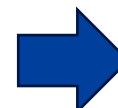
↓ 化简

$$nT(n) = (n+1)T(n-1) + \boxed{c_1 n}$$

↓ 变形

$$\frac{T(n)}{n+1} = \frac{T(n-1)}{n} + \frac{c_1}{n+1}$$

n >= 1



$$\begin{aligned}\frac{T(n)}{n+1} &= \frac{T(n-1)}{n} + \frac{c_1}{n+1} = \dots \\ &= c_1 \left[\frac{1}{n+1} + \frac{1}{n} + \dots + \frac{1}{3} \right] + \frac{T(1)}{2} \\ &= c_1 \left[\frac{1}{n+1} + \frac{1}{n} + \dots + \frac{1}{3} \right] \\ &= \Theta(\log n)\end{aligned}$$



求解递推方程

$$T(n) = aT\left(\frac{n}{b}\right) + f(n)$$

a : 归约后的子问题个数

n/b : 归约后子问题的规模

$f(n)$: 归约过程及组合子问题的解的工作量

二分检索: $T(n) = T\left(\frac{n}{2}\right) + 1$

二分归并排序: $T(n) = 2T\left(\frac{n}{2}\right) + n - 1$

一趟快排: $T(n) = \frac{2}{n} \sum_{i=1}^{n-1} T(i) + O(n), \quad n \geq 2$

$$T(1) = 0$$

定义

定理：设 $a \geq 1, b > 1$ 为常数, $f(n)$ 为函数, $T(n)$ 为非负整数, 且 $T(n) = aT(n/b) + f(n)$, 则

情况1

1. 若 $f(n) = O(n^{\log_b a - \varepsilon})$, $\varepsilon > 0$, 那么 $T(n) = \Theta(n^{\log_b a})$

存在 ε

情况2

2. 若 $f(n) = \Theta(n^{\log_b a})$, 那么 $T(n) = \Theta(n^{\log_b a} \log n)$

情况3

若 $f(n) = \Omega(n^{\log_b a + \varepsilon})$, $\varepsilon > 0$,

且对于某个常数 $c < 1$ 和充分大的 n 有 $af\left(\frac{n}{b}\right) \leq c f(n)$,

存在
 c 和 n

那么 $T(n) = \Theta(f(n))$



证明

迭代

$$T(n) = aT\left(\frac{n}{b}\right) + f(n)$$

设 $n = b^k$

$$\begin{aligned} T(n) &= aT\left(\frac{n}{b}\right) + f(n) \\ &= a[aT\left(\frac{n}{b^2}\right) + f\left(\frac{n}{b}\right)] + f(n) \\ &= a^2T\left(\frac{n}{b^2}\right) + af\left(\frac{n}{b}\right) + f(n) \\ &= \dots \end{aligned}$$

$$T(n) = a^k \cdot T\left(\frac{n}{b^k}\right) + a^{k-1} \cdot f\left(\frac{n}{b^{k-1}}\right) + \dots + af\left(\frac{n}{b}\right) + f(n)$$

$$= a^k \cdot T(1) + \sum_{j=0}^{k-1} a^j f\left(\frac{n}{b^j}\right)$$

$$= c_1 n^{\log_b a} + \sum_{j=0}^{k-1} a^j f\left(\frac{n}{b^j}\right) \quad (T(1) = c_1)$$

$$a^{\log_b n} = n^{\log_b a}$$

- 第一项为所有最小子问题的计算工作量
- 第二项为迭代过程归约到子问题及综合解的工作量



情况1

$$f(n) = O(n^{\log_b a - \epsilon})$$

$$\begin{aligned} T(n) &= c_1 n^{\log_b a} + \sum_{j=0}^{k-1} a^j f\left(\frac{n}{b^j}\right) \\ &= c_1 n^{\log_b a} + O\left(\sum_{j=0}^{\log_b n - 1} a^j \left(\frac{n}{b^j}\right)^{\log_b a - \epsilon}\right) \\ &= c_1 n^{\log_b a} + O(n^{\log_b a - \epsilon} \sum_{j=0}^{\log_b n - 1} \frac{a^j}{(b^{\log_b a - \epsilon})^j}) \end{aligned}$$

$$\frac{1}{(b^{\log_b a - \epsilon})^j} = \frac{b^{\epsilon j}}{(b^{\log_b a})^j} = \frac{b^{\epsilon j}}{a^j}$$

$$\begin{aligned} &= c_1 n^{\log_b a} + O(n^{\log_b a - \epsilon} \sum_{j=0}^{\log_b n - 1} (b^\epsilon)^j) \\ &= c_1 n^{\log_b a} + O(n^{\log_b a - \epsilon} \frac{b^{\epsilon \log_b n} - 1}{b^\epsilon - 1}) \\ &= c_1 n^{\log_b a} + O(n^{\log_b a - \epsilon} n^\epsilon) \\ &= \Theta(n^{\log_b a}) \end{aligned}$$



情况2

$$f(n) = \Theta(n^{\log_b a})$$

$$T(n) = c_1 n^{\log_b a} + \sum_{j=0}^{k-1} a^j f\left(\frac{n}{b^j}\right)$$

$$= c_1 n^{\log_b a} + \Theta\left(\sum_{j=0}^{\log_b n - 1} a^j \left(\frac{n}{b^j}\right)^{\log_b a}\right)$$

$$= c_1 n^{\log_b a} + \Theta\left(n^{\log_b a} \sum_{j=0}^{\log_b n - 1} \frac{a^j}{b^j}\right)$$

$$= c_1 n^{\log_b a} + \Theta(n^{\log_b a} \log n) = \Theta(n^{\log_b a} \log n)$$



情况3

$$f(n) = \Omega(n^{\log_b a + \varepsilon}) \quad (1)$$

$$af(n/b) \leq c f(n) \quad (2)$$

$$T(n) = c_1 n^{\log_b a} + \sum_{j=0}^{k-1} \boxed{a^j f\left(\frac{n}{b^j}\right)} \leq c \cdot a^{j-2} \cdot c \cdot f\left(\frac{n}{b^{j-2}}\right)$$

$$\leq c^2 \cdot a^{j-3} \cdot a \cdot f\left(\frac{n}{b^{j-3}}\right)$$

$$\leq c^2 \cdot a^{j-3} \cdot c \cdot f\left(\frac{n}{b^{j-3}}\right)$$

利用条件2

$$a^j f\left(\frac{n}{b^j}\right) \leq a^{j-1} c f\left(\frac{n}{b^{j-1}}\right)$$

$$\leq c a^{j-1} f\left(\frac{n}{b^{j-1}}\right) \leq \dots \leq c^j f(n)$$

两边同时乘 a^{j-1} , n 换为 $\frac{n}{b^{j-1}}$

$$a^j \cdot f\left(\frac{n}{b^j}\right) \leq a^{j-1} \cdot c \cdot f\left(\frac{n}{b^{j-1}}\right)$$

$$\leq c \cdot a^{j-1} \cdot f\left(\frac{n}{b^{j-1}}\right)$$

$$= c \cdot a^{j-2} \cdot a \cdot f\left(\frac{n}{b^{j-1}}\right)$$

⋮

$$\leq c^j \cdot f(n)$$



情况3

$$f(n) = \Omega(n^{\log_b a + \varepsilon}) \quad (1)$$

$$af(n/b) \leq c f(n) \quad (2)$$

利用条件2

$$a^j f\left(\frac{n}{b^j}\right) \leq a^{j-1} c f\left(\frac{n}{b^{j-1}}\right)$$

$$\leq c a^{j-1} f\left(\frac{n}{b^{j-1}}\right) \leq \dots \leq c^j f(n)$$

$$\begin{aligned} T(n) &= c_1 n^{\log_b a} + \sum_{j=0}^{k-1} a^j f\left(\frac{n}{b^j}\right) & T(n) &\leq c_1 n^{\log_b a} + \sum_{j=0}^{\log_b n - 1} c^j f(n) \\ &\leq c_1 n^{\log_b a} + \sum_{j=0}^{\log_b n - 1} c^j f(n) & &= c_1 n^{\log_b a} + f(n) \frac{(c^{\log_b n} - 1)}{c - 1} \\ &&&= c_1 n^{\log_b a} + \Theta(f(n)) \end{aligned}$$

c 小于1等比数列

主定理的简化版包括三种情况：

1. 情况1：当 $f(n) = O(n^{\log_b a})$ 且 $\varepsilon > 0$ 时， $T(n) = \Theta(n^{\log_b a})$ 。
2. 情况2：当 $f(n) = \Theta(n^{\log_b a} \cdot \log^k n)$ 且 $k \geq 0$ 时， $T(n) = \Theta(n^{\log_b a} \cdot \log^{k+1} n)$ 。
3. 情况3：当 $f(n) = \Omega(n^{\log_b(a+\varepsilon)})$ 且 $a \cdot f\left(\frac{n}{b}\right) \leq c \cdot f(n)$ 对于某个常数 $c < 1$ 和所有足够大的 n 成立时， $T(n) = \Theta(f(n))$ 。

主定理的一般形式

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

$$f(n) = \Omega(n^{\log_b a + \varepsilon}) \quad \varepsilon > 0$$



总结

主定理的简化版包括三种情况：

1. 情况1：当 $f(n) = O(n^{\log_b a})$ 且 $\epsilon > 0$ 时， $T(n) = \Theta(n^{\log_b a})$ 。
2. 情况2：当 $f(n) = \Theta(n^{\log_b a} \cdot \log^k n)$ 且 $k \geq 0$ 时， $T(n) = \Theta(n^{\log_b a} \cdot \log^{k+1} n)$ 。
3. 情况3：当 $f(n) = \Omega(n^{\log_b(a+\epsilon)})$ 且 $a \cdot f(\frac{n}{b}) \leq c \cdot f(n)$ 对于某个常数 $c < 1$ 和所有足够大的 n 成立时， $T(n) = \Theta(f(n))$ 。

主定理的一般形式



示例1

例1 求解递推方程

$$T(n) = 9T(n/3) + n$$

解 上述递推方程中的

$$a = 9, b = 3, f(n) = n$$

$$n^{\log_3 9} = n^2, f(n) = O(n^{\log_3 9-1})$$

相当于主定理的Case1，其中 $\varepsilon = 1$

根据定理得到 $T(n) = \Theta(n^2)$

若 $f(n) = O(n^{\log_b a-\varepsilon})$, $\varepsilon > 0$ 那么

$$T(n) = \Theta(n^{\log_b a})$$



示例2

例2 求解递推方程

$$T(n) = T(2n/3) + 1$$

解 上述递推方程中的

$$a = 1, b = 3/2, f(n) = 1,$$

$$n^{\log_{3/2} 1} = n^0 = 1,$$

若 $f(n) = \Omega(n^{\log_b a})$, 那么

$$T(n) = \Theta(n^{\log_b a} \log n)$$

相当于主定理的 Case2,

根据定理得到 $T(n) = \Theta(\log n)$



算法评估-主定理的应用

示例3

例2 求解递推方程

$$T(n) = 3T(n/4) + n \log n$$

解 上述递推方程中的

$$a = 3, b = 4, f(n) = n \log n$$

$$n \log n = \Omega(n^{\log_4 3 + \varepsilon}) \approx \Omega(n^{0.793 + \varepsilon})$$

取 $\varepsilon = 0.2$ 即可。

要使 $af\left(\frac{n}{b}\right) \leq c f(n)$ 成立,

代入 $3\left(\frac{n}{4}\right) \log\left(\frac{n}{4}\right) \leq cn \log n$, 得到

$$3\left(\frac{n}{4}\right) \log\left(\frac{n}{4}\right) \leq cn \log n$$

只要 $c \geq \frac{3}{4}$, 上述不等式可以对所有充分大的 n 成立
相当于主定理的 Case3,

因此有 $T(n) = \Theta(f(n)) = \Theta(n \log n)$



示例3

递归算法分析

二分检索：

$$W(n) = W\left(\frac{n}{2}\right) + 1, \quad W(1) = 1$$

$$a = 1, b = 2, n^{\log_2 1} = 1, f(n) = 1,$$

属于 Case2,

$$W(n) = \Theta(\log n)$$

二分归并排序：

$$W(n) = 2W\left(\frac{n}{2}\right) + n - 1, \quad W(1) = 0$$

$$a = 2, b = 2, n^{\log_2 2} = n, f(n) = n - 1$$

属于 Case2,

$$W(n) = \Theta(n \log n)$$



程序运行时间测量

- **Wall time (挂钟时间)**: 测量从开始到结束所经过的时间。它包括所有时间间隔，例如等待I/O操作、CPU花费时间等
- **CPU time (CPU时间)** :CPU忙于处理程序指令的时间。等待其他事情完成（例如I / O操作）所花费的时间不包括在CPU时间中

常用的计算挂钟时间的函数：

`<sys/time.h>` 中的 `gettimeofday()`: 微妙精度

`<chrono>` 中的 `high_resolution_clock`: 纳秒精度





程序运行时间测量

```
struct timeval
{
    __time_t tv_sec;          /* Seconds. */
    __suseconds_t tv_usec;    /* Microseconds. */
};

struct timeval start_time, end_time;
// Record the start time
gettimeofday(&start_time, NULL);
// Call the function you want to measure
myFunction();
// Record the end time
gettimeofday(&end_time, NULL);
// Calculate the elapsed time in microseconds
long elapsed_time = (end_time.tv_sec - start_time.tv_sec) * 1e6
                    + (end_time.tv_usec - start_time.tv_usec);
// Output the elapsed time
printf("Time taken by function: %ld microseconds\n", elapsed_time);
return 0;
```



时间复杂度 $T(n)=O(n^3)$

```
for (int i=0;i<N;++i)
    for (int j=0;j<N;++j){
        C[i][j]=0;
        for (int k=0;k<N;++k)
            C[i][j] += A[i][k]*B[k][j];
    }
```

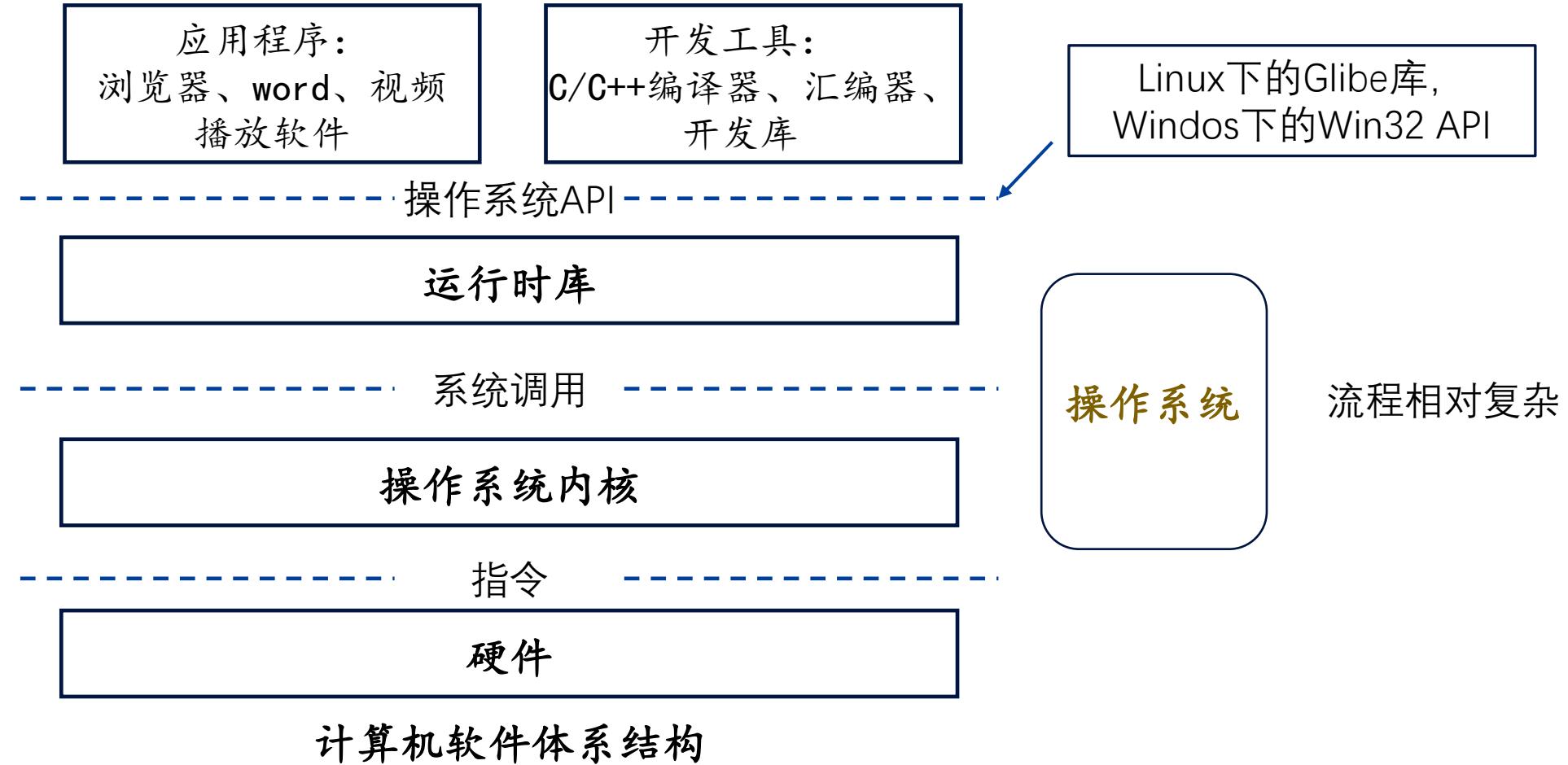
实际运行时间为T1

```
for (int i=0;i<N;++i)
    for (int j=0;j<N;++j){
        for (int k=0;k<N;++k)
            printf("hello\n");
```

实际运行时间为T2

当序列长度一定时， $T1 << T2$

程序在硬件上运行





Printf函数干了什么

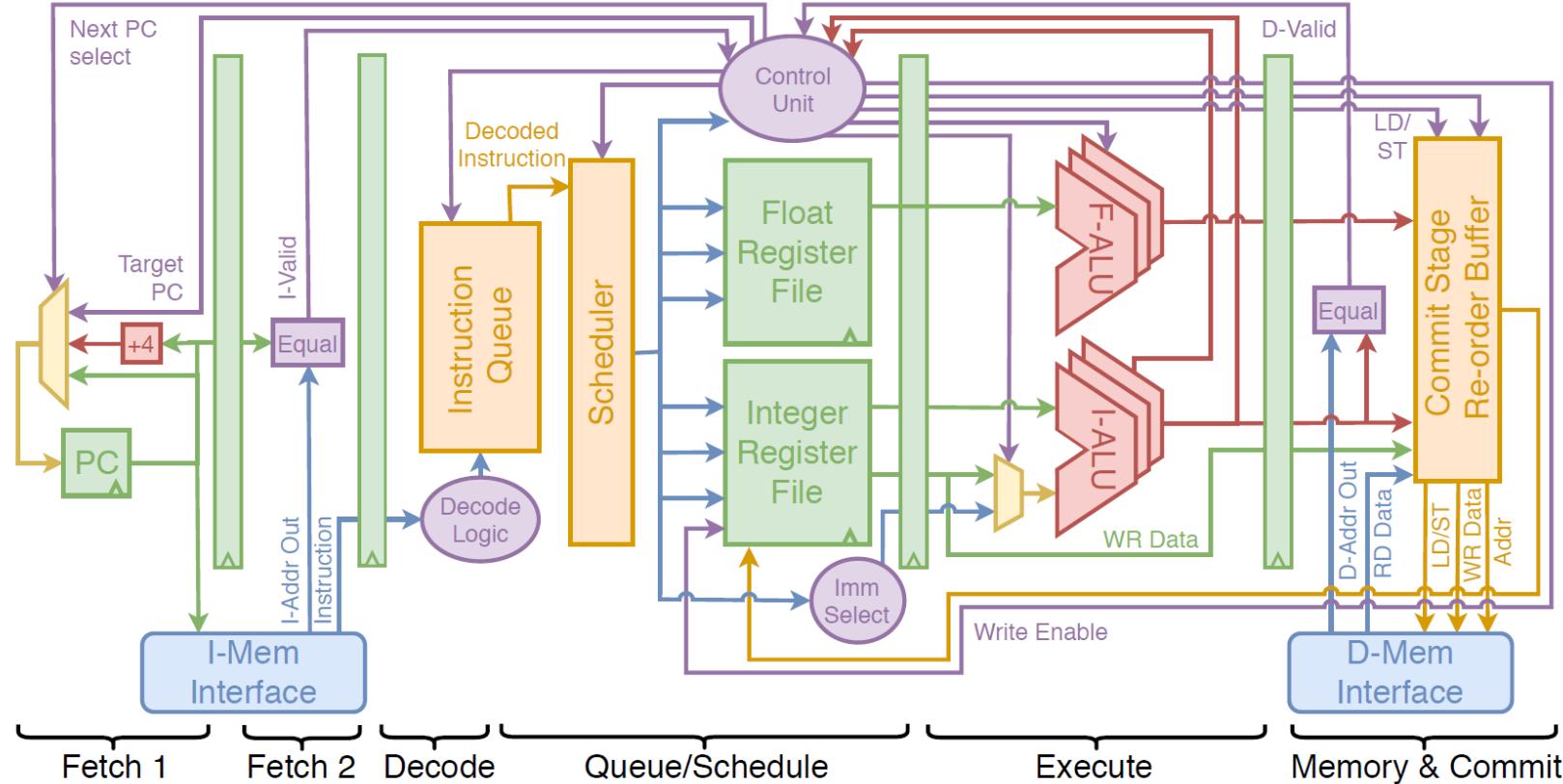
解析格式字符串，确定要输出的变量的类型、数量和格式。

从变长参数列表中获取相应的变量的值，并进行类型转换和格式化。

将格式化后的字符串存放到一个缓冲区中，等待输出。

调用底层的系统函数，将缓冲区中的字符串输出到标准输出流。

直接从最底层观察



支持乱序发射的五级流水线CPU结构示意图

计算量、访存量

并行角度：流水化

计算资源（结构冲突）

计算顺序（数据冲突）



时间复杂度 $T(n)=O(n^3)$

```
for (int i=0;i<N;++i)
for (int j=0;j<N;++j){
    C[i][j]=0;
    for (int k=0;k<N;++k)
        C[i][j] +=A[i][k]*B[k][j];
}
```

实际运行时间为T1

```
for (int i = 0; i < N; i += Tn) {
    for (int j = 0; j < N; j += Tn) {
        for (int ii = i; ii < i + Tn; ++ii) {
            for (int jj = j; jj < j + Tn; ++jj) {
                C[ii][jj] = 0;
                for (int k = 0; k < N; k += Tn) {
                    for (int kk = k; kk < k + Tn; ++kk)
                        C[ii][jj] += A[ii][kk] * B[kk][jj];
                }
            }
        }
    }
}
```

实际运行时间为T3

当序列长度足够大时， $T1 > T3$



- 存储空间：内存开销+额外空间开销

- 有时候，特定问题的应用背景对可用的内存大小进行了约束
- 额外空间开销：

- 比如，排序时，可能需额外的临时空间来存储一个或更多待排序的记录；
- 隐式的额外空间需求：递归方式实现的算法对栈空间的需求

- 示例：

问题：800电话号码有如下的格式：800-8222657，其中有效的800免费电话不超过800万个，比如不存在以0或1开头的有效免费电话。现要求对这些800免费电话号码进行排序，要求内存不超过1MB。（巧妙选择数据存储方式）

提示：可以使用位图法(BitMap)存放数据。

如何度量？（理论估算，实际测量）

电话号码排序



● 位图表示

举例来说，我们可以使用一个16位的字符串来表示一个小型的小于16的非负整数。

如{1,2,3,5,8,13}表示成如下字符串：

index:	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
value:	0	0	1	0	0	0	0	1	0	0	1	0	1	1	1	0

第1位表示数字0，在集合中没有0所以这一位为0

第2位表示数字1，在集合中有1所以这一位为1

第3位表示数字2，在集合中有2所以这一位为1

第4位表示数字3，在集合中有3所以这一位为1

第5位表示数字4，在集合中有3所以这一位为0

本题中，我们使用800万位的字符表示这整个文件中的整数。

表示[0,8000000)之间的所有的数。当文件中有这个数时，用1表示，否则为默认值0。这样实际

使用的内存是 $800\ 0000/8=1000000 < 1\text{MB}$ 。

电话号码排序



● 位图表示

1. 第一阶段关闭所有位，将集合初始化为空。
2. 第二阶段读取文件中的每个整数，并打开**相应位**。
3. 第三阶段**检查每个位**，如果某个位为1，就写出相应的整数，写入文件中。

index:	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
value:	0	0	1	0	0	0	0	0	0	0	0	0	0	0	0	0



第1个8位(char) 第0个8位(char)

i=13时：如果用char字符，则在第1个字符（0开始）的第5位置1

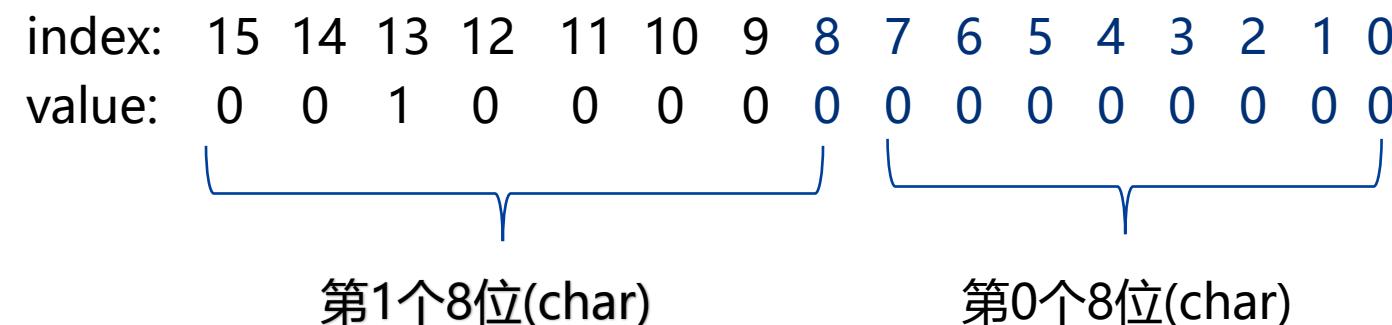
电话号码排序



● 位图表示

1. 第一阶段关闭所有位，将集合初始化为空。
2. 第二阶段读取文件中的每个整数，并打开**相应位**。
3. 第三阶段**检查每个位**，如果某个位为1，就写出相应的整数，写入文件中。

1bit数据类型？



i=13时：如果用char字符，则在第1个字符（0开始）的第5位置1

```
#define BITSPERWORD 32
#define SHIFT 5
#define MASK 0x1F
#define N 8000000
int a[1 + N/BITSPERWORD];
```

S1

```
// 设置整数 i 对应的位为1
void set(int i) {
    a[i >> SHIFT] |= (1 << (i & MASK));
}
```

S2

```
// 将整数 i 对应的位清零
void clr(int i) {
    a[i >> SHIFT] &= ~(1 << (i & MASK));
}
```

S3

```
// 检查整数 i 对应的位是否为1
int test(int i) {
    return
        a[i >> SHIFT] & (1 << (i & MASK));
}
```

S4

测量：实际的内存空间开销



```
#define MEASURE(T, text) {\
    std::cout << text << "\t" << sizeof(T)<<"字节\t" << "实际消耗的内存空间为\n()";\
    ptrdiff_t lastp2 = 0;\
    T *sentinel;\
    for (int i = 0; i < 10; i++) {\
        sentinel = new T;\
        ptrdiff_t thisp2 = (ptrdiff_t)sentinel;\
        std::cout << " " << thisp2 - lastp2;\
        lastp2 = thisp2;\
    }\
    std::cout << "\n";\
}
```



```
int main()
{
    MEASURE(int,"int");
    MEASURE(double,"double");
    struct structc {char c;};
    struct structc7 {char c[7];};
    struct structc9 {char c[9];};
    struct structc12 {char c[12];};
    struct structc15 {char c[15];};
    struct structc20 {char c[20];};
    struct structc21 {char c[21];};
    struct structic {int i;char c;};
    struct structip {int i;structip *p;};
    struct structdc {double d;char c;};
    MEASURE(structc,"structc");
    MEASURE(structc9,"structc9");
    MEASURE(structc7,"structc7");
    MEASURE(structc12,"structc12");
    MEASURE(structc15,"structc15");
    ...
}
```

申请1~8B：实际分配32B；

申请9~16B：实际分配40B；

申请17~24B：实际分配48B；

申请分配内存空间1次，额外开销为：24~31B



e:\SouceCodeForTeaching\ProgramPeals\Release...							
int	4字节	实际消耗的内存空间为: (字节)	3757680	32	32	32	32
32	32						
double	8字节	实际消耗的内存空间为: (字节)	3758288	32	32	32	32
32	32						
structc	1字节	实际消耗的内存空间为: (字节)	3758640	32	32	32	32
32	32						
structc5	5字节	实际消耗的内存空间为: (字节)	3758992	32	32	32	32
2	32	32	32	32			
structc9	9字节	实际消耗的内存空间为: (字节)	3759344	40	40	40	40
0	40	40	40	40			
structc7	7字节	实际消耗的内存空间为: (字节)	3759784	32	32	32	32
2	32	32	32	32			
structc12	12字节	实际消耗的内存空间为: (字节)	3760136	40	40	40	40
0	40	40	40	40			
structc17	17字节	实际消耗的内存空间为: (字节)	3760576	48	48	48	48
8	48	48	48	48			
structc14	14字节	实际消耗的内存空间为: (字节)	3761104	40	40	40	40
0	40	40	40	40			
structc15	15字节	实际消耗的内存空间为: (字节)	3761544	40	40	40	40
0	40	40	40	40			
structc20	20字节	实际消耗的内存空间为: (字节)	3761984	48	48	48	48
8	48	48	48	48			
structc21	21字节	实际消耗的内存空间为: (字节)	3762512	48	48	48	48
8	48	48	48	48			
structic	8字节	实际消耗的内存空间为: (字节)	3763040	32	32	32	32
2	32	32	32	32			
structip	8字节	实际消耗的内存空间为: (字节)	3763392	32	32	32	32
2	32	32	32	32			
structdc	16字节	实际消耗的内存空间为: (字节)	3763744	40	40	40	40
0	40	40	40	40			



结论：应尽可能减少申请分配内存空间的次数

- ✓ 好处：
 - 减少额外开销
 - 可提高运行速度，原因有二：
 - 1) 减少额外开销后，Cache中的元素数增加了，可以提高处理速度（Malloc之后的Cache情况）
 - 2) 内存的分配和销毁，属于耗时操作。
- ✓ 如何减少申请分配内存空间的次数？
 - 一次性手动申请大片的内存空间作为内存池（从这些预分配的块中分配内存，以避免频繁的系统调用或内存碎片）
 - 按实际需要从内存池中取出相应大小的空间来使用，若内存池中空闲空间不够，则再申请新的大片内存空间。
 - 内存空间用完后不直接回收，而是返回给内存池；内存池不用时，需手动释放。

内存池的简单应用



初始化数据类型

```
typedef struct {
    char data[BLOCK_SIZE];
} Block;

typedef struct {
    Block* blocks[POOL_SIZE];
    int nextFreeIndex;
} MemoryPool;
```

```
MemoryPool* createMemoryPool() {
    MemoryPool* pool =
(MemoryPool*)malloc(sizeof(MemoryPool));
    if (pool == NULL) {
        perror("Failed to create memory pool");
        exit(EXIT_FAILURE);
    }

    // 初始化内存池中的块
    for (int i = 0; i < POOL_SIZE; ++i) {
        pool->blocks[i] =
(Block*)malloc(sizeof(Block));
        if (pool->blocks[i] == NULL) {
            perror("Failed to allocate memory block");
            exit(EXIT_FAILURE);
        }
    }
    pool->nextFreeIndex = 0;
    return pool;
}
```

内存池的简单应用



```
void* allocateFromPool(MemoryPool* pool) {  
    if (pool->nextFreeIndex < POOL_SIZE) {  
        return (void*)pool->blocks[pool-  
        >nextFreeIndex++];  
    } else {  
        fprintf(stderr, "Memory pool is full\n");  
        return NULL;  
    }  
}  
  
void deallocateToPool(MemoryPool* pool, void*  
block) {  
    if (pool->nextFreeIndex > 0) {  
        pool->blocks[--pool->nextFreeIndex] =  
        (Block*)block;  
    } else {  
        fprintf(stderr, "Memory pool is empty\n");  
    }  
}
```

```
void destroyMemoryPool(MemoryPool* pool)  
{  
    for (int i = 0; i < POOL_SIZE; ++i)  
    {  
        free(pool->blocks[i]);  
    }  
    free(pool);  
}
```

多次析构

```
int** array = new int*[m];  
for(i) array[i] = new int[n];  
  
for(i) delete []array[i];  
delete []array;
```

1.4 设计算法的步骤



- Step1：问题定义；
- Step2：系统结构的设计。 (指将大型系统进行模块分解，是属于程序设计思想的范畴)。（本课程假定所要解决的问题不涉及复杂的大系统，因而此步骤不予考虑）
- Step3：算法设计及数据结构的选择。依据问题定义、输入数据的特征和要求输出的数据的特征，分析广泛的解决方案（数据结构+算法），并选择最佳的解决方案；（本课程的重点）
 - 解决方案：数据结构+算法
 - 最佳的解决方案的确定：依赖于良好的数据结构和算法的选择。

1.4 设计算法的步骤



• Step4：代码调优。实现并优化代码。

- 效率 vs. 清晰的程序结构/可维护性
- 原则：
 - 1) 尽量减少输入输出；减少函数调用的次数；限制 计算密集型操作（浮点运算，除法运算）；
 - 2) 然后，确定最耗时的操作，并提高其性能；（如，冒泡排序中，应关注比较和交换）；
 - 可以用测量和跟踪工具（如，Profiler, AQTime）；
 - 也可以用手动测试进行性能监视（打印所花费的时间与输入规模之间的关系，后面在“二分查找”部分会介绍）

1.4 设计算法的步骤



- 问题分析：
 - 问题1：给定一个英语词典，找出其中的所有变位词集合。例如，“pots”、“stop” 和“tops” 互为变位词。
 - 问题规模？
 - 变位词具有什么性质？

1.4 设计算法的步骤



- 问题分析：

- 问题2：给定一个最多包含40亿个随机排列的32 bits非负整数的顺序文件，找出一个不在文件中的32 bits整数（在文件中必然缺失一个这样的数——为什么？）。
 - 问题规模？40亿（ $4e9$ ）个INT型
 - 找到一个不在或者多个不在文件中的值？
 - 在具有足够内存的情况下，如何解决该问题？
 - 如果有几个外部的“临时”文件可用，但是仅有几百字节的内存，又该如何解决？

1.5 最佳算法选择的决定因素



1) 问题的约束:

- 比如, 可用内存空间, 运行时间的上限约束等。

2) 数据的存储方式

- 存储什么信息? (比如, 位图法)
- 选用何种数据结构? (取决于在数据上的常用操作类型 (静态? 动态?) 、以及内存空间的限制)
- 比如, 顺序表上进行插入/删除比较慢, 所以, 在顺序表上实现交换2个元素的算法要尽量回避这些速度慢的操作。

3) 输入数据的特征: 是否要求有序?

4) 输出数据的特征: 是否要求有序? 是否包含重复记录。



中国科学技术大学
University of Science and Technology of China

谢谢！



中国科学技术大学