数据结构与算法

(一)

2 算法复杂度

杜育根

ygdu@sei.ecnu.edu.cn

主要内容

- 计算的资源
- 算法的定义
- 算法的评估

计算的资源

- **程序运行时需要的资源有两种**

  时间：程序运行需要的时间。

  空间：程序运行需要的存储空间。

✓资源是有限的

Time Limit和Memory Limit

- OJ上的题目中，常常有对运行时间和空间的说明，例如：
- Time Limit: 2000/1000 MS (Java/Others)
- Memory Limit: 65536/65536 K (Java/Others)

- 这2个限制条件非常重要，是检验程序性能的参数。
- 注意：在现场赛中，为了增加迷惑性，可能不会告诉你，需要自己判断。

- 程序必须在限定的时间和空间内运行结束。
- 问题的“有效”解决，不仅在于能否得到正确答案，更重要的是能在合理的时间和空间内给出答案。

算法的定义

- 算法（Algorithm）：对特定问题求解步骤的一种描述，是指令的有限序列。有5个特征：

(1)有穷性：一个算法必须在执行有穷步之后结束，且每一步都在有穷时间内完成。

(2)确定性：算法中的每一条指令必须有确切的含义，对于相同的输入只能得到相同的输出。

(3)输入：一个算法有零个或多个输入。以刻画运算对象的初始情况，所谓0个输入是指算法本身定出了初始条件；

(4)输出：一个算法有一个或多个输出，以反映对输入数据加工后的结果。没有输出的算法是毫无意义的。

(5)可行性：算法描述的操作可以通过已经实现的基本操作执行有限次来实现。

李开复的故事

- “1988年，贝尔实验室副总裁亲自来访问我的学校，目的就是为了想了解为什么他们的语音识别系统比我开发的慢几十倍，而且，在扩大至大词汇系统后，速度差异更有几百倍之多……

- 在与他们探讨的过程中，我惊讶地发现一个O(n*m)的动态规划居然被他们做成了O(n*n*m)……贝尔实验室的研究员当然绝顶聪明，但他们全都是学数学、物理或电机出身，从未学过计算机科学或算法，才犯了这么基本的错误。”

算法的复杂性

*算法的复杂性*是算法效率的度量，是评价算法优劣的重要依据。一个算法的复杂性的高低体现在运行该算法所需要的计算机资源的多少上面，所需的资源越多，我们就说该算法的复杂性越高；反之，所需的资源越低，则该算法的复杂性越低。

不言而喻，对于任意给定的问题，设计出复杂性尽可能低的算法是我们在设计算法时追求的一个重要目标；另一方面，当给定的问题已有多种算法时，选择其中复杂性最低者，是我们在选用算法适应遵循的一个重要准则。因此，算法的复杂性分析对算法的设计或选用有着重要的指导意义和实用价值。

时间复杂性和空间复杂性

- 计算机的资源，最重要的是时间和空间（即存储器）资源。因而，算法的复杂性有*时间复杂性*和*空间复杂性*之分。
- 一个好的算法可以在时间和空间上达到最优，但更多情况下，两者是有冲突的，这时候需要选择时间或空间优先。用时间换空间的常用方法是重复计算，用空间换时间的常用方法是预处理。
- 关于算法的复杂性，有两个问题要弄清楚：用怎样的一个量来表达一个算法的复杂性；对于给定的一个算法，怎样具体计算它的复杂性。

考虑问题1

- 已知不重复且已经按从小到大排好的m个整数的数组A[0..m-1]。对于给定的整数c，要求寻找一个下标i，使得A[i]=c；若找不到，则返回一个0。

问题1的一个简单的算法是：

从头到尾扫描数组A。照此，或者扫到A的第i个分量，经检测满足A[i]=c；或者扫到A的最后一个分量，经检测仍不满足A[i]=c。

```
int Search (int c){
    int i=0;
    while (a[i]!=c) &&(i<m))    i++;
    if (a[i]==c) then return i;
            else return -1;
    }
```

二分查找算法

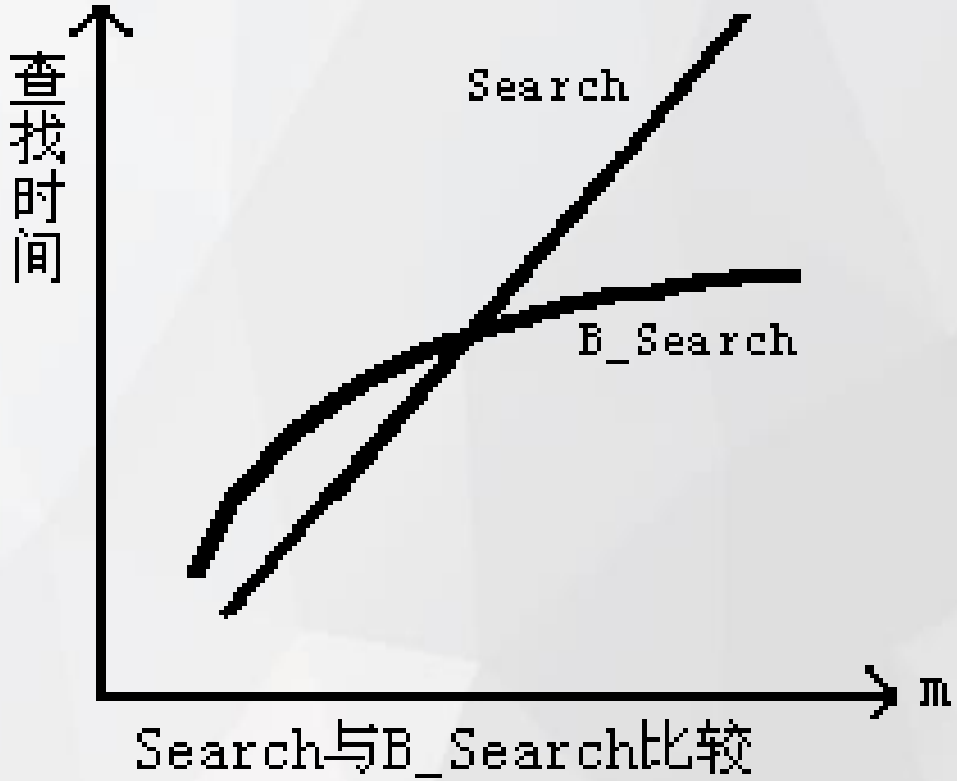
二分查找算法利用到已知条件中A已排好序的性质。它首先拿A的中间分量A[mid]与c比较，如果A[mid]=c则解已找到。如果A[mid]>c，则c只可能在A[0],A[2],..,A[mid-1]之中，因而下一步只要在A[0]，A[2]，..，A[mid-1]中继续查找；如果A[mid]<c，则c只可能在A[mid+1],A[mid+2]，...，A[m-1]之中，因而下一步只要在A[mid+1],A[mid+2],..,A[m-1]中继续查找。不管哪一种情形，都把下一步需要继续查找的范围缩小了一半。再拿这一半的子数组的中间分量与c比较，重复上述步骤。照此重复下去，总有一个时候，或者找到一个i使得A[i]=c，或者子数组为空（即子数组下界大于上界）。前一种情况找到了等于c的分量，后一种情况则找不到。

二分查找算法描述

```
int B_Search ( int c){
    int L,U,mid;   //U和L分别是要查找的数组的下标的上界和下界}
    bool Found;
    L=0; U=m-1; //初始化数组下标的上下界
    Found=false; //当前要查找的范围是A[L]..A[U]。
    while  ((!Found) && (U>=L))  { //当等于c的分量还没有找到且U>=L时，继续查找
            mid=(U+L) /2; //找数组的中间分量
            if (c==a[mid]) then Found=Ture;
                                else if c>a[mid] then L=mid+1;
                                                     else U=mid-1;
            }
    if (Found) then return mid;
                else return -1;
}
```

复杂度比较

算法Search和B_Search解决的是同一个问题，但在最坏的情况下（所给定的c不在A中），两个算法所需要检测的分量个数却大不相同，前者要m个，后者只要$\log_2^m+1$个。可见算法B_Search比算法Search高效得多。



复杂性的计量

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作时间复杂性，需要的空间（即存储器）资源的量称作空间复杂性。这个量应该集中反映算法中所采用的方法的效率，而从运行该算法的实际计算机中抽象出来。换句话说，这个量应该是只依赖于算法要解的问题的规模、算法的输入和算法本身的函数。*如果分别用*N、I和A*来表示算法要解问题的规模、算法的输入和算法本身，*用*C*表示算法的复杂性，那么应该有：$C = F(N, I, A)$

其中$F(N, I, A)$是N, I, A的一个确定的三元函数。如果把时间复杂性和空间复杂性分开，并分别用*T*和*S*来表示，那么应该有：

$$T = T(N, I, A) \qquad (2.1)$$

$$S = S(N, I, A) \qquad (2.2)$$

通常，我们让A隐含在复杂性函数名当中，因而将（2.1）和（2.2）分别简写为$T = T(N, I)$和$S = S(N, I)$

以T(N,I)为例，将复杂性函数具体化

- 根据$T(N,I)$的概念，它应该是算法在一台抽象的计算机上运行所需的时间。设此抽象的计算机所提供的元运算有k种，他们分别记为$O_1, O_2, .., O_k$；再设这些元运算每执行一次所需要的时间分别为$t_1, t_2, .., t_k$。对于给定的算法，设经过统计，用到元运算O_i的次数为e_i，$i = 1, 2, .., k$，很明显，对于每一个i，$1 \le i \le k$，e_i是N和I的函数，即$e_i = e_i(N, I)$。那么有：

$$T(N,I) = \sum_{i=1}^{k} t_i \cdot e_i(N,I) \qquad (2.3)$$

其中t_i，$i = 1, 2, .., k$,是与N, I无关的常数。

考虑三种情况的复杂性

即最坏情况、最好情况和平均情况下的时间复杂性，并分别记为$Tmax(N)$、$Tmin(N)$和$Tavg(N)$。其中，D_N是规模为N的合法输入的集合；I^*是D_N中一个使$T(N,I^*)$达到$Tmax(N)$的合法输入，$\tilde{I}$是D_N中一个使$T(N,\tilde{I})$到$Tmin(N)$的合法输入；而$P(I)$是在算法的应用中出现输入I的概率。在数学上有：

$$T_{\max}(N)=\max_{I\in D_N}T(N,I)=\max_{I\in D_N}\sum_{i=1}^{k}t_i\cdot e_i(N,I)=\sum_{i=1}^{k}t_i\cdot e_i(N,I^*)=T(N,I^*) \quad (2.4)$$

$$T_{\min}(N)=\min_{I\in D_N}T(N,I)=\min_{I\in D_N}\sum_{i=1}^{k}t_i\cdot e_i(N,I)=\sum_{i=1}^{k}t_i\cdot e_i(N,\tilde{I})=T(N,\tilde{I}) \quad (2.5)$$

$$T_{avg}(N)=\max_{I\in D_N}P(I)\cdot T(N,I)=\sum_{I\in D_N}P(I)\sum_{i=1}^{k}t_i\cdot e_i(N,I) \quad (2.6)$$

复杂性的渐近性态及其阶

设T(N)是在第二段中所定义的关于算法A的复杂性函数。一般说来，当N单调增加且趋于∞时，T(N)也将单调增加趋于∞。对于$T(N)$，如果存在$T'(N)$，使得当N→∞时有：

$$(T(N) - T'(N))/T(N) \rightarrow 0$$

那么，我们就说$T'(N)$是$T(N)$当N→∞时的*渐近性态*，或叫$T'(N)$为算法A当N→∞的*渐近复杂性(asymptotic complexity)*而与T(N)相区别，因为在数学上，$T'(N)$是T(N)当N→∞时的渐近表达式。

直观上，$T'(N)$是$T(N)$中略去低阶项所留下的主项。所以它无疑比T(N)来得简单。

举例

$T(N) = 3N^2 + 4Nlog_2N + 7$时，$T'(N)$的一个答案是$3N^2$，因为这时有：

$$(T(N) - T'(N)) / T(N) = \frac{4N\log_2 N + 7}{3N^2 + 4N\log_2 N + 7} \to 0, \qquad 当N \to \infty$$

显然$3N^2$比$3N^2 + 4Nlog_2N + 7$简单得多。

由于当$N \to \infty$时$T(N)$渐近于$T'(N)$，我们有理由用$T'(N)$来替代$T(N)$作为算法在$N \to \infty$时的复杂性的度量。而且由于于$T'(N)$明显地比$T(N)$简单，这种替代明显地是对复杂性分析的一种简化。

O、Ω、θ、o和ω

以下设$f(N)$和$g(N)$是定义在正数集上的正函数。

如果存在正的常数C和自然数N_0，使得当$N \geq N_0$时有$f(N) \leq Cg(N)$。则称函数$f(N)$当N充分大时有上界，且$g(N)$是它的一个*上界*，记为$f(N) = O(g(N))$。这时我们还说$f(N)$的阶不高于$g(N)$的阶。

关于记号Ω，文献里有两种不同的定义。本文只采用其中的一种，定义如下：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$f(N) = \Omega(g(N))$。这时我们还说$f(N)$的阶不低于$g(N)$的阶。

明白了记号O和Ω之后，记号θ将随之清楚，因为我们定义$f(N) = \theta(g(N))$则$f(N) = O(g(N))$且$f(N) = \Omega(g(N))$。这时，我们说$f(N)$与$g(N)$*同阶*。最后，如果对于任意给定的$\varepsilon > 0$，都存在非负整数$N0$，使得当$N \geq N0$时有$f(N) \leq \varepsilon g(N)$，则称函数$f(N)$当$N$充分大时比$g(N)$低阶，记为$f(N) = o(g(N))$，而$f(N) = \omega(g(N))$定义为$g(N) = o(f(N))$。

举例

若 $f(N) = 3N^2 + 4Nlog_2N + 7$

$g(N) = 4Nlog_2N + 7$

则 $f(N) = O(N^3)$

$f(N) = \Omega(g(N))$

$f(N) = \theta(N^2)$

$g(N) = o(f(N))$

$f(N) = \omega(g(N))$

上面这个语音识别的例子

- O(n*m)和O(n*n*m)：就是时间复杂度

- 符号’O’表示复杂度，O(n*m)可以粗略地理解为运行次数是n*m。
- O(n*n*m)比O(n*m)运行时间大n倍

- 假如n=100
- 李开复的算法O(n*m)，设运行时间是1s；
- 贝尔实验室的系统O(n*n*m)，则需要100s。

如何测量程序的运行时间?

用clock函数统计运行时间。

下面程序中的for语句，循环次数是n。

```
#include <bits/stdc++.h>
using namespace std;
int main(){
    int i,k;
    int n=1e8;
    clock_t start, end;
    start = clock();
    for(i = 0; i < n; i++)
       k++;
    end = clock();
    cout << (double)(end - start) /
         CLOCKS_PER_SEC << endl;
}
```

- PC机的算力：
  当n = 1e8 = 10^8时，运行时间0.164s。

  当n = 1e9时，运行时间1.645s。

- 如果题目要求“Time Limit: 2000/1000 MS (Java/Others)”，那么内部的循环次数应该满足n<=10^8，即1亿次以内。

例子：不同算法的效率

hdu 1425 sort

Time Limit: 6000/1000MS (Java/Others) Memory Limit: 64M/32M (Java/Others)

给n个整数，按从大到小的顺序输出其中前m大的数。

输入：每组测试数据有两行，第一行有两个数n，m(0 < n, m < 1,000,000)，第二行包含n个各不相同，且都处于区间[-500,000, 500,000]的整数。

输出：对每组测试数据按从大到小的顺序输出前m大的数。

Sample Input

5 3

3 -35 92 213 -644

Sample Output

213 92 3

算法1. 冒泡排序

```
#include<bits/stdc++.h>
using namespace std;
int a[1000001];                                    //记录数字
#define swap(a, b) {int temp = a; a = b; b = temp;} //交换
int n, m;
void bubble_sort(){                    //冒泡排序，结果仍放在a[]中
    for(int i = 1; i <= n-1; i++)
        for(int j = 1; j <= n-i; j++)
            if(a[j] > a[j+1])
                swap(a[j], a[j+1]);
}
int main(){
    while(~scanf("%d%d", &n, &m)){
        for(int i=1; i<=n; i++)  scanf("%d", &a[i]);
        bubble_sort();
        for(int i = n; i >= n-m+1; i--){    //打印前m大的数
            if(i == n-m+1)  printf("%d\n", a[i]);
            else            printf("%d ", a[i]);
        }
    }
    return 0;
}
```

冒泡排序的复杂度

```
void bubble_sort(){    //冒泡排序，结果仍放在a[]中
    for(int i = 1; i <= n-1; i++)
        for(int j = 1; j <= n-i; j++)
            if(a[j] > a[j+1])
                swap(a[j], a[j+1]);
}
```

- 在bubble_sort()中有2层循环，循环次数是：
  $n-1 + n-2 + \ldots + 1 \approx n^2/2;$
- 在swap(a, b)中做了3次操作；总的计算次数是 $3n^2/2$，复杂度记为$O(n^2)$。
- n = 100万时，计算1万亿次。

算法2. 快速排序

- **STL的sort()函数，是改良版的快速排序。**
  在上面的程序中，把bubble_sort()；改为：

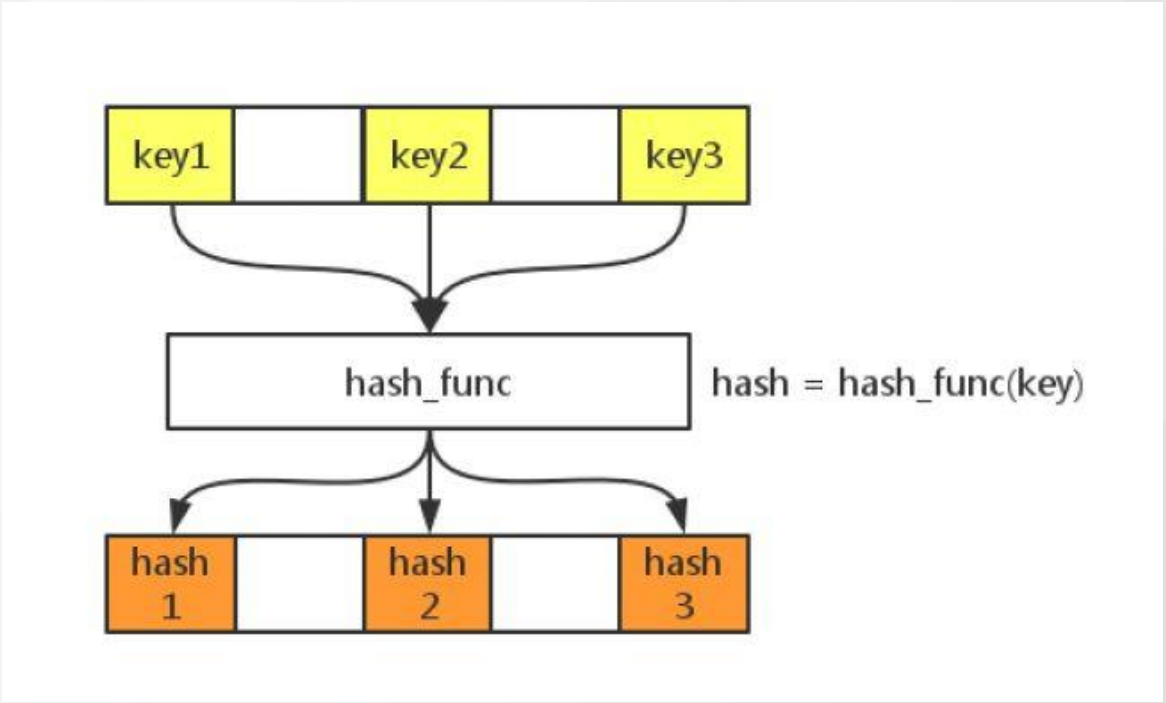
  sort(a + 1, a + n + 1);

  就完成了a[1]到a[n]的排序。

- **算法的时间复杂度是O(nlogn)**
  当n = 100万时，100万 × log2100万 ≈ 2000万。

算法3．哈希

- 哈希算法是一种以空间换取时间的算法。



本题的哈希思路

- 输入一个数字t的时候，对应a[500000 + t]这个位置，记录a[500000 + t] = 1;

- 输出的时候，逐个检查a[i]，如果a[i]等于1，表示这个数存在，打印出前m个。

```
#include<bits/stdc++.h>
using namespace std;
const int MAXN = 1000001;
int a[MAXN];
int main(){
    int n,m;
    while(~scanf("%d%d", &n, &m)){
        memset(a, 0, sizeof(a));
        for(int i=0; i<n; i++){
            int t;
            scanf("%d",&t);//此题数据多，如果用很慢的cin输入，肯定TLE
            a[500000+t]=1; //数字t，登记在500000+t这个位置
        }
        for(int i=MAXN; m>0; i--)
            if(a[i]){
                if(m>1)  printf("%d ", i-500000);
                else      printf("%d\n", i-500000);
                m--;
            }
    }
    return 0;
}
```

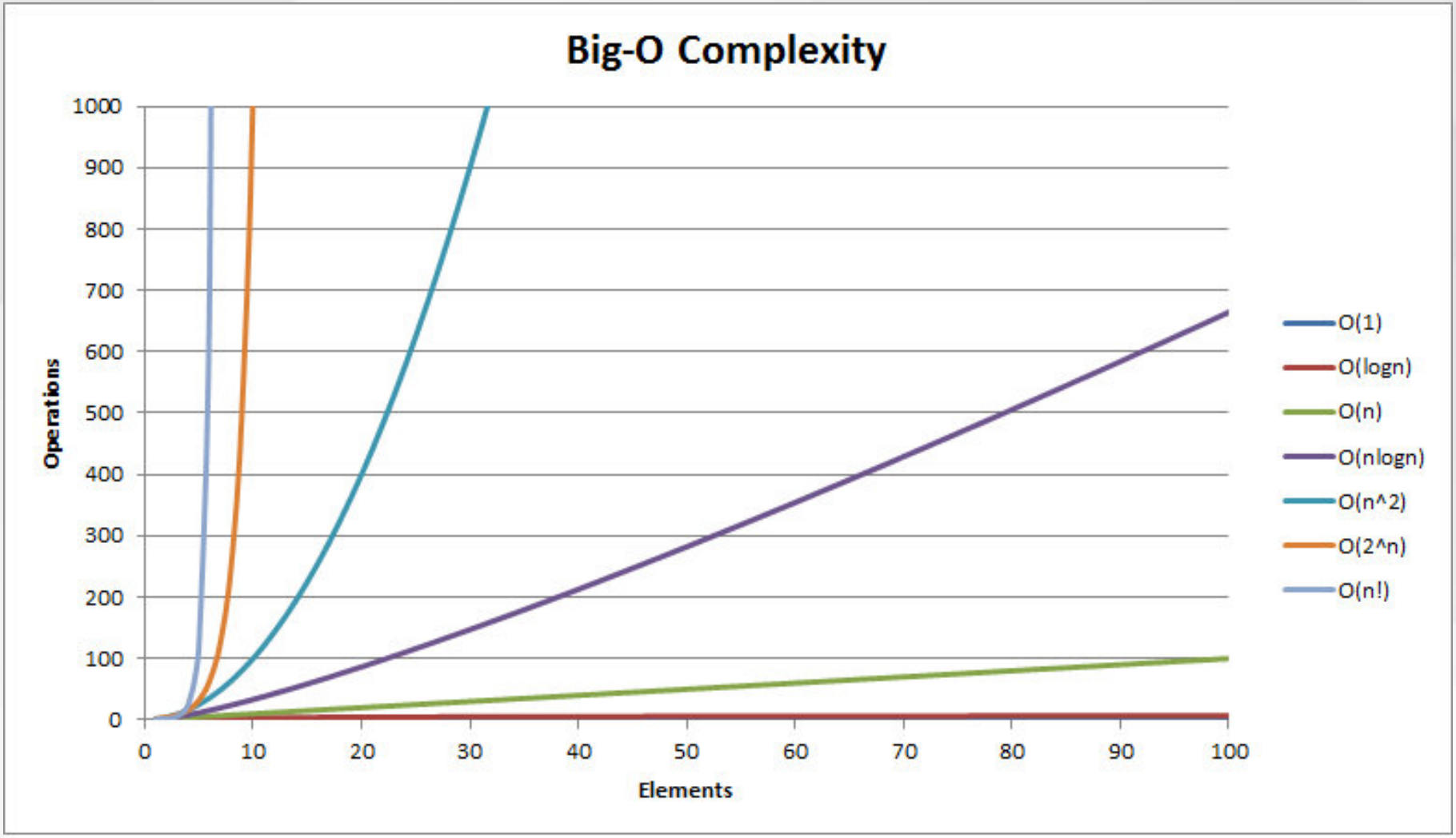
哈希的复杂度

- 程序并没有做显式的排序，只是每次把输入的数按哈希直接插入到对应位置，只有1次操作；
- n个数输入完毕，就相当于排好了序。
- 总的时间复杂度是O(n)。

复杂度的表示

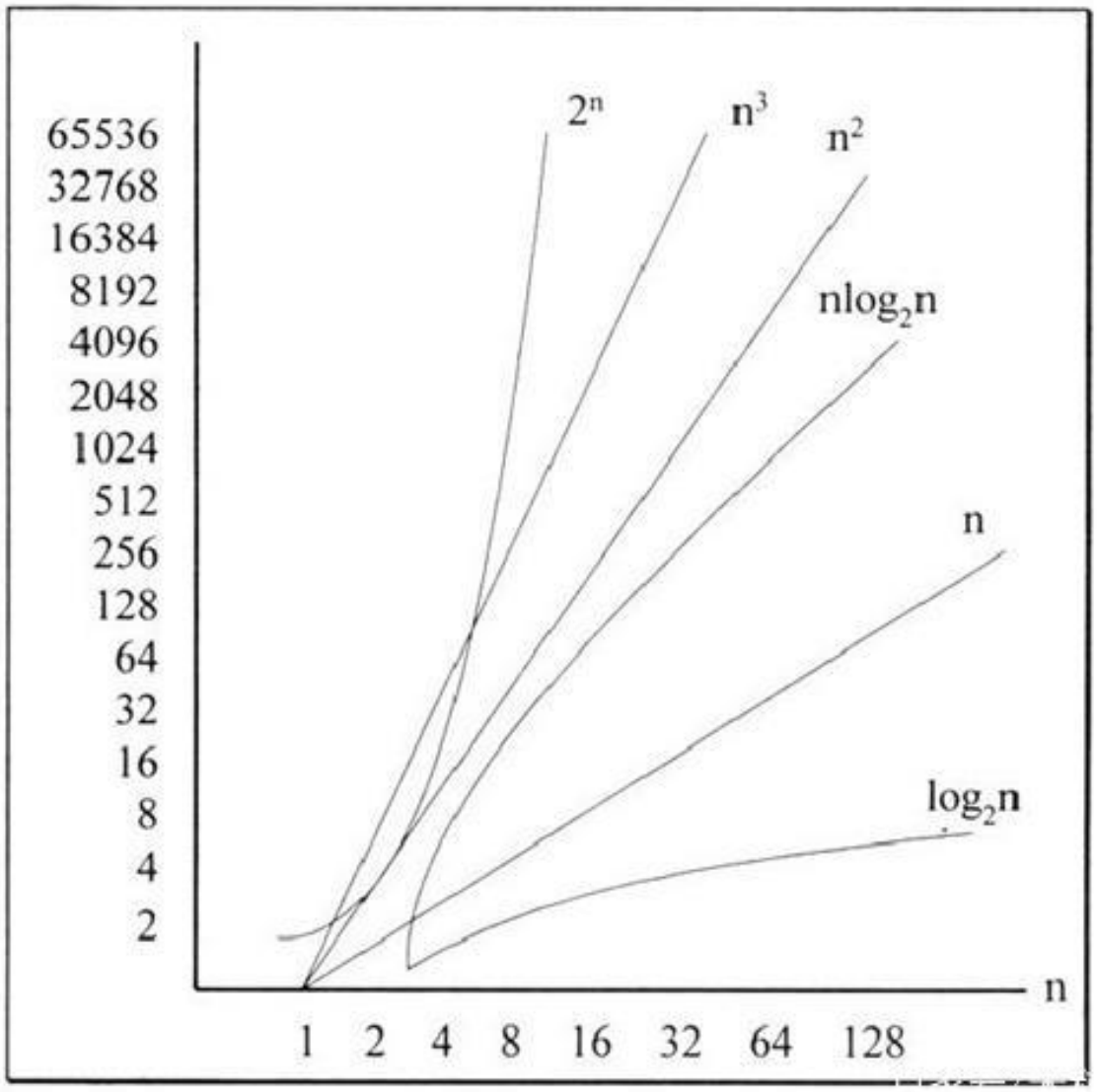
- **上述例子中，把n称为问题的数据规模，把程序的复杂度记为O(n)。**
- **复杂度只是一个估计，不需要精确计算。**
  例如在一个有n个数的无序数列中，查找某个数a，可能第一个数就是a，也可能最后一个数才是a。平均查找时间是n/2次，但是仍然把查找的时间复杂度记为O(n)。
- **在算法分析中，规模n前面的系数被认为是不重要的。**

有哪些复杂度?



图表引用原文https://www.bigocheatsheet.com/

时间复杂度变化曲线图


65536
32768
16384
8192
4096
2048
1024
512
256
128
64
32
16
8
4
2
2^n
n^3
n^2
$n\log_2 n$
n
$\log_2 n$
n
1 2 4 8 16 32 64 128

算法的评估

- **O(1) 计算时间是一个常数，和问题的规模n无关。**
  用公式计算时，一次计算的复杂度就是O(1)，例如hash算法。在矩阵A[M][N]中查找i行j列的元素，只需要一次访问A[i][j] 。
- **O(logn) 计算时间是对数。**
  通常是以2为底的对数，每一步计算后，问题的规模减小一倍。例如在一个长度为n的有序数列中查找某个数，用折半查找的方法，只需要logn次就能找到。
- **O(n) 计算时间随规模n线性增长。**
  在很多情况下，这是算法能达到的最优复杂度，因为对输入的n个数，程序一般需要处理所有的数，即计算n次。例如查找一个无序数列中的某个数，可能需要检查所有的数。

- O(nlogn) 算法可能达到的最优复杂度。
  快速排序算法是典型例子。
- $O(n^2)$ 一个两重循环的算法，复杂度是$O(n^2)$。
  例如冒泡排序，是典型的两重循环。
- $O(n^3)$、$O(n^4)$等等。
- $O(2^n)$ 一般对应集合问题。
  例如一个集合中有n个数，要求输出它的所有子集。
- O(n!) 在集合问题中，如果要求按顺序输出所有的子集，那么复杂度就是O(n!)

分类

把上面的复杂度分成两类：

- **多项式**复杂度，包括O(1)、O(n)、O(nlogn)、$O(n^k)$等，其中k是一个常数；

$$O(1)<O(\log n)<O(n)<O(n\log n)<O(n^2)<O(n^3)$$

- **指数**复杂度，包括$O(2^n)$、O(n!)等。

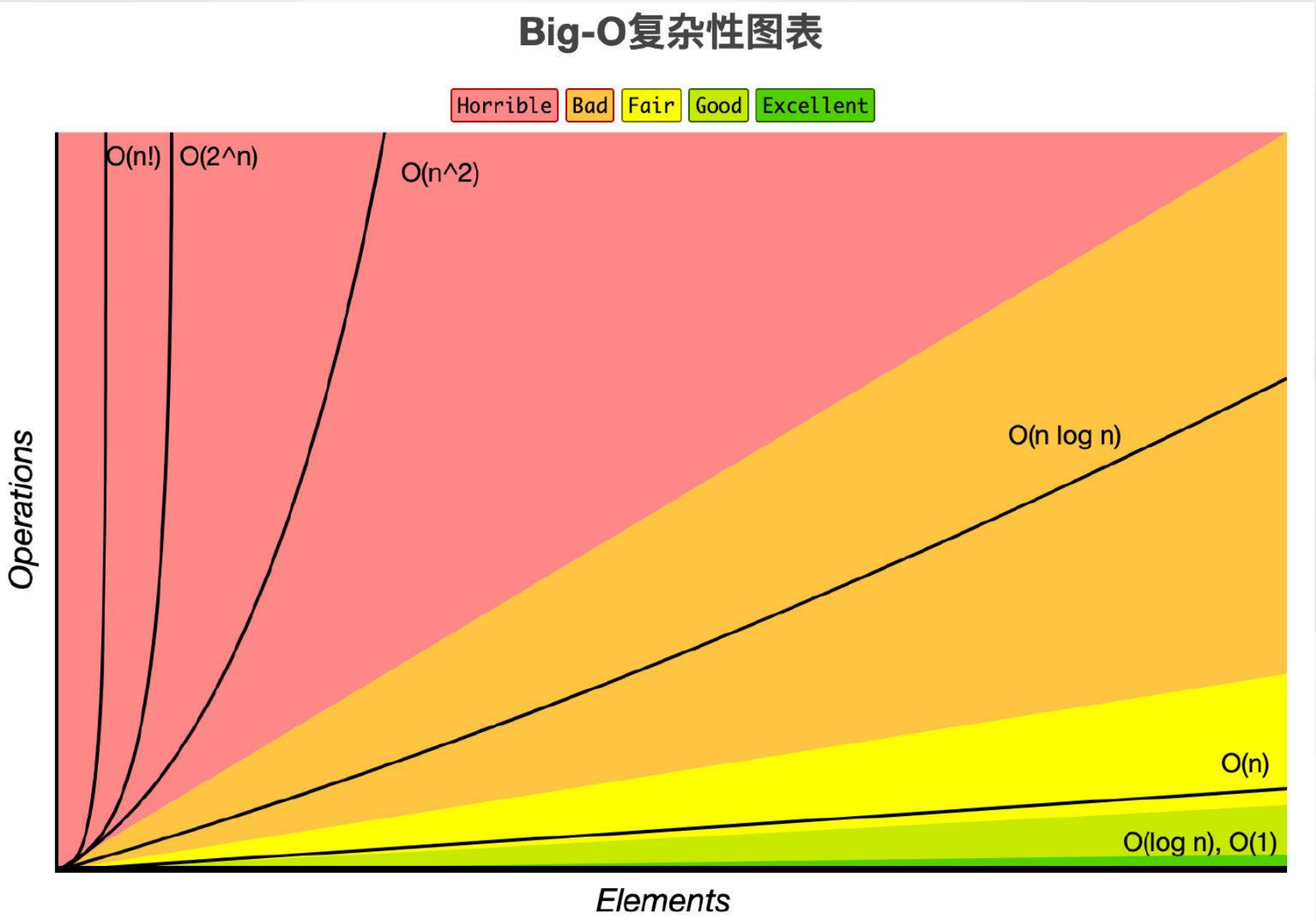
$$O(2^n)<O(n!)<O(n^n)$$

易解问题 - - 难解问题：用多项式时间来区分

- 一个算法是多项式复杂度：“高效”算法。
- 指数复杂度：“低效”算法。

- 多项式复杂度的算法，随着规模n的增加，可以通过堆叠硬件来实现，“砸钱”是行得通的；
- 指数复杂度的算法，增加硬件也无济于事，其增长的速度超出了想象力。

算法复杂度好坏示意图



图表引用原文https://www.bigocheatsheet.com/

数据结构的操作复杂度

数据结构	时间复杂度								空间复杂度
	平均				最差				最差
	访问	搜索	插入	删除	访问	搜索	插入	删除	
顺序表	$O(1)$	$O(n)$	$O(n)$	$O(n)$	$O(1)$	$O(n)$	$O(n)$	$O(n)$	$O(n)$
栈	$O(n)$	$O(n)$	$O(1)$	$O(1)$	$O(n)$	$O(n)$	$O(1)$	$O(1)$	$O(n)$
单链表	$O(n)$	$O(n)$	$O(1)$	$O(1)$	$O(n)$	$O(n)$	$O(1)$	$O(1)$	$O(n)$
双链表	$O(n)$	$O(n)$	$O(1)$	$O(1)$	$O(n)$	$O(n)$	$O(1)$	$O(1)$	$O(n)$
跳表	$O(\log(n))$	$O(\log(n))$	$O(\log(n))$	$O(\log(n))$	$O(n)$	$O(n)$	$O(n)$	$O(n)$	$O(n\log(n))$
散列表	–	$O(1)$	$O(1)$	$O(1)$	–	$O(n)$	$O(n)$	$O(n)$	$O(n)$
二叉搜索树	$O(\log(n))$	$O(\log(n))$	$O(\log(n))$	$O(\log(n))$	$O(n)$	$O(n)$	$O(n)$	$O(n)$	$O(n)$
笛卡尔树	–	$O(\log(n))$	$O(\log(n))$	$O(\log(n))$	–	$O(n)$	$O(n)$	$O(n)$	$O(n)$
B–树	$O(\log(n))$	$O(\log(n))$	$O(\log(n))$	$O(\log(n))$	$O(\log(n))$	$O(\log(n))$	$O(\log(n))$	$O(\log(n))$	$O(n)$
红黑树	$O(\log(n))$	$O(\log(n))$	$O(\log(n))$	$O(\log(n))$	$O(\log(n))$	$O(\log(n))$	$O(\log(n))$	$O(\log(n))$	$O(n)$
伸展树	–	$O(\log(n))$	$O(\log(n))$	$O(\log(n))$	–	$O(\log(n))$	$O(\log(n))$	$O(\log(n))$	$O(n)$
AVL 树	$O(\log(n))$	$O(\log(n))$	$O(\log(n))$	$O(\log(n))$	$O(\log(n))$	$O(\log(n))$	$O(\log(n))$	$O(\log(n))$	$O(n)$

数组的各种排序算法复杂度

算法	时间复杂度			空间复杂度
	最佳	平均	最差	最差
快速排序	$O(n \log(n))$	$O(n \log(n))$	$O(n^2)$	$O(\log(n))$
归并排序	$O(n \log(n))$	$O(n \log(n))$	$O(n \log(n))$	$O(n)$
Timsort	$O(n)$	$O(n \log(n))$	$O(n \log(n))$	$O(n)$
堆排序	$O(n \log(n))$	$O(n \log(n))$	$O(n \log(n))$	$O(1)$
冒泡排序	$O(n)$	$O(n^2)$	$O(n^2)$	$O(1)$
插入排序	$O(n)$	$O(n^2)$	$O(n^2)$	$O(1)$
选择排序	$O(n^2)$	$O(n^2)$	$O(n^2)$	$O(1)$
希尔排序	$O(n)$	$O((n \log(n))^2)$	$O((n \log(n))^2)$	$O(1)$
桶排序	$O(n + k)$	$O(n + k)$	$O(n^2)$	$O(n)$
基数排序	$O(nk)$	$O(nk)$	$O(nk)$	$O(n + k)$

图、堆操作复杂度

图操作

节点 / 边界管理	存储	增加顶点	增加边界	移除顶点	移除边界	查询
邻接表	$O(\|V\| + \|E\|)$	$O(1)$	$O(1)$	$O(\|V\| + \|E\|)$	$O(\|E\|)$	$O(\|V\|)$
邻接矩阵	$O(\|V\|^2)$	$O(\|V\|^2)$	$O(1)$	$O(\|V\|^2)$	$O(1)$	$O(1)$

堆操作

类型	时间复杂度					
	建堆	查找最大值	分离最大值	提升键	插入	删除	合并
（排好序的）链表	–	$O(1)$	$O(1)$	$O(n)$	$O(n)$	$O(1)$	$O(m+n)$
（未排序的）链表	–	$O(n)$	$O(n)$	$O(1)$	$O(1)$	$O(1)$	$O(1)$
二叉堆	$O(n)$	$O(1)$	$O(\log(n))$	$O(\log(n))$	$O(\log(n))$	$O(\log(n))$	$O(m+n)$
二项堆	–	$O(1)$	$O(\log(n))$	$O(\log(n))$	$O(1)$	$O(\log(n))$	$O(\log(n))$
斐波那契堆	–	$O(1)$	$O(\log(n))$	$O(1)$	$O(1)$	$O(\log(n))$	$O(1)$

多项式函数与指数函数的增长对比

问题规模n	多项式函数					指数函数	
	logn	n	nlogn	n^2	n^3	2^n	n!
1	0	1	0.0	1	1	2	1
10	3.3	10	33.2	100	1000	1024	3628800
20	4.3	20	86.4	400	8000	1048376	2.4E18
50	5.6	50	282.2	2500	125000	1.0E15	3.0E64
100	6.6	100	664.4	10000	1000000	1.3E30	9.3E157

问题规模和可用算法

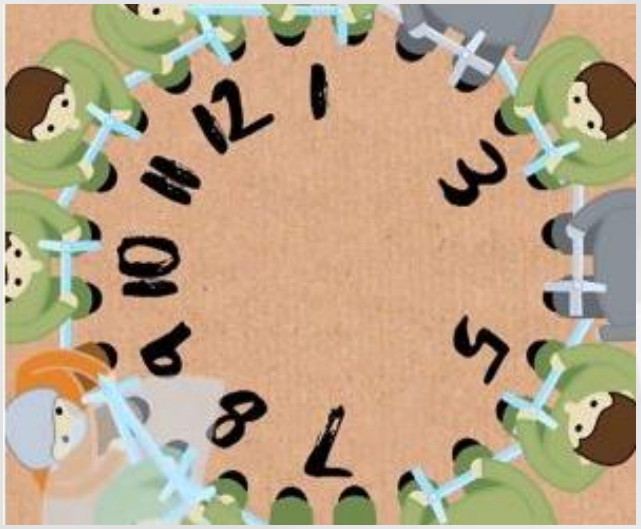
问题规模 n	可用算法的时间复杂度					
	O(logn)	O(n)	O(nlogn)	$O(n^2)$	$O(2^n)$	O(n!)
$n \leq 11$	√	√	√	√	√	√
$n \leq 25$	√	√	√	√	√	×
$n \leq 5000$	√	√	√	√	×	×
$n \leq 10^6$	√	√	√	×	×	×
$n \leq 10^7$	√	√	×	×	×	×
$n > 10^8$	√	×	×	×	×	×

最快的编程方法：打表法

- 一个程序题，你能猜出它所有的输入和输出。（“下界”）
- 如果数据量不大，就可以用“打表”的办法编程。

打表法编程

- Joseph问题http://poj.org/problem?id=1012
- 有k个坏人和k个好人坐成一圈，前k个为好人，后k个为坏人。
- 现在有一个报数m，从编号为1的人开始报数，报到m的人，就杀了他。
- 问当m为什么值时，可以使得在好人死之前，k个坏人先全部死掉?
- 0 < k < 14



一个数一个数地试：

当k=1时，1个好人1个坏人，m=2。

当k=2时，2个好人2个坏人，m=7。

……

当k=14时，m＝13482720。

“打表”：直接输出答案

```
#include <stdio.h>
int a[15]={0, 2, 7, 5, 30, 169, 441, 1872, 7632,
  1740, 93313, 459901, 1358657, 2504881, 13482720};
int main(){
      int k;
      while(scanf("%d", &k)==1){
          if(k==0)
              break;
          printf("%d\n", a[k]);
      }
      return 0;
}
```

课外阅读

- 《算法导论》Thomas H. Cormen，机械工业出版社
- Introduction to Algorithms, 3rd Edition (The MIT Press) (英语) Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, Clifford Stein
- Data Structures and Algorithms in Java (2nd Edition) (英语) - Robert Lafore (Author)
- 《算法设计与分析基础》Anany Levitin著，潘彦译