算法设计与分析

Algorithm Design and Analysis

课程内容 (64学时)

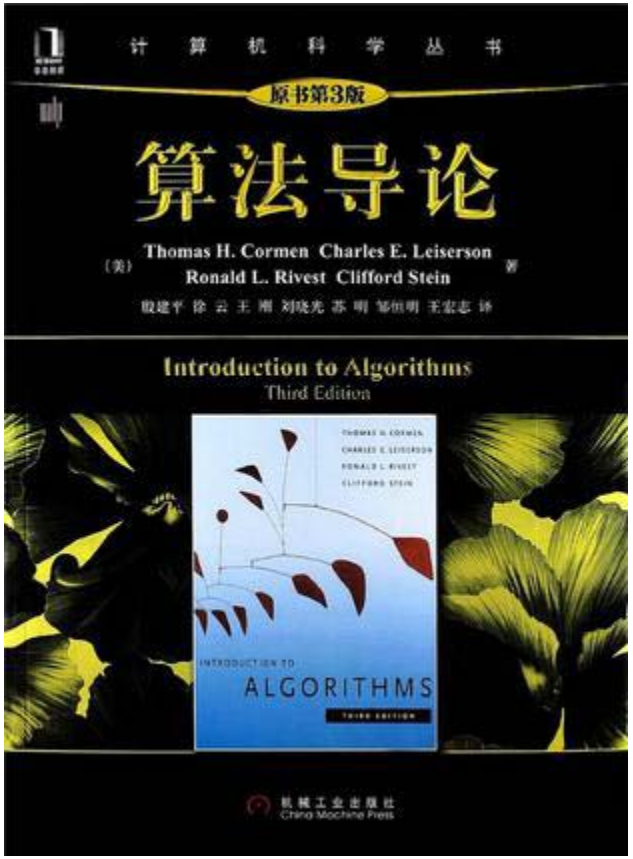
- 常用的算法设计策略(包括分治策略、动态规划、贪心策略、回溯法、随机算法等)
- 算法复杂度分析方法(计算迭代次数、使用递归方程、频度分析等)
- 其中：实验内容为16学时

课程要求

- 掌握常用的算法设计策略、算法复杂度分析方法
- 授课方式：讲授
- 考核方式：闭卷笔试

教材

- 《算法导论》(第3版)，机械工业出版社，2016



相关参考文献

- **Algorithm Design Techniques and Analysis, M. H. Alsuwaiyel. (影印本). 电子工业出版社. 2013**
- 计算机算法设计与分析(第4版), 王晓东, 电子工业出版社, 2012.
- 算法设计与分析， 王红梅编著， 清华大学出版社，2006.

引例一：斐波那契数列
目　的：考察不同算法的差异性

- 斐波那契数列是由13世纪的意大利数学家、来自Pisa的 Leonado Fibnacci发现。
- 斐波那契数列是由1，1开始，之后的每一项等于前两项之和：
- 1，1，2，3，5，8，13，21，34，55，89，144..... 。
- 这个数列有如下一些特性：
  - 前2个数相加等于第3个数
  - 前1个数除以后一个数越往后越无限接近于0.618 (黄金分割)
  - 相邻的两个比率必是一个小于0.618一个大于0.618
  - 后1个数除以前一个数越往后越无限接近于1.618
  - …

$$a_n = \begin{cases} 1 & , \ if \ n = 1, 2 \\ a_{n-1} + a_{n-2} & , \ if \ n \geq 3 \end{cases}$$

问题：编写一个程序(设计一个算法)，输入为整数n,
输出为第n项斐波那契数列的值。

1 算法引论

- 历史背景
- 算法复杂度与渐近分析方法
- 如何估计算法的复杂度
- 算法复杂度分析的意义

历史背景简介

- 阶段1：二十世纪30年代，能否使用一个有效的过程(相当于现在算法的概念)来求解一个给定的问题一直是人们所关注的。当时的焦点是将问题进行分类：可解或是不可解。关注问题是否可以求解的领域称为可计算理论(computability theory or theory of computation)。出现了一系列的计算模型，例如：calculus of Church、Post machines of Post、Turing machines of Turing、RAM model of computation。
- 阶段2：随着数字计算机的出现，人们越来越关注于那些可求解的问题。一开始，人们满足于能够在一定的时间内解决一个特定的问题，而不去关注所需要的资源。慢慢地，人们需要考虑在有限资源的条件下高效地解决问题。这就导致了计算复杂度(computational complexity)这一新学科的诞生。在这个领域，主要是研究解决可求解问题时所需要的资源，主要是时间和空间复杂性。有时候，其他的资源也需要考虑，例如，通信代价、需要使用的处理器的个数(使用并行计算模型)等等。

引例二：搜索问题

- 给定已经排好序(不妨假设为非降序)的n个元素A[1...n],现在要判定一个给定的元素x是否在此数组中出现。
  - 方法1：顺序搜索
  - 方法2：二分搜索

二分搜索算法

输入：非降序排列的数组A[1...n]和元素x
输出：如果x=A[j], 1 ≤j ≤n,则输出j,否则输出0.

```
1.  low←1; high ←n; j←0
2.  while(low ≤high)  and  (j=0)
3.      mid ←⌊( low+high)/2⌋
4.      if x=A[mid] then  j ←mid
5.      else if x<A[mid]  then  high ←mid-1
6.      else low ←mid+1
7.  end while
8.  return j
```

分析

- 最好情形：比较1次
- 最坏情形：比较$\lfloor \log n \rfloor+1$次
  - 每次循环都要抛弃一些元素，例如第二次循环时，剩余元素为A[1...mid-1]或A[mid+1...n]，不妨设为A[mid+1...n]，则剩余的元素个数是$\lfloor n/2 \rfloor$
  - 第j次while循环时，剩余元素的个数是$\lfloor n/2^{j-1} \rfloor$
  - 或者找到x，或者程序在子序列长度达到1时终止搜索，此时$\lfloor n/2^{j-1} \rfloor=1 \Leftrightarrow 1 \le n/2^{j-1} <2 \Leftrightarrow 2^{j-1} \le n< 2^{j} \Leftrightarrow \log n<j\le \log n+1 \Leftrightarrow j=\lfloor \log n \rfloor+1$

算法及其性质

- 算法是对问题求解过程的准确描述，由**有限条**指令组成，这些指令能在**有限时间**内执行完毕并产生确定性的输出。
- 算法需满足的4个性质：
  - **输入**：零个或多个外部量作为输入。
  - **输出**：至少产生一个量作为输出，它(们)与输入量之间存在某种特定的联系。
  - **确定性**：组成算法的每条指令都是清晰、无歧义的。
  - **有限性**：每条指令的执行次数有限，执行每条指令的时间也有限。
- 程序与算法的区别：
  - 程序是算法用某种程序设计语言的具体实现。
  - 程序可以不满足算法的有限性性质。

算法的复杂度

- 复杂度：算法运行时需要耗费计算机资源的量，可以分为时间复杂度和空间复杂度。
- 算法复杂度依赖于三个方面：待求解问题的规模、算法的输入和算法本身。(如何理解)
- 用n，I，A分别表示问题的规模、算法的输入和算法本身，用表示C复杂度，则有：C=F(n,I,A)。如果将时间复杂度和空间复杂度分开，则有T=T(n,I,A)和S=S(n,I,A)。通常，我们让A隐藏在复杂度函数名中，所以有T=T(n,I)和S=S(n,I)。
- 由于时间复杂度和空间复杂度概念类似，计量方法类似，且空间复杂度分析相对简单，因此本课程主要讨论算法的时间复杂度。

T=T(n,I)的具体化

- 假设计算机提供k种元运算，分别记为$O_1, O_2,...,O_k$。所谓元运算指的是这样一类运算：不管使用什么样的算法和输入数据，该运算的上阶是一个常数时间(关于阶，参见后文描述)。例如，算术运算、比较、逻辑、赋值等。
- 元运算O_i每执行一次需要的时间为t_i。
- 对于给定的算法A，已经知道用到的元运算O_i的执行次数为e_i, i = 1,...,k。很显然e_i是n和I的函数，即$e_i = e_i(n, I)$。
- 至此，就有：

$$T(n,I) = \sum_{i=1}^{k} t_i e_i(n,I)$$

分析

- T(n,I)给出的是在问题规模为n，输入为I时算法的时间复杂度。但是我们不可能，也没必要对每种可能的输入都去求其时间复杂度。通常可以考虑3种情形下的时间复杂度：最坏情形、最好情形以及平均情形。

$$T_{\max}(n)=\max_{I\in D_n}\sum_{i=1}^{k}t_i e_i(n,I)=\sum_{i=1}^{k}t_i e_i(n,I^*)=T(n,I^*)$$

$$T_{\min}(n)=\min_{I\in D_n}\sum_{i=1}^{k}t_i e_i(n,I)=\sum_{i=1}^{k}t_i e_i(n,I')=T(n,I')$$

$$T_{avg}(n)=\sum_{I\in D_n}P(I)T(n,I)=\sum_{I\in D_N}P(I)\sum_{i=1}^{k}t_i e_i(n,I)$$

实践表明：可操作性最强、最有实际价值的是算法最坏情形下的时间复杂度。

自此，如不作特殊说明，使用T(n)表示给定算法在最坏情形下的时间复杂度.

使用算法的**绝对运行时间**来度量其时间复杂度？

- 一个编程实现了的算法的**绝对运行时间**，不仅仅和算法本身相关，还和很多其他因素密切相关：机器性能、编程语言、编译器、编程技巧等等。
- 在分析一个算法的运行时间时，通常将该算法和其他算法在同一问题、甚至是不同的问题上进行比较，因而运行时间只能是相对的，而不是绝对的。
- 希望算法的描述不仅独立于机器，并且可以以任何语言来加以描述，包括自然语言。
- 希望使用度量算法运行时间的准则不依赖于软硬技术的进步。
- 不仅仅关注小规模输入下的，而且还关注在大规模输入下的情形。

渐近分析(Asymptotic Analysis)

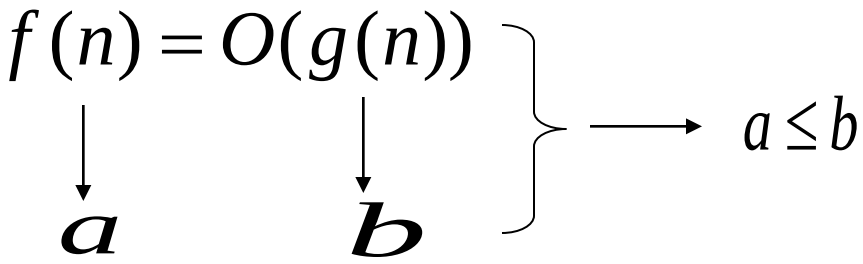
- 对于T(n)，当n单调递增并趋于∞时，T(n)也是单调增加并趋于∞。为此,如果存在一个T*(n)，使得当n → ∞时有(T(n) – T*(n)) /T(n) → 0,就称T*(n)是T(n)当n → ∞的渐近性态，或称T*(n)是给定算法在n →∞时的渐近复杂度。
- 显然,T*(n)不是唯一的。我们可以尽可能的选择简单的T*(n)，然后使用T*(n)来替代T(n)作为n → ∞的复杂度度量。渐近复杂度分析只要关心T*(n)的阶就可以了(在n充分大时)，不必关心其中的常数因子(原因后释)。

4种阶：O, Ω, Θ,和o

- 假设：f (n) 和g(n) 是定义在正数集上的正函数。(为何如此假设？实际意义?)
- 定义1 (O)：如果存在正的常数C和自然数n_0，使得当$n \geq n_0$时，有$f(n) \leq C \cdot g(n)$，则称函数f (n) 在n 充分大时有上有界，且g(n) 是它的一个上界，记做f (n) = O(g(n))，并称f (n) 的阶不高于g(n) 的阶。

例子

- 例：$f(n)=n^2$，$g(n)=n^3$。因为：存在$n_0=1$，C=1，当$n \geq n_0$时，有$n^2 \leq Cn^3$，所以：$n^2=O(n^3)$
- 例：$f(n)=n^2$，$g(n)=n^2$。因为：存在$n_0=1$，C=1，$n \geq n_0$时，有$n^2 \leq Cn^2$，所以：$n^2=O(n^2)$
- 例：$f(n)=n^2+n\log(n)$，$g(n)=n^2$。同样有$n^2+n\log(n)=O(n^2)$
- 例：$f(n)=an^2+n\log(n)$，$g(n)=n^2$。同样有$an^2+n\log(n)=O(n^2)$
- 例：$f(n)=an^2+n\log(n)+c$，$g(n)=n^2$。同样有$an^2+n\log(n)+c=O(n^2)$

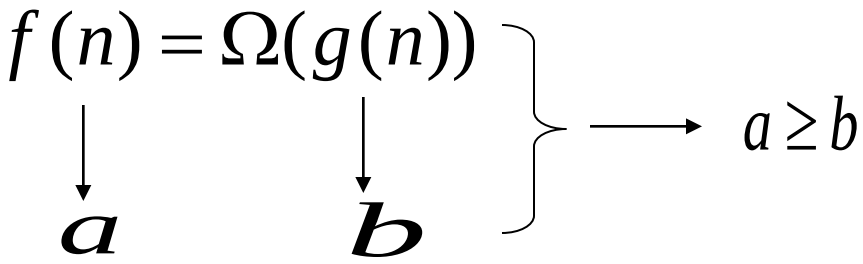

$f(n) = O(g(n))$
a
b
$a \leq b$

小结

- 在进行阶的运算时，常系数、低的阶以及常数项可以忽略。
- 根据O的定义，得到的是在问题规模充分大时，算法复杂度的一个上界。上界的阶越低则评估越有价值。
- 运算规则
  - O(f) + O(g) = O(max(f , g));
  - O(f)·O(g) = O(f·g);
  - O(C·f (n)) = O(f (n));
  - f = O(f);

Ω

- 定义2(Ω)：如果存在正的常数C和自然数n_0，使得当$n \geq n_0$时，有$f(n) \geq C \cdot g(n)$，则称函数f(n)在n充分大时有下有界，且g(n)是它的一个下界，记做$f(n) = \Omega(g(n))$，并称f(n)的阶不低于g(n)的阶。
- 下界的阶越高，则评估精度越高，也就越有价值。

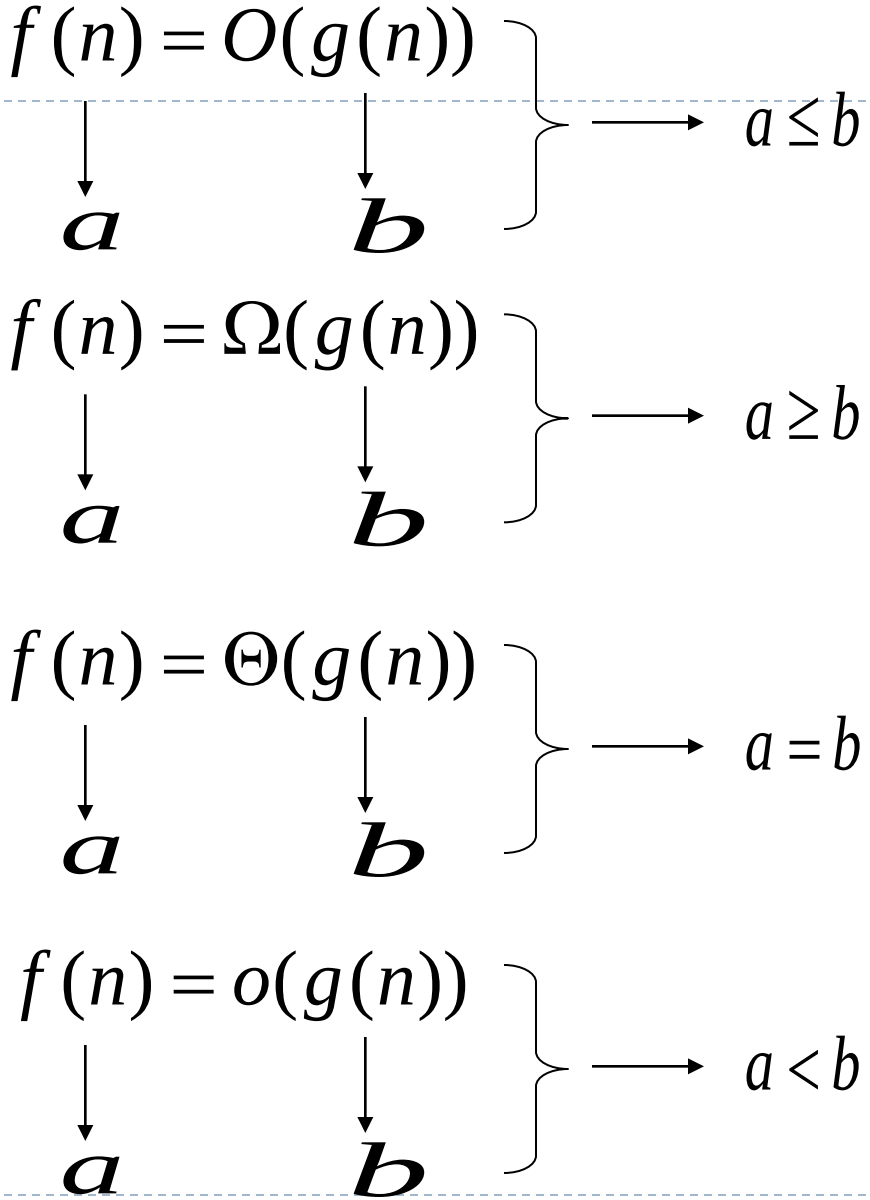

$f(n) = \Omega(g(n))$
a
b
$a \geq b$

Θ和o

- 定义3(Θ)： $f(n) = \Theta(g(n))$，当且仅当$f(n) = O(g(n))$，且$f(n) = \Omega(g(n))$，称$f(n)$和$g(n)$是同阶。
- 定义4(o)：对于任意给定的$\varepsilon > 0$，存在正整数n_0，使得当$n \geq n_0$时，有$f(n) / g(n) \leq \varepsilon$，则称函数$f(n)$在n充分大时的阶比$g(n)$低，记为$f(n) = o(g(n))$。

小结

- 进行算法的时间复杂度分析，就是求其T(n)，并用O、Ω或是Θ以尽可能简单的形式进行表示。
- 理想情况下，希望能够使用Θ表示算法的时间复杂性。退而求其次，可以使用O或是Ω。
- 使用O时，希望估计的上界的阶越小越好。
- 使用Ω时，希望估计的下界的阶越大越好。


$f(n) = O(g(n))$
a
b
$a \leq b$
$f(n) = \Omega(g(n))$
a
b
$a \geq b$
$f(n) = \Theta(g(n))$
a
b
$a = b$
$f(n) = o(g(n))$
a
b
$a < b$

算法耗费时间随问题规模的变化

Algorithm	1	2	3	4	5
Time function(ms)	$33n$	$46n \lg n$	$13n^2$	$3.4n^3$	2^n
Input size(n)	Solution time				
10	0.00033 sec.	0.0015 sec.	0.0013 sec.	0.0034 sec.	0.001 sec.
100	0.0033 sec.	0.03 sec.	0.13 sec.	3.4 sec.	4×10^{16} yr.
1,000	0.033 sec.	0.45 sec.	13 sec.	0.94 hr.	
10,000	0.33 sec.	6.1 sec.	22 min.	39 days	
100,000	3.3 sec.	1.3 min.	1.5 days	108 yr.	
Time allowed	Maximum solvable input size (approx.)				
1 second	30,000	2,000	280	67	20
1 minute	1,800,000	82,000	2,200	260	26

阶运算的几个实例

(1) $n! = \Theta(n^n)$

解：因为 $0 < \frac{n!}{n^n} = \frac{n(n-1)(n-2)\cdots 1}{nn\cdots n} < \frac{nnn\cdots 1}{nn\cdots n} \le \frac{1}{n}$ 所以 $n! = o(n^n)$

(2) $\log n! = \Theta(\log n^n)$

解: a. 因为 $\log n! < \log n^n$ 有 $\log n! = O(\log n^n)$

b.又因为 $\log n! = \sum_{j=1}^{n} \log j$

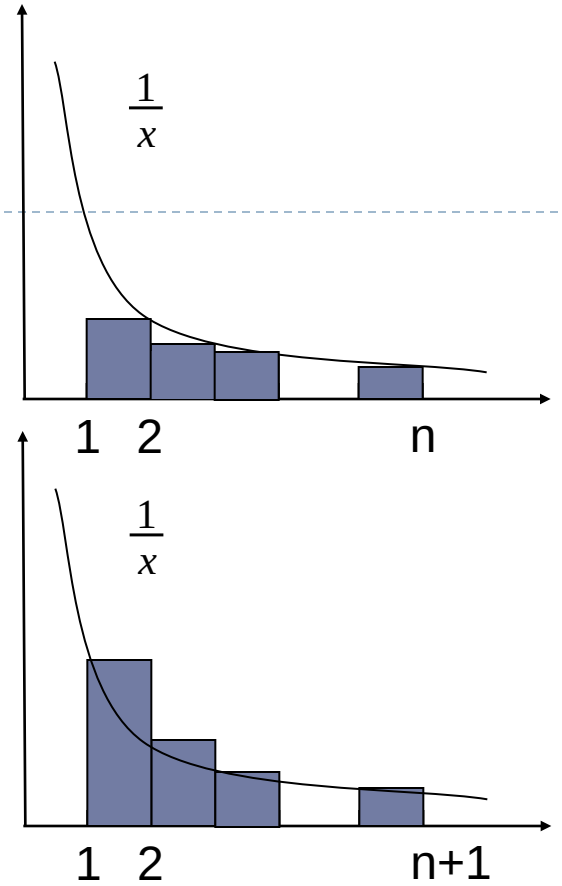
$$= \log 1 + \log 2 + \cdots + \log(\frac{n}{2}) + \cdots + \log n$$

$$\ge \sum_{j=1}^{\lfloor n/2 \rfloor} \log(\frac{n}{2}) = \lfloor n/2 \rfloor \log(\frac{n}{2}) = \lfloor n/2 \rfloor \log(n) - \lfloor n/2 \rfloor$$

则有 $\log n! = \Omega(\lfloor n/2 \rfloor \log n - \lfloor n/2 \rfloor) = \Omega(n \log n)$

命题成立

(3) $n\sum_{i=1}^{n}\frac{1}{i}=\Theta(n\log n)$



解:

a. $\sum_{i=1}^{n}\frac{1}{i}=1+\sum_{i=2}^{n}\frac{1}{i}\le 1+\int_{1}^{n}\frac{1}{x}dx=1+\ln n$

b. $\sum_{i=1}^{n}\frac{1}{i}\ge \int_{1}^{n+1}\frac{1}{x}dx=\ln(n+1)$

所以 $\ln(n+1)\le\sum_{i=1}^{n}\frac{1}{i}\le 1+\ln n$

换底公式 $\frac{\log(n+1)}{\log e}\le\sum_{i=1}^{n}\frac{1}{i}\le 1+\frac{\log n}{\log e}$

所以 $\sum_{i=1}^{n}\frac{1}{i}=O(1+\frac{\log n}{\log e})=O(\log n)$

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

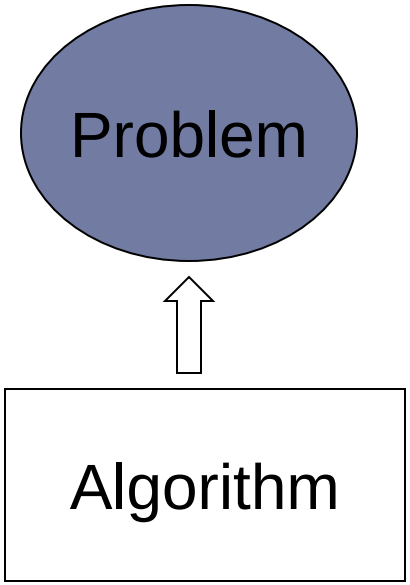
亦即 $n\sum_{i=1}^{n}\frac{1}{i}=\Theta(n\log n)$

(4) $f(n) = O(g(n)) \Rightarrow f(n) = o(g(n))$ x

$f(n) = o(g(n)) \Rightarrow f(n) = O(g(n))$ √

$f(n) = o(g(n)) \Rightarrow g(n) = \Omega(f(n))$ √

如何计算一个算法的时间复杂度



$T(n) = ?$

- 计算迭代次数
- 使用递归方程
- 频度分析

计算迭代次数

- 通常，算法耗费的时间和算法的迭代次数成比例。因此计算算法的迭代次数就可以作为算法运行时间的指示器。这在很多算法中都可以得到应用，如查找、排序等等。

例：

输入： n (n = 2^k， k 为某一正整数)

输出： count

1. count ← 0
2. while n ≥ 1　　$n \geq 2^0$
3. for j ← 1 to n
4. count ← count+1　　//执行一次耗费时间设为a
5. end for
6. n ← n/2　　//执行一次耗费时间设为d
7. end while
8. return count

分析：while 迭代的次数是k +1次(因为n≥1 可以写成$n \geq 2^0$，算法运行过程 $n=2^k \rightarrow 2^0$)

显然，k = log n 。在每次while 循环里面for 依次执行 $n/2^{\log n}, \ldots, n/2^2, n/2^1, n/2^0$次。所以，算法耗费的时间为：

$$T(n) = \sum_{j=0}^{k} a(n/2^j) = an\sum_{j=0}^{k} 2^{-j} = an(2 - 1/2^k) = 2an - a = \Theta(n)$$

思考？

- 为什么在上面计算算法的时间复杂度时不考虑step 6，而只是考虑step 4呢？
- 如果同时考虑 step 4和step 6，我们有：

$$T(n) = (\sum_{j=0}^{k} n/2^j)a + (k+1)d$$

$$= an\sum_{j=0}^{k} 2^j + (k+1)d = na(2 - 1/2^k) + (k+1)d$$

$$= 2an - a + d + d\log n = \Theta(n)$$

小结：

使用计算迭代次数的技术来分析算法的时间复杂度时，只需要考虑最深层次的迭代。

例：

输入：正整数n

输出：step 5 的执行次数

1. count ← 0
2. for i ← 1 to n
3. m←⌊n/i⌋
4. for j ← 1 to m
5. count ← count+1
6. end for
7. end for
8. return count

分析：Step5 的执行次数依次为：
⌊n/1⌋, ⌊n/2⌋, ⌊n/3⌋, …, ⌊n/n⌋

$$T(n)=\sum_{i=1}^{n}\left\lfloor\frac{n}{i}\right\rfloor$$

因为 $\sum_{i=1}^{n}(\frac{n}{i}-1)\le\sum_{i=1}^{n}\left\lfloor\frac{n}{i}\right\rfloor\le\sum_{i=1}^{n}\frac{n}{i}$

所以 $T(n)=\Theta(n\log n)$

2 使用递归方程

例：二分搜索（算法略）

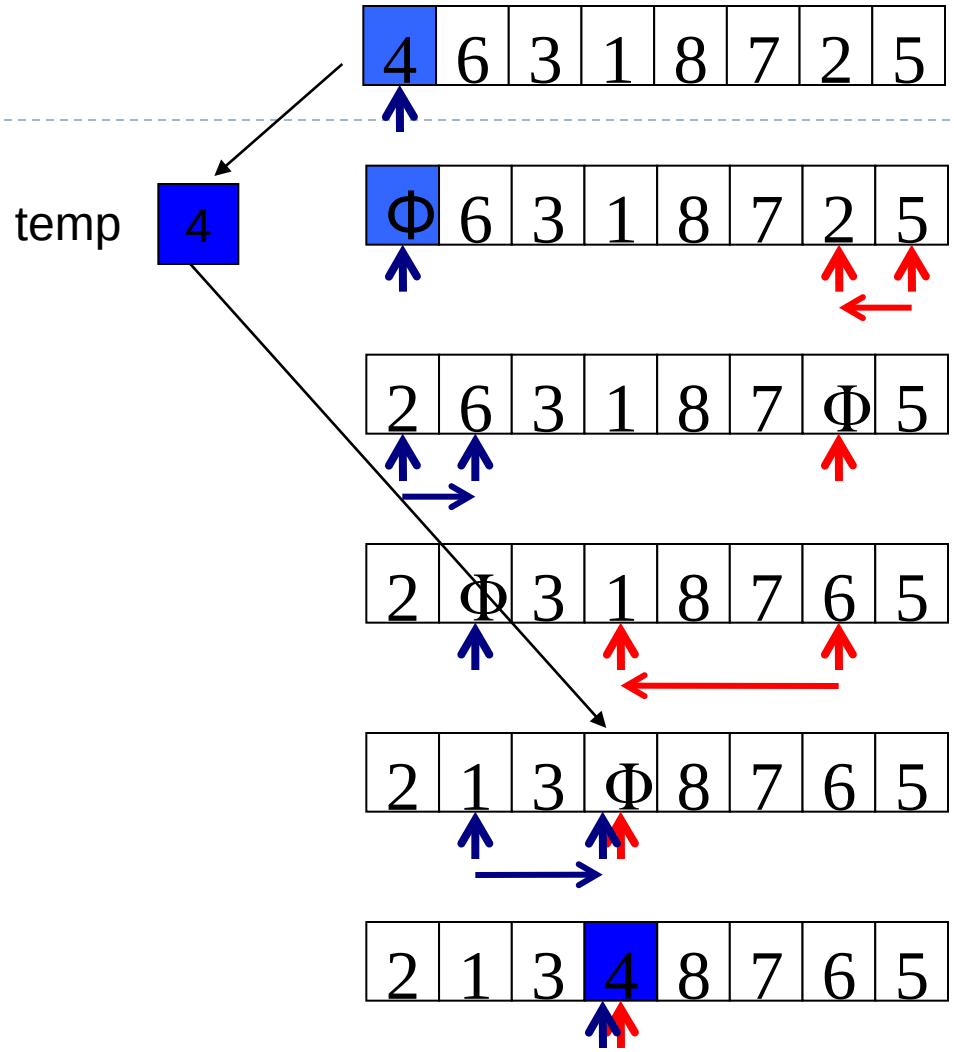
$$T(n)=\begin{cases}1 & if\ n=1\\ T(n/2)+1 & if\ n\geq 2\end{cases}$$

$$\Rightarrow T(n)=T(n/2)+1=T(n/2^2)+1+1$$

$$=T(n/2^{\log n})+\log n=1+\log n=\Theta(\log n)$$

例：快速排序

- 使用递归方程分析快速排序的时间复杂度。
- 什么条件下快速排序就“不快”了？
- 如何解决？


temp
4
4 6 3 1 8 7 2 5
Φ 6 3 1 8 7 2 5
2 6 3 1 8 7 Φ 5
2 Φ 3 1 8 7 6 5
2 1 3 Φ 8 7 6 5
2 1 3 4 8 7 6 5

理想情形：每次SPLIT后得到的左右子数组规模相当，因此有：

$$T(n) = 2T(n/2) + n \quad \Longrightarrow \quad T(n) = \Theta(n \log n)$$

最差情形(已经排好序或是逆序的数组)：每次SPLIT后，只得到左或是右子数组，因此有：

$$T(n) = \begin{cases} O(1) & if\ n = 1 \\ T(n-1) + \Theta(n) & if\ n > 1 \end{cases} \quad \Longrightarrow \quad T(n) = \Theta(n^2)$$

3 频度分析

- 对于有些算法，要像前面那样准确地分析算法的运行时间非常麻烦，有时甚至是不可能的。这时候，可以使用频度分析，我们将在后续章节中进行讲解。(单源最短路径、Prim 算法等等)

算法复杂度分析的意义

- 已知待求解问题的多种算法时，挑选复杂度尽可能低的算法进行应用。
- 给定待求解的问题，设计复杂度尽可能低的算法。
- 设计出算法后，不要急于实现，而是先进行复杂度分析后；若该算法确实可行，才有实现的价值与必要。

研究算法还有必要么？

- 随着计算机设计和制造技术的突飞猛进，计算机的计算速度和存储容量在不断增长。有的人因此认为不必要再去苦苦追求高效率的算法。认为低效的算法可以由高速的计算机来弥补，认为在可接受的一定时间内用低效的算法完不成的任务，只要移植到高速的计算机上就能完成。
- 造成上述错误认识的原因是他们没看到：随着经济的发展、社会的进步、科学研究的深入，要求计算机解决的问题越来越复杂、规模越来越大，也呈增长之势；而问题复杂程度和规模的线性增长导致的时耗的增长和空间需求的增长，对低效算法来说，都是非线性的，决非计算机速度和容量的线性增长带来的时耗减少和存储空间的扩大所能抵销。

几点说明

- “低阶复杂度的算法比高阶复杂度的算法效率高”这个结论，只是在问题的规模充分大时才成立。
  - 复杂度分别为n^3与$10n^2$：当$n>10$时，$n^3>10n^2$; 而$n<10$时，$n^3<10n^2$，即复杂度为n^3的算法更有效。
- 在问题规模较小时，我们往往并不一味追求低复杂度的算法，而是更侧重于算法的简单性。
- 当两个算法的渐近复杂度的阶相同时，必须进一步综合考察渐近复杂度表达式中的低阶及常数因子才能判别它们的优劣。
  - 例如：复杂度为n1ogn/l00 的算法显然比复杂度为l00n1ogn的算法来得有效。

作业

- 课堂练习

  题3.2, 3.4。

3-2 （相对渐近增长） 为下表中的每对表达式(A, B)指出A是否是B的O、o、Ω、ω或Θ。假设$k \geqslant 1$、$\epsilon > 0$且$c > 1$均为常量。回答应该以表格的形式，将“是”或“否”写在每个空格中。

	A	B	O	o	Ω	ω	Θ
a.	$\lg^k n$	n^ϵ					
b.	n^k	c^n					
c.	$\sqrt{n}$	$n^{\sin n}$					
d.	2^n	$2^{n/2}$					
e.	$n^{\lg c}$	$c^{\lg n}$					
f.	$\lg(n!)$	$\lg(n^n)$					

3-4 （渐近记号的性质） 假设$f(n)$和$g(n)$为渐近正函数。证明或反驳下面的每个猜测。

a. $f(n) = O(g(n))$蕴涵$g(n) = O(f(n))$。

b. $f(n) + g(n) = \Theta(\min(f(n), g(n)))$。

c. $f(n) = O(g(n))$蕴涵$\lg(f(n)) = O(\lg(g(n)))$，其中对所有足够大的n，有$\lg(g(n)) \geqslant 1$且$f(n) \geqslant 1$。

d. $f(n) = O(g(n))$蕴涵$2^{f(n)} = O(2^{g(n)})$。

e. $f(n) = O((f(n))^2)$。

f. $f(n) = O(g(n))$蕴涵$g(n) = \Omega(f(n))$。

g. $f(n) = \Theta(f(n/2))$。

h. $f(n) + o(f(n)) = \Theta(f(n))$。