数据结构:算法分析

Data Structure

主讲教师：屈卫兰

Office number：基地203

Email：5604293@qq.com

教学要点

- 算法基本概念
- 算法分析术语
- 算法分析概念
- 算法分析方法

算法基本概念

- 算法：
  - 指令的有限序列，其中每一条指令表示一个或多个操作
- 问题与算法
  - 算法可以看为解决问题的方法和步骤
  - 一个问题可以用多种算法来解决，需要比较不同算法的效率，为此引入算法代价

算法的代价

- 算法的代价是算法的效率的度量
- 是算法运行所需要的计算机资源的量，常包括：
  - 时间代价
    - 需要的时间资源的量
  - 空间代价
    - 需要的空间(即存储器)资源的量
- 应集中反映算法所采用的方法的效率，而与运行该算法的硬件、软件环境无关

算法分析

- 概述
- 最佳、最差、平均情况分析
- 换一台更快的计算机，还是换一种更快的算法
- 渐近分析
- 程序运行时间的计算

3.1概述

如何比较两种算法解决问题的效率呢?

- 事后统计方法，就是用源程序分别实现这两种算法，然后输入适当的数据运行，测算两个程序各自的开销。
- 事前分析估算的方法，称为渐进算法分析(asymptotic algorithm analysis)，简称算法分析(algorithm analysis)。它可以估算出当问题规模变大时，一种算法及实现它的程序的效率和开销。

运行时间

- 编译时间
- 运行时间

通常情况下，程序的运行时间取决于下列因素：

算法所采用的策略

问题规模

书写程序采用的语言

机器速度

基本操作

为便于分析和对比，不考虑所用的机器，并约定用同一种语言来书写程序。这样一来，仅需考虑算法策略和问题规模对时间的影响情况了。

在这种情况下，以算法中基本操作 (*basic operation*) 数衡量算法的时间性能。一般情况下，执行次数是问题规模的函数。

- 规模：一般是指输入量的数目。
- 基本操作：一个“基本操作”必须具有这样的性质：完成该操作所需时间与操作数的具体取值无关。

举例

例　以迭代方式求累加和的函数

```
行   float sum ( float a[ ], const int n )
 1   {
 2      float s = 0.0;
 3      for ( int i=0; i<n; i++ )
 4         s += a[i];
 5      return s;
 6   }
```

在求累加和程序中加入count语句

```
float sum ( float a[ ], const int n )  {
   float s = 0.0;
   count++;   //count统计执行语句条数
   for ( int i=0; i<n; i++ ) {
        count++;          //针对for语句
         s += a[i];
         count++;
    }   //针对赋值语句
    count++;  //针对for的最后一次
    count++;  //针对return语句
    return s;
}
```

执行结束得 程序步数 count =2 * n + 3

时间代价

经常把执行算法所需要的时间T写成输入规模n的函数，记为T(n)。

程序段：

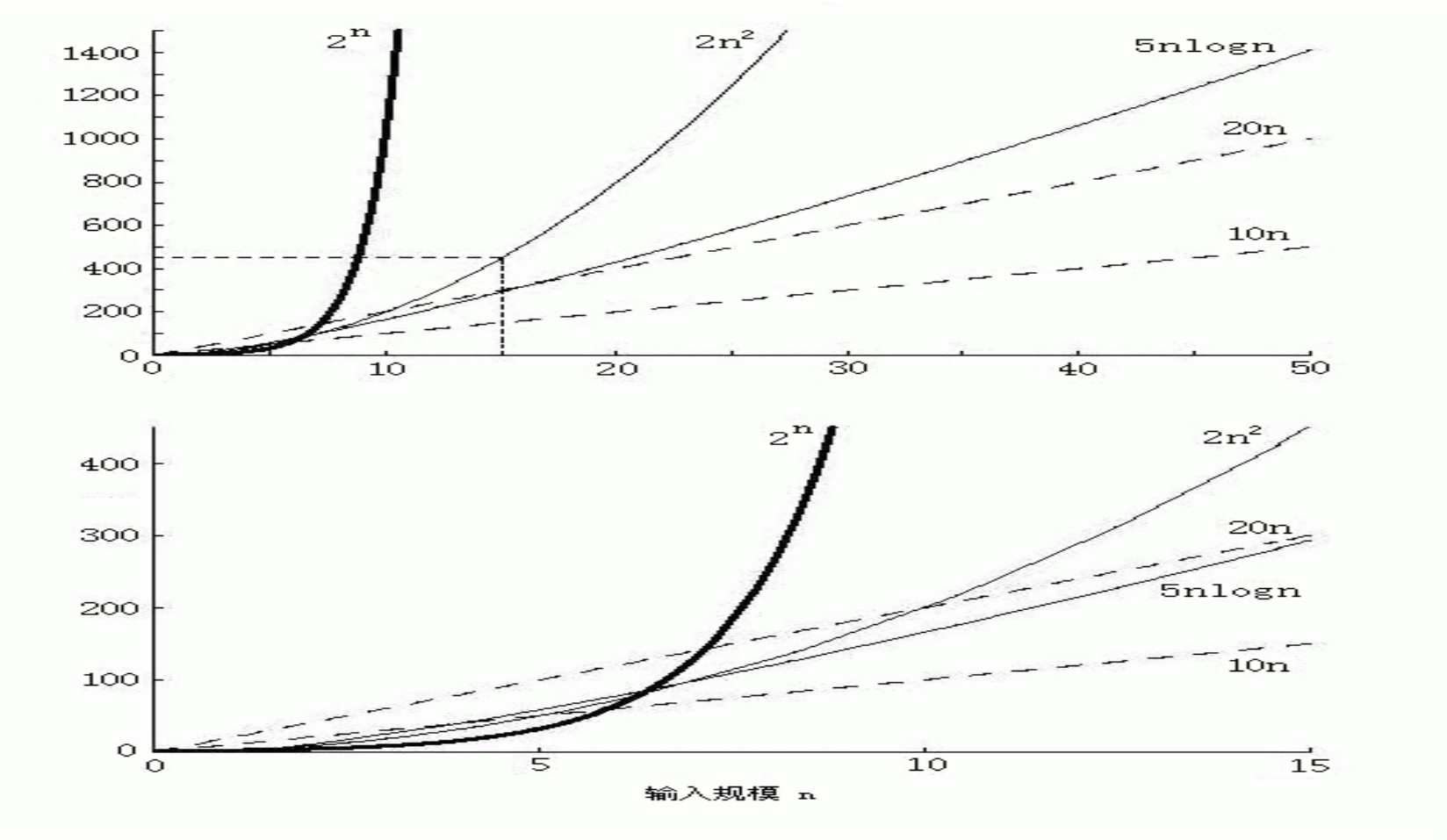
```
sum = 0;
for (i=1; i<=n; i++)
  for (j=1; j<=n; j++)
    sum++;
```

$T(n) = cn^2$

常见时间代价

n	log n	n	n log n	n^2	n^3	2^n
1	0	1	0	1	1	2.E+00
10	3	10	33	100	1000	1.E+03
20	4	20	86	400	8000	1.E+06
30	5	30	147	900	27000	1.E+09
40	5	40	213	1600	64000	1.E+12
50	6	50	282	2500	125000	1.E+15
60	6	60	354	3600	216000	1.E+18
70	6	70	429	4900	343000	1.E+21
80	6	80	506	6400	512000	1.E+24
90	6	90	584	8100	729000	1.E+27
100	7	100	664	10000	1000000	1.E+30

增长率函数曲线


1400
1200
1000
800
600
400
200
0
0
10
20
30
40
50
2^n
$2n^2$
5nlogn
20n
10n
400
300
200
100
0
0
5
10
15
2^n
$2n^2$
20n
5nlogn
10n
输入规模 n

3.2 最佳、最差、平均情况分析

- 对于不同的输入情况，算法的时间代价不一样
  - 例如，在一个数组中查找元素K
- 往往分为最佳、最差、平均情况分析的方式。

	Time for $f(n)$ instructions on a 10^9 instr/sec computer						
n	$f(n)=n$	$\log_2 n$	n^2	n^3	n^4	n^{10}	2^n
10	.01μs	.03μs	.1μs	1μs	10μs	10sec	1μs
20	.02μs	.09μs	.4μs	8μs	160μs	2.84hr	1ms
30	.03μs	.15μs	.9μs	27μs	810μs	6.83d	1sec
40	.04μs	.21μs	1.6μs	64μs	2.56ms	121.36d	18.3min
50	.05μs	.28μs	2.5μs	125μs	6.25ms	3.1yr	13d
100	.10μs	.66μs	10μs	1ms	100ms	3171yr	$4*10^{13}$yr
1,000	1.00μs	9.96μs	1ms	1sec	16.67min	$3.17*10^{13}$yr	$32*10^{283}$yr
10,000	10μs	130.03μs	100ms	16.67min	115.7d	$3.17*10^{23}$yr	
100,000	100μs	1.66ms	10sec	11.57d	3171yr	$3.17*10^{33}$yr	
1,000,000	1.0ms	19.92ms	16.67min	31.71yr	$3.17*10^{7}$yr	$3.17*10^{43}$yr	

μs = microsecond = 10^{-6} seconds
ms = millisecond = 10^{-3} seconds
sec = seconds
min = minutes
hr = hours
d = days
yr = years

3.3 换一台更快的计算机，还是换一种更快的算法

- 计算机相同，不同算法的时间代价差异可能很大

例：假设CPU每秒处理10^6个指令，对于输入规模为$n=10^8$的问题，时间代价为$T(n)=2n^2$的算法要运行多长时间？

- 操作次数为 $T(n)=T(10^8)=2\times(10^8)^2=2\times10^{16}$
- 运行时间为 $2\times10^{16}/10^6=2\times10^{10}$秒
- 每天有86,400秒，因此需要231480 天(634年)

运行时间估计

例：假设CPU每秒处理10^6个指令，对于输入规模为$n=10^8$的问题，时间代价为$T(n)=n\log n$的算法要运行多长时间？

–操作次数为

$$T(n)=T(10^8)= 10^8 \times \log 10^8 = 2.66\times 10^9$$

–运行时间为

$$2.66\times 10^9/10^6 = 2.66\times 10^3 \text{秒}，即44.33分钟$$

运行时间估计

- 假设CPU每秒处理10^6个指令，则每小时能够解决的最大问题规模
  - $T(n)/10^6 \leqslant 3600$
- 对$T(n) = 2n^2$，
  - 即$2n^2 \leqslant 3600 \times 10^6$
  - $n \leqslant 42,426$
- $T(n) = n\log n$
  - 即$n\log n \leqslant 3600 \times 10^6$
  - $n \leqslant 133,000,000$

运行时间估计

T(n)	处理输入规模为n = 10^8		1小时内解决的问题规模	
	10^6指令/秒	10^8指令/秒	10^6指令/秒	10^8指令/秒
nlogn	44.33分钟	0.4433分钟	1.33 亿	~100亿
$2n^2$	634年	6.34年	42，426	~424,264

- 设CPU每秒处理10^8个指令（快100倍）
  - –处理时间都降为原来的1/100
  - –解决问题的规模

    对$T(n)=2n^2$，规模增加10倍； $T(n)=n\log n$，规模增加75倍

快10倍的计算机所能解决的问题规模

T(n)	n	n'	Change	n'/n
$10n$	1,000	10,000	$n' = 10n$	10
$20n$	500	5,000	$n' = 10n$	10
$5n \log n$	250	1,842	$\sqrt{10}\, n < n' < 10n$	7.37
$2n^2$	70	223	$n' = \sqrt{10n}$	3.16
2^n	13	16	$n' = n + 3$	-----

因此，换电脑对于指数级时间代价的算法没有什么作用

3.4 渐近分析

渐近分析是指当输入规模$n \to \infty$时，对算法运行时间函数T(n)的渐近性态的估算，它提供了对算法资源开销进行评估的简单化的模型。

3.4.1 上限

渐近分析——大0表示法

算法运行时间的上限(upper bound)用来表示该算法可能有的最高增长率。增长率的上限用符号O表示, 称为大O表示法(big-Oh notation)。

定义3.1 对于非负函数T(n), 若存在两个正常数c和n0, 使得当n>n0时有T(n)≤cf(n), 则称函数T(n)当n充分大时有上限, 且f(n)是它的一个上限, 记为T(n)∈O(f(n)), 或T(n)在集合O(f(n))中。也称T(n)的阶不高于f(n)的阶。

举例

例 考虑找出数组中某个特定元素的顺序检索法。若访问并检查数组中的一个元素需要时间c_s(c_s为正数), 那么在平均情况下$T(n)=c_sn/2$。当$n>1$时, $c_sn/2 \le c_sn$, 所以根据定义，$T(n)$在$O(n)$中，$n_0=1$，$c=c_s$。

例 某一算法平均情况下 $T(n)=c_1n^2+c_2n$, c_1、c_2为正数。当$n>1$时, $c_1n^2+c_2n \le c_1n^2+c_2n^2 \le (c_1+c_2)n^2$。因此取$c=c_1+c_2$, $n_0=1$, 有$T(n) \le cn^2$。根据定义, $T(n)$在$O(n^2)$中。

例 把数组中第一个元素的值赋给一个变量，这个算法的运行时间是一个常量，与数组大小无关。因此，在最佳、最差和平均情况下恒有$T(n)=c$ 。我们可以认为这种情况下$T(n)$在$O(c)$中。

3.4.2 下限

算法的下限用符号Ω表示，称为大Ω表示法。

定义3.2 对于非负函数T(n)，若存在两个正常数c和n0，使得当n>n0时有T(n)≥cg(n)，则称函数T(n)当n充分大时有下限，且g(n)是它的一个下限，记为T(n)∈Ω(g(n))，或T(n)在集合Ω(g(n))中。也称T(n)的阶不低于g(n)的阶。

举例

例 假定$T(n)=c_1n^2+c_2n$ $(c_1, c_2>0)$，则有

$c_1n^2+c_2n \geqslant c_1n^2$ $(n>1)$

因此，取$c=c_1$，$n_0=1$，有$T(n) \geqslant cn^2$，根据定义，$T(n)$在$\Omega(n^2)$中。

3.4.3 Θ表示法

大O表示法和Ω表示法使我们能够描述某一算法的上限和下限。当上、下限相等时，我们就可以用**Θ表示法。**

定义3.3 如果非负函数T(n)既在O(h(n))中，又在Ω(h(n))中，则称T(n)是Θ(h(n))。这时也说T(n)与h(n)同阶。

例：在平均情况下，顺序检索法既在O(n)中，又在Ω(n)中，所以说平均情况下它是Θ(n)。

Θ表示法

定义3.5 渐近分析化简四法则

1、若f(n)在O(g(n))中，且g(n)在O(h(n))中，则f(n)在O(h(n))中;（传递性）

2、若f(n)在O(kg(n))中，对于任意常数k>0成立，则f(n)在O(g(n))中;（常数系数可忽略）

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;（取大值）

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。（函数相乘则复杂度相乘）

3.5 程序运行时间的计算

- 例 一个给整型变量赋值的简单语句:

```
a = b;
```

该语句执行时间为一常量，为Θ(1)。

- 例 一个简单的for循环:

```
sum = 0;
for (i=1; i<=n; i++)
sum += n;
```

第一个语句的时间代价为Θ(1)。第二行的for循环重复执行了n次，第三行的语句时间代价为一常量，根据简化法则4，后两行的for循环总时间代价为Θ(n)。根据法则3，整个程序段的代价也是Θ(n)。

程序运行时间的计算

- 例 一个含有多个for循环的程序段, 其中有些是有嵌套的。

```
sum = 0;
for (i=1; i<=n; i++)
   for (j=1; j<=i; j++)  sum++;
for (k=0; k<n; k++)  a[k]=k;
```

该程序段有三个相对独立的片断: 一个赋值语句和两个for循环结构。赋值语句的时间代价为常量，记作c_1。第二个for循环，时间代价为c_2n。

程序运行时间的计算

第一个for循环是一个双重循环，从内层循环入手:

运行sum++;需要时间为一常量，记作c_3，内层循环执行i次，根据法则4，时间开销为$c_3 i$ 。外层循环的控制变量i从1递增到n。因此，总的时间开销是从1累加到n再乘以c_3，可以得出:

$$c_3 \sum_{i=1}^{n} i = \frac{n(n+1)}{2} c_3$$

即$\Theta(n^2)$，根据法则3，总运行时间为$\Theta(c_1+c_2 n+c_3 n^2)$, 可简化为$\Theta(n^2)$。

相同时间代价等级的程序，实际运行时间也可能有差异

例 比较下面两段程序的算法分析：

```
sum1 = 0;
for(i=1;i<=n;i++)  for(j=1;j<=n;j++) sum1++;
sum2 = 0;
for (i=1; i<=n; i++) for (j=1; j<=i; j++)  sum2++;
```

在第一个双重循环中，内层for循环总是执行n次。因为外层循环执行n次，所以sum++;语句执行n^2次。而第二个循环与上题的例子相仿，时间代价近似为$n^2/2$。因此，两个二重循环的时间代价都是$\Theta(n^2)$，不过第二个程序段的运行时间约为第一个的一半。

二分法检索

适用于已排序顺序线性表或二叉查找树

- 设数组中间位置为mid，相应的元素值记为kmid。
  - 如果kmid =k ，那么检索工作就完成了。
  - 当kmid >k时，检索继续在前半部分进行。
  - 相反地，若kmid < k，检索继续在后半部分进行。
  - Kmid ≠k的两种情况，都缩小了一半的检索范围。

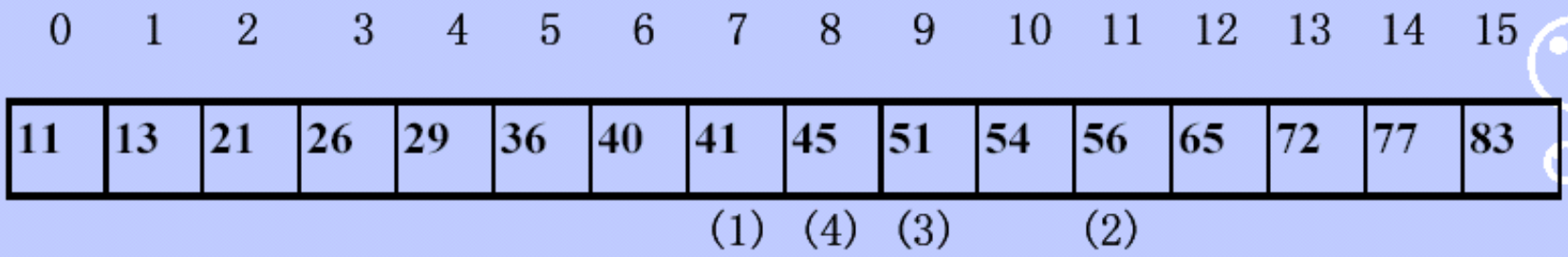
二分法检索

- 二分法的下一步工作是检查k可能存在的那部分元素中的中间位置。该位置上的元素值使我们又能缩小一半的检索范围。
- 重复这个过程，就能找到指定的元素(或确定它不在数组中)。
- 该过程至多需要重复 $\lceil \log_2(n+1) \rceil$ 次。

二分法检索

```
int binary(int K, ELEM* array, int left, int right) {
    // 返回值为K的元素位置
    // l and r are beyond the bounds of array
   int l = left-1;
    int r = right+1;
   while (l+1 != r) {   // Stop when l and r meet
      int mid = (l+r)/2;   // Look at middle
      if (K < array[mid].key) r = mid;     // In left half
      if (K == array[mid].key) return mid; // Found it
      if (K > array[mid].key) l = mid;     // In right half
    }
    return UNSUCCESSFUL; // value K not in array
   }
```

二分法检索



检索关键码45。l=0; r=15; K=45

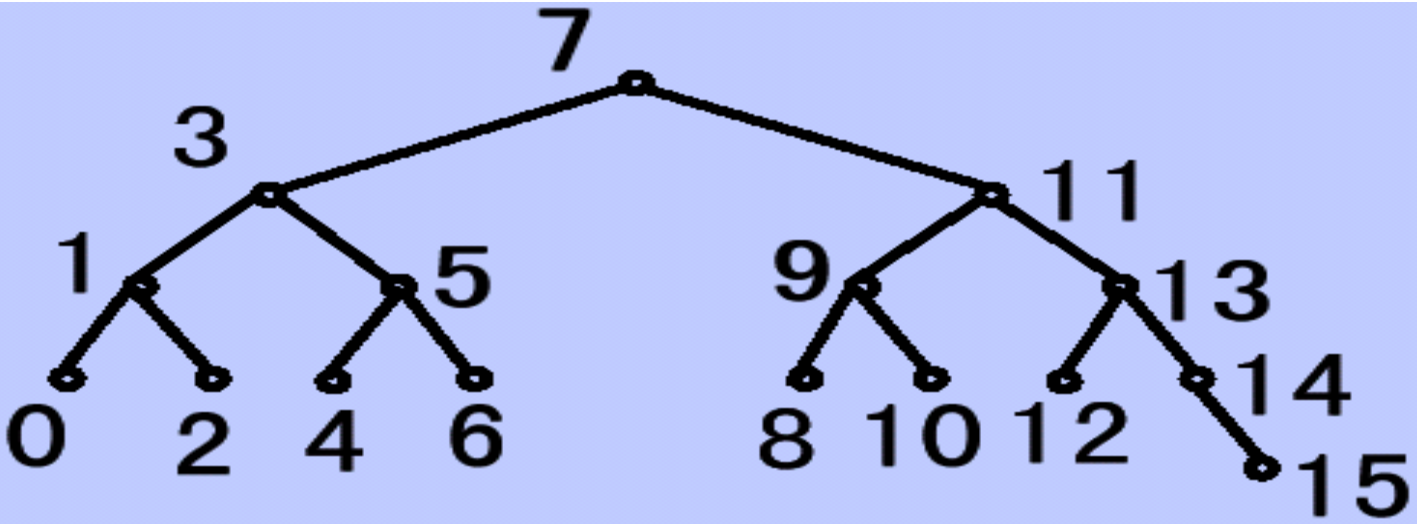
第一次：mid=7; array[7]=41<45
l=7; (r=15)

第二次：mid=11; array[11]=56>45
r=11; (l=7)

第三次：mid=9; array[9]=51>45
r=9; (l=7)

第四次：mid=8; array[8]=45==45; return 8

二分法检索——二叉查找树



1) 最大检索长度为 $\lceil \log_2(n+1) \rceil$

2) 失败的检索长度是 $\lceil \log_2(n+1) \rceil$

$\lfloor \log_2(n+1) \rfloor$

二分法检索

3） 成功的平均检索长度为：

$$E(n)=\frac{1}{n}\left(\sum_{i=1}^{j} i\cdot 2^{i-1}\right)$$

$$=\frac{n+1}{n}\log_2(n+1)-1$$

$$\approx \log_2(n+1)-1 \qquad (n > 50)$$

4） 优缺点

- 优点：平均检索长度与最大检索长度相近，检索速度快
- 缺点：要排序、顺序存储，不易更新(插/删)