Algorithm Complexity

Zibin Zheng（郑子彬）

School of Data and Computer Science， SYSU

http://www.inpluslab.com

课程主页：http://inpluslab.sysu.edu.cn/dsa2016/

Analyzing algorithms

- 如何度量算法的效率？

- 事后统计方法
  - 通过设计好的测试程序和数据，利用计时器对程序的运行时间进行比较，从而确定算法效率的高低。

- 事前分析估算方法
  - 在计算机程序编制前，依据统计方法对算法进行估算

Analyzing algorithms

- 一个程序在计算机上运行时消耗的时间取决于：
  - 依据的算法选用何种策略
  - 问题的规模：例如求100以内还是1000以内的素数
  - 书写程序的语言，语言级别越高，执行效率越低
  - 编译程序产生的机器代码的质量
  - 机器执行指令的速度
- 同一算法用不同语言实现，用不同编译器，或是在不同计算机上运行，效率均不同
- 使用绝对时间衡量算法效率不合适
- **基本操作重复执行的次数作为算法的时间度量**

A case

- 求和1+2+…+100?

第一种算法:

```
int i, sum = 0,n = 100;          /* 执行 1 次 */
for(i = 1; i <= n; i++)          /* 执行了 n+1 次 */
{
    sum = sum + i;               /* 执行 n 次 */
}
printf("%d", sum);               /* 执行 1 次 */
```

1+(n+1)+n+1=2n+3次

第二种算法:

```
int sum = 0,n = 100;             /* 执行一次 */
sum = (1 + n) * n/2;             /* 执行一次 */
printf("%d", sum);               /* 执行一次 */
```

1+1+1=3次

A case

- 再延伸一下这个例子：

```
int i, j, x = 0,sum = 0,n = 100;    /* 执行一次 */
for(i = 1; i <= n; i++)
{
    for (j = 1; j <= n; j++)
    {
        x++;                        /* 执行n×n次 */
        sum = sum + x;
    }
}
printf("%d", sum);                  /* 执行一次 */
```

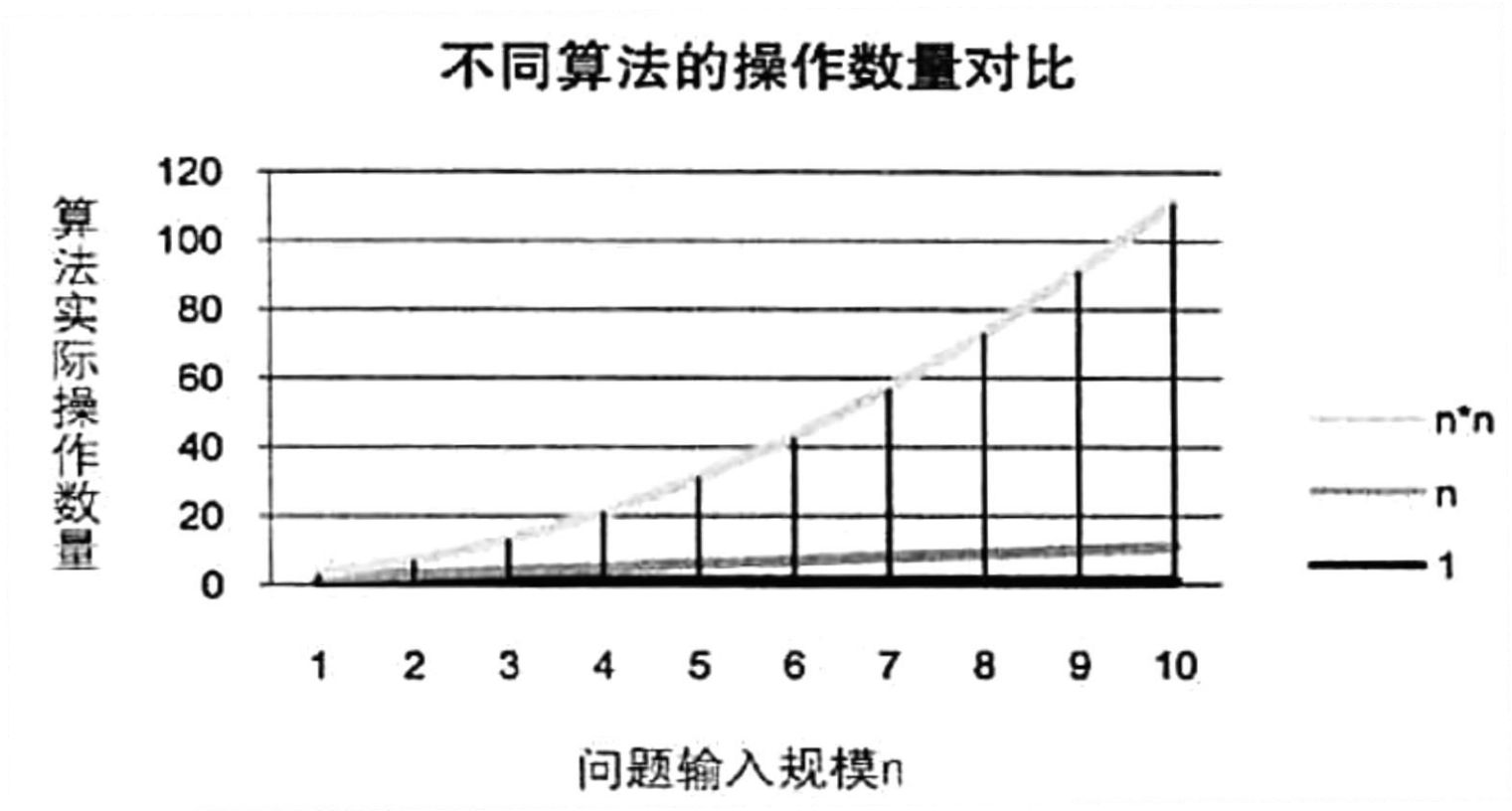
循环部分执行n^2次

算法的执行时间随着n的增加也将远远多于前面2个算法。

A case

- 从上述例子中可以得到启示，同样的输入规模是n
- 第一种方法运行代码n次，则操作数量f(n)=n，显然运行100次是运行10次的10倍时间
- 第二种方法则无论n为多少，运行次数都为1，即f(n)=1
- 第三种方法，由于$f(n)=n^2$，即运算100次是运算10次的100倍。

A case


不同算法的操作数量对比
算法实际操作数量
120
100
80
60
40
20
0
1
2
3
4
5
6
7
8
9
10
问题输入规模n
n*n
n
1

Analyzing algorithms

- 函数的渐进增长
- 算法A做2n+3次操作，算法B做3n+1次操作，谁更快？

次数	算法 A（2n + 3）	算法 A'（2n）	算法 B（3n + 1）	算法 B'（3n）
n = 1	5	2	4	3
n = 2	7	4	7	6
n = 3	9	6	10	9
n = 10	23	20	31	30
n = 100	203	200	301	300

- 在输入规模n没有限制的情况下，只要超过一个数值N，这个函数就总是大于另一个函数，则称函数是渐进增长的。

- 注：忽略加法常数，不影响算法变化。

Analyzing algorithms

- 算法C是4n+8，算法D是$2n^2+1$

次数	算法 C（4n+8）	算法 C'（n）	算法 D（$2n^2+1$）	算法 D'（n^2）
n = 1	12	1	3	1
n = 2	16	2	9	4
n = 3	20	3	19	9
n = 10	48	10	201	100
n = 100	408	100	20 001	10 000
n = 1000	4 008	1 000	2 000 001	1 000 000

- 注：与最高次项相乘的常数并不重要。

Analyzing algorithms

- 算法E是$2n^2+3n+1$，算法F是$2n^3+3n+1$

次数	算法 E（$2n^2+3n+1$）	算法 E'（n^2）	算法 F（$2n^3+3n+1$）	算法 F'（n^3）
n = 1	6	1	6	1
n = 2	15	4	23	8
n = 3	28	9	64	27
n = 10	231	100	2 031	1 000
n = 100	20 301	10 000	2 000 301	1 000 000

- 注：最高次项的指数大的，函数随着n的增长，结果也会变得增长特别快。

Analyzing algorithms

- 算法G是$2n^2$，算法H是$3n+1$，算法I是$2n^2+3n+1$

次数	算法 G（$2n^2$）	算法 H（$3n+1$）	算法 I（$2n^2+3n+1$）
n = 1	2	4	6
n = 2	8	7	15
n = 5	50	16	66
n = 10	200	31	231
n = 100	20 000	301	20 301
n = 1,000	2 000 000	3 001	2 003 001
n = 10,000	200 000 000	30 001	200 030 001
n = 100,000	20 000 000 000	300 001	20 000 300 001
n = 1,000,000	2 000 000 000 000	3 000 001	200 000 3000 001

随着n的值越来越大，H已经远小于G和I，而G和I越来越相近。

- 结论：判断一个算法的效率时，函数中的常数和其他次要项常常可以忽略，而更应该关注主项（最高阶项）的阶数。

算法的时间复杂度

- 定义：
  - 在进行算法分析时，语句总的执行次数T(n)是关于问题规模n的函数，进而分析T(n)随n的变化情况，并确定T(n)的数量级。
  - 算法的时间复杂度，记作：T(n)=O(f(n))。表示随问题规模n的增大，算法执行时间的增长率和f(n)的增长率相同，称作算法的渐近时间复杂度，简称为时间复杂度。
  - 其中f(n)是问题规模n的某个函数。
- 用大写O()来体现算法时间复杂度的记法，称之为大O记法。
- O(1)为常数阶，O(n)为线性阶，$O(n^2)$为平方阶。

算法的时间复杂度

- 常数阶

```
int sum = 0,n = 100;      /* 执行一次 */
sum = (1+n)*n/2;          /* 执行一次 */
printf("%d", sum);        /* 执行一次 */
```

3次

```
int sum = 0, n = 100;     /* 执行1次 */
sum = (1+n)*n/2;          /* 执行第1次 */
sum = (1+n)*n/2;          /* 执行第2次 */
sum = (1+n)*n/2;          /* 执行第3次 */
sum = (1+n)*n/2;          /* 执行第4次 */
sum = (1+n)*n/2;          /* 执行第5次 */
sum = (1+n)*n/2;          /* 执行第6次 */
sum = (1+n)*n/2;          /* 执行第7次 */
sum = (1+n)*n/2;          /* 执行第8次 */
sum = (1+n)*n/2;          /* 执行第9次 */
sum = (1+n)*n/2;          /* 执行第10次 */
printf("%d",sum);         /* 执行1次 */
```

12次

只有执行次数的差异，跟问题规模n的取值无关，所以为O(1)时间复杂度

算法的时间复杂度

- 线性阶

```
int i;
for (i = 0; i < n; i++)
{
    /* 时间复杂度为O(1)的程序步骤序列 */
}
```

循环n次，所以为O(n)时间复杂度

- 对数阶

```
int count = 1;
while (count < n)
{
    count = count * 2;
    /* 时间复杂度为O(1)的程序步骤序列 */
}
```

由$2^x=n$，得$x=\log_2 n$，所以为O(logn)时间复杂度

算法的时间复杂度

- 平方阶

```
int i,j;
for(i = 0; i < n; i++)
{
    for (j = 0; j < n; j++)
    {
        /* 时间复杂度为O(1)的程序步骤序列 */
    }
}
```

内循环n次，再进行外循环n次，所以为$O(n^2)$时间复杂度

```
int i,j;
for(i = 0; i < m; i++)
{
    for (j = 0; j < n; j++)
    {
        /* 时间复杂度为O(1)的程序步骤序列 */
    }
}
```

内循环n次，外循环m次，所以为O(mn)时间复杂度

Complicated cases

- 嵌套循环

```
int i,j;
for(i = 0; i < n; i++)
{
    for (j = i; j < n; j++)   /* 注意j = i而不是0 */
    {
        /* 时间复杂度为O(1)的程序步骤序列 */
    }
}
```

i=0时，内循环执行n次
i=1时，内循环执行n-1次
.....
i=n-1时，内循环执行1次

- 总的执行次数为： $n+(n-1)+(n-2)+\cdots+1=\frac{n(n+1)}{2}=\frac{n^2}{2}+\frac{n}{2}$

- 最终时间复杂度为$O(n^2)$

Complicated cases

```
void function(int count)
{
    int j;
    for (j = count; j < n; j++)
    {
        /* 时间复杂度为O(1)的程序步骤序列 */
    }
}
```

```
n++;                              /* 执行次数为 1 */
function(n);                      /* 执行次数为 n */
int i,j;
for(i = 0; i < n; i++)            /* 执行次数为 n^2 */
{
    function (i);
}
for(i = 0; i < n; i++)            /* 执行次数为 n(n + 1)/2 */
{
    for (j = i;j < n; j++)
    {
        /* 时间复杂度为O(1)的程序步骤序列 */
    }
}
```

执行次数：

$$f(n)=1+n+n^2+\frac{n(n+1)}{2}=\frac{3}{2}n^2+\frac{3}{2}n+1$$

时间复杂度：$O(n^2)$

算法的时间复杂度

- 常见的时间复杂度：

执行次数函数	阶	非正式术语
12	O(1)	常数阶
2n+3	O(*n*)	线性阶
$3n^2+2n+1$	$O(n^2)$	平方阶
$5\log_2 n+20$	O(log*n*)	对数阶
$2n+3n\log_2 n+19$	O(*n*log*n*)	nlogn 阶
$6n^3+2n^2+3n+4$	$O(n^3)$	立方阶
2^n	$O(2^n)$	指数阶

- 所耗费的时间从小到大排列：

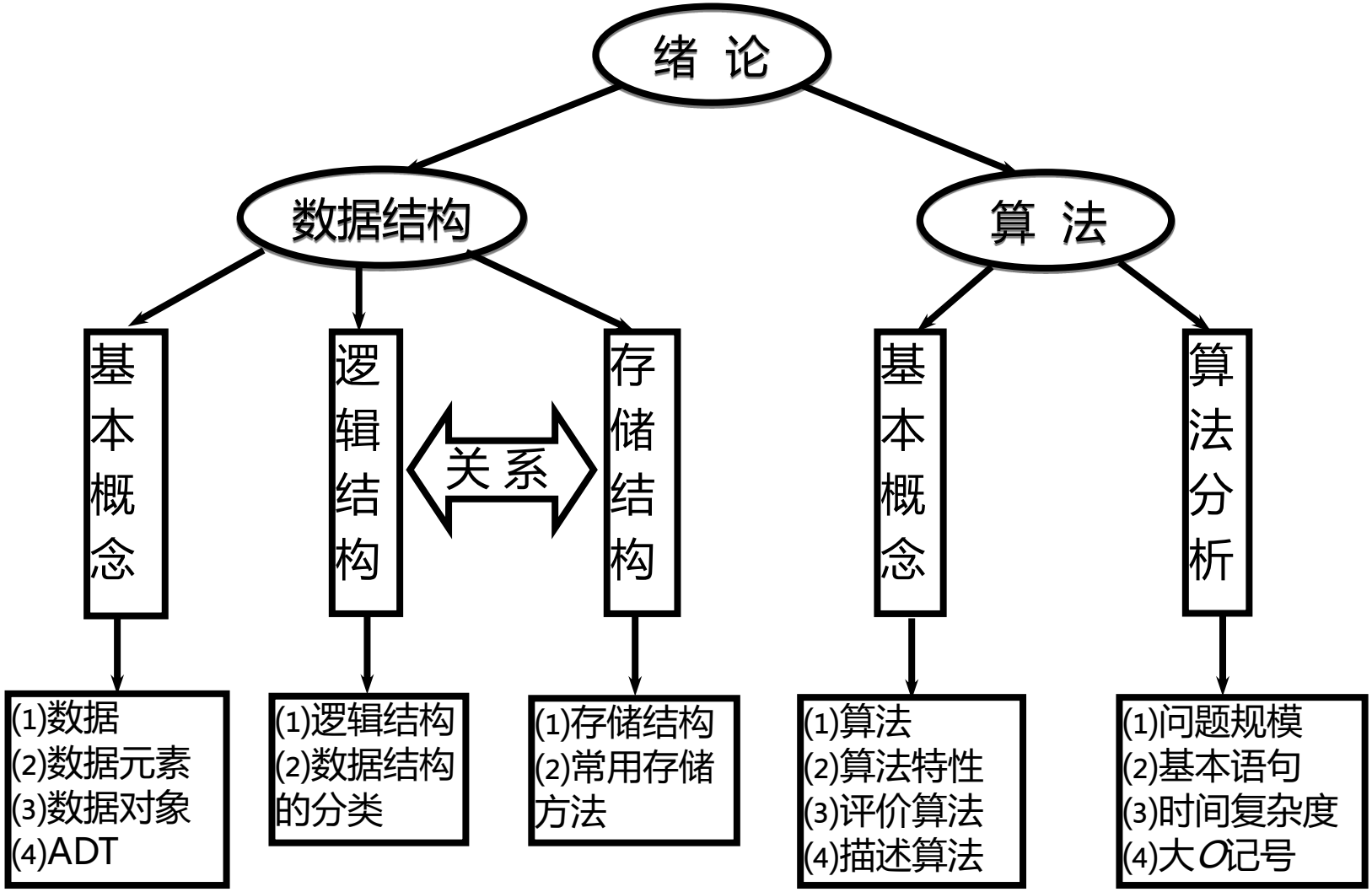
$$O(1) < O(\log n) < O(n) < O(n\log n) < O(n^2) < O(n^3) < O(2^n) < O(n!) < O(n^n)$$

算法的空间复杂度

- 算法的空间复杂度通过计算算法所需的存储空间实现，记作S(n)=O(f(n))，其中n为问题的规模，f(n)为语句关于n所占存储空间的函数。

- 若算法执行时所需的辅助空间相对于输入数据量而言是个常数，则称此算法的空间复杂度为O(1)。

本章小结——知识结构图


绪 论
数据结构
算 法
基本概念
逻辑结构
关 系
存储结构
基本概念
算法分析
(1)数据
(2)数据元素
(3)数据对象
(4)ADT
(1)逻辑结构
(2)数据结构的分类
(1)存储结构
(2)常用存储方法
(1)算法
(2)算法特性
(3)评价算法
(4)描述算法
(1)问题规模
(2)基本语句
(3)时间复杂度
(4)大O记号

谢谢！