

本节内容

算法

效率的度量

知识总览

算法效率的度量

时间复杂度

空间复杂度

如何评估算法时间开销?



让算法先运行，事后统计运行时间？



存在什么问题？

- 和机器性能有关，如：超级计算机 v.s. 单片机
- 和编程语言有关，越高级的语言执行效率越低
- 和编译程序产生的机器指令质量有关
- 有些算法是不能事后再统计的，如：导弹控制算法

能否排除与算法本身无关的外界因素

能否事先估计？



算法时间复杂度

事前预估算法时间开销 $T(n)$ 与问题规模 n 的关系（ T 表示“time”）

算法的时间复杂度



用算法表白——“爱你n遍”

算法的时间复杂度

//算法1— 逐步递增型爱你

void loveYou(int n) { //n 为问题规模

① int i=1; //爱你的程度

② while(i<=n){

③ i++; //每次+1

④ printf("I Love You %d\n", i);

}

⑤ printf("I Love You More Than %d\n", n);

}

```
int main(){
    loveYou(3000);
}
```

I Love You 2994
I Love You 2995
I Love You 2996
I Love You 2997
I Love You 2998
I Love You 2999
I Love You 3000
I Love You 3001
I Love You More Than 3000

语句频度:

① ——1次

② ——3001次

③④ ——3000次

⑤ ——1次

$T(3000) = 1 + 3001 + 2 * 3000 + 1$

时间开销与问题规模 n 的关系:

$T(n) = 3n + 3$

思考中.....



问题1: 是否可以忽略表达式某些部分?

问题2: 如果有好几千行代码, 按这种方法需要一行一行数?

算法的时间复杂度

全称：渐进时间复杂度

思考中.....



问题1：是否可以忽略表达式某些部分？

当问题规模 n 足够大时...

大O表示“同阶”，同等数量级。即：当 $n \rightarrow \infty$ 时，二者之比为常数

$$\begin{aligned}T_1(n) &= O(n) \\T_2(n) &= O(n^2) \\T_3(n) &= O(n^3)\end{aligned}$$

简化

时间开销与问题规模 n 的关系：

$$T_1(n) = 3n + 3$$

$$T_2(n) = n^2 + 3n + 1000$$

$$T_3(n) = n^3 + n^2 + 99999999$$

若 $n=3000$ ，则

$$3n = 9000$$

$$n^2 = 9,000,000$$

$$n^3 = 27,000,000,000$$

V.S.

$$T_1(n) = 9003$$

V.S.

$$T_2(n) = 9,010,000$$

V.S.

$$T_3(n) = 27,018,999,999$$

结论1：可以只考虑阶数高的部分

当 $n=3000$ 时， $9999n = 29,997,000$ 远小于 $n^3 = 27,018,999,999$
当 $n=1000000$ 时， $9999n = 9,999,000,000$ 远小于 $n^2 = 1,000,000,000,000$

结论2：问题规模足够大时，常数项系数也可以忽略

算法的时间复杂度

思考中.....



问题1: 是否可以忽略表达式某些部分?

当问题规模 n 足够大时...

时间开销与问题规模 n 的关系:

$$T_1(n) = 3n + 3$$

$$T_2(n) = n^2 + 3n + 1000$$

$$T_3(n) = n^3 + n^2 + 99999999$$

$$T_1(n) = O(n)$$

$$T_2(n) = O(n^2)$$

$$T_3(n) = O(n^3)$$

简化

a) 加法规则

$$T(n) = T_1(n) + T_2(n) = O(f(n)) + O(g(n)) = O(\max(f(n), g(n)))$$

多项相加, 只保留最高阶的项, 且系数变为1

b) 乘法规则

$$T(n) = T_1(n) \times T_2(n) = O(f(n)) \times O(g(n)) = O(f(n) \times g(n))$$

多项相乘, 都保留

$$\begin{aligned} \text{Eg: } T_3(n) &= n^3 + n^2 \log_2 n \\ &= O(n^3) + O(n^2 \log_2 n) \\ &= ? ? ? \end{aligned}$$

算法的时间复杂度

$$O(1) < O(\log_2 n) < O(n) < O(n \log_2 n) < O(n^2) < O(n^3) < O(2^n) < O(n!) < O(n^n)$$



问题：两个算法的时间复杂度分别如下，哪个的阶数更高（时间复杂度更高）？

$$T_1(n) = O(n)$$

$$T_2(n) = O(\log_2 n)$$

$$\begin{aligned} \text{Eg: } T_3(n) &= n^3 + n^2 \log_2 n \\ &= O(n^3) + O(n^2 \log_2 n) \\ &= O(n^2 n) + O(n^2 \log_2 n) \end{aligned}$$

$$\lim_{n \rightarrow \infty} \frac{\log_2 n}{n} \xrightarrow{\text{洛必达}} \lim_{n \rightarrow \infty} \frac{1}{n \ln 2} = 0$$

当 $n \rightarrow \infty$ 时， n 比 $\log_2 n$ 变大的速度快很多

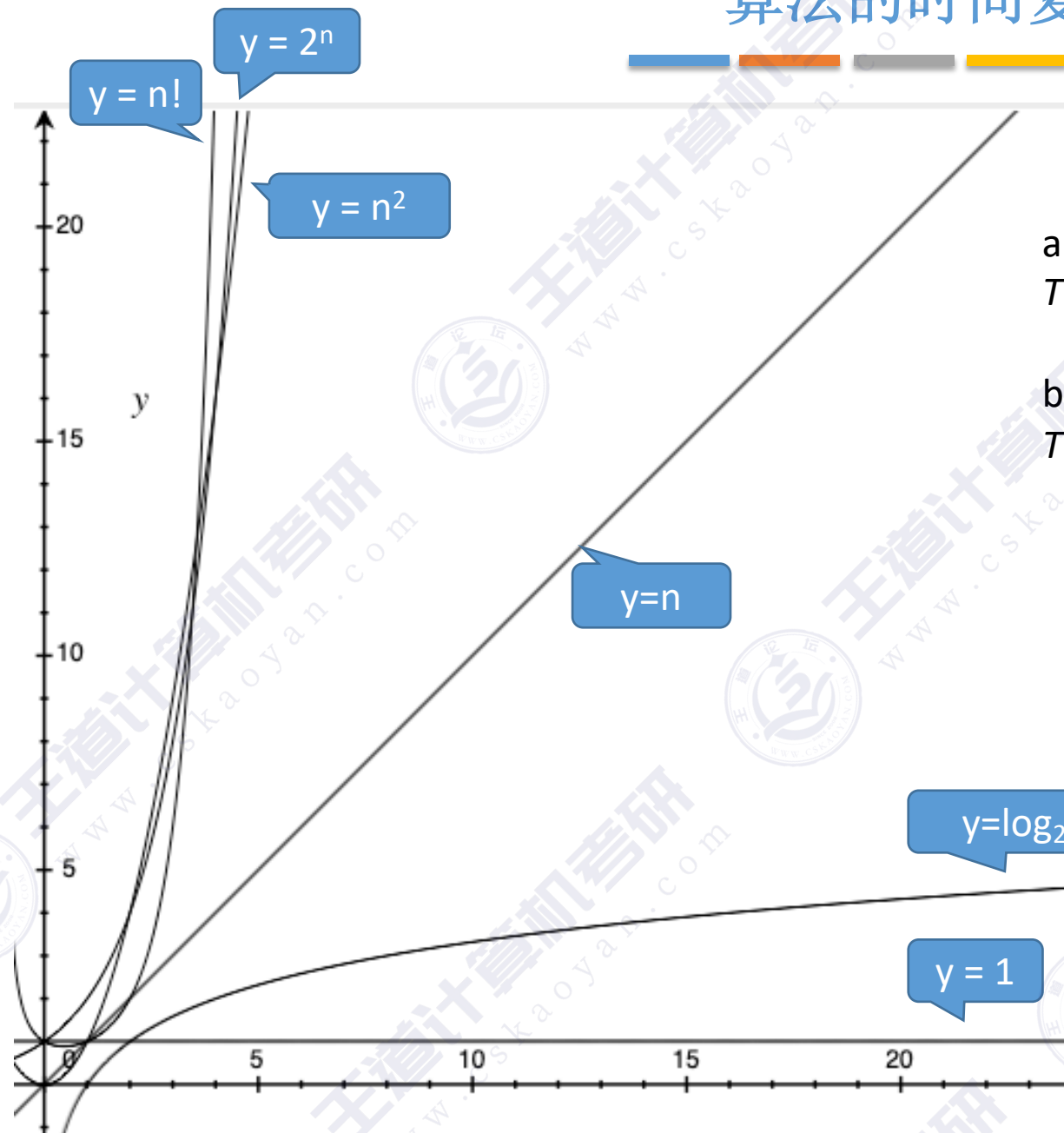
$$\lim_{n \rightarrow \infty} \frac{n^2}{2^n} \xrightarrow{\text{洛必达}} \lim_{n \rightarrow \infty} \frac{2n}{2^n \ln 2} \xrightarrow{\text{洛必达}} \lim_{n \rightarrow \infty} \frac{2}{2^n \ln^2 2} = 0$$

当 $n \rightarrow \infty$ 时， 2^n 比 n^2 变大的速度快很多

别紧张 放轻松 😊



算法的时间复杂度



只保留更高阶的项

a) 加法规则

$$T(n) = T_1(n) + T_2(n) = O(f(n)) + O(g(n)) = O(\max(f(n), g(n)))$$

b) 乘法规则

$$T(n) = T_1(n) \times T_2(n) = O(f(n)) \times O(g(n)) = O(f(n) \times g(n))$$



巴拉拉能量——

常对幂指阶

数据结构 +20分

算法的时间复杂度

//算法1— 逐步递增型爱你

void loveYou(int n) { //n 为问题规模

① int i=1; //爱你的程度

② while(i<=n){

③ i++; //每次+1

④ printf("I Love You %d\n", i);

}

⑤ printf("I Love You More Than %d\n", n);

}

```
int main(){
    loveYou(3000);
}
```

I Love You 2994
I Love You 2995
I Love You 2996
I Love You 2997
I Love You 2998
I Love You 2999
I Love You 3000
I Love You 3001
I Love You More Than 3000

语句频度:

① ——1次

② ——3001次

③④ ——3000次

⑤ ——1次

$T(3000) = 1 + 3001 + 2 * 3000 + 1$

时间开销与问题规模 n 的关系:

$T(n) = 3n + 3 = O(n)$

思考中.....



问题1: 是否可以忽略表达式某些部分?

只考虑阶数, 用大O记法表示

问题2: 如果有好几千行代码, 按这种方法需要一行一行数?

算法的时间复杂度

//算法1— 逐步递增型爱你

void loveYou(int n) { //n 为问题规模

```
① int i=1;    //爱你的程度
② while(i<=n){
③     i++;    //每次+1
④     printf("I Love You %d\n", i);
}
⑤ printf("I Love You More Than %d\n", n);
}
```

此处插入1000行顺序执行的代码

语句频度:

- ① ——1次
- ② ——3001次
- ③④ ——3000次
- ⑤ ——1次

$T(3000) = 1 + 3001 + 2 * 3000 + 1$
时间开销与问题规模 n 的关系:

$$T(n) = 3n + 3 = O(n)$$

结论1: 顺序执行的代码只会影响常数项, 可以忽略

结论2: 只需挑循环中的一个基本操作分析它的执行次数与 n 的关系即可



机智如我

$T(3000) = 1000 + 1 + 3001 + 2 * 3000 + 1$
时间开销与问题规模 n 的关系:

$$T(n) = 3n + 1003 = O(n)$$

算法的时间复杂度

//算法2— 嵌套循环型爱你

```
void loveYou(int n) { //n 为问题规模
    int i=1; //爱你的程度
    while(i<=n){
        i++; //每次+1
        printf("I Love You %d\n", i);
        for (int j=1; j<=n; j++){
            printf("I am Iron Man\n");
        }
    }
    printf("I Love You More Than %d\n", n);
}
```

外层循环执行n次

嵌套两层循环

内层循环共执行 n^2 次

时间开销与问题规模 n 的关系:

$$T(n) = O(n) + O(n^2) = O(n^2)$$

结论1: 顺序执行的代码只会影响常数项, 可以忽略

结论2: 只需挑循环中的一个基本操作分析它的执行次数与 n 的关系即可

结论3: 如果有多层嵌套循环, 只需关注最深层循环循环了几次



机智如我

算法的时间复杂度

//算法1— 逐步递增型爱你

void loveYou(int n) { //n 为问题规模

① int i=1; //爱你的程度

② while(i<=n){

③ i++; //每次+1

④ printf("I Love You %d\n", i);

}

⑤ printf("I Love You More Than %d\n", n);

}

```
int main(){
    loveYou(3000);
}
```

I Love You 2994

I Love You 2995

I Love You 2996

I Love You 2997

I Love You 2998

I Love You 2999

I Love You 3000

I Love You 3001

I Love You More Than 3000

语句频度:

① ——1次

② ——3001次

③④ ——3000次

⑤ ——1次

$T(3000) = 1 + 3001 + 2 * 3000 + 1$

时间开销与问题规模 n 的关系:

$T(n) = 3n + 3 = O(n)$

思考中.....



问题1: 是否可以忽略表达式某些部分?

只考虑阶数, 用大O记法表示

问题2: 如果有好几千行代码, 按这种方法需要一行一行数?

只需考虑最深层循环的循环次数与 n 的关系

小练习1

//算法3— 指数递增型爱你

```
void loveYou(int n) { //n 为问题规模
    int i=1; //爱你的程度
    while(i<=n){
        i=i*2; //每次翻倍
        printf("I Love You %d\n", i);
    }
    printf("I Love You More Than %d\n", n);
}
```

```
I Love You 32
I Love You 64
I Love You 128
I Love You 256
I Love You 512
I Love You 1024
I Love You 2048
I Love You 4096
I Love You More Than 3000
```

计算上述算法的时间复杂度 $T(n)$:

设最深层循环的语句频度（总共循环的次数）为 x ，则由循环条件可知，循环结束时刚好满足 $2^x > n$

$$x = \log_2 n + 1$$

$$T(n) = O(x) = O(\log_2 n)$$

小练习2

//算法4— 搜索数字型爱你

```
void loveYou(int flag[], int n) { //n 为问题规模
    printf("I Am Iron Man\n");
    for(int i=0; i<n; i++){ //从第一个元素开始查找
        if(flag[i]==n){ //找到元素n
            printf("I Love You %d\n", n);
            break; //找到后立即跳出循环
        }
    }
}
```

//flag 数组中乱序存放了 1~n 这些数

```
int flag[n]={1...n};
loveYou(flag, n);
```

计算上述算法的时间复杂度 $T(n)$

最好情况：元素n在第一个位置

最坏情况：元素n在最后一个位置

平均情况：假设元素n在任意一个位置的概率相同为 $\frac{1}{n}$

很多算法执行时间与输入的数据有关

——最好时间复杂度 $T(n)=O(1)$

——最坏时间复杂度 $T(n)=O(n)$

——平均时间复杂度 $T(n)=O(n)$

$$\text{循环次数 } x = (1+2+3+\dots+n) \frac{1}{n} = \left(\frac{n(1+n)}{2}\right) \frac{1}{n} = \frac{1+n}{2}$$

$$T(n)=O(x)=O(n)$$

算法的时间复杂度



最坏时间复杂度：最坏情况下算法的时间复杂度

平均时间复杂度：所有输入示例等概率出现的情况下，算法的期望运行时间

最好时间复杂度：最好情况下算法的时间复杂度

知识回顾与重要考点

时间复杂度

如何计算

- ① 找到一个基本操作（最深层循环）
- ② 分析该基本操作的执行次数 x 与问题规模 n 的关系 $x=f(n)$
- ③ x 的数量级 $O(x)$ 就是算法时间复杂度 $T(n)$

加法规则: $O(f(n)) + O(g(n)) = O(\max(f(n), g(n)))$

常用技巧

乘法规则: $O(f(n)) \times O(g(n)) = O(f(n) \times g(n))$

“常对幂指阶”

$O(1) < O(\log_2 n) < O(n) < O(n \log_2 n) < O(n^2) < O(n^3) < O(2^n) < O(n!) < O(n^n)$

三种复杂度

最坏时间复杂度: 考虑输入数据“最坏”的情况

平均时间复杂度: 考虑所有输入数据都等概率出现的情况

最好时间复杂度: 考虑输入数据“最好”的情况

小故事: 算法的性能问题只有在 n 很大时才会暴露出来。