本文内容是对时间复杂度的梳理和总结，本文内容包括：

1 评价排序算法

武松和鲁智深都写了一个排序算法，如何评价他们俩谁写的算法好呢？

评价一个算法通常从两个角度考虑

- 执行速度
- 占用存储空间

如果执行速度块，说明算法好。这种评价的角度称为**时间复杂度**

如果执行过程中占用存储空间少，说明算法好。这种评价的角度称为**空间复杂度**

而实际上对于用户来说时间复杂度更重要，因为时间复杂度是用户能感受到的，而空间复杂度是用户感受不到的。为了让用户感受速度，还可以用空间换时间，例如缓存（redis，memcache）技术就是用空间换时间。

2 时间频度

要学习时间复杂度，要先知道什么是时间频度。

2.1 什么是时间频度

时间频度：一个算法花费的时间与算法中语句的执行次数成正比例，哪个算法中语句执行次数多，它花费时间就多。**一个算法中的语句执行次数称为语句频度或时间频度。**记为T(n)。

例如：计算1-100所有数字之和

第一种算法：

```
    int sum = 0;
    int end = 100;
    for (int i = 0; i <= end; i++) {
        sum += i;
    }
```

时间频度为：T(n+1)，说明随着end值的增加，时间频度也会增加。

第二种算法：

```
    int total = 0;
    int end = 100;
    total = (1+end )*end /2;
```

时间频度为：T(1)，说明随着end值的增加，时间频度不会增加。

2.1.1 忽略常数项

下表是一个时间频度的函数表

n的值	T(n)=2n+20	T(n)=2*n	T(n)=T(3n+10)	T(n)=T(3n)
1	22	2	13	3
2	24	4	16	6
5	30	10	25	15
8	36	16	34	24

n的值	T(n)=2n+20	T(n)=2*n	T(n)=T(3n+10)	T(n)=T(3n)
15	50	30	55	45
30	80	60	100	90
100	220	200	310	300
300	620	600	910	900

观察上表，发现随着n的增大，时间频度也会增大。

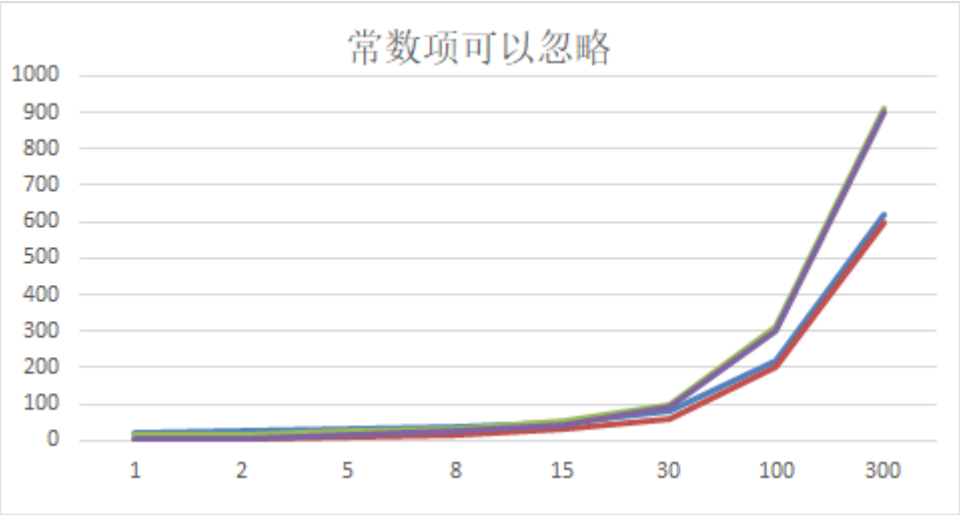
例如：

- 当n为300时，2n+20的值是620
- 当n为300时，2*n的值是600

再例如：

- 当n为300时，T(3n+10) 的值是910
- 当n为300时，T(3n) 的值是900

随着n的增大，时间频度的变化如下图所示：



我们发现：

- 2n+20 和 2n 随着n 变大，执行曲线无限接近, 20可以忽略
- 3n+10 和 3n 随着n 变大，执行曲线无限接近, 10可以忽略

结论：时间频度的**常数项可以忽略**

2.1.2 忽略低次项

下表是一个时间频度的函数表

n的值	T(n) = $2n^2+3n+10$	T(n) = $T(2n^2)$	T(n) = $T(n^2+5n+20)$	T(n) = $T(n^2)$
1	15	2	26	1
2	24	8	34	4
5	75	50	70	25
8	162	128	124	64
15	505	450	320	225
30	1900	1800	1070	900
100	20310	20000	10520	10000

观察上表，发现随着n的增大，时间频度也会增大。

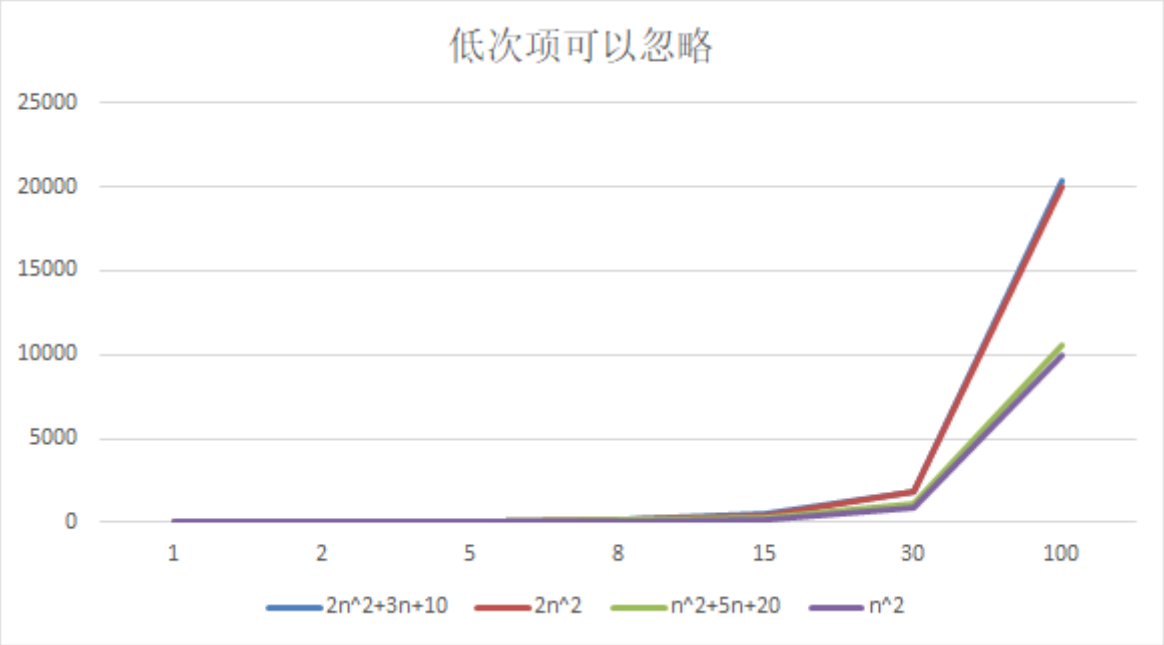
例如：

- 当n为100时，$2n^2+3n+10$的值是20310
- 当n为100时，$T(2n^2)$ 的值是20000

再例如：

- 当n为100时，$T(n^2+5n+20)$ 的值是10520
- 当n为100时，$T(n^2)$ 的值是 10000

随着n的增大，时间频度的变化如下图所示：



我们发现：

- 2n^2+3n+10 和 2n^2 随着n 变大, 执行曲线无限接近, 可以忽略 3n+10

- n^2+5n+20 和 n^2 随着n 变大,执行曲线无限接近, 可以忽略 5n+20

结论：时间频度的**低次项可以忽略**

2.1.3 忽略系数

下表是一个时间频度的函数表

n的值	T(3n^2+2n)	T(5n^2+7n)	T(n^3+5n)	T(6n^3+4n)
1	5	12	6	10
2	16	34	18	56
5	85	160	150	770
8	208	376	552	3104
15	705	1230	3450	20310
30	2760	4710	27150	162120
100	30200	50700	1000500	6000400

观察上表，发现随着n的增大，时间频度也会增大。

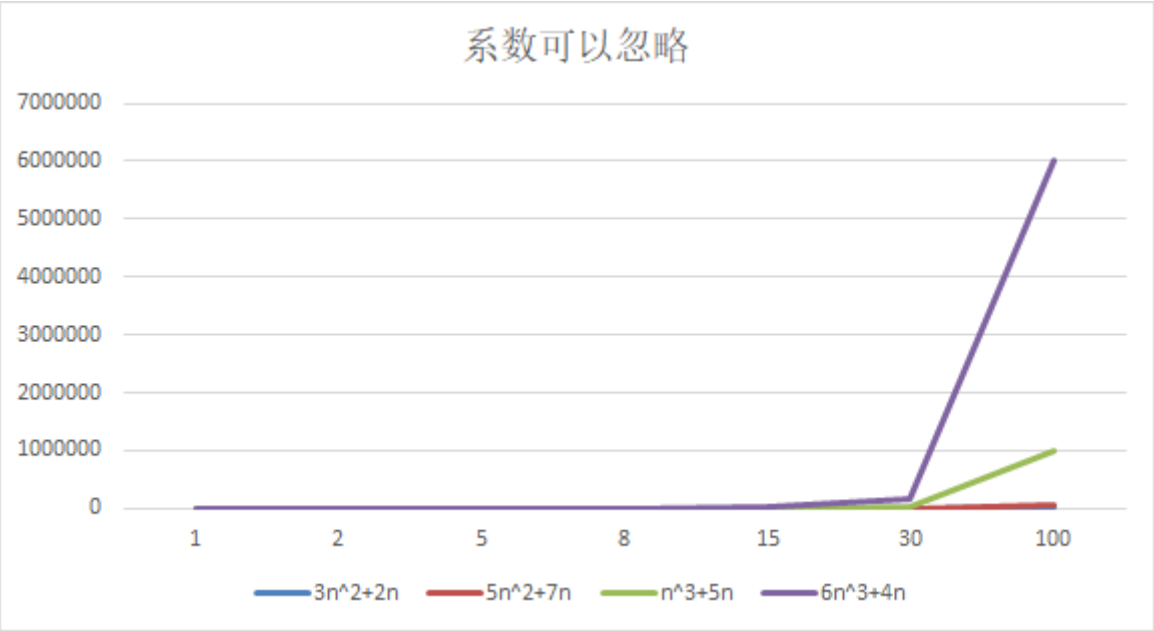
例如：

- 当n为100时，3n^2+2n的值是30200
- 当n为100时，5n^2+7n的值是50700

再例如：

- 当n为100时，n^3+5n 的值是1000500
- 当n为100时，6n^3+4n的值是6000400

随着n的增大，时间频度的变化如下图所示：



我们发现：

- 5n^2+7n 和 3n^2 + 2n ， 随着n值变大，执行曲线重合, 说明这种情况下, 5和3可以忽略。
- n^3+5n 和 6n^3+4n ， 随着n值变大，执行曲线分离，说明多少次方式关键。

结论：时间频度的**系数可以忽略**

2.2 时间复杂度

2.2.1 什么是时间复杂度

一般情况下，算法中的基本操作语句的重复执行次数是问题规模n的某个函数，用T(n)表示，若有某个辅助函数f(n)，使得当n趋近于无穷大时，T(n) / f(n) 的极限值为不等于零的常数，则称f(n)是T(n)的同数量级函数。记作 T(n)=O(f(n))，称O(f(n)) 为算法的渐进时间复杂度，简称时间复杂度。

T(n) 不同，但时间复杂度可能相同。 如：T(n)=n^2+7n+6 与 T(n)=3n^2+2n+2 它们的T(n) 不同，但时间复杂度相同，都为O(n^2)。因为时间频度中的常数项、低次向、系数都可以忽略。

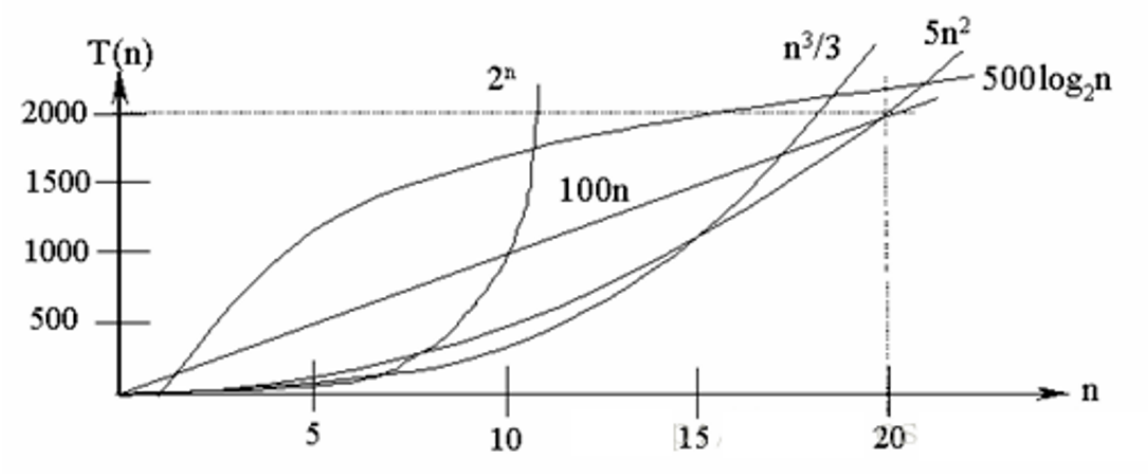
2.2.2 计算时间复杂度的方法

1. 用常数1代替运行时间中的所有加法常数 T(n)=n^2+7n+6 => T(n)=n^2+7n+1
2. 修改后的运行次数函数中，只保留最高阶项 T(n)=n^2+7n+1 => T(n) = n^2
3. 去除最高阶项的系数 T(n) = n^2 => T(n) = n^2 => O(n^2)

2.2.3 常见的时间复杂度

- 常数阶O(1)
- 对数阶O(log2n)

- 线性阶O(n)
- 线性对数阶O(nlog2n)
- 平方阶O(n^2)
- 立方阶O(n^3)
- k次方阶O(n^k)
- 指数阶O(2^n)



说明：

- 常见的算法时间复杂度由小到大依次为：O(1) < O(log2n) < O(n) < O(nlog2n) < $O(n^2)$ < $O(n^3)$ < $O(n^k)$ < $O(2^n)$
- 随着问题规模n的不断增大，上述时间复杂度不断增大，算法的执行效率越低
- 从图中可见，我们应该尽可能避免使用指数阶的算法

2.3 常见的时间复杂度

2.3.1 常数阶O(1)

无论代码执行了多少行，只要是没有循环等复杂结构，那这个代码的时间复杂度就都是O(1)

```
    int i = 1;
    int j = 2;
    i++;
    j++;
    int sum = i+j;
```

上述代码在执行的时候，它消耗的时间并不随着某个变量的增长而增长，那么无论这类代码有多长，即使有几万几十万行，都可以用O(1)来表示它的时间复杂度。

2.3.2 对数阶O($\log_2 n$)

```
    int i = 1;
    while(i<=n) {
        i = i * 2;
    }
```

说明：在while循环里面，每次都将 i 乘以 2，乘完之后，i 距离 n 就越来越近了。假设循环x次之后，i 就大于 2 了，此时这个循环就退出了，也就是说 2 的 x 次方等于 n，那么 x = $\log_2 n$也就是说当循环 $\log_2 n$ 次以后，这个代码就结束了。因此这个代码的时间复杂度为：O($\log_2 n$) 。 O($\log_2 n$) 的这个2 时间上是根据代码变化的，i = i * 3 ，则是 O($\log_3 n$) .

2.3.3 线性阶O(n)

```
    for (int i = 1; i <=n; i++) {

    }
```

说明：这段代码，for循环里面的代码会执行n遍，因此它消耗的时间是随着n的变化而变化的，因此这类代码都可以用O(n)来表示它的时间复杂度。

2.3.4 线性对数阶O($n\log_2 N$)

```
    for (int m = 1; m <=n; m++) {
        int i = 1;
        while(i<n) {
            i = i * 2;
        }
    }
```

说明：线性对数阶O(nlogN) 其实非常容易理解，将时间复杂度为O(logn)的对数阶代码循环N遍的话，那么它的时间复杂度就是 n * O(logN)，也就是了O(nlogN)。

2.3.5 平方阶O(n^2)

```
    for (int i = 1; i <= n; i++) {
        for(int j=1;j <= n;j++) {

        }
    }
```

说明：平方阶$O(n^2)$ 就更容易理解了，如果把 O(n) 的代码再嵌套循环一遍，它的时间复杂度就是 $O(n^2)$，这段代码其实就是嵌套了2层n循环，它的时间复杂度就是 O(n * n)，即 $O(n^2)$ 如果将其中一层循环的n改成m，那它的时间复杂度就变成了 O(m * n)。

2.3.6 立方阶$O(n^3)$

参考上面的$O(n^2)$ 去理解就好了，$O(n^3)$相当于三层n循环，其它的类似。

2.3.7 K次方阶O(n^k)

参考上面的$O(n^2)$ 去理解就好了，$O(n^k)$相当于三层n循环，其它的类似。

2.3.8 平均时间复杂度、最坏时间复杂度

- **平均时间复杂度**：考虑各种情况及其发生的概率，得到的时间复杂度。
- **最好时间复杂度**：在最理想的情况下，执行这段代码的时间复杂度。
- **最坏时间复杂度**：在最糟糕的情况下，执行这段代码的时间复杂度。

一般讨论的时间复杂度均是最坏情况下的时间复杂度。 这样做的原因是：最坏情况下的时间复杂度是算法在任何输入实例上运行时间的界限，这就保证了算法的运行时间不会比最坏情况更长。

各种排序算法的最好、最坏、平均时间复杂度如下表所示：

排序方法	平均时间	最好情况	最坏情况	辅助存储	稳定性
选择排序	$O(n^2)$	$O(n^2)$	$O(n^2)$	O(1)	不稳定
插入排序	$O(n^2)$	O(n)	$O(n^2)$	O(1)	稳定
冒泡排序	$O(n^2)$	$O(n^2)$	$O(n^2)$	O(1)	稳定
希尔排序	$O(n^{1.25})$	--	--	O(1)	不稳定
快速排序	O(nlogn)	O(nlogn)	$O(n^2)$	O(nlogn)	不稳定
堆排序	O(nlogn)	O(nlogn)	O(nlogn)	O(1)	稳定
归并排序	O(nlogn)	O(nlogn)	O(nlogn)	O(n)	稳定
基数排序	O(d (n+rd))	O(d (n+rd))	O(d (n+rd))	O(rd)	稳定

小贴士：排序稳定性

> 稳定排序：是指能保证排序前，两个相等的数其在序列的前后位置顺序和排序后它们两个的前后位置顺序相同。

> 例如，如果Ai = Aj，Ai原来在位置前，排序后Ai还是要在Aj位置前。
>
> 不稳定排序：排序之后在序列中相等的值的相对位置发生变化。

3 排序术语

- **稳定**：如果 A 原本在 B 前面，而 A=B，排序之后 A 仍然在 B 的前面。
- **不稳定**：如果 A 原本在 B 的前面，而 A=B，排序之后 A 可能会出现在 B 的后面。
- **内排序**（In-place）：所有排序操作都在内存中完成。
- **外排序**（Out-place）：由于数据太大，因此把数据放在磁盘中，而排序通过磁盘和内存的数据传输才能进行。
- **时间复杂度**： 定性描述一个算法执行所耗费的时间。
- **空间复杂度**：定性描述一个算法执行所需内存的大小。