算法基础复习

1

算法基础

渐进记号

- Θ 渐进上届与渐进下届相等
- O 小于等于渐进上届
- o 小于
- Ω 大于等于
- ω 大于

什么是时间复杂度

- 时间频度
  - 算法中语句的执行次数
  - 用 T(n) 表示 n为问题规模
- 时间复杂度
  - 如果有一个辅助函数 f(n)
  - 在 n 趋近无穷大的时候，$\frac{T(n)}{f(n)}$ 的极限值等于一个不为零的常量
  - 可以近似地用 f(n) 替代 T(n) 记为 T(n)=O(f(n))
  - 只保留最耗时的部分，舍去常数部分

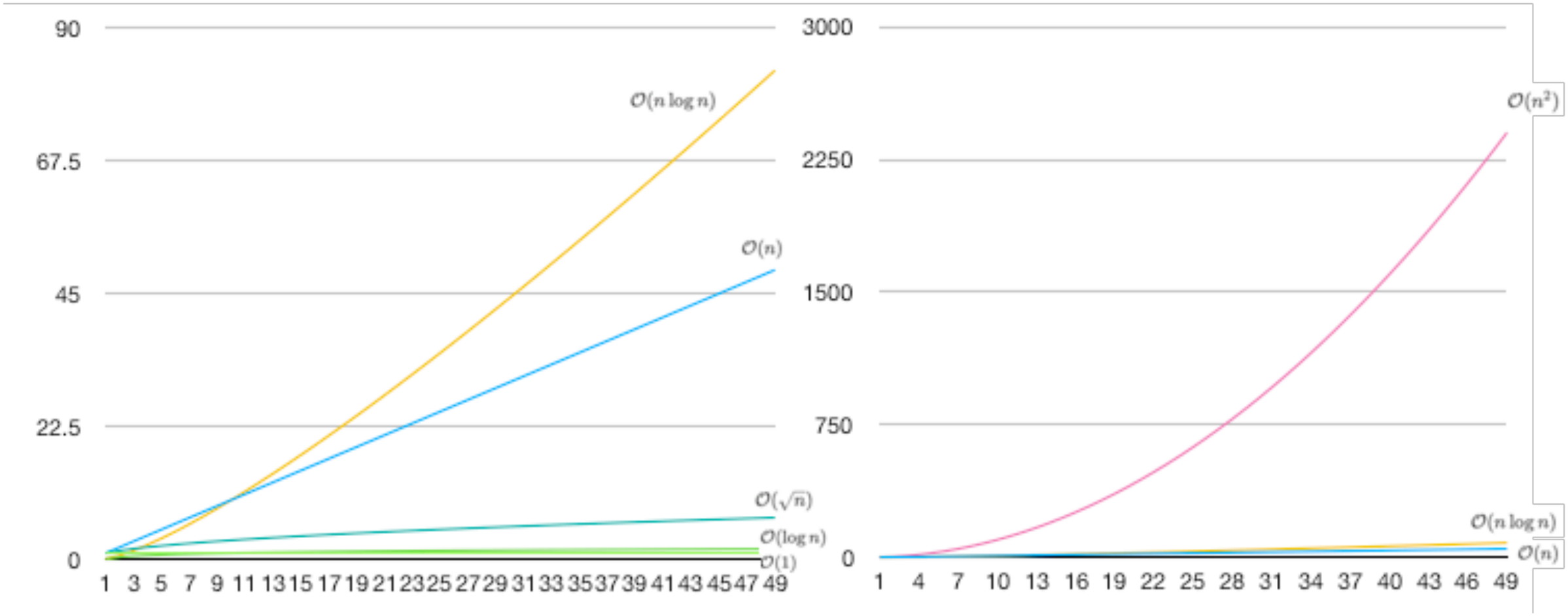
什么是时间复杂度

- 时间频度
- 时间复杂度
  - 只保留最耗时的部分，舍去常数部分
  - 例如：T(n) = $3n^3 + 2n^2 + 4n + 1$
  - 时间复杂度为 O(n^3)
- 如果一个算法需要 f(n) 次操作
  - f(n) = $3n^2 + 2n + 4$
  - 时间复杂度可以是 O(n), Θ(n)和 Ω(n)

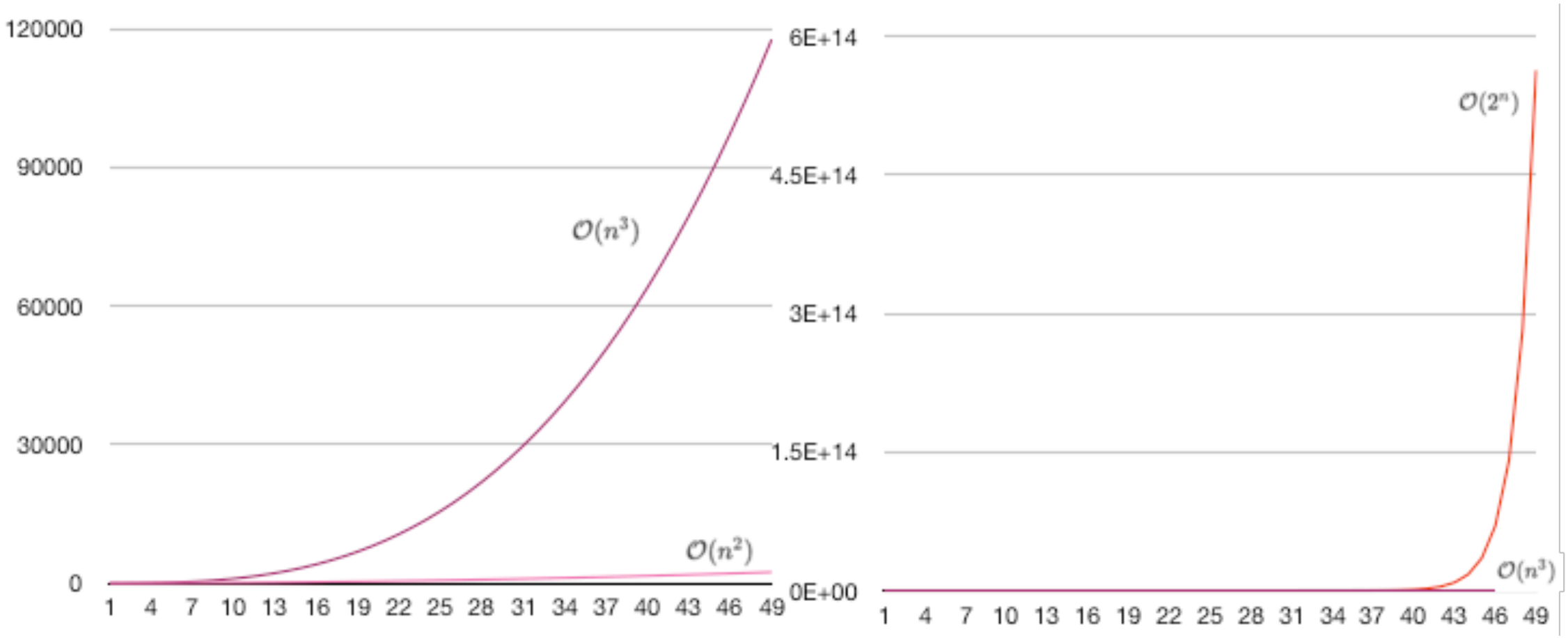
什么是时间复杂度

常数阶	对数阶	平方根阶	线性阶	线性对数阶	平方阶	立方阶	指数阶	阶乘阶
$\mathcal{O}(1)$	$\mathcal{O}(\log n)$	$\mathcal{O}(\sqrt{n})$	$\mathcal{O}(n)$	$\mathcal{O}(n \log n)$	$\mathcal{O}(n^2)$	$\mathcal{O}(n^3)$	$\mathcal{O}(2^n)$	$\mathcal{O}(n!)$

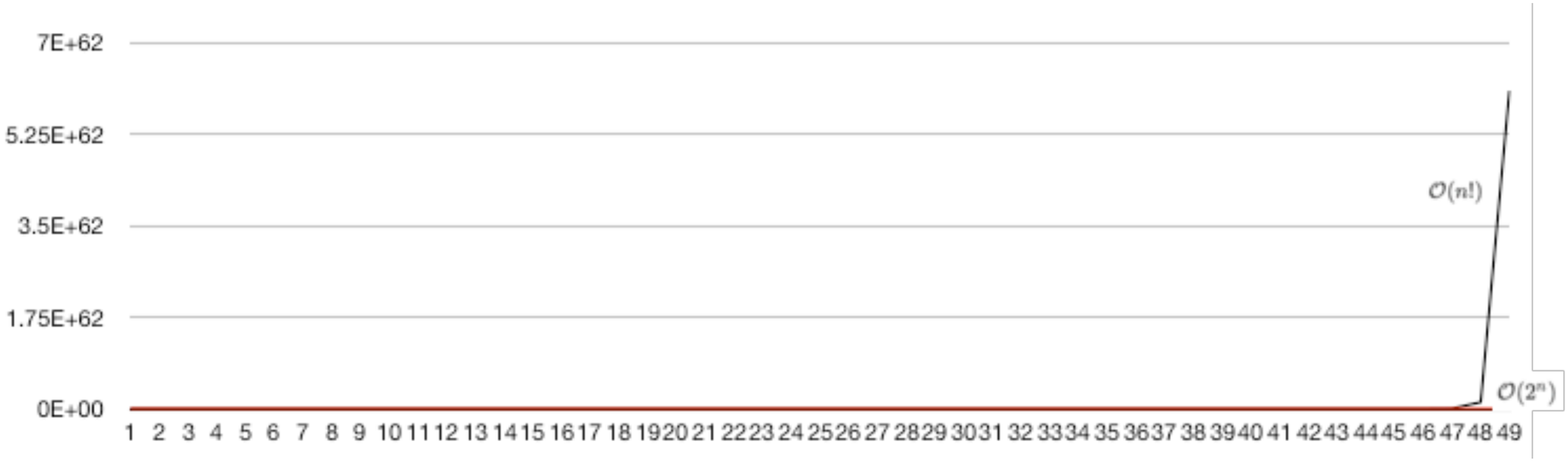
什么是时间复杂度


90
67.5
45
22.5
0
3000
2250
1500
750
0
$\mathcal{O}(n \log n)$
$\mathcal{O}(n)$
$\mathcal{O}(\sqrt{n})$
$\mathcal{O}(\log n)$
$\mathcal{O}(1)$
$\mathcal{O}(n^2)$
1 3 5 7 9 11 13 15 17 19 21 23 25 27 29 31 33 35 37 39 41 43 45 47 49
1 4 7 10 13 16 19 22 25 28 31 34 37 40 43 46 49

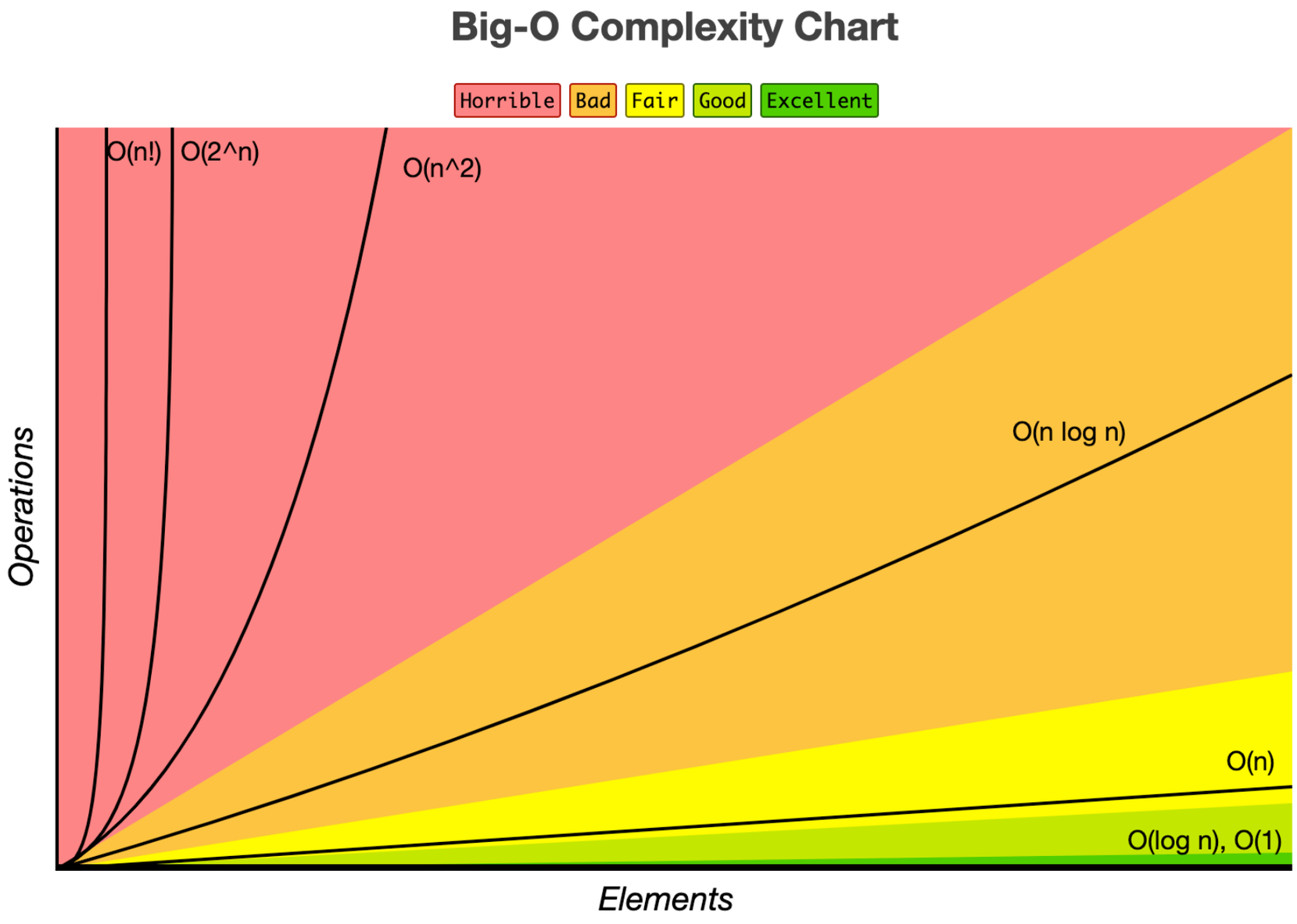
什么是时间复杂度


120000
90000
60000
30000
0
6E+14
4.5E+14
3E+14
1.5E+14
0E+00
$\mathcal{O}(n^3)$
$\mathcal{O}(n^2)$
$\mathcal{O}(2^n)$
$\mathcal{O}(n^3)$
1 4 7 10 13 16 19 22 25 28 31 34 37 40 43 46 49
1 4 7 10 13 16 19 22 25 28 31 34 37 40 43 46 49

什么是时间复杂度


7E+62
5.25E+62
3.5E+62
1.75E+62
0E+00
O(n!)
O(2^n)
1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 21 22 23 24 25 26 27 28 29 30 31 32 33 34 35 36 37 38 39 40 41 42 43 44 45 46 47 48 49

什么是时间复杂度


Big-O Complexity Chart
Horrible
Bad
Fair
Good
Excellent
O(n!)
O(2^n)
O(n^2)
O(n log n)
O(n)
O(log n), O(1)
Operations
Elements

什么是时间复杂度

- O(1) 操作
  - 数组的访问
  - 栈和队列的插入与删除
  - 链表的插入与删除
  - 平均情况的哈希表的查询、插入与删除

什么是时间复杂度

- O(logn) 操作
  - 平均情况的二叉搜索树的访问、查询、插入与删除

什么是时间复杂度

- O(n) 操作
  - 数组的查询、插入与删除
  - 栈和队列的访问与查询
  - 链表的访问与查询
  - 最坏情况的哈希表的查询、插入与删除

什么是时间复杂度

- O(nlogn) 算法
  - 平均情况的快速排序、归并排序
  - 平均情况的堆排序

什么是时间复杂度

- O(n^2) 算法
  - 平均情况的冒泡排序、插入排序和选择排序

如何计算时间复杂度

- 第一行执行了n次 第二行执行n次
  - 语句1执行了n^2次
- 语句2执行了n次
- 语句3执行了logn次
- T(n) = n^2 + n + logn
- 于是该代码时间复杂度为 O(n^2)

```
1  for (int i = 0; i < n; i++) {
2      for (int j = 0; j < n; j++) {
3          a++; //语句 1
4      }
5  }
6  for (int i = 0; i < n; i++) {
7      b++; //语句 2
8  }
9  while (n) {
10     n = n / 2; //语句 3
11 }
```

如何计算时间复杂度

- 小明在美术课上给马上要过生日的妈妈做了张贺卡，为了装饰这张贺卡，小明买了一条彩带，但是彩带上并不是所有颜色小明都喜欢，于是小明决定裁剪这条彩带，以取得最好的装饰效果。 现已知彩带由 n 种不同的颜色顺次相接而成，而每种颜色的装饰效果用一个整数表示（包括正整数，0，或负整数），从左到右依次为 a_1, a_2,..., a_n，小明可以从中裁剪出连续的一段用来装饰贺卡，而装饰效果就是这一段上各个颜色装饰效果的总和，小明需要选取装饰效果最好的一段颜色来制作贺卡（取该段颜色数值之和的最大值）。当然，如果所有颜色的装饰效果都只能起到负面的作用（即 a_i<0），小明也可以放弃用彩带来装饰贺卡（获得的装饰效果为 0）。

如何计算时间复杂度

- 题意是求一个数组的最大子段和
  - 暴力枚举起点位置 i 和重点位置 j
  - 计算和 sum，并更新答案
  - T(n) = $\sum_{i=1}^{n}\sum_{j=i}^{n}(j-i+1)$
    $=\frac{n(n+1)(n+2)}{6}=\frac{n^3+3n^2+2n}{6}$
- 算法复杂度是 O(n^3)

```
int ans = 0;
for (int i = 1; i <= n; i++) {
    for (int j = i; j <= n; j++) {
        int sum = 0;
        for (int k = i; k <= j; k++) {
            sum += a[k];
        }
        if (sum > ans) {
            ans = sum;
        }
    }
}
```

如何计算时间复杂度

- 三层循环没有必要
  - 可以在j向后移的时候 维护 sum
  - 时间复杂度降到了 O(n^2)

```
int ans = 0;
for (int i = 1; i <= n; i++) {
    int sum = 0;
    for (int j = i; j <= n; j++) {
        sum += a[j];
        if (sum > ans) {
            ans = sum;
        }
    }
}
```

- 动态规划
  - 时间复杂度降到了 O(n)

```
int ans = 0, sum = 0;
for (int i = 1; i <= n; i++) {
    sum += a[i];
    if (sum > ans) {
        ans = sum;
    }
    if (sum < 0) {
        sum = 0;
    }
}
```

如何计算时间复杂度

- 教室的墙上挂满了气球，五颜六色，小朋友们非常喜欢。 刚一下课，小朋友们就打算去抢这些气球。每个气球在墙上都有一定的高度，只有当小朋友跳起来时，手能够到的高度大于等于气球的高度，小朋友才能摘到这个气球。为了公平起见，老师让跳的低的小朋友先摘，跳的高的小朋友后摘。小朋友都很贪心，每个小朋友在摘气球的时候都会把自己能摘的气球都摘掉。 很巧的是，小朋友们跳起来手能够着的高度都不一样，这样就不会有跳起来后高度相同的小朋友之间发生争执了。 一共有 n 个小朋友，m 个气球，求每个小朋友能摘多少气球。

如何计算时间复杂度

- 根据题意模拟过程
  - 跳的低的小朋友先摘，所以将小朋友身高从小到大排
  - 对每一个小朋友都枚举一遍所有气球，把能摘到的气球标记为摘过
- 时间复杂度
  - sort的时间复杂度是 O(nlogn)
  - 模拟摘气球的过程是 O(nm)
  - 所以时间复杂度是 O(nlogn + nm)

```
1  sort(stu, stu + n, cmp);
2  for (int i = 0; i < n; i++) {
3      for (int j = 0; j < m; j++) {
4          if (!vis[j] && stu[i].h >= ball[j]) {
5              ans[stu[i].id]++;
6              vis[j] = true;
7              }
8          }
9  }
```

如何计算时间复杂度

- 没有必要每次都遍历所有的气球
  - 只需要枚举没有被摘掉的
  - 气球也可以排序
  - 从上一次没有摘掉的开始
  - pos代表未被摘掉的气球位置
- 时间复杂度
  - 两次sort是 O(nlogn + mlogm)
  - 循环的复杂度是 O(n+m)
  - 时间复杂度是 O(nlogn + mlogm)

```
1  sort(stu, stu + n, cmp);
2  sort(ball, ball + m);
3  int pos = 0;
4  for (int i = 0; i < n; i++) {
5      while (pos < m && ball[pos] <= stu[i].h) {
6          ans[stu[i].id]++;
7          pos++;
8      }
9  }
```

如何计算时间复杂度

- 设有n个活动的集合E={1,2,…,n}，其中每个活动都要求使用同一资源，如演讲会场等，而在同一时间内只有一个活动能使用这一资源。每个活动i都有一个要求使用该资源的起始时间si和一个结束时间fi,且si < fi。如果选择了活动i，则它在半开时间区间[si, fi)内占用资源。若区间[si, fi)与区间[sj, fj)不相交,则称活动i与活动j是相容的。也就是说，当si≥fj或sj≥fi时，活动i与活动j相容。活动安排问题就是要在所给的活动集合中选出最大的相容活动子集合。

如何计算时间复杂度

i	1	2	3	4	5	6	7	8	9	10	11
S[i]	1	3	0	5	3	5	6	8	8	2	12
f[i]	4	5	6	7	8	9	10	11	12	13	14

如何计算时间复杂度

- 贪心算法
  - 总是选择当前最优解
  - 先对活动按照结束时间排序，总是选择结束最早的活动
  - 选中一个活动，移除所有与之冲突的活动
  - 继续循环选择下一个活动

```
1  int greedy(int s[],int f[],int a[],int k)
2  {
3        int i;
4        int j = 0;
5        for(i=0;i<k;i++)
6        {
7          a[i] = 0;//初始所有活动都未被安排
8        }
9        a[0] = 1;  //安排第一个活动
10       int count = 1;
11       for(i=1;i<k;i++)
12    {
13       if(s[i] > f[j])
14       {
15          a[i] = 1;
16          printf("开始%d,结束%d.",s[i],f[i]);
17          j = i;
18          count++;
19          printf("第%d个活动被安排\n",i+1);
20       }
21    }
22    return count;
```

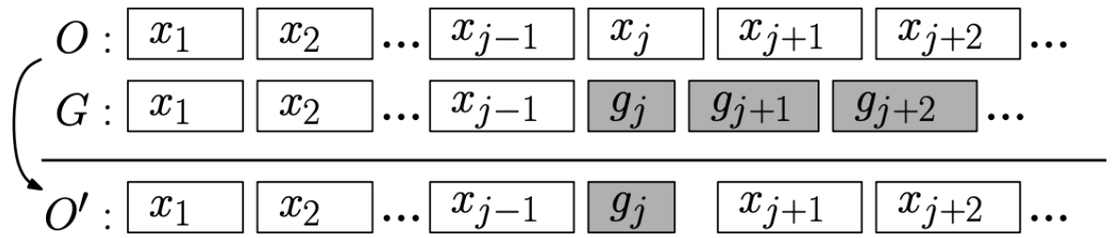
如何计算时间复杂度

- 算法复杂度
  - O(n)
- 为什么贪心能够效率高?
  - 将一个问题划分为两个子问题
  - 一个子问题为空
  - 继续循环另外一个子问题

```
int greedy(int s[],int f[],int a[],int k)
{
      int i;
      int j = 0;
      for(i=0;i<k;i++)
      {
        a[i] = 0;//初始所有活动都未被安排
      }
      a[0] = 1;  //安排第一个活动
      int count = 1;
      for(i=1;i<k;i++)
      {
         if(s[i] > f[j])
         {
            a[i] = 1;
            printf("开始%d,结束%d.",s[i],f[i]);
            j = i;
            count++;
            printf("第%d个活动被安排\n",i+1);
         }
      }
      return count;
```

如何计算时间复杂度

- 为什么贪心能够最优?
  - 假设最优集为：O = {x1, x2,…,xk}
  - 假设贪心集为：G = {g1, g2,…,gk}
  - 如果两者相同，证明结束
  - 如果两者不同，那么肯定存在一个序号j，两个集合元素开始不同
    - A = {x1,…,xj-1,xj,xj+1…}
    - B = {x1,…,xj-1,gj,gj+1…}
    - bj的完成时间一定不晚于xj （因为贪心）
    - bj不会和xj+1冲突
    - G至少和O一样好



如何计算时间复杂度

- 赫夫曼编码是一种无损数据压缩算法。在计算机数据处理中，赫夫曼编码使用变长编码表对源符号（如文件中的一个字母）进行编码，其中变长编码表是通过一种评估来源符号出现机率的方法得到的，出现机率高的字母使用较短的编码，反之出现机率低的则使用较长的编码，这便使编码之后的字符串的平均长度、期望值降低，从而达到无损压缩数据的目的。例如，在英文中，e的出现机率最高，而z的出现概率则最低。当利用赫夫曼编码对一篇英文进行压缩时，e极有可能用一个比特来表示，而z则可能花去25个比特（不是26）。用普通的表示方法时，每个英文字母均占用一个字节（byte），即8个比特。二者相比，e使用了一般编码的1/8的长度，z则使用了3倍多。倘若我们能实现对于英文中各个字母出现概率的较准确的估算，就可以大幅度提高无损压缩的比例。

如何计算时间复杂度

- 字典序最小问题
  - 输入一个整数，然后在输入N长度的字符串，每次从字符串开头或者末尾取一个字母，组成新的字符串，组成字符串的字典序最小
  - 例如 6 ACDBCB 输出 ABCBCD
  - 贪心