COURSE NOTES / DAY 04

算法设计与分析　Day4

随机快速排序、排序下界与三道作业详解

RedMoon　排序进阶　2026 年 10 月 15 日

目录

一、课堂热身：多数元素为什么可以两两抵消 1
二、分治排序：归并排序与快速排序的分工 2
三、随机快速排序：划分与实现 3
四、输入、随机选择与运行时间 5
五、随机快速排序的期望时间推导 5
六、Quickselect：只递归包含目标秩的一侧 8
七、堆排序与自底向上建堆 9
八、计数排序与值域大小 10
九、比较排序下界：为什么至少需要 $\Omega(n \log n)$ 次比较 11
十、常见排序算法的复杂度对照 12
十一、作业 1：非递归的线性时间最大子数组 12
十二、作业 2：用最小堆合并 k 个有序表 14
十三、作业 3：截断快速排序，再对全数组做插入排序 16
十四、复习题 20
附 • 资料下载 20

RedMoon • 算法设计与分析 • Day4

快速排序的一次划分会把枢轴放到最终位置。剩下的工作量，取决于枢轴两侧各有多少元素。本次课从划分规模出发，推导随机快速排序的期望时间，再结合堆排序、计数排序和决策树，讨论排序算法利用的信息及其复杂度下界。

文末详解三道作业：线性时间最大子数组、用最小堆合并有序表，以及快速排序与插入排序的组合。每题包含题意、算法、执行示例、正确性证明和复杂度分析。

记 $\lg n = \log_2 n$，$\ln n$ 为自然对数；区间 $A[l..r]$ 包含两端。Python 代码使用 **0-based 下标**，返回的最大子数组区间也包含左右端点。时间分析默认一次比较、数组访问及机器字范围内的算术运算花费常数时间；涉及输入编码长度时，会单独说明。

一、课堂热身：多数元素为什么可以两两抵消

1. 多数元素与众数的区别

课件第 2 页的题目是 LeetCode 169「多数元素」：给定一个非空数组，保证存在出现次数 **严格大于** $\lfloor n/2 \rfloor$ 的元素，找出它。

众数只要求出现次数最多，多数元素则要求占据一半以上。例如 $[1, 1, 2, 2, 3]$ 中，1 和 2 都是众数，但没有多数元素。严格多数至多只有一个，因为两个数的出现次数不可能都超过 $n/2$。

直接统计频次即可求解。用哈希表记录，期望时间为 $O(n)$，最坏空间为 $O(n)$；若已知整数值域为 $1..K$，也可以使用长度为 K 的计数数组，时间为 $O(n + K)$，空间为 $O(K)$。值域越大，初始化计数数组的成本越高。

2. Boyer–Moore 投票法

从数组中删去两个值不同的元素，原来的多数元素仍然是剩余数组中的多数元素。

设多数元素为 x，原来出现 $c > n/2$ 次。若删去的一对中包含 x，它还剩 $c-1$ 次，且 $c-1 > (n-2)/2$；若这一对都不是 x，它的次数不变，占比反而更高。因此，不断抵消后留下的候选值就是 x。

实现时维护一个候选值 `candidate` 和未被抵消的数量 `count`：

```
def majority_element(a):
    if not a:
        raise ValueError("input must be nonempty")
    candidate, count = None, 0
    for x in a:
        if count == 0:
            candidate = x
        count += 1 if x == candidate else -1
    return candidate
```

以 $[2,2,1,1,1,2,2]$ 为例：

下标	读入值	处理后的候选值	count	发生的操作
0	2	2	1	空状态建立候选值
1	2	2	2	同值累积
2	1	2	1	抵消一对不同值
3	1	2	0	当前候选全部抵消
4	1	1	1	重新建立候选值
5	2	1	0	再次抵消
6	2	2	1	剩下候选值 2

`count` 记录尚未被抵消的候选数量，不代表候选值在整个数组中的出现次数。算法只扫描一次，时间 $\Theta(n)$，辅助空间 $O(1)$。

若题目不保证多数元素存在，还需在第二遍扫描中统计候选值，检查它是否出现超过 $\lfloor n/2 \rfloor$ 次。例如 $[1,2,3]$ 第一遍也会产生候选值，但数组中没有多数元素。

二、分治排序：归并排序与快速排序的分工

排序的输出既要按非降序排列，也要与输入包含完全相同的元素及重复次数。正确性证明需要同时覆盖有序性与元素保留，避免遗漏或增加元素。

课堂先区分了问题、算法和分析。排序规定要解决什么问题，归并排序与快速排序给出具体做法，正确性和运行时间分析则回答这些做法是否正确、需要多少代价。同一个问题可以有多种算法，分析时也可以采用不同模型。

分治环节	归并排序	快速排序
边界	长度不超过 1 时返回	长度不超过 1 时返回
分	按下标切成两半，切分本身简单	扫描元素，按枢轴值划分
治	排序左右两半	排序枢轴两侧
合	线性合并两个有序区间	枢轴已经就位，无须再线性合并
子问题规模	两侧约为 $n/2$	可能从 $0, n-1$ 到接近均分

归并排序的主要线性工作在合并阶段，快速排序则在划分阶段。分析快速排序时，需要继续考察划分后两个子问题的规模。

三、随机快速排序：划分与实现

1. 枢轴必须退出后续递归

先从数组中均匀随机选取一个位置，以该位置的元素为枢轴 x。单独取出这次出现，再将其余元素分成：

$$B = \{a : a \le x\}, \qquad C = \{a : a > x\}.$$

递归结果为：

$$\mathrm{sort}(B) + [x] + \mathrm{sort}(C).$$

这里的加号表示顺序拼接。枢轴本身不再进入 B 或 C，所以两侧规模之和是 $n-1$，每个递归子问题都严格变小。

数组含重复值时，也必须排除选作枢轴的这次出现。否则，把枢轴放回「小于等于」的一侧后，全相等数组可能原样进入下一次递归，无法结束。原地划分会固定一个枢轴位置，使后续区间严格缩小。

2. 原地划分的四个区域

课件第 8 页使用 Lomuto 划分。先把随机选中的枢轴换到 $A[r]$，令 $x = A[r]$，再扫描 $A[p..r-1]$。

循环每次开始时维护：

区域	语义
$A[p..i]$	已处理且全部 $\le x$
$A[i+1..j-1]$	已处理且全部 $> x$
$A[j..r-1]$	尚未检查
$A[r]$	暂存枢轴 x

若 $A[j] \le x$，先令 i 加 1，再交换 $A[i]$ 与 $A[j]$，把这个元素纳入左区间；若 $A[j] > x$，则直接推进扫描位置。

扫描结束后，交换 $A[i+1]$ 与 $A[r]$。令 $q = i+1$，得到：

$$A[p..q-1] \le A[q], \qquad A[q+1..r] > A[q].$$

```
from random import Random


def randomized_partition(a, p, r, rng):
    pivot_index = rng.randrange(p, r + 1)
    a[pivot_index], a[r] = a[r], a[pivot_index]
    x = a[r]
    i = p - 1
    for j in range(p, r):
        if a[j] <= x:
            i += 1
            a[i], a[j] = a[j], a[i]
    a[i + 1], a[r] = a[r], a[i + 1]
    return i + 1
```

```python
def quicksort(a, rng=None):
    if rng is None:
        rng = Random()

    def sort(p, r):
        if p >= r:
            return
        q = randomized_partition(a, p, r, rng)
        sort(p, q - 1)
        sort(q + 1, r)

    sort(0, len(a) - 1)
    return a
```

3. 课件示例的交换过程

输入 $[40, 30, 10, 20]$，取末尾的 20 为枢轴。使用 0-based 下标，初始 $i = -1$：

扫描阶段	比较结果与动作	数组
$j = 0$	$40 > 20$，不交换	$[40, 30, 10, 20]$
$j = 1$	$30 > 20$，不交换	$[40, 30, 10, 20]$
$j = 2$	$10 \le 20$，令 $i = 0$，交换位置 0 和 2	$[10, 30, 40, 20]$
扫描结束	交换位置 $i + 1 = 1$ 和末尾枢轴	$[10, 20, 40, 30]$

返回 $q = 1$。20 已经就位，但右侧 $[40, 30]$ 还没有排序。一次划分只保证枢轴两侧满足大小关系。

4. 正确性与空间

用循环不变式证明划分正确。初始化时，两个已处理区域均为空；扫描中，每次交换将符合条件的元素放进左区间，并保持右区间的性质；扫描结束时，未知区域为空，最后一次交换将枢轴放到两区之间。

再对区间长度归纳，即可证明快速排序正确。长度不超过 1 时，区间已有序。更长的区间经过划分后，枢轴两侧满足大小关系；由归纳假设，两个更短区间递归排序后也各自有序，拼接起来便得到整个有序区间。全过程只交换已有元素，因此输入的多重集合保持不变。

长度为 $m = r - p + 1$ 的划分需要 $m - 1$ 次关键字比较，时间 $\Theta(m)$，辅助空间 $O(1)$。上面的快速排序仍有递归栈：互异元素、随机枢轴时，期望栈深为 $O(\log n)$，最坏可达 $O(n)$。若每次只递归较短一侧、用循环处理较长一侧，可以把栈空间最坏限制在 $O(\log n)$。

5. 重复值与稳定性

Lomuto 使用 `<=` 判断。所有元素相等时，每次划分都把其余 $m - 1$ 个元素留在左侧，无论随机选中哪个位置，时间都是 $\Theta(n^2)$。随机化改变了枢轴的选择方式，但这个划分规则仍会把所有相等元素放到同一侧。

课件之外，还可以用三路划分处理重复值：将元素分成 $< x$、$= x$、$> x$ 三区，只递归两侧；所有值相等时，一次线性扫描即可结束。后续期望递推假设 **元素互异**，此时每个枢轴秩恰好对应一种左右规模。

这种交换式快速排序通常不稳定。例如给相同键附上标签，$[(2, a), (2, b), (1, c)]$ 以 1 划分后成为 $[(1, c), (2, b), (2, a)]$，两个键为 2 的记录已交换先后。

四、输入、随机选择与运行时间

1. 运行时间有两个自变量

记 $T(A,R)$ 为输入数组 A、随机选择序列 R 下的运行时间。即使固定 A，不同的枢轴序列也会产生不同的递归树。

「随机快速排序的期望时间」通常对算法的随机选择取期望：

$$\max_{|A|=n} \mathbb{E}_R[T(A,R)].$$

若元素互异，这个期望为 $O(n\log n)$。分析时先固定任意输入，再对枢轴的随机选择取平均，无须假设输入本身是随机排列。

平均情况 $\mathbb{E}_A[T(A)]$ 则对输入取平均，需要先给定输入分布。它可能与随机算法的期望时间具有相同的渐近阶，但两者取平均的对象不同。

2. 不带有序性预检查的普通版本

若每次都接近均分：

$$T(n) = 2T(n/2) + \Theta(n) = \Theta(n\log n).$$

若每次都选到当前最小值或最大值：

$$T(n) = T(n-1) + \Theta(n) = \Theta(n^2).$$

后一种情况下，扫描规模累加为 $n + (n-1) + \cdots + 2$。随机快速排序也可能连续选到这样的枢轴，因此最坏时间仍为二次量级，未降至 $O(n\log n)$。

3. 加入有序性检查后的版本

课件第 11 页另加了一步：先扫描数组，发现已经有序就返回。对这个变体，已排序输入的运行时间是 $\Theta(n)$，因为只需完成一次检查。

带预检查版本在有序输入上花费的时间，与普通快速排序在均衡划分下的 $\Theta(n\log n)$ 对应不同条件。几种情况如下：

所讨论的情况	时间
普通版本，划分始终接近均衡	$\Theta(n\log n)$
普通版本，划分始终极端不均衡	$\Theta(n^2)$
带预检查版本，输入原本有序	$\Theta(n)$
互异元素的随机版本，对随机枢轴取期望	$\Theta(n\log n)$

预检查本身需要线性时间。下一节沿用课件的规范化递推，分析不提前返回的普通随机版本。

五、随机快速排序的期望时间推导

1. 按枢轴秩分类

设 $T(n)$ 为规模 n 的期望代价。元素互异、从当前子数组均匀选择枢轴时，枢轴可能有 $0, 1, \dots, n-1$ 个更小的元素，每种情况概率都是 $1/n$。

当左侧规模为 i 时，右侧规模为 $n-1-i$。将一次线性处理的代价规范化写成 n，得到：

$$T(n) = \frac{1}{n}\sum_{i=0}^{n-1}\big(T(i) + T(n-1-i)\big) + n.$$

两部分求和包含相同的项，只是次序相反，因此：

$$T(n) = \frac{2}{n}\sum_{i=0}^{n-1} T(i) + n.$$

若保留具体实现中的常数项和线性项系数，推导方式与渐近阶不变。边界代价 $T(0), T(1)$ 均为常数；相邻规模作差时，要从两个递推式都成立的规模开始，再把有限个初值并入常数。

2. 乘以规模，再对相邻式子作差

两边乘以 n：

$$nT(n) = 2\sum_{i=0}^{n-1} T(i) + n^2.$$

把规模换为 $n-1$：

$$(n-1)T(n-1) = 2\sum_{i=0}^{n-2} T(i) + (n-1)^2.$$

两式相减，求和只留下最后一项：

$$nT(n) - (n-1)T(n-1) = 2T(n-1) + 2n - 1,$$

于是：

$$nT(n) = (n+1)T(n-1) + 2n - 1.$$

求和项由此消去，递推只剩相邻两个规模。

3. 除以 $n(n+1)$，得到可累加的形式

$$\frac{T(n)}{n+1} = \frac{T(n-1)}{n} + \frac{2n-1}{n(n+1)} \le \frac{T(n-1)}{n} + \frac{2}{n+1}.$$

令 $S(n) = T(n)/(n+1)$，就有 $S(n) \le S(n-1) + 2/(n+1)$。展开至一个固定初值 n_0：

$$\frac{T(n)}{n+1} \le \frac{T(n_0)}{n_0+1} + 2\sum_{j=n_0+1}^{n}\frac{1}{j+1} = O(\log n).$$

因此：

$$\boxed{T(n) = O(n \log n)}.$$

若选择使递推在 $n = 1$ 也成立的规范化初值，便可直接写成课件中的 $T(1)/2+2(1/3+\cdots+1/(n+1))$。若程序在长度为 1 时立即返回，从 $n_0 = 2$ 开始展开即可，渐近结论一致。

调和级数产生对数，因为：

$$\sum_{j=1}^{n} \frac{1}{j} \le 1 + \int_{1}^{n} \frac{dx}{x} = 1 + \ln n.$$

均衡递归树因为每次规模约减半而产生对数层数。这里则先对所有枢轴秩取平均，再由调和级数得到对数因子，因此允许部分划分很不均衡。

4. 为什么随机版本仍可能很慢

互异元素下，每次恰好选到当前最小值这一种具体序列的概率为：

$$\frac{1}{n} \cdot \frac{1}{n-1} \cdots \frac{1}{2} = \frac{1}{n!}.$$

这个概率对应一条指定的极端路径。分析算法整体变慢的概率，还要考虑其他坏路径。上面的递推证明了期望为 $O(n \log n)$；若要得到指数衰减的尾界等更强的高概率结论，还需额外论证，不能由期望上界直接推出。

5. 补充推导：精确比较次数

如果只统计划分时的关键字比较，令 C_n 为期望比较次数，$C_0 = C_1 = 0$，则：

$$C_n = n - 1 + \frac{2}{n} \sum_{i=0}^{n-1} C_i.$$

设排好序后各元素为 $z_1 < \cdots < z_n$。一对 z_i, z_j 只有在区间 $z_i, \ldots, z_j$ 中最先成为枢轴的是某个端点时才会相互比较，因此：

$$\Pr(z_i \text{ 与} z_j \text{ 比较}) = \frac{2}{j - i + 1}.$$

利用期望的线性性求和：

$$\mathbb{E}[C_n] = \sum_{1 \le i < j \le n} \frac{2}{j - i + 1} = 2(n+1)H_n - 4n, \qquad H_n = \sum_{j=1}^{n} \frac{1}{j}.$$

最坏比较次数为 $n(n-1)/2$。下图由这两个公式绘制，比较的是理论计数：

期望比较次数约为 $2n \ln n$，最坏比较次数按二次增长。这里的 C_n 只统计关键字比较；前面将每层全部线性工作简写为 n 的 $T(n)$，采用的是另一种计量单位。

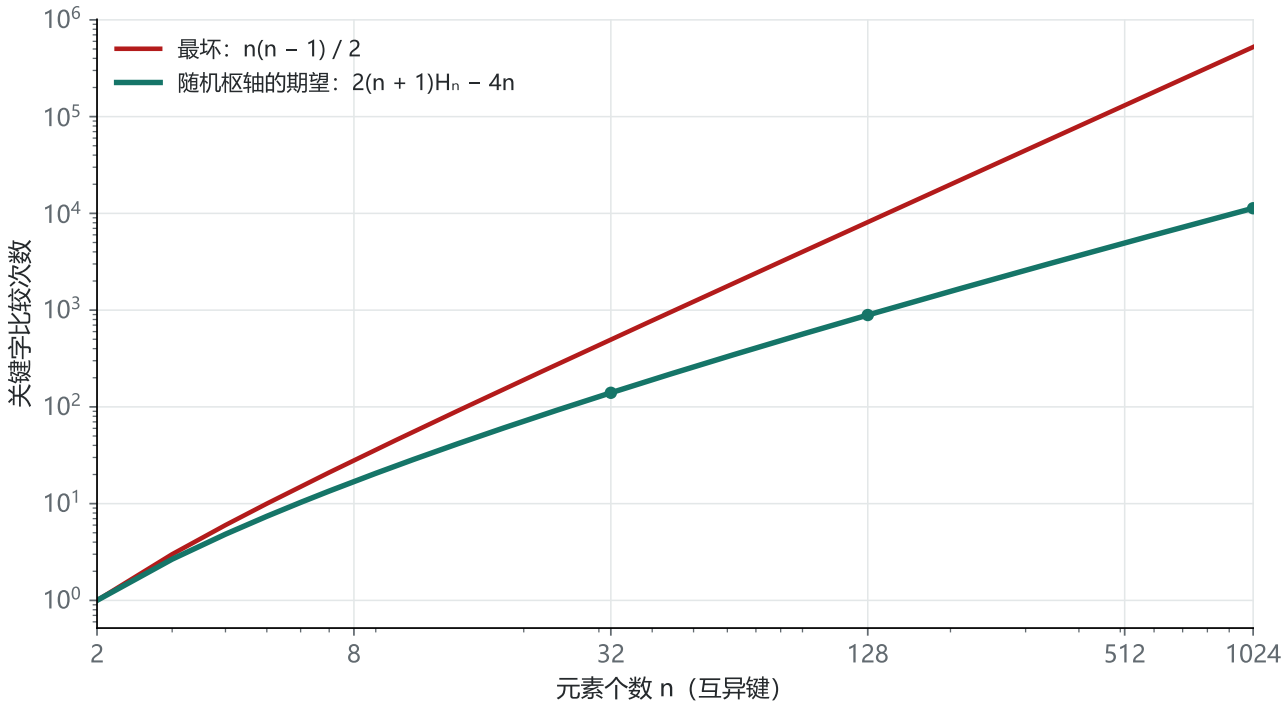


图 1: 互异元素下，随机快速排序的期望比较次数与最坏比较次数；双对数坐标展示增长速度。

六、Quickselect：只递归包含目标秩的一侧

若目标是第 k 小元素，每次划分后只需判断枢轴的最终位置：

- 枢轴位置恰好为目标位置，直接返回。
- 目标在左侧，只继续处理左侧。
- 目标在右侧，只继续处理右侧。

```python
def quickselect(a, rank, rng=None):
    if not 0 <= rank < len(a):
        raise ValueError("rank must be a valid zero-based index")
    if rng is None:
        rng = Random()
    p, r = 0, len(a) - 1
    while p <= r:
        q = randomized_partition(a, p, r, rng)
        if q == rank:
            return a[q]
        if rank < q:
            r = q - 1
        else:
            p = q + 1
```

这里的 `rank` 是 0-based 排名，例如奇数长度数组的中位数对应 `rank = len(a) // 2`。算法会改动输入数组。

若每次保留的规模约减半，总扫描代价为 $n + n/2 + n/4 + \cdots = O(n)$。随机枢轴下的期望线性时间，可以用归纳法证明。设 $S(n)$ 为所有目标秩中最大的期望代价。固定任意目标秩，枢轴左边有 i 个元素时，记下一步实际处理的规模为 m_i；若恰好找到目标，则记为 0。于是：

$$0 \le m_i \le \max(i, n-1-i) < n.$$

对于 $n \ge 2$，

$$\frac{1}{n}\sum_{i=0}^{n-1}\max(i,n-1-i)\le\frac{3n}{4}.$$

用归纳假设 $S(m)\le am$ 控制所有更小规模，对任何目标秩都有：

$$\mathbb{E}[T]\le cn+\frac{a}{n}\sum_{i=0}^{n-1}m_i\le cn+\frac{3an}{4}\le an,\qquad a\ge 4c.$$

同一个常数 a 对所有目标秩都成立，故 $S(n)\le an$，有限初值并入常数。因此，互异元素、均匀随机枢轴下的期望时间为 $\Theta(n)$，最坏仍为 $\Theta(n^2)$。

课件第 19 页将找中位数的期望时间写成 $O(\log n)$，应改为 $O(n)$。初始划分就要扫描线性数量的元素，分析总时间还需累加后续各层的扫描工作。要保证确定性的最坏线性时间，则需采用其他枢轴策略，例如 median-of-medians。

七、堆排序与自底向上建堆

1. 建堆、查看堆顶与取出堆顶

每次扫描剩余数组找最小值，连续取出 n 次的总代价为 $\Theta(n^2)$。最小堆要求父节点不大于子节点，因此全局最小值始终在堆顶。

操作	代价	做了什么
自底向上建堆	$O(n)$	从整个输入建立堆序
查看最小值	$O(1)$	读取堆顶，不删除
取出最小值	$O(\log n)$	删除堆顶，并恢复堆序

课堂将最小堆视作黑盒：先建堆，再反复弹出最小值。

```
import heapq


def heap_sort(a):
    heap = list(a)
    heapq.heapify(heap)
    result = []
    while heap:
        result.append(heapq.heappop(heap))
    return result
```

建堆花费 $O(n)$，随后 n 次删除各需至多 $O(\log n)$，总时间为 $O(n\log n)$。上面的实现复制了输入，并单独保存输出，额外空间为 $O(n)$。教材中常见的 **原地堆排序**使用最大堆，将最大值交换到数组末尾，辅助空间可降至 $O(1)$。空间结论取决于采用哪种实现。

2. 补充推导：建堆的 $O(n\log n)$ 上界为何可以收紧

逐个插入 n 个数，时间上界为 $O(n\log n)$。`heapify` 采用自底向上调整：大多数节点靠近叶子，调整距离很短，只有少数上层节点可能下降很多层。

高度为 h 的节点数量至多为 $O(n/2^{h+1})$，每个节点的调整代价为 $O(h)$。因此：

$$\sum_{h\ge 0} O\left(\frac{n}{2^{h+1}}\right) O(h) = O(n) \sum_{h\ge 0} \frac{h}{2^{h+1}} = O(n).$$

课堂主要使用建堆、取堆顶的接口及其复杂度。上面的求和补充了线性建堆的原因，后面的多路归并也会用到这一结论。

八、计数排序与值域大小

1. 算法与例子

假设输入都是正整数，值域为 $1..K$。建立计数数组 $C[0..K]$，让 $C[x]$ 记录 x 出现的次数，再按 $1, 2, \dots, K$ 的顺序输出。

对课堂例子 $[1, 5, 3, 2, 2]$：

值 x	1	2	3	4	5
次数 $C[x]$	1	2	1	0	1

输出为 $[1, 2, 2, 3, 5]$。

```
def counting_sort(a):
    if not a:
        return []
    upper = max(a)
    counts = [0] * (upper + 1)
    for x in a:
        counts[x] += 1
    result = []
    for x in range(1, upper + 1):
        result.extend([x] * counts[x])
    return result
```

这段代码要求输入为正整数；若要支持 0 或负整数，需要调整扫描范围或增加偏移。找最大值、统计频次及输出共需 $O(n)$，初始化和遍历值域需 $O(K)$，所以：

$$\text{时间} = O(n + K), \qquad \text{辅助空间} = O(K).$$

若把返回的输出数组也计入额外存储，总空间为 $O(n + K)$。

这个简化版本只输出整数键值。若每条记录还带有其他字段，稳定排序就要保留完整记录及相同键的原有先后顺序，应采用累计计数与稳定放置等标准写法。重复输出相同整数，并不能完成这种记录排序。

2. 数值大小与二进制编码长度

复杂度式中的 K 表示数值大小，而计算复杂性中的「多项式时间」以输入编码长度 L 为参照。

正整数 a_i 的二进制长度是 $\lfloor \lg a_i \rfloor + 1$。变长编码下，总输入长度还要加上分隔或长度信息；通常可以把主要部分写为：

$$L \asymp \sum_{i=1}^{n} (\lfloor \lg a_i \rfloor + 1).$$

若统一用足够容纳 $1..K$ 的定长字段，单个字段需要 $\lceil \lg(K+1) \rceil$ 位，总长度约为 $n \lg K$。

数值可以相对编码长度呈指数增长。例如数组只含一个数 2^b，存储它需要 $b+1$ 位；若算法初始化从 1 到 2^b 的计数表，单这一步就需要 $\Theta(2^b)$ 的时间和空间。

因此，$O(n+K)$ 虽然是数值参数的多项式，却未必是二进制输入长度的多项式。这类依赖数值大小的界称为 **伪多项式时间**。若输入额外满足 $K = O(n^c)$，其中 c 为固定常数，则该算法在这一受限输入族上是多项式时间；当 $K = O(n)$ 时，时间进一步降为线性。

3. 桶排序、基数排序与 Sleep Sort 的条件

课堂还提到了桶排序和基数排序，作为课件延伸。桶排序的期望线性时间依赖适当的输入分布和桶内处理条件；输入不是正整数这一点，本身不足以保证桶排序线性完成。基数排序则需计入位数及每一位的排序成本。

Sleep Sort 按数值设置等待时间，再按唤醒顺序输出。分析它时，要把计时、调度、并发资源和值域大小纳入计算模型；只数代码中的显式比较会漏掉这些代价。系统调度也未必能让任意输入都可靠有序。这个例子说明，复杂度结论必须与计算模型一起理解。

九、比较排序下界：为什么至少需要 $\Omega(n \log n)$ 次比较

1. 用决策树表示算法

限制算法只能通过比较获得键之间的顺序信息。对 n 个互异元素，一次比较只会产生两个可能结果，可表示为二叉树的一个分支。

树根对应第一次比较；向左或向右表示比较结果；后续节点对应由此前结果决定的下一次比较。到达叶子时，算法已经能确定一个正确的输出下标顺序。

一条根到叶子的路径长度，就是该输入执行的比较次数；树高就是最坏输入所需的比较次数。

2. 为什么至少要有 $n!$ 个叶子

把输入位置标记为 $a_1, \ldots, a_n$。当键互异时，它们可能有 $n!$ 种严格全序。一个固定的输出下标顺序，例如 (a_2, a_1, a_3)，只能在与它匹配的全序下正确。

正确的比较排序必须覆盖所有这些可能性。两个需要不同输出顺序的全序不能最终共享同一个输出叶子，因此叶子数至少是 $n!$。

这里区分的是元素相对顺序。即使键可以取无限多种数值，只要全序相同，它们仍可以沿同一条比较路径到达同一叶子。

3. 由叶子数推出高度

高度为 h 的二叉树最多有 2^h 个叶子，因此：

$$2^h \geq n!, \qquad h \geq \lceil \lg(n!) \rceil.$$

再证明 $\lg(n!) = \Theta(n \lg n)$。上界直接用 $n! \leq n^n$：

$$\lg(n!) \leq n \lg n.$$

下界取阶乘中最大的至少 $n/2$ 个因子，每个都不小于 $n/2$：

$$n! \geq (n/2)^{n/2}, \qquad \lg(n!) \geq \frac{n}{2}\lg\frac{n}{2} = \Omega(n\lg n).$$

由此得到：

> 任意正确的比较排序，最坏情况需要$\Omega(n\log n)$ 次比较。

课件第 30 页在阶乘附近标注了 $\Theta(n^n)$。这里应使用粗上界 $n! \leq n^n$，并在取对数后得到 $\lg(n!) = \Theta(n\lg n)$，不能把前一个标记当作阶乘的等价式。

例如 $n = 3$ 时有 6 种全序，二叉树至少要有 6 个叶子，最坏路径至少为 $\lceil \lg 6 \rceil = 3$。一般的叶子计数只给出下界，不保证存在某个算法恰好达到这个整数值。

4. 下界的适用范围

该下界针对最坏情况。带有序性预检查的算法遇到已排序输入时，仍可在线性扫描后返回；某些输入较容易处理，与最坏情况下界一致。

计数排序通过键值直接寻址计数表，利用了比较模型以外的信息，因此其 $O(n + K)$ 时间与比较排序下界相容。特定分布、重复值结构及其他受限输入族，也可能允许更快的算法。

推广到随机算法的期望下界，还需对输入分布取平均。让 $n!$ 种全序等概率出现，任何固定的正确比较决策树，平均叶深都至少为 $\lg(n!)$。再对随机选择取平均，平均比较次数仍为 $\Omega(n\log n)$，因此至少有一个输入的期望比较次数达到这一量级。这里要区分两个量词顺序：固定随机选择后存在坏输入，并不等于存在一个输入对所有随机选择都坏。

一次二元比较至多提供一比特的分支信息，而排序需要区分 $n!$ 种顺序。决策树用叶子数与树高的关系，给出了这一信息量直觉的严格证明。

十、常见排序算法的复杂度对照

下表列出常见实现的复杂度，以及平均或期望结论所需的条件。

算法	最坏时间	平均或期望时间	辅助空间与条件
插入排序	$\Theta(n^2)$	随机排列下 $\Theta(n^2)$	$O(1)$；近乎有序时为 $\Theta(n + I)$
归并排序	$\Theta(n\log n)$	$\Theta(n\log n)$	常见数组实现为 $O(n)$
随机快速排序	$\Theta(n^2)$	互异元素下 $\Theta(n\log n)$	原地划分；常见递归栈期望 $O(\log n)$
原地堆排序	$O(n\log n)$	$O(n\log n)$ 上界	$O(1)$；使用最大堆原地放置
计数排序	$\Theta(n + K)$	$\Theta(n + K)$	计数表 $O(K)$；含输出为 $O(n + K)$

其中 I 为当前数组的逆序对数：若 $i < j$ 且 $A[i] > A[j]$，两者就构成一个逆序对。插入排序每次向右移动一个前驱元素，都会消除一个逆序对，第三道作业将据此计算移动次数。

十一、作业 1：非递归的线性时间最大子数组

1. 题意与状态分析

来源：课件第 32 页，CLRS 练习 **4.1-5**。

给定非空数组 A，元素可以为正、负或零。求一个 **非空连续子数组**，使元素和最大。要求从左向右扫描，采用非递归算法，在线性时间内完成。

题目提示：已知前缀 $A[0..j]$ 的答案，扩展到 $A[0..j+1]$ 后，最优解只有两种位置关系：

1. 完全位于旧前缀中。
2. 以新元素 $A[j+1]$ 结尾。

因此需要同时记录整个前缀的最佳值，以及恰好在当前位置结束的最佳值。前者保存已有答案，后者决定新元素接在哪一段后面；只保留全局最大值会缺少后一项信息。

2. 两个状态与递推

定义：

$$E_j = \max_{0\le i\le j} \sum_{t=i}^{j} A[t],$$

即以位置 j 结尾的最大非空子数组和。另设：

$$B_j = \max_{0\le i\le t\le j} \sum_{u=i}^{t} A[u],$$

即整个前缀 $A[0..j]$ 中的最大非空子数组和。

以 j 结尾的最优区间，要么只选 $A[j]$，要么把一个以 $j-1$ 结尾的最优区间延长：

$$E_j = \max\{A[j], E_{j-1} + A[j]\}.$$

前缀最佳值则为：

$$B_j = \max\{B_{j-1}, E_j\}.$$

初值为 $E_0 = B_0 = A[0]$。旧后缀贡献为负时，从当前元素重新开始更优。这一选择取决于以当前位置结束的状态，不能由此前的全局最佳值决定。

3. 返回最大和与左右端点的实现

```python
def max_subarray(a):
    if not a:
        raise ValueError("nonempty subarray requires nonempty input")
    end_sum = best_sum = a[0]
    end_left = best_left = best_right = 0

    for j in range(1, len(a)):
        if end_sum + a[j] < a[j]:
            end_sum = a[j]
            end_left = j
        else:
            end_sum += a[j]
        if end_sum > best_sum:
            best_sum = end_sum
            best_left, best_right = end_left, j

    return best_sum, best_left, best_right
```

返回值为 **(最大和，左端点，右端点)**。相等时保留已有答案，因此有多个最优区间时，会返回最先记录的那个，满足题目求一个最优区间的要求。

4. 手算完整过程

取：

$$A = [-2, 1, -3, 4, -1, 2, 1, -5, 4].$$

j	$A[j]$	E_j	当前结尾区间	B_j	已记录的最佳区间
0	-2	-2	$[0,0]$	-2	$[0,0]$
1	1	1	$[1,1]$	1	$[1,1]$
2	-3	-2	$[1,2]$	1	$[1,1]$
3	4	4	$[3,3]$	4	$[3,3]$
4	-1	3	$[3,4]$	4	$[3,3]$
5	2	5	$[3,5]$	5	$[3,5]$
6	1	6	$[3,6]$	6	$[3,6]$
7	-5	1	$[3,7]$	6	$[3,6]$
8	4	5	$[3,8]$	6	$[3,6]$

答案为 $[4,-1,2,1]$，总和为 6，端点为 $[3,6]$。处理到位置 7 时，结尾状态的和下降，但记录的全局最佳区间保持不变。

5. 正确性证明

循环处理完下标 j 后，保持两个不变式：

1. `end_sum` 等于所有以 j 结尾的非空子数组中的最大和，`end_left` 是对应起点。
2. `best_sum` 等于前缀 $A[0..j]$ 中所有非空子数组的最大和，记录的端点对应一个达到该和的区间。

初始 $j = 0$ 时只有一个非空区间 $[0,0]$，两个状态均取 $A[0]$，成立。

假设处理完 $j-1$ 时两个不变式成立。以 j 结尾的非空区间，要么从 j 开始，要么由以 $j-1$ 结尾的区间延长。后一类都加上同一个新元素，选旧区间中和最大的那个即可，因此第一次更新得到 E_j。前缀的最优区间则要么不经过 j，要么以 j 结尾，第二次更新据此得到 B_j。两次选择都同步维护端点，所以不变式继续成立。

循环结束时 $j = n-1$，前缀就是整个数组，因此返回值正确。

6. 边界与复杂度

数组全为负数时，例如 $[-5,-2,-8]$，答案是单个元素 $[-2]$，返回 $(-2,1,1)$。若把最佳值初始化为 0，便会错误地返回一个不允许的空子数组。

长度为 1 时，直接返回唯一元素。空数组不满足输入要求，代码会抛出异常。若题目改为允许空子数组，可以将答案下限设为 0，但也要相应修改状态定义与证明，和本题的非空约定区分开。

每个元素只更新固定数量的变量，因此时间为 $\Theta(n)$，辅助空间为 $O(1)$。上一课的分治方法花费 $\Theta(n \log n)$；这里用一个最佳结尾状态，保存下一次扩展所需的跨边界信息。

十二、作业 2：用最小堆合并 k 个有序表

1. 题意与关键观察

来源：课件第 33 页，CLRS 练习 **6.5-9**。

给定 k 个已经分别按非降序排列的表，总元素数为 n。将它们合成一个有序表，要求时间 $O(n \lg k)$。提示是使用最小堆进行多路归并。

每张有序表中，第一个尚未输出的元素就是剩余部分的最小值。因此，只需比较各非空表的当前表头，就能找到下一个输出值。

堆中保存至多 k 个候选值。取出一个表头后，只需从它所属的表补入后继。

2. 堆节点保存什么

对于数组形式的输入表，堆节点设为：

$$(\text{value}, \text{list_id}, \text{index}).$$

第一个字段决定数值大小；后两个字段记录来源，便于找到后继。值相等时，Python 元组继续比较表号和下标，因此可以正常处理重复值，也不需要比较不可排序的自定义节点。

```
def merge_sorted_lists(lists):
    heap = [
        (row[0], list_id, 0)
        for list_id, row in enumerate(lists)
        if row
    ]
    heapq.heapify(heap)
    result = []
    while heap:
        value, list_id, index = heapq.heappop(heap)
        result.append(value)
        next_index = index + 1
        if next_index < len(lists[list_id]):
            next_value = lists[list_id][next_index]
            heapq.heappush(heap, (next_value, list_id, next_index))
    return result
```

若输入是链表，则每个堆节点保存当前节点及表号，弹出后沿 `next` 指针加入后继，算法结构不变。

3. 一次合并的全过程

设：

$$L_0 = [1, 4, 7], \quad L_1 = [1, 3, 8], \quad L_2 = [2, 6].$$

初始候选为 $(1, 0, 0), (1, 1, 0), (2, 2, 0)$。下表为便于阅读，按数值和表号列出候选集合；实际堆数组的排列可能不同：

第几次取出	输出值与来源	加入的后继	取出后尚在堆中的候选值
1	1，来自 L_0	4	$1(L_1), 2(L_2), 4(L_0)$
2	1，来自 L_1	3	$2(L_2), 3(L_1), 4(L_0)$
3	2，来自 L_2	6	$3(L_1), 4(L_0), 6(L_2)$
4	3，来自 L_1	8	$4(L_0), 6(L_2), 8(L_1)$

第几次取出	输出值与来源	加入的后继	取出后尚在堆中的候选值
5	4，来自 L_0	7	$6(L_2), 7(L_0), 8(L_1)$
6	6，来自 L_2	无	$7(L_0), 8(L_1)$
7	7，来自 L_0	无	$8(L_1)$
8	8，来自 L_1	无	空

最终结果是 $[1, 1, 2, 3, 4, 6, 7, 8]$。每个输入元素进入输出一次；表耗尽后不再向堆中放入候选。

4. 正确性证明

循环开始时保持：

1. 对每个仍有元素未输出的表，堆中恰有它的第一个未输出元素。
2. 已输出部分有序，并且不大于任何尚未输出的元素。

初始化把各非空表的第一个元素加入堆，输出为空，成立。

各表内部有序，所以未输出元素都不小于本表的当前表头。堆顶又是所有当前表头中的最小值，也就是全体未输出元素的最小值。将它接在输出末尾，仍保持有序；再加入同一张表的后继，便恢复了每张未耗尽的表恰有一个候选值的不变式。

每轮恰好输出一个元素，循环共进行 n 轮。堆空时，各表都已经耗尽，所以输出有序并且包含全部输入元素。

5. 时间、空间与特殊情况

若 k 个表都非空且 $k \geq 2$，初始化用 $O(k)$ 时间，`heapify` 也是 $O(k)$；共弹出 n 次，每次至多再插入一次，堆大小始终不超过 k，因此：

$$T(n, k) = O(k + n \lg k) = O(n \lg k),$$

最后一步使用了 $k \leq n$。堆的辅助空间为 $O(k)$，若把返回的输出也计入存储，则为 $O(n + k)$。

如果允许大量空表，扫描这 k 个输入表头本身就需 $O(k)$。设其中非空表数为 k'，更完整的表达是：

$$O\big(k + n \log \max(2, k')\big).$$

$k = 0$ 时返回空表；只有一个非空表时，复制输出仍需 $\Theta(n)$，即使 $\log 1 = 0$，总时间也不为零。题目常用的 $O(n \lg k)$ 写法默认至少合并两张非空表。

逐个扫描所有当前表头，每输出一个元素需要 $O(k)$，总计 $O(nk)$；最小堆把这一步的维护成本降到了 $O(\log k)$。

十三、作业 3：截断快速排序，再对全数组做插入排序

1. 题意中的执行顺序

来源：课件第 34 页，CLRS 练习 **7.4-5**。

修改快速排序：调用到一个 **少于 k 个元素**的子数组时，直接返回，让这个小区间暂时无序。整个快速排序阶段结束后，再对 **整个数组**执行一次插入排序。

要求证明期望时间为：

$$\boxed{O(nk + n\lg(n/k))},$$

并讨论理论上、实践中如何选择 k。

分析采用互异元素和均匀随机枢轴，设 $2 \le k \le n$。截止条件严格为 $m < k$，长度恰好为 k 的子数组仍要划分。

2. 完整算法

```
def hybrid_sort(a, k, rng=None):
    if k < 2:
        raise ValueError("cutoff k must be at least 2")
    if rng is None:
        rng = Random()

    def partial_quicksort(p, r):
        if r - p + 1 < k:
            return
        q = randomized_partition(a, p, r, rng)
        partial_quicksort(p, q - 1)
        partial_quicksort(q + 1, r)

    partial_quicksort(0, len(a) - 1)

    for j in range(1, len(a)):
        key = a[j]
        i = j - 1
        while i >= 0 and a[i] > key:
            a[i + 1] = a[i]
            i -= 1
        a[i + 1] = key
    return a
```

代码复用前面的 `randomized_partition`，可接受空数组及 $k > n$ 的情况；后者会直接进入全数组插入排序。题目中含 $\lg(n/k)$ 的复杂度分析，采用的范围为 $2 \le k \le n$。

前面的划分只交换元素，最后的插入排序又会排好整个数组，因此算法正确。分析运行时间时，需要先说明第一阶段留下的数组有多接近有序。

3. 未排序小块之间的顺序

递归在小于 k 的区间上停止。这些终止区间内部可以无序，但它们之间的顺序已经由祖先枢轴确定。

任取两个不同的终止区间，考虑把它们首次分开的祖先划分。左侧全部元素小于该枢轴，右侧全部元素大于该枢轴，所以左区间中的每个元素都小于右区间中的每个元素。已经就位的枢轴也不会与其他区间形成逆序。

因此，剩余逆序对只可能出现在同一个未排序小块内部。

以 $k = 4$ 为例，一种合法的第一阶段结果是：

$$[3, 1, 2] \mid 4 \mid [6, 5] \mid 7 \mid [9, 8].$$

其中 4 和 7 是已经固定的枢轴，三个小块的长度分别为 3、2、2，均小于 4。小块内部尚未排序，块间已满足前块所有值小于后块所有值的关系。

4. 插入排序的代价用逆序对计量

标准插入排序插入 key 时，每向右移动一个比它大的前驱元素，就消除一个逆序对。每个原始逆序对恰好引起一次这种移动，加上每轮固定开销，运行时间为：

$$\Theta(n+I),$$

其中 I 是第一阶段结束后的逆序对数。

设未排序小块长度为 $m_1, m_2, \dots, m_s$。这些块互不重叠，满足 $m_i < k$、$\sum_i m_i \le n$。长度为 m_i 的块至多有 $\binom{m_i}{2}$ 个逆序对，所以：

$$I \le \sum_{i=1}^{s} \binom{m_i}{2} \le \frac{k-1}{2} \sum_{i=1}^{s} m_i \le \frac{n(k-1)}{2} = O(nk).$$

因此，最后一次 **全数组**插入排序需要 $O(n+nk) = O(nk)$。全局逆序对数限制了整个扫描过程的移动次数，所以即使不逐块排序，也能得到这个上界。

前面的具体例子中，$[3,1,2]$ 有 2 个逆序对，$[6,5]$ 和 $[9,8]$ 各有 1 个，总计 $I = 4$。完整数组的插入过程只需 4 次右移：

插入的键	被右移的元素	局部结果
1	3	$[1,3,2]$
2	3	$[1,2,3]$
5	6	$[5,6]$
8	9	$[8,9]$

其余插入轮次只进行固定数量的检查。最终数组为 $[1,2,3,4,5,6,7,8,9]$。

5. 截断快速排序的期望代价

若每次恰好均分，递归约有 $\lg(n/k)$ 层，每层处理 $O(n)$ 个元素，于是得到 $O(n \lg(n/k))$。这是均衡划分下的直观分析。要覆盖随机快速排序中的不均衡划分，还需计算达到截止规模之前的期望工作量。

将长度为 m 的一次划分所需的 $\Theta(m)$ 工作，平均分摊给区间内的每个元素。固定任意原始元素 x，跟踪它在尚需划分的子问题中经历的区间规模。

称某次枢轴为「好枢轴」，若它的 1-based 秩 r 满足：

$$\lfloor m/4 \rfloor + 1 \le r \le \lceil 3m/4 \rceil.$$

这些秩至少占一半，所以每次条件概率至少为 $1/2$。取到好枢轴后，若 x 本身就是枢轴，它退出递归；否则它进入的那一侧规模至多为 $3m/4$，取整不会破坏这个上界。

从规模 n 缩小到小于 k，至多需要：

$$g = \left\lfloor \log_{4/3} \frac{n}{k} \right\rfloor + 1$$

次这样的成功缩小。即使中间出现坏划分，每等到一次好枢轴，期望尝试次数也至多为 2：给定此前任意划分历史，下一次选中好枢轴的条件概率始终至少为 $1/2$。如果元素提前成为枢轴，跟踪还会更早结束。

因此，一个元素参与划分的期望次数为：

$$O\left(1+\log\frac{n}{k}\right).$$

再对 n 个元素求和，利用期望的线性性：

$$\mathbb{E}[\text{第一阶段代价}] = O\left(n + n\log\frac{n}{k}\right).$$

不同元素的递归路径可以相互依赖。上述求和只用到每步随机枢轴的条件概率，以及期望的线性性，无须假设路径独立或每层均衡。

6. 合并两阶段的界

加入插入排序以及递归调用的线性级开销：

$$\mathbb{E}[T(n,k)] = O\left(n + n\lg\frac{n}{k} + nk\right) = \boxed{O\left(nk + n\lg\frac{n}{k}\right)}.$$

最后一步利用 $k \geq 2$，将 $O(n)$ 吸收到 $O(nk)$ 中。特别地，$k = n$ 时虽然 $\lg(n/k) = 0$，第一阶段仍需一次线性划分。这部分由总界中的 nk 项覆盖，单独计算第一阶段时仍要保留。

这个界是随机枢轴下的期望结论。若连续选到很差的枢轴，第一阶段仍可能花费二次时间。输入含大量重复值、继续采用二路 Lomuto 划分时，互异元素下的期望证明也不适用。

7. 理论上怎样选择 k

若目标是保持最优的渐近量级，取一个固定常数 k 即可：

$$nk = O(n), \qquad n\lg(n/k) = \Theta(n\log n).$$

取 $k = O(\log n)$，总界也保持在 $O(n\log n)$，与固定常数的渐近阶相同；取 $k = n$ 时，上界则为 $O(n^2)$。

若进一步比较两阶段的成本常数，可以使用模型：

$$F(k) = a\,n\ln(n/k) + b\,nk, \qquad a, b > 0.$$

把 k 暂当作连续变量，则：

$$F'(k) = -\frac{an}{k} + bn = 0 \quad\Longrightarrow\quad k = \frac{a}{b}.$$

这个模型给出的理想阈值为常数规模，具体取决于两阶段的成本系数。它不能确定一个适用于所有机器和程序的整数；渐近大 O 隐藏了这些系数，实际最优阈值仍需测量。

8. 实践中怎样选择 k

固定实现与硬件，在同一批数据上比较几个小阈值。测试应覆盖随机、近乎有序、逆序及重复值较多的输入，并通过重复运行观察稳定耗时，减少随机枢轴波动对判断的干扰。

短数组上，插入排序的循环紧凑，局部访问连续；快速排序则要承担枢轴选择、划分和递归管理的开销。阈值决定何时切换两种算法，合适的取值还受元素大小、比较函数成本和缓存行为影响，需要通过测量确定。

9. 证明中需要保留的条件

- 截止条件是「少于 k」，长度等于 k 时仍划分。
- 插入排序执行于整个数组；它的代价由全局剩余逆序对数控制。
- 小块内部无序，但块间顺序已经确定，否则 $I = O(nk)$ 无法成立。
- 期望划分代价需要概率论证；理想均分的递归树只能提供直观解释。
- 小区间划分使用 $O(1)$ 辅助空间，不等于整段递归算法也只用 $O(1)$ 空间。

十四、复习题

1. 多数元素的 `count` 为什么不是实际出现次数？去掉存在性保证后应补哪一步？
2. Lomuto 划分的四个区域是什么？为什么最后交换后还不能说左右两侧已经有序？
3. 为什么互异元素下，枢轴秩的均匀分布足以分析任意固定输入？
4. 快速排序的期望递推为什么能通过相邻规模作差消掉求和？
5. 递归层数为对数，为什么 Quickselect 的时间仍为线性？
6. 自底向上建堆为什么是线性时间？查看堆顶与取出堆顶的成本有何区别？
7. $O(n + K)$ 为什么未必是关于二进制编码长度的多项式？
8. 比较排序的叶子为什么至少有 $n!$ 个？下界为什么允许某些输入在线性时间内完成？
9. 最大子数组的两个状态各自覆盖哪些候选区间？
10. 多路归并为什么只需为每张表保留一个候选元素？
11. 混合排序第一阶段之后，为什么逆序对只能留在小块内部？
12. 怎样在允许随机坏划分的情况下，证明第一阶段的期望代价？

附•资料下载

主要依据为课程课件《04-advanced_sorting》、本次课堂转写《算法课排序复杂度与决策树证明》，以及当天课堂摘要。三道作业的编号、输入要求和目标复杂度均以课件末尾题目为准。

- 课堂算法与三道作业的 Python 实现
- 算法与边界条件验证脚本
- 理论比较次数图的生成脚本

单页手写提纲用于快速回顾课堂主线，详细推导和作业解答见本文：

单页手写速览另存为《Day4-单页手写速览.pdf》，可与本册配合阅读。