《计算之魂》共读-个人学习讲义

以往习惯了Typora笔记，本次尝试练习 LaTeX 与费曼方法，理清概念、自我教学，按共读计划更新个人讲义。提醒自己：学习切忌闭门造车。

补充：

【致谢】我非常意外，原本小打小闹的自我要求与自我教学内容，竟会得到许多共读伙伴的反响，非常感谢小伙伴们的反馈与鼓舞，让我在共读的过程中，又多了一份自信与激情。更令人意外的是，这份个人讲义竟能够同步开源到Datawhale 《计算之魂》共读项目里，这是我作为一名初学者莫大的荣幸，也给予了我更为持续的动力，非常感谢Datawhale 社区给到的这个机会。

【关于勘误】既然存在进一步传播的可能，我必须要对自己的内容负责。这里记录了我的思考过程，尽管我已多次核对，但探索学习的过程难免有误，如若得到反馈，那将是我的荣幸。本人邮箱 634798049@qq.com

【关于更新】后续随共读活动或个人阅读进度而更新，新版本将同步更新到以下链接：

https://rf8j0i8s4f.feishu.cn/file/boxcn1foj2Y0pHLpD1NCbEJtR7b

by CELFS/陈晓宇

2022年9月13日

Contents

0　计算的本质——从机械到电子

0.1　什么是计算机

概要

0.2　机械计算机、布尔代数和开关电路

概要

0.3　图灵机：计算的本质是机械运动

概要

0.4　人工智能的极限

概要

直播回顾　（20220825）

《计算之魂》共读活动直播课01（主讲人：高博），我的几点触动，总结如下：

- （1）计算机实际上在做**分类与组合**的工作，这也是当前讨论的相关计算智能的基础，涉及组合数学的内容；
- （2）**图论**非常重要，可以作为衡量一个人的计算机能力的指标之一，因为这体现了编程思维从线性的，过渡到**非线性**的；
- （3）正确对待书中的习题、思考题——目的是要掌握题目背后的思想、算法的精髓，重在思维的过程，而非题目的结果——“要**真正的掌握**”；
- （4）读到后面，要适当**回顾前面**的内容，或许会有更深入的、不一样的理解。

分享感悟后，共读群友对（2）的补充：

- 计算机中**争夺打印和计算资源**本质上也是一种图算法，就是**共圈问题**；
- 还有一些**底层大规模并行**，如何设计**处理器架构**也是图论问题；
- 图论：指**广义**上的图论分支，算法中的图论只是一部分，还有一些**子图问题**等等都有很强的实际应用。

组合数学好书补充：《组合数学》陈景润（待购），《组合数学》冯荣权（已购）

1　毫厘之差——大O概念

1.1　算法的规范化和量化度量

概要

- 本节从上世纪40 年代电子计算机的雏形讲起，冯·诺依曼（John von Neumann）等人完善了计算机硬件架构的顶层设计，即基于EDVAC 设计草案完善了“冯·诺依曼体系结构”——沿用至今，同时也标志着**硬件**设计有了一定的规范标准。
- 随着战争的结束，经济复苏，计算机的这项伟大的发明（也是战争的功臣），逐渐被人们认识，计算机**从军事应用一步步地推广到商业应用**。
- 在这个背景下，或者说在不同的经济主体驱动下，市场需求对计算机的**软件**层面提出了要求：可重复性、易用性——而实现这种要求的前提是：规范化软件设计。
- 自然地，高德纳（Donald Ervin Knuth）成为了**弥补这方面不足**的人，为软件的设计提供了“语法”标准，也提出了衡量“语法”优劣的量纲——“算法分析”一词，携着属于它的“宪法”正式进入计算机领域。
- 总体而言，冯·诺依曼与高德纳，分别奠定了现代计算机硬件与软件的**顶层设计**基础。有了顶层设计，意味着搭好了计算机软硬件的**基本结构**，也意味着给出了软硬件设计的**方向与边界**。日后，科学家们就可以围绕着这些基本结构，沿着**前人的足迹**，构建更宏大的计算机大厦了。

前面概括了算法提出的主要背景，我适当地进行了补充并按照个人的理解进行了复述，其实要挖掘的话，还有好多有趣的事，推荐一本书《计算思维史话》（Tony Hey），该书比较详细地讲述了计算机及其主要领域的发展脉络、历史故事，从布尔代数、巴贝奇（Charles Babbage）的机械式计算机（差分机）讲起，最后用科幻小说中的机器人与人工智能画上了一个颇具“魔幻现实主义”色彩的句号。

思考题 1.1（★☆☆☆☆）

世界上还有什么产品类似于计算机，是软硬件分离的?

我思：

- 最初的理解（认为有误）：汽车、乐器、玩具、书籍、一支笔、手机等可交互的产品。其实进一步想，但凡是“可交互”的产品（如此看来一切的物品，都能称为“软硬件分离”？），都可以在外部找到承担“软件”功能的角色，即使回到计算机的软件，也可以存储在光盘、U盘、硬盘.....等若干外部的媒介中，而对于像“书籍”这样的硬件，其软件部分天然地存在于人类的大脑中。
- 理解的误区：将“媒介”的形式，当成了“软件”本身，其实还有一点很重要“软件要回到产品本身”，因为原本软件的概念就是一层层地从计算机硬件的逻辑里面抽象出来的，所有才能说“分离”。
- 目前的理解：手机、平板等类似的电子产品本质上也是计算机，那么有什么不是计算机，但也属于“软硬件分离”的产品呢?
- – **一段讨论的插曲**：

- 就上述疑问，我在小队群里抛出了我的想法：对于1.1思考题，一开始看着“一颗星”觉得想当然，但是细细想想之后，觉得有点奇怪。
- 傍晚在休息的时候，看着家里中午喝剩下的半瓶天地一号，不禁思考：酒瓶和酒，算不算软硬件分离呢?
- 队友：想到了生命。
- 这个回答具有启发性，我对比自己的疑问，又作了进一步的思考：
  - 可能吧，但怎么解释“分离”呢？如果要从智慧生物“意识”的角度，和躯体的组成之间做一个划分的话。
  - 另外，生命能不能称为“产品”呢?
  - 感觉讨论这些，容易陷入自然语言表述上的误区。不过，这类思考题总归是开放性的，意味着每个人的解读不同，“答案”也可以包罗万象（如果这个是事实，间接也说明了：这类问题很难有所谓的标准答案，而且有时候也很难二元化地给出对与错，就像我前面标注的“认为有误”那样，只不过是我找到了一个判断为“有误”的逻辑自洽的理由罢了）。
- 队友：那就得看产品怎么定义了。
- 还是具有启发性的回答，“定义”一词，进一步引发了我的思考：
  - 计算机可以做“软硬件分离”，有其结构的原因，从计算机中来，又到计算机中去。
  - 如果用生命来类比的话，似乎也能找到一些共性，只是结构层面我们难以“分离”，这里的共性指的是：人有意识，可以用文字等不同的载体表达自己的感受，相当于把大脑中的“软件”，拷贝出来了（当然也可以反映在其行为表现上，留下“活动痕迹”或许也能算是“存在的证明”），但不同的是，这些“痕迹”难以衡量。比如，计算机可以把软件、把文件剪切出来，相当于从自己身上剥离了一部分，但是生命的意识，似乎不存在“剪切”的操作（但DNA 遗传，是否相当于一部分的“剪切”？）。
  - 我感觉这题过于开放了，但如果说题中“类似计算机”就是一个范畴限定的话，那么“产品就是一般意义的工业产品”，“分离也应该是类似计算机中能够根据结构，进行清晰的划分”
  - 而“软件，则有点难界定了，因为范畴基本局限在计算机领域；如果要用‘摸不着’的东西来类比软件，那么范围似乎就太宽泛了”。
  - 而且如果用“摸不着”来界定软件，我前面酒瓶和酒的例子也不对了（如果查询软件的定义，可以知道“无形”是软件的特征之一）。
- 至此，尽管我仍然很好奇，但感觉讨论到上述类似的那种无穷尽的问题就该适可而止了（又或许解铃还须系铃人吧）。

■

这里有个背景想提一下：以前和现在的“硬件”通常都是指那些**看得见摸得着**的，而“软件”则不然，早期的计算机，大多是通过“打孔卡/穿孔卡(punched card)”来运行程序和输入数据的（本质是打孔与否组成的二进制表示，相当于直译成机器码；其实也有存储功能），更没有“操作系统”的概念了，换句话说，当时的“软件”其实是一张张**看得见、摸得着的卡片**（当时还有专门管理卡片的“操作员”）。后面随着冯·诺依曼体系结构的提出，用户对软件需求的复杂化，以及各种编程语言的发展，软件也逐渐成为了我们今天熟知的**“摸不着”**的东西。

无论软件是以何种形式存在，其目的都是想更加方便地控制计算机，去做那些“计算机可以做的事情”。而软件的本质，可以用“信息”一词概括。

就此，尽管读完了1.1 小节的内容，但我们还可以问自己：本节的标题意思，我们理解了吗？此篇行文的结构如何，get到文章大意了吗？为什么要提这样一个开放性的思考题？这番讨论，对理解本章标题“毫厘、大O 概念”有什么帮助？我们如何衡量自己是否理解了呢？

复盘总结

待复盘

引申

- First Draft of a Report on the EDVAC, John Von Neumann（1945），提出“冯·诺依曼体系结构”的草案
- https://en.wikipedia.org/wiki/Punched_card，维基百科-穿孔卡
- 《计算思维史话》Tony Hey, Gyurl Pápay

1.2 大数和数量级的概念

概要

- 上一节介绍了“规范化”的必要性，本讲进一步讨论“如何进行算法的规范化”。
- 开篇借助朗道（Lev Davidovich Laudau）对物理学家的五级分类，引入了“数量级”概念，以衡量同类事物不同级别之间的整体差异。
- 接着，借助自身的一次“搞砸了”的实习经历，讲述了算法效率中的“数量级”对程序运行的长期影响。
- 其次，借助George Gamow 书中例子、日常生活金钱数值例子，提醒我们“人的本能受限于小数字的常识世界，容易忽略大数情况下可能出现的种种意外”，需要重视“运用在大数字的计算机世界”中的算法选择。
- 为了进一步论证算法选择的重要性，鼓励大家以计算机的思维，去思考算法的效率。借助“辩证法”的例子，引入了一个并不能广泛适用于计算机思维的**常识误区**——“以为具体问题具体分析总是正确的”，这将导致计算机给出“模棱两可”的答案，即远离了我们使用计算机解决问题的初衷——远离了确定性。
- 基于以上的讨论——“如何选择合适的算法”成为了一个亟待解决的问题，但若想解决这个问题，**需优先**对算法度量进行严格的规范化。
- 自然地，我们的思路回到了早期，回到了算法度量标准的制定历程——五十多年前，两位科学家Juris Haetmanis 和Richard Stearns 提出了算法复杂度的概念，几年后，由我们熟悉的Knuth 进行了严格的量化——即提出了沿用至今的**“大O ”概念**。
- 最后，以例题的形式，讨论了Knuth 算法分析思想的**几个核心特征：讨论大数、抹除常数、抓“主”变化、量级归类**，近似地衡量算法的性能。
  - 讨论大数：即只考虑理论上数据无穷多的情况；
  - 抹除常数：在数据无穷多的情况下，常数微不足道；
  - 抓“主”变化：在数据无穷多的情况下，只需考虑最主要的影响，进行近似。
  - 量级规律：在数据无穷多的情况下，通过比值结果，可区分不同数量级的算法。

通常，计算机的思想可以追溯到数学的思想。我们讨论“用计算机解决的问题”，要回到计算机设计的初衷——弥补人类在面临大量计算问题时，“易出错、难计算、过于复杂、不便维护”等短板。所以，通常鼓励要用“大数思维”来思考计算机要面对的问题——而处理大数，甚至是理论上“无穷”的情况，最容易想到的数学思想就是“极限”、“无穷大”的相关概念。正如书中讲述的，大O概念在数学上有定义，如若想深入了解，可以找些相关的资料学习（如Knuth 《计算机程序设计艺术（第一卷）》P85）。

尽管“大O”的思想可以回到数学进行理解，但其作为一门工具引入计算机领域，是有**从工程的角度**作出简化的考虑，正如书中提到“一个算法的复杂度由一高一低两部分组成.....后面数量级低的那部分可以直接省略.....这在数学上显然不成立，但是在计算机算法上是被认可的”。

我想起了以往纠结过的一些问题，如今也值得回味“为什么可以‘抹掉’常数”、“为什么只需要讨论‘多项式最高次项’”、到底是算“执行次数”还是算“执行次数的增长率”，为什么有时候“看似计算执行次数，实则在讨论增长趋势”……？

如果想直观地理解“数量级”的差异，可以对几种主要的算法复杂度以及它们的组合，打个表进行对比。

摘录 （P33）

根据数学上对大O概念的定义，如果两个函数$f(N)$和$g(N)$，在N趋近于无穷大时比值只差一个常数，那么它们就被看成同一个数量级的函数。而在计算机科学中相应的算法，也就被认为是具有相同的复杂度。

例题 1.1（★☆☆☆☆）

围棋有多复杂?

我思： 这道例题，实则说明了一个事实：我们以为的数，可能大得超乎我们的想象。例如，比宇宙间基本粒子还要多得多的变化，可以浓缩在一个小小的围棋棋盘当中——存在于能够列明的“可能性”概念当中，这是很有意思的事（如同人类如谜的大脑般浓缩）。我不禁想起了“国王棋盘大米”的故事，也想起了“64片黄金圆盘-汉诺塔”的故事，其实如出一辙。

■

例题 1.2（★★★☆☆）

一句有20 个单词的英语语句可以有多少种组合?

我思： $\binom{100000}{20} \approx (\frac{10^5}{c})^{20} \approx 10^{100}$，即Googol 数。题外话：想起有个故事，谷歌在成立的时候，创始人本来是想写Googol 这个单词的，由于读音太像，当时并没有发现拼写的错误，直到注册成功的时候，才发现写成了Google。

■

这两道例题，似乎都是在探索我们对数字大小的“感觉边界”，从能够理解的个、十、百、千……，逐渐拓展到超越现实世界能够感知的大数，最后以葛立恒数划定了这次探索的边界。

思考题 1.2（★★☆☆☆）

如果一个程序只运行一次，在编写它的时候，你是采用最直观但是效率较低的算法，还是依然寻找复杂度最优的算法?

我思：

- 前提：只运行一次。选择：最直观但效率低VS 复杂度最优
- 我首先想到的不是怎么在上面两者之中做选择，而是“什么程序只运行一次”?
- “只运行一次”，意味着很多本来可以辩证看待、“具体问题具体分析”的条件都受到了限制：例如是否便于维护（强调直观-可读性）、是否需考虑输入数据量的改变（影响效率）……——在确保了算法正确性的情况下，我的理解是：这些统统都不需要考虑。

- 因为这时，无论数据量大或小，选择复杂度最优的算法都是合适的，而选择直观但低效的算法，则条件更苛刻：
  - 情况一：在数据量小的情况下，有些效率低的算法确实有较好的表现，那么对于复杂度最优的算法，这种情况下两者差距将很小（除非你是在打什么比赛竞速，一定得扣那几十毫秒，或者几秒的误差）。
  - 情况二：在数据量大的情况下，假设有某种场景，是对大的数据进行分段处理，而这种情况下，似乎把问题转换成了“情况一”，那么能不能讨论累加效应，从而辩证地分析？
    * 因为程序“只执行一次”，分段了意味着内部处理单元需运行不止一次，这就有点套娃的味道了，看似在解决数据量大的问题，实则是若干批次“小”数据的处理；
    * 但这种情况下，意味着分段处理的所谓“小”数据，也很有可能比前面情况一的数据量大；
    * 否则，所有片段合起来，也只能说是“小”数据，这就跟前提条件矛盾了；
    * 再者，如果我把一个大数据，无限细分，在内部处理单元重复执行次数上的消耗，可能会抵消细分批次使用低效算法带来的效率增益，也就是说：将得不偿失。这种情况下，还不如选择合适的数据块大小，一次过处理呢。
- 这个问题，让我想起了一道题目：
  - 假设你对小兰一无所知，以下哪个选项最可能是对的？
    * （a）小兰是一个摇滚明星，还是一名银行职员
    * （b）小兰是一名文静的银行职员
    * （c）小兰是一名文静且内向的银行职员
    * （d）小兰是一名诚实的银行职员
    * （e）小兰是一名银行职员
    * （f）以上选项都不太可能
- 正是因为上述的选项都有“提供事实”的意思，而“银行职员”涵盖的情况最多，于是，在不完全否定所有“事实”的可能性前提下，选择（e）。而对于本思考题，基于我上述列举的一些事实，我作出的选择也同样是考虑到谁的泛用性更广。
- 那么，有没有什么复杂度最优的算法，在数据量低的情况下，效率是非常低的呢？或许有吧。但是无论选择怎样的算法，本该花的时间，一点也少不了。比如我们解决一个排序问题，最快也要$O(n \log n)$ 的复杂度，这是排序问题本身决定的，也就给出了这类问题的边界。

■

复盘总结

待复盘

1.3 怎样寻找最好的算法

概要

- 本节借助一道例题，用四种方法详细地展开分析，讲述了计算复杂度的差异来源。
- （总和最大区间问题/股票最长有效增长期问题）**四种方法**
  - **方法1**：三重循环，排列组合的方法，$O(K^3)$ 复杂度；
  - **方法2**：两重循环，在已求和的基础上，再做一次加法，而非再一次循环，$O(K^2)$ 复杂度；
  - **方法3**：分治（Divide-and-Conquer）算法，将序列一分为二，判断两个子序列的总和最大区间之间是否有隔断，从而进行分类讨论：【待缕清完善】
    - 无隔断，则比较$[p, \frac{K}{2}], [\frac{K}{2}+1, q], [p, q]$ 三种区间；
    - 有隔断，则比较$[p_1, q_1], [p_2, q_2], [p_1, q_2]$ 三种区间；
    - $O(K \log K)$ 复杂度；
  - **方法4**：正、反两遍扫描（**可优化为正向一遍**）
    - 部分实现的方法：
      - 正向计算一遍最大区间和，以确定右区间边界；
      - 反向计算一遍最大区间和，以确定左区间边界；
      - 依据左右区间边界，求得全局最大区间和。
    - 上述方法存在问题：
      - 连续累加和至某个元素，可能出现累加和跌到零以下，并一直在零以下，导致右边界在左边界的左边，将无法据此求得最大区间和。（P40）
    - 方法完善：
    -
    -
- **三个要点：边界认知、检查冗余、逆向思维**。
  - 边界认知：确定问题是否有**(上)下界**，界为多少，可为最优解法的探索提供方向；
  - 检查冗余：优化算法最常用的就是检查其中的**无用功**（即冗余的操作）；
  - 逆向思维：换个角度，甚至反过来思考问题，从而寻找突破口。

本小节的内容较之前“借史传道”的两小节要硬核不少，一道例题尽显计算思维的百态，尽管是块硬骨头，但非常值得花时间去仔细分析。我想起以前看过的一个关于数学家林群院士的讲座视频，林院士大意是说：“微积分实际上是**‘假传万卷书，真传一例子’**。”这里的“真”与“假”，指的不是“True or False”或者“Real or Fake”，而是表面上借助万卷书来阐述微积分的本质、性质，看似宏大而繁杂的内容，其实都可以浓缩到某个具有广泛意义的例子上面。

说到这里，我又想起了3Blue1Brown《微积分的本质》系列视频，第一课当中引用了大卫·希尔伯特（David Hilbert）的语录：**“The art of doing mathematics is finding that special case that contains**

all the germs of generality."，翻译为"数学之道在于找出一个这样的特例，它包含普遍原则的全部萌芽。"如此回想，竟觉有异曲同工之妙。

读《计算之魂》这部分的时候，我不由自主地屡次想起前面那番话，计算思维有很多种，有时候也看似非常宏大而繁杂，但通过"例题1.3"这样的例子，**掰开、揉碎又重新合成，从简单情况到复杂情况的不断缕清，从一个表层的认知，逐渐找到深入理解的窗口**，确实算得上是一次"假传万道，真传一例"的体验。

正如吴军老师书中讲到"这是一个很好的练习题，它不仅可以帮助大家理解不同算法在复杂度上的差异，而且可以让大家不断深入思考和寻找更好的答案并且想清楚各种细节.....这道例题则能够帮助大家**领悟计算机科学的精髓**。"

摘录 （P44）

人通常是喜欢从前到后顺着想问题，不喜欢反过来从后往前思考；喜欢做加法、做乘法，而不喜欢做减法、做除法；喜欢从小到大看、从下往上归纳，而不喜欢从大往小看、从上往下演绎。有些很简单的问题，正向思维难以找到答案，而逆向思维却马上迎刃而解。人的思维很多时候和计算机科学应有的思维是矛盾的，要成为一流的计算机科学家或工程师，需要有意识地改变自己的思维方式，突破常规。

前与后、加与减、乘与除、小与大、下与上、正与反。

例题 1.3（★★★☆☆）

总和最大区间问题：给定一个实数序列，设计一个最有效的算法，找到一个总和最大的区间。

例如序列：1.5, -12.3, 3.2, -5.5, 23.2, 3.2, -1.4, -12.2, 34.2, 5.4, -7.8, 1.1, -4.9

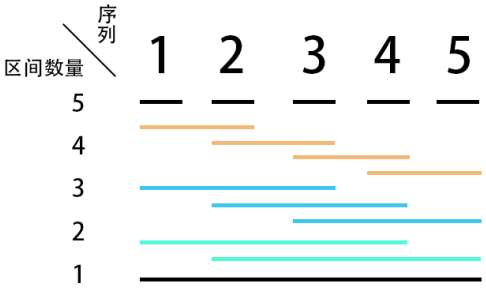
一段讨论引发的勘误插曲：针对书本P35 倒数第三行"计算$S(p,q)$ 平均要做$K/4$ 次加法"这句话，看到共读群里有同学提出疑问与想法，究竟其中的是$K/4$，还是$K/3$? 这个问题引起了我的兴趣，遂尝试证明。问题整理后如下所示：

假设序列有K 个数，依次是$a_1, a_2, a_3, \ldots, a_K$ 。假定区间起始的数字序号为p，结束的数字序号为q，这些数字的总和为$S(p,q)$ ，则$S(p,q) = a_p + a_{p+1} + \ldots + a_q$。$p$ 可以从1 一直到K，q 可以从p 一直到K，这是两重循环了，因此区间一头一尾的组合有$O(K^2)$ 种。在每一种组合中，计算$S(p,q)$ 平均要做的加法次数?

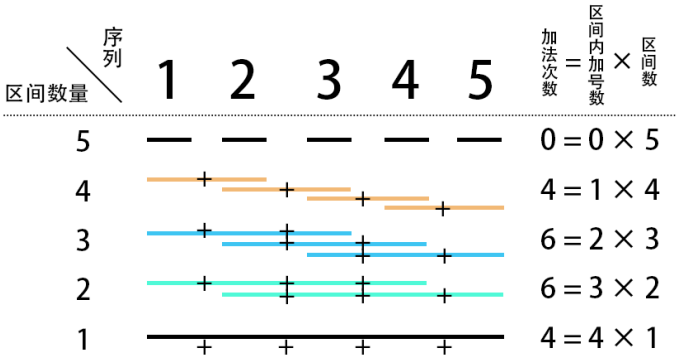
我思： 需计算加法总次数、数字总和的组合数、平均加法次数.

I. 先考虑$K = 5$ 时：

①计算数字总和的组合数, 理解的关键是"给定的数组不可改变顺序", 设区间长度为l：共有5 种不同长度的区间组合情况：$l = K, K-1, K-2, K-3, K-4$, 即$l = 5, 4, 3, 2, 1$, 区间组合数（区间总数）$C$为：$C = 5+4+3+2+1 = 15$.



②计算加法总次数T, 总加法次数如下图所示



即:

$$T = 0 \times 5 + 1 \times 4 + 2 \times 3 + 3 \times 2 + 4 \times 1 = 20$$

不难发现: 式子具有对称性. 因此, 上式可化简为:

$$T = 2 \times (0 \times 5 + 1 \times 4 + 2 \times 3) = 20$$

考虑到序列项数的奇偶数, 其对称中心会有所改变, 可考虑当$K = 6$ 时, 将有如下公式:

$$T = 2 \times (0 \times 6 + 1 \times 5 + 2 \times 4) + 3 \times 3 = 35$$

③计算$S(p, q)$ 平均要做的加法次数, 设为m, 则有:

$$m = \frac{T}{C} = \frac{20}{15} \approx 1.33$$

II. 将情况推广至K 个时:

①计算数字总和的组合数, 即当$l = 1, 2, 3, \ldots, K$ 时, 区间长度的组合数C 可通过等差数列求和公式得: $C = \frac{(1+K)K}{2}$. 这也是书中说"区间一头一尾的组合有$O(K^2)$ 种"的原因, 采用了"大O" 近似.

②计算加法总次数T , 利用数学归纳, 不难得出:

$$T = \begin{cases} 2 \times [0 \times K + 1 \times (K-1) + 2 \times (K-2) + \ldots + (\frac{K-1}{2})(K - \frac{K-1}{2})], & K = 1, 3, 5, 7, \ldots \\ 2 \times [0 \times K + 1 \times (K-1) + 2 \times (K-2) + \ldots + (\frac{K-2}{2})(K - \frac{K-2}{2})] + \frac{K}{2} \times \frac{K}{2}, & K = 2, 4, 6, 8, \ldots \end{cases}$$

当$K = 1, 3, 5, 7, \ldots$ 时, 记奇项数序列的加法总次数为T_{odd}, 下列公式的第一项方括号内（除首项0 外）共

有$\frac{K-1}{2}$ 项, 化简可得:

$$\begin{aligned}
T_{odd} &= 2\times[0\times K+1\times(K-1)+2\times(K-2)+\ldots+(\frac{K-1}{2})(K-\frac{K-1}{2})]\\
&= 2\times[K-1^1+2K-2^2+3K-3^2+\ldots+(\frac{K-1}{2})K-(\frac{K-1}{2})(\frac{K-1}{2})]\\
&= 2\times[(1+2+3+\ldots+(\frac{K-1}{2}))K-(1+2^2+3^2+\ldots+(\frac{K-1}{2})(\frac{K-1}{2}))]\\
&= 2\times[\frac{(K+1)(K-1)}{8}K-\frac{1}{6}\times(\frac{K-1}{2})(\frac{K-1}{2}+1)(2\times\frac{K-1}{2}+1)]\\
&= 2\times[\frac{(K+1)(K-1)}{8}K-\frac{1}{6}\times\frac{(K+1)(K-1)}{4}K]\\
&= \frac{(K+1)(K-1)K}{4}-\frac{1}{3}\times\frac{(K+1)(K-1)K}{4}\\
&= \frac{(K+1)(K-1)K}{6}
\end{aligned}$$

当$K=2,4,6,8,\ldots$ 时, 记偶项数序列的加法总次数为T_{even}, 下列公式的第一项方括号内（除首项0 外）共有$\frac{K-2}{2}$ 项, 化简可得:

$$\begin{aligned}
T_{even} &= 2\times[0\times K+1\times(K-1)+2\times(K-2)+\ldots+(\frac{K-2}{2})(K-\frac{K-2}{2})]+\frac{K}{2}\times\frac{K}{2}\\
&= 2\times[K-1^1+2K-2^2+3K-3^2+\ldots+(\frac{K-2}{2})K-(\frac{K-2}{2})(\frac{K-2}{2})]+\frac{K}{2}\times\frac{K}{2}\\
&= 2\times[(1+2+3+\ldots+(\frac{K-2}{2}))K-(1+2^2+3^2+\ldots+(\frac{K-2}{2})(\frac{K-2}{2}))]+\frac{K}{2}\times\frac{K}{2}\\
&= 2\times[\frac{K(K-2)}{8}K-(1+2^2+3^2+\ldots+(\frac{K-2}{2})(\frac{K-2}{2}))]+\frac{K}{2}\times\frac{K}{2}\\
&= 2\times[\frac{K(K-2)}{8}K-\frac{1}{6}\times\frac{K(K-2)}{4}(K-1)]+\frac{K}{2}\times\frac{K}{2}\\
&= \frac{K(K-2)}{4}K-\frac{1}{3}\times\frac{K(K-2)}{4}(K-1)+\frac{K}{2}\times\frac{K}{2}\\
&= \frac{(K-2)K^2}{4}-\frac{1}{3}\times\frac{(K-2)K^2}{4}+\frac{1}{3}\times\frac{(K-2)K}{4}+\frac{K}{2}\times\frac{K}{2}\\
&= \frac{(K-2)K^2}{6}+\frac{(2K-1)K}{6}\\
&= \frac{(K^2-1)K}{6}\\
&= \frac{(K+1)(K-1)K}{6}
\end{aligned}$$

可得$T=T_{odd}=T_{even}$

③计算$S(p,q)$ 平均要做的加法次数, 设为m, 则有:

$$m=\frac{T}{C}=\frac{\frac{(K+1)(K-1)K}{6}}{\frac{(1+K)K}{2}}=\frac{K-1}{3}$$

因此，书本P35 倒数第三行“计算$S(p,q)$ 平均要做$K/4$ 次加法” 中的$K/4$，应为$K/3$.

■

为了验证这个结论，尝试用代码跑了一遍（PS：在LaTeX中插入代码不太友好，尝试过使用“minted” 和“lstlisting” 语法，效果都不太理想，而且复制文本时忽略行号弄了许久也没实现，最后只能删掉了行号。另外就是代码缩进，尽管官方提供了一些选项显示缩进的空格，但要么就是乱码难处理，要么就还是不能带空格

复制。尽管最后选择了显示更友好的“minted”，但还不懂如何将一长串的参数封装到自定义环境中。这方面还是Typora 那类好用啊，后面有时间得再看看如何操作更合适）。这里贴出来的代码，暂时也就只能看看了。

```
# 讨论当 K = 1000 时，计算 S(p, q) 的平均次数 average_times
# k_list = [3, 10, 100, 1000, 10000, 100000]
# k_list = [5, 6, 10]
k_list = [times for times in range(1, 10001)]

for K in k_list:                                    # 一个 K 对应一轮
    i, j = 1, K - 1
    temp = 0                                        # 存储每一轮的加法总次数（公式）中的一部分
    if K % 2 != 0:                                  # 根据奇偶项数，分别处理中间项，作为循环结束条件
        mid = (K - 1) / 2
    else:
        mid = (K - 2) / 2

    while i <= mid:
        temp = temp + i * j
        i += 1
        j -= 1
    if K % 2 != 0:
        add_times = 2 * temp                        # 奇数项，计算每一轮的加法总次数
    else:
        add_times = 2 * temp + (K / 2) ** 2         # 偶数项，计算每一轮的加法总次数

    combine_times = (1 + K) * K / 2                 # 每一轮的区间组合数（区间总数），等差数列求和
    average_times = add_times / combine_times       # 每一轮的平均加法次数

    print(f'【{K = }】')
    print(f'{combine_times = :<{10}}   {add_times = :<{15}}     {average_times = :<{10}.5}    {K / 3 = :<{10}.5}
    ↪  {K / 4 = :<{10}}')
```

```
【K = 9992】
combine_times = 49925028.0   add_times = 166266984916.0     average_times = 3330.3     K / 3 = 3330.7     K / 4 = 2498.0
【K = 9993】
combine_times = 49935021.0   add_times = 166316909944       average_times = 3330.7     K / 3 = 3331.0     K / 4 = 2498.25
【K = 9994】
combine_times = 49945015.0   add_times = 166366844965.0     average_times = 3331.0     K / 3 = 3331.3     K / 4 = 2498.5
【K = 9995】
combine_times = 49955010.0   add_times = 166416789980       average_times = 3331.3     K / 3 = 3331.7     K / 4 = 2498.75
【K = 9996】
combine_times = 49965006.0   add_times = 166466744990.0     average_times = 3331.7     K / 3 = 3332.0     K / 4 = 2499.0
【K = 9997】
combine_times = 49975003.0   add_times = 166516709996       average_times = 3332.0     K / 3 = 3332.3     K / 4 = 2499.25
【K = 9998】
combine_times = 49985001.0   add_times = 166566684999.0     average_times = 3332.3     K / 3 = 3332.7     K / 4 = 2499.5
【K = 9999】
combine_times = 49995000.0   add_times = 166616670000       average_times = 3332.7     K / 3 = 3333.0     K / 4 = 2499.75
【K = 10000】
combine_times = 50005000.0   add_times = 166666665000.0     average_times = 3333.0     K / 3 = 3333.3     K / 4 = 2500.0
```

下面开始分析四种方法。

【测试用例】

```
# 13 个元素 测试中间情况【P35、P39】
# section = [1.5, -12.3, 3.2, -5.5, 23.2, 3.2, -1.4, -12.2, 34.2, 5.4, -7.8, 1.1, -4.9]
# 13 个元素 修改负值【P40】
section = [1.5, -12.3, 3.2, -5.5, 23.2, 3.2, -1.4, -62.2, 44.2, 5.4, -7.8, 1.1, -4.9]
# 14 个元素 测试右边界
# section = [1.5, -12.3, 3.2, -5.5, 23.2, 3.2, -1.4, -12.2, 34.2, 5.4, -7.8, 1.1, -4.9, 1000]
# 14 个元素 测试左边界
# section = [1000, 1.5, -12.3, 3.2, -5.5, 23.2, 3.2, -1.4, -12.2, 34.2, 5.4, -7.8, 1.1, -4.9]
# 测试全负数序列
# section = [-1, -2, -3, -4, -5]
```

【方法1】

这里的边界条件处理得不是很好。

```
# 方法 1， 三重循环 O(K^3) 优化
max_sum, left, right = 0, 0, 0

for p in range(len(section)):                # 每轮循环固定一个起点
    for q in range(p, len(section) + 1):     # 区间终点
        temp_sum = 0
        for num in range(p, q):              # 区间内元素求和
            temp_sum += section[num]         # 此处便是重复计算的部分
            if temp_sum > max_sum:
                max_sum = temp_sum
                left, right = p, q - 1

    if left == right:                        # 处理全序列小于零的情况
        max_sum = section[left]

print(f'最大区间索引值为[{left}, {right}]，序列为{section[left:right + 1]}，最大区间和为{max_sum}')
```

【方法2】

这里方法比较清晰，边界条件也比较统一。

```
# 方法 2 ，两重循环 O(K^2)
max_sum, temp_sum, left, right = 0, 0, 0, 0

for p in range(len(section)):            # 每轮固定起点，计算在该起点下，不同长度区间的最大值
    for q in range(p, len(section)):
        temp_sum += section[q]
        if temp_sum > max_sum:
            max_sum = temp_sum           # 符合条件则更新最大值
            left, right = p, q
```

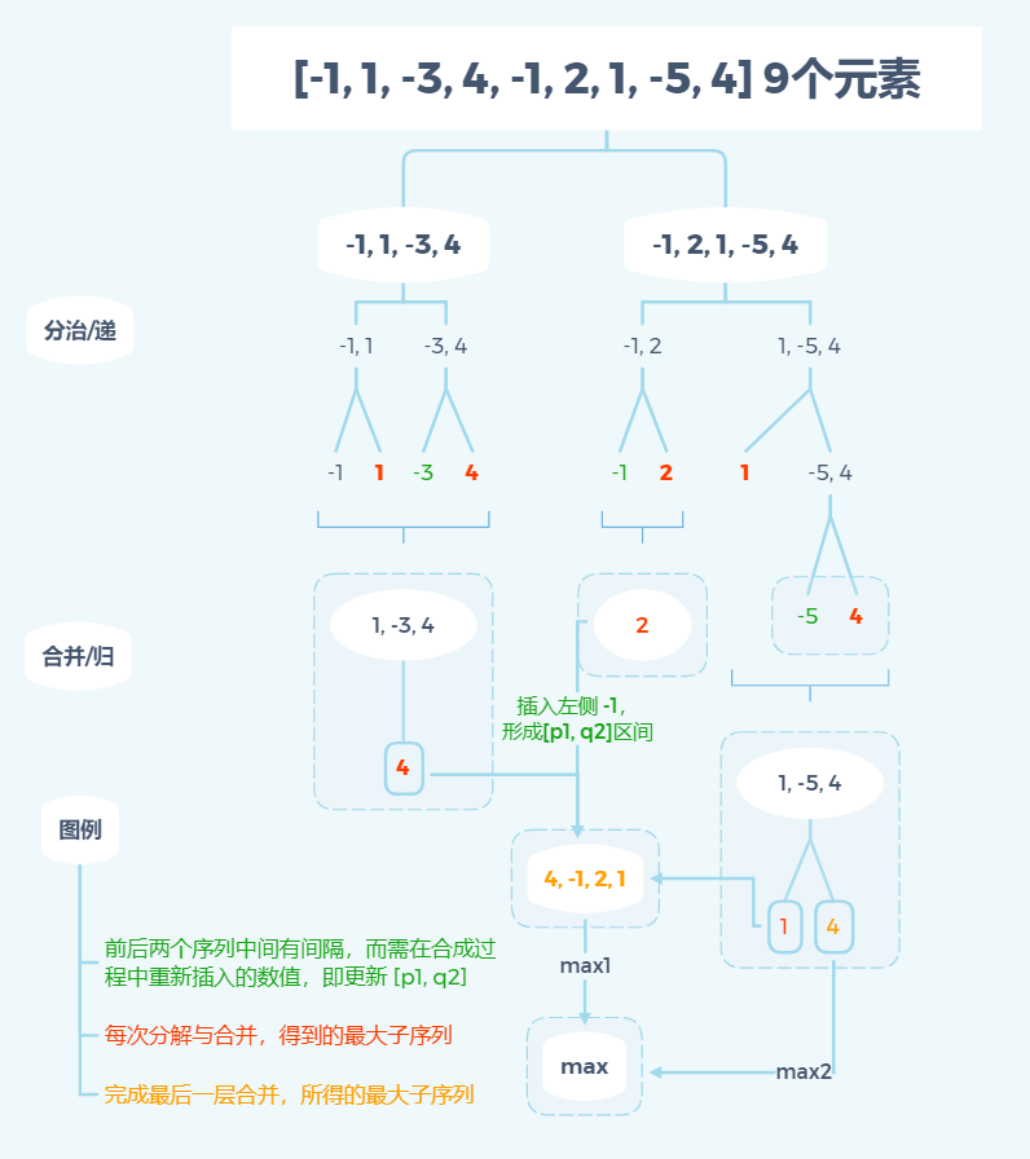
```
    temp_sum = 0                        # 新的区间起点，重新计算区间和

    if left == right:                   # 处理全序列小于零的情况
        max_sum = section[left]

print(f'最大区间索引值为[{left}, {right}]，序列为{section[left:right + 1]}，最大区间和为{max_sum}')
```

【方法3】【代码暂略】

我对方法3 的细节依然不清晰，尤其是合并的过程。但能确定的是，要先分治，后合并——三种形式的区间$[p_1, q_1], [p_2, q_2], [p1, q2]$ 在不断迭代，当完成最后的合并过程，最大区间将为：

$$max = max\{[p_1, q_1], [p_2, q_2], [p1, q2]\}$$

某天晚上，共读群里对第三个区间$[p1, q2]$ 面临的“中间间隔情况” 展开了讨论，尽管最后的讨论结果依然没有缕清其中的细节，但也是具有启发性的。共读群友列举了一个序列（改自力扣053），我觉得该特例很不错，于是在尝试理解方法3 的过程中，就该特例用树状图进行了分析，以期更好地理解分治的思想。



画完图后，我才发现自己当时有一个**理解误区：分析的时候只考虑了一层的分解与合并，没有全局考虑迭代之后的结果**。而且，这个图也间接证实了第三种区间$[p1, q2]$ 是可以获得最大区间和的。

因为**分治**的步骤对整个序列一分二、二分四、四分八.....直到一个区间只有一个元素（这个划分的过程是树状的），所以$[p_1, q_1], [p_2, q_2]$ 中的p, q 值是不断变化的，即区间不断发生改变（直到每个分支都到达了叶子节点）。

接着，就是一层层往回**合并**，如果中间有间隔，其中的判断条件就需要进一步考虑第三种区间，即$[p1, q2]$。

```
# 方法 3 ，分治算法【略，待缕清】
section = [-1, 1, -3, 4, -1, 2, 1, -5, 4]
```

【方法4 】【部分实现】

本代码依据书本（P38-39）的步骤逐一实现，即未处理“右边界反而在左边界的左边”这种情况，造成这个问题的原因是“从一开始累加的总和在遇到某个元素时下跌到零以下，然后一直在零以下（P40）”。

```
def maxSubArray(alist):                    # 方法 4 ，依据书本 P38-39 【部分实现】，即未处理 S(p, q) < 0 的情况
    l, r = 0, 0
    K = len(alist)
    maxf, maxb = 0, 0
    sum_max, sum_forward, sum_back = 0, 0, 0

    for i in range(K):                     # 步骤 1（P38）
        pass

    for q in range(0, K):                  # 步骤 2，正向计算 maxf，以确定右边界
        sum_forward = sum_forward + alist[q]
        if sum_forward > maxf:
            maxf = sum_forward
            r = q

    for i in range(K - 1, 0, -1):          # 步骤 2，反向计算 maxb，以确定左边界
        sum_back = sum_back + alist[i]
        if sum_back > maxb:
            maxb = sum_back
            l = i

    for i in range(l, r + 1):              # 通过左右边界，求最大区间和
        sum_max += alist[i]

    return sum_max, maxf, maxb, [l + 1, r + 1], alist[l:r + 1]
    # (52.4, 39.3, 40.8, [5, 10], [23.2, 3.2, -1.4, -12.2, 34.2, 5.4]) 与书中数据一致
```

【方法4 】【完整实现】

基于上述的【部分实现】代码，继续对照书本（P40-42）的步骤，完善代码。对于“子序列和从大于零跌到零以下” 的情况，需要多加一步判断。

【方法4 优化——正向一趟扫描】

该方法参考共读群友的实现，本来看代码注释，以为思路跟书本是一样的，结果发现是一个更优解。同为$O(K)$ 复杂度，这种方法仅需要正向一趟扫描，即可求得最大区间和。

我的理解是：该方法将$S(p,q)<0$ 作为更新区间的条件，每当“当前区间和（$sum\_forward$）”出现**小于零的情况**，则重新统计区间和，并移动区间左端点；而当“当前区间和（$sum\_forward$）”出现**减小的情况**，由两个条件语句自动忽略该情况。因此，总能保证$maxf$ 记录的是当前的最大区间和，由此更新的区间范围，自然也是数组的索引位置。

```
def maxSubArray(alist):                    # 这种方法是正向一趟计算，用一个区间框住 S(p, q) 并不断更新
    l, r = 0, 0
    K = len(alist)
    maxf = 0
    sum_forward = 0                        # S(p, q) 当前区间和
    p = 0
    for i in range(K):                     # 步骤 1
        if alist[i] <= 0:
            continue
        else:
            p = i                          # 删除序列头部连续的负数或零，指向第一个正数，将左边界 p 固定在该位置
            break

    for q in range(p, K):                  # 类似步骤 2‘ （P40），但不需要反向计算的过程
        sum_forward = sum_forward + alist[q] # 计算 S(p, q)
        if sum_forward > maxf:
            maxf = sum_forward
            l = p
            r = q
        if sum_forward < 0:                # S(p, q) < 0
            sum_forward = 0                # 重置 S(p, q) 以计算下一段区间和
            p = q + 1                      # 区间左端点移动

    return maxf, [l + 1, r + 1]            # 结果返回最大区间和、序列的区间位置（从 1 算起）
```

思考题 1.3（p45）

Q1. 将例题1.3 的线性复杂度算法写成伪代码。（★★☆☆☆）

Q2. 在一个数组中寻找一个区间，使得区间内的数字之和等于某个事先给定的数字。（★★★☆☆）

Q3. 在一个二维矩阵中，寻找一个矩形的区域，使得其中的数字之和达到最大值。（★★★★☆）

我思：　Q2（★★★☆☆）这题让我想起了“力扣001.两数之和”，不同的是有所延伸，即求“ n 数之和” 的第

一个符合要求的区间。“两数之和”是有哈希复杂度$O(1)$ 解法的，但我当时做的哈希方法太乱了，以至于并没有真正掌握那种方法。

于是乎，在我做Q2 这题的时候，想要通过修改那时候的代码来实现一个$O(1)$ 解，发现当时写得实在难以入目，以至于我难以回想当时的思路，可见良好的代码可读性多么重要！我把当时的“两数之和” 代码也贴在后面了，**目的提醒自己“丑代码是我成长的拦路石”**。

```python
# l = [-10, 1, -2, 5, 11, 7, 9, -3, -2, 3]
l = [1.5, -12.3, 3.2, -5.5, 23.2, 3.2, -1.4, -62.2, 44.2, 5.4, -7.8, 1.1, -4.9]
target = -18
# Q2. 两重循环 O(K^2)
def target_sub_array(nums, target):
    temp_sum, left, right, p = 0, 0, 0, 0
    stop = 0

    while p < len(nums) and not stop:
        for q in range(p, len(nums)):
            temp_sum += nums[q]
            if temp_sum == target:
                left, right = p, q
                stop = 1
                print(left, right)
                break
        p += 1
        temp_sum = 0

    return left, right, nums[left:right + 1]

target_sub_array(l, target)
```

```python
# 力扣 001.两数之和，散列实现【我的丑代码，成长的拦路石】
class Solution:
    def twoSum(self, nums: List[int], target: int) -> List[int]:
        knownNums = dict()
        res = []
        for i in nums:
            knownNums[i] = target - i
            if knownNums[i] in nums:
                if knownNums[i] == i:
                    if  nums.count(knownNums[i]) > 1:
                        res = [nums.index(i), nums.index(knownNums[i], nums.index(i) + 1)]
                    else:
                        continue
                else:
                    res = [nums.index(i), nums.index(knownNums[i])]
                return res
```

■

我思： Q3（★★★★☆）

（1）思路一（有bug）：先找出单行最大区间所在行，接着围绕该区间向四周寻找增量大于0，宽度一致的区间（但直觉上这个方法有bug，比如当一列是最大区间的时候，某一行如果寻找到长度大于1的区间，那么前面的做法就会出现问题）。

（2）思路二（暴力破解），从区间长度为1 开始，一个个地记录，可以使用前面两重循环解决，但面对二维的数组，这种方法“可解但不可取”。

（3）思路三（降维映射）

■

复盘总结

aaa

引申

- https://www.bilibili.com/video/BV1g4411D7bg?spm_id_from=333.337.search-card.all.click&vd_source=1e9ce768698d539dc8e4dbecfe72afaa，林群院士演讲
- https://www.bilibili.com/video/BV1cx411m78R?spm_id_from=333.999.0.0&vd_source=1e9ce768698d539dc8e4dbecfe72afaa，3Blue1Brown 《微积分的本质》01课
- https://share.weiyun.com/uDta0nMf，【共读交流分享视频】共读群友的习题讲解
- https://leetcode.cn/problems/maximum-subarray/ 力扣053.最大子数组和/最大子序和

1.4 关于排序的讨论

这一节的目标：进一步理解和识别算法的无用功——不可或缺VS 无用功。

概要 （1.4.1 ~ 1.4.2）

- 本节借助排序算法，从“识别有效与无效” 的角度，进一步剖析了“分治与递归” 的思维与步骤。
- 排序算法$O(N^2)$ VS $O(N\log N)$
  - $O(N^2)$ 通常较为直观，符合认知直觉（专业背景要求不高）；
  - $O(N\log N)$ 通常不直观，需要找出并移除无用功（专业背景要求较高）；
  - **理解的关键：计算思维，突破直觉，拥抱递归与分治**。
- 选择排序（Selection Sort）$O(N^2)$
  - 步骤：（排队）先找到最矮的人，让他站在最前面，然后在剩下的人里找最矮的，排在第一个的后面，**以此类推，直到**把所有人排好；（例子来源：《未来算法》诸葛越）
  - 核心：**一趟一最值，结束换位置**。（书中步骤则用了冒泡排序“两两交换” 的思想）
  - **无用功**：
    - 无谓的比较——由于**不等式的传递性**，$A<B, B<C$, 必然有$A<C$ （P48）；
    - 无谓的位置互换——应抓住**“有效移动”**。
- 插入排序（Insertion Sort）$O(N^2)$
  - 步骤：（抓牌）先抓一张牌在手，第二张开始跟旧牌比大小，然后把新牌插入合适的地方；
  - 核心：**抓一比一，合适插入**。
- 归并排序（Merge Sort）$O(N\log N)$
  - 步骤：（排队）男孩从矮到高排好，女孩同样，然后每次从两个队的**队首挑选**出比较矮的孩子，依次合成一个队；
  - 核心：**不断假设子问题已解决而形成递归结构，分组、比较、合并**；
  - **缺点**：需额外存储空间$O(N)$ 保留中间结果，归并时过多地一对一比较大小（P51、56）。
- 堆排序（Heap Sort）、快速排序（Quick Sort）$O(N\log N)$ —— 不满足稳定性。
- 就地特征（in place characteristic）：不占用额外空间。例如用三个变量迭代生成斐波那契数列。

摘录 （P53）

接近理论最佳值的算法可能有很多种，除了单纯考量计算时间外，可能还有很多考量的维度，因此有时不存在一种算法就比另一种绝对好的情况，只是在设定的边界条件下，某些算法比其他的更合适罢了。

这段摘录，让我想起了西瓜书（《机器学习》周志华，P9）中提到的“没有免费的午餐”定理（No Free

Lunch Theorem，NFL），讲的是“无论一个学习算法多聪明，另一个多笨拙，它们的期望性能竟然是相同的”，**书中有这么一段话**“NFL 定理最重要的寓意，是让我们清楚地认识到，**脱离具体问题，空泛地谈论‘什么学习算法更好’毫无意义**，因为若考虑所有潜在的问题，则所有学习算法都一样好。要谈论算法的相对优劣，必须要针对具体的学习问题；在某些问题上表现好的学习算法，在另一些问题上却可能不尽如人意，**学习算法自身的归纳偏好与问题是否相配，往往会起到决定性的作用**。”

我对这段话之所以印象深刻，是因为这个定理坚定了我的某种认知，强调了“无需执着于比较，合适的才是最好的”这样一个道理，书中的那番话不只是算法，也不止于算法。

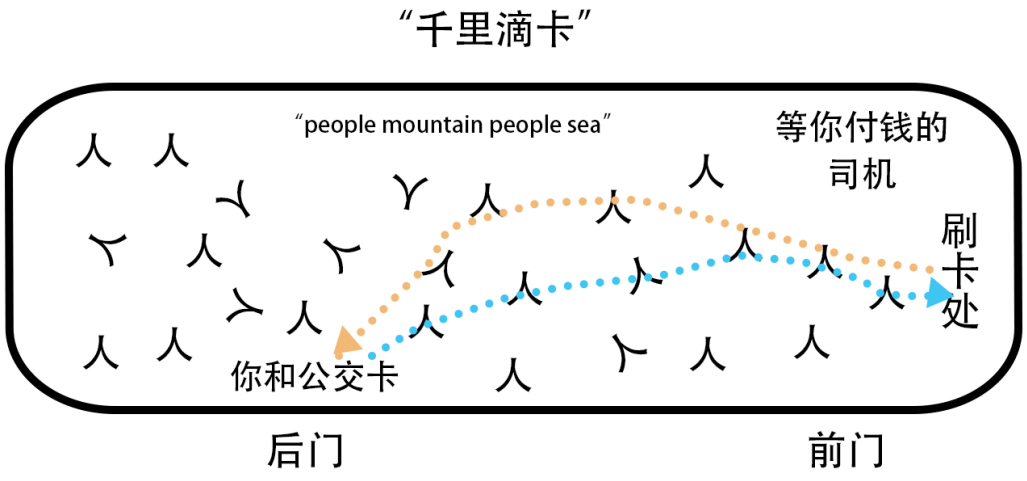
【归并排序】

分治法的思想：将原问题分解为几个规模较小但类似于原问题的子问题，递归地求解这些子问题，然后在合并这些子问题的解来建立原问题的解（《算法导论》P16）。

归并排序，如果按照《算法导论》的说法，主要分为**三步骤：分解、解决、合并**。由于这个算法是分治与递归的结合体，在理解的过程中，难免会出现各种反直觉的情况——明明“分解”的时候开开心心，结果要说“解决”问题，就开始“套娃”了。两三层的套娃好像还能理解，但套娃一多，就开始晕头转向，陷入云雾缭绕之景——其实如果人人都能理解套娃，那或许就不需要用计算机来下套和解套了。套娃虽有趣，但并不适合人的思维习惯——钻进去，就容易出不来了，这是一个“思维虎穴”。

而**理解递归的关键**，就是吴军老师书中反复提及的**“假设”**一词，将一个序列一分为二，我们面对的是直勾勾的两个序列，**怎么解?**

我想到一个例子，当我们必须挤进一趟看似爆满的公交车，司机通常会让我们“从后门上车”，这时候就有一个问题：**“我们该怎么刷卡呢？”**想必通常的做法就是：**往刷卡处看去**，把你的公交卡递给离你最近的乘客。由于惯性（笑），接到你公交卡的乘客就会把你的公交卡又往前面递，一个接一个地递，你的**公交卡就送到了距离刷卡处最近的人手上**了——之后你听到了“滴”的一声。可是任务还没完成呢，你的卡还得回到你的手里，于是帮你刷到卡的乘客就**往你的方向看**，开始重复你的操作，**最后你取回了你的公交卡**，并表达了感谢。这样，**一次“千里滴卡”就完成了**。



我们可以想想，如果是你接到了这样一张公交卡，你通常会怎么做？哈哈，往前面递呗，总能给递到最前面那个人手里。**这个想法就好像我们使用递归算法的心理状态**——我们拿到了名为“两个子序列”的这样一张“公交卡”，我们能做的自然就是“递给谁”，正如我们相信公交卡可以递到刷卡处一样，我们也相信把“两个

子序列”递给某种东西，可以帮助我们到达某处——而这种东西在算法里通常就是**递归结构**。

不难发现，这个过程**我们总是“假设谁可以”帮助我们解决问题，并且我们找到了“这个谁”**，因此面对直勾勾的“两个子序列”，我们就当作它可以排好序、已经排好序，当“这个谁”接到排序任务，他又会想找“下一个谁”帮他解决这个问题，**直到问题分解到可以直接解决的程度**（比如归并排序，根据其定义，分解到每个区间只有一个元素的时候，就默认/相当于排好序了，这一步就是成功的“刷卡”）。可见，这个过程体现了“假设”的魔力。

不同的是，传递公交卡的一来一回，可以由不同的人，即沿着不同的路线进行；而递归的过程，一来一回是固定在一条根据问题自动生成的路线、沿着相反的顺序进行，并且都是同一个人在干活。

```python
# 归并排序，模块化写法，O(NlogN)
def merge(left: list, right: list, nums: list) -> None: # 【合并】N 次
    i, j, k = 0, 0, 0                       # 也可以使用 append、pop 压缩代码量，但随之带来的复杂度提升需具体处理
    size_left, size_right = len(left), len(right)

    while i < size_left and j < size_right: # 比较左右子列表的首位元素
        if left[i] < right[j]:
            nums[k] = left[i]
            i += 1
        else:
            nums[k] = right[j]
            j += 1
        k += 1

    while i < size_left:            # 当左列表有剩余元素，可直接合并
        nums[k] = left[i]
        i += 1
        k += 1
    while j < size_right:           # 同理，剩余元素直接合并
        nums[k] = right[j]
        j += 1
        k += 1

def mergeSort(nums: list) -> list:  # 【分解】logN 次
    size = len(nums)
    if size > 1:                    # 当某一子列表 size 等于 2，它的左右子列表 size 等于 1，再次递归调用，将直接返回
                                    # 此时，其中一条递归小分支结束，将开始往回读取函数栈的数据，并开始往下执行语句
        mid = size // 2
        left, right = nums[0:mid], nums[mid:] # 切片，一分为二

        mergeSort(left)             # 【解决】假设子列表已经排好序，把求解过程交给递归结构
        mergeSort(right)
        merge(left, right, nums)    # 【合并】等待某条递归小分支的左右小小分支调用完毕，这部分才执行

    return nums
```

归并排序的算法复杂度为$O(N\log N)$，其中$O(\log N)$源于**分解的过程**，这部分解释书中讲得比较详尽，

就不再赘述；$O(N)$ 源于**比较与合并的过程**，这部分由于跟合并过程挂钩，为了顺便理清合并的思路，就稍作分析。

在比较与合并的过程中，我们要考虑**“最多”**要做多少次比较，**“最多”**要移动多少个数字，因此，我们可以跳到最后一步来理解——即只有长度为$\frac{N}{2}$ 的两个子序列的时候，合并完这两个序列，我们就完成了整个序列的排序了，因此它们都被传入了上面代码的$merge$ 函数当中，这时候，第一个$while$ 循环，就会将它们的首元素依次比较与更新（移动到下一个位置，首元素就更新了），然后把“比出来”的元素放到合适的位置（这里是覆盖了原来的$nums$ 序列），直到其中一个子序列“比完了”。

这个过程中，因为我们是一个一个地比较，所以比较的次数就是两个子序列放进$nums$ 序列的元素总数。当然，还可能有零星的“没比完” 的元素，但这已经不影响我们去估算整个过程的比较次数和移动数字的个数了，显然就是N 了（P50）。或者再直接点，合并的过程有三个平行的循环，一个循环当作一个N（正常情况下，后面两个循环要处理的元素非常少），加起来最多就是$O(3N)$，化简后就是$O(N)$ 。

思考题 1.4（★★★☆☆）

Q2.区间排序

如果有N 个区间$[l_1, r_1], [l_2, r_2], \ldots, [l_N, r_N]$ ，只要满足下面的条件我们就说这些区间是有序的：存在$x_i \in [l_i, r_i]$ ，其中$i = 1, 2, \ldots, N$.

比如, $[1, 4]$ 、$[2, 3]$ 和$[1.5, 2.5]$ 是有序的, 因为我们可以从这三个区间中选择1.1 、2.1 和2.2 三个数. 同时$[2, 3]$ 、$[1, 4]$ 和$[1.5, 2.5]$ 也是有序的, 因为我们可以选择2.1 、2.2 和2.4. 但是$[1, 2]$ 、$[2.7, 3.5]$ 和$[1.5, 2.5]$ 不是有序的.

对于任意一组区间, 如何将它们进行排序?

我思： 这一题实际上是借助“元素有序”来定义“区间有序”，其中的元素包含浮点数——而我们不难感知到：区间内的浮点数是无穷的。因此**我在想一个问题：究竟是从无穷当中找到某些合适的值，来顺着题目要求排序？还是找到其中的不变关系（某种绝对性），比如字面上的端点关系、比如把区间看作一段集合进行运算（作差），或者锁定不成立的情况以排除呢？**

进一步分析发现，如果是对区间的集合作差（目的是求出真正影响整体次序的“有效区间”，此时我们可以得出若干数量的小区间），通过比较小区间的数量和其在数轴上的先后关系，应当可以证明给定的一组区间是否能得出“有序解”，这可以确定解的存在性，但并不能直接得出具体的排序，还得绕点弯从中选出合适的值，进而回到对应的区间当中。但这样分析之后，尽管筛选出了一部分的“不变性”，但还是得从无穷当中找到合适的值，所以我对这种方法的理解就止步于此了。

回到明文上的东西，我们可以尝试从端点的关系入手。画几个数轴，我们可以看出一个事实：无序的情况下，总是存在至少一个更大的区间，在排序的时候，放在了更小区间的前面。例如书中例子：$[1, 2], [2.7, 3.5], [1.5, 2.5]$ 。**概括来说：这其实引发了客观顺序（数轴顺序）与主观顺序（取值排序）的矛盾**。反过来说，如果在排序的过程中，能够避免这种矛盾情况，无论实际排序的结果如何，这个结果都将是有序的——因为我们都能够在其中找到符合“区间有序”要求的“有序元素”，只需要在得出结果后取值即可。

这让我**想起了一个类似的结构与过程：判断下列哪种出栈顺序是正确/错误的**。我们或许可以尝试将这道“区间排序”问题，转化为“判断正确出入栈的序列”问题。不同的是，出入栈的元素是离散的、独立的，而区间是可以相交、包含的。这里只列举一下思路的可能性。

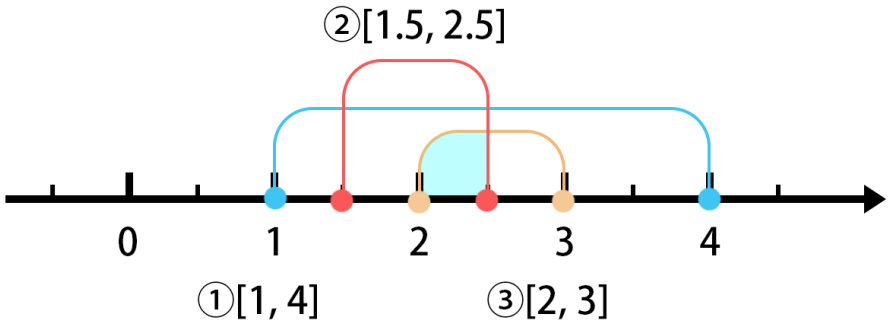
要避免前面的矛盾，我们可以**先确定好“给定区间组的客观顺序”**，接着基于这样的客观事实做文章。**有**

趣的是，有了上面的结论，似乎我们只需要将给定区间按照左端点（即区间最小值）排序，就一定可以找到满足“区间有序”的x_i 值。

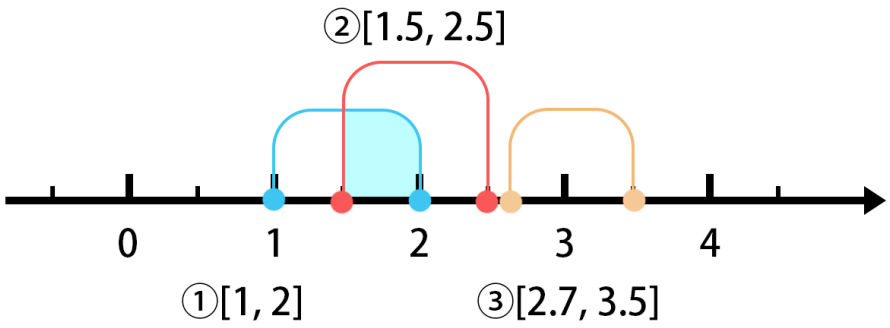
首先，这个结论是符合直觉的；其次，如果需要一个数学证明——那么就是**利用数轴本身的有序性**，一维数轴本身的定义。我们的工作不过是把一个个的区间，以左端点为参考，重新摆放到数轴上罢了。

这种方法主要有两个局限：第一点，上述“客观顺序与主观顺序产生矛盾”的一般性，我没能给出相关的证明；第二点，基于数轴，只能得出一种符合的区间排序，而通过排列组合与排除法，显然符合的排序可以有多个。

这种方法，尽管难免有“取巧”的性质，但起码能得出一个结果。如果想要求出所有符合要求的区间排序，得花更多的时间去寻找问题中的不变性了。对于后者，**这里只列出思路**，我们依然可以基于数轴（从左往右升序）的不变性，先利用左端点l_i 排序得到一个有序结果；接着可以用右端点r_i 排序，又得到一个不同的有序结果；而对于$[2,3],[1,4],[1.5,2.5]$ 这样的区间，由于三个区间中间存在交集（如下图所示），因此无论怎么排列，都是有序的。



而对于$[1,2],[2.7,3.5],[1.5,2.5]$ 这样的区间，只有两个区间存在交集，**没有交集的区间顺序受数轴限制**。



由此可见，**“存在交集的区间”意味着这样一个事实：它们在区间排序的时候可交换位置顺序**，原因也是直观的：我们总能在它们的交集区间内，找到符合题目要求的x_i 值。**那么，在算法设计的时候，我们可以尝**

试通过区间的端点关系，找出或者标记出哪些区间有交集，哪些没有交集，从而基于数轴的顺序，对有交集的区间互相交换、排列组合，就可以得出所有可能的有序结果了。

```python
# 1.4 Q2, 利用数轴本身的有序性，可以给出一种或两种排序结果。 冒泡排序 O(N^2)
def bubbleSort(nums: list) -> list:
    for i in range(len(nums)):
        for j in range(i + 1, len(nums)):
            if nums[i][0] > nums[j][0]:                # 若需根据右端点排序，此处改为 [1] 即可
                nums[i], nums[j] = nums[j], nums[i]
    return nums

if __name__ == '__main__':
    # nums = [99, -1, 3, 0, 77, 17, 33]
    # nums = [[1, 4], [2, 3], [1.5, 2.5]]
    # nums = [[2.7, 3.5], [1, 4], [1.5, 2.5]]
    nums = [[1, 2], [2.7, 3.5], [1.5, 2.5]]            # 将区间以二维数组的形式传参
    print(bubbleSort(nums))
```

■

复盘总结

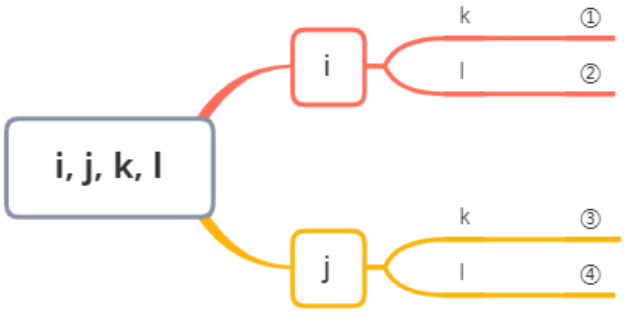
引申

- https://www.cs.usfca.edu/~galles/visualization/Algorithms.html，算法可视化网站
- https://github.com/python/cpython/blob/main/Objects/listsort.txt，跳跃式预测，P55
- https://thonny.org/，Python IDE for beginners，带调试器（步进功能）的轻量IDE
- 《未来算法》诸葛越

概要 （1.4.3）

- 混合排序算法（Hybrid Sorting Algorithm）：取长补短。
- 内省排序（Introspective Sort, Introsort）
  - 快速排序+ 堆排序
  - 应用：如今大多数标准函数库（STL）中的排序函数使用的算法
- 蒂姆排序（Timsort）$O(N \log N)$ 以内
  - 插入排序（节省空间、简单直观）+ 归并排序（节省时间、效率高）
  - 性质：时间复杂度上界控制在$O(N \log N)$，同时保证稳定性，相当于以块（run）为单位的归并排序，块内部有序；
  - **思想：利用数据连续有序的特性，减少排序中的比较和数据移动次数**；
  - 步骤：
    - 找出序列中的所有单调子序列，用插入排序整理较短子序列，用二分查找确定插入位置；整理后放入临时堆栈空间；
    - 跳跃式（galloping）预测，确定块中各组边界；按规则合并各块，先短后长，批处理归并（成组归并）；
  - 应用：如今Java、Python 和安卓（Android）操作系统内部使用的排序算法，广泛用于多列表的排序。
- 排序算法的复杂度**下界/下限**为$O(N \log N)$
  - 从讨论两个序列的有序性，到讨论一个序列的有序性，即从不同序列的比较，到单一序列的排列组合之间的比较——从而**定义**我们在排序算法中讨论的**排序原则**（P58）；
  - 这个原则可体现在一个序列的M 种排列组合，即对这M 个**序列的大小比较需要进行多少次元素比较**当中——可以讨论这样两个序列以求得结果：
    - 两个序列的第i 位元素与第j 位元素互换；
      - $a_1, a_2, \ldots, a_i, \ldots, a_j, \ldots, a_N$
      - $a_1, a_2, \ldots, a_j, \ldots, a_i, \ldots, a_N$
    - 如果$a_i \leq a_j$ ，第一个序列小于第二个序列；反之，第二个序列小于第一个序列；
      - 结论：两个元素做一次比较，最多能分出两个不同的序列大小。
    - 推广：进行两次比较，最多能分出四种序列（一棵二叉树），以此类推，做k 次比较，最多可区分2^k 种不同序列的大小。
      - **结论：反过来，有M 种序列，需区分它们的大小，需要$\log M$ 次比较**。
  - 采用斯特林（Stirling）公式，计算大数阶乘的近似解：
    - $\ln N! = N \ln N - N + O(\ln N)$
    - $\log N! = O(N \log N)$

我思：　进行两次比较，最多能分出的序列数目，如下图所示：



书中（P58）讨论**元素之间的比较次数**与**可区分的序列数量**的问题，可转换为一道经典的题目：**求n 元素集合的子集的个数?** 我们可以对每一个元素做一次选择，并判断它是否属于这个子集，这样一共分成n 个阶段，每一个阶段有两种选择，这样的子集总数为：

$$\underbrace{2 \cdot 2 \ldots 2}_{n\text{次}} = 2^n$$

此时，原问题中的**“一次比较”**，对应此处的**“一个阶段”**，**一次比较有两种结果，两次比较在第一次的两种结果基础上乘以2 ，得四种结果**。因此，对于n 个阶段，结果可得2^n 个不同子集，即原问题中的k 次比较，可判断2^k 个不同序列。

■

我思：

斯特林公式：

$$n! \approx \sqrt{2\pi n}(\frac{n}{e})^n$$

尝试计算$\ln N!$ 与$\log N!$ ：

$$\begin{aligned}
\because \ln N! &= \ln \sqrt{2\pi N}(\frac{N}{e})^N \\
&= \ln \sqrt{2\pi N} + \ln (\frac{N}{e})^N \\
&= \ln \sqrt{2\pi N} + N(\ln N - \ln e) \\
&= \ln \sqrt{2\pi N} + N \ln N - N \\
&= N \ln N - N + \ln \sqrt{2\pi N} \\
&= N \ln N - N + O(\ln N) \\
\therefore \log N! &\approx O(\ln N!) \approx O(N \ln N - N + O(\ln N)) \approx O(N \ln N) \\
&\approx O(N \log N)
\end{aligned}$$

■

思考题 1.5（★★★☆☆）

Q1.赛跑问题（GS）

假定有25 名短跑选手比赛争夺前三名，赛场上有五条赛道，一次可以有五名选手同时比赛。比赛并不计时，只看相应的名次。假如选手的发挥是稳定的，也就是说如果约翰比张三跑得快，张三比凯利跑得快，约翰一定比凯利跑得快。最少需要几组比赛才能决出前三名?

我思:

这题要计算得出答案的最小信息量。考虑几个条件：五条赛道，同组同时比赛，一组比赛结束到下一组，赛场上存在三名最快的选手。

（1）先考虑用最一般的方法，即每次选出前三名继续比赛，得出的答案**最多需要多少组**:

- 5 组，选出赛后各组前三，剩15人;
- 3 组，选出赛后各组前三，剩9 人;
- 2 组，选出赛后各组前三，剩6 人;
- 2 组，选出赛后各组前三，剩4 人;
- 1 组，选出赛后各组前三，剩3 人;
- **共**13 **组**。

（2）为了考虑所有的情况，每个人至少参与一次比赛，即**至少需要**5 **组**。也就是说，按照题意，若存在某种方法可以更快地决出25 个人当中的前三名，而这种方法需要的**组别数目范围在**$5 \sim 13$ **之间**。

（3）不难发现，每次每组决出的前三名，存在大量的**冗余情况**，例如全局的前三名可能都源于每组的第一名，或者某组的前三名就是全局的前三名，又或者局部分散开来，但能确定的是:

- 1 组，全局的第一名就在第一轮的五个第一名当中诞生;
- 1 组，当全局第一名决出，就可以在五个组目前排名最前面的人当中继续比较，即全局第一名那个组的第二名，参与下一轮比较，由此可以决出全局第二名;
- 1 组，同理，全局第二名后面的那个人，参与下一轮的比较，决出全局第三名;
- **这样，一共需要**$5+1+1+1=8$ **个组，可决出全局的前三名**。

■

1）**是否还有更优的方法? 如何证明上面的方法是或不是最优的? 而如果问"有什么方法可以直接求出这个最少组别数? "就有点懵了**。这题吴军老师书中有**解答过程（P213 - P216）**，既然往深想不出来，那么尝试从中学习优化思想。

2）进一步地，利用**方法（3）**中第二轮比赛的情况，将最后两名及其所在的组员排除，由于整体第一名已决出，则下一轮比赛可空出三个名额。此时，假设比赛排名依次为$A1, B1, C1$，正如书中（P215）所言，角逐第②名的只剩下$B1, A2$ ，角逐第③名的只剩下$C1, B2, A3$ ，如下图所示:

	①	②	③		
	A组	B组	C组	D组	E组
①	A1	B1	C1	D1	E1
②	A2	B2	C2	D2	E2
③	A3	B3	C3	D3	E3
	A4	B4	C4	D4	E4
	A5	B5	C5	D5	E5

如此，共需要$5+1+1=7$ **组，即可决出整体前三名**。

3）不得不佩服，老师的这种方法非常巧妙地识别出**哪些有用、哪些无用，哪些资源在什么情况下可用**。

```
# 暂略 未能理解``有什么方法可以直接求出这个最少组别数? ''似乎可以用图的思想、路径的思想来解决
```

我思：

究竟如何才能想到上面的最优方法？“如何活学活用，不生搬硬套？（P215）”明明已经将“识别无用功”铭记于心，摆正了一部分在算法优化的态度，但实践的时候似乎它们总藏在许多难以发觉的犄角旮旯，可能还是见得少，练得少吧。那么见一次，理一次，记一次，练一次，总归还是有用的，或许以后面对类似的问题就能一步步地站得更稳了。

这一次的算法无用功，藏在了一次比较后分割出来的排名当中，关键是要清晰地讨论一组比赛的始末，清晰地思考比赛前后这一组哪些是重要的、哪些是不重要的。并且，**在讨论多变的情况时，要作出某种不失一般性的假设，固定出一种具有代表性的静态情况，进而推广到动态的情况**。

无论是“有用功”还是“无用功”，都是对资源利用的一种度量。本质上就是在计算机的层面，去讨论各种计算资源的利用情况。这次可以说是侥幸想出来了8 组的解法，但看了书才知道里面就是归并排序的合并思想，所以也说明了后者掌握得不熟练。

回过头来分析，正如书中提到的上面**方法（3）**实际是归并排序算法的合并过程，尝试将问题转化为归并排序理解：

- 第一轮，五个组，对应五个有序子序列；
- 第二轮，一个组，对应每个子序列的队首元素，弹出一个元素；
- 第三轮，一个组，对应每个子序列的队首元素，弹出一个元素；
- 第四轮，一个组，对应每个子序列的队首元素，弹出一个元素。

在第二轮的基础上，使用上面**方法2）**的思想，类似上一节提到的**“就地特征”**，全局获胜的人和全局淘汰的人，从对应赛道中移除，因此赛道腾出了3 个位置，随后将位置留给了剩余可能的选手，从而“恰好”满足了条件。其中也体现了在**整体与局部之间灵活转换**的思想，对整体与局部而言，什么对象在何种情况下无用。

■

复盘总结

待复盘

读完了第一章，不禁要问自己：

- 你懂得判断某些算法的无用功了吗?
- 学完一章，印象最深刻的是什么?
  - （什么有用、什么无用，何时有用、何时无用）
- 你能否简单概括这一章的收获?
- 学习的过程中，有没有什么难倒你了?
  - （1.3 方法3、方法4、思考题Q3；GS代码）
- 那些难倒你的东西，你解决了多少？是怎样解决的呢?
  - （已部分解决方法4）（庖丁解牛，逐步击破）
- 没有解决的那些你又准备怎么做呢?
  - （继续回退，多学几遍，与后续内容一起思考，直到解决为止）
- 你如何衡量那些你认为“学懂了”的内容?
- 讲义的方式是否适合自己？写得详细，投入的时间是否值得?
  - （就目前而言，适合且值得；时间多用于思考；但仍在不断地寻找更合适的方法——目的是重于理解，而非形式）
- ……

2 逆向思考——从递推到递归

概要

2.1 递归：计算思维的核心

2.2 遍历：递归思想的典型应用

2.3 堆栈和队列：遍历的数据结构

2.4 嵌套：自然语言的结构特征

3 万物皆编码——抽象与表示

概要

4　智能的本质——分类与组合

概要

4.1　这是选择分类问题

4.2　组织信息：集合与判定

4.3　B+ 树、B* 树：数据库中的数据组织方式

4.4　卡特兰数

5　工具与算法——图论及应用

概要

5.1　图的本质：点与线

5.2　图的访问：

5.3　构建网络爬虫的工程问题

5.4　动态规划：寻找最短路径的有效方法

5.5　最大流：解决交通问题的方法

5.6　最大配对：流量问题的扩展

6　化繁为简——分治思想及应用

概要

6.1　分治：从$O(N^2)$ 到$O(NlogN)$

6.2　分割算法：快速排序和中值问题

6.3　并行初探：矩阵相乘和MapReduce

6.4　从机器学习到深度学习：Google大脑

7　权衡时空——理解存储

概要

7.1 访问：顺序vs. 随机

7.2 层次：容量vs. 速度

7.3 索引：地址vs. 内容

8　并行与串行——流水线和分布式计算

概要

8.1　流水线：逻辑串行和物理并行

8.2　摩尔定律的两条分水岭

8.3　云计算揭秘：GFS 和MapReduce

9　状态与流程——等价性与因果关系

概要

9.1　从问题到状态

9.2　等价性：抽象出状态的工具

9.3　因果关系：建立状态之间的联系

10　确定与随机——概率算法及应用

概要

10.1　信息指纹：寓确定于随机之中

10.2　随机性和量子通信

10.3　置信度：成本与效果的平衡

11　理论与实战——典型难题精解

概要