

杭州市北京航空航天大学国际创新研究院 (北京航空航天大学国际创新学院)  
Hangzhou International Innovation Institute, Beihang University



## AI 赋能优化的《算法设计与分析》教学

By  
欧阳真超

工程师通用技术科教平台  
教学改革项目 (2024 年)

杭州，中国  
2024.07-2025.07



# Acknowledgments

本研究得到杭州市北京航空航天大学国际创新研究院 (北京航空航天大学国际创新学院)2024 教改项目-AI 赋能优化的《算法设计与分析》教学项目资助, 项目编号为 JG202417;《空间双臂机器人跨模态多任务学习系统》科研启动经费资助, 项目编号为 2024KQ011。特别感谢参与本项目的硕士 (贾皇城、张宇辰、梁瞳尧) 和本科生 (周诣博和吴俊达)。

# Contents

<b>List of Tables</b>	<b>vi</b>
<b>List of Figures</b>	<b>vii</b>
<b>Abstract</b>	<b>viii</b>
<b>1 Introduction</b>	<b>1</b>
1.1 项目简介	1
1.2 现状分析与存在的问题	3
1.3 大语言模型使用注意事项	5
1.3.1 OpenAI Chat-GPT 辅助功能介绍及注意事项	6
1.4 章节组织	7
<b>2 大语言模型交互编程</b>	<b>8</b>
2.1 生成式大语言模型在交互编程中的应用	8
2.2 大语言模型	8
2.2.1 通用大语言模型	8
2.2.2 生成式编程模型	11
2.3 大语言模型交互技巧	13
2.4 交互编程问题的形式化规约	14
2.5 量化评估指标	15
2.6 章节小结	15
<b>3 算法设计与分析基础概念学习</b>	<b>17</b>
3.1 算法设计与分析基础概念学习	17
3.2 学习顺序	18
3.2.1 冒泡排序	19
3.2.2 选择排序	21
3.2.3 插入排序	23
3.3 章节小结	29

<b>4</b>	<b>批判性思维的交互学习</b>	<b>30</b>
4.1	批判性思维应用 . . . . .	30
4.1.1	多种替代方案 . . . . .	30
4.2	章节小结 . . . . .	40
<b>5</b>	<b>IDE 集成编程开发环境学习</b>	<b>41</b>
5.1	IDE 集成编程开发环境配置 . . . . .	41
5.1.1	pycharm 界面简介 . . . . .	41
5.1.2	CodeGeeX 界面功能简介 . . . . .	41
5.1.3	CodeGeeX 界面功能简介 . . . . .	43
5.2	方案驱动的 AI 辅助算法设计流程 . . . . .	43
5.3	学习实例 . . . . .	44
5.3.1	归并排序 . . . . .	44
5.4	章节小结 . . . . .	60
<b>6</b>	<b>总结</b>	<b>62</b>

# List of Tables

3.1	排序算法对比 . . . . .	17
3.2	选择排序两个版本对比 . . . . .	23
3.3	插入排序两个版本对比 . . . . .	27
4.1	测试用例 1 . . . . .	30
4.2	结果对比 . . . . .	34
4.3	算法间对比 . . . . .	36
4.4	两个模型生成结果对比 . . . . .	38

# List of Figures

2.1	在 Pycharm 以插件形式的 CodeGeeX 交互界面 . . . . .	11
2.2	在 Pycharm 以插件形式的 Trae 交互界面 . . . . .	12
3.1	学习顺序 . . . . .	18
3.2	冒泡排序的时间复杂度测试。 . . . .	20
3.3	冒泡排序的内存测试。 . . . .	20
3.4	冒泡排序早停对比测试。 . . . .	21
3.5	选择排序对比测试。 . . . .	23
3.6	选择排序空间占用对比测试。 . . . .	24
3.7	选择排序对比测试。 . . . .	28
4.1	对比结果-代码差异分析 . . . . .	39
4.2	对比结果-预处理方式 . . . . .	39
4.3	对比结果-输出差异 . . . . .	39
4.4	对比结果-总结 . . . . .	40
4.5	Gemini 错误矫正 . . . . .	40
5.1	pycharm 界面下配置 AI 助手插件示意图 . . . . .	42
5.2	CodeGeeX 界面功能示意 (左) 和 Trae 界面功能示意 (右) . . . . .	42
5.3	面向方案驱动的 AI 辅助算法设计流程。 . . . .	43
5.4	CodeGeeX 在对 merge() 函数生成单元测试用例的错误 . . . . .	50
5.5	Trae 在对 merge() 函数生成单元测试用例通过结果 . . . . .	54
5.6	人工补充测试用例测试结果 . . . . .	55
5.7	Trae 在对 Merge_Sort_Recursive() 函数生成单元测试用例通过结果 . . . . .	57
5.8	一次运行下的性能测试结果 . . . . .	59
5.9	多轮次运行下的性能测试结果 . . . . .	60

# Abstract

针对传统《算法设计和分析》前导课程多且复杂、教学内容丰富，研究生无上机课，但又依赖学生课余时间编程实践学习的问题；以及杭州国际创新学院学生跨专业背景差异大、基础课程缺失、编程能力参差不齐，导致学生在算法编程实践中难以找到适合自身基础学习抓手的双重挑战。本项目希望结合自身在计算机基础、编程实践经验和人工智能领域的知识背景，以及以往自学和研究过程中对相关编程和人工智能技术的应用经验，探索如何利用生成式人工智能的强大记忆与认知能力、数据压缩和高效搜索能力、以及亲和的交互能力，改善《算法设计和分析》的教学和学生的编程实践过程。通过该项目与学生共同探索如何在人工智能时代，借助技术改善编程实践过程，辅助信息检索、代码逻辑分析、算法设计与优化，降低《算法》课程可对背景知识的要求、加速编程过程，形成一套适合个人知识储备的 AI 工具包，并将这一学习方法推广到今后的学习和工作中。



# Chapter 1

## Introduction

### 1.1 项目简介

《算法设计与分析》<sup>[1]</sup>(简称“算法”)是计算机专业必修基础课程,涵盖算法设计、算法分析和算法实现等内容。学习该课程不仅能提高学生利用算法工具解决实际问题的能力,还促进了计算机科学理论研究,增强了学生的编程能力,并为后续课程和工作奠定了坚实基础。经典算法作为一种将已有知识固化,并利用计算机技术辅助执行的先进技术,能够显著优化诸多电子化的工作流程和数据处理任务的效率,降低对硬件设备性能的依赖,在各个学科和工业领域广泛应用,展现了强大的溢出效应。在本科和研究生阶段学习《算法设计与分析》,有助于学生形成计算思维和数据驱动思维,从而在解决自身领域研究问题时,利用高效程序优化和改进传统处理流程,提高效率和缩短研发周期。

《算法》课程在计算机学科体系属于高阶专业课程,依赖于包括如《计算机导论》、《操作系统》、《计算机组成原理》和《数据结构》多门前导专业课程,以及《高等数学》、《概率论与数理统计》、《线性代数》和《矩阵》等在内的基础数学课,课程内容复杂,知识点交叉、掌握难度大,依赖理论结合实践的学习方式,才能全面理解算法的设计原理、进行代码实践、并掌握计算编程用语、进行新算法知识检索和迭代更新,能够灵活的应用于后续遇到的实际工程任务中。而算法的实践过程,即利用所学算法求解新问题需要结合批判性思维、逻辑分析、算法设计、编程,以及学生自身对自然语言的理解;整个任务的每个环节均具有引入人工智能大语言模型辅助的潜在可能。

国新院学生跨专业背景差异大,非计算技术专业还包括电子信息、集成电路、控制工程、交通运输、机械工程、航空工程、智能制造、机器人、材料与化工、化学、能源动力等;这类专业学生的计算机类基础前导课程缺失、知识点交叉,编程能力参差不齐,非计算机背景学生直接选修《算法》课程掌握难度大,导致学生在算法编程实践中难以找到适合自身基础知识的学习方法、可信的交互对象和高效实践练习手段。在教学过程中如何降低学生对《算法设计与分析》的学习门槛,与其它学科形成交叉,赋能计算机/电子以外的专业,发挥计算思维的实践优势,形成算法对其它学科的溢出效应成为一个挑战。

中国特色的互联网公司发展起步晚，被商业资本裹挟逐利，近年来互联网共享精神几乎消亡，百度搜索引擎的商业化排序、CSDN[2] 等互联网毒瘤近乎无法提供有效编程信息、还充斥大量垃圾和低效的转述信息，极大的阻碍了软件开发人员对有效信息的搜索，而大型商业网络资源公司对开源文档进行收费等卑劣行为更是令人发指。而尚未在国内形成的开源风气，又被所谓的知识付费市场冲击，使得有效的资料检索更为耗时费力。原版参考书籍、开源代码和函数 API 多由国外 GNU 社区发起和维护，文字和配图主要采用英语，对国内学生的语言能力有一定的要求。而互联网监管则使得 GitHub[3]、StackOverflow[4]、Stack Exchange[5]、Reddit[6] 等网站的访问速度缓慢，从而进一步限制了学生对高质量编程信息的有效检索途径。

随着人工智能技术和相关应用的发展，特别是在自然语言处理领域的生成预训练 (Generative Per-trained Transformer)[?] 大语言模型 (Large Language Models, LLMs)[8] 的发布，AI 模型具有了基于自然语言生成计算机代码的能力，从而极大的提升了传统编程开发的流程，被广大程序员群体应用于辅助开发流程，并受到广泛好评。Z 世代学生从小伴随着网络和电子产品成长，视野开阔、对新兴技术兴趣度高、掌握速度快。当然也有部分研究人员和教师担心在编程基础学习的阶段，过早引入 AI 辅助会使得学生对工具产生依赖性，从而阻碍他们掌握基本编程的能力。部分保守派过度担忧利用 AI 助手进行论文、作业的自动生成，进而研究如何对基于 GPT 生成文档的检测和禁止 [9]；本研究则持相反态度，因为作为新兴工具，AI 助手的渗透率将逐渐提高，在教学中融入 AI 助手辅助，并让学生了解当前技术的局限、辨别其中的谬误，与 AI 助手共同成长将成为未来的趋势。

就算法学习和编程实践而言，程序员利用 Copilot[10] 提升工作效率的经验表明，AI 助手有望成为初步接触编程的学生的学习助手，协助完成包括编程问题解释、流程设计、算法代码生成、代码注释、代码优化和 Debug 等功能，并极大提升学生学习效率的潜力。而大语言模型还存在生成错误和“幻觉”问题，而学习初期的学生由于缺少基本的判断能力，会学习错误的知识，进而编写错误的代码。此外，与大语言模型交互时，需要额外注意通过使用特定的提示 (Prompt)[11] 能够提高交互效率，而计算机编程也有自身的专业术语，因此学生还需要掌握计算机编程术语的 Prompt 的交互经验。

针对上述问题，本项目对利用编程大语言模型改进编程教学的相关研究进行了调研，并希望利用“国产编程辅助语言模型”提升具有交叉学科背景、以及尚不具备熟练编程经验的学生的编程能力，从而加速学生对《算法》课程内容的掌握。同时，学习与编程大模型交互所需使用的 Prompt，以及在面对新任务时能够使用已有算法设计和编程实践对问题进行求解；此外，更为重要的是在与 AI 模型进行交互时，具备对生成内容的正确性、准确性和真实性的识别能力，从而在利用现有不完美的 AI 助手的同时，避免掉入大模型的陷阱。

## 1.2 现状分析与存在的问题

随着大语言模型被用于代码生成和低代码开发工作,相关工具 [12, 13] 也不断发展和改进,使得当前的学生能够使用上述模型解决编程作业和学习编程技术。研究者开始关注编程初学者与 AI 智能代理之间的交互过程 [14, 15, 16, 16, 17, 18], 其中提示词研究 [19, 20] 由于无需实际编程和依赖大规模计算硬件,备受非人工智能学科的研究关注。

通过系统文献综述,布鲁内尔大学的学者 [21] 探讨了对话式 AI, 如 Chat-GPT、GitHub Copilot 等,对软件工程教育与实践的影响。研究基于 2018 年至 2024 年的 183 篇文献,从三个视角展开分析:对话 AI 在软件工程教育中的采用、计算教育的整体影响,以及软件工程实践与教育之间的差距。研究发现,尽管对话 AI 在代码生成、测试和需求分析等软件开发生命周期任务中展现出潜力,但其在教育中的应用仍处于早期阶段,主要集中在入门编程课程,且存在学术诚信、过度依赖等挑战。文章指出,当前软件工程教育需调整课程设计,培养学生批判性使用 AI 工具的能力,同时强调未来研究应关注 AI 与人类协作的伦理框架、个性化学习工具开发,以及跨学科课程整合,以应对技术快速演进带来的变革需求。

其中,卢索丰纳大学的研究人员 [22] 分析了 69 名本科新生在未接受提示训练的情况下,完成课程项目编程任务时使用的提示策略。研究同时阐述了实验设计规则:通过特定练习框架引导学生与 AI 交互时培养批判性思维。结果表明,尽管学生使用的提示技术较为初级,但多数人成功借助 GPT 生成解决方案并将其整合至项目中。尤为重要的是,半数学生展现出对多个 GPT 生成方案的判别能力,这表明他们在评估 AI 生成代码的过程中发展了批判性思维技能。作者提出 AI 辅助的编程中的批判性思维应包含:向代理询问多套方案,并通过分析结论选择其中的最优方案。

来自四所不同大学的研究人员 [23] 针对现有数据集编程问题仅包含单一专家提示的情况下,招募了 80 名低年级编程初学者进行 AI 辅助的编程研究,并构建了针对 48 个编程问题的非专家、多对一提示数据集,也就是针对每个编程问题,梳理了来自不同学生的共计 1700 多条提示,并命名为 StudentEval 数据集,分析了学生交互过程的提示变化,以及首次提交提示没解决问题、首次提交提示解决问题、多次提交成功解决问题,和多次提交也无法解决问题的样本分布。通过统计分析,作者总结了现有 AI 编程代理的缺陷,一些与编程强相关的高度形式化提示词集合,以及多次提交过程中学生如何改进提示描述。

来自葡萄牙大学的研究人员 [16] 对面向对象编程 (OOP) 问题,利用基于 GPT-3 的 AI 智能体研究对互相关联类的生成结果。通过六个真实 OOP 作业 (以葡萄牙语描述) 对 GPT-3 进行测试,并通过开源自动评估工具 (AAT) 对生成的代码进行客观评价。研究发现,虽然 GPT-3 能够满足直接的功能需求,但在 OOP 设计方面表现欠佳,例如频繁使用显式类型检查 (如 `instanceof`)、继承关系处理不当以及代码质量不达标等问题。基于实验结果,作者为计算机科学教育者提出了建议,包括采用项目制评估、在作业中嵌入学生个性化信息等策略,以应对学生可能滥用此类工具的挑战。该研究为 AI 代码生成技术在教育领域 (尤其是非英语环境和高级编程范式) 的应用提供了新的实证



参考。

文献 [20] 提出了一种将“用简单英语解释代码”(EiPE) 任务与大型语言模型 (LLMs) 相结合的创新方法, 旨在同步提升学生的代码理解能力和提示工程技能。研究团队通过让学生用自然语言描述代码功能, 并利用这些描述作为 LLM 的输入提示来生成代码, 实现了对代码理解能力的自动化评估。实验结果表明, 采用高层次抽象描述 (关联结构) 的学生提示最有可能生成功能正确的代码, 且该方法在包含 900 名学生的入门编程课程中取得了高达 98% 的任务完成率。这项工作不仅为传统 EiPE 任务提供了客观的自动评分方案, 解决了人工评分主观性强的长期难题, 还展示了 LLMs 在编程教育中的双重价值: 既可作为评估工具, 又能帮助学生发展新兴的 AI 协作技能。研究进一步指出, 这种方法的优点包括支持多语言解释、增强学习参与度, 但也面临提示稳定性、复杂任务扩展性等挑战, 为未来 AI 时代编程教学法的革新提供了重要参考。

JAMES PRATHER 等人 [14] 研究了 19 名代码初学者使用 C++ 与 AI 编程助手的两种不同的交互方式: 1) 通过利用 Copilot 自动生成的代码提示, 引导它朝着解决方案的方向发展, 而不是专注于从零开始编写代码并整合 Copilot 的建议; 这种行为被称为“引导 (shepherding)”。2) 一些学生则被 Copilot 的不正确建议一步步引导, 逐渐在不同建议中漂移, 最终迷失方向; 这种现象被称为“漂移 (drifting)”。他们还观察到, 学生在使用该工具时面临认知与元认知上的困难。最后, 基于这些发现, 提出了相应的伦理考量和设计指导原则。

针对现有研究主要面向一年级学生的编程对比, 文献 [18] 研究了 AI 编程工具在计算机二年级学生的数据结构和算法课 (data structures and algorithms), 并通过与传统学习的学生的作答情况进行对比, 表明 AI 比大多数学生获得更高的成绩。同时, 分析表明, Codex 在起点为空白、且需求明确具体的任务中表现较好; 而题目描述较长的任务, 其完成率则显著下降。需要修改已有代码或需处理隐含边界情况的任务, Codex 的表现通常较差。这些使 Codex 面临挑战的特性, 如模糊或隐含的要求、复杂的边界情况、冗长的问题描述以及对现有代码的理解与修改需求, 同样也会增加学生作答的难度。研究结果进一步表明, Codex 在解决中等复杂度的代码阅读与编写问题时表现优异, 但在遇到罕见的边界条件或较为晦涩的需求时, 性能显著下降。

文章 [19] 提出了一种新的编程教学练习形式——提示问题 (Prompt Problems), 旨在培养学生针对大型语言模型进行有效提示设计 (prompt engineering) 的能力。研究开发了配套工具 Promptly, 支持学生通过自然语言编写提示词, 并自动评估 LLM 生成代码的正确性。通过在计算机学生一二年级课程中的应用实验, 作者发现 Prompt Problems 能够有效激发学生的计算思维, 帮助其学习新的编程结构与表达方式, 同时也暴露出部分学生对于依赖 AI 生成代码的担忧与抵触。结果表明, Prompt Problems 作为新型编程任务形式, 在提升学生逻辑表达、问题分析与代码生成指导能力方面具有积极作用, 并为未来生成式 AI 环境下的计算机教育改革提供了实践依据。

文献 [24] 则通过构建层级化分类标准分析了人工智能模型对算法的理解程度, 通过对比本科和研究生群体来论证该分层分类框架的有效性。基于此框架, 作者解释了大语言模型在对算法问题的理解程度上与人类理解能力之间存在的共性和差异, 并有望层

位对算法理解评估标准的监测和量化工具。

来自伦敦大学国王学院的研究人员同样希望利用 AI 模型辅助学生利用课外时间提升课程学习。通过招募 44 名 JAVA 语言初学者进行在线学习，利用学生的真实代码状态构建了提示-生成场景 (hint-generation scenarios)，结合比较判断法 (comparative judgment) 对 AI 模型和 5 个人类专家生成的 JAVA 提示进行排序；对比组的参与的学生在未经告知的情况下，同样对上述 JAVA 提示进行排序。通过分析参与者对提示生成的反馈，大语言模型 (GPT-4) 所生成的提示在教学意义上甚至超过人类专家。

澳大利亚墨尔本大学的研究人员 [25] 则设计了一种基于大语言模型的编程流程，从初始代码生成和后续改进两个方面考虑。其中，在方案生成阶段先通过大语言模型将问题拆分为多个子问题，然后生成对应测试用例对方案进行验证；接着在代码实现阶段，利用大语言模型基于上述方案生成对应代码，并利用测试用例进行验证。如果代码无法通过测试，方案验证环境将作为预设自然语言解决方案，持续引导代码调错直到通过。此外，文章还进一步提出基于 LPW 的改进的多重采样方法进行多方案生成和迭代优化进而提升模型生成方案的通过率。虽然该研究为端到端建模，缺少引入人工交互认知反馈带来的差异和考量，但总结了代码优化如何基于执行结果反馈。

虽然仅有极少研究 (仅一篇) 针对计算机二年级数据结构和算法考试作为研究对象，但文献仅考虑使用 AI 代理进行考试，未考虑如何在教学过程中引入 AI 模型。此外，当前研究虽在 AI 辅助编程领域取得显著进展，但主要聚焦于一年级入门编程课程。其余研究则主要以编程过程为研究对象，未考虑系统性教学的师生互动中如何引入 AI 智能体。因此，面向 AI 赋能的教学尚处于初步研究阶段，需要综合考虑 1) 场景适配不足：现有工具在复杂算法问题 (如动态规划、NP 难问题) 的生成质量、边界条件处理上表现欠佳，且缺乏针对算法教学特性的结构化提示模板；2) 教学闭环缺失：多数研究仅关注代码生成结果，未构建“算法思想讲解-案例生成-可视化验证-错误模式分析”的全流程 AI 支持框架；3) 认知反馈薄弱：忽视教师对 AI 生成算法 (如贪心策略的局部最优性证明) 的实时批注与修正引导，导致算法设计思维的培养断层。

## 1.3 大语言模型使用注意事项

本研究会采用不同公司开源、或开放功能的大语言模型参与研究，其中不同公司的模型、以及所有此类模型都存在一定的隐私泄露风险，以及独立的安全措施。通过简短的进行介绍，本研究还将相关使用过程的注意事项完整链接奉上，供学生在学习前进行学习、同时供其它人进行查阅 (注：部分功能会随公司和 IDE 工具的更新而发生些许变化)。

首先，一些学术机构包括期刊和会议，不允许使用由任何 GPT 生成的论文进行发表，并将相关行为列为学术诚信问题。其次，部分比赛，特别是艺术类大赛，也同样将使用生成式模型绘制的画作等艺术品排除在候选作品内容。最后，部分学校也考虑到学生会直接提交由生成式人工智能生成的作业，因此也将相关行为纳入学术诚信政策中。一方面凸显了新兴技术对传统政策产生了影响，另一方面，也促使我们从不同的角度思

考如何在教学过程中利用和驾驭相关技术，并促进相关政策制定合理监管上述技术的同  
时，为教学提供便利、为学生学习赋能加速，而非变成一个偷懒工具。

### 1.3.1 OpenAI Chat-GPT 辅助功能介绍及注意事项

#### 1.3.1.1 用户年龄限制

以 Deepmind 为例，Chat-gpt 对用户年龄要求，必须年满 13 岁，13 至 18 岁之间的用户必须获得父母或监护人的许可才能使用该平台。考虑到个别特例，大多数学生已经成年，加上使用过程在老师的指导下，能够满足相关要求。

#### 1.3.1.2 大语言模型的局限性

大语言模型虽然功能强大，能够回答任意提问，但包括“幻觉 hallucination”在内的问题仍然很严重，如大语言模型会一本正经的胡说八道，列举不存在的文献资料；大模型也存在巨大的认知边界，给出的答案并不完整；在计算器也能完成的基础数学计算，GPT 可能给出错误的解；实时性上，各公司也会陈述所使用训练的模型的数据集截至时间。上述问题受训练数据的局限性、有偏性和错误性、模型工作机制通过统计关联而非推理，以及对上下文的误解、用户输入的模糊性、以及结果的生成机制等诸多因素造成。此外，在模型输出上，不同的公司还有额外的数据后处理标准，从而限制危险、消极结果的输出，也同样对输出结果的内容和质量造成影响。

因此，在使用过程中教师和学生都需要正确认识现有工具的局限性，利用传统教学资料 (书本、教案、PPT)、传统互联网信息 (wikipedia、在线社区、mooc)、通过不同公司的模型产品交叉对比等手段进行验证。

#### 1.3.1.3 对话超链接

我们将利用学生与生成式大模型对话的记录进行统计分析，部分产品如 Chat-GPT 提供分享链接 (Shared Links) 功能 (Google Gemini 的分享功能) 均能够让其他人访问用户已经产生的对话，并通过 URL 的形式进行分发，并进行自定义备注以进行命名的区分。通过该功能可以便捷地掌握学生在学习算法中与大语言模型交互过程中的每轮对话的具体内容；同时，通过对比长周期的对话所使用的提示词的变化，能够分析学生在算法背景知识的掌握熟练度，以及与生成式大模型的交互技巧，对 AI 生成结果的批判思维过程。

此外，由于分享链接存在一定的隐式泄漏风险，我们也将在此注明在研究过程中如何通过强调安全风险，从而避免产生相应事故。首先，分享的对话过程默认不会包含用户的账户和个人信息，除非选择分享个人名称，仅在分享链接 URL 自定义时可以进行命名。研究过程中需要确认对话产生的实体，但又需要进行匿名保护，因此我们建议学生以学号等形式作为匿名化标注。然后，教学过程中可以用采用“精准提示词”与大语言模型交互产生的对话链接作为教学示例；同学之间或小组之间也能够通过分享已有对

话进行便捷沟通。接着，虽然 GPT 等工具在主页菜单具有相应的目录对对话进行管理，但是当使用多个不同公司的模型进行交叉学习时，则额外需要进行对比。这时，可以通过分享链接的形式按对话的主题内容进行组织记录、方便后续访问。最后，分享的对话链接可主动关闭，或通过删除相关对话进行删除，相应的超链接也会随之失效。

## 1.4 章节组织

第二章我们将以具体算法为例，介绍学习对于算法时所采用的基本提示词流程框架，以期使用最高效、准确的提示词对不同算法进行学习；对于特定的复杂算法引入额外的提示词，实现对特定知识点的掌握。

第三章

第四章

总结

# Chapter 2

## 大语言模型交互编程

### 2.1 生成式大语言模型在交互编程中的应用

本章节介绍研究中拟采用的 AI 大模型工具特性；并总结文献调研中发现的，利用大语言模型进行编程辅助的交互技巧经验；通过对比分析，将在后续的研究中选择其中的大模型工具和交互技巧作为本研究的交互对象，以及 AI 辅助编程的算法教学与学习的关键知识，从而提升学生理论学习的效率和降低实践编程的难度。

### 2.2 大语言模型

#### 2.2.1 通用大语言模型

##### 2.2.1.1 OpenAI Chat-GPT

Chat-GPT[26] 是由 OpenAI[27] 开发的生成式大语言模型，代表了人工智能在自然语言处理领域的重要进展。作为 GPT(Generative Pretrained Transformer) 系列模型的一部分，Chat-GPT 基于 Transformer 架构，采用了海量文本数据进行预训练，使其能够理解和生成多样化的自然语言文本。OpenAI 从 GPT-1 到 GPT-4 的不断迭代，极大地提升了模型的能力，尤其是在生成性文本、语言理解、推理能力等方面。Chat-GPT 以其强大的自然语言生成和理解能力，广泛应用于各行各业，包括教育、科研、商业、技术支持等领域。

Chat-GPT 的核心功能包括但不限于：生成高质量的文本内容 (如文章、报告、邮件、对话等)，提供技术支持与代码生成，帮助用户进行文本编辑和优化，解决数学、科学、编程等复杂问题，提供多语言翻译和语言学习支持，进行数据分析和报告生成等。其独特的能力在于能够根据用户的需求进行灵活、准确的文本创作，且具有强大的上下文理解与推理能力，能够生成逻辑性强且符合特定要求的内容。在生成式大语言模型的领域，OpenAI 的 GPT 系列标志着人工智能从单一任务解决向多任务处理的转型。GPT-4 的推出不仅增强了生成能力，还提升了模型的稳定性和可靠性，使其在应对更加复杂的



任务时能够保持更高的准确性和一致性。随着大语言模型技术的不断进步，Chat-GPT 在自动化工作、教育、科研、创意创作等方面展现了广泛的应用潜力。

其中，人机问答对话，也就是自然语言处理与生成是我们最常用的功能。Chat-GPT 可以理解用户输入的自然语言，并生成具有逻辑性和连贯性的回应，能够回答问题、进行对话、撰写文章、报告、邮件等各种类型的文本。同时能够提供技术支持：Chat-GPT 提供编程、算法、数学、物理等领域的技术帮助，解答用户提出的各种技术难题，生成代码，解决计算问题，并能提供算法设计与实现的指导。文本创作与编辑：Chat-GPT 能够帮助用户完成各类文本创作，进行论文写作、文案创作、内容优化，提供语法、结构和流畅性的修改建议，使文章更加规范和易读。数据分析与报告生成：通过对输入的数据进行分析，Chat-GPT 能帮助用户进行数据可视化、趋势分析、报告撰写等，为决策提供支持。学术与科研支持：Chat-GPT 具有为学术研究提供辅助的能力，如文献综述、研究问题的探讨、实验设计、模型建立等，帮助用户优化研究方向与方法。教育与学习辅助：对于语言学习、学术学习、职业技能学习，Chat-GPT 提供个性化的学习支持，帮助用户进行语言翻译、解答学术问题、提供课程建议。

除了自然语言文本的生成之外，GPT 还能够根据用户需求，自动生成符合标准的代码片段，并支持多种编程语言，如 Python、C++、Java、JavaScript 等。通过简明的描述，用户可以得到针对特定问题的解决方案或算法实现，并且可以进一步优化代码结构和性能。Chat-GPT 还能够在用户提出修改要求时，快速调整生成的代码，提供调试建议，帮助用户理解代码逻辑、找到潜在问题，并为复杂问题提供高效的算法或数据结构实现。对于学习编程的用户，Chat-GPT 不仅可以提供代码示例，还能帮助用户理解不同算法的实现原理和优缺点。

《算法设计与分析》是计算机科学与技术专业的核心课程之一，旨在教授学生如何设计高效的算法，以及如何分析算法的时间复杂度和空间复杂度。这门课程不仅涵盖了算法的基本理论，还重点讲解如何将算法应用到实际问题中，并在不断变化的技术环境中优化算法性能。课程的核心内容包括经典的排序算法、查找算法、图算法、动态规划、分治法、贪心算法、回溯算法等，还会深入探讨算法的设计技巧和不同算法的适用场景。

在计算机专业中，《算法设计与分析》是基础课程之一，也是后续高级课程和实际项目的基石。它对于学生理解计算机系统和软件工程有着至关重要的作用。无论是从理论的角度，还是从实践的角度，这门课程都在计算机领域占据着举足轻重的地位。基于 Chat-GPT 的编程能力，它有在《算法设计与分析》课程中有发挥重要的辅助作用的潜力。对于教师而言，Chat-GPT 可以协助课程内容的设计和组织的最新研究成果、教材内容的拓展、以及教学内容的多样化。此外，教师在准备课程时，Chat-GPT 可以帮助编写理论讲解的文本、生成算法的讲解示例，甚至可以根据课程安排设计习题和编程练习题，丰富课程内容。在编程实践方面，Chat-GPT 能为学生提供实时的代码生成与调试支持。学生在进行算法实践时，遇到问题或编程难题时，Chat-GPT 可以迅速提供算法的实现代码，帮助学生理解不同算法的实现细节与复杂度分析。学生还可以通过与 Chat-GPT 的互动，深入理解数据结构与算法的应用场景，掌握常见的优化技巧与编程实践。此外，Chat-GPT 可以在学生自主学习时，通过自动化

答疑和生成针对性的学习材料，如算法推导过程、代码实现的优化方法等，帮助学生更好地理解并掌握算法设计与分析的核心知识，增强理论学习与编程实践的结合。

### 2.2.1.2 Google Gemini

Gemini[28] 是 Google AI 开发的一款大型语言模型，代表了当前自然语言处理技术的顶尖水平。它经过了海量文本数据的训练，能够理解和生成人类语言，并执行各种语言相关的任务。相比于早期的语言模型，Gemini 在理解上下文、生成多样化文本、执行复杂推理等方面有了显著的提升。Gemini 的发展历程可以追溯到 Google 长期以来在机器学习和自然语言处理方面的深厚积累。通过不断迭代和优化，Gemini 在模型架构、训练数据、算法等方面都取得了突破性的进展。其核心功能包括：文本生成、问答系统、机器翻译、文本摘要、代码生成等。虽然 Gemini 和 Chat-GPT 都是强大的语言模型，但它们在架构、训练数据、侧重点等方面存在一些差异。Gemini 在多模态方面具有更强的能力，能够处理图像、视频等多种形式的数​​据，而 Chat-GPT 声称自己更擅长文本生成和对话。此外，两者的训练数据和算法也有所不同，导致它们在某些任务上的表现会有差异。

Gemini 同样在编程领域展现出了强大的辅助能力。它能根据自然语言描述生成代码片段或完整的函数，大大减少了开发者的人工编码量。此外，Gemini 还具备代码补全、解释、调试和优化等功能。例如，当开发者遇到复杂的算法时，Gemini 可以提供详细的解释，帮助开发者深入理解。更进一步，Gemini 还能根据代码生成文档，提高代码的可读性和可维护性。通过这些功能，Gemini 能够帮助开发者更高效地完成编程任务，将更多精力集中在问题的解决上。同样，利用 Gemini 自然语言处理功能，进行问答和编程辅助，可以辅助教师和学生进行 < 算法 > 的理论教学与学习、算法设计、以及编程辅助。

### 2.2.1.3 DeepSeek

DeepSeek(深度求索)[29] 是一家聚焦人工智能与大模型技术的前沿科技公司，以推动自然语言处理 (NLP) 和通用人工智能 (AGI) 发展为核心使命。通过自主研发高性能大模型 (如 DeepSeek-V3、DeepSeek Coder 和 DeepSeek Math)，公司在代码生成、数学推理、长文本分析等领域实现突破。2025 年初，DeepSeek 因推出全球首个支持 200K 上下文窗口的 DeepSeek-V4 模型引发行业轰动，其多模态交互能力与工业级推理效率被广泛集成至金融、教育、医疗等领域，成为全球 AGI 赛道现象级企业。

DeepSeek 大模型以多模态理解、超长上下文支持 (128K→200K) 为核心优势，采用创新的稀疏计算架构与混合训练框架，显著降低推理成本的同时提升生成质量。技术亮点包括：代码大模型：支持跨语言代码生成与全栈调试，效率超越同类工具；数学推理引擎：融合符号计算与神经网络，解决复杂工程问题；企业级文档智能：支持百页级 PDF/Excel 解析与结构化输出；开放生态：API 调用成本较主流模型降低 40%，吸引超 10 万开发者入驻。

## 2.2.2 生成式编程模型

生成式编程模型也叫 AI 编程助手，通常不作为独立工具使用，而是作为集成开发环境 (IDE) 的插件，增强编译器界面的功能，提供各种基于人工智能的编程辅助。这些助手能够理解用户的按键输入，并自动提供后续代码建议，实现代码的自动补全和生成。市面上涌现的很多编程助手许多都带有各自的特色，例如 iflycode[30] 的语音输入、文心快码 [31] 支持的超 100 种编程语言、CodeWhisperer 与亚马逊 AWS 深度集成的 Amazon Q Developer[32] 等，这里只介绍普遍使用且具有代表性的两个生成式编程模型。

### 2.2.2.1 CodeGeeX

CodeGeeX[33] 是一款免费开源的生成式编程模型，专为满足项目特色需求而设计。它兼容 Python、Java 和 C++ 等多种编程语言，能够将自然语言指令转换为相应代码，特别适合需要开源工具灵活性和自定义开发的技术开发者。虽然处理复杂逻辑时准确度可能有所下降，但在主流语言中仍能提供高质量的代码生成。其核心功能包括代码生成、智能补全、代码注释、代码翻译、代码修复、单元测试生成和智能问答，显著提升了开发效率和代码质量。特色功能如代码重构、交互式编程、代码审查、Commit 生成和幽灵注释等，进一步扩展了其应用广度和深度。此外，通过集成工具箱 Code Interpreter，CodeGeeX 还提供了批量文件处理、数据分析与可视化、数学函数绘图、代码分析及多格式文件修改等实用功能，大幅提升了工作效率。



Figure 2.1: 在 Pycharm 以插件形式的 CodeGeeX 交互界面

CodeGeeX 与 PyCharm 的集成通过插件形式实现 (如图2.1)，为开发者提供了便捷的交互界面。作为社区驱动项目，其更新频率和功能扩展依赖于社区贡献，适合个人开发者、企业团队以及教育场景。在教学应用中，CodeGeeX 的问题解释、算法代码生成、自动注释、代码优化和 Debug 等功能，能够帮助学生理解编程概念、减少编码挫败感并快速定位问题，成为编程教育的强大辅助工具。尽管复杂逻辑可能仍需人工干预，但 CodeGeeX 显著提高了开发效率和代码质量，同时使编程教育更加个性化和高效。

### 2.2.2.2 Trae

Trae[34] 是一款功能强大的编程助手，集成了自动代码补全、代码生成等基础功能，并进一步扩展到调试支持、个性化学习建议、单元测试生成等高级特性。这款工具旨在通过智能化的辅助功能，提升开发者的编程效率和代码质量，同时降低编程的复杂性，让开发者能够更加专注于创新和解决问题。2025 年更名问 Trae 插件，如图 2.2 所示。

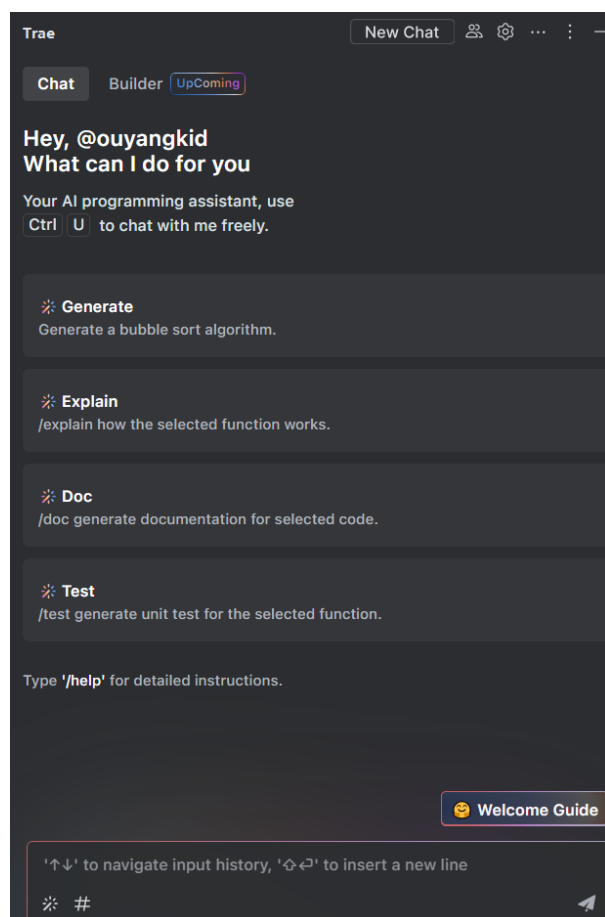


Figure 2.2: 在 Pycharm 以插件形式的 Trae 交互界面

Trae 在辅助代码生成的同时，提供强大的调试支持和问题定位功能。其内置的智能分析引擎能够实时分析代码，迅速识别出潜在的错误和不良编程实践，从而帮助用户快速定位并解决问题。此外，根据用户的编程经验和历史修复行为，Trae 能够提供定制化的错误修复建议，这些建议覆盖了代码修复、设计模式优化以及架构调整等多个层面。Trae 极大地简化了单元测试的创建过程，通过其一键生成单元测试的功能，用户可以轻松地选中代码片段自动创建对应的测试用例，显著减少了编写测试代码的工作量。此外，Trae 不仅止步于代码生成，它还能对这些测试用例进行文字总结，快速向用户展示测试覆盖的范围和结果，这对于编写测试报告和进行代码审查来说是一个宝贵的工具，确保了测试过程的透明度和效率。Trae 的代码补全 Pro 功能通过智能分析现有代码，自动预测下一个可能的改动点，并提供相应的推荐代码，极大地提升了代码修改和重构的效率。与传统的代码补全工具相比，它减少了对用户手动输入提示或关键词的

依赖，使得整个代码补全过程更加流畅和自然，从而为用户提供了一种更加直观和高效的编程体验。Trae 在主流编程语言中展现出卓越的代码生成稳定性，特别是在实现复杂算法时，能够提供完整且准确的代码建议。尽管面对一些新兴编程语言或特定领域的框架时，其准确性可能会有所下降，但 Trae 依然能够满足需要多语言支持的开发者的需求，特别是在处理复杂项目和代码优化方面，它仍然是一个强有力的助手。

通过智能化的功能，Trae 能够适应不同专业背景和编程能力参差不齐的学生群体，为他们提供个性化的学习体验。特别是单元测试后的总结，可以帮助学生理解算法的特殊情况，加深对算法和算法边界的理解。Trae 改善了教学和编程实践过程，通过其强大的调试支持和问题定位功能，帮助学生快速识别并解决编程中的错误，从而加深对编程概念的理解。

## 2.3 大语言模型交互技巧

本模块介绍一些与 AI 大语言模型交互进行算法编程的注意事项，以及 IDE 中可安装 AI 插件的功能介绍。

### 2.3.0.1 通用编程 Prompt 经验

1. 使用算法设计中的专用名词，如：时间复杂度、空间复杂度、最优性保证、内部排序、外部排序、数据结构类型 (列表、数组、链表、树、图) 等；2. 问题定义明确，定义输入输出要求、指出可能的性能瓶颈了；3. 对复杂问题进行按需拆分，逐条列出子任务/问题；4. 算法对比提问，在适用场景 x 中比较算法 1 和算法 2，使用表格形式展示对比维度，使用-来定义对比维度；5. 明确时间/空间复杂度要求，指定算法范式 (如分治、递归、动态规划)；6. 考虑算法的实现指标：如递归调用次数限制、内存使用模式、缓存友好性、并行化潜力；7. 需求拆分后先生成伪代码，检查伪代码逻辑后再生成相应代码；8. 列出所有可能的越界访问、边界测试情况；9. 内存使用分析考虑：峰值内存、不同规模下的增长趋势分析、优化方案；10. 先生成代码注释，在优化代码或进行测试，能够提升改进效率。11. 信息密度控制：每段 Prompt 包含 1 个核心指令 +1 个示例 +1 个约束条件。12. 采用 APEX 原则进行 prompt 描述，即 A(Action)：明确动词 (“生成”、“优化”、“调试”)；P(Parameter)：量化参数 (时间复杂度  $O(n)$ 、内存限制 1GB)；E(Example)：最少一个示例；X(X-factor)：独特约束 (“不用递归”、“仅用标准库”)。

### 2.3.0.2 CodeGeeX 内置指令

AI 插件内置的固定功能，可实现对代码编辑栏 (Workspace) 中的代码选择后完成相应操作，通过功能向导可查阅详细功能。以下介绍基于 win10 操作系统下的 pycharm。

- `/explain`：解释编辑器中选中的代码。若没有选择代码，则解释全部代码。

- `/comment`: 为编辑器中选中的代码添加逐行注释。若没有选中代码, 则对所有代码添加注释。
- `/fixbug`: 修复编辑器中选中的代码中的错误。若没有选中代码, 则对所有代码进行修复。
- `/tests`: 为编辑器中选中的代码生成单元测试代码。
- `/review`: 为编辑器中选中的代码生成代码审查报告。
- `@file`: 从当前打开的文件中选一个进行提问, 默认选择当前文件。
- `@recentFiles`: 可以同时多文件提问, 会默认引用打开的所有文件和最近查看的文件。
- `@repo`: 对 git 常用仓库进行提问。

### 2.3.0.3 Trae 内置指令

- `#workspace`: 引用整个工作空间内的所有代码文件, 基于相应内容进行 AI 交互。
- `#file`: 引用指定的文件名, 基于相应内容进行 AI 交互。
- `#code`: 引用指定代码片段, 基于相应内容进行 AI 交互。
- `Tab`: 在对话框中使用后进行代码自动补齐。
- `Shift + Tab`: 接受当前第一条推荐生成结果。
- `ESC`: 放弃生成的代码建议。
- `Ctrl + Shift + Enter`: 激活高级代码补全功能 (Pro 功能默认关闭, 耗时更长)。
- `/help`: 选定代码后, 生成帮助信息。
- `/test`: 选定代码后, 生成单元测试。
- `/doc`: 选定代码后, 生成代码注释。
- `/fix`: 选定代码后, 修复代码。

## 2.4 交互编程问题的形式化规约

AI 辅助的编程又可以被定义为文本-代码 (Text-to-code) 生成问题, 也就是由使用者通过对问题描述的理解, 撰写自然语言描述文本作为大语言模型输入, 进而引导模型产生相应代码, 并通过测试用例的测试。上述过程可以被定义为  $P = \langle Q, T_v, T_h \rangle$ , 其中  $Q$  为用自然语言描述的问题规约,  $T_v$  和  $T_h$  分别为可见和隐式测试, 每个包含一组输入-输出  $(t^i, t^o) \in T = T_u \cup T_h$ 。该过程的目标是希望学生能够利用大语言模型  $M$  生成一个程序  $f$ ,  $M \rightarrow f$ , 能够处理所有来自  $T$  中的输入  $t^i$  并得到对应的输出  $t^o$ , 也就是  $f(t^i) = t^o$ , 对于所有满足的  $(t^i, t^o) \in T$ , 其中包含的隐式测试  $T_h$  在整个过程对用户 (即学生) 不可见, 当且仅当由大语言模型生成的  $f$  能够通过所有  $T_v$  的测试。

在 AI 辅助的算法设计学习中, 可以班级为单位考察多个学生在大语言模型协同下完成同一/多个算法习题的通过率, 作为人在环交互的 AI 辅助算法学习的一个度量维度。衡量最终通过学习, 学生可以将算法和 AI 辅助交互相结合的实际应用效果。



## 2.5 量化评估指标

首次通过准确率 (Pass@1 accuracy): 是评估代码生成模型性能的核心指标, 特指模型首次生成代码即通过所有测试用例的概率。一般用于低时延下的实时运行评估。

前  $k$  次生成通过率 (Pass@ $k$ ): 表示模型生成  $k$  个候选代码, 其中至少有一个能通过测试用例单元测试的概率, 是评估模型在多次 ( $k$  次) 尝试下, 生成有效代码的能力。

平均首次通过率 (Mean Pass@1 Rate): 是评估大语言模型代码生成性能的核心指标, 定义了给定一组提示的情况下, 模型首次生成的代码解决方案中能通过所有单元测试的比例的算术平均值, 公式如 Eq. 2.1。其中,  $N$  为测试提示的总数;  $\mathbb{I}$  为指示函数, 通过为 1, 否则为 0;  $y_i$  和  $\hat{y}_i$  为第  $i$  个样本的真实结果和预测结果, 当预测结果可以通过测试时两者相等。

$$\text{mean pass@1} = \frac{1}{N} \sum_{i=1}^N \mathbb{I}(y_i = \hat{y}_i) \quad (2.1)$$

因此, 在针对个人样本评估时, 可采用首次通过准确率和前  $k$  次生成通过率; 当针对多名学生进行整体评估时, 则可以采用平均首次通过率指标。

## 2.6 章节小结

由于网络访问、以及跨境支付的报销问题, 虽然市面上有诸多新型公司推出的大量生成式对话大模型, 和大量的 AI 辅助编程工具; 但综合考虑学校访问, 为每个志愿者进行账号注册的难度, 研究的时间、对话过程的可分享性, 我们仅考虑了有限的工具尝试, 这虽然会对本研究产生一定的限制, 但考虑研究处于相对初级阶段, 重点还是探索 AI 在编程辅助教学, 以及在算法进阶课程上的可用性。

### 2.6.0.1 引入多模型原因

正如在第一章说描述的, 现有的大语言模型具有显著的局限性, 通常情况下具备健全背景知识的领域专家能够辨别此类模型的错误和问题, 从而通过修改问题边界、给出新的条件、以及替换文字描述等方式驱动大模型进行优化。而初始学习的学生经验缺失, 潜在可能盲目轻信 AI 模型的输出结果。

引入多个模型处理相同的问题, 用户能够从不同的角度获得差异性的回答 (假设不同公司采用的模型架构、训练语料、限定条件存在差异), 通过对比多个不同模型的回答, 以及对同一问题的连续描述的修改和完善, 获得的回答进行对比和分析, 思考其中的差异, 以及这些差异与交互过程的关联, 可以达到以下目的: 1) 获得多样性回答, 受模型架构等因素的差异, AI 模型在处理用户输入的自然语言文本等内容时, 编码和计算过程存在差异, 会产生不同的回答; 2) 互补性结果, AI 模型的回复会随着交互过程产生偏向性回答, 部分模型以加入联网搜索功能, 在回复长度相对固定的情况下, 能够形成的差异性回答能够形成互补信息; 3) 提升结果的可靠性, 用户在接收到差异性回答

后会产生对比，并促进进一步的搜索和检验，避免受模型幻觉生成的虚构答案误导。此外，4) 模型差异性功能互补：生成式大模型发展至今已衍生出大量在特定任务上的微调，通过在不同的任务环节引入不同的模型，可以利用模型在特定数据集微调的优势，发挥不同模型在特定领域的，如本方案在对话学习使用 Chat-GPT/Gemini，但在编程辅助过程，则介绍 CodeGeeX 和 Trae 的使用方法。



# Chapter 3

## 算法设计与分析基础概念学习

### 3.1 算法设计与分析基础概念学习

本研究以排序问题`Sorting algorithm`为例，尝试在大语言模型辅助下完成对这一概念的学习，突破传统教科书讲述的固有框架。一方面，排序问题十分基础，是很多其它算法的组成部分 (如后续用例问题均)，也具有算法的原子性，属于最小算法单元。另一方面，排序算法种类较多，它依赖不同的数据结构来进行效率优化，原本分布在整个算法课程的多个部分较为离散，很少在学习初期进行横向对比学习。此外，排序算法常用于引出算法设计与分析常用的时间复杂度、空间复杂度、递归、算法稳定性、串并行、适应性等基础概念。

为了进一步简化对比学习的难度，本研究仅考虑采用以数组为数据对象的比较排序算法，相关算法及其简要对比如表 3.1所示。

Table 3.1: 排序算法对比

算法名称	最佳	平均	最坏	空间占用	稳定性	是否就地	方法
块排序 Block sort	n	$n \log_n$	$n \log_n$	1	Yes		Insertion & Merging
冒泡排序 Bubble sort	n	$n^2$	$n^2$	1	Yes	Yes	Exchanging
交换排序 Exchange sort	$n^2$	$n^2$	$n^2$	1	No		Exchanging
堆排序 Heapsort	$n \log_n$	$n \log_n$	$n \log_n$	1	No	Yes	Selection
插入排序 Insertion sort	n	$n^2$	$n^2$	1	Yes	Yes	Insertion
内排序 Introsort	$n \log_n$	$n \log_n$	$n \log_n$	$n \log_n$	No		Partitioning & Selection
归并排序 Merge sort	$n \log_n$	$n \log_n$	$n \log_n$	n	Yes	No	Merging
奇偶排序 Odd-even sort	n	$n^2$	$n^2$	1	Yes		Exchanging
快速排序 Quicksort	$n \log_n$	$n \log_n$	$n^2$	1	No	No	Partitioning
选择排序 Selection sort	$n^2$	$n^2$	$n^2$	1	No	Yes	Selection
希尔排序 Shellsort	$n \log_n$	$n^{\frac{4}{3}}$	$n^{\frac{3}{2}}$	1	No		Insertion

## 3.2 学习顺序

通过将上述表格输入 deepseek，询问如果需要顺序学习排序算法，如何开始？获得以下反馈。

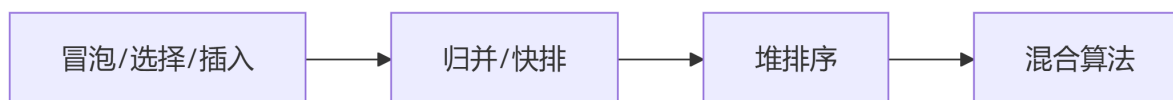


Figure 3.1: 学习顺序

第一阶段：基础排序算法 (理解核心思想)

冒泡排序 (Bubble Sort) 理由：最直观的排序思想，帮助理解”比较交换”机制。学习重点：双重循环结构、相邻元素比较、优化策略 (提前终止)。

选择排序 (Selection Sort) 理由：引入”选择极值”的思想学习重点：线性扫描找最小值、不稳定性的原因 (跨位置交换)。

插入排序 (Insertion Sort) 理由：理解”增量排序”思想，为高级算法奠基学习重点：元素插入操作、对部分有序数据的高效性

第二阶段：中级排序算法 (掌握分治思想)

归并排序 (Merge Sort) 理由：经典分治算法代表学习重点：递归分治策略、合并两个有序数组的技巧、空间复杂度分析

快速排序 (Quick Sort) 理由：最常用的高效排序学习重点：分区 (Partition) 操作、随机化优化、递归栈空间理解

希尔排序 (Shell Sort) 理由：插入排序的进阶版学习重点：递减增量序列、跨元素比较思想

第三阶段：高级排序算法 (理解特殊优化)

堆排序 (Heap Sort) 理由：树形选择排序学习重点：二叉堆构建、原地排序实现、优先级队列关联

块排序 (Block Sort) 推荐理由：归并排序的工程优化学习重点：分块策略、混合排序思想、内存访问优化

内省排序 (Intro Sort) 理由：快速排序的工业级实现学习重点：混合算法设计、递归深度检测、算法退化处理

以下学习将基于上述推荐顺序展开，同时为了防止大模型幻觉，本研究加入以下四方面的交叉确认机制：1. 对每个算法附上了 wikipedia 的官方链接；2. 推荐电子教材 <Introduction to Algorithms> the 4th edition 的 pdf；3. 相匹配的 stanford-cs161-master 课件；4. 在 pycharm 环境下进行编程实现和评估。其中，还将在最后一章详细介绍如何采用两个不同的 Pycharm 内置大模型插件进行交互编程。

## 3.2.1 冒泡排序

### 3.2.1.1 算法定义-wikipedia

冒泡排序 Bubble Sort，是一种简单的排序算法。它重复地走访过要排序的数列，一次比较两个元素，如果它们的顺序错误就把它们交换过来。走访数列的工作是重复地进行直到没有再需要交换，也就是说该数列已经排序完成。这个算法的名字由来是因为越小的元素会经由交换慢慢“浮”到数列的顶端。

### 3.2.1.2 算法分析

```
1 def bubble_sort(arr):
2     n = len(arr)
3     for i in range(n):          # 外层循环控制轮次
4         for j in range(0, n-i-1): # 内层循环比较相邻元素
5             if arr[j] > arr[j+1]:
6                 arr[j], arr[j+1] = arr[j+1], arr[j] # 交换
7     return arr
```

Listing 3.1: 冒泡排序

同时，为了分析时间复杂度，使用 python 下的 big\_o 库进行理论分析测试。对应表格中冒泡排序的最佳和最坏情况的时间复杂度分别是  $O(n)$  和  $O(n^2)$ ，实际对应的输入应该是已经排序好的输入序列和逆排序的情况，前者无需排序，仅需要遍历一轮数组顺序即可；而后者需要在两轮循环中将每相邻两个元素调换一次顺序。因此，首先通过以下代码片段生成已排序数组 *best\_case* 和逆排序数组 *worst\_case* 作为输入，分别对冒泡排序算法执行 100 次，并输出结果。

```
1 best_case = lambda n: sorted(range(n)) # 最优情况，数组已经有序
2 worst_case = lambda n: sorted(range(n), reverse=True) # 最坏情况，数
   组倒序
3
4 # 分析时间复杂度
5 print("最优情况(已排序数组): ")
6 best, _ = big_o(bubble_sort, best_case, n_measures=10, min_n=10, max_n
   =100)
7 print(best) # 期望输出: O(n)(优化后)或 O(n^2)(未优化)
8
9 print("\n最坏情况(逆序数组): ")
10 worst, _ = big_o(bubble_sort, worst_case, n_measures=10, min_n=10,
   max_n=100)
11 print(worst) # 期望输出: O(n^2)
```

Listing 3.2: 时间复杂度测试用例

经过等待 (单线程测试), Pycharm 终端打印结果如下图 3.2所示, 结果与分析一致。其中, 最优情况和最差情况随机用 10-100 长度的随机数组测试 10 次, 最优情况的执行时间为 0.000096s, 最差情况执行的时间为 0.00021s, 相差约 2 倍。

```

最优情况 (已排序数组):
Constant: time = 9.6E-05 (sec)

最坏情况 (逆序数组):
Constant: time = 0.00021 (sec)

```

Figure 3.2: 冒泡排序的时间复杂度测试。

接着使用 memory\_profiler 分析算法的空间复杂度, 采用如下推荐代码片, 即通过长度为 100、500 和 1000 的逆序数组测试 bubble\_sort()。

```

1 for n in [100, 500, 1000]:
2     arr = list(range(n, 0, -1))
3     bubble_sort(arr.copy())

```

Listing 3.3: 空间复杂度测试用例

以 500 长度的顺序数组测试结果为例, 如图 3.3所示, 可以看到算法在初期占用 61.4MB 后, 后续代码执行的 Increment=0, 也就是排序过程无需占用额外的空间存储中间变量, 满足表 3.1中空间占用  $O(1)$ , 而代码的 Line6 仅涉及交换操作 (Exchanging)。

Filename: H:\中法\基金申请\2024教改项目\算法设计与分析\code\pythonProject\BubbleSort.py

Line #	Mem usage	Increment	Occurrences	Line Contents
5	61.4 MiB	61.4 MiB	1	@profile
6				def bubble_sort(arr):
7	61.4 MiB	0.0 MiB	1	n = len(arr) # 获取数组长度
8	61.4 MiB	0.0 MiB	501	for i in range(n): # 外层循环控制轮次
9	61.4 MiB	0.0 MiB	125250	for j in range(0, n - i - 1): # 内层循环比较相邻元素
10	61.4 MiB	0.0 MiB	124750	if arr[j] > arr[j + 1]:
11	61.4 MiB	0.0 MiB	124750	arr[j], arr[j + 1] = arr[j + 1], arr[j] # 交换
12	61.4 MiB	0.0 MiB	1	return arr # 返回排序后的数组

Figure 3.3: 冒泡排序的内存测试。

此外, 模型还给出通过增加一个 Flag 变量来记录每次遍历是否进行了数据交换的早停版本, 代码如下。通过初始化布尔变量 swapped(Line4), 并在发生交换时修改其值 (Line8), 以及在每轮 i 遍历最后进行状态变换判断 (Line9-10) 实现早停。

```

1 def bubble_sort_optimized(arr):
2     n = len(arr)
3     for i in range(n):
4         swapped = False # 标记本轮是否发生交换
5         for j in range(0, n-i-1):
6             if arr[j] > arr[j+1]:

```

```

7         arr[j], arr[j+1] = arr[j+1], arr[j]
8         swapped = True
9         if not swapped: # 无交换则提前退出
10            break
11    return arr

```

Listing 3.4: 早停冒泡排序

由于早停机制触发依赖于测试用例中包含已排序好连续子区间，本研究将使用 `pytest` 和 `pytest-benchmark` 两个库进行两个版本的冒泡排序算法的评估，采用如下测试用例。其中，使用长度为 200 的随机数组进行测试。

```

1 @pytest.mark.parametrize("sort_func", [bubble_sort,
    bubble_sort_optimized])
2 def test_bubble_sort(benchmark, sort_func):
3     arr = list(range(200)) # 小规模数据
4     shuffle(arr) # 随机打乱
5     benchmark(sort_func, arr.copy())

```

Listing 3.5: 冒泡排序对比测试用例

测试需要依赖终端输入命令 `pytest BubbleSort.py -v`，结果局部截图如下图 3.4，可以看到增加早停机制能够节约大量时间。不采取早停机制的情况下原算法无论顺序如何都进行对比，最小、最大和平均时间一样长；而增加早停机制后平均时间降至几乎最小值。

Name	(time in us)	Outliers	OPS	Rounds	Min	Max	Mean
-----							
test_bubble_sort[bubble_sort_optimized]					9.2000 (1.0)	29.8000 (1.0)	9.5055 (1.0)
	14;35	105,202.3321 (1.0)		619	1		
test_bubble_sort[bubble_sort]					463,156.3000 (>1000.0)	467,260.1000 (>1000.0)	464,867.3000 (>1000.0)
	.0)	2;0	2.1512 (0.00)	5	1		

Figure 3.4: 冒泡排序早停对比测试。

## 3.2.2 选择排序

### 3.2.2.1 算法定义-wikipedia

选择排序 Selection sort 是一种简单直观的排序算法。它的工作原理如下，首先在未排序序列中找到最小 (大) 元素，存放到排序序列的起始位置，然后，再从剩余未排序元素中继续寻找最小 (大) 元素，然后放到已排序序列的末尾。以此类推，直到所有元素均排序完毕。选择排序的主要优点与数据移动有关。如果某个元素位于正确的最终位置上，则它不会被移动。选择排序每次交换一对元素，它们当中至少有一个将被移到其最终位置上，因此对  $n$  个元素的表进行排序总共进行至多  $n - 1$  次交换。在所有的完全依靠交换去移动元素的排序方法中，选择排序属于非常好的一种。

### 3.2.2.2 算法分析

选择排序通过不断选择剩余元素中的最小值，并将其放到已排序部分的末尾。其中，大语言模型分别给出两个版本，仅考虑最小值的 `selection_sort` 版本和同时考虑最大、最小值的 `selection_sort_optimized` 版本。

```
1 def selection_sort(arr):
2     n = len(arr)
3     for i in range(n):
4         min_idx = i # 假设当前位置是最小值
5         for j in range(i+1, n):
6             if arr[j] < arr[min_idx]:
7                 min_idx = j # 更新最小值索引
8         arr[i], arr[min_idx] = arr[min_idx], arr[i] # 交换
9     return arr
```

Listing 3.6: 选择排序

```
1 def selection_sort_optimized(arr):
2     n = len(arr)
3     for i in range(n//2):
4         min_idx, max_idx = i, i
5         for j in range(i, n-i):
6             if arr[j] < arr[min_idx]:
7                 min_idx = j
8             if arr[j] > arr[max_idx]:
9                 max_idx = j
10        # 将最小值放到前面
11        arr[i], arr[min_idx] = arr[min_idx], arr[i]
12        # 如果最大值被移动过，需要修正索引
13        if max_idx == i:
14            max_idx = min_idx
15        # 将最大值放到后面
16        arr[n-i-1], arr[max_idx] = arr[max_idx], arr[n-i-1]
17    return arr
```

Listing 3.7: 选择排序改进

两个版本的理论时空复杂度一样，因为没有早停机制，以及任何情况下均需要进行所有数据对的比较。同时引入对最大值的计算可以减少最外层的循环次数和交换次数。

进一步使用 `Big_o` 进行测试，采用与前面相同的用例分别对两个版本进行测试，执行结果如图 3.5 所示。可以看到两组最坏和最优情况下的花费十分接近，也均为  $O(n^2)$ 。

空间复杂度对比分析如图 3.6，截图仅显示采用长度为 100 的逆序数组（最坏情况），可以看到：首先，两个算法在初始均占用 66MB 内存，之后 `Increment` 为 0，即无额外

Table 3.2: 选择排序两个版本对比

版本	时间复杂度	空间复杂度	交换次数 (最坏)	比较次数 (固定)
基础版	$O(n^2)$	$O(1)$	$n-1$	$n(n-1)/2$
优化版	$O(n^2)$	$O(1)$	$n//2$	$n^2/2$

```

普通版最优情况（已排序数组）：
Quadratic: time = -0.00068 + 2.2E-08*n^2 (sec)

普通版最坏情况（逆序数组）：
Quadratic: time = 0.0015 + 2.3E-08*n^2 (sec)

改进版最优情况（已排序数组）：
Quadratic: time = 0.00073 + 1.8E-08*n^2 (sec)

改进版最坏情况（逆序数组）：
Quadratic: time = -2.8E-05 + 1.8E-08*n^2 (sec)

```

Figure 3.5: 选择排序对比测试。

占用情况；接着，对比普通版本 (上) 的外层循环为 101，而改进版 (下) 以及其内的循环次数均为  $1/2$ ，比较次数不变均为 2500，交换次数同样降为 50，以前表 3.2 分析一致。

最后，基于冒泡排序可以通过早停机制减少对比次数，想大模型提问是否能为选择排序增加早停机制？回答为否定：标准选择排序无法有效加入早停机制，因为它的核心逻辑要求必须遍历所有未排序元素才能确定最小值。潜在的改进有两种：1. 减少不必要的交换 (但比较次数不变)。不影响时间复杂度仍为  $O(n^2)$ ，仅优化常数因子。2. 如果剩余部分已经有序，可以提前终止。但额外检查会增加比较次数，仅对部分有序数据有轻微优化。

结论：仅在数据接近有序时可能有轻微优化。随机/逆序数据无改进，甚至可能更慢，因为额外检查增加了开销，因此不采用上述方案进行算法改进。

### 3.2.3 插入排序

#### 3.2.3.1 算法定义-wikipedia

插入排序 (Insertion Sort) 是一种简单直观的排序算法。它的工作原理是通过构建有序序列，对于未排序数据，在已排序序列中从后向前扫描，找到相应位置并插入。插入排序在实现上，通常采用 in-place 排序 (即只需用到  $O(1)$  的额外空间的排序)，因而在从后向前扫描过程中，需要反复把已排序元素逐步向后挪位，为最新元素提供插入空间。



Line #	Mem usage	Increment	Occurrences	Line Contents
6	66.0 MiB	66.0 MiB	1	@profile
7				def selection_sort(arr):
8	66.0 MiB	0.0 MiB	1	n = len(arr)
9	66.0 MiB	0.0 MiB	101	for i in range(n):
10	66.0 MiB	0.0 MiB	100	min_idx = i # 假设
11	66.0 MiB	0.0 MiB	5050	for j in range(i+1
12	66.0 MiB	0.0 MiB	4950	if arr[j] < ar
13	66.0 MiB	0.0 MiB	2500	min_idx =
14	66.0 MiB	0.0 MiB	100	arr[i], arr[min_id
15	66.0 MiB	0.0 MiB	1	return arr
Filename: H:\中法\基金申请\2024教改项目\算法设计与分析\code\pythonProject\Sel				
Line #	Mem usage	Increment	Occurrences	Line Contents
16	66.0 MiB	66.0 MiB	1	@profile
17				
18				def selection_sort_optimiz
19	66.0 MiB	0.0 MiB	1	n = len(arr)
20	66.0 MiB	0.0 MiB	51	for i in range(n//2):
21	66.0 MiB	0.0 MiB	50	min_idx, max_idx =
22	66.0 MiB	0.0 MiB	2600	for j in range(i,
23	66.0 MiB	0.0 MiB	2550	if arr[j] < ar
24	66.0 MiB	0.0 MiB	2500	min_idx =
25	66.0 MiB	0.0 MiB	2550	if arr[j] > ar
26				max_idx =
27	66.0 MiB	0.0 MiB	50	arr[i], arr[min_id
28	66.0 MiB	0.0 MiB	50	if max_idx == i:

Figure 3.6: 选择排序空间占用对比测试。

### 3.2.3.2 插入排序算法流程

1. 从第一个元素开始，该元素可以认为已经被排序
2. 取出下一个元素，在已经排序的元素序列中从后向前扫描
3. 如果该元素 (已排序) 大于新元素，将该元素移到下一位置
4. 重复步骤 3，直到找到已排序的元素小于或者等于新元素的位置
5. 将新元素插入到该位置后
6. 重复步骤 2 ~ 5

### 3.2.3.3 插入排序伪代码

插入排序的伪代码如算法 1 所示。  
值得注意的是，Chat-GPT 等工具同样具备对伪代码的修改功能。



---

**Algorithm 1** INSERTION-SORT

---

**Require:** 输入参数 1: 待排序数组  $A$

输入参数 2: 数组长度  $n$

**Ensure:** 排序后的数组  $A$

```
for  $i = 2$  to  $n$  do
     $key = A[i]$ 
    while  $j > 0$  and  $A[j] > key$  do
         $A[j+1] = A[j]$ 
         $j = j - 1$ 
    end while
     $A[j+1] = key$ 
end for
return  $A$ 
```

---

### 3.2.3.4 插入排序算法定义-GPT

以下定义由 GPT 生成:

插入排序 (insertion sort) 是一种基于比较的排序算法, 它通过构建一个部分有序的数组, 将未排序部分的每个元素插入到有序部分的适当位置, 从而逐步完成整个数组的排序。

**详细定义: 初始状态:** 将数组划分为两部分, 其中第一部分包含一个元素, 默认为有序, 其余部分为未排序。

**核心操作:** 从未排序部分逐个取出元素, 与有序部分的元素从后向前进行比较, 直到找到合适的位置并插入。

**结束状态:** 当所有未排序的元素都插入到有序部分后, 整个数组变为有序。

算法性质:

1. 稳定性: 插入排序是稳定的排序算法, 保持相同元素的相对顺序。

2. 时间复杂度:

(a) 最佳情况 (数组已排序):  $O(n)$ 。

(b) 最差情况 (数组逆序):  $O(n^2)$ 。

(c) 平均情况:  $O(n^2)$ 。

3. 空间复杂度:  $O(1)$ , 该算法为**原地排序**, 不需要额外的存储空间。

插入排序的优点: 对小规模数据效果较好; 对于部分已排序的数据具有较高的效率。

### 3.2.3.5 插入排序算法代码-GPT

以下 python 代码由 GPT 生成: 使用 [Prompt]: 生成插入排序的 python 代码。

```

1 def insert_sort(arr):
2     for i in range(1, len(arr)):
3         key = arr[i]
4         j = i - 1
5         while j >= 0 and key < arr[j]:
6             arr[j + 1] = arr[j]
7             j = j - 1
8         arr[j + 1] = key
9     return arr

```

Listing 3.8: 插入排序

在这段代码中，本研究首先从数组的第二个元素开始遍历，因为第一个元素已经被认为是排序好的。对于每个元素，本研究将其设为关键字，并与前一个元素进行比较。如果前一个元素大于关键字，本研究就将前一个元素向后移动一位，并继续与前一个元素进行比较，直到找到关键字应该插入的位置。最后，本研究将关键字插入到正确的位置，并返回排序后的数组。同时，交互 AI 还会提供一个测试用例，以及调用语句，如下：

```

1 arr = [3, 1, 4, 1, 5, 9, 2, 6, 5, 3, 5]
2 sorted_arr = insert_sort(arr)
3 print(sorted_arr)

```

Listing 3.9: 插入排序-测试用例

使用 [Prompt]: 为插入排序代码进行注释。

使用 [Prompt]: 使用 @file \*.py，指定具体需要注释的代码文件。

```

1 def insert_sort(arr):
2     # 遍历数组中的每个元素，从第二个元素开始
3     for i in range(1, len(arr)):
4         # 将当前元素设为关键字
5         key = arr[i]
6         # 将当前元素的索引减一，得到前一个元素的索引
7         j = i - 1
8         # 当前一个元素大于关键字且前一个元素的索引大于等于0时
9         while j >= 0 and key < arr[j]:
10            # 将前一个元素向后移动一位
11            arr[j + 1] = arr[j]
12            # 将前一个元素的索引减一
13            j -= 1
14            # 将关键字插入到正确的位置
15            arr[j + 1] = key
16        # 返回排序后的数组
17    return arr

```

Listing 3.10: 插入排序 (注释)

插入排序可通过二分查找，即将数组从中间一分为二 (Line7)，分别扫描将比较次数降低，如下：

```
1 def insertion_sort_optimized(arr):
2     for i in range(1, len(arr)):
3         key = arr[i]
4         left, right = 0, i - 1
5         # 二分查找插入位置(减少比较次数)
6         while left <= right:
7             mid = (left + right) // 2
8             if arr[mid] < key:
9                 left = mid + 1
10            else:
11                right = mid - 1
12            # 整体后移
13            arr[right+2:i+1] = arr[right+1:i]
14            arr[right+1] = key
15        return arr
```

Listing 3.11: 插入排序改进

两个算法的对比如表 3.3所示。

Table 3.3: 插入排序两个版本对比

对比维度	基础版	优化版
查找插入位置	线性扫描 $O(n)$	二分查找 $O(\log_n)$
元素移动方式	逐个后移 (while 循环)	整体切片赋值 (批量移动)
比较次数	最多 $i-1$ 次 ( $i$ 为当前轮次)	最多 $\log_2(i)$ 次
赋值操作次数	较多 (每次移动需单独赋值)	较少 (批量移动)
代码复杂度	简单直观	需理解二分查找边界
适用场景	小规模数据或链表排序	大规模数据 (但仍有 $O(n^2)$ 限制)

通过以下代码块分别测试数组规模为 [100, 1000, 5000] 三种不同情况下两组代码的执行时间，结果如下图 3.7所示。但输入规模较小时，优化版本提升 62.6% 的计算效率，规模增长到 1000 和 5000 时，则能够提升 90%-94.3% 的计算效率，主要得益于二分查找的时间复杂度为  $O(\log_n)$ 。由于数组规模过大时，移动操作占比将增加，优化效率减弱。

```
1 def test_performance():
2     sizes = [100, 1000, 5000] # 不同数据规模
3     for size in sizes:
4         arr = random.sample(range(size * 10), size)
5
6         # 测试基础版
```

```

7     time_basic = timeit.timeit(
8         lambda: insertion_sort(arr.copy()),
9         number=10
10    )
11
12    # 测试优化版
13    time_optimized = timeit.timeit(
14        lambda: insertion_sort_optimized(arr.copy()),
15        number=10
16    )
17
18    print(f"数据规模: {size}")
19    print(f"基础版: {time_basic:.5f}s")
20    print(f"优化版: {time_optimized:.5f}s")
21    print(f"优化幅度: {(1 - time_optimized/time_basic)*100:.1f}%\n")

```

Listing 3.12: 插入排序测试用例

```

InsertionSort.py::te
数据规模: 100
基础版: 0.00187s
优化版: 0.00070s
优化幅度: 62.6%

数据规模: 1000
基础版: 0.18740s
优化版: 0.01880s
优化幅度: 90.0%

数据规模: 5000
基础版: 4.98149s
优化版: 0.28475s
优化幅度: 94.3%

```

Figure 3.7: 选择排序对比测试。

剩余中高级排序算法均可沿用上述方案进行组合学习，由于内容重复不在此赘述。本研究将在后续章节使用剩余更复杂排序算法开展 AI 辅助的交互学习研究。

### 3.3 章节小结

本章节利用大语言模型辅助分析了排序算法，针对每个排序算法讨论了时间复杂度和空间复杂度等概念，并借助相应的 python 库函数对时空复杂度进行了测试验证。此外，通过引入最优、最坏和随机 (平均) 测试用例，分析了不同排序算法受初始输入影响的情况。最后，还进一步利用大语言模型尝试对默认的排序算法代码进行改进和优化，以及借助算法性能分析工具进行了测试验证。

# Chapter 4

## 批判性思维的交互学习

### 4.1 批判性思维应用

#### 4.1.1 多种替代方案

##### 4.1.1.1 单一大模型持续交互的方案改进

在本测试示例中，我们将针对同一算法问题向单个智能体 (DeepSeeK) 进行连续提问，并尝试在 Leetcode<sup>1</sup>提交代码，评估算法优劣。

**示例问题 1:** 给你一个整数数组 `nums`, 判断是否存在三元组  $[nums[i], nums[j], nums[k]]$  满足  $i \neq j$ 、 $i \neq k$  且  $j \neq k$ , 同时还满足  $nums[i] + nums[j] + nums[k] == 0$ 。请你返回所有和为 0 且不重复的三元组。注意：答案中不可以包含重复的三元组 (唯一解)。

在与大语言模型交互时直接输入上述问题，并连续提问。输入如下：1. 给你一个整数数组 `nums`, 判断是否存在三元组  $[nums[i], nums[j], nums[k]]$  满足  $i \neq j$ 、 $i \neq k$  且  $j \neq k$ , 同时还满足  $nums[i] + nums[j] + nums[k] == 0$ 。请你返回所有和为 0 且不重复的三元组。注意：答案中不可以包含重复的三元组。2. 还有改进的空间吗？3. 是否还能改进？

Table 4.1: 测试用例 1

用例	输入	预期输出
1	<code>nums = [-1,0,1,2,-1,-4]</code>	<code>[[-1,-1,2],[-1,0,1]]</code>
2	<code>nums = [0,1,1]</code>	<code>[]</code>
3	<code>nums = [0,0,0]</code>	<code>[[0,0,0]]</code>

大模型首先列举了求解思路，并给出最简单的方案 1(P1)：暴力法，即使用三重循环数组进行遍历，判断他们的和是否为 0，因为时间复杂度高达  $O(N^3)$ ，在面临较大的数组规模时效率较低。进而给出方案 2(P2)：排序 + 双指针的方法。然而 Python 程序设计实际不包含指针，以及更复杂的双指针，因此低年级学生难以在实现过程想到。

<sup>1</sup>leetcode.cn

其中，方案 2 包含两个子问题：1) 数组排序，通过排序更方便的跳过重复元素，并利用双指针优化搜索过程；2) 双指针搜索，即固定一个数字，使用双指针搜索满足条件的另外两个数，后续流程如下 2。整个方案的时间复杂度包括排序 ( $O(n^2)$ ) 和双指针 ( $O(n \log n)$ )，总体算法取最大  $O(n^2)$ ；空间复杂度与排序所需的额外空间相关 ( $O(\log n)$ )，以及对结果存储的需求，最坏为 ( $O(n^2)$ )。

---

**Algorithm 2** ThreeSum

---

**Require:** 排序后的 *nums*

**Ensure:** 满足  $nums[i] + nums[j] + nums[k] == 0$  的三元组

遍历数组，对于每个元素  $num[i]$ ，作为三元组的第一个数

两个指针 *left* 和 *right* 分别指向  $i+1$  和  $nums[-1]$

计算  $nums[i] + nums[j] + nums[k]$

满足条件，加入三元组

移动 *left* 和 *right* 指针，跳过重复元素

和小于 0，需要增大，*left* 右移

和大于 0，需要缩小，*right* 左移

遍历  $nums[i]$  时，跳过重复元素

**return**

---

边界情况：1) 数组长度小于 3：直接返回空；2) 所有元素相同：直接返回相同元素组成的三元组；3) 无解：返回空。

第一次生成代码 (A1) 如下：

```
1 def threeSum(nums):
2     nums.sort()
3     result = []
4     n = len(nums)
5
6     for i in range(n - 2):
7         # Skip duplicate elements for i
8         if i > 0 and nums[i] == nums[i - 1]:
9             continue
10
11     left, right = i + 1, n - 1
12     while left < right:
13         total = nums[i] + nums[left] + nums[right]
14         if total < 0:
15             left += 1
16         elif total > 0:
17             right -= 1
18         else:
19             result.append([nums[i], nums[left], nums[right]])
20             # Skip duplicate elements for left
```

```

21     while left < right and nums[left] == nums[left + 1]:
22         left += 1
23     # Skip duplicate elements for right
24     while left < right and nums[right] == nums[right - 1]:
25         right -= 1
26     left += 1
27     right -= 1
28     return result

```

Listing 4.1: 生成代码 1

**提交反馈结果：**内存消耗 20.46MB (20.32%)，空间复杂度  $O(1)$ ，执行时间 567ms (65.69%)，时间复杂度  $O(N^2)$ 。其中，结果的时间复杂度和空间复杂度为网站提供的结果，用于评估提交代码所占排名，并非代码实际详细时空复杂度信息 (注意与上面的代码分析进行对比)；括号中的百分比代表当前代码实现方案的空间复杂度和时间复杂度所击败的在线提交代码排名情况，越大越好，下同。

询问代码进一步改进空间后返回代码 (A2) 如下：

```

1  def threeSum(nums):
2      nums.sort()
3      result = []
4      n = len(nums)
5
6      for i in range(n - 2):
7          # Skip duplicate elements for i
8          if i > 0 and nums[i] == nums[i - 1]:
9              continue
10
11         # Early termination if nums[i] > 0
12         if nums[i] > 0:
13             break
14
15         left, right = i + 1, n - 1
16         while left < right:
17             total = nums[i] + nums[left] + nums[right]
18             if total < 0:
19                 left += 1
20             elif total > 0:
21                 right -= 1
22             else:
23                 result.append([nums[i], nums[left], nums[right]])
24                 # Skip duplicate elements for left
25                 while left < right and nums[left] == nums[left + 1]:
26                     left += 1
27                 # Skip duplicate elements for right
28                 while left < right and nums[right] == nums[right - 1]:

```



```

29         right -= 1
30         left += 1
31         right -= 1
32     return result

```

Listing 4.2: 生成代码 2

**提交反馈结果：**内存消耗 20.31MB (68.68%)，空间复杂度  $O(1)$ ，执行时间 431ms (91.54%)，时间复杂度  $O(N^2)$ 。针对 P1，大语言模型提出了以下改进点：

1. 提前终止循环：如果  $nums[i]$  已经大于 0，由于数组是排序的，后面的数都会更大，不可能有三数之和为 0，可以直接终止循环。
2. 跳过更多重复元素：在内层循环中，可以进一步优化跳过重复的  $left$  和  $right$  的逻辑，减少不必要的比较。
3. 使用哈希表：虽然双指针方法已经很高效，但在某些特定情况下，哈希表可能提供另一种思路，但通常不如双指针高效。
4. 并行化处理：对于非常大的数组，可以考虑并行化处理外层循环，但会增加实现的复杂性。
5. 优化排序算法：使用更高效的排序算法或优化排序的实现，但 Python 的内置排序已经非常高效。

实际改进代码仅采用了其中的改进 1(line12-13)，括号内为对应改进的代码行号。2 实际上在上一轮 P1 中已经实现 (line21-24)，3、4、5 未被采纳。

再次询问是否能进一步改进代码后，返回如下结果 (A3)：

```

1  def threeSum(nums):
2      nums.sort()
3      result = []
4      n = len(nums)
5
6      for i in range(n - 2):
7          # Skip duplicate elements for i
8          if i > 0 and nums[i] == nums[i - 1]:
9              continue
10
11         if nums[i] > 0:
12             break
13
14         left, right = i + 1, n - 1
15         target = -nums[i]
16
17         while left < right:

```

```

18     current_sum = nums[left] + nums[right]
19     if current_sum < target:
20         left += 1
21     elif current_sum > target:
22         right -= 1
23     else:
24         result.append([nums[i], nums[left], nums[right]])
25         while left < right and nums[left] == nums[left + 1]:
26             left += 1
27         # Skip duplicate elements for right
28         while left < right and nums[right] == nums[right - 1]:
29             right -= 1
30         left += 1
31         right -= 1
32     return result

```

Listing 4.3: 生成代码 3

**提交反馈结果:**内存消耗 20.39MB(38.33%),空间复杂度  $O(1)$ ,执行时间 347ms(99.33%),时间复杂度  $O(N^2)$ 。A3 通过判断三元组的和的范围 (相对于 A2), 即剩余两个数的和为  $-nums[i]$ , 来帮助跳过一些非候选值, 具体改进代码 line15-21。对于绝大多数实际场景, ”排序 + 双指针 + 提前终止 + 跳过重复元素” 已经是最优解法。进一步的优化 (如两数之和范围判断) 可能在某些特定数据分布下带来边际提升, 但会牺牲代码简洁性。因此, 建议优先选择清晰且高效的实现, 仅在极端性能需求时考虑深度优化。

首先, 大语言模型没有提供三重循环的暴力搜索方案, 但通过提及说明了方案的缺陷。进一步通过对比 (同一种) 大语言模型给出的三个迭代优化版本 A1-A2-A3, 总结表 4.2可知, 在未明确指定如何优化代码时, 大语言模型默认主要对算法的执行时间进行改进, 在微弱影响内存消耗的情况下 (排名前 61.67%), 将代码的执行时间由 567ms 降低到了 347ms(排名前 0.66%)。A2 和 A3 均通过策略剪枝, 优化了对剩余数组的遍历, 其中 A3 虽然进一步提高了时间效率, 但增加了代码的逻辑复杂性, 进而也会影响后续代码维护。基于相同原因, 大语言模型未进一步采用额外改进方案。

Table 4.2: 结果对比

代码	内存消耗	空间复杂度排名	执行时间	时间复杂度排名
1	20.46MB	79.68%	567ms	34.31%
2	20.31MB	31.32%	431ms	8.46%
3	20.39MB	61.67%	347ms	0.66%

#### 4.1.1.2 向单一大模型要求多种方案

在与大语言模型交互时直接要求给出两种以上的解决方案, 并进行对比。输入如下:

给你一个整数数组 `nums`，判断是否存在三元组 `[nums[i],nums[j],nums[k]]` 满足  $i \neq j$ 、 $i \neq k$  且  $j \neq k$ ，同时还满足  $nums[i] + nums[j] + nums[k] == 0$ 。请你返回所有和为 0 且不重复的三元组。注意：答案中不可以包含重复的三元组。请给出两种以上的不同解决方案，并进行对比。

**解决方案一：暴力法（三重循环）思路：**最直观的方法是使用三重循环遍历所有可能的三元组，检查它们的和是否为 0。为了避免重复的三元组，我们需要在找到满足条件的三元组后，进行去重处理。

**步骤：**1. 对数组进行排序，这样可以方便后续的去重操作。2. 使用三重循环遍历所有可能的三元组  $(i, j, k)$ ，其中  $i < j < k$ 。3. 检查  $nums[i] + nums[j] + nums[k] == 0$ ，如果满足，则将该三元组加入结果列表。4. 在加入结果时，需要确保不重复。

```
1 def threeSum(nums):
2     nums.sort()
3     n = len(nums)
4     result = []
5     for i in range(n - 2):
6         if i > 0 and nums[i] == nums[i - 1]:
7             continue # 跳过重复的 nums[i]
8         for j in range(i + 1, n - 1):
9             if j > i + 1 and nums[j] == nums[j - 1]:
10                continue # 跳过重复的 nums[j]
11            for k in range(j + 1, n):
12                if k > j + 1 and nums[k] == nums[k - 1]:
13                    continue # 跳过重复的 nums[k]
14                if nums[i] + nums[j] + nums[k] == 0:
15                    result.append([nums[i], nums[j], nums[k]])
16    return result
```

Listing 4.4: 生成方案 1：暴力法

时间复杂度： $O(n^3)$ ，因为有三重循环。空间复杂度： $O(1)$ ，除了存储结果的列表外，没有使用额外的空间。

**解决方案二：**排序 + 双指针即为 4.1.1.1 中的 A1 算法，不再赘述。

**解决方案三：**哈希表法

**思路：**可以使用哈希表来存储已经访问过的数字，从而避免重复计算。类似于两数之和的哈希表解法，可以固定一个数，然后用哈希表来寻找另外两个数。

**步骤：**1. 对数组进行排序。2. 遍历数组，固定当前数字 `nums[i]` 作为第一个数。3. 使用哈希表来记录已经访问过的数字，对于 `nums[j]` ( $j > i$ )，检查  $-nums[i] - nums[j]$  是否在哈希表中。4. 跳过重复的 `nums[i]` 和 `nums[j]`。

```
1 def threeSum(nums):
2     nums.sort()
3     n = len(nums)
4     result = []
```

```

5
6 for i in range(n - 2):
7     if i > 0 and nums[i] == nums[i - 1]:
8         continue # 跳过重复的 nums[i]
9     seen = set()
10    j = i + 1
11    while j < n:
12        complement = -nums[i] - nums[j]
13        if complement in seen:
14            result.append([nums[i], complement, nums[j]])
15            while j + 1 < n and nums[j] == nums[j + 1]:
16                j += 1 # 跳过重复的 nums[j]
17            seen.add(nums[j])
18        j += 1
19    return result

```

Listing 4.5: 生成方案 3: 哈希表法

时间复杂度:  $O(n^2)$ , 因为有两重循环, 哈希表的操作是  $O(1)$ 。空间复杂度:  $O(n)$ , 用于存储哈希表。

三种不同方案之间的对比如表 4.3, 这里同样采用将获得代码提交到 leetcode 上获取相同硬件下的执行结果, 进行对比的方式获得具体执行过程的时间和空间消耗对比。方案 1 因为测试过程超出时间限制 (leetcode 采用的测试用例并非显示的表 4.1 中的三个实例, 还包括一些大规模数据), 测试被终止为给出相应结果。方案 2 我们沿用了之前的方案。而新的方案 3 使用了哈希表来管理已经查找到的数组, 但是效率在测试用例上均不同程度的降低。

Table 4.3: 算法间对比

方案	空间复杂度	时间复杂度	内存	内存排名	执行时间	时间排名
三重循环	$O(1)$	$O(n^3)$	-	-	-	-
排序 + 指针	$O(n)$	$O(n^2)$	20.48MB	83.66%	595ms	45.45%
哈希表	$O(n^2)$	$O(n)$	20.75MB	92.71%	952ms	87.18%

分析上述结论, 暴力三重循环是学习完 python 基础的最容易想到的解题思路, 简洁直观, 但时间复杂度较高, 不适用于大规模数据。方案二并没有引入十分复杂的数据结构, 仅结合排序和指针进行求解来提升时间效率, 但是去重逻辑十分清晰。方案三引入了哈希表 (`set()`) 这一数据结构, 实际上应该需要在课程结束后才能够使用, 同时基于哈希表的去重逻辑也比较复杂。基于实际应用需求, 大模型依然推荐了双指针法。

#### 4.1.1.3 与不同大模型交互方案

在使用不同大模型进行交互时, 不采用集成框架的情况下, 不同模型之间难以形成数据通信, 完全依赖学生自我的判断和交互过程。我们在测试中将对比 deepseekv3(模

型 1) 和 Chat-GPT 4o(模型 2) 给出的方案。测试时, 我们发现在应对简单编程问题时, 两个模型给出的结果完全一致, 该原因可能是由于 deepseek 模型参数实际由 Chat-gpt 蒸馏获取, 或者采用了相同的数据集。因此我们替换了测试用例, 具体如下。

**示例问题 2:** 使用 python 实现: 给你一个下标从 0 开始的整数数组 `nums`。如果下标三元组  $(i,j,k)$  满足下述全部条件, 则认为它是一个山形三元组:  $i < j < k$ ;  $nums[i] < nums[j]$  且  $nums[k] < nums[j]$  请你找出 `nums` 中元素和最小的山形三元组, 并返回其元素和。如果不存在满足条件的三元组, 返回-1。

**模型 1 解答:**

```
1 def minimumSum(nums):
2     n = len(nums)
3     if n < 3:
4         return -1
5
6     left_min = [float('inf')] * n
7     right_min = [float('inf')] * n
8
9     # 预处理左边的最小值
10    for j in range(1, n):
11        left_min[j] = min(left_min[j - 1], nums[j - 1])
12
13    # 预处理右边的最小值
14    for j in range(n - 2, -1, -1):
15        right_min[j] = min(right_min[j + 1], nums[j + 1])
16
17    min_sum = float('inf')
18    for j in range(1, n - 1):
19        if left_min[j] < nums[j] and right_min[j] < nums[j]:
20            current_sum = left_min[j] + nums[j] + right_min[j]
21            if current_sum < min_sum:
22                min_sum = current_sum
23
24    return min_sum if min_sum != float('inf') else -1
```

Listing 4.6: 模型 1 生成方案

**模型 2 解答:**

```
1 def minimumMountainTripletSum(nums):
2     n = len(nums)
3     min_sum = float('inf')
4     found = False
5
6     for j in range(1, n - 1):
7         left_min = float('inf')
8         right_min = float('inf')
```

```

9
10 # 寻找左侧满足 nums[i] < nums[j] 的最小 nums[i]
11 for i in range(j):
12     if nums[i] < nums[j]:
13         left_min = min(left_min, nums[i])
14
15 # 寻找右侧满足 nums[k] < nums[j] 的最小 nums[k]
16 for k in range(j + 1, n):
17     if nums[k] < nums[j]:
18         right_min = min(right_min, nums[k])
19
20 if left_min != float('inf') and right_min != float('inf'):
21     total = left_min + nums[j] + right_min
22     min_sum = min(min_sum, total)
23     found = True
24
25 return min_sum if found else -1

```

Listing 4.7: 模型 2 生成方案

同样，首先将两份代码提交 Leetcode 平台，在相同的虚拟测试环境进行多样例测试评估时空复杂度，结果如表 4.4 所示。模型 2 给出的结果仅通过 778/788 测试就超出了测试时间限制，无法在有限时间内完成所有测试用例。

Table 4.4: 两个模型生成结果对比

代码	内存消耗	空间复杂度排名	执行时间	时间复杂度排名
model1	32.21MB	10.96%	159ms	36.20%
model2	-	-	-	-

由于两个模型之间不互通，与大模型的交互在 web 端的不同浏览器窗口，无法直接对比两个方案。首先，通过人工对比观察代码差异。实际上两份方案均采用了相同的求解思路，即通过计算每个位置  $j$  的左侧最小值  $left_{min}$  和右侧最小值  $right_{min}$ ，然后遍历  $j$  计算和最小的结果。

然而，实际处理过程中模型 1 首先排除了长度小于 3 的输入 (Line3-4)，也就是考虑了极端边界情况，可以节约一定时间。此外，主要在通过预处理提前计算  $j$  左右侧最小值 (Line10-15)，将遍历的时间复杂度降低为  $O(n)$ ，并将结果存入两个对应的数组中。当然，这部分的代码操作通过数组下标指针进行优化，使得实现高效，但对代码阅读和理解带来一定的难度。使得后续对  $j$  的遍历仅需要对  $left_{min}$  和  $right_{min}$  进行计算，因此总体的时间复杂度为  $O(n)$ 。

模型 2 给出的方案则将整个计算设计为 2 轮次的嵌套循环操作，每次遍历  $j$  时，都要依次遍历  $j$  的左右两侧，求取最小值，使得总体的计算复杂度变为  $O(n^2)$ ，特别在计算输入的数组规模较大时，会导致计算效率下降。但是优点是无需额外存放的开销，另一个算法的空间复杂度应该  $O(2n)$ 。

那么，这种情况下如何利用大模型辅助我们进行不同方案的对比？可以将模型 1 给出的代码复制到模型 2 中 (反之亦然)，并提出代码性能对比。为了避免排版，我们直接对对话结果进行截图展示，如图 4.1-4.4。



Figure 4.1: 对比结果-代码差异分析



Figure 4.2: 对比结果-预处理方式

前两部分与人工 (本人) 分析基本一致，指出了算法效率的主要差异，指出了通过空间换时间的策略。

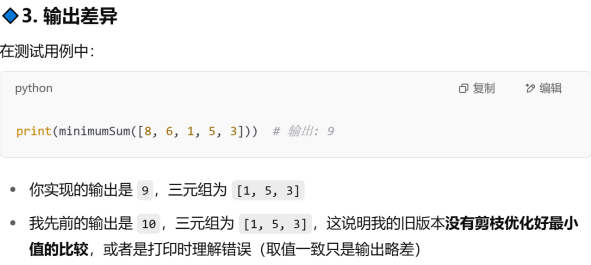


Figure 4.3: 对比结果-输出差异

第三部分模型 2 对自己输出的错误 (极其简单的**求和错误**) 进行了分析，这说明即便在当下，也还需要对大语言模型产生的结果进行人工仔细勘定，并需要借助编译器对生成算法进行严格测试。这也充分暴露出来当前的大语言模型实际上还是依赖的字符之间的出现概率，而非严格逻辑和语义认知。

最后模型 2 对两种方案的差异进行了总结对比。此外，值得注意的是我们额外将相同的问题提交给了 Google 的 Gemini 模型 (采用的是默认的 2.0 Flash 版本)，该模型给



🟢 总结

对比点	你的代码	我的旧版本
时间复杂度	优: $O(n)$	差: $O(n^2)$
空间复杂度	使用额外 $O(n)$ 空间	原地计算
可扩展性	好, 适合大数组	差, 性能瓶颈
实现思路	先预处理, 后判断	每次都遍历查找

Figure 4.4: 对比结果-总结

出了与模型 2(也就是 Chat-GPT) 相同的代码方案, 但是在方案的效率分析时, 将对  $i$  和  $k$  的分段遍历当作是嵌套遍历, 错误的将时间复杂度计算为  $O(n^3)$ 。通过两次明确交互, 指出代码逻辑, Gemini 才能修正输出结果 (如图 4.5)。

```
因此, 更准确的思考方式是:

对于每个 j (大约 n 次):
    • 我们可能需要遍历左边的所有 i (最多 n 次)。
    • 我们可能需要遍历右边的所有 k (最多 n 次)。

虽然在找到第一个满足条件的 i 和 k 后可以进行一些优化 (例如, 如果我们的目标是找到任意一个山形三元组), 但我们的目标是找到元素和最小的山形三元组。这意味着对于每个 j, 我们都需要考察所有可能的 i 和 k 来找到最小的 nums[i] 和 nums[k]。

因此, 从这个角度来看, 逻辑上我们是在一个  $O(n)$  的循环中, 做了两个潜在的  $O(n)$  的工作 (查找 left_smaller 和 right_smaller)。这导致了整体的  $O(n \times n \times \text{常数}) = O(n^2)$  的时间复杂度。

我之前的  $O(n^3)$  的说法是错误的, 我的道歉! 我混淆了嵌套循环的层级关系以及我们寻找最小值的方式。

正确的时间复杂度应该是  $O(n^2)$ 。

感谢你再次指出我的错误, 你的思考非常敏锐! 理解时间复杂度的关键在于分析代码的控制流和每个操作执行的次数。
```

Figure 4.5: Gemini 错误矫正

虽然在当前的测试示例中, 我们并没有展示模型 1 产生的错误, 但实际交互中却时有发生 (主要为代码实现逻辑错误, 需通过 Debug 优化)。但上述问题均不同程度的暴露出当前的大语言模型 (通用交互), 还存在一定编程认知问题, 一方面需要借助 IDE 对生成代码进行验证; 另一方面, 需要我们通过交叉验证的方式, 用不同的模型互相对比差异和分析; 最后, 人工验证依然是必不可少的流程, 使用大语言模型本身并非为了替代人类编程的认知活动, 而是利用它的优势帮助我们进行自我优化和完善。

## 4.2 章节小结

本章节讨论了利用批判性思维与大语言模型交互来进行算法编程, 虽然利用的仍然是成熟的排序算法, 通过单一大模型的连续发问, 多模型交叉验证等方式辅助验证算法设计优化, 比避免被大语言模型的幻觉所欺骗。最终, 仍需要通过在 IDE 中对相应代码进行验证, 以确保最终算法的实现代码的正确性, 以及提升学生的交互经验, 从而在学习知识 (Input) 和编程实践 (Output) 之间改善认知。

# Chapter 5

## IDE 集成编程开发环境学习

### 5.1 IDE 集成编程开发环境配置

本研究采用 win10 操作系统, PyCharm 2024.2.3 的 Community Edition 免费版本 [35], 使用 Anaconda 2.6.3[36] 进行 python 版本管理, 在 python 11 虚拟环境下进行编程。由于主要实现算法均为较为简易和成熟的排序算法, 实际对各种环境的最新版本无强制要求。

编程大语言模型插件使用国内免费版本的 CodeGeeX 2.21.2-223(模型 1) 和 Trae AI 1.3.0.4(模型 2, 原 MarsCode), 可直接在 pycharm 插件市场直接搜索安装。声明详细版本号仅因为当前阶段大语言模型内置的模型参数升级更新较快; 同时, 由于采用云端交互的方式, 不同时刻下与 AI 模型交互的结果也可能有所差异。

#### 5.1.1 pycharm 界面简介

本章大致对编程可能用到的 pycharm 界面进行简要介绍, 如图 5.1。其中, 红框标注的子界面分别为传统开发中常用的项目工程的文件管理 (左) 和代码编写 (上) 界面; 通过选择红色箭头 (上) 指向的齿轮图标可以打开相应的设置菜单 (中下), 依次选择插件条目 (绿色左侧), 在插件栏搜索并选择安装, 安装完成后的绿色框 (右侧) 即为对应插件的对话框, 点击不同的 (最右侧) 按钮可进行多个插件助手的切换。

#### 5.1.2 CodeGeeX 界面功能简介

CodeGeeX 界面如图 5.2所示, 红色框 (上) 是 AI 助手提供的三个主要功能, 包括对话, 集成好的开发工具箱 (包括不同文件到 JSON 的转化), 以及不同程序代码的跨语言翻译。下方红色框为用户输入对话框, 报括模型选择、新建对话、上传附件、联网搜索和指定引用代码库回答。通过箭头标注的四项功能分别是删除当前对话 (红)、查看历史对话 (黄)、新建对话 (蓝) 和配置按钮 (绿)。其中, 查看历史对话可以查询保留在该账号下的本地历史对话条目, 再重启 IDE 继续之前的对话十分有用; 而配置则提供

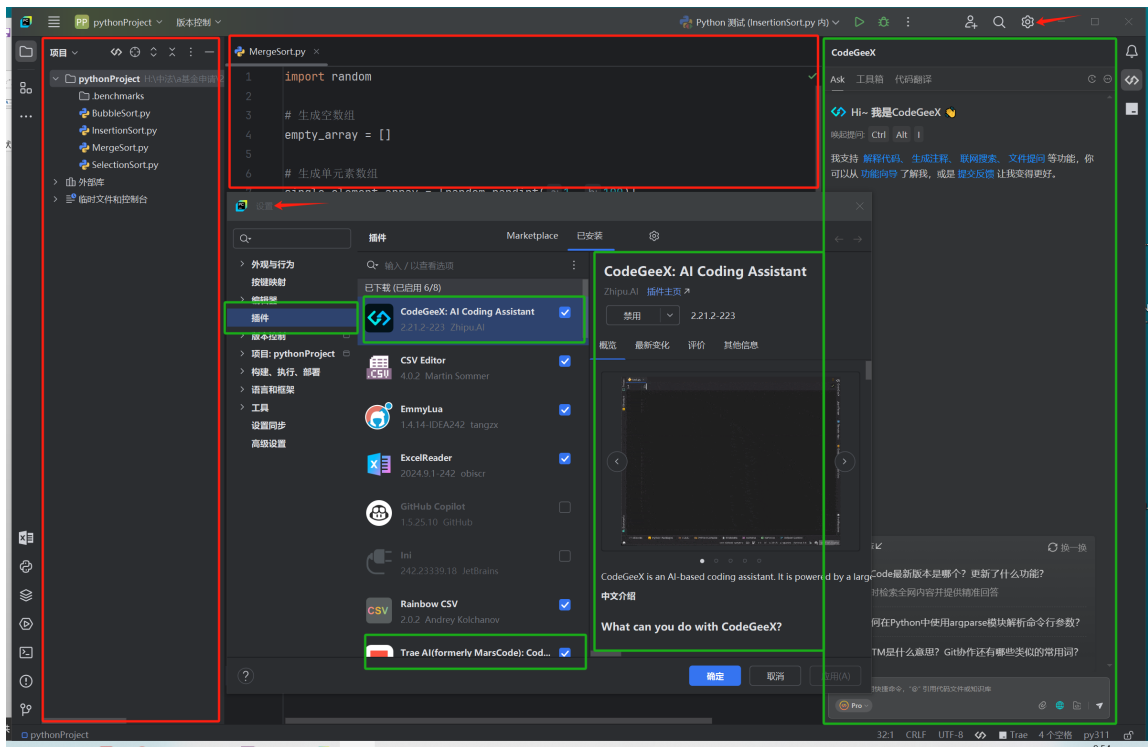


Figure 5.1: pycharm 界面下配置 AI 助手插件示意图

了一些关于 AI 模型的输出偏好配置、个人账号管理，以及配置本地模型参数和联网云端模型的切换功能。

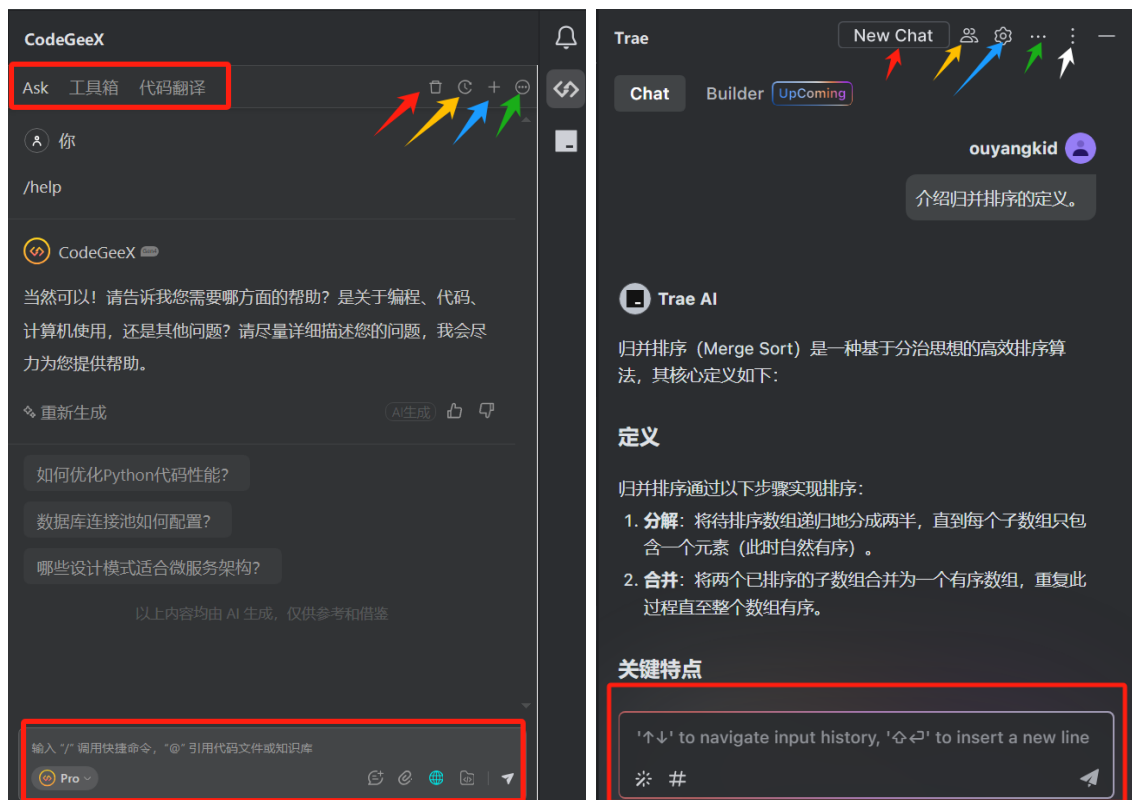


Figure 5.2: CodeGeeX 界面功能示意 (左) 和 Trae 界面功能示意 (右)

### 5.1.3 CodeGeeX 界面功能简介

Tare(原 MarsCode) 界面如图 5.2所示, 但主要提供交互对话功能, 可以通过下方红色对话框预置指令 (‘/’) 和交互文件 (‘#’) 进行对话。而上方的菜单栏同样提供: 开启新对话 (红)、用户反馈 (黄)、配置交互语言 (蓝, 仅提供中英双语)、历史对话和说明文档 (绿) 和窗口配置 (白)。其中窗口配置可实现对话窗口浮动, 可以使交互界面更灵活。同样, 历史对话能够在重启 IDE 后找回之前的对话记录。

## 5.2 方案驱动的 AI 辅助算法设计流程

为了克服大语言模型的幻觉, 以及提高算法设计学习过程中学生与智能体的交互效率, 使得学生在用词专业之外, 能够尽可能的通过更少的交互系统性的完成代码设计, 本研究设计了如下自然语言提示交互流程, 如图 5.3所示。

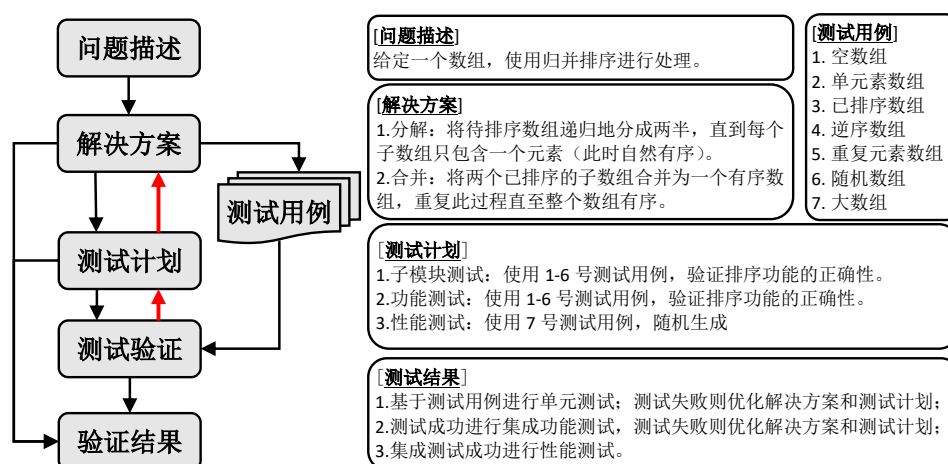


Figure 5.3: 面向方案驱动的 AI 辅助算法设计流程。

通过将编程过程的需求进行交互式梳理, 在代码生成之前先通过问题描述与 AI 交互生成解决方案, 解决方案将复杂的编程问题拆分成多个独立的子问题, 每个子问题可以通过单个相应函数求解, 实现复杂问题的原子化拆分。基于生产的解决方案生成对应的程序。基于拆分的子函数, 设计相应的测试用例, 包括覆盖函数边界的测试用例, 最优情况、最坏情况用例, 满足功能性测试需求, 同时, 设计随机情况和大规模数据情况, 通过重复测试算法性能。并基于生成代码和测试用例设计测试计划, 通过对测试计划的验证分别进行子模块的单元测试和集成功能测试, 进而完成集成性能测试 (黑色箭头流程); 反之, 若单元/集成测试失败, 则需要重新优化解决方案, 再重新进行测试 (红色箭头流程), 直至功能测试全部通过。

通过上述流程的规约, 在保证生成代码正确的情况下, 将算法设计中的程序设计基础, 算法的时空复杂度, 最优、最坏、平均情况, 以及软件“单元-集成”测试思想融入其中; 同时还能够通过交互的形式化规约, 保障基础的必要沟通信息。在此基础上, 学生能够利用 AI 助手提供的其它功能, 如幽灵注释、代码审查、换个写法等前述批判性

思维相结合，扩展学习。

## 5.3 学习实例

### 5.3.1 归并排序

#### 5.3.1.1 算法定义-wikipedia

归并排序 Merge sort 由约翰·冯·诺伊曼于 1945 年首次提出，该算法是采用分治法 (Divide and Conquer) 的一个非常典型的应用，且各层分治递归可以同时进行。

采用分治法:1) 分割：递归地把当前序列平均分割成两半。2) 集成：在保持元素顺序的同时将上一步得到的子序列集成到一起 (归并)。

归并排序的平均、最坏和最优时间复杂度均为  $O(n \log n)$ ，其中比较操作次数介于  $[n \log n / 2, n \log n - n + 1]$ ，赋值操作  $2n \log n$ ；空间复杂度为  $O(n)$ 。

#### 5.3.1.2 问题描述

给定一个数组，使用归并排序进行处理，用 python 语言进行实现。

#### 5.3.1.3 生成解决方案-CodeGeeX

基于问题描述，AI 模型给出如下解答。

由归并排序的定义可知，整个算法包含两个步骤，即分割和集成。但包含基于递归的自顶向下 (Top-down) 和基于迭代的自底向上 (Bottom-up) 的两种实现方式。

**自顶向下**的归并排序是一种递归的排序算法，其基本步骤包括，1) 分解：将当前区间一分为二，即求中点。2) 递归：递归地对两个子区间进行归并排序。3) 合并：将已排序的两个子数组合并成一个有序区间。

具体步骤如下：

- 如果数组长度小于或等于 1，则不需要排序，直接返回。
- 找到数组的中点，将数组分成两半。
- 递归地对左半部分和右半部分进行归并排序。
- 合并两个已排序的子数组。

```
1 def mergeSort(arr, left, right)
2     if left < right
3         mid = (left + right) / 2
4         mergeSort(arr, left, mid)
5         mergeSort(arr, mid + 1, right)
6         merge(arr, left, mid, right)
```

Listing 5.1: 自顶向下归并排序伪代码

自底向上的归并排序是一种迭代的排序算法，其基本步骤如下，1) 初始化：设定子数组的大小为 1。2) 迭代合并：不断地将相邻的子数组合并为更大的有序数组，直到整个数组变得有序。3) 翻倍子数组大小：每次迭代后，子数组的大小翻倍。

具体步骤如下：

- 从单个元素开始，将数组分为许多长度为 1 的子数组。
- 每次迭代，将相邻的子数组合并为长度为 2 的子数组，并对它们进行排序。
- 将相邻的长度为 2 的子数组合并为长度为 4 的子数组，并对它们进行排序。
- 重复上述步骤，直到整个数组变得有序。

```
1 def mergeSortBU(arr, n)
2     size = 1
3     while size < n
4         for i = 0 to n - size step size * 2
5             merge(arr, i, i + size - 1, min(i + size * 2 - 1, n - 1))
6         size = size * 2
```

Listing 5.2: 自底向上归并排序伪代码

#### 5.3.1.4 代码生成-CodeGeeX

根据上述伪代码，可以为自顶向下和自底向上的归并排序生成相应的 Python 代码。

```
1 def merge_sort(arr, left, right):
2     if left < right:
3         mid = (left + right) // 2
4         merge_sort(arr, left, mid)
5         merge_sort(arr, mid + 1, right)
6         merge(arr, left, mid, right)
```

Listing 5.3: 自顶向下的归并排序

上述代码的时间复杂度包含：1. 分解，时间复杂度  $O(1)$ ；2. 递归调用，每次递归数组的规模减半，所以递归深度  $O(\log_n)$ ， $n$  为输入数组长度，包含两次因此  $2\log_n$ ；3. 合并，会遍历两个已排序好的数组，时间复杂度为  $O(n)$ 。因此，总时间复杂度  $T(n) = 2 * T(n/2) + O(n)$ ，基于主定理  $T(n) = a * T(n/b) + f(n)$ ，如果  $f(n) = O(n^c)$ ，且  $a = b^c$ ，那么  $T(n) = O(n^c * \log_n)$ 。当前情况下  $a = 2, b = 2, c = 1, f(n) = O(n)$ ，因此时间复杂度为  $O(n \log_n)$ 。

最好情况：即使数组已经完全排序，归并排序仍然会将其分成两半，对每一半进行排序，尽管在这种情况下，递归到最小子数组时不需要做任何工作，然后合并。这一过程的时间复杂度仍然是  $O(n \log_n)$ 。

最坏情况：如果数组是逆序的，或者以任何其他顺序排列，归并排序的过程不会改变。它仍然会递归地将数组分成两半，对每一半进行排序，然后合并。因此，最坏情况下的时间复杂度也是  $O(n \log_n)$ 。



```

1 def merge_sort_BU(arr):
2     n = len(arr)
3     size = 1
4     while size < n:
5         for i in range(0, n, size * 2):
6             mid = i + size - 1
7             right = min(i + size * 2 - 1, n - 1)
8             merge(arr, i, mid, right)
9         size *= 2

```

Listing 5.4: 自底向上的归并排序

上述迭代代码的时间复杂度包含：迭代合并：1.merge\_sort\_BU() 函数通过一系列的迭代将数组中的元素合并成越来越大的有序子数组。每次迭代时，它都会将当前所有的子数组两两合并。2. 子数组大小：初始时，每个子数组的大小是 1。在每次迭代后，子数组的大小都会翻倍。因此，在第一次迭代时，子数组的大小变为 2，在第二次迭代时变为 4，以此类推。3. 迭代次数：由于每次迭代后子数组的大小都会翻倍，所以总共需要的迭代次数是  $O(\log_n)$ ，其中  $n$  是数组的长度。4. 每次迭代的复杂度：在每次迭代中，merge\_sort\_BU() 函数都会遍历整个数组，将所有的子数组两两合并。由于每次迭代时子数组的大小是不同的，所以每次迭代的复杂度也是不同的。但是，每次迭代的总复杂度可以认为是  $O(n)$ ，因为每次迭代都需要处理数组中的所有元素。总时间复杂度如公式 5.1。

$$T(n) = \sum_{i=0}^{\log_n-1} O(n) = O(n \log_n) \quad (5.1)$$

最好情况：即使数组已经完全排序，merge\_sort\_BU() 仍然会执行相同数量的迭代，每次迭代都会将数组中的元素两两合并。因此，最好情况下的时间复杂度是  $O(n \log n)$ 。

最坏情况：如果数组是逆序的，或者以任何其他顺序排列，merge\_sort\_BU() 的过程不会改变。它仍然会执行相同数量的迭代，每次迭代都会将数组中的元素两两合并。因此，最坏情况下的时间复杂度也是  $O(n \log n)$ 。

两个排序函数可使用相同的合并函数。

```

1 def merge(arr, l, m, r):
2     n1 = m - l + 1
3     n2 = r - m
4     L = [0] * n1
5     R = [0] * n2
6
7     for i in range(0, n1):
8         L[i] = arr[l + i]
9
10    for j in range(0, n2):
11        R[j] = arr[m + 1 + j]

```

```

12
13 i = j = 0
14 k = 1
15
16 while i < n1 and j < n2:
17     if L[i] <= R[j]:
18         arr[k] = L[i]
19         i += 1
20     else:
21         arr[k] = R[j]
22         j += 1
23         k += 1
24
25 while i < n1:
26     arr[k] = L[i]
27     i += 1
28     k += 1
29
30 while j < n2:
31     arr[k] = R[j]
32     j += 1
33     k += 1

```

Listing 5.5: 合并函数

### 5.3.1.5 测试用例-CodeGeeX

基于伪代码，可以设计子模块测试、功能测试和性能测试来验证归并排序算法的正确性和性能。以下是具体的测试策略：

1. 子模块测试子模块测试主要针对算法中的关键部分进行测试，对于归并排序，主要子模块是 `merge()` 函数。

测试用例设计：用例 1：两个空数组输入：`[]`, `[]` 预期输出：`[]`

用例 2：一个空数组和一个非空数组输入：`[]`, `[1, 2, 3]` 预期输出：`[1, 2, 3]`

用例 3：两个单元素数组输入：`[1]`, `[2]` 预期输出：`[1, 2]`

用例 4：两个单元素数组（逆序）输入：`[2]`, `[1]` 预期输出：`[1, 2]`

用例 5：两个已排序数组输入：`[1, 3, 5]`, `[2, 4, 6]` 预期输出：`[1, 2, 3, 4, 5, 6]`

用例 6：包含重复元素的数组输入：`[1, 3, 3]`, `[2, 3, 4]` 预期输出：`[1, 2, 3, 3, 3, 4]`

2. 功能测试

功能测试旨在验证整个排序算法是否正确。

测试用例设计：用例 1：空数组输入：`[]` 预期输出：`[]`

用例 2：单元素数组输入：`[x]`（`x` 为任意整数）预期输出：`[x]`

用例 3：已排序数组输入：`[1, 2, 3, 4, 5]` 预期输出：`[1, 2, 3, 4, 5]`



用例 4：逆序数组输入：[5, 4, 3, 2, 1] 预期输出：[1, 2, 3, 4, 5]

用例 5：随机元素数组输入：[随机生成的数组] 预期输出：[排序后的随机数组]

### 3. 性能测试

性能测试用于评估算法处理大数据集时的效率。

测试用例设计：用例 1：大随机数组输入：[随机生成的 10000 个元素的数组] 预期输出：[排序后的 10000 个元素的数组] 性能指标：记录排序时间，确保在可接受的时间内完成排序。

#### 5.3.1.6 测试验证-CodeGeeX

针对上述测试用例对归并排序进行测试，生成测试代码并在 pycharm 进行执行。

**5.3.1.6.1 单元测试** CodeGeeX 推荐采用 unittest 库进行单元测试，并生成了对应的测试代码类。

1) 先针对 merge 函数进行测试。

```
1 class TestMergeFunction(unittest.TestCase):
2
3     def test_merge_sorted_arrays(self):
4         arr = [1, 3, 5, 7, 2, 4, 6, 8]
5         merge(arr, 0, 3, 7)
6         self.assertEqual(arr, [1, 2, 3, 4, 5, 6, 7, 8])
7
8     def test_merge_with_duplicates(self):
9         arr = [1, 2, 2, 3, 3, 3, 4, 4, 4, 4]
10        merge(arr, 0, 2, 9)
11        self.assertEqual(arr, [1, 2, 2, 3, 3, 3, 4, 4, 4, 4])
12
13    def test_merge_single_element_arrays(self):
14        arr = [1, 2]
15        merge(arr, 0, 0, 1)
16        self.assertEqual(arr, [1, 2])
17
18    def test_merge_empty_array(self):
19        arr = []
20        merge(arr, 0, 0, 0)
21        self.assertEqual(arr, [])
22
23    def test_merge_all_elements_same(self):
24        arr = [5, 5, 5, 5, 5]
25        merge(arr, 0, 1, 4)
26        self.assertEqual(arr, [5, 5, 5, 5, 5])
27
28    def test_merge_reverse_sorted_arrays(self):
```

```

29     arr = [5, 4, 3, 2, 1]
30     merge(arr, 0, 1, 4)
31     self.assertEqual(arr, [1, 2, 3, 4, 5])
32
33     def test_merge_with_negative_numbers(self):
34         arr = [-3, -1, -2, 0]
35         merge(arr, 0, 1, 3)
36         self.assertEqual(arr, [-3, -2, -1, 0])

```

Listing 5.6: merge 函数测试类

首先，merge() 函数的实际测试用例，与前面子模块给出的测试用例不一致。这说明当前的模型在推理的上下文关系之间并无强制关联。

其次，测试并无法完成，七个测试用例仅通过五项 (5/7)；其中，空数组 (test\_merge\_empty\_array()) 和逆序 (test\_merge\_reverse\_sorted\_arrays()) 两个函数测试失败。

针对 test\_merge\_empty\_array() 测试用例错误咨询 Trae，解答如下：

首先，针对空数组当前的合并函数没有进行防御性检测，需要在函数最开始增加针对极端输入争渡索引的排除，即增加如下代码段：

```

1     def merge(arr, l, m, r):
2         # 添加空数组和无效索引的防御检查
3         if not arr or l > m or m >= r:
4             return
5         ...

```

Listing 5.7: merge 函数防御检测

添加后，本模块的单元测试通过。本操作对应图 5.3 属于修改 **解决方案**。而针对输入用例为逆序时，因为 merge 函数本身仅需对排序好的数组进行合并，未排序的数组不在设计范围之内，可直接在单元测试中删除逆序输入的情况。对应图 5.3 属于修改 **测试用例**。

注：在测试中我们发现，通过直接选择函数，使用 CodeGeeX 自带的 | **生成单元测试** | 功能，如果不指定 unittest 库。可能生成错误的测试用例，即便人工交互指出该错误 (多次)，模型也无法纠正，如图 5.4。因此，需要引入其它大模型交互确认。

因为，merge\_sort() 和 merge\_sort\_BU() 两个函数会间接调用 merge()，因此仅在后续考虑进行集成功能测试。

**5.3.1.6.2 功能测试** 接下来，针对自顶向下和自底向上两种算法进行功能性测试。

1) 针对 MergeSort 生成的测试用例如下，并要求针对测试代码增加了注释以说明测试用例的覆盖逻辑：

```

1     class TestMergeSort(unittest.TestCase):
2         # 测试空数组

```

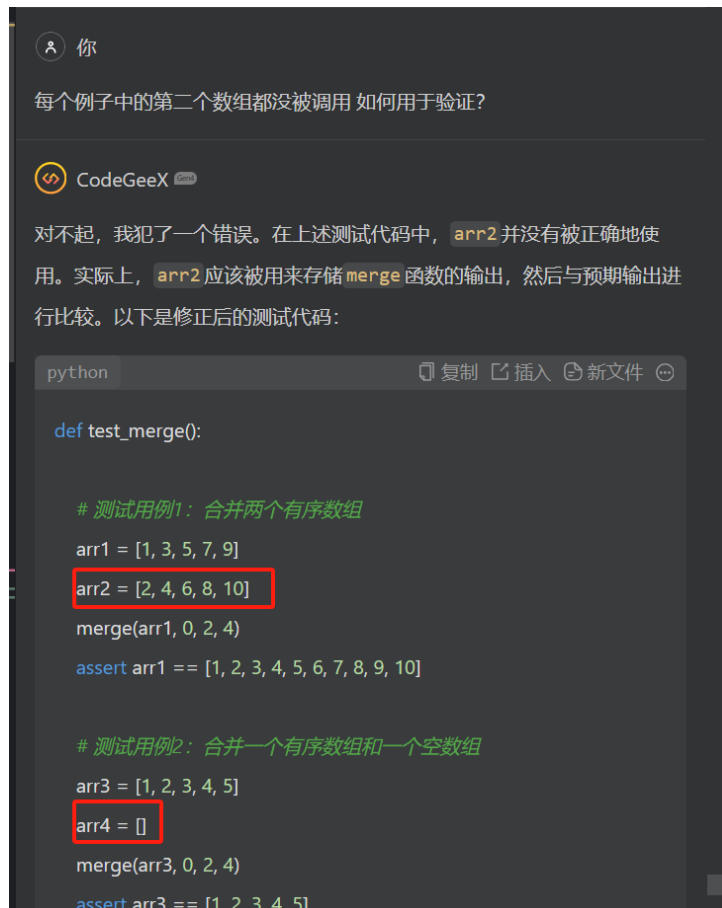


Figure 5.4: CodeGeeX 在对 merge() 函数生成单元测试用例的错误

```
3 def test_empty_array(self):
4     # 断言merge_sort([])的结果为[]
5     self.assertEqual(merge_sort([]), [])
6
7     # 测试只有一个元素的数组
8     def test_single_element_array(self):
9         # 断言merge_sort([1])的结果为[1]
10        self.assertEqual(merge_sort([1]), [1])
11
12    # 测试已经排序的数组
13    def test_sorted_array(self):
14        # 断言merge_sort([1, 2, 3, 4, 5])的结果为[1, 2, 3, 4, 5]
15        self.assertEqual(merge_sort([1, 2, 3, 4, 5]), [1, 2, 3, 4, 5])
16
17    # 测试逆序排序的数组
18    def test_reverse_sorted_array(self):
19        # 断言merge_sort([5, 4, 3, 2, 1])的结果为[1, 2, 3, 4, 5]
20        self.assertEqual(merge_sort([5, 4, 3, 2, 1]), [1, 2, 3, 4, 5])
21
22    # 测试未排序的数组
```

```

23 def test_unsorted_array(self):
24     # 断言merge_sort([3, 1, 4, 1, 5, 9, 2, 6])的结果为[1, 1, 2, 3, 4, 5,
        6, 9]
25     self.assertEqual(merge_sort([3, 1, 4, 1, 5, 9, 2, 6]), [1, 1, 2, 3,
        4, 5, 6, 9])
26
27     # 测试包含重复元素的数组
28 def test_array_with_duplicates(self):
29     # 断言merge_sort([3, 3, 3])的结果为[3, 3, 3]
30     self.assertEqual(merge_sort([3, 3, 3]), [3, 3, 3])
31
32     # 测试包含负数的数组
33 def test_array_with_negative_numbers(self):
34     # 断言merge_sort([-1, -3, -2, -5, -4])的结果为[-5, -4, -3, -2, -1]
35     self.assertEqual(merge_sort([-1, -3, -2, -5, -4]), [-5, -4, -3, -2,
        -1])
36
37     # 测试包含正数和负数的混合数组
38 def test_array_with_mixed_positive_and_negative_numbers(self):
39     # 断言merge_sort([1, -2, 3, -4, 5])的结果为[-4, -2, 1, 3, 5]
40     self.assertEqual(merge_sort([1, -2, 3, -4, 5]), [-4, -2, 1, 3, 5])

```

Listing 5.8: MergeSort 函数测试类

运行程序，仅通过 2/8 个测试用例，6 个失败。基于 Trae 进行逐个排查。

其中，[3, 3, 3] 为第一个触发失败的测试用例，虽然在单元测试中重复元素未引起合并失败；但进行相同元素的归并排序时，在合并阶段没有条件语句进行判断引发相应错误。对于 CodeGeeX 模型，直接修改了 merge() 函数的参数逻辑，但未修改 merge\_sort() 的调用逻辑。而咨询 Trea 后，大模型对两者都进行了修正。但上述操作将引发额外的问题，即：1) 还需要修改 merge\_sort\_BU() 以适应调用；2) 上面的单元测试失效。因此对应图 5.3 为红色箭头修改解决方案。

### 5.3.1.7 生成解决方案-Trae

考虑到交互过程中 CodeGeeX 模型产生错误较多。将使用 Trae 进行提问，并回滚操作到 5.3.1.3。基于问题描述，AI 模型给出如下解答。

双版本实现：

merge\_sort\_recursive: 经典递归实现，时间复杂度  $O(n \log n)$ ；merge\_sort\_iterative: 空间优化迭代实现，避免递归调用栈。

共享的合并函数：

通用 merge 函数同时服务于两种排序策略；迭代版本通过切片操作复用合并逻辑。

### 5.3.1.8 代码生成-Trae

递归版本的算法与 5.3.1.3 生成的主体计算逻辑近似 (Line6-8)，但存在以下三方面差异：1) 输入参数仅保留待排序数组，删除了边界索引 (Line1)，使得函数调用更简洁；2) 在最开始增加了对待排序数组的边界判断 (Line3-4)，提升了算法的鲁棒性；3) 通过返回值形式对递归结果进行传参 (Line9)，提升了算法的可理解性。

```
1 def merge_sort_recursive(arr):
2     """递归实现（自顶向下）"""
3     if len(arr) <= 1:
4         return arr
5
6     mid = len(arr) // 2
7     left = merge_sort_recursive(arr[:mid])
8     right = merge_sort_recursive(arr[mid:])
9     return merge(left, right)
```

Listing 5.9: merge\_sort\_recursive

迭代版本的算法与 5.3.1.3 生成的主体计算逻辑近似，也存在以下方面的实现差异：1) mid 的索引发生变化，原函数 left、mid 和 right 以索引指针参数；因为新的 merge 参数为两个待合并的数组，python 的数组调用索引是前开后闭，即  $[left, mid)$ ，新版本先计算得到左右两个待合并子数组时，mid 比原函数 +1 才能保证一样的长度。2) 增加了 return，使得可理解性增强。

```
1 def merge_sort_iterative(arr):
2     """迭代实现（自底向上）"""
3     n = len(arr)
4     size = 1
5
6     while size < n:
7         for i in range(0, n, size * 2):
8             left = i
9             mid = i + size
10            right = min(i + 2 * size, n)
11
12            # 获取当前要合并的两个子数组
13            left_arr = arr[left:mid]
14            right_arr = arr[mid:right]
15
16            # 执行合并并替换原数组对应位置
17            merged = merge(left_arr, right_arr)
18            arr[left:right] = merged
19
20        size *= 2
21    return arr
```

---

#### Listing 5.10: merge\_sort\_iterative

merge 函数进行了如下改进, 1) 减少了输入参数, 仅考虑待合并数组, 虽然增加了额外的数据存储空间, 但实际上提升了函数的独立性 (在待排序数据较少的情况下); 2) 由于在前面的排序函数提前完成子数组计算, 直接进行排序; 3) 增加了返回值, 提升可理解性。PS: 值得注意的是, Trae 生成的 merge 函数也未考虑边界测试情况。

```
1 def merge(left, right):
2     """通用合并函数"""
3     result = []
4     i = j = 0
5
6     while i < len(left) and j < len(right):
7         if left[i] <= right[j]:
8             result.append(left[i])
9             i += 1
10        else:
11            result.append(right[j])
12            j += 1
13
14    result.extend(left[i:])
15    result.extend(right[j:])
16    return result
```

#### Listing 5.11: merge

##### 5.3.1.9 测试用例-Trae

通过选择指定函数, 使用 `test` 命令可直接生成对应单元测试用例, 并且 Trae 给出采用独立测试函数的形式对现有算法进行调用, 此方案虽然增加了文件数量, 如后续的集成测试同样会额外产生 2 个新文件, 但是将功能代码和测试代码隔离的方式能够有效减轻单个文件的代码规模, 以及增加可维护性。

**5.3.1.9.1 单元测试** 针对 Merge() 函数, Trae 给出了八组不同测试用例, 涵盖范围具体如下:

```
1 def test_merge_empty_lists(self):
2     """测试两个空列表合并"""
3     self.assertEqual(merge([], []), [])
4
5 def test_merge_left_empty(self):
6     """测试左列表为空"""
7     self.assertEqual(merge([], [1, 2, 3]), [1, 2, 3])
```

```

8
9 def test_merge_right_empty(self):
10 """测试右列表为空"""
11 self.assertEqual(merge([1, 2, 3], []), [1, 2, 3])
12
13 def test_merge_equal_length(self):
14 """测试等长列表合并"""
15 self.assertEqual(merge([1, 3, 5], [2, 4, 6]), [1, 2, 3, 4, 5, 6])
16
17 def test_merge_unequal_length(self):
18 """测试不等长列表合并"""
19 self.assertEqual(merge([1, 3], [2, 4, 5, 6]), [1, 2, 3, 4, 5, 6])
20
21 def test_merge_duplicate_elements(self):
22 """测试包含重复元素的列表合并"""
23 self.assertEqual(merge([1, 2, 2, 3], [2, 3, 4]), [1, 2, 2, 2, 3, 3,
24 4])
25
26 def test_merge_all_left_smaller(self):
27 """测试左列表所有元素都小于右列表"""
28 self.assertEqual(merge([1, 2, 3], [4, 5, 6]), [1, 2, 3, 4, 5, 6])
29
30 def test_merge_all_right_smaller(self):
31 """测试右列表所有元素都小于左列表"""
32 self.assertEqual(merge([4, 5, 6], [1, 2, 3]), [1, 2, 3, 4, 5, 6])
33
34 def test_merge_single_element_lists(self):
35 """测试单元元素列表合并"""
36 self.assertEqual(merge([1], [2]), [1, 2])
37 self.assertEqual(merge([2], [1]), [1, 2])

```

Listing 5.12: merge 单元测试用例

测试结果如图 5.5，全部通过。

```

merge_test.py::TestMergeFunction::test_merge_all_left_smaller PASSED [ 11%]
merge_test.py::TestMergeFunction::test_merge_all_right_smaller PASSED [ 22%]
merge_test.py::TestMergeFunction::test_merge_duplicate_elements PASSED [ 33%]
merge_test.py::TestMergeFunction::test_merge_empty_lists PASSED [ 44%]
merge_test.py::TestMergeFunction::test_merge_equal_length PASSED [ 55%]
merge_test.py::TestMergeFunction::test_merge_left_empty PASSED [ 66%]
merge_test.py::TestMergeFunction::test_merge_right_empty PASSED [ 77%]
merge_test.py::TestMergeFunction::test_merge_single_element_lists PASSED [ 88%]
merge_test.py::TestMergeFunction::test_merge_unequal_length PASSED [100%]

```

Figure 5.5: Trae 在对 merge() 函数生成单元测试用例通过结果

值得注意的是，当前测试没对两个完全相同的数组的合并的用例，即5.3.1.6.1中测试出现的第一个失败案例。通过人工添加，进行补充测试。

```

1 def test_merge_equal_element(self):
2     """测试相同元素数组合并"""
3     self.assertEqual(merge([3, 3, 3], [3, 3, 3]), [3, 3, 3, 3, 3, 3])

```

Listing 5.13: merge 单元测试用例-相同元素数组合并

经测试，可以通过 5.6。

```

collecting ... collected 10 items

merge_test.py::TestMergeFunction::test_merge_all_left_smaller PASSED [ 10%]
merge_test.py::TestMergeFunction::test_merge_all_right_smaller PASSED [ 20%]
merge_test.py::TestMergeFunction::test_merge_duplicate_elements PASSED [ 30%]
merge_test.py::TestMergeFunction::test_merge_empty_lists PASSED [ 40%]
merge_test.py::TestMergeFunction::test_merge_equal_element PASSED [ 50%]
merge_test.py::TestMergeFunction::test_merge_equal_length PASSED [ 60%]
merge_test.py::TestMergeFunction::test_merge_left_empty PASSED [ 70%]
merge_test.py::TestMergeFunction::test_merge_right_empty PASSED [ 80%]
merge_test.py::TestMergeFunction::test_merge_single_element_lists PASSED [ 90%]
merge_test.py::TestMergeFunction::test_merge_unequal_length PASSED [100%]

===== 10 passed in 0.06s =====

```

Figure 5.6: 人工补充测试用例测试结果

**5.3.1.9.2 集成测试** 集成测试仍属于功能测试，分别针对两个归并排序算法展开。即分别选择 `merge_sort_recursive` 和 `merge_sort_iterative` 两个函数，使用 `test` 命令生成单元测试用例及测试文件。测试用例包括对空数组、单个元素数组、顺序数组、逆序数组、重复元素数组、奇数和偶数长度数组、以及混合正负数的数组 8 种不同情形，比 CodeGeeX 的用例覆盖更全面。其中，测试用例 2 虽然是测试只有单个元素的列表，可以通过增加长度变成重复元素数组排序。

```

1 # 测试空列表的情况
2 def test_empty_list(self):
3     # 断言merge_sort_recursive([])的结果应该是[]
4     self.assertEqual(merge_sort_recursive([]), [])
5
6 # 测试只有一个元素的列表
7 def test_single_element(self):
8     # 断言merge_sort_recursive([5])的结果应该是[5]
9     self.assertEqual(merge_sort_recursive([5]), [5])
10
11 # 测试已经排序好的列表
12 def test_sorted_list(self):
13     # 断言merge_sort_recursive([1, 2, 3, 4])的结果应该是[1, 2, 3, 4]
14     self.assertEqual(merge_sort_recursive([1, 2, 3, 4]), [1, 2, 3, 4])
15
16 # 测试逆序排序的列表

```



```

17 def test_reverse_sorted_list(self):
18     # 断言merge_sort_recursive([4, 3, 2, 1])的结果应该是[1, 2, 3, 4]
19     self.assertEqual(merge_sort_recursive([4, 3, 2, 1]), [1, 2, 3, 4])
20
21 # 测试包含重复元素的列表
22 def test_duplicate_elements(self):
23     # 断言merge_sort_recursive([3, 1, 2, 1, 3])的结果应该是[1, 1, 2, 3, 3]
24     self.assertEqual(merge_sort_recursive([3, 1, 2, 1, 3]), [1, 1, 2, 3,
25         3])
26
27 # 测试奇数长度的列表
28 def test_odd_length_list(self):
29     # 断言merge_sort_recursive([5, 2, 8, 1, 3])的结果应该是[1, 2, 3, 5, 8]
30     self.assertEqual(merge_sort_recursive([5, 2, 8, 1, 3]), [1, 2, 3, 5,
31         8])
32
33 # 测试偶数长度的列表
34 def test_even_length_list(self):
35     # 断言merge_sort_recursive([6, 4, 2, 1, 3, 5])的结果应该是[1, 2, 3, 4,
36         5, 6]
37     self.assertEqual(merge_sort_recursive([6, 4, 2, 1, 3, 5]), [1, 2, 3,
38         4, 5, 6])
39
40 # 测试包含负数的列表
41 def test_negative_numbers(self):
42     # 断言merge_sort_recursive([-3, -1, -2, -4])的结果应该是[-4, -3, -2,
43         -1]
44     self.assertEqual(merge_sort_recursive([-3, -1, -2, -4]), [-4, -3, -2,
45         -1])
46
47 # 测试包含正数和负数的混合列表
48 def test_mixed_numbers(self):
49     # 调用merge_sort_recursive函数对列表[0, -1, 2, -3, 4]进行排序
50     # 预期结果是[-3, -1, 0, 2, 4]，即从小到大排序
51     self.assertEqual(merge_sort_recursive([0, -1, 2, -3, 4]), [-3, -1, 0,
52         2, 4])

```

Listing 5.14: merge\_sort 测试用例

通过编写和执行测试文件对 MergeSortRecursive 进行测试，获得如下结果：

考虑到如果考虑对 merge\_sort\_iterative() 实现的是相同功能，对其进行功能测试，是否可以沿用上述测试用例？是否还需要补充其它情况，或者剔除上述测试用例中的不需要的情况？

模型给出以下解答：

- 测试空列表，验证边界情况处理

```
merge_sort_recursive_test.py::TestMergeSortRecursive::test_duplicate_elements PASSED [ 11%]
merge_sort_recursive_test.py::TestMergeSortRecursive::test_empty_list PASSED [ 22%]
merge_sort_recursive_test.py::TestMergeSortRecursive::test_even_length_list PASSED [ 33%]
merge_sort_recursive_test.py::TestMergeSortRecursive::test_mixed_numbers PASSED [ 44%]
merge_sort_recursive_test.py::TestMergeSortRecursive::test_negative_numbers PASSED [ 55%]
merge_sort_recursive_test.py::TestMergeSortRecursive::test_odd_length_list PASSED [ 66%]
merge_sort_recursive_test.py::TestMergeSortRecursive::test_reverse_sorted_list PASSED [ 77%]
merge_sort_recursive_test.py::TestMergeSortRecursive::test_single_element PASSED [ 88%]
merge_sort_recursive_test.py::TestMergeSortRecursive::test_sorted_list PASSED [100%]

===== 9 passed in 0.05s =====
```

Figure 5.7: Trae 在对 Merge\_Sort\_Recursive() 函数生成单元测试用例通过结果

- 测试单元素列表，验证最小非空情况
- 测试已排序列表，验证算法不会破坏已排序状态
- 测试完全逆序列表，验证算法能处理最坏情况
- 测试普通随机顺序列表
- 测试包含重复元素的情况
- 测试奇数长度列表，验证非 2 的幂次长度处理
- 测试大型列表，验证算法性能和正确性
- 测试包含负数的情况

执行相应测试代码全部通过。

**5.3.1.9.3 性能测试** 初始 Trea 没有区分功能测试和性能测试，通过纠正错误，给出了基于 memory-profiler 的性能测试用例。测试方案说明：

时间复杂度验证：

1. 生成 10 万级不同特征 (随机/有序/逆序) 的数据
2. 记录排序耗时，验证  $O(n\log n)$  增长趋势
3. 预期结果：不同数据特征耗时相近，符合算法稳定性

空间复杂度验证：

使用 memory\_profiler 监测内存使用；预期结果：空间复杂度接近  $O(n)$  级别。

以下函数 (`generate_test_data()`) 为模型生成的三种不同随机范围的用例，通过随机数组长度和测试类型 (`case_type=` 随机/有序/逆序) 进行选择，`performance_test()` 依次选择不同类型的数据并生成长度为 100000 规模的数组进行一次测试，并依次记录当前用例的计算时间和内存占用。存在问题：1. 仅考虑了自底向上的迭代版本 (可通过人工添加)；2. 未考虑数据波动带来的影响，该测试用例依然比较单一。

```
1 def generate_test_data(size, case_type):
2     """生成不同特征的测试数据"""
3     if case_type == 'random':
4         return [random.randint(-10000, 10000) for _ in range(size)]
5     elif case_type == 'sorted':
```

```

6     return sorted([random.randint(-10000, 10000) for _ in range(size)])
7 elif case_type == 'reverse_sorted':
8     return sorted([random.randint(-10000, 10000) for _ in range(size)],
9                   reverse=True)
10
11 def performance_test():
12     """性能测试主函数"""
13     test_cases = {
14         'random': 100000, # 10万随机数
15         'sorted': 100000, # 10万已排序数
16         'reverse_sorted': 100000 # 10万逆序数
17     }
18
19     for case_type, size in test_cases.items():
20         # 生成测试数据
21         data = generate_test_data(size, case_type)
22
23         # 时间复杂度测试
24         start = time.perf_counter()
25         merge_sort_iterative(data.copy())
26         elapsed1 = time.perf_counter() - start
27
28         # 人工添加
29         start = time.perf_counter()
30         merge_sort_recursive(data.copy())
31         elapsed2 = time.perf_counter() - start
32
33
34         # 空间复杂度测试
35         mem_usage = max(memory_usage((merge_sort_iterative, (data.copy(),)))
36                          )
37
38         print(f"\n测试类型: {case_type} (规模: {size})")
39         print(f"耗时: {elapsed:.4f}秒")
40         print(f"峰值内存: {mem_usage:.2f} MiB")
41
42         # 人工添加
43         mem_usage2 = max(memory_usage((merge_sort_recursive, (data.copy(),)))
44                          )
45
46         print(f"耗时: {elapsed2:.4f}秒")
47         print(f"峰值内存: {mem_usage2:.2f} MiB")

```

Listing 5.15: merge\_sort 性能测试用例

通过对人工改进的测试用进行测试，结果如图 5.8。可以发现虽然归并排序在任意情况下的理论时间复杂度是  $n\log n$ ，但在随机情况下耗时要比已排序和逆序输入用时高，而自顶向下的递归方案在三种情况下均要比自底向上的归并略快，不同情况下的两种算法空间占用均近似。

```
性能测试结果：

测试类型：random (规模：1000000)
merge_sort_iterative:
耗时：0.3018秒
峰值内存：55.37 MiB
merge_sort_recursive:
耗时：0.2763秒
峰值内存：57.07 MiB

测试类型：sorted (规模：1000000)
merge_sort_iterative:
耗时：0.2029秒
峰值内存：59.54 MiB
merge_sort_recursive:
耗时：0.1784秒
峰值内存：60.57 MiB

测试类型：reverse_sorted (规模：1000000)
merge_sort_iterative:
耗时：0.2014秒
峰值内存：59.21 MiB
merge_sort_recursive:
耗时：0.1796秒
峰值内存：59.66 MiB
```

Figure 5.8: 一次运行下的性能测试结果

观测输入在增长规模下性能是另一种角度，如果能够进行多轮次 (如 10 次、100 次) 计算平均值来统计算法执行的时空复杂度。通过多轮次沟通 (依赖人工经验编写 Prompt)，希望包括：1) 对两种实现的归并算法进行测试；2) 只考虑性能测试；3) 对每种情况下进行多轮次测试求均值；4) 控制测试规模递增。得到如下相应测试代码：

```
1
2 # 改为动态规模控制
3 sizes = [1000, 10000, 100000, 1000000] # 新增规模控制列表
4 data_types = ["sorted", "random", "reverse"]
5 # 性能测试参数化配置
6 IMPLEMENTATIONS = [
7 ("iterative", merge_sort_iterative),
```

```

8 ("recursive", merge_sort_recursive)
9 ]
10
11 # 重构后的数据配置生成逻辑
12 DATA_CONFIG = {
13     f"{data_type}_{size}": (size, data_type)
14     for size in sizes
15     for data_type in data_types
16 }

```

Listing 5.16: merge\_sort 最终性能测试用例

通过数组和字典两种形式组织了所需要进行测试的不同测试参数，如增长规模 sizes、数据的类型、待测试的算法，并通过 pytest(前一章节介绍的) 进行多轮次测试和统计评估。相关测试函数经过一轮交互修正 (由不同 pytest-benchmark 版本下 mean 的调用方式修改造成)，测试结果如下图 5.9 所示，测试按不同输入规模、算法和数据类型进行了多轮次测试，并顺利通过测试，计算的时间增长近似  $n\log n$ ：

===== 24 passed in 121.54s (0:02:01) =====					
PASSED	[ 4%]	iterative	sorted	1000	平均耗时: 0.001280s
PASSED	[ 8%]	recursive	sorted	1000	平均耗时: 0.001120s
PASSED	[ 12%]	iterative	random	1000	平均耗时: 0.001293s
PASSED	[ 16%]	recursive	random	1000	平均耗时: 0.001604s
PASSED	[ 20%]	iterative	reverse	1000	平均耗时: 0.001287s
PASSED	[ 25%]	recursive	reverse	1000	平均耗时: 0.001148s
PASSED	[ 29%]	iterative	sorted	10000	平均耗时: 0.016207s
PASSED	[ 33%]	recursive	sorted	10000	平均耗时: 0.014109s
PASSED	[ 37%]	iterative	random	10000	平均耗时: 0.016334s
PASSED	[ 41%]	recursive	random	10000	平均耗时: 0.021590s
PASSED	[ 45%]	iterative	reverse	10000	平均耗时: 0.016471s
PASSED	[ 50%]	recursive	reverse	10000	平均耗时: 0.014401s
PASSED	[ 54%]	iterative	sorted	100000	平均耗时: 0.187444s
PASSED	[ 58%]	recursive	sorted	100000	平均耗时: 0.166752s
PASSED	[ 62%]	iterative	random	100000	平均耗时: 0.189086s
PASSED	[ 66%]	recursive	random	100000	平均耗时: 0.296768s
PASSED	[ 70%]	iterative	reverse	100000	平均耗时: 0.189765s
PASSED	[ 75%]	recursive	reverse	100000	平均耗时: 0.169989s
PASSED	[ 79%]	iterative	sorted	1000000	平均耗时: 2.148805s
PASSED	[ 83%]	recursive	sorted	1000000	平均耗时: 1.957065s
PASSED	[ 87%]	iterative	random	1000000	平均耗时: 2.168426s
PASSED	[ 91%]	recursive	random	1000000	平均耗时: 3.456657s
PASSED	[ 95%]	iterative	reverse	1000000	平均耗时: 2.142602s
PASSED	[ 100%]	recursive	reverse	1000000	平均耗时: 1.954634s

Figure 5.9: 多轮次运行下的性能测试结果

## 5.4 章节小结

本章节通过归并排序这一成熟算法为编程实例，介绍了在 pycharm 集成开发环境下，与两种不同的 AI 大模型交互编程学习的过程。归并排序存在两种实现方案，在算法是现实上通过复用 merge 函数提升了算法的模块功能独立性，但是导致代码实现比前两章的排序算法更复杂 (互相调用的子函数)。因此，在算法时空复杂度分析过程，对

应的测试用例生成变得更加复杂，即需要同时考虑单元测试和集成测试，既要考虑功能测试，还要考虑性能测试。在此基础上，由于存在子算法的互相调用，使得算法测试的用例生成和调用过程变得更加复杂，推理链条更长。

针对此问题，我们设计了一种基于方案驱动的 AI 辅助算法设计流程，除了算法本身，通过解决方案进行复杂问题拆分，测试用例生成，测试计划和验证，以及验证结果分析形成闭环流程；同时，引入两个不同公司的 AI 插件进行互相修正，进而保证生成的算法的正确性，以及通过全面的测试用例来保证测试的完备性，在交互过程中通过迭代的交互验证、交叉验证，最终理解算法设计与分析的编程实践。

# Chapter 6

## 总结

本研究通过在《算法设计与分析》课程中引入三类不同的、与 AI 大语言模型辅助的交互学习方式；为了保证知识内容的正确性，还给出了对应教材、在线课程 PPT、wikipedia 链接等基准信息。通过实际算法学习，引入了交互学习的基本方法、批判性思维如何利用 AI 辅助进行算法的设计和分析。并针对实际使用过程中出现的 AI 幻觉进行了展示，以及相应的应对方案。通过阶梯式的推进学习难度，以及与 AI 交互的技巧，将算法设计、算法分析的知识内容嵌入到了 AI 交互过程中，通过批判性思维驱动不同的模型进行左右互搏，为学习知识的准确性、算法实现的正确性、算法设计的高效性提供参考依据。

首先，在算法设计与分析基础概念学习中，介绍了如何利用 web 端大语言模型交互对话，以排序算法基础概念为核心，基于不同排序算法的难易程度、依赖关系进行了学习日程的推荐。尝试了简单的交互代码生成和 pycharm 本地测试。通过与 deepseek 和 Chat-GPT 大语言模型交互，构建了排序算法性能对比和学习顺序；在此基础上选择最简易的冒泡排序、选择排序和插入排序进行实操。利用大语言模型生成代码，持续进行代码完善；介绍了边界测试、最优情况和最差情况的算法效率度量，并引入了 Big\_O、memory\_profiler 和 pytest 等测试库，进行交互式算法效率测试，完成对算法分析的基础概念的学习和基本编程实践环境的搭建和使用。

接着，在批判性思维的交互学习一章中，通过引入两个不同 AI 大语言模型 (Chat-GPT 4o、deepseek v3 和 Gemini) 尝试了交叉对比验证，并通过在线的 Leetcode 算法平台进行算法性能测试。其中，介绍了连续对话的算法递进优化 (单一大模型持续交互的方案改进)、向单一大模型要求多种方案和与不同大模型交互方案三种策略。并将获得的多种方案在 Leetcode 的在线编译环境进行时间复杂度和空间复杂度的测试。并针对大语言模型出现的幻觉和错误，采用“多模型-多方案”交叉验证，代码在线运行测试，以及人工验证进行功能和性能上的确认。

最后，针对现有 AI 模型的潜在交互学习中可能出现的错误，本研究提出了一种面向方案驱动的 AI 辅助算法设计流程。该流程通过将复杂问题进行分治求解，生成对应的详细解决方案，并作为 Prompt 用于生成代码，以及全面的测试用例。基于不同的 Pycharm 内置的插件 AI 编程专用大模型 (CodeGeeX 和 Trae)，以及相关内置功能，开

展算法实现和测试验证。研究发现即便是十分成熟的算法，在利用实现逻辑相对复杂(算法具有两种实现方式，需要设计子函数)的归并排序进行实战时，任意单一模型均无法一次性生成正确、完整的解决方案，并通过测试。因此，通过在所总结的算法设计流程中融入以往章节中介绍的交互技巧、测试库、批判性思维等技巧，进行了多轮次的循环，最终在生成正确代码的基础上，结合人工经验完善了测试用例，在本地代码开发环境进行时空复杂度分析，并对 AI 所生成的算法进行了功能测试(单元测试、集成测试)，以及基于统计的性能测试分析。

研究发现，现有大语言模型，以及专用的编程大语言模型仍然对算法设计存在基础、且根本的认知问题，由于幻觉和错误无法避免面，使得直接、完全依赖 AI 进行教学，特别是逻辑思辨能力要求较强的 < 算法设计与分析 > 课程还有一定的差距。一方面，研究通过阶梯式的知识引入和交互技巧使用，介绍了不同的工具链组合(如第三章的 web 端 AI+pycharm+python 测试库、第四章的 web 端 AI+ 在线编译器、第五章的 Pycharm 的 AI 插件)，方便学生在任何配置电脑上不依赖 IDE 与否都能够开展学习。工具组合的另一方面是思辨能力，即算法设计与分析的基础知识要点，以及相应的批判性思维方式的形成和锻炼。最终，在不同环境下的编程实践是检验交互结果的唯一渠道，通过多种测试概念和设计方式的介绍能够辅助学生掌握基础、全面的测试评估思想。

由于时间有限、个人认知的限制，本研究尚处于较为初级的阶段，所采用的算法实例也较为简单。原本设计的志愿者教学对比试验也因为种种原因，未进行落实。因此，上述经验仅在小规模已有一定编程经验的学生上测试获得，希望后续有机会在仅掌握初级编程经验的学生上进行中等规模的测试，以便检验所提出的-方案驱动的 AI 辅助算法设计流程-是否具对学生有普适性的提升能力。



# Bibliography

- [1] M. T. Goodrich and R. Tamassia, *Algorithm design: foundations, analysis, and internet examples*. John Wiley & Sons, 2001.
- [2] 北京创新乐知网络技术有限公司. (2025) Csdn. [Online]. Available: [www.csdn.net](http://www.csdn.net)
- [3] github. (2025) Github. [Online]. Available: [github.com](https://github.com)
- [4] stack overflow. (2025) stack overflow. [Online]. Available: [stackoverflow.co](https://stackoverflow.co)
- [5] stack exchange. (2025) stack exchange. [Online]. Available: [stackexchange.com](https://stackexchange.com)
- [6] Raddit. (2025) Raddit. [Online]. Available: [www.reddit.com](http://www.reddit.com)
- [7] G. Yenduri, G. Srivastava, P. K. R. Maddikunta, R. H. Jhaveri, W. Wang, A. V. Vasilakos, T. R. Gadekallu *et al.*, “Generative pre-trained transformer: A comprehensive review on enabling technologies, potential applications, emerging challenges, and future directions,” *arXiv preprint arXiv:2305.10435*, 2023.
- [8] Y. Chang, X. Wang, J. Wang, Y. Wu, L. Yang, K. Zhu, H. Chen, X. Yi, C. Wang, Y. Wang *et al.*, “A survey on evaluation of large language models,” *ACM transactions on intelligent systems and technology*, vol. 15, no. 3, pp. 1–45, 2024.
- [9] A. Jobin, M. Ienca, and E. Vayena, “The global landscape of ai ethics guidelines,” *Nature machine intelligence*, vol. 1, no. 9, pp. 389–399, 2019.
- [10] Copilot. (2025) Copilot. [Online]. Available: [copilot.microsoft.com](https://copilot.microsoft.com)
- [11] L. Reynolds and K. McDonell, “Prompt programming for large language models: Beyond the few-shot paradigm,” in *Extended abstracts of the 2021 CHI conference on human factors in computing systems*, 2021, pp. 1–7.
- [12] F. F. Xu, U. Alon, G. Neubig, and V. J. Hellendoorn, “A systematic evaluation of large language models of code,” in *Proceedings of the 6th ACM SIGPLAN international symposium on machine programming*, 2022, pp. 1–10.

- [13] G. Destefanis, S. Bartolucci, and M. Ortu, “A preliminary analysis on the code generation capabilities of gpt-3.5 and bard ai models for java functions,” *arXiv preprint arXiv:2305.09402*, 2023.
- [14] J. Prather, B. N. Reeves, P. Denny, B. A. Becker, J. Leinonen, A. Luxton-Reilly, G. Powell, J. Finnie-Ansley, and E. A. Santos, “‘it’ s weird that it knows what i want” : Usability and interactions with copilot for novice programmers,” *ACM transactions on computer-human interaction*, vol. 31, no. 1, pp. 1–31, 2023.
- [15] J. Savelka, A. Agarwal, M. An, C. Bogart, and M. Sakr, “Thrilled by your progress! large language models (gpt-4) no longer struggle to pass assessments in higher education programming courses,” in *Proceedings of the 2023 ACM Conference on International Computing Education Research-Volume 1*, 2023, pp. 78–92.
- [16] B. P. Cipriano and P. Alves, “Gpt-3 vs object oriented programming assignments: An experience report,” in *Proceedings of the 2023 Conference on Innovation and Technology in Computer Science Education V. 1*, 2023, pp. 61–67.
- [17] —, “Llms still can’t avoid instanceof: An investigation into gpt-3.5, gpt-4 and bard’s capacity to handle object-oriented programming assignments,” in *Proceedings of the 46th International Conference on Software Engineering: Software Engineering Education and Training*, 2024, pp. 162–169.
- [18] J. Finnie-Ansley, P. Denny, A. Luxton-Reilly, E. A. Santos, J. Prather, and B. A. Becker, “My ai wants to know if this will be on the exam: Testing openai’ s codex on cs2 programming exercises,” in *Proceedings of the 25th Australasian Computing Education Conference*, 2023, pp. 97–104.
- [19] P. Denny, J. Leinonen, J. Prather, A. Luxton-Reilly, T. Amarouche, B. A. Becker, and B. N. Reeves, “Prompt problems: A new programming exercise for the generative ai era,” in *Proceedings of the 55th ACM Technical Symposium on Computer Science Education V. 1*, 2024, pp. 296–302.
- [20] P. Denny, D. H. Smith IV, M. Fowler, J. Prather, B. A. Becker, and J. Leinonen, “Explaining code with a purpose: An integrated approach for developing code comprehension and prompting skills,” in *Proceedings of the 2024 on Innovation and Technology in Computer Science Education V. 1*, 2024, pp. 283–289.
- [21] C. Sengul, R. Neykova, and G. Destefanis, “Software engineering education in the era of conversational ai: current trends and future directions,” *Frontiers in Artificial Intelligence*, vol. 7, p. 1436350, 2024.

- [22] P. Alves and B. P. Cipriano, ““give me the code”—log analysis of first-year cs students’ interactions with gpt,” *arXiv preprint arXiv:2411.17855*, 2024.
- [23] H. M. Babe, S. Nguyen, Y. Zi, A. Guha, M. Q. Feldman, and C. J. Anderson, “Studenteval: A benchmark of student-written prompts for large language models of code,” *arXiv preprint arXiv:2306.04556*, 2023.
- [24] M. Reid and S. S. Vempala, “Does gpt really get it? a hierarchical scale to quantify human and ai’ s understanding of algorithms,” in *Proceedings of the AAAI Conference on Artificial Intelligence*, vol. 39, no. 2, 2025, pp. 1492–1500.
- [25] C. Lei, Y. Chang, N. Lipovetzky, and K. A. Ehinger, “Planning-driven programming: A large language model programming workflow,” *arXiv preprint arXiv:2411.14503*, 2024.
- [26] J. Achiam, S. Adler, S. Agarwal, L. Ahmad, I. Akkaya, F. L. Aleman, D. Almeida, J. Altenschmidt, S. Altman, S. Anadkat *et al.*, “Gpt-4 technical report,” *arXiv preprint arXiv:2303.08774*, 2023.
- [27] OpenAI. (2025) Openai. [Online]. Available: [openai.com](https://openai.com)
- [28] G. Team, R. Anil, S. Borgeaud, J.-B. Alayrac, J. Yu, R. Soricut, J. Schalkwyk, A. M. Dai, A. Hauth, K. Millican *et al.*, “Gemini: a family of highly capable multimodal models,” *arXiv preprint arXiv:2312.11805*, 2023.
- [29] A. Liu, B. Feng, B. Xue, B. Wang, B. Wu, C. Lu, C. Zhao, C. Deng, C. Zhang, C. Ruan *et al.*, “Deepseek-v3 technical report,” *arXiv preprint arXiv:2412.19437*, 2024.
- [30] M. P. Cosentino and T. Stamper, “ifly: Code development for an app to support automating entomological data collection,” 2016.
- [31] comate. (2025) comate. [Online]. Available: [comate.baidu.com](https://comate.baidu.com)
- [32] A. Ghoshal, “Amazon q, la respuesta de aws a copilot de microsoft.” *Computerworld Spain*, pp. NA–NA, 2023.
- [33] Q. Zheng, X. Xia, X. Zou, Y. Dong, S. Wang, Y. Xue, L. Shen, Z. Wang, A. Wang, Y. Li *et al.*, “Codegeex: A pre-trained model for code generation with multilingual benchmarking on humaneval-x,” in *Proceedings of the 29th ACM SIGKDD Conference on Knowledge Discovery and Data Mining*, 2023, pp. 5673–5684.
- [34] marscode. (2025) marscode. [Online]. Available: [www.marscode.com](https://www.marscode.com)

- [35] jetbrains. (2025) pycharm. [Online]. Available: [www.jetbrains.com.cn/en-us/pycharm/](http://www.jetbrains.com.cn/en-us/pycharm/)
- [36] P. Wang and T. Oliphant. (2025) anaconda. [Online]. Available: <https://www.anaconda.com/>