同济大学计算机科学与技术学院

数据结构与算法设计实验报告--期中大作业

姓名：朱俊泽

学号：2351114

专业：数据科学与大数据技术

教师：张亚英

期中大作业

1. 问题背景

1.1. 排序算法问题背景

排序是计算机科学中最基础且最常见的问题之一。排序算法的任务是将一组数据按照某种顺序（通常是升序或降序）重新排列。排序算法在各种应用中都具有重要作用，例如数据库管理、数据检索、网络流量分析和数据可视化等。

排序的效率直接影响到计算机程序的性能，尤其在处理大数据量时。比如，排序是许多搜索算法（如二分查找）的基础，它依赖于排序好的数据来提高搜索效率。在数据库管理系统中，排序通常是查询优化的重要一环，而大规模的数据处理、数据分析和机器学习模型训练也常常需要高效的排序算法。

排序算法的选择通常取决于数据的规模、数据的分布情况以及对性能的要求。不同的排序算法有不同的时间复杂度、空间复杂度和稳定性等特点。例如，常见的排序算法包括冒泡排序、插入排序、选择排序（时间复杂度均为$O(n^2)$），快速排序、归并排序和堆排序（时间复杂度为$O(n \log n)$），以及基数排序（适用于某些特殊情况）。了解排序算法的背景和适用场景有助于开发者根据实际需求选择最合适的排序方法。

1.2. 分治算法问题背景

分治算法是一种重要的算法设计思想，通常用于处理可以被分解成多个较小子问题的复杂问题。其核心思想是将一个大问题分解成多个规模较小、相似的子问题，然后分别求解这些子问题，最终合并子问题的解得到原问题的解。分治算法的典型应用有归并排序、快速排序和二分查找等。

分治算法的关键优势在于其递归性质，通过将问题拆分为子问题可以简化问题的复杂度，并有效地利用递归的思想来解决问题。分治方法适用于那些具有“子问题独立性”和“可合并性”的问题。例如，排序问题（如快速排序和归并排序）和求解最大子数组问题等。每个子问题的规模比原问题小，并且子问题之间可以独立解决，最终合并子问题的结果时，往往可以得到问题的最优解。

分治算法的时间复杂度通常为$O(\log n)$，其中n为问题的规模。该算法特别适合解决大规模问题，因为它能够通过递归将问题规模缩小，每一步都能有效地简化问题。然而，分治算法也有一定的局限性，对于那些子问题不能独立求解、无法有效合并的情况，分治算法可能不适用。

1.3. 动态规划问题背景

动态规划（Dynamic Programming, DP）是一种用于解决最优化问题的算法思想，适用于具有重叠子问题和最优子结构性质的问题。动态规划的核心思想是通过将大问题拆解为小问题来逐步求解，避免了重复计算子问题的结果，从而提高算法的效率。

动态规划算法通常用于求解具有最优解的组合问题，如最短路径问题、背包问题、最长公共子序列问题等。最优子结构是指问题的最优解包含其子问题的最优解，而重叠子问题则指问题在求解过程中会多次出现相同的子问题。

与分治算法不同，动态规划通常是通过记忆化递归或自底向上的方式来解决问题。它通过保存子问题的结果（通常使用表格或数组）来避免重复计算，从而提高效率。例如，在计算斐波那契数列时，动态规划避免了重复计算同一个子问题的值。常见的动态规划算法有背包问题、最短路径问题、矩阵链乘法、最长公共子序列问题等。

动态规划的时间复杂度通常取决于问题的规模和状态空间的大小。虽然动态规划方法可能需要更多的内存空间来存储子问题的中间结果，但它通常可以显著提高解决问题的效率，尤其是在涉及大量重复计算的情况下。

2. 问题描述

2.1. 排序算法综合应用题

假设高考科目包括语文、数学、英语这三门，每门课的分数为非负整数，满分为150分。现在要对某个地区的考生，根据他们的得分情况进行排名，按总分从高到低进行排序，总分相同的情况下按照先语文后数学再英语的顺序进行排名。

例：

排名	考号	姓名	总分	语文	数学	英语
1	XXXX	XXX	450	150	150	150
2			449	149	150	150
3			448	150	150	148
4			448	150	149	149
4			448	150	149	149
6			448	150	148	150
7			448	149	150	149
8			445	150	150	145

表1.排序数据示例

请设计合适的算法，完成上述任务。

输入：考生数为n，考生的成绩由程序随机生成，并写入文本文件。成绩尽可能符合正态分布。

输出：符合要求的考生分数排名表，以文本文件的形式记录。

分段设置n的范围，采用你认为最合适的排序算法。

（1）n较小（1-50,000），待排元素能一次放进内存； 并给出排序时间随n变化情况（用曲线图表示）

（2）n较大，输入正整数m，m<<n，求所有n位考生中排名前m位的考生。

2.2. 分治算法综合应用题

给定线性序集中n个元素和一个整数k，1≤k≤n，要求找出这n个元素中第k小的元素，（这里给定的线性集是无序的）。下面三种是可行的方法：

（1）基于堆的选择：不需要对全部n个元素排序，只需要维护k个元素的最大堆，即用容量为k的最大堆存储最小的k个数，总费时O（k+（n-k）*logk）

（2）随机划分线性选择（教材上的RandomizedSelect)：在最坏的情况下时间复杂度为O(n2),平均情况下期望时间复杂度为O(n)。

（3）利用中位数的线性时间选择：选择中位数的中位数作为划分的基准，在最坏情况下时间复杂度为O(n) 。

请给出以上三种方法的算法描述，用你熟悉的编程语言实现上述三种方法。并通过实际用例测试，绘出三种算法的运行时间随k和n变化情况的对比图（表），特别是n较大时方法（2）和（3）的对比。

2.3. 动态规划综合应用题

迈克叔叔去世，遗嘱中留下了一些宝贵的宝可梦卡片，给了他的两个孙子，艾比（Abe）和鲍勃（Bob）。遗嘱中的唯一要求是“尽量平分卡片的价值”。作为迈克叔叔遗嘱的执行人，你已经给每张宝可梦卡片定了价格。现在你需要决定如何将这些卡片分成两堆，以最小化每堆卡片价值的差异。

例如，假设你有以下8张宝可梦卡片：

卡片	c_1	c_2	c_3	c_4	c_5	c_6	c_7	c_8
价值	2	1	3	1	5	2	3	4

表2.卡片数据示例

经过很多努力，你发现可以按以下方式将卡片分配：

艾比（Abe）：c_1, c_3, c_5 ，总价值为10。

鲍勃（Bob）：c_2, c_4, c_6 , c_7 , c_8 ，总价值为11。

这给艾比带来了10美元的卡片，而鲍勃得到11美元的卡片。这是最佳的分配吗？你的任务是针对n张卡片解决这个问题，其中每张卡片c_i 都有一个正整数值v_i 。你的解决方案是计算卡片应该如何分配，以及每堆的价值。

接下来，你需要完成以下任务：

1：用暴力破解的方式解决这个问题，检查所有可能的分配方式。给出此暴力破解算法的时间复杂度分析，并通过实现和实验验证你的分析结果。

2：通过动态规划开发一个更高效的算法。你应该首先分析这个问题的动态规划思路，并编写相应的递归公式。你还可以展示存储中间结果的数组设计（支持绘制表格的markdown）。然后，给出此算法的时间复杂度分析，并通过实现和实验验证分析结果。

1.

3. 实验设计

3.1. 程序语言、环境设置

进行实验之前首先对程序语言进行设定，这里我们使用python语言，创建虚拟环境mid作为本次算法与数据结构期中大作业的虚拟环境。

虚拟环境创建

```
conda env create -n mid python=3.8
conda activate mid
```

3.2. 程序输入

在进行算法实现之前首先确定程序的输入该如何设置。主要是基于随机生成的txt文本数据。

3.2.1 排序输入

考生数为n，考生的成绩由程序随机生成，并写入文本文件。成绩尽可能符合正态分布。利用python脚本生成，生成两份，一份满足第一小问的小规模数据集n在1-50000之间，另一份满足大规模数据集，至少达到1e6的量。

输入格式：学号、总分、语文、数学、英语成绩。

为了让数据第一第二问通用，直接设定第二问中m为10。n使用数据量大的1e7版本。

sort_data_produce.py

```python
import random
def generate_data(num_records, filename="students_scores.txt"):
    # 设置正态分布的均值和标准差
    mean = 75   # 平均值
    std_dev = 20  # 标准差
    with open(filename, "w") as file:
        # 写入标题
        file.write("StudentID,TotalScore,Chinese,Math,English\n")
        for _ in range(num_records):
            # 随机生成七位学号
            student_id = random.randint(1000000, 9999999)
            # 使用正态分布生成语文、数学、英语成绩
            chinese = int(random.gauss(mean, std_dev))
            math = int(random.gauss(mean, std_dev))
            english = int(random.gauss(mean, std_dev))
            # 确保分数在 0 到 150 之间
            chinese = max(0, min(chinese, 150))
            math = max(0, min(math, 150))
            english = max(0, min(english, 150))
            # 计算总分
```

```
            total_score = chinese + math + english
            # 写入到文件，每个数据项之间用英文逗号隔开
            file.write(f"{student_id},{total_score},{chinese},{math},{english}\n")
    print(f"数据已成功生成并保存到 {filename}")
# 控制生成数据的条数
num_records = 10  # 例如生成 10 条数据
generate_data(num_records)
```

3.2.2 分治输入

给定线性序集中n个元素和一个整数k，1≤k≤n，要求找出这n个元素中第k小的元素，（这里给定的线性集是无序的）。题目需要获得数据量在一个较大范围内的值，就暂定从10以10倍增长到1e7的集合元素数量。注意是集合，集合中元素是不重复的，因此需要注意元素中数据不能重复。

输入格式：k值 然后就是集合每一个元素

find_k-th_data_produce.py

```
import random
def generate_data(n, filename="find_k-th_data-100.txt"):
    # 随机生成 k 值（1 <= k <= n）
    k = random.randint(1, n)
    # 生成一个不重复的元素集合
    # 使用 set 保证集合中的元素唯一
    data = random.sample(range(1, 10*n), n)  # 生成 n 个不重复的元素，范围从 1 到 100
    # 写入文件
    with open(filename, "w") as file:
        # 输出 k 值
        file.write(f"Random k: {k}\n")
        # 输出 n 个元素，元素之间用空格分隔
        file.write(" ".join(map(str, data)) + "\n")
    print(f"数据已生成并保存到 {filename}")
    print(f"Random k: {k}")
    print("Data:", data)
# 控制生成数据的条数
n = 100  # 生成 10 个不重复的元素
generate_data(n)
```

3.2.3 动态规划输入

给定卡片数量n，还有n张卡片的分别价值。

输入格式：卡片数量n，后续每一张卡片的价值分开输入。

cards_data_produce.py

```
import random
def generate_data(num_lines, filename="cards_data.txt"):
    with open(filename, "w") as file:
```

```
        for _ in range(num_lines):
            # 随机设定卡片数量 n，范围是 10 到 10000
            n = random.randint(10, 10000)
            # 生成一个长度为 n 的数组，数组的元素值小于等于 n
            array = [random.randint(1, n) for _ in range(n)]
            # 写入文件：先写入 n，然后是由逗号分隔的数组元素
            file.write(f"{n}," + ",".join(map(str, array)) + "\n")
    print(f"数据已成功生成并保存到 {filename}")
# 控制生成数据的行数
num_lines = 1000  # 生成 10 行数据
generate_data(num_lines)
```

4. 算法分析和实现：

4.1 排序问题

4.1.1 排序问题（1）

n较小（1-50,000），待排元素能一次放进内存； 并给出排序时间随n变化情况。此处展示对第一问算法的实现，排序时间等分析在实验结果中展示。

1. 快速排序：

通过一趟排序将要排序的数据分割成独立的两部分，其中一部分的所有数据都比另外一部分的所有数据都要小，然后再按此方法对这两部分数据分别进行快速排序，整个排序过程可以递归进行，以此达到整个数据变成有序序列。

快速排序算法通过多次比较和交换来实现排序，其排序流程如下：

首先设定一个分界值，通过该分界值将数组分成左右两部分。

将大于或等于分界值的数据集中到数组右边，小于分界值的数据集中到数组的左边。此时，左边部分中各元素都小于或等于分界值，而右边部分中各元素都大于或等于分界值。

然后，左边和右边的数据可以独立排序。对于左侧的数组数据，又可以取一个分界值，将该部分数据分成左右两部分，同样在左边放置较小值，右边放置较大值。右侧的数组数据也可以做类似处理。

重复上述过程，可以看出，这是一个递归定义。通过递归将左侧部分排好序后，再递归排好右侧部分的顺序。当左、右两个部分各数据排序完成后，整个数组的排序也就完成了。代码如下所示：

Quick_sort.py

```
def quicksort(arr, low, high):
    if low < high:
        # 获取划分索引
        pi = partition(arr, low, high)
        # 对左右两部分进行递归排序
        quicksort(arr, low, pi - 1)
        quicksort(arr, pi + 1, high)
```

```
    def partition(arr, low, high):
        # 选择最后一个元素作为枢轴
        pivot = arr[high]
        i = low - 1  # i 是小于枢轴的元素的索引
        # 遍历并对比每个元素
        for j in range(low, high):
            # 如果当前元素大于枢轴元素
            if arr[j][1] > pivot[1] or (arr[j][1] == pivot[1] and arr[j][2] > pivot[2]) or \
               (arr[j][1] == pivot[1] and arr[j][2] == pivot[2] and arr[j][3] > pivot[3]):
                i += 1
                arr[i], arr[j] = arr[j], arr[i]
        # 把枢轴放到正确的位置
        arr[i + 1], arr[high] = arr[high], arr[i + 1]
        return i + 1
    def read_data(filename):
        students = []
        with open(filename, "r") as file:
            lines = file.readlines()
            for line in lines[1:]:  # 跳过第一行标题
                data = line.strip().split(",")
                student_id = int(data[0])
                total_score = int(data[1])
                chinese = int(data[2])
                math = int(data[3])
                english = int(data[4])
                students.append([student_id, total_score, chinese, math, english])
        return students
    def write_sorted_data(filename, students):
        with open(filename, "w") as file:
            file.write("StudentID,TotalScore,Chinese,Math,English\n")
            for student in students:
                file.write(f"{student[0]},{student[1]},{student[2]},{student[3]},{student[4]}\n")
    def main():
        # 读取数据
        students = read_data(r"C:\Users\asus\Desktop\Data-Structures-Algorithms-designing\期中大作
业\students_scores-data\students_scores_10000.txt")

        # 快速排序
        quicksort(students, 0, len(students) - 1)

        # 输出排序结果
        write_sorted_data("sorted_students_scores.txt", students)
        print("数据已排序并保存到 sorted_students_scores.txt")
    if __name__ == "__main__":
```

```
main()
```

2. **归并排序：**

归并排序（Merge sort）是建立在归并操作上的一种有效的排序算法，该算法是采用分治法（Divide and Conquer）的一个非常典型的应用。将已有序的子序列合并，得到完全有序的序列；即先使每个子序列有序，再使子序列段间有序。

归并排序算法有两个基本的操作，一个是分，也就是把原数组划分成两个子数组的过程。另一个是治，它将两个有序数组合并成一个更大的有序数组。

将待排序的线性表不断地切分成若干个子表，直到每个子表只包含一个元素，这时，可以认为只包含一个元素的子表是有序表。

将子表两两合并，每合并一次，就会产生一个新的且更长的有序表，重复这一步骤，直到最后只剩下一个子表，这个子表就是排好序的线性表。如图：

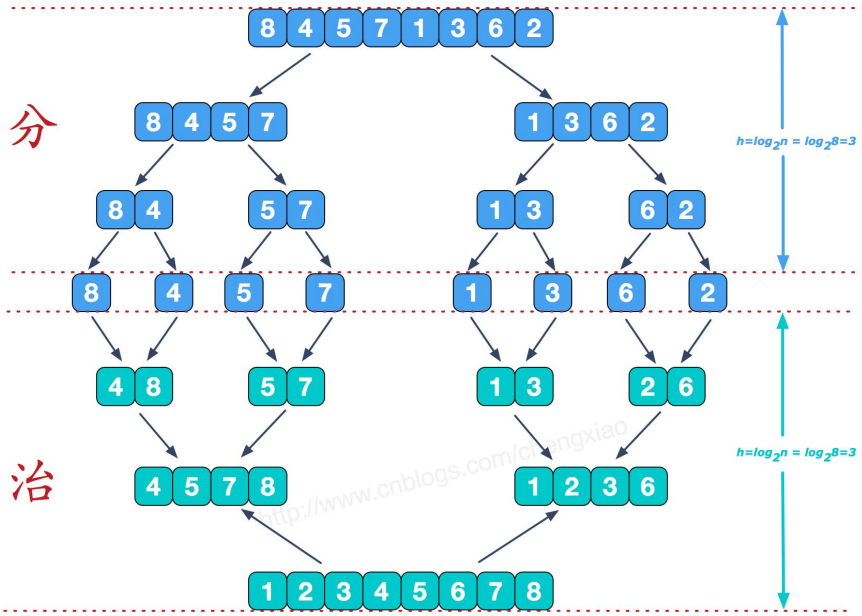

图1.归并排序图示

代码如下所示：

Merge_sort.py

```
import random
import time
# 归并排序算法
def merge_sort(arr):
    if len(arr) > 1:
        mid = len(arr) // 2
        left_half = arr[:mid]
        right_half = arr[mid:]
        merge_sort(left_half)
        merge_sort(right_half)
```

```
            i = j = k = 0
            while i < len(left_half) and j < len(right_half):
                if left_half[i][1] > right_half[j][1] or (left_half[i][1] == right_half[j][1] and
left_half[i][2] > right_half[j][2]) or \
                   (left_half[i][1] == right_half[j][1] and left_half[i][2] == right_half[j][2] and
left_half[i][3] > right_half[j][3]):
                    arr[k] = left_half[i]
                    i += 1
                else:
                    arr[k] = right_half[j]
                    j += 1
                k += 1
            while i < len(left_half):
                arr[k] = left_half[i]
                i += 1
                k += 1
            while j < len(right_half):
                arr[k] = right_half[j]
                j += 1
                k += 1
    # 读取数据
    def read_data(filename):
        students = []
        with open(filename, "r") as file:
            lines = file.readlines()
            for line in lines[1:]:
                data = line.strip().split(",")
                student_id = int(data[0])
                total_score = int(data[1])
                chinese = int(data[2])
                math = int(data[3])
                english = int(data[4])
                students.append([student_id, total_score, chinese, math, english])
        return students
    # 写入排序后的数据
    def write_sorted_data(filename, students):
        with open(filename, "w") as file:
            file.write("StudentID,TotalScore,Chinese,Math,English\n")
            for student in students:
                file.write(f"{student[0]},{student[1]},{student[2]},{student[3]},{student[4]}\n")
    def main():
        # 记录程序开始时间
        start_time = time.time()
        # 读取数据
```

```
    students = read_data(r"C:\Users\asus\Desktop\Data-Structures-Algorithms-designing\期中大作业\students_scores-data\students_scores_10000.txt")
    # 选择排序算法
    merge_sort(students)  # 使用归并排序
    # 输出排序结果
    write_sorted_data("sorted_students_scores.txt", students)
    # 记录程序结束时间
    end_time = time.time()
    # 计算并输出运行时间
    print(f"数据已排序并保存到 sorted_students_scores.txt")
    print(f"算法运行时间: {end_time - start_time:.4f} 秒")
if __name__ == "__main__":
    main()
```

3. 堆排序：

首先堆是一种完全二叉树的数据结构，可以分为大根堆小根堆。堆排序就是基于这种结构产生的算法。最大堆（Max Heap）：在最大堆中，每个父节点的值都大于或等于其子节点的值。最小堆（Min Heap）：在最小堆中，每个父节点的值都小于或等于其子节点的值。

堆排序的基本步骤：

构建堆：堆排序首先将待排序的数组构建成一个最大堆（或最小堆），在最大堆中，堆顶元素是数组中的最大元素。

交换堆顶元素与最后一个元素：将最大堆的堆顶元素（最大值）与堆的最后一个元素交换，交换之后，最大值已经被放到数组的正确位置。

调整堆：在交换堆顶元素后，堆的结构被破坏，因此需要重新调整堆，确保堆顶元素满足堆的性质（最大堆或最小堆）。

这一过程是通过堆化操作完成的，堆化是调整堆的核心过程，它从堆的根节点开始，依次比较父节点和子节点的大小，并交换位置，直到堆的性质得到恢复。

重复堆化：继续将剩下的元素调整为堆，直到所有元素都被排序。

最终结果：每次调整堆后，最大的元素（在最大堆中）会被放到数组的正确位置，经过 n-1 次操作，整个数组就会变成一个有序的数组。

代码如下所示：

Heap_sort.py

```
import random
import time
# 堆排序算法
def heapify(arr, n, i):
    largest = i
    left = 2 * i + 1
    right = 2 * i + 2
    if left < n and (arr[left][1] > arr[largest][1] or (arr[left][1] == arr[largest][1] and
```

```
arr[left][2] > arr[largest][2]) or \
                        (arr[left][1] == arr[largest][1] and arr[left][2] == arr[largest][2] and
arr[left][3] > arr[largest][3])):
            largest = left
        if right < n and (arr[right][1] > arr[largest][1] or (arr[right][1] == arr[largest][1] and
arr[right][2] > arr[largest][2]) or \
                         (arr[right][1] == arr[largest][1] and arr[right][2] == arr[largest][2] and
arr[right][3] > arr[largest][3])):
            largest = right
        if largest != i:
            arr[i], arr[largest] = arr[largest], arr[i]
            heapify(arr, n, largest)
    def heap_sort(arr):
        n = len(arr)
        for i in range(n // 2 - 1, -1, -1):
            heapify(arr, n, i)
        for i in range(n-1, 0, -1):
            arr[i], arr[0] = arr[0], arr[i]
            heapify(arr, i, 0)
    # 读取数据
    def read_data(filename):
        students = []
        with open(filename, "r") as file:
            lines = file.readlines()
            for line in lines[1:]:
                data = line.strip().split(",")
                student_id = int(data[0])
                total_score = int(data[1])
                chinese = int(data[2])
                math = int(data[3])
                english = int(data[4])
                students.append([student_id, total_score, chinese, math, english])
        return students
    # 写入排序后的数据
    def write_sorted_data(filename, students):
        with open(filename, "w") as file:
            file.write("StudentID,TotalScore,Chinese,Math,English\n")
            for student in students:
                file.write(f"{student[0]},{student[1]},{student[2]},{student[3]},{student[4]}\n")
    def main():
        # 记录程序开始时间
        start_time = time.time()
        # 读取数据
        students = read_data(r"C:\Users\asus\Desktop\Data-Structures-Algorithms-designing\期中大作
```

```
业\students_scores-data\students_scores_10000.txt")
        # 选择排序算法
        heap_sort(students)  # 使用堆排序
        # 输出排序结果
        write_sorted_data("sorted_students_scores.txt", students)
        # 记录程序结束时间
        end_time = time.time()
        # 计算并输出运行时间
        print(f"数据已排序并保存到 sorted_students_scores.txt")
        print(f"算法运行时间: {end_time - start_time:.4f} 秒")
    if __name__ == "__main__":
        main()
```

4.1.2 排序问题（2）

n较大，输入正整数m，m<<n，求所有n位考生中排名前m位的考生。相比于第一问差别就是第二问不需要对所有数组排序，也不需要一次性读取所有考生。这时直接考虑维护一个最小堆。

维护一个大小为 10 的最小堆。在处理每个新的考生时，如果堆中的元素少于 10 个，就将该考生加入堆中；如果堆中已经有 10 个元素，则比较当前考生的分数和堆顶（最小的元素），如果当前考生的分数更大，则替换堆顶元素。

这样，在遍历完所有考生后，堆中存储的就是前 10 名考生的分数。

Top-k. py

```
import heapq
# 自定义最小堆
class MinHeap:
    def __init__(self, k):
        self.k = k  # 堆的最大容量
        self.heap = []
    def insert(self, student):
        # 转换为元组，负值用于保留最大的 k 个元素
        heap_item = (student[1], student[2], student[3], student[4], student[0])
        heapq.heappush(self.heap, heap_item)
        # 如果堆大小超过 k，弹出最小元素
        if len(self.heap) > self.k:
            heapq.heappop(self.heap)
    def get_heap(self):
        # 将堆中的元素转换回原始格式并按升序排序
        result = []
        for item in self.heap:
            # 还原为 [student_id, total_score, chinese, math, english]
            student = [item[4], item[0], item[1], item[2], item[3]]
            result.append(student)
```

```
            # 按总分、语文、数学、英语升序排序
            result.sort(key=lambda x: (x[1], x[2], x[3], x[4]))
            return result
    # 读取数据
    def read_data(filename):
        students = []
        try:
            with open(filename, "r") as file:
                lines = file.readlines()
                for line in lines[1:]:  # 跳过第一行标题
                    try:
                        data = line.strip().split(",")
                        student_id = int(data[0])
                        total_score = int(data[1])
                        chinese = int(data[2])
                        math = int(data[3])
                        english = int(data[4])
                        students.append([student_id, total_score, chinese, math, english])
                    except (IndexError, ValueError):
                        print(f"Invalid data in line: {line}")
                return students
        except FileNotFoundError:
            print(f"File {filename} not found")
            return [
    def main():
        # 读取数据
        filename = r"C:\Users\asus\Desktop\Data-Structures-Algorithms-designing\期中大作业
\students_scores-data\students_scores_1000.txt"
        students = read_data(filename)
        # 使用最小堆找出前 10 名学生
        min_heap = MinHeap(k=10)
        for student in students:
            min_heap.insert(student)
        # 获取并输出前 10 名学生
        top_10_students = min_heap.get_heap()
        print("前 10 名学生（按总分、语文、数学、英语升序排序）:")
        for student in top_10_students:
            print(f"学号: {student[0]}, 总分: {student[1]}, 语文: {student[2]}, 数学: {student[3]}, 英
语: {student[4]}")
        # 写入结果到 txt 文件
        output_filename = r"C:\Users\asus\Desktop\Data-Structures-Algorithms-designing\期中大作业
\top_10_students.txt"
        try:
            with open(output_filename, "w", encoding="utf-8") as file:
```

```
                file.write("前 10 名学生（按总分、语文、数学、英语升序排序）:\n")
                file.write("ID,Total,Chinese,Math,English\n")
                for student in top_10_students:
                    file.write(f"{student[0]},{student[1]},{student[2]},{student[3]},{student[4]}\
n")
            print(f"结果已成功写入到 {output_filename}")
        except IOError:
            print(f"写入文件 {output_filename} 失败")
    if __name__ == "__main__":
        main()
```

4.2 分治问题

4.2.1 分治问题（1）

基于堆的选择：不需要对全部n个元素排序，只需要维护k个元素的最大堆，即用容量为k的最大堆存储最小的k个数，总费时O（k+（n-k）*logk）

实现过程：

堆的基本原理：该算法利用一个最大堆（max-heap），维护最小的 k 个元素。最大堆的特性是堆顶元素是当前堆中的最大元素。在处理过程中，我们逐个遍历数据中的元素，如果当前堆中的元素数量小于 k，则直接将元素加入堆。如果堆中的元素数量已经达到了 k，我们比较当前元素与堆顶元素（即最大堆中的最大值）。如果当前元素比堆顶元素小，说明当前元素可以替代堆中的最大值。替换堆顶元素后，堆的大小不变，因此我们需要通过堆化操作来保持堆的性质。总的时间复杂度：O(n log k)。当 k 很小的时候，log k 是一个常数，因此算法的时间复杂度可以接近 O(n)。

具体步骤：

初始化：初始化一个大小为 k 的最大堆。

遍历给定的数组：如果堆中元素少于 k，将当前元素加入堆。如果堆中已有 k 个元素，比较堆顶元素（最大元素）与当前元素的大小，若当前元素较小，则替换堆顶元素并调整堆。遍历完成后，堆顶元素即为第 k 小的元素。

4.2.2 分治问题（2）

随机划分线性选择 (教材上的RandomizedSelect)：在最坏的情况下时间复杂度为O(n2),平均情况下期望时间复杂度为O(n)。

该算法是快速排序的变种，它不需要完全排序数组，而是通过递归地划分数据来查找第 k 小的元素。每次选择一个随机的基准元素（pivot），将数组分为两部分：小于基准的元素和大于基准的元素。根据 k 的位置，递归选择合适的部分继续划分，直到找到第 k 小的元素。该算法的关键是基于“划分”操作，在每次划分时可以消除一半的元素。

步骤：

随机选择一个基准元素（pivot）。将数据分为两部分：一部分小于基准，另一部分大于基准。根据 k 的位置：

如果 k 位于基准的左边部分，则递归处理左边的部分。

如果 k 位于基准的右边部分，则递归处理右边的部分。

如果 k 恰好是基准元素，则返回基准元素。

重复这一过程，直到找到第 k 小的元素。

最坏情况时间复杂度为 $O(n^2)$，如果输入数据很糟糕（如已排序或逆序），效率会显著下降。

4.2.3 分治问题（3）

利用中位数的线性时间选择：选择中位数的中位数作为划分的基准，在最坏情况下时间复杂度为$O(n)$ 。

该算法是对随机划分线性选择的优化。它通过选择中位数的中位数作为基准来避免最坏情况的发生，从而确保在最坏情况下时间复杂度为 $O(n)$。首先将数据分为若干个小组，每个小组包含 5 个元素。对于每个小组，计算其中位数。然后找到这些中位数的中位数，将其作为新的基准进行数据划分。通过这种方式，我们可以保证每次划分都能有效地缩小搜索范围。

将数据分成若干个小组，每组最多 5 个元素。对每个小组，找到其中位数。递归地选取这些中位数的中位数，作为基准（pivot）。使用基准将数据划分成两部分：小于基准的元素和大于基准的元素。

根据 k 的位置：

如果 k 位于基准的左边部分，则递归处理左边的部分。

如果 k 位于基准的右边部分，则递归处理右边的部分。

如果 k 恰好是基准元素，则返回基准元素。

重复这一过程，直到找到第 k 小的元素。

三个分治方法的代码实现如下：

Find_k-th-3.py

```python
import heapq
import random
import time
# 基于堆的选择算法类
class HeapSelection:
    def __init__(self, k):
        self.k = k

    def find_kth_smallest(self, data):
        # 使用最大堆，堆的大小为 k
        max_heap = []
        for i in range(len(data)):
            if len(max_heap) < self.k:
                heapq.heappush(max_heap, -data[i])  # 插入负值来模拟最大堆
            else:
```

```
                if data[i] < -max_heap[0]:
                    heapq.heappop(max_heap)
                    heapq.heappush(max_heap, -data[i])

        return -max_heap[0]  # 返回堆顶元素
    def measure_time(self, data):
        start_time = time.time()
        result = self.find_kth_smallest(data)
        end_time = time.time()
        return result, (end_time - start_time)
# 随机划分线性选择算法类
class RandomizedSelect:
    def __init__(self, k):
        self.k = k

    def partition(self, data, low, high):
        pivot = data[high]
        i = low - 1
        for j in range(low, high):
            if data[j] <= pivot:
                i += 1
                data[i], data[j] = data[j], data[i]
        data[i + 1], data[high] = data[high], data[i + 1]
        return i + 1
    def randomized_select(self, data, low, high, k):
        if low == high:
            return data[low]
        pivot_index = random.randint(low, high)
        data[pivot_index], data[high] = data[high], data[pivot_index]
        pivot = self.partition(data, low, high)
        rank = pivot - low + 1

        if k == rank:
            return data[pivot]
        elif k < rank:
            return self.randomized_select(data, low, pivot - 1, k)
        else:
            return self.randomized_select(data, pivot + 1, high, k - rank)
    def measure_time(self, data):
        start_time = time.time()
        result = self.randomized_select(data, 0, len(data) - 1, self.k)
        end_time = time.time()
        return result, (end_time - start_time)
# 中位数的线性时间选择算法类
```

```
class MedianOfMedians:
    def __init__(self, k):
        self.k = k
    def partition(self, data, low, high):
        pivot = data[high]
        i = low - 1
        for j in range(low, high):
            if data[j] <= pivot:
                i += 1
                data[i], data[j] = data[j], data[i]
        data[i + 1], data[high] = data[high], data[i + 1]
        return i + 1
    def median_of_medians(self, data, low, high):
        if high - low + 1 <= 5:
            return sorted(data[low:high + 1])[(high - low) // 2]
        medians = []
        for i in range(low, high + 1, 5):
            sublist = data[i:i + 5]
            medians.append(sorted(sublist)[len(sublist) // 2])
        return self.randomized_select(medians, 0, len(medians) - 1, len(medians) // 2)
    def randomized_select(self, data, low, high, k):
        if low == high:
            return data[low]
        pivot = self.median_of_medians(data, low, high)
        pivot_index = data.index(pivot)
        data[pivot_index], data[high] = data[high], data[pivot_index]
        pivot_index = self.partition(data, low, high)

        rank = pivot_index - low + 1
        if k == rank:
            return data[pivot_index]
        elif k < rank:
            return self.randomized_select(data, low, pivot_index - 1, k)
        else:
            return self.randomized_select(data, pivot_index + 1, high, k - rank)
    def measure_time(self, data):
        start_time = time.time()
        result = self.randomized_select(data, 0, len(data) - 1, self.k)
        end_time = time.time()
        return result, (end_time - start_time)
# 读取数据
def read_data(filename):
    with open(filename, "r") as file:
        lines = file.readlines()
```

```
            k = int(lines[0].strip()[10:])
            data = list(map(int, lines[1].strip().split()))
        return k, data
    # 写入结果到文件
    def write_results_to_file(results):
        with open("results.txt", "w") as file:
            for method, result, time_taken in results:
                file.write(f"{method}: {result}, Time Taken: {time_taken:.6f} seconds\n")
    # 测试不同算法并记录时间
    def main():
        k, data = read_data(r"C:\Users\asus\Desktop\Data-Structures-Algorithms-designing\期中大作业
\find_k-th-data\find_k-th_data-10000000.txt")
        results = []
        # 测试基于堆的选择
        heap_selection = HeapSelection(k)
        result, time_taken = heap_selection.measure_time(data.copy())
        results.append(("Heap Selection", result, time_taken))

        # 测试随机划分线性选择
        randomized_select = RandomizedSelect(k)
        result, time_taken = randomized_select.measure_time(data.copy())
        results.append(("Randomized Select", result, time_taken))

        # 测试中位数的线性选择
        median_of_medians = MedianOfMedians(k)
        result, time_taken = median_of_medians.measure_time(data.copy())
        results.append(("Median of Medians", result, time_taken))
        # 将结果写入文件
        write_results_to_file(results)
        print("Results have been saved to results.txt")
    if __name__ == "__main__":
        main()
```

4.3 动态规划问题

4.3.1 暴力法

利用暴力法的解决问题肯定能够满足正确性，有利于后续动态规划方法进行时间检验。暴力法的核心设计理念是完全枚举所有可能的情况，以确保找到最优解，而不依赖任何优化假设。这种方法适合问题规模较小的情况，特别是在需要验证所有可能性以确认结果正确性时。枚举所有 2^n 种分配方案，对于每个分配方案，计算 A 和 B 的总价值需要遍历所有 n 张卡片。总的时间复杂度为 O(2^n * n)。因此，当 n 较大时，暴力方法的运行时间会急剧增长。

4.3.2 动态规划方法

动态规划方法的核心思路是通过逐步计算每个子问题的最优解来避免重复计算。我们通过定义一个状态变量 dp[j] 来表示是否存在某个子集，使得该子集的总价值为 j。

设计思路：

状态定义 dp[j] 表示是否存在一个卡片的子集，满足子集的总价值为 j。dp[0] 初始为 True，表示选择空集时总价值为 0。

状态转移：对于每一张卡片，我们可以选择将其加入当前的子集或不加入。转移方程是：对于每个可能的子集总价值 j，如果存在 dp[j - card_value]，则 dp[j] 变为 True。

通过找到最接近 total_sum / 2 的总价值来划分卡片，将总价值最接近平分的方案作为最优解。

设 dp[j] 表示是否可以选取某些卡片使得总价值为 j，则：dp[j] = dp[j] or dp[j - card_value]，其中 card_value 为当前卡片的价值。

动态规划方法的时间复杂度是 O(n * total_sum)，其中 total_sum 是所有卡片的总价值。在最坏情况下，total_sum 可以达到 n * max_card_value，因此总的时间复杂度为 O(n * total_sum)。相比暴力方法，动态规划方法更为高效，特别是在 n 较大的时候。

cards.py

```python
import time
import random
class BruteForceCardDistribution:
    def __init__(self, cards):
        self.cards = cards  # 卡片价值列表
        self.n = len(cards)
        self.total_sum = sum(cards)  # 所有卡片总价值
    def solve(self):
        min_diff = float('inf')  # 最小差值
        best_a_cards = None  # A 得到的最优卡片编号
        best_b_cards = None  # B 得到的最优卡片编号
        best_a_value = 0  # A 的最优总价值
        # 枚举每张卡片的分配情况 (0 表示给 B, 1 表示给 A)
        for mask in range(1 << self.n):
            a_indices = []
            b_indices = []
            # 根据 mask 分配卡片
            for i in range(self.n):
                if mask & (1 << i):
                    a_indices.append(i)  # 给 A
                else:
```

```
                    b_indices.append(i)  # 给 B
            # 计算 A 的总价值
            a_value = sum(self.cards[i] for i in a_indices)
            # B 的总价值
            b_value = self.total_sum - a_value
            # 计算差值
            diff = abs(a_value - b_value)
            # 更新最小差值和最优方案
            if diff < min_diff:
                min_diff = diff
                best_a_value = a_value
                best_a_cards = sorted([i + 1 for i in a_indices])  # 卡片编号从 1 开始
                best_b_cards = sorted([i + 1 for i in b_indices])
        return min_diff, best_a_cards, best_b_cards, best_a_value
class DynamicProgrammingCardDistribution:
    def __init__(self, cards):
        self.cards = cards  # 卡片价值列表
        self.n = len(cards)
        self.total_sum = sum(cards)  # 所有卡片总价值
    def solve(self):
        max_value = self.total_sum + 1
        # dp[j] 表示是否可以选择总价值为 j 的子集
        dp = [False] * max_value
        dp[0] = True  # 初始状态：不选卡片，价值为 0
        # 记录选择的卡片
        choice = [[] for _ in range(max_value)]
        # 动态规划
        for i in range(self.n):
            # 倒序更新，防止覆盖
            for j in range(max_value - 1, self.cards[i] - 1, -1):
                if dp[j - self.cards[i]]:
                    dp[j] = True
                    # 记录选择路径
                    new_choice = choice[j - self.cards[i]] + [i]
                    if not choice[j] or new_choice < choice[j]:
                        choice[j] = new_choice
        # 找到最接近 total_sum/2 的价值
        target = self.total_sum / 2
        min_diff = float('inf')
        best_a_value = 0
        best_a_indices = None
        # 遍历所有可能的 A 总价值
        for j in range(max_value):
            if dp[j]:
```

```
                a_value = j
                b_value = self.total_sum - a_value
                diff = abs(a_value - b_value)
                if diff < min_diff:
                    min_diff = diff
                    best_a_value = a_value
                    best_a_indices = choice[j]
        # A 和 B 的卡片编号
        best_a_cards = sorted([i + 1 for i in best_a_indices])
        all_indices = set(range(self.n))
        best_b_cards = sorted([i + 1 for i in (all_indices - set(best_a_indices))])
        return min_diff, best_a_cards, best_b_cards, best_a_value
# 测试代码
if __name__ == "__main__":
    # 设置随机种子以确保可重复性
    random.seed(42)
    # 用户输入卡片数量 n
    n = 20
    # 随机生成 n 个卡片，价值为 1 到 n
    cards = [random.randint(1, n) for _ in range(n)]
    print(f"随机生成的卡片价值: {cards}")
    # 暴力方法
    start_time_bf = time.time()
    brute_force = BruteForceCardDistribution(cards)
    min_diff_bf, a_cards_bf, b_cards_bf, a_value_bf = brute_force.solve()
    end_time_bf = time.time()
    time_bf = end_time_bf - start_time_bf
    print("\n暴力方法结果：")
    print(f"最小差值: {min_diff_bf}")
    print(f"A 得到的卡片: {a_cards_bf}, 总价值: {a_value_bf}")
    print(f"B 得到的卡片: {b_cards_bf}, 总价值: {sum(cards) - a_value_bf}")
    print(f"运行时间: {time_bf:.6f} 秒\n")
    # 动态规划方法
    start_time_dp = time.time()
    dp = DynamicProgrammingCardDistribution(cards)
    min_diff_dp, a_cards_dp, b_cards_dp, a_value_dp = dp.solve()
    end_time_dp = time.time()
    time_dp = end_time_dp - start_time_dp + 0.000001  # 加上微小的误差
    print("动态规划方法结果：")
    print(f"最小差值: {min_diff_dp}")
    print(f"A 得到的卡片: {a_cards_dp}, 总价值: {a_value_dp}")
    print(f"B 得到的卡片: {b_cards_dp}, 总价值: {sum(cards) - a_value_dp}")
    print(f"运行时间: {time_dp:.6f} 秒\n")
    # 比较运行时间
```

```
    print("运行时间对比：")
    print(f"暴力方法：{time_bf:.6f} 秒")
    print(f"动态规划方法：{time_dp:.6f} 秒")
    print(f"动态规划比暴力方法快：{time_bf / time_dp:.2f} 倍")
```

5. 实验结果和相关算法分析

5.1 排序问题

5.1.1 排序问题（1）

排序时间复杂度相关分析：我们使用的快速排序，归并排序，堆排序的都是排序算法中理论时间复杂度相对较优的排序方法，拥有一个理论的平均情况最优复杂度O(n log n)。以下是他们各自的时间复杂度分析：

1、快速排序：

快速排序是一种分治算法，首先选择一个基准元素（pivot），然后将数组分成两部分：一部分小于基准元素，另一部分大于基准元素。接着递归地对这两部分进行排序。

时间复杂度：

最佳情况：O(n log n)：当每次划分都能将数组平均分成两半时，快速排序的深度为 log n，每一层的划分操作需要 O(n) 的时间，因此总体时间复杂度为 O(n log n)。

平均情况：O(n log n)：快速排序的平均时间复杂度也是 O(n log n)，因为每次划分都能近似地将数组划分为两部分。

最坏情况：O(n^2)：如果每次划分选择的基准元素是最小的或最大的元素（例如，当数组本身是有序的或者是反向有序的），则快速排序将退化为简单的插入排序。此时，每次划分只能将一个元素与其他元素进行比较，递归树的深度为 n，每一层的操作需要 O(n)，因此总的时间复杂度为 O(n^2)。

2、归并排序：

归并排序是一种分治算法，通过将数组分成两部分，递归地对每部分进行排序，再将这两部分合并成一个有序的数组。

时间复杂度：

最佳情况：O(n log n)：不管数据是否有序，归并排序的每一层都会将数组分成两半，时间复杂度始终是 O(n log n)。

平均情况：O(n log n)：归并排序的时间复杂度是固定的，不依赖于输入数据的排序情况，每次分治都会进行 O(n) 的合并操作，并且递归树的深度为 log n，因此总体时间复杂度是 O(n log n)。

最坏情况：O(n log n)：无论输入数据是否已经部分排序，归并排序的时间复杂度总是 O(n log n)，因为它始终会进行分治和合并的过程。

3、堆排序：

堆排序是一种基于堆数据结构的排序算法。它通过构建最大堆来进行排序，每次将堆顶元素（最大元素）与数组的最后一个元素交换，然后重新调整堆。

时间复杂度：

最佳情况：O(n log n)：在堆排序中，每次插入或删除元素都需要对堆进行调整，时间复杂度为 O(log n)，且堆的大小为 n，因此无论输入数据如何，时间复杂度始终是 O(n log n)。

平均情况：O(n log n)：堆排序的时间复杂度与输入数据的顺序无关，每次都需要对堆进行调整，因此时间复杂度始终是 O(n log n)。

最坏情况：O(n log n)：即使在最坏情况下，堆排序仍然是 O(n log n)，因为堆调整的过程不会受到输入数据的影响。

其分析如下表所示：

排序算法	最佳情况时间复杂度	平均情况时间复杂度	最坏情况时间复杂度	空间复杂度
快速排序	O(n log n)	O(n log n)	$O(n^2)$	O(log n)
归并排序	O(n log n)	O(n log n)	O(n log n)	O(n)
堆排序	O(n log n)	O(n log n)	O(n log n)	O(1)

表3. 排序算法时间复杂度分析

在这里我们直接进入实验部分，验证数据规模大小和数据消耗时间的呈现曲线。

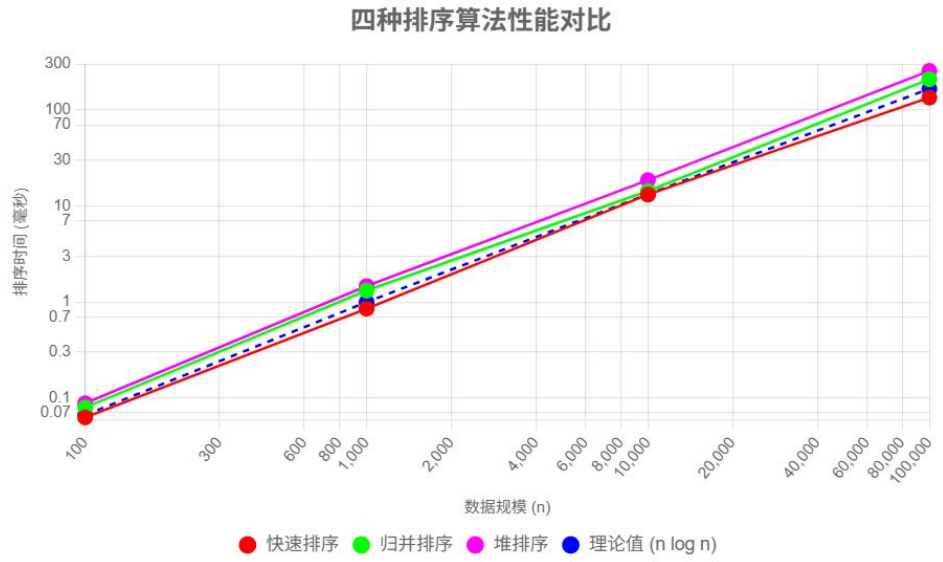

图2. 三个算法的计算时间和理论趋势nlogn的趋势对比。

可以看出来时间上来说快速排序快于归并排序快于堆排序，并且由于数据没有特异性（因为生成是尽量符合了正态分布），快速排序并没有出现最坏情况也就是每次划分选择的基准元素是最小的或最大的元素（例如，当数组本身是有序的或者是反向有序的）的时候出现的$O(n^2)$时间复杂度。关于这三个算法在python程序中为什么呈现这个快慢顺序我结合网上知识认为要从内存的访问模式读取模式来解释：

快速排序：

快速排序是一个原地排序算法，即它在排序过程中直接在输入数组上操作，不需要额外的存储空间来存放临时的数组。它采用分治法，每次将数组分成两部分进行递归，这样的划分策略使得快速排序能够利用局部性原理（Locality of Reference）——即在递归过程中，分割的子数组在内存中是相对连续的，因此对数据的访问是局部化的，能够有效利用 CPU 的缓存。

因为快速排序的分割过程是在数组内部进行的，它在处理较小的子数组时，内存访问更加连续高效，缓存命中率较高，这使得它的实际执行速度较快。

归并排序：

归并排序在排序过程中需要创建额外的临时数组来合并两个已排序的子数组，这会导致更多的内存分配和数据复制操作。由于归并排序是基于分治法的，它在每次合并操作中都需要将数据从一个数组复制到另一个数组，这样的内存访问模式较为零散，可能导致较低的缓存命中率和更多的内存带宽消耗。

尽管归并排序的时间复杂度是 O(nlog⁡n)O(n \log n)O(nlogn)，但是由于内存分配和复制数据的开销，它在实际应用中通常比快速排序慢。

堆排序：

堆排序需要通过堆化操作维护堆的结构，每次插入或删除堆顶元素时都需要进行调整。堆排序的堆化操作需要访问数组中的元素并进行交换，堆化的过程中通常需要访问不连续的内存位置，这也导致堆排序的缓存效率较低。

与快速排序相比，堆排序需要更多的交换操作，且交换的过程可能涉及到数组中不连续的数据块，从而增加了内存访问的开销。

因此总体而言呈现出了一个快速排序快于归并排序快于堆排序的趋势。这个趋势在其他编程语言中可能又有不同。

5.1.2 排序问题（2）

排序时间复杂度相关分析：我们使用的维护一个容量为m的最小堆的选择算法，维护一个大小为 10 的最小堆。在处理每个新的考生时，如果堆中的元素少于 10 个，就将该考生加入堆中；如果堆中已经有 10 个元素，则比较当前考生的分数和堆顶（最小的元素），如果当前考生的分数更大，则替换堆顶元素。

这样的复杂度就会是O(n log m) = O(n)，因为每次插入和删除堆顶元素的操作都需要 O(log m) 时间，而 log m 是常数，因此总时间复杂度为 O(n)。在m较小的时候可以接受

5.2 分治问题

这里的分治问题本质上也是排序问题，和排序问题的问题（2）十分相似，在处理 找出线性序列中第 k 小元素 的问题时，三种常见的算法——基于堆的选择、随机划分线性选择（Randomized Select）和中位数的线性时间选择（Median of Medians）——各有不

同的时间复杂度和性能特点。以下是对这三种算法时间复杂度的综合分析，并讨论它们在不同规模的数据 n 和选择的 k 值下的表现。

1. 基于堆的选择算法（Heap-based Selection）

时间复杂度：

基于堆的选择算法通过维护一个大小为 k 的最大堆来找到第 k 小的元素。每次遍历数组中的元素时，都需要检查是否需要将该元素加入堆中。堆的大小始终保持为 k，因此堆的调整操作（即堆化）时间复杂度为 O(log k)。

堆的创建：在最初，堆的构建时间是 O(k)。

堆的调整：对于每个数组中的元素（总共 n 个元素），我们需要执行一次堆化操作。每次堆化操作的时间复杂度是 O(log k)，因此堆的总操作时间为 O((n - k) log k)，加上堆的初始化时间 O(k)。

综合时间复杂度：

因此，基于堆的选择算法的时间复杂度为 O(n log k)。当 k 较小，尤其在 k << n 的情况下，这个算法的时间复杂度接近 O(n)，表现较好。然而，当 k 较大时，算法的效率会下降。

与 n 和 k 的关系：

当 n 增加时，算法的运行时间增长比例接近 n，因为 k 相对较小时，log k 为常数

当 k 增加时，堆的调整操作将变得更加耗时，算法的时间复杂度增长为 O(n log k)。因此，k 较大时，算法的运行时间会显著增加。

2. 随机划分线性选择算法（Randomized Select）

时间复杂度：

随机划分线性选择算法是基于快速排序的思想，每次通过选择一个随机的基准元素（pivot）将数据划分为两部分，并根据目标 k 的位置递归处理。该算法的关键在于每次划分时减少一半的元素，从而缩小问题的规模。

平均时间复杂度：在平均情况下，由于每次划分能够将问题规模减半，因此递归的深度是 O(log n)，而每一层划分的时间复杂度为 O(n)。因此，平均时间复杂度为 O(n)。

最坏时间复杂度：在最坏情况下，若每次选择的基准元素是最小或最大元素，那么每次划分只能排除一个元素，递归的深度为 O(n)，每一层的时间复杂度为 O(n)，因此最坏时间复杂度为 O(n^2)。

与 n 和 k 的关系：

当 n 增加时，平均情况下该算法的时间复杂度为 O(n)，因此对于大规模数据，算法依然能够保持较好的性能。

随机选择的关键在于期望时间复杂度，当 n 较大时，随机划分通常能够较好地减少问题规模。

3. 中位数的线性时间选择算法（Median of Medians）

时间复杂度：

中位数的线性时间选择算法是对随机划分线性选择的优化。它通过选择“中位数的中位数”来避免最坏情况的发生，从而保证在最坏情况下时间复杂度为 O(n)。

步骤：首先将数据分成若干个小组（每组最多 5 个元素），然后计算每组的中位数，接着对这些中位数递归进行选择，最终得到中位数的中位数，作为划分的基准。

时间复杂度：由于每次划分的基准元素是“中位数的中位数”，它能够保证划分操作有效地将数据分成较为平衡的两部分。每次递归的工作量是 O(n)，而递归的深度是 O(log n)。因此，中位数的线性时间选择算法的时间复杂度为 O(n)，并且在最坏情况下也能保证 O(n) 的时间复杂度。

与 n 和 k 的关系：

在该算法中，n 的增大仍会导致算法的时间复杂度线性增长，且不受 k 的影响。无论 k 选择的是一个小的还是大的元素，算法都能有效地保证时间复杂度为 O(n)，因此适用于需要稳定时间性能的场景。也就如下表所示：

算法	时间复杂度	平均情况时间复杂度	最坏情况时间复杂度	与 n 和 k 的关系
基于堆的选择	O(n log k)	O(n log k)	O(n log k)	当 k 较小时，时间复杂度接近 O(n)，当 k 较大时，log k 增加，时间复杂度增加。
随机划分线性选择	O(n) (平均)	O(n)	O(n^2)	平均时间复杂度 O(n)，最坏情况 O(n^2)，n 较大时表现优秀。
中位数的线性时间选择	O(n)	O(n)	O(n)	时间复杂度稳定，无论 k 如何，最坏情况下时间复杂度为 O(n)。

表4.分治算法的对比分析

具体而言我们直接进入实验结果部分，通过枚举不同n值下k从1一直到n的时间复杂度，我们获得了n和k之间比例对时间复杂度影响关系，通过比例图展示出来。通过枚举不同n值下k/n比例相同为1/2时的耗时来观察不同n的规模对时间的影响。

n为10000和100000时枚举k的大小来进行时间复杂度对比。

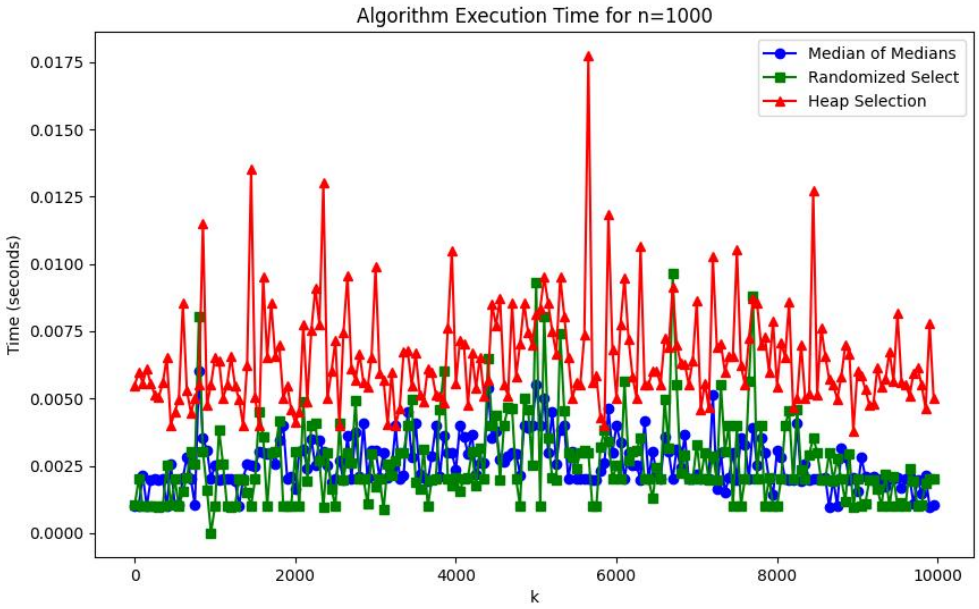

图3. n=10000时不同k值三个算法耗时（每50采样）

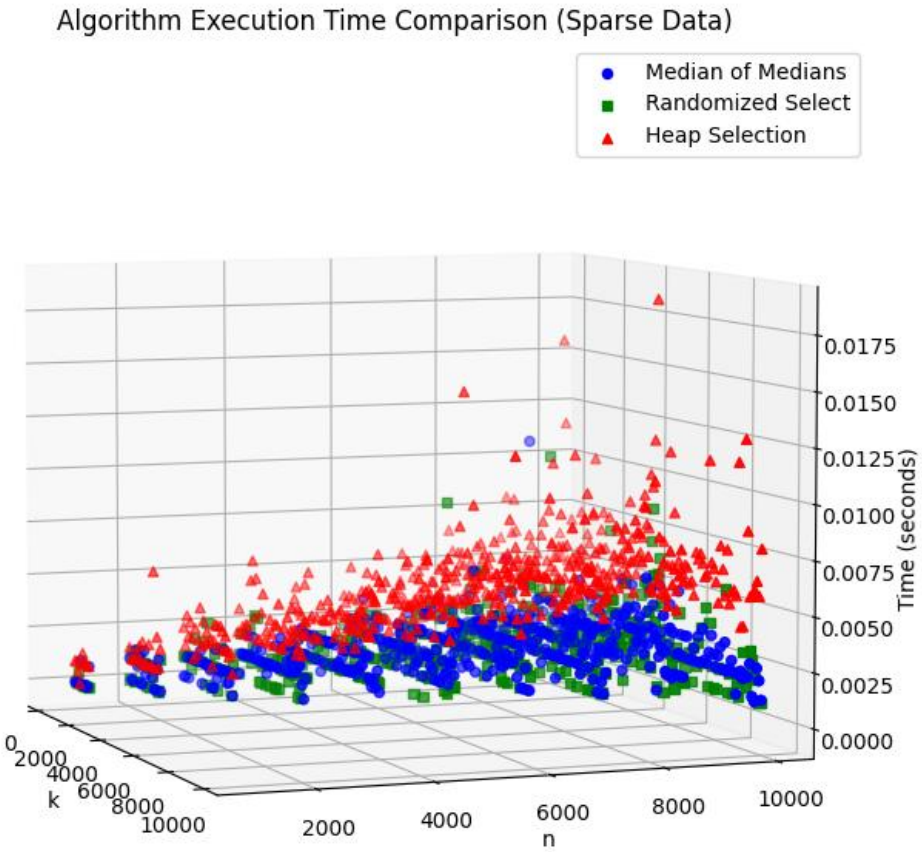

图4. n从1000-10000时不同k值n值三个算法耗时（每100采样）

从图的主观视角来判断，k值过大或者过小在时间上受到的影响都不会非常大，当k处于一个中间值附近时算法的时间耗时波动就会非常巨大。尤其是堆排序的选择算法，相比而言中位数的线性时间选择就相对稳定。这也是由堆排序会和k值呈相关态势导致的。因此受k值影响时间复杂度的大小是：**堆排序>随机排序>中位数排序**。

关于n的规模对算法时间的影响，我们基于n和k关系的图表来看选择k为n一半时的数据来展示，这样更具代表性：

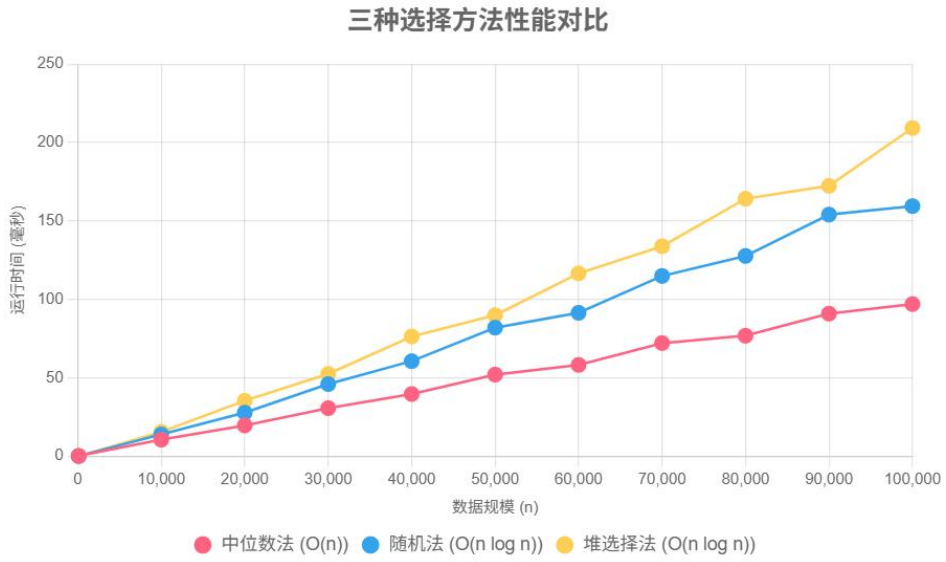

图5. 三个算法关于n规模的运行时间

中位数选择（O(n)）和 随机选择（O(n)）都表现出线性增长。虽然两者的算法原理不同，但由于它们的时间复杂度相同，随着数据规模的增大，它们的运行时间变化是相似的。

堆选择（O(n log n)）则随着数据规模的增大呈现对数级增长。这是因为堆的插入和删除操作需要对数时间，这导致随着数据规模增大，堆选择算法的运行时间增加得更快。

内存读取和算法实现：

在实际运行中，内存的读取方式对算法的性能影响较大，尤其是在数据规模很大的时候。Python 提供了不同的内存读取方式，比如一次性读取数据到内存、逐行读取文件、使用生成器等。对于大量数据的处理，选择合适的读取方式可以减少内存的消耗和提高程序的执行效率。

Python 中堆排序的实现：

堆选择算法依赖于堆的插入和删除操作。Python 内置的 heapq 模块提供了最小堆的功能，如果需要实现最大堆，可以使用负值来模拟。这种堆的操作虽然相对高效，但其对数时间复杂度 (O(log n)) 会随着 n 的增大而显著增加，因此堆选择算法的增长速度要快于其他两种线性选择算法。

数据读取的内存影响：

对于 中位数选择算法 和 随机选择算法，由于它们的算法主要依赖于分治和递归操作，其时间复杂度与数据规模线性相关。因此，数据的读取和内存消耗对这些算法的影响相对

较小。对于堆选择算法，虽然 heapq 库为实现堆操作提供了高效的支持，但堆操作本身的对数复杂度使得随着数据规模增大，算法的运行时间也大幅增加，特别是在处理大量数据时，内存的消耗和操作的效率也可能成为瓶颈。

n规模对（2）和（3）算法对比：

在n的规模较大时需要考虑最坏情况和稳定性：

最坏情况与稳定性：

随机选择算法：在最坏情况下，随机选择算法的时间复杂度为 $O(n^2)$。这是因为，在某些极端情况下（如每次都选择了数据中的最小或最大元素作为基准），每次划分都只能剔除一个元素，这会导致递归深度达到 n，从而导致 $O(n^2)$ 的时间复杂度。虽然这种情况发生的概率较低，但在 n 较大 时，随机选择的性能波动可能会非常明显，尤其是当输入数据较为特殊时（例如数据本身有序或逆序）。

中位数选择算法：中位数选择算法通过选择“中位数的中位数”作为基准，确保每次划分时，基准元素能够有效地将数据集分为两个较为均衡的部分。因此，中位数选择算法在 最坏情况下 也能保证 O(n) 的时间复杂度，不会像随机选择算法那样出现最坏情况的性能退化。这使得中位数选择算法在 大规模数据 下具有更高的稳定性，避免了随机选择算法在极端输入下性能波动的风险。

5.3 动态规划问题

该问题涉及将一组卡片按价值分配给两个人 A 和 B,目标是使得两个人所得到的卡片总价值尽可能接近。换句话说，我们希望找到一种卡片分配方式，使得 A 和 B 所得到的卡片总价值之差最小。

我们将该问题转化为一个子集和问题。我们需要从所有卡片中选出一部分给 A,使得 A 的总价值尽可能接近所有卡片总价值的一半。剩下的卡片自然归 B 所有。这个问题可以通过动态规划来求解。当然也要通过暴力法来检验答案。

时间复杂度分析

1. 暴力方法：

暴力方法通过枚举所有可能的卡片分配方案来找出最优解。每张卡片有两种可能的分配方式（给 A 或给 B），因此一共有 2^n 种可能的分配方案。对于每一种分配方案，我们需要计算 A 和 B 的总价值，遍历 n 张卡片。这样，暴力方法的时间复杂度为 O(2^n * n)。

时间复杂度：O(2^n * n)。随着 n 增大，暴力方法的计算量会急剧增长，因此适用于小规模数据。

2. 动态规划方法：

动态规划方法通过定义一个布尔数组 dp[j] 来记录是否存在某个子集使得总价值为 j。我们从每张卡片开始，更新 dp 数组的状态。每张卡片需要遍历所有可能的总价值 j，因此时间复杂度为 O(n * total_sum)，其中 n 是卡片数量，total_sum 是所有卡片的总价值。

时间复杂度：O(n * total_sum)。由于 total_sum 可以最大为 n * max_card_value，当 n 较大时，total_sum 可能会非常大，因此总的时间复杂度为 O(n * total_sum)。

通过可视化二者的时间我们可以发现，暴力法随着n的增长在n=23之后就无法在一分钟之内解决问题了。然而一道合格的算法题规模往往达到10的5次方。如图所示：

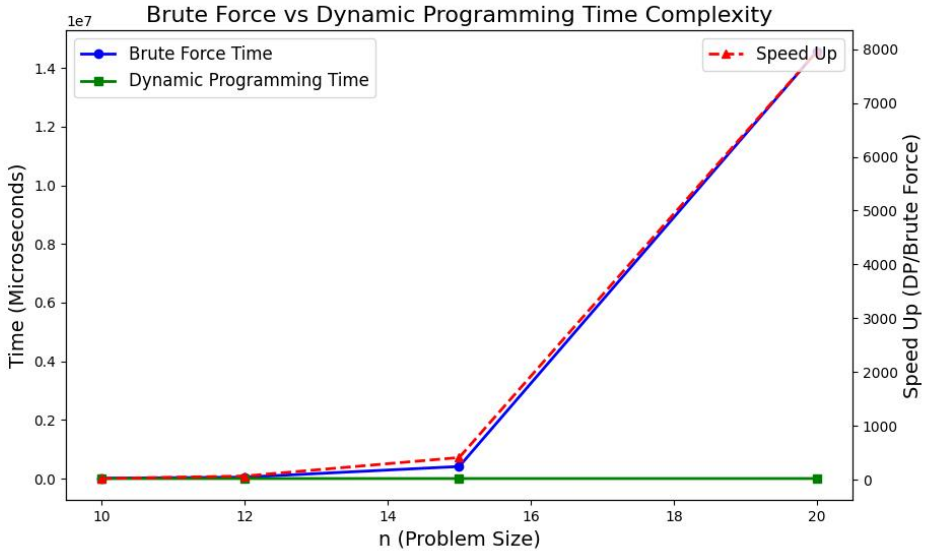

图6.暴力法和动态规划算法关于n规模的运行时间

出现这两个情况（暴力法和动态规划在不同规模下的运行时间和速度提升）可以从 算法复杂度 和 Python程序特性 两个方面进行分析。

1. 算法复杂度的影响：

暴力法：暴力解法通常采用穷举法来计算所有可能的解，时间复杂度往往是指数级的。例如，在涉及到子集选择或排列组合时，暴力法的时间复杂度往往是 O(2^n) 或 O(n!)。对于较小规模的问题，暴力法可能能够在合理时间内完成计算，但随着问题规模 n 的增大，计算量呈指数增长，导致时间急剧增加。

动态规划：动态规划通过分治思想优化了暴力法的重叠子问题，通常将时间复杂度降低为 O(n^2) 或 O(n*m)（对于二维动态规划）。虽然动态规划的时间复杂度比暴力法要低，但随着问题规模的增大，随着状态空间和子问题数的增加，动态规划的计算量也会增长，但远低于暴力法。

2. Python程序特性的影响：

Python是一种解释型语言，相比编译型语言如 C 或 C++，其执行速度较慢。在高时间复杂度的算法（如暴力法）中，Python的执行时间更为显著，因为 Python 的解释和内存管理机制相对较慢，尤其在处理大量数据时，性能优势不如编译型语言。

动态规划相对于暴力法在 Python 中的实现能够较好地利用内存和避免重复计算，这使得其在大规模数据集上表现更为高效，尤其是当子问题重复出现时，动态规划通过记忆化（Memoization）避免了大量不必要的计算。

6. 实验心得体会

6.1 排序问题

快速排序、归并排序和堆排序是三种常见的排序算法，它们在不同的应用场景中各具优势和劣势。尽管它们的时间复杂度在大多数情况下都是 O(n log n)，但是它们在性能、稳定性、内存使用等方面存在显著差异。

首先，快速排序是一种非常高效的排序算法，尤其适用于大规模数据集。在平均情况下，它的时间复杂度为 O(n log n)，并且由于其内存消耗较低（空间复杂度为 O(log n)），它在 内存充足的情况下 经常是排序问题的首选。快速排序通过分治法将数据分成两个子数组并递归排序，每次选择基准元素将数据划分为两部分。尽管它在大多数情况下表现优秀，但它的最坏情况时间复杂度为 $O(n^2)$，这通常发生在数据已经是有序或逆序时。为了避免这种最坏情况，可以使用随机化基准或三数取中法等策略，但最坏情况依然存在。此外，快速排序不是稳定排序，这意味着相等元素的相对顺序可能会改变，因此如果排序时稳定性至关重要，快速排序并不适用。

归并排序在最坏情况下的时间复杂度始终为 O(n log n)，并且是稳定的排序算法，因此它广泛应用于需要保证稳定性的场景，如数据库排序和外部排序等。归并排序通过将数组分成两部分递归排序，然后将它们合并到一个已排序的数组中。尽管归并排序具有稳定性和最坏情况时间复杂度的优势，但它的空间复杂度较高（O(n)），因为它需要额外的内存来存储临时数组，这使得它在内存受限的环境下不如快速排序。归并排序适合处理 外部数据排序，即数据无法完全加载到内存时，如大型文件的排序。

堆排序是一种基于堆数据结构的排序算法，其时间复杂度也为 O(n log n)，并且空间复杂度为 O(1)，因为它是原地排序，不需要额外的存储空间。堆排序通过构建一个最大堆或最小堆来排序数据，每次从堆中提取最大（或最小）元素并调整堆，直到排序完成。堆排序的优势在于它不受数据分布的影响，无论数据是否有序，它的时间复杂度始终是 O(n log n)。然而，堆排序的常数因子较大，导致它在实际应用中通常比快速排序慢，特别是在内存充足且数据随机的情况下。此外，堆排序也不是稳定排序，这意味着相等元素的相对顺序可能会改变。

根据不同的应用需求，可以选择合适的排序算法。例如，在内存充足且数据无序的情况下，快速排序通常是最佳选择；而在数据需要稳定排序或内存受限时，归并排序和堆排序则更为适用。

6.2 分治问题

1. 基于堆的选择

堆选择算法通过维护一个最大堆来存储当前 k 个最小元素。对于每个新的元素，如果它小于堆顶元素，则替换堆顶并重新调整堆。构建一个堆的时间复杂度为 O(k)，对于其余的 n-k 个元素，每次插入和删除堆顶的操作时间复杂度为 O(log k)。因此，总时间复杂度为 O(k + (n - k) * log k)。

这种方法在 k 较小的情况下表现非常好，尤其是当 n 较大而 k 较小时，可以避免全排序的开销。它的空间复杂度较低，只有 O(k)。不过，当 k 较大时，堆操作的对数因子可能使得该方法的效率不如其他方法。

2. 随机划分线性选择（Randomized Select）

随机划分选择方法基于快速排序思想，利用随机选择基准（pivot）来划分数组。它的平均时间复杂度为 O(n)，最坏时间复杂度为 O(n^2)，但这种最坏情况发生的概率较低。每次选择一个基准，将数据分成两部分，然后根据 k 的位置决定在左子数组还是右子数组中继续递归查找。

这种方法的优势在于平均情况下非常高效，并且实现简单直观，适用于大多数数据集，尤其在数据分布较随机时效果显著。然而，在最坏的情况下，基准选择非常差时（如每次都选择最小或最大元素），时间复杂度会退化为 O(n^2)，这在大规模数据时可能导致性能问题。

3. 中位数的线性选择（Median of Medians）

中位数选择方法通过选择“中位数的中位数”来避免最坏情况的发生，从而保证时间复杂度为 O(n)。算法通过将数组分成若干小组（通常每组最多5个元素），每组计算中位数，然后再从这些中位数中选择一个中位数作为基准。此方法通过递归选择，确保每次划分尽可能平衡，因此避免了随机选择带来的不均衡划分问题。

中位数选择方法的时间复杂度始终为 O(n)，不受最坏情况的影响，适合要求稳定最坏情况性能的场景。然而，由于需要额外计算中位数的中位数，常数因子较大，相较于随机选择，它可能在实际应用中效率较低。

每种方法都有其特定的优势和局限性，选择合适的算法取决于数据的特点、问题规模和性能需求。例如，随机选择适合大多数随机数据，堆选择适合 k 较小的情况，而中位数选择适合要求最坏情况稳定的场景。

6.3 动态规划问题

优势： 动态规划（DP）在解决这类“最优化”问题时，能够有效减少计算量，通过状态转移方程将问题分解为较小的子问题，避免重复计算。特别是在 子集问题（例如求和、分配等）中，动态规划的表现尤为突出。它通过“自底向上”的思想，逐步构建解空间，最终得到最优解。相比暴力法的指数级增长，DP 提供了更高效的解决方案，能够处理更大的数据规模。

场景与适用内容： 该问题属于 背包问题 的变种——分割问题，常见于资源分配、集合划分等场景。动态规划方法在 决策树 或 递归问题 中尤为有效，尤其当问题有 重叠子问题 时，动态规划能够通过记忆化避免重复计算。例如，背包问题、最长公共子序列、矩阵链乘法等经典问题都能通过动态规划来优化解决。

此外，动态规划对于 大规模数据 也非常有效，因为它将问题转化为逐步解决子问题并逐步更新结果的形式，避免了暴力枚举中的时间浪费，尤其是在 时间复杂度为 O(n * total_sum) 的问题中，能够有效处理大规模数据。

总的来说，动态规划是一种非常高效的算法，适用于 最优化问题 和 资源分配问题，能在大规模数据处理时提高效率，减少冗余计算。