**2022-2023学年第二学期《数据结构与算法设计2》 期中大作业题目和要求**

**要求**

（1）本作业包含三个题目，第1题是必做题， 第2和3题中可选择其中1道题完成；

（2）以电子文档形式提交，需要提交的内容包括报告和源码，报告主要内容包括题目、算法思想、选择编程语言及环境、典型的测试用例、系统的输入输出运行结果截图、算法分析、其他说明（包括但不限于求解过程中遇到的问题及如何解决等）；

（3）输入输出测试用例可以采用文件的形式；

（4）提交格式：所有提交文件压缩在一个压缩文件中，文件名格式：学号-姓名-期中大作业.xxx,；

1. **排序算法综合应用题**

**问题定义：**

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 |

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）

1. 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. **解答:**

**注：**为了方便生成测试数据与后续的结果画图处理，本题我统一采用了Python 3.11的环境，通过Python语言编写实现。

本题考查的是排序算法的应用，我们通过一个排序学生三科成绩的方式来体会排序算法的特性，适用范围，来帮助我们对排序算法有更多的了解，首先按照题目要求我们来生成测试数据，题目要求学生成绩尽可能满足正态分布，所以在生成随机数据的时候我们调用了Python中的服从正太分布的随机数，我们选取每科分数均值为75，方差为25进行随机数据的生成，这里我们每个学生通过其id来唯一区别，为了方便起见，所有学生id从1开始依次递增，在第一问中我们分别选取n为50、500、5000、50000四种不同规模的n生成的测试数据分别导入score1.txt、score2.txt、score3.txt、score4.txt中，其中文件内容为每行5个数，分别代表这个学生的id、语文成绩、数学成绩、英语成绩、总成绩。

生成测试数据的代码如下：

1. **import** numpy as np
3. # 生成随机数
4. n = 50000  # 考生数量
5. mu = 75  # 期望
6. sigma = 25  # 标准差
7. data = np.random.normal(mu, sigma, size=(n, 3))
9. # 转换为整数
10. data = np.round(data).astype(int)
12. # 保存到文件
13. id=0
14. with open('scores4.txt', 'w') as f:
15. **for** row **in** data:
16. id=id+1
17. f.write('{} {} {} {} {}\n'.format(id,row[0], row[1], row[2],row[0]+row[1]+row[2]))

其次我们再来考虑排序算法的选取方面，这里我打算选择归并排序，归并排序是利用了递归分治的思想进行排序，归并排序是一种稳定且快速的排序算法，在面对任何数据的时候都能维持其O(nlogn)的优秀时间复杂度，而且本题相对于一般的排序算法只是做了简单的改变变为多关键字排序了，所以归并排序的代码也不需要进行很多额外的改动，同时本题要求计算并统计每种规模数据对应的具体排序时间，这里我借用了Python中的timeit库，多次运行同一段代码取平均值，最后借用Python的画图功能实现了结果的可视化，同时我也把排序完成的结果按照与输入数据相同的个数输出到了文件里。具体Python代码如下：

1. **import** timeit
3. # 归并排序函数
4. **def** Less\_Than(a,b):
5. **if** a[4]!=b[4]: **return** a[4]<b[4]
6. **elif** a[1]!=b[1] :**return** a[1]<b[1]
7. **elif** a[2]!=b[2] :**return** a[2]<b[2]
8. **return** a[3]<b[3]
10. **def** merge\_sort(arr):
11. # 归并函数，用于合并两个有序数组
12. **def** merge(left, right):
13. result = []
14. i, j = 0, 0
15. **while** i < len(left) **and** j < len(right):
16. **if** Less\_Than(right[j],left[i]):
17. result.append(left[i])
18. i += 1
19. **else**:
20. result.append(right[j])
21. j += 1
22. result += left[i:]
23. result += right[j:]
24. **return** result
26. # 递归函数
27. **def** merge\_sort\_recursion(arr):
28. **if** len(arr) <= 1:
29. **return** arr
30. mid = len(arr) // 2
31. left = arr[:mid]
32. right = arr[mid:]
33. left = merge\_sort\_recursion(left)
34. right = merge\_sort\_recursion(right)
35. **return** merge(left, right)
37. **return** merge\_sort\_recursion(arr)

40. # 读取成绩
41. stu\_info=[]
42. #利用一个二维列表来存储所有的学生信息，
43. #每个一维列表中分别有五个元素表示id,chines\_score,math\_score,English\_score,total\_score
44. with open('scores4.txt', 'r') as f:
45. line = f.readline()
46. **while** line:
47. info=[]
48. **try**:
49. num\_list = line.strip().split()
50. **for** num\_str **in** num\_list:
51. num = int(num\_str)
52. info.append(num)
53. **except** ValueError:
54. **print**(f"Invalid input: {line}")
55. stu\_info.append(info)
56. line = f.readline()

59. # 记录排序所用时间
61. # 排序
62. sorted\_info = merge\_sort(stu\_info)
64. with open('scores4\_res.txt', 'w') as f:
65. **for** info **in** sorted\_info:
66. f.write('{} {} {} {} {}\n'.format(info[0],info[1], info[2], info[3],info[4]))
68. t = timeit.Timer(**lambda**: merge\_sort(stu\_info))
69. times = t.repeat(repeat=10, number=1)
70. mean\_time = sum(times) / len(times)
71. **print**("Mean execution time:", mean\_time)

画图代码如下：

1. **import** matplotlib.pyplot as plt
2. **import** numpy as np
4. # 假设已经获取到了n和时间数据
5. n\_values = [ 50, 500, 5000, 50000]
6. time\_values = [0.06606, 1.8101, 10.329, 137.58]
8. # 绘制折线图
9. plt.figure()
10. plt.plot(n\_values, time\_values)
11. plt.title('Sorting Time vs. n')
12. plt.xlabel('n')
13. plt.ylabel('Time')
14. plt.show()

为了保证求出的执行时间可信，我使用了python库中的timeit重复执行一段代码多次求平均值来得出最终不同的n对应的执行时间，具体表示如下：

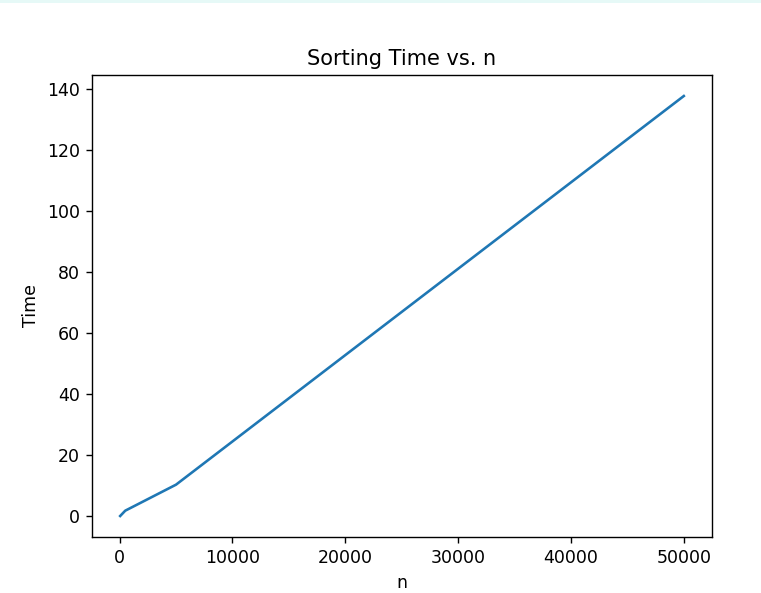








折线图如下：



我们知道归并排序时间复杂度是O(nlogn)，这几个测试数据中n属于较小规模，logn常数很小，所以图像上近似表现为线性。

（2）因为m<<n,所以我们考虑使用快速排序的算法来解决本问题，快速排序对于这种数据量很大的排序问题往往可以实现很好的排序时间复杂度，我们利用稍加改进的快速排序算法即可快速得到前m个学生的成绩，在数据上我们选择m=100,n=1000000进行测试：

代码如下：

1. **import** timeit

4. m=100
5. # 归并排序函数
6. **def** Less\_Than(a,b):
7. **if** a[4]!=b[4]: **return** a[4]<b[4]
8. **elif** a[1]!=b[1] :**return** a[1]<b[1]
9. **elif** a[2]!=b[2] :**return** a[2]<b[2]
10. **return** a[3]<b[3]

13. **def** quick\_sort(scores, left, right, m):
14. **if** left >= right:
15. **return**
16. pivot = scores[left]
17. i = left
18. j = right
19. **while** i < j:
20. **while** i < j **and** Less\_Than(scores[j],pivot):
21. j -= 1
22. scores[i] = scores[j]
24. **while** i < j **and** Less\_Than(pivot,scores[i]):
25. i += 1
26. scores[j] = scores[i]
28. scores[i] = pivot
30. **if** i+1 == m:  # 前m个数已经有序
31. **return**
32. **elif** i+1 < m:
33. quick\_sort(scores,  i+1, right, m)
34. **else**:
35. quick\_sort(scores, left, i-1, m)
36. # 读取成绩
37. stu\_info=[]
38. #利用一个二维列表来存储所有的学生信息，
39. #每个一维列表中分别有五个元素表示id,chines\_score,math\_score,English\_score,total\_score
40. with open('scores\_big.txt', 'r') as f:
41. line = f.readline()
42. **while** line:
43. info=[]
44. **try**:
45. num\_list = line.strip().split()
46. **for** num\_str **in** num\_list:
47. num = int(num\_str)
48. info.append(num)
49. **except** ValueError:
50. **print**(f"Invalid input: {line}")
51. stu\_info.append(info)
52. line = f.readline()

55. # 记录排序所用时间
57. # 排序
58. t = timeit.Timer(**lambda**: quick\_sort(stu\_info,0,len(stu\_info)-1,m))
59. sorted\_info = stu\_info[:m]
60. kase=0
61. with open('scores\_big\_res.txt', 'w') as f:
62. **for** info **in** sorted\_info:
63. kase=kase+1
64. f.write('{} {} {} {} {}\n'.format(info[0],info[1], info[2], info[3],info[4]))
65. **if** kase==m:
66. **break**

69. times = t.repeat(repeat=1, number=1)
70. mean\_time = sum(times) / len(times)
71. **print**("Score\_big:Mean execution time:", mean\_time)
72. **警卫巡逻问题（未选择）**

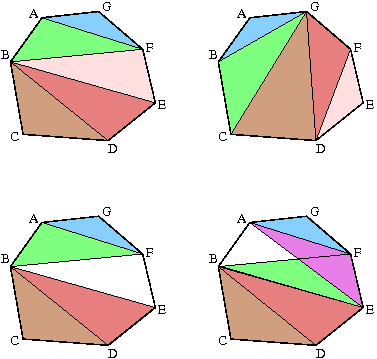
**问题定义**：博物馆是具有n个顶点的凸多边形的形状。博物馆由警卫队通过巡逻来确保馆内物品的安全。博物馆的安全保卫工作遵循以下规则，以尽可能时间经济的方式确保最大的安全性：

* 1. 警卫队中每个警卫巡逻都沿着一个三角形的路径；该三角形的每个顶点都必须是多边形的顶点。
  2. 警卫可以观察其巡逻路径三角形内的所有点，并且只能观察到这些点；我们说这些点由该警卫守护并覆盖。
  3. 博物馆内的每一处都必须由警卫人员守护。
  4. 任何两个警卫巡逻所在的三角形在其内部不重叠，但它们可能具有相同的边。

在这些限制条件下，警卫的成本是警卫巡逻所沿路径的三角形的周长。

我们的目标是找到一组警卫，以使警卫队的总成本（即各个警卫的成本之和）尽可能小。给定博物馆顶点的x坐标和y坐标以及这些顶点沿博物馆边界的顺序，设计一种算法求解该问题，并给出算法的时间复杂性。

请注意，我们并未试图最小化警卫人数。我们希望使警卫队巡逻的路线的总长度最小化，假定任何线段的长度都是线段端点之间的欧几里得距离，并且可以在恒定时间内计算该长度。



上面是说明本问题的四个图形。博物馆是多边形ABCDEFG（顶点的逆时针序）。每个彩色（阴影）三角形对应一个警卫，

上面的两个图显示了一组警卫（它们的三角形），它们满足安全保卫规则。在左上方，警卫队巡逻了三角形AFG（蓝色），ABF（绿色），BEF（淡红色），BDE（浅红色）和BCD（棕色）的边界。在右上方，守卫巡逻ABG（蓝色），BCG（绿色），CDG（棕色），DFG（浅红色）和DEF（浅红色）。

底部的两个图显示了一组不满足这些规则的三角形：在左下图中，博物馆的一部分没有任何警卫守护（覆盖）（无阴影三角形BEF），而在右下图中，粉色三角形（AEF）和绿色三角形（BEF）相交。

1. **基因序列比对问题（选择）**

**问题定义**：基因序列比对是生物信息学中的一个重要问题。给定两个基因序列和，要求找到它们之间的最长公共子序列（LCS）。但是，在比对基因序列时，还需要考虑到一些特殊情况，例如基因突变、缺失、插入等情况。为了解决这些问题，人们通常会引入一个惩罚机制，即为不同的情况赋予不同的惩罚值，例如插入和删除操作会有更高的惩罚，而替换操作会有较低的惩罚。假设有一个矩阵，其中第行第列的元素表示在中插入一个字符转换为中字符的惩罚值。现在，请你设计一个动态规划算法，计算和之间的最短编辑距离（需要经过若干次插入、删除、替换操作才能变成）。

1. 请给出一个递归式，描述该问题的动态规划转移方程。
2. 请设计一个动态规划算法，计算和之间的最短编辑距离，并说明算法的时间复杂度。
3. 在实际应用中，基因序列通常非常长，算法的时间复杂度很高，无法直接使用。请说明如何对算法进行优化，降低时间复杂度。

**解答：**

**注：**本题采用C++编译环境下运行

1. 我们定义dp[i][j]表示S1的前i个字符和S2的前j个字符之间的最短编辑距离，状态转移方程如下：

若s1[i]==s2[j],那么dp[i][j]=dp[i-1][j-1];

若s1[i]!=s2[j],那么dp[i][j]=min(dp[i-1][j-1],dp[i][j-1],dp[i-1][j])+M[i][j];

其中，M[i][j]是矩阵M中第i行第j列的元素，表示在S1中插入一个字符i转换为S2中字符j的惩罚值。上式中的三个部分分别对应替换、插入和删除操作。选择最小的操作数，并加上对应的惩罚值，即可得到当前状态的最优解。

(2)

对于给定的两个基因序列S1和S2，我们可以创建一个二维数组dp，其中dp[i][j]表示S1的前i个字符和S2的前j个字符之间的最短编辑距离。初始状态为dp[0][0] = 0，即两个空序列之间的编辑距离为0。

接下来，我们可以按照递归式进行动态规划计算。具体地，对于每个dp[i][j]，我们可以根据dp[i-1][j-1]、dp[i][j-1]和dp[i-1][j]的值，以及矩阵M中的对应值，计算出dp[i][j]的值。最终，dp[n][m]就是S1和S2之间的最短编辑距离，其中n和m分别表示S1和S2的长度。

算法的时间复杂度为O(nm)，其中n和m分别表示S1和S2的长度。因为算法需要填充一个n\*m的二维数组dp，每个位置需要O(1)的时间计算。因此，总时间复杂度为O(nm)。

C++实现代码如下：

1. #include<bits/stdc++.h>
2. **using** **namespace** std;

5. **int** minEditDistance(string s1, string s2,vector<vector<**int**>>M) {
6. **int** m = s1.size(), n = s2.size();
7. vector<vector<**int**>> dp(m+1, vector<**int**>(n+1, 0));
9. // 初始化第一行和第一列
10. **for** (**int** i = 1; i <= m; i++) {
11. dp[i][0]=dp[i-1][0]+M[i][0];
12. }
13. **for** (**int** j = 1; j <= n; j++) {
14. dp[0][j] = dp[0][j-1]+M[0][j];
15. }
17. // 计算dp数组
18. **for** (**int** i = 1; i <= m; i++) {
19. **for** (**int** j = 1; j <= n; j++) {
20. **int** cost = (s1[i-1] == s2[j-1]) ? 0 : M[i][j];
21. dp[i][j] = min(dp[i-1][j], min(dp[i][j-1] , dp[i-1][j-1]))+M[i][j];
22. }
23. }
25. **return** dp[m][n];
26. }
28. **int** main()
29. {
30. string s1,s2;
31. cin>>s1>>s2;
32. **int** len1=s1.length();
33. **int** len2=s2.length();
34. vector<vector<**int**>>M(len1+1,vector<**int**>(len2+1));
35. **for**(**int** i=0;i<=len1;i++)
36. **for**(**int** j=0;j<=len2;j++)
37. cin>>M[i][j];
38. cout<<minEditDistance(s1,s2,M);
39. **return** 0;
40. }

（3）

在实际应用中，基因序列通常非常长，算法的时间复杂度很高，无法直接使用。因此，我们需要对算法进行优化，降低时间复杂度。

一种常见的优化方法是使用滚动数组（rolling array）来压缩存储空间。具体地，我们可以使用两个一维数组prev和curr，分别表示上一行和当前行的dp值。在计算当前行的dp值时，我们只需要使用上一行和当前行的dp值，因此可以将二维数组压缩为两个一维数组，从而节省空间。

另一种优化方法是使用分治法（divide and conquer）。具体地对于长序列，我们可以将它们分成若干个较短的子序列，分别计算它们之间的最短编辑距离。假设我们将S1和S2分别划分为k个子序列，那么总的时间复杂度为O(k\*n^2)，其中n表示子序列的平均长度。这种方法的优点是可以充分利用现代多核处理器的并行计算能力，提高计算速度。

另外，我们还可以使用启发式搜索（heuristic search）来加速算法。启发式搜索是一种基于启发式函数的搜索算法，它可以帮助我们快速找到最优解。对于基因序列比对问题，我们可以定义一个启发式函数，根据当前状态估计到目标状态的最小编辑距离。然后，我们可以使用A\*算法等启发式搜索算法来寻找最优解，从而加速计算。

综上所述，针对基因序列比对问题，我们可以使用动态规划算法、滚动数组、分治法和启发式搜索等方法进行优化，以降低时间复杂度并提高计算效率。

**注**：提交文件说明：

1.（1）data\_gen.py:随机数据产生文件

（2）merge\_sort.py:第一问归并排序算法文件

（3）pic\_draw.py:画图文件

（4）score\*.txt:原来数据文件；score\*\_res.txt:排序后结果文件

（5）quick\_sort.py:快速排序文件

3.（1）dp.cpp:动态规划代码文件