最长公共子序列问题：枚举法和dp方法时间对比

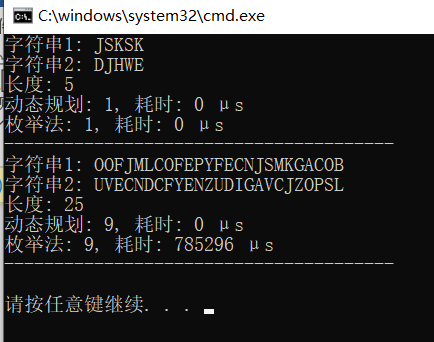
1. 代码：
2. #include <bits/stdc++.h>
3. #define MAXM 10000
4. **using** **namespace** std;
5. **using** **namespace** chrono;
7. string a, b;
8. **int** dp[MAXM][MAXM], m;
10. // 动态规划求最长公共子序列
11. **int** find\_dp(){
12. **for**(**int** i=1;i<=m;i++){
13. **for**(**int** j=1;j<=m;j++){
14. **if**(a[i-1] == b[j-1]) dp[i][j] = dp[i-1][j-1] + 1;
15. **else** dp[i][j] = max(dp[i-1][j], dp[i][j-1]);
16. }
17. }
18. **return** dp[m][m];
19. }
21. // 枚举法求最长公共子序列
22. **int** find\_list() {
23. **int** m = b.length();
24. **int** ans = 0;
25. **for** (**long** **long** **int** i = 0; i < (1 << m); i++) {
26. **int** current\_length = 0;
27. **int** last\_position = -1;
28. **for** (**int** j = 0; j < m; j++) {
29. **if** (i & (1 << j)) {
30. **int** pos = a.find\_first\_of(b[j], last\_position + 1);
31. **if** (pos == string::npos) **break**;
32. **else** {
33. current\_length++;
34. last\_position = pos;
35. }
36. }
37. }
38. ans = max(ans, current\_length);
39. }
40. **return** ans;
41. }
43. // 随机生成字符串
44. string generate\_random\_string(**int** length) {
45. string result;
46. **for** (**int** i = 0; i < length; i++) {
47. result.push\_back('A' + rand() % 26);
48. }
49. **return** result;
50. }
52. **int** main(){
53. ios::sync\_with\_stdio(**false**);
54. srand(time(NULL));
55. **int** lens[] = {5, 25};
56. **for** (**int** size : lens) {
57. a = generate\_random\_string(size);
58. b = generate\_random\_string(size);
59. m = a.size();
61. auto start1 = high\_resolution\_clock::now();
62. **int** ans1 = find\_dp();
63. auto end1 = high\_resolution\_clock::now();
64. auto time1 = duration\_cast<microseconds>(end1 - start1);
66. auto start2 = high\_resolution\_clock::now();
67. **int** ans2 = find\_list();
68. auto end2 = high\_resolution\_clock::now();
69. auto time2 = duration\_cast<microseconds>(end2 - start2);
70. cout << "字符串1: " << a << endl;
71. cout << "字符串2: " << b << endl;
72. cout << "长度: " << size << endl;
73. cout << "动态规划: " << ans1 << ", 耗时: " << time1.count() << " μs" << endl;
74. cout << "枚举法: " << ans2 << ", 耗时: " << time2.count() << " μs" << endl;
75. cout << "---------------------------------------" << endl;
76. }
77. **return** 0;
78. }

代码组成: 随机生成字符串（generate\_random\_string）

枚举法算法（find\_list）

动态规划算法（find\_dp）

1. 分析
2. 实验结果：



1. 正确性验证：
2. 枚举法：枚举法通过遍历字符串A的所有子序列，在字符串B中查找这些子序列的出现情况。由于我们遍历了字符串A的所有可能子序列，并在字符串B中查找它们的出现情况，因此我们可以保证找到最长的公共子序列。这个过程的正确性来源于其穷举的特性：遍历了所有可能的情况，找到最优解。
3. 动态规划法：动态规划法的关键在于状态转移方程和边界条件。在本实验中，我们使用了以下状态转移方程：

\* 如果a[i-1] == b[j-1]，则dp[i][j] = dp[i-1][j-1] + 1；

\* 否则，dp[i][j] = max(dp[i-1][j], dp[i][j-1])。

这个状态转移方程是基于以下思想：对于字符串A的第i个字符和字符串B的第j个字符，如果它们相等，那么它们可以构成一个新的公共子序列，长度为之前的公共子序列长度加1；否则，最长公共子序列的长度为去掉一个字符后的最长公共子序列的长度。通过遍历字符串A和B，并根据状态转移方程填充dp数组，我们可以得到最长公共子序列的长度。

同时，最终动态规划法和枚举法得到的最长公共子序列长度相同。这表明两种方法在求解最长公共子序列问题时能够得到一致的解，同样验证了正确性。

1. 结果分析：从实验结果可以看出，动态规划法在处理最长公共子序列问题时，运行时间明显优于枚举法。在字符串长度为5时，两种方法的耗时都较短，但当字符串长度增加到25时，枚举法的耗时显著增加，而动态规划法耗时仍然很短。这说明动态规划法在处理大规模问题时具有更好的性能。