最长公共子序列问题：枚举法和 dp 方法时间对比

一、 代码：

```cpp
#include <bits/stdc++.h>
#define MAXM 10000
using namespace std;
using namespace chrono;

string a, b;
int dp[MAXM][MAXM], m;

// 动态规划求最长公共子序列
int find_dp(){
    for(int i=1;i<=m;i++){
        for(int j=1;j<=m;j++){
            if(a[i-1] == b[j-1]) dp[i][j] = dp[i-1][j-1] + 1;
            else dp[i][j] = max(dp[i-1][j], dp[i][j-1]);
        }
    }
    return dp[m][m];
}

// 枚举法求最长公共子序列
int find_list() {
    int m = b.length();
    int ans = 0;
    for (long long int i = 0; i < (1 << m); i++) {
        int current_length = 0;
        int last_position = -1;
        for (int j = 0; j < m; j++) {
            if (i & (1 << j)) {
                int pos = a.find_first_of(b[j], last_position + 1);
                if (pos == string::npos) break;
                else {
                    current_length++;
                    last_position = pos;
                }
            }
        }
        ans = max(ans, current_length);
    }
    return ans;
}
```

```cpp

// 随机生成字符串
string generate_random_string(int length) {
    string result;
    for (int i = 0; i < length; i++) {
        result.push_back('A' + rand() % 26);
    }
    return result;
}

int main(){
    ios::sync_with_stdio(false);
    srand(time(NULL));
    int lens[] = {5, 25};
    for (int size : lens) {
        a = generate_random_string(size);
        b = generate_random_string(size);
        m = a.size();

        auto start1 = high_resolution_clock::now();
        int ans1 = find_dp();
        auto end1 = high_resolution_clock::now();
        auto time1 = duration_cast<microseconds>(end1 - start1);

        auto start2 = high_resolution_clock::now();
        int ans2 = find_list();
        auto end2 = high_resolution_clock::now();
        auto time2 = duration_cast<microseconds>(end2 - start2);
        cout << "字符串 1: " << a << endl;
        cout << "字符串 2: " << b << endl;
        cout << "长度: " << size << endl;
        cout << "动态规划: " << ans1 << ", 耗时: " << time1.count() << " μs" << endl;
        cout << "枚举法: " << ans2 << ", 耗时: " << time2.count() << " μs" << endl;
        cout << "--------------------------------------" << endl;
    }
    return 0;
}
```

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）

二、 分析

（1） 实验结果：

（2） 正确性验证：

1. 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。
2. 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。

（3） 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。