

Lab 5: 最长公共子序列算法

PB21020718 曾健斌

一、实验内容

编程实现最长公共子序列(LCS)算法,并理解其核心思想

- 时间复杂度 $O(mn)$,空间复杂度 $O(mn)$,求出 LCS 及其长度
- 时间复杂度 $O(mn)$,空间复杂度 $O(2 \times \min(m, n))$,求出 LCS 的长度
- 时间复杂度 $O(mn)$,空间复杂度 $O(\min(m, n))$,求出 LCS 的长度

二、算法实现

类定义

```
1 class LCS {
2 public:
3     LCS(const string& s1, const string& s2) : s1_(s1), s2_(s2) {}
4
5     string solve();
6     string solve2();
7     string solve2sub(const string& s1, const string& s2);
8     string solve3();
9     string solve3sub(const string& s1, const string& s2);
10
11 private:
12     string s1_, s2_;
13 };
```

空间复杂度 $O(mn)$

定义枚举来标识方向:

```
1 enum Direction { UP, LEFT, DIAG };
```

该算法使用动态规划策略, 与课本中给出的算法伪码思路一致

```
1 string
2 LCS::solve()
3 {
4     // c is the length of the longest common subsequence
5     // b is the direction of the longest common subsequence
6     int m = s1_.size(), n = s2_.size();
7     vector<vector<int>> c(m + 1, vector<int>(n + 1, 0));
8     vector<vector<Direction>> b(m + 1, vector<Direction>(n + 1, UP));
9     for (int i = 1; i <= m; i++) {
10         for (int j = 1; j <= n; j++) {
11             if (s1_[i - 1] == s2_[j - 1]) {
12                 c[i][j] = c[i - 1][j - 1] + 1;
```

```

13         b[i][j] = DIAG;
14     } else if (c[i - 1][j] >= c[i][j - 1]) {
15         c[i][j] = c[i - 1][j];
16         b[i][j] = UP;
17     }
18     else {
19         c[i][j] = c[i][j - 1];
20         b[i][j] = LEFT;
21     }
22 }
23 }
24
25 string ans;
26 int i = m, j = n;
27 while (i > 0 && j > 0) {
28     if (b[i][j] == DIAG){
29         ans += s1[i - 1];
30         i--, j--;
31     }
32     else if (b[i][j] == UP)
33         i--;
34     else
35         j--;
36 }
37 reverse(ans.begin(), ans.end());
38 return ans;
39 }

```

line 6 - 24 是建立自底向上建立最长公共子串的长度表与方向表, line 26 - 38 通过查找两表得到 LCS

空间复杂度 $O(2 \times \min(m, n))$

```

1  string
2  LCS::solve3()
3  {
4      if (s1_.size() > s2_.size())
5          return solve3sub(s1_, s2_);
6      return solve3sub(s2_, s1_);
7  }
8
9  string
10 LCS::solve3sub(const string& s1, const string& s2)
11 {
12     int m = s1.size(), n = s2.size();
13     string ans;
14     int tmp;
15     int nowindex = 0; // 1
16     vector<vector<int>> c(2, vector(n + 1, 0));
17     for (int i = 1; i <= m; i++) {
18         nowindex = i & 1;
19         tmp = 0;
20         for (int j = 1; j <= n; j++) {
21             tmp = c[nowindex][j];
22             if (s1[i - 1] == s2[j - 1])

```

```

23         c[nowindex][j] = c[nowindex ^ 1][j - 1] + 1;
24     else if (c[nowindex ^ 1][j] >= c[nowindex][j - 1])
25         c[nowindex][j] = c[nowindex ^ 1][j];
26     else
27         c[nowindex][j] = c[nowindex][j - 1];
28     }
29 }
30
31 int i = m, j = n;
32 while (i > 0 && j > 0) {
33     if (s1[i - 1] == s2[j - 1]) {
34         ans += s1[i - 1];
35         while (c[nowindex][j - 1] == c[nowindex][j])    // 2
36             j--;
37         i--, j--;
38     }
39     else if (c[nowindex][j - 1] < c[nowindex][j])
40         i--;
41     else
42         j--;
43 }
44 reverse(ans.begin(), ans.end());
45 return ans;
46 }

```

本实现使用一个 $2 \times \min(m, n)$ 的矩阵 c ，并使用 // 1 处变量 `nowindex` 来实现两个数组的循环使用，具体方法是：

- 在每一个外层循环中使 `nowindex` 的值为 `i & 1`
- 在循环过程中使用 `nowindex ^ 1` 来取得另外一个数组

在第二步通过 c 建立 LCS 时，需注意在 // 2 处将 j 减小至 c 数组增加处，本质上，这个循环时保证了左移的优先级高于上移（本实验的一大半时间都在找这个bug），从而使得重建过程与建立 c 的过程保持一致

后者在建表过程中从左向右扫描两个序列，故匹配的字符位于左侧，若不在重建过程中手动提高左移优先级，会导致匹配右侧符号时 `c[nowindex][j - 1]` 与 `c[nowindex][j]` 相等的情况。如此时直接进行 `i--, j--;` 将很有可能丢失后续匹配

空间复杂度 $O(\min(m, n))$

```

1  string
2  LCS::solve2()
3  {
4      if(s1_.size() < s2_.size())
5          return solve2sub(s2_, s1_);
6      return solve2sub(s1_, s2_);
7  }
8
9  string
10 LCS::solve2sub(const string& s1, const string& s2)
11 {
12     int m = s1.size(), n = s2.size();
13     string ans;

```

```

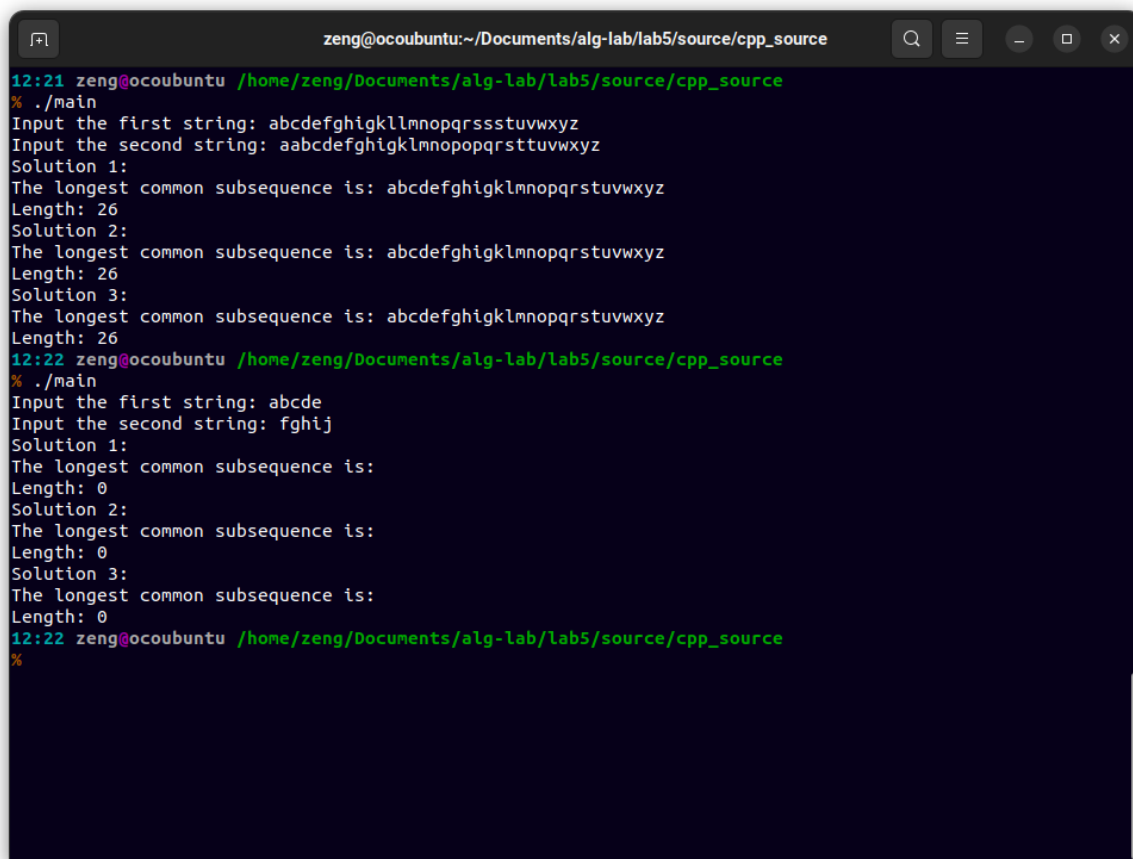
14     int tmp;
15     vector<int> c(n + 1, 0);
16     for (int i = 1; i <= m; i++) {
17         int last = 0;
18         tmp = 0;
19         for (int j = 1; j <= n; j++) {
20             tmp = c[j];
21             if (s1[i - 1] == s2[j - 1])
22                 c[j] = last + 1;
23             else if (c[j - 1] >= c[j])
24                 c[j] = c[j - 1];
25             last = tmp;           // 1
26         }
27     }
28
29     int i = m, j = n;
30     while (i > 0 && j > 0) {
31         if (s1[i - 1] == s2[j - 1]) {
32             ans += s1[i - 1];
33             while (c[j - 1] == c[j])      // 2
34                 j--;
35             i--, j--;
36         }
37         else if (c[j - 1] < c[j])
38             i--;
39         else
40             j--;
41     }
42     reverse(ans.begin(), ans.end());
43     return ans;
44 }

```

本实现使用一个长度为 $\min(m, n)$ 的一维数组，以及变量 `tmp` 与 `last` 作为额外空间，具体方法是：

- `last` 用于记录当前计算位置左上角的位置（在一个虚拟矩阵中），而 `tmp` 用于在计算过程中保存当前位置初始值（当前位置正上方位置），并在内层循环结束时（// 1）将 `last` 置为 `tmp`
- 注意在 // 2 同上一个实现一样需要手动提高左移的优先级

三、实验结果



```
zeng@ocoubuntu:~/Documents/alg-lab/lab5/source/cpp_source
12:21 zeng@ocoubuntu /home/zeng/Documents/alg-lab/lab5/source/cpp_source
% ./main
Input the first string: abcdefghigklmnopqrssstuvwxyz
Input the second string: aabcdefghigklmnopopqrstuvwxyz
Solution 1:
The longest common subsequence is: abcdefghigklmnopqrstuvwxy
Length: 26
Solution 2:
The longest common subsequence is: abcdefghigklmnopqrstuvwxy
Length: 26
Solution 3:
The longest common subsequence is: abcdefghigklmnopqrstuvwxy
Length: 26
12:22 zeng@ocoubuntu /home/zeng/Documents/alg-lab/lab5/source/cpp_source
% ./main
Input the first string: abcde
Input the second string: fghij
Solution 1:
The longest common subsequence is:
Length: 0
Solution 2:
The longest common subsequence is:
Length: 0
Solution 3:
The longest common subsequence is:
Length: 0
12:22 zeng@ocoubuntu /home/zeng/Documents/alg-lab/lab5/source/cpp_source
%
```

三种实现均运行正常，输出正确的 LCS 及其长度

四、复杂度分析

时间复杂度

三个实现的时间复杂度均为 $O(mn)$ ，主要耗时在于二层循环建表上

空间复杂度

- 实现一建立了两个 $m \times n$ 的矩阵，空间复杂度为 $O(m \times n)$
- 实现二使用了一个 $2 \times \min(m, n)$ 的矩阵，空间复杂度为 $O(2 \times \min(m, n))$
- 实验三使用了一个长度为 $\min(m, n)$ 的一维矩阵以及若干临时变量，空间复杂度为 $O(\min(m, n)) + O(1)$

五、实验总结

- 本次实验总体较为简单，主要是需要处理好使用一维数组来模拟在矩阵上操作的相关细节
- 值得注意的是，在使用一维数组时需要指明左移与上移的优先级，而在附加信息更多的矩阵上则无需顾虑