

<https://leetcode.com/problems/edit-distance/description/>

Given two strings word1 and word2, return *the minimum number of operations required to convert word1 to word2*.

You have the following three operations permitted on a word:

- Insert a character
- Delete a character
- Replace a character

Example 1:

Input: word1 = "horse", word2 = "ros"

Output: 3

Explanation:

horse -> rorse (replace 'h' with 'r')

rorse -> rose (remove 'r')

rose -> ros (remove 'e')

Example 2:

Input: word1 = "intention", word2 = "execution"

Output: 5

Explanation:

intention -> inention (remove 't')

inention -> enention (replace 'i' with 'e')

enention -> exention (replace 'n' with 'x')

exention -> exection (replace 'n' with 'c')

exection -> execution (insert 'u')

Constraints:

- $0 \leq \text{word1.length}, \text{word2.length} \leq 500$
- word1 and word2 consist of lowercase English letters.

Attempt 1: 2023-07-12

(1) 基于L115文献的逆向递归进化到DP的思路

word1, word2 scan from left to right

在L115中文文献中，从递归的角度讲，“顶”就是递归最开始在主体方法中被呼叫的状态，本题中就是 $i == 0$ 和 $j == 0$ 时，“底”在本题中就是当递归到达 $i == \text{s1.length}()$ 和 $n == \text{s2.length}()$ 时，也就是递归实际方法中的base condition，递归就是先从“顶”即 $i == 0$ 和 $j == 0$ 逐层到达“底”即 $i == \text{s1.length}()$ 和 $n ==$

s2.length(), 然后在到达"底"后再通过返回语句逐层从"底"返回到"顶", 而DP能够省略掉递归中"从顶到底"的过程, 而"直接由底向顶", 这也意味着从二维数组DP状态表的角度讲, 从右下角逆推到左上角的过程, 也就是i == s1.length()(底) --> i == 0(顶), n == s2.length()(底) --> n == 0(顶)的过程

第一步: 实现一个基本递归(逆向版本):

在递归的过程就是由顶到底再回到顶

递归中由顶到底的过程:

我们的递归始于i == 0和j == 0时, i == 0(顶) --> i == s1.length()(底), j == 0(顶) --> j == s2.length()(底), 然后在到底的时候触碰到base condition开启return返回过程

递归中再由底回到顶的过程:

在从顶到底并触碰到base condition开启return之后, 逐层返回, i == s1.length()(底) --> i == 0(顶), j == s2.length()(底) --> j == 0(顶), 此时最终状态实际上在顶, 也就是i == 0和j == 0时取得, 和二维DP中最终状态在左上角[0, 0]处获得形成一致

```
1 class Solution {
2     public int minDistance(String word1, String word2) {
3         // 从顶i = 0和j = 0开始递归
4         return helper(word1, 0, word2, 0);
5     }
6     private int helper(String s1, int i, String s2, int j) {
7         // 在底i == s1.length()和j == s2.length()触底开启逐层返回到顶过程
8         // Base condition 1:
9         // When s1 scan index as i reach the end, the deviation
10        // between s2 scan index as j and the length of s2 is the
11        // minimum number of operations required to match s1 and s2
12        if(i == s1.length()) {
13            return s2.length() - j;
14        }
15        // Base condition 2:
16        // When s2 scan index as j reach the end, the deviation
17        // between s1 scan index as i and the length of s1 is the
18        // minimum number of operations required to match s1 and s2
19        if(j == s2.length()) {
20            return s1.length() - i;
21        }
22        // Divide
```

```

23     int result = 0;
24     // If current position pair match
25     if(s1.charAt(i) == s2.charAt(j)) {
26         // no +1 as current position match, not require change and move on to next
27         result = helper(s1, i + 1, s2, j + 1);
28     // If current position pair not match
29     } else {
30         // Insert
31         // Try to modify s1 to match s2, if insert new character
32         // at the head of s1
33         // e.g s1 = "bcd", i = 0, s2 = "abc", j = 0, insert 'a' at the
34         // head of s1 then s1 = "abcd", but we don't really add 'a', the
35         // dummy 'a' is used to flatten the first different pair of
36         // character's deviation as 'b' in s1 and 'a' in s2, mock it as
37         // 'a' in s1 and 'a' in s2.
38         // Now we don't have to change index as i in s1, keep i = 0,
39         // still point to the original first character as 'b' in original
40         // s1 = "bcd", which also the second character in new s1 = "abcd",
41         // but for index as j in s2, we have to shift to next position
42         // because its current position as j = 0 character as 'a' already
43         // "perished" after utilization as counterpart with dummy 'a' on s1.
44         // In current example, i = 0 in s1 keep pointing to 'b' in updated
45         // s1 = "abcd", j = 0 update to j = 1 in s2 will skip same character
46         // as 'a' it original point to in s2 and point to next character as
47         // 'b' in s2 = "abc"
48         // s1 = "bcd"                s1 = "abcd"
49         //      ^  --> i = 0  ==>      ^  --> i = 0
50         // s2 = "abc"                s2 = "abc"
51         //      ^  --> j = 0          ^  --> j + 1 = 1
52         int insert_step = helper(s1, i, s2, j + 1);
53         // Delete
54         // Removes the first character, shifting s1 character to left.
55         // Since we do not actually delete the character, incrementing i
56         // simulates skipping this character
57         int delete_step = helper(s1, i + 1, s2, j);
58         // Replace
59         // We replace the cur char with the char we need from s2, then
60         // increment i and j to look at the next char
61         int replace_step = helper(s1, i + 1, s2, j + 1);
62         // +1 => each operation take one more step

```

```

63         result = Math.min(Math.min(insert_step, delete_step), replace_step) + 1;
64     }
65     return result;
66 }
67 }

```

第二步：递归配合Memoization(逆向版本):

```

1  class Solution {
2      public int minDistance(String word1, String word2) {
3          Integer[][] memo = new Integer[word1.length() + 1][word2.length() + 1];
4          // 从顶i = 0和j = 0开始递归
5          return helper(word1, 0, word2, 0, memo);
6      }
7
8      private int helper(String s1, int i, String s2, int j, Integer[][] memo) {
9          if(memo[i][j] != null) {
10              return memo[i][j];
11          }
12          // 在底i == s1.length()和j == s2.length()触底开启逐层返回到顶过程
13          // Base condition 1:
14          // When s1 scan index as i reach the end, the deviation
15          // between s2 scan index as j and the length of s2 is the
16          // minimum number of operations required to match s1 and s2
17          if(i == s1.length()) {
18              return s2.length() - j;
19          }
20          // Base condition 2:
21          // When s2 scan index as j reach the end, the deviation
22          // between s1 scan index as i and the length of s1 is the
23          // minimum number of operations required to match s1 and s2
24          if(j == s2.length()) {
25              return s1.length() - i;
26          }
27          // Divide
28          int result = 0;
29          // If current position pair match

```

```

30     if(s1.charAt(i) == s2.charAt(j)) {
31         // no +1 as current position match, not require change and move on to next
32         result = helper(s1, i + 1, s2, j + 1, memo);
33     // If current position pair not match
34     } else {
35         // Insert
36         // Try to modify s1 to match s2, if insert new character
37         // at the head of s1
38         // e.g s1 = "bcd", i = 0, s2 = "abc", j = 0, insert 'a' at the
39         // head of s1 then s1 = "abcd", but we don't really add 'a', the
40         // dummy 'a' is used to flatten the first different pair of
41         // character's deviation as 'b' in s1 and 'a' in s2, mock it as
42         // 'a' in s1 and 'a' in s2.
43         // Now we don't have to change index as i in s1, keep i = 0,
44         // still point to the original first character as 'b' in original
45         // s1 = "bcd", which also the second character in new s1 = "abcd",
46         // but for index as j in s2, we have to shift to next position
47         // because its current position as j = 0 character as 'a' already
48         // "perished" after utilization as counterpart with dummy 'a' on s1.
49         // In current example, i = 0 in s1 keep pointing to 'b' in updated
50         // s1 = "abcd", j = 0 update to j = 1 in s2 will skip same character
51         // as 'a' it original point to in s2 and point to next character as
52         // 'b' in s2 = "abc"
53         // s1 = "bcd"                s1 = "abcd"
54         //      ^  --> i = 0  ==>      ^  --> i = 0
55         // s2 = "abc"                s2 = "abc"
56         //      ^  --> j = 0          ^  --> j + 1 = 1
57         int insert_step = helper(s1, i, s2, j + 1, memo);
58         // Delete
59         // Removes the first character, shifting s1 character to left.
60         // Since we do not actually delete the character, incrementing i
61         // simulates skipping this character
62         int delete_step = helper(s1, i + 1, s2, j, memo);
63         // Replace
64         // We replace the cur char with the char we need from s2, then
65         // increment i and j to look at the next char
66         int replace_step = helper(s1, i + 1, s2, j + 1, memo);
67         // +1 => each operation take one more step
68         result = Math.min(Math.min(insert_step, delete_step), replace_step) + 1;
69     }

```

```

70         return memo[i][j] = result;
71     }
72 }

```

第三步：基于递归的2D DP(逆向版本)：

DP能够省略掉递归中"从顶到底"的过程，而"直接由底向顶"，这也意味着从二维数组DP状态表的角度讲，从右下角逆推到左上角的过程，也就是 $i == s1.length()$ (底) $\rightarrow i == 0$ (顶)， $j == s2.length()$ (底) $\rightarrow j == 0$ (顶)的过程

这里我们用一个二维数组 $dp[i][j]$ 对应于从 $s[i, s1.length())$ 所代表的的字符串需要多少步变成 $s2[j, s2.length())$ 。

当 $i == s1.length()$ ，意味着 $s1$ 是空串，此时 $dp[s1.length()][j]$ ，取值随 j 变化，即 $s2.length() - j$

当 $j == s2.length()$ ，意味着 $s2$ 是空串，此时 $dp[i][s2.length()]$ ，取值随 i 变化，即 $s1.length() - i$

然后状态转移的话和解法一分析的一样。如果求 $dp[i][j]$ 。

- $s1[i] == s2[j]$ ，当前两个字符相等，需要多少步 $s1$ 能变成 $s2$ 取决于同时跳过当前字符时两个字符串的关系，之前需要多少步现在仍需要多少步

$dp[i][j] = dp[i+1][j+1]$

- $s1[i] != s2[j]$ ，有三种情况，1.去掉一个字符，2.加入一个字符，3.替换一个字符，取三种方案中所需步骤最少的方案

$dp[i][j] = \text{Math.min}(\text{Math.min}(dp[i+1][j], dp[i][j+1]), dp[i+1][j+1])$

insert -> 对应 $dp[i][j+1]$

delete -> 对应 $dp[i+1][j]$

replace -> 对应 $dp[i+1][j+1]$

代码就可以写了。

```

1  class Solution {
2      /**
3       * 在底  $i == s1.length()$  和  $j == s2.length()$  触底开启逐层返回到顶  $i = 0$  和  $j = 0$  过程
4       * 观察基础状态在  $i$  和  $j$  到达底(即原字符串长度时)获得，尤其当  $i$  和  $j$  同时到达底时，即
5       *  $dp[s1.length()][s2.length()] = 0$ 
6       *  $dp[s1.length()][j] = s2.length() - j = \{3, 2, 1, 0\}$ 
7       *  $dp[i][s2.length()] = s1.length() - i = \{5, 4, 3, 2, 1, 0\}$ 
8       * if( $i == s1.length()$ ) {
9       *     return  $s2.length() - j$ ;
10    }

```

```

11         if(j == s2.length()) {
12             return s1.length() - i;
13         }
14         然后是递推关系也符合递归（s1 and s2 scan from left to right）中的逻辑，即
15         if(word1.charAt(i) != word2.charAt(j)) {
16             dp[i][j] = Math.min(Math.min(dp[i + 1][j], dp[i][j + 1]), dp[i + 1][j +
17         1]) + 1;
18         } else {
19             dp[i][j] = dp[i + 1][j + 1];
20         }
21
22         e.g s1 = "horse", s2 = "ros"
23
24         0 1 2 3
25         s2 r o s '' -> j
26
27         s1
28         0 h   3 3 4 5
29         1 o   3 2 3 4
30         2 r   2 2 2 3
31         3 s   3 2 1 2
32         4 e   3 2 1 1
33         5 ''  3 2 1 0
34
35         -> i
36
37         */
38
39         public int minDistance(String word1, String word2) {
40             int w1_len = word1.length();
41             int w2_len = word2.length();
42             int[][] dp = new int[w1_len + 1][w2_len + 1];
43             // No need set up below since it covered by two base conditions
44             // dp[w1_len][w2_len] = 0
45             // 对应递归底时的基础状态1: i到底（即word1的长度）
46             // dp[word1.length()][j] = word2.length() - j
47             for(int j = 0; j <= w2_len; j++) {
48                 dp[w1_len][j] = w2_len - j;
49             }
50             // 对应递归底时的基础状态2: j到底（即word2的长度）
51             // dp[i][word2.length()] = word1.length() - i
52             for(int i = 0; i <= w1_len; i++) {
53                 dp[i][w2_len] = w1_len - i;
54             }
55             // 倒着进行，word1每次增加一个字母（row维度）

```

```

50     for(int i = w1_len - 1; i >= 0; i--) {
51         // 倒着进行，word2每次增加一个字母（column维度）
52         for(int j = w2_len - 1; j >= 0; j--) {
53             // 当两个字母不相等，当前状态取决于上一层insert, delete, replace三种情况的
54             // 最小结果，然后加一步
55             // 对应递归中的关系：result = Math.min(Math.min(insert_step,
delete_step), replace_step) + 1;
56             // 同原始递归解法中一致，依然以改变word1（row维度i）去匹配word2（column维度
j）
57             // insert -> 对应dp[i][j + 1]
58             // delete -> 对应dp[i + 1][j]
59             // replace -> 对应dp[i + 1][j + 1]
60             /**
61                 e.g
62                 s1 = "horse", s2 = "ros"
63                     0 1 2 3
64                 s2 r o s '' -> j
65                 s1
66                 0 h   ?   5
67                 1 o       4
68                 2 r       3
69                 3 s       2
70                 4 e       1 1
71                 5 ''   3 2 1 0
72                 -> i
73                 dp[4][2] = Math.min(Math.min(dp[5][2], dp[4][3]), dp[5][3])
74                 即需要多少步把s1 = "e"变成s2 = "s"呢？
75                 -----
--
76                 insert -> 和递归中一样采用头插法，在s1前面虚拟插入字符's'，此时s1变
成"se"，
77                 s2依然是"s"，遵循递归中的思路，扫描s1的坐标i不用改变，保持原位，即i = 4
不变，
78                 但s2的坐标j需要向下一个位置移动，因为此时s1中虚拟插入的字符's'已经和s2当
前
79                 j = 2所指示的's'匹配并抵消了，要判断s1是否和s2一致需要看s1中i = 4所指代
的字
80                 符和s2中j + 1 = 2 + 1 = 3所指代的字符是否一致，不过我们看到s2并没有下一
个字符，
81                 此时直接到达s2到头的边界条件（对应递归中的底之一，j == s2.length()），那
么还
82                 需要多少步实现s1和s2一致呢？dp[4][3] = 1的结果和递归中以下返回值保持了一
致性，

```



```
83         if(j == s2.length()) return s1.length() - i; --> 5 - 4 = 1
84
85 变的
86
87 一步
88
89
90
91
92
93
94
95
96
97
98
99
100
101
102
103
104
105
106
107
108
109
110
111
112
113
114
115
116
117
118
119
120
121
122
123
124
125
126
127
128
129
130
131
132
133
134
135
136
137
138
139
140
141
142
143
144
145
146
147
148
149
150
151
152
153
154
155
156
157
158
159
160
161
162
163
164
165
166
167
168
169
170
171
172
173
174
175
176
177
178
179
180
181
182
183
184
185
186
187
188
189
190
191
192
193
194
195
196
197
198
199
200
201
202
203
204
205
206
207
208
209
210
211
212
213
214
215
216
217
218
219
220
221
222
223
224
225
226
227
228
229
230
231
232
233
234
235
236
237
238
239
240
241
242
243
244
245
246
247
248
249
250
251
252
253
254
255
256
257
258
259
260
261
262
263
264
265
266
267
268
269
270
271
272
273
274
275
276
277
278
279
280
281
282
283
284
285
286
287
288
289
290
291
292
293
294
295
296
297
298
299
300
301
302
303
304
305
306
307
308
309
310
311
312
313
314
315
316
317
318
319
320
321
322
323
324
325
326
327
328
329
330
331
332
333
334
335
336
337
338
339
340
341
342
343
344
345
346
347
348
349
350
351
352
353
354
355
356
357
358
359
360
361
362
363
364
365
366
367
368
369
370
371
372
373
374
375
376
377
378
379
380
381
382
383
384
385
386
387
388
389
390
391
392
393
394
395
396
397
398
399
400
401
402
403
404
405
406
407
408
409
410
411
412
413
414
415
416
417
418
419
420
421
422
423
424
425
426
427
428
429
430
431
432
433
434
435
436
437
438
439
440
441
442
443
444
445
446
447
448
449
450
451
452
453
454
455
456
457
458
459
460
461
462
463
464
465
466
467
468
469
470
471
472
473
474
475
476
477
478
479
480
481
482
483
484
485
486
487
488
489
490
491
492
493
494
495
496
497
498
499
500
501
502
503
504
505
506
507
508
509
510
511
512
513
514
515
516
517
518
519
520
521
522
523
524
525
526
527
528
529
530
531
532
533
534
535
536
537
538
539
540
541
542
543
544
545
546
547
548
549
550
551
552
553
554
555
556
557
558
559
560
561
562
563
564
565
566
567
568
569
570
571
572
573
574
575
576
577
578
579
580
581
582
583
584
585
586
587
588
589
590
591
592
593
594
595
596
597
598
599
600
601
602
603
604
605
606
607
608
609
610
611
612
613
614
615
616
617
618
619
620
621
622
623
624
625
626
627
628
629
630
631
632
633
634
635
636
637
638
639
640
641
642
643
644
645
646
647
648
649
650
651
652
653
654
655
656
657
658
659
660
661
662
663
664
665
666
667
668
669
670
671
672
673
674
675
676
677
678
679
680
681
682
683
684
685
686
687
688
689
690
691
692
693
694
695
696
697
698
699
700
701
702
703
704
705
706
707
708
709
710
711
712
713
714
715
716
717
718
719
720
721
722
723
724
725
726
727
728
729
730
731
732
733
734
735
736
737
738
739
740
741
742
743
744
745
746
747
748
749
750
751
752
753
754
755
756
757
758
759
760
761
762
763
764
765
766
767
768
769
770
771
772
773
774
775
776
777
778
779
780
781
782
783
784
785
786
787
788
789
790
791
792
793
794
795
796
797
798
799
800
801
802
803
804
805
806
807
808
809
810
811
812
813
814
815
816
817
818
819
820
821
822
823
824
825
826
827
828
829
830
831
832
833
834
835
836
837
838
839
840
841
842
843
844
845
846
847
848
849
850
851
852
853
854
855
856
857
858
859
860
861
862
863
864
865
866
867
868
869
870
871
872
873
874
875
876
877
878
879
880
881
882
883
884
885
886
887
888
889
890
891
892
893
894
895
896
897
898
899
900
901
902
903
904
905
906
907
908
909
910
911
912
913
914
915
916
917
918
919
920
921
922
923
924
925
926
927
928
929
930
931
932
933
934
935
936
937
938
939
940
941
942
943
944
945
946
947
948
949
950
951
952
953
954
955
956
957
958
959
960
961
962
963
964
965
966
967
968
969
970
971
972
973
974
975
976
977
978
979
980
981
982
983
984
985
986
987
988
989
990
991
992
993
994
995
996
997
998
999
1000
```

变的

一步

就可以变成"s"，此时匹配了s2

```
s1 = "e"                s1 = "se"
      ^  --> i = 4  ==>      ^  --> i = 4
s2 = "s"                s2 = "s"
      ^  --> j = 2          ^  --> j + 1 = 3
```

反映在2D DP数组中dp[4][2]可以是dp[4][3] + 1

--

delete -> 删除一个字符，在s1中删除一个字符，此时s1变成""空字符串，此时直接

到达s1到头的边界条件（对应递归中的底之一，i == s1.length()），s2依然是

"s"，遵循递归中的思路，扫描s2的j不用改变，扫描s1的i需要向下一个位置移动，

因为

当前

少步

实现s1和s2一致呢？dp[5][2] = 1的结果和递归中以下返回值保持了一致性，

```
if(i == s1.length()) return s2.length() - j; --> 3 - 2 = 1
```

可以理解为要让dp[4][2]在s1中delete一个字符的情况下去匹配没有改变的s2还需

dp[5][2]所代表的步骤，即还需要1步，s1 = ""

```

113         i = 4和j = 2的情况下同时后移一个位置同时到头，同时达到边界条件（对应递归
    中底
114         的两种情况：i == s1.length()和j == s2.length()），照说此时s1和s2已经完
    成了匹配，
115         那么还需要多少步实现s1和s2一致呢？理论上不再需要步骤了，应该为0步，我们来
    看看
116         dp[5][3]的结果是否符合猜想，dp[5][3]在初始化中就同时被包含在以下2个初始
    化中：
117         for(int j = 0; j <= w2_len; j++) {dp[w1_len][j] = w2_len - j;}
118         for(int i = 0; i <= w1_len; i++) {dp[i][w2_len] = w1_len - i;}
119         无论从哪个角度都是dp[5][3] = 0，本质含义就是当s1和s2都是空串的时候不需要
    步骤
120         来完成匹配了，这也符合递归中"底"（同时满足两个base condition）的表述
121         s1 = "e"                s1 = ""
122             ^ --> i = 4 ==>         ^ --> i + 1 = 5
123         s2 = "s"                s2 = ""
124             --> j = 2                ^ --> j + 1 = 3
125
126         反映在2D DP数组中dp[4][2]可以是dp[5][3] + 1
127         -----
128         --
129         */
130         if(word1.charAt(i) != word2.charAt(j)) {
131             dp[i][j] = Math.min(Math.min(dp[i + 1][j], dp[i][j + 1]), dp[i + 1]
[j + 1]) + 1;
132             // 当两个字母相等，两个字符串都不需要做任何变动，当前状态和上一层状态一致，
133             // 直接跳过当前层，不需要加一步
134             // 对应递归中的关系：result = helper(s1, i + 1, s2, j + 1)
135             } else {
136                 dp[i][j] = dp[i + 1][j + 1];
137             }
138         }
139         return dp[0][0];
140     }
141 }

```

第四步：基于2D DP的空间优化1D DP(逆向版本)：

优化为2 rows (相对于L115真正展现2 rows array替代2D array的本质)

```

1  class Solution {
2      public int minDistance(String word1, String word2) {
3          int w1_len = word1.length();
4          int w2_len = word2.length();
5          // 原2D DP数组中的定义: word1是row维度, word2是column维度
6          //int[][] dp = new int[w1_len + 1][w2_len + 1];
7          // No need set up below since it covered by two base conditions
8          // dp[word1.length()][word2.length()] = 0
9          // 对应递归底时的基础状态1: i到底 (即word1的长度)
10         // dp[word1.length()][j] = word2.length() - j
11         //for(int j = 0; j <= w2_len; j++) {
12         //    dp[w1_len][j] = w2_len - j;
13         //}
14         // 对应递归底时的基础状态2: j到底 (即word2的长度)
15         // dp[i][word2.length()] = word1.length() - i
16         //for(int i = 0; i <= w1_len; i++) {
17         //    dp[i][w2_len] = w1_len - i;
18         //}
19         // -> 现在只保留了column维度, 因为本质上是row的维度上"上一行只依赖于下一行", 在原2D数
           组中上一行是dp[i], 下一行是dp[i + 1], 现在由于去掉了row维度, dp[i][j]平行替换为dp[j], dp[i
           + 1][j]平行替换为dpPrev[j], dp[i + 1][j + 1]平行替换为dpPrev[j + 1]
20         int[] dp = new int[w2_len + 1];
21         int[] dpPrev = new int[w2_len + 1];
22         // -> 去掉row维度后初始化状态进化为只需要设定剩下column维度
23         for(int i = 0; i <= w2_len; i++) {
24             // 注意: 这题和L115中随便初始化dp或者dpPrev效果一致不同, 这题的初始化让人更透彻的
25             // 理解如何用2 rows array来模拟2D array的效果
26             // 如下使用s1 = "horse", s2 = "ros"演示逆向模式 (以dp[0][0]为结束) 下的情况:
27             // 因为我们是使用dpPrev和dp两个1D array迭代来替代原先的2D array, 在实际替代的初始
           化中,
28             // 第一个被替换的就是原先2D array中的最后一行, 但是不像L115中随使用dp或者dpPrev
29             // 替代那样效果相当, 不出问题。实际上dpPrev必须是最优先被赋值的, 因为得用dpPrev推
           算出
30             // dp, 但在L115中巧合的是dpPrev即使和dp初始化为同样的值也不影响, 因为在L115逆向
           模式
31             // 中倒数第一行和倒数第二行的最后一个数字都是一样的 (都为1), 而且L115中计算dp时
           用到
32             // 的公式为dp[i] = dpPrev[i + 1] + dpPrev[i], 并不需要dp本行的元素参与, 但是本
           题中
33             // 倒数第一行和倒数第二行的最后一个数字并不是一样的 (一个为0, 一个为1), 而且本题
           计算

```

```

34         // dp时用到的公式为dp[j] = Math.min(Math.min(dpPrev[j], dp[j + 1]),
dpPrev[j + 1]) + 1,
35         // 我们发现dp[j]用到了本行元素dp[j + 1]。举个例子，假设倒数第一行在初始化中用
dpPrev
36         // 代表，倒数第二行在初始化中用dp代表，假设倒数第二行的倒数第二个元素是dp[j]，那
么在
37         // 计算该元素的时候不仅会用到倒数第一行的元素dpPrev[j + 1]和dpPrev[j]，也会用到
倒数
38         // 第二行的倒数第一个元素dp[j + 1]，所以实际上dpPrev和dp必须是严格分开定义的，不
可混淆。
39         // 总体来说，在本题的逆向模式中dpPrev代表原2D DP array的倒数第一行，dp则是倒数第
二行，
40         // 并且使用dpPrev来推算，但在推算dp，也即倒数第二行所有其他元素前，其最后一个元素
也得
41         // 附上数值，原因如前述
42         /**
43             e.g s1 = "horse", s2 = "ros"
44                 0 1 2 3
45             s2 r o s '' -> j
46             s1
47             0 h   3 3 4 5
48             1 o   3 2 3 4
49             2 r   2 2 2 3
50             3 s   3 2 1 2
51             4 e   3 2 1[1] -> the last element '1' in second last row equal
initial dp array
52             5 ''   3 2 1 0 -> the last row equal initial dpPrev array (not dp
array)
53             -> i
54             */
55         //dp[i] = w2_len - i; --> wrong way, since last row initialize must only
allocate to dpPrev
56         dpPrev[i] = w2_len - i;
57     }
58     // 倒着进行，word1每次增加一个字母（row维度）
59     // -> 外层循环依旧为row维度，而且dpPrev/dp在row维度的反复替换也在外层循环发生，为了维
持row维度的替换，外层循环必须使用row维度
60     for(int i = w1_len - 1; i >= 0; i--) {
61         // -> 根据上述细节叙述，因为dp推算公式中含有dp同行元素，推算dp其他元素前必须初始
化dp的最后一个元素(因为逆向模式中是从右往左推导)
62         dp[w2_len] = w1_len - i;
63         // 倒着进行，word2每次增加一个字母（column维度）
64         for(int j = w2_len - 1; j >= 0; j--) {

```

```

65         if(word1.charAt(i) != word2.charAt(j)) {
66             //dp[i][j] = Math.min(Math.min(dp[i + 1][j], dp[i][j + 1]), dp[i +
1][j + 1]) + 1;
67             // -> 现在由于去掉了row维度, dp[i][j]平行替换为dp[j], dp[i + 1][j]平行
替换为dpPrev[j], dp[i][j + 1]平行替换为dp[j + 1], dp[i + 1][j + 1]平行替换为dpPrev[j + 1]
68             dp[j] = Math.min(Math.min(dpPrev[j], dp[j + 1]), dpPrev[j + 1]) +
1;
69             // 当两个字母相等, 两个字符串都不需要做任何变动, 当前状态和上一层状态一致,
70             // 直接跳过当前层, 不需要加一步
71             // 对应递归中的关系: result = helper(s1, i + 1, s2, j + 1)
72         } else {
73             //dp[i][j] = dp[i + 1][j + 1];
74             dp[j] = dpPrev[j + 1];
75         }
76     }
77     dpPrev = dp.clone();
78 }
79 return dpPrev[0];
80 }
81 }

```

进一步优化为1 row (不是真正的1 row方案, 内层循环不需要反转, 因为只是用2个变量替代了2 rows 中的1 row)

```

1  class Solution {
2      public int minDistance(String word1, String word2) {
3          int w1_len = word1.length();
4          int w2_len = word2.length();
5          //int[] dp = new int[w2_len + 1]; -> remove dp array
6          int[] dpPrev = new int[w2_len + 1];
7          for(int i = 0; i <= w2_len; i++) {
8              dpPrev[i] = w2_len - i;
9          }
10         int prev = 0;
11         for(int i = w1_len - 1; i >= 0; i--) {
12             // Special handling to store current row(dpPrev)'s last
13             // element in each round as a single variable 'prev',
14             // since it will be used in formula below to replace

```

```

15         // 'dpPrev[j + 1]' further
16         // dp[j] = Math.min(Math.min(dpPrev[j], dp[j + 1]), dpPrev[j + 1]) + 1;
17         prev = dpPrev[w2_len];
18         // Then update current row(dpPrev)'s last element to next round
19         // row(dp)'s last element value(w1_len - i) which suppose
20         // belongs to dp, but we don't create dp to represent
21         // next round row now
22         dpPrev[w2_len] = w1_len - i;
23         // No need to reverse the scanning order like L115, but why ?
24         // Because its NOT the REAL 1D array solution, its just use two
25         // variables as 'prev' to represent the necessary last element in
26         // 'dpPrev' array and 'temp' to update 'prev' each round, not
27         // fully dismiss 'dp' array, just use two variables to MOCK another
28         // 'dp' array
29         // for(int j = 0; j <= w2_len - 1; j++) { --> Wrong Way !!!
30         for(int j = w2_len - 1; j >= 0; j--) {
31             int temp = dpPrev[j];
32             if(word1.charAt(i) != word2.charAt(j)) {
33                 //dp[j] = Math.min(Math.min(dpPrev[j], dp[j + 1]), dpPrev[j + 1])
+ 1; -> 'prev' replace dpPrev[j + 1]
34                 dpPrev[j] = Math.min(Math.min(dpPrev[j], dpPrev[j + 1]), prev) + 1;
35             } else {
36                 //dp[j] = dpPrev[j + 1]; -> 'prev' replace dpPrev[j + 1]
37                 dpPrev[j] = prev;
38             }
39             prev = temp;
40         }
41         //dpPrev = dp.clone();
42     }
43     return dpPrev[0];
44 }
45 }

```

Refer to

<https://leetcode.com/problems/edit-distance/solutions/25846/c-o-n-space-dp/>

```

1 class Solution {
2 public:

```

```

3     int minDistance(string word1, string word2) {
4         int m = word1.size(), n = word2.size(), pre;
5         vector<int> cur(n + 1, 0);
6         for (int j = 1; j <= n; j++) {
7             cur[j] = j;
8         }
9         for (int i = 1; i <= m; i++) {
10            pre = cur[0];
11            cur[0] = i;
12            for (int j = 1; j <= n; j++) {
13                int temp = cur[j];
14                if (word1[i - 1] == word2[j - 1]) {
15                    cur[j] = pre;
16                } else {
17                    cur[j] = min(pre, min(cur[j - 1], cur[j])) + 1;
18                }
19                pre = temp;
20            }
21        }
22        return cur[n];
23    }
24 };

```

(2) 基于L115文献的正向递归进化到DP的思路

word1, word2 scan from right to left

第一步：实现一个基本递归(正向版本)：

在LeetCode解答中，从递归的角度讲，“顶”就是递归最开始在主体方法中被呼叫的状态，本题中就是 $i == s1.length()$ 和 $j == s2.length()$ 时，“底”在本题中就是当递归到达 $i == 0$ 和 $j == 0$ 时，也就是递归实际方法中的base condition，递归就是先从“顶”即 $i == s1.length()$ 和 $j == s2.length()$ 逐层到达“底”即 $i == 0$ 和 $j == 0$ ，然后在到达“底”后再通过返回语句逐层从“底”返回到“顶”，而DP能够省略掉递归中“从顶到底”的过程，而“直接由底向顶”，这也意味着从二维数组DP状态表的角度讲，从左上角正推到右下角的过程，也就是 $i == 0$ (底) $--> i == s1.length()$ (顶)， $j == 0$ (底) $--> j == s2.length()$ (顶)的过程

递归中由顶到底的过程：

我们的递归始于*i* == *s1.length()*和*j* == *s2.length()*时, *i* == *s1.length()*(顶) --> *i* == 0(底), *j* == *s2.length()*(顶) --> *j* == 0(底), 然后在到底的时候触碰到base condition开启return返回过程

递归中再由底回到顶的过程:

在从顶到底并触碰到base condition开启return之后, 逐层返回, *i* == 0(底) --> *i* == *s1.length()*(顶), *j* == 0(底) --> *j* == *s2.length()*(顶), 此时最终状态实际上在顶, 也就是*i* == *s1.length()*和*j* == *s2.length()*时取得, 和二维DP中最终状态在右下角[*s1.length()*, *s2.length()*]处获得形成一致

```
1 class Solution {
2     public int minDistance(String word1, String word2) {
3         return helper(word1, word1.length(), word2, word2.length());
4     }
5
6     public int helper(String s1, int i, String s2, int j) {
7         // Imaging s1 is empty string "" now, how many steps for s2 to become
8         // empty string "" also ? At least delete all its characters requires
9         // j steps of 'delete' operation, hence return j
10        if(i == 0) {
11            return j;
12        }
13        // Imaging s2 is empty string "" now, how many steps for s1 to become
14        // empty string "" also ? At least delete all its characters requires
15        // i steps of 'delete' operation, hence return i
16        if(j == 0) {
17            return i;
18        }
19        int result = 0;
20        // If current pair of character in both string equals, no need any step,
21        // directly move on
22        if(s1.charAt(i - 1) == s2.charAt(j - 1)) {
23            result = helper(s1, i - 1, s2, j - 1);
24        } // If not equal characters, move on with 3 choices but add 1 more step required
25        else {
26            int insert_step = helper(s1, i, s2, j - 1);
27            int delete_step = helper(s1, i - 1, s2, j);
28            int replace_step = helper(s1, i - 1, s2, j - 1);
29            result = Math.min(Math.min(insert_step, delete_step), replace_step) + 1;
30        }
31    }
32 }
```



```

31         return result;
32     }
33 }

```

第二步：递归配合Memoization(正向版本):

```

1  class Solution {
2      public int minDistance(String word1, String word2) {
3          Integer[][] memo = new Integer[word1.length() + 1][word2.length() + 1];
4          return helper(word1, word1.length(), word2, word2.length(), memo);
5      }
6
7      public int helper(String s1, int i, String s2, int j, Integer[][] memo) {
8          if(memo[i][j] != null) {
9              return memo[i][j];
10         }
11         // Imaging s1 is empty string "" now, how many steps for s2 to become
12         // empty string "" also ? At least delete all its characters requires
13         // j steps of 'delete' operation, hence return j
14         if(i == 0) {
15             return j;
16         }
17         // Imaging s2 is empty string "" now, how many steps for s1 to become
18         // empty string "" also ? At least delete all its characters requires
19         // i steps of 'delete' operation, hence return i
20         if(j == 0) {
21             return i;
22         }
23         int result = 0;
24         // If current pair of character in both string equals, no need any step,
25         // directly move on
26         if(s1.charAt(i - 1) == s2.charAt(j - 1)) {
27             result = helper(s1, i - 1, s2, j - 1, memo);
28         } // If not equal characters, move on with 3 choices but add 1 more step required
29         else {
30             int insert_step = helper(s1, i, s2, j - 1, memo);
31             int delete_step = helper(s1, i - 1, s2, j, memo);

```

```

32         int replace_step = helper(s1, i - 1, s2, j - 1, memo);
33         result = Math.min(Math.min(insert_step, delete_step), replace_step) + 1;
34     }
35     return memo[i][j] = result;
36 }
37 }

```

第三步：基于递归的2D DP(正向版本):

```

1  class Solution {
2      /**
3       s1.charAt(i - 1) == s2.charAt(j - 1)
4       -> dp[i][j] = dp[i - 1][j - 1];
5
6       s1.charAt(i - 1) != s2.charAt(j - 1)
7       -> dp[i][j] = Math.min(Math.min(dp[i - 1][j], dp[i][j - 1]), dp[i - 1][j - 1]);
8
9       e.g s1 = "horse", s2 = "ros"
10
11           0 1 2 3
12       s2 ' ' r o s -> j
13       s1
14       0 ' '  0 1 2 3
15       1 h   1 1 2 3
16       2 o   2 2 1 2
17       3 r   3 2 2 2
18       4 s   4 3 3 2
19       5 e   5 4 4 3
20       -> i
21       */
22     public int minDistance(String word1, String word2) {
23         int w1_len = word1.length();
24         int w2_len = word2.length();
25         int[][] dp = new int[w1_len + 1][w2_len + 1];
26         for(int i = 0; i <= w1_len; i++) {
27             dp[i][0] = i;
28         }

```

```

29     for(int j = 0; j <= w2_len; j++) {
30         dp[0][j] = j;
31     }
32     for(int i = 1; i <= w1_len; i++) {
33         for(int j = 1; j <= w2_len; j++) {
34             if(word1.charAt(i - 1) == word2.charAt(j - 1)) {
35                 dp[i][j] = dp[i - 1][j - 1];
36             } else {
37                 dp[i][j] = Math.min(Math.min(dp[i - 1][j], dp[i][j - 1]), dp[i - 1]
[j - 1]) + 1;
38             }
39         }
40     }
41     return dp[w1_len][w2_len];
42 }
43 }

```

第四步：基于2D DP的空间优化1D DP(正向版本，基于第三步)：

优化为2 rows (相对于L115真正展现2 rows array替代2D array的本质)

```

1  class Solution {
2      /**
3       * s1.charAt(i - 1) == s2.charAt(j - 1)
4       * -> dp[i][j] = dp[i - 1][j - 1];
5
6       * s1.charAt(i - 1) != s2.charAt(j - 1)
7       * -> dp[i][j] = Math.min(Math.min(dp[i - 1][j], dp[i][j - 1]), dp[i - 1][j - 1]);
8
9       * e.g s1 = "horse", s2 = "ros"
10
11         0 1 2 3
12         s2 '' r o s -> j
13         s1
14         0 ''    0 1 2 3 -> the first row equal initial dpPrev array (not dp array)
15         1 h    [1]1 2 3 -> the first element '1' in second row equal initial dp array
16         2 o    2 2 1 2

```

```

17      3 r      3 2 2 2
18      4 s      4 3 3 2
19      5 e      5 4 4 3
20      -> i
21      */
22      public int minDistance(String word1, String word2) {
23          int w1_len = word1.length();
24          int w2_len = word2.length();
25          //int[][] dp = new int[w1_len + 1][w2_len + 1];
26          //for(int i = 0; i <= w1_len; i++) {
27              //    dp[i][0] = i;
28              //}
29          //for(int j = 0; j <= w2_len; j++) {
30              //    dp[0][j] = j;
31              //}
32          int[] dp = new int[w2_len + 1];
33          int[] dpPrev = new int[w2_len + 1];
34          for(int i = 0; i <= w2_len; i++) {
35              dpPrev[i] = i;
36          }
37          // 正着进行，word1每次增加一个字母（row维度）
38          // -> 外层循环依旧为row维度，而且dpPrev/dp在row维度的反复替换也在外层循环发生，为了维持row维度的替换，外层循环必须使用row维度
39          for(int i = 1; i <= w1_len; i++) {
40              // -> 根据上述细节叙述，因为dp推算公式中含有dp同行元素，推算dp其他元素前必须初始化dp的第一个元素(因为正向模式中是从左往右推导)
41              dp[0] = i;
42              // 正着进行，word2每次增加一个字母（column维度）
43              for(int j = 1; j <= w2_len; j++) {
44                  if(word1.charAt(i - 1) == word2.charAt(j - 1)) {
45                      //dp[i][j] = dp[i - 1][j - 1];
46                      // -> 现在由于去掉了row维度，dp[i][j]平行替换为dp[j]，dp[i - 1][j]平行替换为dpPrev[j]，dp[i][j - 1]平行替换为dp[j - 1]，dp[i - 1][j - 1]平行替换为dpPrev[j - 1]
47                      dp[j] = dpPrev[j - 1];
48                      // 当两个字母相等，两个字符串都不需要做任何变动，当前状态和上一层状态一致，
49                      // 直接跳过当前层，不需要加一步
50                      // 对应递归中的关系：result = helper(s1, i - 1, s2, j - 1)
51                  } else {
52                      //dp[i][j] = Math.min(Math.min(dp[i - 1][j], dp[i][j - 1]), dp[i - 1][j - 1]) + 1;

```

```

53         dp[j] = Math.min(Math.min(dpPrev[j], dp[j - 1]), dpPrev[j - 1]) +
1;
54     }
55 }
56     dpPrev = dp.clone();
57 }
58     return dpPrev[w2_len];
59 }
60 }

```

进一步优化为1 row (不是真正的1 row方案，内层循环不需要反转，因为只是用2个变量替代了2 rows 中的1 row)

```

1  class Solution {
2      /**
3       s1.charAt(i - 1) == s2.charAt(j - 1)
4       -> dp[i][j] = dp[i - 1][j - 1];
5
6       s1.charAt(i - 1) != s2.charAt(j - 1)
7       -> dp[i][j] = Math.min(Math.min(dp[i - 1][j], dp[i][j - 1]), dp[i - 1][j - 1]);
8
9
10     e.g s1 = "horse", s2 = "ros"
11
12         0 1 2 3
13     s2 ' ' r o s -> j
14     s1
15     0 ' ' 0 1 2 3 -> the first row equal initial dpPrev array (not dp array)
16     1 h  [1]1 2 3 -> the first element '1' in second row equal initial dp array
17     2 o  2 2 1 2
18     3 r  3 2 2 2
19     4 s  4 3 3 2
20     5 e  5 4 4 3
21     -> i
22     */
23     public int minDistance(String word1, String word2) {
24         int w1_len = word1.length();

```

```

25     int w2_len = word2.length();
26     //int[] dp = new int[w2_len + 1]; -> remove dp array
27     int[] dpPrev = new int[w2_len + 1];
28     for(int i = 0; i <= w2_len; i++) {
29         dpPrev[i] = i;
30     }
31     int prev = 0;
32     for(int i = 1; i <= w1_len; i++) {
33         // Special handling to store current row(dpPrev)'s first
34         // element in each round as a single variable 'prev',
35         // since it will be used in formula below to replace
36         // 'dpPrev[j - 1]' further
37         // dp[j] = Math.min(Math.min(dpPrev[j], dp[j - 1]), dpPrev[j - 1]) + 1;
38         prev = dpPrev[0];
39         // Then update current row(dpPrev)'s first element to next round
40         // row(dp)'s first element value(i) which suppose
41         // belongs to dp, but we don't create dp to represent
42         // next round row now
43         dpPrev[0] = i;
44         // No need to reverse the scanning order like L115, but why ?
45         // Because its NOT the REAL 1D array solution, its just use two
46         // variables as 'prev' to represent the necessary last element in
47         // 'dpPrev' array and 'temp' to update 'prev' each round, not
48         // fully dismiss 'dp' array, just use two variables to MOCK another
49         // 'dp' array
50         // for(int j = w2_len; j >= 0; j--) { --> Wrong Way !!!
51         for(int j = 1; j <= w2_len; j++) {
52             int temp = dpPrev[j];
53             if(word1.charAt(i - 1) != word2.charAt(j - 1)) {
54                 //dp[j] = Math.min(Math.min(dpPrev[j], dp[j - 1]), dpPrev[j - 1])
+ 1; -> 'prev' replace dpPrev[j - 1]
55                 dpPrev[j] = Math.min(Math.min(dpPrev[j], dpPrev[j - 1]), prev) + 1;
56             } else {
57                 //dp[j] = dpPrev[j - 1]; -> 'prev' replace dpPrev[j - 1]
58                 dpPrev[j] = prev;
59             }
60             prev = temp;
61         }
62         //dpPrev = dp.clone();
63     }

```

```
64         return dpPrev[w2_len];
65     }
66 }
```

Refer to

<https://leetcode.com/problems/edit-distance/solutions/25846/c-o-n-space-dp/>

```
1  class Solution {
2  public:
3      int minDistance(string word1, string word2) {
4          int m = word1.size(), n = word2.size(), pre;
5          vector<int> cur(n + 1, 0);
6          for (int j = 1; j <= n; j++) {
7              cur[j] = j;
8          }
9          for (int i = 1; i <= m; i++) {
10             pre = cur[0];
11             cur[0] = i;
12             for (int j = 1; j <= n; j++) {
13                 int temp = cur[j];
14                 if (word1[i - 1] == word2[j - 1]) {
15                     cur[j] = pre;
16                 } else {
17                     cur[j] = min(pre, min(cur[j - 1], cur[j])) + 1;
18                 }
19                 pre = temp;
20             }
21         }
22         return cur[n];
23     }
24 };
```

Refer to

<https://leetcode.com/problems/edit-distance/solutions/25895/step-by-step-explanation-of-how-to-optimize-the-solution-from-simple-recursion-to-dp/comments/562196>

I was having trouble with this and figuring out the recurrences for the edit operations. i and j basically keep track of the current characters that are getting compared, each operation shifts them differently. The important thing to note is that **we are simulating the edit operations by moving i and j around**, not actually changing the input strings.

EXAMPLE

$c1 = \text{sample}$, $c2 = \text{example}$

$i = 0$ (s), $j = 0$ (e)

Replace is simplest for me to understand, we replace the cur char with the char we need from word2. We then increment i and j to look at the next char.

Replace -> $\text{match}(c1, c2, i+1, j+1)$

$c1 = \text{eample}$, $c2 = \text{example}$

$i = 1$ (a), $j = 1$ (x)

Delete removes the first character, shifting word 1 character to left. Since we do not actually delete the char, incrementing i simulates skipping this char.

Delete -> $\text{match}(c1, c2, i+1, j)$

$c1 = \text{sample}$, $c2 = \text{example}$

$i = 1$ (a), $j = 0$ (e)

Insert is the opposite of delete, we insert the char we need, shifting word 1 to the right. Since we do not actually add a char, leave i alone. It's the similar as doing:

$c1 = \text{"e"} + c1$;

$\text{match}(c1, c2, i+1, j+1)$ //Since we added "e", $i+1$ would point to "s"

Insert -> $\text{match}(c1, c2, i, j+1)$

$c1 = \text{esample}$, $c2 = \text{example}$

$i = 0$ (s), $j = 1$ (x)

Refer to

 [L115.Distinct Subsequences](#)