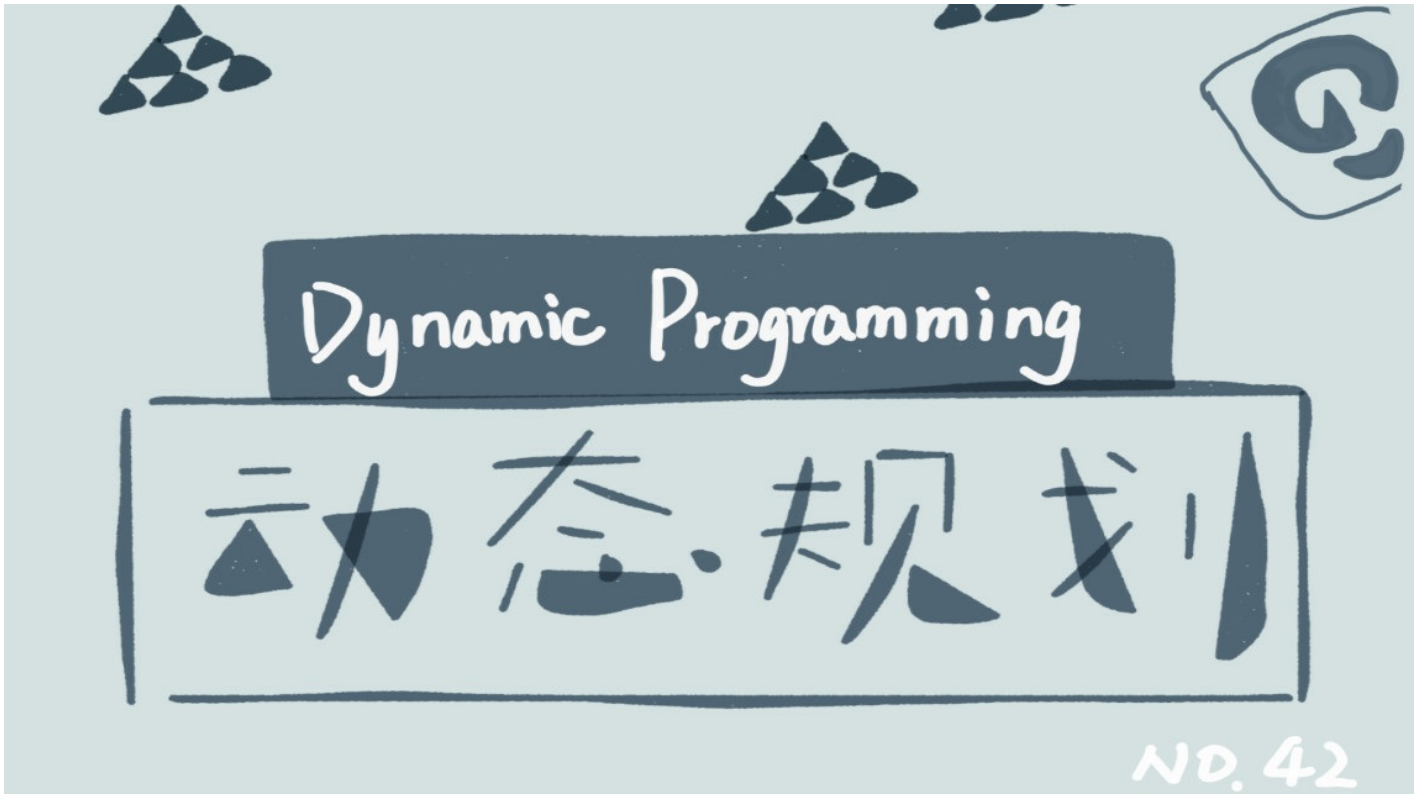
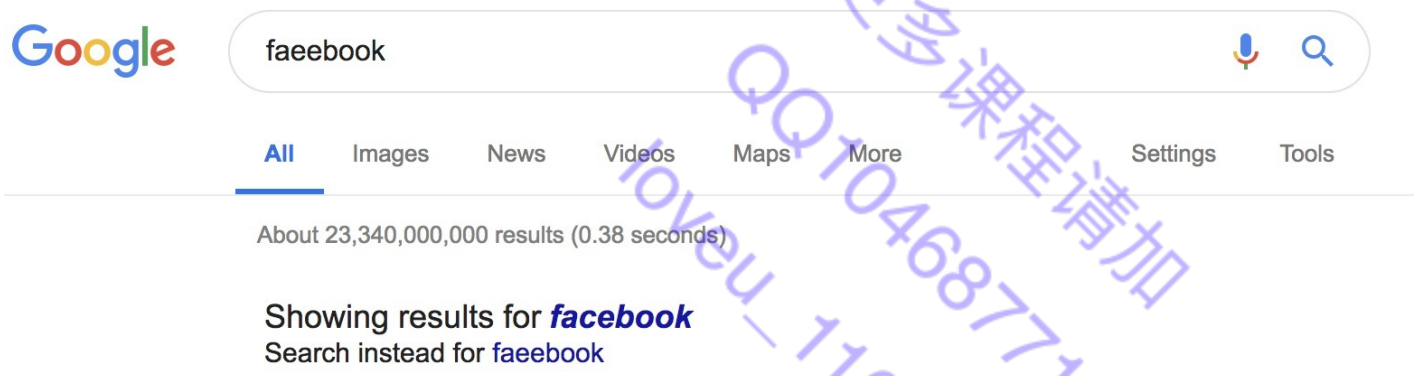
42讲动态规划实战：如何实现搜索引擎中的拼写纠错功能



在Trie树那节我们讲过，利用Trie树，可以实现搜索引擎的关键词提示功能，这样可以节省用户输入搜索关键词的时间。实际上，搜索引擎在用户体验方面的优化还有很多，比如你可能经常会用的拼写纠错功能。

当你在搜索框中，一不小心输错单词时，搜索引擎会非常智能地检测出你的拼写错误，并且用对应的正确单词来进行搜索。作为一名软件开发工程师，你是否想过，这个功能是怎么实现的呢？



如何量化两个字符串的相似度？

计算机只认识数字，所以要解答开篇的问题，我们就要先来看，如何量化两个字符串之间的相似程度呢？有一个非常著名的量化方法，那就是编辑距离（Edit Distance）。

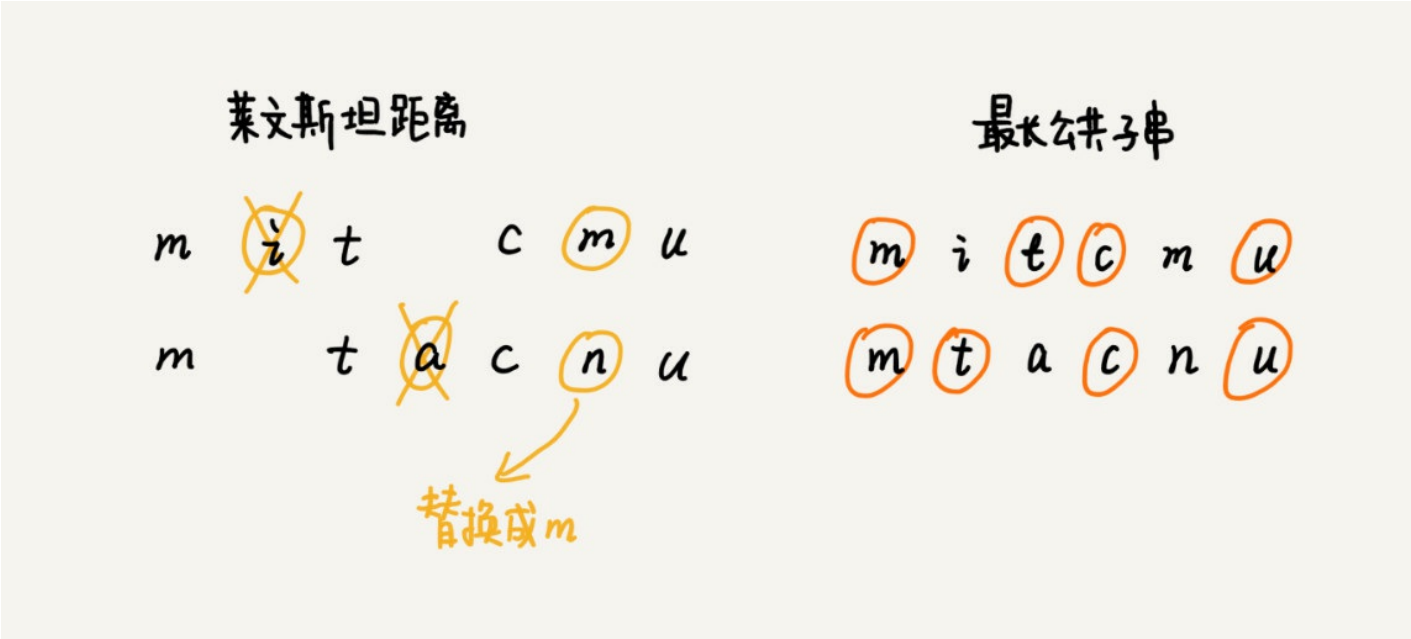
顾名思义，**编辑距离**指的就是，将一个字符串转化成另一个字符串，需要的最少编辑操作次数（比如增加一个字符、删除一个字符、替换一个字符）。编辑距离越大，说明两个字符串的相似程度越小；相反，编辑距离就越小，说明两个字符串的相似程度越大。对于两个完全相同的字符串来说，编辑距离就是0。

根据所包含的编辑操作种类的不同，编辑距离有多种不同的计算方式，比较著名的有**莱文斯坦距离**（Levenshtein distance）

和**最长公共子串长度**（Longest common substring length）。其中，莱文斯坦距离允许增加、删除、替换字符这三个编辑操作，最长公共子串长度只允许增加、删除字符这两个编辑操作。

而且，莱文斯坦距离和最长公共子串长度，从两个截然相反的角度，分析字符串的相似程度。莱文斯坦距离的大小，表示两个字符串差异的大小；而最长公共子串的大小，表示两个字符串相似程度的大小。

关于这两个计算方法，我举个例子给你说明一下。这里面，两个字符串mitcmu和mtacnu的莱文斯坦距离是3，最长公共子串长度是4。



了解了编辑距离的概念之后，我们来看，如何快速计算两个字符串之间的编辑距离？

如何编程计算莱文斯坦距离？

之前我反复强调过，思考过程比结论更重要，所以，我现在就给你展示一下，解决这个问题，我的完整的思考过程。

这个问题是求把一个字符串变成另一个字符串，需要的最少编辑次数。整个求解过程，涉及多个决策阶段，我们需要依次考察一个字符串中的每个字符，跟另一个字符串中的字符是否匹配，匹配的话如何处理，不匹配的话又如何处理。所以，这个问题符合**多阶段决策最优解模型**。

我们前面讲了，贪心、回溯、动态规划可以解决的问题，都可以抽象成这样一个模型。要解决这个问题，我们可以先看一看，用最简单的回溯算法，该如何来解决。

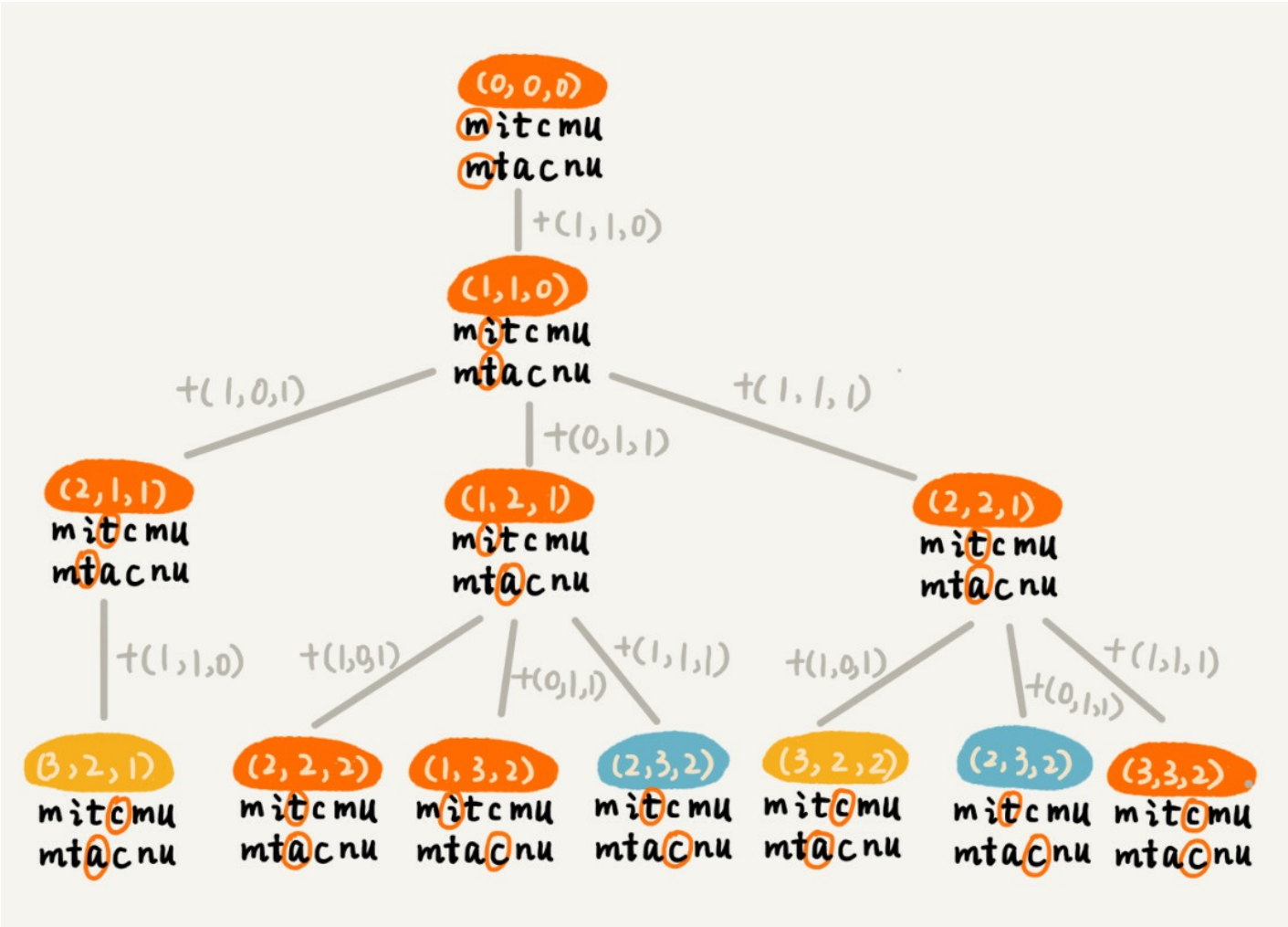
回溯是一个递归处理的过程。如果a[i]与b[j]匹配，我们递归考察a[i+1]和b[j+1]。如果a[i]与b[j]不匹配，那我们有多种处理方式可选：

- 可以删除a[i]，然后递归考察a[i+1]和b[j]；
- 可以删除b[j]，然后递归考察a[i]和b[j+1]；
- 可以在a[i]前面添加一个跟b[j]相同的字符，然后递归考察a[i]和b[j+1]；
- 可以在b[j]前面添加一个跟a[i]相同的字符，然后递归考察a[i+1]和b[j]；
- 可以将a[i]替换成b[j]，或者将b[j]替换成a[i]，然后递归考察a[i+1]和b[j+1]。

我们将上面的回溯算法的处理思路，翻译成代码，就是下面这个样子：

```
private char[] a = "mitcmu".toCharArray();
private char[] b = "mtacnu".toCharArray();
private int n = 6;
private int m = 6;
private int minDist = Integer.MAX_VALUE; // 存储结果
// 调用方式 lwstBT(0, 0, 0);
public lwstBT(int i, int j, int edist) {
  if (i == n || j == m) {
    if (i < n) edist += (n-i);
    if (j < m) edist += (m - j);
    if (edist < minDist) minDist = edist;
    return;
  }
  if (a[i] == b[j]) { // 两个字符匹配
    lwstBT(i+1, j+1, edist);
  } else { // 两个字符不匹配
    lwstBT(i + 1, j, edist + 1); // 删除a[i]或者b[j]前添加一个字符
    lwstBT(i, j + 1, edist + 1); // 删除b[j]或者a[i]前添加一个字符
    lwstBT(i + 1, j + 1, edist + 1); // 将a[i]和b[j]替换为相同字符
  }
}
```

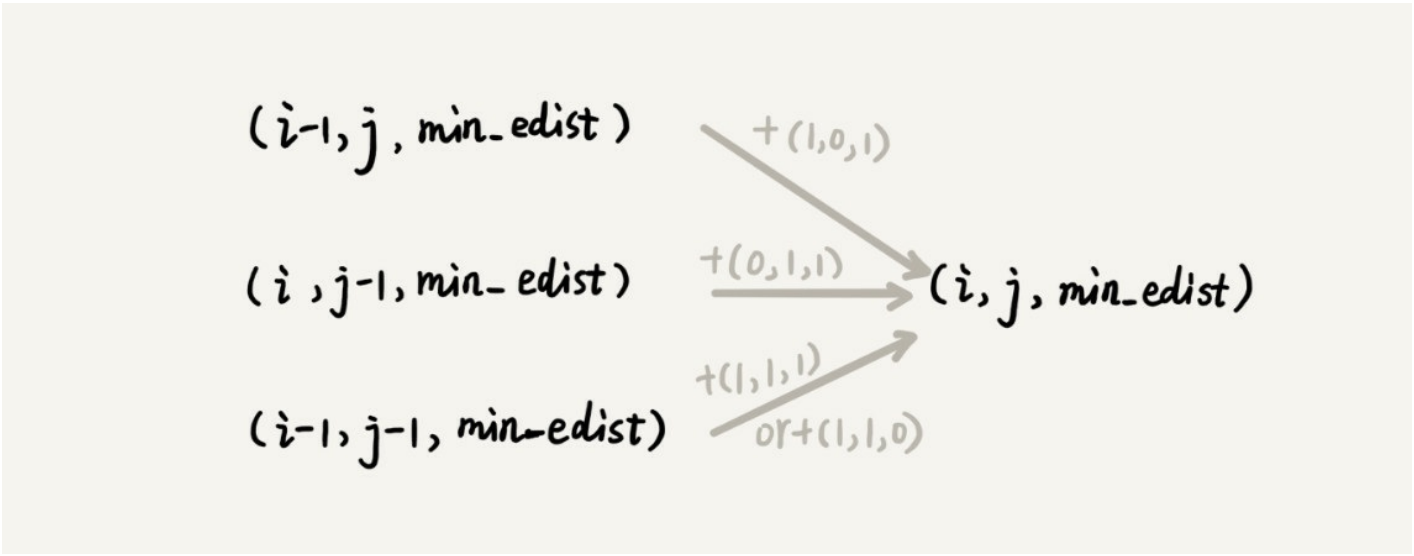
根据回溯算法的代码实现，我们可以画出递归树，看是否存在重复子问题。如果存在重复子问题，那我们就可以考虑能否用动态规划来解决；如果不存在重复子问题，那回溯就是最好的解决方法。



在递归树中，每个节点代表一个状态，状态包含三个变量(i, j, edist)，其中，edist表示处理到a[i]和b[j]时，已经执行的编辑操作的次数。

在递归树中，(i, j)两个变量重复的节点很多，比如(3, 2)和(2, 3)。对于(i, j)相同的节点，我们只需要保留edist最小的，继续递归处理就可以了，剩下的节点都可以舍弃。所以，状态就从(i, j, edist)变成了(i, j, min_edist)，其中min_edist表示处理到a[i]和b[j]，已经执行的最少编辑次数。

看到这里，你有没有觉得，这个问题跟上两节讲的动态规划例子非常相似？不过，这个问题的状态转移方式，要比之前两节课中讲到的例子都要复杂很多。上一节我们讲的矩阵最短路径问题中，到达状态(i, j)只能通过(i-1, j)或(i, j-1)两个状态转移过来，而今天这个问题，状态(i, j)可能从(i-1, j)，(i, j-1)，(i-1, j-1)三个状态中的任意一个转移过来。



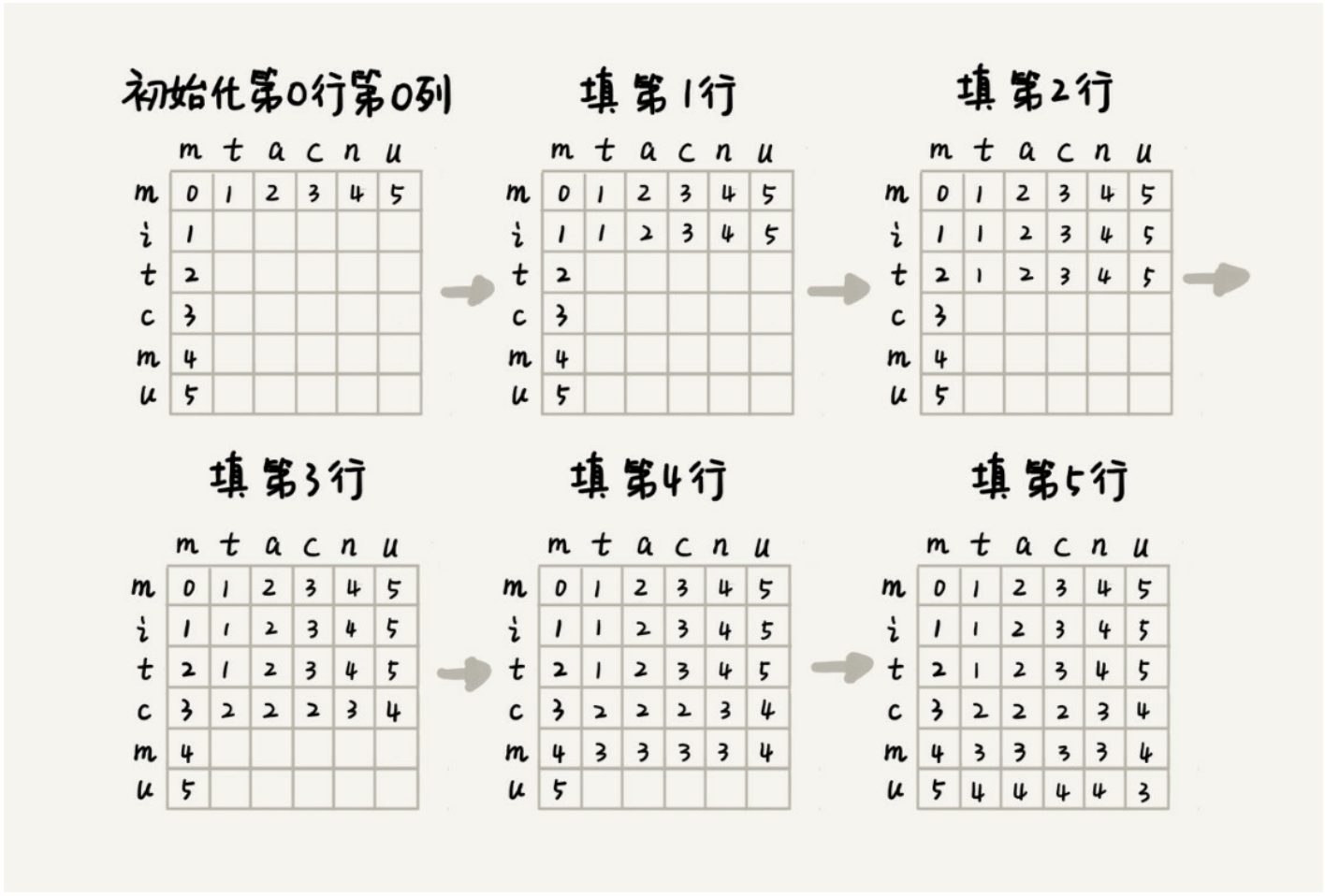
基于刚刚的分析，我们可以尝试着将把状态转移的过程，用公式写出来。这就是我们前面讲的状态转移方程。

```
如果：a[i]!=b[j]，那么：min_edist(i, j)就等于：
min(min_edist(i-1,j)+1, min_edist(i,j-1)+1, min_edist(i-1,j-1)+1)

如果：a[i]==b[j]，那么：min_edist(i, j)就等于：
min(min_edist(i-1,j)+1, min_edist(i,j-1)+1，min_edist(i-1,j-1))

其中，min表示求三数中的最小值。
```

了解了状态与状态之间的递推关系，我们画出一个二维的状态表，按行依次来填充状态表中的每个值。



我们现在既有状态转移方程，又理清了完整的填表过程，代码实现就非常简单了。我将代码贴在下面，你可以对比着文字解释，一起看下。

```
public int lwstDP(char[] a, int n, char[] b, int m) {
  int[][] minDist = new int[n][m];
  for (int j = 0; j < m; ++j) { // 初始化第0行:a[0..0]与b[0..j]的编辑距离
    if (a[0] == b[j]) minDist[0][j] = j;
    else if (j != 0) minDist[0][j] = minDist[0][j-1]+1;
    else minDist[0][j] = 1;
  }
  for (int i = 0; i < n; ++i) { // 初始化第0列:a[0..i]与b[0..0]的编辑距离
    if (a[i] == b[0]) minDist[i][0] = i;
    else if (i != 0) minDist[i][0] = minDist[i-1][0]+1;
    else minDist[i][0] = 1;
  }
  for (int i = 1; i < n; ++i) { // 按行填表
    for (int j = 1; j < m; ++j) {
      if (a[i] == b[j]) minDist[i][j] = min(
          minDist[i-1][j]+1, minDist[i][j-1]+1, minDist[i-1][j-1]);
      else minDist[i][j] = min(
          minDist[i-1][j]+1, minDist[i][j-1]+1, minDist[i-1][j-1]+1);
    }
  }
  return minDist[n-1][m-1];
}

private int min(int x, int y, int z) {
  int minv = Integer.MAX_VALUE;
  if (x < minv) minv = x;
  if (y < minv) minv = y;
  if (z < minv) minv = z;
  return minv;
}
```

你可能会说，我虽然能看懂你讲的思路，但是遇到新的问题的时候，我还是会感觉到无从下手。这种感觉是非常正常的。关于复杂算法问题的解决思路，我还有一些经验、小技巧，可以分享给你。

当我们拿到一个问题的时候，**我们可以先不思考，计算机会如何实现这个问题，而是单纯考虑“人脑”会如何去解决这个问题**。人脑比较倾向于思考具象化的、摸得着看得见的东西，不适合思考过于抽象的问题。所以，我们需要把抽象问题具象化。那如何具象化呢？我们可以实例化几个测试数据，通过人脑去分析具体实例的解，然后总结规律，再尝试套用学过的算法，看是否能够解决。

除此之外，我还有一个非常有效、但也算不上技巧的东西，我也反复强调过，那就是**多练**。实际上，等你做多了题目之后，自然就会有感觉，看到问题，立马就能想到能否用动态规划解决，然后直接就可以寻找最优子结构，写出动态规划方程，然后将状态转移方程翻译成代码。

如何编程计算最长公共子串长度？

前面我们讲到，最长公共子串作为编辑距离中的一种，只允许增加、删除字符两种编辑操作。从名字上，你可能觉得它看起来跟编辑距离没什么关系。实际上，从本质上来说，它表征的也是两个字符串之间的相似程度。

这个问题的解决思路，跟莱文斯坦距离的解决思路非常相似，也可以用动态规划解决。我刚刚已经详细讲解了莱文斯坦距离的动态规划解决思路，所以，针对这个问题，我直接定义状态，然后写状态转移方程。

每个状态还是包括三个变量(i, j, max_lcs)，max_lcs表示a[0…i]和b[0…j]的最长公共子串长度。那(i, j)这个状态都是由哪些状态转移过来的呢?

我们先来看回溯的处理思路。我们从a[0]和b[0]开始，依次考察两个字符串中的字符是否匹配。

- 如果a[i]与b[j]互相匹配，我们将最大公共子串长度加一，并且继续考察a[i+1]和b[j+1]。
- 如果a[i]与b[j]不匹配，最长公共子串长度不变，这个时候，有两个不同的决策路线：
- 删除a[i]，或者在b[j]前面加上一个字符a[i]，然后继续考察a[i+1]和b[j]；
- 删除b[j]，或者在a[i]前面加上一个字符b[j]，然后继续考察a[i]和b[j+1]。

反过来也就是说，如果我们要求a[0…i]和b[0…j]的最长公共长度max_lcs(i, j)，我们只有可能通过下面三个状态转移过来：

- (i-1, j-1, max_lcs)，其中max_lcs表示a[0…i-1]和b[0…j-1]的最长公共子串长度；
- (i-1, j, max_lcs)，其中max_lcs表示a[0…i-1]和b[0…j]的最长公共子串长度；
- (i, j-1, max_lcs)，其中max_lcs表示a[0…i]和b[0…j-1]的最长公共子串长度。

如果我们把这个转移过程，用状态转移方程写出来，就是下面这个样子：

```
如果：a[i]==b[j]，那么：max_lcs(i, j)就等于：
max(max_lcs(i-1,j-1)+1, max_lcs(i-1, j), max_lcs(i, j-1));

如果：a[i]!=b[j]，那么：max_lcs(i, j)就等于：
max(max_lcs(i-1,j-1), max_lcs(i-1, j), max_lcs(i, j-1));

其中max表示求三数中的最大值。
```

有了状态转移方程，代码实现就简单多了。我把代码贴到了下面，你可以对比着文字一块儿看。

```
public int lcs(char[] a, int n, char[] b, int m) {
  int[][] maxlcs = new int[n][m];
  for (int j = 0; j < m; ++j) {//初始化第0行：a[0..0]与b[0..j]的maxlcs
    if (a[0] == b[j]) maxlcs[0][j] = 1;
    else if (j != 0) maxlcs[0][j] = maxlcs[0][j-1];
    else maxlcs[0][j] = 0;
  }
  for (int i = 0; i < n; ++i) {//初始化第0列：a[0..i]与b[0..0]的maxlcs
    if (a[i] == b[0]) maxlcs[i][0] = 1;
    else if (i != 0) maxlcs[i][0] = maxlcs[i-1][0];
    else maxlcs[i][0] = 0;
  }
  for (int i = 1; i < n; ++i) { // 填表
    for (int j = 1; j < m; ++j) {
      if (a[i] == b[j]) maxlcs[i][j] = max(
          maxlcs[i-1][j], maxlcs[i][j-1], maxlcs[i-1][j-1]+1);
      else maxlcs[i][j] = max(
          maxlcs[i-1][j], maxlcs[i][j-1], maxlcs[i-1][j-1]);
    }
  }
  return maxlcs[n-1][m-1];
}

private int max(int x, int y, int z) {
  int maxv = Integer.MIN_VALUE;
  if (x > maxv) maxv = x;
  if (y > maxv) maxv = y;
  if (z > maxv) maxv = z;
  return maxv;
}
```

解答开篇

今天的内容到此就讲完了，我们来看下开篇的问题。

当用户在搜索框内，输入一个拼写错误的单词时，我们就拿这个单词跟词库中的单词一一进行比较，计算编辑距离，将编辑距离最小的单词，作为纠正之后的单词，提示给用户。

这就是拼写纠错最基本的原理。不过，真正用于商用的搜索引擎，拼写纠错功能显然不会就这么简单。一方面，单纯利用编辑距离来纠错，效果并不一定好；另一方面，词库中的数据量可能很大，搜索引擎每天要支持海量的搜索，所以对纠错的性能要求很高。

针对纠错效果不好的问题，我们有很多种优化思路，我这里介绍几种。

- 我们并不仅仅取出编辑距离最小的那个单词，而是取出编辑距离最小的TOP 10，然后根据其他参数，决策选择哪个单词作

为拼写纠错单词。比如使用搜索热门程度来决定哪个单词作为拼写纠错单词。

- 我们还可以用多种编辑距离计算方法，比如今天讲到的两种，然后分别编辑距离最小的TOP 10，然后求交集，用交集的结果，再继续优化处理。
- 我们还可以通过统计用户的搜索日志，得到最常被拼错的单词列表，以及对应的拼写正确的单词。搜索引擎在拼写纠错的时候，首先在这个最长被拼错单词列表中查找。如果一旦找到，直接返回对应的正确的单词。这样纠错的效果非常好。
- 我们还有更加高级一点的做法，引入个性化因素。针对每个用户，维护这个用户特有的搜索喜好，也就是常用的搜索关键词。当用户输入错误的单词的时候，我们首先在这个用户常用的搜索关键词中，计算编辑距离，查找编辑距离最小的单词。

针对纠错性能方面，我们也有相应的优化方式。我讲两种分治的优化思路。

- 如果纠错功能的TPS不高，我们可以部署多台机器，每台机器运行一个独立的纠错功能。当有一个纠错请求的时候，我们通过负载均衡，分配到其中一台机器，来计算编辑距离，得到纠错单词。
- 如果纠错系统的响应时间太长，也就是，每个纠错请求处理时间过长，我们可以将纠错的词库，分割到很多台机器。当有一个纠错请求的时候，我们就将这个拼写错误的单词，同时发送到这多台机器，让多台机器并行处理，分别得到编辑距离最小的单词，然后再比对合并，最终决定出一个最优的纠错单词。

真正的搜索引擎的拼写纠错优化，肯定不止我讲的这么简单，但是万变不离其宗。掌握了核心原理，就是掌握了解决问题的方法，剩下就靠你自己的灵活运用和实战操练了。

内容小结

动态规划的三节内容到此就全部讲完了，不知道你掌握得如何呢?

动态规划的理论尽管并不复杂，总结起来就是“一个模型三个特征”。但是，要想灵活应用并不简单。要想能真正理解、掌握动态规划，你只有多练习。

这三节中，加上课后思考题，总共有8个动态规划问题。这8个问题都非常经典，是我精心筛选出来的。很多动态规划问题其实都可以抽象成这几个问题模型，所以，你一定要多看几遍，多思考一下，争取真正搞懂它们。

只要弄懂了这几个问题，一般的动态规划问题，你应该都可以应付。对于动态规划这个知识点，你就算是入门了，再学习更加复杂的就会简单很多。

课后思考

我们有一个数字序列包含n个不同的数字，如何求出这个序列中的最长递增子序列长度？比如2, 9, 3, 6, 5, 1, 7这样一组数字序列，它的最长递增子序列就是2, 3, 5, 7，所以最长递增子序列的长度是4。

欢迎留言和我分享，也欢迎点击“请朋友读”，把今天的内容分享给你的好友，和他一起讨论、学习。

精选留言

ext4

Trie树和编辑距离，很多年前我去Google面试的时候都被考过。还记得Trie树是问我怎么存储美国的10位电话号码，可以最快速查找一个号码是否是空号，我答上来了；不过关于编辑距离我当时没想出来用dp。

2019-01-02 08:35

zixuan

补充一下，中文纠错很多时候是通过拼音进行的，比如 "刘得花"->"liudehua"->"刘德华". 拼音检索方法也有很多，比如可以把热门词汇的拼音字母组织成Trie树，每个热词的结尾汉字的最后一个拼音字母就是叶子，整体性能就是O(n)的，n为query的拼音总长度. 除了拼音外也有根据字形（二维文字版的编辑距离？）甚至语义等做的纠错策略。

传统搜索引擎中的查询词智能提示、纠错、同义词、近义词、同好词、相关搜索、知识图谱等系列功能统称为用户的意图识别模块。

2019-01-03 23:00

作者回复

好厉害

2019-01-04 09:38

Kudo

思考题解答：

状态转移公式：maxLen[i] = max(maxLen[j]+(1 if j<i else 0)) for any j < i

python代码：

```
def maxOrderedSeq(seq):
maxLen = [1] * len(seq) # 初始化为1

for i in range(1, len(seq)): # i从1开始
for j in range(i-1,-1,-1): # j从i-1到0
if seq[j] <= seq[i]:
maxLen[i] = maxLen[j] + 1
break # 满足则退出
if maxLen[i] == 1: # 比前面所有元素小
```

```
maxLen[i] = maxLen[i-1]

print(maxLen)

# usage
seq = [2, 9, 3, 6, 5, 1, 7]
maxOrderedSeq(seq)
```

2019-01-03 12:39

feifei

要自己思考这个问题，感觉真不容易，因为思路错误，走了不少弯路，花了5天的休息时间，终于解出来了，写出来了，感觉是容易了，但这个思考的过程，感觉自己收获不少，尝试了很多种的解法，然后都一一的否决！

这是我写的递归求解，

```
public int recursionCount4(int[] arrays, int index) {

if (index == 0) {
return 1;
}
int max = 0;
//此问题的解，递归的核心就是在之前的序列中找到最大递增子序列加1
//所以需要遍历此此之前的全部数据项
for (int i = 0; i < index; i++) {
//递归求解每项的最递增序列
int value = recursionCount4(arrays, i);
if (arrays[i] < arrays[index]) {
if (value > max) {
max = value;
}
}
}

return max + 1;
}



public void countDynamic(int[] arrays) {
int length = arrays.length;

int[] status = new int[length];

status[0] = 1;

int commMax = 0;

for (int i = 1; i < length; i++) {
int max = 0;
for (int j = 0; j < i; j++) {
if (arrays[j] < arrays[i]) {
```

```
if (status[j] > max) {
max = status[j];
}
}
}
status[i] = max + 1;

if (status[i] > commMax) {
commMax = status[i];
}
}

System.out.println("最大递增序列为 ： " + commMax);
int maxComp = commMax;
System.out.println("递增:" + Arrays.toString(status));

for (int i = length - 1; i >= 0; i--) {
if (status[i] == maxComp) {
System.out.print("-->" + arrays[i]);
maxComp = maxComp - 1;
}
}
}
```

2019-01-11 12:26

spark

我怎么感觉，分析完问题后，能不能写出代码的关联点是能不能通过前面的分析写出状态转移方程，跟着老师学的越多，越感觉自己是以前真的是个码农，根本算不上程序员。虽然我才真正做java一年，还是半路出家的，跟着老师学到好多东西，哈哈

2019-01-08 00:36

Sharry

思考题的解法还是很精妙的

递推公式:

a[0...i] 的最长子序列为: a[i] 之前所有比它小的元素中子序列长度最大的 + 1

代码实现:

```
#include<iostream>

using namespace std;

// 动态规划求 a 的最上升长子序列长度
#include<iostream>

using namespace std;

// 动态规划求 a 的最上升长子序列长度
int longestSubsequence(int *a, int n) {
// 创建一个数组, 索引 i 对应考察元素的下标, 存储 arr[0...i] 的最长上升子序列大小
int *lss_lengths = new int[n];
```

```
// 第一个元素哨兵处理
lss_lengths[0] = 1;
// 动态规划求解最长子序列
int i, j, max;
for (i = 1; i < n; i++) {
// 计算 arr[0...i] 的最长上升子序列
// 递推公式: lss_lengths[i] = max(condition: j < i && a[j] < a[i] value: lss_lengths[j] + 1)
max = 1;
for (j = 0; j < i; j++) {
if (a[i] > a[j] && lss_lengths[j] >= max) {
max = lss_lengths[j] + 1;
}
}
lss_lengths[i] = max;
}
int lss_length = lss_lengths[n - 1];
delete[]lss_lengths;
return lss_length;
}

void main() {
const int n = 7;
int arr[n] = { 2, 9, 3, 6, 5, 1, 7 };;
cout << longestSubsequence(arr, n) << endl;
getchar();
}
```

2019-01-03 21:01

SevenHe

思考题还有一个比较tricky的做法，用二分查找维护最长子序列（并不一定是目标子序列，只是与之等长），时间复杂度更低，从O(n^2)降到O(nlogn)

2019-01-02 16:02

。。。

思考题

```
public int recursionCount02(int[] arrays, int index) {
if (index == 0) {
return 1;
}
int max = 0;
for (int i = 0; i < index; i++) {
// 取到 index 到 i 的最大值
int value = recursionCount(arrays, i);
if (arrays[i] < arrays[index]) {
if (value + 1 > max) {
max = value + 1;
}
} else {
if (value > max) {
max = value;
}
}
}
```

```
return max;
}

dp
public int dynamicP(int[] array) {
int[] status = new int[array.length];

status[0] = 1;
for (int i = 1; i < array.length; i++) {
int max = 0;
for (int j = 0; j < i; j++) {
if (array[i] > array[j]) {
if (status[j] + 1 > max) {
max = status[j] + 1;
}
} else {
if (status[j] > max) {
max = status[j];
}
}
}
status[i] = max;
}
return status[status.length - 1];
}

@Test
public void test03() {
int[] array = new int[]{2, 3, 4, 6, 8, 1};
System.out.println(recursionCount02(array, array.length - 1));
System.out.println(dynamicP(array));
}
```

2019-01-13 11:05

Alexis何春光

“根据回溯算法的代码实现，我们可以画出递归树，看是否存在重复子问题。如果存在重复子问题，那我们就可以考虑能否用动态规划来解决；如果不存在重复子问题，那回溯就是最好的解决方法。”

2019-01-13 05:27

Dylan

以前遇到DP问题，都是一上来假设第i个状态已经搞定，去想下一步的状态方程，加上作者前面的回溯和递归分析，可以让整个想DP的思维更圆满～

2019-01-12 19:02

梅坊帝卿

通过思考题再次发现 找到状态转移方程是多么重要 其他基本都是套路 迭代取极值 到结束

2019-01-08 21:23

作者回复

光讲状态转移方程 新手是没法很好的理解的

2019-01-09 10:31

郭霖

思考题java版解答：

```
public int longestIncreaseSubArrayDP(int[] array) {
if (array.length < 2) return array.length;
```

```
int[] state = new int[array.length];
state[0] = 1;
for (int i = 1; i < state.length; i++) {
int max = 0;
for (int j = 0; j < i; j++) {
if (array[j] < array[i]) {
if (state[j] > max) max = state[j];
}
}
state[i] = max + 1;
}
int result = 0;
for (int i = 0; i < state.length; i++) {
if (state[i] > result) result = state[i];
}
return result;
}
```

2019-01-08 15:44

Lily

这门课程真的很赞，篇篇干货，简洁明晰；计算莱文斯坦距离的递归树的图右下角有处笔误： (3,2,3)应是(3,2,2)吧

2019-01-05 23:28

作者回复

多谢 我改下 不好意思

2019-01-07 09:50

煦暖

计算最长公共子串长度的代码中，max函数：
原文中的“int maxv = Integer.MIN_VALUE;”应该是MAX。

2019-01-05 21:56

作者回复

应该是MIN的

2019-01-07 10:00

想当上帝的司机

最长的1，3，5，7，9 五位吧 或者我理解错题目了

2019-01-05 15:14

作者回复

顺序不能变的

2019-01-07 10:03

微秒

状态方程中的条件如果：a[i]!=b[j]，那么：min_edist(i, j) 就等于，这里应该是a[i-1]!=b[j-1]吧？？？

2019-01-05 10:44

Ricky

老师，您好，您这节内容讲的很清晰透彻，我以前做动态规划问题是直接寻找状态转移方程，基本只能处理一些简单的动态规划问题，没有形成系统的解题思路，听了您这一节后，我觉得将回溯简单思路逐步转化为动规思路让我受益匪浅，但是当我试着将这套思路应用于求解最长递增子序列时却感觉回溯更麻烦，不知能否指点一二

2019-01-04 21:10

强哥爱飞飞

递归树那张图，最右侧的叶子节点的数字有问题。

2019-01-04 20:32

往事随风，顺其自然

初始化第零行没看懂代码啥意思

2019-01-03 21:59

往事随风，顺其自然

填表的那个图没看明白