动态规划实战：如何实现搜索引擎中的拼写纠错功能?

Trie树那一节讲述的是实现搜索引擎的关键词提示，但是很多时候搜索的时候，会输错单词，搜索引擎会非常智能的检测出你的拼写错误，并且用对应的正确单词来进行搜索；

如何量化两个字符的相似度

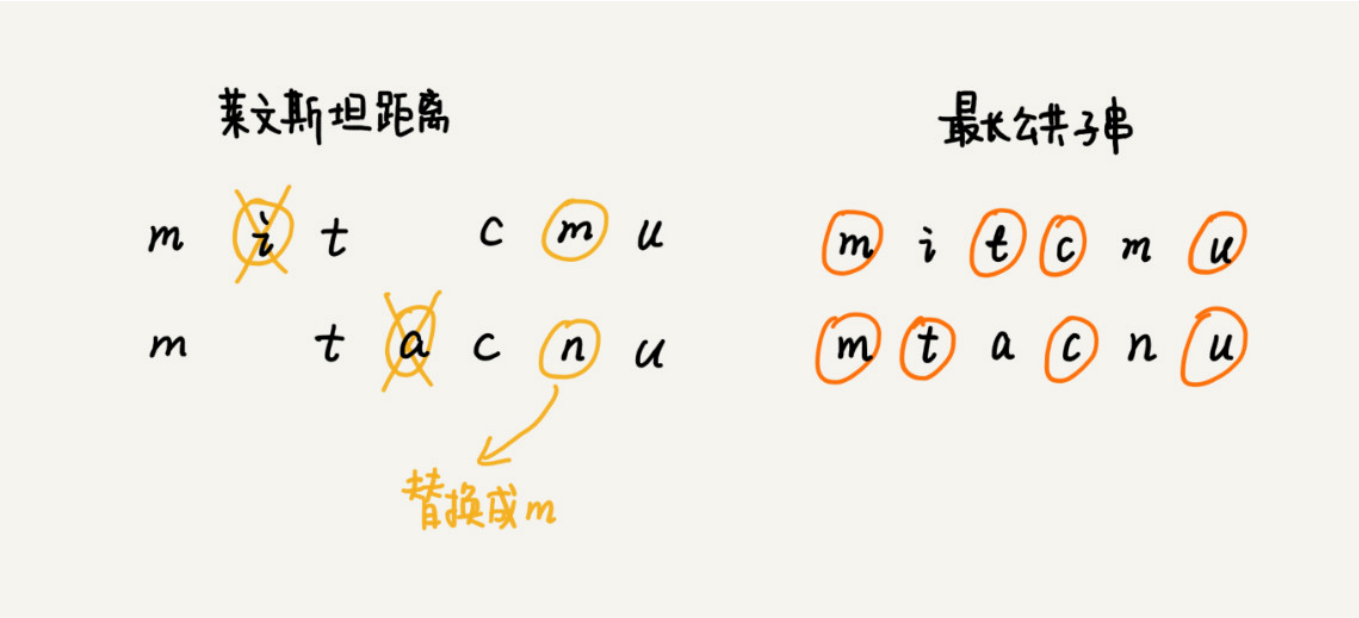
编辑距离（Edit distance）；

编辑距离就是将一个字符串转化成另一个字符串需要的最小编辑操作次数（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。

两种计算方式：**莱文斯坦距离和最长公共子串长度**，莱文斯坦距离允许增加，删除，替换字符三个操作，最长子串长度只允许增加，删除字符这两个编辑操作；

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

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。



如何编程计算莱文斯坦距离?

这个问题是把一个字符串变成另一字符串，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一个字符，跟另一个字符串中的字符是否匹配，匹配的话如何处理，不匹配的话如何处理，所以这个问题是符合多阶段决策最优模型的；

先用最简单的回溯算法，如何解决呢?

回溯是一个递归处理的过程，如果a[i]与b[j]匹配，我们考察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：

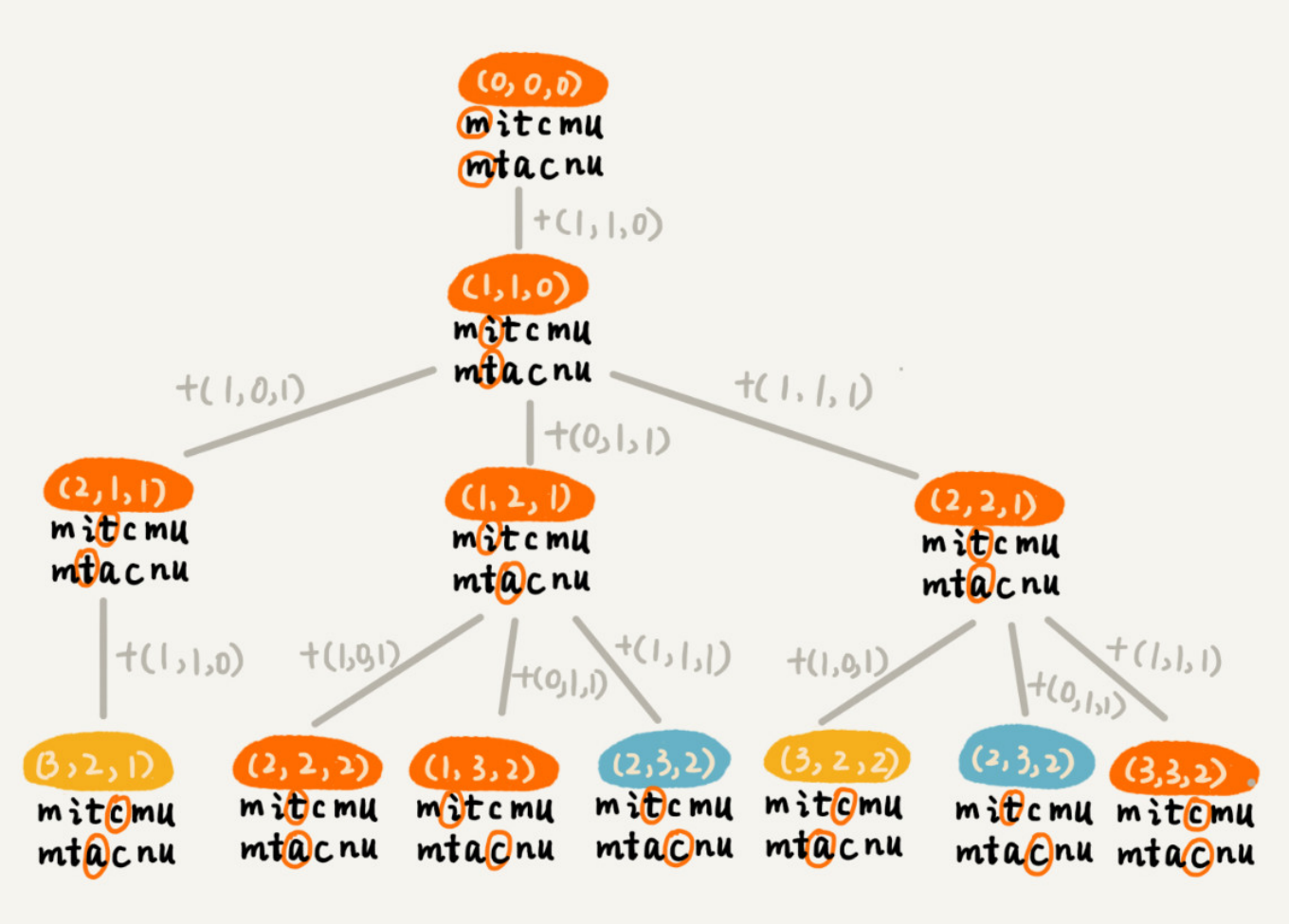
1. 可以删除a[i],然后递归考察a[i+1]和b[j];
2. 可以删除b[j],然后递归考察a[i]和b[j+1];

3. 可以在a[i]之前添加一个与b[j]相同的字符，继续考察a[i]和b[j+1];
4. 可以在b[j]之前添加一个与a[i]相同的字符，继续考察a[i+1]和b[j];
5. 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]。

所以实现的代码是：

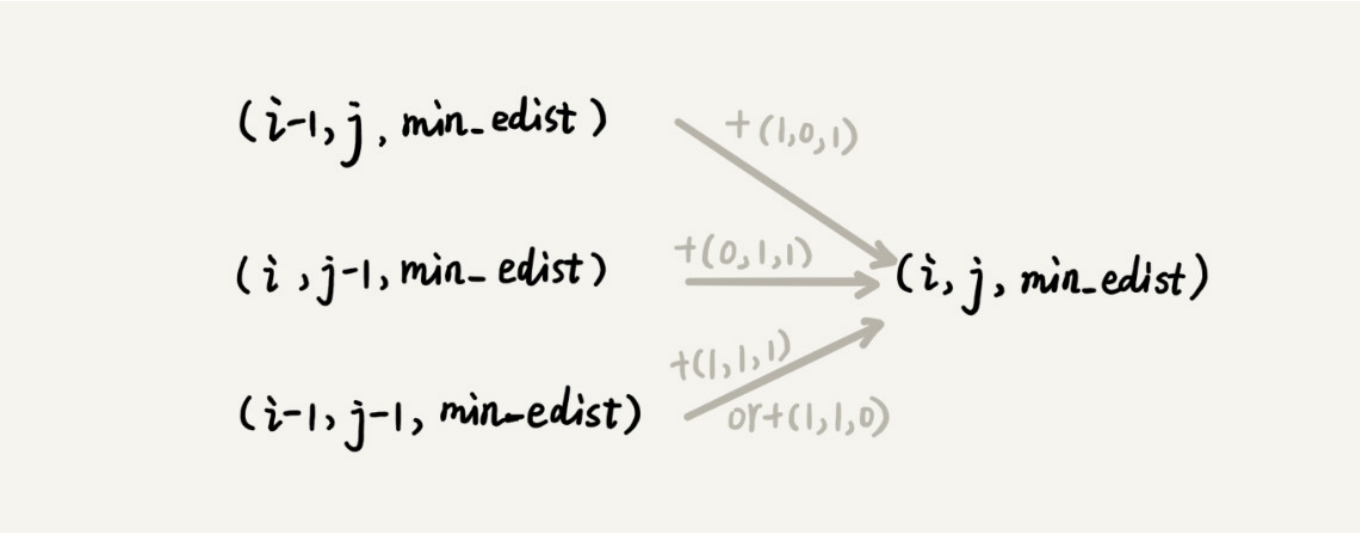
```
/**
 * 回溯算法求编辑距离
 * @param i
 * @param j
 * @param edist
 */
public void lwstBT(int i,int j,int edist){
    if(i == n || j == m){
        if(i < n) {
            edist += (n-i);
        }
        if(j < m) {
            edist += (m - j);
        }
        if(edist < minDist) {
            minDist = edist;
            return;
        }
    }
    if(a[i] == b[j]){
        lwstBT(i+1,j+i,edist);
    }else{
        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
    }
}
```

然后画出相应的递归树为：



在节点中，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。

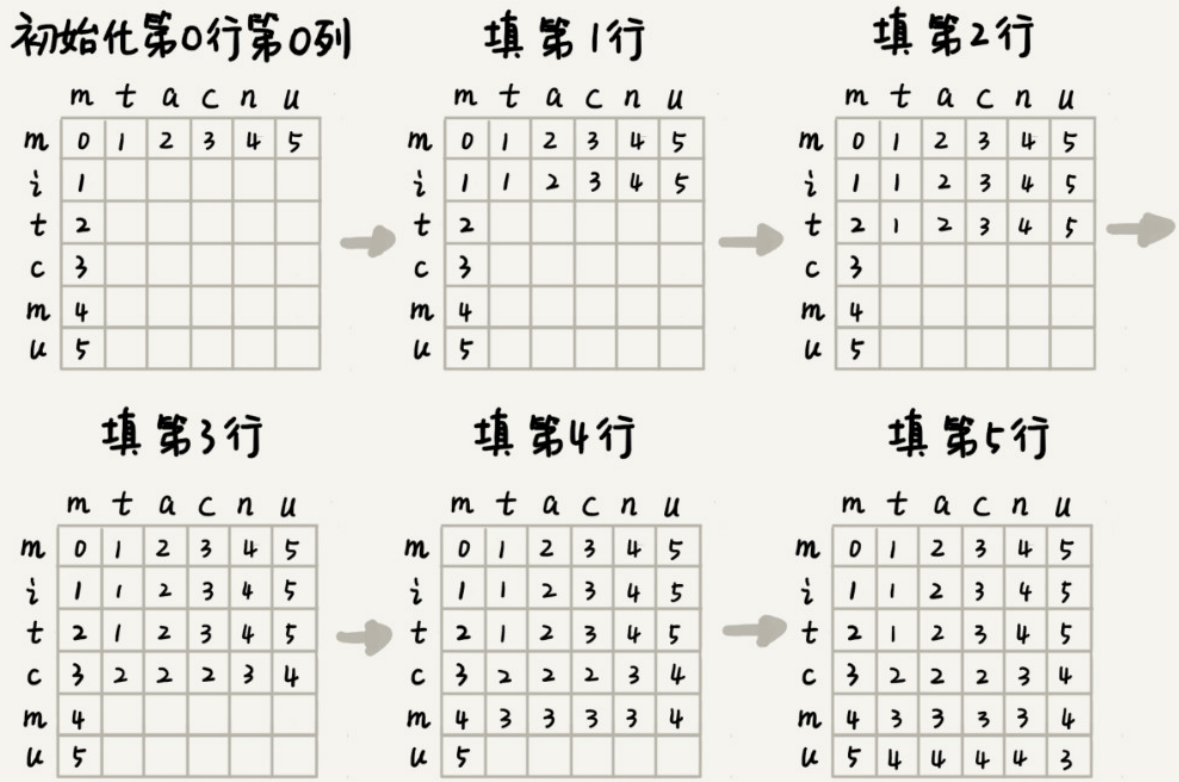
在递归树中，(i, j) 两个变量重复的节点很多，比如 (3, 2) 和 (2, 3)。对于 (i, j) 相同的节点，我们只需要保留 edist 最小的，继续递归处理就可以了，剩下的节点都可以舍弃。所以，状态就从 (i, j, edist) 变成了 (i, j, min_edist)，其中 min_edist 表示处理到 a[i]和 b[j]，已经执行的最少编辑次数。



如果a[i] != b[j],那么min_dist(i,j)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))

按照上述转移方程画出相应的转移状态图：




```
/**
 * 动态规划求编辑距离
 * @param a
 * @param n
 * @param b
 * @param m
 * @return
 */
public int lwstDP(char[] a,int n,char[] b,int m){
    int[][] minDist = new int[n][m];
    for(int j = 0; j < m; j++){
        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++){
        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++){
        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;
}
```

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**； 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。

如何编辑计算最长公共子串长度呢?

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。

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?

1. 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]。
2. 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：
3. 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]；
4. 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]。

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

1. (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；
2. (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；
3. (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。

状态转移方程是：

```
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))；
```

代码实现是：

```
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

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。

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。

1. 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。
2. 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。
3. 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，首先在这个最常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。
4. 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。

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。