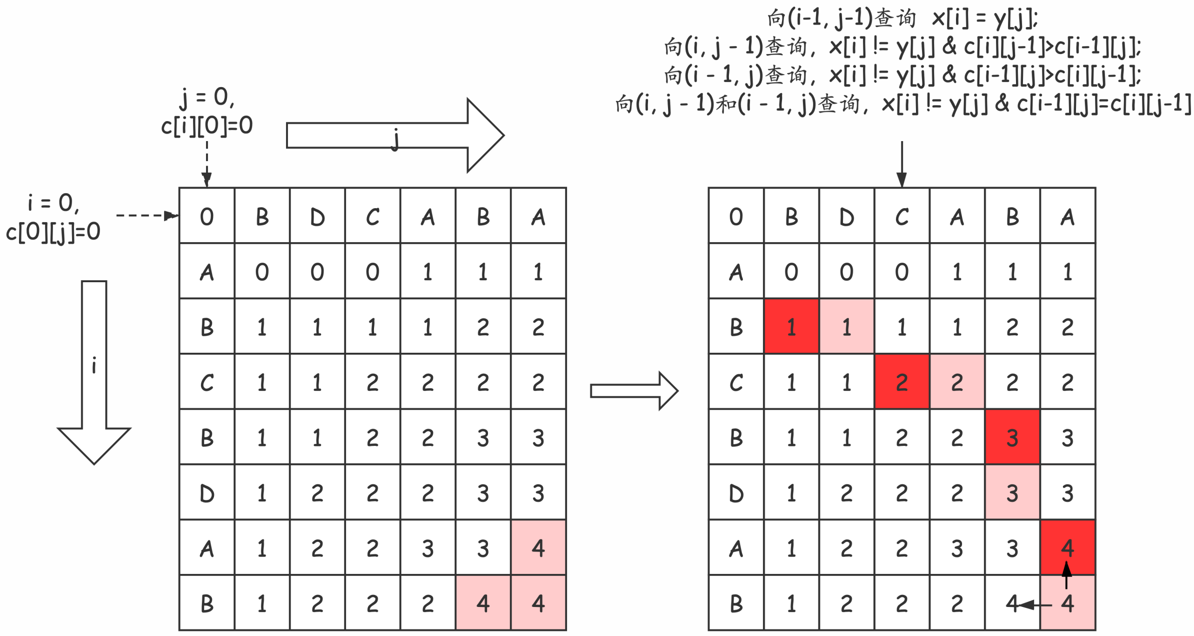
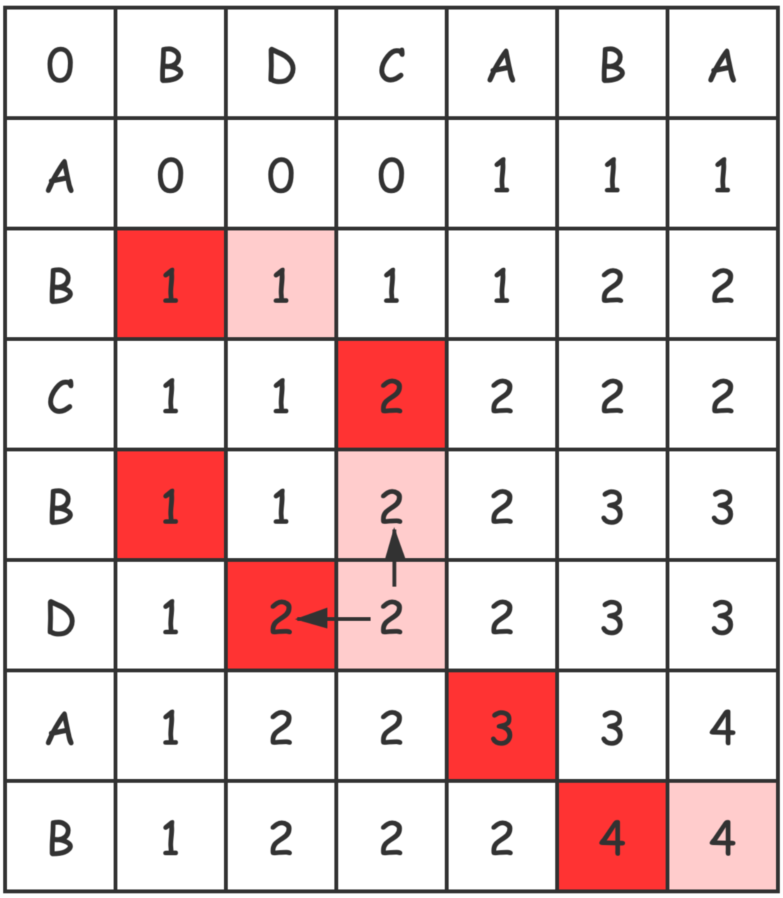
关于最长公共子序列的动态规划算法分析

Date：2021.9.29

- 示例引出：
  - 给定序列 $X = \{A, B, C, B, D, A, B\}$ 和 $Y = \{B, D, C, A, B, A\}$，要求找出它们的最长公共子序列。
- 什么是子序列？
  - 简单来说，就是从某个给定的序列中，按照从左向右的顺序提取出某些元素构成的序列。
  - 那么对于上述示例中的X来说，$\{B, C, D, B\}$就是其中的一个子序列.
  - 那么最长公共子序列就是求两个序列中最长相同的子序列，对于上述示例来说，$\{B, C, B, A\}, \{B, D, A, B\}, \{B, C, A, B\}$是两者的最长公共子序列。
- 结构分析：
  - 最长公共子序列具有最优子结构的性质：某个问题的最优解包含其子问题的最优解。
  - 这里给出递推关系：
  - 假设序列$X_n = \{x_1, x_2, ..., x_n\}$和序列$Y_m = \{y_1, y_2, ..., , y_m\}$，而两者的最长公共子序列$Z_k = \{z_1, z_2, ..., z_k\}$；
  - 那么：
    - 当$x_n = y_m$时，有$z_k = x_n = y_m$，Z_{k-1}为X_{n-1}和Y_{m-1}的最长公共子序列；
    - 当$x_n \neq y_m$且$z_k = y_m$时，Z_{k-1}为X_n和Y_{m-1}的最长公共子序列；
    - 当$x_n \neq y_m$且$z_k = x_n$时，Z_{k-1}为X_{n-1}和Y_m的最长公共子序列.
  - 可以给出递归关系：
    - 这里使用`c[i][j]`来存储序列X_i和序列Y_j的最长公共子序列，
    - $c[i][j] = \begin{cases} 0,\ \ i > 0;\ j = 0 \\ c[i-1][j-1] + 1,\ \ i, j > 0;\ x_i = y_j \\ max(c[i-1][j], c[i][j-1]),\ \ i, j > 0;\ x_i \neq y_j \end{cases}$
    - 这里给出上述示例的最大长度计算图示，左边是长度数组，右边是标记数组：
    -



- 这里我们可以看到，二维数组中最大值为4，说明我们的最长公共子序列的长度为4。
- 而我们可以记录下长度最大的坐标，并从此进行查询。
- 如上图中右边的部分，对于(7, 6)来说，由于`c[6][6] = c[7,5]`，所以需要同时向上和向下查询，之后的按照如上规则查询即可。
- 这里再给出另外一条查询线路，如下图：
-



- 这里给出实现代码如下，为了避免重复编写比较代码，这里使用了标记数组b[n+1][m+1]来指示回溯方向：

```
#include <iostream>
#include <vector>
using namespace std;

const int maxn = 100;
vector<int> temp;

/**
* @description: 获取最长公共子序列的长度
* @param {int} n: 序列X的长度
* @param {int} m: 序列Y的长度
* @param {char*} x: 序列X，索引从1开始到n结束
* @param {char*} y: 序列Y，索引从1开始到m结束
* @param {int**} c: 长度二维数组，大小为(m + 1) * (n + 1)
* @param {int**} b: 标记二维数组，大小为(m + 1) * (n + 1)
* @return {int} 最长公共子序列的长度
*/
int Longest_Common_SubSequence_Length(int n, int m, char *x, char *y, int **c,
int **b)
{
  for(int i = 0; i <= n; i++)
  {
    c[i][0] = 0;
  }
  for(int j = 0; j <= m; j++)
  {
    c[0][j] = 0;
  }
  for(int i = 1; i <= n; i++)
  {
    for(int j = 1; j <= m; j++)
    {
      if(x[i] == y[j])
      {
        c[i][j] = c[i-1][j-1] + 1;
        b[i][j] = 1;
      }
      else if(c[i][j-1] > c[i-1][j])
      {
        c[i][j] = c[i][j-1];
        b[i][j] = 2;
      }
      else if(c[i][j-1] < c[i-1][j])
      {
        c[i][j] = c[i-1][j];
        b[i][j] = 3;
      }
      else if(c[i][j-1] == c[i-1][j])
      {
```

```
        c[i][j] = c[i-1][j];
        b[i][j] = 4;
      }
    }
  }
  return c[n][m];
}


/**
 * @description: 输出最长公共子序列
 * b[i][j] = 1, 入栈，再向(i-1, j-1)查找
 * b[i][j] = 2, 向(i, j-1)查找
 * b[i][j] = 3, 向(i-1, j)查找
 * b[i][j] = 4, 向(i-1, j)和(i, j-1)同时查找
 */
void LCS(char *x, int **b, int i, int j, int longest)
{
  if(i == 0 || j == 0)
  {
    if(temp.size() == longest)
    {
      for(int i = temp.size() - 1; i >= 0; i--)
      {
        printf("%c ",x[temp[i]]);
      }
      printf("\n");
    }
    return ;
  }
  if(b[i][j] == 1)
  {
    temp.push_back(i);
    LCS(x, b, i-1, j-1, longest);
    temp.pop_back();
  }
  else if(b[i][j] == 2)
  {
    LCS(x, b, i, j-1, longest);
  }
  else if(b[i][j] == 3)
  {
    LCS(x, b, i-1, j, longest);
  }
  else
  {
    LCS(x, b, i-1, j, longest);
    LCS(x, b, i, j-1, longest);
  }
}
```

```c
int main(void)
{
  int n, m;
  int **c, **b;
  char x[maxn], y[maxn];
  scanf("%d%d", &n, &m);

  c = (int **) malloc(sizeof(int*) * (n+1));
  b = (int **) malloc(sizeof(int*) * (n+1));
  for(int i = 0; i <= n; i++)
  {
    c[i] = (int *)malloc(sizeof(int) * (m+1));
    b[i] = (int *)malloc(sizeof(int) * (m+1));
  }
  scanf("%s",(x+1));
  scanf("%s",(y+1));

  Longest_Common_SubSequence_Length(n, m, x, y, c, b);
  LCS(x, b, n, m, c[n][m]);
}
```

- 输入测试：

```
7 6
ABCBDAB
BDCABA
```

- 输出结果：

```
B C B A
B C A B
B D A B
```

- 算法复杂度分析：
  - 计算最长公共子序列长度：在`int Longest_Common_SubSequence_Length(int n, int m, char *x, char *y, int **c, int **b)`中，对于每个单元`c[i][j]`来说，计算耗费时间为$O(1)$，总计时间复杂度为$O(n*m)$。
  - 获取最长公共子序列：在`void LCS(char *x, int **b, int i, int j, int longest)`中回溯寻找某一个最大公共子序列的比较次数最大为m+n，时间复杂度为$O(n+m)$。
  - 这里我们使用了标记数组来避免重复编写比较代码，我们也可以不使用标记数组，这样可以节省一定的空间。同时，**如果我们只是为了得到最长公共子序列的长度，我们可以将空间复杂度减少到$O(minn, m)$，因为在之前的计算中可以发现，我们只需要保留数组c中的第i行和第$i-1$行的数据即可**。