DP

dynamic programming（动态规划）

Cat

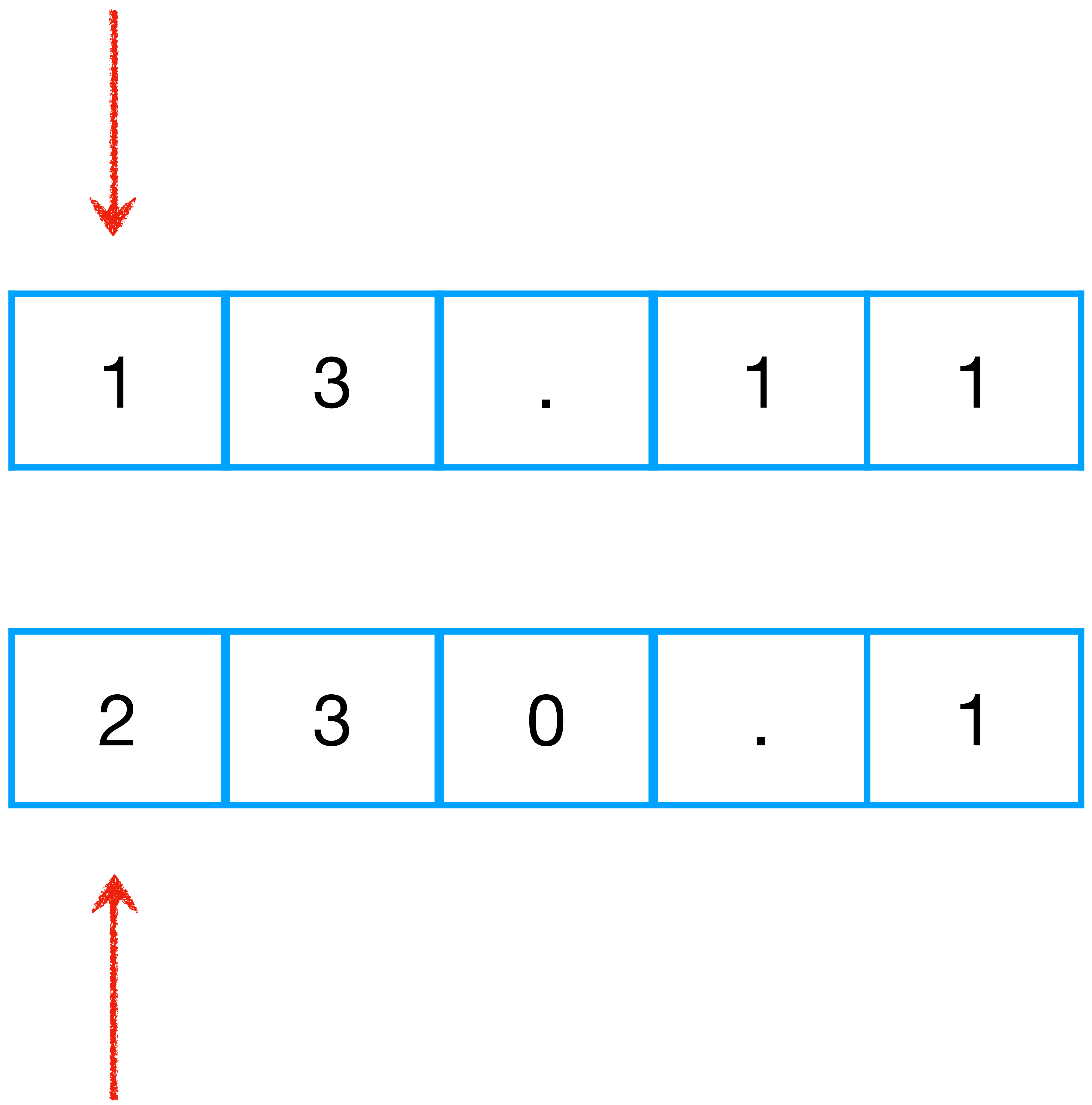
165. 比较版本号

给你两个版本号 `version1` 和 `version2` ，请你比较它们。

- 版本号由一个或多个修订号组成，各修订号由一个 '.' 连接。每个修订号由 多位数字 组成，可能包含 前导零 。每个版本号至少包含一个字符。修订号从左到右编号，下标从 0 开始，最左边的修订号下标为 0 ，下一个修订号下标为 1 ，以此类推。例如，2.5.33 和 0.1 都是有效的版本号。

- 比较版本号时，请按从左到右的顺序依次比较它们的修订号

165. 比较版本号



前四节课回顾

- 双指针
- 快慢指针
- 有特性的数据结构

DP

动态规划常常适用于有重叠子问题和最优子结构性质的问题，并且记录所有子问题的结果，因此动态规划方法所耗时间往往远少于朴素解法。

- 动态规划有两种解决问题的方式：
  - 自底向上，即递推
  - 自顶向下，即记忆化递归
- 使用动态规划解决的问题有个明显的特点：
  - 一旦一个子问题的求解得到结果，以后的计算过程就不会修改它，这样的特点叫做无后效性
  - 动态规划只解决每个子问题一次，具有天然剪枝的功能，从而减少计算量。

DP

动态规划解题思路分为两步：

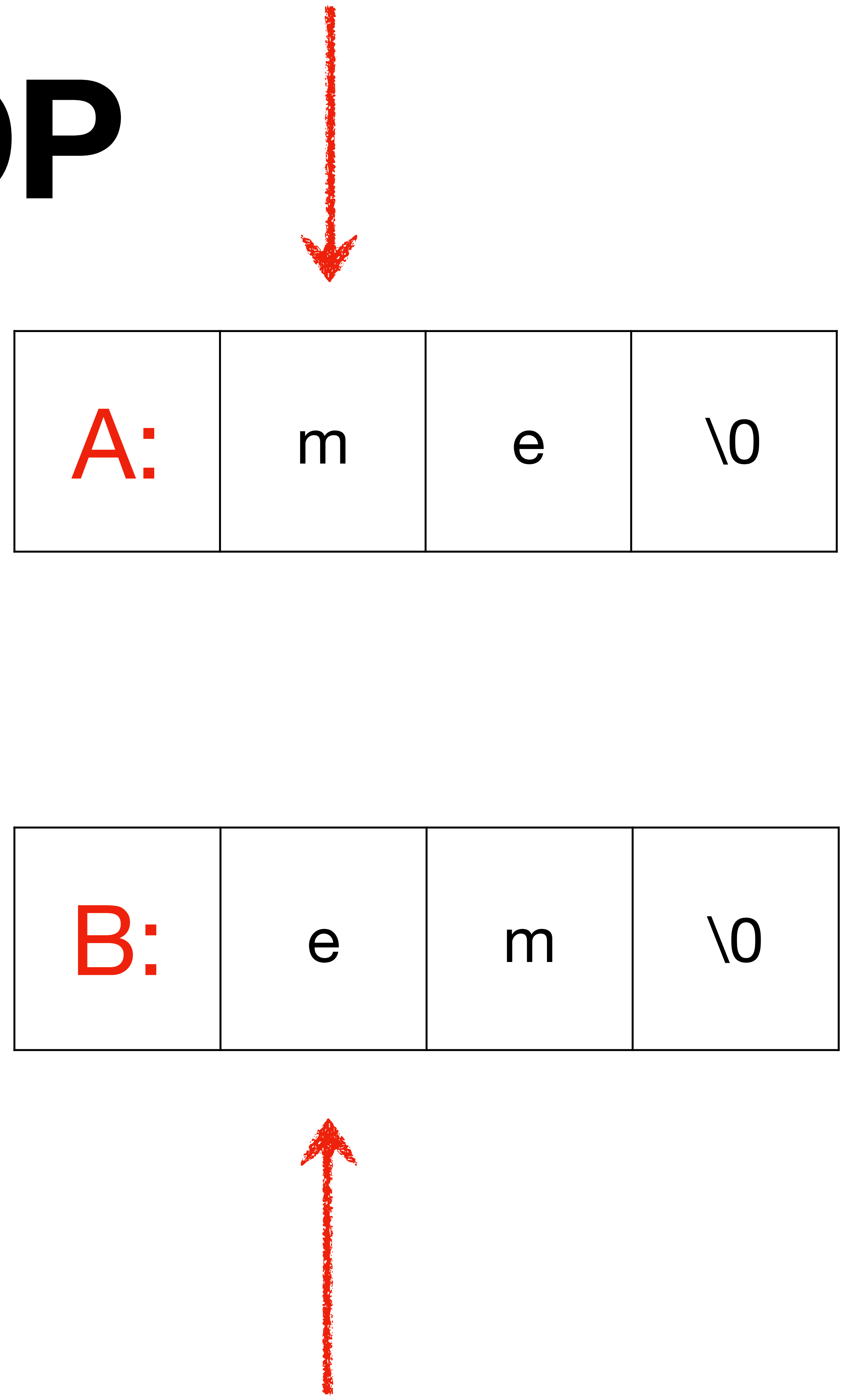
- 状态定义：定义动态规划过程中涉及的变量状态
- 转移方程：归纳和抽象出子问题对应的数学方程

1143. 最长公共子序列

给定两个字符串，求这两个字符串的最长公共子序列（LCS）。

- 子序列是指：由原字符串在不改变字符的相对顺序的情况下删除某些字符（也可以不删除任何字符）后组成的新字符串
- 如 S1={2,5,2,7,9,2,3} 和 S2={5,6,7,9,3,7} 的最长公共子序列是 {5,7,9,3}
- 应用：
    - Diff：文件比较

DP



DP

A:	a	p
B:	p	a

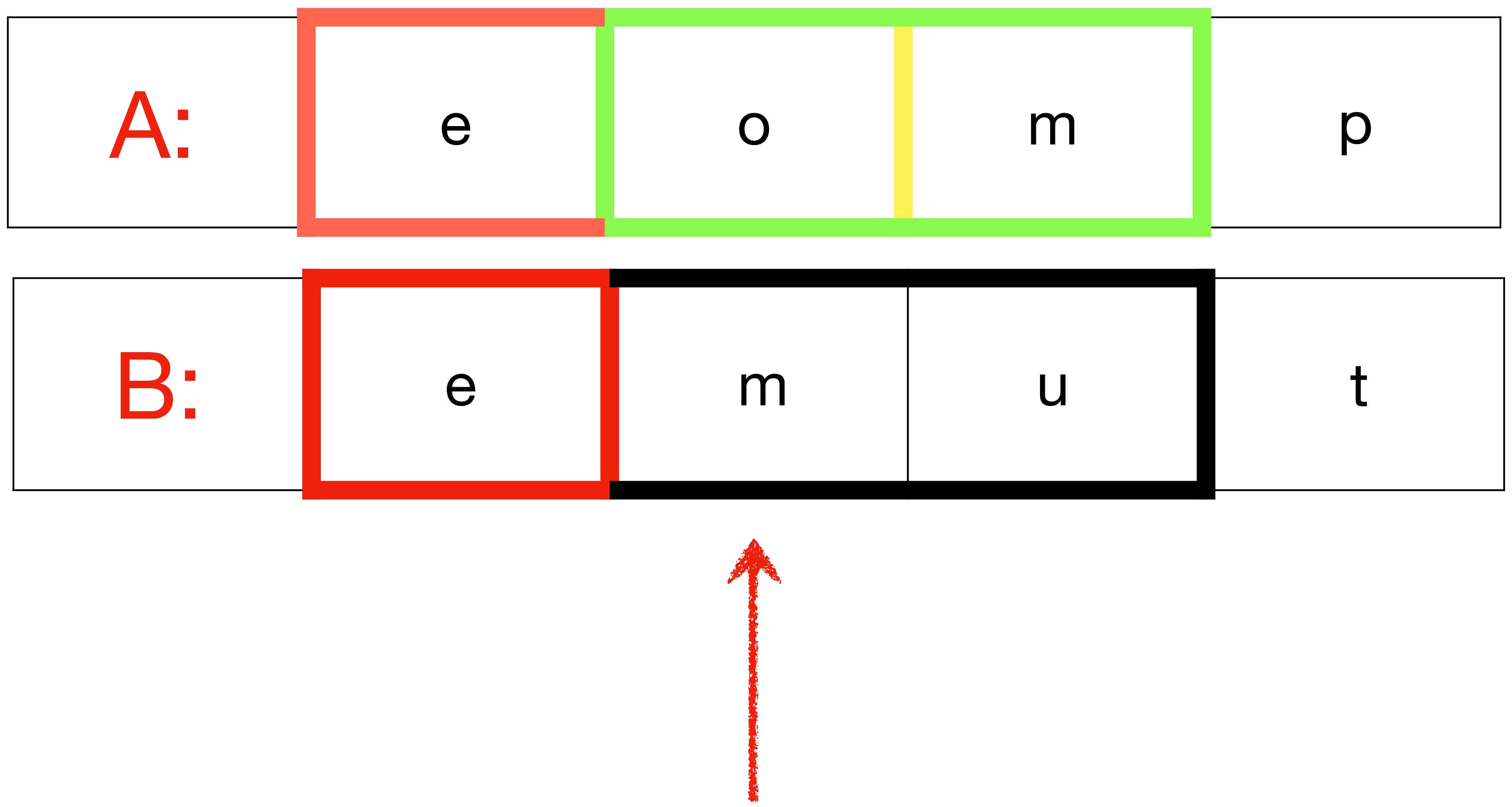
DP

A:	lcs_length	a
B:	lcs_length	a

lcs_length(A[-1],B[-1])+1

A:	a
B:	a

DP



DP

A:	e	m	a	p	o	d	e	k	n	o	w

B:	e	m	p	t	y

- 第一种情况：A[1] == B[1] length = lcs_length(A[2], B[2]) + 1
- 第二种情况：A[2] != B[2] length = max(lcs_length(A[3], B[2]), lcs_length(A[2], B[3]))

DP

A:	e	m	a	p	o	d	e	k	n	o	w

B:	e	m	p	t	y

- 第一种情况：A[1] == B[1] length = lcs_length(A[2], B[2]) + 1
- 第二种情况：A[2] != B[2] length = max(lcs_length(A[3], B[2]), lcs_length(A[2], B[3]))

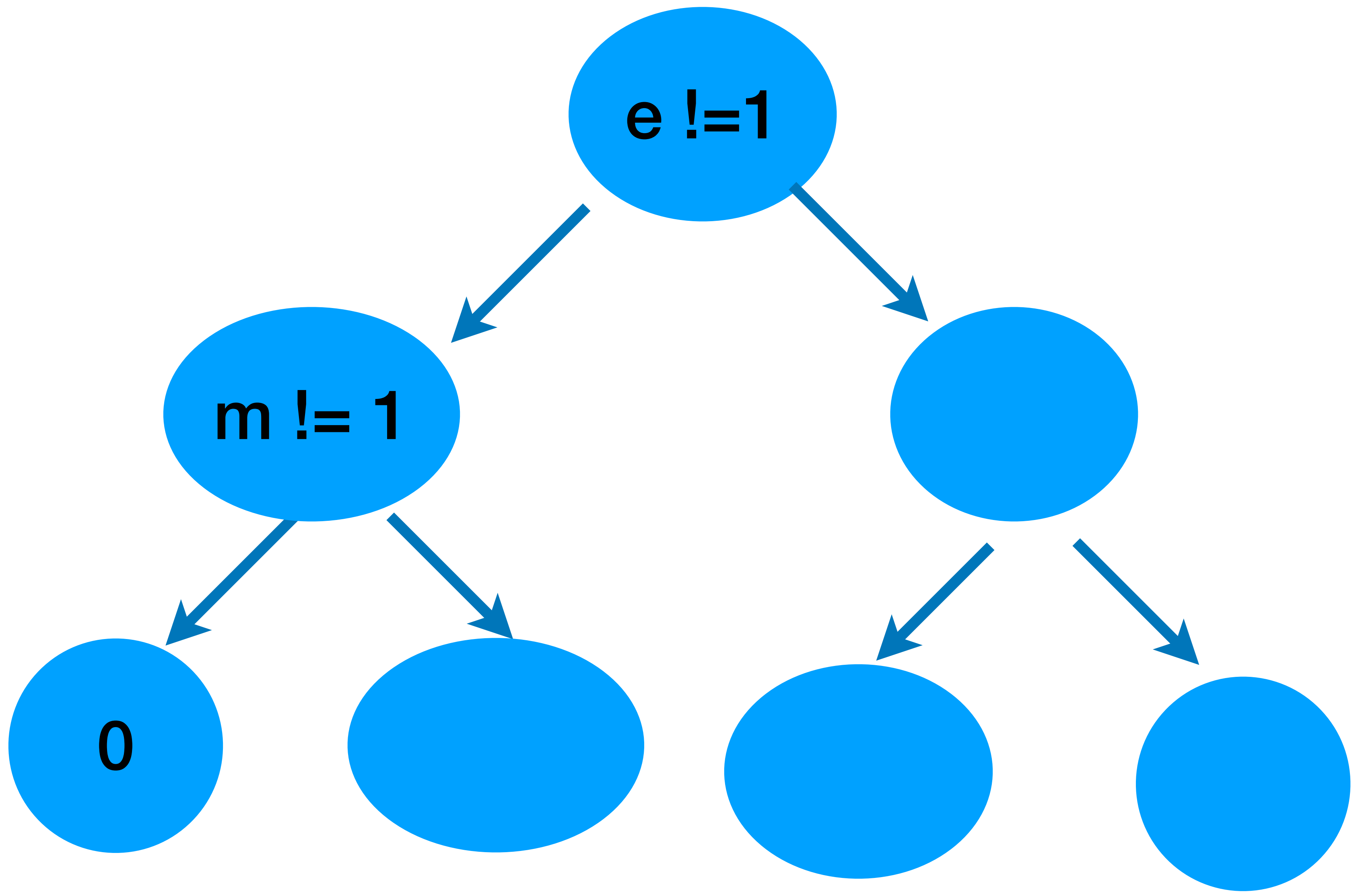
```
int lcs_length(char * A, char * B)
{
    if (*A == '\0' || *B == '\0') return 0;
    else if (*A == *B) return 1 + lcs_length(A+1, B+1);
    else return max(lcs_length(A+1,B), lcs_length(A,B+1));
}
```

if m = n，时间复杂度 = O(2^n)

DP

A:	e	m

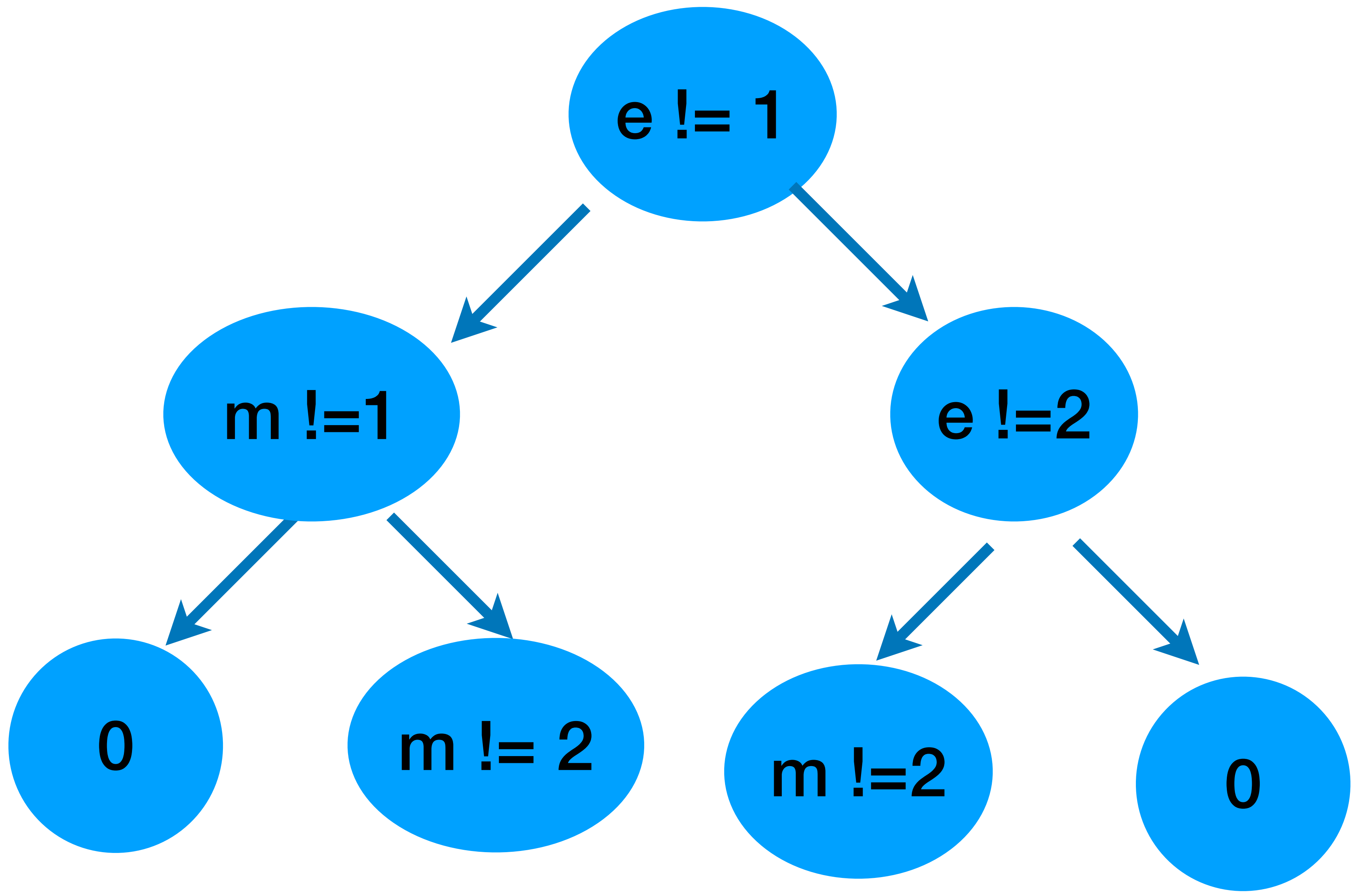
B:	1	2



DP

A:	e	m

B:	1	2



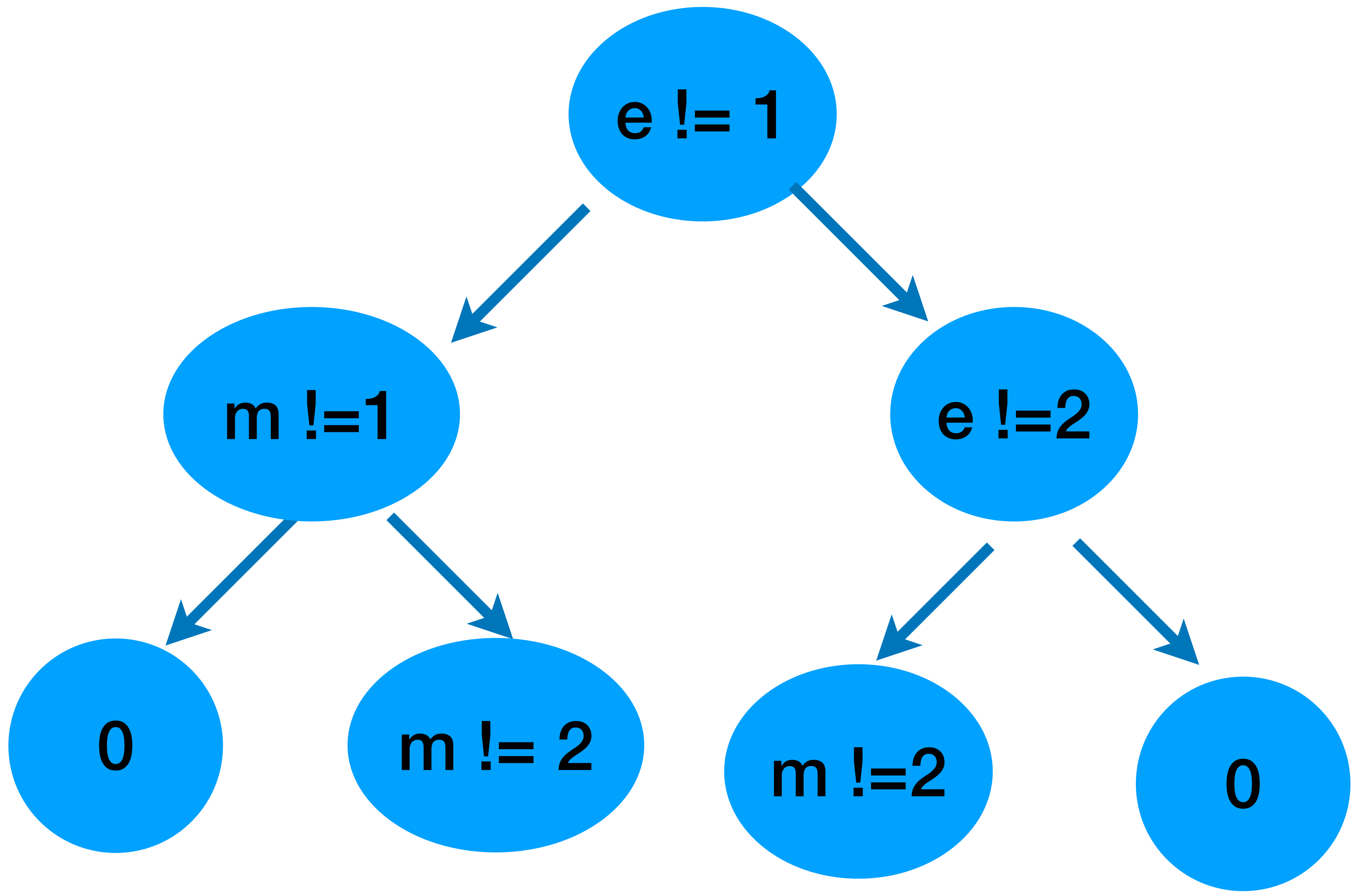

```c
int lcs_length(char * A, char * B)
{
    if (*A == '\0' || *B == '\0') return 0;
    else if (*A == *B) return 1 + lcs_length(A+1, B+1);
    else return max(lcs_length(A+1,B), lcs_length(A,B+1));
}
```

if m = n, 时间复杂度 = O(2^n)

DP

A:	e	m

B:	1	2



递归解决方案的问题是相同的子问题会被多次调用

DP ——自上而下

A:	e	m	a	p	o	d	e	k	n	o	w

B:	e	m	p	t	y

- 第一种情况：A[i] == B[j]，dp[i][j] = dp[i+1][j+1] + 1
- 第一种情况：A[i] != B[j], dp[i][j] = max(dp[i+1][j], dp[i][j+1])
- 动态规划算法，只需要使用一个数组来存储子问题结果。当想要一个子问题的解决方案时，首先查看数组，并检查那里是否已经存在解决方案。如果有，取出；否则执行计算并存储结果

DP

```
int lcs_length(char * AA, char * BB)
{
    A = AA; B = BB;
    L;
    for (i = 0; i <= m; i++)
        for (j = 0; j <= m; j++)
            L[i,j] = -1;

    return subp(0, 0);
}

int subp(int i, int j)
{
    if (L[i,j] < 0) {
        if (A[i] == '\0' || B[j] == '\0') L[i,j] = 0;
        else if (A[i] == B[j]) L[i,j] = 1 + subp(i+1, j+1);
        else L[i,j] = max(subp(i+1, j), subp(i, j+1));
    }
    return L[i,j];
}
```

DP ——自下而上

```
int lcs_length(char * A, char * B)
{
    L;
    for (i = m; i >= 0; i--)
        for (j = n; j >= 0; j--)
        {
            if (A[i] == '\0' || B[j] == '\0') L[i,j] = 0;
            else if (A[i] == B[j]) L[i,j] = 1 + L[i+1, j+1];
            else L[i,j] = max(L[i+1, j], L[i, j+1]);
        }
    return L[0,0];
}
```

DP

		0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	
		n	e	m	a	t	o	d	e	_	k	n	o	w	l	e	d	g	e	
0	e																			
1	m																			
2	p																			
3	t																			
4	y																			
5	_																			
6	b																			
7	o																			
8	t																			
9	t																			
10	l																			
11	e																			

DP

		0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	
		n	e	m	a	t	o	d	e	_	k	n	o	w	l	e	d	g	e	
0	e	7	7	6	5	5	5	5	5	4	3	3	3	2	2	2	1	1	1	0
1	m	6	6	6	5	5	4	4	4	4	3	3	3	2	2	1	1	1	1	0
2	p	5	5	5	5	5	4	4	4	4	3	3	3	2	2	1	1	1	1	0
3	t	5	5	5	5	5	4	4	4	4	3	3	3	2	2	1	1	1	1	0
4	y	4	4	4	4	4	4	4	4	4	3	3	3	2	2	1	1	1	1	0
5	_	4	4	4	4	4	4	4	4	4	3	3	3	2	2	1	1	1	1	0
6	b	3	3	3	3	3	3	3	3	3	3	3	3	2	2	1	1	1	1	0
7	o	3	3	3	3	3	3	3	3	3	3	3	3	2	2	1	1	1	1	0
8	t	3	3	3	3	3	2	2	2	2	2	2	2	2	2	1	1	1	1	0
9	t	3	3	3	3	3	2	2	2	2	2	2	2	2	2	1	1	1	1	0
10	l	2	2	2	2	2	2	2	2	2	2	2	2	2	2	1	1	1	1	0
11	e	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	0
		0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0