```
/*
接著實作遞迴函數 solver，當 dp[l1][l2][len] != -1，則代表答案已經被求得，直接回傳 dp[l1][l2][len] 的答案。

否則可以先進行兩個剪枝，isSame 用來判別兩次字串是否相等。cnt 則用來判別兩字串的字母組成是否相同，cnt[0] 代表字母 a 的數量，cnt[1] 代表字母 b 的數量…，以此類推，在字串 s1 將字母用加的加入 cnt，字串 s2 則將字母用扣

若 isSame == true 則回傳 true，並且要將答案紀錄在 dp 陣列中，因此可以簡單的寫成 return (dp[l1][l2][len] = true)。

若 cnt 中有任何一項不為 0，代表字母組成不同，則回傳 false。

最後，枚舉切點以及是否交換順序。i 代表字串 s1 = x1 + y1 或是 s1 = y1 + x1 的 x1 字串長度。

當 s1 = x1 + y1, 則 s2 = x2 + y2 的 x2 長度等於 x1 長度等於 i，而 y1, y2 長度等於 s1 長度 len 減去 i，也就是 len - i，因此若 solver(l1, l2, i) == true && solver(l1 + i, l2 + i, len - i) == true,

當 s1 = y1 + x1, 則 s2 = x2 + y2 的 x2 長度等於 y1 長度等於 len - i，而 y2, x1 長度等於 i，因此若 solver(l1 + i, l2, len - i) == true && solver(l1, l2 + len - i, i) == true，則回傳 true。

最後，若都沒有符合的，則回傳 false。
*/
#define MAX_LENGTH          (30)
#define HASH_MAP_SIZE       (1024)
#define HASH_CODE           (1024)
#define SEED     0x12345678

typedef struct node
{
    char* s1;
    int s1_len;
    char* s2;
    int s2_len;
}node_t;

struct node** hashmap;
int hashmap_length;
int dp[31][31][31];

uint32_t FNV(const void* key, uint32_t h)
{
    h ^= 2166136261UL;
    const uint8_t* data = (const uint8_t*)key;
    for(int i = 0; data[i] != '\0'; i++)
    {
        h ^= data[i];
        h *= 16777619;
    }
    return h;
}


uint32_t hash_get_key(struct node* node)
{
    uint32_t h;
    int index;

    h = SEED;
    h ^= 2166136261UL;

    for(index = 0; index < node->s1_len; index++)
    {
        h ^= node->s1[index];
        h ^= node->s2[index];
        h *= 16777619;
    }

    return h;
}

struct node* hash(char* s1, int s1_len, char* s2, int s2_len)
{
    struct node ref_node;
    uint32_t tmp_key;
    uint32_t key;

    ref_node.s1 = s1;
    ref_node.s1_len = s1_len;
    ref_node.s2 = s2;
    ref_node.s2_len = s2_len;

    key = hash_get_key(&ref_node);
    tmp_key = key;

    while(true)
    {
        if( NULL != hashmap[tmp_key%hashmap_length] )
        {
            if( hashmap[tmp_key%hashmap_length][0].s1 == s1 &&
                hashmap[tmp_key%hashmap_length][0].s2 == s2 &&
                hashmap[tmp_key%hashmap_length][0].s1_len == s1_len &&
                hashmap[tmp_key%hashmap_length][0].s2_len == s2_len)
            {
                break;
            }

        }else
        {
            break;
        }

        tmp_key++;
    }

    return  hashmap[tmp_key%hashmap_length];
}

void hash_insert(char* s1, int s1_len, char* s2, int s2_len)
{
    struct node node;
    uint32_t key;
    uint32_t tmp_key;

    node.s1 = s1;
    node.s1_len = s1_len;
    node.s2 = s2;
    node.s2_len = s2_len;
    key = hash_get_key(&node);
    tmp_key = key;

    while(true)
    {
        if( NULL == hashmap[tmp_key%hashmap_length] )
        {
            hashmap[tmp_key%hashmap_length] = (struct node*)malloc(sizeof(struct node));
            hashmap[tmp_key%hashmap_length][0].s1 = s1;
            hashmap[tmp_key%hashmap_length][0].s1_len = s1_len;
            hashmap[tmp_key%hashmap_length][0].s2 = s2;
            hashmap[tmp_key%hashmap_length][0].s2_len = s2_len;
            break;
        }else if(  hashmap[tmp_key%hashmap_length][0].s1 == s1 &&
                   hashmap[tmp_key%hashmap_length][0].s1_len == s1_len &&
                   hashmap[tmp_key%hashmap_length][0].s2 == s2 &&
                   hashmap[tmp_key%hashmap_length][0].s2_len == s2_len)
        {
             break;
        }

        //printf("collision, s1:0x%x len:%d s2:0x%x len:%d\n", s1, s1_len, s2, s2_len);
        tmp_key++;

        if(key == tmp_key)
        {
            hashmap_length += HASH_MAP_SIZE;
            hashmap = (struct node**)realloc(hashmap, sizeof(struct node*)*hashmap_length);
        }

    }
}



bool cmp(char* s1, char* s2, int len)
{
    int index;


    for(index = 0; index < len; index++)
    {
        if(s1[index] != s2[index])
        {
```

```
            return false;
        }
    }

    return true;
}


bool _isScramble(char* s1, int s1_index, char* s2, int s2_index, int len)
{
    int letters[26] = {0};
    int index;


     if( -1 != dp[s1_index][s2_index][len] )
    {
        return dp[s1_index][s2_index][len];
    }

    if ( true == cmp(&s1[s1_index], &s2[s2_index], len) )
    {
        dp[s1_index][s2_index][len] = true;
        return true;
    }


    for (index = 0; index < len; index++)
    {
        letters[s1[s1_index + index] - 'a']++;
        letters[s2[s2_index + index] - 'a']--;
    }

    for (index = 0; index < 26; index++)
    {
        if (letters[index])
        {
            dp[s1_index][s2_index][len] = false;
            return false;
        }
    }

    for (index = 1; index < len; index++)
    {
       if ( _isScramble(s1, s1_index, s2, s2_index, index) &&
            _isScramble(s1, s1_index + index, s2, s2_index + index, len-index)
          )
        {
            dp[s1_index][s2_index][len] = true;
            return true;

        }

        if( _isScramble(s1, s1_index + index, s2, s2_index, len-index) &&
            _isScramble(s1, s1_index, s2, s2_index + len - index, index)
          )
        {
             dp[s1_index][s2_index][len] = true;
             return true;
        }
    }

    dp[s1_index][s2_index][len] = false;

    return dp[s1_index][s2_index][len];

}

bool isScramble(char* s1, char* s2) {
    int s1_len;
    int s2_len;
    int index_l1;
    int index_l2;
    int index_l3;

    s1_len = strlen(s1);
    s2_len = strlen(s2);


    if (s1_len != s2_len)
    {
        return false;
    }

    for(index_l1 = 0; index_l1 < 31; index_l1++)
    {
         for(index_l2 = 0; index_l2 < 31; index_l2++)
        {
             for(index_l3 = 0; index_l3 < 31; index_l3++)
                {
                    dp[index_l1][index_l2][index_l3] = -1;
                }
        }
    }

    return  _isScramble(s1, 0, s2, 0, s1_len);

}
```