## 1、字符串所有组合

**问题描述：**  
输入一个字符串，输出该字符串中字符的所有组合。  
例如如果输入abc，它的组合有a、b、c、ab、ac、bc、abc。

字符串长度为n，子字符串长度为m

1. 首先排序，m根据从1到n循环构建，对于每个m，循环对个每个字符构建；
2. 首先往组合中加入第一个字符，再依次加入后面的字符

## 2、最长不重复子串

给定一个字符串，找出这个字符串中最长的不重复子串。

比如对于字符串“sadus”，那么返回的结果应该是“sadu”或者“adus”（返回一个即可）；

[复制代码](javascript:void(0);)

int lengthOfLongestSubstring(string s) {

vector<int> dict(256, -1);

int maxLen = 0, start = -1;

for (int i = 0; i != s.length(); i++) {

if (dict[s[i]] > start)

start = dict[s[i]];

dict[s[i]] = i;

maxLen = max(maxLen, i - start);

}

return maxLen;

}

[复制代码](javascript:void(0);)

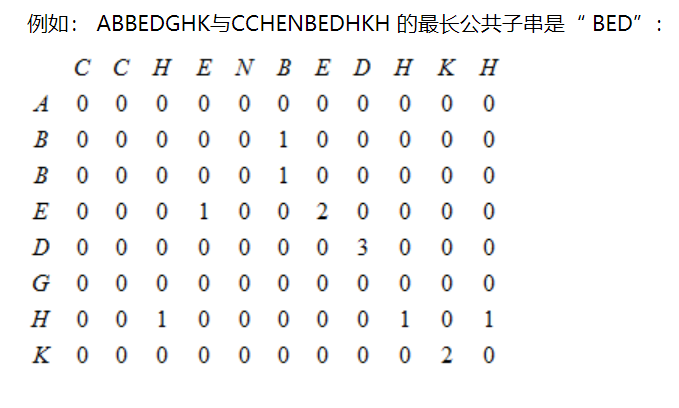
## 3、动态规划经典例题——最长公共子序列和最长公共子串

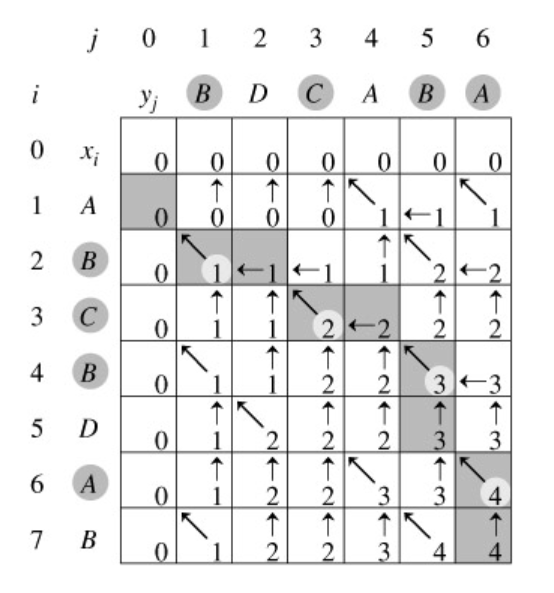
如果xi ! = yj, 那么c[i][j] = 0；

如果xi == yj， 则 c[i][j] = c[i-1][j-1]+1。

最后求Longest Common Substring的长度等于：

max{ c[i][j], 1<=i<=n， 1<=j<=m}





## 4、从字符串中删除指定字符

利用两个指针，一个读指针，一个写指针，读指针在每个循环中走一步，写指针根据当前读取的字符来决定是否前进。

如果读到的当前字符是要删除的字符，则写指针原地不动，等待下一个读操作来赋值。

[复制代码](javascript:void(0);)

static void DelInStr(char\* str, char c)

{

char \*pr = str;

char \*pw = str;

while (\*pr) {

\*pw = \*pr++;

pw += (\*pw != c);

}

\*pw = '\0';

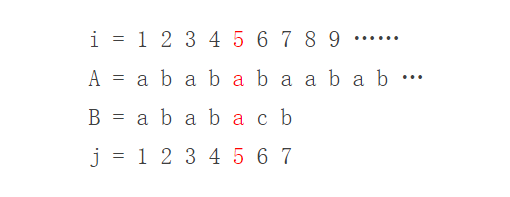
}

假如要从字符串中删除多个指定的字符，方法也是一样的，同样是判断当前读到的字符是否是待删除字符中的一个。

判断可以使用一个循环，将当前字符与所有要删除的字符依次比较，这样做效率比较慢，我们考虑用一个hash的结构来代替查找过程。

我们知道一个ASCII字符表示的范围是0～255，故我们构造一个长度为256的数组，并将其所有元素初始化为0，然后将要删除字符的ASCII码作为索引的元素置1。

## 5、KMP算法



此时，A[6]<>B[6]。这表明，此时j不能等于5了，我们要把j改成比它小的值j'。j'可能是多少呢？仔细想一下，我们发现，j'必须要使得B[1..j]中的头j'个字母和末j'个字母完全相等（这样j变成了j'后才能继续保持i和j的性质）。这个j'当然要越大越好。在这里，B [1..5]="ababa"，头3个字母和末3个字母都是"aba"。而当新的j为3时，A[6]恰好和B[4]相等。于是，i变成了6，而j则变成了 4；

针对B获取前缀字符串与后缀字符串最大匹配长度，可采用动态规划法；

6、字典序求字符串全排列**：**

**要求 123456789 这个序列的全排列时，我们可以知道这个序列的取值范围**

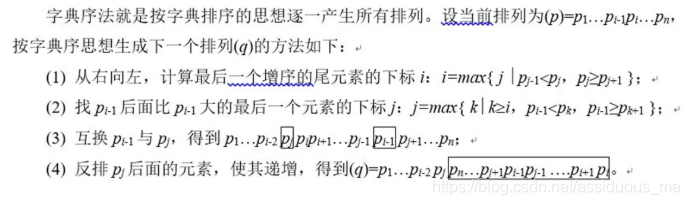
**范围：123456789-987654321**

**如何进一步在这个范围内生成我们所需要的所有序列**

**思想：让连续的两个子序列的公共前缀尽可能的长**

**比如：当生成序列为 123456978，那么下一个序列是多少呢？当然是：123456987了**  
**这样下来，就可以一个序列接着一个序列生成，最后生成所有的全排列的所有序列了**

**算法步骤** **：**



**算法案例：**

**对于 346987521 下一个排列是什么？**

**从后向前找到第一个 arr[i] < arr[i+1]，也就是递减位置，找到 6 的位置为递减的位置，因为 6 < 9 且 9 前面的都是递增的序列**

**再从后向前找到第一个 arr[j] > arr[i]，也就是比 6 大的第一个数字，找到 7，因为 7 > 6 且 7 后面的都小于 6**

**将 7 和 6 调换位置得到序列：347986521**

**再将 7 所在位置的后缀（不包含 7）反转，从而得到：347125689**

**这样就得到了 346987521 的下一个排列：347125689**

**那么什么时候是结束呢？**

**当从后向前查找发现这个序列是一个递增序列的时候就结束了（因为已经到了生成序列的最大值了）**

**注意要点：要求一个字符串的全排列时，首先要对这个字符串内部字符进行排序，从而得到生成序列的最小值**

/\*

\* Permute full array of input string by general recusion

\* @ char\* perm [in/out] The string need to do permutation

\* @ int from [in] The start position of the string

\* @ int to [in] The end position of the string

\*/

void CalcAllPermutation(char\* perm, int from, int to)

{

if (to <= 1)

{

return;

}

if (from == to)

{

//all characters has been permuted

for (int i = 0; i <= to; i++)

cout << perm[i];

cout << endl;

}

else

{

// always select one character, then full array the left ones.

for (int j = from; j <= to; j++)

{

swap(perm[j], perm[from]); //swap the selected character to the beginning of string

CalcAllPermutation(perm, from + 1, to); // Permute left characters in full array.

swap(perm[j], perm[from]); //recovery the string to original one (swap the selected character back to its position.)

}

}

}

/\*

\* Find out the next (bigger) permutation of current string.

\* @ char\* perm [in/out] String

\* @ int num [in] The length of string.

\*/

bool FindNextPermutation(char\* perm, int num)

{

int i = 0;

for(i = num - 2; (perm[i] >= perm[i+1]) && i >= 0; --i)

; // Find x

if(i < 0)

{

return false; // The input string is a single character

}

int j = 0;

for(j = num - 1; (j > i) && perm[j] <= perm[i]; --j)

; // Find y

swap(perm[i], perm[j]); // swap x and y

reverse(perm + i + 1, perm + num); // reverse B'

return true;

}

void CalcByDictionary(const string &str)

{

char\* perm = const\_cast<char\*>(str.c\_str());

sort(perm, perm+str.size());

cout<<str<<endl;

while(true)

{

if(!FindNextPermutation(perm, str.size()))

{

break;

}

cout<<s<<endl;

}

}

**7、LeetCode - 字符串转整数（atoi）**

public static int myAtoi(String str) {

*// 合法性判断*

if (str.isEmpty()) return 0;

*// 正负号标记*

int sign = 1;

*// 转换值*

int base = 0;

*// 索引位数*

int i = 0;

*// 剔除开始空白字符*

while (str.charAt(i) == ' ')

i++;

*// 判断正负号*

if (str.charAt(i) == '-' || str.charAt(i) == '+')

sign = str.charAt(i++) == '-' ? -1 : 1;

*// 索引有效数字字符*

while (i < str.length() && str.charAt(i) >= '0' && str.charAt(i) <= '9') {

*// that statement is used to test if the num is bigger than INT\_MAX after the str[i] is handled, if base > INT\_MAX/10,*

*// then base10+ str[i] -‘0’> base10>INT\_MAX, or when base== INT\_MAX/10, that means all the places are the same as INT\_MAX except the ones place, so str[i]>‘7’ is needed.*

*// 上面这段是LeetCode国外站对下面代码的解释。*

*// 简单来说就是*

*// 如果`base > MAX\_VALUE/10`，那么`base\*10 + new\_value` > `base\*10` > `MAX\_VALUE`。这个应该很容易理解，这种情况下就会发生溢出。*

*// 若`base == INT\_MAX/10`，而且`new\_value = str.charAt(i++) - '0'`大于`7`，也会发生溢出。因为`MAX\_VALUE = 2147483647`；*

if (base > Integer.MAX\_VALUE / 10 || (base == Integer.MAX\_VALUE / 10 && str.charAt(i) - '0' > 7)) { // 负值时大于 8

return (sign == 1) ? Integer.MAX\_VALUE : Integer.MIN\_VALUE;

}

*// 计算转换值*

base = 10 \* base + (str.charAt(i++) - '0');

}

*// 计算结果值*

return base \* sign;

}

**8、最长回文子串**

暴力法

中心法

动态规划：

 dp[i][j] =  dp[i+1][j-1]    &&    i和j位置的值是否相等

     注：dp[i][j]表示以i开头，以j结尾的子字符串是否是回文子串。

class Solution {

public:

    string longestPalindrome(string s) {

        int size = s.size();

        if (size <=1 )  return s;  //边界条件处理

        int len=1;  //记录回文子串的长度，回文子串的最小长度为1。

        int start=0; // 回文子串的开始位置

        int end;

         vector<vector<int>>dp (size,vector<int>(size,0));

        for(int i = 0;i<size;i++)  dp[i][i]=1;  //设置表格中对角线位置的值

        for (int j = 1; j<size;j++)  //i是开始位置，j是结束位置

           {

               for (int i=0;i<j;i++)

               {

                  if (s[i] == s[j])

                  {

                      if (j-i<3)

                      dp[i][j] = 1; //特殊情况处理

                      else

                      dp[i][j] = dp[i+1][j-1];

                  }

                 if ( dp[i][j] )   //更新长度

                   {

                       if(j-i+1 > len)

                       {

                           len = j-i+1;

                           start = i;

                       }

                   }

               }

           }

        return s.substr(start,len);

    }

};