**最长公共子串（连续）**

int getCommonStrLength(string str1, string str2){

int len1 = str1.size();

int len2 = str2.size();

//dp[i][j] --表示以str1[i]和str2[j]为结尾的最长公共子串

vector<vector<int>> dp(len1 + 1, vector<int>(len2 + 1, 0));

for (int i = 1; i <= len1; i++)

for (int j = 1; j <= len2; j++){

if (str1[i] == str2[j])

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

else

dp[i][j] = 0; //与最长公共子序列的区别

}

int max = 0;

for (int i = 0; i <= len1; i++){

for (int j = 0; j <= len2; j++){

if (max < dp[i][j])

max = dp[i][j];

}

}

return max;

}

**最长公共子序列（可以不连续）**

int getCommonStrLength(string str1, string str2){

int len1 = str1.size();

int len2 = str2.size();

//dp[i][j] --表示子串str1[0...i]和子串str[0...j]的最长公共子序列

vector<vector<int>> dp(len1 + 1, vector<int>(len2 + 1, 0));

for (int i = 1; i <= len1; i++)

for (int j = 1; j <= len2; j++){

if (str1[i] == str2[j])

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

else

dp[i][j] = max(dp[i - 1][j], dp[i][j - 1]);

}

return dp[len1][len2];

}

**字符串数组的最长公共前缀**

string longestCommonPrefix(vector<string>& strs) {

if (strs.empty())

return "";

int n = strs.size();

string maxPre = strs[0];

for (int i = 1; i < n; ++i)

{

string s = strs[i];

int j = 0;

while (j < min(maxPre.size(), s.size()) && maxPre[j] == s[j])

++j;

maxPre = maxPre.substr(0, j);

}

return maxPre;

}

**电话号码的字符组合**

vector<string> letterCombinations(string digits) {

if (digits.empty())

return{};

vector<string> res;

string charmap[10] = { "", "", "abc", "def", "ghi", "jkl", "mno", "pqrs", "tuv", "wxyz" };

res.push\_back("");

for (int i = 0; i < digits.size(); i++)

{

vector<string> tempres;

string chars = charmap[digits[i] - '0'];

for (int c = 0; c < chars.size(); c++)

for (int j = 0; j < res.size(); j++)

tempres.push\_back(res[j] + chars[c]);

res = tempres;

}

return res;

}

**合法括号对**

bool isValid(string s) {

stack<char> st;

for (char c : s)

{

switch (c) {

case '(':

case '{':

case '[': st.push(c); break;

case ')': if (st.empty() || st.top() != '(') return false; else st.pop(); break;

case '}': if (st.empty() || st.top() != '{') return false; else st.pop(); break;

case ']': if (st.empty() || st.top() != '[') return false; else st.pop(); break;

default:; // pass

}

}

return st.empty();

}

**最长合法的括号对**

int longestValidParentheses(string s) {

stack<int> st;

st.push(-1);

int maxL = 0;

for (int i = 0; i < s.size(); ++i)

{

int t = st.top();

if (t != -1 && s[t] == '(' && s[i] == ')')

{

st.pop();

maxL = max(maxL, i - st.top());

}

else

st.push(i);

}

return maxL;

}

**表达式的不同组合**

vector<int> diffWaysToCompute(string input) {

vector<int> res;

int size = input.size();

for (int i = 0; i < size; i++) {

char cur = input[i];

if (cur == '+' || cur == '-' || cur == '\*') {

vector<int> res1 = diffWaysToCompute(input.substr(0, i));

vector<int> res2 = diffWaysToCompute(input.substr(i + 1));

for (auto n1 : res1) {

for (auto n2 : res2) {

if (cur == '+')

res.push\_back(n1 + n2);

else if (cur == '-')

res.push\_back(n1 - n2);

else

res.push\_back(n1 \* n2);

}

}

}

}

if (res.empty())

res.push\_back(atoi(input.c\_str()));

return res;

}

**字符串大数相乘**

vector<int> multiply(string num1, string num2) {

int n1 = num1.size();

int n2 = num2.size();

vector<int> re(n1 + n2, 0);

reverse(num1);

reverse(num2);

for (int i = 0; i < n1; ++i)

for (int j = 0; j < n2; ++j)

re[i + j] += (num1[i] - '0')\*(num2[j] - '0');

for (int i = 0; i < n1 + n2 - 1; ++i)

{

re[i + 1] += re[i] / 10;

re[i] %= 10;

}

return re;

}

void reverse(string &s)

{

int i = 0;

int j = s.size() - 1;

while (i < j)

swap(s[i++], s[j--]);

}

**两个字符串的编辑距离（相似度）**

int minDistance(string word1, string word2) {

int m = word1.size();

int n = word2.size();

vector<vector<int>> dp(m + 1, vector<int>(n + 1));

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

dp[i][0] = i;

for (int j = 0; j <= n; j++)

dp[0][j] = j;

for (int i = 1; i <= m; i++)

{

for (int j = 1; j <= n; j++)

{

if ((word1[i - 1] == word2[j - 1]))

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

else

{

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

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

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

dp[i][j] = min(min(Ins, Del), Rep);

}

}

}

return dp[m][n];

}

**翻转句子**

“The sky is blue” 变成 “blue is sky The”。

void reverseWords(string &s) {

if (s.empty())

return;

reverse(s, 0, s.size() - 1); //先翻转整个句子

string res("");

string t;

istringstream is(s);

while (getline(is, t, ' ')) {

reverse(t, 0, t.size() - 1); //再翻转每个单词

if (t != "")

res += " ";

res += t;

}

if (!res.empty())

res = res.substr(1);

s.swap(res);

}

void reverse(string &s, int start, int end)

{

while (start < end)

swap(s[start++], s[end--]);

}

**字符串分割成所有回文子串**

class Solution {

public:

vector<vector<string>> res;

vector<string> t;

vector<vector<string>> partition(string s) {

if (s.empty())

return res;

backTrack(s);

return res;

}

void backTrack(string s)

{

if (s.empty())

{

res.push\_back(t);

return;

}

for (int i = 1; i <= s.size(); i++)

{

if (isPalindrome(s.substr(0, i)))

{

t.push\_back(s.substr(0, i));

backTrack(s.substr(i));

t.pop\_back();

}

}

}

bool isPalindrome(const string &s) {

int start = 0;

int end = s.size() - 1;

while (start <= end) {

if (s[start++] != s[end--])

return false;

}

return true;

}

};

**字符串匹配**

//Brute force 暴力搜索

int bf(const string &src, const string &pstr)

{

int i = 0, j = 0;

for (; i < src.size(); ++i)

{

if (src[i] == pstr[j])

++j;

else

{

i -= j;

j = 0;

}

if (j == pstr.size())

return i - j + 1;

}

return -1;

}

//KMP 算法

void getNext(const string &pstr, int next[])

{

next[0] = 0;//字符串的第一个字符的最大前后缀长度为0

int j = 1;

int k = 0;

for (; j < pstr.size(); ++j)//从第二个字符开始

{

while (k > 0 && pstr[j] != pstr[k])//递归的求出P[0]···P[i]的最大的相同的前后缀长度 k

k = next[k - 1];

if (pstr[j] == pstr[k])//如果相等，那么最大相同前后缀长度加1

k++;

next[j] = k;

}

}

int kmp(const string src, const string pstr, int next[])

{

int i = 0;

int k = 0;

for (; i < src.size(); ++i) //每次循环i向后移动1位

{

while (k > 0 && src[i] != pstr[k]) //如果不匹配，j 移动到开始位（j=0）或者匹配的位置

k = next[k - 1];

if (src[i] == pstr[k]) //如果匹配，则j 也向后移动1位

k++;

//找到一个就返回

if (k == pstr.size())

return i - pstr.size() + 1;

}

return -1;

}

//sunday 算法

#define CHAR\_MAX 256

int sunday(const string &src, const string &pstr)

{

int map[CHAR\_MAX];

int j;

int plen = pstr.size();

//preprocess

for (j = 0; j < CHAR\_MAX; j++)

map[j] = plen;

for (j = 0; j < plen; j++)

map[pstr[j]] = plen - j-1;

//match process

int i=0;

j = 0;

for (; i < src.size(); ++i)

{

if (src[i] == pstr[j])

++j;

else

{

i += map[src[i + plen]];

j = 0;

}

if (j == plen)

return i - j + 1;

}

return -1;

}

//Bitap算法

int bitap(const string &src, const string &pstr)

{

int plen = pstr.size();

char map[CHAR\_MAX];

memset(map, 0, CHAR\_MAX);

map[0] = 1;

for (int i = 0; i < src.size(); i++)

{

for (int j = plen - 1; j >= 0; j--)

{

map[j + 1] = map[j] & (src[i] == pstr[j]);

}

if (map[plen] == 1)

{

return i - plen + 1;

}

}

return -1;

}