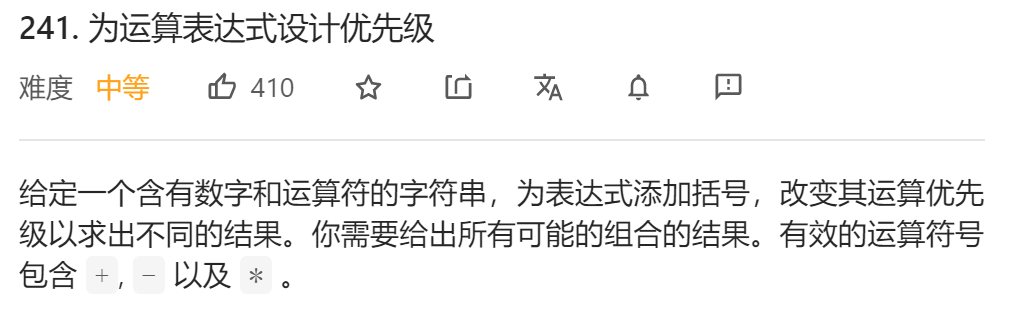
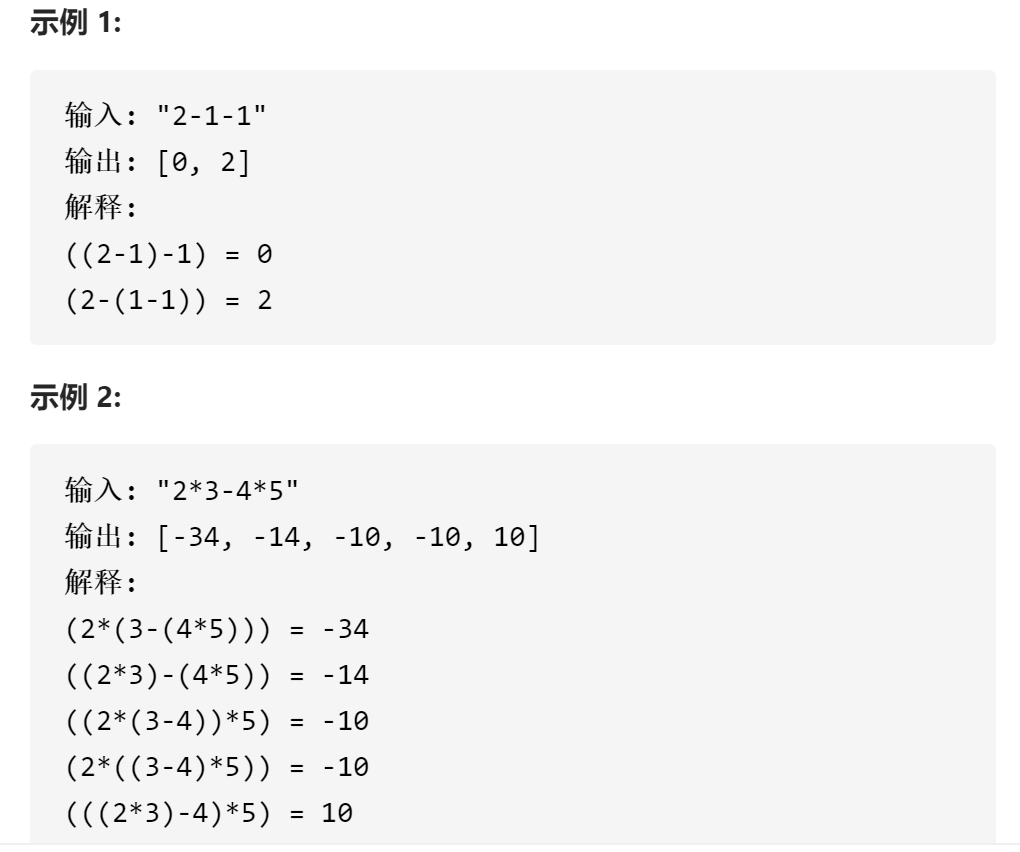
# 经典分治算法分析

## 分治思想：

1. **典型特征：一个问题可以分解成多个相似的小问题来解决；大问题的解可以由这些小问题的解来得到**
2. **解决分治类问题的两个关键：如何将一个大问题分解成相似的两个或多个小问题；如何将小问题的解合成成大问题的解**
3. **分治的算法实现一般用递归**
4. **分治算法中的一些情况可能会重复的计算；使用动态规划的话可以用空间换取一点时间**

## 问题描述：





## 问题分析：

1. **如何将该问题分解成相似的小问题？以运算符+，-，\*为界限，分割成左右两个部分，左右两个部分各为一个问题。**
2. **如何将小问题合成大问题？根据运算符+，-，\*进行合并**

## 代码分析：

class Solution {  
public:  
 vector<int> diffWaysToCompute(string expression) {  
 //分治算法  
 //将一个问题分成更小的子问题  
 vector<int>ways;  
 int n=expression.size();  
 for(int i=0;i<n;i++){  
 char c=expression[i];  
 if(c=='+'||c=='-'||c=='\*'){  
 //\*\*如何将该问题分解成相似的小问题？以运算符+，-，\*为界限，分割成左右两个部分，左右两个部分各为一个问题。\*\*  
 vector<int>left=diffWaysToCompute(expression.substr(0,i));  
 vector<int>right=diffWaysToCompute(expression.substr(i+1));  
 //\*\*如何将小问题合成大问题？根据运算符+，-，\*进行合并\*\*  
 for(const int&l:left){  
 for(const int&r:right){  
 switch(c){  
 case '+':ways.push\_back(l+r);  
 break;  
 case '-':ways.push\_back(l-r);  
 break;  
 case '\*':ways.push\_back(l\*r);  
 break;   
 }  
 }  
 }  
 }  
 }  
 if(ways.empty())ways.push\_back(stoi(expression));  
 return ways;  
 }  
};

## 动态规划，记忆化搜索优化

**我们发现，某些被 divide 的子字符串可能重复出现多次，因此我们可以用 memoization 来去重。或者与其我们从上到下用分治处理 +memoization，不如直接从下到上用动态规划处理。**

class Solution {  
public:  
 vector<int> diffWaysToCompute(string expression) {  
 //动态规划，记忆化搜索  
 //字符输入流  
 istringstream input(expression+'+');//加上一个加号是为了好操作  
 vector<int>data;//数据  
 vector<char>ops;//操作符  
 int num=0;//初始化  
 char op=' ';//初始化  
 while(input>>num&&input>>op){  
 data.push\_back(num);  
 ops.push\_back(op);  
 }  
 //动态规划  
 //dp【i】【j】表示在i,j范围内数字得到的可能的结果  
 int n=data.size();  
 vector<vector<vector<int>>>dp(n,vector<vector<int>>(n,vector<int>()));  
 for(int i=0;i<n;i++){  
 for(int j=i;j>=0;j--){  
 if(i==j){  
 //动态规划的数组初始化  
 dp[i][j].push\_back(data[i]);  
 }else{  
 for(int k=j;k<i;k++){  
 //代码功底要雄厚  
 //对于边界的处理要头脑清醒  
 for(const int&r:dp[j][k]){  
 for(const int&l:dp[k+1][i]){  
 switch(ops[k]){  
 case '+':dp[j][i].push\_back(r+l);break;  
 case '-':dp[j][i].push\_back(r-l);break;  
 case '\*':dp[j][i].push\_back(r\*l);break;  
 }  
 }  
 }  
  
 }  
 }  
 }  
 }  
 return dp[0][n-1];  
 }  
};