表达式求值

```
//表达式计算
/*
 表达式是由若干项用 + - 链接的
 项是是有由若干因子用 * / 链接的
 因子是由 表达式组成的 或者 是整数组成的
*/
#include <bits/stdc++.h>

using namespace std;
int FactorValue();//输入一个因子计算其值
```

```
int TermValue();//输入一个项计算其值
int ExpressionValue();//输入一个表达式计算其值

int main() {
    printf("%d",ExpressionValue());
    return 0;
}

int ExpressionValue() {
    int term = TermValue();
    bool more = true;//表示是否还有其他项
    while (more) {
        char op = cin.peek();
        if (op == '+' || op == '-' ) {
            cin.get();
            if (op == '+' ) {
                term += TermValue();
            } else if (op == '-') {
                term -= TermValue();
            }
        } else {
            more = false;
        }
    }
    return term;
}
int TermValue() {
    int factor = FactorValue();
    bool more = true;//表示是否还有其它因子
    while (more) {
        char op = cin.peek();//查看下一个字符是什么
        if (op == '*' || op == '/' ) {
            cin.get();
            if (op == '*') {
                factor *= TermValue();
            } else if (op == '/') {
                factor /= TermValue();
            }
        } else {
            more = false;
```

```
        }
    }
    return factor;
}
int FactorValue() {
    int result = 0;
 char ch = cin.peek();

 if (ch == '(' ) {
  cin.get();
  result = ExpressionValue();
  cin.get();
 } else {
  while(isdigit(ch)) {
   result = 10 * result + ch - '0';
   cin.get();
   ch = cin.peek();
  }
 }
    return result;
}
```

227 基本运算器II

主要通过判断前一个运算符，加减直接存到栈里，乘除从栈里取出数字运算再存回栈里。
对于有括号的，把

```
int calculate(string s) {
    long num = 0 ;
    char op = '+';
    stack<int> st;
    for (int i = 0; i < s.size(); ++i) {
        if (isdigit(s[i])) {
            num = num * 10 + s[i] - '0';
        }
        if ((!isdigit(s[i]) && s[i] != ' ') || i == s.size() - 1) {
            if (op == '+')
              st.push(num);
```

```
            else if (op == '-')
              st.push(-num);
            else if (op == '*' || op == '/') {
                int tmp = (op == '*') ? st.top() * num : st.top() /
num;
                st.pop();
                st.push(tmp);
            }
            op = s[i];
            num = 0;
        }
    }
    long res = 0;
    while (!st.empty()) {
        res += st.top();
        st.pop();
    }
    return res;
}
```

对于有括号的，把这对括号内的字符提取出来递归调用

```
 if (isdigit(s[i])) {
            num = num * 10 + s[i] - '0';
        }
else if (s[i] == '(') {
                int count = 0,start = i+1;
                for(; i < s.size(); ++i){
                    if(s[i] == '(')
                        ++count;
                    else if (s[i] == ')')
                        --count;
                    if(count == 0)
                        break;
                }
                num = calculate(s.substr(start,i-start));
            }
```

003无重复字符的最长子串（middle）

- **题目**
  给定一个字符串，请你找出其中不含有重复字符的 最长子串 的长度。
  示例 1:

  > 输入: "abcabcbb"
  > 输出: 3
  > 解释: 因为无重复字符的最长子串是 "abc"，所以其长度为 3。

- **思路**
  left保存无重复子串中的左端，max_len保存最大的无重复子串。
  使用不对关键字进行排序的unordered_map，存储当前字符，出现的位置。
  1、通过count(关键字)函数判断当前关键字有无出现过，若出现过且出现在left和i之间的子串，则把left更新为res[s[i]]出现过的位置。
  2、res[s[i]] = i;
  3、 max_len = max_len > i-left ? max_len:i-left; :其中i-left相当于局部无重复子串的长度

- **基础知识**
  创建的哈希表无序的即可，所以用ordered_map，否则用map。

- **代码实现**

```
class Solution {
public:
    int lengthOfLongestSubstring(string s) {
        int len = s.length();
        if(len < 2)
            return len;
        unordered_map<char, int> res;
        int max_len = 0;
        for(int i = 0, left = -1; i < len; ++i){
            if(res.count(s[i]) && res[s[i]] > left) //出现与之前重复的字符，且出现在left的右边（即出现在当前子串中）
                left = res[s[i]]; //更新left的值
            res[s[i]] = i;
            max_len = max_len > i-left ? max_len:i-left; //保留之前最长子串长度与当前子串长度之间的最大值
        }
```

```
        return max_len;
    }
};
//解法2
class Solution {
public:
    int lengthOfLongestSubstring(string s) {
        int len = s.length();
        if(len < 2)
            return len;
        vector<int> res(128,-1); //键盘输入的字符范围为0到128
        int max_len = 0;
        for(int i = 0, left = -1; i < len; ++i){
            left = left > res[s[i]] ? left:res[s[i]]; //left保存当前无重复字符串最左边的位置
            res[s[i]] = i;
            max_len = max_len > i-left ? max_len:i-left;
        }
        return max_len;
    }
};
```

最长回文子串（马拉车算法）

```
string longestPalindrome(string s){
    string manaStr = "$#";
    for (int i = 0; i < s.size(); ++i) {//首先构造出新的字符串
      manaStr = manaStr + s[i] + '#';
    }
    //用一个辅助数组来记录最大的回文串长度，注意这里记录的是新串的长度，原串的长度要减去1
    vector<int> rd(manaStr.size(), 0);
    int pos = 0, mx = 0;//pos为中心点坐标，mx为右边界坐标
    int start = 0, maxLen = 0;
    for (int i = 1; i < manaStr.size(); ++i) {
        if (i < mx) {
   //2*pos-i是关于pos的对称点(完全更新)，mx-i为i到边界的距离(局部更新)
            rd[i] = min(rd[2 * pos - i], mx - i);
        }
```

```
        else {
            rd[i] = 1;
        }
         //更新当前位置i的 rd[i] (对于在对称点pos范围内的则不需要再更新)
      while (i+rd[i] < manaStr.size() && i-rd[i] > 0 && manaStr[i + rd[i]] == manaStr[i - rd[i]])
          rd[i]++;
      //如果新计算的最右侧端点大于mx,则更新pos和mx
      if (i + rd[i] > mx) {
        pos = i;
        mx = i + rd[i];
      }
      if (rd[i] - 1 > maxLen){
        start = (i - rd[i]) / 2;
        maxLen = rd[i] - 1;
      }
    }
    return s.substr(start, maxLen);
  }
```

312 戳气球

301 删除无效的括号

300 最长上升子序列

```
class Solution {
public:
    int lengthOfLIS(vector<int>& a) {
        if (a.empty())
            return 0;
        vector<int> res(1,a[0]);
        for (int i = 1; i < a.size(); ++i) {
        if (a[i] > *res.rbegin())
            res.push_back(a[i]);
        else
            res[lower_bound(res.begin(), res.end(), a[i]) - res.begin()] = a[i];
        }
```

```
        return res.size();
    }
};
```

297 二叉树的序列化与反序列化

序列化是将一个数据结构或者对象转换为连续的比特位的操作，进而可以将转换后的数据存储在一个文件或者内存中，同时也可以通过网络传输到另一个计算机环境，采取相反方式重构得到原数据。

```
typedef TreeNode * T;
class Codec {
public:
    // Encodes a tree to a single string.
    string serialize(TreeNode* root) {
        if(!root)
            return "";
        string res = "";
        queue<T> q;
        q.push(root);
        while (!q.empty()) {
            T temp = q.front();
            q.pop();
            if (temp) {
                res += to_string(temp->val) + ' ';
                q.push(temp->left);
                q.push(temp->right);
            } else {
                res += "# ";
            }
        }
        return res;
    }

    // Decodes your encoded data to tree.
    TreeNode* deserialize(string data) {
        if (data.empty())
            return nullptr;
        istringstream in(data);
```

```
        queue <T> q;

        string val;
        in >> val;
        T root = new TreeNode(stoi(val)), cur = root;
        q.push (cur);
        while(!q.empty()){
            T temp = q.front();
            q.pop();
            if (!(in >> val))
                break;
            if (val != "#") {
               temp-> left = new TreeNode(stoi(val));
                q.push(temp->left);
            }
            if (!(in >> val))
                break;
            if (val != "#") {
               temp-> right = new TreeNode(stoi(val));
                q.push(temp->right);
            }
        }
        return root;
    }
};
```

寻找重复数

给定一个包含 n + 1 个整数的数组 nums，其数字都在 1 到 n 之间（包括 1 和 n），可知至少存在一个重复的整数。假设只有一个重复的整数，找出这个重复的数。

- 要求
  不能更改原数组（假设数组是只读的）。
  只能使用额外的 O(1) 的空间。
  时间复杂度小于 O(n2) 。
  数组中只有一个重复的数字，但它可能不止重复出现一次。
- 方法
  比如数组范围是0到n，那么统计0到n/2的个数count，若个数大于等于count,则说明有重复；在0到n/2区间统计0到n/4的数字

```cpp
class Solution {
public:
    int findDuplicate(vector<int>& nums) {
        int left = 0, right = nums.size();
        while (left < right){
            int mid = (left + right) >> 1;
            int count = 0;
            for (int i = 0; i < nums.size(); ++i) {
                if (nums[i] <= mid){
                    ++count;
                }
            }
            if (count > mid) {
                right = mid;
            } else {
                left = mid+1;
            }
        }
        return left;
    }
};
```

279 完全平方数

给定正整数 n，找到若干个完全平方数（比如 1, 4, 9, 16, ...）使得它们的和等于 n。你需要让组成和的完全平方数的个数最少。

- 四平方和定理：
  任意一个正整数均可表示为4个整数的平方和（或更少）
- 优化
  一个数如果含有因子4，那么我们可以把4都去掉，并不影响结果
  如果一个数除以8余7的话，那么肯定是由4个完全平方数组成

```cpp
class Solution {
public:
    int numSquares(int n) {
        while (n % 4 == 0)
            n /= 4;
```

```
        if (n % 8 == 7)
            return 4;
        for (int i = 0; i * i <= n; ++i) {
            int temp = sqrt(n - i*i);
            if (i*i + temp*temp == n) {
                return !!i + !!temp; //存在i*i=n，temp=0 （返回1）或者i和temp都不为0（返回2）的两种情况
            }
        }
        return 3;
    }
};
```

239 滑动窗口最大值

给定一个数组 nums，有一个大小为 k 的滑动窗口从数组的最左侧移动到数组的最右侧。你只可以看到在滑动窗口内的 k 个数字。滑动窗口每次只向右移动一位。
返回滑动窗口中的最大值。

- 方法
  set保存k个值，最大值为set的最后一个值，出队通过nums[i-k+1]（时间复杂度比较高）
  优先队列 priority_queue 维护添加进来的数据和下标，下标超过i-k范围内的出队，队头为最大值。
  双向数组来维护滑动窗口，队头为最大值的下标

```
class Solution {
public:
    vector<int> maxSlidingWindow(vector<int>& nums, int k) {
        deque<int> data;
        vector<int> res;
        for (int i = 0; i < nums.size(); ++i) {
            if (!data.empty() && data.front() == i - k)
               data.pop_front();

            while (!data.empty() && nums[data.back()] < nums[i])
                data.pop_back();
            data.push_back(i);

```

```
            if (i >= k-1)
                res.push_back(nums[data.front()]);
        }
        return res;
    }
};
```

236 二叉树的最近公共祖先

两个节点分为三种情况：p和q分别位于左右子树（返回根节点）、p和q同时位于左子树（返回较高的节点）、p和q同时位于左子树

```
class Solution {
public:
    TreeNode* lowestCommonAncestor(TreeNode* root, TreeNode* p, TreeNode* q) {
        if (!root || p == root || q == root)
            return root;
        TreeNode * left = lowestCommonAncestor(root->left, p, q);
        TreeNode * right = lowestCommonAncestor(root->right, p, q);
        if(left && right)
            return root;
        return left ? left : right;
    }
};
```

221 最大正方形

在一个由 0 和 1 组成的二维矩阵内，找到只包含 1 的最大正方形，并返回其面积。

```
class Solution {
public:
    int maximalSquare(vector<vector<char>>& matrix) {
        if (matrix.empty() || matrix[0].empty()) {
            return 0;
        }
        int row = matrix.size(), col = matrix[0].size();
        vector< vector<int>> dp(row ,vector<int>(col));
```

```
        int res = 0;
        for (int i = 0; i < row; ++i) {
            for (int j = 0; j < col; ++j) {
                if (i == 0 || j == 0) { //边界处理
                    dp[i][j] = matrix[i][j] - '0';
                }
                else {
                    if (matrix[i][j] == '1') {
                        dp[i][j] = min(dp[i][j-1], min(dp[i-1][j-1], dp[i-1][j])) + 1;  //取 左边、左上、上边的最小值
                    }
                }
                res = max(dp[i][j], res);
            }
        }
        return res*res;
    }
};
```

字典树（Trie）

Trie树，又称字典树，单词查找树或者前缀树，是一种用于快速检索的多叉树结构，如英文字母的字典树是一个26叉树，数字的字典树是一个10叉树。

```
class TrieNode {
public:
    TrieNode * child[26];
    bool isword;
    TrieNode():isword(false) {
        for(auto &a : child)
            a = nullptr;
    }
};

class Trie {
private:
    TrieNode * root;
public:
    /** Initialize your data structure here. */
```

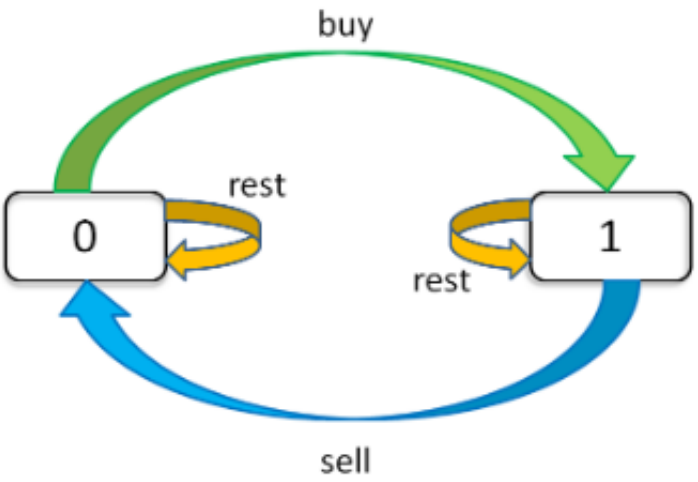
```
16      Trie() {
17          root = new TrieNode();
18      }
19
20      /** Inserts a word into the trie. 添加字符串*/
21      void insert(string word) {
22          TrieNode * p = root;
23          for (auto &a : word) {
24              int i = a - 'a';
25              if (!p->child[i])
26                  p->child[i] = new TrieNode();
27              p = p->child[i];
28          }
29          p->isword = true;//单词结尾
30      }
31
32      /** Returns if the word is in the trie. 查找是否有指定单词 */
33      bool search(string word) {
34          TrieNode * p = root;
35          for (auto a : word) {
36              int i = a - 'a';
37              if (!p->child[i])
38                  return false;
39              p = p->child[i];
40          }
41          return p->isword;
42      }
43
44      /** Returns if there is any word in the trie that starts with the
    given prefix. 查找是否含有指定前缀字符*/
45      bool startsWith(string prefix) {
46          TrieNode * p = root;
47          for (auto a : prefix) {
48              int i = a - 'a';
49              if (!p->child[i])
50                  return false;
51              p = p->child[i];
52          }
53          return true;
54      }
```

```
};
```

股票类题目

第几天-最多进行交易次数-是否持有股票
dp[3][2][1] 的含义就是：今天是第三天，我现在手上持有着股票，至今最多进行 2 次交易。
再比如 dp[2][3][0] 的含义：今天是第二天，我现在手上没有持有股票，至今最多进行 3 次交易。

- 状态转移框架



通过这个图可以很清楚地看到，每种状态（0 和 1）是如何转移而来的。根据这个图，我们来

- 写一下状态转移方程：
  dp[i][k][0] = max(dp[i-1][k][0], dp[i-1][k][1] + prices[i])
  max(选择 rest , 选择 sell)
  dp[i][k][1] = max(dp[i-1][k][1], dp[i-1][k-1][0] - prices[i])
  max(选择 rest , 选择 buy)
- 初始化
  dp[-1][k][0] = 0 //为开始，利润为0
  dp[-1][k][1] = INT_MIN //未开始，不可能持有股票
  dp[i][0][0] = 0 //未交易时，利润为0
  dp[i][0][1] = INT_MIN //未持有股票时不允许交易
- 总结

```
base case:
```

```
dp[-1][k][0] = dp[i][0][0] = 0
dp[-1][k][1] = dp[i][0][1] = -infinity

状态转移方程：
dp[i][k][0] = max(dp[i-1][k][0], dp[i-1][k][1] + prices[i])
dp[i][k][1] = max(dp[i-1][k][1], dp[i-1][k-1][0] - prices[i])
```

309 最佳买卖股票时机含冷冻期

给定一个整数数组，其中第 i 个元素代表了第 i 天的股票价格 。
设计一个算法计算出最大利润。在满足以下约束条件下，你可以尽可能地完成更多的交易（多次买卖一支股票）:

- 你不能同时参与多笔交易（你必须在再次购买前出售掉之前的股票）。
- 卖出股票后，你无法在第二天买入股票 (即冷冻期为 1 天)。

思路：
dp[i][0] = max(dp[i-1][0], dp[i-1][1] + prices[i])
dp[i][1] = max(dp[i-1][1], dp[i-2][0] - prices[i])
解释：第 i 天选择 buy 的时候，要从 i-2 未持有股票那天的钱来买（第i-1天冷冻期钱不变）。

```
class Solution {
public:
    int maxProfit(vector<int>& prices) {
        int dp_1 = INT_MIN, dp_0 = 0;
        int dp_pre_0 = 0; //代表 dp[i-2][0]
        for(int price : prices){
            int temp = dp_0;
            dp_0 = max(dp_1 + price, dp_0); //出售或者维持
            dp_1 = max(dp_pre_0 - price, dp_1); //购买或者维持
            dp_pre_0 = temp;
        }
        return dp_0;
    }
};
```

714 买卖股票的最佳时机含手续费

```cpp
class Solution {
public:
    int maxProfit(vector<int>& prices, int fee) {
        int dp_0 = 0, dp_1 = INT_MIN>>1;
        for (int price : prices) {
            int dp_pre_0 = dp_0;
            dp_0 = max(dp_0, dp_1 + price - fee); //未持有状态，由上个状态的 保持和出售转换而来
            dp_1 = max(dp_1, dp_pre_0 - price);  //出售状态，由上个状态的 保持和购买转换而来
        }
        return dp_0;
    }
};
```

188 买卖股票的最佳时机 IV

利用局部最优和全局最优来解答

```cpp
class Solution {
public:
    int maxProfit(int k, vector<int>& prices) {
        if(prices.empty())
            return 0;
        if( k >= prices.size()) //交易次数大于股票数量的时候
            return solveMaxProfit(prices);
        int globle[k+1] = {0}, local[k+1] = {0};
        for(int i = 1; i < prices.size(); ++i){
            int profit = prices[i] - prices[i-1];
            for(int j = k; j >=1; --j){
                local[j] = max(globle[j-1] + max(0,profit), local[j]+profit);
                globle[j] = max(globle[j], local[j]);
            }
        }
        return globle[k];
    }
    int solveMaxProfit(vector<int> &prices) {
        int res = 0;
```

```
        for (int i = 1; i < prices.size(); ++i) {
            if (prices[i] - prices[i - 1] > 0) {
                res += prices[i] - prices[i - 1];
            }
        }
        return res;
    }

};
```

123 买卖股票的最佳时机 III

给定一个数组，它的第 i 个元素是一支给定的股票在第 i 天的价格。
设计一个算法来计算你所能获取的最大利润。你最多可以完成 **两笔** 交易。

> 注意: 你不能同时参与多笔交易（你必须在再次购买前出售掉之前的股票）。
>
> - 思路 动态规划，globle[i][j]表示 i 天内买卖 j 次最优的全局利润。local[i][j]表示 i 天内（且第 i 天卖出股票），买卖 j 次最优的局部利润。 local[i][j] = max(globle[i-1][j-1] + max(profit,0), local[i-1][j]+profit);
> - globle[i-1][j-1] + max(profit,0) i-1天内j-1次买卖的全局最优解 + 当前大于0的利润值
> - local[i-1][j]+profit 局部最优 local[i][j] 或 i-1天j次买卖的局部最优解 + 第i天卖出利润值 的最大值（因为i-1天交易j次，也可以当做第i天卖出交易j次）

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

- 全局最优解取 i-1天内交易j次的全局利润 和 当前局部最优 的最大值。

```
class Solution {
public:
    int maxProfit(vector<int>& prices) {
        if(prices.empty())
            return 0;
        int n = prices.size(), globle[n][3] = {0}, local[n][3] = {0};
        for(int i = 1;i < prices.size(); ++i){
            int profit = prices[i] - prices[i-1];
            for(int j = 1;j <= 2; ++j){
```

```
                local[i][j] = max(globle[i-1][j-1] + max(profit,0), local[i-1][j]+profit);
                globle[i][j] = max(globle[i-1][j], local[i][j]);
            }
        }
  return globle[n-1][2];
    }
};
```

148 排序链表

在 O(n log n) 时间复杂度和常数级空间复杂度下，对链表进行排序。

```
class Solution {
public:
    ListNode* sortList(ListNode* head) {
        if(!head || !head->next){
   return head;
  }
  ListNode * pre = head, * slow = head, *quick = head;
  while(quick && quick->next){
   pre = slow;
   slow = slow->next;
   pre = pre->next->next;
  }
  pre->next = NULL;
  return Merge(sortList(head), sortList(slow));    //链表分层两块进行归并，子块递归进行归并排序
    }
 ListNode * Merge(ListNode * left, ListNode * right){
  if(!left){
   return right;
  }
  if(!right){
   return left;
  }
  if(left->val < right->val){
   left->next = Merge(left->next, right);
   return left;
```

```
  }
  else{
   right->next = Merge(left, right->next);
   return right;
  }
 }
};
```

152 乘积最大子序列

```
class Solution {
public:
    int maxProduct(vector<int>& nums) {
        int res = nums[0], Max = res, Min = res;
  for(int i = 1; i < nums.size(); ++i){
   if(nums[i] < 0){   //负值时进行调换最大值和最小值
    swap(Max, Min);
   }
   Max = max(nums[i], Max*nums[i]);//求保存包含nums[i]的最大值
   Min = min(nums[i], Min*nums[i]); //求保存包含nums[i]的最小值
   res = max(res, Max);
  }
  return res;
    }
};
```