



**《程序设计原理》实验报告**

**——问题 G: 卡牌大师**

**学 号 3020205094**

**姓 名 刘宏伟**

**学 院 智能与计算学部**

**年 级 2021级**

**任课教师 杜朴风**

**2022年 12月 11日**

# 实验目的

掌握C++全排列的实现与递归构造所有情况，掌握表达式求值的基本方法，掌握使用递归实现的主符号法和使用栈进行模拟这两个经典的方法进行表达式求值，并进行一定的对比分析。

# 二．实验内容

1. **设计思路**

首先用一个数据存放输入的卡片点数，由题意可知需要列举所有的情况进行判断，并且可以移动卡片顺序，这里可以考虑对卡片的位置摆放进行一个全排列，首先对卡片数组排序，然后利用next\_permutation获取接下来每个的卡牌顺序，然后对这个顺序再进行各个空隙符号的摆放，这里使用递归，循环四个符号，摆满即进行判断，这里我同时实现了主符号法和栈模拟法两种方法，如果判断为24点即计数并输出。

1. **代码实现**（写上必要的注释）

#include <bits/stdc++.h>

using namespace std;

// 定义n与答案计数器

int n, cnt = 0;

// 定义符号映射表

char op[5] = {' ', '+', '-', '\*', '/'};

// 定义符号数组

int posture[6];

// 定义卡牌数组

int card[6];

// 查找主符号，\*/优先级大于+-

int find\_main\_symbol(int l, int r) {

for (int i = r - 1; i >= l; i--) {

if (posture[i] == 1 || posture[i] == 2) {

return i;

}

}

for (int i = r - 1; i >= l; i--) {

if (posture[i] == 3 || posture[i] == 4) {

return i;

}

}

return -1;

}

// 计算表达式值的递归函数

int calculate(int l, int r) {

// 查找主符号的位置

int pos = find\_main\_symbol(l, r);

if (pos == -1) {

return card[l];

}

// 根据主符号位置左右分治递归

int left = calculate(l, pos);

int right = calculate(pos + 1, r);

switch (posture[pos]) {

case 1:

return left + right;

case 2:

return left - right;

case 3:

return left \* right;

case 4:

return left / right;

default:

break;

}

return 0;

}

int stack\_calculate() {

// 定义数字栈和符号栈

int num\_stack[6], num\_p = -1, op\_stack[6], op\_p = 0;

op\_stack[0] = -3;

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

num\_stack[++num\_p] = card[i];

if (i < n - 1)

// 如果将要入栈的元素优先级小于等于栈顶元素

if (posture[i] / 3 <= op\_stack[op\_p] / 3) {

switch (op\_stack[op\_p]) {

case 1:

num\_stack[num\_p - 1] += num\_stack[num\_p];

break;

case 2:

num\_stack[num\_p - 1] -= num\_stack[num\_p];

break;

case 3:

num\_stack[num\_p - 1] \*= num\_stack[num\_p];

break;

case 4:

num\_stack[num\_p - 1] /= num\_stack[num\_p];

break;

default:

break;

}

num\_p--;

op\_stack[op\_p] = posture[i];

} else {

op\_stack[++op\_p] = posture[i];

}

}

// 清空符号栈

while (op\_p > 0) {

switch (op\_stack[op\_p]) {

case 1:

num\_stack[num\_p - 1] += num\_stack[num\_p];

break;

case 2:

num\_stack[num\_p - 1] -= num\_stack[num\_p];

break;

case 3:

num\_stack[num\_p - 1] \*= num\_stack[num\_p];

break;

case 4:

num\_stack[num\_p - 1] /= num\_stack[num\_p];

break;

default:

break;

}

num\_p--;

op\_p--;

}

return num\_stack[0];

}

// 通过递归来填满各个空位的符号

void recurrence(int k) {

// 如果填满了，使用相应方法验证点数是否为24从而计数输出

if (k == n - 1) {

// 使用栈模拟法

// int ans = stack\_calculate();

// 使用主符号法

int ans = calculate(0, n);

if (ans == 24) {

cnt++;

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

cout << card[i];

if (i < n - 1)

cout << op[posture[i]];

}

cout << endl;

}

return;

}

// 否则继续填

for (int i = 1; i < 5; i++) {

posture[k] = i;

recurrence(k + 1);

}

}

int main() {

// 输入n张卡牌并进行排序

cin >> n;

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

cin >> card[i];

}

sort(card, card + n);

// 对卡牌全排列中的每一种情况进行递归

do {

recurrence(0);

} while (next\_permutation(card, card + n));

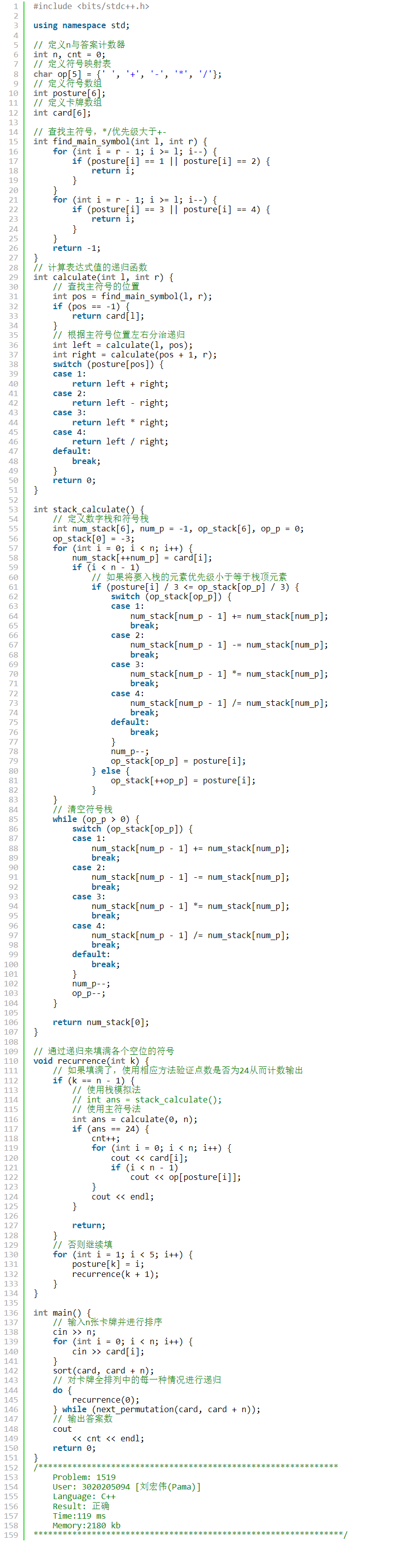
// 输出答案数

cout<< cnt << endl;

return 0;

};

1. **结果展示**（Oj平台提交记录）



# 三．实验中遇到的问题和解决办法

两种方法的对比分析：

主符号法和栈模拟法的主要区别在于它们的实现方式不同。主符号法通过确定表达式中的主符号，来决定应该如何解析表达式。而栈模拟法则通过使用一个栈来模拟表达式的解析过程，并在遇到符号时，根据符号的优先级来决定如何进行解析。

主符号法和栈模拟法都有各自的优点和缺点。主符号法的优点在于它的实现相对简单，不需要使用栈来模拟表达式的解析过程。但是，主符号法的缺点在于它可能不够灵活，无法处理一些特殊的表达式。

栈模拟法的优点在于它的实现相对灵活，可以处理一些特殊的表达式。但是，栈模拟法的缺点在于它的实现复杂度高，需要使用栈来模拟表达式的解析过程。

遇到的问题：

这里我采用了一个数组来储存符号状态，而不是之前常用的字符串等方法，避免了转换上的麻烦，但同时也对算法理解有了更高的要求。同时在主符号法中，主符号的优先级与遍历顺序也有讲究，需要十分小心。这里也体现了其不够灵活的特性。