24点游戏   
  
  
数字游戏题解   
by starfish   
  
[说明：此文改编自我写的一篇解题报告，原题是某年国家集训队组队赛题目]   
  
  
问题描述   
  
80年代全世界流行一种数字游戏，在中国我们把这种游戏称为“24点”。现在我们   
把这个有趣的游戏推广一下：您作为游戏者将得到6个不同的自然数作为操作数，   
以及另外一个自然数作为理想目标数，而您的任务是对这6个操作数进行适当的算   
术运算，要求运算结果小于或等于理想目标数，并且我们希望所得结果是最优的，   
即结果要最接近理想目标数。   
您可以使用的运算只有：+，-，\*，/，您还可以使用（）来改变运算顺序。注意：   
所有的中间结果必须是整数，所以一些除法运算是不允许的（例如，（2\*2）/4是   
合法的，2\*（2/4）是不合法的）   
下面我们给出一个游戏的具体例子：   
若给出的6个操作数是：1，2，3，4，7和25，理想目标数是573；   
则最优结果是573：（（（4\*25-1）\*2）-7）\*3。   
  
输入：   
  
输入文件名为game.in。输入文件仅一行，包含7个整数，前6个整数Mi,   
1<=Mi<=100，表示操作数，最后一个整数T, 1<=T<=1000,表示理想目标数。   
  
输出：   
  
输出文件名为game.out。输出文件有两行，第一行仅一个整数，表示您的程序计算   
得到的最优结果；第二行是一个表达式，即您得到的最优结果的运算方案。   
  
输入输出示例：   
  
输入文件   
  
1 2 3 4 7 25 573   
  
输出文件   
  
573   
  
((4\*25-1)\*2)-7)\*3   
  
  
算法分析   
        
首先我们要对这个问题进行数学抽象。   
  
定义1：对于有理数组成的多重集合S ， f(S) 定义如下：   
  
如果 S 是空集或只包含一个元素，则 f(S)=S ；否则   
f(S)=∪ f( ( S-{r1, r2}) ∪ {r} ) ，对于每一个 r=r1+r2 , r1-r2 , r1×r2   
，r1÷r2(r2≠0)，且r1, r2取遍 S 中所有元素的组成的二元组。   
  
定义1说明：要计算集合S中的元素通过四则混合运算所能得到的所有值，我们只需   
要任取 S 中的两个元素 r1 , r2 ，分别计算 r1 , r2 的加减乘除运算，然后用   
所得的结果与 S 中剩下的其他数字进行四则混合运算。只要取遍所有的 r1 ,   
r2 ，最后得到的所有结果的并集就是 S 中的元素通过四则混合运算所能得到的所   
有值的集合。   
  
根据上述定义，在本问题中，集合 S 就是由输入中给定的6个正整数组成的集合，   
题目所求就是找出 f(S) 中小于或等于目标数的最大数。   
  
定义2：给定两个多重集合 S1 , S2，定义   
        comb( S1, S2 ) = ∪ { r1+r2 , r1-r2, r1×r2, r1÷r2(r2≠0) }          
  
(1.1)   
其中 ( r1 , r2 ) ∈ S1 × S2。   
  
定义2实际上定义了两个集合中的元素两两进行加减乘除运算所能得到的结果集合   
。   
  
定理1：对于有理数组成的多重集合 S ，如果 S 至少有两个元素，则   
  
        f(S)=∪ comb( f(S1), f(S - S1) )                      (1.2)   
  
其中 S1 取遍 S 的所有非空真子集。   
  
定理1的含义是：要计算 S 中的元素通过四则混合运算所能得到的所有值，可以先   
将 S 分解为两个子集 S1 和 S- S1 ，分别计算 S1 和 S-S1 中的元素进行四则混   
合运算所能得到的结果集合，即 f(S1) 和 f(S-S1) ，然后对这两个集合中的元素   
进行加减乘除运算，即 comb( f(S1), f(S-S1) ) ，最后得到的所有集合的并集就   
是 f(S) 。限于篇幅，定理1的正确性易用数学归纳法证明。   
  
定义1和定理1实际上分别给出了计算f(S)的两种不同的方法。根据定义1，可以递   
归地计算f(S) ，其算法伪代码如下：   
  
算法1   
  
function f(S)   
begin   
1.  if |S| < 2   
2.    then return S   
3.    else begin   
4.           T ← Φ   
5.           for each (r1, r2) in S do   
6.           begin   
7.             r ← r1 + r2;   
8.             T ← T + f(S – {r1, r2} + {r});   
9.             r ← r1 - r2;   
10.            T ← T + f(S – {r1, r2} + {r});   
11.            r ← r1 \* r2;   
12.            T ← T + f(S – {r1, r2} + {r});   
13.            if (r2 <> 0) and (r1 mod r2 = 0) then   
14.            begin   
15.              r ← r1 / r2;   
16.              T ← T + f(S – {r1, r2} + {r});   
17.            end   
18.          end   
19.          return T;   
20.   end   
end   
  
上述伪代码中使用了+, - 来分别表示集合的并和差运算。算法1每次选择两个数字   
进行某种运算，然后将结果与剩下的数字递归地进行运算，最后求得所有数字进行   
四则混合运算的结果。当然，在具体实现该算法的过程中有很多可以优化的地方，   
比如根据加法交换律， a+b+c=a+c+b ，因此我们可以规定：如果上一层递归作了   
加法运算，这一层仅当满足当前的操作数大于上一层的两个操作数的时候才进行加   
法运算，以确保 a+b+c 这样的式子中的操作数总是从小到大排列，这样就可以避   
免重复进行等价的加法计算。类似地我们可以对乘法也作此规定。在进行减法的时   
候，我们可以规定只能计算大数减小数，因为最后所需计算得到的目标数是一个正   
数，如果计算过程中出现负数，肯定有另外一个较大的正数与其作加法或者有另外   
一个负数与其做乘除法以消除负号。因此我们总可以调整运算次序使得四则混合运   
算的每一步的中间结果都是正数。在作除法的时候，因为题目规定中间结果只能是   
整数，所以也只需要用大数除小数，且仅当能除尽的时候才进行除法。对于本题而   
言，初始的集合 S 中一共有6个操作数，每次递归都可以合并两个操作数，所以递   
归到第5层的时候集合 S 中只剩下一个数，这个数就是原先的6个操作数进行四则   
混合运算所能得到的结果。本题只要求最接近目标值的结果，所以实现上述算法的   
时候可以只记录当前最优的结果。对于本题也可以利用递归回溯构造出所有的四则   
混合运算的语法树，但本质上与算法1是没有区别的。   
  
定理1则给出了另一种计算f(S)的方法。我们当然也可以根据(1.2)式直接地递归计   
算f(S)，但那样的话会有很多冗余计算。例如对于S={1,2,3,4}，   
   f(S) = comb( f({ 1 }), f({ 2,3,4}) )∪ ... ∪ comb( f({ 1,2 }), f({   
3,4 }) ) ∪ ...;   
计算f(S)的时候需要计算 f({ 2,3,4 })和f({ 3,4 }) ，又因为   
   f({2,3,4}) = comb(f({ 2 }), f({3,4})) ∪ ...;   
在计算 f({ 2,3,4}) 的时候又要重复地计算 f({ 3,4 }) ，这就产生了冗余的计   
算。这种情况下直接地递归就不适用。必须按照一定的顺序，递推地进行计算。这   
种将递归改为递推，以解决冗余的算法设计策略，就叫做动态规划。   
  
下面我们具体阐述一下该算法的步骤。设初始时集合 S 中的 n 个数字分别为   
x[0], x[1],...,x[n-1] ，我们可以用一个二进制数k来表示S 的子集 S[k] ，   
x[i] ∈ S[k] 当且仅当二进制数k的第i位为1。于是我们用一个数组 F[0..2^n-1]   
就可以保存函数f对于S的所有子集的函数值（注意，函数f的函数值是一个集合）   
，且 F[2^n-1]=f(S) 就是所求。   
  
算法2   
1.  for i ← 0 to 2^n-1   
2.    do F[i]←Φ;   
3.  for i ← 0 to n-1   
4.    do F[2^i]← {x[i]};   
5.  for x ← 1 to 2^n-1 do   
6.  begin   
7.    for i ← 1to x-1 do            
8.    begin   
9.      if x∧i=i then   
10.     begin   
11.       j ← x – i;   
12.       if i < j   
13.         then F[x] ← F[x] + comp(F[i],F[j]);   
14.     end;   
15.   end;   
16. end;                    
17. return F[ 2 n ?1] ;   
  
上述伪代码中使用了+表示集合的并运算。算法2的第1～2行将F中所有的集合初始   
化为空；第3～4行中 2^i 即表示只包含元素 x[i]的子集（因为 2^i 只有第 i 位   
上是1），根据定义1我们知道当集合中只有一个元素的时候函数 f 的函数值就是   
那唯一的元素组成的集合，所以3～4行计算出了函数 f 对于所有只有一个元素的   
子集的函数值；第5～17行按照一定的顺序计算函数 f 对于 S 的所有子集的函数   
值。对于 S 的两个子集 S[i] 和 S[x] ， S[i]真包含于S[x]的充要条件是 x∧   
i=i ，这里 ∧ 是按位进行与操作，而 x∧i=i 的必要条件是 i<x 。因而第7~15   
行的循环将S[x]拆成两个子集S[i]和S[j]，并在第13行根据(1.2)式计算所有的   
comp( f(S[i]),f(S[j]) ) 的并。第12行的判断语句是为了优化算法的效率，因为   
将 S[x]拆成两个子集 S[i]和 S[j]的过程是对称的，所以我们对于 comp(   
f(S[i]),f(S[j]) ) 和 comp( f(S[j]),f(S[i]) ) 两者只取一个进行计算。下面   
是函数comp的伪代码：   
  
算法3   
function comp(S1, S2)   
1.  T ← Φ ;   
2.  for each x in S1 do   
3.  begin   
4.    for each y in S2 do   
5.    begin   
6.      T ← T + {(x + y)};   
7.      T ← T + {(x \* y)};   
8.      if x > y then   
9.      begin   
10.       T ← T + {(x – y)};   
11.       if (y <> 0) and (x mod y = 0)   
12.          then T ← T + {(x / y)};   
13.     end   
14.     else begin   
15.       T ← T + {(y – x)};   
16.       if (x <> 0) and (y mod x = 0)   
17.         then T ← T + {(y / x)};   
18.     end;   
19.   end;   
20. end;   
21. return T;   
  
comp在进行计算的时候不考虑参数集合S1和S2的顺序，进行减法的时候始终用大   
数减小数，这样保证运算过程中不出现负数（这样做的理由前文已经阐明）。   
  
因为我们只关心最后的f(S)中最接近目标值的数字，并且题目只要求求出任何一组   
最优解，所以算法2中的集合不需要是多重集合，只要是一般的集合即可。换句话   
说，集合F[i]中所有的元素互不相同，重复出现元素的我们只保留其中一个。这样   
可以大大减少计算中的冗余。做了这样的处理后，算法2的效率至少不会比算法1差   
，因为算法1中所能采用的主要剪枝手段是排除等价的表达式，但因为等价的两个   
表达式计算出的结果也一定相同，而算法2排除了所有结果相同的表达式，所以算   
法2的效率至少不会比算法1差，算法2中所进行的计算基本上都是得到最优解所必   
需的计算。   
  
在实现算法2的过程中，集合可以用一个链表加上一个哈希表来实现。链表中保存   
每个表达式及其值，哈希表用来记录该集合中是否存在某个特定值的表达式。当向   
集合中插入一个新的表达式的时候，首先检查哈希表，看看该集合是否已经有和新   
表达式值相同的表达式，如果有的话就不插入，否则将新的表达式追加到链表末尾   
。采用这种数据结构，可以在常数时间内完成集合的插入和删除操作。利用链表，   
集合的并操作也很容易高效地实现。   
  
在实现算法2的过程中，可以不必保存表达式的字符串，只需要记录下当前的值是   
由哪两个集合中的元素通过哪种运算得到的，最后再根据最优解递归地计算出最优   
解的表达式。这样只在最后构造最优解的表达式时才进行字符串操作，程序运行效   
率能提高7～8倍左右。另外，在comb函数中进行乘法运算的时候要注意考虑运算结   
果超出整数范围的情况。   
  
经过以上优化，利用算法2实现的程序对于100个随机生成的测试数据总共只需要5   
秒左右就可以出解，平均每个数据只需要50毫秒即可出解（测试用的CPU为赛扬   
1GB）。这样的效率已经非常令人满意了。   
  
  
  
附录：   
  
1。根据算法1计算24点的代码   
  
#include <iostream>   
#include <string>   
#include <cmath>   
  
using namespace std;   
  
const double PRECISION = 1E-6;   
const int COUNT\_OF\_NUMBER  = 4;   
const int NUMBER\_TO\_CAL = 24;   
  
double number[COUNT\_OF\_NUMBER];   
string expression[COUNT\_OF\_NUMBER];   
  
bool Search(int n)   
{   
    if (n == 1) {   
        if ( fabs(number[0] - NUMBER\_TO\_CAL) < PRECISION ) {   
            cout << expression[0] << endl;   
            return true;   
        } else {   
            return false;   
        }   
    }   
  
    for (int i = 0; i < n; i++) {   
        for (int j = i + 1; j < n; j++) {   
            double a, b;   
            string expa, expb;   
  
            a = number[i];   
            b = number[j];   
            number[j] = number[n - 1];   
  
            expa = expression[i];   
            expb = expression[j];   
            expression[j] = expression[n - 1];   
  
            expression[i] = '(' + expa + '+' + expb + ')';   
            number[i] = a + b;   
            if ( Search(n - 1) ) return true;   
              
            expression[i] = '(' + expa + '-' + expb + ')';   
            number[i] = a - b;   
            if ( Search(n - 1) ) return true;   
              
            expression[i] = '(' + expb + '-' + expa + ')';   
            number[i] = b - a;   
            if ( Search(n - 1) ) return true;   
                          
  
            expression[i] = '(' + expa + '\*' + expb + ')';   
            number[i] = a \* b;   
            if ( Search(n - 1) ) return true;   
  
            if (b != 0) {   
                expression[i] = '(' + expa + '/' + expb + ')';   
                number[i] = a / b;   
                if ( Search(n - 1) ) return true;   
            }   
            if (a != 0) {   
                expression[i] = '(' + expb + '/' + expa + ')';   
                number[i] = b / a;   
                if ( Search(n - 1) ) return true;   
            }   
  
            number[i] = a;   
            number[j] = b;   
            expression[i] = expa;   
            expression[j] = expb;   
        }   
    }   
    return false;   
}   
  
void main()   
{   
    for (int i = 0; i < COUNT\_OF\_NUMBER; i++) {   
        char buffer[20];   
        int  x;   
        cin >> x;   
        number[i] = x;   
        itoa(x, buffer, 10);   
        expression[i] = buffer;   
    }   
  
    if ( Search(COUNT\_OF\_NUMBER) ) {   
        cout << "Success." << endl;   
    } else {   
        cout << "Fail." << endl;   
    }          
}   
  
2。根据算法2计算解决题目的程序代码：   
  
#include <fstream>   
#include <algorithm>   
#include <string>   
#include <sstream>   
#include <list>   
#include <cmath>   
#include <climits>   
#include <bitset>   
using namespace std;   
  
const char\* INPUT\_FILE  = "game.in";   
const char\* OUTPUT\_FILE = "game.out";   
const int NUMBER\_COUNT  = 6;   
const int STATE\_COUNT   = (1 << NUMBER\_COUNT);   
const int MAX\_NUMBER    = 100;   
const int MAX\_EXPECTION = 1000;   
const int MAX\_VALUE             = MAX\_EXPECTION \* MAX\_NUMBER;   
  
struct Node {   
        int value;   
        int left, right;          
        int leftvalue, rightvalue;   
        char opr;   
};   
  
typedef list<Node> NodeList;   
  
struct State {   
        bitset<MAX\_VALUE+10> exist;   
        NodeList nodelist;   
};   
  
int number[NUMBER\_COUNT], expection;   
State state[STATE\_COUNT];   
  
void ReadData()   
{   
        ifstream fin(INPUT\_FILE);   
          
        for (int i = 0; i < NUMBER\_COUNT; i++) {   
                fin >> number[i];   
        }   
        fin >> expection;   
}   
  
void Init()   
{   
        Node node ;   
        for (int i = 0; i < NUMBER\_COUNT; i++) {   
                node.value              = number[i];   
                node.left = node.right = -1;   
                state[(1 << i)].nodelist.push\_back(node);   
                state[(1 << i)].exist[node.value] = true;   
        }   
}   
  
void Merge(int a, int b, int x)   
{         
        Node node;        
        NodeList::const\_iterator i, j;   
  
        for (i = state[a].nodelist.begin(); i != state[a].nodelist.end();   
i++) {   
                for (j = state[b].nodelist.begin(); j != state[b].nodelist.en   
  
d(); j++)   
{                                        
                        node.value = (\*i).value + (\*j).value;   
                        node.left  = a;   
                        node.right = b;                   
                        node.leftvalue  = (\*i).value;   
                        node.rightvalue = (\*j).value;   
                        node.opr   = '+';   
                        if ( (node.value <= MAX\_VALUE) && (!state[x].exist[no   
  
de.value]) ) {   
                                state[x].nodelist.push\_back(node);   
                                state[x].exist[node.value] = true;   
                        }   
  
                        /////////////////////////////////////////////////////   
  
                        double tmp = double((\*i).value) \* double((\*j).value);   
  
                        if (tmp < INT\_MAX) {   
                                node.value = (\*i).value \* (\*j).value;   
                                node.left  = a;   
                                node.right = b;   
                                node.leftvalue  = (\*i).value;   
                                node.rightvalue = (\*j).value;   
                                node.opr   = '\*';   
                                if ( (node.value <= MAX\_VALUE) && (!state[x].   
  
exist[node.value]) )   
{   
                                        state[x].nodelist.push\_back(node);   
                                        state[x].exist[node.value] = true;   
                                }   
                        }   
  
                        /////////////////////////////////////////////////////   
  
                        if ((\*i).value >= (\*j).value) {   
                                node.value = (\*i).value - (\*j).value;   
                                node.left  = a;   
                                node.right = b;   
                                node.leftvalue  = (\*i).value;   
                                node.rightvalue = (\*j).value;   
                                node.opr   = '-';   
                        } else {   
                                node.value = (\*j).value - (\*i).value;   
                                node.left  = b;   
                                node.right = a;   
                                node.leftvalue  = (\*j).value;   
                                node.rightvalue = (\*i).value;   
                                node.opr   = '-';   
                        }   
                                                  
                        if ( (node.value <= MAX\_VALUE) && (!state[x].exist[no   
  
de.value]) ) {   
                                state[x].nodelist.push\_back(node);   
                                state[x].exist[node.value] = true;   
                        }   
  
                        /////////////////////////////////////////////////////   
  
  
                        if ( ((\*j).value != 0) && ((\*i).value >= (\*j).value) &   
&   
                                        ((\*i).value % (\*j).value == 0) )   
                        {   
                                node.value = (\*i).value / (\*j).value;   
                                node.left  = a;   
                                node.right = b;           
                                node.leftvalue  = (\*i).value;   
                                node.rightvalue = (\*j).value;   
                                node.opr   = '/';   
                        } else if ( ((\*i).value != 0) && ((\*j).value >= (\*i).   
  
value) &&   
                                        ((\*j).value % (\*i).value == 0) )   
                        {   
                                node.value = (\*j).value / (\*i).value;   
                                node.left  = b;   
                                node.right = a;   
                                node.leftvalue  = (\*j).value;   
                                node.rightvalue = (\*i).value;   
                                node.opr   = '/';   
                        }   
                                                  
                        if ( (node.value <= MAX\_VALUE) && (!state[x].exist[no   
  
de.value]) )   
{                                 
                                state[x].nodelist.push\_back(node);   
                                state[x].exist[node.value] = true;   
                        }                         
                        /////////////////////////////////////////////////////   
  
                }   
        }   
}   
  
void Solve()   
{   
        Init();   
          
        for (int x = 2; x < STATE\_COUNT; x++) {   
                for (int i = 1; i < x; i++) {                     
                        if ( (x & i) == i ) {   
                                int j = x - i;   
                                if (i <= j) {   
                                        Merge(i, j, x);           
                                }   
                        }   
                }   
        }   
}   
  
void PrintExpression(ostream& out, Node node)   
{   
        if (node.left == -1) {   
                out << node.value;   
        } else {   
                NodeList::const\_iterator iter;   
                  
                out << "(";   
  
                for (iter = state[node.left].nodelist.begin();   
                                iter != state[node.left].nodelist.end();   
                                iter++)   
                {   
                        if ((\*iter).value == node.leftvalue) {   
                                PrintExpression(out, \*iter);   
                                break;   
                        }   
                }   
  
                out << node.opr;   
  
                for (iter = state[node.right].nodelist.begin();   
                                iter != state[node.right].nodelist.end();   
                                iter++)   
                {   
                        if ((\*iter).value == node.rightvalue) {   
                                PrintExpression(out, \*iter);   
                                break;   
                        }   
                }   
  
                out << ")";   
        }                 
}   
  
void Output()   
{   
        ofstream fout(OUTPUT\_FILE);   
  
        int bestValue = -INT\_MAX;         
        NodeList::const\_iterator iter, bestIter;   
  
        NodeList& nodelist = state[STATE\_COUNT-1].nodelist;   
  
        for (iter = nodelist.begin(); iter != nodelist.end(); iter++)   
        {         
                if ( ((\*iter).value <= expection) && (bestValue < (\*iter).val   
  
ue) ) {   
                        bestValue = (\*iter).value;   
                        bestIter  = iter;   
                }   
        }         
        fout << bestValue << endl;   
        PrintExpression(fout, \*bestIter );   
        fout << endl;   
}   
  
int main()   
{   
        ReadData();   
        Solve();   
        Output();   
        return 0;   
}