□号匹配

 alt text □栈来模拟括号匹配的过程 用栈是因为栈可以解决具有完全包含关系的问题

```
#include<bits/stdc++.h>
using namespace std;
int main()
{
int n;
int flag = 0;
cin >> n;
vector<string> temp[n] , s;
string str;
string target;
for(int i = 0; i < n; i++)
{
    cin >> str;
        temp[i] = str;
        str = NULL ;
}
cin >> target;
for(int i = 0; i < n; i++)
{
    if(temp[i] == target)
    {
        s.push_back(temp[i]);
        flag = 1;
        break;
    }
    if(temp[i] == "return")
    s.pop_back(temp[i]);
    else
    s.push_back(temp[i]);
}
if(flag)
{
    for(int i = 0; i < s.size();i++)
    {
        if(i) cout << "->"
            cout << s[i];
    }
    cout << endl;
    else
        {
            cout << "NOT REFERENCED"
        }
}
return 0;
}
```

考研题 三元组最小距离

思路挺巧妙的  alt text

- □里面的数字是从小到大排列的
- 因此我们只需要找到三个数字中最小的那个
    - 对于他来说，其它两个数字再取□后面的数字，其距离会越来越大
    - 因此我们找到三个数字中的最小数字，就说明当前答案已经是对于这个数字来说的最小距离
    - 我们便可以将它从□中删除
    - 同时要记录最小值
- 当有一个□为空战时，就说明已经无数字可比，此时找到那个记录的最小值，输出即可

```
#include <iostream>
#include <cstdlib>
#include <queue>
using namespace std;

int min_num(int a, int b, int c) {
    if (a > b) swap(a, b);
    if (a > c) swap(a, c);
    return a;
}

int func(queue<int> que1, queue<int> que2, queue<int> que3) {
    int min = 0x3f3f3f;  //□个是16进制表示法，可以看成是正无穷
    while(!que1.empty() && !que2.empty() && !que3.empty())
    {
        int a = que1.front(),b = que2.front(),c = que3.front();
        int min_n = abs(a-b) + abs(a-c) + abs (b-c);
        if(min_n<min)  min = min_n;
        int d = min_num(a,b,c);
        if(a == d) que1.pop();
        if(b == d) que2.pop();
        if(c == d) que3.pop();
}
    return min;
}

int main() {
    int m, n, k, x;
    queue<int> que1, que2, que3;
    cin >> m >> n >> k;
    for (int i = 0; i < m; i++) {
        cin >> x;
        que1.push(x);
    }
    for (int i = 0; i < n; i++) {
        cin >> x;
```

```
            que2.push(x);
        }
        for (int i = 0; i < k; i++) {
            cin >> x;
            que3.push(x);
        }
        cout << func(que1, que2, que3) << endl;
        return 0;
    }
```

比较含退格的字符串

 alt text 使用栈的思想

- 遇到#就出□，否则就入□
- 比较俩□的字符

```
class Solution {
public:
    bool backspaceCompare(string s, string t) {
        stack<char> temp1,temp2;
        for(int i = 0;s[i];i++)
        {
            if(s[i] == '#')   //判断是否为退格字符
            {
                if(!temp1.empty()) //判断栈是否非空 若无□个判断条
            temp1.pop(); 件 遇到a##b时便会把#压入栈，使得逻辑错误
            }
            else temp1.push(s[i]);
        }
        for(int i = 0;t[i];i++)
        {
            if(t[i] == '#')
            {
                if(!temp2.empty())
            temp2.pop();
            }
            else temp2.push(t[i]);
        }
        if(temp1.size() != temp2.size()) return false;
        while(!temp1.empty())
        {
            if(temp1.top() != temp2.top() ) return false;
            temp1.pop();
            temp2.pop();
        }
        return true;
    }
};
```

火车进栈

 alt text

- 全排列输出
- 重点是判断输出序列是否为合法序列
- 用□模拟
    1. 假设有一组输出序列a1，a2，a3，a4，□顶元素为x
    2. 现在轮到a2输出，用a2与x比较
    3. a2 > x 说明a2还没进□，此时应该继续进□，直到a2进□
    4. a2 = x □顶元素即进站的火车编号为x，此时a2弹出□
    5. a2 < x 说明a2在□顶元素下方，无法弹出，此时序列不合法
- 一种是使用next_permutation函数来进行全排列
- 一种是使用递归的排列型枚举来进行全排列
- **next_permutation**函数做法

  #include #include #include #include #include #include using namespace std; bool valid(int a[],int n) { stack s; int x = 1; for(int i = 0;i < n;i++) { if(s.empty() || s.top() < a[i]) //血的教训！！ 先判空再比较 { while (x<=a[i]) s.push(x),x+=1; } if(s.top() != a[i]) return false; s.pop(); } return true; } int main() { int a[25],n,cnt = 20; cin >> n; for(int i =0;i < n;i++) a[i] = i + 1; do { if(valid(a,n)) { for(int i = 0;i <n;i++) { cout << a[i]; } cout << endl; cnt-=1; } }while(next_permutation(a,a+n) && cnt); return 0; }

递归的排列型枚举做法

```
#include<iostream>
#include<cstdlib>
#include<cstdio>
#include<string>
#include<cstring>
#include<stack>
#include<algorithm>
#include<queue>
using namespace std;
int vis[25]={0};
int a[25],b[25];
int times = 0;
bool valid(int n)
{
    stack<int> s;
    int x = 1;
    for(int i = 0;i < n;i++)
    {
        if(s.empty() || s.top()<a[i])
        {
            while (x<=a[i])
            {
```

```
                s.push(x);
                x+=1;
            }
        }
        if(s.top()!=a[i]) return false;
        s.pop();
    }
    return true;
}
void print(int n)
{
    for(int i = 0;i < n;i++)
    {
            cout << b[i];
    }
    cout << endl;
}

void f(int i,int n)
{
    if(times == 20) return ;
    if(i == n && valid(n))
    {
        print(i);
        times ++;
        return;
    }
    for(int j = 1;j <= n;j++)
    {
        if(vis[j]) continue;
        b[i] = j;
        a[i] = j;
        vis[j] = 1;
        f(i+1,n);
        vis[j] = 0;
    }
}
int main()
{
    int n;
    cin >> n;
    f(0,n);
}
```

重点解释一下**next_permutation**函数

- 在头文件include< algorithm >中
- 接受两个迭代器作为参数，并返回一个bool值，表示是否成功生成下一个排列
  - 迭代器：访问数据结构中的元素
  - 函数是对这两个迭代器之间的数据进行字典序排序
- 当序列已经是字典序最大排列，返回false，循环结束
- 作用：生成给定序列的下一个较大排序，直到序列按降序排列为止

- 常与do while联用
  - 若与while连用，则会丢失初始时的排序
  - 若你想要得到所有的排序结果，初始时要按照升序来排序

验证栈序列

 alt text 题目意思是按照所给顺序对空栈进行入栈和出栈操作 若执行到最后栈为空，则**return true**

- 要对出□序列进行分析
- 若出□序列当前元素与□顶元素相同，则pop
- 不相同则一直入□
- 若将入□序列遍历完也没有找到当前元素，且□顶元素也与其不同，return false
  - 因为该元素在□顶元素下方 无法pop

>  *alt text*

```
class Solution {
public:
    bool validateStackSequences(vector<int>& pushed, vector<int>& popped) {
        int x = 0,n = pushed.size();
        stack<int> s;
        for(int i = 0;i < n;i++)
        {
            if(s.empty() || s.top() != popped[i])
            {
        while(x < pushed.size() && pushed[x] != popped[i])
        {
            s.push(pushed[x]);
            x += 1;
        }
        if(x == pushed.size()) return false;
        s.push(pushed[x]);
        x +=1 ;
            }
        s.pop();
        }
        return true;
    }
};
```

□号画家

 alt text 要让匹配到的一对□号做上标记

```
#include<iostream>
#include<stack>
using namespace std;
stack<int> s;
```

```
char str[10005];
int match[10005];
int main()
{
    cin >> (str + 1);  //让字符数组下标从1开始
    for(int i = 1;str[i];i++)
    {
        switch(str[i])
        {
        case'(':
        case'[':
        case'{': s.push(i); break;
        case')':
            if(!s.empty() && str[s.top()] == '(')
            {
                match[s.top()] = i;  //表明s.top()和i位置是匹配的
                s.pop();
            }
            else  s.push(i); //表明i位置之前的括号序列非法，做一个信息阻隔
            break;
        case']':

            if(!s.empty() && str[s.top()] == '[')
            {
                match[s.top()] = i;
                s.pop();
            }
            else s.push(i);
            break;
        case'}':
            if(!s.empty() && str[s.top()] == '{')
            {
                match[s.top()] = i;
                s.pop();
            }
            else s.push(i);
            break;
            }
    }
    int temp_ans = 0,ans = 0,i = 1;
    while(str[i])
    {
        if(match[i])
        {
            temp_ans += (match[i] - i + 1); //临时的长度  +=是因为可能会出现([]){}的形式
            i = match[i] + 1;             //跳转到匹配位置的下一个位置，寻找下一个美观长度
        }
        else
        {
            temp_ans = 0;
            i++;
        }
```

```
        if(temp_ans > ans) ans = temp_ans; //取最大长度
    }
    cout << ans;
    return 0;
}
```

设计循环队列

alt text

本题难点：需要确定初始时尾指针位置

- 与队列结构定义不一样，队列机构定义头尾指针采取左闭右开式
- 在本题头尾指针采取左闭右闭式
- 因此尾指针初始位置应该在头指针前一位才能表示队列为空

```
struct Node { int data; Node *next; }; class MyCircularQueue { public: int count,size; Node
*head,*tail; MyCircularQueue(int k) { head = new Node(); tail = head; for(int i = 0;i < k;i++) {
tail->next = new Node(); tail = tail->next; } count = 0; size = k; tail->next = head; }
bool enQueue(int value) { if(isFull()) return false; tail = tail->next; tail->data = value;
count+=1; return true; }
bool deQueue() { if(isEmpty()) return false; head = head->next; count-=1; return true; }
int Front() { if(isEmpty()) return -1; return head->data; }
int Rear() { if(isEmpty()) return -1; return tail->data; }
bool isEmpty() { return count == 0; } bool isFull() { return count == size; } };
```

表达式求值

alt text

- 使用递归的思想
    - 找到计算优先度最低的符号进行分割
    - 对符号左右进行求值
    - 最后返回计算结果
- 要找计算优先度最低的符号须计算权重比
    - 对于括号内部的符号权重比加100
    - 考虑负数，对于数字前的负号，权重比更大，加1000
        - 因为负数与相减相比优先度更高
        - 负数实现过程是用负号分割。用0减去当前数字
- 为了让分割位置指向符号，我们让数字优先度达到正无穷
    - pos代表要分割的位置
    - cur_pri统计当前位置的权重
    - temp_pri统计遇到括号的权重
    - 设定pri为正无穷-1
        - 目的是为了找出分割位置
        - 减去1是为了不让pos指向数字

- □□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□ □□**7-5-2**□□□□7-5□□□□□□□□□□

```
#include<bits/stdc++.h>
using namespace std;
#define INF 0x3f3f3f3f
string str;
bool is_operator(char c)
{
    switch(c)
    {
    case'+':
    case'-':
    case'*':
    case'/':
    case'^': return true;
    default : return false;
    }
    return false;
}

long long result(string &s, long long l,long long r)
{
    long long pos = -1 , pri =INF - 1 , cur_pri , temp_pri = 0;
    for(long long i = l;i < r;i++)
    {
        cur_pri = INF;
        switch(s[i])
        {
        case'(':
            temp_pri += 100;
            break;
        case')':
            temp_pri -=100;
            break;
        case'+':
        case'-':
            cur_pri = 1 + temp_pri;
            break;
        case'*':
        case'/':
            cur_pri = 2 + temp_pri;
            break;
        case'^':
            cur_pri = 3 + temp_pri;
            break;
        }
        if((s[i] == '-' || s[i] == '+') && (i -1 < 0 ||
is_operator(s[i - 1])))
        cur_pri += 1000;
        if(pri >= cur_pri)  //□□□□□□□□□□□
        {
            pri = cur_pri;  // □□□□□
```

```
            pos = i;
        }
    }
    if(pos == -1)
    {
    long long num = 0;
    for(long long i = l;i < r;i++)
    {
        if(s[i] < '0'|| s[i] > '9') continue;
        num = num * 10 + (s[i]-'0');
    }
    return num;
    }
    else
    {
        long long a = result(s,l,pos);
        long long b = result(s,pos+1,r);
        switch(s[pos])
        {
            case'+': return a+b;
            case'-': return a-b;
            case'*': return a*b;
            case'/': return a/b;
            case'^': return pow(a,b);
        }
    }
}
int main()
{
    cin >> str;
    cout << result(str,0,str.size());
    return 0;
}
```