栈题目选讲

河南省实验中学信息技术组

2026 年 02 月 03 日

【例】push，pop，getMin

实现一个栈，支持 push(入栈)、pop(出栈并输出栈顶) 和 getMin(查询栈中最小的值) 三个操作，要求时间复杂度均为 O(1)。

```
push(-1);
push(3);
push(-4);
getMin();   --> Returns -4.
pop();
top();      --> Returns  3.
getMin();   --> Returns -1.
```

【例】Push，Pop，GetMin

- push 和 pop 默认时间复杂度为 O(1)。
- 如果我们只用一个变量记录最小值，当发生出栈操作时，如果最小值恰好被出栈，就无法得知新的最小值是什么。
- 因此需要记录历史上每个时刻的最小值，在出栈后还原即可。
- 定义两个栈 a 和 b，用 a 表示原始栈，用栈 b 存储栈 a 中以栈底开头的每段数据的最小值。
  - 当执行 push(x) 操作时，在 a 中插入 x，在 b 中插入 $min(b$ 的栈顶数据，$x)$。
  - 在执行 pop 操作时，在 a、b 中分别弹出栈顶。
  - 在执行 getMin 操作时，直接输出 b 的栈顶数据。
- 每个操作的时间复杂度都是 O(1)。

【例】Editor

【题目描述】

维护一个整数序列的编辑器，有以下五种操作，操作总数不超过 10^6。

- `I x`：在当前光标位置之后插入一个整数 $x(|x| \leq 1000)$，插入以后光标移动到 x 之后。
- `D`：将光标前面的第一个元素删除，如果前面没有元素，则忽略此操作。
- `L`：将光标向左移动，跳过一个元素，如果左边没有元素，则忽略此操作。
- `R`：将光标向右移动，跳过一个元素，如果右边没有元素，则忽略此操作。
- `Q k`：假设此刻光标之前的序列为 $a_1, a_2, \cdots, a_n$，输出 $\max\limits_{1 \leq i \leq k} s_i$，其中 $s_i = a_1 + a_2 + \cdots + a_i$，其中 k 不超过当前光标的位置。。

【输入格式】

第一行包含一个整数 Q，表示指令的总数。

接下来 Q 行，每行一个指令，具体指令格式如题目描述。

【输出格式】

每一个 `Q k` 指令，输出一个整数作为结果，每个结果占一行。

【例】Editor

【样例输入】

```
8
I 2
I -1
I 1
Q 3
L
D
R
Q 2
```

【样例输出】

```
2
3
```

【样例解释】

```
I 2         2|
I -1        2 -1|
I 1         2 -1 1|
Q 3         2 -1 1|
L           2 -1|1
D           2|1
R           2 1|
Q 2         2 1|
```

【例】Editor

- 所有的操作均在光标所在位置发生，对于插入和删除操作显然可以用栈实现。
- 但是操作中存在光标左移和右移，序列会被分成两段，利用一个栈无法维护，可以设计一个“对顶栈”。
- 建立两个对顶栈，栈 a 存储从序列开头到光标位置的子序列，栈 b 存储从光标位置到序列结尾的子序列，二者都以光标所在的一端为栈顶。
- 为了查询最大前缀和，定义 $s[]$ 表示栈 a 的前缀和，定义 $f[]$ 维护栈 a 的前缀和最大值。设栈 a 的大小为 i，则：
  - 对于 `I x` 操作：将 x 插入栈 a；更新 $s[i] = s[i-1] + x$；更新 $f[i] = max(f[i-1], s[i])$。
  - 对于 `D` 操作：把 a 的栈顶出栈。
  - 对于 `L` 操作：弹出 a 的栈顶，插入到栈 b 中。
  - 对于 `R` 操作：弹出 b 的栈顶，插入到栈 a 中；更新 $s[i] = s[i-1] + a.top()$；更新 $f[i] = max(f[i-1], s[i])$。
  - 对于 `Q k` 操作：输出 $f[k]$ 即可。

【例】Editor

```
char op[5]; scanf("%s", op);
if(op[0] == 'I')
{
    int x; scanf("%d", &x);
    a.push(x);
    int i = a.size();
    s[i] = s[i - 1] + x, f[i] = max(f[i - 1], s[i]);
}
else if(op[0] == 'D')
{
    if(!a.empty()) a.pop();
}
else if(op[0] == 'L')
{
    if(!a.empty()) b.push(a.top()), a.pop();
}
// 其他操作请读者自行完成
```

【例】出栈序列问题

【题目描述】
给定一个无限大的栈和 $1 \sim n$ 这 n 个整数，每个数都要进栈并出栈一次。如果进栈的顺序为 $1, 2, \cdots, n$，那么可能的出栈序列有多少种?
【输入格式】
一行一个整数 n，表示数字的个数。对于 50% 的数据，$n \leq 15$；对于 100% 的数据，$n \leq 1000$。
【输出格式】
一行一个整数，表示可能的出栈序列种数。

【样例输入】

```
3
```

【样例输出】

```
5
```

【样例解释】
当 $n = 3$ 时的出栈序列可能为:

```
1 2 3
1 3 2
2 1 3
2 3 1
3 2 1
```

【例】出栈序列问题

- 算法 1：搜索
- 入栈和出栈共 $2n$ 次操作，对于每次操作有两种选择：
  1. 将下一个数入栈；
  2. 将当前栈顶出栈 (栈非空)。
- 时间复杂度：$O(2^{2N})$。

【例】出栈序列问题

- 算法 2：递推
- 定义 $f(n)$ 表示进栈序列为 $1, 2, \cdots, n$ 的可能出栈序列总数。
- 考虑数字 1 在出栈序列的中的位置，设数字 1 在第 k 个位置出栈，那么整个序列的进出栈的过程就是：
  1. 数字 1 入栈；
  2. 数字 $2 \sim k$ 这 $k-1$ 个数按照某种顺序进出栈，这个方案数显然为 $f(k-1)$。
  3. 数字 1 出栈，正好排在第 k 个位置。
  4. 数字 $k+1 \sim n$ 这 $n-k$ 个数按照某种顺序进出栈，这个方案数显然为 $f(n-k)$。
- 于是整个问题就被数字 1 划分成了“$k-1$ 个数进出栈”和“$n-k$ 个数进出栈”这两个子问题，得到递推公式：

$$f(n) = \sum_{k=1}^{n} f(k-1) f(n-k)$$

- 时间复杂度：$\mathrm{O}(N^2)$。

【例】出栈序列问题

- 算法 3：动态规划
- 在任何一个时刻，我们实际上只关心有多少个数入栈、有多少个数已经出栈，并做出一步合法的操作，并不关心这些数具体是哪些。
- 定义 $f(i,j)$ 表示有 i 个数已经入栈，有 j 个数已经出栈的方案数。
- 显然有 $f(i,0)=1$；结束时所有的数都已经出入栈，所以目标为 $f(n,n)$。
- 对于状态 $f(i,j)$，它由入栈一个数的状态 $f(i-1,j)(i-1\geq j)$ 或者出栈一个数的状态 $f(i,j-1)$ 达到，状态转移方程为

$$f(i,j)=f(i-1,j)+f(i,j-1)$$

- 时间复杂度：O(N^2)。

【例】出栈序列问题

- 算法 4：计数问题
- 该问题等价于求第 n 项的卡特兰数，通项公式为

$$f(n) = \mathrm{C}_{2n}^{n} - \mathrm{C}_{2n}^{n-1} = \frac{\mathrm{C}_{2n}^{n}}{n+1}$$

- 不过一般计算时会选择用递推公式：

$$f(n) = \frac{f(n-1)(4n-2)}{n+1}$$

- 时间复杂度：O(N)。

【例】出栈序列问题

- 对于 50% 的数据，$n \leq 15$。上述四种方法均可使用。
- 对于 100% 的数据，$n \leq 1000$。上述哪些方法可用？代码实现难度如何？

【例】表达式

小 C 热衷于学习数理逻辑。有一天，他发现了一种特别的逻辑表达式。在这种逻辑表达式中，所有操作数都是变量，且它们的取值只能为 0 或 1，运算从左往右进行。如果表达式中有括号，则先计算括号内的子表达式的值。特别的，这种表达式有且仅有以下几种运算：

1. 与运算：`a & b`。当且仅当 a 和 b 的值都为 1 时，该表达式的值为 1。其余情况该表达式的值为 0。
2. 或运算：`a | b`。当且仅当 a 和 b 的值都为 0 时，该表达式的值为 0。其余情况该表达式的值为 1。
3. 取反运算：`!a`。当且仅当 a 的值为 0 时，该表达式的值为 1。其余情况该表达式的值为 0。

小 C 想知道，给定一个逻辑表达式和其中每一个操作数的初始取值后，再取反某一个操作数的值时，原表达式的值为多少。

【例】表达式

为了化简对表达式的处理，我们有如下约定：
表达式将采用后缀表达式的方式输入。后缀表达式的定义如下：

1. 如果 E 是一个操作数，则 E 的后缀表达式是它本身。
2. 如果 E 是 E_1 op E_2 形式的表达式，其中 op 是任何二元操作符，且优先级不高于 E_1、E_2 中括号外的操作符，则 E 的后缀式为 $E_1'E_2'$op，其中 E_1'、E_2' 分别为 E_1、E_2 的后缀式。
3. 如果 E 是 E_1 形式的表达式，则 E_1 的后缀式就是 E 的后缀式。

同时为了方便，输入中：

1. 与运算符（&）、或运算符（|）、取反运算符（!）的左右均有一个空格，但表达式末尾没有空格。
2. 操作数由小写字母 x 与一个正整数拼接而成，正整数表示这个变量的下标。例如：x10，表示下标为 10 的变量 x_{10}。数据保证每个变量在表达式中出现恰好一次。

【例】表达式

【输入格式】
第一行包含一个字符串 s，表示上文描述的表达式。
第二行包含一个正整数 n，表示表达式中变量的数量。表达式中变量的下标为 $1, 2, \cdots, n$。
第三行包含 n 个整数，第 i 个整数表示变量 x_i 的初值。
第四行包含一个正整数 q，表示询问的个数。
接下来 q 行，每行一个正整数，表示需要取反的变量的下标。注意，每一个询问的修改都是临时的，即之前询问中的修改不会对后续的询问造成影响。
数据保证输入的表达式合法。变量的初值为 0 或 1。
【输出格式】
输出一共有 q 行，每行一个 0 或 1，表示该询问下表达式的值。

【例】表达式

【样例 1 输入】

```
x1 x2 & x3 |
3
1 0 1
3
1
2
3
```

【样例 1 输出】

```
1
1
0
```

【样例 1 解释】

该后缀表达式的中缀表达式形式为 $(x_1 \& x_2)|x_3$。

- 对于第一次询问，将 x_1 的值取反。此时，三个操作数对应的赋值依次为 0，0，1。原表达式的值为 $(0\&0)|1 = 1$。
- 对于第二次询问，将 x_2 的值取反。此时，三个操作数对应的赋值依次为 1，1，1。原表达式的值为 $(1\&1)|1 = 1$。
- 对于第三次询问，将 x_3 的值取反。此时，三个操作数对应的赋值依次为 1，0，0。原表达式的值为 $(1\&0)|0 = 0$。

【例】表达式

【样例 2 输入】

```
x1 ! x2 x4 | x3 x5 ! & & ! &
5
0 1 0 1 1
3
1
3
5
```

【样例 2 输出】

```
0
1
1
```

【样例 2 解释】

该表达式的中缀表达式形式为 $(!x_1)\&(!((x_2|x_4)\&(x_3\&(!x_5))))$。

【数据范围与约定】

- 对于 20% 的数据，表达式中有且仅有与运算（&）或者或运算（|）。
- 对于另外 30% 的数据，$|s| \leq 1000$，$q \leq 1000$，$n \leq 1000$。
- 对于另外 20% 的数据，变量的初值全为 0 或全为 1。
- 对于 100% 的数据，$1 \leq |s| \leq 1 \times 10^6$，$1 \leq q \leq 1 \times 10^5$，$2 \leq n \leq 1 \times 10^5$。

其中，$|s|$ 表示字符串 s 的长度。

【例】表达式

- 后缀表达式求值，利用栈实现。
- 注意输入字符串 s 需要整行输入。
- 时间复杂度：$O(Q|s|)$，实际得分：30 分。

【例】表达式

- 每次都直接对原表达式进行后缀计算，分析字符串比较浪费时间。
- 利用原表达式建立后缀表达式树，对于每次修改，对表达式树进行后序遍历即可求解。
- 对于 `a & b` 运算，如果 $a = 0$ 或 $b = 0$，那么结果必然为 0；对于 `a | b` 运算，如果 $a = 1$ 或 $b = 1$，那么结果必然为 1，这就是逻辑运算的“短路”效应。可以利用该效应加快速度。
- 时间复杂度：$\mathrm{O}(QN)$；预计得分：75 分。

【例】表达式

- 不管如何修改，询问的答案只有 0 和 1。
- 考虑“短路“效应，如果修改的值 x_i 在被短路的子树下，那么它的修改对原始结果没有影响，否则会导致原始答案发生变化 (取反)。
- 因此需要提前预处理出 x_i 的修改对答案是否有影响，考虑到以 rt 为根的子树：
  - 如果根的运算符为!，那么它的孩子变化会对其有影响；
  - 如果根的运算符为 &，那么如果它的左孩子运算的结果为 0，那么整个右子树无论如何修改对答案都没有影响，同理如果它的右孩子运算结果为 0，那么整个左子树无论如何修改对答案都没有影响。
  - 如果根的运算符为 |，那么如果它的左孩子运算的结果为 1，那么整个右子树无论如何修改对答案都没有影响，同理如果它的右孩子运算结果为 1，那么整个左子树无论如何修改对答案都没有影响。
- 时间复杂度：$O(|s| + N + Q)$；预计得分：100 分。

【例】括号子串匹配个数

【题目描述】
给定一个括号字符串，其中只包含（和），请编程计算：字符串中满足括号匹配的不同子串个数。不同字符串定义：如果两个括号字符串的最左边或最右边的括号在原始串中位置不同即认为是两个不同的字符串。
【输入格式】
一行一个字符串 s。对于 40% 的数据，$|s| \leq 1200$；对于 80% 的数据，$|s| \leq 10^5$；对于 100% 的数据，$|s| \leq 2 \times 10^7$。
【输出格式】
一行一个整数，表示符合要求的子串个数。

【样例 1 输入】

```
(()())
```

【样例 1 输出】

```
4
```

【样例 2 输入】

```
))((
```

【样例 2 输出】

```
0
```

【样例 1 解释】
(()()) 共有 4 个满足条件子串，分别是 $s[1:6]$、$s[2:3]$、$s[4:5]$、$s[2:5]$。

【例】括号子串匹配个数

- 算法 1：枚举所有的子串，判断子串是否满足括号匹配。
- 时间复杂度 $O(N^3)$。

```
bool check(int l, int r)
{
    stack<char> st;
    for(int i = l; i <= r; ++i)
    {
        if(s[i] == '(') st.push(s[i]);
        else if(s[i] == ')' && st.empty()) return false;
        else st.pop();
    }
    return st.empty();
}

long long cnt = 0;
for(int i = 0; i < n; ++i)
  for(int j = i + 1; j < n; ++j)
    if(check(i, j)) ++cnt;
```

【例】括号子串匹配个数

- 算法 2：枚举括号子串的起点 $s[i]$，统计以 $s[i]$ 开头的满足条件的子串个数。
- 通过观察可以发现，如果从 $s[i]$ 开始进行括号匹配算法，在右括号匹配成功的情况下如果栈也为空，说明找到了一个满足条件的子串。
- 例如：子串 (((()))((只会出现 1 次栈空，而子串 ()()()((会出现 3 次栈空。
- 时间复杂度：$O(N^2)$。

```
long long cnt = 0;
for(int i = 0; i < n; ++i) {
    // 成功匹配的时候栈空的次数就是以 s[i] 开头的满足条件的子串个数
    stack<char> st;
    for(int j = i; j < n; ++j) {
        if(s[j] == '(') st.push(s[j]);
        else if(s[j] == ')' && st.empty()) break;  // 匹配失败后续也必然失败
        else {
            st.pop();
            if(st.empty()) ++cnt; // 右括号与左括号匹配并且栈空
        }
    }
}
```

【例】括号子串匹配个数

- 算法 3：递推
- 在括号匹配过程中，不考虑失败，每一个右括号都能找到与之匹配的左括号，例如 (()()) 和 ()()()，那么显然右括号和与之匹配的左括号之间就是一个匹配子串。
- 那么不妨在右括号匹配成功时统计因该匹配成功对答案的贡献，但是有两种情况：
  1. 右括号匹配的左括号是最长的，那么这个右括号对答案的贡献就是 1，例如))))(()()) 。
  2. 右括号匹配的左括号前面的子串也是匹配的，那么这个右括号对答案的贡献就是 1+ 与前面匹配的子串组成的子串的个数，例如，()()()，最后一个匹配的括号对答案的贡献有 ()、()()、()()()。

  但无论如何都与当前匹配的左括号的上一个字符有关。

(	(	(	)	)	(	)	(	)	)
0	0	0	1+0	1+0	0	1+1	0	1+2	1+0

【例】括号子串匹配个数

- 定义 $f[i]$ 表示以 $s[i]$ 为尾的括号匹配个数 (对答案的贡献)，那么
  - 如果 $s[i]=($ ，那么 $f[i]=0$;
  - 如果 $s[i]=)$，那么如果 $s[i]$ 没有匹配的左括号，那显然 $f[i]=0$，如果 $s[i]$ 成功匹配且匹配的左括号为 $s[l]$ ，那么 $f[i]=f[l-1]+1$ 。
- 时间复杂度：$O(N)$。

```
s = " " + s; // 为了方便递推，字符下标从 1 开始
stack<int> st;
for(int i = 1; i <= n; ++i) {
    if(s[i] == '(') f[i] = 0, st.push(i); // 新增左括号 贡献为 0
    else if(s[i] == ')' && st.empty()) f[i] = 0; // 例如 ()) 这个右括号贡献为 0
    else {
        f[i] = f[st.top() - 1] + 1;
        st.pop();
        // 成功和栈顶 ( 匹配 情况 1: (())()(( ) 情况 2: (())()( )
        // 这两种情况都和栈顶左括号的上一位置符号有关
    }
}
long long cnt = 0;
for(int i = 1; i <= n; ++i) cnt += f[i];
```

练习

- push,pop,getMin(COGS 2857)
- Editor(COGS 1745)
- 栈 [NOIP 2003](COGS 1033)
- 出栈序列统计 (COGS 1103)
- 表达式 [CSP-J 2020](COGS 3500)
- 逻辑表达式 [CSP-J 2022](COGS 2022)

练习

- 括号子串匹配个数 (COGS 3312)
- 最长括号匹配子串 (COGS 1744)
- 括号画家 (COGS 2316)
- 解压缩 (COGS 4036)
- 括号匹配 (COGS 1214)
- 括号树 [CSP-S 2019](COGS 3290)
- 消消乐 [CSP-S 2023](COGS 3932)
- 焰硝庭火 (COGS 3895)