栈与单调栈

后进先出 LIFO 原理、表达式求值与 $O(n)$ 单调栈

HGNU-ACM 实验室算法组

配套：数组模拟与 STL 栈 + 括号与后缀求值 + 单调栈
$O(n)$ 论证 + 4 道例题精讲





本章目录（1/2）

栈（Stack）后进先出与数组/STL 实现

括号匹配与后缀表达式求值

单调栈：$O(n)$ 找左侧/右侧首个更大或更小

承接上一篇《BFS 最短路与双向 A 星》，先从“后进先出”的栈结构讲起，再深入括号匹配、后缀求值与均摊 $O(n)$ 单调栈。

本章目录（2/2）

四大经典真题与模板实战精讲

常见错误、检查清单与练习安排

后两节逐题精讲《括弧匹配检验》《车厢调度》《后缀表达式的值》《右侧首个更大元素》，并附上高频防错检查清单与下一章预告。

承接上一篇：从“排队水波”切换到“死胡同回头”

- 在上一篇《BFS 最短路与双向 A 星》中，求最少步数用的是先来先出去的队列，像湖面水波一样一圈一圈往外扩。
- 但在做 DFS 回溯时，计算机悄悄帮你记着“刚才走过的岔路口”——每次走不通回头，必须先退回最近刚走过的那一步。
- 今天我们正式把这种“最后放进去的，最先拿出来”的容器请上台面，它就是数据结构里的栈（**Stack**）。

先做个小实验

依次把数字 $10, 20, 30$ 放进一个窄口筒子里，每次只能从筒口取最上面的数，拿出来的顺序一定是：$30 \to 20 \to 10$！

本页要点

队列讲究“先来先办”，栈讲究“后来先办”——凡是需要就近回溯、嵌套闭合的问题都归栈管。

生活引入：洗好的一摞盘子与单行死胡同

场景一：食堂一摞盘子

- 阿姨洗完盘子，总是把刚洗好的放到最顶上。
- 同学打饭取盘子时，也只能从最顶上拿走一个，不能从中间抽。
- 最早放进去的盘子压在最底下，最后才会被用上。

场景二：单车道死胡同

- 一条只容一车通行的死胡同，A 车先开进去，B 车后开进去。
- 想退出胡同，必须让后进去的 **B** 车先倒出来，A 车才能出来。
- 如果强行让 A 先出，两辆车就会堵死在里面。

本页要点

记住这个画面：入口和出口在同一端（顶部），只能从顶部放、从顶部拿！

为什么需要栈？暴力数组反复删改为什么会 TLE？

假设我们要维护一个“最近待处理任务”清单，有 $n = 10^5$ 次插入和删除最近一项的操作：

- 笨办法（无序数组每次从头找或中间删）：如果每次都要从头扫描到最后一个有效元素，或者在数组第 0 位插入并把后面全部右移，单次操作需要 $O(n)$ 步。
- 做 10^5 次操作总计 $\frac{n(n+1)}{2} \approx 5 \times 10^9$ 次运算，远超 1 秒约 10^8 次运算的极限，必定 **TLE**！
- 栈的做法：只用一个下标变量 `top` 记住“最顶上那格在哪”。放入只改 `top+1` 的格子，取出只把 `top` 减 1，每次只要 $O(1)$（**1** 步）！
- 10^5 次操作最多耗时 2×10^5 步，几毫秒轻松通过！

本页要点

栈把原本可能需要翻遍整张表或搬移元素的工作，压缩成只盯住“栈顶”这一个位置，耗时降到 $O(1)$！

新术语大白话：栈、栈顶、入栈、出栈、LIFO

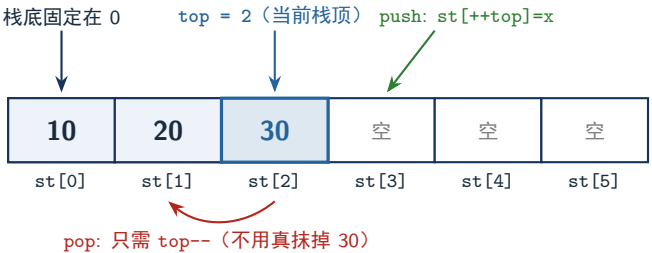
- 栈（**Stack**）：像窄口水桶一样的线性表，只允许在桶口那一端放东西和拿东西。
- 栈底（**Bottom**）：桶底最先放进去、最后才能出来的那一端（数组下标 0 那边）。
- 栈顶（**Top**）：桶口目前最上面那个元素所在的位置；新元素放这，取出也拿这。
- 入栈（**Push**）：把一个新元素压到栈顶上面（盘子往上摞一个）。
- 出栈（**Pop**）：把当前栈顶元素拿走，露出它下面那个元素。
- **LIFO**（**Last In, First Out**）：“后进先出”的英文缩写，就是栈的核心脾气。

本页要点

学栈只要记住四个动作：判空 `empty`、看顶 `top/back`、入栈 `push`、出栈 `pop`！

TikZ 图解：数组模拟栈的真实槽位与 top 指针

用一维数组 st[0..5] 和整型变量 top 模拟栈（初值 top = -1 表示空栈）：



本页要点

出栈时不用费事把 st[2] 清零，只要 top 退回 1，下次入栈就会自动覆盖 st[2]！

手算跟踪表：依次执行 6 步入栈与出栈（含判空终止）

初始 top = -1（代表栈空），跟踪每次操作后 top 与有效栈元素变化：

步序	操作	操作前 top	操作后 top	栈内有效元素（底 → 顶）与输出
1	push(10)	-1	0	[10]（先 ++top 再写入 st[0]）
2	push(20)	0	1	[10, 20]
3	pop()	1	0	取出顶 20，剩余 [10]
4	push(30)	0	1	[10, 30]（覆盖原 st[1]）
5	pop()	1	0	取出顶 30，剩余 [10]
6	pop()	0	-1	取出顶 10，剩余 []
7	判空终止	-1	-1	top == -1 成立，停止取顶（防 RE!）

本页要点

第 7 步判空最关键：top == -1 时再做 pop() 会变成 -2 或读负下标导致运行崩溃！

为什么对：数组模拟栈的不变量三段论证

为什么只维护一个整型下标 top，就能永远精准代表整个栈的顶部和元素个数？

阶段	保证成立的事实（不变量）
初始化	令 top = -1，此时区间 st[0..top] 为空，栈内恰好有 0 个元素。
保持	若每次 push(x) 都先 ++top 再赋值，pop() 仅在 top >= 0 时执行 top--，则每步后有效元素始终精确落在 st[0..top]，总数为 top + 1，且 st[top] 必为最后入栈者。
终止	当不再有合法元素可弹出时必有 top == -1，此时停下即可确保零越界。

本页要点

只要遵守“先判 top >= 0 再读/弹”，st[0..top] 永远是一段不重不漏的合法区间！

写法对比：手写数组栈 vs STL stack vs vector

竞赛里有三种常见写法，核心操作全部都是 $O(1)$：

操作需求	手写数组（top=-1）	stack<int> st	vector<int> st
判空	`top == -1`	`st.empty()`	`st.empty()`
元素个数	`top + 1`	`st.size()`	`st.size()`
入栈	`st[++top] = x;`	`st.push(x);`	`st.push_back(x);`
看栈顶	`st[top]`	`st.top()`	`st.back()`
出栈	`top--;`	`st.pop();`	`st.pop_back();`
能否遍历调试	能（0..top）	不能直接遍历	能（范围 for）

本页要点

平时解题最推荐 `vector<int> st`：既自动扩容防爆数组，又支持下标访问与调试打印！

模板代码（1/2）：手写数组模拟栈

```
const int MAXN = 100005;
int st[MAXN], top = -1;                  // top = -1 代表栈为空

void push_val(int x) {
    st[++top] = x;                       // 先让 top 加 1，再存入新元素
}
int read_top() {
    if (top == -1) return -1;            // 访问前必须先判空，防负数下标 RE
    return st[top];
}
void pop_val() {
    if (top >= 0) top--;                 // 只在非空时退回一格
}
```

本页要点

手写数组在常数要求极严时最快；若有多组测试数据，每组开头务必重新赋值 `top = -1`！

模板代码（2/2）：STL vector / stack 安全写法

```cpp
#include <bits/stdc++.h>
using namespace std;
int main() {
    vector<int> st;                    // 用 vector 充当动态栈
    st.push_back(10);                  // 入栈：10
    st.push_back(20);                  // 入栈：20
    if (!st.empty()) {                 // 读取与弹出前永远紧贴着判空！
        cout << st.back() << '\n';     // 查看栈顶：输出 20
        st.pop_back();                 // 弹出栈顶 20
    }
    cout << st.size() << '\n';         // 当前剩余元素个数：1
    return 0;
}
```

本页要点

pop() 无返回值！想拿栈顶必须先读 top()/back() 再弹出。

上机验证：3 组必测数据与反事实追问

把上面代码输入以下 3 组操作序列，核对输出是否完全符合预期：

1. 普通入出序列：依次 `push(10)`，`push(20)`，`pop()`，`push(30)`，`top()` $\Longrightarrow$ 应输出 30，此时 `size == 2`。
2. 单元素清空边界：依次 `push(5)`，`pop()`，`empty()` $\Longrightarrow$ 应返回 `true`（数组版 `top == -1`）。
3. 空栈安全拦截：初始为空时尝试读取栈顶或弹栈 $\Longrightarrow$ 因为有 `if (!st.empty())` 保护，程序不崩溃且忽略非法弹出。

反事实追问

如果数组模拟时把 `top` 初值改成 0（想用 1-based），却忘了把入栈语句从 `st[++top] = x` 改掉，会怎样？

答：第一个元素存进 `st[1]`，但 `st[0]` 空着；若判空仍写 `top == -1` 会永远判非空！判空必须同步改成 `top == 0`。

易错对照：栈操作最致命的 4 种崩溃与 WA

错误类型	常见错误写法	后果与正确写法
空栈读顶或弹栈	不写 `!st.empty()` 直接 `st.top()`	立刻越界或段错误 **RE**；必须先判 `!st.empty()`
误以为 `pop()` 返回值	写 `int x = st.pop();`	编译报错 **CE**；应先 `x = st.top();` 再 `st.pop();`
多组数据未清空	只写 `top = -1;` 在 `main` 开头	下一组读到上一组残余元素 **WA**；每组重置 `top = -1`
数组开小	`int st[1005];` 处理 10^5 数据	栈深超限写穿内存 **RE**；按最大入栈量开够或用 `vector`

本页要点

任何时候只要在代码里看到 `.top()`、`.back()`、`.pop()`，眼睛立刻往左看有没有 `!empty()`！

生活引入：只数“左括号和右括号一样多”行不行？

试看这两个短串，左括号和右括号的数量完全相等，它们合法吗？

- 串 A：`())(` ——左圆括号 2 个，右圆括号 2 个，但第 2 个 `')'` 前面根本没有待匹配的 `'('`！
- 串 B：`[(])` ——方括号 1 对、圆括号 1 对，可里面拆不开：`'('` 还没关闭，外面的 `']'` 就强行插队闭合，交错打架！

- 括号合法的真正规律叫就近配对：每当遇到一个右括号，它必须和前面距离它最近、且尚未被闭合的左括号正好配成一对！
- 这正好就是栈的后进先出（**LIFO**）：最后打开的左括号蹲在栈顶，最先等着被右括号消掉。

本页要点

括号不能只靠计数器，因为多种括号嵌套时，必须记住“最近还没关掉的那一个”是谁！

暴力痛点：每次扫描消去内层括号为什么会 TLE？

如果不使用栈，还有一种直觉写法：反复从头扫描字符串，找到挨在一起的 "()" 或 "[]" 就把它们从串里删掉，直到删无可删。

- 举个坏例子：输入长度 $n = 10^5$ 的深层嵌套串 "((...(())...))"（共 5×10^4 层）。
- 第 1 遍扫描走 10^5 步，只在正中间删掉 1 对括号；第 2 遍再走 $10^5 - 2$ 步删掉第 2 对……
- 总步数约为 $\frac{10^5 \times 5 \times 10^4}{2} = 2.5 \times 10^9$ 步，远超 1 秒限制，必定 **TLE**！
- 改用栈后，每个字符只进栈 1 次、出栈 1 次，从头到尾只需扫一趟，时间复杂度降到 $O(n)$！

本页要点

把“反复在串中寻找最近左括号”换成“栈顶直接存着最近左括号”，复杂度从 $O(n^2)$ 降到 $O(n)$！

新术语大白话：就近配对、中缀表达式、后缀表达式

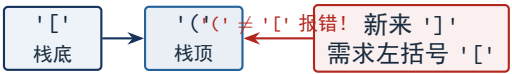
- 就近配对：右括号只跟“当前还没闭合的左括号里最后出现的那一个”配对，匹配成功就把两人一起消掉。
- 中缀表达式：平时数学课写的式子，运算符夹在两个数中间，比如 `16 - 9 * (4 + 3)`，需要靠括号和先乘除后加减决定顺序。
- 后缀表达式（逆波兰表达式）：把运算符挪到两个数字后面，比如 `16 9 4 3 + * -`。它完全不需要括号，从左到右一趟就能算！
- 右操作数与左操作数：做减法 $a - b$ 时，a 在左边叫左操作数，b 在右边叫右操作数。入栈顺序是先 a 后 b，所以先弹出的是右操作数 b！

本页要点

后缀表达式没有括号也没有优先级烦恼，计算机最喜欢用栈从左到右直接求值！

TikZ 图解：括号匹配 [(]) 在第 3 步当场抓错

从左到右扫描 "[(])"，用栈记录“等待闭合的左括号”：



第 1 步：压入 '[' 第 2 步：压入 '(' 第 3 步：立刻输出 Wrong

本页要点

遇到右括号时只看栈顶：对得上就 pop，栈空或对不上立刻判定不合法！

手算跟踪表：合法嵌套串 [([])] 完整 7 步（含结尾判空）

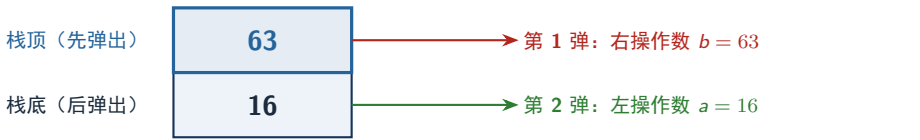
步序	读入字符	动作与判定	动作后栈内容（底 → 顶）
1	'['	左括号，直接入栈	['[']
2	'('	左括号，直接入栈	['[', '(']
3	'['	左括号，直接入栈	['[', '(', '[']
4	']'	栈非空且栈顶为 '['，匹配弹栈	['[', '(']
5	')'	栈非空且栈顶为 '('，匹配弹栈	['[']
6	']'	栈非空且栈顶为 '['，匹配弹栈	[]
7	串结束	检查 st.empty() 为真	[]（所有左括号清空，输出 **OK**）

本页要点

千万不能少了第 7 步！如果输入是 "(((", 前 3 步都不报错，全靠结尾 st.empty() 抓出错误！

TikZ 图解：后缀求值遇到 '-' 时，谁是被减数？

计算 `16 9 4 3 + * - @` 到最后一步减号 `'-'` 时的栈状态：



本页要点

加和乘满足交换律写反没事，但减法和除法必须严格用“后弹出的 a”去减/除以“先弹出的 b”！

手算跟踪表：16 9 4 3 + * - @ 逐项解析与求值

步序	读到完整项	动作	栈内数值（底 → 顶）
1	16	拼出多位数 16，压栈	[16]
2	9	拼出数字 9，压栈	[16，9]
3	4	拼出数字 4，压栈	[16，9，4]
4	3	拼出数字 3，压栈	[16，9，4，3]
5	'+'	先弹 $b=3$,后弹 $a=4$,压入 $4+3=7$	[16，9，7]
6	'*'	先弹 $b=7$,后弹 $a=9$,压入 $9\times7=63$	[16，63]
7	'-'	先弹 $b=63$，后弹 $a=16$，压入 $16-63=-47$	[-47]
8	'@'	终止符，退出循环并输出栈顶	输出 -47

本页要点

看到 16 后面紧跟空格时才把拼好的 16 压栈；不能把 '1' 和 '6' 拆成两个数！

为什么对：括号匹配与后缀求值的不变量

阶段	括号匹配的不变量	后缀求值的不变量
初始化	扫描前栈空，表示尚未有待闭合的左括号。	扫描前栈空，表示尚未读入任何运算数。
保持	每读一个字符：左括号入栈；右括号必须消去栈顶对应左括号。栈中始终按出现顺序保存“未闭合左括号”。	每读一个数字压入栈；每遇运算符必把最近两个已算好的子结果弹出合并为一个新结果压回。
终止	串扫完且从未冲突，当且仅当栈空（无剩余左括号）时全串合法。	遇 @ 停止，合法表达式此时栈中恰好只剩 1 个值，即为整式结果。

本页要点

“栈里永远只留还没配对/还没被运算符吃掉的东西”，这正是栈处理嵌套与表达式的根本保证！

模板代码（1/2）：括号匹配核心判断

```
vector<char> st;
for (char ch : s) {
    if (ch == '(' || ch == '[') {
        st.push_back(ch);                          // 左括号进栈等待配对
    } else if (ch == ')' || ch == ']') {
        char need = (ch == ')' ? '(' : '[');       // 期待的左括号类型
        if (st.empty() || st.back() != need) {     // 栈空或栈顶不匹配立刻报错
            cout << "Wrong\n"; return 0;
        }
        st.pop_back();                             // 配对成功，弹出栈顶
    }
}
cout << (st.empty() ? "OK" : "Wrong") << '\n';  // 结尾必须判空！
```

本页要点

这段核心代码只有 13 行，把“右括号找栈顶”和“结尾判空”这两道防线守死就绝不会漏。

模板代码（2/3）：后缀表达式数字拼装

```
vector<long long> st; long long cur = 0; bool in_num = false;
for (char c : text) {
    if (c == '@') break;                          // 遇到结束符立即停止
    if (c >= '0' && c <= '9') {
        cur = cur * 10 + (c - '0');               // 拼装多位数
        in_num = true;
    } else if (isspace((unsigned char)c)) {
        if (in_num) { st.push_back(cur); cur = 0; in_num = false; }
    } else {                                      // 遇到运算符（见下一页）
```

本页要点

遇到空格时，若 in_num 为真就把拼好的整数 cur 压栈并清零复位！

模板代码（3/3）：后缀表达式运算符弹栈

```
        if (in_num) { st.push_back(cur); cur = 0; in_num = false; }
        long long b = st.back(); st.pop_back(); // 第一弹：右操作数
        long long a = st.back(); st.pop_back(); // 第二弹：左操作数
        if (c == '+') st.push_back(a + b);
        else if (c == '-') st.push_back(a - b);
        else if (c == '*') st.push_back(a * b);
        else if (c == '/') st.push_back(a / b);
    }
}
```

本页要点

运算符分支前也要把 in_num 收尾压栈，防止数字与运算符之间没空格时漏掉最后一个数字！

上机验证：3 组测试与反事实追问

1. 正常嵌套：输入 `[([])]` $\Longrightarrow$ 输出 `OK`；后缀 `16 9 4 3 +*-@` $\Longrightarrow$ 输出 `-47`。
2. 全左括号与基本运算边界：括号输入 `(((` $\Longrightarrow$ 输出 `Wrong`；后缀输入 `10 3 -@` $\Longrightarrow$ 输出 `7`。
3. 单右括号边界：输入 `)` $\Longrightarrow$ 在第 1 步 `st.empty()` 为真，立刻安全输出 `Wrong`（不崩溃）。

反事实追问

1. 如果括号代码最后一行改成直接 `cout << "OK";`，输入 `"((("` 会发生什么？
答：循环中根本不会进右括号分支，程序会错误地输出 `OK`！
2. 如果后缀求值把减法写成了 `b - a`，样例 `16 9 4 3 +*-@` 会错成多少？
答：算出 $63 - 16 = 47$，丢掉负号导致 **WA**！

易错对照：括号与后缀表达式的 4 大错因

错误类型	常见错误写法	评测结果与正确写法
空栈取顶	只写 `if (st.back() != need)`	遇 `")"` 时空栈崩溃 **RE**；先写 `st.empty() || ...`
结尾漏判空栈	循环安全跑完就直接输出 `OK`	对 `"((("` 误判 **WA**；结尾写 `st.empty() ? "OK" : "Wrong"`
减除顺序写反	先弹出的值当左操作数 `b - a`	结果正负号/商全错 **WA**；第一弹给 b，第二弹给 a
多位数拆分错误	逐字符 `st.push_back(c - '0')`	把 16 拆成 1 和 6 导致 **WA**；用 `cur = cur*10 + (c-'0')` 拼完整

本页要点

写完表达式或括号程序，必测三件套：`")"`（空栈保护）、`"((("`（结尾判空）、`"5 2 -@"`（减法方向）！

小例子引入：奶牛向右看，谁第一个挡住你？

手动试做：给 5 个身高 $a = [2, 1, 5, 3, 6]$（下标 $1 \sim 5$），找每个人右侧第一个严格比自己高的人下标（没有就写 0）

- 第 1 个（高 2）：右边是 1（不够高），再往右是第 3 个（高 $5 > 2$） $\Longrightarrow$ 答案 3。
- 第 2 个（高 1）：紧挨着右边第 3 个（高 $5 > 1$） $\Longrightarrow$ 答案 3。
- 第 3 个（高 5）：右边第 4 个高 3 不行，第 5 个高 $6 > 5$ $\Longrightarrow$ 答案 5。
- 第 4 个（高 3）：右边第 5 个高 $6 > 3$ $\Longrightarrow$ 答案 5。
- 第 5 个（高 6）：右边没人了 $\Longrightarrow$ 答案 0。

本页要点

答案数组为 `[3，3，5，5，0]`。手工只有 5 个数可以一眼看出，如果换成 10^5 个数怎么办？

暴力痛点：双重循环向右扫描为什么会 TLE？

最直观的暴力写法：对每个位置 $i = 1 \dots n$，开一层内循环 $j = i+1 \dots n$ 挨个往右找，直到发现 $a[j] > a[i]$ 就 `break`。

- 当 $n = 10^5$ 且序列是严格降序 $[10^5, 10^5 - 1, \dots, 1]$ 时，每一个人向右都找不到更高的人！
- 内循环每次都要跑到末尾才停：第 1 个人查 $n-1$ 次，第 2 个人查 $n-2$ 次……
- 总比较次数 $= \frac{n(n-1)}{2} \approx 5 \times 10^9$ 次，远远超过 1 秒 10^8 次运算的极限，必定 **TLE**！
- 病根在于：前面每个人找过的路，后面的人完全没有复用，每次都重新从头扫！

本页要点

要把 $O(n^2)$ 降到 $O(n)$，必须让“已经被高个子挡住的矮个子”趁早离开舞台，不再重复比较！

新术语大白话：单调栈、弹栈结算、均摊 $O(n)$

- 单调栈（**Monotonic Stack**）：还是一个普通的栈，只是我们定下一条家规——从栈底到栈顶的元素大小必须永远保持单调（比如从大到小排）。
- 破坏单调性就踢人：每当新来一个数准备入栈，如果它打破了家规（比如比栈顶还要高），就把违反家规的旧栈顶赶出栈，直到栈顶符合家规或栈空。
- 弹栈结算：旧栈顶被新元素 i 踢出去的那一刻，刚好说明“i 就是它右边第一个比它大的元素”，顺手记下答案！
- 均摊 $O(n)$：虽然某一次可能一口气弹出好几个元素，但每个人总共只进栈 1 次、出栈 1 次，平均每人只花常数时间。

本页要点

单调栈没有发明新结构，只是在普通栈入栈前加了一个 `while` 循环维持“单调排队秩序”！

核心洞察：为什么被高个子跨过的矮个子“再也没用了”？

假设你前面站着下标 1（身高 2）和下标 2（身高 1），现在新来了下标 3（身高 5）：

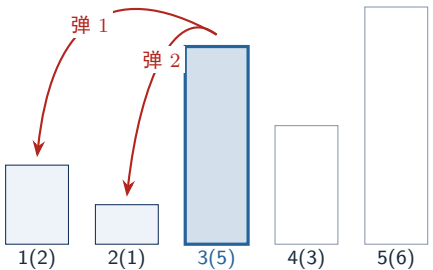
- 对下标 $1,2$ 自己而言：它们向右终于等到了比自己高的 3（高 5），它们的任务已经完成（答案锁定为 3）。
- 对未来还没出场的人 $k>3$ 而言：如果往左找，只要 5 挡在前面，根本轮不到更矮更远的 2 和 1 说话！
- 因此，把 2 和 1 永久弹出栈，既结算了它们的答案，又把栈精简成只有 $[3_{(5)}]$ 的单调序列！

本页要点

“答案已定、未来又挡不住谁”的元素留在栈里纯属累赘，新元素一到立刻清理出栈！

TikZ 图解：a = [2, 1, 5, 3, 6] 在 i=3 处的弹栈结算

当 $i = 3$（身高 5）到达时，栈里原本蹲着按高度递减的下标 $[1_{(2)}, 2_{(1)}]$：



原栈：$[1_{(2)}, 2_{(1)}]$

新来 $3_{(5)}$：踢掉 $2, 1$
记录 ans[2]=3，ans[1]=3

入栈后：$[3_{(5)}]$（单调递减）

本页要点

因为栈底到栈顶原本是从高到矮，新来的 5 只要不断把比它矮的栈顶踢掉，留下的栈天然还是从高到矮！

手算跟踪表：a = [2，1，5，3，6] 完整 6 步（含结束清算）

栈 st 内只存下标（括号内注明对应的身高，方便心算）：

i ($a[i]$)	栈顶 $<$ $a[i]$？	弹栈并写 ans	动作后 st	当前 ans[1..5]
1 (2)	栈空	无	$[1_{(2)}]$	[0,0,0,0,0]
2 (1)	假（$2<1$ 假）	无	$[1_{(2)}, 2_{(1)}]$	[0,0,0,0,0]
3 (5)	真（$1<5, 2<5$）	弹 2 记 3，弹 1 记 3	$[3_{(5)}]$	[3,3,0,0,0]
4 (3)	假（$5<3$ 假）	无	$[3_{(5)}, 4_{(3)}]$	[3,3,0,0,0]
5 (6)	真（$3<6, 5<6$）	弹 4 记 5，弹 3 记 5	$[5_{(6)}]$	[3,3,5,5,0]
结束	循环终止	留在栈里的 5 无更大值	$[5_{(6)}]$ 保持默认 0	[3,3,5,5,0]

本页要点

最后留在栈里的元素（比如 5）始终没等到比它高的后来者，因为 ans 事先全初始化为 0，不用再单独清空栈！

均摊 $O(n)$ 论证：为什么 for 套 while 不是 $O(n^2)$？

很多同学看到 for 循环里面套了一个 while 循环，就习惯性以为是 $O(n^2)$，其实大错特错！

- 数一数每个下标 $i \in \{1, 2, \ldots, n\}$ 一辈子进出栈的次数：
  1. 在 for 循环第 i 轮末尾，下标 i 只会入栈恰好 1 次，总入栈次数 $= n$。
  2. 每个下标一旦在 while 里被 pop_back() 弹出，就再也不会回到栈里，因此最多被弹出 1 次！
  3. 整个程序跑完，内层 while 循环体在所有轮次加起来最多执行 n 次！
- 总操作次数 $\leq n$（入栈） $+ n$（出栈） $= 2n$，因此时间复杂度严格为 $O(n)$！

本页要点

不要只看循环有几层，要看每个元素被操作多少次：进 1 次、出至多 1 次 $\Longrightarrow$ 严格 $O(n)$！

循环不变量：为什么弹栈记下的答案一定是“最近且更大”？

阶段	保证成立的事实（不变量）
初始化	处理 $i = 1$ 前，栈空且所有 ans 为 0，显然单调不增成立。
保持	每轮处理完 i 后：(1) 栈中下标从底到顶严格递增，对应数值单调不增（即 $a[\text{底}] \geq \cdots \geq a[\text{顶}]$）；(2) 所有已经出栈的下标都已精确填好右侧首个更大位置；(3) 仍在栈中的下标，在它右侧到 i 为止尚未出现严格大于它的数。
终止	当 $i = n + 1$ 退出循环时，出栈者答案全对，仍留栈者右侧到 n 确实没有更大数，保持 0 完全正确！

本页要点

如果下标 j 在 i 之前还没出栈，说明 $(j, i-1)$ 里没人大于 $a[j]$；一旦 $a[i] > a[j]$，则 i 必是右边第一个！

关键设计：为什么栈里存“下标 i”而不存“数值 a[i]”？

初学单调栈，十个人有八个会把 `st.push_back(a[i])` 写出来，为什么必须改成 `st.push_back(i)`？

1. 写答案需要位置：本题要填的是 `ans[idx] = i`，如果你只往栈里塞了数值 2，当 5 来踢掉 2 时，你根本不知道这个 2 原来站在第几号位置！
2. 算距离/宽度必须有下标：在《柱状图最大矩形》《奶牛排队》等题里，需要用下标相减算矩形宽度。
3. 有下标就能 $O(1)$ 查数值：只要手里握着下标 `idx`，写 `a[idx]` 立刻就能拿到数值；反过来只存数值却找不回下标！

本页要点

单调栈与单调队列的黄金铁律：栈/队列里永远存下标，比较大小时写成 `a[st.back()]`！

四向变形速查表：左/右 × 更大/更小 × 严格/不严格

单调栈有 4 个常见方向，只需调整“while 踢人条件”和“什么时候记答案”：

目标问题	栈内顺序（底 → 顶）	while 踢栈顶条件	何时记答案
右侧首个严格更大（$>$）	单调不增（$\geq$）	`a[st.back()] < a[i]`	弹栈时记 `ans[top]=i`
右侧首个不小于（$\geq$）	严格递减（$>$）	`a[st.back()] <= a[i]`	弹栈时记 `ans[top]=i`
左侧首个严格更大（$>$）	严格递减（$>$）	`a[st.back()] <= a[i]`	入栈前看 `st.back()`
左侧首个严格更小（$<$）	严格递增（$<$）	`a[st.back()] >= a[i]`	入栈前看 `st.back()`

本页要点

记住口诀：找“更大”就让栈从底到顶递减（把小的全踢掉）；找“左侧”在踢完后看此时新栈顶！

模板代码（1/2）：右侧首个严格更大（核心循环）

```cpp
vector<int> ans(n + 1, 0), st;                  // 找不到默认 0；st 存下标
for (int i = 1; i <= n; ++i) {
    // 若新元素严格大于栈顶对应值，说明栈顶元素找到了答案
    while (!st.empty() && a[st.back()] < a[i]) {
        ans[st.back()] = i;                     // 记下右侧首个更大下标 i
        st.pop_back();                          // 功成身退，弹出栈顶
    }
    st.push_back(i);                            // 当前下标入栈
}
```

本页要点

这 9 行代码就完成了全部求解！重点看第 4 行：`!st.empty()` 写在前面，比较用的是 `a[st.back()]`。

模板代码（2/2）：只改两行变成“找左侧首个严格更大”

```
vector<int> ans(n + 1, 0), st;                    // 找不到左侧更大则默认 0
for (int i = 1; i <= n; ++i) {
    // 把所有 <= a[i] 的栈顶踢走，剩下的栈顶就一定严格 > a[i]
    while (!st.empty() && a[st.back()] <= a[i]) {
        st.pop_back();
    }
    ans[i] = st.empty() ? 0 : st.back();          // 此时栈顶就是左边最近且 > a[i] 的
    人
    st.push_back(i);                              // 把自己压入栈供右边的人查询
}
```

本页要点

对比两页代码：找“右侧”是在弹栈时给别人记答案；找“左侧”是踢掉矮个子后给自己找现成栈顶！

上机验证：3 组测试与反事实追问

使用上面“右侧首个严格更大”模板运行以下 3 组输入，核对输出：

1. 普通混合：$n=5, a=[2,1,5,3,6] \implies$ 应输出 `3 3 5 5 0`。
2. 严格降序边界：$n=4, a=[9,7,4,1] \implies$ 全程没人被弹出，应输出 `0 0 0 0`。
3. 全部相等边界：$n=4, a=[3,3,3,3] \implies$ 因为要求“严格大于”，谁也踢不动谁，应输出 `0 0 0 0`。

反事实追问

如果把第 4 行的严格小于 `a[st.back()] < a[i]` 不小心改成了 `<=`，在全相等输入 `[3, 3, 3, 3]` 上输出会变成什么？

答：第 2 个 3 会把第 1 个 3 踢掉，输出变成 `2 3 4 0`（找成了“右侧首个不小于”）！

易错对照与算法边界：什么时候该换“单调队列”？

错误 / 场景	常见错误写法	正确做法与原因
栈内误存数值	`st.push_back(a[i]);`	无法写回 `ans[idx]` 也无法算距离，必须存下标 `i`
比较漏写 `a[...]`	`while (... && st.back() < a[i])`	把下标直接拿去跟数值比了，必 **WA**！写 `a[st.back()]`
忘记判非空	漏写 `!st.empty()`	栈空时读 `st.back()` 直接崩溃 **RE**
严格/非严格混写	题目要严格更大却写 `<=`	相等元素会互相踢出导致 **WA**

与下一章《队列与单调队列》的衔接边界

单调栈解决的是没有固定长度限制的“左/右侧首个更大或更小”；如果题目限定了固定窗口长度 k（比如求每连续 k 个数的最大值），旧元素过 k 步必须从队头踢走，那就得用下一章的单调队列！

例 1 | 括弧匹配检验（一本通 1354）[入门]：先读懂要求

大白话题意

给你一行只含圆括号 () 和方括号 [] 的字符串（字符个数小于 255），判断括号的嵌套和配对是否全部合法。

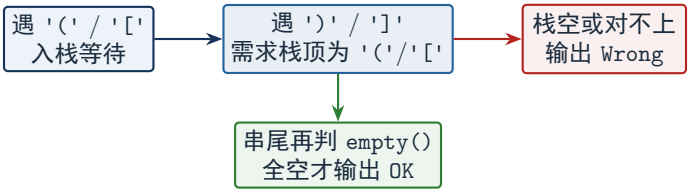
- 输入：一行括号字符串（例如 [(] 或 [([][])]）。
- 处理：检验每一个右括号能否与前面最近未闭合的左括号配对，且扫描结束时没有剩余未闭合的左括号。
- 输出：合法输出 OK，不合法输出 Wrong（注意大小写完全一致）。

本页要点

样例 [(] 应输出 Wrong；样例 [([][])] 应输出 OK。

例 1 | 怎么想到的？用栈管住“最近打开的左括号”

- 从左到右扫描字符：左括号就像“开了一扇门”，暂时先记在栈顶；右括号就像“找门来关”，必须刚好关掉最晚打开的那扇门！



本页要点

只有 '(' 与 ')'、'[' 与 ']' 才配对；遇到其他空白符或回车直接忽略即可安全兼容评测机换行符。

例 1 | 样例手推跟踪表：对比 [(]) 与 (((

样例	第几步	读入字符	栈状态（底 → 顶）与判定
[(])	1	'['	入栈 → ['[']
[(])	2	'('	入栈 → ['[', '(']
[(])	3	']'	需求 '['，但栈顶是 '('，不匹配 ⟹ 输出 **Wrong** 并结束
((()	1∼3	'(','(','('	连续入栈 → ['(', '(', '(']
((()	4	')'	与栈顶 '(' 匹配并弹出 → ['(', '(']
((()	结束	串尾	检查 st.empty() 为假（还剩 2 个左括号）⟹ 输出 **Wrong**

本页要点

两类失败方式全都在表里了：中途右括号失配（或空栈），以及结尾栈没清空！

例 1 | 核心步骤与相对模板改动

按顺序计算	纸上得到	核对问题
读整行串并初始化空栈	`getline(cin, s)`	串长 < 255，遇左括号直接入栈吗？
右括号算出期待左括号	`need = (ch==')'?'(':'[')`	判断 `st.empty()` 写在 `st.back()` 前面吗？
循环结束后终检	`st.empty() ? "OK" : "Wrong"`	全左括号有没有被漏判成 OK？

相对基础栈模板改了哪里？

1. 用字符变量 `need` 把 `')'` 和 `']'` 的判断合并成同一套逻辑，不写两遍重复 `if`。
2. 使用 `else if (ch == ')' || ch == ']')` 过滤行尾可能存在的 `'\r'` 等无关字符。

例 1 | 完整可编译代码（1/2）：读入与初始化

以下源码与 code/bracket_check.cpp 完全一致：

```cpp
#include <bits/stdc++.h>
using namespace std;
// 一本通 1354 括弧匹配检验：成功输出 OK，否则 Wrong
int main() {
    ios::sync_with_stdio(false);
    cin.tie(nullptr);
    string s;
    if (!getline(cin, s)) return 0;
    vector<char> st;
```

本页要点

用 getline(cin, s) 读入整行，即使测试数据里有空行或特殊空白也不会错位。

例 1 | 完整可编译代码（2/2）：括号配对与结尾判空

```
    for (char ch : s) {
        if (ch == '(' || ch == '[') {
            st.push_back(ch);                          // 左括号入栈等待匹配
        } else if (ch == ')' || ch == ']') {
            char need = (ch == ')' ? '(' : '[');
            if (st.empty() || st.back() != need) { cout << "Wrong\n"; return 0; }
            st.pop_back();                             // 与栈顶配对成功，弹栈
        }
    }
    cout << (st.empty() ? "OK" : "Wrong") << '\n';  // 必须栈完全空才合法
    return 0;
}
```

本页要点

`st.empty() || st.back() != need` 先判空再读栈顶，防止空栈 RE！

例 1 | 易错点与 3 组自测边界

- 易错 **1**：输出拼写成 "Ok" 或 "WRONG"，大小写与题面 OK / Wrong 不符导致 **WA**。
- 易错 **2**：只写 if (st.back() != need) 忘了前面的 st.empty() ||，遇到开头是右括号就 **RE**。
- 易错 **3**：结束时直接打印 OK，忘了检查 st.empty()。

3 组必跑自测边界

1. 交错样例：[(]) $\Longrightarrow$ Wrong（测试栈顶不匹配拦截）。
2. 单右括号：) $\Longrightarrow$ Wrong（测试空栈不崩溃）。
3. 全左括号：[[(($\Longrightarrow$ Wrong（测试结尾判空）。

例 2 | 车厢调度（1357）[基础]：先读懂要求

大白话题意

n 节车厢（$n \le 1000$）按照 $1, 2, 3, \ldots, n$ 的固定顺序从 A 方向驶入。中间有一个只能一端进出的死胡同中转站 C（可以停任意多节车厢）。车厢可以随时开进 C，也可以从 C 退到出口 B（但进去 B 就不能再退回 C）。问能否让车厢以指定顺序 $a_1, a_2, \ldots, a_n$ 从 B 驶出？

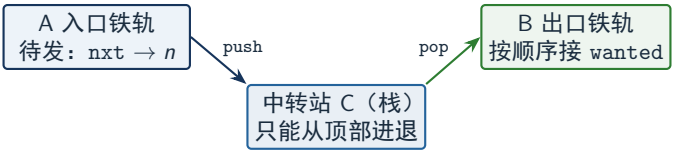
- 输入：第 1 行整数 n；第 2 行 n 个整数，代表期望的出站序列 $a_1 \ldots a_n$。
- 处理：用栈模拟中转站 C，判断能否按给定次序逐一从站顶调度出去。
- 输出：能做到输出 `YES`，否则输出 `NO`（全大写）。

本页要点

样例输入 `5` / `5 4 3 2 1`，全部推进 C 再倒序弹出即可，输出 `YES`。

例 2 | 怎么想到的？铁路中转站 C 就是后进先出栈

如果暴力枚举每节车厢“何时进站、何时出站”，$n = 1000$ 有指数级方案必 **TLE**。仔细想一想，其实根本不用选择：



- 假设当前想让编号 `wanted` 出站：如果它还没进过中转站 C（即 `nxt <= wanted`），那就必须先把 `nxt` 到 `wanted` 一节节推入 C！
- 推完之后，`wanted` 要么正好停在 C 的栈顶（立刻弹出到 B），要么被后面更大号的车厢死死压在底下——只要栈顶不是 `wanted`，就绝对无解！

本页要点

用变量 `nxt` 记录下一个还没进站的车厢号；要出 `wanted` 时，把 $\leq$ `wanted` 的新车全推进栈再看栈顶！

例 2 | 样例手推跟踪表：对比 5 4 3 2 1 与反例 3 1 2

先走一遍失败反例 $n=3$，目标出站序列 3 1 2（初始 nxt = 1，栈 siding = []）：

第几辆	目标 wanted	入栈过程（nxt<=wanted）	检查栈顶前 siding	结果与说明
1	3	压入 $1,2,3$（nxt 变 4）	[1, 2, 3]	栈顶是 3，匹配并弹出 → [1, 2]
2	1	$4 \le 1$ 假，不入栈	[1, 2]	栈顶是 $2 \neq 1$！**1** 被 **2** 压死，输出 **NO**
对照	5 4 3 2 1	第 1 步压入 $1 \sim 5$	[1,2,3,4,5]	随后连续 5 次恰好从顶弹出 $5,4,3,2,1$ →**YES**

本页要点

口诀：想让某车出站时，比它小的所有未入站车厢都必须先入站；如果已经入站的更大车厢挡在它上面就无解！

例 2 | 核心步骤与相对模板改动

按顺序计算	纸上得到	核对问题
设待入站号 `nxt = 1`	空栈 `siding`	每读一个 `wanted` 都需要重置 `nxt` 吗？（不需要，全局递增！）
推进未进站车厢	`while (nxt <= wanted)`	是否在推完后立刻检查了 `siding.empty()`？
比对栈顶并出站	`siding.back() == wanted`	一旦失败是不是立刻输出 `NO` 并退出？

相对基础栈模板改了哪里？

1. 不是简单读完再弹，而是“一边读目标 `wanted`，一边按需用 `while (nxt <= wanted)` 补进栈”。
2. 若中途输出 `NO` 直接 `return 0`，不需要再读完后面的数字。

例 2 | 完整可编译代码（1/2）：读入规模与初始化

以下源码与 code/train_dispatch.cpp 完全一致：

```cpp
#include <bits/stdc++.h>
using namespace std;

// 一本通 1357 车厢调度 (train): 用中转栈模拟 1..n 能否调出目标出站序列
int main() {
    ios::sync_with_stdio(false);
    cin.tie(nullptr);

    int n;
    if (!(cin >> n)) return 0;
    vector<int> siding;
    int nxt = 1;
```

本页要点

nxt = 1 代表 1 号车厢还没有开进中转站；siding 代表中转站 C 的轨道栈。

例 2 | 完整可编译代码（2/2）：调度模拟主循环

接上一页 code/train_dispatch.cpp 结尾：

```
    for (int i = 0; i < n; ++i) {
        int wanted; cin >> wanted;
        while (nxt <= wanted) siding.push_back(nxt++); // 还没到的车厢依次推进中转栈
        if (siding.empty() || siding.back() != wanted) { cout << "NO\n"; return 0;
    }
        siding.pop_back();                              // 栈顶恰好是要出站的车厢
    }
    cout << "YES\n";
    return 0;
}
```

本页要点

虽然循环内也有 while，但 nxt 从 1 单调增至 $n+1$，总入栈仅 n 次，总复杂度严格 $O(n)$！

例 2 | 易错点与 3 组自测边界

- 易错 **1**：把 `nxt` 写进 `for` 循环里每轮重新设成 1，会把已经进站的车厢重复入栈 **WA**。
- 易错 **2**：只看“若 3 出了后面不能出现 $1, 2$ 升序”，手推规则漏掉更深层嵌套导致 **WA**；直接用栈模拟最稳！
- 易错 **3**：没判 `siding.empty()`，若输入非法序列会 **RE**。

3 组必跑自测边界

1. 逆序出站：`5 / 5 4 3 2 1` $\Longrightarrow$ `YES`（全进后再全出）。
2. 经典非法：`3 / 3 1 2` $\Longrightarrow$ `NO`（1 被 2 挡住）。
3. 最小规模：`1 / 1` $\Longrightarrow$ `YES`（测试 $n = 1$ 边界）。

例 3 | 后缀表达式的值（1331）[基础]：先读懂要求

大白话题意

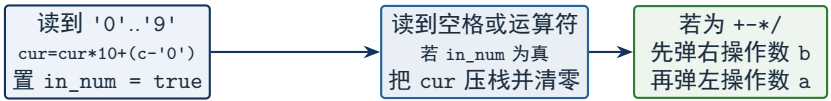
从键盘读入一个以 `@` 结尾的后缀表达式字符串（长度 < 250），只包含数字 `0-9`、加减乘除 `+ - * /` 以及分隔数字的空格（注意运算数可能是多位数，比如 `16`）。题目保证表达式合法、除法能整除、且中间结果与最终结果绝对值均在 2^{64} 范围内。计算整个式子的整数值。

- 输入：一行后缀表达式字符串，以 `@` 结束（例如 `16 9 4 3 +*-@`）。
- 处理：拼出完整多位数压栈；遇运算符弹出栈顶两数做运算后再压回。
- 输出：一个整数（样例输出 `-47`）。

本页要点

看清两大坑： 数字可能是多位数（16）； 范围很大且含负数，栈内类型要开 `long long`！

例 3 | 怎么想到的？词法拼装 + 遇符号弹双数



- 为什么需要布尔标记 in_num？如果直接判断 cur != 0 才压栈，万一表达式里有数字 0（如 0 5 +@），就会把 0 吞掉！
- 有了 in_num，就算数字本身是 0，只要读过 '0' 就会老老实实被压进栈。

本页要点

切记：数字 0 也合法，不能用 cur > 0 判断有没有正在读数，必须用专门的 bool in_num！

例 3 | 样例手推跟踪表：完整演算 16 9 4 3 +*-@

扫描段	cur / in_num	栈内数值（底 → 顶）	操作说明
16	$16 \to 0$ / 假	[16]	'1' 得 1，'6' 得 16，遇空格压入 16
9 4 3	0 / 假	[16，9，4，3]	空格分别触发压入 9，4，3
'+'	0 / 假	[16，9，7]	弹 $b=3, a=4$，算 $a+b=7$ 压回
'*'	0 / 假	[16，63]	弹 $b=7, a=9$，算 $a\times b=63$ 压回
'-'	0 / 假	[-47]	弹 $b=63, a=16$，算 $a-b=-47$ 压回
'@'	停止	[-47]	break 退出，输出栈顶 **-47**

本页要点

注意样例里 +*- 之间没有空格，运算符分支里先清 in_num 再弹栈，两种写法都稳稳兼容！

例 3 | 核心步骤与相对模板改动

按顺序计算	纸上得到	核对问题
读取输入拼接到 text	`while (getline...)`	若题面中间换行也能读全吗？
数字字符拼装成 cur	`cur = cur*10+(c-'0')`	遇数字 0 时 in_num 有置 true 吗？
遇空格或符号先收尾数字	`st.push_back(cur)`	cur 和 in_num 都清零了吗？
运算符弹 *b* 再弹 *a*	算 `a - b` 与 `a / b`	左右操作数有没有搞反？

相对基础栈模板改了哪里？

1. 栈类型从 int 升级为 long long，避免大数相乘溢出。
2. 加入了多位数词法状态机（cur + in_num），逐字符拼出十进制整数再入栈。

例 3 | 完整可编译代码（1/3）：读取全文

以下源码与 code/postfix_eval.cpp 完全一致：

```cpp
#include <bits/stdc++.h>
using namespace std;
// 一本通 1331 后缀表达式的值：遇数字入栈，遇运算符弹两个操作数计算再压回
int main() {
    ios::sync_with_stdio(false);
    cin.tie(nullptr);
    string line, text;
    while (getline(cin, line)) text += line + '\n';
    vector<long long> st;
    long long cur = 0; bool in_num = false;
```

本页要点

用 while (getline(cin, line)) 拼接全文，跨行或多余空白也能正常解析到 @。

例 3 | 完整可编译代码（2/3）：数字拼装与空格收尾

接上一页 code/postfix_eval.cpp：

```
    for (char c : text) {
        if (c == '@') break;
        if (c >= '0' && c <= '9') {
            cur = cur * 10 + (c - '0');
            in_num = true;
        } else if (isspace((unsigned char)c)) {
            if (in_num) { st.push_back(cur); cur = 0; in_num = false; }
        } else {
```

本页要点

看到空格时若 in_num 为真，立刻把拼好的整数 cur 压栈并清零。

例 3 | 完整可编译代码（3/3）：弹栈运算与输出

接上一页 code/postfix_eval.cpp 结尾：

```
            if (in_num) { st.push_back(cur); cur = 0; in_num = false; }
            long long b = st.back(); st.pop_back(); // 先弹出的是右操作数 b!
            long long a = st.back(); st.pop_back(); // 后弹出的是左操作数 a!
            if (c == '+') st.push_back(a + b);
            else if (c == '-') st.push_back(a - b);
            else if (c == '*') st.push_back(a * b);
            else if (c == '/') st.push_back(a / b);
        }
    }
    cout << st.back() << '\n'; return 0;
}
```

本页要点

永远按“先 b 后 a”顺序弹出，保证 a - b 与 a / b 万无一失。

例 3 | 易错点与 3 组自测边界

- 易错 **1**：用 `int` 存栈内元素，连乘超出 2×10^9 时发生有符号溢出 **WA**。
- 易错 **2**：只靠 `if (cur != 0)` 判断是否压栈，导致输入里有 `0` 时漏掉数字 **WA**。
- 易错 **3**：减法或除法把 `a` 和 `b` 顺序写反。

3 组必跑自测边界

1. 原题样例：`16 9 4 3 +*-@` $\Longrightarrow$ `-47`（含多位数与负数减法）。
2. 含数字 `0` 与整除：`100 0 + 25 /@` $\Longrightarrow$ `4`（检验 `in_num` 处理 `0`）。
3. 大数乘法：`100000 100000 *@` $\Longrightarrow$ `10000000000`（检验 `long long` 不溢出）。

例 4 | 右侧首个更大元素 [基础]：先读懂要求

大白话题意

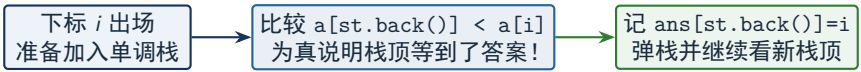
给定一个长度为 n（$1 \le n \le 10^5$）的整数数列 $a_1, a_2, \ldots, a_n$，请你为每一个位置 i 找出它右侧第一个严格大于 a_i 的元素下标 j（即最小的 $j > i$ 且 $a_j > a_i$）；如果右边没有任何数比它大，输出 0。

- 输入：第 1 行 n；第 2 行 n 个整数 $a_1 \ldots a_n$。
- 处理：用 $O(n)$ 单调栈从左向右扫一遍，遇到更高元素时弹出矮栈顶并记录答案。
- 输出：一行 n 个整数（样例 `5 / 2 1 5 3 6` 输出 `3 3 5 5 0`）。

本页要点

这是后续做《柱状图最大矩形》《奶牛排队》《每日温度》等一大批单调栈真题的直接母题！

例 4 | 怎么想到的？让还没找到答案的下标在栈里排好队



- 栈里的每个下标，都代表“还没找到右边第一个比它大的元素”的等待者。
- 因为谁也没找到更高的，所以它们从小到大顺序站立时，身高必定满足从栈底到栈顶单调不增（否则后面那个早就把前面矮的踢出去了）。

本页要点

单调栈就像“失物招领处”：还没找到右边更大值的下标都暂存栈里，谁等来了高个子就立刻领走答案并离队！

例 4 | 样例手推跟踪表：输入 5 / 4 4 2 5 1

为了练一下带相等元素的情况，我们手推 $a=[4,4,2,5,1]$（注意前面两个 4 相等）：

i ($a[i]$)	判断 `a[top] < a[i]`	弹出谁	压入 i 后栈 `st`	当前 `ans[1..5]`
1 (4)	栈空	无	$[1_{(4)}]$	`[0,0,0,0,0]`
2 (4)	$4<4$ 为假！不弹 1	无	$[1_{(4)},2_{(4)}]$	`[0,0,0,0,0]`
3 (2)	$4<2$ 为假	无	$[1_{(4)},2_{(4)},3_{(2)}]$	`[0,0,0,0,0]`
4 (5)	$2<5,\ 4<5,\ 4<5$ 全真	依次弹 $3,2,1$	$[4_{(5)}]$	`[4,4,4,0,0]`
5 (1)	$5<1$ 为假	无	$[4_{(5)},5_{(1)}]$	`[4,4,4,0,0]`

本页要点

注意第 2 步：因为要求严格大于，第二个 4 不能踢掉第一个 4，它们会一起在栈里等到第 4 步的 5，答案同为 4！

例 4 | 核心步骤与相对模板改动

按顺序计算	纸上得到	核对问题
采用 1-based 读入数组	`a(n+1), ans(n+1,0)`	找不到时天然输出 0 而不与下标 0 混淆吗？
遇更高值不断弹栈写答案	`ans[st.back()] = i`	`pop_back()` 写在赋值之后吗？
当前下标压栈等待	`st.push_back(i)`	压进去的是 `i` 而不是 `a[i]` 吗？

为什么这题选用 1-based 下标？

- 因为题面用 0 表示“不存在更大的元素”。如果数组从 0 开始编号，你就分不清“找不到”和“第 0 号元素”！
- 从 1 开始编号并把 `ans` 全体初始化为 0，既直接符合题意，结尾也不用把还在栈里的下标再弹出来赋 0。

例 4 | 完整可编译代码（1/2）：读入与数组准备

以下源码与 code/mono_stack.cpp 完全一致：

```cpp
#include <bits/stdc++.h>
using namespace std;
// 单调栈 O(n) 模板：求右侧第一个严格大于它的元素下标（无则 0）
int main() {
    ios::sync_with_stdio(false);
    cin.tie(nullptr);
    int n;
    if (!(cin >> n)) return 0;
    vector<int> a(n + 1), ans(n + 1, 0), st;
    for (int i = 1; i <= n; ++i) cin >> a[i];
```

本页要点

开启 sync_with_stdio(false) 与 cin.tie(nullptr)，避免 $n = 10^5$ 读入超时。

例 4 | 完整可编译代码（2/2）：单调栈主循环与输出

接上一页 code/mono_stack.cpp 结尾：

```
    for (int i = 1; i <= n; ++i) {
        while (!st.empty() && a[st.back()] < a[i]) { // 新元素比栈顶高，栈顶终于找到了右侧更大
            ans[st.back()] = i;
            st.pop_back();
        }
        st.push_back(i);                                    // 栈内始终保持下标对应值单调不增
    }
    for (int i = 1; i <= n; ++i) cout << ans[i] << (i == n ? '\n' : ' ');
    return 0;
}
```

本页要点

第 8 行格式输出 `(i == n ? '\n' : ' ')`，末尾不带多余空格，是防 PE 的好习惯！

例 4 | 易错点与 3 组自测边界

- 易错 **1**：把 `ans[st.back()] = i;` 写在了 `st.pop_back();` 后面，写入的是下一个栈顶甚至空栈 **WA/RE**！
- 易错 **2**：把 `st.push_back(i)` 写成 `st.push_back(a[i])`，导致下标越界 **RE** 或答案全乱 **WA**。
- 易错 **3**：数据含负数或 0 时没有初始化 `ans` 或错误使用 0-based 导致混淆。

3 组必跑自测边界

1. 混合数组：`5 / 2 1 5 3 6` $\Longrightarrow$ `3 3 5 5 0`。
2. 全相等数组：`4 / 4 4 4 4` $\Longrightarrow$ `0 0 0 0`（验证严格大于）。
3. 严格递减：`4 / 8 6 4 2` $\Longrightarrow$ `0 0 0 0`（验证不弹栈且无越界）。

栈与单调栈高频错误对照表

题目 / 场景	常见错误写法	正确做法与原因
空栈取顶或弹栈	不写 `!st.empty()` 直接 `st.pop()`	遇右括号 `")"` 会立刻段错误 **RE**
括弧匹配剩余左括号	循环没报错就直接输出 OK	对全左括号 `"((("` 会漏检 **WA**，结尾必判 `st.empty()`
车厢调度重置 `nxt`	在每轮 `for` 内重新写 `nxt = 1`	已经进站的车厢不能重复进站，`nxt` 应全局递增
后缀表达式减除颠倒	先弹出的数减后弹出的数 `b - a`	后进先出，先弹的是右操作数 b，必须算 `a - b`
单调栈存值不存下标	栈里压入 `a[i]` 而非 `i`	找不到原下标写回 `ans`，必须存下标并比 `a[st.back()]`

本页要点

涉及栈操作，任何读顶或弹栈前面都必须紧贴着写 `!st.empty()`（数组版则是 `top >= 0`）！

常见错误：看懂评测结果再修改

提交代码看到 OJ 返回结果时，按下面顺序快速定位：

- 编译错误（**CE**）：检查 `<stack>` / `<vector>` 头文件是否包含；注意 `st.pop()` 没有返回值，不能写 `int x = st.pop();`。
- 答案错误（**WA**）：查三处——结尾有没有判 `st.empty()`、减法除法是不是写成了 `b - a`、单调栈严格与不严格符号（`<` 还是 `<=`）是否搞混、中间连乘有没有忘开 `long long`。
- 格式错误（**PE**）：核对 `OK` / `Wrong` 与 `YES` / `NO` 的大小写，以及行末是否多打了空格。
- 运行错误（**RE**）：95% 是对空栈调用了 `top()/back()/pop()`，或者手写数组栈把 `top` 初值设为 0 导致越界、数组开太小。
- 超时（**TLE**）：检查是不是用了 $O(n^2)$ 暴力双重循环，或读入 10^5 数据忘了关同步流。

本页要点

定位口诀：**RE** 优先查空栈与数组边界，**WA** 优先查操作数顺序、结尾判空与 `</<=`！

提交前检查清单（六必查）

1. 每一次调用 `st.top()` / `st.back()` / `st.pop()` 前，是否都在同一条件前判断了 `!st.empty()`？
2. 括号匹配或合法出栈模拟结束后，是否校验过栈最终已经完全清空（`st.empty()`）？
3. 表达式求值中，弹出两个操作数做减法/除法时，是否保证用后弹出的减先弹出的（`a - b`）？
4. 解析带多位数和数字 0 的表达式时，是否用 `bool in_num` 拼装完整整数，并在空格或运算符前及时压栈？
5. 写单调栈时，栈内保存的是不是元素下标，比较时有没有写对 `a[st.back()]`，且先赋值后弹栈？
6. 若中间结果连乘超过 2×10^9，栈内元素类型是否开成了 `long long`？

最后一步

造一组空串/单右括号 `")"`、全左括号 `"((("`、减法顺序 `5 2 -@` 和全相等单调栈 `4 4 4` 跑通再提交！

本章全部练习与完成标准

1. [入门] 括弧匹配检验（一本通 1354，code/bracket_check.cpp）：练圆/方括号就近配对与结尾判空。
2. [入门] 表达式括号匹配（题库 表达式括号匹配 (stack)）：练过滤字母运算符、只校验圆括号并以 @ 结尾。
3. [基础] 车厢调度（一本通 1357，code/train_dispatch.cpp）：练单向进出中转栈模拟与合法性判定。
4. [基础] 后缀表达式的值（一本通 1331，code/postfix_eval.cpp）：练多位数词法解析与操作数左右顺序。
5. [基础] 右侧首个更大元素（code/mono_stack.cpp）：练均摊 $O(n)$ 单调栈存下标与弹栈结算。
6. [提高] 中缀表达式值与奶牛排队（题库 中缀表达式值 (expr)、一本通 1747）：练双栈表达式转换与单调栈区间边界。

完成标准

能独立写出零越界报错的括号检验与后缀求值；能用语言讲清为什么单调栈每位最多进出一次故总计 $O(n)$；独立 AC 前 5 题并尝试提高题。

课后安排与下一章预告

- 复习：合上课件在草稿纸上完整画一遍 $a=[2,1,5,3,6]$ 的单调栈跟踪表，并默写“右侧首个严格更大”核心 9 行循环。
- 练习：独立复写 `code/` 下 4 份程序并用边界样例测试，重点核对 `16 9 4 3` `+*-@` 与 `3 1 2`。
- 下一章预告：第 2 章《队列与单调队列》——把“后进先出”换成公平排队的“先进先出”，学循环队列取模 `(tail+1)%M` 防假溢出，并用双端单调队列在 $O(n)$ 扫完定长滑动窗口极值！

本章一句话

栈用“后进先出”管住就近回溯与嵌套，单调栈只留下真正有潜力的候选下标让复杂度降至 $O(n)$。

谢谢！