栈的实现 (af:/

```
struct stack {
    int s[1000], t = 0;
    void push(int c) {
        s[t++] = c;
    }
    int top() {
        return s[t - 1];
    }
    void pop() {
        t--;
    }
    bool empty() {
        return !t;
    }
} s;

stack<int> s;
```

队列的实现 (af:/

```
struct queue {
    int p = 0, q = 0, que[1000];
    void push(int c) {
        que[q++] = c;
    }
    void pop() {
        p++;
    }
    int front() {
        return que[p];
    }
    bool empty() {
        return p == q;
    }
} q;

queue<int> q;
```

查询其他的函数 (af:/

https://en.cppreference.com/w/cpp/container/stack (https://en.cppreference.com/w/cpp/container/stack)
https://en.cppreference.com/w/cpp/container/queue (https://en.cppreference.com/w/cpp/container/queue)

dfs 的举例 1 —— 给定一张图，判断能否走到终点 (af:/

```
bool dfs(int x, int y) {
    if (x == n && y == m) return 1;
    for (ri i = 0; i < 4; i++) {
        int nx = x + dx[i], y = dy[i];
        if (nx < 1 || nx > n || ny < 1 || ny > m || map[nx][ny] == '#' || vis[nx][ny]) continue;
        vis[nx][ny] = 1;
        if (dfs(nx, ny)) return 1;
    }
    return 0;
}
```

构成一个系统栈的结构：

```
(1,1)                                    (1, 1) 进入系统栈
(1,1) -> (1, 2)                           (1, 2) 进入系统栈
(1,1) -> (1, 2) -> (1, 3)                  (1, 3) 进入系统栈
(1,1) -> (1, 2)                           (1, 3) 退出系统栈
(1,1) -> (1, 2) -> (2, 2)                  (2, 2) 进入系统栈
(1,1) -> (1, 2) -> (2, 2) -> (2, 3)         (2, 3) 进入系统栈
(1,1) -> (1, 2) -> (2, 2)                  (2, 3) 退出系统栈
(1,1) -> (1, 2)                           (2, 2) 退出系统栈
(1,1)                                    (1, 2) 退出系统栈
                                         (1, 1) 退出系统栈，递归过程结束
```

栈溢出：递归太多层导致调用太多系统栈
段错误的三种情况：① 栈溢出 ② 数组访问越界 ③ 试图对 `NULL` 指向的元素操作

dfs 的举例 2 —— 八皇后问题（回溯） (af:/

```
// 3 个标记，分别记录这一列有没有被占用、这一斜线有没有被占用、另外一条斜线有没有被占用
void dfs(int x) {
    if (x == n + 1) {
        if (++tot <= 3) {
            for (int i = 1; i <= n; i++) printf("%d ", ans[i]);
            puts("");
        }
        return;
    }
    for (int i = 1; i <= n; i++) if (!vis1[i] && !vis2[x + i] && !vis3[x - i + 13]) {
        vis1[i] = vis2[x + i] = vis3[x - i + 13] = 1;
        ans[x] = i; dfs(x + 1);
        vis1[i] = vis2[x + i] = vis3[x - i + 13] = 0;
    }
}
```

函数的递归 —— 先写结束条件。

为什么第一题不用回溯，而第二题用回溯? (af:/

第一题如果已经经过了一个节点，我们已经把它的所有后续节点都走过了，不需要重复走。
而第二题如果我们在一个地方放置了一个棋子，后来需要把它撤销，再在这一行做其他的尝试。

简单的说，第一题的状态仅有 (x, y)，是记忆化搜索；
而第二题的棋子摆放的每种局面都是一个状态。

如果我们第一题也回溯（加一句 `vis[nx][ny] = 0`），答案是一样的，但是复杂度会发生翻天覆地的变化 $O(n^2) \Rightarrow O(4^{4n})$。

撤销操作，还原状态 (af:/

如果没有系统栈，撤销操作也应该按相反的顺序，即栈的顺序

① 举例：莫队

```
void add(int pos){
    cur += cnt[s[pos] ^ val];
    ++cnt[s[pos]];
}
void sub(int pos){
    --cnt[s[pos]];
    cur -= cnt[s[pos] ^ val];
}
```

原题见：https://yhx-12243.github.io/OI-transit/records/cf617E.html (https://yhx-12243.github.io/OI-transit/records/cf617E.html)

② 有的题需要构造这个栈，栈的元素包含两个成员 —— 指针和原来的值，例子我没找到。

尾递归优化 (af:/

```
优化前：
5 -> 4 -> 3 -> 2 -> 1    // 5, 4, 3, 2, 1 依次入栈

优化后：
5                      // 5 入栈
                       // 5 出栈
4                      // 4 入栈
                       // 4 出栈
3                      // 3 入栈
                       // 3 出栈
2                      // 2 入栈
                       // 2 出栈
1                      // 1 入栈
                       // 1 出栈
```

```
int dfs1(int x) {
    do_sth();
    return dfs(x - 1);
}
// 会触发

int dfs2(int x) {
    do_sth();
    return dfs(x - 1) + 2;
}
```

```
// 不会触发

void dfs3(int x) {
    do_sth();
    dfs(x - 1);
}
// 会触发
```

最大矩形问题 (af:/.

题面被 x_faraway_x 拉到 vjudge 上面了。

某位同学提供的 $O(n^2)$ 做法（略）

考虑 $O(n)$ 做法：

r_i 第 i 个楼房左边第一个比他矮的楼房
l_i 第 i 个楼房右边第一个比他矮的楼房

$(r_i - l_i + 1) \times h_i$

我们写单调栈，只要明确一点 —— 什么样的点是没有用的。

维护单调栈两部曲：把没用的去掉 + 把自己加上

```
h[n + 1] = 1e9 + 1;
s.push(n + 1);
for (int i = n; i >= 1; i--) {
    while (h[i] >= h[s.top()]) s.pop();
    ans[i] = s.top();
    s.push(i);
}
```

关闭流同步两部曲： (af:/.

```
ios::sync_with_stdio(false); cin.tie(NULL);
```

合法的出栈序列 (af:/.

三点总结：

1. 答案是卡特兰数（出栈序列和括号序列一一对应，括号序列与合法的路径一一对应）
2. 卡特兰数怎么算（见底下公式）
3. 怎么折叠（最后一次接触 $y = x - 1$ 的点，以及这个点以后的部分 沿 $y = x - 1$ 折叠，终点从 (n, n) 变成了 $(n + 1, n - 1)$）。

$$C_n = \binom{2n}{n} - \binom{2n}{n+1}$$

bfs (af:/.

题目见 dfs 举例 1，把判断能否走到换成求最短的步数是多少

```
    q.push((node){1, 1});
    for (ri i = 1; i <= n; i++)
        for (ri j = 1; j <= m; j++) dis[i][j] = -1;
    dis[1][1] = 0;
    while (!q.empty()) {
        node cur = q.front(); q.pop();
        for (int i = 0; i < 4; i++) {
            int nx = cur.x + dx[i], ny = cur.y + dy[i];
            if (nx < 1 || nx > n || ny < 1 || ny > m || dis[nx][ny] != -1 || c[nx][ny] == '#') continue;
            dis[nx][ny] = dis[cur.x][cur.y] + 1;
            q.push((node){nx, ny});
        }
    }
```

01bfs（双端队列） (af:/.

还是同样的题，加一个 `!`，从 `.` 走到 `!` 上不消耗体力：

```
    q.push_front((node){1, 1});
    for (ri i = 1; i <= n; i++)
        for (ri j = 1; j <= m; j++) dis[i][j] = -1;
    dis[1][1] = 0;
```

```
    while (!q.empty()) {
        node cur = q.front(); q.pop_front();
        for (int i = 0; i < 4; i++) {
            int nx = cur.x + dx[i], ny = cur.y + dy[i];
            if (nx < 1 || nx > n || ny < 1 || ny > m || dis[nx][ny] != -1 || c[nx][ny] == '#') continue;
            if (c[nx][ny] == '.') {
                dis[nx][ny] = dis[cur.x][cur.y] + 1;
                q.push_back((node){nx, ny});
            }
            else {
                dis[nx][ny] = dis[cur.x][cur.y];
                q.push_front((node){nx, ny});
            }
        }
    }
```

滑动窗口 (af:/

题目也被 x_faraway_x 拉到 vjudge 上面了

需要 `deque` 维护，期中一端当成单调栈维护，另外一边判断如果随着窗口的移动被
注意答案是另外一边的

```
    for (ri i = k, cur = 1, p = 0, q = 0; i <= n; i++) {
        //i 是每个滑动窗口的右端点，每次把 cur 加进滑动窗口中。
        for (;cur <= i; cur++) {
            while (p < q && a[cur] <= a[que[q - 1]]) q--;
            if (p < q && que[p] <= i - k) p++;
            que[q++] = cur;
        }
        cout << a[que[p]] << " ";
    }
    cout << endl;
```

优先队列 —— 就是一个堆 (af:/

每一个操作的复杂度是 $O(\log n)$ ，我需要进行的操作数和优先队列里面的元素的对数成比例。

之前我们学的栈、队列每一个操作的复杂度 $O(1)$ 。只需要进行常数次操作。

$O(n)$ 大致能跑 $n \leq 10^8$
$O(n \log n)$ 大致能跑 $n \leq 10^6 \sim 2 \times 10^6$ ，稍微差一些，但也很优秀。（树状数组的 $\log$ 比较小，可能跑 10^7）
$O(n^2)$ 大致能跑 $n \leq 2 \times 10^4$。

堆优化 dijkstra (af://

关于 `vector` 的使用

```
v.push_back(1);
cout << v[0] << endl;
v.push_back(2);
cout << v[1] << endl;
```

领接表：一个 vector 数组，也可认为一个二维数组，但第二维的长度可变。

一个值得注意的点：重载小于号那里，我希望距离小的优先，却写的是大于号。

懒删除：打标记，如果发现已被删除就什么都不做。
懒修改：不能“直接修改 `dis[y]`，比较的时候用 `dis[x] > dis[rhs.x]`”，因为修改一个堆中的点会使堆的内部结构基础不再满足，所以我们只好把新的 `dis` 值以另一个成员放到堆的元素里。

```
struct node {
    int x, d;
    bool operator < (const node &rhs) const {
        return d > rhs.d;
    }
};
priority_queue<node> pq;

memset(dis, 0x3f, sizeof(dis));
dis[s] = 0;
pq.push((node){s, 0});
while (!pq.empty()) {
    int x = pq.top().x; pq.pop();
    if (vis[x]) continue;
    vis[x] = 1;
    for (ri i = 0; i < to[x].size(); i++) {
        int y = to[x][i];
```

```
        if (dis[x] + le[x][i] < dis[y]) {
            dis[y] = dis[x] + le[x][i];
            pq.push((node){y, dis[y]});
        }
    }
}
```

并查集 (af:/

```
struct unionset {
    int f[N];
    void init() {
        for (ri i = 1; i <= n; i++) f[i] = i;
    }
    int getf(int x) {
        if (f[x] == x) return x;
        return getf(f[x]);
    }
    bool insame(int u, int v) {
        return getf(u) == getf(v);
    }
    void merge(int u, int v) {
        if (insame(u, v)) return;
        f[getf(u)] = getf(v);
    }
} us;
```