该库用来记录在 acwing 的代码模板

##

快速排序

```
#include <iostream>

using namespace std;

const int N = 1e6 + 10;

int q[N];

void quick_sort(int q[], int l, int r) {
    if (l >= r) return ;

    int x = q[l + r >> 1], i = l - 1, j = r + 1; //如果超时的话建议修改一下中值 x

    while (i < j) {
        do i ++; while (q[i] < x);
        do j --; while (q[j] > x);
        if (i < j) swap(q[i], q[j]);
    }
    quick_sort(q, l, j);
    quick_sort(q, j + 1, r);
}

int main() {
    int n;

    scanf("%d", &n);

    for (int i = 0; i < n; i ++) scanf("%d", &q[i]);

    quick_sort(q, 0, n - 1);

    for (int i = 0; i < n; i ++) printf("%d ", q[i]);

    return 0;
}
```

归并排序

```
#include <iostream>

using namespace std;

const int N = 1e6 + 10;
```

```
int n;

int q[N], temp[N];

void merge_sort(int q[], int l, int r)
{
    //递归的终止情况
    if(l >= r) return;

    //第一步：分成子问题
    int mid = l + r >> 1;

    //第二步：递归处理子问题
    merge_sort(q, l, mid ), merge_sort(q, mid + 1, r);

    //第三步：合并子问题
    int k = 0, i = l, j = mid + 1, tmp[r - l + 1];
    while(i <= mid && j <= r) {
        if(q[i] <= q[j]) tmp[k++] = q[i++];
        else tmp[k++] = q[j++];
    }
    while(i <= mid) tmp[k++] = q[i++];
    while(j <= r) tmp[k++] = q[j++];

    for(k = 0, i = l; i <= r; k++, i++) q[i] = tmp[k];
}

int main() {
    scanf("%d", &n);

    for (int i = 0; i < n; i ++)
        scanf("%d", &q[i]);

    merge_sort(q, 0, n - 1);

    for (int i = 0; i < n; i ++)
        printf("%d ", q[i]);

    return 0;
}
```

第二章数据结构(一)

单链表

```
#include <iostream>

using namespace std;

const int N = 1000010;
```

```
// head 表示头节点 e[i] 表示节点 i 的值 ne[i] 表示节点 i 的 next 指针是多少
// idx 存储当前已经用到哪个节点

int head, e[N], ne[N], idx;

// 初始化
init () {
    head = -1;
    idx = 0;
}

// 将x插到头节点
void add_to_head (int x) {
    e[idx] = x, ne[idx] = head, head = idx, idx ++;
}

// 将 x 插到下标是 k 的点后面
void add (int k, int x) {
    e[idx] = x;
    ne[idx] = ne[k];
    ne[k] = idx;
    indx ++;
}

// 将下标是 k 的点后面的点删掉
remove (int k) {
    ne[k] = ne[ne[k]]; // 删除的时候并没有关联到 idx
}

int main () {
    int m;
    cin >> m;

    init();

    while (m --) {
        int k, x;
        char op;

        cin >> op;

        if (op == 'H') {
            cin >> x;
            add_to_head(x);
        } else if (op == 'D') {
            cin >> k;
            if (!k) head = ne[head]; // 对头节点的一个特判
            remove(k - 1);
        } else {
            cin >> k >> x;
            add(k - 1, x);
        }
    }

    for (int i = head; i != -1; i = ne[i]) cout << e[i] << " ";
```

```
    cout << endl;

    return 0;
}
```

双链表

```
#include <iostream>

using namespace std;

const int N = 100010;

int m;
int e[N], l[N], r[N], idx;

// 初始化
void init () {
    // 0表示左端点 1表示右端点
    r[0] = 1, l[0] = 0;
    idx = 2;
}

// 在下标是 k 的点的右边插入 x (如果是在左边插入 其实也可以直接调用这个 但参数需要换一下)
void add (int k, int x) { //顺序别写反了
    e[idx] = x;
    r[idx] = r[k];
    l[idx] = k;
    l[r[k]] = idx;
    r[k] = idx;
}

// 删除操作
void remove (int k, int x) {
    r[l[k]] = r[k];
    l[r[k]] = l[k];

}
```

模拟栈

```
#include <iostream>

using namespace std;

const int N = 100010;

int stk[N], tt; // tt 表示栈顶元素
```

```
// 插入
stk[++ tt] = x;

// 弹出
tt --;

// 判断栈是否为空
if (tt > 0) not empty
esle empty

// 栈顶
stk[tt];
```

模拟队列

-普通队列

```
#include <iostream>

using namespace std;

const int N = 100010;

int q[N], hh, tt = -1; // hh 表示的是队头 tt 表示的是队尾 (注意这里栈初始的是 -1 而栈初始的是0)

// 插入一个元素
q[ ++ tt] = x;

// 弹出一个元素
hh ++;

// 判断是否为空
if (hh <= tt) not empty
else empty

// 取出队头元素
q[hh]
```

-循环队列

```
// hh 表示队头，tt表示队尾的后一个位置
int q[N], hh = 0, tt = 0;

// 向队尾插入一个数
q[tt ++ ] = x;
if (tt == N) tt = 0;

// 从队头弹出一个数
hh ++ ;
if (hh == N) hh = 0;
```

```
// 队头的值
q[hh];

// 判断队列是否为空
if (hh != tt) {

}
```

单调栈

```
#include <iostream>

using namespace std;

const int N = 100010;

int n;
int stk[N], tt;

int main () {
    cin >> n;

    for (int i = 0; i < n; i ++) {
        int x;
        cin >> x;

        while (tt && stk[tt] >= x) tt --;

        if (tt) cout << stk[tt] << " ";
        else cout << -1 << " ";

        stk[ ++ tt] = x;
    }
}
```

单调队列

```
#include <iostream>

using namespace std;

const int N = 1000010;

int n;
int a[N], q[N];

int main () {
    scanf ("%d%d", &n, &k);
```

```
    for (int i = 0; i < n; i ++) scanf("%d", &a[i]);


    int hh = 0, tt = -1;

    for (int i = 0; i < n; i ++) {
        // 判断队头是否滑出窗口 q[hh] 里存的是数组下标
        if (hh <= tt && i - k + 1 > q[hh]) hh ++;

        while (hh < = tt && a[q[tt]] >= a[i]) tt --;

        q[ ++ tt] = i;

        if (i >= k - 1) print("%d", a[q[hh]]);
    }

    puts(" ");

    /*
    如果是求窗口里的最大值

    int hh = 0, tt = -1;

    for (int i = 0; i < n; i ++) {
        // 判断队头是否滑出窗口 q[hh] 里存的是数组下标
        if (hh <= tt && i - k + 1 > q[hh]) hh ++;

        while (hh < = tt && a[q[tt]] <= a[i]) tt --; // 就把这里的符号改一下

        q[ ++ tt] = i;

        if (i >= k - 1) print("%d", a[q[hh]]);
    }

    puts(" ");
    */

    return 0;
}
```

KMP字符串

```
#include <iostream>

using namespace std;

const int N = 10010, M = 1000010;

int n, m;

char P[N], s[M];
int ne[N]; // next 数组
```

```
int main () {
    cin >> n >> p + 1 >> m >> s + 1; // 下标从 1 开始

    // 求 next 过程
    for (int i = 2, j = 0; i <= n; i ++) {
        while (j && p[i] != p[j + 1]) j = ne[j];

        if (p[i] == p[j + 1]) j ++;

        ne[i] = j;
    }

    // kmp 匹配过程
    for (int i = 1, j = 0; i <= m; i ++) {

        while (j && s[i] != p[j + 1]) j = ne[j];

        if (s[i] == p[j + 1]) j ++;

        if (j == n) { // 匹配成功
            printf("%d", i - n);

            j = ne[j]; //
        }
    }
}
```

第二章 数据结构(二)

Trie数(字典树)

```
#include <iostream>

using namespace std;

const int N = 100010;

int son[N][26], cnt[N], idx; // son 是子节点 cnt 以当前这个字母结尾的单词有多个 idx 当前用到的下标 下标是 0 的点既是根节点 又是空节点
char str[N];

// 插入操作
void insert (char str[]) {
    int p = 0; // 从根节点开始

    for (int i = 0; str[i]; i ++) {
        int u = str[i] - 'a';

        if (!son[p][u]) { // 如果当前节点的子节点没有这个字母
            son[p][u] = ++ idx;
        }
```

```
        p = son[p][u];
    }

    cnt[p] ++;
}

// 查询操作
int query (char str[]) { // 返回的是这个字符串出现多少次
    int p = 0;

    for (int i = 0; str[i]; i ++) {
        int u = str[i] - 'a';

        if (!son[p][u]) return 0;

        p = son[p][u];
    }

    return cnt[p];
}

int main () {
    int n;

    scanf("%d", &n);
    while (n --) {
        char op[2];
        scanf("%s%s", op, str);

        if (op[0] == 'I') insert(str);
        else print("%d\n", query(str));
    }

    return 0;
}
```

并查集

```
#include <iostream>

using namespace std;

const int N = 100010;

int n, m;
int p[N]; // 每个元素的父节点是谁

int find (int x) { // 返回 x 所在集合的编号 x 的祖宗节点 + 路劲压缩
    if (p[x] != x) p[x] = find(p[x]);
    return p[x];
}

int main  () {
```

```
    scanf("%d%d", &n ,&m);

    for (int i = 1; i <= n; i ++) p[i] = i; // 初始的时候每个集合里只有一个元素 因此每个元素都是自己的父节点

    while (m --) {
        char op[2];
        int a, b;

        scanf("%s%d%d". op, &a, &b);

        if (op[0] == 'M') p[find(a)] == find(b); // 合并集合
        else {
            if (find(a) == find(b)) puts("Yes");
            else puts("No");
        }

    }
    return 0;
}
```

堆

```
#include <iostream>
#include <algorithm> // 导入额外的库

using namespace std;

const int N = 100010;

int n , m;

int h[N], size; // size 表示当前 h 有多少元素

void down (int u) { // down 操作
    int t = u;
    if (u * 2 <= size && h[u * 2] < h[t]) t = u * 2;
    if (u * 2 + 1 <= size && h[u * 2 + 1] < h[t]) t = u * 2 + 1;

    if (u != t) { // 如果 u 不等于 t 说明根节点不是最小值
        swap(h[u], h[t]); // 交换一下最小值 继续执行 down 操作
        down(t);
    }
}

void up (int u) { // up 操作
    while (u / 2 && h[u / 2] > h[u]) {
        swap(h[u / 2], h[u]);
        u /= 2;
    }
}

int main () {
```

```
    scanf("%d", &n);
    for (int i = 1; i <= n; i ++) scanf("%d", &h[i]);
    size = n;

    // 构建堆
    for (int i = n / 2; i; i --) down(i);

    while (m --) {
        printf("%d", h[1]); // 输出堆顶元素
        // 维护堆
        h[1] = h[size];
        size --;
        down(1);
    }

    return 0;
}
```

数据结构(三)

hash表

拉链法

```
#include <iostream>
#include <cstring> // memset 所在库

using namespace std;

int const N = 100003; // 取模时应该取质数 而且要离 2 的整数幂尽可能的远(这么取出错的概率最小)

int h[N], e[N], ne[], idx;

// 拉链法

void insert (int x) {
    int k = (x % N + N) % N // x % N 如果 x 是一个负数那么余数也是负数 所以 + N 让结果为正再取模

    e[idx] = x, ne[idx] = h[k], h[k] = idx ++;
}

bool find (int x) {
    int k = (x % N + N) % N;

    for (int i = h[k]; i != -1; i = ne[i]) {
        if (e[i] == x) return true;
    }

    return false;
}
```

```
int main () {
    int n;

    scanf("%d", &n);

    memset(h, -1, sizeof h); // 将数组清空

    while (n --) {
        char op[2];
        int x;
        scanf("%s%d", op, &x);

        if (op == 'I') insert(x);
        else {
            if (find(x)) puts("Yes");
            else puts("No");
        }
    }

    return 0;
}
```

开放寻址法

```
#include <iostream>
#include <cstring> // memset 所在库

using namespace std;

int const N = 200003, null = 0x3f3f3f3f; // 开到两倍 开放定址法的数组长度一般需要开到题目数据的两到三倍

int h[N];

// 开放寻址法

int find (int x) {
    int k = (x % N + N) % N;

    while (h[k] != null && h[k] != x) {
        k ++;

        if (k == N) k = 0;
    }

    return k;
}

int main () {
    int n;

    scanf("%d", &n);
```

```
    memset(h, 0x3f, sizeof h); // 将数组清空 按字节来的 memset 而不是按数

    while (n --) {
        char op[2];
        int x;
        scanf("%s%d", op, &x);

        int k = find(x);
        if (op == 'I') h[k] = x;
        else {
            if (h[k] != null) puts("Yes");
            else puts("No");
        }
    }

    return 0;
}
```

字符串 hash(快速判断两个字符串是不是相等 O(1)的复杂度)

```
#include <iostream>

using namespace std;

typedef unsigned long long ULL; // 用 ULL 来表示 unsigned long long

const int N = 100010, P = 131; // P 常取 131 或者 13331

int n, m;
char str[N];
ULL h[N], p[N]; // h 数组表示某一前缀的 hash 值 p 数组表示 p 进制

ULL get (int l, int r) {
    return h[r] - h[l - 1] * p[r - l + 1];
}


int main () {
    scanf("%d%d%s", &n, &m, str + 1);

    p[0] = 1;

    for (int i = 0; i <= n; i ++) {
        p[i] = p[i + 1] * P;

        h[i] = h[i - 1] * P + str[i];
    }

    while (m --) {
        int l1, r1, l2, r2;

        scanf("%d%d%d%d", &l1, &r1, &l2, &r2);
```

```
        if (get(l1, r1) == get(l2, r2)) puts("Yes");
        else puts("No");
    }

    return 0;
}
```

STL

常用的 STL

- vector 变长数组 倍增的思想

```
    size()  -> 返回元素个数

    empyt()  -> 返回是否为空

    clear()  -> 清空

    front() / back() -> 返回第一个数 / 返回最后一个数

    push_back() / pop_back() -> 在最后插入一个数 / 把最后一个数删掉

    begin() / end() -> vector 的第 0 个数 / vector 的最后一个数的后面一个数
```

```
#include <cstdio>
#include <cstring>
#include <iostream>
#include <algorithm>
#include <vector>

using namespace std;

int main () {
    vector<int> a;

    for (int i = 0; i < 10; i ++) a.push_back(i);

    for (int i = 0; i < a.size(); i ++) cout << a[i] << ' ';
    cout << endl;

    // vector 的迭代器来遍历
    for (vector<int> :: iterator i = a.begin(); i != a.end(); i ++) cout << *i << ' ';
// a.begin() 其实就是 a[0] a.end() 就是 a.size()
    cout << endl;

    for (auto i = a.begin(); i != a.end(); i ++) // 也可以把上一个这样写

    for (auto x : a) cout << x << ' ';
    cout << endl;

    // 支持比较运算
```

```
    vector<int> a(4, 3) b(3, 4);

    if (a < b) { // 可以比较 vector 之间的大小 按字典序来比

    }

    return 0;
}
```

- pair<int, int> 存储一个二元组

```
    first -> 第一个元素

    second -> 第二个元素

    支持比较运算 也是按字典序 以 first 为第一关键字 以 second 为第二关键字
```

```
#include <cstdio>
#include <cstring>
#include <iostream>
#include <algorithm>

using namespace std;

int main () {
    pair<int, string> p;

    // 初始化 pair 的两种方式
    // 第一种方式
    p = make_pair(10, "wjm");
    //第二种方式
    p = {20, "wjm"};

    // 也可以用 pair 存储三个属性
    pair<int, pair<int, int>> p;

    return 0;
}
```

- string 字符串

```
    substr() -> 返回某一个字串

    c_str() -> 返回 string 对应的字符数组的头指针

    size()

    empty()

    clear()
```

```
#include <cstdio>
```

```cpp
#include <cstring>
#include <iostream>
#include <algorithm>

using namespace std;

int main () {
    string a = "wjm";

    // 在字符串后面添加字符
    a += "def";
    a += "c";

    cout << a << endl;

    cout << a.substr(1, 2) << endl; // 第一个参数是字串的起始位置 第二个参数是字串的长度 当长度超过字符串长度时 输出到末尾为止
    cout << a.substr(1) << endl; // 把第二个参数省略掉 就会返回从 1 开始的整个字符串

    printf("%s\n", a.c_str()) // 这样也可以输出整个字符串

    return 0;

}
```

- queue 队列

```
    push() -> 往队尾插入

    front() -> 返回队头元素

    back() -> 返回队尾元素

    pop() -> 把队头弹出

    size()

    empty()
```

```cpp
#include <cstdio>
#include <cstring>
#include <iostream>
#include <algorithm>
#include <queue>

using namespace std;

int main () {
    queue<int> q;

    // 如果想要清空 queue 那么重新构造一个就可以了
    q = queue<int>();
}
```

- priority_queue 优先队列(是一个堆 默认是大根堆)

```
    push() -> 往堆里插入一个元素

    top() -> 返回堆顶

    pop() -> 把堆顶弹出
```

```cpp
#include <cstdio>
#include <cstring>
#include <iostream>
#include <algorithm>
#include <queue>
#include <vector>

int main () {
    priority_queue<int> heap; // 默认是大根堆

    // 如果想是小根堆 那么插入负数 负数就是按从小到大排序
    heap.push(-x);

    // 如果想直接定义小根堆 定义的时候多加两个参数
    priority_queue<int, vector<int>, greater<int>> heap;

    return 0;
}
```

- stack

```
    size()

    empty()

    push() -> 往栈顶添加一个元素

    top() -> 返回栈顶元素

    pop() -> 弹出栈顶元素
```

```
    和队列的操作差不多
```

- deque 双端队列

```
    size()

    empty()

    clear()

    front() -> 返回第一个元素

    back() -> 返回最后一个元素
```

```
push_back() / pop_back() -> 在最后插入一个元素 / 弹出最后一个元素

push_front() / pop_front() -> 在队首插入一个元素 / 弹出队首元素

begin() / end()

[]
```

- set map multiset multimap 基于平衡二叉树(红黑树) 动态维护有序数列

```
size()

empty()

clear()

begin() / end() -> ++ / -- 操作 返回前驱和后继  时间复杂度 O(logn)
```

- ** set / multiset

```
insert() -> 插入一个数

find() -> 查找一个数

count() -> 返回某一个数的个数

erase() 有两种参数
    (1) 输入是一个数 x 删除所有 x O(k + logn) k 是 x 的个数
    (2) 输入是迭代器 删除这个迭代器

lower_bound() / upper_bound() 最核心的两个操作
    lower_bound(x) -> 返回大于等于 x 的最小的数的迭代器
    upper_bound(x) -> 返回大于 x 的最小的数迭代器
```

- ** map / multimap

```
insert() -> 插入的数是一个 pair

erase() -> 输入的参数是一个 pair 或者是迭代器

find()

[] 时间复杂度是 O(logn)

lower_bound() / upper_bound()
```

- unordered_set unordered_map unordered_multiset unordered_multimap 没有顺序 基于 hash 表实现的

```
** 和上面类型 增删改查的时间复杂度是 O(1) 不支持 lower_bound() 和 upper_bound() 不支持迭代器的 ++ / -- 操作 和排序有关的操作都是不支持的**
```

- bitset 压位

```
bitset<10000> s; -> <> 里存的是个数

支持位运算操作
~s -> 取反

& | ^(异或) >> << == !=

[] -> 取出某一位

count() -> 返回有多少个 1

any() / none() -> 判断是否至少有一个 1 / 判断是否全为 0

set() -> 把所有位置改成 1

set(k, v) -> 将第 k 位变成 v

reset() -> 把所有变成 0

flip() -> 把所有位取反 等价于 ~

flip() -> 把第 k 位取反
```