基础数据结构

王晓鹏

广州大学附属中学

2023 年 11 月

目录

相关定义

- **数据 (data)** 是对客观事物的数值符号的表示。例如数值、图像、声音都属于数据的范畴。
- **数据元素 (data element)** 是数据的基本单位。
- **数据对象（data object）** 是性质相同的数据元素的集合，是数据的一个子集。
- **数据结构（data structure）** 是相互之间存在一种或多种特定关系的数据元素的集合。

数据结构的内涵

1. “操作”的对象：数据。
2. 数据与数据间的关系。
3. 针对数据的基本操作。

据此，数据结构可以形式定义为：

- 数据结构是一个二元组 $Data\_Structure = (D, S)$

- **逻辑结构**：数据元素之间的逻辑关系。
- **物理结构**：数据结构在计算机中的存储方式。
- **顺序存储结构**：
  - 借助元素在存储器中的相对位置来表示数据元素之间的逻辑关系。
  - 逻辑上关联的数据元素，物理存储结构中相邻。
- **链式存储结构** ：
  - 借助元素存储地址的指针（pointer）表示数据元素之间的逻辑关系。
  - 逻辑上关联的数据元素，物理存储结构中不一定相邻。

目录

数组

- 数组是顺序存储的数据集合。
- C++ 中的数组，可以存储一个固定大小的相同类型元素的顺序集合。
- 数组是用来存储一系列数据，但它往往被认为是一系列相同类型的变量。

二分查找

- 问题描述：从 n 个有序的数中查找是否有 x 这个数。
- 例如：从下面 9 个有序的数中查找是否有 12 这个数。

1	3	8	12	18	22	24	27	31

算法分析

1. 初始状态下，将整个序列作为搜索区域（假设为 $[l, r]$）;
2. 找到搜索区域内的中间元素（假设所在位置为 mid ），和目标元素进行比对。
   - 如果相等，则搜索成功;
   - 如果中间元素小于目标元素，表明目标元素位于中间元素的右侧，将 $[mid + 1, r]$ 作为新的搜索区域;
   - 如果中间元素大于目标元素，表明目标元素位于中间元素的左侧，将 $[l, mid - 1]$ 作为新的搜索区域;
3. 重复执行第二步，直至找到目标元素。如果搜索区域无法再缩小，且区域内不包含任何元素，表明整个序列中没有目标元素，查找失败。

- 源程序

```cpp
#include <bits/stdc++.h>
using namespace std;
const int N = 105;
int a[N];
int main() {
    int n, x;
    cin >> n >> x;
    for (int i = 1; i <= n; i++) cin >> a[i];
    bool flag = false; // 标记是否存在x
    int l = 1, r = n;
    while (l <= r) {
        int mid = (l + r) / 2;
        if (a[mid] == x) flag = true;
        else if (a[mid] < x) l = mid + 1;
        else r = mid - 1;
    }
    if (flag) cout << "Yes" << "\n";
    else cout << "No" << "\n";
    return 0;
}
```

问题描述：输入 n，输出 n 以内的所有质数。

- 一个自然的想法是对于小于等于 n 的每个数进行一次质数检验，这种暴力的做法时间复杂度 $O(n^2)$，显然不能达到最优复杂度;
- 考虑优化：对于任意一个大于 1 的正整数 n，那么它的 x 倍就是合数（$x > 1$）。利用这个结论，我们可以避免很多次不必要的检测;
- 埃氏筛核心：从 2 开始删去素数本身倍数，向后找到的第一个数字一定是素数;
- 证明：设已找到第 n 个素数，删去此数自身倍数后找到剩下的第一个数字 L，知 L 之前有且仅有 n 个素数，且都无法整除 L，即 L 无法被小于自身的所有素数整除，推出 L 是素数（L 就是第 $n+1$ 个素数）。

- 源代码

```
bool is_prime[N];
int prime[N];
void Eratosthenes(int n) {
    int tot = 0;
    for (int i = 2; i <= n; ++ i) is_prime[i] = 1;
    for (int i = 2; i <= n; ++ i) {
        if (is_prime[i]) {
            prime[++ tot] = i;
            for (int j = i * i; j <= n; j += i) is_prime[j] = 0;
        }
    }
}
```

- 时间复杂度：$O(n\ ln\ (ln\ n))$；证明略。

- 考虑在埃氏筛的基础上进一步优化。
- 观察发现，$30 = 2 \times 15 = 3 \times 10 = 5 \times 6$，30 被筛了 3 次；显然会将一个合数重复多次标记。能不能只筛一次？
- 只要保证每一个数都被且仅被其最小的质因数筛掉，即被筛的数 i 能被当前的质数整除时（最小质数），就 break 退出，这样后面的数 (后面所有含有 i 的数不会被当前循环筛掉) 就不会被重复篮掉。

线性筛图例

- 线性筛图例

	2	3	5	7	11	13	17	...
2	2*2							
3	2*3	3*3						
4	2*4							
5	2*5	3*5	5*5					
6	2*6							
7	2*7	3*7	5*7	7*7				
8	2*8							
9	2*9	3*9						
10	2*10							
11	2*11	3*11	5*11	7*11	11*11			
12	2*12							
13	2*13	3*13	5*13	7*13	11*13	13*13		
14	2*14							
15	2*15	3*15						
16	2*16							
17	2*17	3*17	5*17	7*17	11*17	13*17	17*17	
...								
2~n	all_2	all_3	all_5	all_7	all_11	all_13	all_17	...

- 源代码

```cpp
void Euler(int n) {
    int cnt = 0;
    for (int i = 2; i <= n; ++i) {
        if (!vis[i]) pri[++cnt] = i;
        for (int j = 1; j <= cnt; j ++) {
            if (i * pri[j] > n) break;
            vis[i * pri[j]] = 1;
            if (i % pri[j] == 0) break;
        }
    }
}
```

- 时间复杂度：$O(n)$；证明略。

- 排序；
- 差分数组；
- 前缀和；
- 二维前缀和；
- 请听本次培训其他专题

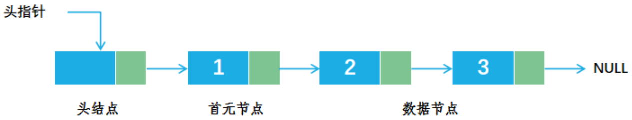
目录

- 链表，别名链式存储结构或单链表，用于存储逻辑关系为" 一对一" 的数据；
- 与顺序表不同，链表不限制数据的物理存储状态，换句话说，使用链表存储的数据元素，其物理存储位置是随机的；
- 动态分配内存，指针实现。

链表的结构

一个完整的链表需要由以下几部分构成：

1. **头指针**：头指针用于指明链表的位置，指向链表第一个节点的位置；
2. **头节点**：通常作为链表的第一个节点 (一般不存数据)，头节点不是必须的，它的作用只是为了方便解决某些实际问题；
3. **首元节点**：链表中称第一个存有数据的节点为首元节点。某些表述的一个称谓，没有实际意义；
4. **其他节点**：链表中其他的节点；



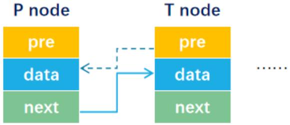
- 指针实现

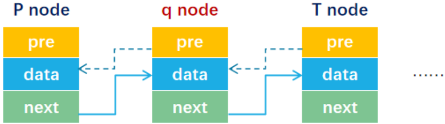
```
struct Node{
    int value;// 数据域
    int *prev,*next;// 指针
}
void initialize(){
    head = new Node();
    tail = new Node();
    head->next = tail;
    tail->prev = head;
}
```

链表增删
———指针实现

- 插入节点

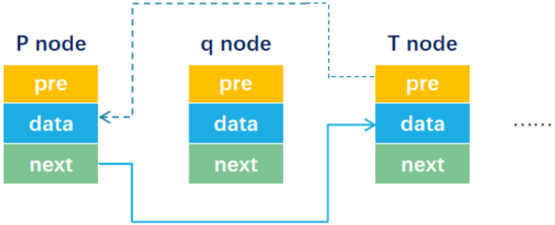
```
void insert (Node *p,int val){
    q = new Node();
    q->value = val;
    p->next->prev = q;
    q->next = p->next;
    p->next = q;
    q->prev = p;
}
```





- 删除节点

```
void remove(Node *p){
    p->prev->next = p->next;
    p->next->prev = p->prev;
    delete p;
}
```



- 静态链表 (数组模拟)

```
struct Node{
    int value;
    Node prev,next;
} node[Maxsize];

int head,tail,tot;
void initialize(){
    tot = 2,head = 1, tail = 2;
    node[head].next = tail;
    node[tail].prev = head;
}

void insert(int p,int val){
    q = ++ tot;
    node[q].val = val;
    node[node[p].next].prev = q;
    node[q].next = node[p].next;
    node[p].next = q;
    node[q].prev = p;
}

void remove(int p){
    node[node[p].prev].next = node[p].next;
    node[node[p].next].prev = node[p].prev;
}
```

假设有一个链表的节点定义如下：

```
struct Node {
    int data;
    Node* next;
};
```

现在有一个指向链表头部的指针：Node* head。如果想要在链表中插入一个新节点，其成员 data 的值为 42，并使新节点成为链表的第一个节点，下面哪个操作是正确的？（ ）

A.
```
Node* newNode = new Node;
newNode->next = head;
head = newNode;
```

B.
```
Node* newNode = new Node;
head->data = 42;
newNode->next = head;
head = newNode;
```

C.
```
Node* newNode = new Node;
newNode->data = 42;
head->next = newNode;
```

D.
```
Node* newNode = new Node;
newNode->data = 42;
newNode->next = head;
```

- 解析:
- D 选项没有更新 head 的值，导致没法插入新的数据，BC 选项对 head 的 data 进行修改;
- 答案: A

链表和数组的区别包括（　）。

A. 数组不能排序，链表可以
B. 链表比数组能存储更多的信息
C. 数组大小固定，链表大小可动态调整
D. 以上均正确

- 解析:
- C 是数组和链表的重要特征;
- 答案：C

以下哪组操作能完成在双向循环链表结点 p 之后插入结点 s 的效果（其中，next 域为结点的直接后继，prev 域为结点的直接前驱）：（　）。

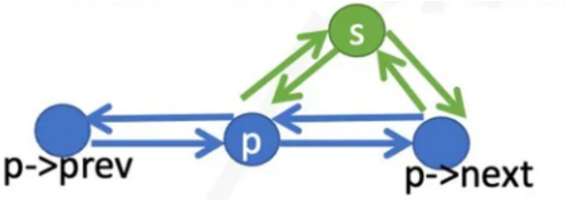
A. $p \to next \to prev = s; s \to prev = p; p \to next = s; s \to next = p \to next;$

B. $p \to next \to prev = s; p \to next = s; S \to prev = p; s \to next = p - \rangle next;$

C. $s \to prev = p; s \to next = p \to next; p \to next = s; p \to next \to preves;$

D. $s \to next = p \to next; p \to next \to prev = s; s \to prev = p; p \to next = s;$

- 解析:
- D 选项能如实反映出下图的修改关系;
- 答案: D



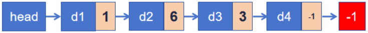
静态链表

❶ 对于线性链表，我们同样也可以使用数组来存储，并实现链式结构，称为“静态链表”;

❷ 静态链表的数组每个元素都有一个数据域和一个指针域，数组元素的下标充当当前元素的地址;

❸ 静态链表便于在没有指针类型的高级程序设计语言中使用链表结构。（竞赛写法中常用）

	val	nxt
下标	初始head=2	
1	d2	6
2	d1	1
3	d4	−1
4		
5		
6	d3	3
7		
8		
9		




```
for (int i = head; i != -1; i = nxt[i]) cout << val[i] << endl;
```

- 源代码

```
#include<bits/stdc++.h>
using namespace std;
const int N = 105;
int cnt, head, tail;
int val[N], nxt[N];
void headInsert(int x) {
    val[++ cnt] = x;
    nxt[cnt] = head;
    head = cnt;
}
int main(){
    int n, x;
    cin >> n;
    head = -1, cnt = 0;
    for (int i = 1; i <= n; i++) {
        cin >> x;
        headInsert(x);
    }
    for (int i = head; i != -1; i = nxt[i]) cout << val[i] << endl;
}
```

约瑟夫问题

【题目描述】

- 有 n 个人围坐在一个圆桌周围，把这 n 个人一次编号为 $1, 2, \ldots, n$。
- 从编号是 1 的人开始报数，数到第 m 个人就出列，然后从出列的下一个人重新开始报数，数到第 m 个人又出列……如此反复，直到所有人都出列为止。
- 给出 n 和 m 的值，输出出列顺序。$n, m \leq 1000$

【样例输入】

```
6 5
```

【样例输出】

```
5 4 6 2 3 1
```

- 使用环形链表模拟约瑟夫问题的过程。
- 一个人出列即删除链表中的一个节点。
- 源代码

```
#include<bits/stdc++.h>
int main(){
    int n,m,i,a[1001],k=1;
    cin >> n >> m;
    for(i = 1; i < n; i ++) a[i] = i + 1;
    a[n]=1;
    while(n){
        for(i = 2; i < m; i ++) k = a[k];
        cout << a[k] << "␣";
        a[k] = a[a[k]];
        k = a[k];
        n--;
    }
}
```

【题目描述】

- 你有一个破损的键盘。键盘上所有的键都能正常工作，但有时 *Home* 键或者 *End* 键会自动按下。
- 注意：按下 *home* 键光标会跳到一行的开头，按下 *end* 键，光标会跳到一行的结尾。
- 给你一段使用该键盘输入的文本，求出正常的文本。
- 每行文本包含不超过 100000 个字母、下划线、字符 '['或者 ']'。其中字符 '['表示 *HOME* 键，']'表示 *END* 键。

【样例输入】

```
This_is_a_[Beiju]_text
[[]][][]Happy_Birthday_to_U
```

【样例输出】

```
BeijuThis_is_a__text
Happy_Birthday_to_U
```

- 使用链表维护输入的文本。
- 记录链表当前节点的指针，当输入一个字符时，创建一个节点保存输入的字符，并将节点插入到链表当前节点的后面，然后将新的节点设为当前节点。
- 当输入的字符是 [时，将当前节点的指针设为链表的头节点。
- 当输入的字符时] 时，将当前节点的指针设为链表的尾节点。

```cpp
#include <bits/stdc++.h>
using namespace std;
char c[100005];
int nxt[100005];
string s;
int main(){
    while (cin >> s){
        memset(nxt, -1, sizeof(nxt));
        int head = 1 ,tail = 1, tot = 1, cur = 1;
        for (int i = 0; i < s.size(); i++){
            if (s[i] == '[') cur = head;
            else if (s[i] == ']') cur = tail;
            else{
                c[++ tot] = s[i];
                nxt[tot] = nxt[cur];
                nxt[cur] = tot;
                if (cur == tail) tail = tot;
                cur = tot;
            }
        }
        cur = nxt[head];
        while (~cur) {
            printf("%c", c[cur]);
            cur = nxt[cur];
        }
        printf("\n");
    }
}
```

【题目描述】

- 依次读入一个 32 位整数序列，每当读入的整数个数为奇数的时候，输出已经读入的整数构成的序列的中位数。
- 序列长度 M $(1 \le M \le 999)$，M 保证为奇数，序列中不存在相同元素

【样例输入】

```
3
1 9
1 2 3 4 5 6 7 8 9
2 9
9 8 7 6 5 4 3 2 1
3 23
23 41 13 22 -3 24 -31 -11 -8 -7 3 5 103 211 -311 -45 -67 -73 -81 -99 -33 24 56
```

【样例输出】

```
1 5
1 2 3 4 5
2 5
9 8 7 6 5
3 12
23 23 22 22 13 3 5 5 3 -3 -7 -3
```

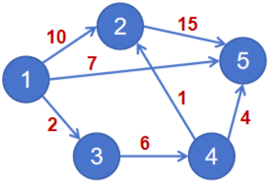
- 当 m 为偶数时，中位数的个数是 $\frac{m}{2}$。
- 当 m 为奇数时，中位数的个数是 $\frac{m+1}{2}$。
- 将整体读入之后快排，建立链表，首先求出最后一个中位数，然后按读入顺序从后向前每次删去两个数，有如下三种情况：
  1. 如果删去的两个数都大于等于中位数，那么将中位数的位置移到没有被删去的比当前中位数小的最大的数。
  2. 如果删去的两个数都小于等于中位数，那么将中位数的位置移到没有被删去的比当前中位数大的最小的数。
  3. 如果两个数中一个比中位数大而另一个比中位数小，那么当前中位数位置不动。

- 链式前向星用静态链表实现，可以快速访问一个顶点的所有邻接点；算法竞赛广泛采用；
- 链式前向星存储由两个结构组成：
  - 边集数组：edge[]，edge[i] 存储第 i 条边包含的所有信息；
  - 头结点数组：head[]，head[i] 存以 i 为起点的第一条边在边集数组的位置 (在 edge[] 中的下标)
- 链式前向星的存储结构

```
struct node{
    int to,next,w;
}edge[Maxe];//边集数组

int head[Maxn];//头结点数组
```

- 输入一个有 n 个节点，m 条边的有向图；输入格式为 u, v, w，分别代表边 (u, v) 以及边的权值 w；

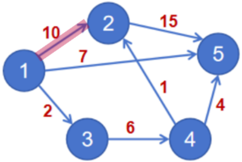



```
5 7
1 2 10
3 4 6
1 3 2
4 2 1
4 5 4
1 5 7
2 5 15
```

- 建边代码

```
void add(int u,int v,int w){
    edge[++cnt].to = v;
    edge[cnt].w = w ;
    edge[cnt].next = head[u];
    head[u] = cnt;
}
```

- 建边 $(1, 2)$




```
5 7
1 2 10
3 4 6
1 3 2
4 2 1
4 5 4
1 5 7
2 5 15
```

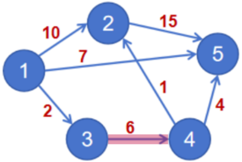
head[ ]

i	
1	1
2	-1
3	-1
4	-1
5	-1

Edge[ ]

i	to	w	next
1	2	10	-1
2			
3			
4			
5			
6			
7			

- 建边 $(3, 4)$




```
5 7
1 2 10
3 4 6
1 3 2
4 2 1
4 5 4
1 5 7
2 5 15
```

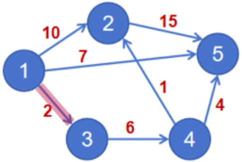
head[ ]

i	
1	1
2	-1
3	2
4	-1
5	-1

Edge[ ]

i	to	w	next
1	2	10	-1
2	4	6	-1
3			
4			
5			
6			
7			

- 建边 $(1, 3)$




```
5 7
1 2 10
3 4 6
1 3 2
4 2 1
4 5 4
1 5 7
2 5 15
```

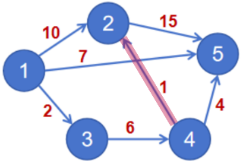
head[ ]

i	
1	3
2	-1
3	2
4	-1
5	-1

Edge[ ]

i	to	w	next
1	2	10	-1
2	4	6	-1
3	3	2	1
4			
5			
6			
7			

- 建边 $(4,2)$




```
5 7
1 2 10
3 4 6
1 3 2
4 2 1
4 5 4
1 5 7
2 5 15
```

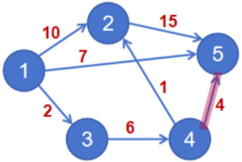
head[ ]

i	
1	3
2	-1
3	2
4	4
5	-1

Edge[ ]

i	to	w	next
1	2	10	-1
2	4	6	-1
3	3	2	1
4	2	1	-1
5			
6			
7			

- 建边 $(4,5)$




```
5 7
1 2 10
3 4 6
1 3 2
4 2 1
4 5 4
1 5 7
2 5 15
```

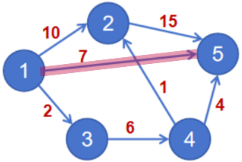
head[ ]

i	
1	3
2	-1
3	2
4	5
5	-1

Edge[ ]

i	to	w	next
1	2	10	-1
2	4	6	-1
3	3	2	1
4	2	1	-1
5	5	4	4
6			
7			

- 建边 $(1,5)$




```
5 7
1 2 10
3 4 6
1 3 2
4 2 1
4 5 4
1 5 7
2 5 15
```

head[ ]

i	
1	6
2	-1
3	2
4	5
5	-1

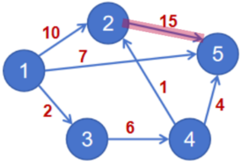
Edge[ ]

i	to	w	next
1	2	10	-1
2	4	6	-1
3	3	2	1
4	2	1	-1
5	5	4	4
6	5	7	3
7			

链式前向星

————存储

- 建边 $(2,5)$




```
5 7
1 2 10
3 4 6
1 3 2
4 2 1
4 5 4
1 5 7
2 5 15
```

head[ ]

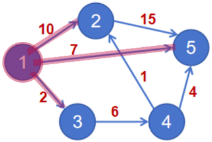
i	
1	6
2	7
3	2
4	5
5	-1

Edge[ ]

i	to	w	next
1	2	10	-1
2	4	6	-1
3	3	2	1
4	2	1	-1
5	5	4	4
6	5	7	3
7	5	15	-1

- 遍历端点 u 为出点的所有边集

```
for (int i = head[u]; i != -1; i = Edge[i].next)
    cout << Edge[i].v << "␣" << Edge[i].w;
```




```
5 7
1 2 10
3 4 6
1 3 2
4 2 1
4 5 4
1 5 7
2 5 15
```

head[ ]

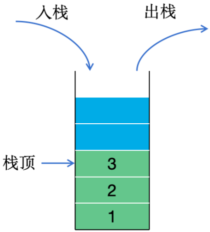
i	
1	6
2	7
3	2
4	5
5	-1

Edge[ ]

i	to	w	next
1	2	10	-1
2	4	6	-1
3	3	2	1
4	2	1	-1
5	5	4	4
6	5	7	3
7	5	15	-1

目录

- 只允许在一端插入和删除的线性表
- 允许插入和删除的一端称为栈顶（top），另一端称为栈底（bottom）
- 特点：后进先出（LIFO）



1. 栈的定义

```
int stack[N], top; //栈数组, 栈顶指针
```

2. 初始化

```
void init() { top = 0; }
```

3. 进栈 PUSH

```
void push(int x) { stack[++top] = x; }
```

4. 出栈 POP

```
void pop() { top--; }
```

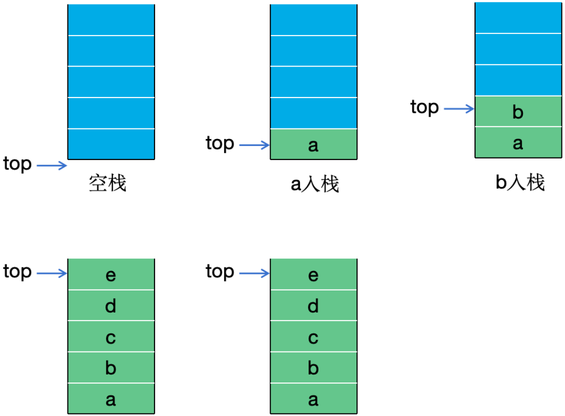
5. 取栈顶元素

```
int getTop() { return stack[top]; }
```

6. 判断栈是否非空

```
bool isEmpty() { return top == 0; }
```

入栈



空栈　a入栈　b入栈

c、d、e入栈　f入栈，溢出

有 6 个元素，按照 6、5、4、3、2、1 的顺序进入栈 S，请问下列哪个出栈序列是非法的（　）。

A. 5 4 3 6 1 2

B. 4 5 3 1 2 6

C. 3 4 6 5 2 1

D. 2 3 4 1 5 6

- 解析:
- 对于 C 的出栈顺序，按照栈先进后出的特点，3，4 先出的话，6，5 还在栈中，6 先进，5 后进，则 6 不可能比 5 先出。
- 答案：C

对假设栈 S 和队列 Q 的初始状态为空。存在 $e1 \sim e6$ 六个互不相同的数据，每个数据按照进栈 S、出栈 S、进队列 Q、出队列 Q 的顺序操作，不同数据间的操作可能会交错。已知栈 S 中依次有数据 e1、e2、e3、e4、e5 和 e6 进栈，队列 Q 依次有数据 e2、e4、e3、e6、e5 和 e1 出队列。则栈 S 的容量至少是 (　) 个数据。

A. 2　　B. 3　　C. 4　　D. 6

- 解析:
- 出队的顺序即入队的顺序。因为每个数据 4 种操作依次进行，因此，入队顺序也是出栈的顺序。那么就看为了达到这个出栈顺序，栈的容量是多少。
- 根据出栈序列，进出栈顺序应该为：e1 进，e2 进，e2 出，e3 进，e4 进，e4 出，e3 出，e5 进，e6 进，e6 出，e5 出，e1 出。容量至少 3。
- 答案：B

后缀表达式“6 2 3 + - 3 8 2 / + * 2 ^ 3 +”对应的中缀表达式是（　）

A. ((6 - (2 + 3)) * (3 + 8 / 2)) ^ 2 + 3
B. 6 - 2 + 3 * 3 + 8 / 2 ^ 2 + 3
C. (6 - (2 + 3)) * ((3 + 8 / 2) ^ 2) + 3
D. 6 - ((2 + 3) * (3 + 8 / 2)) ^ 2 + 3

- 解析:
- 根据后缀表达式到中缀表达式的转换规则，我们遍历后缀表达式并计算:
- 遇到数字时，直接将其入栈；遇到运算符时，从栈中弹出相应数量的操作数，并按照运算符和操作数之间的优先级进行括号添加，然后将结果再次入栈。
- 对于给定的后缀表达式"6 2 3 + - 3 8 2 / + * 2 ^ 3 +"，我们可以通过上述方法得到中缀表达式为:
  ((6-(2+3))*(3+(8/2)))^2+3
- 因此，选项 A. ((6 - (2 + 3)) * (3 + 8 / 2)) ^ 2 + 3 是正确的答案。

【题目描述】

- 所谓后缀表达式是指这样的一个表达式：式中不再引用括号，运算符号放在两个运算对象之后，所有计算按运算符号出现的顺序，严格地由左而右新进行（不用考虑运算符的优先级）。
- 如：3*(5–2)+7对应的后缀表达式为：3．5．2．-*7．+@。@ 为表达式的结束符号。.为操作数的结束符号。
- 运算符只有加减乘除。
- 字符串长度，1000 内。

【样例输入】

```
3.5.2.-*7.+@
```

【样例输出】

```
16
```

后缀表达式
————题目解析

- 维护一个数字栈。
- 表达式中有加减乘除，后缀表达式不用考虑运算的优先级，遵循从左到右计算，所以:
- 当读入的是数字（完整的运算数）时，压入数字栈;
- 当读入的是符号时，弹出栈顶的两个数字，然后执行符号的运算，将将运算结果压入数字栈;
- 最终数字栈顶就是后缀表达式的计算结果。

```
#include <bits/stdc++.h>
using namespace std;
const int N = 1000 + 5;
int stk[N], top;
int main() {
    string s;
    cin >> s;
    int num = 0;
    for (int i = 0; i < s.size() - 1; i++) {
        if (s[i] >= '0' && s[i] <= '9') {
            num = num * 10 + s[i] - '0';
        } else if (s[i] == '.') {
            stk[++top] = num;
            num = 0;
        } else {
            int b = stk[top--];
            int a = stk[top--];
            if (s[i] == '+') stk[++top] = a + b;
            else if (s[i] == '-') stk[++top] = a - b;
            else if (s[i] == '*') stk[++top] = a * b;
            else if (s[i] == '/') stk[++top] = a / b;
        }
    }
    cout << stk[top] << '\n';
}
```

【题目描述】

- 给定一个只包含加法和乘法的算术表达式，请你编程计算表达式的值（结果输出最后 4 位）。

【样例输入】

```
1+1*3+4
1+1234567890*1
1+1000000003*1
```

【样例输出】

```
8
7891
4
```

- 维护两个栈：一个数字栈，一个符号栈。
- 表达式中只有乘法和加法，先读入第一个数并压入数字栈，然后每次读入一个符合和一个数字，因为乘法的优先级比加法的高，所以：
- 当读如的符号是乘号时，将数字栈顶取出，并和读入的数字相乘，结果再压入数字栈中。
- 当读如的符号是加法时，将加号压入符号栈，数字压入数字栈。
- 表达式读入完毕后，如果符号栈不为空，则循环执行弹出加号，然后弹出数字栈最上面的两个数字，相加后压入数字栈，直到符号栈为空。
- 最终数字栈顶就是表达式的计算结果。

单调栈

- 顾名思义，单调栈即满足单调性的栈结构。
- 用途：利用单调栈特性，可以加速某些计算过程。

山峰

【题目描述】

- 有 n 个山峰，一字排开，从西向东依次编号为 $1, 2, 3, \ldots, n$。每个山峰的高度都是不一样的。编号为 i 的山峰高度为 h_i。
- 在第 i 座山峰，记录下回头能看到的山峰数 s_i（如果在第 i 座山峰，存在 $j < k < i$ $h_j < h_k$，那么第 j 座山峰就是不可见的。除了不可见的山峰，其余的山峰都是可见的。）
- 计算 $\sum_i^n s_i$。
- $n \leq 15000, h_i \leq 10^9$。

【样例输入】

```
5
2
1
3
5
9
```

【样例输出】

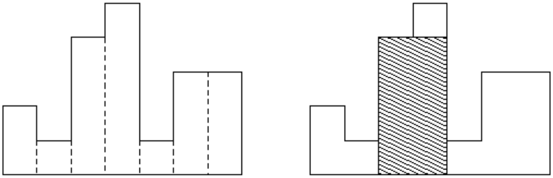
```
5
```

- 暴力扫描的时间复杂度是 $O(n^2)$ 的，会超时。
- 因为高的山峰会挡住低的山峰，所以我们可以维护一个单调递减的栈。
- 当加入一个山峰时，栈中小于要加入的山峰的都弹出去，表示后面的山峰都看不到这些弹出去的山峰了。
- 每个山峰能看到的山峰数就是它入栈前，栈中山峰的数量。

```cpp
#include <bits/stdc++.h>
using namespace std;
int n, num, sum = 0;
stack<int> st;
int main(){
    scanf("%d", &n);
    for(int i = 1; i <= n; i++) {
        scanf("%d", &num);
        if(i == 1) st.push(num);
        else {
            sum += st.size();
            while(!st.empty()) {
                if(num >= st.top()) st.pop();
                else break;
            }
            st.push(num);
        }
    }
    printf("%d", sum);
}
```

【题目描述】

- 如图，有 n 条面积为 $1 \times h_i$ 的小木棒依次排在一起，现在木匠想在其中取一块矩形来作为工件，问工件最大面积可能是多少？
- 图例显示了由高度为 $2, 1, 4, 5, 1, 3, 3$，最大面积如阴影显示



- $1 \le N \le 100000, 0 \le hi \le 1000000000$

【样例输入】

```
10
1 2 3 4 5 6 7 8 9 10
```

【样例输出】

```
30
```

- 从左往右扫描，扫描到当前小木棒 i，若当前 $h_i > h[s.top()]$，那么该小木棒 i 下标入栈；反之，一直将栈顶小木棒 $s.top()$ 出栈；此时 $s.top()+1 \to i$ 的小木棒可以组成连续的工件；则记录 $l[i]$ 为 i 开始往左可以拓展到不低于自身高度的位置；
- 如何确定以当前扫描到的小木棒 i 的高 $h[i]$ 为顶的最大矩阵面积？
- 该最大矩阵的宽度一定是从 i 两边出发，直到遇到比该小木棒高度矮的小木棒所拼成的宽度；即同理，从右往左再扫描一次 $r[i]$。最大面积 $ans = (r[i] - l[i] - 1) * h[i]$

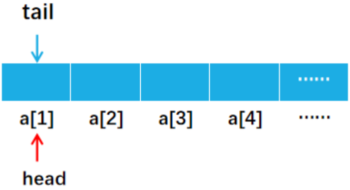
```
#include<bits/stdc++.h>
using namespace std;
long long n, h[100005], l[100005], r[100005], ans = -1;
stack<long long> s;
int main(){
    scanf("%lld", &n);
    for(int i = 1;i <= n;i ++) scanf("%lld", &h[i]);
    for(int i = 1;i <= n;i ++) {
        while(!s.empty() && h[s.top()] >= h[i]) s.pop();
        if(s.empty()) l[i] = 0;
        else l[i] = s.top();
        s.push(i);
    }
    while(!s.empty()) s.pop();
    for(int i = n;i >= 1;i --) {
        while(!s.empty() && h[s.top()] >= h[i]) s.pop();
        if(s.empty()) r[i] = n + 1;
        else r[i] = s.top();
        s.push(i);
    }
    for(int i = 1; i <= n; i ++) ans = max(ans, h[i] * (r[i] - l[i] - 1));
    printf("%lld",ans);
}
```

目录

队列

- 队列：只允许在一端进行插入数据操作，在另一端进行删除数据操作的特殊线性表，队列具有先进先出 FIFO(First In First Out) 的特点；
- 入队列：进行插入操作的一端称为队尾；
- 出队列：进行删除操作的一端称为队头。

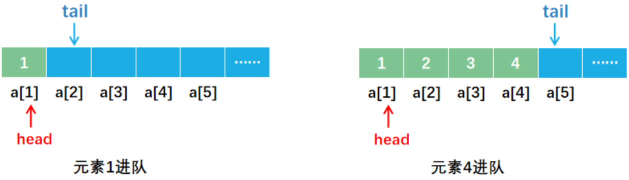
- 数组模拟实现，head 首指针指向队首，tail 尾指针指向队尾



- 初始化

```
head = 1; tail = 1;
```

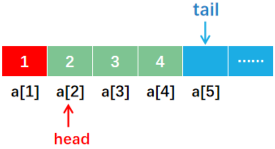
- 元素入队



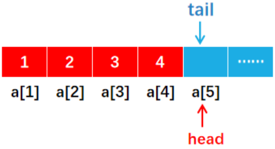
- 入队操作

```
q[tail] = data; tail ++;
```

- 元素出队



元素1出队



元素4出队

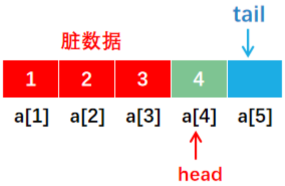
- 出队操作

```
head ++;
```

- 判断队列是否非空

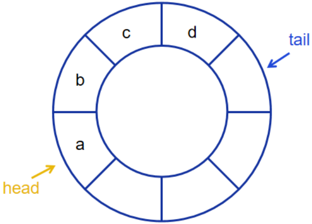
```
bool isEmpty() { return tail == head; }
```

- 假溢出



- 可以发现，当随着不断出队，前面会空出许多空间；而当有新的入队时，那些空间是无法被利用的。如何提高空间利用率呢？

- 循环队列：让 tail 能走到 head 前面的区域，增长一定程度后回到开始，把队列做成一个环，成为循环队列。



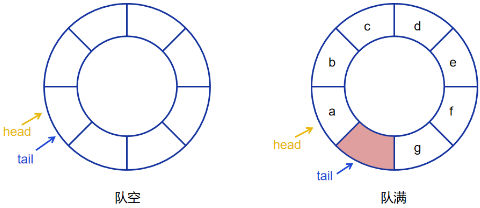
- 循环队列入队

```
q[tail] = data;  tail = (tail + 1) % maxsize;
```

- 循环队列出队

```
head = (head + 1) % maxsize;
```

- 判断队满队空



- 队空

```
head == tail
```

- 队满

```
(tail + 1) % maxsize == head % maxsize;
```

已知队列（13，2，11，34，41，77，5，7，18，26，15），第一个进入队列的元素是 13，则第五个出队列的元素是（　）。

A. 5　　B. 41　　C. 77　　D. 13

- 解析
- 队列先进先出的性质，第 5 个数是 41，则第 5 个出队列元素即是 41。
- 答案：B

设循环队列中数组的下标范围是 $1 \sim n$，其头尾指针分别为 f 和 r，则其元素个数为（ ）。

A. r-f

B. r-f+1

C. (r-f)MOD n+1

D. (r-f+n)MOD n

- 解析
- 循环队列中 f 和 r 不一定满足 $f < r$，如果满足 $f < r$，则元素个数为 $r - f$；但有时也会有 $r < f$，所以需要加 n 然后再对 n 求余数才能统一表示。
- 答案：D

队列快照是指某一时刻对列中的元素组成的有序序列。例如，当元素 1、2、3 入队，元素 1 出队后，此刻的队列快照“2 3”。当元素 2，3 也出队后，队列快照是“”，即为空。

现在有 3 个整数元素一次入队、出队。已知它们的和为 8，则共有 ______ 种可能的不同队列快照（不同队列的相同快照只计一次）。

例如，“5 1”，“4 2 2”，“”都是可能的队列快照；而“7”不是可能的队列快照，因为剩下的 2 个正整数的和不可能为 1。

- 解析
- 第一种情况是“”;
- 第二种情况是一位数，一共有 6 个;
- 第三种情况是两位数 11 到 16 共有 6 个，21 到 25 共 5 个，31 到 34 共 4 个，41 到 43 共 3 个，51 到 52 共 2 个，61 为 1 个，共 $1 + \cdots + 6 = 21$ 个;
- 第三种情况三位数的有 116、125、134、143、152、161，2 开头的有 215、224、233、242、251、...... 同理可得 $6 + 5 + \cdots + 1 = 21$ 个，所以全部加起来共 $1 + 6 + 21 + 21 = 49$ 个。
- 答案：49

【题目描述】

- 蛐蛐国里现在共有 n 只蚯蚓。第 i 只蚯蚓的长度为 a_i（$a_i \geq 0$）
- 每一秒，神刀手会在所有的蚯蚓中，准确地找到最长的那一只（如有多个则任选一个）将其切成两半。
- 神刀手切开蚯蚓的位置由常数 p（$0 < p < 1$）决定，设这只蚯蚓长度为 x，神刀手会将其切成两只长度分别为 $\lfloor px \rfloor$ 和 $x - \lfloor px \rfloor$ 的蚯蚓。特殊地，如果这两个数的其中一个等于 0，则这个长度为 0 的蚯蚓也会被保留。此外，除了刚刚产生的两只新蚯蚓，其余蚯蚓的长度都会增加 q（$q \leq 0$）。
- 蛐蛐国王希望知道这 m 秒内：
  - m 秒内，每一秒被切断的蚯蚓被切断前的长度（有 m 个数）；
  - m 秒后，所有蚯蚓的长度（有 $n + m$ 个数）。
- $1 \leq n \leq 10^5, 0 \leq m \leq 7 \times 10^6, 0 \leq a_i \leq 10^8, 0 \leq q \leq 200$

- 手动模拟一下切割的过程，发现切割过的蚯蚓长度具有单调性，即先切的蚯蚓的左段一定大于等于后切蚯蚓的左段，先切蚯蚓的右段一定大于等于后切蚯蚓的右段。
- 使用三个队列维护蚯蚓，第一个队列是刚开始的 n 只按长度排序后的蚯蚓。第二个队列是存储蚯蚓切割出来的左段，第三个队列是存储蚯蚓切割出来的右段。
- 每次取蚯蚓的时候，取三个队列队头最长的蚯蚓来进行切割，切割完后将蚯蚓左段放入第二个队列，蚯蚓右段放入第三个队列。
- 取蚯蚓的顺序确定了，长度如何维护呢?
- 我们可以考虑，队列中存储的蚯蚓都先不增加长度，当执行 x 秒后，取出来的蚯蚓会增加 qx 的长度，然后分裂成的两只蚯蚓这轮不会增加 q，所以要减去 $q(x+1)$ 的长度后再进入队列。
- 最终每只蚯蚓的长度要加上 qm。

双端队列

【题目描述】

- 有 N 个整数需要排序，只能借助若干个双端队列。
- 依次处理这 N 个数，对于每个数，能做以下两件事：
  1. 新建一个双端队列，并将当前数作为这个队列中的唯一的数；
  2. 将当前数放入已有的队列的头之前或者尾之后。
- 对所有的数处理完成之后，将这些队列排序后就可以得到一个非降的序列。
- 求最少需要的双端队列数。

【样例输入】

```
6
3 6 0 9 6 3
```

【样例输出】

```
2
```

- 从最后的结果考虑，最终序列是单调不下降的。

	原序列	下标序列
原序列	[2 1 3 6 4 5]	[1 2 3 4 5 6]
排序后	[1 2 3 4 5 6]	[2 1 3 5 6 4]

- 可以发现只有下标序列的一段满足先递减再递增（单谷），才能用一个双端队列实现。因为最小的数先处理，然后才能处理左右序号比它大的数。

- 再看题目中的样例：

	原序列	下标序列
原序列	[3 6 0 9 6 3]	[1 2 3 4 5 6]
排序后	[0 3 3 6 6 9]	[3 1 6 2 5 4]
	-	[3 1 6 5 2 4]

- 不过原序列中相同的数对应的下标序列中的数可以交换位置，使得答案更优。如将 [2,5] 交换位置可以只用 2 个队列，若不交换则需要 3 个。考虑贪心使得当前尽量形成单谷。

- 排序时用下标作为第二关键字，对于排序好的序列，记录相同的值的起点 st 和终点 ed，显然两者分别为下标最小和最大值。
- 然后遍历每一个值，记录之间序列中最后一个下标 $last$ 和序列状态 $flag$（0 表示递减，反之递增），作如下分类讨论：
  - 序列递减且 $ed[i] > last$ 没法继续单调递减，只有单调递增，更新 $\text{last} = ed[i]$
  - 序列递增且 $st[i] < last$，没法继续递增，改为单调递减，同时 $\text{ans}++$，更新 $\text{last} = ed[i]$
  - 否则继续递减或递增

```
#include<bits/stdc++.h>
using namespace std;
struct node {
    int s,id;
} s[200005];
int n,cnt,st[200005],ed[200005];
bool cmp(const node &x,const node &y) {
    return x.s == y.s ? x.id < y.id : x.s < y.s;
}
int main(){
    cin >> n;
    for (int i = 1; i <= n; i ++) cin >> s[i].s, s[i].id = i;
    sort(s + 1, s + n + 1, cmp);
    st[++ cnt] = s[1].id;
    for (int i = 2; i <= n; i ++){
        if (s[i].s == s[i-1].s) continue;
        ed[cnt] = s[i-1].id;
        st[++ cnt] = s[i].id;
    }
    ed[cnt] = s[n].id;
    int last = st[1], ans = 1;  bool now=0;
    for (int i = 2; i <= cnt; i ++) {
        if (!now && ed[i] > last) now = 1;
        else if (now && st[i] < last) ans ++, now = 0;
        last = now ? ed[i] : st[i];
    }
    cout << ans;
}
```

- 顾名思义，单调队列的重点分为「单调」和「队列」。
- 「单调」指的是元素的「规律」——递增（或递减)。
- 「队列」指的是元素只能从队头和队尾进行操作。
- 单调队列中的" 队列" 是双端队列（STL 中有类似的数据结构 deque)。

【题目描述】

- 有一个长为 n 的序列 a，以及一个大小为 k 的窗口。现在这个从左边开始向右滑动，每次滑动一个单位，求出每次滑动后窗口中的最大值和最小值。
- 例如序列为 $[1, 3, -1, -3, 5, 3, 6, 7]$，$k = 3$ 时。

Window position	Minimum value	Maximum value
[1 3 -1] -3 5 3 6 7	-1	3
1 [3 -1 -3] 5 3 6 7	-3	3
1 3 [-1 -3 5] 3 6 7	-3	5
1 3 -1 [-3 5 3] 6 7	-3	5
1 3 -1 -3 [5 3 6] 7	3	6
1 3 -1 -3 5 [3 6 7]	3	7

- $1 \leq k \leq n \leq 10^6$，$a_i \in [-2^{31}, 2^{31})$

- 先只考虑如何求最大值。
- 1 3 -1 -3 5 3 6 7
- 观察一下这组数据。当 5 被加入窗口后，如果之后的窗口能覆盖到 -1 或 -3，也一定可以覆盖到 5。因此之后的最大值不可能是 -1 或 -3。
- 从而可以得出一个结论：若窗口中存在 a，当前加入一个数 b，满足 $b > a$，那么 a 已经没有保留的必要。
- 如果后面的数比前面的大，那么前面的数不可能成为最大值。如果后面的数比前面的小，那么两个数都可能成为最大值。

滑动窗口

————题目解析

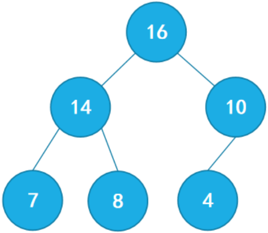
- 进而产生一种想法：维护一个队列存储可能成为最大值的数，同时保证任何时刻队列元素单调减。
- 每次窗口向后移动会加入一个新的数，从队尾不断删去比其小的数，然后把这个数插入队尾。
- 同时判断如果队首元素不在当前窗口内则删去队首。队首元素就是当前窗口的最大值。

窗口位置	单调队列中的元素
1 3 -1 -3 5 3 6 7	3 -1
1 3 -1 -3 5 3 6 7	3 -1 -3
1 3 -1 -3 5 3 6 7	5
1 3 -1 -3 5 3 6 7	5 3
1 3 -1 -3 5 3 6 7	6
1 3 -1 -3 5 3 6 7	7

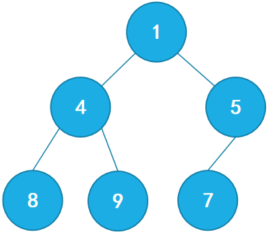
- 由于每个元素各入队、出队一次，时间复杂度为 $O(n)$。

目录

- 堆是一棵树，其每个节点都有一个键值，且每个节点的键值都大于等于/小于等于其父亲的键值。
- 大根堆：所有结点的值不大于父亲结点，最大值为根结点；
- 小根堆：所有结点的值不小于父亲结点最小值为根结点；

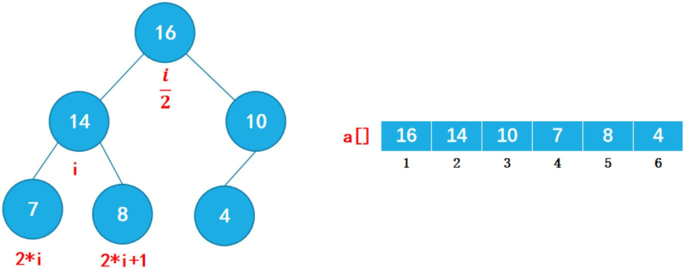


大根堆

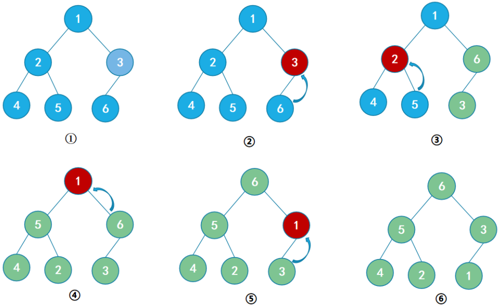


小根堆

- 完全二叉树结构的堆可以使用数组存储。
- 已知结点：i，父亲结点：$\frac{i}{2}$，左孩子：$2 \times i$，右孩子：$2 \times i + 1$



1. 读入初始数组；从第一个**非叶子节点**即 $\frac{n}{2}$ 处开始从上往下调整
2. 如果当前结点大于父亲结点，则交换它们的值，并把父亲结点置为当前结点；不断重复直到当前结点小于等于父结点，结束。



- 堆的调整

```
void heap(int *a, int i, int m){
    int j;
    j = 2*i;
    while(j<=m){
        if(j<m && a[j]<a[j+1])j++;
        if(a[i] > a[j]) break;
        else{
            swap(a[i],a[j]);
            i = j;
            j = 2*i;
        }
    }
}
```

小根堆 {0, 3, 2, 5, 7, 4, 6, 8}，在删除堆顶元素 0 之后，其结果是（　）。

A. {3, 2, 5, 7, 4, 6, 8}
B. {2, 3, 5, 7, 4, 6, 8}
C. {2, 3, 4, 5, 7, 8, 6}
D. {2, 3, 4, 5, 6, 7, 8}

- 解析:
- 将堆顶元素 0 和堆中最后一个元素 8 交换并删除堆顶元素 0 后，向下调整重新建堆;
- 第一次 8 和 2 交换，第二次 4 和 8 交换，所以结果是 {2, 3, 4, 5, 7, 8, 6};
- 答案：C

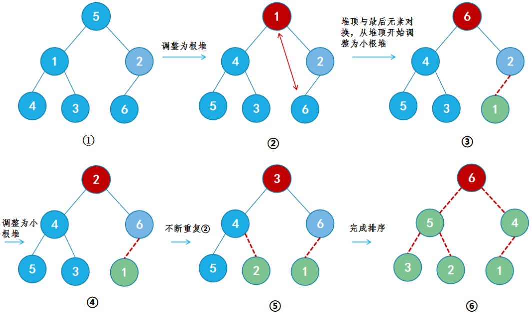
已知小根堆为 $\{8, 15, 10, 21, 34, 16, 12\}$，删除关键字 8 之后需重建堆，在此过程中，关键字之间的比较次数是（　）。

A. 1　　B. 2　　C. 3　　D. 4

- 解析:
- 将堆顶元素 8 和堆中最后一个元素 12 交换并删除堆顶元素 8 后，向下调整重新建堆;
- 首先比较 12 的左右孩子，然后用较小的关键字和 12 进行比较，由于 $12 > 10$，所以交换这两个元素，然后接着向下调整;
- 因为此时以 12 为根的子树没有右孩子，所以 12 只需要和其左孩子进行比较，由于 $12 < 16$，所以不需要再调整了; 整个过程中，关键字之间的比较次数为 3 次。
- 答案: C

【题目描述】

- 给出 n 个数，用堆排序进行排序，从小到大输出。
  - 读入元素，先调整堆为小根堆;
  - 输出最小 (根结点)，从堆顶开始调整堆，反复这个过程直到堆为空;



堆排序

- 堆排序

```cpp
#include<bits/stdc++.h>
using namespace std;
int n,a[1000];
void heap(int *a, int i, int m){
    int j = 2 * i;
    while(j <= m){
        if(j < m && a[j] < a[j+1]) j ++;
        if(a[i] > a[j]) break;
        else{
            swap(a[i],a[j]);
            i = j;
            j = 2 * i;
        }
    }
}
int main(){
    cin >> n;
    for (int i = 1;i <= n;i ++) cin >> a[i];
    for(int i = n / 2; i > 0;i --) heap(a,i,n);
    for(int i = n; i >= 2; i--){
        cout << a[1] << " ";
        a[1] = a[i];
        heap(a, 1, i - 1);
    }
    cout << a[1];
}
```

合并果子

【题目描述】

- 有 n 堆果子要两两合并到一起，每堆果子的重量为 a_i，每次合并消耗的体力等于两堆果子的重量之和。求总共消耗的体力最小值。
- $1 \leq n \leq 10000, 1 \leq a_i \leq 20000$

【样例输入】

```
3
1 2 9
```

【样例输出】

```
15
```

- 贪心，容易想到每次合并时选取最小的两堆; 合并完再放入堆中;
- 先把 n 个数都 *push* 到小根堆里，合并 $n-1$ 次: 每次取出堆顶的两个数，更新答案，再把它们的和 push 到堆里;
- 使用 *STL* 模板库里的 *priority_queue* 实现小根堆。

- 源代码

```
#include<bits/stdc++.h>
using namespace std;
int n, x, ans;
priority_queue<int,vector<int>,greater<int> >q;
int main(){
    cin >> n;
    for(int i = 1; i <= n; i ++) {
        cin >> x;
        q.push(x);
    }
    while(q.size() >= 2){
        int a=q.top();
        q.pop();
        int b=q.top();
        q.pop();
        ans += (a + b);
        q.push(a + b);
    }
    cout << ans << endl;
}
```

【题目描述】

- 丑数是一些质因子只有 $2, 3, 5$ 的数。
- 数列 $1, 2, 3, 4, 5, 6, 8, 9, 10, 12, 15 \ldots$ 写出了从小到大的前 11 个丑数，1 属于丑数。
- 现在请你编写程序，找出第 1500 个丑数是什么。

- 先将 1 push 进堆里，每次取出堆顶，将它分别乘上 $2, 3, 5$ push 进堆里，弹出堆顶。
- 第 1500 次取的堆顶即为答案。
- 入堆的数不能重复，可以使用数组标记某个数是否入堆过。

- 源代码

```
#include<bits/stdc++.h>
using namespace std;
bool vis[1000000005];
#define int long long
priority_queue <int, vector<int>,greater<int> > q;
signed main(){
    int cnt = 0, tmp;
    q.push(1);
    while(cnt < 1500){
        tmp = q.top();
        q.pop();
        if(vis[tmp]) continue;
        vis[tmp] = 1;
        cnt ++;
        q.push(2 * tmp);
        q.push(3 * tmp);
        q.push(5 * tmp);
    }
    cout << tmp;
}
```

中位数

【题目描述】

- 给定一个长度为 N 的非负整数序列 A，对于前奇数项求中位数。
- $1 \leq N\ 100000$，$0 \leq A_i \leq 10^9$

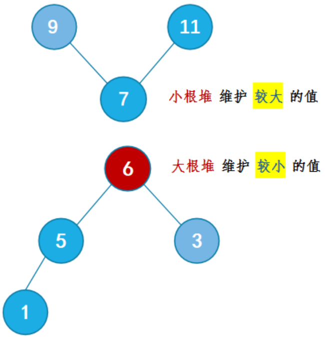
【样例输入 1】

```
7
1 3 5 7 9 11 6
```

【样例输出 1】

```
1
3
5
6
```

- 使用对顶堆维护序列，其中大根堆维护较小的值，小根堆维护较大的值;
- 只要保证，大根堆元素比小根堆元素多一，则中位数为大根堆堆顶;



- 源代码

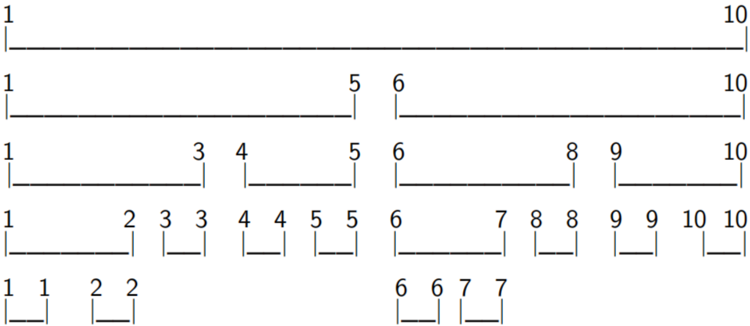
```cpp
#include <bits/stdc++.h>
using namespace std;
priority_queue<int> qmax;
priority_queue<int, vector<int>, greater<int>> qmin;
int main(){
    int n, tmp;
    cin >> n;
    for (int i = 1; i <= n; ++i)    {
        cin >> tmp;
        qmax.push(tmp);
        if (i == 1){
            cout << tmp << "\n";
            continue;
        }
        if (i & 1){
            qmin.push(qmax.top());  qmax.pop();
            while (qmin.top() < qmax.top()) {
                qmin.push(qmax.top());
                qmax.push(qmin.top());
                qmin.pop();
                qmax.pop();
            }
            cout << qmax.top() << "\n";
        }
    }
}
```

目录

- 给你一段长度为 N 的序列 $A[]$, 求:
  1. 单点修改, 单点查询;
  2. 单点修改, 区间查询;
  3. 区间修改, 单点查询;
  4. 区间修改, 区间查询;
- $N \le 100000$
- 修改操作包括但不限于 $+C, *C, \ldots$
- 查询操作包括但不限于 max, min, sum, ...

- 线段树将一个区间划分成一些单元区间，每个单元区间对应线段树中的一个叶结点；根节点对应区间为 $[1, N]$;
- 对于线段树中的每一个非叶子节点 $[a, b]$，它的左儿子表示的区间为 $[a, \frac{a+b}{2}]$，右儿子表示的区间为 $[\frac{a+b}{2}+1, b]$，最后的叶子节点数目为 N;
- 采用堆式存贮：父亲节点编号为 i，则左儿子编号为 $i \times 2$，右儿子编号为 $i \times 2 + 1$。

假设 $N = 10$

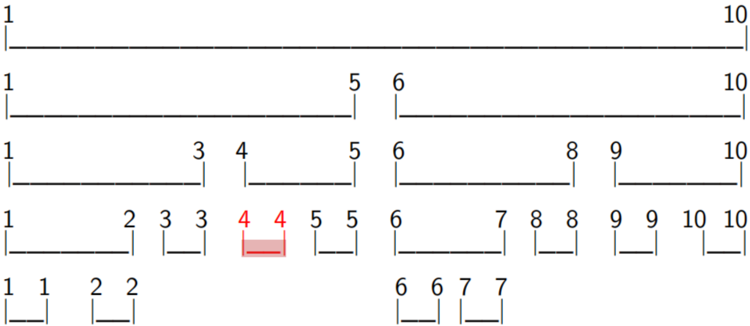


由线段树的存储方式，易知树高为 $\log N$

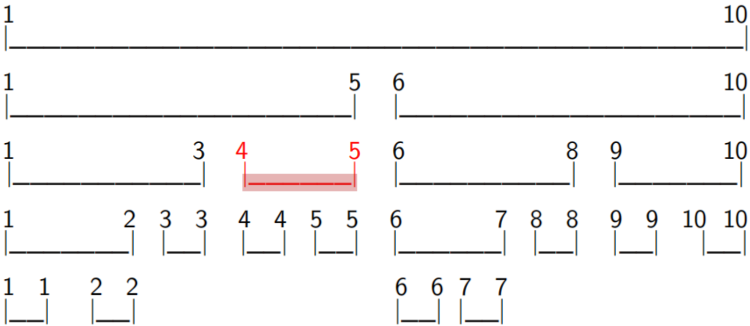
```
void buildTree(int o, int l, int r)
{
    if (l == r) { sum[o] = a[l]; return; }
    int mid = l + r >> 1;
    buildTree(o << 1, l, mid);
    buildTree(o << 1 | 1, mid + 1, r);
    sum[o] = sum[o << 1] + sum[o << 1 | 1];
}
```

修改

修改 $A[4]$

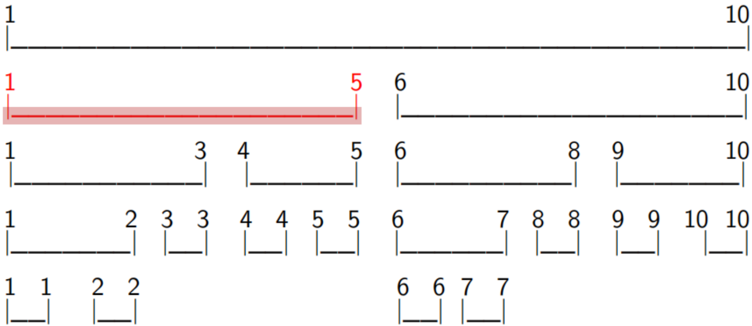


修改 $A[4]$

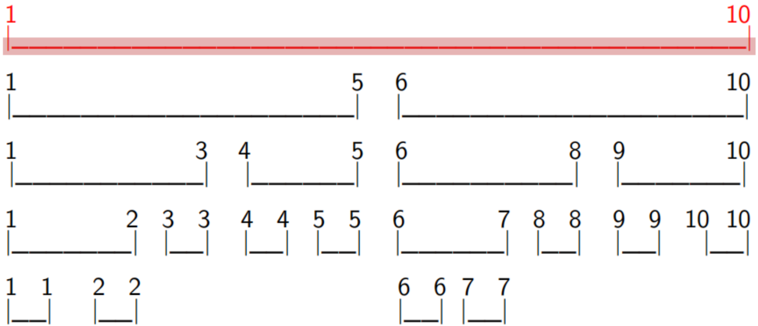


修改

修改 $A[4]$



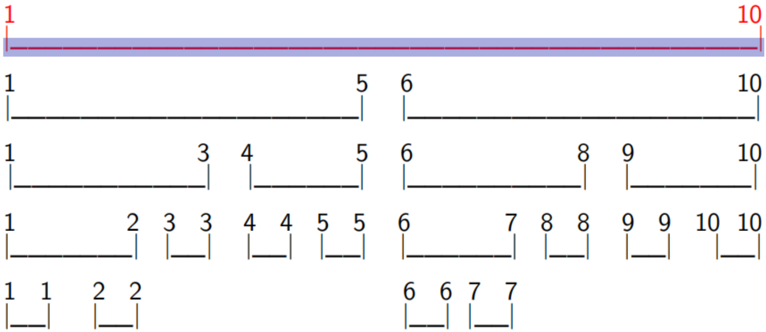
修改 $A[4]$



- 易知单点修改复杂度为 $\log N$。
- 单点查询类似

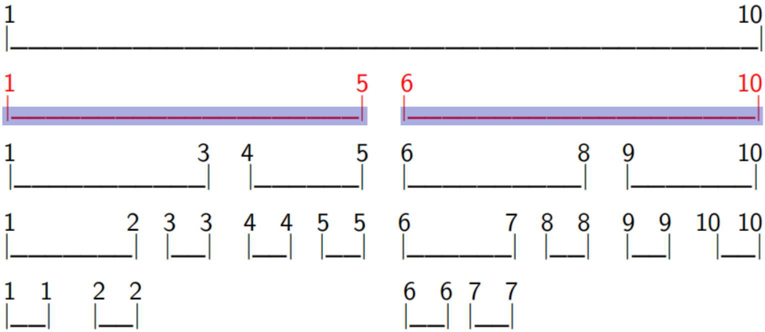
```
void update(int o, int l, int r) // A[x] = y
{
    if (l == r) { sum[o] = y; return; }
    int mid = l + r >> 1;
    if (x <= mid) update(o << 1, l, mid);
    else update(o << 1 | 1, mid + 1, r);
    sum[o] = sum[o << 1] + sum[o << 1 | 1];
}
```

查询 $A[2...9]$

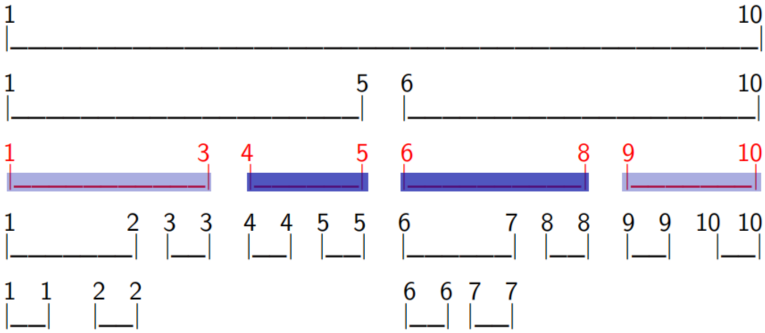


查询

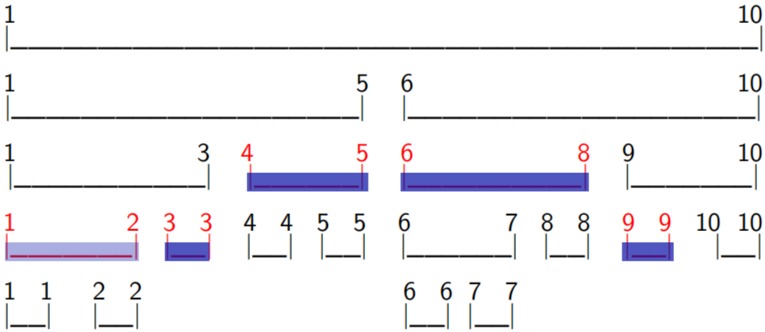
查询 $A[2\ldots9]$



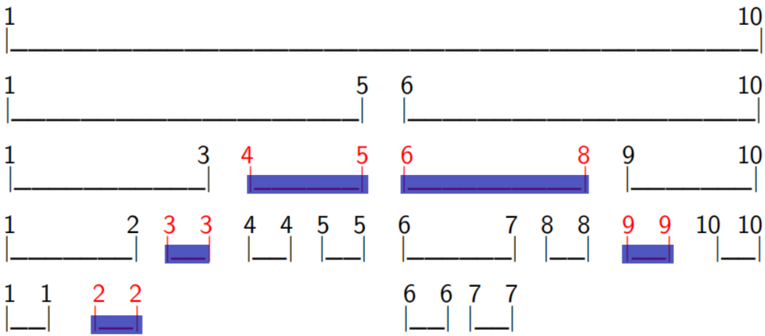
查询 $A[2\ldots9]$



查询 $A[2\ldots9]$



查询 $A[2\ldots 9]$



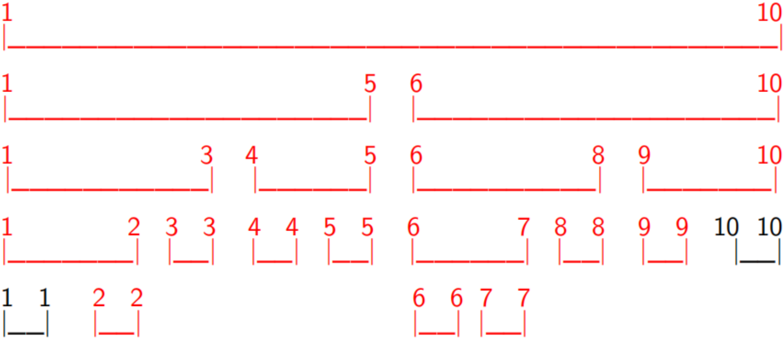
- Q: 为什么不直接查 $[2,9]$?
- A: 因为没有 $[2,9]$ 这段区间啊…
- 易知能够通过访问不超过 $2\times\log N$ 个线段树上的区间来获得任意区间 $[l,r]$ 的答案。

```
void query(int o, int l, int r) //A[x..y]
{
    if (x <= l && r <= y) { ans += sum[o]; return; }
    int mid = l + r >> 1;
    if (x <= mid) query(o << 1, l, mid);
    if (mid < y) query(o << 1 | 1, mid + 1, r);
}
```

- 区间修改类似吗?

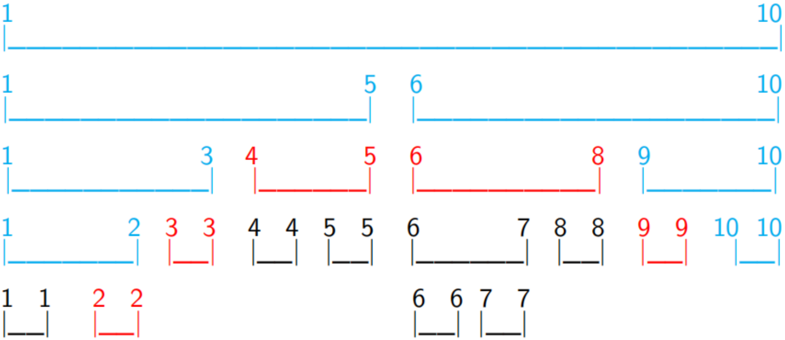
- Lazy-Tag 记录的是每一个线段树节点的变化值。
- 当这部分区间的一致性被破坏时，就将这个变化值传递给子区间。
- 每个节点存一个 Tag 值，表示这个区间进行的变化。
- 每当访问到某一个节点时,Tag 下传。

如果把 $A[2\ldots9]$ 每个数都 $+C$, 那么真正要修改的节点大概这么多



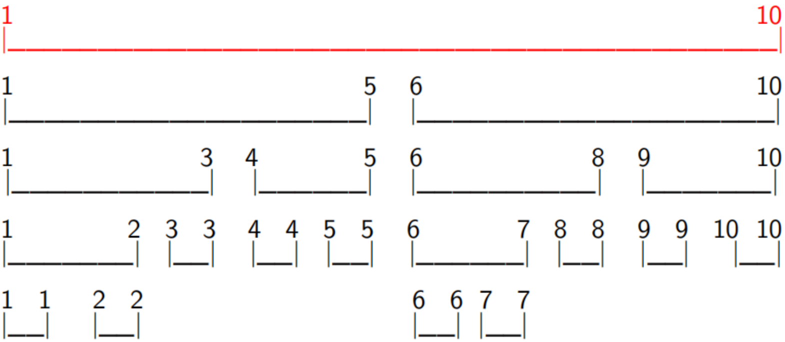
Lazy tag

使用 Lazy Tag 以后，情况是这样的:



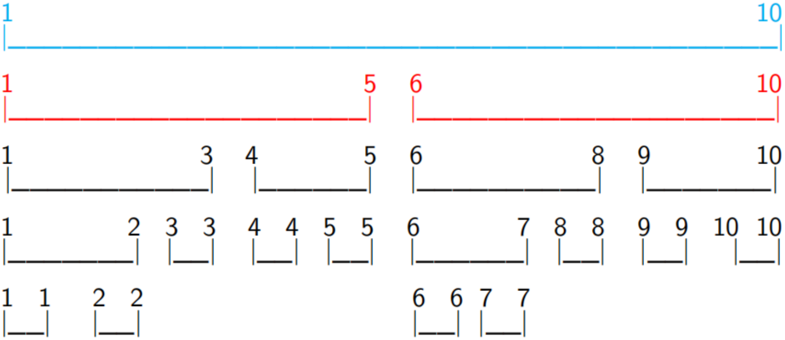
蓝色是要把信息及时维护的节点，红色是本次区间修改操作 Lazy Tag 下传停止的位置

修改 $A[2\ldots 9]$

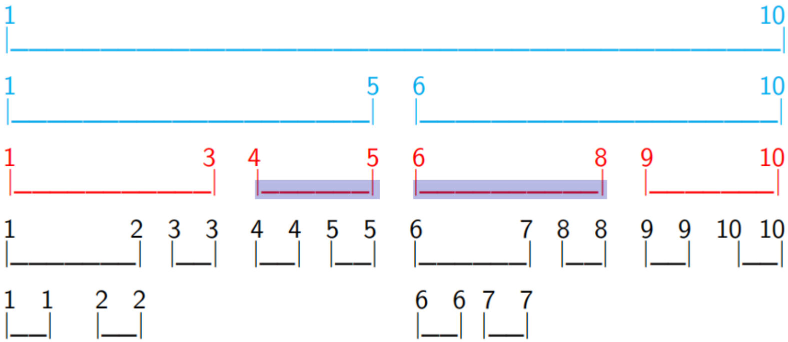


Lazy tag

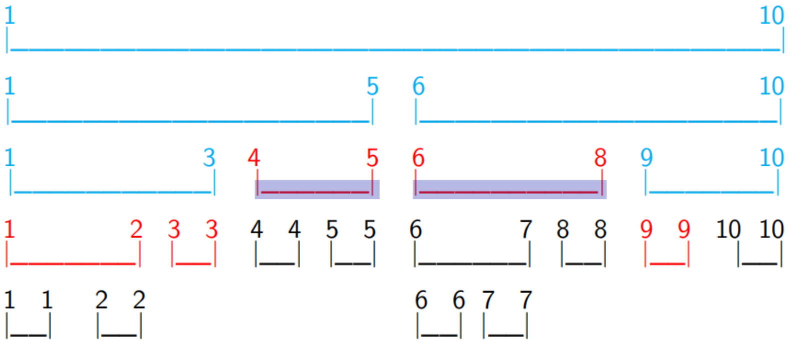
修改 $A[2\ldots9]$



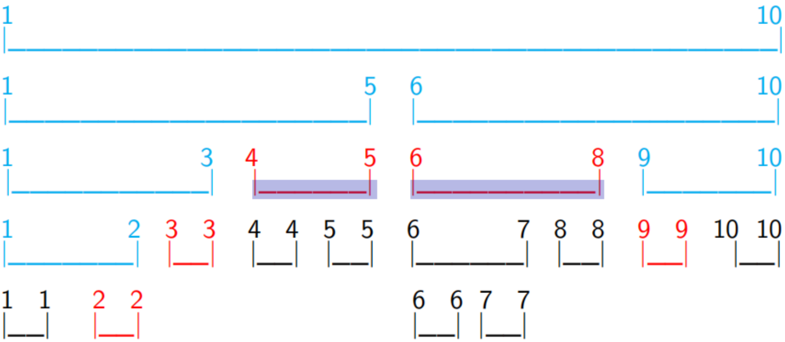
修改 $A[2...9]$



修改 $A[2\dots9]$



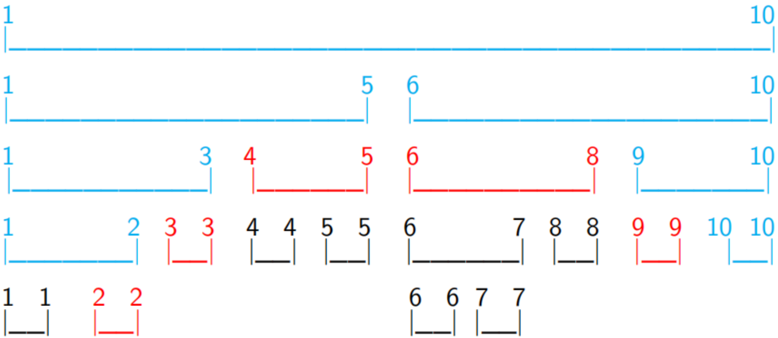
修改 $A[2...9]$



- 由于每一行最多只有两个蓝色区间和两个红色区间，因此线段树区间修改的自带常数为 4.

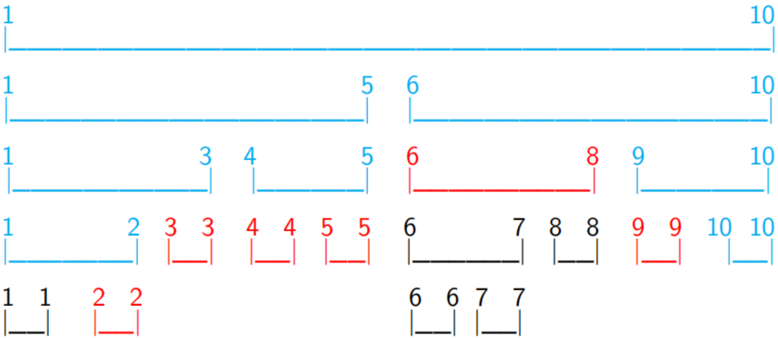
Lazy tag

要查询 $A[5]$

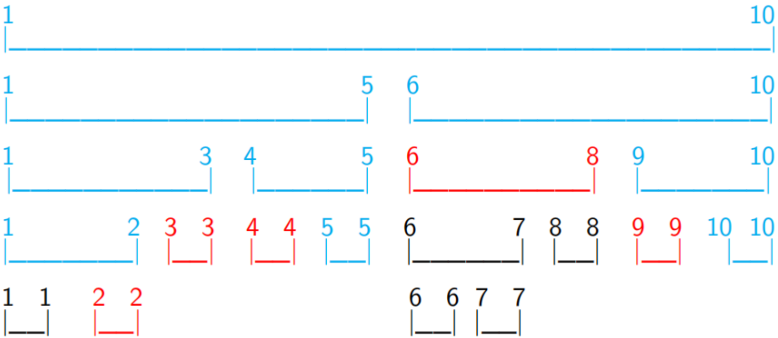


Lazy tag

要查询 $A[5]$



要查询 $A[5]$



- 多个 Lazy Tag 咋办?
- 考虑打标记运算的优先级, 优先级高的先下传;
- 线段树里面每个节点都要记录统计量和 Lazy Tag (修改量);
- 建议写相关的函数都传 3 个参: `(int o, int l, int r)`;
- 线段数一般要开 4 倍空间。

什么时候要用到线段树?

- 统计量可合并
- 修改量可合并
- 通过修改量可直接修改统计量
- 一句话: 满足区间加法 (结合律) 即可使用线段树维护信息

【题目描述】

- 李华需要对编号为 1 到 n 的信用卡记账。
- 对信用卡分为三种操作：
  1. 'Query L R' 表示查询由 L 到 R 的余额之和
  2. 'Add X VAL' 表示编号为 X 的信用卡存入 VAL 个单位的资金
  3. 'Sub X VAL' 表示编号为 X 的信用卡取出 VAL 个单位的资金
- $n \leq 50000$，操作数 ≤ 40000

【样例输入】

```
10
1 2 3 4 5 6 7 8 9 10
Query 1 3
Add 3 6
Query 2 7
Sub 10 2
Add 6 3
Query 3 10
End
```

【样例输出】

```
6
33
59
```

- 线段树维护区间资金量，单点修改，区间查询即可。

买票

【题目描述】

- 现有一条空队伍，编号为 id 的 n 个人分别按照顺序排到当前队伍的第 pos 个人后面去。
- 给出 n 和每个人的 id , pos（不同人的 id 可以重复），请你输出队伍最后的排列情况。
- $n = 200000, 0 \leq id \leq 32767$

【样例输入】

```
4
0 76
1 51
1 33
2 66
4
0 20532
1 19234
1 3890
0 31492
```

【样例输出】

```
76 33 66 51
31492 20532 3890 19234
```

- 由题意思考，对于第 i 个人，如果从前往后放，并不能确定具体位置；
- 反着思考：可以观察最后一个人即第 n 个人的位置是确定的，再观察从后往前可以唯一确定正确的排列；
- 使用权值线段树维护区间还有多少个位置可以站，然后从后往前遍历，查询第 $pos+1$ 个可以站的位置是多少，查询到的位置就是这个人最终要站的位置；
- 这个人站完后记得将这个位置删除。

逆序对

【题目描述】

- 对于一个数列 $A_1, A_2, ..., A_n$ 来说，若存在 A_i, A_j ，满足 $i < tj$ $A_i < A_j$ 则称这一对数为一个逆序对，一个数列的逆序对数即该数列逆序对的总数。
- 对于一个数列 $A_1, A_2, ..., A_n$ ，我们可以通过每次将第一个数移到数列的末尾来形成一个新的数列，这样可以形成的数列共有 n 个

- $A_2, A_3, ..., A_n, A_1$
- $A_3, A_4, ..., A_1, A_2$
- ...
- $A_n, A_1, ..., A_{n-2}, A_{n-1}$
- $A_1, A_2, ..., A_{n-1}, A_n$

【样例输入】

```
10
1 3 6 9 0 8 5 7 4 2
```

【样例输出】

```
16
```

- 现在请你找出以上所有数列的逆序对数中最小的那一个。
- $n \leq 5000, A_i \leq 10^9$

- 首先对数据进行离散化处理，并用线段树求出第一个序列的逆序对的数量；
- 将一个数 x 放到末尾，逆序对的数量变化为减少了比 x 小的个数，增加了比 x 大的个数；
- 从前往后计算逆序对数量变化，取个最小的即可。

告示板

【题目描述】

- 告示板大小为 h 行，w 列，高为 1 ，长为 L 的告示会贴到告示板上。张贴告示时，要求把告示张贴在空间足够的最高那一行的尽可能靠左的位置；如果空间不够，这张告示将不会被张贴。
- 给你告示板和各张告示的大小，请你找到每张告示应张贴在哪一行，不能张贴的输出 -1。
- $n \le 300000, 1 \le h, w \le 10^9$

【样例输入】

```
3 5 5
2
4
3
3
3
```

【样例输出】

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

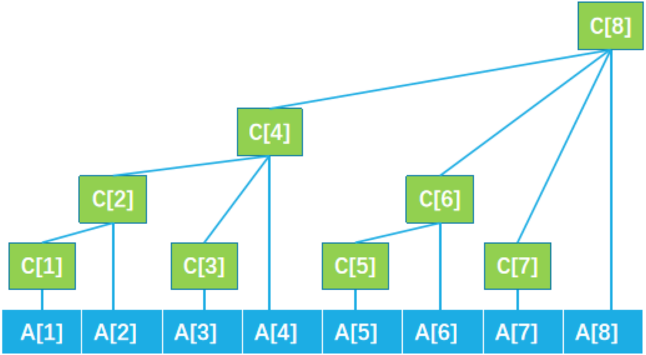
- 线段树维护区间 $[1, \min(h, n)]$ 告示板最大剩余长度;
- 如果最大剩余长度小于要贴的告示长度 L，则不可贴，否则找到最大的位置，这就是这张告示要贴的位置;
- 贴完后，修改告示板长度。

目录

- 区间更新和求区间和。
- 不是线段树也可以吗，学这个干嘛
- 编码复杂度低，常数小
- 有多低
- 很低……

什么是树状数组?

先来看个图



蓝色数组代表原来的数组（下面用 $A[i]$ 代替），绿色结构代表我们的树状数组 (下面用 $C[i]$ 代替)，发现没有，每个位置只有一个方框，令每个位置存的就是子节点的值的和，则有：

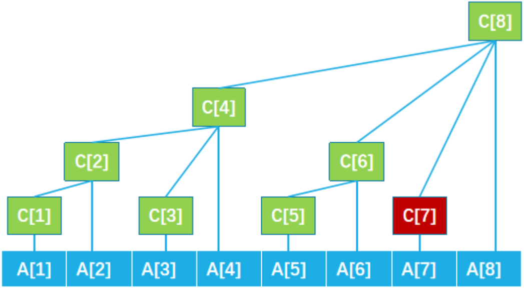
- $C[1] = A[1];$
- $C[2] = A[1] + A[2];$
- $C[3] = A[3];$
- $C[4] = A[1] + A[2] + A[3] + A[4];$
- $C[5] = A[5];$
- $C[6] = A[5] + A[6];$
- $C[7] = A[7];$
- $C[8] = A[1] + A[2] + A[3] + A[4] + A[5] + A[6] + A[7] + A[8];$

- 可以发现，这颗树是有规律的；
- $C[i] = A[i - 2^k + 1] + A[i - 2^k + 2] + \cdots + A[i]$;
- k 为 i 的二进制下末尾连续 0 的个数；
- 这个怎么实现求和呢，比如我们要找前 7 项和，那么应该是 $SUM(7) = C[7] + C[6] + C[4]$;
- 其实树状数组就是二进制分解划分区间；2^k 该怎么求呢？

```
int lowbit (int x) { return x & -x; }
```

lowbit 原理

- 求 $Sum(7)$ 为例：

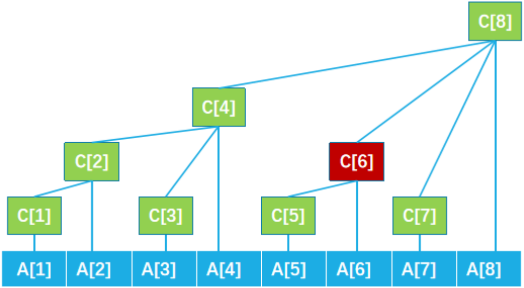


lowbit	7	-7
原码	0111	1111
反码	0111	1000
补码	0111	1001
lowbit	1	

```
7 - lowbit(7) = 7 - 1 = 6
```

lowbit 原理

- 求 $Sum(7)$ 为例：

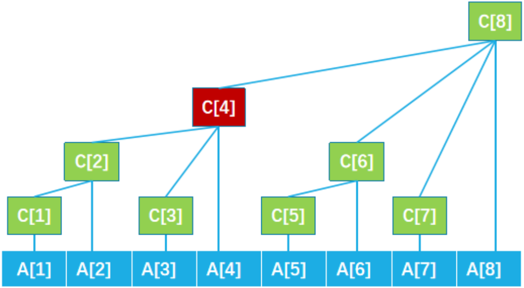


lowbit	6	-6
原码	0110	1110
反码	0110	1001
补码	0110	1010
lowbit	2	

```
6 - lowbit(6) = 6 - 2 = 4
```

lowbit 原理

- 求 $Sum(7)$ 为例：



lowbit	4	-4
原码	0100	1100
反码	0100	1001
补码	0100	1100
lowbit	4	

```
4 - lowbit(4) = 4 - 4 = 0
```

```cpp
#include<bits/stdc++.h>
using namespace std;
const int N = 100005;
int c[N], n, m;
int lowbit(int x) { return x & (-x); }
int getSum(int x){
    int ans = 0;
    while (x) {
        ans += c[x];
        x -= lowbit(x);
    }
    return ans;
}
void update(int x, int val){
    while (x <= n){
        c[x] += val;
        x += lowbit(x);
    }
}
int main(){
    cin >> n;
    for(int i = 1; i <= n; i++)    {
        cin >> m;
        update(i, m);
    }
    int a, b;
    cin >> a >> b;
    cout << getSum(b) - getSum(a-1) << endl;
}
```