



清华大学  
Tsinghua University



# 简单线性 & 树形算法入门 到入坟

---

清华大学学生算法协会 Mys\_C\_K

2024.08.07. 18:30 - 20:30

## ◆ Q & A

- Q: 你看起来好可爱菜你是谁啊?
- A: 我是清华算协的 Mys\_C\_K 确实非常可爱菜。
- Q: 你凭什么给我们讲课啊? 你都拿过什么奖啊?
- A: 2015 普及组二等, 2017 提高组一等里面垫底, WC2018 没牌, APIO2017 铜牌, NOI2018 铜牌, WC2019 铜牌我就是炼铜术士。
- Q: 会不会讲的太简单啊?
- A: .....如果觉得简单可以狠狠嘲讽睡觉.....
- Q: Zzz...
- A: 阿巴阿巴



清華大學  
Tsinghua University



## Part I: 线性数据结构入门

---



## 概要

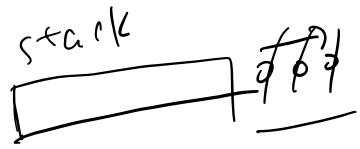
- 尽管所有线性数据结构都能够被其余数据结构在时间开销多  $O(\log n)$  的情况下代替，但总归还是有卡这一点的凉心出题人。
- 而且一些线性数据结构着重强调单调性的重要性，可以在某些题目里面给出一些启发。
- 不过想来想去貌似能讲的东西不多，所以强行加上了某些时间复杂度和线性有关的算法XD



# 目录

- 栈和队列
- 单调栈和单调队列
- 链表
- 差分/前缀和
- 基数排序

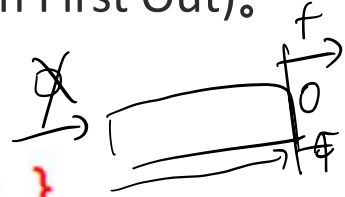
# ◆ 栈&队列



- 栈 (stack) 和队列(queue)是两个有序列表，唯一不同之处在于前者是先进后出 (FILO, First In Last Out)，后者是先进先出 (FIFO, First In First Out)。

手写栈

```
int s[MAX_N], top=0;
void push(int x) { s[++top]=x; }
int top(int x) { return s[top]; }
void pop() { top--; }
bool empty() { return top==0; }
```



STL中的栈

```
std::stack<int> s;
s.push(x);
s.pop();
s.top();
s.empty();
```

- 栈 (stack) 和队列(queue)是两个有序列表，唯一不同之处在于前者是先进后出 (FILO, First In Last Out)，后者是先进先出 (FIFO, First In First Out)。

手写队列

```
int q[MAX_N], fp=1, rp=0;
void push(int x) { q[++rp]=x; }
int front() { return q[fp]; }
void pop() { fp++; }
bool empty() { return fp<=rp; }
```

手写循环队列

```
int q[MAX_N], fp=1, rp=0, cnt=0;
void push(int x) { sz++;rp++;if(rp==MAX_N) rp=1;q[rp]=x; }
void pop() { sz--;rp--;if(rp==0) rp=MAX_N-1; }
int front() { return q[fp]; }
bool empty() { return sz==0; }
```

STL中的队列

```
std::queue<int> q;
q.push(x);
q.pop();
q.front();
q.empty();
```

## ◆ 栈&队列

- 栈 (stack) 和队列(queue)是两个有序列表，唯一不同之处在于前者是先进后出 (FILO, First In Last Out)，后者是先进先出 (FIFO, First In First Out)。

STL中的双端队列

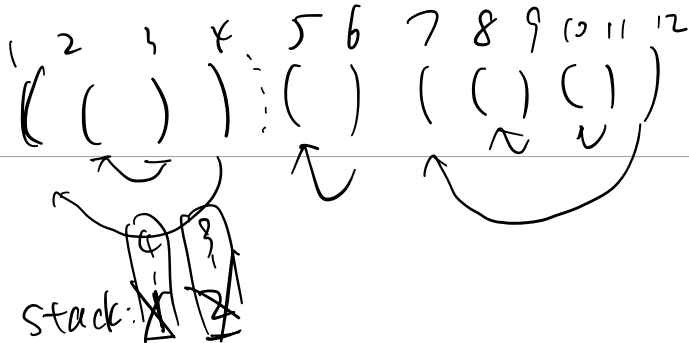
```
std::deque<int> dq;  
dq.push_back(x);  
dq.push_front(x);  
dq.pop_back();  
dq.pop_front();  
dq.back();  
dq.front();  
dq.empty();
```



## ◆ Eg. 括号序列

- 关于合法括号序列的定义：
  - 空串是合法的括号序列。
  - 若  $s$  是合法的括号序列，则  $(s)$  是合法的括号序列。
  - 若  $s$  和  $t$  分别是合法的括号序列，则  $st$  也是合法的括号序列。
- 比如， $()(())$  是合法的，但是  $()())$  就是不合法的。
- 现在给出一个长度不超过  $10^6$  的括号序列，判断其是否合法；若是，求出每个左括号对应的右括号。

## ◆ Eg. 括号序列



- 遇到左括号：
  - 把这个位置压入栈中。
- 遇到右括号：
  - 若栈是空的，则该序列不合法。
  - 否则，这个位置和栈顶的左括号匹配，弹出栈顶。

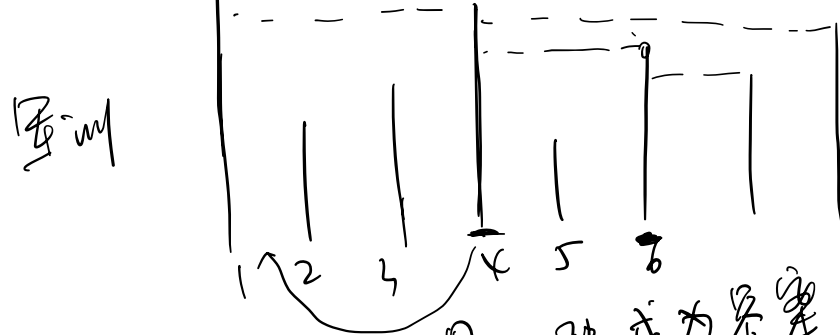
$O(n)$

# 单调栈

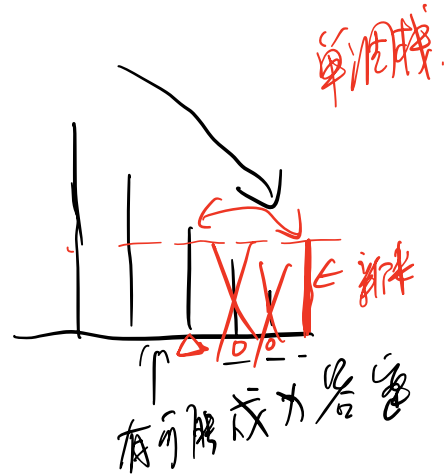
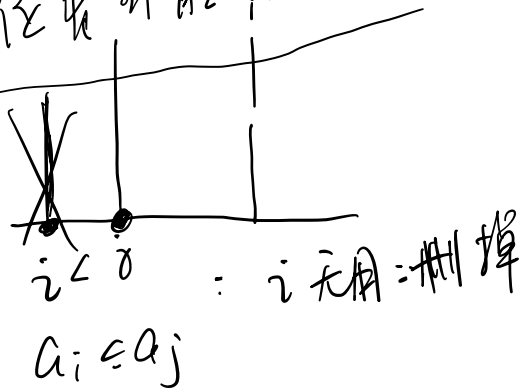
$a_1 \sim a_n$

左

- 考虑这样一个问题：给你一系列数字，对每个数字求出其右边第一个值大于等于它的数字的位置。要求做到  $O(n)$ 。



一：哪些位置可能成为答案



单调栈

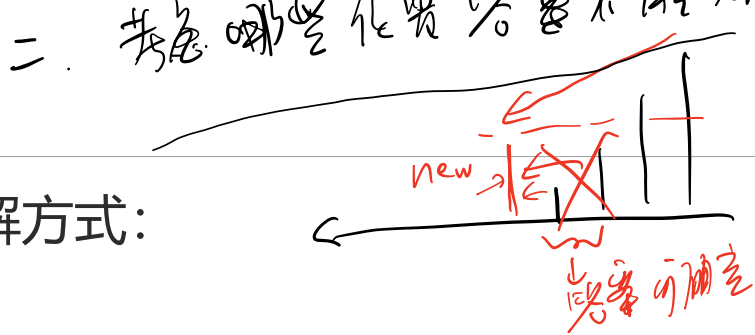
复杂度？

总  $O(n)$

均摊  $O(1)$

如何保证？  
如何保证？  
如何保证？

## ◆ 单调栈



- 有两种理解方式:
- 方案一:
  - 考虑从左到右扫整个序列, 并用栈维护当前还有哪些元素的答案**未被确定**。
  - 可以发现栈中元素自顶到底依次变小 (否则有些位置的答案就能被确定了)。
  - 每次加入一个元素, 则从栈顶开始连续的一段的答案会被确定, 将这些元素从栈顶删除, 并加入新的元素。
- 方案二:
  - 考虑哪些位置可能会成为后续元素的答案。
  - 其余和方案一基本相同。
- 复杂度: 每个元素只会入栈出栈一次, 所以复杂度是线性的。

# Eg1.

- 有一列  $n$  个数字  $a[1] \dots a[n]$ , 对所有  $1 \leq L \leq R \leq n$  求  $\max(a[L], a[L+1], \dots, a[R])$  并求和,  $n \leq 1e6$ .

$$\sum_{L=1}^n \sum_{R=L}^n \max(a_L, \dots, a_R)$$

$L$  与  $a_p$  取得,  $p \in [L, R]$

$$\approx \sum_{p=1}^n a_p \sum_{[L, R]} [a_L \sim a_R \text{ has max to } p \text{ 取得}]$$

单增排一下

$\frac{1}{2} L_p < L \leq p, p \leq R < R_p$

$a_L \sim a_R \text{ has max.}$

$= (p - L_p)(R_p - p)$

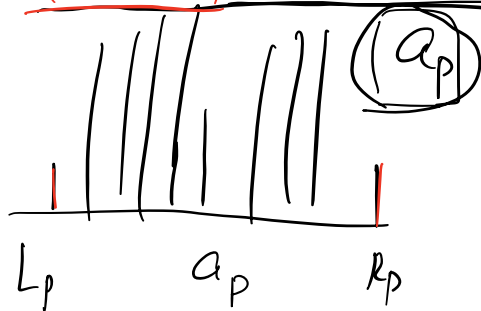
$\sum_{p=1}^n a_p \cdot (p - L_p)(R_p - p)$

## ◆ Eg1.

- 这种题的一个常见套路就是考虑一个数字会在哪些区间中被算到
- 不妨假设数字互不相同（否则，认为编号小的更小）。
- 考虑数字  $a[p]$  会成为哪些区间的  $\max$ ，用上文方法求出左面和右面第一个比它大的位置  $l[p]$  和  $r[p]$ （认为  $a[0]$  和  $a[n+1]$  是正无穷即可避免一些特判），那么当  $l[p] < L \leq p \leq R < r[p]$  的时候， $\max(a[L], \dots, a[R]) = a[p]$ 。因此答案就是  $a[p] * (p - l[p]) * (r[p] - p)$  的和。

## ◆ Eg2.

- 考虑有一列  $n$  个数字, 求  $1 \leq L \leq R \leq n$  使得  $(R-L+1) * \min(a[L], a[L+1], \dots, a[R])$  最大。  
 $n \leq 1e6$ 。



$$\left\{ \begin{array}{l} L \in (L_p, R_p) \\ R \in [L_p, R_p) \\ a_L \sim a_R \text{ 以 } a_p \text{ 为最小值} \end{array} \right.$$

取  $L = L_p + 1, R = R_p - 1$



## Eg2.

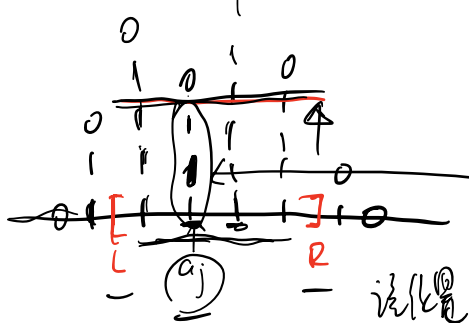
- 和上一个题一样，无甚区别。



# Eg2+.

- 有一个矩阵，每个位置是0或者1。
- 求最大的全1子矩阵。  $n * m \leq 1000000$ 。

4个自由度：上下左右边界  $O(n^2 m^2)$   
 $\Rightarrow$  先枚举下边界  $O(n)$ 。令  $a_j$  表示  $a_{ij}$  向上的1的长度最长是？  
 (枚举  $L \sim R$  列作为左右边界)  
 上边界 - 下边界最大为



$$\max_{L, R} (R - L + 1) \min(a_L, \dots, a_R)$$

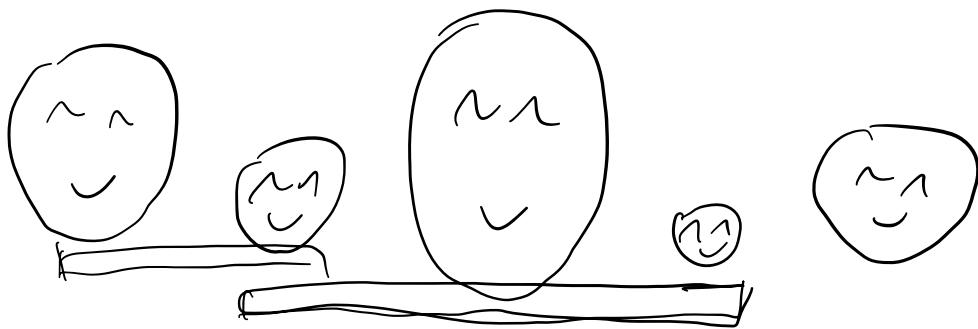
$$a_{ij} = \begin{cases} a_{i-1, j} + 1, & \text{if } a_{ij} = 1 \\ 0, & \text{else} \end{cases}$$

$O(nm)$

$O(m)$   
 总  $O(nm)$

## ◆ Eg2+.

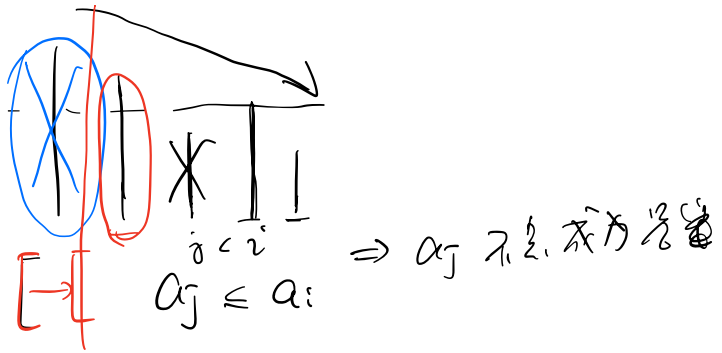
- 对每个位置维护其向上极长的 1 的段的长度，然后每行转化为刚刚的 Eg2. 即可。



## ◆ 单调队列

- 单调队列通常用于解决一个叫滑动窗口的问题。
- 这个问题是这样的：有一列数  $\{a_n\}$  和  $m$  个区间  $[L(i), R(i)]$ ，满足  $L(i) \leq L(i+1)$ ， $R(i) \leq R(i+1)$ 。对每个区间求区间最大值。
- 由于这看上去像是有个大小不固定的窗口在移动，然后你透过窗口观察能看到的数字的最值，所以称为滑动窗口问题。
- 这个问题至少能用随便什么数据结构在  $O(n \lg n)$  时间内完成，但是使用单调队列可以做到  $O(n)$ 。

实现：考虑哪些位置可能成为答案



—



## 单调队列

- 我们维护从左到右哪些数字有可能成为答案。
- 那么每次右边新增一个数字 $a[x]$ ，若其比目前最右边的可能成为答案的数字 $a[y]$ 还大，那么意味着之后无论怎么询问， $a[y]$ 都不可能是答案，删掉即可。然后接着看，直到 $a[x]$ 比 $a[y]$ 小为止。
- 发现这样维护出来的东西，从左到右有可能成为答案的数字是单调变小的。并且目前的区间的最大值就是最左面有可能成为答案的。
- 删除一个数字就看删掉的是不是目前最左面的那个数字即可。

## ◆◆ More then 单调栈/队列

- 实际上，上述“观察哪些东西还未确定答案/可能成为答案”是一种很经典的思想，可以应用到很多题的分析里。具体到一些涉及大小关系的题里，可能可以帮助提炼出单调性，从而使问题据有更好的性质。



## Eg3.

- 给一系列数  $a[1], \dots, a[n]$ , 求最大的区间  $[L, R]$  使得该区间的数字之和  $S[L, R]$  不小于 0。
- $n \leq 1e6$ 。

### ◆ Eg3.

- 令  $S[r]=S[r-1]+a[r]$ 。
- 则问题转为求最大的  $R-L$ ，使得  $S[R] \geq S[L]$ 。
- ~~当然，总是可以直接上个权值线段树之类的。~~
- 考虑枚举  $R$ ，若  $L_1 < L_2 < R$  且  $S[L_1] < S[L_2]$ ，则  $R$  一定不会和  $L_2$  匹配成为答案
- 因此只有一列单调递减的  $S[L]$  可能和  $S[R]$  匹配。
- 此时已经可以在其中二分做到  $O(n \lg n)$ 。
- 不过可以接着分析单调性，类似的，如果  $L < R_1 < R_2$  且  $S[R_1] < S[R_2]$ ，则  $L$  一定不会和  $R_1$  匹配成为答案，因此  $R_1$  也无需进行询问。
- 所以实际上询问也有单调性，双指针即可。





## 链表

- 虽然在现实中很基础且常见，但在做题意义上基本上无甚卵用。

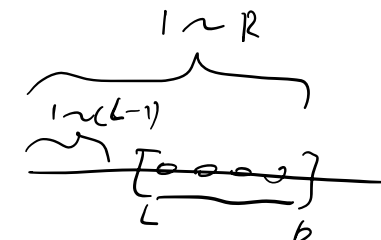
# 前缀和

$$a_1 \sim a_n$$



- 我们用  $O(F(n))$ - $O(G(n))$  表示一个算法需要  $O(F(n))$  时间预处理,  $O(G(n))$  时间单次询问。

- 有一列数字  $\{a_n\}$ , <sup>很</sup>多次询问一个区间的和。  $n, m \leq 1000000$ 。  $[L, R]$  问  $a_L + \dots + a_R$ 。

- 做法很简单, 令  $b[p] = b[p-1] + a[p] = a[1] + a[2] + \dots + a[p]$ , 那么: 
- $a[L] + a[L+1] + \dots + a[R] = b[R] - b[L-1]$  for  $(i = 1 \dots n)$   
$$b_i = b_{i-1} + a_i \quad // \quad b_i = a_1 + \dots + a_i$$
  
$$a \xrightarrow{\text{前缀和}} b$$

- 复杂度  $O(n)$ - $O(1)$ 。

$$O(1) : a_L + \dots + a_R = b_R - b_{L-1}$$

# 差分

$$a \quad [L, R] \quad a_p += v \quad \forall p \in [L, R]$$

• 有一列数，我们多次做区间加操作，最后询问每个位置是啥。

• 考虑前缀和的逆变换，令  $b[p] = a[p] - a[p-1]$ ，也就是  $a$  是  $b$  的前缀和。

• 这样每次  $a$  的一个区间  $a[L, R] += v$ ，等价于  $b[L] += v, b[R+1] -= v$ 。

$$a_i = a_{i-1} + b_i$$

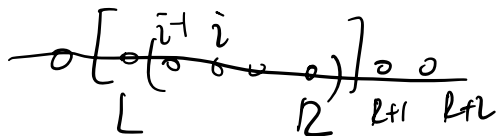
$$\Downarrow$$

$$b_i = a_i - a_{i-1}$$

$$a_i \sim a_{i-1} \quad b \xrightarrow{\text{差分}} a$$

$$\text{令 } \underline{b}_i = \underline{a}_i - \underline{a}_{i-1}$$


• 最后再做一遍前缀和还原回来即可。



对  $i \in [1, L-1]$ .  $a_i \neq a_{i-1}$  为假  $\Rightarrow b_i$  为假  
 $[R+2, n]$   
 $[L+1, R]$ .  $a_i, a_{i-1} += v \Rightarrow b_i$  为假

## 二维前缀和/差分

当  $i=L$  时  $a_i + v$   $a_{i-1}$  没变  $\Rightarrow b_{L+1} = v$   
 $i=L+1$   $a_i$  没变  $a_{i-1} + v$   $b_{L+1} = v$



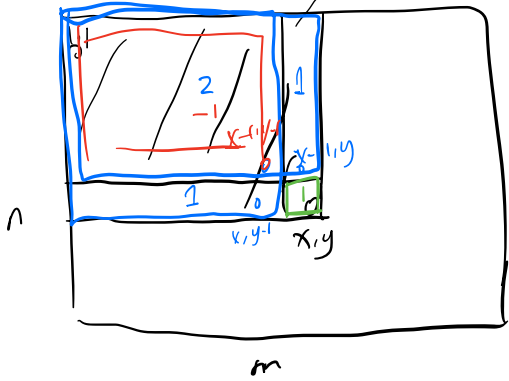
清华大学  
Tsinghua University

$$b_{R+1} = \frac{a_R - a_{R-1}}{+v}$$

- 问题是一样的，不过搬到了二维平面上。
- 二维前缀和： $b[x,y]=b[x-1,y]+b[x,y-1]-b[x-1,y-1]+a[x,y]$
- 矩阵求和： $S(x1,y1,x2,y2)=b[x2,y2]-b[x1-1,y2]-b[x2,y1-1]+b[x1-1,x2-1]$
- 二维差分： $b[x,y]=a[x,y]+a[x-1,y-1]-a[x-1,y]-a[x,y-1]$
- 修改矩形 $[x1,y1,x2,y2]$ 等价于： $b[x1,y1]+=v$ ,  $b[x2+1,y2+1]+=v$ ,  $b[x1,y2+1]-=v$ ,  $b[x2+1,y1]-=v$
- 对更高维的情况，其实质可以理解为容斥原理，在这里就不展开了。

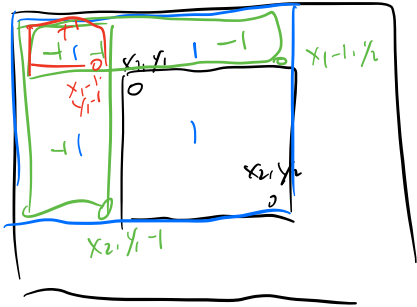
前缀和  
a —> b

前缀和



$$b_{x,y} = b_{x,y-1} + b_{x-1,y} - b_{x-1,y-1} + a_{x,y}$$

$O(nm)$



$$S(x_1, y_1, x_2, y_2)$$

$$= b_{x_2, y_2} - b_{x_2, y_1 - 1} - b_{x_1 - 1, y_2} + b_{x_1 - 1, y_1 - 1}$$

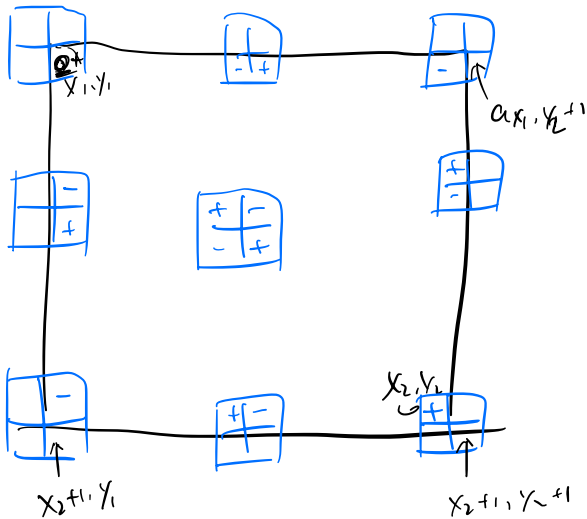
$\xrightarrow{\text{前缀和}}$   
 $\xleftarrow{\text{前缀和}}$

差分

$$b_{x,y} = b_{x,y-1} + b_{x-1,y} - b_{x-1,y-1} + a_{x,y}$$

$$\Downarrow$$

$$a_{x,y} = b_{x,y} - b_{x,y-1} - b_{x-1,y} + b_{x-1,y-1}$$



$$[x_1, y_1, x_2, y_2] += V$$

$$\Downarrow$$

$$\begin{cases} a_{x_1, y_1} += V \\ a_{x_2+1, y_1} -= V \\ a_{x_1, y_2+1} -= V \\ a_{x_2+1, y_2+1} += V \end{cases}$$



## 基数排序

- 如何在 $O(n+32768)$ 的时间复杂度内对int类型进行排序?
- 总之就是将一个int类型看作前16位和后16位。
- 然后用类似桶排先对后16位进行排序，再在此基础上对前16位排序。

WC 17118? 桶排.



清華大學  
Tsinghua University



## Part II: 树形数据结构入门

---



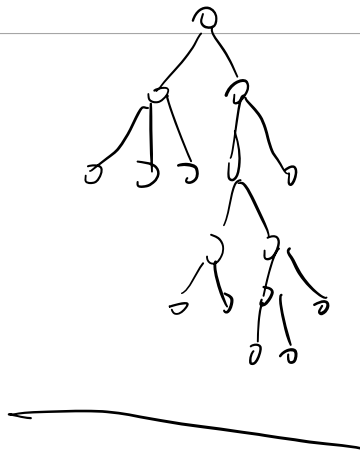
# 目录

- 堆
- 并查集
- ~~• 平衡树~~
- ~~• 树上的几个常用算法:~~
- ~~• 倍增~~
- ~~• 树链剖分: 重链剖分、长链剖分~~



# ◆ 树是什么

- 树就是一类植物的统称。



# 堆

- 世界上目前有很多种堆，比如二叉堆、斐波那契堆、配对堆、左偏树等等，不过由于各种原因，一般默认提到的堆都是二叉堆。下文的堆都是指的二叉堆。
- 二叉堆是一种有根二叉树，每个点有个权值，以小根堆为例，每个点的权值都大于它的父节点的权值。大根堆同理。下文假设是小根堆。
- 能做的事情非常有限，可以在单次严格  $O(\lg n)$  的时间复杂度内插入任意数字 (push)、删除最小的数字 (pop)，并严格  $O(1)$  询问最小的数字。
- 就是各位熟悉的优先队列 <sup>std::</sup>priority\_queue<sup>z.</sup>，所以实际上没有学习其实现的必要。

z. push(x) : 插入  
z. pop() : 弹出最大值  
z. top() → 返回最大值

## ◆ 堆：删除任意数字

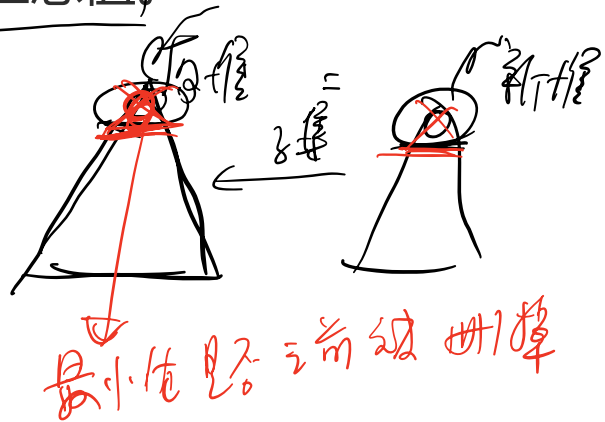
- 一般的堆只支持删除最小值，现在希望支持删除任意值。

- 保证删除的值在堆中出现。

- 只要在维护一个堆，记录哪些值需要被删除即可。

- 每当两个堆堆顶相同时，两个堆同时弹出堆顶

- 好处是使用这个技巧可能会比 `std::set` 快



## ◆◆ Eg1. 堆排序

$O(\lg n)$

$O(\lg n)$

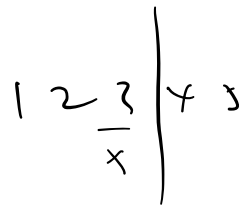
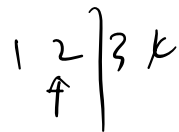
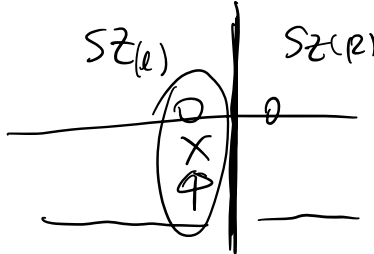
- 啊这真是太简单了，把所有数字 push 进去然后依次 pop 出来即可。
- $O(n \lg n)$ 。

# ◆ Eg2. 带插入的中位数

可重集合  $S = \phi$   
例定数 两个不同

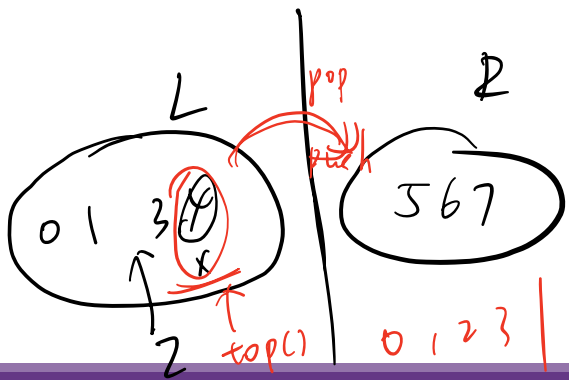
- 每次插入一个数字，然后询问所有数字的中位数。  $1e5$ 。

中位数



$$S_L - S_R = \begin{cases} 0 \\ 1 \end{cases}$$

只需要把  $S$  分成两部分  $L, R$  使  $L$  中元素  $<$   $R$  中元素 ①  
 $|L| - |R| = \begin{cases} 0 \\ 1 \end{cases}$  ②



$\Rightarrow$   $L$  中最大值为中位数

新来  $y$

$y < x$  放到  $L$   
 $y > x$  放到  $R$

①



## Eg2. 带插入的中位数

luogu



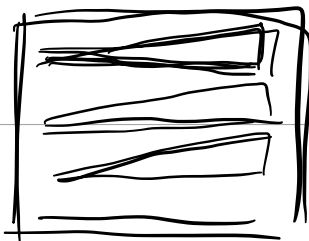
push



清华大学  
Tsinghua University

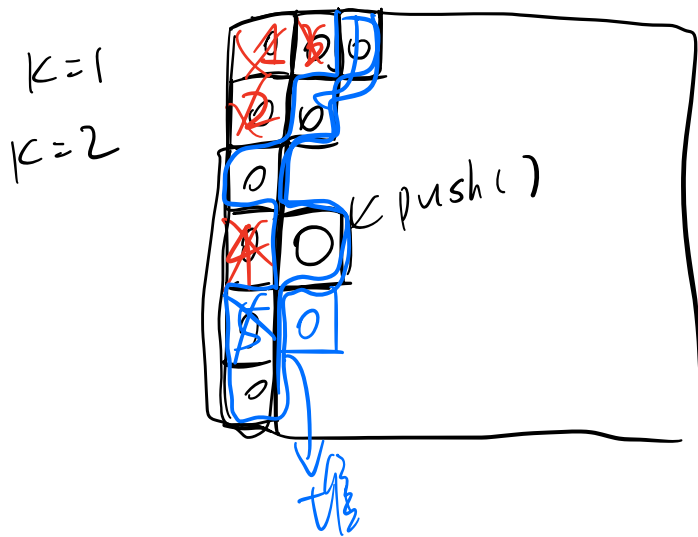
- 每次插入一个数字，然后询问所有数字的中位数。1e5。
- 做法很简单，维护一个大根堆一个小根堆，使得大根堆维护的是前一半的元素，小根堆维护剩下的。每次插入视情况放到小根堆/大根堆里面，并，比如大根堆元素个数不足一半，就从小根堆中拿走最小的这种感觉。每次只有常数级别的变化，因此复杂度  $O(n \lg n)$ 。

## ◆ Eg3. 行有序数表第 k 大



- 有一个  $n*m$  的数表，每一行数字从小到大。
- 一开始这个表并不在内存里，但你可以  $O(1)$  的查询任一位置的值。
- 询问这个数表中第  $k$  小的数字，要求复杂度  $O((n+k)\lg n)$ 。

原理：考虑哪些元素可能成为答案



依次对  $i = 1 \dots k$  看第  $i$  是哪个

在每一行 当前最大的没被删去的  
元素中选最小的，删之

$\text{top}() - \text{pop}()$

$$O((n+k)\lg(n+k))$$

## ◆ Eg3. 行有序数表第 k 大

- 有一个  $n*m$  的数表，每一行数字从小到大。
- 一开始这个表并不在内存里，但你可以  $O(1)$  的查询任一位置的值。
- 询问这个数表中第  $k$  小的数字，要求复杂度  $O((n+k)\lg n)$ 。
- 做法也很经典，有可能成为答案的只会是每一行最左侧的还未被取走的元素，从里面选一个最小的取走即可，发现可以用堆维护之。



# ◆ Eg3+. 第k小点对和

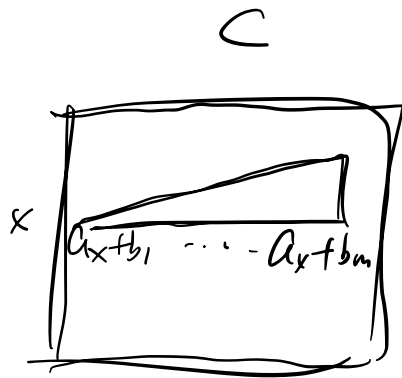
$n \times m$   $(x, y)$   
 $a_x + b_y$

- 给定两个序列  $a_1 \sim a_n$  和  $b_1 \sim b_m$ , 对所有  $(x, y)$  求  $a[x] + b[y]$  中的第  $k$  小。
- $n, m, k \leq 100000$ 。

把  $a, b$  从小到大排序

令  $c_{x,y} = a_x + b_y$

eg3.



## ◆◆ Eg3+. 第k小点对和

- 给定两个序列  $a[1:n]$  和  $b[1:m]$ , 对所有  $(x, y)$  求  $a[x]+b[y]$  中的第  $k$  小。
- $n, m, k \leq 100000$ 。
- 对  $a, b$  排序后令  $c[x, y] = a[x] + b[y]$ , 然后应用 Eg3. 即可。

## ◆ Eg4. 合并石子

- 有  $n$  堆石子，每次你可以选择任意不同的两堆合并，代价是两堆石子的个数之和，问合并成一堆的最小代价。
- $n \leq 100000$ 。

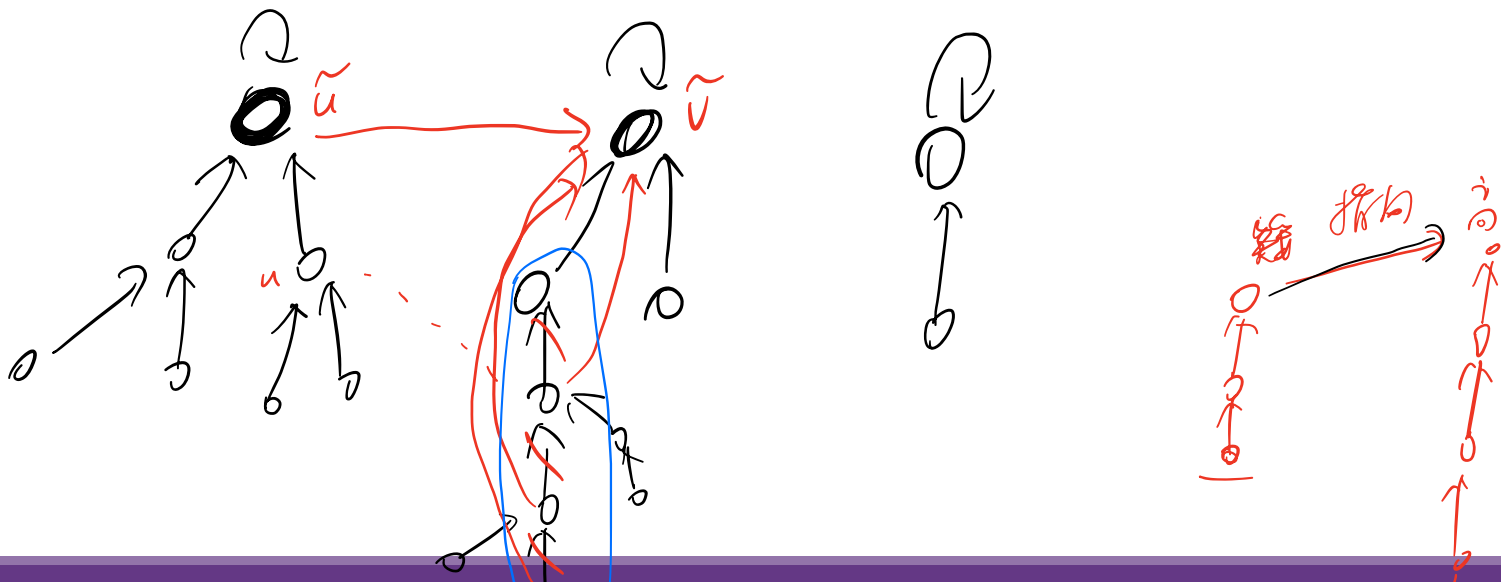
## ◆ Eg4. 合并石子

- 有  $n$  堆石子，每次你可以选择任意不同的两堆合并，代价是两堆石子的个数之和，问合并成一堆的最小代价。
- $n \leq 100000$ 。
- 显然每次取两堆最小的合并即可。

# 并查集

$\{1\}$   $\{2\}$  ...  $\{n\}$

- 能够做的事情很简单：支持合并两个不相交集合，或者询问两个元素是否在同一个集合里。
- 用图论的说法就是：支持用一条无向边连接两个点，或询问两点是否连通。





## ◆ 并查集实现

- 每个集合选一个代表元。初始时，各元素自己作为代表元形成一个集合。
- 假设可以使每个集合形成一个有根树的结构，每个元素沿着父亲方向走，最后到根就是代表元，则：
  - 当试图合并两个集合时，设两集合代表元为  $x, y$ ，则设置  $x$  的父亲为  $y$  即可。
  - 当查询两个元素是否在同一集合时，看他们所在有根树的根是否一样即可。
- 显然，此时的复杂度最坏可能是  $O(n^2)$  的（例如连接成链）。

## ◆ 并查集实现

- 两个优化：
  - 路径压缩：每从  $x$  进行一次查询得到  $x$  的根  $y$ ，就将  $x$  到  $y$  的路径上的所有点的父亲设置为  $y$ 。这样下次对这些点只需要询问 1 次就能得到答案。
  - 按秩合并：当合并两集合的代表元为  $x, y$  时，总是将树高度小的代表元的父亲设置为树高度大的代表元。

tarjan  $O(n \cdot \alpha(n))$

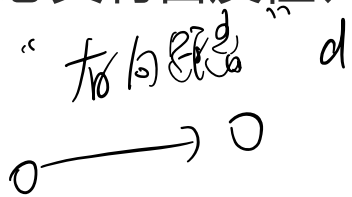
- 可以证明使用上述两个优化，其时间复杂度为均摊  $O(\alpha(n))$ ，其中  $\alpha(n)$  为一个增长极其缓慢的函数，因此并查集在实践中近似线性。
- 不过一般只写路径压缩就足够了。



- 按秩合并也可以改为启发式合并（即树大小小的指向大小大的），从而在某些场景下做到单次操作严格  $O(\lg n)$ ，而非均摊。

## ◆ 加权并查集

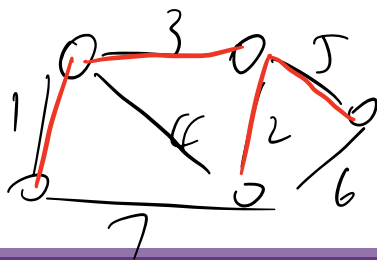
- 并查集中每个点指向父亲的边可以设置一个权值，表示两点间的某种信息
- 通常需要该信息具有自反性、反对称性、传递性



$$d(x, x) = 0$$

$$d(x, y) = -d(y, x)$$

$$d(x, y) + d(y, z) = d(x, z)$$

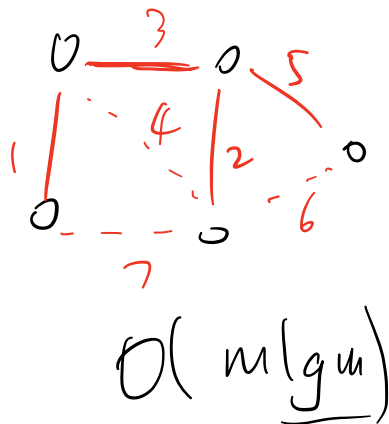
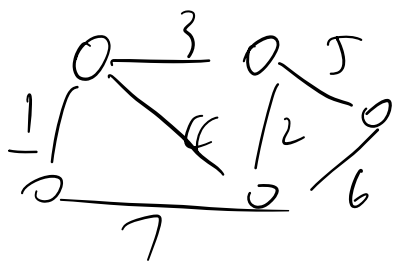


$$1 + 2 + 3 + 5$$



## ◆ Eg5. 最小生成树 Kruskal

- 给一张图，求其最小生成树。  $n, m \leq 100000$ 。
- 一般最小生成树有三种算法：Prim, Kruskal 和 Boruvka (Sollin), 通常使用第二种；第三种在少数题里用于帮助分析性质；第一种则通常只在稠密图上使用。
- 做法就是按边权排序然后能加就加。





## Eg7.

JSOI 2008, 星球大战



清华大学  
Tsinghua University

- 给一张图，每次删掉一个点及相连的边，求剩下的图中的联通块数。  $n \leq 100000$ 。
- ~~动态连通图问题可以使用ETT来解决，如果要写的话估计也就十几k左右吧。~~

## ◆ Eg7.

- 给一张图，每次删掉一个点及相连的边，求剩下的图中的联通块数。  $n \leq 100000$ 。
- ~~动态连通图问题可以使用ETT来解决，如果要写的话估计也就十几k左右吧。~~
- 我们注意到并查集没法删除。
- 因此我们倒着从空图往回做，就变成了加边求联通块数的问题。

# Eg8.

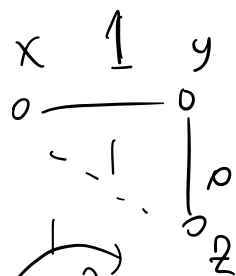
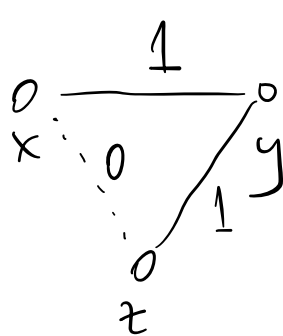
NOIP 2010 关押罪犯



- 有一张图，边有边权。
- 你要给每个点确定是黑色或者白色，使得两端颜色相同的边的边权最大值最小。

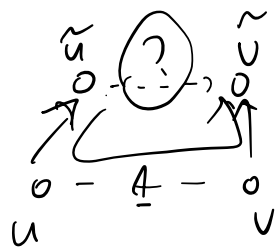
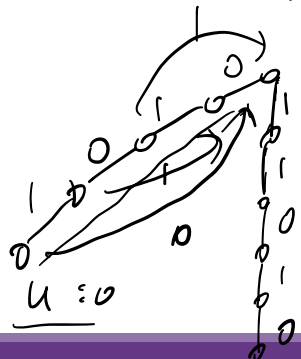
$n \leq 100000$ .

把边从大到小排序，加边，如果两端颜色不同  
直到做不到为止



$$c(x, y) = \begin{cases} 0, & \text{if } x, y \text{ 相同} \\ 1, & \text{else} \end{cases}$$

$$c(x, y) + c(y, z) \equiv c(x, z) \pmod{2}$$



$$c(\tilde{u}, \tilde{v}) = c(\tilde{u}, u) + 1 + c(v, \tilde{v}) \pmod{2}$$

v:1

## Eg8.

- 有一张图，边有边权。
- 你要给每个点确定是黑色或者白色，使得两端颜色相同的边的边权最大值最小。  
 $n \leq 100000$ 。
- 一个直观想法是尽可能让边权大的边，两端颜色不同。
- 因此我们从大到小加边，看能否钦定两端的颜色不同。
- 带权并查集维护每个点和他并查集上的父节点是否颜色相同即可。

类似 100. NOIP 2023 之 颜色图

## ◆ Eg9.

- 有三种生物A, B, C, 其中A吃B, B吃C, C吃A。
- 有n个生物, 每个要么是A, 要么B, 要么C。
- 每次告诉你谁吃谁或者谁和谁同类, 或者问你谁吃谁/谁和谁同类是否一定不成立。  $1e5$ 。

## ◆ Eg9.

- 有三种生物A, B, C, 其中A吃B, B吃C, C吃A。
- 有n个生物, 每个要么是A, 要么B, 要么C。
- 每次告诉你谁吃谁或者谁和谁同类, 或者问你谁吃谁/谁和谁同类是否一定不成立。  $1e5$ 。
  
- 带权并查集维护一个点和其父节点的关系 (谁吃谁或者是否同类) 。
- 实现上其实这就是一个模 3 剩余类, 0 表示同类 1 表示吃 2 表示被吃。
- 还有一种维护三个并查集的做法并不推荐, 感觉缺乏拓展性。

# ◆ Eg10.

每次告诉你一个区间的和，或者问能否确定一个区间的和。

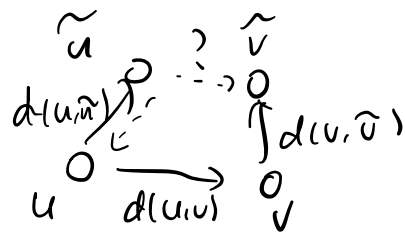
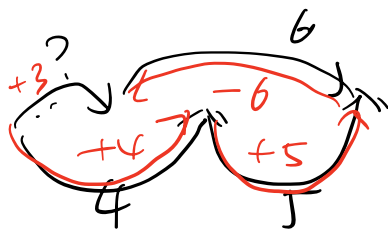
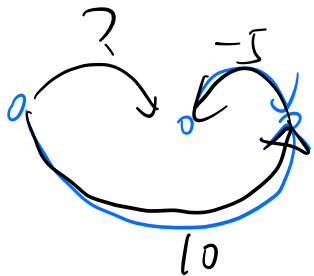
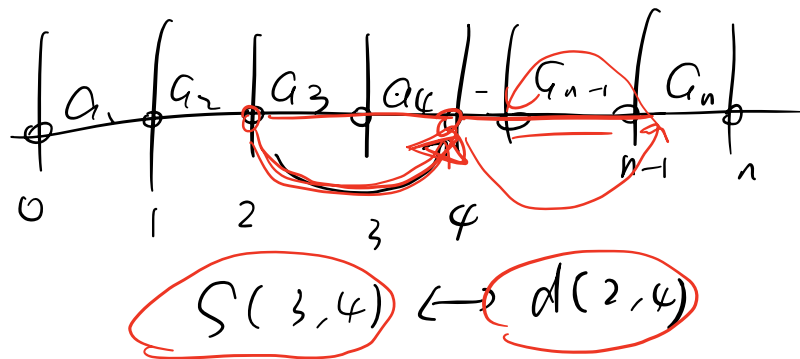
- 有一列数字，每次告诉你一个区间的和，或者问能否确定一个区间的和。  
 $n \leq 100000$ .

eg.  $S(1, 3) = 4$

$S(4, 5) = 5$

$S(3, 5) = 6$

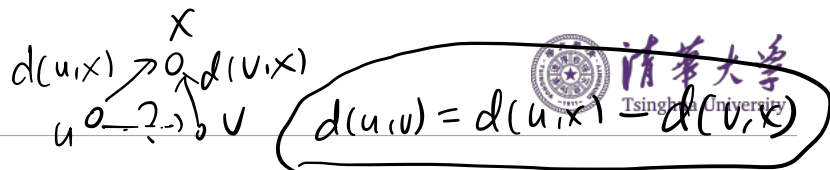
$\Rightarrow S(1, 2) = 3$



$$d(\tilde{u}, \tilde{v}) = -d(u, \tilde{u}) + d(u, v) + d(v, \tilde{v})$$



## ◆ Eg10.



- 有一列数字，每次告诉你一个区间的和，或者问能否确定一个区间的和。  
 $n \leq 100000$ 。
- 考虑告诉你一个区间的和本质上就是在讲从一个缝隙走到另一个缝隙的距离。  
这里的距离满足  $\text{dis}(x,y) + \text{dis}(y,x) = 0$ 。
- 然后并查集维护一个点和父节点的距离即可。