长河乱语 四

OTTF 2024 6 6 起写

OTTF 2024 9 16 终写

第0章 引子

日月同辉，只是一个正升，一个正落。我们的主人公还是在石凳上，朝着对面的沃柔德说道：“你准备好了吗？这次会非常有趣的。”

“哦？”沃柔德灵动地闪过眼睛，“准备好了，现在就开始吧。”

第1章 并查集万岁

> 独学而无友，则孤陋而寡闻。
>
> ——《礼记·学记》

你有朋友吗？你有许多朋友吗？好吧，都耐着性子到这里了，你或许并没有许许多多能与你共处的朋友，不然你也就不会来了。

但是总有人会有很多朋友，而这个人都有这么多朋友了，也没准儿会让这些朋友们相互认识一下。换言之，我们先作出这样的假定：若A为B的朋友，且A为C的朋友，则B为C的朋友。

看起来很欢乐！这样大家就都能有更多朋友了！不过，请设想这样一种场景：A曾经与B结识，这两个人相谈甚欢，于是便成为了朋友。之后，A又碰到了C，并且与之谈笑风生，也成为了朋友。但是，当B见到C时，他们并不知道彼此有A这个共同的朋友——这实在是糟糕极了。

我承认这是个比较牵强的例子，但是，我们有没有一种方法，能使让这两个“朋友”相互知道彼此是“朋友”呢？确实有。

不如接着说故事吧：A和B的关系实在太好了，于是B对A说道：“A啊，我要用笔记录下来我们之间的友情，我要在纸上写下‘B与A是朋友’！”A就点了点头：“太好了！这样下来，只要你和我看到这张纸，就会因我们之间的友情而感到快乐了！”

后来，C也感到自己与A的关系很好，就也在一张纸上写下了“C与A是朋友”。而在B与C碰面

时，B看到了C的那张纸，就惊讶地拿出了自己的纸——就这样，B与C也互相知道彼此是朋友了。

你看，只要某些人在纸上记录自己的一个朋友，人们就能知道互相是不是朋友了。现在我们来扩展这种行为：如果有一个人D结识了B，这两个人也成为了朋友，所以为了建立朋友之间的连接，D在纸上写下“D与B是朋友”；然后E又与C成为了朋友，便在纸上写下“E与C是朋友”。我们用这样的图来表示现在的纸条系统：

```
 A
/ \
B C
| |
D E
```

某日D与E碰面了，这两个人突然想检查自己的朋友情况。于是D拿着纸条找到B，然后又根据B的纸条找到A，A没有写纸条，现在D可以把A当作自己的“最高朋友”。而E也如此做，也发现A是自己的“最高朋友”。这样一来，D和E肯定是朋友了。

但是，D和E觉得如果每次检查朋友情况都要大费周章地先找一个朋友，再找到另一个朋友，未免也太繁琐了。于是D修改了自己的纸条，直接写上了自己的最高朋友，也就是“D与A是朋友”；而E也把自己的纸条改成了“E与A是朋友”。于是我们让D和E也在简图中与A相连：

```
D-A-E
 / \
B   C
```

看起来确实不错！但是还有一个问题，现在假设F和G是一对朋友，G的纸条上写着“G与F是朋友”。之后，G结识了E，我们又该如何处理朋友关系呢？G和E都写过纸条了，如果让这两个人中的一个修改纸条，就肯定会失去某些朋友。我们还有什么方法呢？

且住，根据我们的理论，A和F都是最高朋友，而且都没有写纸条。那么，让没有写纸条的人写一个关于对方的纸条，就可以维护朋友关系了。换言之，让A写一个“A是F的朋友”或者让F写一个“F是A的朋友”都可以解决问题。

那么该让谁来写呢？我们注意到，让某一个人来写，其实就是取消了其最高朋友的身份，这时其原本的朋友离最最高朋友也就远了一点，这些人在日后检查朋友关系时也就慢了一点，我们要让这些人尽可能少。观察到A原本的朋友更多，F原本的朋友更少，所以我们让F写“F是A的朋

友”。现在A处于一个画面的中心：

```
  G
  |
  F
  |
D-A-E
 / \
B   C
```

想必你已经发现了，我们现在为止，已经建立了一个可以处理人们之间朋友关系的系统。具体而言：每个人都会有一个写下自己一个朋友关系的纸条；当两人从不是朋友成为朋友时，两人找到自己的最高级朋友，让朋友数少的那个最高级朋友写下纸条，表示另一个最高级朋友是自己的朋友；如果两人想要知道对方是不是自己的朋友，就只需要比对这两人的最高级朋友是否相同；顺便，在一个人找到自己的最高级朋友后，就会将纸条直接写为最高级朋友。

其实，我们的这个系统大有用处，它不仅可以处理朋友关系，还可以处理一切带有“传递性”的关系。比如说，如果直线l_1平行于l_2，l_2又平行于l_3，那么l_1就肯定平行于l_3，这与朋友关系是类似的，我们的系统也可以处理平行关系。

为了处理各种带有传递性的关系，我们把这这种系统用伪代码表示：

```
一列数nums，长度为n
一列数size，长度为n

初始化（）：
    i从1到n循环：
        nums[i] 为 i
        size[i] 为 1

查找（一个数num）：
    如果nums[num] 不等于 num
        nums[num] 为 查找（nums[num]）
    返回 num

合并（一个数one，一个数two）：
    one 为 查找（one）
    two 为 查找（two）
    如果（one 不等于 two）：
        如果 size[one] 小于 size[two]：
            nums[one] 为 two
            size[two] 为 size[two] + size[one]
        否则：
            nums[two] 为 one
            size[one] 为 size[one] + size[two]
```

在上面的伪代码中，`nums`相当于小纸条，但不同的是在初始时，每个人的小纸条上都会写着其自己是自己的朋友。而`size`则表示当一个元素相当于“最高级朋友”时，它的“朋友”的总数，如果一个元素不是“最高级朋友”，这个值就没有什么用处。

这种系统叫做并查集，你可能会对它感到神奇。实际上，它确实十分神奇。在未来的一段时间中，我们将看到很多神奇的东西。这些东西被我们用来存储数据、应用数据，我们称它们为“数据结构”。在欢乐的信息世界中，数据结构是你忠实的朋友。

第2章 栈与队列

> 难道先入住的就必须最后才能走吗？
>
> ——某人对“客栈”一词的疑问

你对“数组”这个词，估计很熟悉了。而现在，让我介绍一个听起来更高端的名词——“线性表”。它就是很多元素排成一条线形成的东西——这好像还是“数组”。

这个名词——或者说这个概念——有什么用呢？好吧，我现在将介绍两种数据结构，而它们都属于线性表。

想象一下，既然我们让众多元素排成一条线，那么我们如何从零添加元素呢？嗯……在末尾添加元素怎么样？既然是一条线，我们就能定义它的“开头”和“末尾”。我们再选择一种删除元素的操作，就在末尾删除元素吧。然后，嗯，既然我们只能在末尾添加与删除元素，要不我们再激烈一点，只能访问末尾的元素怎么样？好极了，现在我们拥有了一个“栈”。

是的，这种只能在末尾进行操作的线性表叫做“栈”，你会发现栈就是一个大瓶子，你可以把元素——想象成球——一个一个从瓶口放进去，或者一个一个从瓶口拿出来，同时你也只能看到瓶口的元素。

只要稍加改动，我们还能创造出另一种线性表。虽然我们只能在末尾添加元素，但我们可以把删除元素的位置改成开头。这时为了方便，我们就让访问元素的操作既能在开头又能在末尾吧。这样，元素从末尾进来，从开头出去，就像是在排队一样，我们把这种线性表叫做“队列”。

栈和队列好像只是在数组上套了一层皮，反正我有这种感觉。但是当然了，这两个概念是有用的。其实在实现搜索算法时，我们就会用到它们：你或许记得，我们在实现宽搜时提前使用了队列的概念；而深搜则需要递归，在计算机中，为了实现递归，系统会建立一个栈。

你会发现，栈和队列的长度都可能会增加或者减小，这意味着我们需要对数组上下其手，毕竟在我们的为代码中，数组的长度是规定好的。就拿栈来说，如何扩张一个栈的长度呢？我们可以先采用一个长度，在栈满了的时候，建立一个更大的数组，让这个数组替代原来的数组，使得下一次“满了的时候”远一点儿。

很不错的主意！但是，如何“让这个数组替代原来的数组”呢？我们可以采用一种叫“指针”的技术。简单来说，设立一个指针`p`，先让它指向“一段空间”，我们可以不给这段空间起名字，因为实际上我们会用`p`来操控它,就像`p`是一个数组名一样。在这段空间满了的时候，我们建立“一段新空间”，这段空间要更大，比如说是原空间的两倍，它被`newp`所指向。我们再让这段新空间前面一段的值等于原空间，然后让`p`变为`newp`，这就相当于替代了原来的数组。当然，我们还要告诉机器原来的空间我们不用了。

非常不错的想法！你可能会问：欸？我们就不能直接改变一个数组的长度吗？这是个很好的问题，某些编程语言确实提供了改变数组长度的功能，但请假定我们的伪代码不能，这样，我们就能看看指针的绚丽操作了。

在接下来的代码中，我们用到一个名为`capacity`的变量，它表示`p`指向的空间的长度，又或者说是容量。还有一个名为`last`的变量，它表示栈的末尾是数组的第几项，这使得它的值是栈的长度。值得一提，在栈中，我们可以把末尾称为栈顶。

```
一个指针p，一开始指向长度为8的空间，里面都是数，指向的空间的长度为capacity
一个数last，一开始为0

栈顶（）：
    返回p[last]

设置长度（一个数size）：
    一个指针newp，指向一个长度为size的空间
    i从1到last循环：
        newp[i] 为 p[i]
    p 为 newp
    capacity 为 size

添加（一个数num）：
    如果 last 大于等于 capacity：
        设置长度（capacity*2）
    last 为 last+1
    p[last] 为 num
```

而如果要删除栈顶呢？我们可以只见让`last`减一，而如果删除的元素太多，空间却没有变化，就会显得很浪费，这时，我们就让长度减小一点。

```
删除（）：
    last 为 last-1
    如果 last 小于等于 capacity/2：
        设置长度（capacity/2）
```

那么队列呢？其实也是一样的道理。我们将队列的开头称为队头，并将队列的末尾称为队尾。为了处理删除队头，我们需要一个变量`first`，让它的值等于队头在数组中的项数减一。

```
一个指针p，一开始指向长度为8的空间，里面都是数，指向的空间的长度为capacity
一个数first，一开始为0
一个数last，一开始为0

队头 () :
    返回p[first+1]

队尾 () :
    返回p[last]

设置长度 (一个数size) :
    一个指针newp，指向一个长度为size的空间
    i从first+1到last循环:
        newp[i - first] 为 p[i]
    p 为 newp
    capacity 为 size
    last 为 last-first
    first 为 0

添加 (一个数num) :
    如果 last-first 大于等于 capacity:
        设置长度 (capacity*2)
    last 为 last+1
    p[last] 为 num

删除 () :
    first 为 first+1
    如果 last-first 小于等于 capacity/2:
        设置长度 (capacity/2)
```

我们好像一直没有处理当栈或队列为空时，访问其中元素的情况，我们理应作出反应的。你可以在对应的函数上动手动脚，自己实现一个人性化的提示。

值得一提，我们的代码中出现了`p 为 newp`这种句子。在这种句子之后，`p`本来指向的空间就不被使用了。某些真正的计算机语言要求你明确“释放”这些空间，不然就不会处理它们。而另一些真正的计算机语言则会“智能地”处理它们。甚至还有某些计算机语言允许你采用多种方法操控内存。在伪代码中，我们认为不被使用的空间会立即被好心的计算机之灵处理，让我们为它欢呼。

第3章 链表

> 若以大船小船各皆配搭，或三十为一排，或五十为一排，首尾用铁环连锁，上铺阔板，休言人可渡，马亦可走矣，乘此而行，任他风浪潮水上下，复何惧哉？
>
> ——《三国演义》庞统

你肯定发现了，虽然在栈和队列中，简单的添加和删除元素是$O(1)$的，但与之相伴的使用一块新空间的复杂度却是$O(n)$的。虽然我们使用每次扩展成两倍或减小为一半的方式尝试减少这种操作，但我们或许还是期望一种任何时候都能做到$O(1)$添加或删除的数据结构。

欸？如果我们在添加新元素时，就的空间已经满了，能不能直接开辟一点新空间，通过某些手段把新旧空间联系起来，而不直接挪动元素呢？又或者说，要不更激进一点，把元素的存储位置完全打乱，转而在每个元素中存储自己的下一个元素在哪里，这样不管在什么位置添加或删除元素，都可以通过一些操作，来达到$O(1)$。

但是话又说回来，既然我们的元素存储被完全打乱，我们也就不能通过元素的编号直接找到这个元素，而是需要进行一个$O(n)$的查找。这么奇怪的数据结构有存在的必要吗？当然有，你会发现它在头部或尾部的插入和删除其实可以做到$O(1)$。

可以联想到，要让一个元素储存自己的下一个元素在哪里，可以用指针实现，这个时候指针指向的就是一个元素了。而由于元素自己有一个值，比如在我们的例子中，是一个整数，我们就需要进行一种“捆绑”，把一个整数和一个指针“绑”在一起，变成一个“大元素”。我们可以写：

```
一个大元素由一个整数num和一个指针next组成
```

为了添加一个大元素，我们或许需要明确新添加的元素在哪个位置，又或者说——哪个元素的后面。所以我们需要一个指针`p`，表示添加的元素在`p`所指向的元素的后面。但是，在一个元素都没有的时候，我们仍然需要一个指针。所以我们可以先写一个名叫`head`的指针，然后实现添加操作。

```
一个指针head，指向一个空大元素

添加（一个整数num，一个指向大元素的指针p）：
    一个指针newp，指向一个大元素
    newp的num 为 num
    newp的next 为 p的next
    p的next 为 newp
```

在写完这些后，我们就可以用`添加（0，head）`这样的语句来添加元素。但是，我相信你一定在问：`newp的next 为 p的next`和`p的next 为 newp`到底是什么意思啊？！

首先，我们知道，我们在某些时候把指针当作它指向的元素来书写伪代码，所以`的`前面的两个指针分别代表一个大元素，我们显然是在为这两个大元素的指针赋予新值。所谓`newp的next`，就是我们新添加的大元素中，指向下一个元素的指针，我们要把它变成什么呢？我们要把它变成`p的next`。这是为什么呢？新添加的大元素前面，应该会有一个前老元素，前老元素后面本来也会有一个后老元素，它们本来是挨着的。现在新元素位于前老元素的后面，那么新元素的后面就是后老元素，我们可以借用前老元素的`next`来得到后老元素的位置。

稍等？那么本来没有老元素，或者前老元素后面没有元素的情况该怎么办呢？这种情况下，我们认为所谓“前老元素”和“后老元素”都是虚空，虚空中的东西都是虚空，指向虚空不会出什么问题，这代表新元素后面没有元素了。

既然这样，`p的next`那一句也好说了——我们需要为前老元素的`next`赋予一个新值，现在新元素就在前老元素的后面，我们自然要让这个指针指向新元素。

解决了添加元素，我们自然而然地想要处理删除元素的操作。在这里，我们删除`p`所指向的那个元素：

```
删除（一个指向大元素的指针p）：
    p的num 为 p的next的num
    p的next 为 p的next的next
```

稍等，这是删除吗？这好像是用`p`所指向元素的下一个元素的值来覆盖了`p`所指向的元素。哦！这样一来，`p`所指向的位置变成了原来的下一个元素，甚至包括`next`指针，所以`p`原来指向的位置的元素也就灰飞烟灭了。那么那个元素的下一个元素呢？在它的值被复制后，没有东西去指向它了，可怜的它不在处于我们的数据结构中，被处理垃圾空间的计算机之灵收走了。

这种数据结构显然建立于指针这种能够指向其他东西的工具之上，又或者说，这种数据结构热衷于把元素连起来。因此，我们把这种数据结构称为“链表”。

注意到，我们的链表实现非常简单，其实这是因为我们实现的是比较简单的“单向不循环链表”。这意味着我们的每个元素都只指向它的下一个元素，同时链表的首部和尾部也不会相连。是的，你猜的不错——在某些时候，我们不仅需要让链表中的元素指向它的下一个元素，还需要让它们指向上一个元素，这称为“双向链表”；而在另一些时候，我们会让链表首尾相接，成为一个大圈，这称为“循环链表”；你更可以把它们结合起来，制作出“双向循环链表”。如果你想要实现这些酷炫的数据结构，你可以自己去尝试。在遇到问题的时候，你可以在互联网技术博文中尽情冲浪。

第4章 二叉堆

> 人生就是不公平的，你慢慢习惯吧你。
>
> ——《海绵宝宝》派大星

我们的数据结构真是太厉害了，不仅能建立朋友之间的关系，还能支持奇形怪状的线性表。话说，我们能构造出可以排序的数据结构吗？当然可以！请看这张示意图：

```
    2
   / \
  6   10
 /
9
```

可以看到，在这个示意图中，数字下方可能会伸出一根或两条线，线上方的那个数字必定小于线下方的那个数字。这样一来，数据结构最上方的数字肯定是最小的。既然这样，对于一些数字，我们只需要先把它们加入这个数据结构，然后每一次查看最上方的数字，并把它删除，这样循环往复，就能实现排序了。

说的很不错！现在我们需要实现在这个数据结构中添加一个数以及删除其最上方数字的两个操作。先让我们想想，如何添加一个数呢？

我们发现，图上的这些数好像也可以被认为是从上到下、从左到右依次排列的。那么，不管我们要添加什么，就可以先在现在的“下一个位置”添加上去。比如说，让我们添加一个7：

```
    2
   / \
  6   10
 / \
9   7
```

没有出问题，但是为什么呢？你会发现，这是因为`7`上面的`6`要小于它。如果我们添加的是`1`呢？

```
    2
   / \
  6   10
 / \
9   1
```

哦吼，我们的数据结构出了一点问题——`1`上面的`6`要大于它！该怎么解决呢？嗯……把这两个数交换怎么样？

```
    2
   / \
  1   10
 / \
9   6
```

嗯哼，好像确实不错，`1`下面的位置没有错误了。不过——它上面的数还是大于它。我们自然可以再进行一次交换。

```
    1
   / \
  2   10
 / \
9   6
```

完全没有问题了！所以说，要想添加一个数，我们只需要先把它放在“末尾”，然后检查它上面的数是否大于它，如果是，那就把它们交换。一直这样检查下去，直到它上面的数小于它。

如果想用伪代码来书写一下，我们可以怎么样呢？貌似可以用指针来表示那些线条。不过，继续按照“从上到下、从左到右依次排列”的逻辑思考，我们可以给这里的每个数赋予一个编号。

```
    1
   / \
  2   3
 / \
4   5
```

看出来了吧：如果一个数在这里的编号是n，那么它用线连向下方的两个数的编号就分别是$2n$与$2n+1$；相对应的，它用线连向上方的那个数的编号就是$n/2$，不过可能需要向下取整。

有了这种利用编号操纵数据结构的方式，我们可以轻松用数组来完成这些操作。

```
一列数nums，一开始为空，其长度为n

最小值（）：
    返回nums[1]

添加（一个数num）：
    nums添加num

    一个数id，为n
    如果nums[向下取整（id/2）] 大于 nums[id]，重复执行：
        交换（nums[向下取整（id/2）], nums[id]）
        id 为 向下取整（id/2）
```

稍等，这会不会太暴力了？哈哈，注意到如果我们一共有n个数，那么这个数据结构在视觉上最多只有$logn$层，而每一次交换操作都会“上升”一层。所以添加的时间复杂度是$O(logn)$，值得信赖。

那么删除最上方的数该怎么做呢？我们发现，操作数组的最后一项真是太容易了！只需要直接添加或删除就可以了！所以我们干脆把最上方的数与最后一项交换，然后直接删除它。

```
    5
   / \
  2   3
 / \
4   1
```

```
    5
   / \
  2   3
 /
4
```

现在最上方的数——或者说最小值——被我们抛弃了，然后不出意料地，数据结构又需要修正了。观察那个`5`，它大于下面的那两个数，我们很自然地想到应该与其中的一个交换——不过是哪个呢？嗯……应该与较小的那个交换！

来设想一下，如果`5`与并不较小的`3`交换，那么`3`称为最上面的数后，会大于它下面的`2`。这样，我们甚至需要在这一层再进行一次交换，这可不好。所以，我们选择与较小的那个数交换。

```
删除 () :
    交换 (nums[1], nums[n])
    nums删除末尾

    一个数id，为1
    如果nums[id] 大于 最小 (nums[id*2], nums[id*2+1]) ，重复执行:
        如果 nums[id*2] 小于 nums[id*2+1]:
            交换 (nums[id], nums[id*2])
            id 为 id*2
        否则:
            交换 (nums[id], nums[id*2+1])
            id 为 id*2+1
```

很显然$O(logn)$的，删除操作也是。有了删除操作，我们只需要进行n次添加，n次删除，就可以进行一次排序，总复杂度是$O(nlogn)$。

这种数据结构的名字是什么呢？你看，我们的数据结构就像是把数字堆在了一起，所以我们可以叫它“堆”。而每个数字下面最多有两个分叉，所以我们叫它“二叉堆”。另外，我们管利用它

的排序方法叫做“堆排序”。

第5章 稀疏表

> 把“Sparse Table”叫做“ST表”？不要把递归用在奇怪的地方上啊！
>
> ——某语言学爱好者

你肯定记得，2是一个很神奇的数字，我们可以通过它的非负整数次方完成各种奇怪的任务。现在，我们来看看“区间中的可重复贡献问题”。

这是什么意思呢？对于一个运算，如果在一个数上重复进行两次一样的运算，最后得到的结果总是一样的，那么这种运算就是“可重复贡献”的。比如说，定义一个“取最大值”的运算，就是把一个数变成其与另一个数中较大的。现在有一个数6，我们用7对它进行运算，得到的结果就是7。我们还是用7，再对7进行运算，得到的结果还是一样的。相反，加法就没有这种性质。

有“可重复贡献”了，那么“区间”呢？很简单，就是给你一列数，每次划定一个区间，求出区间中这些数进行“可重复贡献”运算后的结果。比如说，给定$2, 4, 6, 7, 8$这五个数，划定从第2个数到第4个数的区间，需要求出区间中所有数的最大值，那么答案就是7。

记序列长度为n，我们当然可以用$O(n)$的暴力扫描解决每个询问，但如果询问也有很多，就有点麻烦了。现在，由我来介绍稀疏表——一种可以用$O(1)$的复杂度解决每个询问的数据结构！

$O(1)$？这太疯狂了！哈哈，为了做到这么妙的复杂度，我们其实需要提前付出一点代价——进行一点预处理。

还是看$2, 4, 6, 7, 8$这五个数，如果我问你“从第1个数开始的1个数中，最大的数是几”，你肯定知道答案是2。而如果我把问题改成“从第2个数开始的1个数中，最大的数是几”，你也一眼就能看到答案是4。那么，如果我问“从第1个数开始的2个数中，最大的数是几”，答案是？好吧，这也不难，是4。

但是等一下，发现了吗？“取最大值”是我们使用的运算，而把前两个问题的答案进行这个运算后，得到的正是最后一个问题的答案，这是巧合吗？显然不是，一眼就能看出来，把前两个问题所对应的区间拼起来，得到的就是最后一个问题所对应的区间。

那么，我把三个问题改成“从第1个数开始的2个数中，最大的数是几”“从第3个数开始的2个数中，最大的数是几”“从第1个数开始的4个数中，最大的数是几”，它们的答案还是满足上面的关系。你肯定发现了，除了区间可以正好拼起来之外，每个问题中的那个表示区间长度的数

还正好是2的非负整数次幂。这就使得我们可以很方便的预处理像这样的问题的答案，比如“从第3个数开始的2个数中，最大的数是几”这个问题的答案就可以由“从第3个数开始的1个数中，最大的数是几”“从第4个数开始的1个数中，最大的数是几”这两个问题推出。

嗯，假设我们已经推导出了每一个“从第i个数开始的第2^a个数中，最大的数是几”这样的问题，就把它的值表示为$init(i,a)$吧。对于一个更一般的问题，比如“从第1个数到第5个数，最大的数是几”，又该怎么解决呢?

啊哈！我们实际上是处理了很多长度为2的非负整数次幂的区间的答案，而现在要计算一个更一般的区间的答案。我们只需要用两个预处理过的区间完全覆盖要求答案的区间，并且做到不覆盖区间外的内容，就可以直接得出答案了！至于重复覆盖？不用管！这是可重复贡献问题！比如刚才的问题，我们用$init(1,2)$与$init(2,2)$两个值就可以得出答案。

注意到，确定所需要的区间时，最重要的是那个a。为了完全覆盖所求区间，设其长度为len，我们要让$2\times 2^a\geq len$，也就是$2^{a+1}\geq len$，也就是$a\geq log(len)-1$。为了方便，我们可以直接让a等于$log(len)$，可以想象到这不会让我们使用的两个预处理区间超出所求区间范围。

我们可以写出这样的伪代码：

```
一列数nums，长度为n
一列元素init，长度为n，每个元素为一列数，长度为log(n)

预处理 () :
    i从1到n:
        init[i][0] 为 nums[i]

    i从1到向下取整 (log (n) ) 循环:
        j从1到n循环:
            如果 j+2^i-1 大于 n:
                退出此层循环
            init[j][i] 为 最大 (init[j][i-1], init[j+2^i-1][i-1])

查询 (一个数l，一个数r) :
    一个数a，为向下取整 (log (r-l+1) )
    返回 最大 (init[l][a], init[r-2^a+1][a])
```

这样一来，我们是$O(nlogn)$的预处理，$O(1)$的处理询问，快极了。但是……还有一个问题。我们的代码好像使用了很多`log`函数，它应该也是有复杂度的，能把这个复杂度消掉吗？确实可

以！我们可以改为使用一个数组，根据log的定义一个个推出它向下取整的值。

```
一列数log，长度为n，第1个数为0，第2个数为1

log预处理（）：
    i从3到n循环：
        log[i] 为 log[向下取整（i/2）] + 1
```

这样，询问的复杂度就能是真正的$O(1)$了。

为什么这种数据结构被称为“稀疏表”呢？嗯……可能是因为它的预处理方式很奇妙，很稀疏？嗯……

第6章 区间求和

> 君子和而不同，小人同而不和。
>
> ——《论语》中的孔子

我们提到过，加法不是可重复贡献的运算，我们也就不能用稀疏表解决一段区间上的加法询问。不过加法有另一个性质——可差分。

这又是什么意思呢？你看，$1+2=3$，而$3-2=1$。在进行一次加法运算后，我们可以轻松地将结果通过减法转变为原来的数。换言之，即使我们多进行了一些加法运算，只要我们拥有操作数的信息，我们就可以消除这些多余操作。

还是没懂吗？哼哼，来看一个例子。还是$2,4,6,7,8$这个数列，我们要去解决很多次区间求和询问，也可以用一些预处理来作为帮助。简单来说，要是我们知道从第1个数到其他每个数的区间和这几个信息，就可以用减法得到每个问题的答案了。

举个例子，我们想要知道第3个数到第4个数的和。我们发现第1个数到第4个数的和与这个区间只有一小段不同，而这一小段就是第1个数到第2个数的和。所以说，如果我们能预处理出第1个数到第i个数的和$init(i)$，从第l个数到第r个数的和就可以被表示成$init(r)-init(l)$。我们可以称此为“前缀和”。

```
一列数nums，长度为n
一列数init，长度为n

前缀和（）：
    init[1] 为 nums[1]
    i从2到n循环：
        init[i] 为 init[i-1] + nums[i]

询问（一个数l，一个数r）：
    返回 init[r] - init[l-1]
```

我们用$O(n)$的预处理达到了每次$O(1)$的询问，这很快。但是，注意到了吗？我们好像始终没有尝试改变序列，然后进行询问。是否有数据结构能让我们改变序列，然后处理改变后的各个询问呢？

尝试用前缀和来进行这种操作。如果我们给一个数加了1，那么——从它开始之后每个位置上的前缀和都要加1——这是$O(n)$的修改。我们可不可以加快这个操作呢？

当然可以！让我们以一种新颖的方式来建立“前缀和”的改进版本——“树状数组”：

```
12345678
*------- 第1行
**------ 第2行
--*----- 第3行
****---- 第4行
----*--- 第5行
----**-- 第6行
------*- 第7行
******** 第8行
```

最上方是数组的编号，而下面的几行中，每一行的`*`代表树状数组的该项会处理那个位置的和。比如说，树状数组的第6项会处理数组中第5和第6两个数的和。

嘶……好玄幻。很容易看出，树状数组中的每项维护的都是一段连续区间，而对于树状数组的第i项，其维护的区间的右端点就是i。但是——左端点是几呢？换句话说——区间的长度是几呢？

好吧，这可能需要一些奇妙的内容：规定i的二进制表示中最低一位为第0位，我们设i的二进制

表示中最低的为1的一位为第k位，那么区间的长度是2^k。

其实，我们就是把i的二进制表示从最后的1开始，做了一个截断，比如6，又或者说$110_{(2)}$，就变成了$10_{(2)}$，即2。

觉得我在胡言乱语？好吧，首先，你可以自己搜索。其次，接下来，我将把上面的“截断操作”包装成$lowbit$这个黑箱，即使你不懂它的原理也没有关系。现在，你只需要知道$lowbit(i)$就是树状数组的第i项所代表的区间长度就可以了，比如$lowbit(6)=2$。

利用这种数据结构，我们可以实现$O(\log n)$的修改——给一项添加一个值，以及$O(\log n)$的前缀和计算。如果能得到前缀和，我们自然能得到每个区间的值。

先说修改：如果在一个位置上添加一个值，我们自然需要在树状数组中每一个维护这个位置的项添加一个值。比如说，如果要在第3项添加1，我们需要在树状数组的第$3,4,8$项各自添加1。这个操作非常容易实现，你看：$4-3=1$，而$lowbit(3)$，也就是树状数组的那一项所维护的长度，也是1；同理，$8-4=4=low$。换言之，对于在第i项添加一个之，想象从树状数组的第i项开始往后寻找需要添加值的每一项，只需要把i添加$lowbit(i)$就可以了。

```
一列数tree，长度为n

添加（一个数i，一个数num）：
    如果 i 小于 n，重复：
        tree[i] 为 tree[i] + num
        i 为 i + lowbit (i)
```

可以想到，树状数组中顶多有$\log n+1$个项同时维护一个位置。这样，我们可以用$O(\log n)$的复杂度在一个位置上做加法。

那么求前缀和呢？注意到，对于在i结尾的前缀和，树状数组的第i项会维护其中的一些内容，而前面则可能空一块——只需要再从空的部分的末尾开始新一次计算，或者说，让i变为$i-lowbit(i)$并继续循环，就可以了。

```
前缀和（一个数i）：
    一个数res，一开始为0
    如果 i 大于 0，重复：
        res 为 res + tree[i]
        i 为 i - lowbit (i)
    返回 res
```

很显然，前缀和也是$O(\log n)$的。有了前缀和，任意区间的和也很好写了。

```
区间和（一个数l，一个数r）：
    返回 前缀和（r） - 前缀和（l - 1）
```

你可能会想：能不能做到复杂度低于$O(n)$的区间修改——把区间中的每个数都加上一个数呢？我们确实有这么做的方法。

想想看，一个数列$1, 1, 1, 1, 1$，它的前缀和数列自然是$1, 2, 3, 4, 5$。如果让数列变成$1, 2, 1, 1, 0$，也就是把第2项加1，把第5项减1，那么它的前缀和会变成——$1, 3, 4, 5, 5$，第2项到第4项都加了一。这好像也是一种区间修改。

既然这样，我们不妨让树状数组所维护的“前缀和”变成我们实际需要的数，那么每个“前缀和询问”就是一个实际上的“单点询问”。同样的，我们可以通过两个“单点修改”来达到一个“区间修改”。

```
一列数tree，长度为n

添加（一个数i，一个数num）：
    如果 i 小于 n，重复：
        tree[i] 为 tree[i] + num
        i 为 i + lowbit (i)

区间添加（一个数l，一个数r，一个数num）：
    添加（l，num）
    添加（r + 1，-num）

单点询问（一个数i）：
    一个数res，一开始为0
    如果 i 大于 0，重复：
        res 为 res + tree[i]
        i 为 i - lowbit (i)
    返回 res
```

这样，我们其实实现了区间修改和单点询问，这种用一加一减配合前缀和来达到区间修改的操作叫做“差分”——这正是“可差分”这种性质的意思，你可以用一加一减配合前缀和进行一个区间加。顺便，你可能希望既要区间修改，又要区间询问，但这有一点难度，在此略过。

第7章 区间再求和

> 在我的后园，可以看见墙外有两株树，一株是枣树，还有一株也是枣树。
>
> ——鲁迅《秋夜》

考虑另一种对数组区间求和的处理方式。

```
12345678
******** 第1行
****---- 第2行
----**** 第3行
**------ 第4行
--**---- 第5行
----**-- 第6行
------** 第7行
*------- 第8行
-*------ 第9行
--*----- 第10行
---*---- 第11行
----*--- 第12行
-----*-- 第13行
------*- 第14行
-------* 第15行
```

不错，第1行包含整个数组，而对于每一个第i行，如果它处理的区间还能再平分，那么第$2i$行就处理左边那一个，第$2i+1$行就去处理右边那一个。与树状数组的示意图一样，第几行就是我们的“神奇数组”的第几项。由于这种“神奇数组”操控一个区间——或者说一个线段——而且它一层层分割，像一棵树，我们便把它叫做“线段树”。

很显然，按照这种方式，只要有一个原数组就能方便地递归建立起线段树。使用线段树，我们可以快速做到各种区间操作，比如区间求和和区间修改。想一想，在上面的线段树例子中，我们可以怎样得到从第3个数到第6个数的和呢？显然，我们可以把这个区间从中间劈开，变成从第3个数到第4个数的和从第5个数到第6个数的和这两个区间，然后把它们的答案加起来就可以了。这启示我们，当我们想要进行区间求和时，我们也只需要进行递归就行了。

很不错的想法，而且这是$O(\log n)$的。但是，当我们想要用类似的方法进行区间修改时，就会出现问题——一般的区间修改肯定是$O(n)$的，因为这时，把区间分割成小区间并没有用，一个区间只有维护这里的和的功能，并没有维护它“所添加的值”的功能。稍等，或许它可以有？

我们尝试，在对一个由线段树的一项维护的区间中的数都添加某值时，并不在这个区间中的每个数上直接添加，而是在这一区间所对应的一个“标记”上添加这个值。当然，我们还需要将这里的和添加上这里的长度和所要添加的值。在这之后，如果我们要查询一个完全包含这个区间的区间的和，由于我们对这里的区间和做过处理，所以并不会有问题。而如果我们要求这个区间中的小区间的和，只需要把“标记”下放给小区间，然后清零自己的“标记”就可以了。这样，相当于把添加的操作放在了以后，所以我们称这种标记为“懒标记”。显然，懒标记被下放到“最下

方”的概率比较小，所以我们可以用这种方式减少很多运算。

说了这么多，我们来看一些伪代码吧！首先，我们要做一些准备工作：

```
一列数nums，长度为n
一列数tree，长度为4*n
一列数mark，长度为4*n
```

很显然，`nums`是原数组，`tree`和`mark`则分别是区间的和与标记，但为什么后两者的长度是`4*n`呢？其实，这就是在问一个有n个数的数列在按照上面的方式分割时会被分成多少个区间。人们用一些方式算出来，这个值不会超过$4n$，所以我们把数组长度设成$4*n$。

考虑到懒标记是我们的重要操作之一，我们可以把它有关的操作单独写成函数。而对于一个区间，在它所包含的区间被更新后，我们需要更新它，这个操作十分简单，但也值得单列。于是，我们有以下`赋标记`（其实用所添加的值同时更新了一下区间和）`下放标记`和`更新和`（是用小区间来更新）三个函数：

```
赋标记（一个数l，一个数r，一个数i，一个数num）：
    tree[i] 为 tree[i] + num * (r - l + 1)
    mark[i] 为 mark[i] + num

下放标记（一个数l，一个数r，一个数i）：
    一个数m，为 向下取整 ((l + r) / 2)
    赋标记（l，m，2 * i，mark[i]）
    赋标记（m + 1，r，2 * i + 1，mark[i]）
    mark[i]为0

更新和（一个数i）：
    tree[i] 为 tree[2 * i] + tree[2 * i + 1]
```

然后，让我们来用递归简单实现一下树的建立：

```
建树（一个数l，一个数r，一个数i）：
    如果 l = r：
        tree[i] 为 nums[l]
        返回

    一个数m，为 向下取整 ((l + r) / 2)
    建树（l，m，2 * i）
    建树（m，r，2 * i + 1）
    更新和（i）
```

我们可以用`建树（1，n，1）`的方式来调用这个函数，然后它会自己递归。

区间求和与区间修改的为代码也可以写出来，只是要注意以下`l`和`r`代表我们已经递归到的区间，而`allL`和`allR`代表我们需要处理的区间。这意味着我们仅当递归到的区间处于所需要处理的区间内时，才会进行直接的处理，否则就要进行递归：

```
求和（一个数allL，一个数allR，一个数l，一个数r，一个数i）：
    如果 allL 小于等于 l 且 r 小于等于 allR：
        返回tree[i]

    一个数m，为 向下取整 ((l + r) / 2)
    一个数res，为0

    下放标记 (l, r, i)
    如果 allL 小于等于 m
        res 为 res + 求和 (allL, allR, l, m, 2 * i)
    如果 m + 1 小于等于 allR
        res 为 res + 求和 (allL, allR, m + 1, r, 2 * i + 1)

    返回 res

修改（一个数allL，一个数allR，一个数l，一个数r，一个数i，一个数num）：
    如果 allL 小于等于 l 且 r 小于等于 allR：
        赋标记 (l, r, i, num)

    一个数m，为 向下取整 ((l + r) / 2)
    一个数res，为0

    下放标记 (l, r, i)
    如果 allL 小于等于 m
        修改 (allL, allR, l, m, 2 * i, num)
    如果 m + 1 小于等于 allR
        修改 (allL, allR, m + 1, r, 2 * i + 1, num)
    更新和 (i)
```

你看，我们为了做到$O(\log n)$的区间求和、区间修改，花了多大的力气啊！你说，如果我们降低对事件的要求，有没有其他的处理方式呢？来看这种划分方式：

```
123456789
***------ 第1行
---***--- 第2行
------*** 第3行
```

这种划分方式，好像是把整个数组划分成了众多大小相等的块，而每个块的长度是……$\sqrt{n}$？没

错，不过考虑到数组长度很有可能不是平方数，我们可能会留下最后的一个小块。这样，我们一共会有向上取整（$\frac{n}{\sqrt{n}}$）个块。不如我们把$sqrtn$设成B，而把块数设成L吧。顺便，我们记录一下每一块的和。我们可以写一个简单的代码，表示每个数所属于的块和每个块的左端点与右端点：

```
一列数nums，长度为n
一个数B，为 向下取整 (sqrt(n))
一个数L，为 向上取整 (n / B)
一列数belong，长度为n
一列数left，长度为L
一列数right，长度为L
一列数sum，长度为L

一个数nowBelong，为0
i从1到n循环：
    如果 (i - 1) % B = 0
        nowBelong 为 nowBelong + 1
        left[nowBelong] 为 i
    如果 i % B = 0 或 i = n：
        right[nowBelong] 为 i

    belong[i] 为 nowBelong
    sum[belong[i]] 为 sum[belong[i]] + nums[i]
```

在这之中，`%`表示取余，就是做除法但是把余数作为运算结果。我们显然可以用余数是否等于零来判断一个数是否是另一个数的倍数。

如果我们要进行区间修改，那么，我们可以把它分为覆盖了整块的中间部分和没有覆盖整块的零散数字。可以注意到，中间部分覆盖的整块的数量不会超过L，而零散数字的数量不会超过$2B$，所以，只需要分开处理它们，时间复杂度就不会超过$O(\sqrt{n})$。只不过需要注意两点：第一，我们可以对每一个块添加标记，当集体增加值时直接在其上添加数值，与懒标记优化那样类似，只是不下放；第二，如果处理的数字甚至没有包含一个整块，那我们可以直接处理它们，不用进行划分。而对于区间求和，也是这个思路，甚至更简单。

```
一列数mark，长度为L

区间求和（一个数l，一个数r）：
    如果 belong[l] 不等于 belong[r]：
        res为0
        i从l到right[belong[l]]循环：
            res 为 res + nums[i] + mark[belong[i]]
        i从left[belong[r]]到r循环
            res 为 res + nums[i] + mark[belong[i]]
        i从belong[l]+1到belong[r]-1循环
            res 为 sum[i] + mark[i]
        返回res
    否则：
        res为0
        i从l到r循环：
            res 为 res + nums[i] + mark[belong[i]]
        返回res

区间修改（一个数l，一个数r，一个数num）：
    如果 belong[l] 不等于 belong[r]：
        i从l到right[belong[l]]循环：
            nums[i] 为 nums[i] + num
        i从left[belong[r]]到r循环
            nums[i] 为 nums[i] + num
        i从belong[l]+1到belong[r]-1循环
            mark[i] 为 mark[i] + (right[i] - left[i] + 1) * num
    否则：
        i从l到r循环：
            nums[i] 为 nums[i] + num
```

你看，虽然代码量可能还是很长，但它的思维理解度好像确实没有这么复杂。事实上，确实有些人用这种数据结构解决各种问题，我们管这个处理方式叫“分块”。因为这种方式其实很暴力，所以我们戏称它为“优雅的暴力”。

第-1章 结语

何乐乌讲完，摆弄起了土里捡的几颗石子，“感觉怎么样？喜欢数据结构吗？”

“喜欢，”沃柔德看着石子，“或许就像玩石头一样快乐吧。”