让我们做得更好

——从《 parity 》的解法谈程序优化

福州第三中学高三（3）　　孙林春

Parity 题意描述

一个序列全部由 0 和 1 构成。你将知道其中某些连续的区间段中（例如，从第三位到第五位）含有的 1 的个数是奇数还是偶数，这些信息按照给出顺序编号。然而，这些信息有可能是自相矛盾的。你的任务是编程求出一个最大编号，使得存在一个序列，满足此编号及此编号之前的所有信息。

样例输入：

10{ 序列的长度 L， 1<=L<=1 000 000 000}

5 { 信息总数 N,1<=N<=5000}

1 2 even

{ 表示从第一位到第二位中含有偶数个 1}

3 4 odd

{ 表示从第三位到第四位中含有奇数个 1}

5 6 even

1 6 even

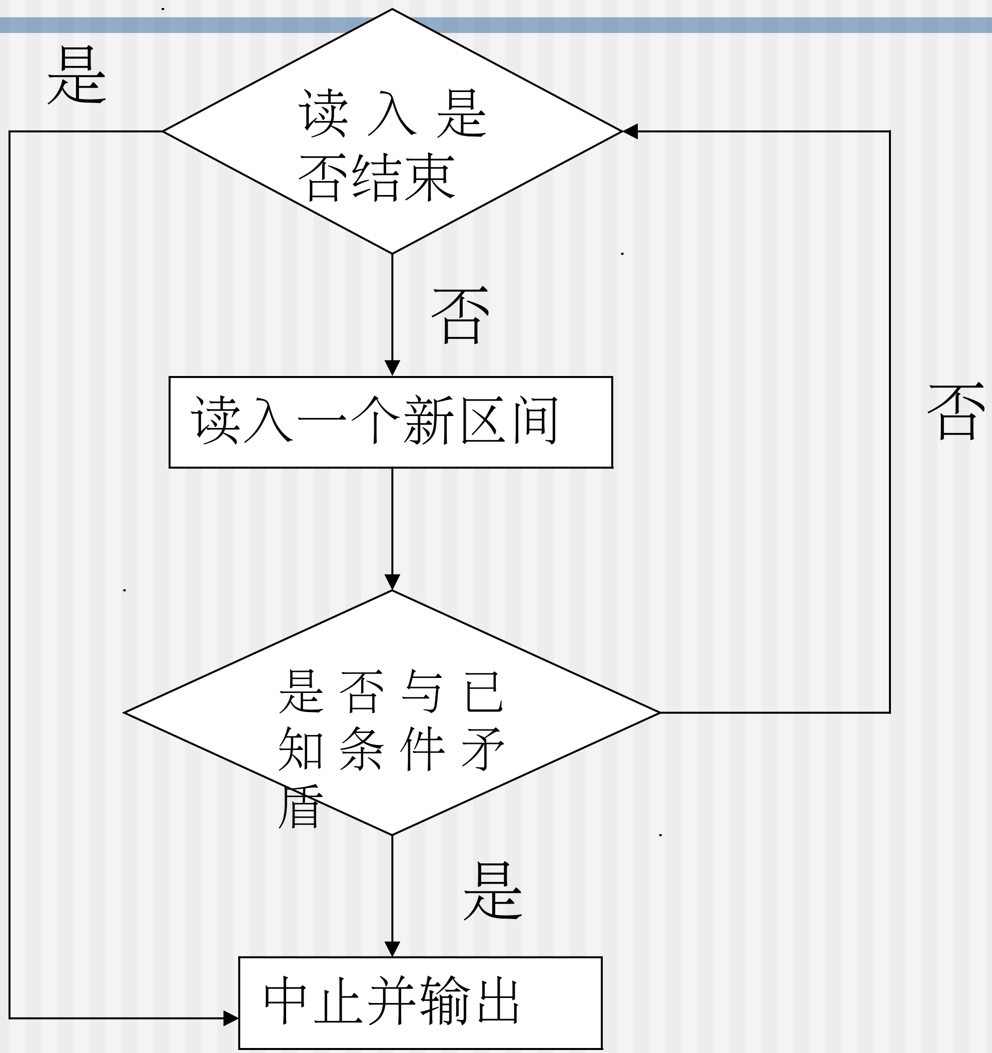
7 10 odd

样例输出：

3

{ 即可以找到一个序列，使之满足前三条信息，但找不到一个序列，使之满足前四条信息}

算法框架



原始的考虑——算法一

将当前区间的信息分别与每个已知的区间的信息进行比较，判断是否出现矛盾。

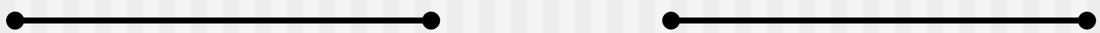
- 预备：两个区间之间的关系

具体做法是：将当前的区间与已知区间逐个进行比较：如果存在某个已知区间与之重合，则直接判断是否出现矛盾；否则，如果有左端点或右端点与其相同的区间，则对区间进行删截，同时修改区间信息，并将得到的新区间重新与已知区间比较，直至与所有已知区间的左右端点都不相同为止；最后将剩下的区间插入已知区间的队列中。

两个区间之间的关系

1　相离

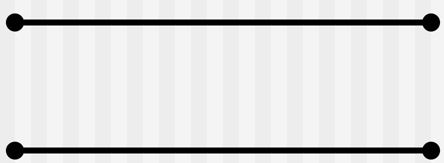
区间 1　　　　　　　　区间 2



2　重合

区间 1

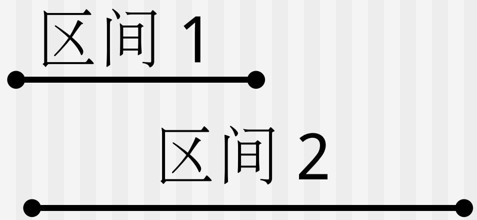
区间 2



两个区间之间的关系

3　相交

4　包含



深入分析——算法二

改进点：保留部分重复的区间及信息，而把注意力集中到其中一些能够直接导出矛盾信息的区间上来。

做法：当读入一个新的区间并进行判断时：若已知区间队列中有与其具有共同的左端点的区间，我们只需留下它们之中右端点小的一个，较长区间长出的部分则可以看成是一个新的区间，并重新与其他已知区间进行比较；若没有一个已知区间与当前区间有相同的左端点，则将当前区间作为一个新的左端点的代表区间插入队列中。

局部优化——算法三

离散化端点值，提高查找效率

最简单的办法：开一个数组，将左端点的值作为数组的下标，数组中的值表示该左端点的代表区间的右端点的值，若这样的区间不存在，则值记为 0 。

改进：将原有的端点值离散化后对端点重新编号。我们将所有出现过的端点值放入另一个数组中，并对该数组进行快速排序，然后把用二分法在该数组中查找一个端点值所得到下标作为该端点的新编号。

挖掘本质——算法四

- 区间奇偶性描述法：以某个区间段中所含的 1 的个数的奇偶性来描述 01 序列
-

$$P[i,j]=\begin{cases} true\text{（从第i位到第j位有奇数个1）} \\ false\text{（从第i位到第j位有偶数个1）} \end{cases}$$

前缀奇偶性描述法：以前 i 位中所含 1 的个数的奇偶性来描述 01 序列

$$Parity[i]=\begin{cases} true\text{（从第一位到第i位有奇数个1）} \\ false\text{（从第一位到第i位有偶数个1）} \end{cases}$$

两种描述法的对应关系

若 p[i， j]=true，

则 parity[i-1] xor parity[j]=true；

若 p[i， j]=false，

则 parity[i-1] xor parity[j]=false；

parity 数组的所有下标构成了集合

A ＝ {1， 2， …， L} 。

这个集合根据元素 i 所对应的 parity[i] 的值是 True 还是 False 被划分成了两个等价类 A_1 和 A_2，

所有 parity[i]=True 的 i 归入 A_1 中，

所有 parity[i] ＝ False 的 i 则归入 A_2 中。

根据对应关系， **p[i， j]** 的值是 True 还是 False 决定了 parity[i-1] 的值与 parity[j] 的值是否相同；实际上也就决定了 **i － 1** 和 **j** 是否属于同一个等价类。

这样，原来对每个区间 [i， j] 进行约束的条件就转化成了元素 i-1， j 是否属于同一个等价类的判断条件。

具体做法是

定义

集合 same[i] 表示已知与 i 在同一个等价类中的元素集合

集合 opp[i] 表示已知与 i 不在同一个等价类中的元素集合

根据已知条件，若有：

parity[j] ＝ parity[i]，则 j∈same[i]；

parity[j]≠parity[i]，则 j∈opp[i]；

初始时， same[i]={i}， opp[i]=Φ 。

依次处理每条区间信息：

```
if 与已知条件不矛盾 then
   if p[i,j]=true then
     begin
        合并(same[i],same[j])；    合并(opp[i],opp[j])；
     end
   else
     begin
          合并(same[i],opp[j])；     合并(same[j],opp[i])；
     end
   else 中止判断；
```

具体实现时，我们可以应用并查集来完成所需的操作。理论上已经证明，如果利用按秩合并与路径压缩等技巧对程序进行充分优化，并查集这种数据结构的时间复杂度是 **O**（ N*α(N)) 的。其中， α(N) 是单变量阿克曼函数的逆，它是一个增长速度比 logN 慢的多但又不是常数的函数。

同时，我们已经将算法主体部分的时间复杂度降为 **O**（ N*α(N)) 的，查找部分再用 O （ logN ）的二分查找就显得不合适了，因此我们考虑用更加高效的哈希表来实现查找。哈希表的查找时间是 O （ 1 ）的，因此，算法中的查找时间是 O （ N ）的。

综合两部分，整个程序的时间复杂度为（ N*α(N)) 。

运行时间比较

测试数据	算法一的实现	算法二的实现	算法三的实现	算法四的实现
N＝500	0．00s	0．00s	0．00s	0．00s
N＝5000	3．41s	3．41s	0．22s	0．22s
N＝2003	1．87s	0．77s	0．38s	0．05s
N＝4505	35．93s	0．71s	0．27s	0．22s
N＝5000	10．77s	7．36s	7．14s	0．22s

总 结

通过解决这道题，我们看到了充分优化算法的重要作用，并从中总结出优化算法的一些一般规律：

1 从问题出发，深入挖掘题目本质；

- 2 针对当前算法的不足加以改进。

但归根到底，这些优化都是建立在对问题本身及数据结构深刻理解的基础上的。只有充分认识问题，理解问题，并熟练掌握各种数据结构，才能针对问题设计出高效而实用的算法并加以实现。

让我们做得更好！

谢谢！