THUWC/PKUWC 2019 Simulation Day 1

Nanjing Foreign Language School

题目名称	排列	子序列	Lambda
可执行文件名	per	sub	N/A
输入文件名	标准输入	标准输入	lambda.in
输出文件名	标准输出	标准输出	lambda.out
测试点时限	3s	1s	N/A
内存限制	512MB	256MB	N/A
是否有部分分	否	否	否
题目类型	传统型	传统型	提交答案型

1 排列

1.1 题目描述

垫底哥学了两年 OI 最近刚刚学会了 xor 和排列。

他现在有 n 个数 $a_1, a_2, a_3, ..., a_n$。他想找到一个长度为 n 的排列 $p_1, p_2, p_3, ..., p_n$，使得 $\max_{i=1}^{n-1}(a_{p_i} \operatorname{xor} a_{p_{i+1}})$ 最小。如果有多组，输出 p 字典序最小的一组。

然而垫底哥除了垫底其它什么都不会，他就找你解决这个问题。

1.2 输入格式

第一行一个正整数 n。

第二行 n 个整数，分别表示 $a_1, a_2, a_3, ..., a_n$。

1.3 输出格式

一行 n 个整数，$p_1, p_2, p_3, ..., p_n$。

1.4 样例输入

8

1 9 2 6 0 8 1 7

1.5 样例输出

1 2 6 5 3 4 7 8

1.6 数据规模与约定

对于所有数据，满足 $2 \leq n \leq 3 \times 10^5, 0 \leq a_i \leq 10^9$。

- Subtask 1（20%）：$n \leq 10$；
- Subtask 2（30%）：$n \leq 1000$；
- Subtask 3（50%）：无特殊限制。

2 子序列

2.1 题目描述

垫底哥学了两年 OI 最近刚刚学会了 xor 和子序列。

他得到了一个长度为 n 的数组 $a_1, a_2, a_3, ..., a_n$，他想知道 xor 值为 0 的子序列最长是多少。

然而垫底哥除了垫底其它什么都不会，他就找你解决这个问题。

2.2 输入格式

第一行一个正整数 n。

第二行 n 个整数，分别表示 $a_1, a_2, a_3, ..., a_n$。

2.3 输出格式

一行 1 个整数，表示答案。

2.4 样例输入

8

1 9 2 6 0 8 1 7

2.5 样例输出

7

2.6 数据规模与约定

对于所有数据，满足 $1 \leq n \leq 5 \times 10^5, 0 \leq a_i \leq 5 \times 10^5$。

记 $A = \max\{a_1, a_2, \cdots, a_n\}$：

- Subtask 1（10%）：$n \leq 20$；
- Subtask 2（20%）：$n \leq 100$，$A \leq 100$；
- Subtask 3（30%）：$n \leq 3000$，$A \leq 3000$；
- Subtask 4（40%）：无特殊限制。

3 Lambda

3.1 题目背景

1930 年代，人们在对这样一个问题感到好奇：我们到底可以计算什么？两个人回答了这个问题。阿兰 · 图灵构想出在一个写满 0 和 1 的纸带上运作的图灵机。而阿隆佐 · 邱奇发明了 λ-演算（λ 读作 lambda）。这两个模型都可以表达我们计算的方式。

两个模型截然不同，那么它们是否互有优劣呢？有没有一个模型能表示的计算，而另一个模型不能完成呢？这个问题被称为**邱奇 - 图灵论题**。通常可以认为这个论题为真，这两个模型是等价的：任何一个 λ-演算表达式都可以表示成一台图灵机上的程序，反之亦然。

这道题需要你用 λ-演算的表达式表达一系列计算。

3.2 语法

我们在这里简称 λ-演算的表达式为表达式。表达式有如下 3 种形式：

1. a：表示一个变量 a 的值。

2. $\lambda x.N$：其中 N 是一个表达式。这个形式表示一个函数，这个函数有一个参数叫 x，函数的返回值是将 N 中的 x 替换为调用这个函数的实际参数的结果。如果 $f = \lambda x.N$，可以用伪代码表示如下：

```
function f(x) {
    return (将 N 中的 'x' 替换为 x 的实际值);
}
```

3. $F\ A$：其中 F 和 A 都是表达式。这个形式表示函数调用，即将 A 代入 F 的参数，得到返回值。例如如果 F 为 f，则用伪代码可以表示成：

```
f(A)
```

我们约定：

- 调用（即形式 3）是左结合的，即 $x\ y\ z$ 表示 $(x\ y)\ z$ 而不是 $x\ (y\ z)$。
- 当用 λ 表示一个函数开始时（即形式 2），函数体的右端会延申仅可能远，如 $\lambda x.x\ x\ x$ 表示 $\lambda x.(x\ x\ x)$ 而不是 $(\lambda x.x)\ x\ x$ 或 $(\lambda x.(x\ x))\ x$

注意到 λ-演算中没有命令的说法，只有表达式和表达式的化简。这和递等式计算差不多。

这里有几个例子：

- $I = \lambda x.x$ 表示一个接受一个参数 x 的函数，返回 x 本身。- 对于任意表达式 a，$I\ a = a$，因为将 a 带入 I 的参数 x，可以将函数体 x 替换为 a。

- $true = \lambda x.\lambda y.x$ 表示一个接受一个参数 x 的函数，它返回一个接受参数 y 的函数，第二个函数的返回值为 x。至于为什么叫 $true$，可以参见测试点描述。用伪代码也可以这样表示：

```
function true (x) {
        // 返回的是一个函数！
        // 这个返回的函数无论传什么参数进去都会返回传给 true 的 x。
        return function (y) {
                return x;
        };
}
```

  换而言之，对于表达式 a 和 b，

$$
\begin{aligned}
& (true\ a)\ b \\
=& \Big(\big(\lambda x.(\lambda y.x) \big)\ a \Big)\ b && \text{展开 } true \\
=& (\lambda y.a)\ b && \text{将 } a \text{ 代入} \\
=& a && \text{将 } b \text{ 代入}
\end{aligned}
$$

  或者把上面的计算过程表示为伪代码：

```
func = true(a);
func(b) // 永远为 a
```

  你可以感性地认为这是一个接受双参数的函数，不过参数必须要一个一个地传进去。

- $one = \lambda f.(\lambda x.(f\ x))$ 表示一个接受参数 f 的函数，它返回一个接收参数 x 的函数，第二个函数的返回值是将 x 当作 f 的参数进行调用的结果。至于为什么叫 one，可以参见测试点描述。换而言之，对于函数 f 和表达式 a，

$$
\begin{aligned}
&(one\ f)\ a \\
=&\Big(\lambda f.\big(\lambda x.(f\ x)\big)\ f\Big)\ a \\
=&\big(\lambda x.(f\ x)\big)\ a \\
=&f\ a
\end{aligned}
$$

这可以表示我们通常写作 $f(a)$ 的结果。这说明 one 可以认为是一个接受两个参数，返回值是用第二个参数调用第一个参数的结果。

- $\Omega = \big(\lambda x.(xx)\big)\big(\lambda x.(xx)\big)$，虽然看起来它的最外层是一个调用，可以将后面一半代入前面一半，但是你会发现这次调用之后的结果仍然是 Ω 本身。

我们再来看一个化简的例子：
设 $S = \lambda f.(\lambda g.(\lambda x.(f\ x)\ (g\ x)))$，$K = \lambda x.(\lambda y.x)$，则有

$$
\begin{aligned}
&(S\ K)\ K && && (1)\\
=&\Big(\lambda g.\big(\lambda x.(K\ x)\ (g\ x)\big)\Big)\ K && && (2)\\
=&\lambda x.(K\ x)\ (K\ x) && \text{两个 } K \text{ 分别填入了 } f \text{ 和 } g && (3)\\
=&\lambda x.(\lambda y.x)\ (\lambda y.x) && \text{此处将第二个 } \lambda y.x \text{ 代入第一个中的 } y && (4)\\
=&\lambda x.x && && (5)
\end{aligned}
$$

另外因为涉及到名字的问题，我们认为 $\lambda x.x$ 和 $\lambda y.y$、$\lambda x.f\ (x\ x)$ 和 $\lambda y.f\ (y\ y)$ 等只是参数名字不同的函数是等价的。

如果外层函数和内层函数参数名字相同，内层函数内部的这个名字总是表示内层函数的参数，如 $\lambda x.\lambda x.x = \lambda x.\lambda y.y$，而 $\lambda x.x\ (\lambda x.x) = \lambda x.x\ (\lambda y.y)$。

以上是 λ-演算的语法的简单描述。

3.3 答题格式

你不容易在键盘上打出 λ，所以我们用反斜杠（\）代替 λ；“.”仍然是“.”；每个函数的参数名称，以及表达式内部用到的变量名都应该是由一个或多个小写字母构成的字符串。

每个测试点会要求你定义一些表达式，如测试点 1 要求你定义 $true$（如上，$true = \lambda x.\lambda y.x$），你应该在测试点 1 的提交文件 lambda1.txt 中写下这样一行（这里的空格不是必须的，没有空格也能知道表达式的意思）：

true = \x . \y . x

另外，你还需要定义 $false$，所以你应该另起一行写：

false = < 这里是 false 的定义 >

两行的顺序是无所谓的。

注意一个表达式里是可以出现没有在表达式内部作为某一层函数的参数的名字的，如 $\lambda x.y$ 对于任何参数 x 都返回 y，但如果没有定义 y，我们并不知道这个返回值究竟是什么。在本题的答题过程中，这样的用法是被限制的。具体来说，**你可以引用你在之前的行中定义的表达式**，例如如果你先定义 $true$ 再定义 $false$，$false$ 的定义中可以引用 $true$；但 $false$ 的定义不能引用 $false$ 本身或者之后定义的东西。

你可以定义额外的辅助表达式，你可以随便给这些表达式起名字。

你不可以定义一个表达式两遍。

因为测试点之间可能会有依赖关系，对于测试点 i，你实际的答案是 lambda1.txt、lambda2.txt、……、lambda<i>.txt 连接在一起的结果。也就是说，如果你在某一个测试点的答案中定义了 $true$，那么之后的测试点都可以使用 $true$；反过来说，如果你某一个测试点的格式有问题，后面的测试点也会连坐。

当然，如果你不会定义 $false$，你可以选择不写 false = <false 的定义 >，或者将它定义成一个合法但不满足测试点要求的表达式

这样虽然第一个点得不到分数，但是只要后面的某个测试点没有明确需要引用 $false$，没有定义 $false$ 这件事不会影响该测试点的得分。也就是说，不要写格式不合法的定义，后果严重。

下发文件中有一个 evaluator，可以帮助你在本地化简表达式，用法类似提交格式，另外可以用?? < 表达式 > 输出表达式化简的结果，细节将在题目末尾给出。在里面输入 help 也可以获得帮助。

3.4 测试点

第一部分：布尔逻辑

3.4.1 测试点 1

定义 $true$ 和 $false$，使得对于任意 x 和 y，$(true\ x)\ y = x$，$(false\ x)\ y = y$。

分值：10

3.4.2 测试点 2

定义 not、and 和 or，使得它们满足一般的逻辑运算规律。注意这依赖于测试点 1。

具体来说，设 $a = (and\ true)\ true$，则要求 $(a\ x)\ y = x$。其他类似。

提示：可以引用测试点 *1* 中的 $true$ 和 $false$。

分值：10

第二部分：用函数表示数据

3.4.3 测试点 3

定义 $pair$、$first$ 和 $second$，使得对于任意 x 和 y，$first\ (pair\ x\ y) = x$，$second\ (pair\ x\ y) = y$。你的 $first$ 和 $second$ 不需要对于其他参数负责。

提示：$true$ 和 $false$ 具有选择功能；虽然 $pair$ 是双参数的，在接受两个参数后其实也可以接受更多参数。

分值：10

3.4.4 测试点 4

定义列表！具体来说，需要定义：

- nil 表示空列表。它可以不接受参数。
- $insert$ 表示在列表首部插入，具体来说，对于列表 l 和任意 x，$insert\ x\ l$ 表示在 l 首部插入 x 的结果。（评测机会从 nil 开始，不断对一个列表进行 $insert$，从而得到一个非空列表。例如 $insert\ a\ (insert\ b\ nil)$ 表示列表 $[a, b]$。）
- $head$ 表示获得列表首部元素。具体来说，对于非空列表 $l = insert\ x\ l'$，$head\ l = x$。
- $tail$ 表示获得除首部元素以外其他元素构成的列表。具体来说，对于非空列表 $l = insert\ x\ l'$，$tail\ l = l'$。
- $isnil$ 表示判断列表是否为空。具体来说，$isnil\ nil = true$，$isnil\ (insert\ x\ l) = false$，其中 l 是一个列表。

你不需要支持其他操作（如按下标寻找元素）。

提示：参考测试点 *3*。

分值：5

第三部分：高阶函数

3.4.5 测试点 5

高阶函数是参数或返回值是函数的函数。
这里，你需要定义：

- *compose* 表示接合两个函数。设 $h = compose\ f\ g$，则 $h\ x = f\ (g\ x)$，即 *compose* 将两个函数合成一个。注意第二个参数（g）先作用于 x。
- *map* 表示将一个函数应用到一个数据结构上的所有元素中。这里我们只要定义支持 *pair* 的简单版本。具体来说，只要求 $(map\ f)\ (pair\ x\ y) = pair\ (f\ x)\ (f\ y)$。

分值：10

第四部分：自然数

3.4.6 测试点 6

我们跟随邱奇的脚步，用 λ-演算表达式来定义自然数：

$0 = \lambda f.\lambda x.x$

$k = \lambda f.\lambda x.(f\ \cdots(f\ (f\ (f\ x)))\cdots)$　　（k 是正整数，共有 k 个 f）

或者用一般的数学语言：

$0(f)(x) = x$

$k(f)(x) = f(f(f(\cdots f(x)\cdots)))$　　（k 是正整数，共有 k 个 f）

这里你需要定义：

- *zero*，表示 0。
- *three*，表示 3。
- *succ*，表示获得下一个自然数。具体来说，如果 k 表示自然数 K，则 $succ\ k$ 可以表示 $K + 1$。

注意你不能在你的提交中使用数字。当然，你可以手动定义 *one*、*two* 等等，但它们不计入分数。

之后的测试点均依赖 *zero* 和 *succ*。

提示：你可以在一些函数里预先保留 f 和 x 的位置，毕竟必须代入 f 和 x 才能知道这个数字具体的值。

分值：10

3.4.7 测试点 7

定义 *plus*、*mul*、*exp*，使得如果 a 表示自然数 A，b 表示自然数 B，

- *plus a b* 表示 $A+B$；
- *mul a b* 表示 $A\times B$；
- *exp a b* 表示 A^B。

分值：10

3.4.8 测试点 8

定义 *pred*，表示获得前一个自然数。具体来说，

- $pred\ zero = zero$；
- 如果 a 表示正整数 A，*pred a* 表示 $A-1$。

注意第一条。

提示：妥善利用 *true*、*false*、*pair* 等。

分值：10

3.4.9 测试点 9

定义 *minus*，表示减法。具体来说，如果 a 表示自然数 A，b 表示自然数 B，$A\geq B$，则 *minus a b* 表示 $A-B$。

分值：5

3.4.10 测试点 10

定义 $iszero$、leq 和 eq，其中：

- $iszero\ zero = true$；对于表示正整数 A 的 a，$iszero\ a = false$。
- 如果 a 表示自然数 A，b 表示自然数 B，如果 $A \leq B$，$leq\ a\ b = true$；否则 $leq\ a\ b = false$。
- 如果 a 表示自然数 A，b 表示自然数 B，如果 $A = B$，$eq\ a\ b = true$；否则 $eq\ a\ b = false$。

分值：10

第五部分：递归

3.4.11 测试点 11

显然，λ-演算是不能直接递归的：一个表达式无法包含自己。
考虑如下表达式：

$$y = \lambda f.\big(\lambda x.f\ (x\ x)\big)\ \big(\lambda x.f\ (x\ x)\big)$$

y 被称为 Y 组合子（Y Combinator），或者不动点组合子，因为对于任意函数 f，均有：

$$\begin{aligned}
&y\ f \\
=&\big(\lambda x.f\ (x\ x)\big)\ \big(\lambda x.f\ (x\ x)\big) && \text{展开 } y \\
=&f\ \Big(\big(\lambda x.f\ (x\ x)\big)\ \big(\lambda x.f\ (x\ x)\big)\Big) && \text{进行最外层调用} \\
=&f\ (y\ f)
\end{aligned}$$

即 $y\ f$ 为任何函数 f 的不动点。

考虑（非形式化的）递归函数 $f = \ldots f \ldots$（即其中引用了 f 本身），构造函数 $f' = \lambda d.(\ldots d \ldots)$，则如果对于某个 d，$d = f'\ d$，就相当于 $f'\ d$ 引用了 $f'\ d$ 本身。这可以由 y 实现，即 $y\ f'$ 就是我们想要的 f。

定义 $divide$，使得对于表示自然数 A 的 a 和表示正整数 B 的 b，$divide\ a\ b$ 表示 a 整除 b 的结果，或者 $\lfloor \frac{a}{b} \rfloor$。

提示：使用 $minus$ 和 leq。

分值：10

3.5 evaluator 的使用

如果你不是 64 位 Windows，可以自行编译 evaluator.cpp，需要 C++11。这是一个交互式的程序：

- 你可以通过 f = < 表达式 > 设置名字 f 对应的表达式。注意在这之前 f 不能已经定义。
- 你可以通过 del f 删去名字 f。这样你就可以重新定义 f。
- 你可以通过?? < 表达式 > 打印表达式化简的具体步骤；? 表达式则直接打印最终结果。在步骤较多的时候，建议直接打印最终结果。
- 如果表达式最外层是个函数，evaluator 不会继续化简内部。但是评测时保证所有答案都会化到最简。
- 你可以通过 save < 文件名 > 将当前的所有定义存入文件。
- 你可以通过 read < 文件名 > 读入文件中的所有定义；但是**已定义的名字不会修改**。
- 你可以通过 help 打印帮助信息。
- 你可以通过 exit 退出。