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

最大值一定是最高位为 1 和 0 的两个数异或得到。把所有数字分成两个集合 S_0, S_1，其中 S_0 是最高位为 0 的数的集合，S_1 是最高位为 1 的数的集合。

可以找到异或最大值最小是多少，然后每次把异或和不超过答案的点连边。每次从小往大试，判断是否合法。

可以发现并不需要试所有的点，只有三种点是有用的：

- 跳到另一个集合
- 同一个集合的最小点
- 同一个集合的次小点

用数据结构维护即可。

2 子序列

2.1 算法一

$\Theta(2^n)$ 枚举每个数选不选。

2.2 算法二

$dp[i][x]$ 表示前 i 个数异或和为 x 时候最多选多少个，复杂度 $\Theta(nA^2)$。

2.3 算法三

令 $X = \oplus_{i=1}^{n} a_i$，变为删去最少的数使得异或和为 X。

考虑最短路，u 向 v 连边当且仅当存在 a_i 使得 $u \oplus a_i = v$。以 0 为起点 BFS 即可。

时间复杂度：$\Theta(A^2)$。

2.4 算法四

设 $f[i][x]$ 表示选 i 个数，异或和能否是 x。可以发现有用的 i 是 $\Theta(\log_2 n)$ 级别的，转移可以 FWT 优化。

时间复杂度：$\Theta(A \log_2^2 A)$，也可以做到 $\Theta(A \log_2 A)$。

3 Lambda

首先，$\lambda x.N$ 的伪代码我感觉写得还不够清楚，这样说可能更准确且容易理解：如果 $f = \lambda x.N$，可以用伪代码表示如下：

```
function f(x) {
    return 将 x = (x 的实际值) 代入 N 中的结果;
}
```

下面我们来按测试点顺序解题。

3.1 测试点 1

$true$ 的定义已经给出，

```
true = \x . \y . x
```

只要读完题（的主要部分）就知道

```
false = \x . \y . y
```

观察题面可以注意到这样一件事：

$$f\ x = N \Longleftrightarrow f = \lambda x.N$$

所以 $false$ 的推断是显然的：

$$\begin{aligned} & (false\ x)\ y &=& \ y \\ \Longleftrightarrow \quad & false\ x &=& \ \lambda y.y \\ \Longleftrightarrow \quad & false &=& \ \lambda x.(\lambda y.y) = \lambda x.\lambda y.y \end{aligned}$$

下面会不加说明地使用这样的结论。

3.2 测试点 2

我们会希望有一个类似 a ? b : c 的运算符，或者说 if。这样就可以：

```
not x = x ? false : true
and x y = x ? (y ? true : false) : false
or x y = x ? true : (y ? true : false)
```

仔细观察 $true$ 和 $false$ 的定义，会发现它们本身就能达到这样的要求，即如果 a 是 $true$ 或 $false$，则 a ? b : c 可以表示成 $(a\ b)\ c$，所以

```
not x = x false true
and x y = x (y true false) false
or x y = x true (y true false)
```

当然，$y\ true\ false$ 就是 y ? true : false，和 y 是等价的，所以一个简洁的答案是：

```
not = \x . x false true
and = \x . \y . x y false
or = \x . \y . x true y
```

3.3 测试点 3

提示提到 $true$ 和 $false$ 有选择功能，所以能够取出 $pair\ x\ y$ 中的 x 和 y，将它们当作 $true$ 或 $false$ 的参数，就可以实现 $first$ 和 $second$：

```
pair x y = ?
first (pair x y) = true x y
second (pair x y) = false x y
```

可惜 λ-演算不内置模式匹配，没法这么做。我们在 $first$ 和 $second$ 的函数体里，只能将参数 $pair\ x\ y$ 当作一个整体，而无法直接拆开 x 和 y。但是提示中又说：可以让 $pair$ 接受更多参数。那我们只好把 $true$ 和 $false$ 送进 $pair\ x\ y$，试图让它把这两个选择器作用于 x 和 y：

```
pair x y = ?
first p = p true
second p = p false
```

那么 $pair$ 就必须接受第三个参数：

```
pair x y = \f . ?
```

考虑? 是什么。$(pair\ x\ y)\ true = true\ x\ y$，$(pair\ x\ y)\ false = false\ x\ y$，所以可以让

```
pair x y = \f . f x y
```

即

```
pair = \x . \y . \f . f x y
first = \p . p true
second = \p . p false
```

这可能有点绕，我们回想一下 $first\ (pair\ a\ b)$ 是如何计算的：

$$
\begin{aligned}
&first\ (pair\ a\ b)\\
=&(pair\ a\ b)\ true\\
=&(\lambda x.\lambda y.\lambda f.f\ x\ y)\ a\ b\ true\\
=&(\lambda f.f\ a\ b)\ true\\
=&true\ a\ b\\
=&a
\end{aligned}
$$

注意到 $pair$ 在作用在 a 和 b 上后，得到的 $(\lambda f.f\ a\ b)$ 捕获了 a 和 b，从而我们可以再传入 f，对 a 和 b 进行操作。在这样的基础上，我们不仅可以传入 $true$ 和 $false$，甚至可以传入任意的双参数函数，使得两个参数是 $pair$ 的两个元素。通过这样的方式，我们也能构造任意数量成员的简单元组：

```
triple = \x . \y . \z . \f . f x y z
first = \t . t (\x . \y . \z . x)
second = \t . t (\x . \y . \z . y)
third = \t . t (\x . \y . \z . z)
```

这样，我们就用函数表示了数据。我们在这里简单谈一点对“数据”这个概念的理解。

$pair\ x\ y$ 这个数据本身是一个函数，似乎并不是很符合直觉。不过数据本身不一定要是内存中的二进制字节，一个表达式也可以将数据编码，只要我们能通过一定方法构造这种数据（$pair$），然后通过一定方法读取数据中我们需要的部分（$first$ 和 $second$），并使得这种数据满足某些约定。

在这里，$pair$ 被称为**构造函数**，而 $first$ 和 $second$ 被称为**选择器**，我们可以认为数据就是由构造函数、选择器和一些数据上的约定组成的。事实上，例如在 C++ 中，make_pair 可视为 pair 的构造函数（不是 C++ 意义上的构造函数），而.first 和.second‘就作为选择器而存在（虽然语法上不是一个函数，但通过这种方法我们得以选择数据中的两个部分）。这个例子中，$pair$ 上并没有很明显的限制条件。通过构造函数和选择器，我们可以在数据上进行一系列操作。

这样来观察数据，我们也得以进行一层抽象，将数据的使用和数据的内部表示隔离，不同的表示可以进行相同的使用，例如如果我们这样完成这个测试点：

```
pair = \x . \y . \f . f y x
first = \p . p false
second = \p . p true
```

我们在之后的测试点使用这三个函数的时候，并不会和之前的实现方式有什么不同。类似这样的剥离接口和实现的抽象在程序设计中是非常重要的。

3.4 测试点 4

显然我们的列表会形如

$$insert\ a_0\ (insert\ a_1\ (insert\ a_2\ \cdots (insert\ a_{n-1}\ nil))),$$

即满足

$$l \equiv insert\ (head\ l)\ (tail\ l).$$

这和 *pair* 的形式相当接近，所以我们会想：

```
insert = pair
head = first
tail = second
```

这样，列表成为了一棵用 *pair* 组合而成的向右的单边树，或者说，$first$ 是值、*second* 是指针的链表。每个节点的值是列表中该位置的值（*head*），指针指向下一个节点，即后面的元素组成的子列表（*tail*）。

的确，在不考虑 *isnil* 的情况下一切都能顺利工作。如果需要支持 *isnil*，我们可能需要在节点上记录更多的值（参考上面的 *triple*）：

```
makenode value isnil next = \f . f value isnil next
getvalue node = node (\x . \y . \z . x)
getisnil node = node (\x . \y . \z . y)
getnext node = node (\x . \y . \z . z)
```

然后将节点连在一起：

```
insert value list = makenode value false list
// 列表本身就是一个节点

head list = getvalue list
tail list = getnext list

// 只有 makenode 的第二个参数 isnil 有实际意义，其他两个 false 是随便填的
nil = makenode false true false

isnil list = getisnil list
```

整理一下，可以得到

```
makenode = \value . \isnil . \next . (\f . f value isnil next)
getvalue = \node . node (\x . \y . \z . x)
getisnil = \node . node (\x . \y . \z . y)
getnext = \node . node (\x . \y . \z . z)

insert = \x . \l . makenode x false l
head = \l . getvalue l
tail = \l . getnext l
nil = makenode false true false
isnil = \l . getisnil l
```

当然，可以不必把前四个函数显式地写出来；一个节点只要能包含值、是否是 nil、下一个节点这样三个信息，具体实现也是无所谓的，如

```
nil = pair true true
isnil = \p . first p
head = \l . first (second l)
tail = \l . second (second l)
insert = \x . \l . pair false (pair x l)
```

还有一些不用每个节点存三个信息的表示方法，但相比这样的做法可能稍微有点难理解，故在此不再赘述。

注意到这里没有要求对链表做加入首节点之外的操作，如将列表倒序，这是因为这样的处理整个列表的操作需要递归，而递归在 λ-演算中没有普通的编

程语言中那么直白。在第十个测试点介绍 Y 组合子后，我们将可以实现全列表的操作。

现在，我们可以实现一些局部的操作，如给列表换头：

```
changehead list newhead = insert newhead (tail list)
```

当然，由于一个已经构造出的表示列表的表达式不可能真的改变 *head*，所以这里事实上构造了一个新的列表，其 *tail* 和原列表相同，而 *head* 是新的值。这种不可以真的改变，而只能通过构造新的实例来得到改变后的结果的数据被称为“不可变数据”。不可变数据在 OI 中与可持久化是密切相关的（如函数式线段树）。

3.5 测试点 5

对于 *compose*，

```
(compose f g) x = f (g x)
compose f g = \x . f (g x)
compose = \f . \g . \x . f (g x)
```

这是显然的。通常在数学中 *compose* f g 用 $f \circ g$ 表示。

对于 *pair* 的 *map* 也是显然的：

```
(map f) (pair x y) = pair (f x) (f y)
map f = \p . pair (f (first p)) (f (second p))
map = \f . \p . pair (f (first p)) (f (second p))
```

通常情况下 *map* 是在列表上做的，但因为需要递归，所以在这里只要求实现一个简化版本。如果允许（直接）递归，列表上的 *map* 可以这么实现：

```
map = \f . \list . insert (f (head list)) (map f (tail list))
```

从而用比较简洁的语法（不需要显式地循环）将函数 f 作用于列表 *list* 的所有元素。

高阶函数在函数式程序设计中是一个相当重要的概念。（在一个复杂的程序中）通过将函数作为参数传递或当作返回值返回，可以避免重复写相似的代码，将共同的部分抽象出来。C++11 允许将函数作为“第一类公民”，标准库中提供了若干常用的高阶函数，如 std::transform 和上面的 *map* 功能类似。

（之所以这里只有两个简单的例子是因为列表操作需要递归以及高阶函数不容易测试正确性）

3.6 测试点 6

一个数字的表示是比较简单的（$zero$ 和 $false$ 是一样的，这很有意思）：

```
zero = \f . \x . x
three = \f . \x . f (f (f x)))
```

或

```
three = \f . compose f (compose f f)
```

对于 $succ$，比较 n 和 $n+1$ 的表示：

$$n = \lambda f.\lambda x.\underbrace{f\ (f\ \cdots\ (f\ x))}_{\text{共有 } n \text{ 个 } f}$$

$$succ\ n = \lambda f.\lambda x.\underbrace{f\ (f\ \cdots\ (f\ x))}_{\text{共有 } n+1 \text{ 个 } f}$$

如果我们把 $\lambda f.\lambda x.$ 移到等式左边，可以得到

$$n\ f\ x = \underbrace{f\ (f\ \cdots\ (f\ x))}_{\text{共有 } n \text{ 个 } f}$$

$$\begin{aligned}(succ\ n)\ f\ x &= \underbrace{f\ (f\ \cdots\ (f\ x))}_{\text{共有 } n+1 \text{ 个 } f} \\ &= f\ (\underbrace{f\ (f\ \cdots\ (f\ x))}_{\text{共有 } n \text{ 个 } f}) \\ &= f\ (n\ f\ x)\end{aligned}$$

所以可以用拆开邱奇数的 $\lambda f.\lambda x.$ 的方式定义 $succ$：

```
succ = \n . \f . \x . f (n f x)
```

另一方面，我们会发现

$$n\ f = \underbrace{f \circ f \circ f \circ \cdots \circ f}_{\text{共有 } n \text{ 个 } f} = f^{\circ n}$$

其中 $\circ$ 表示之前的 $compose$。从这个角度上看，

$$(succ\ n)\ f = f^{\circ(n+1)} = f \circ f^{\circ n} = f \circ (n\ f)$$

所以

```
succ = \n . \f . compose f (n f)
```

3.7 测试点 7

从 $n\ f = f^{\circ n}$ 的角度看，$plus$ 和 mul 会比较显然：

对于 $plus$，$f^{\circ(m+n)} = f^{\circ m} \circ f^{\circ n}$，所以有

```
plus m n = \f . compose (m f) (n f)
```

对于 mul，$f^{\circ(m\times n)} = (f^{\circ m})^{\circ n}$，所以有

```
mul m n = \f . n (m f)
```

对于 exp，

$$\begin{aligned}
&(exp\ m\ n)\ f\\
=&f^{\circ m^n}\\
=&f^{\circ(\overbrace{m\times m\times\cdots\times m}^{\text{共有 } n \text{ 个 } m})}\\
=&\underbrace{m\ (m\ (m\ \cdots\ (m\ f)))}_{\text{共有 } n \text{ 个 } m}
\end{aligned}$$

我们会发现这和邱奇数的形式非常接近：邱奇数 n 将 f 在 x 上作用了 n 次，而 $exp\ m\ n$ 将 m 在 f 上作用了 n 次。因此，

```
exp m n = n m
```

3.8 测试点 8

我们没有办法通过 $O(1)$ 的操作来从将 $f^{\circ n} = f \circ f^{\circ(n-1)}$ 中将最外面的 f 直接剥去。所以，可以采取一个看起来比较蠢的办法：

- 因为我们可以用邱奇数将函数 f 在 x 上作用 n 遍，而 $n-1$ 与 n 接近，我们可以考虑对 0 进行 n 次左右的 $succ$。
- 但是，进行 n 次 $succ$ 就和 n 一样了，所以我们希望能特判第一次，即第一次什么都不做，之后每次对现在的数进行 $succ$。
- 考虑平时写程序时用一个布尔变量记录“现在是不是第一次”的方法，因为我们已经有了 $true$ 和 $false$，并可以通过 bxy（b 是 $true$ 或 $false$）来选择进行 x 或 y（类似于 if），所以这是可行的。

具体来说，我们用一个 *pair* 记录现在是否是第一次，以及被 *succ* 的数。

```
init = pair true zero
```

我们根据这个 *pair* 中的 $first$ 决定不动（或者返回 0），还是进行 *succ*：

```
succifnotfirst p = (first p) zero (succ (second p))
```

不过我们要进行 n 次的操作必须也返回一个 *pair*，不然无法对其作用 n 次。返回的 *pair* 的 $first$ 是 $false$，表示已经至少进行过一次操作了：

```
operation p = pair false (succifnotfirst p)
```

这样，我们只要对 *init* 进行 n 次 *operation*：

```
result n = n operation init
```

由于返回值是一个 *pair*，我们要取出 *pair* 的 *second* 作为 *pred* 的返回值：

```
pred n = second (result n)
```

整理得到

```
init = pair true zero
operation = \p . pair false ((first p) zero (succ (second p)))
pred = \n . second (n operation init)
```

当然也有其他解法，如

```
extract = \container . container (\u . u)
pred = \n . \f . \x . extract (n (\g . \h . h (g f)) (\u . x))
```

这里

- 将 $v = f^{\circ n}(x)$ 包装成 $\lambda h.h\ v$ 的容器，通过 *extract* 提取容器中的值
- $\lambda g.\lambda h.h\ (g\ f)$ 通过对容器 g 里的值作用 f，并用 $\lambda h.h$ 重新包装成容器，从而达到类似 *succ* 的效果
- $\lambda u.x$ 通过这种容器的设置，起到了一个类似容器中包装 -1 的效果，即 $g\ f$ 中 $g = \lambda u.x$ 时，$g\ f = x$，类似于 *succ* 后为 $0\ f\ x$。
- 最后从容器中 *extract* 出 $(pred\ n)\ f\ x$。

3.9 测试点 9

有了 $pred$ 减法就相当简单了。一次 $pred$ 能让一个数减 1，n 次 $pred$ 就能减 n，所以

```
minus m n = n pred m
```

3.10 测试点 10

对于 $iszero$，我们可以用类似 $pred$ 的套路：一开始是 $true$，进行一次之后就永远是 $false$，一共进行 n 次。这里我们没有必要用包装一个 $pair$，因为除了这个布尔值，我们没有什么要存的东西：

```
iszero n = n (\x . false) true
```

在之前的测试点中，有一个可能有些奇怪的规定：$pred\ zero = zero$。这意味着，对于 $a \leq b$，$minus\ a\ b = b\ pred\ a = zero$。这样一来，$leq$ 就可以用 $minus$ 和 $iszero$ 实现：

```
leq m n = iszero (minus m n)
```

而 eq 显然是可以用 leq 实现的：如果 $a \leq b$ 且 $b \leq a$，那么必然有 $a = b$，所以

```
eq m n = and (leq m n) (leq n m)
```

3.11 测试点 11

在邱奇数的编码中，减法已经相当困难了，如果直接做除法可能难以达成。如果我们思考我们平时是怎么做除法的以及高精度除法的实现，结合关于递归的提示，可以发现除法能用递归的减法表示：

$$m \div n = \text{if } m < n \text{ then } 0 \text{ else } 1 + (m - n) \div n$$

用 λ-演算表示的话，就是

$$divide' = \lambda m.\lambda n.(leq\ (succ\ m)\ n)\ zero\ (succ\ (divide'\ (minus\ m\ n)\ n))$$

这里用了 $divide'$ 而不是 $divide$，是因为这并不是真的 λ-表达式的项，它的定义中引用了自身。

根据 Y 组合子的介绍，我们要考虑将 $divide'$ 变成这样：

$$div = \lambda c.\lambda m.\lambda n.(leq\ (succ\ m)\ n)\ zero\ (succ\ (c\ (minus\ m\ n)\ n))$$

使得本来应该递归调用自身的地方，用一个传入的参数 c 代替，并祈祷 Y 组合子将 $divide'$ 当作 c 传进来，从而使函数体变得和 $divide'$ 等价。

幸运的是，Y 组合子干的正是这件事。根据题面中的推导，Y 组合子能找到 div 的不动点，即 $\mathsf{Y}\ div \equiv div\ (\mathsf{Y}\ div)$。观察等式可知，左侧的 $\mathsf{Y}\ div$ 就是右侧将 $\mathsf{Y}\ div$ 当作参数 c 传入 div 的结果，写成等式就是：

$$\mathsf{Y}\ div = div\ (\mathsf{Y}\ div) = \lambda m.\lambda n.(leq\ (succ\ m)\ n)\ zero\ (succ\ ((\mathsf{Y}\ div)\ (minus\ m\ n)\ n))$$

这时，将 $\mathsf{Y}\ div$ 视为一个整体，上式就和 $divide'$ 的（不严谨）定义完全一样，每处 $divide'$ 都被替换成 $\mathsf{Y}\ div$。$\mathsf{Y}\ div$ 就是我们想要的 $divide'$。

这样一来，我们就可以得到一个答案：

```
y = \f . (\x . f (x x)) (\x . f (x x))
div = \c . \m . \n . (leq (succ m) n) zero (succ (c (minus m n) n))
divide = y div
```

当然，为了效率考虑，计算 $m < n$ 和 $m - n$ 事实上都进行了一次减法，减法是相当耗时间的事情。（邱奇数的运算相当慢，所以测试数据中只能用 10 以内的数字进行除法。）如果能合并两次类似的减法，效率能够大幅度提升。因为 $m < n$ 用 $m+1 \le n$ 计算，而 $m+1 \le n$ 包含的减法是 $(m+1)-n$ 而不是 $m-n$（$((m+1)-n)-1$ 并不永远等于 $m-n$），所以似乎要将 $m+1 \le n$ 改成 $m \le n$ 才能合并。我们把 div 修改为 $divone$，这个函数使 $\mathsf{Y}\ div$ 计算 $(m-1) \div n$，这样一来两边的减法就一样了：

```
divone = \c . \m . \n . (leq m n) zero (succ (c (minus m n) n))
divide = \m . \n . (y divone) (succ m) n
```

此时我们还没有真的合并两次减法。我们先 inline $leq\ m\ n$：

```
divone = \c . \m . \n . (iszero (minus m n)) zero (succ (c (minus m n) n))
```

怎么将两个相同的项合并到一起去呢？我们可能希望有一个 let $a = v$ in B 的语法，表示将 $a = v$ 代入 B，这样一来就可以只算一次：

```
divone = \c . \m . \n . let d = (minus m n) in (iszero d) zero (succ (c d n))
```

可惜并没有这种语法。但是，参考我在一开始提到的，将 $\lambda x.N$ 解释为这样一个函数，它调用时返回将 x 的值代入 N 的计算结果，这个概念和我们想要的 let ... in ... 是相当类似的。事实上，我们可以这样实现 let $a = v$ in B：

$$\text{let } a = v \text{ in } B \Longleftrightarrow (\lambda a.B)\ v$$

通过这样的方式，我们可以在当前上下文增加一个名字 a 的绑定。所以

```
divone = \c . \m . \n . (\d . (iszero d) zero (succ (c d n))) (minus m n)
```

当然，这个题目的测试数据并没有卡常，所以这个优化在这里是无关紧要的。