Verified fast formulas for control bits for permutation networks

Daniel J. Bernstein[1,2]

[1] Department of Computer Science, University of Illinois at Chicago, USA
[2] Horst Görtz Institute for IT Security, Ruhr University Bochum, Germany
djb@cr.yp.to

Abstract. This paper presents detailed and computer-verified proofs of formulas that, given a permutation π of 2^m indices with $m \geq 1$, produce control bits for a standard permutation network that uses $2^m(m-1/2)$ swaps to apply π to a list. The formulas match the control bits computed by a serial algorithm of Stone (1968) and a parallel algorithm of Nassimi–Sahni (1982). The proofs are a step towards computer-verified correctness proofs for efficient implementations of these algorithms.

Keywords: sorting networks, permutation networks, Beneš networks, Clos networks, Stone's algorithm, Nassimi–Sahni algorithm, constant-time algorithms, parallelization, vectorization

1 Introduction

If π is a permutation of $\{0, 1, \ldots, n-1\}$ then sorting the n pairs

$$(\pi(0), 0), (\pi(1), 1), \ldots, (\pi(n-1), n-1)$$

into increasing order produces the n pairs

$$(0, \pi^{-1}(0)), (1, \pi^{-1}(1)), \ldots, (n-1, \pi^{-1}(n-1)).$$

The problem of inverting a permutation of $\{0, 1, \ldots, n-1\}$ thus reduces to the problem of sorting a size-n list of pairs.

More generally, sorting the pairs $(\pi(0), a_0), (\pi(1), a_1), \ldots, (\pi(n-1), a_{n-1})$ for any array $(a_0, a_1, \ldots, a_{n-1})$ gives $(0, a_{\pi^{-1}(0)}), (1, a_{\pi^{-1}(1)}), \ldots, (n-1, a_{\pi^{-1}(n-1)})$. Sorting the pairs $(\pi^{-1}(0), a_0), (\pi^{-1}(1), a_1), \ldots, (\pi^{-1}(n-1), a_{n-1})$ into increasing order produces the pairs $(0, a_{\pi(0)}), (1, a_{\pi(1)}), \ldots, (n-1, a_{\pi(n-1)})$. The problem of applying a permutation to a size-n array thus also reduces to the sorting problem.

This work was supported by the German Research Foundation under EXC 2092 CASA 390781972 "Cyber Security in the Age of Large-Scale Adversaries", by the U.S. National Science Foundation under grant 1913167, and by the Cisco University Research Program. "Any opinions, findings, and conclusions or recommendations expressed in this material are those of the author(s) and do not necessarily reflect the views of the National Science Foundation" (or other funding agencies). Permanent ID of this document: 782136154c6ea5bb93257e2a5f667a579df7a48e. Date: 2020.09.23.

Wait a minute. Why not simply run through each i, using $\pi(i)$ as an index into the RAM storing the input array to obtain the desired output $a_{\pi(i)}$? If the objective is to compute $\pi^{-1}(i)$ or more generally $a_{\pi^{-1}(i)}$, why not simply run through each j, storing a_j at position $\pi(j)$ in the output array? Here are two answers:

- Security: On many popular computer architectures, RAM leaks information about indices through timings. If π is secret then one has to figure out whether too much information is leaked, whereas there are sorting algorithms that avoid this issue.
- Speed: RAM uses more and more transistors as n grows to look up a single array element, and n lookups occupy RAM for many cycles. There are much faster ways to sort n items using that amount of hardware.

So let's return to the sorting approach.

A **comparator** sorts two items in place. Given two inputs p, q, let's say 64-bit integers, the comparator overwrites p, q with $\min\{p, q\}, \max\{p, q\}$ respectively. It is easy to build a fast comparator that takes time independent of p and q.

Assume for simplicity that $n = 2^m$ is a power of 2. Batcher's bitonic sorting network sorts n items using $(m^2 + m)n/4$ comparators. Even better, Batcher's odd-even sorting network sorts n items using $(m^2 - m + 4)n/4 - 1$ comparators. Each network consists of $m(m + 1)/2$ stages, where each stage applies $\leq n/2$ comparators in parallel. See Batcher's original paper [**5**]. Readers interested in the non-power-of-2 case should see Knuth's merge-exchange sort [**16**, Algorithm 5.2.2M]; this is a simple sorting network that works for any size, and aside from data layout it is equivalent to the odd-even sorting network for power-of-2 sizes.

There are at least three ways in which this method of permuting a list is known to be suboptimal:

- The AKS sorting network [**4**] and its descendants have only $\Theta(m)$ stages. However, the survey in [**13**, Section 5] indicates that the hidden Θ constant is above 1000, whereas competing with Batcher's networks for any reasonable value of m would require a Θ constant far below 100.
- Some marginally smaller sorting networks are known for reasonable values of m, for example using 60 comparators for $m = 4$ where Batcher's odd-even sorting network uses 63. However, this generally seems outweighed by the relatively simple structure of Batcher's networks.
- Often a permutation π is going to be applied to *many* lists. There is a standard technique that saves time in each application of π, at the expense of a one-time precomputation that depends only on π. Specifically, for $m \geq 1$, one can permute a list using a standard permutation network containing just $2m - 1$ stages, where each stage has $n/2$ conditional swaps controlled by π.

This paper focuses on the last point, and more specifically on algorithms that, given π, compute $(2m-1)n/2$ control bits for conditional swaps in the standard permutation network.

There are various algorithms of this type in the literature, including fast parallel algorithms. The parallel algorithms are usually stated in terms of parallel

RAM accesses, which in turn can be converted into sorting steps.[3] There are various reports of software and hardware implementations computing the desired control bits, such as fast vectorized constant-time software introduced as part of McBits [**10**], and modified software introduced as part of Classic McEliece [**9**] using the sorting library from [**8**] to save time in the sorting steps.

Could there be bugs in the algorithms, or in the implementations? Perhaps testing some random inputs is adequate, or perhaps not; the literature has not studied what sort of bugs happen here, and the probability of triggering each bug. A bug that triggers only occasional failures will not be caught by typical tests. The computations are not very complicated, but the broader history of computing shows disastrous bugs appearing in simpler computations.

Testing control bits immediately after they are computed is relatively cheap, and can be recommended in any case as protection against hardware faults. But does the application have the flexibility to try another permutation if one permutation fails? Even if it does, perhaps the distribution of permutations in that application triggers constant failures, or frequent enough failures to be a performance problem. If failures are *not* common then experience shows that implementors often omit tests for failures, even when the tests are cheap, simply because the tests require nonzero implementor effort.

Probably the most convincing way forward, for a reviewer interested in high assurance, would be computer-verified proofs of

- theorems saying that the implementations, given a permutation π as input, compute control bits defined by particular mathematical formulas; and
- theorems saying that the control bits defined by those formulas are control bits for π.

Today it is surprisingly difficult to find theorems of the second type—never mind the first, and never mind computer verification. An algorithm statement is not conceptually different from a formula, but the algorithm statements in the literature are generally intertwined with descriptions of how the algorithms were developed, usually have layers that distract from verification, and are rarely accompanied by proofs.

The goals of this paper are (1) to present mathematical formulas that convert a permutation π into control bits, and (2) to present computer-verified proofs that the control bits produced by those formulas are control bits for π. Sections 2 through 5 of this paper present the formulas and detailed proofs in a traditional mathematical format. Appendix A presents computer-verified proofs. Sections 6 and 7 present further material that has *not* been similarly verified.

The formulas were derived from, and produce the same control bits as, a 1968 algorithm [**24**] by Stone and a 1982 parallel algorithm [**19**] by Nassimi–Sahni. (If I've correctly understood a 1981 parallel algorithm [**18**] by Lev–Pippenger–Valiant then that algorithm often produces different control bits for the same permutation, as explained below.) The core ideas used in the proofs are also not new—although I doubt that the core ideas are where bugs are likely to occur!

[3] Sorting is thus used some number of times to compute control bits that are in turn used to avoid sorting, like an employee training the employee's replacement.

2 Cycles

This section defines "cycle π", which maps x to the cycle of x under permutation π, and "cyclemin π", which maps x to the minimum element of the cycle of x under π. This section also defines intermediate results in a fast parallel algorithm to compute cyclemin π.

Definition 2.1. *Let S be a set. Let π be a permutation of S. Then "*$\operatorname{cycle}\pi$*" means the function from S to 2^S that maps x to $\left\{\pi^k(x) : k \in \mathbf{Z}\right\}$.*

Theorem 2.2. *Let S be a set. Let π be a permutation of S. Define $C = \operatorname{cycle}\pi$. Then $C(\pi(x)) = C(x)$ for each $x \in S$.*

Proof. $\left\{\pi^k(\pi(x)) : k \in \mathbf{Z}\right\} = \left\{\pi^{k+1}(x) : k \in \mathbf{Z}\right\} = \left\{\pi^k(x) : k \in \mathbf{Z}\right\}$. □

Definition 2.3. *Let S be a set of nonnegative integers. Let π be a permutation of S. Then "*$\operatorname{cyclemin}\pi$*" means the function from S to S that maps x to $\min((\operatorname{cycle}\pi)(x))$.*

This definition can be generalized from subsets of $\{0, 1, 2, \ldots\}$ to subsets of any well-ordered set.

Theorem 2.4. *Let S be a set of nonnegative integers. Let π be a permutation of S. Define functions $c_0, c_1, \cdots : S \to S$ as follows: $c_0(x) = x$; $c_{i+1}(x) = \min\{c_i(x), c_i(\pi^{2^i}(x))\}$. Then $c_i(x) = \min\{x, \pi(x), \ldots, \pi^{2^i-1}(x)\}$.*

Proof. Induct on i.

Case 0: $i = 0$. Then $\{x, \pi(x), \ldots, \pi^{2^i-1}(x)\} = \{x\}$ and $c_i(x) = x = \min\{x\}$ as claimed.

Case 1: $i \geq 1$. By the inductive hypothesis,

$$c_{i-1}(x) = \min\{x, \pi(x), \ldots, \pi^{2^{i-1}-1}(x)\}$$

for all x, so

$$c_{i-1}(\pi^{2^{i-1}}(x)) = \min\{\pi^{2^{i-1}}(x), \pi^{2^{i-1}+1}(x), \ldots, \pi^{2^i-1}(x)\},$$

so

$$c_i(x) = \min\{x, \pi(x), \ldots, \pi^{2^i-1}(x)\}$$

as claimed. □

Theorem 2.5. *In the situation of Theorem 2.4, let x be an element of S, and assume that $\#((\operatorname{cycle}\pi)(x)) \leq 2^i$. Then $c_i(x) = (\operatorname{cyclemin}\pi)(x)$.*

Proof. Write $x_k = \pi^k(x)$ for each $k \in \mathbf{Z}$. Then $\{x_0, x_1, \ldots, x_{2^i}\} \subseteq \{x_k : k \in \mathbf{Z}\}$, so $\#\{x_0, x_1, \ldots, x_{2^i}\} \leq 2^i$, so $x_a = x_b$ for some distinct $a, b \in \left\{0, 1, \ldots, 2^i\right\}$. Without loss of generality assume $a < b$.

Now $x_0 = \pi^{-a}(x_a) = \pi^{-a}(x_b) = x_{b-a}$. By induction $x_k = x_{k \bmod (b-a)}$ for every $k \in \mathbf{Z}$. Now $k \bmod (b-a) \in \{0, 1, \ldots, b-a-1\} \subseteq \left\{0, 1, \ldots, 2^i - 1\right\}$. By Theorem 2.4, $c_i(x) = \min\{x_k : 0 \leq k < 2^i\} = \min\{x_k : 0 \leq k < b-a\} = \min\{x_k : k \in \mathbf{Z}\} = (\operatorname{cyclemin}\pi)(x)$. □

3 Controlled swaps

Given a b-bit sequence $s = (s_0, s_1, \ldots, s_{b-1})$, this section defines an involution "Xif s" of $\{0, 1, \ldots, 2b-1\}$. The involution swaps 0 and 1 if the first bit s_0 is 1, else leaves 0 and 1 in place; swaps 2 and 3 if the next bit s_1 is 1, else leaves 2 and 3 in place; etc. The bits in s are called **control bits** for this involution.

Theorem 3.1. *If b is a nonnegative integer and $x \in \{0, 1, \ldots, 2b-1\}$ then $x \oplus 1 \in \{0, 1, \ldots, 2b-1\}$ and $\lfloor (x \oplus 1)/2 \rfloor = \lfloor x/2 \rfloor$.*

Proof. Case 0: x is even. Then $0 \leq x \leq 2b-2$; $x \oplus 1 = x + 1 \leq 2b-1$; and $\lfloor (x \oplus 1)/2 \rfloor = \lfloor (x+1)/2 \rfloor = \lfloor x/2 \rfloor$.

Case 1: x is odd. Then $1 \leq x \leq 2b-1$; $x \oplus 1 = x - 1 \geq 0$; and $\lfloor (x \oplus 1)/2 \rfloor = \lfloor (x-1)/2 \rfloor = \lfloor x/2 \rfloor$. □

Definition 3.2. *Let b be a nonnegative integer. Let $s = (s_0, s_1, \ldots, s_{b-1})$ be an element of $\{0,1\}^b$. Define a function S on $\{0, 1, \ldots, 2b-1\}$ as follows: $S(x) = x \oplus s_{\lfloor x/2 \rfloor}$. Then "*Xif *$s$" means S.*

Theorem 3.3. *Let b be a nonnegative integer. Let s be an element of $\{0,1\}^b$. Then* Xif *s is an involution of $\{0, 1, \ldots, 2b-1\}$.*

Proof. Write s as $(s_0, s_1, \ldots, s_{b-1})$, and write $S = \text{Xif}\, s$.

Fix $x \in \{0, 1, \ldots, 2b-1\}$, and write $y = S(x) = x \oplus s_{\lfloor x/2 \rfloor}$.

Case 0: $s_{\lfloor x/2 \rfloor} = 0$. Then $y = x$ so $y \in \{0, 1, \ldots, 2b-1\}$ and $S(y) = S(x) = x$.

Case 1: $s_{\lfloor x/2 \rfloor} = 1$. Then $y = x \oplus 1$. By Theorem 3.1, $y \in \{0, 1, \ldots, 2b-1\}$ and $\lfloor y/2 \rfloor = \lfloor x/2 \rfloor$, so $S(y) = y \oplus s_{\lfloor y/2 \rfloor} = (x \oplus s_{\lfloor y/2 \rfloor}) \oplus s_{\lfloor y/2 \rfloor} = x$.

Either way $y \in \{0, 1, \ldots, 2b-1\}$ and $S(y) = x$. □

4 Back and forth

This section defines, for each permutation π of $\{0, 1, \ldots, 2b-1\}$, a permutation "XbackXforth π" of the same set.

Definition 4.1. *Let b be a nonnegative integer. Let π be a permutation of $\{0, 1, \ldots, 2b-1\}$. Then "*XbackXforth *π" is the function from $\{0, 1, \ldots, 2b-1\}$ to $\{0, 1, \ldots, 2b-1\}$ that maps x to $\pi(\pi^{-1}(x \oplus 1) \oplus 1)$.*

Theorem 4.2. *Let b be a nonnegative integer. Let π be a permutation of $\{0, 1, \ldots, 2b-1\}$. Then* XbackXforth *π is a permutation of $\{0, 1, \ldots, 2b-1\}$.*

Proof. By definition $\overline{\pi}$ is a composition of the following four permutations: $x \mapsto x \oplus 1$, which is a permutation of $\{0, 1, \ldots, 2b-1\}$ by Theorem 3.1; π^{-1}; $x \mapsto x \oplus 1$ again; and π. □

Theorem 4.3. *Let b be a nonnegative integer. Let π be a permutation of $\{0, 1, \ldots, 2b-1\}$. Define $\overline{\pi} =$* XbackXforth *π. Fix $x \in \{0, 1, \ldots, 2b-1\}$, and define $x_k = \overline{\pi}^k(x)$ for all $k \in \mathbf{Z}$. Then $\overline{\pi}^k(x_k \oplus 1) = x \oplus 1$; $x_j \neq x_k \oplus 1$ for all $j, k \in \mathbf{Z}$; and $\#\{x_k : k \in \mathbf{Z}\} \leq b$.*

In other words, for each cycle $x_0 \to x_1 \to \cdots \to x_{c-1} \to x_c = x_0$ of $\overline{\pi}$, there is a cycle $x_c \oplus 1 \to x_{c-1} \oplus 1 \to \cdots \to x_1 \oplus 1 \to x_0 \oplus 1 = x_c \oplus 1$ of $\overline{\pi}$, and these cycles are disjoint, each of length $c \le b$.

Proof. By definition $x_{k+1} = \overline{\pi}(x_k) = \pi(\pi^{-1}(x_k \oplus 1) \oplus 1)$ and $\overline{\pi}(x_{k+1} \oplus 1) = \pi(\pi^{-1}(x_{k+1}) \oplus 1)$ so $\overline{\pi}(x_{k+1} \oplus 1) = \pi(\pi^{-1}(x_k \oplus 1)) = x_k \oplus 1$. Iterate to see that $\overline{\pi}^k(x_k \oplus 1) = x \oplus 1$.

Suppose there are pairs (j, k) with $x_j = x_k \oplus 1$. If $x_j = x_k \oplus 1$ then $x_k = x_j \oplus 1$ so there are pairs (j, k) with $x_j = x_k \oplus 1$ and $j \le k$. Take some such pair (j, k) with $k - j$ minimal.

If $k = j$ then $x_j = x_j \oplus 1$, contradiction. If $k = j + 1$ then $x_j = x_{j+1} \oplus 1$ so $\pi^{-1}(x_{j+1}) = \pi^{-1}(x_j \oplus 1) \oplus 1 = \pi^{-1}(x_{j+1}) \oplus 1$, contradiction. If $k \ge j + 2$ then $x_{j+1} = \overline{\pi}(x_j) = \overline{\pi}(x_k \oplus 1) = x_{k-1} \oplus 1$ so there is a pair with smaller $k - j$, contradiction. Hence x_j never equals $x_k \oplus 1$.

Write $S_0 = \{x_k : k \in \mathbf{Z}\}$ and $S_1 = \{x_k \oplus 1 : k \in \mathbf{Z}\}$. Then $S_0 \cup S_1 \subseteq \{0, 1, \ldots, 2b - 1\}$, so $\#(S_0 \cup S_1) \le 2b$. Also S_0 and S_1 are disjoint, so $\#(S_0 \cup S_1) = \#S_0 + \#S_1$; and $\#S_0 = \#S_1$, since $\oplus 1$ is a permutation. Hence $2\#S_0 \le 2b$; i.e., $\#S_0 \le b$. □

Theorem 4.4. *Let b be a nonnegative integer. Let π be a permutation of $\{0, 1, \ldots, 2b - 1\}$. Define $\overline{\pi} = \mathrm{XbackXforth}\,\pi$. Define $c = \mathrm{cyclemin}\,\overline{\pi}$. Then $c(x \oplus 1) = c(x) \oplus 1$ for each $x \in \{0, 1, \ldots, 2b - 1\}$.*

Proof. Write $x_k = \overline{\pi}^k(x)$ for $k \in \mathbf{Z}$. Find j that minimizes x_j. By definition $c(x) = x_j$.

Also write $y_k = \overline{\pi}^k(x \oplus 1)$ for $k \in \mathbf{Z}$. Then $y_{-k} = x_k \oplus 1$ by Theorem 4.3.

By definition $c(x \oplus 1) = \min\{y_k : k \in \mathbf{Z}\}$. One of the entries in the min is $y_{-j} = x_j \oplus 1 = c(x) \oplus 1$, so $c(x \oplus 1) \le c(x) \oplus 1$.

If $c(x \oplus 1) < c(x) \oplus 1$ then $y_k < c(x) \oplus 1$ for some k, so $x_{-k} \oplus 1 < x_j \oplus 1$, but $x_j \le x_{-k}$, forcing $x_j = x_{-k} \oplus 1$, which contradicts Theorem 4.3. Hence $c(x \oplus 1) = c(x) \oplus 1$ as claimed. □

5 Control bits from permutations

This section writes any permutation π of $\{0, 1, \ldots, 2b - 1\}$ as

$$\mathrm{Xif}\,\mathrm{firstcontrol}\,\pi \circ \mathrm{middleperm}\,\pi \circ \mathrm{Xif}\,\mathrm{lastcontrol}\,\pi,$$

where $\mathrm{middleperm}\,\pi$ is a permutation that preserves the parity of its inputs. The functions firstcontrol, middleperm, and lastcontrol are defined below.

Definition 5.1. *Let b be a nonnegative integer. Let π be a permutation of $\{0, 1, \ldots, 2b - 1\}$. Define*

- $c = \mathrm{cyclemin}\,\mathrm{XbackXforth}\,\pi$*;*
- $f_j = c(2j) \bmod 2$ *for each* $j \in \{0, 1, \ldots, b - 1\}$*;*
- $F = \mathrm{Xif}\,f$ *where* $f = (f_0, f_1, \ldots, f_{b-1})$*;*

- $\ell_k = F(\pi(2k)) \bmod 2$ *for each* $k \in \{0, 1, \ldots, b-1\}$*; and*
- $L = \text{Xif}\,\ell$ *where* $\ell = (\ell_0, \ell_1, \ldots, \ell_{b-1})$.

Then

- "firstcontrol π" *means the sequence* $f \in \{0,1\}^b$*;*
- "lastcontrol π" *means the sequence* $\ell \in \{0,1\}^b$*; and*
- "middleperm π" *means the function* $F \circ \pi \circ L : \{0, 1, \ldots, 2b-1\} \to \{0, 1, \ldots, 2b-1\}$.

Theorem 5.2. *Let* b *be a nonnegative integer. Let* π *be a permutation of* $\{0, 1, \ldots, 2b-1\}$*. Define* $f = \text{firstcontrol}\,\pi$*. Then* $f_0 = 0$.

Proof. Write $\overline{\pi} = \text{XbackXforth}\,\pi$. By definition $\overline{\pi}$ is a permutation of $\{0, 1, \ldots, 2b-1\}$.

Write $C = \text{cycle}\,\overline{\pi}$. By definition $0 \in C(0) \subseteq \{0, 1, \ldots, 2b-1\}$.

Write $c = \text{cyclemin}\,\overline{\pi}$. By definition $c(0) = \min C(0) = 0$.

By definition $f_0 = c(0) \bmod 2$, so $f_0 = 0$. □

Theorem 5.3. *Let* b *be a nonnegative integer. Let* π *be a permutation of* $\{0, 1, \ldots, 2b-1\}$*. Define* $c = \text{cyclemin XbackXforth}\,\pi$*. Define* $F = \text{Xif firstcontrol}\,\pi$*. Then* $F(x) \equiv c(x) \pmod 2$ *for each* $x \in \{0, 1, \ldots, 2b-1\}$.

Proof. Write firstcontrol π as $(f_0, \ldots, f_{b-1})$. By definition $F(x) = x + f_{\lfloor x/2 \rfloor}$.

Case 0: $x = 2j$ for some $j \in \{0, 1, \ldots, b-1\}$. Then $F(x) \equiv x + f_j \equiv f_j \equiv c(2j) = c(x) \pmod 2$.

Case 1: $x = 2j + 1$ for some $j \in \{0, 1, \ldots, b-1\}$. By Theorem 4.4, $c(x) = c(2j) \oplus 1$, so $F(x) \equiv x + f_j \equiv 1 + f_j \equiv 1 + c(2j) \equiv c(x) \pmod 2$. □

Theorem 5.4. *Let* b *be a nonnegative integer. Let* π *be a permutation of* $\{0, 1, \ldots, 2b-1\}$*. Define* $F = \text{Xif firstcontrol}\,\pi$ *and* $L = \text{Xif lastcontrol}\,\pi$*. Then* $L(x) \equiv F(\pi(x)) \pmod 2$ *for each* $x \in \{0, 1, \ldots, 2b-1\}$.

Proof. Define $\overline{\pi} = \text{XbackXforth}\,\pi$; $c = \text{cyclemin}\,\overline{\pi}$; $f = \text{firstcontrol}\,\pi$; and $\ell = \text{firstcontrol}\,\pi$. By definition $f_j = c(2j) \bmod 2$ and $\ell_k = F(\pi(2k)) \bmod 2$.

Case 0: $x = 2k$ for some $k \in \{0, 1, \ldots, b-1\}$. By definition $L(x) = L(2k) \equiv \ell_k \equiv F(\pi(2k)) = F(\pi(x)) \pmod 2$ as claimed.

Case 1: $x = 2k + 1$ for some $k \in \{0, 1, \ldots, b-1\}$. Write $u = \pi(2k)$. By definition $\overline{\pi}(u \oplus 1) = \pi(\pi^{-1}(u) \oplus 1) = \pi(2k+1) = \pi(x)$, so $c(u \oplus 1) = c(\pi(x))$ by Theorem 2.2, so $c(u) \oplus 1 = c(\pi(x))$ by Theorem 4.4, so $F(u) + 1 \equiv F(\pi(x)) \pmod 2$ by Theorem 5.3, so $L(x) = L(2k+1) \equiv 1 + \ell_k \equiv 1 + F(u) \equiv F(\pi(x)) \pmod 2$ as claimed. □

Theorem 5.5. *Let* b *be a nonnegative integer. Let* π *be a permutation of* $\{0, 1, \ldots, 2b-1\}$*. Define* $M = \text{middleperm}\,\pi$*. Then* M *is a permutation of* $\{0, 1, \ldots, 2b-1\}$*, and* $M(x) \equiv x \pmod 2$ *for each* $x \in \{0, 1, \ldots, 2b-1\}$.

Proof. Define c, f, F, ℓ, L as in Definition 5.1. By Theorem 3.3, F and L are involutions of $\{0, 1, \ldots, 2b-1\}$. M is a permutation since it is the composition of permutations F, π, L.

By Theorem 5.4, $F(\pi(x)) \equiv L(x) \pmod 2$ for each x. Substitute $L(x)$ for x to see that $M(x) = F(\pi(L(x))) \equiv L(L(x)) = x \pmod 2$. □

6 Permutation networks

Consider the problem of using a permutation π of $\{0,1,\ldots,n-1\}$ to permute a list $(a_0,a_1,\ldots,a_{n-1})$ in place, overwriting the list with $(a_{\pi(0)},a_{\pi(1)},\ldots,a_{\pi(n-1)})$. For example, applying such an algorithm to the list $(0,1,\ldots,n-1)$ obtains $(\pi(0),\pi(1),\ldots,\pi(n-1))$.

Assume $n=2b$. Section 5 constructed sequences $f=\operatorname{firstcontrol}\pi\in\{0,1\}^b$ and $\ell=\operatorname{lastcontrol}\pi\in\{0,1\}^b$, and another permutation $M=\operatorname{middleperm}\pi$ of $\{0,1,\ldots,2b-1\}$, with the following properties:

- π decomposes as $F\circ M\circ L$, where F is the involution $\operatorname{Xif} f$ and L is the involution $\operatorname{Xif}\ell$.
- $M(x)$ has the same parity as x. Consequently M permutes $\{0,2,\ldots,2b-2\}$ and separately permutes $\{1,3,\ldots,2b-1\}$.

One can rewrite $M(2x)$ as $2M_0(x)$, and rewrite $M(2x+1)$ as $2M_1(x)+1$, where M_0 and M_1 are permutations of $\{0,1,\ldots,b-1\}$.

These properties can be used to permute a list $(a_0,a_1,\ldots,a_{2b-1})$ in place as follows:

- First stage, using the control bits f: Overwrite (a_0,a_1) with $(a_{F(0)},a_{F(1)})$, overwrite (a_2,a_3) with $(a_{F(2)},a_{F(3)})$, etc. (Now a_i is what was $a_{F(i)}$ at the beginning.)
- Middle stage, using the permutation M: Overwrite $(a_0,a_2,\ldots,a_{2b-2})$ with $(a_{M(0)},a_{M(2)},\ldots,a_{M(2b-2)})$, and overwrite $(a_1,a_3,\ldots,a_{2b-1})$ with $(a_{M(1)},a_{M(3)},\ldots,a_{M(2b-1)})$. (Now a_i is what was $a_{M(i)}$ before this stage, i.e., what was $a_{F(M(i))}$ at the beginning.)
- Last stage, using the control bits ℓ: Overwrite (a_0,a_1) with $(a_{L(0)},a_{L(1)})$, overwrite (a_2,a_3) with $(a_{L(2)},a_{L(3)})$, etc. (Now a_i is what was $a_{L(i)}$ before this stage, i.e., what was $a_{F(M(L(i)))}$ at the beginning.)

This pattern of operations is sometimes called a "three-stage Clos network", although this is a misnomer; see Section 6.5.

An **in-place Beneš network** for $b=2^m$ uses the same technique recursively to handle the two half-size permutations in the middle stage. Overall an in-place Beneš network has $2m-1$ stages, each stage consisting of $n/2$ parallel conditional swaps. The conditional swaps use indices at distance 1 in the first stage, distance 2 in the second stage, distance 4 in the third stage, and so on up through 2^{m-1}, and then back down through 1. See Figure 6.1 for $m=3$.

It is most convenient to organize the control bits here as being from top to bottom in each of the $2m-1$ stages. The control bits in the first stage are f; the control bits in the last stage are ℓ; the control bits in the remaining stages interleave the control bits for M_0 and M_1.

The first control bit f_0 is 0 by Theorem 5.2, so the corresponding conditional swap can be eliminated from the Beneš network, producing a slightly smaller permutation network. Applying the same observation recursively saves $2^{m-1}-1$

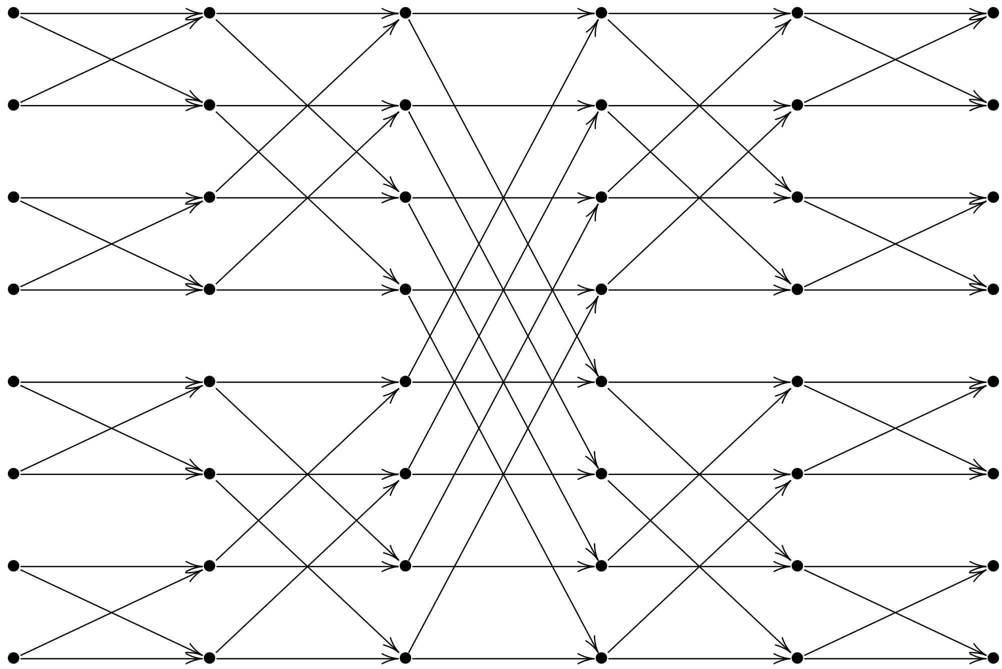

Fig. 6.1. Data flow in an in-place Beneš network for permuting 8 inputs.

control bits, reducing the total number of control bits from $(2m-1)2^{m-1}$ to $(2m-2)2^{m-1}+1=(m-1)2^m+1$.

6.2. Data layout. The long-distance conditional swaps in Figure 6.1 are more expensive than nearby conditional swaps in models of computation that assign costs to distance. One can reduce this cost with a **shuffled Beneš network**, as illustrated in Figure 6.3 for $m=3$. This network shuffles the positions of intermediate results so that each stage applies its control bits to conditionally swap $(a_0,a_1),(a_2,a_3),\ldots$. Beware that the top-to-bottom order of control bits in Figure 6.1 corresponds to a different order of control bits in Figure 6.3 for the middle stages.

The shuffling here still involves long-distance data movement, even if not as much as before. One obtains somewhat better scalability with "cache-oblivious" algorithms, and further improvements with "cache-tuned" algorithms. Some long-distance data movement is unavoidable given that each input position could end up at any output position.

6.4. Inversion and mirroring. Some applications naturally follow the pattern above: after constructing a permutation π they naturally move position $\pi(i)$ in the input to position i in the output. Other applications, after constructing a permutation π, naturally move position i in the input to position $\pi(i)$ in the output.

Mirroring the control bits from left to right in the Beneš network replaces the permutation with its inverse. This means exchanging stage 0 with stage $2m-2$, exchanging stage 1 with stage $2m-3$, etc., while preserving the order of control bits from top to bottom in each stage.

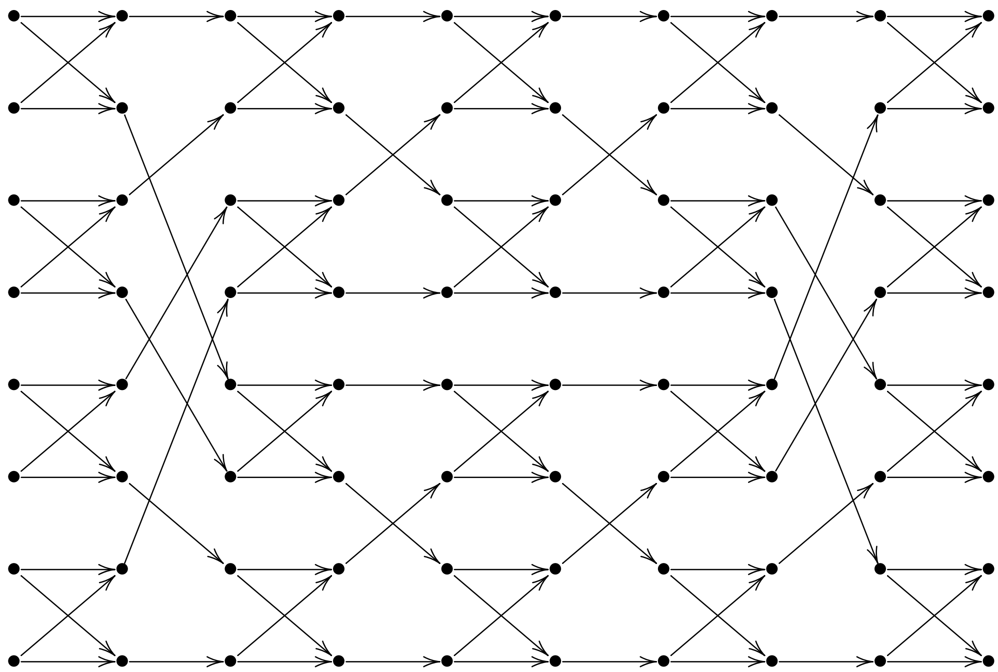

Fig. 6.3. Data flow in a shuffled Beneš network for permuting 8 inputs.

One can instead replace π with π^{-1} in Section 5 to compute control bits for π^{-1}. This usually produces different control bits from mirroring; note that for $b > 0$ there are multiple sequences of control bits computing the same permutation. Concretely, Section 5 always produces $f_0 = 0$ for π (and not necessarily $\ell_0 = 0$) by Theorem 5.2, so mirroring always produces $\ell_0 = 0$ for π^{-1} (and not necessarily $f_0 = 0$), whereas replacing π with π^{-1} produces $f_0 = 0$ for π^{-1}. One can replace π with π^{-1} *and* mirror to produce $\ell_0 = 0$ for π if desired.

Often the literature defines control bits as being for π^{-1} where this paper defines the same control bits as being for π. Perhaps there is an equal split of applications in both directions, but it would be useful to synchronize notation, and there is an advantage of setting up definitions so that applying a permutation network for π to a standard input produces the list of π values as output.

6.5. Network history. Beneš presented the shuffled Beneš network in 1965 [**7**, pages 123–125]—except for merging the middle three stages out of $2m-1$, which was an improvement in some of the cost metrics that Beneš considered.

Waksman pointed out in 1968 [**24**], modulo mirroring, that one can always take $f_0 = 0$, and so on recursively, eliminating $2^{m-1} - 1$ conditional swaps from the network. The simplest proof is as follows: if $f_0 = 1$, flip all f and ℓ bits and exchange M_0 with M_1.

Beneš's 1962 paper [**6**] had already considered the following more general structure, calling it a **three-stage Clos network**:

- First stage: The n inputs are distributed across r networks $F_0, \ldots, F_{r-1}$, each mapping n/r inputs to μ outputs.

- Middle stage: The $r\mu$ outputs are then distributed across μ middle networks $M_0, \ldots, M_{\mu-1}$, each mapping r inputs to r outputs.
- Last stage: The $r\mu$ outputs from the subpermutations are then distributed across r networks $L_0, \ldots, L_{r-1}$, each mapping μ inputs to n/r outputs.

Specializing to $\mu = 2$ and $r = n/2$, and then recursively decomposing the middle stage in the same way, produces the Beneš network.

What Clos's 1953 paper [12] had actually proposed was networks with $\mu = 2(n/r) - 1$. These networks were developed in the context of telephone calls, connecting caller $\pi(i)$ to callee i by setting control bits as physical switches. Working in place, which requires $\mu = n/r$, is meaningless in this context, whereas there are good reasons to take μ larger than n/r: Clos observed that, with $\mu = 2(n/r) - 1$, a new call could be routed immediately through the network. In other words, one can decide on control bits compatible with $(\pi(0), 0)$, and then decide on control bits compatible with $(\pi(1), 1)$, and then decide on control bits compatible with $(\pi(2), 2)$, and so on, never having to change a previous decision.

Followup papers observed that with smaller μ one could still handle a new call $(\pi(i), i)$ by changing a limited number of previous decisions. In particular, Beneš's paper [6] (using the names "m" and "n" and "r" for μ and n/r and r here) showed that at most $r - 1$ existing calls need to be rearranged, assuming $\mu \geq n/r$. For example, at most 1 existing call needs to be rearranged if $r = 2$ and $\mu = n/2$. Beneš credited Paull [21] with the special case $\mu = r = \sqrt{n}$.

A weaker statement is simply that one can handle a new call—never mind the number of previous decisions that need to be changed. In other words, for every permutation π of n items, there exists a sequence of control bits for π for the three-stage Clos network (and hence for the Beneš network, when n is a power of 2). Beneš credited Slepian and Duguid with proving this fact, as an application of Hall's matching theorem. Beneš also stated that the proof in [6] was "not depending on the Hall combinatorial result".

Would specializing Hall's proof [14] to this context produce the proof from [6]? Would it produce the proof of Theorem 4.4? I have not found such comparisons in the literature.

6.6. Fast sequential algorithms. Stone introduced an algorithm that, given any permutation π, uses $\Theta(m2^m)$ RAM operations to compute Beneš-network control bits for π, Perhaps one can extract an algorithm running at the same speed from Hall's theorem (or Beneš's theorem), and simplify the algorithm down to Stone's algorithm, but Stone's algorithm was introduced as an algorithm directly handling the power-of-2 case, without reference to other constructions.

Stone's algorithm first chooses $f_0 = 0$; equivalently, it chooses $F(0) = 0$. It then visits each other x in the cycle of 0 under $\overline{\pi}$, choosing each $f_{\lfloor x/2 \rfloor}$ so that $F(x) \equiv 0 \pmod 2$. Note that these control bits match Theorem 5.3 for each x in the cycle of 0, i.e., each x where $c(x) = 0$; the only other possibility is to flip all of these control bits.

Obtaining each x as a value of $\overline{\pi}$ naturally passes through computing $\pi^{-1}(x) \oplus 1$. Stone's algorithm chooses $\ell_{\lfloor \pi^{-1}(x)/2 \rfloor}$ so that $L(\pi^{-1}(x)) \equiv F(x) \pmod 2$, as in Theorem 5.4.

This is not the complete algorithm yet, unless the cycle was long enough to set all the control bits. The complete algorithm to compute f and ℓ is as follows:

- Search for the smallest even c for which $f_{c/2}$ has not yet been chosen. If there is no such c, stop.
- For each x in the cycle of c under $\overline{\pi}$, choose $f_{\lfloor x/2 \rfloor}$ so that $F(x) \equiv 0 \pmod 2$. These control bits again match Theorem 5.3, since c is even.
- Also, for each such x, choose $\ell_{\lfloor \pi^{-1}(x)/2 \rfloor}$ so that $L(\pi^{-1}(x)) \equiv F(x) \pmod 2$, matching Theorem 5.4.

This takes $\Theta(2^m)$ RAM operations. Handling M_0 and M_1 in the same way recursively takes, in total, $\Theta(m2^m)$ RAM operations to set all $\Theta(m2^m)$ control bits in the complete Beneš network.

Stone's algorithm was first published as part of Waksman's paper [**24**, page 161, "constructive proof"], Waksman's paper credits Stone only in a footnote, and says little about Beneš's results, so it is not surprising that Waksman is often miscredited for (1) the algorithm and (2) the fact that the Beneš network can apply any permutation. The same algorithm is also sometimes miscredited to Opferman and Tsao-Wu [**20**], who presented this "looping algorithm" as if it were new, despite citing Waksman's paper from three years earlier.

6.7. Fast parallel algorithms. Parallel versions of Stone's algorithm were introduced by Lev–Pippenger–Valiant [**18**], independently Nassimi–Sahni [**19**], and later Lee–Liew [**17**].

Each of these algorithms takes $\Theta(m)$ steps to find control bits $f_0, \dots, f_{2^{m-1}-1}$ and $\ell_0, \dots, \ell_{2^{m-1}-1}$ for the first and last stages of a 2^m-input Beneš network, and thus $\Theta(m^2)$ steps to find all control bits for a Beneš network, where each step involves $\Theta(2^m)$ RAM operations. Overall there are $\Theta(m^2 2^m)$ RAM operations. This is a factor $\Theta(m)$ more than Stone's algorithm, but now the RAM operations are parallel accesses to distinct locations, and can thus be implemented by a fast parallel sorting network.

The main idea in these parallel algorithms is Theorem 2.4, which computes $\operatorname{cyclemin} \pi$ with a number of stages logarithmic in the maximum cycle length: the first stage computes (c_1, π^2), the second stage computes (c_2, π^4), the third stage computes (c_3, π^8), etc. These $\Theta(m)$ stages, each with $\Theta(2^m)$ RAM operations, explain the extra $\Theta(m)$ factor in the number of RAM operations.

The Nassimi–Sahni [**19**] algorithm applies this idea to compute $\operatorname{cyclemin} \overline{\pi}$ in parallel, and then sets the same control bits as Stone. The Lev–Pippenger–Valiant [**18**] algorithm is obscured by unnecessary layers of abstraction but seems to compute $\operatorname{cyclemin} \sqrt{\pi}$, where $\sqrt{\pi}$ is the double-size permutation that maps $(x, 0)$ to $(\pi^{-1}(x \oplus 1), 1)$ and maps $(x, 1)$ to $(\pi(x \oplus 1), 0)$. Sometimes the minimum of a cycle of $\sqrt{\pi}$ has the form $(x, 1)$ rather than $(x, 0)$, and this can change $x \bmod 2$, producing control bits for the cycle that are the opposite of Stone's control bits.

The Lee–Liew [**17**] algorithm was introduced as being more general than [**19**], for example in allowing the input to be a partial permutation. The Lee–Liew algorithm works with edges $\lfloor x/2 \rfloor \mapsto \lfloor \overline{\pi}(x)/2 \rfloor$ between just $n/2$ vertices

$\{0, 1, \ldots, n/2 - 1\}$, compared to the permutation of n elements in Nassimi–Sahni and the permutation of $2n$ elements in Lev–Pippenger–Valiant; and follows edges only to smaller positions, removing the factor 2 redundancy between x and $x \oplus 1$. However, there can now be two edges out of one vertex, complicating and slowing down the composition of edges in the inner loop.

6.8. Using sorting inside fast parallel algorithms. As noted in Section 1, starting with the list of values of a function a and the list of values of π, one can compute the list of values of $a \circ \pi^{-1}$ by sorting the pairs $(\pi(x), a(x))$. One can, as a special case, compute π^{-1} with one sorting step, and then compute $a \circ \pi$ with another sorting step.

To compose two functions a, b with the same π, one can compose the pair (a, b) with π. In other words, first sort the pairs $(\pi(x), x)$, obtaining pairs $(x, \pi^{-1}(x))$; and then sort the triples $(\pi^{-1}(x), a(x), b(x))$, obtaining triples $(x, a(\pi(x)), b(\pi(x)))$. This is typically somewhat more efficient than sorting $(\pi^{-1}(x), a(x))$ and separately sorting $(\pi^{-1}(x), b(x))$.

Recall that the cyclemin π computation in Theorems 2.4 and 2.5 starts with the identity map c_0 and computes successively

$$\begin{aligned} c_1 &= \min\{c_0, c_0 \circ \pi_0\}, \\ c_2 &= \min\{c_1, c_1 \circ \pi_1\}, \\ c_3 &= \min\{c_2, c_2 \circ \pi_2\}, \end{aligned}$$

etc., where π_i means π^{2^i}. Each iteration then composes (c_{i-1}, π_{i-1}) with π_{i-1}, except that the first iteration can simply compute $c_1 = \min\{c_0, \pi_0\}$, and the last iteration can skip its π_i computation.

One can reduce latency by computing π_i^{-1} in preparation for the next iteration as $\pi_{i-1}^{-1} \circ \pi_{i-1}^{-1}$ at the same time as computing π_i, rather than computing π_i^{-1} by inverting π_i. Each iteration, except near the beginning and end, then involves, in parallel,

- sorting pairs $(\pi_{i-1}(x), \pi_{i-1}^{-1}(x))$ to obtain pairs $(x, \pi_i^{-1}(x))$, and
- sorting triples $(\pi_{i-1}^{-1}(x), c_{i-1}(x), \pi_{i-1}(x))$ to obtain $(x, c_{i-1}(\pi_{i-1}(x)), \pi_i(x))$,

or optionally splitting the sorting of triples into sorting pairs $(\pi_{i-1}^{-1}(x), c_{i-1}(x))$ and sorting pairs $(\pi_{i-1}^{-1}(x), \pi_{i-1}(x))$. By Theorem 2.5, i iterations handle cycle lengths as large as 2^i.

This is not asymptotically optimal. One can, for example, compute $\pi^{-1}, \pi^2, \pi^{-3}, \pi^5, \pi^{-8}, \pi^{13}, \ldots$ with just one sorting step per iteration, while redefining c_i accordingly. Each iteration then gains a factor asymptotically $(1 + \sqrt{5})/2 \approx 1.61803$ instead of 2, so the number of iterations grows by a factor asymptotically $\log_{(1+\sqrt{5})/2} 2 \approx 1.44042$, but each iteration costs only one step of sorting triples, without the extra step of sorting pairs. This makes each iteration $1.5\times$ cheaper in the worst case that sorting triples costs twice as much as sorting pairs. The advantage is larger if sorting triples is cheaper than this.

7 Software

Three Python functions are shown in Figure 7.1: `permutation`, `composeinv`, and `controlbits`.

The `permutation` function takes as input a list of $(2m-1)2^{m-1}$ bits, where m is a positive integer. The function outputs a permutation π of $\{0,1,\ldots,2^m-1\}$, represented as a list whose entries are $\pi(0),\pi(1),\ldots,\pi(2^m-1)$. The list starts as $0,1,\ldots,2^m-1$, and then each input bit controls a conditional swap of list entries at a power-of-2 distance. The structure of the function, simply exchanging pairs of list entries without otherwise inspecting the entries, implies that generalizing $0,1,\ldots,2^m-1$ to an arbitrary list $a_0,a_1,\ldots,a_{2^m-1}$ at the beginning would produce $a_{\pi(0)},a_{\pi(1)},\ldots,a_{\pi(2^m-1)}$ as output. This generalization can be taken as a formal definition of an in-place Beneš network.

The `controlbits` function takes as input a permutation π of $\{0,1,\ldots,2^m-1\}$. The function outputs a list of $(2m-1)2^{m-1}$ bits. The correctness goal for this function is for `permutation` $\circ$ `controlbits` to be the identity map, returning the same permutation π that was provided as input. Beware that `controlbits` $\circ$ `permutation` is not the identity map: as noted earlier, each permutation is represented by multiple sequences of bits.

The `composeinv` function is a subroutine in `controlbits`, computing $c\circ\pi^{-1}$ given a function c on $\{0,1,\ldots,2^m-1\}$ and a permutation π of $\{0,1,\ldots,2^m-1\}$. The idea, as explained in Section 1, is to sort pairs $(\pi(x),c(x))$ to see pairs $(x,c(\pi^{-1}(x)))$. Outside the calls to `composeinv`, all array accesses in `controlbits` are local, implementable in hardware with short wire lengths independent of m.

The computation inside `controlbits` for $m\ge 2$ matches the definitions in Section 5. First c is computed as $\operatorname{cyclemin}\overline{\pi}$ where $\overline{\pi}=\operatorname{XbackXforth}\pi$; this is the bulk of the computation, and is explained in more detail below. Then $f=\operatorname{firstcontrol}\pi$, $F=\operatorname{Xif} f$, $\ell=\operatorname{lastcontrol}\pi$, $L=\operatorname{Xif}\ell$, $M=F\circ\pi\circ L$ are computed as in Definition 5.1. Then `subM[0]` and `subM[1]` are set to the permutations M_0,M_1 of $\{0,1,\ldots,2^{m-1}-1\}$; control bits `subz[0]` and `subz[1]` for M_0 and M_1 respectively are set by recursive calls to `controlbits`; these control bits are then interleaved, and sandwiched between `f` and `l`.

The initial computation of $\operatorname{XbackXforth}\pi$ works as follows. First p is set to $x\mapsto\pi(x\oplus 1)$, and q is set to $x\mapsto\pi(x)\oplus 1$; then p is overwritten with $p\circ q^{-1}=\overline{\pi}$. Meanwhile q is overwritten with $q\circ p^{-1}=\overline{\pi}^{-1}$, and an invariant $q=p^{-1}$ holds after each of the subsequent computations.

At the same time π^{-1} is computed. One could delay this until the last iteration of the main loop without any latency impact, simplifying space allocation for π^{-1}. Alternatively, one can view a computation of π^{-1} at this moment as a computation of the initial p^{-1}, since $\pi^{-1}(x)=p^{-1}(x)\oplus 1$. The computation of p^{-1}, in turn, can be merged with the initial $q\circ p^{-1}$ computation into an $(\text{identity},q)\circ p^{-1}$ computation as explained in Section 6.8. Figure 7.1 omits this refinement for readability.

The main loop computes $\operatorname{cyclemin}\overline{\pi}$ as follows. By Theorem 4.3, each cycle of $\overline{\pi}$ has length at most 2^{m-1}. Define c_i as in Theorem 2.4, substituting $\overline{\pi}$ for

```
def permutation(c):
  m = 1
  while (2*m-1)<<(m-1) < len(c): m += 1
  assert (2*m-1)<<(m-1) == len(c)

  n = 1<<m
  pi = list(range(n))
  for i in range(2*m-1):
    gap = 1<<min(i,2*m-2-i)
    for j in range(n//2):
      if c[i*n//2+j]:
        pos = (j%gap)+2*gap*(j//gap)
        pi[pos],pi[pos+gap] = pi[pos+gap],pi[pos]

  return pi

def composeinv(c,pi):
  return [y for x,y in sorted(zip(pi,c))]

def controlbits(pi):
  n = len(pi)
  m = 1
  while 1<<m < n: m += 1
  assert 1<<m == n

  if m == 1: return [pi[0]]
  p = [pi[x^1] for x in range(n)]
  q = [pi[x]^1 for x in range(n)]

  piinv = composeinv(range(n),pi)
  p,q = composeinv(p,q),composeinv(q,p)

  c = [min(x,p[x]) for x in range(n)]
  p,q = composeinv(p,q),composeinv(q,p)
  for i in range(1,m-1):
    cp,p,q = composeinv(c,q),composeinv(p,q),composeinv(q,p)
    c = [min(c[x],cp[x]) for x in range(n)]

  f = [c[2*j]%2 for j in range(n//2)]
  F = [x^f[x//2] for x in range(n)]
  Fpi = composeinv(F,piinv)
  l = [Fpi[2*k]%2 for k in range(n//2)]
  L = [y^l[y//2] for y in range(n)]
  M = composeinv(Fpi,L)
  subM = [[M[2*j+e]//2 for j in range(n//2)] for e in range(2)]
  subz = map(controlbits,subM)
  z = [s for s0s1 in zip(*subz) for s in s0s1]
  return f+z+l
```

Fig. 7.1. Python functions to compute the permutation for an in-place Beneš network given control bits, and to compute control bits given a permutation. See text for details.

π; then $c_{m-1} = \text{cyclemin}\,\overline{\pi}$ by Theorem 2.5. The loop invariant is that c, p, q are $c_i, \overline{\pi}^{2^i}, \overline{\pi}^{-2^i}$. Then $c \circ p$ is computed as $c \circ q^{-1}$, and $c_{i+1}(x)$ is computed as $\min\{c(x), c(p(x))\}$.

The $i = 0$ iteration is handled before the loop, so as to skip the redundant composition $c_0 \circ \overline{\pi} = \overline{\pi}$ as in Section 6.8; the loop thus starts with $i = 1$. Some further refinements from Section 6.8 are, like the initial p^{-1} refinement above, omitted for readability: merging `composeinv(c,q)` and `composeinv(p,q)` into `composeinv(zip(c,p),q)`; skipping the computations of p and q in the final iteration $i = m - 2$; skipping the computation of p in the previous iteration $i = m - 3$.

7.2. Small values of m. Figure 7.1 special-cases $m = 1$, taking $\pi(0)$ as the output control bit. This is adequate for the base case of the recursion, but one can also special-case larger values of m for efficiency.

For $m = 2$, the following sequence of 11 bit operations (7 XORs and 4 ANDs) computes the same control bits $c_0, c_1, c_2, c_3, c_4, c_5$ as above given the high and low bits of $\pi(0), \pi(1), \pi(2)$:

$$\begin{aligned}
c_0 &= 0\\
t_0 &= \pi(1)_0 \oplus \pi(0)_0\\
t_1 &= \pi(1)_1 \oplus \pi(0)_1\\
t_2 &= t_1 \cdot t_0\\
c_1 &= t_2 \oplus t_1\\
t_3 &= c_1 \cdot \pi(0)_1\\
t_4 &= c_1 \cdot \pi(2)_1\\
c_4 &= t_3 \oplus \pi(0)_0\\
c_5 &= t_4 \oplus \pi(2)_0\\
t_5 &= c_4 \cdot t_1\\
c_2 &= t_5 \oplus \pi(0)_1\\
c_3 &= t_5 \oplus \pi(1)_1
\end{aligned}$$

This sequence was found by a straightforward combinatorial search through instruction sequences ("superoptimization"), applied to all 24 permutations in bitsliced form. The search did not find any shorter sequences, even when ORs were allowed, even when the bits of $\pi(3)$ were considered as two further inputs. (The low bits of $\pi(0), \pi(1), \pi(2), \pi(3)$ sum to 0 modulo 2, as do the high bits, so it is not surprising that $\pi(3)$ is unnecessary.) Perhaps the flexibility of taking a different sequence of control bits for the same permutation would allow a shorter sequence.

Could similar techniques handle larger values of m? The search for $m = 2$ took on the scale of a day on one core, so much more searching is possible; skipping ORs would have saved time; searching for each output bit separately would have produced somewhat worse results but would have saved more time. For $m = 3$ one can easily enumerate all 40320 permutations; for $m = 4$ one can enumerate all 20922789888000 permutations; to handle larger m one would need to be able to verify an instruction sequence in much less time than trying the sequence on all permutations. Presumably superoptimization using a random selection of 64 permutations would have very few false positives—although it is obviously important to verify the results for all permutations. The main question for $m = 3$ and $m = 4$ is whether each output bit has a short enough sequence to be found by a search.

7.3. Space allocation. One can squeeze all temporary variables into a 2^m-entry array, each entry having $4m$ bits. The inner loop sorts $(p, \text{identity})$ in half of the space, obtaining $(\text{identity}, p^{-1})$, while remembering (p, c) in the other half; then sorts (p^{-1}, p), obtaining $(\text{identity}, p^2)$, while remembering (p^{-1}, c); then sorts (p^{-1}, c), obtaining $(\text{identity}, c \circ p)$, while remembering (p^2, c). For comparison, the input permutation π is stored in an untouched array having 2^m entries, each entry having m bits.

Other steps in the computation are easy to fit into the same space. This space can also be used for the recursion and for the output. This amount of space is also compatible with merging two pair-sorting steps into a triple-sorting step: sort (p^{-1}, p, c) in 3/4 of the space while remembering c in the other 1/4.

Fast sorting software typically expects input words having 8 bits, 16 bits, 32 bits, etc.—although it is not clear that this is essential for performance; most time is spent sorting relatively small subarrays, and the rest of the array can be stored in packed form. To reuse word-oriented sorting software inside control-bit computation, one can expand $2m$ or $3m$ bits to full words. For example, the fast 32-bit sorting software from [**8**] handles triples for m as large as 10, and pairs for m as large as 16.

References

[1] — (no editor), *AFIPS conference proceedings, volume 32: 1968 Spring Joint Computer Conference, Reston, Virginia*, Thompson Book Company, 1968. See [5].

[2] — (no editor), *Proceedings of the 25th annual ACM symposium on theory of computing*, Association for Computing Machinery, New York, 1983. See [4].

[3] — (no editor), *Proceedings IEEE INFOCOM '96, the Conference on Computer Communications, fifteenth annual joint conference of the IEEE Computer and Communications Societies, networking the next generation, San Francisco, CA, USA, March 24–28, 1996*, IEEE Computer Society, 1996. ISBN 0-8186-7292-7. See [17].

[4] Miklós Ajtai, János Komlós, Endre Szemerédi, *An $O(n \log n)$ sorting network*, in [**2**] (1983), 1–9. Citations in this document: §1.

[5] Kenneth E. Batcher, *Sorting networks and their applications*, in [**1**] (1968), 307–314. URL: `https://www.cs.kent.edu/~batcher/conf.html`. Citations in this document: §1.

[6] Václav E. Beneš, *On rearrangeable three-stage connecting networks*, Bell System Technical Journal **41** (1962), 1481–1492. Citations in this document: §6.5, §6.5, §6.5, §6.5.

[7] Václav E. Beneš, *Mathematical theory of connecting networks and telephone traffic*, Academic Press, 1965. ISBN 0-12-087550-0. Citations in this document: §6.5.

[8] Daniel J. Bernstein, *djbsort* (2018). URL: https://sorting.cr.yp.to. Citations in this document: §1, §7.3.

[9] Daniel J. Bernstein, Tung Chou, Tanja Lange, Ingo von Maurich, Rafael Misoczki, Ruben Niederhagen, Edoardo Persichetti, Christiane Peters, Peter Schwabe, Nicolas Sendrier, Jakub Szefer, Wen Wang, *Classic McEliece: conservative code-based cryptography*, "Supporting Documentation" (2019). URL: https://csrc.nist.gov/projects/post-quantum-cryptography/round-2-submissions. Citations in this document: §1.

[10] Daniel J. Bernstein, Tung Chou, Peter Schwabe, *McBits: fast constant-time code-based cryptography*, in CHES 2013 [**11**] (2013), 250–272. URL: https://cr.yp.to/papers.html#mcbits. Citations in this document: §1.

[11] Guido Bertoni, Jean-Sébastien Coron (editors), *Cryptographic hardware and embedded systems—CHES 2013—15th international workshop, Santa Barbara, CA, USA, August 20–23, 2013, proceedings*, Lecture Notes in Computer Science, 8086, Springer, 2013. ISBN 978-3-642-40348-4. See [10].

[12] Charles Clos, *A study of non-blocking switching networks*, Bell System Technical Journal **32** (1953), 406–424. Citations in this document: §6.5.

[13] Michael T. Goodrich, *Zig-zag sort: a simple deterministic data-oblivious sorting algorithm running in $O(n \log n)$ time*, in STOC 2014 [**22**] (2014), 684–693. URL: https://arxiv.org/abs/1403.2777. Citations in this document: §1.

[14] Philip Hall, *On representatives of subsets*, Journal of the London Mathematical Society **1** (1935), 26–30. Citations in this document: §6.5.

[15] John Harrison, *HOL Light: A tutorial introduction*, in FMCAD 1996˜[**23**] (1996), 265–269. Citations in this document: §A.

[16] Donald E. Knuth, *The art of computer programming, volume III: sorting and searching* (1973). ISBN 0-201-03803-X. Citations in this document: §1.

[17] Tony T. Lee, Soung-Yue Liew, *Parallel routing algorithms in Benes-Clos networks*, in INFOCOM 1996 [**3**] (1996), 279–286. Citations in this document: §6.7, §6.7.

[18] Gavriela Freund Lev, Nicholas Pippenger, Leslie G. Valiant, *A fast parallel algorithm for routing in permutation networks*, IEEE Transactions on Computers **C-30** (1981), 93–100. Citations in this document: §1, §6.7, §6.7.

[19] David Nassimi, Sartaj Sahni, *Parallel algorithms to set up the Benes permutation network*, IEEE Transactions on Computers **C-31** (1982), 148–154. Citations in this document: §1, §6.7, §6.7, §6.7.

[20] David C. Opferman, Nelson T. Tsao-Wu, *On a class of rearrangeable switching networks, part I: control algorithm*, Bell System Technical Journal **50** (1971), 1579–1600. Citations in this document: §6.6.

[21] M. C. Paull, *Reswitching of connection networks*, Bell System Technical Journal **41** (1962), 833–855. Citations in this document: §6.5.

[22] David B. Shmoys (editor), *Symposium on theory of computing, STOC 2014, New York, NY, USA, May 31–June 03, 2014*, ACM, 2014. ISBN 978-1-4503-2710-7. See [13].

[23] Mandayam K. Srivas, Albert John Camilleri (editors), *Formal methods in computer-aided design, first international conference, FMCAD '96, Palo Alto, California, USA, November 6–8, 1996, proceedings*, Lecture Notes in Computer Science, 1166, Springer, 1996. ISBN 3-540-61937-2. See [15].

[24] Abraham Waksman, *A permutation network*, Journal of the ACM **15** (1968), 159–163. Citations in this document: §1, §6.5, §6.6.

A Computer verification of theorems

There are several tools available for computers to verify—and, to some extent, construct—mathematical proofs. A reviewer still has to check each theorem statement used, along with the underlying definitions, but can skip checking the proofs in favor of checking the verifier.

The following proofs are written in the HOL Light [15] language and have passed the August 2020 version of the HOL Light verifier. This verifier is often described as small. Beware that the verifier depends on a large stack of software for a general-purpose programming language (OCaml), and on the hardware running that software; one hopes that bugs in these tools do not interfere with the correctness of the verifier.

The time for the verifier to check all of the proofs below is under 1 minute on a 1.7GHz CPU core, and the human time to write the proofs below was an unrecorded fraction of a two-week period. Surely these times can be reduced—I'm still learning the tools available for constructing HOL Light proofs—but both times are fast enough to not be problematic.

A.1. Permutations in HOL Light. For each set S, permutations of S can be identified with permutations of the whole universe that leave the rest of the universe alone. For example, for each n, a permutation π of $\{0, 1, \ldots, n-1\}$ corresponds to a permutation π' of $\{0, 1, \ldots\}$ that fixes $\{n, n+1, \ldots\}$: define $\pi'(x) = \pi(x)$ for $x < n$ and $\pi'(x) = x$ for $x \geq n$; conversely, define π as the restriction of π'. In the HOL Light language, it is easy to talk about functions such as π' from $\{0, 1, \ldots\}$ to $\{0, 1, \ldots\}$, whereas it is considerably harder to talk about functions such as π from $\{0, 1, \ldots, n-1\}$ to $\{0, 1, \ldots, n-1\}$, since the language is type-theoretic without "dependent types". The theorems below thus talk about π' rather than π. This puts a small burden on the reviewer to translate π' theorems into π theorems.

Modulo this translation, `XbackXforth_cyclemin_xor1` below is Theorem 4.4. As stated, it is more general than Theorem 4.4, since it does not require π to fix $\{2b, 2b+1, \ldots\}$.

Similarly, a function from $\{0, 1, \ldots, n-1\}$ to $\{0, 1\}$ is most easily expressed as HOL Light as a function from $\{0, 1, \ldots\}$ to $\{\text{False}, \text{True}\}$ that is supported on $\{0, 1, \ldots, n-1\}$, i.e., takes $\{n, n+1, \ldots\}$ to False. The string $s \in \{0, 1\}^b$ in Theorem 3.3 is then translated to a function $s : \{0, 1, \ldots\} \to \{\text{False}, \text{True}\}$ supported on $\{0, 1, \ldots, b-1\}$. If one identifies subsets of T with functions $T \to \{\text{False}, \text{True}\}$, as is conventional in HOL Light, then saying that s is supported on $\{0, 1, \ldots, b-1\}$ is the same as saying that s is a subset of $\{0, 1, \ldots, b-1\}$.

`Xif` s below is defined as a function from $\{0, 1, \ldots\}$ to $\{0, 1, \ldots\}$ for each subset s of $\{0, 1, \ldots\}$; `xif_involution` below says that this is an involution of $\{0, 1, \ldots\}$. If s is a subset of $\{0, 1, \ldots, b-1\}$ then `xif_fixesge` below says that `Xif` s fixes all $x \geq 2b$, so `Xif` s can be viewed as an involution of $\{0, 1, \ldots, 2b-1\}$, exactly the involution Xif s in Theorem 3.3.

To simplify spot checks that what is being proven is what is desired, Table A.2 is an index mapping further definitions and theorem statements in this appendix

`xor1` n	$n \oplus 1$ for $n \in \{0, 1, 2, \ldots\}$
`involution` π	$\pi \circ \pi =$ identity
`bijection` π	π is a bijection
`inverse` π	π^{-1}
π `pow_num` n	π^n for $n \in \{0, 1, 2, \ldots\}$
π `pow_int` k	π^k for $k \in \mathbf{Z}$
`cycle` π	$x \mapsto \{\pi^k(x) : k \in \mathbf{Z}\}$; Definition 2.1
`cyclemin` π	$x \mapsto \min\{\pi^k(x) : k \in \mathbf{Z}\}$; Definition 2.3
`part2powcycle`	$(i, \pi, x) \mapsto \{\pi^k(x) : 0 \le k < 2^i\}$
`fastcyclemin`	$(i, \pi, x) \mapsto c_i(x)$ in Theorem 2.4
π `fixesge` n	$\pi(x) = x$ for each $x \ge n$
`Xif` s	$x \mapsto x \oplus s_{\lfloor x/2 \rfloor}$; Definition 3.2
`XbackXforth` π	$x \mapsto \pi(\pi^{-1}(x \oplus 1) \oplus 1)$; Definition 4.1
`firstcontrol` π	firstcontrol π in Definition 5.1
`lastcontrol` π	lastcontrol π in Definition 5.1
`middleperm` π	middleperm π in Definition 5.1
`cycle_shift`	Theorem 2.2
`fastcyclemin_part2powcycle`	Theorem 2.4
`fastcyclemin_works`	Theorem 2.5
`xor1_lteven`	$x \oplus 1 \in \{0, 1, \ldots, 2b - 1\}$ in Theorem 3.1
`xor1_div2`	$\lfloor (x \oplus 1)/2 \rfloor = \lfloor x/2 \rfloor$ in Theorem 3.1
`xif_fixesge`	Xif s fixes $\ge 2b$ in Theorem 3.3
`xif_involution`	Xif s is an involution in Theorem 3.3
`XbackXforth_fixesge`	XbackXforth π fixes $\ge 2b$ in Theorem 4.2
`XbackXforth_bijection`	XbackXforth π is a permutation in Theorem 4.2
`XbackXforth_powpow_xor1`	$\overline{\pi}^k(x_k \oplus 1) = x \oplus 1$ in Theorem 4.3
`XbackXforth_powpow_avoids_xor1`	$x_j \ne x_k \oplus 1$ in Theorem 4.3
`XbackXforth_cycle_size`	$\#\{x_k : k \in \mathbf{Z}\} \le b$ in Theorem 4.3
`XbackXforth_cyclemin_xor1`	Theorem 4.4
`firstcontrol_subsetrange`	$f \in \{0, 1\}^b$ in Definition 5.1
`lastcontrol_subsetrange`	$\ell \in \{0, 1\}^b$ in Definition 5.1
`firstcontrol_0`	Theorem 5.2
`firstcontrol_parity`	Theorem 5.3
`lastcontrol_first`	Theorem 5.4
`middleperm_fixesge`	M fixes $\ge 2b$ in Theorem 5.5
`middleperm_bijection`	M is a permutation in Theorem 5.5
`middleperm_parity`	$M(x) \equiv x \pmod 2$ in Theorem 5.5

Table A.2. Index of definitions and theorem statements.

to definitions and theorem statements in the main body of the paper. To simplify re-running the verifier, all of the following HOL Light proofs also appear at https://cr.yp.to/2020/controlbits-20200923.ml for easy download.

```
(* ----- Definitions *)

let xor1 = new_definition
  `xor1 (n:num) = if EVEN n then n+1 else n-1`;;
  (* "num" in HOL Light is the type {0,1,2,...}. *)
  (* Could have simply said "xor1 n = ...";
     type-checker will take n:num since EVEN wants num. *)
  (* Warning: If n:num is 0 then n-1 is 0 in HOL Light.
     This definition uses n-1 only for positive n. *)

let involution = new_definition
  `involution (f:A->A) <=> !x. f(f x) = x`;;
  (* "!x" in HOL Light is "for all x of this type".
     Type-checker automatically chooses type of x here
     as A since x is an f input (and an f output). *)

(* Simpler special case of built-in SURJ. *)
let surjection = new_definition
  `surjection (f:A->B) <=> !y. ?x. f x = y`;;
  (* "?x" in HOL Light is "there exists x of this type". *)

(* Simpler special case of built-in INJ. *)
let injection = new_definition
  `injection (f:A->B) <=> !x y. f x = f y ==> x = y`;;

(* Simpler special case of built-in BIJ. *)
let bijection = new_definition
  `bijection (f:A->B) <=> surjection f /\ injection f`;;
  (* /\ in HOL Light is "and". *)

(* Same as "inverse" in permutations.ml. *)
let inverse = new_definition
  `inverse (f:A->B) = \y. @x. f x = y`;;
  (* \y.f(y) in HOL Light is lambda notation
     for the function y|->f(y). *)
  (* @x.P(x) in HOL Light is some x such that P(x).
     Warning: returns an arbitrary element if P(x) is never true. *)

parse_as_infix("pow_num",(24,"left"));;
let pow_num = define
  `((p:A->A) pow_num n) =
    if n = 0 then (I:A->A) else (p o (p pow_num(n-1)))`;;
  (* "I" in HOL Light is the identity map. *)
  (* "define" is like "new_definition" but allows recursion. *)
```

```
parse_as_infix("pow_int",(24,"left"));;
let pow_int = new_definition
  `(p:A->A) pow_int (k:int)
    = if (&0 <= k) then (p pow_num(num_of_int k))
      else (inverse p pow_num(num_of_int(--k)))`;;
  (* "&" in HOL Light maps naturals to integers.
     Or to real numbers! Important to declare k as an int. *)
  (* "--k" in HOL Light is the negative of k. *)
  (* "num_of_int k" is @n. &n = k.
     Warning: returns an arbitrary natural if k is negative. *)

let cycle = new_definition
  `cycle (p:A->A) x = { y | ?k:int. y = (p pow_int k)(x) }`;;

let cyclemin = new_definition
  `cyclemin p x = (minimal) (cycle p x)`;;
  (* HOL Light defines minimal only for sets of naturals,
     so this definition forces p:num->num. *)

let partcycle = new_definition
  `partcycle (i:num) (p:A->A) x =
   { y | ?k:num. k < i /\ y = (p pow_num k)(x) }`;;

let part2powcycle = new_definition
  `part2powcycle (i:num) (p:A->A) x = partcycle (2 EXP i) p x`;;

let fastcyclemin = define
  `fastcyclemin i p x =
   if i = 0 then x else
   MIN (fastcyclemin (i-1) p x)
       (fastcyclemin (i-1) p ((p pow_num (2 EXP (i-1))) x))`;;

parse_as_infix("vanishesifonbothsidesof",(12,"right"));;
let vanishesifonbothsidesof = new_definition
  `(p:A->A) vanishesifonbothsidesof (q:A->A) <=> p o q o p = q`;;

let range = new_definition
  `range n = {m:num | m < n}`;;

parse_as_infix("subsetrange",(12,"right"));;
let subsetrange = new_definition
  `(s:num->bool) subsetrange n <=> !x. s x ==> x < n`;;
```

```
(* Part of "permutes" from permutations.ml. *)
(* There isn’t a type {0,1,...,n-1} in HOL Light;
   easiest way to describe a permutation of {0,1,...,n-1}
   is as a permutation of the integers that fixes {n,n+1,...}. *)
parse_as_infix("fixesge",(12,"right"));;
let fixesge = new_definition
  `(p:num->num) fixesge n <=> !x. x >= n ==> p(x) = x`;;

let xif = new_definition
  `Xif s n = if s(n DIV 2) then xor1 n else n`;;

let XbackXforth = new_definition
  `XbackXforth p = p o xor1 o inverse p o xor1`;;

let firstcontrol = new_definition
  `firstcontrol p j = ODD(cyclemin(XbackXforth p)(2*j))`;;

let lastcontrol = new_definition
  `lastcontrol p k = ODD(Xif (firstcontrol p) (p(2*k)))`;;

let middleperm = new_definition
  `middleperm p = Xif(firstcontrol p) o p o Xif(lastcontrol p)`;;

(* ----- involution *)

let involution_I = prove(
  `involution (I:A->A)`,
  MESON_TAC[I_DEF;involution]);;

let involution_o = prove(
  `!f:A->A. involution f <=> f o f = I`,
  REWRITE_TAC[FUN_EQ_THM;o_THM;I_THM;involution]);;

(* ----- surjection *)

let surjection_ifinvolution = prove(
  `!p:A->A. involution p ==> surjection p`,
  REWRITE_TAC[involution;surjection] THEN
  REPEAT STRIP_TAC THEN
  EXISTS_TAC `(p:A->A)(y)` THEN
  ASM_MESON_TAC[]);;

let surjection_I = prove(
  `surjection (I:A->A)`,
  MESON_TAC[involution_I;surjection_ifinvolution]);;
```

```
let surjection_composes = prove(
  `!p:A->B q:B->C. surjection p /\ surjection q ==> surjection(q o p)`,
  REWRITE_TAC[surjection;o_THM] THEN
  MESON_TAC[]);;

(* ----- injection *)

let injection_ifinvolution = prove(
  `!(p:A->A). involution p ==> injection p`,
  REWRITE_TAC[involution;injection] THEN
  MESON_TAC[]);;

let injection_I = prove(
  `injection (I:A->A)`,
  MESON_TAC[involution_I;injection_ifinvolution]);;

let injection_composes = prove(
  `!p:A->B q:B->C. injection p /\ injection q ==> injection(q o p)`,
  REWRITE_TAC[injection] THEN
  REWRITE_TAC[o_THM] THEN
  MESON_TAC[]);;

(* ----- bijection *)

let bijection_ifinvolution = prove(
  `!p:A->A. involution p ==> bijection p`,
  MESON_TAC[surjection_ifinvolution;injection_ifinvolution;bijection]);;

let bijection_I = prove(
  `bijection (I:A->A)`,
  MESON_TAC[involution_I;bijection_ifinvolution]);;

let bijection_composes = prove(
  `!p:A->B q:B->C. bijection p /\ bijection q ==> bijection(q o p)`,
  REWRITE_TAC[bijection] THEN
  MESON_TAC[surjection_composes;injection_composes]);;

let bijection_inverseisinjection = prove(
  `!p:A->B. bijection p ==> injection(inverse p)`,
  REWRITE_TAC[bijection;surjection;injection;inverse] THEN
  MESON_TAC[]);;

let bijection_inverseissurjection = prove(
  `!p:A->B. bijection p ==> surjection(inverse p)`,
  REWRITE_TAC[bijection;surjection;injection;inverse] THEN
  MESON_TAC[]);;
```

```
let bijection_inverseisbijection = prove(
  `!p:A->B. bijection p ==> bijection(inverse p)`,
  MESON_TAC[bijection_inverseisinjection;
    bijection_inverseissurjection;bijection]);;

let bijection_inverse_composition1 = prove(
  `!p:A->B x. bijection p ==> p((inverse p)x) = x`,
  REWRITE_TAC[bijection;surjection;injection;inverse] THEN
  MESON_TAC[]);;

let bijection_inverse_composition2 = prove(
  `!p:A->B x. bijection p ==> (inverse p)(p x) = x`,
  REWRITE_TAC[bijection;surjection;injection;inverse] THEN
  MESON_TAC[]);;

let bijection_inverse_composition3 = prove(
  `!p:A->B. bijection p ==> !x. (p o inverse p)x = x`,
  MESON_TAC[bijection_inverse_composition1;o_THM]);;

let bijection_inverse_composition4 = prove(
  `!p:A->B. bijection p ==> !x. (inverse p o p)x = x`,
  MESON_TAC[bijection_inverse_composition2;o_THM]);;

let bijection_inverse_composition5 = prove(
  `!p:A->B. bijection p ==> p o inverse p = I`,
  MESON_TAC[bijection_inverse_composition3;I_THM;EQ_EXT]);;

let bijection_inverse_composition6 = prove(
  `!p:A->B. bijection p ==> inverse p o p = I`,
  MESON_TAC[bijection_inverse_composition4;I_THM;EQ_EXT]);;

let bijection_frominverse = prove(
  `!p:A->B q:B->A. q o p = I /\ p o q = I ==>
                   bijection p /\ q = inverse p`,
  REWRITE_TAC[bijection;surjection;injection;inverse] THEN
  MESON_TAC[I_THM;o_THM;EQ_EXT]);;

let bijection_matchinverse = prove(
  `!p:A->B. bijection p ==> !q:B->A. p o q = I ==> q = inverse p`,
  REWRITE_TAC[bijection;surjection;injection;inverse] THEN
  MESON_TAC[I_THM;o_THM;EQ_EXT]);;

let bijection_matchinverse2 = prove(
  `!p:A->B. bijection p ==> !q:B->A. q o p = I ==> q = inverse p`,
  REWRITE_TAC[bijection;surjection;injection;inverse] THEN
  MESON_TAC[I_THM;o_THM;EQ_EXT]);;
```

```
let bijection_inverse_inverse = prove(
  `!p:A->B. bijection p ==> inverse(inverse p) = p`,
  MESON_TAC[bijection_inverse_composition5;
    bijection_inverse_composition6;bijection_frominverse]);;

(* ----- EVEN, ODD *)

let suc_isadd1 = prove(
  `!n. SUC n = n+1`,
  MESON_TAC[ONE;ADD_SUC;ADD_0]);;

let nonzero_sub1add1 = prove(
  `!n. ~(n = 0) ==> (n-1)+1 = n`,
  MESON_TAC[LTE_CASES;LE_1;SUB_ADD]);;

let nonzero_subsuc = prove(
  `!n. ~(n = 0) ==> SUC(n-1) = n`,
  MESON_TAC[nonzero_sub1add1;suc_isadd1]);;

let nonzero_pre_le_suc = prove(
  `!n. ~(n = 0) ==> n-1 <= SUC(n-1)`,
  MESON_TAC[LE_REFL;LE]);;

let pre_le_ifnonzero = prove(
  `!n. ~(n = 0) ==> n-1 <= n`,
  MESON_TAC[nonzero_pre_le_suc;nonzero_subsuc]);;

let pre_le_ifzero = prove(
  `!n. n = 0 ==> n-1 <= n`,
  CONV_TAC ARITH_RULE);;

let pre_le = prove(
  `!n. n-1 <= n`,
  MESON_TAC[pre_le_ifzero;pre_le_ifnonzero]);;

let even_odd_add1 = prove(
  `!n. EVEN n ==> ODD(n+1)`,
  REWRITE_TAC[ODD_ADD;ODD;EVEN;ONE;NOT_ODD]);;

let even_add1_odd = prove(
  `!n. EVEN(n+1) ==> ODD n`,
  MESON_TAC[ONE;NOT_EVEN;EVEN_ADD;EVEN;ODD]);;

let odd_even_sub1 = prove(
  `!n. ODD n ==> EVEN(n-1)`,
  MESON_TAC[EVEN_SUB;ODD;EVEN;NOT_EVEN;ONE;LTE_CASES]);;
```

```
let odd_sub1add1 = prove(
  `!n. ODD n ==> (n-1)+1 = n`,
  MESON_TAC[nonzero_sub1add1;ODD]);;

let odd_sub1distinct = prove(
  `!n. ODD n ==> ~(n-1 = n)`,
  GEN_TAC THEN
  ASSUME_TAC(ARITH_RULE `~(n-1 = (n-1)+1)`) THEN
  ASM_MESON_TAC[odd_sub1add1]);;

let odd_isdoubleplus1 = prove(
  `!n. ODD n ==> ?m. n = 2*m+1`,
  MESON_TAC[ODD_EXISTS;suc_isadd1]);;

let sandwich_suc = prove(
  `!m n. m < n /\ n <= SUC m ==> SUC m = n`,
  MESON_TAC[LE_ANTISYM;LE_SUC_LT]);;

let sandwich_add1 = prove(
  `!m n. m < n /\ n <= m+1 ==> m+1 = n`,
  MESON_TAC[sandwich_suc;suc_isadd1]);;

let sandwich_even_add1_odd = prove(
  `!n b. n < 2*b /\ 2*b <= n+1 ==> ODD n`,
  MESON_TAC[EVEN_DOUBLE;sandwich_add1;even_add1_odd;NOT_EVEN]);;

(* ----- range *)

let empty_ifnever = prove(
  `!s:A->bool. (!x. ~(x IN s)) ==> s = EMPTY`,
  REWRITE_TAC[IN;FUN_EQ_THM;EMPTY;IN_ELIM_THM]);;

let range_in = prove(
  `!k n. k IN range n <=> k < n`,
  MESON_TAC[range;IN_ELIM_THM]);;

let range_in_flip = prove(
  `!k n. k < n <=> k IN range n`,
  MESON_TAC[range_in]);;

let range_0_lemma = prove(
  `!x. ~(x IN range 0)`,
  REWRITE_TAC[range;IN_ELIM_THM] THEN
  ARITH_TAC);;

let range_0 = prove(
  `range 0 = EMPTY`,
  MESON_TAC[empty_ifnever;range_0_lemma]);;
```

```
let range_suc = prove(
  `!n. range(SUC n) = n INSERT range n`,
  REWRITE_TAC[range;INSERT;FUN_EQ_THM;IN_ELIM_THM;LT] THEN
  MESON_TAC[]);;

let range_plus1 = prove(
  `!n. range(n+1) = n INSERT range n`,
  MESON_TAC[range_suc;suc_isadd1]);;

let range_finite = prove(
  `!n. FINITE(range n)`,
  INDUCT_TAC THEN
  ASM_MESON_TAC[FINITE_RULES;range_0;range_suc]);;

let range_card_0 = prove(
  `CARD(range 0) = 0`,
  MESON_TAC[range_0;CARD_CLAUSES]);;

let range_card_lemma = prove(
  `!n. ~(n IN range n)`,
  REWRITE_TAC[range;IN;IN_ELIM_THM] THEN
  ARITH_TAC);;

let range_card = prove(
  `!n. CARD(range n) = n`,
  INDUCT_TAC THEN
  REWRITE_TAC[range_card_0] THEN
  REWRITE_TAC[range_suc] THEN
  ASM_MESON_TAC[CARD_CLAUSES;range_finite;range_card_lemma]);;

(* ----- xor1 *)

let xor1_double = prove(
  `!n. xor1 (2*n) = 2*n+1`,
  ASM_SIMP_TAC[xor1;EVEN_DOUBLE]);;

let xor1_distinct = prove(
  `!n. ~(xor1 n = n)`,
  REWRITE_TAC[xor1] THEN
  GEN_TAC THEN
  ASSUME_TAC(ARITH_RULE `~(n+1 = n)`) THEN
  ASSUME_TAC(SPEC `n:num` odd_sub1distinct) THEN
  DISJ_CASES_TAC (SPEC `n:num` EVEN_OR_ODD) THEN
  ASM_SIMP_TAC[] THEN
  ASM_MESON_TAC[NOT_ODD]);;

let xor1_oneven_isadd1 = prove(
  `!n. EVEN n ==> xor1 n = n+1`,
  MESON_TAC[xor1]);;
```

```
let xor1_oneven_isodd = prove(
  `!n. EVEN n ==> ODD(xor1 n)`,
  MESON_TAC[even_odd_add1;xor1]);;

let xor1_onodd_issub1 = prove(
  `!n. ODD n ==> xor1 n = n-1`,
  MESON_TAC[NOT_EVEN;xor1]);;

let xor1_onodd_iseven = prove(
  `!n. ODD n ==> EVEN(xor1 n)`,
  MESON_TAC[xor1_onodd_issub1;odd_even_sub1]);;

let xor1_parity = prove(
  `!n. ODD(xor1 n) <=> ~ODD n`,
  MESON_TAC[xor1_onodd_iseven;xor1_oneven_isodd;NOT_EVEN]);;

let xor1xor1_ifeven = prove(
  `!n. EVEN n ==> xor1(xor1 n) = n`,
  MESON_TAC[xor1_oneven_isodd;xor1_onodd_issub1;
    xor1_oneven_isadd1;ARITH_RULE `(n+1)-1 = n`]);;

let xor1xor1_ifodd = prove(
  `!n. ODD n ==> xor1(xor1 n) = n`,
  MESON_TAC[xor1_onodd_iseven;xor1_onodd_issub1;
    xor1_oneven_isadd1;odd_sub1add1]);;

let xor1xor1 = prove(
  `!n. xor1(xor1 n) = n`,
  MESON_TAC[xor1xor1_ifodd;xor1xor1_ifeven;EVEN_OR_ODD]);;

let xor1_involution_o = prove(
  `xor1 o xor1 = I`,
  REWRITE_TAC[I_DEF;o_DEF;xor1xor1]);;

let xor1_involution = prove(
  `involution xor1`,
  MESON_TAC[xor1xor1;involution]);;

let xor1_lteven = prove(
  `!n b. n < 2*b ==> xor1 n < 2*b`,
  REWRITE_TAC[xor1] THEN
  ASM_CASES_TAC(`EVEN n`) THEN
  MESON_TAC[LTE_CASES;sandwich_even_add1_odd;
    pre_le;NOT_EVEN;LET_TRANS]);;

let xor1_geeven = prove(
  `!n b. n >= 2*b ==> xor1 n >= 2*b`,
  MESON_TAC[xor1xor1;xor1_lteven;NOT_LT;GE]);;
```

```
let xor1_lemma101 = prove(
  `!m n. EVEN m ==> EVEN n ==> ~(n = m + 1)`,
  MESON_TAC[even_odd_add1;NOT_ODD]);;

let xor1_lemma102 = prove(
  `!m n. EVEN m ==> EVEN n ==> n <= m + 1 ==> n <= m`,
  REPEAT STRIP_TAC THEN
  ASSUME_TAC(SPEC `m:num` suc_isadd1) THEN
  ASM_MESON_TAC[xor1_lemma101;LE_LT;LT_SUC_LE]);;

let xor1_lemma103 = prove(
  `!m n. EVEN m ==> EVEN n ==> m < xor1 n /\ n <= xor1 m ==> m = n`,
  REWRITE_TAC[xor1] THEN
  REPEAT STRIP_TAC THEN
  ASM_SIMP_TAC[] THEN
  ASSUME_TAC(ARITH_RULE `n <= m /\ m < n + 1 ==> m = n`) THEN
  ASM_MESON_TAC[xor1_lemma102]);;

let xor1_lemma104 = prove(
  `!m n. ~EVEN m ==> ~EVEN n ==> m < xor1 n ==> xor1 m < n`,
  REPEAT GEN_TAC THEN DISCH_TAC THEN DISCH_TAC THEN
  REWRITE_TAC[xor1] THEN
  ASM_SIMP_TAC[] THEN
  ARITH_TAC);;

let xor1_lemma105 = prove(
  `!m n. ~EVEN m ==> EVEN n ==> m < xor1 n /\ n <= xor1 m ==> m = n`,
  REPEAT GEN_TAC THEN DISCH_TAC THEN DISCH_TAC THEN
  REWRITE_TAC[xor1] THEN
  ASM_SIMP_TAC[] THEN
  ARITH_TAC);;

let xor1_lemma106 = prove(
  `!m n. EVEN m ==> ~EVEN n ==> m < xor1 n ==> xor1 m < n`,
  REPEAT GEN_TAC THEN DISCH_TAC THEN DISCH_TAC THEN
  REWRITE_TAC[xor1] THEN
  ASM_SIMP_TAC[] THEN
  ARITH_TAC);;

let xor1_lemma107 = prove(
  `!m n. m < xor1 n /\ n <= xor1 m ==> m = n`,
  REPEAT GEN_TAC THEN
  DISJ_CASES_TAC (TAUT `EVEN m \/ ~EVEN m`) THEN
  DISJ_CASES_TAC (TAUT `EVEN n \/ ~EVEN n`) THEN
  ASM_MESON_TAC[NOT_LT;xor1_lemma103;xor1_lemma104;
                     xor1_lemma105;xor1_lemma106]);;
```

```
let xor1_sandwich = prove(
  `!m n. m <= xor1 n /\ n <= xor1 m /\ ~(m = n) ==> m = xor1 n`,
  MESON_TAC[xor1_lemma107;LE_LT]);;

(* ----- DIV 2 *)

let double_div2 = prove(
  `!n. (2*n) DIV 2 = n`,
  MESON_TAC[DIV_MULT;ARITH_RULE `~(2 = 0)`]);;

let double_flip_plus1_div2 = prove(
  `!n. (n*2+1) DIV 2 = n`,
  MESON_TAC[DIVMOD_UNIQ;ARITH_RULE `(1 < 2)`]);;

let double_plus1_div2 = prove(
  `!n. (2*n+1) DIV 2 = n`,
  MESON_TAC[double_flip_plus1_div2;ARITH_RULE `2*n = n*2`]);;

let double_orplus1_div2 = prove(
  `!n. (2*n+1) DIV 2 = (2*n) DIV 2`,
  MESON_TAC[double_div2;double_plus1_div2]);;

let double_plus1_orminus1_div2 = prove(
  `!n. (2*n+1) DIV 2 = ((2*n+1)-1) DIV 2`,
  MESON_TAC[double_orplus1_div2;ARITH_RULE `(2*n+1)-1 = 2*n`]);;

let double_suc_orminus1_div2 = prove(
  `!n. SUC(2*n) DIV 2 = (SUC(2*n)-1) DIV 2`,
  MESON_TAC[suc_isadd1;double_plus1_orminus1_div2]);;

let xor1_div2 = prove(
  `!n. xor1 n DIV 2 = n DIV 2`,
  REWRITE_TAC[xor1] THEN
  GEN_TAC THEN
  ASM_CASES_TAC `EVEN n` THEN
  ASM_SIMP_TAC[] THENL
  [
    ASSUME_TAC(SPEC `n:num` EVEN_EXISTS) THEN
    ASM_MESON_TAC[double_orplus1_div2]
  ;
    ASSUME_TAC(SPEC `n:num` EVEN_EXISTS_LEMMA) THEN
    ASM_MESON_TAC[double_suc_orminus1_div2]
  ]);;

let div2_lt_doubling = prove(
  `!x b. x DIV 2 < b ==> x < 2*b`,
  MESON_TAC[RDIV_LT_EQ;ARITH_RULE `~(2 = 0)`]);;
```

```
let div2_lt_doubling_ge = prove(
  `!x b. x DIV 2 < b ==> ~(x >= 2*b)`,
  MESON_TAC[div2_lt_doubling;NOT_LT;GE]);;

(* ----- pow_num: natural powers of a function from A to A *)

let pow_num_eval = prove(
  `!p:A->A. !n. ~(n = 0) ==> !x. (p pow_num n)x = p((p pow_num(n-1))x)`,
  ASM_MESON_TAC[pow_num;o_DEF]);;

let pow_num_0 = prove(
  `!p:A->A. p pow_num 0 = I`,
  REWRITE_TAC[pow_num]);;

let pow_num_0_eval = prove(
  `!p:A->A. !x. (p pow_num 0)x = x`,
  REWRITE_TAC[pow_num_0;I_DEF]);;

let pow_num_11 = prove(
  `!p:A->A. p pow_num (1-1) = I`,
  REWRITE_TAC[ARITH_RULE `1-1=0`;pow_num_0]);;

let pow_num_111 = prove(
  `!p:A->A. p o (p pow_num (1-1)) = p`,
  REWRITE_TAC[pow_num_11;I_O_ID]);;

let pow_num_1 = prove(
  `!p:A->A. p pow_num 1 = p`,
  ASM_MESON_TAC[pow_num;pow_num_111;ARITH_RULE `~(1 = 0)`]);;

let pow_num_1_eval = prove(
  `!p:A->A. !x. (p pow_num 1) x = p x`,
  REWRITE_TAC[pow_num_1]);;

let pow_num_n11 = prove(
  `!n p:A->A. p pow_num(n+1) = p o (p pow_num((n+1)-1))`,
  ASM_MESON_TAC[pow_num;ARITH_RULE `~(n+1 = 0)`]);;

let pow_num_plus1 = prove(
  `!n p:A->A. p pow_num(n+1) = p o (p pow_num n)`,
  ASM_MESON_TAC[pow_num_n11;ARITH_RULE `(n+1)-1 = n`]);;

let pow_num_plus1_eval = prove(
  `!n p:A->A. !x. (p pow_num(n+1)) x = p((p pow_num n) x)`,
  ASM_MESON_TAC[pow_num_plus1;o_DEF]);;
```

```
let pow_num_plus_flip = prove(
  `!m n p:A->A. p pow_num(n+m) = (p pow_num m) o (p pow_num n)`,
  INDUCT_TAC THENL [
    REWRITE_TAC[pow_num_0;I_O_ID;ARITH_RULE `n+0 = n`]
  ;
    REWRITE_TAC[suc_isadd1;ARITH_RULE `n+m+1 = (n+m)+1`;
                pow_num_plus1] THEN
    ASM_MESON_TAC[o_ASSOC]
  ]);;

let pow_num_plus_flip_eval = prove(
  `!m n p:A->A. !x. (p pow_num(n+m))x = (p pow_num m)((p pow_num n) x)`,
  ASM_MESON_TAC[pow_num_plus_flip;o_DEF]);;

let pow_num_plus = prove(
  `!m n p:A->A. p pow_num(m+n) = (p pow_num m) o (p pow_num n)`,
  MESON_TAC[pow_num_plus_flip;ARITH_RULE `n+m = m+n`]);;

let pow_num_plus_eval = prove(
  `!m n p:A->A. !x. (p pow_num(m+n))x = (p pow_num m)((p pow_num n) x)`,
  MESON_TAC[pow_num_plus_flip_eval;ARITH_RULE `n+m = m+n`]);;

let pow_num_plus1_right = prove(
  `!n p:A->A. p pow_num(n+1) = (p pow_num n) o p`,
  MESON_TAC[pow_num_plus;pow_num_1]);;

let pow_num_plus1_right_eval = prove(
  `!n p:A->A. !x. (p pow_num(n+1))x = (p pow_num n)(p x)`,
  MESON_TAC[pow_num_plus1_right;o_DEF]);;

let pow_num_bijection = prove(
  `!p:A->A. bijection p ==> !n. bijection (p pow_num n)`,
  GEN_TAC THEN DISCH_TAC THEN
  INDUCT_TAC THENL
  [ REWRITE_TAC[pow_num_0;bijection_I]
  ; REWRITE_TAC[suc_isadd1] THEN
    ASM_MESON_TAC[pow_num_plus1;bijection_composes]
  ]);;

let pow_num_plus1_bijection = prove(
  `!n p:A->A. bijection p ==>
              inverse p o (p pow_num(n+1)) = p pow_num n`,
  REWRITE_TAC[pow_num_plus1] THEN
  MESON_TAC[bijection_inverse_composition6;o_ASSOC;I_O_ID]);;
```

```
let pow_num_plus1_inverse = prove(
  `!n p:A->A. bijection p ==>
              p o ((inverse p) pow_num(n+1)) = (inverse p) pow_num n`,
  MESON_TAC[pow_num_plus1_bijection;
    bijection_inverseisbijection;bijection_inverse_inverse]);;

(* ----- pow_int: integer powers of a permutation of A *)

let pow_int_num = prove(
  `!p:A->A n. p pow_num n = p pow_int &n`,
  REWRITE_TAC[pow_int] THEN
  REPEAT GEN_TAC THEN
  SIMP_TAC[INT_ARITH `((&0):int) <= &n`] THEN
  MESON_TAC[NUM_OF_INT_OF_NUM] THEN
  ASM_SIMP_TAC[]);;

let pow_int_0 = prove(
  `!p:A->A. p pow_int &0 = I`,
  REWRITE_TAC[pow_int;NUM_OF_INT_OF_NUM;pow_num_0] THEN
  SIMP_TAC[INT_ARITH `&0:int <= &0`]);;

let pow_int_1 = prove(
  `!p:A->A. p pow_int &1 = p`,
  REWRITE_TAC[pow_int;NUM_OF_INT_OF_NUM;pow_num_1] THEN
  SIMP_TAC[INT_ARITH `&0:int <= &1`]);;

let pow_int_bijection_nonnegative = prove(
  `!p:A->A. bijection p ==> !k:int. &0 <= k ==> bijection (p pow_int k)`,
  REPEAT STRIP_TAC THEN
  REWRITE_TAC[pow_int] THEN
  ASM_SIMP_TAC[] THEN
  ASM_MESON_TAC[pow_num_bijection]);;

let pow_int_bijection_negative = prove(
  `!p:A->A. !k:int. bijection p /\ k < &0 ==> bijection (p pow_int k)`,
  REPEAT STRIP_TAC THEN
  REWRITE_TAC[pow_int] THEN
  ASM_MESON_TAC[pow_num_bijection;bijection_inverseisbijection]);;

let pow_int_bijection = prove(
  `!p:A->A. !k:int. bijection p ==> bijection (p pow_int k)`,
  REPEAT STRIP_TAC THEN
  DISJ_CASES_TAC(INT_ARITH `k:int < &0 \/ &0 <= k`) THENL
  [ ASM_MESON_TAC[pow_int_bijection_negative]
  ; ASM_MESON_TAC[pow_int_bijection_nonnegative]
  ]);;
```

```
let pow_int_cancel = prove(
  `!p:A->A k x y. bijection p ==>
                 (p pow_int k) x = (p pow_int k) y ==> x = y`,
  REPEAT GEN_TAC THEN
  ASSUME_TAC(SPEC `p:A->A` pow_int_bijection) THEN
  ASM_MESON_TAC[bijection;injection]);;

let pow_int_plus1_nonnegative = prove(
  `!p:A->A. !k:int. &0 <= k ==> p pow_int(k + &1) = p o (p pow_int k)`,
  REWRITE_TAC[pow_int] THEN
  REPEAT STRIP_TAC THEN
  ASSUME_TAC(INT_ARITH `&0 <= k:int ==> &0 <= k + &1`) THEN
  ASSUME_TAC(SPEC `p:A->A`(SPEC `num_of_int(k:int)` pow_num_plus1)) THEN
  ASSUME_TAC(INT_ARITH `(&0:int) <= &1`) THEN
  ASSUME_TAC(SPEC `1` NUM_OF_INT_OF_NUM) THEN
  ASSUME_TAC(SPECL[`k:int`;`&1:int`]NUM_OF_INT_ADD) THEN
  SUBGOAL_THEN `&0 <= (k:int) + &1` ASSUME_TAC THEN
  ASM_SIMP_TAC[] THEN
  ASM_MESON_TAC[]);;

let int_negative_num_towards0 = prove(
  `!k:int. k < &0 ==> num_of_int(--(k + &1)) + 1 = num_of_int(--k)`,
  REPEAT STRIP_TAC THEN
  ASSUME_TAC(SPECL[`--((k:int) + &1)`;`&1:int`]NUM_OF_INT_ADD) THEN
  ASSUME_TAC(INT_ARITH `k:int < &0 ==> &0 <= --(k + &1)`) THEN
  ASSUME_TAC(INT_ARITH `--((k:int) + &1) + &1 = --k`) THEN
  ASSUME_TAC(INT_ARITH `(&0:int) <= &1`) THEN
  ASSUME_TAC(SPEC `1` NUM_OF_INT_OF_NUM) THEN
  ASM_MESON_TAC[]);;
```

```
let pow_int_plus1_negative = prove(
  `!p:A->A. bijection p ==> !k:int. k < &0 ==>
            p pow_int(k + &1) = p o (p pow_int k)`,
  REWRITE_TAC[pow_int] THEN
  REPEAT STRIP_TAC THEN
  DISJ_CASES_TAC(INT_ARITH `-- &1 <= k:int \/ k < -- &1`) THENL
  [
    ASSUME_TAC(INT_ARITH `-- &1 <= (k:int) /\ k < &0 ==> k = -- &1`) THEN
    ASSUME_TAC(INT_ARITH `k:int = -- &1 ==> ~(&0 <= k)`) THEN
    ASSUME_TAC(INT_ARITH `k:int = -- &1 ==> &0 <= k + &1`) THEN
    ASSUME_TAC(INT_ARITH `-- -- &1:int = &1`) THEN
    ASSUME_TAC(INT_ARITH `-- (&1:int) + &1 = &0`) THEN
    ASSUME_TAC(SPEC `1` NUM_OF_INT_OF_NUM) THEN
    ASSUME_TAC(SPEC `0` NUM_OF_INT_OF_NUM) THEN
    ASSUME_TAC(SPEC `p:A->A` pow_num_0) THEN
    ASSUME_TAC(SPEC `inverse(p:A->A)` pow_num_1) THEN
    ASM_SIMP_TAC[] THEN
    ASM_MESON_TAC[bijection_inverse_composition5]
  ;
    ASSUME_TAC(INT_ARITH `k:int < -- &1 ==> ~(&0 <= k)`) THEN
    ASSUME_TAC(INT_ARITH `k:int < -- &1 ==> ~(&0 <= k + &1)`) THEN
    ASSUME_TAC(SPEC `k:int` int_negative_num_towards0) THEN
    ASSUME_TAC(SPECL[`num_of_int(--((k:int) + &1))`;
                     `p:A->A`]pow_num_plus1_inverse) THEN
    ASM_MESON_TAC[]
  ]);;

let pow_int_plus1 = prove(
  `!p:A->A. bijection p ==> !k:int.
            p pow_int(k + &1) = p o (p pow_int k)`,
  REPEAT STRIP_TAC THEN
  DISJ_CASES_TAC(INT_ARITH `k:int < &0 \/ &0 <= k`)
  THENL[ASM_MESON_TAC[pow_int_plus1_negative];
    ASM_MESON_TAC[pow_int_plus1_nonnegative]]);;
```

```
let pow_int_plusnum = prove(
  `!p:A->A. bijection p ==> !m. !k:int.
            p pow_int(k + &m) = (p pow_num m) o (p pow_int k)`,
  GEN_TAC THEN DISCH_TAC THEN
  INDUCT_TAC THENL
  [
    ASSUME_TAC(INT_ARITH `(k:int) + &0 = k`) THEN
    ASM_SIMP_TAC[] THEN
    REWRITE_TAC[pow_num_0;I_O_ID]
  ;
    GEN_TAC THEN
    REWRITE_TAC[suc_isadd1;pow_num_plus1] THEN
    ASSUME_TAC(INT_ARITH `(k:int) + (&m + &1) = (k + &m) + &1`) THEN
    ASSUME_TAC(SPECL[`m:num`;`1:num`]INT_OF_NUM_ADD) THEN
    ASM_MESON_TAC[o_ASSOC;pow_int_plus1]
  ]);;

let pow_int_plus_nonnegative = prove(
  `!p:A->A. bijection p ==> !j:int. &0 <= j ==>
    !k:int. p pow_int(k + j) = (p pow_int j) o (p pow_int k)`,
  REPEAT STRIP_TAC THEN
  ASSUME_TAC(SPEC `j:int` NUM_OF_INT) THEN
  ASM_MESON_TAC[pow_int_plusnum;pow_int]);;

let pow_int_inverse_lemma1 = prove(
  `!p:A->A. bijection p ==> !j:int. &0 <= j ==>
    p pow_int(--j + j) = (p pow_int j) o (p pow_int --j)`,
  MESON_TAC[pow_int_plus_nonnegative]);;

let pow_int_inverse_lemma2 = prove(
  `!p:A->A. bijection p ==> !j:int. &0 <= j ==>
    p pow_int &0 = (p pow_int j) o (p pow_int --j)`,
  REPEAT STRIP_TAC THEN
  ASSUME_TAC(INT_ARITH `--j + (j:int) = &0`) THEN
  ASM_MESON_TAC[pow_int_inverse_lemma1]);;

let pow_int_inverse_lemma3 = prove(
  `!p:A->A. bijection p ==> !j:int. &0 <= j ==>
    (p pow_int j) o (p pow_int --j) = I`,
  MESON_TAC[pow_int_inverse_lemma2;pow_int_0]);;

let pow_int_inverse_nonnegative = prove(
  `!p:A->A. bijection p ==> !j:int. &0 <= j ==>
    p pow_int --j = inverse(p pow_int j)`,
  MESON_TAC[pow_int_bijection;bijection_matchinverse;
                pow_int_inverse_lemma3]);;
```

```
let pow_int_inverse_negative = prove(
  `!p:A->A. bijection p ==> !j:int. j < &0 ==>
    p pow_int --j = inverse(p pow_int j)`,
  REPEAT STRIP_TAC THEN
  ASSUME_TAC(ARITH_RULE `(j:int) < &0 ==> &0 <= --j`) THEN
  ASSUME_TAC(ARITH_RULE `(j:int) = -- --j`) THEN
  ASM_MESON_TAC[pow_int_bijection;
    bijection_inverse_inverse;pow_int_inverse_nonnegative]);;

let pow_int_inverse = prove(
  `!p:A->A. !j:int. bijection p ==>
    p pow_int --j = inverse(p pow_int j)`,
  REPEAT STRIP_TAC THEN
  DISJ_CASES_TAC(INT_ARITH `j:int < &0 \/ &0 <= j`) THENL
  [ ASM_MESON_TAC[pow_int_inverse_negative]
  ; ASM_MESON_TAC[pow_int_inverse_nonnegative]
  ]);;

let pow_int_plus_lemma1 = prove(
  `!p:A->A. bijection p ==> !j:int. j < &0 ==>
    !k:int. p pow_int(k + --j) = (p pow_int --j) o (p pow_int k)`,
  REPEAT STRIP_TAC THEN
  ASSUME_TAC(ARITH_RULE `(j:int) < &0 ==> &0 <= --j`) THEN
  ASM_MESON_TAC[pow_int_plus_nonnegative]);;

let pow_int_plus_lemma2 = prove(
  `!p:A->A. !j:int. !k:int. bijection p /\ j < &0 ==>
    p pow_int(k - j) = (p pow_int --j) o (p pow_int k)`,
  REPEAT STRIP_TAC THEN
  REWRITE_TAC[ARITH_RULE `(k:int)-(j:int) = k + --j`] THEN
  ASM_MESON_TAC[pow_int_plus_lemma1]);;

let pow_int_plus_lemma3 = prove(
  `!p:A->A. !j:int. !k:int. bijection p /\ j < &0 ==>
    p pow_int(k-j) = inverse(p pow_int j) o (p pow_int k)`,
  MESON_TAC[pow_int_plus_lemma2;pow_int_inverse;pow_int_bijection]);;

let pow_int_plus_lemma4 = prove(
  `!p:A->A j:int k:int. bijection p /\ j < &0 ==>
    (p pow_int j) o (p pow_int(k-j)) = p pow_int k`,
  REPEAT STRIP_TAC THEN
  ASSUME_TAC(SPECL[`p:A->A`;`j:int`;`k:int`]pow_int_plus_lemma3) THEN
  ASSUME_TAC(SPECL[`p:A->A`;`j:int`]pow_int_bijection) THEN
  ASSUME_TAC(ISPEC `(p:A->A) pow_int (j:int)`
                   bijection_inverse_composition5) THEN
  ASM_MESON_TAC[o_ASSOC;I_O_ID]);;
```

```
let pow_int_plus_negative = prove(
  `!p:A->A. bijection p ==> !j:int. j < &0 ==>
    !k:int. (p pow_int j) o (p pow_int k) = p pow_int (k+j)`,
  REPEAT STRIP_TAC THEN
  ASSUME_TAC(SPECL[`p:A->A`;`j:int`;`(k:int)+j`]
                  pow_int_plus_lemma4) THEN
  ASSUME_TAC(ARITH_RULE `((k:int)+(j:int))-j=k`) THEN
  ASM_MESON_TAC[]);;

let pow_int_plus_swap = prove(
  `!p:A->A j:int k:int. bijection p ==>
    p pow_int(k+j) = (p pow_int j) o (p pow_int k)`,
  REPEAT STRIP_TAC THEN
  DISJ_CASES_TAC(INT_ARITH `j:int < &0 \/ &0 <= j`) THENL
  [ ASM_MESON_TAC[pow_int_plus_negative]
  ; ASM_MESON_TAC[pow_int_plus_nonnegative]
  ]);;

let pow_int_plus = prove(
  `!p:A->A j:int k:int. bijection p ==>
    p pow_int(j+k) = (p pow_int j) o (p pow_int k)`,
  REPEAT STRIP_TAC THEN
  ASSUME_TAC(ARITH_RULE `(k:int)+(j:int)=j+k`) THEN
  ASM_MESON_TAC[pow_int_plus_swap]);;

let pow_int_plus_eval = prove(
  `!p:A->A j:int k:int x. bijection p ==>
    (p pow_int(j+k))x = (p pow_int j)((p pow_int k)x)`,
  MESON_TAC[pow_int_plus;o_THM]);;

let pow_int_cancel2 = prove(
  `!p:A->A x j k. bijection p ==>
   (p pow_int j) x = (p pow_int k) x ==> x = (p pow_int (k-j)) x`,
  REPEAT STRIP_TAC THEN
  ASSUME_TAC(ISPECL[`p:A->A`;`j:int`;`k:int-j:int`]
             pow_int_plus_eval) THEN
  ASSUME_TAC(ARITH_RULE `(j:int)+(k-j)=(k:int)`) THEN
  ASSUME_TAC(ISPECL[`p:A->A`;`j:int`;`x:A`;
                    `((p:A->A) pow_int (k-j)) (x:A)`]
                    pow_int_cancel) THEN
  ASM_MESON_TAC[]);;

let pow_int_cancel_num = prove(
  `!p:A->A x j k. bijection p ==>
   (p pow_num j) x = (p pow_num k) x ==>
   x = (p pow_int (&k - &j)) x`,
  MESON_TAC[pow_int_num;pow_int_cancel2]);;
```

```
let pow_int_cancel_num2 = prove(
  `!p:A->A x j k. bijection p ==>
   (p pow_num j) x = (p pow_num k) x /\ j <= k
   ==> x = (p pow_num (k-j)) x`,
  MESON_TAC[pow_int_cancel_num;INT_OF_NUM_SUB;pow_int_num]);;

(* ----- cycle *)

let cycle_contains_start = prove(
  `!p:A->A x. x IN cycle p x`,
  REPEAT GEN_TAC THEN
  REWRITE_TAC[cycle;IN_ELIM_THM] THEN
  EXISTS_TAC `&0:int` THEN
  REWRITE_TAC[pow_int_0;I_THM]);;

let cycle_lemma1 = prove(
  `!p:A->A x y k. bijection p /\ y = (p pow_int k) (p x) ==>
    y = (p pow_int (k + &1)) x`,
  REPEAT STRIP_TAC THEN
  ASM_SIMP_TAC[pow_int_plus_eval;pow_int_1]);;

let cycle_lemma2 = prove(
  `!p:A->A x y k. bijection p /\ y = (p pow_int k) (p x) ==>
    y IN cycle p x`,
  REPEAT STRIP_TAC THEN
  REWRITE_TAC[cycle;IN_ELIM_THM] THEN
  EXISTS_TAC `(k:int) + &1` THEN
  ASM_MESON_TAC[cycle_lemma1]);;

let cycle_lemma3 = prove(
  `!p:A->A x y. y IN cycle p x ==> ?k:int. y = (p pow_int k)(x)`,
  REWRITE_TAC[cycle;IN_ELIM_THM]);;

let cycle_lemma4 = prove(
  `!p:A->A x y k. bijection p /\ y = (p pow_int k) x ==>
    y = (p pow_int (k - &1)) (p x)`,
  REPEAT STRIP_TAC THEN
  ASSUME_TAC(INT_ARITH `(k:int) = (k - &1) + &1`) THEN
  ASM_MESON_TAC[pow_int_plus_eval;pow_int_1]);;

let cycle_lemma5 = prove(
  `!p:A->A x y k. bijection p /\ y = (p pow_int k) x ==>
    y IN cycle p (p x)`,
  REPEAT STRIP_TAC THEN
  REWRITE_TAC[cycle;IN_ELIM_THM] THEN
  EXISTS_TAC `(k:int) - &1` THEN
  ASM_MESON_TAC[cycle_lemma4]);;
```

```
let cycle_shift_forward = prove(
  `!p:A->A x y. bijection p ==> y IN cycle p (p x) ==> y IN cycle p x`,
  REPEAT STRIP_TAC THEN
  MATCH_MP_TAC cycle_lemma2 THEN
  ASM_MESON_TAC[cycle_lemma3]);;

let cycle_shift_backward = prove(
  `!p:A->A x y. bijection p ==> y IN cycle p x ==> y IN cycle p (p x)`,
  REPEAT STRIP_TAC THEN
  MATCH_MP_TAC cycle_lemma5 THEN
  ASM_MESON_TAC[cycle_lemma3]);;

let cycle_shift = prove(
  `!p:A->A x. bijection p ==> cycle p (p x) = cycle p x`,
  MESON_TAC[cycle_shift_backward;cycle_shift_forward;EXTENSION]);;

let cycle_ifpow = prove(
  `!p:A->A x y k. y = (p pow_int k) x ==> y IN cycle p x`,
  REWRITE_TAC[cycle;IN_ELIM_THM] THEN
  MESON_TAC[]);;

let cycle_aspow = prove(
  `!p:A->A x y. y IN cycle p x ==> ?k. y = (p pow_int k) x`,
  REWRITE_TAC[cycle;IN_ELIM_THM]);;

(* ----- cyclemin *)

let cyclemin_in_cycle = prove(
  `!p x. cyclemin p x IN cycle p x`,
  REPEAT GEN_TAC THEN
  REWRITE_TAC[IN] THEN
  REWRITE_TAC[cyclemin] THEN
  MATCH_MP_TAC(prove(`!P. (?n. P n) ==> P((minimal) P)`,
                     MESON_TAC[MINIMAL])) THEN
  ASSUME_TAC(ISPECL[`p:num->num`;`x:num`]cycle_contains_start) THEN
  ASM_MESON_TAC[IN]);;

let cyclemin_le = prove(
  `!p x. cyclemin p x <= x`,
  REPEAT GEN_TAC THEN
  REWRITE_TAC[cyclemin] THEN
  MATCH_MP_TAC MINIMAL_UBOUND THEN
  MESON_TAC[cycle_contains_start;IN]);;

let cyclemin_0 = prove(
  `!p. cyclemin p 0 = 0`,
  MESON_TAC[cyclemin_le;LE_0;LE_ANTISYM]);;
```

```
let cyclemin_shift = prove(
  `!p x. bijection p ==> cyclemin p (p x) = cyclemin p x`,
  MESON_TAC[cycle_shift;cyclemin]);;

let cyclemin_aspow = prove(
  `!p x. ?k. cyclemin p x = (p pow_int k) x`,
  REPEAT STRIP_TAC THEN
  MATCH_MP_TAC cycle_aspow THEN
  ASM_MESON_TAC[cyclemin_in_cycle]);;

(* ----- MIN and minimal *)

let min_le_first = prove(
  `!m n. MIN m n <= m`,
  REWRITE_TAC[MIN] THEN
  MESON_TAC[LE_REFL;NOT_LE;LT_LE]);;

let min_le_second = prove(
  `!m n. MIN m n <= n`,
  REWRITE_TAC[MIN] THEN
  MESON_TAC[LE_REFL;NOT_LE;LT_LE]);;

let minimal_union_le = prove(
  `!s:num->bool t:num->bool.
    ~(s = EMPTY) ==> (minimal) (s UNION t) <= (minimal) s`,
  MESON_TAC[CHOICE_DEF;MINIMAL;IN;IN_UNION;MINIMAL_UBOUND]);;

let minimal_union_le_min = prove(
  `!s:num->bool t:num->bool.
   ~(s = EMPTY) ==> ~(t = EMPTY) ==>
   (minimal) (s UNION t) <= MIN ((minimal) s) ((minimal) t)`,
  REWRITE_TAC[MIN] THEN
  MESON_TAC[minimal_union_le;UNION_COMM]);;

let nonempty_union = prove(
  `!s:A->bool t:A->bool.
  ~(s = EMPTY) ==> ~(s UNION t = EMPTY)`,
  MESON_TAC[CHOICE_DEF;MINIMAL;IN;IN_UNION;MEMBER_NOT_EMPTY]);;

let minimal_union_lemma = prove(
  `!s:num->bool t:num->bool.
   (minimal) (s UNION t) IN s ==>
   (minimal) s <= (minimal) (s UNION t)`,
  MESON_TAC[MINIMAL_UBOUND;IN]);;
```

```
let minimal_union_lemma2 = prove(
  `!s:num->bool t:num->bool.
   (minimal) (s UNION t) IN t ==>
   (minimal) t <= (minimal) (s UNION t)`,
  MESON_TAC[MINIMAL_UBOUND;IN]);;

let minimal_union_lemma3 = prove(
  `!s:num->bool t:num->bool.
   (minimal) (s UNION t) IN s ==>
   MIN ((minimal) s) ((minimal) t) <= (minimal) (s UNION t)`,
  MESON_TAC[minimal_union_lemma;IN;min_le_first;LE_TRANS]);;

let minimal_union_lemma4 = prove(
  `!s:num->bool t:num->bool.
   (minimal) (s UNION t) IN t ==>
   MIN ((minimal) s) ((minimal) t) <= (minimal) (s UNION t)`,
  MESON_TAC[minimal_union_lemma2;IN;min_le_second;LE_TRANS]);;

let minimal_union_lemma5 = prove(
  `!s:num->bool t:num->bool.
   ~(s = EMPTY) ==> ~(t = EMPTY) ==>
   MIN ((minimal) s) ((minimal) t) <= (minimal) (s UNION t)`,
  MESON_TAC[nonempty_union;CHOICE_DEF;MINIMAL;IN;IN_UNION;
            minimal_union_lemma3;minimal_union_lemma4]);;

let minimal_union = prove(
  `!s:num->bool t:num->bool.
   ~(s = EMPTY) ==> ~(t = EMPTY) ==>
   (minimal) (s UNION t) = MIN ((minimal) s) ((minimal) t)`,
  MESON_TAC[minimal_union_lemma5;minimal_union_le_min;LE_ANTISYM]);;

(* ----- pigeonhole principle for range *)

let range_pigeonhole_lemma1 = prove(
  `!f:num->A n.
    (!j k. j IN range n /\ k IN range n /\ f j = f k ==> j = k)
     ==> CARD(IMAGE f (range n)) = CARD(range n)`,
  MESON_TAC[CARD_IMAGE_INJ;range_finite]);;

let range_pigeonhole_lemma2 = prove(
  `!f:num->A n.
   (!j k. j IN range n /\ k IN range n /\ f j = f k ==> j = k)
   ==> CARD(IMAGE f (range n)) = n`,
  MESON_TAC[range_pigeonhole_lemma1;range_card]);;
```

```
let range_pigeonhole_lemma3 = prove(
  `!f:num->A n.
   (!j k. j IN range n /\ k IN range n /\ f j = f k ==> j = k)
   ==> CARD { y | ?k. k IN range n /\ y = f k } = n`,
  REPEAT STRIP_TAC THEN
  ASSUME_TAC(ISPECL[`f:num->A`;`n:num`]
             range_pigeonhole_lemma2) THEN
  ASSUME_TAC(ISPECL[`range n`;`f:num->A`]IMAGE) THEN
  ASM_MESON_TAC[]);;

let range_pigeonhole_lemma4 = prove(
  `!f:num->A n.
   (!j k. j < n /\ k < n /\ f j = f k ==> j = k)
   ==> CARD { y | ?k. k < n /\ y = f k } = n`,
  REWRITE_TAC[range_in_flip] THEN
  MESON_TAC[range_pigeonhole_lemma3]);;

let range_pigeonhole = prove(
  `!f:num->A n.
   CARD { y | ?k. k < n /\ y = f k } < n ==>
   (?j k. j < n /\ k < n /\ f j = f k /\ ~(j = k))`,
  MESON_TAC[range_pigeonhole_lemma4;LT_REFL]);;

(* ----- partcycle *)

let partcycle_subsetcycle = prove(
  `!p:A->A x m. partcycle m p x SUBSET cycle p x`,
  REWRITE_TAC[partcycle;cycle;SUBSET;IN_ELIM_THM] THEN
  MESON_TAC[pow_int_num]);;

let partcycle_monotonic = prove(
  `!p:A->A x m n. m <= n ==> partcycle m p x SUBSET partcycle n p x`,
  REWRITE_TAC[partcycle;SUBSET;IN_ELIM_THM] THEN
  MESON_TAC[LTE_TRANS]);;

let partcycle_lemma1 = prove(
 `!p:A->A x n j k. bijection p ==>
   (p pow_num j) x = (p pow_num k) x /\ j < k /\ k < n
   ==> ?m. 0 < m /\ m < n /\ x = (p pow_num m) x`,
  REPEAT STRIP_TAC THEN
  EXISTS_TAC `k-j` THEN
  ASSUME_TAC(ARITH_RULE `j < k ==> 0 < k - j`) THEN
  ASSUME_TAC(ARITH_RULE `k < n ==> k - j < n`) THEN
  ASM_MESON_TAC[pow_int_cancel_num2;LT_LE]);;
```

```
let partcycle_lemma2 = prove(
  `!p:A->A x n j k. bijection p ==>
   (p pow_num j) x = (p pow_num k) x /\ j < n /\ k < n /\ ~(j = k)
   ==> ?m. 0 < m /\ m < n /\ x = (p pow_num m) x`,
  REPEAT STRIP_TAC THEN
  ASM_CASES_TAC `j < k` THENL
  [
    ASM_MESON_TAC[partcycle_lemma1]
  ;
    ASSUME_TAC(SPECL[`j:num`;`k:num`]LT_CASES) THEN
    ASM_MESON_TAC[partcycle_lemma1]
  ]);;

let partcycle_lemma3 = prove(
  `!p:A->A x n.
   CARD { y | ?k. k < n /\ y = (p pow_num k) x } < n ==>
   (?j k. j < n /\ k < n /\
          (p pow_num j) x = (p pow_num k) x /\ ~(j = k))`,
  REPEAT STRIP_TAC THEN
  ASSUME_TAC(ISPECL[`\m. ((p:A->A) pow_num m) (x:A)`;`n:num`]
                   range_pigeonhole) THEN
  ASM_MESON_TAC[BETA_THM]);;

let partcycle_lemma4 = prove(
  `!p:A->A x n. bijection p ==>
   CARD(partcycle n p x) < n ==>
   ?m. 0 < m /\ m < n /\ x = (p pow_num m) x`,
  REPEAT STRIP_TAC THEN
  ASSUME_TAC(ISPECL[`p:A->A`;`x:A`;`n:num`]
             partcycle_lemma3) THEN
  ASSUME_TAC(ISPECL[`p:A->A`;`x:A`;`n:num`]
             partcycle_lemma2) THEN
  ASM_MESON_TAC[partcycle]);;

let partcycle_lemma5 = prove(
  `!p:A->A x m k.
   0 < m /\ x = (p pow_num m) x ==> x = (p pow_num (k*m)) x`,
  GEN_TAC THEN GEN_TAC THEN GEN_TAC THEN
  INDUCT_TAC THENL
  [
    MESON_TAC[pow_num_0_eval;ARITH_RULE `0*m = 0`]
  ;
    ASM_MESON_TAC[pow_num_plus_eval;
                  ARITH_RULE `(SUC k * m) = m + (k * m)`]
  ]);;
```

```
let partcycle_lemma6 = prove(
  `!p:A->A x m k.
   0 < m /\ x = (p pow_num m) x ==> x = (p pow_int (&k * &m)) x`,
  MESON_TAC[partcycle_lemma5;INT_OF_NUM_MUL;pow_int_num]);;

let partcycle_lemma7 = prove(
  `!p:A->A x m k. bijection p ==>
   0 < m /\ x = (p pow_num m) x ==> x = (p pow_int (--(&k) * &m)) x`,
  REPEAT STRIP_TAC THEN
  ASSUME_TAC(ISPEC `p:A->A` pow_int_0) THEN
  ASSUME_TAC(INT_ARITH `--(&k) * ((&m):int) + &k * &m = &0`) THEN
  ASSUME_TAC(ISPECL[`p:A->A`;`--(&k) * ((&m):int)`;
                    `&k * ((&m):int)`;`x:A`]pow_int_plus_eval) THEN
  ASSUME_TAC(ISPECL[`p:A->A`;`x:A`;`m:num`;`k:num`]
                    partcycle_lemma6) THEN
  ASM_MESON_TAC[I_THM]);;

let partcycle_lemma8 = prove(
  `!p:A->A x m k. bijection p ==>
   0 < m /\ x = (p pow_num m) x ==> x = (p pow_int (k * &m)) x`,
  REPEAT STRIP_TAC THEN
  ASM_CASES_TAC `&0 <= (k:int)` THENL
  [
    ASM_MESON_TAC[INT_OF_NUM_OF_INT;partcycle_lemma6]
  ;
    ASSUME_TAC(INT_ARITH `~(&0 <= (k:int)) ==> &0 <= --k`) THEN
    ASSUME_TAC(INT_ARITH `(k:int) = -- (--k)`) THEN
    ASSUME_TAC(SPEC `--k:int` INT_OF_NUM_OF_INT) THEN
    ASM_MESON_TAC[partcycle_lemma7]
  ]);;

let partcycle_lemma9 = prove(
  `!p:A->A x m j. bijection p ==>
   0 < m /\ x = (p pow_num m) x ==>
   (p pow_int j) x = (p pow_int (j rem &m)) x`,
  REPEAT STRIP_TAC THEN
  ASSUME_TAC(SPECL[`j:int`;`&(m:num):int`]INT_DIVISION) THEN
  ASSUME_TAC(ARITH_RULE `0 < m ==> ~(m = 0)`) THEN
  ASSUME_TAC(SPECL[`m:num`;`0`]INT_OF_NUM_EQ) THEN
  ASSUME_TAC(ARITH_RULE
             `j div &m * &m + j rem &m =
              j rem &m + j div &m * &m`) THEN
  ASSUME_TAC(ISPECL[`p:A->A`;`x:A`;`m:num`;`j div &m`]
             partcycle_lemma8) THEN
  ASSUME_TAC(ISPECL[`p:A->A`;`j rem &m`;`j div &m * &m`;`x:A`]
             pow_int_plus_eval) THEN
  ASM_MESON_TAC[]);;
```

```
let partcycle_lemma10 = prove(
  `!p:A->A x m j. bijection p ==>
   0 < m /\ x = (p pow_num m) x ==>
   (p pow_int j) x IN partcycle m p x`,
  REPEAT STRIP_TAC THEN
  REWRITE_TAC[partcycle;IN_ELIM_THM] THEN
  EXISTS_TAC `num_of_int(j rem &m)` THEN
  ASSUME_TAC(ARITH_RULE `0 < m ==> ~(m = 0)`) THEN
  ASSUME_TAC(SPECL[`m:num`;`0`]INT_OF_NUM_EQ) THEN
  ASSUME_TAC(SPECL[`m:num`]INT_ABS_NUM) THEN
  ASSUME_TAC(SPECL[`j rem &m`]INT_OF_NUM_OF_INT) THEN
  ASSUME_TAC(SPECL[`j:int`;`&(m:num):int`]INT_DIVISION) THEN
  ASSUME_TAC(SPECL[`p:A->A`;`x:A`;`m:num`;`j:int`]
                 partcycle_lemma9) THEN
  ASSUME_TAC(SPECL[`p:A->A`;`num_of_int(j rem &m)`]pow_int_num) THEN
  ASSUME_TAC(SPECL[`num_of_int(j rem &m)`;`m:num`]INT_OF_NUM_LT) THEN
  ASM_MESON_TAC[]);;

let partcycle_lemma11 = prove(
  `!p:A->A x m. bijection p ==>
   0 < m /\ x = (p pow_num m) x ==>
   cycle p x SUBSET partcycle m p x`,
  MESON_TAC[cycle_aspow;partcycle_lemma10;SUBSET]);;

let partcycle_lemma12 = prove(
  `!p:A->A x m n. bijection p ==>
   0 < m /\ m <= n /\ x = (p pow_num m) x ==>
   cycle p x = partcycle n p x`,
  MESON_TAC[partcycle_subsetcycle;partcycle_lemma11;
            SUBSET_ANTISYM;SUBSET_TRANS;partcycle_monotonic]);;

let partcycle_lemma13 = prove(
  `!p:A->A x n. bijection p ==>
   CARD(partcycle n p x) < n ==> cycle p x = partcycle (n-1) p x`,
  MESON_TAC[partcycle_lemma4;
            partcycle_lemma12;
            ARITH_RULE `m < n ==> m <= n-1`]);;

let partcycle_ifcyclelt = prove(
  `!p:A->A x n. bijection p ==> FINITE(cycle p x) ==>
   CARD(cycle p x) < n ==> cycle p x = partcycle (n-1) p x`,
  MESON_TAC[partcycle_subsetcycle;CARD_SUBSET;LET_TRANS;
            partcycle_lemma13]);;

let partcycle_ifcyclele = prove(
  `!p:A->A x n. bijection p ==> FINITE(cycle p x) ==>
   CARD(cycle p x) <= n ==> cycle p x = partcycle n p x`,
  MESON_TAC[partcycle_ifcyclelt;LT_SUC_LE;
            ARITH_RULE `(n+1)-1 = n`;suc_isadd1]);;
```

```
(* ----- part2powcycle *)

let part2powcycle_hasstart = prove(
  `!i p:A->A x. x IN part2powcycle i p x`,
  REWRITE_TAC[part2powcycle;partcycle;IN_ELIM_THM] THEN
  REPEAT GEN_TAC THEN
  EXISTS_TAC `0` THEN
  REWRITE_TAC[pow_num_0_eval] THEN
  MESON_TAC[EXP_EQ_0;ARITH_RULE `~(2 = 0)`;
            ARITH_RULE `~(n = 0) ==> 0 < n`]);;

let part2powcycle_nonempty = prove(
  `!i p:A->A x. ~(part2powcycle i p x = EMPTY)`,
  MESON_TAC[part2powcycle_hasstart;MEMBER_NOT_EMPTY]);;

let part2powcycle_0_lemma = prove(
  `!p:num->num x. { x } = { y | ?k. k = 0 /\ y = (p pow_num k)(x) }`,
  REWRITE_TAC[EXTENSION;IN_ELIM_THM;IN_SING] THEN
  MESON_TAC[pow_num_0_eval]);;

let part2powcycle_0_lemma2 = prove(
  `!x m. m < x ==> ~{x} m`,
  MESON_TAC[LT_REFL;IN_SING;IN]);;

let part2powcycle_0_lemma3 = prove(
  `!x. x = (minimal) {x}`,
  MESON_TAC[part2powcycle_0_lemma2;MINIMAL_UNIQUE;IN_SING;IN]);;

let part2powcycle_0 = prove(
  `!p x. x = (minimal) (part2powcycle 0 p x)`,
  SIMP_TAC[part2powcycle;partcycle] THEN
  REWRITE_TAC[ARITH_RULE `2 EXP 0 = 1`] THEN
  REWRITE_TAC[ARITH_RULE `k < 1 <=> k = 0`] THEN
  MESON_TAC[part2powcycle_0_lemma;part2powcycle_0_lemma3]);;

let part2powcycle_plus1_lemma1 = prove(
  `!i. 2 EXP i <= 2 EXP (i+1)`,
  MESON_TAC[LE_EXP;ARITH_RULE `i <= i+1`;ARITH_RULE `~(2 = 0)`]);;

let part2powcycle_plus1_lemma2 = prove(
  `!i k. k < 2 EXP i ==> k < 2 EXP (i+1)`,
  MESON_TAC[part2powcycle_plus1_lemma1;LTE_TRANS]);;

let part2powcycle_plus1_lemma3 = prove(
  `!i p:A->A x. part2powcycle i p x SUBSET
                part2powcycle (i+1) p x`,
  REWRITE_TAC[part2powcycle;partcycle;SUBSET;IN_ELIM_THM] THEN
  MESON_TAC[part2powcycle_plus1_lemma2]);;
```

```
let part2powcycle_plus1_lemma4 = prove(
  `!i. 2 EXP i + 2 EXP i = 2 EXP (i+1)`,
  MESON_TAC[EXP;suc_isadd1;ARITH_RULE `2*n = n+n`]);;

let part2powcycle_plus1_lemma5 = prove(
  `!i k. k < 2 EXP i ==> k + 2 EXP i < 2 EXP (i+1)`,
  MESON_TAC[part2powcycle_plus1_lemma4;
            ARITH_RULE `m < n ==> m + p < n + p`]);;

let part2powcycle_plus1_lemma6 = prove(
  `!i p:A->A x.
   part2powcycle i p ((p pow_num (2 EXP i)) x)
   SUBSET part2powcycle (i+1) p x`,
  REWRITE_TAC[part2powcycle;partcycle;SUBSET;IN_ELIM_THM] THEN
  MESON_TAC[part2powcycle_plus1_lemma5;pow_num_plus_eval]);;

let part2powcycle_plus1_lemma7 = prove(
  `!k n. k < n+n ==> k < n \/ (k-n < n /\ k = (k-n)+n)`,
  REPEAT STRIP_TAC THEN
  ASM_CASES_TAC `k<n` THEN
  ASM_MESON_TAC[NOT_LT;SUB_ADD;LT_ADD_RCANCEL]);;

let part2powcycle_plus1_lemma8 = prove(
  `!i k. k < 2 EXP(i+1) ==>
         k < 2 EXP i \/
         (k-2 EXP i < 2 EXP i /\ k = (k-2 EXP i)+2 EXP i)`,
  MESON_TAC[part2powcycle_plus1_lemma7;part2powcycle_plus1_lemma4]);;

let part2powcycle_plus1_lemma9 = prove(
  `!i p:A->A x k. k < 2 EXP (i+1) ==> ~(k < 2 EXP i) ==>
   (p pow_num k) x IN
   part2powcycle i p ((p pow_num (2 EXP i)) x)`,
  REWRITE_TAC[part2powcycle;partcycle;IN;IN_ELIM_THM] THEN
  REPEAT STRIP_TAC THEN
  EXISTS_TAC `k - 2 EXP i` THEN
  ASM_MESON_TAC[part2powcycle_plus1_lemma8;pow_num_plus_eval]);;

let part2powcycle_plus1_lemma10 = prove(
  `!i p:A->A x k. k < 2 EXP i ==>
   (p pow_num k) x IN part2powcycle i p x`,
  REWRITE_TAC[part2powcycle;partcycle;IN;IN_ELIM_THM] THEN
  MESON_TAC[]);;
```

```
let part2powcycle_plus1_lemma11 = prove(
  `!i p:A->A x.
   {y | ?k. k < 2 EXP (i+1) /\ y = (p pow_num k) x}
   SUBSET part2powcycle i p x UNION
          part2powcycle i p ((p pow_num (2 EXP i)) x)`,
  REWRITE_TAC[SUBSET;UNION;IN_ELIM_THM] THEN
  MESON_TAC[part2powcycle_plus1_lemma9;part2powcycle_plus1_lemma10]);;

let part2powcycle_plus1_lemma12 = prove(
  `!i p:A->A x.
   part2powcycle (i+1) p x
   = {y | ?k. k < 2 EXP (i+1) /\ y = (p pow_num k) x}`,
  SIMP_TAC[part2powcycle;partcycle]);;

let part2powcycle_plus1_lemma13 = prove(
  `!i p:A->A x.
   part2powcycle (i+1) p x
   SUBSET part2powcycle i p x UNION
          part2powcycle i p ((p pow_num (2 EXP i)) x)`,
  REWRITE_TAC[part2powcycle_plus1_lemma12] THEN
  MESON_TAC[part2powcycle_plus1_lemma11]);;

let part2powcycle_plus1_lemma14 = prove(
  `!i p:A->A x.
   part2powcycle i p x UNION
   part2powcycle i p ((p pow_num (2 EXP i)) x)
   SUBSET part2powcycle (i+1) p x`,
  MESON_TAC[UNION_SUBSET;part2powcycle_plus1_lemma3;
            part2powcycle_plus1_lemma6]);;

let part2powcycle_plus1 = prove(
  `!i p:A->A x.
   part2powcycle (i+1) p x =
   part2powcycle i p x UNION
   part2powcycle i p ((p pow_num (2 EXP i)) x)`,
  MESON_TAC[SUBSET_ANTISYM;part2powcycle_plus1_lemma13;
            part2powcycle_plus1_lemma14]);;

let part2powcycle_ifcyclele = prove(
  `!p:A->A x i. bijection p ==> FINITE(cycle p x) ==>
   CARD(cycle p x) <= 2 EXP i ==> cycle p x = part2powcycle i p x`,
  MESON_TAC[partcycle_ifcyclele;part2powcycle]);;

(* ----- fastcyclemin *)

let fastcyclemin_part2powcycle_0 = prove(
  `!p x. fastcyclemin 0 p x = (minimal) (part2powcycle 0 p x)`,
  MESON_TAC[fastcyclemin;part2powcycle_0]);;
```

```
let fastcyclemin_lemma1 = prove(
  `!i p x.
   fastcyclemin (i+1) p x =
   if i+1 = 0 then x else
   MIN (fastcyclemin ((i+1)-1) p x)
       (fastcyclemin ((i+1)-1) p ((p pow_num (2 EXP ((i+1)-1))) x))`,
  MESON_TAC[fastcyclemin]);;

let fastcyclemin_lemma2 = prove(
  `!i p x. fastcyclemin (i+1) p x =
          MIN (fastcyclemin i p x)
              (fastcyclemin i p ((p pow_num (2 EXP i)) x))`,
  REPEAT GEN_TAC THEN
  ASSUME_TAC(ARITH_RULE `~(i+1 = 0)`) THEN
  ASSUME_TAC(ARITH_RULE `(i+1)-1 = i`) THEN
  ASM_SIMP_TAC[fastcyclemin_lemma1]);;

let fastcyclemin_part2powcycle = prove(
  `!i p x. fastcyclemin i p x = (minimal) (part2powcycle i p x)`,
  INDUCT_TAC THENL
  [
    MESON_TAC[fastcyclemin_part2powcycle_0]
  ;
    REWRITE_TAC[suc_isadd1] THEN
    REWRITE_TAC[part2powcycle_plus1] THEN
    ASSUME_TAC(ISPECL[`i:num`;`p:num->num`;`x:num`]part2powcycle_nonempty)
    THEN
    ASSUME_TAC(ISPECL[`i:num`;`p:num->num`;
              `(p pow_num (2 EXP i)) x:num`]part2powcycle_nonempty) THEN
    ASM_SIMP_TAC[minimal_union] THEN
    ASM_MESON_TAC[fastcyclemin_lemma2]
  ]);;

let fastcyclemin_works = prove(
  `!p:num->num x i. bijection p ==> FINITE(cycle p x) ==>
   CARD(cycle p x) <= 2 EXP i ==> cyclemin p x = fastcyclemin i p x`,
  REWRITE_TAC[cyclemin;fastcyclemin_part2powcycle] THEN
  MESON_TAC[part2powcycle_ifcyclele]);;

(* ----- Xif *)

let xif_parity = prove(
  `!s n. ODD(Xif s n) = ~(s(n DIV 2) = ODD n)`,
  REWRITE_TAC[xif] THEN
  MESON_TAC[xor1_parity]);;
```

```
let xif_double_parity = prove(
  `!s j. ODD(Xif s (2*j)) = s j`,
  REWRITE_TAC[xif_parity;double_div2] THEN
  MESON_TAC[NOT_EVEN;EVEN_DOUBLE]);;

let xif_double_plus1_parity = prove(
  `!s j. ODD(Xif s (2*j+1)) = ~(s j)`,
  REWRITE_TAC[xif_parity;double_plus1_div2] THEN
  MESON_TAC[ODD_DOUBLE;suc_isadd1]);;

let xif_involution = prove(
  `!s. involution(Xif s)`,
  REWRITE_TAC[involution;xif] THEN
  REPEAT GEN_TAC THEN
  ASM_CASES_TAC `(s:num->bool)(x DIV 2)` THEN
  ASM_SIMP_TAC[xor1_div2;xor1xor1]);;

let xif_fixesge = prove(
  `!s b. s subsetrange b ==> Xif s fixesge(2*b)`,
  REWRITE_TAC[subsetrange;fixesge;xif] THEN
  MESON_TAC[div2_lt_doubling_ge]);;

(* ----- vanishesifonbothsidesof *)

let vanishesifonbothsidesof_eval = prove(
  `!p:A->A q:A->A. p vanishesifonbothsidesof q <=> !x. p(q(p x)) = q x`,
  REWRITE_TAC[vanishesifonbothsidesof] THEN
  MESON_TAC[o_DEF;EQ_EXT]);;

let vanishesifonbothsidesof_pow_num_eval = prove(
  `!p:A->A q:A->A. p vanishesifonbothsidesof q ==>
    !n. !x. (p pow_num n)(q((p pow_num n)x)) = q x`,
  REWRITE_TAC[vanishesifonbothsidesof_eval] THEN
  REPEAT GEN_TAC THEN DISCH_TAC THEN INDUCT_TAC THENL
  [
    REWRITE_TAC[pow_num_0;I_DEF]
  ;
    REWRITE_TAC[suc_isadd1] THEN
    ASSUME_TAC(SPECL[`n:num`;`p:A->A`]pow_num_plus1_eval) THEN
    ASSUME_TAC(SPECL[`n:num`;`p:A->A`]pow_num_plus1_right_eval) THEN
    ASM_MESON_TAC[]
  ]);;

let vanishesifonbothsidesof_pow_num_lemma = prove(
  `!p:A->A q:A->A. p vanishesifonbothsidesof q ==>
    !n. (p pow_num n) o q o (p pow_num n) = q`,
  REPEAT GEN_TAC THEN DISCH_TAC THEN
  ASM_MESON_TAC[vanishesifonbothsidesof_pow_num_eval;o_DEF;EQ_EXT]);;
```

```
let vanishesifonbothsidesof_pow_num = prove(
  `!p:A->A q:A->A. p vanishesifonbothsidesof q ==>
    !n. (p pow_num n) vanishesifonbothsidesof q`,
  MESON_TAC[vanishesifonbothsidesof_pow_num_lemma;
            vanishesifonbothsidesof]);;

let vanishesifonbothsidesof_inverse = prove(
  `!p:A->A. bijection p ==> !q:A->A. p vanishesifonbothsidesof q ==>
    inverse p vanishesifonbothsidesof q`,
  REWRITE_TAC[vanishesifonbothsidesof_eval] THEN
  REPEAT STRIP_TAC THEN
  ASSUME_TAC(ISPEC `p:A->A` bijection_inverse_composition1) THEN
  ASSUME_TAC(ISPEC `p:A->A` bijection_inverse_composition2) THEN
  ASM_MESON_TAC[]);;

let vanishesifonbothsidesof_pow_int = prove(
  `!p:A->A. bijection p ==> !q:A->A. p vanishesifonbothsidesof q
    ==> !k:int. p pow_int k vanishesifonbothsidesof q`,
  REWRITE_TAC[pow_int] THEN
  REPEAT STRIP_TAC THEN
  ASSUME_TAC(INT_ARITH `k < &0 ==> ~(&0 <= k)`) THEN
  ASM_MESON_TAC[vanishesifonbothsidesof_pow_num;
                vanishesifonbothsidesof_inverse]);;

(* ----- fixesge *)

let fixesge_double_odd = prove(
  `!p b x. p fixesge(2*b) ==> x >= b ==> ~ODD(p(2*x))`,
  MESON_TAC[fixesge;LE_MULT_LCANCEL;GE;EVEN_DOUBLE;NOT_ODD]);;

let fixesge_inverts_1 = prove(
  `!p n. bijection p ==> p fixesge n ==> inverse p fixesge n`,
  REWRITE_TAC[bijection;surjection;injection;inverse;fixesge] THEN
  MESON_TAC[]);;

let fixesge_inverts_2 = prove(
  `!p n. bijection p /\ (!x. x >= n ==> p x = x) ==>
         !x. x >= n ==> inverse p x = x`,
  MESON_TAC[fixesge_inverts_1;fixesge]);;

let fixesge_lemma1 = prove(
  `!p n m. bijection p ==> p fixesge n ==> p m >= n ==> m >= n`,
  REPEAT STRIP_TAC THEN
  ASSUME_TAC(SPECL[`p:num->num`;`n:num`]fixesge_inverts_1) THEN
  ASSUME_TAC(ISPECL[`p:num->num`;`m:num`]
             bijection_inverse_composition2) THEN
  ASM_MESON_TAC[GE;NOT_LT;fixesge]);;
```

```
let fixesge_lt = prove(
  `!p n m. bijection p ==> p fixesge n ==> m < n ==> p m < n`,
  MESON_TAC[fixesge_lemma1;GE;NOT_LT]);;

let fixesge_I = prove(
  `!n. I fixesge n`,
  REWRITE_TAC[fixesge;I_DEF]);;

let fixesge_composes = prove(
  `!f g n. f fixesge n /\ g fixesge n ==> f o g fixesge n`,
  REWRITE_TAC[fixesge] THEN
  MESON_TAC[o_DEF]);;

let fixesge_pow_lemma1 = prove(
  `!f n m. f fixesge n ==> f pow_num m fixesge n
           ==> f o (f pow_num m) fixesge n`,
  REPEAT STRIP_TAC THEN
  ASM_MESON_TAC[fixesge_composes]);;

let fixesge_pow_lemma2 = prove(
  `!f n m. f fixesge n ==> f pow_num m fixesge n
           ==> f o (f pow_num ((m+1)-1)) fixesge n`,
  REPEAT STRIP_TAC THEN
  ASSUME_TAC(ARITH_RULE `(m+1)-1 = m`) THEN
  ASM_MESON_TAC[fixesge_pow_lemma1]);;

let fixesge_pow_lemma3 = prove(
  `!f n m. f fixesge n ==> f pow_num m fixesge n
           ==> f pow_num (m+1) fixesge n`,
  REPEAT STRIP_TAC THEN
  ASSUME_TAC(ARITH_RULE `~(m+1 = 0)`) THEN
  ASM_MESON_TAC[pow_num;fixesge_pow_lemma2]);;

let fixesge_pow_num = prove(
  `!f n m. f fixesge n ==> f pow_num m fixesge n`,
  GEN_TAC THEN GEN_TAC THEN
  INDUCT_TAC THENL
  [
    ASM_MESON_TAC[pow_num;fixesge_I]
  ;
    REWRITE_TAC[suc_isadd1] THEN
    ASM_MESON_TAC[fixesge_pow_lemma3]
  ]);;

let fixesge_pow_int = prove(
  `!p n k. bijection p ==> p fixesge n ==> p pow_int k fixesge n`,
  MESON_TAC[fixesge_pow_num;pow_int;fixesge_inverts_1]);;
```

```
let fixesge_cyclemin = prove(
  `!p n. bijection p ==> p fixesge n ==> cyclemin p fixesge n`,
  MESON_TAC[fixesge_pow_int;cyclemin_aspow;fixesge]);;

(* ----- XbackXforth *)

let XbackXforth_forward = prove(
  `!p x. bijection p ==> (XbackXforth p)(xor1(p x)) = p(xor1 x)`,
  REWRITE_TAC[XbackXforth] THEN
  REWRITE_TAC[o_DEF] THEN
  MESON_TAC[xor1xor1;bijection_inverse_composition2]);;

let XbackXforth_fixesge = prove(
  `!p b. bijection p ==> p fixesge(2*b) ==> XbackXforth p fixesge(2*b)`,
  REWRITE_TAC[XbackXforth;fixesge;o_THM] THEN
  REPEAT STRIP_TAC THEN
  SUBGOAL_THEN `!x. x >= 2*b ==> xor1 x >= 2*b` ASSUME_TAC
  THENL [MESON_TAC[xor1_geeven];ALL_TAC] THEN
  SUBGOAL_THEN `!x. x >= 2*b ==> p(xor1 x) = xor1 x` ASSUME_TAC
  THENL [ASM_MESON_TAC[];ALL_TAC] THEN
  SUBGOAL_THEN `!x. x >= 2*b ==> xor1(p(xor1 x)) = x` ASSUME_TAC
  THENL [ASM_MESON_TAC[xor1xor1];ALL_TAC] THEN
  SUBGOAL_THEN `!x. x >= 2*b ==>
              inverse p(xor1(p(xor1 x))) = x` ASSUME_TAC
  THENL [ASM_MESON_TAC[fixesge_inverts_2];ALL_TAC]
  THEN ASM_MESON_TAC[]);;

let XbackXforth_bijection = prove(
  `!p. bijection p ==> bijection(XbackXforth p)`,
  REWRITE_TAC[XbackXforth] THEN
  MESON_TAC[xor1_involution;bijection_ifinvolution;
    bijection_inverseisbijection;bijection_composes]);;

let XbackXforth_cyclemin_twist = prove(
  `!p x. bijection p ==>
         cyclemin(XbackXforth p) (xor1(p(x)))
         = cyclemin(XbackXforth p) (p(xor1(x)))`,
  MESON_TAC[XbackXforth_bijection;XbackXforth_forward;
            cyclemin_shift]);;

let XbackXforth_lemma1 = prove(
  `!i:A->A p:A->A q:A->A.
    (!x. (i o i)x = x) /\ (!x. (q o p)x = x) /\ (!x. (p o q)x = x) ==>
    !x. ((p o i o q o i) o i o (p o i o q o i))x = i x`,
  REWRITE_TAC[o_DEF] THEN
  MESON_TAC[]);;
```

```
let XbackXforth_lemma2 = prove(
  `!i:A->A p:A->A q:A->A. i o i = I /\ q o p = I /\ p o q = I ==>
    (!x. (i o i)x = x) /\ (!x. (q o p)x = x) /\ (!x. (p o q)x = x)`,
  REWRITE_TAC[I_DEF;o_DEF] THEN
  MESON_TAC[]);;

let XbackXforth_lemma3 = prove(
  `!i:A->A p:A->A q:A->A. i o i = I /\ q o p = I /\ p o q = I ==>
    !x. ((p o i o q o i) o i o (p o i o q o i))x = i x`,
  REPEAT GEN_TAC THEN DISCH_TAC THEN
  ASSUME_TAC(SPECL[`i:A->A`;`p:A->A`;`q:A->A`]
                   XbackXforth_lemma1) THEN
  ASSUME_TAC(SPECL[`i:A->A`;`p:A->A`;`q:A->A`]
                   XbackXforth_lemma2) THEN
  ASM_MESON_TAC[]);;

let XbackXforth_lemma4 = prove(
  `!i:A->A p:A->A q:A->A. i o i = I /\ q o p = I /\ p o q = I ==>
    (p o i o q o i) o i o (p o i o q o i) = i`,
  MESON_TAC[XbackXforth_lemma3;EQ_EXT]);;

let XbackXforth_lemma5 = prove(
  `!i:A->A p:A->A q:A->A. i o i = I /\ q o p = I /\ p o q = I ==>
    p o i o q o i vanishesifonbothsidesof i`,
  MESON_TAC[XbackXforth_lemma4;vanishesifonbothsidesof]);;

let XbackXforth_lemma6 = prove(
  `!i:A->A. i o i = I ==> !p:A->A. bijection p ==>
    p o i o inverse p o i vanishesifonbothsidesof i`,
  MESON_TAC[XbackXforth_lemma5;
    bijection_inverse_composition5;
    bijection_inverse_composition6]);;

let XbackXforth_vanishesifonbothsidesof_xor1 = prove(
  `!p. bijection p ==> XbackXforth p vanishesifonbothsidesof xor1`,
  REWRITE_TAC[XbackXforth] THEN
  MESON_TAC[xor1_involution_o;XbackXforth_lemma6]);;

let XbackXforth_power_vanishesifonbothsidesof_xor1 = prove(
  `!p k:int. bijection p ==>
              (XbackXforth p pow_int k) vanishesifonbothsidesof xor1`,
  MESON_TAC[XbackXforth_bijection;
    XbackXforth_vanishesifonbothsidesof_xor1;
    vanishesifonbothsidesof_pow_int]);;
```

```
let XbackXforth_powpow_xor1 = prove(
  `!p k:int x. bijection p ==>
               (XbackXforth p pow_int k)
                 (xor1((XbackXforth p pow_int k) x)) = xor1 x`,
  REPEAT STRIP_TAC THEN
  ASSUME_TAC(SPECL[`p:num->num`;`k:int`]
                   XbackXforth_power_vanishesifonbothsidesof_xor1)
  THEN ASM_MESON_TAC[vanishesifonbothsidesof;o_THM]);;

let XbackXforth_even_avoids_xor1 = prove(
  `!p k:int x. bijection p ==>
    ~((XbackXforth p pow_int (k+k)) x = xor1 x)`,
  REPEAT STRIP_TAC THEN
  ASSUME_TAC(SPECL[`p:num->num`;`k:int`;`x:num`]
                  XbackXforth_powpow_xor1) THEN
  ASSUME_TAC(SPEC `p:num->num` XbackXforth_bijection) THEN
  ASSUME_TAC(ISPECL[`XbackXforth p`;`k:int`;`k:int`]
                    pow_int_plus) THEN
  ASSUME_TAC(ISPECL[`XbackXforth p`;`k:int`;
                    `(XbackXforth p pow_int (k:int))(x:num)`;
                    `xor1((XbackXforth p pow_int (k:int))(x:num))`
                   ]pow_int_cancel) THEN
  ASSUME_TAC(SPEC `(XbackXforth p pow_int (k:int)) (x:num)`
             xor1_distinct) THEN
  ASM_MESON_TAC[o_THM]);;

let XbackXforth_1_avoids_xor1 = prove(
  `!p x. bijection p ==> ~(XbackXforth p x = xor1 x)`,
  REWRITE_TAC[XbackXforth;o_DEF] THEN
  MESON_TAC[xor1_distinct;bijection_inverse_composition2]);;
```

```
let XbackXforth_odd_avoids_xor1 = prove(
  `!p k:int x. bijection p ==>
    ~((XbackXforth p pow_int ((k + &1) + k)) x = xor1 x)`,
  REPEAT GEN_TAC THEN DISCH_TAC THEN
  ASSUME_TAC(SPEC `p:num->num` XbackXforth_bijection) THEN
  ASM_SIMP_TAC[ISPECL[`XbackXforth p`;
                      `(k:int)+ &1`;`k:int`]pow_int_plus] THEN
  ASM_SIMP_TAC[o_THM] THEN
  ASM_SIMP_TAC[ISPECL[`XbackXforth p`;
                      `k:int`;`&1:int`]pow_int_plus] THEN
  ASM_SIMP_TAC[o_THM] THEN
  ASM_SIMP_TAC[ISPECL[`XbackXforth p`]pow_int_1] THEN
  ASSUME_TAC(SPECL[`p:num->num`;`k:int`;`x:num`]
                 XbackXforth_powpow_xor1) THEN
  ASSUME_TAC(ISPECL[`XbackXforth p`;`k:int`]pow_int_bijection) THEN
  ASSUME_TAC(ISPECL[`XbackXforth p:num->num`;`k:int`;
                  `(XbackXforth p)((XbackXforth p pow_int k)(x:num))`;
                  `xor1((XbackXforth p pow_int k)(x:num))`
                 ]pow_int_cancel) THEN
  ASSUME_TAC(SPECL[`p:num->num`;`(XbackXforth p pow_int k) (x:num)`]
                XbackXforth_1_avoids_xor1) THEN
  ASM_MESON_TAC[o_THM]);;

let XbackXforth_pow_avoids_xor1 = prove(
  `!p n:int x. bijection p ==>
    ~((XbackXforth p pow_int n) x = xor1 x)`,
  REPEAT STRIP_TAC THEN
  ASSUME_TAC(SPECL[`n:int`;`&2:int`]INT_DIVISION_SIMP) THEN
  DISJ_CASES_TAC(SPEC `n:int` INT_REM_2_CASES) THENL
  [
    ASSUME_TAC(INT_ARITH
             `(n:int) div &2 * &2 + &0 = n div &2 + n div &2`)
    THEN ASM_MESON_TAC[XbackXforth_even_avoids_xor1]
  ;
    ASSUME_TAC(INT_ARITH
             `(n:int) div &2 * &2 + &1 = (n div &2 + &1) + n div &2`)
    THEN ASM_MESON_TAC[XbackXforth_odd_avoids_xor1]
  ]);;
```

```
let XbackXforth_powpow_avoids_xor1 = prove(
  `!p n:int m:int x. bijection p ==>
    ~((XbackXforth p pow_int n) x =
      xor1 ((XbackXforth p pow_int m) x))`,
  REPEAT STRIP_TAC THEN
  ASSUME_TAC(SPEC `p:num->num` XbackXforth_bijection) THEN
  ASSUME_TAC(ISPECL[`XbackXforth p`;`m:int`]pow_int_bijection) THEN
  ASSUME_TAC(ISPECL[`XbackXforth p`;`(n:int)-(m:int)`;`m:int`]
                 pow_int_plus) THEN
  ASSUME_TAC(INT_ARITH `((n:int)-(m:int))+m = n`) THEN
  ASSUME_TAC(SPECL[`p:num->num`;`(n:int)-(m:int)`;
                  `((XbackXforth p pow_int m) x)`
                 ]XbackXforth_pow_avoids_xor1) THEN
  ASM_MESON_TAC[o_THM]);;

let XbackXforth_powpow2_avoids_xor1 = prove(
  `!p n:int m:int x. bijection p ==>
    ~((XbackXforth p pow_int n) x =
       (XbackXforth p pow_int m) (xor1 x))`,
  REPEAT STRIP_TAC THEN
  ASSUME_TAC(SPEC `p:num->num` XbackXforth_bijection) THEN
  ASSUME_TAC(ISPECL[`XbackXforth p`;`m:int`]pow_int_bijection) THEN
  ASSUME_TAC(ISPECL[`XbackXforth p`;`m:int`;
                  `(n:int)-(m:int)`]pow_int_plus) THEN
  ASSUME_TAC(INT_ARITH `m+((n:int)-(m:int)) = n`) THEN
  ASSUME_TAC(ISPECL[`XbackXforth p`;`m:int`;
                  `(XbackXforth p pow_int(n-m))x`;`xor1 x:num`
                 ]pow_int_cancel) THEN
  ASSUME_TAC(SPECL[`p:num->num`;`(n:int)-(m:int)`;`x:num`
                 ]XbackXforth_pow_avoids_xor1) THEN
  ASM_MESON_TAC[o_THM]);;

let XbackXforth_pow_xor1 = prove(
  `!p k:int x. bijection p ==>
    xor1((XbackXforth p pow_int k) x) =
    (XbackXforth p pow_int --k)(xor1 x)`,
  REPEAT STRIP_TAC THEN
  ASSUME_TAC(ISPECL[`p:num->num`]XbackXforth_bijection) THEN
  ASM_SIMP_TAC[ISPECL[`XbackXforth p`;`k:int`]pow_int_inverse] THEN
  ASSUME_TAC(ISPECL[`XbackXforth p`;`k:int`]pow_int_bijection) THEN
  ASSUME_TAC(SPECL[`p:num->num`;`k:int`;`x:num`]
                 XbackXforth_powpow_xor1) THEN
  ASSUME_TAC(ISPECL[`XbackXforth p pow_int k`;
                  `xor1 x:num`]bijection_inverse_composition1) THEN
  ASSUME_TAC(ISPECL[`XbackXforth p`;`k:int`;
                  `inverse(XbackXforth p pow_int k)(xor1 x)`;
                  `xor1((XbackXforth p pow_int k)x)`
                 ]pow_int_cancel) THEN
  ASM_MESON_TAC[]);;
```

```
let XbackXforth_pow_xor1_incycle = prove(
  `!p k:int x. bijection p ==>
    xor1((XbackXforth p pow_int k) x) IN
      cycle (XbackXforth p) (xor1 x)`,
  REPEAT STRIP_TAC THEN
  MATCH_MP_TAC(ISPECL[`XbackXforth p`;`xor1 x`;
                      `xor1((XbackXforth p pow_int k) x)`;`--k:int`
                     ]cycle_ifpow) THEN
  ASM_MESON_TAC[XbackXforth_pow_xor1]);;

let XbackXforth_pow_xor1_gecyclemin = prove(
  `!p x k:int. bijection p ==>
    cyclemin (XbackXforth p) (xor1 x) <=
      xor1((XbackXforth p pow_int k) x)`,
  REPEAT STRIP_TAC THEN
  REWRITE_TAC[cyclemin] THEN
  MATCH_MP_TAC MINIMAL_UBOUND THEN
  ASM_MESON_TAC[XbackXforth_pow_xor1_incycle;IN]);;

let XbackXforth_cyclemin_xor1_gecyclemin = prove(
  `!p x. bijection p ==>
    cyclemin (XbackXforth p) (xor1 x) <=
      xor1(cyclemin (XbackXforth p) x)`,
  REPEAT STRIP_TAC THEN
  ASSUME_TAC(SPECL[`XbackXforth p:num->num`;
                   `x:num`]cyclemin_aspow) THEN
  ASSUME_TAC(SPECL[`p:num->num`;`x:num`]
                 XbackXforth_pow_xor1_gecyclemin) THEN
  ASM_MESON_TAC[]);;

let XbackXforth_cyclemin_xor1_gecyclemin_reverse = prove(
  `!p x. bijection p ==>
    cyclemin (XbackXforth p) x <=
      xor1(cyclemin (XbackXforth p) (xor1 x))`,
  MESON_TAC[xor1xor1;XbackXforth_cyclemin_xor1_gecyclemin]);;

let XbackXforth_cyclemin_xor1_ne = prove(
  `!p x k:int. bijection p ==>
   ~(cyclemin(XbackXforth p) (xor1 x) = cyclemin(XbackXforth p) x)`,
  MESON_TAC[XbackXforth_powpow2_avoids_xor1;cyclemin_aspow]);;

let XbackXforth_cyclemin_xor1 = prove(
  `!p x. bijection p ==>
    cyclemin (XbackXforth p) (xor1 x) =
      xor1(cyclemin (XbackXforth p) x)`,
  MESON_TAC[xor1_sandwich;XbackXforth_cyclemin_xor1_ne;
            XbackXforth_cyclemin_xor1_gecyclemin_reverse;
            XbackXforth_cyclemin_xor1_gecyclemin]);;
```

```
let XbackXforth_cyclemin_xor1_double = prove(
  `!p j. bijection p ==>
         cyclemin (XbackXforth p) (2*j+1) =
         xor1(cyclemin (XbackXforth p) (2*j))`,
  MESON_TAC[XbackXforth_cyclemin_xor1;xor1_double]);;

let XbackXforth_cyclemin_xor1_double_parity = prove(
  `!p j. bijection p ==>
         ODD(cyclemin (XbackXforth p) (2*j+1)) =
         ~ODD(cyclemin (XbackXforth p) (2*j))`,
  MESON_TAC[XbackXforth_cyclemin_xor1_double;xor1_parity]);;

let XbackXforth_cyclemin_twist2 = prove(
  `!p x. bijection p ==>
         xor1(cyclemin(XbackXforth p) (p x))
         = cyclemin(XbackXforth p) (p(xor1 x))`,
  MESON_TAC[XbackXforth_cyclemin_twist;XbackXforth_cyclemin_xor1]);;

let XbackXforth_cyclemin_twist2_parity = prove(
  `!p x. bijection p ==>
         ~ODD(cyclemin(XbackXforth p) (p x))
         = ODD(cyclemin(XbackXforth p) (p(xor1 x)))`,
  MESON_TAC[XbackXforth_cyclemin_twist2;xor1_parity]);;

let XbackXforth_cyclemin_twist2_parity_double = prove(
  `!p j. bijection p ==>
         ~ODD(cyclemin(XbackXforth p) (p(2*j)))
         = ODD(cyclemin(XbackXforth p) (p(2*j+1)))`,
  MESON_TAC[XbackXforth_cyclemin_twist2_parity;xor1_double]);;

(* ----- XbackXforth cycle lengths *)

let XbackXforth_cycle_lemma1 = prove(
  `!p x y k. bijection p ==>
             y = ((XbackXforth p) pow_int k) x ==>
             ~(y IN cycle (XbackXforth p) (xor1 x))`,
  MESON_TAC[cycle_aspow;XbackXforth_powpow2_avoids_xor1]);;

let XbackXforth_cycle_lemma2 = prove(
  `!p x y. bijection p ==>
           y IN cycle (XbackXforth p) x ==>
           ~(y IN cycle (XbackXforth p) (xor1 x))`,
  MESON_TAC[cycle_aspow;XbackXforth_cycle_lemma1]);;

let XbackXforth_cycle_lemma3 = prove(
  `!p x. bijection p ==>
         DISJOINT (cycle (XbackXforth p) x)
                  (cycle (XbackXforth p) (xor1 x))`,
  MESON_TAC[IN_DISJOINT;XbackXforth_cycle_lemma2]);;
```

```
let XbackXforth_cycle_lemma4 = prove(
  `!p x. bijection p ==>
         cycle (XbackXforth p) x INTER
         cycle (XbackXforth p) (xor1 x) = EMPTY`,
  MESON_TAC[DISJOINT;XbackXforth_cycle_lemma3]);;

let XbackXforth_cycle_lemma5 = prove(
  `!p n k x. bijection p ==> p fixesge n ==> x < n ==>
             (p pow_int k) x < n`,
  MESON_TAC[fixesge_pow_int;pow_int_bijection;fixesge_lt]);;

let XbackXforth_cycle_lemma6 = prove(
  `!p b x y. bijection p ==> p fixesge (2*b) ==> x < 2*b ==>
             y IN cycle (XbackXforth p) x ==> y < 2*b`,
  MESON_TAC[XbackXforth_bijection;XbackXforth_fixesge;
            XbackXforth_cycle_lemma5;cycle_aspow]);;

let XbackXforth_cycle_lemma7 = prove(
  `!p b x. bijection p ==> p fixesge (2*b) ==> x < 2*b ==>
           cycle (XbackXforth p) x SUBSET range(2*b)`,
  REWRITE_TAC[SUBSET;range;IN_ELIM_THM] THEN
  MESON_TAC[XbackXforth_cycle_lemma6]);;

let XbackXforth_cycle_lemma8 = prove(
  `!p b x. bijection p ==> p fixesge (2*b) ==> x < 2*b ==>
           FINITE(cycle (XbackXforth p) x)`,
  MESON_TAC[XbackXforth_cycle_lemma7;FINITE_SUBSET;range_finite]);;

let XbackXforth_cycle_lemma9 = prove(
  `!p b x. bijection p ==> p fixesge (2*b) ==> x < 2*b ==>
           cycle (XbackXforth p) (xor1 x) SUBSET range(2*b)`,
  MESON_TAC[xor1_lteven;XbackXforth_cycle_lemma7]);;

let XbackXforth_cycle_lemma10 = prove(
  `!p b x. bijection p ==> p fixesge (2*b) ==> x < 2*b ==>
           cycle (XbackXforth p) x UNION
           cycle (XbackXforth p) (xor1 x) SUBSET range(2*b)`,
  MESON_TAC[XbackXforth_cycle_lemma7;XbackXforth_cycle_lemma9;
            UNION_SUBSET]);;

let XbackXforth_cycle_lemma11 = prove(
  `!p b x. bijection p ==> p fixesge (2*b) ==> x < 2*b ==>
           CARD(cycle (XbackXforth p) x
                UNION cycle (XbackXforth p) (xor1 x))
           <= 2*b`,
  MESON_TAC[XbackXforth_cycle_lemma10;range_card;
            range_finite;CARD_SUBSET]);;
```

```
let XbackXforth_cycle_lemma12 = prove(
  `!p b x. bijection p ==> p fixesge (2*b) ==> x < 2*b ==>
           CARD(cycle (XbackXforth p) x
                UNION cycle (XbackXforth p) (xor1 x))
           = CARD(cycle (XbackXforth p) x)
           + CARD(cycle (XbackXforth p) (xor1 x))`,
  MESON_TAC[CARD_UNION;XbackXforth_cycle_lemma4;
            XbackXforth_cycle_lemma8;xor1_lteven]);;

let XbackXforth_cycle_lemma13 = prove(
  `!p b x. bijection p ==> p fixesge (2*b) ==> x < 2*b ==>
           CARD(cycle (XbackXforth p) x)
           + CARD(cycle (XbackXforth p) (xor1 x)) <= 2*b`,
  MESON_TAC[XbackXforth_cycle_lemma11;XbackXforth_cycle_lemma12]);;

let XbackXforth_cycle_lemma14 = prove(
  `!p x. bijection p ==>
         IMAGE xor1 (cycle (XbackXforth p) x) SUBSET
         cycle (XbackXforth p) (xor1 x)`,
  REWRITE_TAC[IMAGE;SUBSET;IN_ELIM_THM] THEN
  MESON_TAC[cycle_aspow;XbackXforth_pow_xor1_incycle]);;

let XbackXforth_cycle_lemma15 = prove(
  `!p x. bijection p ==>
         cycle (XbackXforth p) (xor1 x) SUBSET
         IMAGE xor1 (cycle (XbackXforth p) x)`,
  REWRITE_TAC[IMAGE;SUBSET;IN_ELIM_THM] THEN
  MESON_TAC[cycle_aspow;XbackXforth_pow_xor1_incycle;xor1xor1]);;

let XbackXforth_cycle_lemma16 = prove(
  `!p x. bijection p ==>
         cycle (XbackXforth p) (xor1 x) =
         IMAGE xor1 (cycle (XbackXforth p) x)`,
  MESON_TAC[XbackXforth_cycle_lemma14;
            XbackXforth_cycle_lemma15;SUBSET_ANTISYM]);;

let XbackXforth_cycle_lemma17 = prove(
  `!p x b. bijection p ==> p fixesge (2*b) ==> x < 2*b ==>
           CARD(IMAGE xor1 (cycle (XbackXforth p) x)) =
           CARD(cycle (XbackXforth p) x)`,
  REPEAT STRIP_TAC THEN
  MATCH_MP_TAC CARD_IMAGE_INJ THEN
  ASM_MESON_TAC[xor1_involution;injection_ifinvolution;
                injection;XbackXforth_cycle_lemma8]);;
```

```
let XbackXforth_cycle_lemma18 = prove(
  `!p x b. bijection p ==> p fixesge (2*b) ==> x < 2*b ==>
          CARD(cycle (XbackXforth p) (xor1 x)) =
          CARD(cycle (XbackXforth p) x)`,
  MESON_TAC[XbackXforth_cycle_lemma16;XbackXforth_cycle_lemma17]);;

let XbackXforth_cycle_lemma19 = prove(
  `!p b x. bijection p ==> p fixesge (2*b) ==> x < 2*b ==>
          2*CARD(cycle (XbackXforth p) x) <= 2*b`,
  MESON_TAC[XbackXforth_cycle_lemma18;XbackXforth_cycle_lemma13;
            ARITH_RULE `2*n = n+n`]);;

let XbackXforth_cycle_size = prove(
  `!p b x. bijection p ==> p fixesge (2*b) ==> x < 2*b ==>
          CARD(cycle (XbackXforth p) x) <= b`,
  MESON_TAC[XbackXforth_cycle_lemma19;LE_MULT_LCANCEL;
            ARITH_RULE `~(2 = 0)`]);;

(* ----- firstcontrol *)

let firstcontrol_0 = prove(
  `!p. ~(firstcontrol p 0)`,
  MESON_TAC[ODD;firstcontrol;cyclemin_0;ARITH_RULE `2*0 = 0`]);;

let firstcontrol_subsetrange = prove(
  `!p b. bijection p ==> p fixesge(2*b) ==>
         firstcontrol p subsetrange b`,
  MESON_TAC[subsetrange;firstcontrol;XbackXforth_fixesge;
            XbackXforth_bijection;fixesge_cyclemin;
            fixesge_double_odd;GE;NOT_LT]);;

let firstcontrol_even = prove(
  `!p j. bijection p ==>
         ODD(Xif(firstcontrol p) (2*j)) =
         ODD(cyclemin(XbackXforth p)(2*j))`,
  MESON_TAC[firstcontrol;xif_double_parity]);;

let firstcontrol_odd_lemma1 = prove(
  `!p j. (firstcontrol p) ((2*j+1) DIV 2) =
         ODD(cyclemin(XbackXforth p)(2*j))`,
  MESON_TAC[firstcontrol;double_plus1_div2]);;

let firstcontrol_odd_lemma2 = prove(
  `!p j. bijection p ==>
         (firstcontrol p) ((2*j+1) DIV 2) =
         ~ODD(cyclemin(XbackXforth p)(2*j+1))`,
  REWRITE_TAC[firstcontrol_odd_lemma1] THEN
  MESON_TAC[XbackXforth_cyclemin_xor1_double_parity]);;
```

```
let firstcontrol_odd = prove(
  `!p j. bijection p ==>
         ODD(Xif(firstcontrol p) (2*j+1)) =
         ODD(cyclemin(XbackXforth p)(2*j+1))`,
  MESON_TAC[firstcontrol_odd_lemma2;xif_parity;
            ODD_DOUBLE;suc_isadd1]);;

let firstcontrol_parity = prove(
  `!p x. bijection p ==>
         ODD(Xif(firstcontrol p) x) =
         ODD(cyclemin(XbackXforth p) x)`,
  REPEAT STRIP_TAC THEN
  DISJ_CASES_TAC (SPEC `x:num` EVEN_OR_ODD) THENL
  [
    ASM_MESON_TAC[EVEN_EXISTS;firstcontrol_even]
  ;
    ASM_MESON_TAC[odd_isdoubleplus1;firstcontrol_odd]
  ]);;

(* ----- lastcontrol *)

let lastcontrol_lemma1 = prove(
  `!p b. bijection p ==> p fixesge(2*b) ==>
         Xif (firstcontrol p) o p fixesge(2*b)`,
  MESON_TAC[firstcontrol_subsetrange;xif_fixesge;fixesge_composes]);;

let lastcontrol_lemma2 = prove(
  `!p b x. bijection p ==> p fixesge(2*b) ==> x >= b ==>
           ~ODD(Xif (firstcontrol p) (p(2*x)))`,
  MESON_TAC[lastcontrol_lemma1;fixesge_double_odd;o_THM]);;

let lastcontrol_subsetrange = prove(
  `!p b. bijection p ==> p fixesge(2*b) ==>
         lastcontrol p subsetrange b`,
  REWRITE_TAC[subsetrange;lastcontrol] THEN
  MESON_TAC[lastcontrol_lemma2;GE;NOT_LT]);;

let lastcontrol_first_even = prove(
  `!p k. bijection p ==>
         ODD(Xif(firstcontrol p) (p(2*k))) =
         ODD(Xif(lastcontrol p) (2*k))`,
  REWRITE_TAC[xif_double_parity;lastcontrol]);;
```

```
let lastcontrol_first_odd = prove(
  `!p k. bijection p ==>
         ODD(Xif(firstcontrol p) (p(2*k+1))) =
         ODD(Xif(lastcontrol p) (2*k+1))`,
  REWRITE_TAC[xif_double_plus1_parity;lastcontrol] THEN
  REPEAT STRIP_TAC THEN
  ASM_MESON_TAC[XbackXforth_cyclemin_twist2_parity_double;
                firstcontrol_parity]);;

let lastcontrol_first = prove(
  `!p x. bijection p ==>
         ODD(Xif(firstcontrol p) (p x)) =
         ODD(Xif(lastcontrol p) x)`,
  REPEAT STRIP_TAC THEN
  DISJ_CASES_TAC (SPEC `x:num` EVEN_OR_ODD) THENL
  [
    ASM_MESON_TAC[EVEN_EXISTS;lastcontrol_first_even]
  ;
    ASM_MESON_TAC[odd_isdoubleplus1;lastcontrol_first_odd]
  ]);;

(* ----- middleperm *)

let middleperm_bijection = prove(
  `!p. bijection p ==> bijection(middleperm p)`,
  REWRITE_TAC[middleperm] THEN
  MESON_TAC[xif_involution;bijection_ifinvolution;
            bijection_composes]);;

let middleperm_fixesge = prove(
  `!p b. bijection p ==> p fixesge(2*b) ==>
         middleperm p fixesge(2*b)`,
  REWRITE_TAC[middleperm] THEN
  MESON_TAC[firstcontrol_subsetrange;lastcontrol_subsetrange;
            xif_fixesge;fixesge_composes]);;

let middleperm_parity = prove(
  `!p x. bijection p ==> ODD(middleperm p x) = ODD(x)`,
  REWRITE_TAC[middleperm;o_DEF] THEN
  MESON_TAC[xif_involution;involution;lastcontrol_first]);;

(* ----- review public definitions and theorems *)
```

```
let xor1 = xor1;;
let involution = involution;;
let surjection = surjection;;
let injection = injection;;
let bijection = bijection;;
let inverse = inverse;;
let pow_num = pow_num;;
let pow_int = pow_int;;
let cycle = cycle;;
let cyclemin = cyclemin;;
let partcycle = partcycle;;
let part2powcycle = part2powcycle;;
let fastcyclemin = fastcyclemin;;
let subsetrange = subsetrange;;
let fixesge = fixesge;;
let xif = xif;;
let XbackXforth = XbackXforth;;
let firstcontrol = firstcontrol;;
let lastcontrol = lastcontrol;;
let middleperm = middleperm;;
let cycle_shift = cycle_shift;;
let fastcyclemin_part2powcycle = fastcyclemin_part2powcycle;;
let fastcyclemin_works = fastcyclemin_works;;
let xor1_lteven = xor1_lteven;;
let xor1_div2 = xor1_div2;;
let xif_fixesge = xif_fixesge;;
let xif_involution = xif_involution;;
let XbackXforth_fixesge = XbackXforth_fixesge;;
let XbackXforth_bijection = XbackXforth_bijection;;
let XbackXforth_powpow_xor1 = XbackXforth_powpow_xor1;;
let XbackXforth_powpow_avoids_xor1 = XbackXforth_powpow_avoids_xor1;;
let XbackXforth_cycle_size = XbackXforth_cycle_size;;
let XbackXforth_cyclemin_xor1 = XbackXforth_cyclemin_xor1;;
let firstcontrol_subsetrange = firstcontrol_subsetrange;;
let lastcontrol_subsetrange = lastcontrol_subsetrange;;
let firstcontrol_0 = firstcontrol_0;;
let firstcontrol_parity = firstcontrol_parity;;
let lastcontrol_first = lastcontrol_first;;
let middleperm_bijection = middleperm_bijection;;
let middleperm_fixesge = middleperm_fixesge;;
let middleperm_parity = middleperm_parity;;
```