# Computing with Cyclic Subgroups of the Symmetric Group

Paul Tarau

**Abstract**

This is a short note on using large cyclical subgroups of the symmetric group $S_n$ to perform computations with the cycle representation of permutations as well as with the (equivalent) product of cyclical subgroups representation. We first compute the value of the Landau function for a given $n$, then we approximate its value with primorials and we define successor, predecessor functions and several bijections between representations, including one involving bijective base-$k$ numbers.

**Keywords:** large cyclical subgroups of $S_n$, Landau's number, primorials, alternative ways for doing arithmetic computations

## 1 Introduction

We assume familiarity with groups of permutations, cycle representation of permutations, modular arithmetic and the cyclic group $Z_n$. We refer to [1] for background in formation, including elementary proofs.

We will describe our algorithms as Haskell code, in the following module:

```
module CycArith where
import Data.List
```

We assume fluency with Haskell's arithmetics, lists and list comprehensions.

## 2 Landau's Number

We want to do arithmetic with a subset of the permutations in the Symmetric Group $S_n$. For that, the largest cyclic subgroup of $S_n$, the order of which is given by the *Landau number* **A000793** computed for each $n$ by Landau $g(n)$ function [1], with initial values listed at [2], is an ideal candidate. As cyclic groups are isomorphic with $Z_n$ which decomposes in a direct product of subgroups with order $p_i^k$ each with $p$ a prime factor of $n$ one can emulate operations with these permutations with modular arithmetic.

We parameterize our code to cover numbers at least up to `maxint`.

```
maxint = 100
```

To compute the *Landau number* we need all the partitions of $n$ as a sum of positive natural numbers.

```
partitions n = ps 1 n where
      ps x 0 = [[]]
      ps x y = [t:ts | t ← [x..y], ts ← ps t (y - t)]
```

Example:

```
*CycArith> partitions 5
[[1,1,1,1,1],[1,1,1,2],[1,1,3],[1,2,2],[1,4],[2,3],[5]]
```

The *Landau number* is then the maximum `lcm` of the elements of such a partition.

```
landau n =  m where
  ps = partitions n
  ls = map (foldl lcm 1) ps
  zs = zip ls ps
  m = foldl1 cmp zs where
    cmp (x,_) (y,ys) | x < y = (y,ys)
    cmp (x,xs) _ = (x,xs)
```

Example:

```
*CycArith> landau 19
(420,[3,4,5,7])
*CycArith> landau 42
(32760,[5,7,8,9,13])
```

The function `lans` finds first partition for which landau number is above $n$.

```
lans n = landau (i+1) where
  f (k,_xs) = k<n
  is = [0..]
  ls = takeWhile f (map landau is)
  ps = zip [0..] ls
  (i,_)=last ps
```

Example:

```
*CycArith> lans 42
(60,[3,4,5])
*CycArith> lans 100
(105,[3,5,7])
```

# 3 Emulating it with Primorials

A fairly close emulation of the partition giving the *Landau number* is given by the factors of a primorial (product of first primes) (see initial values at **A002110** in [2] with possibly some extra powers of 2.

The function `pris` returns the first partition of $n$ such that a primorial times $2^k$ is above $n$.

```
pris 0 = (1,[])
pris 1 = (1,[1])
pris n = (product rs,sort rs) where
  ps = tail primes
  xs = takeWhile (<n) (scanl (*) 1 ps)
  m = last xs
  es = map (2^) [0..]
  ys = takeWhile (<n) (zipWith (*) es (repeat m))
  rs = 2^(length ys):take (length xs-1) ps -- primorials
```

The code for the stream of prime numbers `primes` is given in the**Appendix**. Example:

```
*CycArith> pris 42
(60,[3,4,5])
*CycArith> pris 100
(120,[3,5,8])
```

We can extract an actual "canonical" generator of the cyclic group for $n$ from the largest cyclic subgroup or its primorial approximation, parameterized by `f`.

```
makeGenerator f n =   pss where
  psum xs = scanl (+) 0 xs
  toCycle from to = [from..to-1]
  (p:ps) =  psum (snd (f n))
  pss = zipWith toCycle (p:ps) ps
```

This gives a generator `lGenerator` for a cyclic subgroup of order given by the *Landau number* and as an alternative `pGenerator` when we approximate this with the easier to compute primorials `pGenerator`.

```
lGenerator = makeGenerator lans
pGenerator = makeGenerator pris
```

We chose from now one to work with `pGenerator` that works quite well for $maxint = 2^{128}$ or higher.

```
generator = pGenerator maxint

isGenerator x = x==generator
```

Example:

```
*CycArith> lGenerator 100
[[0,1,2],[3,4,5,6,7],[8,9,10,11,12,13,14]]
*CycArith> pGenerator 100
[[0,1,2],[3,4,5,6,7],[8,9,10,11,12,13,14,15]]
```

Alternatively one could chose

```
generator = lGenerator maxint
```

that our algorithm can efficiently compute only up to a few thousands for `maxint`.

We make a similar choice for `cycTemplate`

```
cycTemplate = pris maxint
```

instead of the alternative

```
cycTemplate = lans maxint -- more precise but slow to compute
```

Example:

```
*CycArith> cycTemplate
(120,[3,5,8])
```

# 4 Successor, Predecessor and Addition with Permutations in Cyclic Form

We are ready to implement successor and predecessor for a permutation in cyclic form as the functions `s` and `s'` that apply a right rotation `rrot` and respectively a left rotation `lrot` to each of the cycles.

```
s xss = map rrot xss

rrot [] = []
rrot (x:xs) = xs ++ [x]

s' xss = map lrot xss

lrot [] = []
lrot xs = last xs : (init xs)
```

Example:

```
*CycArith> generator
[[0,1,2],[3,4,5,6,7],[8,9,10,11,12,13,14,15]]
*CycArith> s generator
[[1,2,0],[4,5,6,7,3],[9,10,11,12,13,14,15,8]]
*CycArith> s' it
[[0,1,2],[3,4,5,6,7],[8,9,10,11,12,13,14,15]]
```

The function `n2p` converts a natural number to a permutation in cyclic form.

```
n2p 0 = generator
n2p n | n >0 = s (n2p (n-1))
```

The function `p2n` converts form a permutation in cyclic form back to a natural number.

```
p2n z | isGenerator z = 0
p2n x = 1+p2n (s' x)
```

One could then define arithmetic operations based on successor arithmetic as:

```
slowAdd x y | isGenerator x = y
slowAdd x y = s (slowAdd (s' x) y)
```

Example:

```
*CycArith> slowAdd (n2p 10) (n2p 20)
[[0,1,2],[3,4,5,6,7],[14,15,8,9,10,11,12,13]]
*CycArith> p2n it
30
```

# 5 Emulating computations in $Z_n$ seen as a direct product of subgroups

Given the isomorphism with $Z_n$ we emulate successor and predecessor in terms of $Z_n$ operations using modular arithmetic in each subgroup $Z_n$ is a product of.

The function `z` implements successor.

```
z ds = zipWith f bs ds where
  bs = snd cycTemplate
  f b d = (d+1) `mod` b
```

Note that each rotation of the cycles of a permutation is mapped to the corresponding modular addition of 1 in these subgroups.

The function `z'` implements predecessor.

```
z' ds = zipWith f bs ds where
  bs = snd cycTemplate
  f b d = (d'-1) `mod` b where
    d' = if d==0 then b else d
```

Example:

```
*CycArith> z [0,0,0]
[1,1,1]
*CycArith> z it
[2,2,2]
*CycArith> z it
[0,3,3]
*CycArith> z' it
[2,2,2]
*CycArith> z' it
[1,1,1]
*CycArith> z' it
[0,0,0]
```

# 6 Computing the Orbits

We define the generator zZero as the list with all components 0.

```
zZero = replicate (length (snd cycTemplate)) 0
```

Example:

```
*CycArith> zZero
[0,0,0]
```

Then a converter form natural numbers is simply the iterated successor `z`.

```
n2zs n = iter z zZero n
```

Its inverse `zs2n` applies predecessor `z'` until it reaches `sZero`.

```
zs2n zs | zs == zZero = 0
zs2n zs = 1+ zs2n (z' zs)
```

Example:

```
CycArith mapM_ print (map n2zs [0..7])
[0,0,0]
[1,1,1]
[2,2,2]
[0,3,3]
[1,4,4]
[2,0,5]
[0,1,6]
[1,2,7]

*CycArith> map (zs2n.n2zs) [0..7]
[0,1,2,3,4,5,6,7]
```

We can now compute the orbit of an element in $Z_n$ with the function `zOrbit`.

```
zOrbit = nub r where
  m=fst cycTemplate
  l=length (snd cycTemplate)
  r =  map n2zs [0..m-1]
```

It uses the function `iter` -that applies the function `f k` times.

```
iter _ x 0 = x
iter f x k = f (iter f x (k-1))
```

Example:

```
 zOrbit
[[0,0,0],[1,1,1],...,[1,3,6],[2,4,7]]
```

Likewise, we can compute the actual orbit for permutations in cyclic form under successor `s`.

```
n2ps n = iter s generator n

sOrbit = nub r where
  m=fst cycTemplate
  r =  map n2ps [0..m-1]
```

And we observe that both cover distinct elements up to and slightly above `maxint` as shown by the functions `permTest` and `znTest` that print out these orbits.

```
permTest = mapM_ print sOrbit
znTest =   mapM_ print zOrbit
```

Example:

```
mapM_ print sOrbit
[[0,1,2],[3,4,5,6,7],[8,9,10,11,12,13,14,15]]
[[1,2,0],[4,5,6,7,3],[9,10,11,12,13,14,15,8]]
...
[[1,2,0],[6,7,3,4,5],[14,15,8,9,10,11,12,13]]
[[2,0,1],[7,3,4,5,6],[15,8,9,10,11,12,13,14]]
```

The main question is now finding a faster conversion mechanism than the one relying on successor and predecessor.

# 7 Fast Conversion to List of Modular Remainders

We will solve it using the Chinese Reminder Theorem, which allows us to recover a number form the residues efficiently as shown by function `n2zs'`.

```
zs2n' zs | ok = crt zs bs where
  bs = snd cycTemplate
  ok = and (zipWith (<) zs bs)

n2zs' n | n<m = map f bs where
  (m,bs) = cycTemplate
  f b = n `mod` b
```

Its inverse, `n2zs'` finds to list of residues simply by applying the `mod` function to each of the elements of the primorial factors (or the Landau-number equivalent). Example:

```
*CycArith> mapM_ print (map n2zs' [0..7])
[0,0,0]
[1,1,1]
[2,2,2]
[0,3,3]
[1,4,4]
[2,0,5]
[0,1,6]
[1,2,7]

*CycArith> map (zs2n'.n2zs') [0..7]
[0,1,2,3,4,5,6,7]
```

We can extend this bijection to the actual permutations in cyclic form.

First we define a bijection between lists of residues and permutations by mapping the residues to rotate each cycle.

The function `zs2cp` generates the cyclic permutation corresponding to a list of residues.

```
zs2cp xs  =   zipWith rotR xs generator

rotR k xs | k≤genericLength xs && k≥0 =
  (genericDrop k xs) ++  (genericTake k xs)
rotL k xs = rotR (genericLength xs-k) xs
```

Its inverse is `zs2cp` generates the list of residues corresponding to a cyclic permutation. Note that it assumes that the permutations are in a canonical form with cycles, each represented as a rotation of a slice `[from..to]`.

```
cp2zs xss = map f xss where f xs = findElem (maximum xs) (reverse xs)

findElem x (y:xs) = if x==y then 0 else 1+findElem x xs
```

Example:

```
*CycArith> map (zs2n'.n2zs') [0..7]
[0,1,2,3,4,5,6,7]
*CycArith> zs2cp [0,0,0]
[[0,1,2],[3,4,5,6,7],[8,9,10,11,12,13,14,15]]
*CycArith> s it
[[1,2,0],[4,5,6,7,3],[9,10,11,12,13,14,15,8]]
*CycArith> cp2zs it
[1,1,1]
*CycArith> z it
[2,2,2]
*CycArith> zs2cp it
[[2,0,1],[5,6,7,3,4],[10,11,12,13,14,15,8,9]]
```

We can now express the bijection between natural numbers and permutations in cycle form as:

```
n2cp n = (zs2cp . n2zs') n

cp2n xss  = (zs2n' . cp2zs) xss
```

Example:

```
*CycArith> n2cp 0
[[0,1,2],[3,4,5,6,7],[8,9,10,11,12,13,14,15]]
*CycArith> cp2n it
0
*CycArith> n2cp 42
[[0,1,2],[5,6,7,3,4],[10,11,12,13,14,15,8,9]]
*CycArith> cp2n it
42
```

# 8 Modular arithmetic on permutation in cycle form

First we define a higher order function `zsOp` that we specialize for each arithmetic operation.

```
zsOp op  xs ys = zipWith3 f bs xs ys where
  bs = snd cycTemplate
  f b x y = (op x y) `mod` b
```

Addition and multiplications are defined as follows.

```
zsAdd  xs ys = zsOp (+) xs ys

zsMul  xs ys = zsOp (*) xs ys
```

Example:

```
*CycArith> n2zs' 42
[0,2,2]
*CycArith> n2zs' 8
[2,3,0]
*CycArith> zsAdd [0,2,2] [2,3,0]
[2,0,2]
*CycArith> zs2n' it
50
```

We can "borrow" these operations from the isomorphic residue list, to work on permutations in cycle form as follows.

```
cpOp op  xss yss = zs2cp (zsOp op (cp2zs xss) (cp2zs yss))

cpAdd xss yss = cpOp (+) xss yss

cpMul xss yss = cpOp (*) xss yss
```

Example:

```
*CycArith> n2cp 11
[[2,0,1],[4,5,6,7,3],[11,12,13,14,15,8,9,10]]
*CycArith> n2cp 7
[[1,2,0],[5,6,7,3,4],[15,8,9,10,11,12,13,14]]
*CycArith> cpMul [[2,0,1],[4,5,6,7,3],[11,12,13,14,15,8,9,10]]
            [[1,2,0],[5,6,7,3,4],[15,8,9,10,11,12,13,14]]
[[2,0,1],[5,6,7,3,4],[13,14,15,8,9,10,11,12]]
*CycArith> cp2n it
77
```

# 9 An alternative bijective mapping from natural numbers to their modular decomposition

Finally we define a converter to a lexicographically ordered representation, that enumerates residues in reverse lexicographic order.

## 9.1 Bijective base-$b$ digit extraction

The function `get_bdigit` extracts one bijective base-$b$ digit form a number $n$ [3]. Intuitively one can see it as reducing the information stored in $n$ by the information consumed for making a choice of a digit between 0 and $b-1$.

```
get_bdigit b n | n>0 =
  if d == 0 then (b-1,q-1) else (d-1,q) where
     (q,d) = quotRem n b
```

Dually, the function `put_bdigit` can be seen as adding to `m` the information stored in a digit $b$ from 0 to $b-1$.

```
put_bdigit b d m | 0 ≤ d && d < b = 1+d+b*m
```

Example:

```
*CycArith> get_bdigit 7 2019
(2,288)
*CycArith> put_bdigit 7 2 288
2019
```

The function `toBaseList` iterates the extraction of digits from $n$ using a list of bases $bs$ instead of a single base $b$.

```
toBaseList bs n  = f bs (skip+n) where
  skip = sum (init (scanl (*) 1 bs))
  f []  0 = []
  f(b:bs) n = d:ds where
    (d,m)=get_bdigit b n
    ds = f bs m
```

Note also that it increments $n$ with the number *skip* of the lists of digits shorter than the very last one, to ensure that exactly the "vectors" of length matching the length of $bs$ are extracted.

Dually, the function `fromBaseList` reverses this process by converting the list of digits $xs$ all assumed bounded by the bases in $bs$ into a single natural number $n$.

```
fromBaseList bs xs = (f bs xs)-skip where
  skip = sum (init (scanl (*) 1 bs))
  f [] [] = 0
  f (b:bs) (x:xs) = put_bdigit b x (f bs xs)
```

Example:

```
*CycArith> mapM_ print (map (toBaseList (snd cycTemplate)) [0..7])
[0,0,0]
[1,0,0]
[2,0,0]
[0,1,0]
[1,1,0]
[2,1,0]
[0,2,0]
[1,2,0]
```

The two functions define, for a given base list a bijection to tuples of the same length as the list and with elements between 0 and $b_i - 1$ for each element $b_i$ of the base list. Note that it enumerates the element tuples in lexicographic order (with rightmost digit most significant) and that this if different from the order induced by applying the successor repeatedly, which, on the other hand, matches the fast conversion provided by the Chinese Reminder Theorem.

# 10 Conclusion

Arithmetic computations in the a cyclic subgroup of $S_n$ can be expressed in terms of permutation operations, or equivalently, in terms of computations using modular arithmetic. With the primorial-based approximation as modular arithmetic happens independently in each subgroup $Z_n$ is a product of, one can realize this computation in parallel, (possibly with GPU-based execution).

# References


[1] W. Miller. The Maximum Order of an Element of a Finite Symmetric Group. *American Mathematical Monthly*, 94:497–506, 1987.

[2] N. J. A. Sloane. The On-Line Encyclopedia of Integer Sequences. 2018. Published electronically at https://oeis.org/.

[3] Paul Tarau. Compact Serialization of Prolog Terms (with Catalan Skeletons, Cantor Tupling and Gödel Numberings) . *Theory and Practice of Logic Programming*, 13(4-5):847–861, 2013.

# Appendix

Simple algorithm for computing the infinite stream of `primes`.

```
primes = 2 : filter is_prime [3,5..]

is_prime p = [p]==to_primes p

to_primes n|n>1 =
  to_factors n p ps where (p:ps) = primes

to_factors n p ps | p*p > n = [n]
to_factors n p ps | 0==n `mod` p =
  p : to_factors (n `div` p)  p ps
to_factors n p ps@(hd:tl) = to_factors n hd tl
```

Extended gcd-based algorithm for Chinese Remainder Theorem.

```
crt :: [Integer]→[Integer]→Integer
crt as ns  = head (dropWhile(<0) [s,s+prod..]) where
    s = sum [ div (x*y*prod) z | (x,y,z)← zip3 as ls ns ]
    prod = abs(product ns)
    ls = [extended_gcd x (div prod x) !! 1 |x← ns]
```

```
extended_gcd:: Integer→Integer→[Integer]
extended_gcd a b | mod a b == 0 = [0,1,b]
extended_gcd a b = [y,x-y*(a `div` b),z] where
  [x,y,z] = extended_gcd b (mod a b)
```