CDM

More Actions and Enumerations

Klaus Sutner
Carnegie Mellon University
www.cs.cmu.edu/~sutner

Battleplan

- Semigroup Actions and DFAs
- Enumeration, Ranking and Unranking
- Orders, Successors and Predecessors
- Exploiting Numeration Systems
- Generation and Isomorph Rejection

Prologue

Status Quo

We have seen Burnside/Polya/Redfield, a rather powerful machinery that solves certain counting problems.

For example, suppose we are interested in the collection S of all bracelets of length n with k colors (lists of length n with entries in $[k]$ under the dihedral group D_n).

Then S really is a partition

$$S = \{S_1, S_2, \ldots, S_N\}$$

where each S_i is a pattern, a class of lists.

The patterns are pairwise disjoint and their union is all of $[k]^n$.

Counting versus Generation

By the BPR method we can determine N, the number of patterns: for odd n we have

$$N = \frac{1}{2n} \sum_{d|n} \varphi(n/d)\, k^d + k^{(n+1)/2}/2$$

Computationally, the key point here is that we can determine N without having to enumerate all of $[k]^n$, or even S.

That's nice, but what if we need to choose one particular representative in each pattern, some $s_i \in S_i$?

We can obviously do this with brute-force (compute the whole space and then the equivalence relation on it).

But we would like an algorithm that uses N rounds to produce just a single representative in each pattern S_i, nothing else.

More Combinatorics

Generating objects of a certain type is yet another typical problem in combinatorics. In general, given some combinatorial structure (data structure if you prefer), we would like to solve at least the following problems.

Existence: Is there such a structure (for certain values of parameters)?
Counting: How many such structures are there?
Generation: How does one produce some/all these structures?

Obviously, existence is of interest only if the structures are relatively complicated. Typical example: Latin squares. Also note that we usually want constructive proofs.

We have seen some interesting approaches to counting, at least for objects related to group actions.

But how about generating algorithms? Is there any systematic way of constructing combinatorial objects such as necklaces, permutations, trees, functions, whatever?

DFAs and Actions

Transition Functions

A DFA is essentially given by its transition function, a matrix

$$\delta : Q \times \Sigma \longrightarrow Q$$

As far as pattern matching on strings is concerned it is convenient to extended this function from single letters to words:

$$Q \times \Sigma^\star \to Q$$
$$\delta(p, \varepsilon) = p$$
$$\delta(p, xa) = \delta(\delta(p, x), a)$$

This looks very similar to the right actions we encountered in Pólya-Redfield counting – except that the set of all words is not a group, just a monoid (or a semigroup if we do not allow the empty word – this turns out to be more usefull overall).

Or: A Right Action

Another way to think of the extended transition function of a DFA is as a right action:

$$Q \times \Sigma^\star \to Q$$
$$p \cdot \varepsilon = p$$
$$p \cdot xa = (p \cdot x) \cdot a$$

This really is an action since

$$p \cdot xy = (p \cdot x) \cdot y$$

for all words x and y.

Incidentally, this is a good example where a right action is more natural, a left action would be awkward.

Is there some more algebraic structure hiding?

The Transformation Semigroup

On occasion we have already interpreted the transition function as a collection of functions

$$\delta_a : Q \longrightarrow Q$$

where $a \in \Sigma$.

The set $\mathsf{Fct}(Q, Q)$ of all functions from Q to Q forms a semigroup (even a monoid) under composition, so we can consider the subsemigroup

$$\mathcal{T} = \langle\, \delta_a \mid a \in \Sigma \,\rangle \subseteq \mathsf{Fct}(Q, Q)$$

Definition 1. *This semigroup is the transformation semigroup of the DFA.*

Note that the size of this monoid is at most n^n where n is the size of the DFA.

A Homomorphism

What are the elements of $\mathcal{T}$?

All functions of the form $p \mapsto \delta(p, w)$ for some word w.

We write δ_w for these functions.

Proposition. *The map $\Phi : \Sigma^+ \longrightarrow \mathsf{Fct}(Q, Q)$, $w \mapsto \delta_w$ is a semigroup homomorphism. The image of this map is $\mathcal{T}$.*

What is the kernel relation of Φ?

It identifies two words u and v if, and only if, $\delta_u = \delta_v$.

In other words, the automaton is completely unable to distinguish between u and v, the transitions induced by these two words are exactly the same (not just from the initial state but from all states).

A Congruence

We write $u \sim v$ if $\delta_u = \delta_v$.

Proposition 1. *The equivalence relation $\sim$ is a congruence:*

$$u \sim v \iff \forall x, y\, (xuy \sim xvy).$$

Recall that congruences are the proper types of equivalence relations when on tries to form a quotient of an algebraic structure.

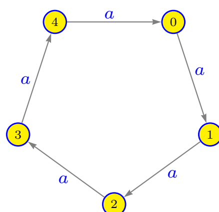
The quotient semigroup $\Sigma^+/\!\sim$ is isomorphic to the transformation semigroup $\mathcal{T}$.

This is very similar to our description of groups in terms of generators and a few equations: the "real" semigroup is $\mathcal{T}$, but it is often easier to compute instead with words representing the semigroup elements.

As before, multiplication corresponds to to concatenation, followed by simplification.

Example: Counting Modulo 5

Let's start with a very simple example, over a single letter alphabet.



It is clear that

$$\delta_{a^k}(p) = p + k \bmod 5$$

The Transformation Semigroup

The semigroup therefore has 5 (rather boring) elements

$$\delta_a = T(1,2,3,4,0)$$
$$\delta_{aa} = T(2,3,4,0,1)$$
$$\delta_{aaa} = T(3,4,0,1,2)$$
$$\delta_{aaaa} = T(4,0,1,2,3)$$
$$\delta_{aaaaa} = T(0,1,2,3,4)$$

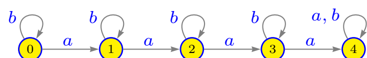
Note that in this case $\mathcal{T}$ is actually a group: we get $\mathbb{Z}_5$. The 5 congruence classes are

$$\{\, a^i \mid i = k \pmod 5 \,\}$$

where $k = 1,2,3,4,5$. There is only one simplification rule: $aaaaa \to \varepsilon$.

Example: Counting to Three

Counting a's over a two-letter alphabet.



In this case δ_b is the identity, but δ_a is a truncated successor function.

$$\delta_a(p) = \begin{cases} p+1 & \text{if } p < 4, \\ p & \text{if } p = 4. \end{cases}$$

The Transformation Semigroup

The semigroup therefore has 5 (rather boring) elements

$$\delta_a = T(1,2,3,4,4)$$
$$\delta_b = T(0,1,2,3,4)$$
$$\delta_{aa} = T(2,3,4,4,4)$$
$$\delta_{aaa} = T(3,4,4,4,4)$$
$$\delta_{aaaa} = T(4,4,4,4,4)$$

This time $\mathcal{T}$ is as far away from a group as possible: the only subsemigroup of $\mathcal{T}$ that is a group is the trivial $\{T(4,4,4,4,4)\}$.

Definition 2. *Semigroups that contain only trivial subgroups are called aperiodic.*

The Simplification Rules

Note that $T(4,4,4,4,4)$ plays the role of a *zero* in $\mathcal{T}$:

$$zx = xz = z \qquad \text{for all } x.$$

Zeros in $\mathcal{T}$ are closely connected to sinks in the automaton.

The simplification rules here can be written very elegantly as

$$a^4 \to 0, b \to 1$$

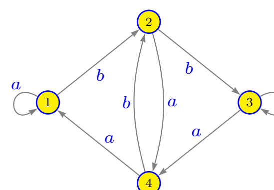
Exercise 1. *Determine the congruence classes for this example.*

Example: -2rd Symbol

Recall our standard example language:

$$L_{-2,a} = \{\, x \in \{a,b\}^\star \mid x_{-2} = a \,\} = \Sigma^\star a \,\{a,b\}\,.$$

We know a DFA for this language (actually, this is the minimal DFA):



The Transformation Semigroup

The two generators are

$$\delta_a = T(1,4,4,1)$$
$$\delta_b = T(2,3,3,2)$$

and the full semigroup consists of these plus the four constants

$$\delta_{aa} = T(1,1,1,1)$$
$$\delta_{ab} = T(2,2,2,2)$$
$$\delta_{ba} = T(3,3,3,3)$$
$$\delta_{bb} = T(4,4,4,4)$$

The constants appear since any word of length 2 or more sends all states to the same state (these are right-nulls in the semigroup).

The Congruence

The congruence $\sim$ is also very easy to determine in this case: there are six classes

$$a, b, \Sigma^\star aa, \Sigma^\star ab, \Sigma^\star ba, \Sigma^\star bb.$$

What are the simplification rules for words in general?

If we apply the spanning tree algorithm as in the group case (which is actually easier here, we don't have to worry about inverses) we see that there are 8 rules:

$$aaa \to aa, baa \to aa, aab \to ab, bab \to ab, \ldots, bbb \to bb$$

Applying these rules we can reduce any non-empty word to one of the six witnesses a, b, aa, ab, ba, bb.

Importing Algebra

At first glance, it might seem like we are just introducing more convoluted terminology to obfuscate a perfectly simple topic.

But algebra is actually extremely useful to study the structure of finite state machines and there are some deep results about decompositions of machines into simple components that depend on this approach.

In particular, there is a famous result, the Krohn-Rhodes Theorem which says that all transformation semigroups can be decomposed (in a certain complicated way) into just aperiodic ones and groups.

This result provides a rather surprising solution for the problem of constructing finite state machines from simpler components, albeit not a very practical one.

Alas, we don't have time to dig any deeper into this.

Exercises

Exercise 2. *Compute the transformation semigroup for the minimal DFA of the language $\Sigma^\star ab\Sigma^\star$. Determine the congruence classes and the simplification rules.*

Exercise 3. *Compute the transformation semigroup for the minimal DFA of the language "even number of a's and even number of b's". Determine the congruence classes and the simplification rules.*

Exercise 4. *Consider some other, simple minimal DFAs and determine their transformation semigroups.*

Exercise 5. *Suppose DFA M is a product $M_1 \times M_2$. What can you say about its transformation semigroup?*

Enumeration

Generation

So suppose we have some collection S of combinatorial objects and we know that its size is n. Hence there is a bijection between S and $[n]$ (though our existence proof may be highly non-constructive, think BPR).

We would like to have an algorithm that enumerates some or perhaps all of the elements of S. One way of doing this is to find an explicit bijection

$$f : S \to [n]$$

There are $n!$ candidates, but for f to be useful one needs to find a bijection such that

- f is easy to compute,
- f^{-1} is easy to compute, and
- f places the elements of S into some natural order.

Ranking and Unranking

Definition 3.
A bijection $f : S \to [n]$ is called a ranking function*, and*
$f^{-1} : [n] \to S$ *the corresponding* unranking functions.

We often write rk for a ranking function, and urk for an unranking function, perhaps with subscripts as in rk_S.

Needless to say, we need

$$\mathrm{rk} \circ \mathrm{urk} = \mathrm{urk} \circ \mathrm{rk} = I$$

so we can go back and forth between ranks and objects.

0-Indexing and 1-Indexing

On occasion it is more convenient to consider ranking and unranking function between S and

$$(n) = \{0, 1, \ldots, n-1\}$$

rather than

$$[n] = \{1, 2, \ldots, n\}.$$

It should always be clear from context whether 0-indexing is used or 1-indexing.

Example 1. *Ranking binary lists of length n is naturally done with 0-indexing.*

CATs

Often the goal is to come up with algorithms whose time complexity is proportional to the number of objects generated.

Ideally, each object should require a constant number of steps (where one may have to be a bit creative regarding the meaning of "step"; Turing machines may be too restrictive).

If a constant number per object does not work, the next interesting class of algorithms is *constant amortized time (CAT)*: over the whole run of the algorithm only a constant number of operations are needed per object – a few objects may require relatively much effort, but others require little.

Exercise 6. *Explain the importance of amortized analysis for splay trees.*

Binary Counter

Consider the problem of generating all binary lists of length n.

If there is no restriction on the order, we can think of this simply as the problem of counting from 0 to $2^n - 1$. Of course, a simple binary counter will do the job, thank you very much.

Suppose we charge one step for each bit that needs to be flipped. If there are $k - 1 < n$ trailing 1's then stepping the counter will cost k steps, a non-constant cost.

But expensive steps are infrequent, over the whole run of the algorithm we need only

$$\sum_{i=1}^{n} i \cdot 2^{n-i} = 2^{n+1} - n - 2$$

steps (n more if we reset all bits to 0).

Hence the natural algorithm is CAT.

Pretty Pictures

Binary expansions for $x = 0, \ldots, 2^6 - 1$, packed into a 6×64 matrix.

The horizontal difference pattern for this matrix: xor between consecutive columns (sometimes called a ruler function).

Exercise 7. *Give a geometric proof that binary counting is CAT based on the second picture.*

Ranking and Random Objects

A lot of work has gone into the construction of good pseudo-random number generators: given some bound n, generate a random integer r in the range 1 to n, uniformly distributed. Equivalently, repeatedly pick a single bit 0 or 1 with probability 1/2.

In conjunction with an unranking function urk this suffices to generate uniformly random object in S:

```
r = rand(n);                 // random integer
return  urk(r);              // random object
```

It can be quite tricky to do this directly in S without producing bias: the selection process should pick each object in S with the same probability.

Direct Approach: Random Permutations

Of course, for some combinatorial objects we don't need to use ranking machinery, we can randomly generate them directly. E.g., here is a standard method to generate random permutations of $[n]$, given a random number generator `rand(k)` that returns an integer in $[k]$, drawn uniformly at random.

```
for k = 1, .., n do
        a[i] = i;

for k = n, .., 2 do
        r = rand(k);
        swap  a[k], a[r];
```

Exercise 8. *Alternatively, we could use a source for random reals in $[0, 1]$, generate a list $((r_1, 1), (r_2, 2), \ldots, (r_n, n))$, sort in lex order and then drop the first component. Check that both methods produce uniformly distributed random permutations.*

Ranking in Ordered Sets

Rankings are closely related to orders.

Suppose the set A is totally ordered, so we have a fixed enumeration

$$A = a_0, a_1, \ldots, a_{k-1}.$$

Definition 4. *For any subset B of A and element $a \in B$ define the rank $\mathrm{rk}(a, B)$ of to be the position of a in B.*

Example 2. *Let $A = a, b, c, d$. Using 0-indexing we have*

$\mathrm{rk}(c, A) = 2$,
$\mathrm{rk}(c, \{b, c, d\}) = 1$ *and*
$\mathrm{rk}(c, \{c\}) = 0$.

And Back

Of course, given any set A together with a ranking function $\mathrm{rk} : A \to (n)$ we can always define a corresponding total order:

$$a <_A b \iff \mathrm{rk}(a) <_{\mathbb{N}} \mathrm{rk}(b).$$

It still is a good idea to distinguish the two notions.

Often, there is a natural order on a set of combinatorial objects (such as lexicographic order on words), and one would like to find a good program for the corresponding ranking and unranking functions.

Rankings and Order

Successors and Predecessors

Given a ranking/unranking pair one can always define the successor and predecessor of an object x in S:

successor: $\mathrm{urk}(\mathrm{rk}(x) + 1)$
predecessor: $\mathrm{urk}(\mathrm{rk}(x) - 1)$

Here we assume 0-indexing and wrap around at 0 and $n - 1$.

It is often challenging to find a computational short-cut: given x, compute the successor/predecessor as efficiently as possible, without resorting to the ranking machinery.

Likewise, we always have a natural notion of predecessor and successor whenever S is totally ordered (and finite).

Programs

Note that in a certain sense it is trivial to find a program that computes, say, rk.

We can just hardwire a table into the code and perform a simple table look-up to get the rank of an element.

a	b	c	$\ldots$	z
0	1	2	$\ldots$	123

The problem is that in many applications this table would be huge (often larger than the physical universe).

This leads naturally to the notion of *program size complexity*: one is interested the size of the program rather than running time or memory requirements (more on this later).

So we need to find a small program to compute the rank.

Cheap Example

Let S be all subsets of A, a set of cardinality k.

Then $n = 2^k$ and we can use binary expansions, padded to k bits, for ranking and unranking.

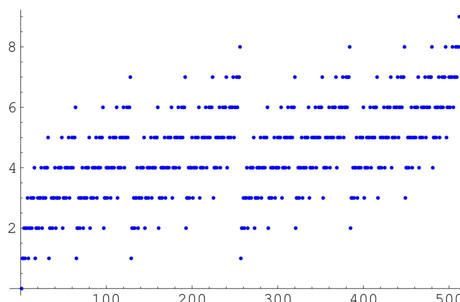
$$\{a, d\} \subseteq \{a, b, c, d\} \longrightarrow (1, 0, 0, 1) \longrightarrow 9$$

Note that this ranking function clobbers other properties that one might be interested in.

For example, there is no good connection between the rank and the size of the corresponding subset.

The next pictures shows the sizes of all subsets of $[9]$ ordered according to this ranking.

Rank versus Size



Less Cheap Example

Using essentially the same approach we can rank and unrank any Cartesian product

$$S = (r_k) \times (r_{k-1}) \times \ldots \times (r_2) \times (r_1)$$

Here $n = \prod r_i$ and we use the ranking function

$$\mathrm{rk}_{\mathsf{CP}}(a_k, \ldots, a_1) = \sum a_i R_i$$

where $R_i = \prod_{j<i} r_j$.

The special case $r_j = r$ is just standard base r notation. In the general case we use a non-standard numeration system.

Medium Priced Example

How about injective functions

$$f : [m] \to [n]$$

where $m \leq n$. Alternatively, we are looking at repetition-free lists of length m with entries in $[n]$.

The number of such functions is the *falling factorial*

$$n^{\underline{m}} = n(n-1)\ldots(n-m+1)$$

with the special case

$$n^{\underline{n}} = n!$$

the number of permutations of n points.

Making a Connection

Perhaps one could somehow establish a correspondence between these injective functions and the Cartesian product

$$S = (n) \times (n-1) \times \ldots \times (n-m+1)$$

which, it so happens, has cardinality $n^{\underline{m}}$?

$f(1)$ selects an arbitrary element a of (n). The order rank of a is any number in (n).

But then $f(2)$ selects an arbitrary element of $(n) - \{a\}$ with order rank in $(n-1)$.

And so on.

▶ Central Idea: If two sets have the same cardinality, there must be a good reason for this.

Ranking Injections

Definition 5. *Let $\vec{a} = (a_1, a_2, \ldots, a_m) \in (n)^m$ be a repetition-free list. Define a new list $\vec{b} = \mathsf{lower}(\vec{a})$ by*

$$b_i = \mathrm{rk}(a_i, (n) - \{a_1, \ldots, a_{i-1}\})$$

for $i = 1, \ldots, m$.

Note that

$$\mathsf{lower}(\vec{a}) \in S = (n) \times (n-1) \times \ldots \times (n-m+1)$$

Proposition 2. *The map*

$$\mathsf{lower} : \mathsf{Inj}(m, n) \to S$$

is a bijection.

Laziness

But then we can reuse the ranking function for Cartesian products to get a ranking function for $\mathsf{Inj}(m, n)$:

$$\mathrm{rk} : \mathsf{Inj}(m, n) \to (n^{\underline{m}})$$
$$\mathrm{rk}(\vec{a}) = \mathrm{rk}_{\mathsf{CP}}(\mathsf{lower}(\vec{a}))$$

Great, but how do we unrank?

We need a dual function raise that converts back from S to $\mathsf{Inj}(m, n)$.

Any bets?

Raise

We need a function that undoes the lower operation. Guess what it might be called.

Definition 6. *Let $\vec{b} \in S$ and define a new list $\vec{a} = \mathsf{raise}(\vec{b}) \in (n)^m$ by*

$$a_i = \mathrm{urk}(b_i, (n) - \{b_1, \ldots, b_{i-1}\})$$

Proposition 3. *The unranking function is given by*

$$\mathrm{urk} : (n^{\underline{m}}) \to \mathsf{Inj}(m, n)$$
$$\mathrm{urk}(x) = \mathsf{raise}(\mathrm{urk}_{CP}(x))$$

Headache?

Example 3. *Here are the ranks for $m = 2$, $n = 5$.*

$(0,1)$	0	$(0,3)$	10
$(1,0)$	1	$(1,3)$	11
$(2,0)$	2	$(2,3)$	12
$(3,0)$	3	$(3,2)$	13
$(4,0)$	4	$(4,2)$	14
$(0,2)$	5	$(0,4)$	15
$(1,2)$	6	$(1,4)$	16
$(2,1)$	7	$(2,4)$	17
$(3,1)$	8	$(3,4)$	18
$(4,1)$	9	$(4,3)$	19

Special Case: Permutations

We get permutations in the case $n = m$. In this case, there is a good combinatorial interpretation of the lower operation: it can be thought of as producing factorial digits.

Definition 7. *The factorial representation of an integer $x \geq 0$ has the form*

$$x_k \cdot k! + x_{k-1} \cdot (k-1)! + \ldots + x_2 \cdot 2! + x_1 \cdot 1!$$

where $0 \leq x_i \leq i$.

Note that this is yet another case where a ranking/unranking problem is solved by means of an apropriate *numeration system*.

To generate all binary lists we can use a binary counter, to generate all permutations we can use a factorial counter (alas, that's a bit harder to implement).

Example

Example 4. *Here is a table of the represenation of all numbers x, $0 \leq x < 24$, padded to 3 factorial digits.*

0	0,0,0	12	2,0,0
1	0,0,1	13	2,0,1
2	0,1,0	14	2,1,0
3	0,1,1	15	2,1,1
4	0,2,0	16	2,2,0
5	0,2,1	17	2,2,1
6	1,0,0	18	3,0,0
7	1,0,1	19	3,0,1
8	1,1,0	20	3,1,0
9	1,1,1	21	3,1,1
10	1,2,0	22	3,2,0
11	1,2,1	23	3,2,1

Note that we only need 3 factorial digits for permutations of $[4]$. Why?

An Unranking Algorithm

There are several ranking/unranking algorithms for permutations in the literature. Here is one standard approach which produces a somewhat strange ordering. It is easier to use 0-indexing for this, so below we deal with permutations on (n).

```
for k = 0, .., n-1 do
        a[i] = i;
unrank( a[], n, r );          // r == rank

unrank( a[], k, r )
        if( k > 0 )
              swap a[k-1], a[r mod k];
              unrank( a, k-1, r div k );
```

This is linear in n (assuming uniform cost function for the arithmetic operations).

A Ranking Algorithm

For the corresponding ranking algorithm we first compute the inverse permutation `ainv`. Running time is again linear in n.

```
for k = 0, .., n-1 do
        ainv[a[i]] = i;
rank( a[], n );

rank( a[], k )
        if( k==1 ) return 0;
        s = a[k-1];
        swap a[k-1], a[ainv[k-1]];
        swap ainv[s], ainv[k-1];
        return  s + k * rank( k-1 );
```

Exercise 9. *Prove that these two ranking/unranking functions really work.*

Back to Lower/Raise

How does one actually compute lower and raise?

$$(2,3,0,1,4) \longrightarrow (2,2,0,0) \longrightarrow 60$$

Our definition is in terms of the ranking function on an ordered set, but the code becomes quite simple if we think of the tuples as arrays of integers.

Ready?

Lower

For example, lower can be implemented by a simple nested loop.

```
for( i = 1; i <= n; i++ )
for( j = i+1; j <= n; j++ )
     if( a[i] < a[j] )  a[j]--;
```

In the end, drop the last element from the array.

Annoyingly, this is quadratic time. The situation for raise is entirely analogous.

Question: Can lower and raise be implemented in time $o(n^2)$?

Enumerating Orbits

Ranking and unranking is fine for the set of all permutations, but what if we want to generate only the permutations of a given list $\vec{a} = (a_1, a_2, \ldots, a_m)$ where not all elements are necessarily distinct.

In other words, we want the orbit of $\vec{a}$ under the natural action of the symmetric group on m letters.

Example 5. *Let $\vec{A} = (a,b,b,c)$.*

Then the stabilizer of $\vec{A}$ is $\{I, (2,3)\}$, a subgroup of size 2 in $\mathbb{S}_4$.

Hence the orbit has size $12 = 4!/2$, and with a lot of computing in $\mathbb{S}_4$ we could produce it using Pólya's machinery: we would need to find representatives $g_1, \ldots, g_{12}$ for the cosets to get the orbit $g_1 \cdot \vec{A}, g_2 \cdot \vec{A}, \ldots, g_{12} \cdot \vec{A}$

Example, contd.

In one-line notation the representatives are

$$T(1,2,3,4), T(1,2,4,3), T(1,3,4,2), T(2,1,3,4), T(2,1,4,3), T(2,3,4,1),$$
$$T(3,1,2,4), T(3,1,4,2), T(3,2,4,1), T(4,1,2,3), T(4,1,3,2), T(4,2,3,1)$$

This produces the following 12 variants of (a,b,b,c):

$$(a,b,b,c), (a,b,c,b), (a,c,b,b), (b,a,b,c), (b,a,c,b), (b,b,a,c),$$
$$(b,b,c,a), (b,c,a,b), (b,c,b,a), (c,a,b,b), (c,b,a,b), (c,b,b,a)$$

Successors

A better way is to find a *successor function* next that enumerates the orbit by iteration:

$$\mathrm{orb}_{\mathsf{next}}(\vec{a}) = \{\, \mathsf{next}^i(\vec{a}) \mid i < 12 \,\}$$

Is there such a successor function that works for all possible lists? Yes:

- Find the largest position i such that $a_i < a_{i+1}$.
- Find the least a_j in positions $i < j \leq n$ larger than a_i.
- Swap a_i and a_j.
- Sort the tail-end $a_{i+1}, \ldots, a_n$ of the permutation.

If the whole permutation is descending, sort the whole thing.

This operation is linear time if we use a fast sorting algorithm (distribution counting).

Example

It is not at all clear that this works. Here is a "proof" by example.

Example 6.

1	2	2	4
1	2	4	2
1	4	2	2
2	1	2	4
2	1	4	2
2	2	1	4
2	2	4	1
2	4	1	2
2	4	2	1
4	1	2	2
4	2	1	2
4	2	2	1

Note that the orbit appears in lex order.

Strictly Increasing Functions

Let's try another combinatorial class: *strictly increasing functions (SIF)* from (m) to (n). Alternatively, we are dealing with strictly increasing lists of length m with elements in (n). Of course, we need $m \leq n$ for these functions to exist.

How many are there?

Each SIF can be identified with its range, and thus with a subset of (n).

Hence there are $\binom{n}{m}$ SIFs.

Following the previous examples, it is tempting to look for a good representation for numbers $0 \leq x < \binom{n}{m}$.

A wild guess would be that binomial coefficients might be involved. From our previous experience with binary and factorial representations, be probably need a new "binomial representation".

Binomial Representation

Lemma 1. *Every number $0 \leq x < \binom{n}{m}$ has a unique representation of the form*

$$x = \binom{x_1}{1} + \binom{x_2}{2} + \ldots + \binom{x_m}{m}$$

given the constraint $0 \leq x_1 < x_2 < \ldots < x_m < n$.

Definition 8. *The x_i above are the* binomial digits *of x.*

Note that unlike with the classical base B representations and the factorial representation from above it is not entirely clear how one would compute binomial digits.

Exercise 10. *Figure out how to compute binomial digits.*

Ranking

So suppose $f : [m] \to (n)$ is strictly increasing. The ranking function is defined by

$$\mathrm{rk}(f) = \binom{f(1)}{1} + \binom{f(2)}{2} + \ldots + \binom{f(m)}{m}$$

The unranking function here is in essence just the computation of the binomial digits.

Note that $f = (0, 1, \ldots, m-1)$ has rank 0 and $f = (n-m, \ldots, n-2, n-1)$ has rank $\binom{n}{m} - 1$.

Example

Example 7. *Here is the case $m = 3$, $n = 6$.*

$(0,1,2)$	0	$(0,1,5)$	10
$(0,1,3)$	1	$(0,2,5)$	11
$(0,2,3)$	2	$(1,2,5)$	12
$(1,2,3)$	3	$(0,3,5)$	13
$(0,1,4)$	4	$(1,3,5)$	14
$(0,2,4)$	5	$(2,3,5)$	15
$(1,2,4)$	6	$(0,4,5)$	16
$(0,3,4)$	7	$(1,4,5)$	17
$(1,3,4)$	8	$(2,4,5)$	18
$(2,3,4)$	9	$(3,4,5)$	19

Note that the order here is a bit strange.

More Binomial Representations

We can generalize the binomial representation from above slightly.

Lemma 2. *For every positive x and m there is a unique representation of the form*

$$x = \binom{x_m}{m} + \binom{x_{m-1}}{m-1} + \ldots + \binom{x_t}{t}$$

given the constraint $x_m > x_{m-1} > \ldots > x_t \geq t$.

Note that this constraint makes it impossible to represent 0, otherwise this decomposition is very similar to the one above.

Exercise 11. *Prove existence of the representation by showing that the natural greedy algorithm works. Then establish uniqueness.*

Colex Order

Lex order (or lexicographic order) is defined by comparing the letters of a string from left to right.

Definition 9. *If we read the string instead from right to left, the corresponding order is called colex order.*

Proposition 4. *The ranking procedure for SIFs from above organizes the functions in colex order.*

Exercise 12. *Prove the last proposition.*

Exercise 13. *Find a rank for a non-decreasing functions.*

Necklaces

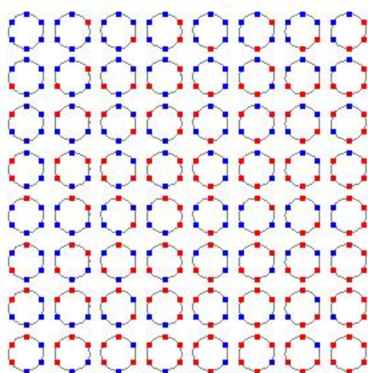
Back to the problem of generating representatives for each equivalence class in some group action. In general, this is a hard problem, but for necklaces where the group in question is just the cyclic group there is a good solution.

Definition 10. *A k-ary necklace is a cyclic word over a k-letter alphabet.*

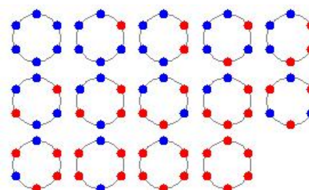
If you prefer, think about lists of some length n with entries in $[k]$, under the cyclic group $\mathbb{Z}_n$ (no reflections, that's only for bracelets).

It is customary to pick a representative in each class: the lexicographically least (linear) word in the equivalence class. The representative is then simply called a necklace.

Example: $n = 6$, $k = 2$



Necklaces Only



There are 64 binary words of length 6, but only 14 binary necklaces:

$$\mathsf{neck}(n, k) = \frac{1}{n} \sum_{d|n} \varphi(d) k^{n/d}$$

Generating Necklaces

We are looking for a good algorithm to generate these necklaces. As a data structure, we are dealing with a simple array of integers in the range 0 to $k - 1$.

One particularly simple algorithm is based on enumerating a slightly larger class of objects.

Definition 11. *A word w is a pre-necklace if it is a prefix of a necklace.*

Example 8. *All binary pre-necklaces of length 4. Note that 0010 is not a necklace.*

$$0000, 0001, 0010, 0011, 0101, 0110, 0111, 1111$$

If we manage to generate all pre-necklaces of length n we have in particular all necklaces, but we still need some mechanism to weed out the non-necklaces.

Lyndon Words

Definition 12. *A non-empty word w is primitive if it is not of the form x^i for any x shorter than w. The root of a word w is the shortest word x such that $x^i = w$ and the exponent i is then the repetition factor of w.*

Thus a primitive word is its own root and has repetition factor 1. A word of prime length is primitive unless it is of the form a^n.

Definition 13. *A Lyndon word is a representative for a necklace that is also primitive.*

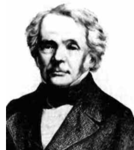
Proposition 5. *The number of Lyndon words of length n over a k symbol alphabet is*

$$\frac{1}{n} \sum_{d|n} \mu(n/d) k^d.$$

where μ is the Möbius function.

Möbius Function

$$\mu(n) = \begin{cases} 1 & \text{if } n = 1, \\ (-1)^r & \text{if } n = p_1 \dots p_r \text{, all distinct primes,} \\ 0 & \text{if } n \text{ is divisible by } p^2 \text{, } p \text{ a prime.} \end{cases}$$

Herr Möbius'  invention is a bit strange. Since $\mu(n)$ depends very much on the prime decomposition of n its behavior is rather erratic.

But it is an exceedingly useful tool for certain types of counting problems.

Back To Necklaces

Define the Lyndon length of w to be

$$\text{lyn}(w) = \text{ length of longest prefix of } w \text{ that is Lyndon}$$

Note that $\text{lyn}(w) \geq 1$. This will turn out to be the right tool to filter out non-necklaces from pre-necklaces.

Lemma 3. *Consider the alphabet $\Sigma_k = \{0, 1, \dots, k-1\}$. Let w be a pre-necklace and set $p = \text{lyn}(w)$. Then wa is a pre-necklace if, and only if,*

- *$a = w_{-p}$, in which case $\text{lyn}(wa) = p$, or*
- *$a > w_{-p}$, in which case $\text{lyn}(wa) = |wa|$.*

Clearly each letter $a \in \Sigma_k$ is a pre-necklace and $\text{lyn}(a) = 1$, so the lemma translates directly into an iterative algorithm for the construction of pre-necklaces.

And Necklaces?

It is easy to keep track of the Lyndon length together with the corresponding pre-necklaces.

To get rid of non-necklaces one can then exploit the following little fact to get a CAT algorithm for the construction of necklaces.

Proposition 6. *Let w be a pre-necklace of length n and Lyndon length p. Then w is a necklace if, and only if, p divides n.*

Exercise 14. *Prove the last proposition.*

Isomorph Rejection

Another closely related computational problem is to determine whether two given objects are equivalent under some given action.

Of course, the point is to avoid having to enumerate the whole orbit (or substantial parts thereof) of one of the objects to search for the other.

Here is a typical example: suppose you are given two one-million bit words.

Are they the same as cyclic words?

That is, do they correspond to the same necklace?

Computing a million rotated versions of a word to find the other is clearly a bad idea.

A Trick

But note the following.

Proposition 7. *Two words x and y are equivalent under rotation if they are conjugate: $x = uv$ and $y = vu$ for some words u and v.*

Proposition 8. *Two words x and y are equivalent under rotation if x is a factor of yy.*

$$yy = v\,\underbrace{u\,v}_{x}\,u$$

The last property can be checked in linear time using standard string-matching algorithms.

Exercise 15. *Prove the two propositions.*

Summary

- Burnside/Pólya/Redfield can be used to solve counting problems involving group actions.
- Beyond mere counting, enumeration (ranking and unranking) is an important technique.
- The corresponding algorithms can be quite intricate; correctness is sometimes difficult to establish.
- Direct generation of non-equivalent objects can be quite difficult; finding fast algorithms is an area of current research.
- Application of CS in areas such as biology and chemistry has created much interest in these topics recently.