Lexicographic generation

Contents

1 Introduction to generation and random generation

Two things we might want to do given a combinatorial class $\mathcal{A}$ would be to generate all elements of $\mathcal{A}_n$ or to generate a random element of $\mathcal{A}_n$.

What does it mean to generate something at random? For a combinatorial class $\mathcal{A}$, we want to describe a process (e.g. algorithm) that takes as input n, and outputs an object α from $\mathcal{A}$ such that any element has probability $\frac{1}{A_n}$ of being generated. We call this Uniform generation.

1.1 Features we might want in an exhaustive generation algorithm

- We might want to be able to generate the elements of $\mathcal{A}_n$ sequentially, so that we can step from one to the next. That is we might want a successor function which takes one element of $\mathcal{A}_n$ and gives the next one in the order. This is useful if we want to iterate through all elements of $\mathcal{A}_n$ and do something with each one.

- Given an object in $\mathcal{A}_n$ we might want to be able to determine the position of this object in the order, that is we might want a ranking function. We also might want to be able to determine the element in position i, that is we might want an unranking function

We would want to be able to do these things efficiently.

Definition. Let S be a finite set. A rank function is a bijection

$$\text{Rank} : S \to \{0, 1, \dots, |S| - 1\}$$

The corresponding unrank function is the inverse bijection

$$\text{Unrank} : \{0, 1, \dots, |S| - 1\} \to S$$

So $\text{Rank}(s) = i$ if and only if $\text{Unrank}(i) = s$.

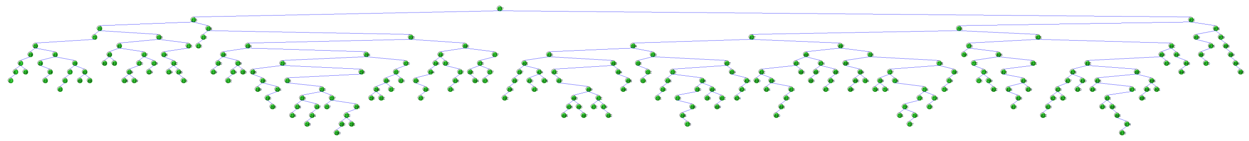
If we have a good unranking function we can generate a random element by first generating a random element of $\{0, \dots, |S| - 1\}$ and then unranking. If we have a good ranking and unranking function we can generate a successor function by $\text{Successor}(c) = \text{Unrank}(\text{Rank}(c) + 1)$. In both cases other approaches might be more efficient.

1.2 What about random generation?

Why would we want to do random generation?

- To get a sense of what the objects in a given class "look" like on average: estimate parameters like average depth of a tree; average number of cycles in a permutation etc.
- To test a hypothesis about the class using an experimental method;
- To feed as input into an algorithm to test the algorithm.

A random plane binary tree on 200 nodes:



Hallmarks of a good random generation scheme

Correct Each element of size n is generated with equal probability.

Efficient How long does it take to generate an element of size n as a function of n?

Space Efficiency How much space does it take to generate an element

Ease of implementation How long would it take to actually code the algorithm?

Re-usability Are there optimizations possible if I am going to run the algorithm multiple times?

Generalizability Is this part of a wider framework, or simply an adhoc solution applicable to a single problem?

Strategies

1. Consider simple models like binary strings, and make a direct argument;
2. Reduce the random generation to a very simple model via a bijection;
3. Surjective method: Find a simpler class that is in a k to one correspondence– uniform generation is still preserved;
4. Ranking/unranking: Order all A_n elements. Generate uniformly a random number k between 1 and A_n, and determine the k-th element.
5. Rejection method: Generate a larger set and then filter the output;
6. Recursive method: Decompose the object by the counting probabilities
7. Markov method: use Markov chains;
8. ...

2 Lexicographic ranking and unranking

(See Donald Kreher and Douglas Stinson, *Combinatorial Algorithms*, chapter 2 for a reference on this material. SFU access is at `https://www.taylorfrancis.com/books/mono/10.1201/9781003068006/combinatorial-algorithms-donald-kreher-douglas-stinson`.)

Definition. Let $A = (a_1, a_2, \dots, a_n)$ and $B = (b_1, b_2, \dots, b_m)$ be distinct lists of integers. $A < B$ in lexicographic order, if $a_1 = b_1, \dots, a_{k-1} = b_{k-1}$ and either $a_k < b_k$ or $n = k - 1$ and $m \geq k$.

For example $(1,2,2) < (2,1)$, $(1,2,2) < (1,3,1)$, $(1,2,2) < (1,2,2,1)$. This is the usual dictionary ordering on words using alphabetical order on the letters (hence the name).

If we can represent our combinatorial objects as lists, then we can use lexicographic order to order them, and from that get Successor, Rank, and Unrank functions.

2.1 Binary strings

Binary strings are simple enough to directly do random generation. Let us represent a binary string of length n by an array of length n with entries 0 or 1. Assume we have a random bit-generator $R()$ which generates a 0 or 1 in constant time.

Algorithm: RandBin (Maple code)

```
> R := rand(0..1);           # R() outputs a random integer in the range 0..1.
> RandBin := proc(n)         # RandBin(n) outputs a random n-bit binary sequence
>   for i from 1 to n do
>       W[i] := R()
>   end do;
>   return seq( W[i], i=1..n )
> end proc;
```

Each string has probability $(0.5)^n$ of being generated, so the distribution is uniform. There is one call to $R()$ made for each of the n bits, so the algorithm is **linear in** n.

Binary strings can be lexicographically ordered since they are lists of 0s and 1s. It is customary to index the digits of a binary word in reverse order, such as $W = (w_k, w_{k-1}, \dots, w_1, w_0)$. Then W is the usual binary representation of the number $\sum_{i=0}^{k} w_i 2^i$, and the successor function corresponds to adding 1 to binary number W. In python-like pseudocode we have

Algorithm: SuccessorBin

```
input: n (length of word), W (binary word)
W_new = W
for i from 1 to n
  if W(i) = 1
    W_new(i) = 0
    if i = n
      return "NO SUCCESSOR"
  else
    W_new(i) = 1
    return W_new
```

Conceptually ranking and unranking is just converting from a binary string to a binary number and back. In pseudocode we have

Algorithm: RankBin

```
input: n (length of word), W (word)
r = 0
for i from 1 to n
  r = r + 2^W(i)
output: r
```

```
                              Algorithm: UnrankBin
input: n (length of word), r (rank)
for i from 1 to n
  W(i) = ( r modulo 2 )
  r = floor(r/2)
output: W
```

2.2 Subsets of an n-set

Here we solve another problem by applying a bijection. Let $\mathcal{S}$ be a set of size n. Suppose we want to generate a subset of $\mathcal{S}$ such that every subset is equally probable. How do we do it? Let us use a bijection to binary strings:

1. Order all elements: $\mathcal{S} = \{s_1, s_2, \ldots, s_n\}$;
2. Generate a random binary string $\beta = (\beta_1, \beta_2, \ldots, \beta_n)$
3. Output the subset $\{s_i : \beta_i = 1\}$.

2.3 k-Subsets of an n-set

A *k-subset* is a subset of cardinality k. It is natural to ask about generating a random k-subset of the n-set $[n] = \{1, \ldots, n\}$. An inefficient way is to generate random subsets of $[n]$ until a subset of size k is produced. We leave as an exercise to find an algorithm which does this more efficiently.

We can also list all the the k-subsets in lexicographic order provided we represent each k-subset by listing its elements from smallest to largest.

For example, when $k = 3$ and $n = 5$, the lexicographic order is

$$(1,2,3),(1,2,4),(1,2,5),(1,3,4),(1,3,5),(1,4,5), \quad (2,3,4),(2,3,5),(2,4,5), \quad (3,4,5) \tag{1}$$

so we have $\mathrm{Rank}((1,4,5)) = 5$ and $\mathrm{Successor}((1,4,5)) = (2,3,4)$.

The successor function for this lexicographic ordering is not hard to find.

```
                              Algorithm: SuccessorkSubset
input: L (current k-subset as a list in increasing order), k, n
Lnew = L
i = k
while (i > 0) and (L(i) = n-k+i)
  i=i-1
if i=0
  return "no successor" (L is the last k-subset in lexicographic order)
for j from i to k
  Lnew(j) = L(i) + 1 + j - i
return Lnew
```

To rank and unrank we need to count how many k-subsets precede a given one.

Proposition. *Let $\{\ell_1, \ldots, \ell_k\} \subseteq [n]$ with $\ell_1 < \cdots < \ell_k$. Write $L = (\ell_1, \ldots, \ell_k)$ and define $\ell_0 = 0$. Then*

$$\mathrm{Rank}(L) = \sum_{i=1}^{k} \sum_{a=\ell_{i-1}+1}^{\ell_i - 1} \binom{n-a}{k-i}.$$

Proof. For every k-subset $\{a_1, a_2, \ldots, a_k\}$ that precedes L, there is exactly one pair (i, a) which satisfies

$$a_1 = \ell_1, \quad a_2 = \ell_2, \quad \ldots, \quad a_{i-1} = \ell_{i-1}, \quad \text{and } a = a_i < \ell_i.$$

We see that $\ell_{i-1} < a < \ell_i$. On the other hand, for any integer pair (i, a) which satisfies $1 \le i \le k$ and $\ell_{i-1} < a$, there are exactly $\binom{n-\ell}{k-i}$ k-subsets (presented as increasing lists) that begin with $\ell_1, \ell_2, \ldots \ell_{i-1}, a$ (Why?). If $a < \ell_i$, then all of these k-subsets precede L in lexicographic order. Summing over all such pairs (i, a) completes the proof. $\square$

This proof translates to the following ranking and unranking algorithms. We will discuss the correctness of these algorithms later in this lecture.

Algorithm: RankkSubset

```
input: L (current k subset as a list in increasing order), k, n
r = 0
L(0) = 0
for i from 1 to k
  for a from L(i-1)+1 to L(i)-1
    r = r + binom(n-a,k-i)
output: r
```

Algorithm: UnrankkSubset

```
input: r (rank), k, n
r_reduced = r
a = 0
for i from 1 to k
  a = a+1
  while binom(n-a,k-i) <= r_reduced
    r_reduced = r_reduced - binom(n-a,k-i)
    a = a+1
    if a>n then output: ERROR ( r is too large )
  L(i) = a
output: L
```

Example. We demonstrate these two algorithms for $n = 10$, $k = 4$, $L = (3, 5, 6, 9)$.

RankkSubset

Input: $n = 10,\ k = 4,\ L = (3, 5, 6, 9)$

$$\text{Rank}(L) = \sum_{i=1}^{k} \sum_{a=\ell_{i-1}+1}^{\ell_i - 1} \binom{n-a}{k-i}$$

i	ℓ_{i-1}	a	$\binom{n-a}{k-i}$	predecessors counted
1	0			
		1	$\binom{9}{3} = 84$	$(1, *, *, *), \{*, *, *\} \in \binom{\{2,3,4,5,6,7,8,9,10\}}{3}$
		2	$\binom{8}{3} = 56$	$(2, *, *, *), \{*, *, *\} \in \binom{\{3,4,5,6,7,8,9,10\}}{3}$
2	3			
		4	$\binom{6}{2} = 15$	$(3, 4, *, *),\ \{*, *\} \in \binom{\{5,6,7,8,9,10\}}{2}$
3	5			
4	6			
		7	$\binom{3}{0} = 1$	$(3, 5, 6, 7)$
		8	$\binom{2}{0} = 1$	$(3, 5, 6, 8)$
	9			
			$\sum_i \sum_a = 157$	

Output: $\text{Rank}((3, 5, 6, 9)) = 157$

`UnrankkSubset`

Input: $n = 10,\ k = 4,\ r = 157$

i	a	$\binom{n-a}{k-i}$	$r_{reduced}$	ℓ_i	predecessors accounted for
			157		
1	1	$\binom{9}{3} = 84$	73		$(1,*,*,*), \{*,*,*\} \in \binom{\{2,3,4,5,6,7,8,9,10\}}{3}$
	2	$\binom{8}{3} = 56$	17		$(2,*,*,*), \{*,*,*\} \in \binom{\{3,4,5,6,7,8,9,10\}}{3}$
	3	$\binom{7}{3} = 35 > 17$		3	
2	4	$\binom{6}{2} = 15$	2		$(3,4,*,*),\ \{*,*\} \in \binom{\{5,6,7,8,9,10\}}{2}$
	5	$\binom{5}{2} = 10 > 2$		5	
3	6	$\binom{4}{1} = 4 > 2$		6	
4	7	$\binom{3}{0} = 1$	1		$(3,5,6,7)$
	8	$\binom{2}{0} = 1$	0		$(3,5,6,8)$
	9	$\binom{1}{0} = 1 > 0$		9	

Output: $\text{Unrank}(157) = (3,5,6,9)$

2.4 General Lexicographic Ranking

The strategy demonstrated demonstrated above works in other settings.

Let $W = (w_1, w_2, \ldots, w_m)$ and $L = (\ell_1, \ell_2, \ldots, \ell_k)$ be two words over alphabet $[n] = \{1, 2, \ldots, n\}$. First, to handle the annoying fact that W and L can have different lengths, we can pretend that the shorter word is padded at the end with a string of 0's. This does not affect the lexicographic order since $0 < 1 < \cdots < n$. This way, we can assume that both W and L are 0-*padded* words of length k. Now if W lexicographically precedes L, then there exists a unique pair of numbers (i, a) where $1 \le i \le k$ and $0 \le a \le \ell_i - 1$ such that

$$w_1 = \ell_1,\ w_2 = \ell_2,\ \ldots,\ w_{i-1} = \ell_{i-1} \quad \text{and} \quad w_i = a. \tag{2}$$

Let $\mathcal{W}$ be a lexicographically sorted collection of 0-padded words of length k, with alphabet $\{0\} \cup [n]$. Let $L = (\ell_1, \ell_2, \ldots, \ell_k)$ be a word, not necessarily in $\mathcal{W}$. For each pair (i, a) with $1 \le i \le k$ and $0 \le a \le n$, we define the set of words

$$\mathcal{P}(L; i, a) = \{\, W \in \mathcal{W} :\ W \text{ satisfies (2)} \,\}$$

and let

$$P(L; i, a) = |\mathcal{P}(L; i, a)|.$$

We notice in particular that $\mathcal{P}(L; i, 0)$ is either empty, or it contains just one word, the truncation of L to length $i - 1$. Every word in $\mathcal{W}$ which preceeds L belongs to exactly one of the sets $\mathcal{P}(L; i, a)$, where $1 \le i \le k$ and $0 \le a \le \ell_i - 1$. Therefore if we can compute $P(L; i, a)$, then we can rank any word $L \in \mathcal{W}$ by performing a double sum.

$$\text{Rank}(L) = \sum_{i=1}^{k} \sum_{a=0}^{\ell_i - 1} P(L; i, a). \tag{3}$$

For example, the algorithm `RankkSubset` arises by observing that for any k-subset L of $[n]$ (presented as an increasing list) we have, assuming $\ell_0 = 0$, that

$$P(L;i,a) = \begin{cases} \binom{n-a}{k-i} & \text{if } 0 \le i \le k \text{ and } \ell_{i-1} < a \le n \\ 0 & \text{otherwise.} \end{cases} \tag{4}$$

In general, using (3) to compute $\mathrm{Rank}(L)$ requires $\mathcal{O}(kn)$ time, since each of the k inner summations has about n terms in the worst case. However `RankkSubset` requires only $\mathcal{O}(n)$ time, since we can omit the values of a in $\{0, 1, 2, \ldots, \ell_{i-1}\}$ in the inner sum or (3), so the inner sum is executed at most n times altogether.

2.5 General Lexicographic Unranking

Provided we can compute $P(L;i,a)$, there is also a natural unranking algorithm for $\mathcal{W}$ that is similar to `UnrankkSubset`. Given $r \ge 0$, we determine the word $L = (\ell_1, \ell_2, \ldots, \ell_k) = \mathrm{Unrank}(r)$ one letter at a time. Again we assume that $\mathcal{W}$ consists of 0-padded words of length k. Suppose we have already determined the values of $\ell_1, \ell_2, \ldots, \ell_{i-1}$. We can determine ℓ_i as follows. For each letter $a = 0, 1, 2, \ldots, n$ we compute the number of words in $\mathcal{W}$ that are predecessors of the word $(\ell_1, \ell_2, \ldots, \ell_{i-1}, a, 0, 0, \ldots, 0)$. Let us call this number $R(L;i,a^-)$.

$$R(L,i,a^-) = \left(\sum_{j=1}^{i-1} \sum_{b=0}^{\ell_j} P(L,j,b) \right) + \sum_{b=0}^{a-1} P(L,i,b).$$

Then ℓ_i is equal to the largest integer $a \in \{0, 1, \ldots, n\}$ for which

$$P(L;i,a) > 0, \quad \text{and} \quad R(L;i,a^-) \le r. \tag{5}$$

If no value of $a \in \{0, 1, 2, \ldots, n\}$ satisfies (5), then there is no word in $\mathcal{W}$ with rank as high as r. In this case we have $r > |\mathcal{W}| - 1$ and the algorithm should report the that r is out of bounds and exit.

Otherwise there is at least one word in $\mathcal{W}$ that begins with $(\ell_1, \ell_2, \ldots, \ell_{i-1}, \ell_i)$. If we find that $\ell_i = 0$, then some word in $\mathcal{W}$, call it L^-, begins with $\ell_1, \ell_2, \ldots, \ell_{i-1}, 0$. This implies that $L^- = (\ell_1, \ell_2, \ldots, \ell_{i-1}, 0, 0, \ldots, 0)$. Since $L^- \in \mathcal{W}$ we have

$$\mathrm{Rank}(L^-) = R(L;i,0^-) \le r.$$

In this case, successor of L^- in $\mathcal{W}$, call it L^+ begins with $\ell_1, \ell_2, \ldots, \ell_{i-1}, a$ for some $a > 0$, yet this value of a failed to satisfy (5). Therefore

$$\mathrm{Rank}(L^+) \ge R(L;i,a^-) \ge r + 1 \ge \mathrm{Rank}(L^-) + 1 = \mathrm{Rank}(L^+).$$

All these inequalities must be equalitites, so $\mathrm{Rank}(L^-) = r$. Therefore $\ell_i = 0$ implies that $L = L^-$ and the algorithm should output $(\ell_1, \ell_2, \ldots, \ell_{i-1})$.

To do this efficiently, for each i we step through the values $a = 0, 1, \ldots, n$, taking note of when $P(L;i,a) > 0$ and updating $R = R(L;i,a^-)$ as we go along. The following algorithm computes $\mathrm{Unrank}(r)$ provided we have subroutine which correctly computes $P(L;i,a)$ for $\mathcal{W}$.

Algorithm: UnrankGeneral

```
input: r (rank), n, P()
R = 0
for i from 1 to infinity
   L[i] = -1
   for a from 0 to n
      if P(L,i,a) > 0
         if  R + P(L,i,a) > r
            L[i] = a
            break (leave "for a" loop)
         else
            R = R + P(L,i,a)
```

```
    if L[i] = 0
       output: (L[1],L[2],...,L[i-1])
       exit
    if L[i] = -1
       output: ERROR ( r IS TOO LARGE )
       exit
```

The algorithm `UnrankkSubset` gives the same output as `UnrankGeneral` does when $P(L;i,a)$ is defined by (4). (This is not strictly true; we need to extend its domain by defining $P(L;i,0) = 1$ if $i > k$.) However `UnrankkSubset` is simplified and made more efficient due to the special form of $P(L;i,a)$. Because all k-subsets have length k, we do not need to pad words with zeros and we are done as soon as $i > k$. Again, for each $i \leq k$, the inner loop can be shortened because $P(L;i,a) = 0$ for $a = 0,1,2,3,\dots,\ell_{i-1}$. This makes `UnrankkSubset` an $\mathcal{O}(n)$ algorithm whereas `UnrankGeneral` is $\mathcal{O}(kn)$ where k is largest length of a word in $\mathcal{W}$. Another slight difference in `UnrankkSubset` is that, instead of using $R = R(L;i,a^-)$, we keep track of the quantity

$$r_{reduced} := r - R(L;i,a^-).$$

2.6 Faster ranking for k-subsets

Both of the algorithms `RankkSubset` and `UnrankkSubset` are $O(n)$ in the worst case. With a bit of cleverness, we can do better and rank in $O(k)$ time. The trick is to consider the reverse lexicographic order of $\binom{[n]}{k}$. Here, we represent each k-subset M as a list in *decreasing* order, before putting the lists into lexicographic order. The corank of M is defined to be the number of predecessors of M in the reverse lexicographic order.

Example. Here is a listing of $\binom{[5]}{3}$ in reverse lexicographic order. Notice that $\text{Coank}((\mathbf{5},\mathbf{3},\mathbf{2})) = 6$.

$$\underbrace{(3,2,1),\ (4,2,1),(4,3,1),(4,3,2),\ (5,2,1),(5,3,1),}_{\xrightarrow[\text{Coank}((\mathbf{5},\mathbf{3},\mathbf{2}))=6]{}}(\mathbf{5},\mathbf{3},\mathbf{2}),(5,4,1),(5,4,2),(5,4,3) \tag{6}$$

The corank of a subset is easier to compute than its rank.

Proposition. *Let* $\{m_1,\dots,m_k\} \subseteq [n]$ *with* $m_1 > m_2 > \cdots > m_k$*. Then*

$$\text{Coank}((m_1,\dots,m_k)) = \sum_{i=1}^{k} \binom{m_i - 1}{k+1-i}$$

Proof. For $i = 1,2,\dots,k$, there are $\binom{m_i - 1}{k-i+1}$ lists which begin with $m_1, m_2, \dots, m_{i-1}$ and have all of its remaining elements less than m_i. (i.e. the $(k+1-i)$ remaining elements are chosen from the set $\{1,2,\dots m_i - 1\}$.) All of these lists come before M in the reverse lexicographic order. Conversely, for every list $(a_1,a_2,\dots,a_k)$ that precedes M in reverse lexicographic order, there is exactly one integer i with $1 \leq i \leq k$ satisfying $a_1 = m_1, a_2 = m_2, \dots, a_{i-1} = m_{i-1}$ and $m_i > a_i > a_{i+1} > \cdots > a_k$. So summing $\binom{m_i-1}{k-i+1}$ over $i = 1,2,\dots,k$ counts every predecessor of $(m_1,\dots,m_k)$ exactly once. □

We need a formula to relate rank with corank. We can do this by "reflecting" the entries of a list in the following way.

Definition. Let $L = (\ell_1,\ell_2,\dots,\ell_k)$ be a list of integers with each ℓ_i in $[n]$. The reflection of L is the list

$$\tilde{L} = (n+1-\ell_1, n+1-\ell_2, \dots, n+1-\ell_k).$$

We observe two facts (Exercise: prove these!):

- If the entries of L are listed in increasing order, then the entries of $\tilde{L}$ appear in decreasing order.
- If A and L are two lists of the same length with increasing entries, then A comes before L in lexicographic order if and only if $\tilde{A}$ comes after $\tilde{L}$ in reverse lexicographic order.

Example. In the first line of the following table, the subsets in $\binom{[5]}{3}$ are ordered lexicographically (each subset is represented by a list L with increasing entries). In the second line, each list L from the first line has been reflected to obtain $\tilde{L}$.

$$\xrightarrow{\text{Rank}((\mathbf{1},\mathbf{3},\mathbf{4}))=3}$$

Lists L (lex-ordered) :	$(1,2,3),(1,2,4),(1,2,5),(\mathbf{1},\mathbf{3},\mathbf{4}),(1,3,5),(1,4,5),\ (2,3,4),(2,3,5),(2,4,5),\ (3,4,5)$
Reversed lists $\tilde{L}$:	$(5,4,3),(5,4,2),(5,4,1),(\mathbf{5},\mathbf{3},\mathbf{2}),(5,3,1),(5,2,1),\ (4,3,2),(4,3,1),(4,2,1),\ (3,2,1)$

$$\xleftarrow[\text{Coank}(\widetilde{(\mathbf{1},\mathbf{3},\mathbf{4})})=\text{Coank}((\mathbf{5},\mathbf{3},\mathbf{2}))=6]{}$$

We can verify the two facts here.

- Each reflected list $\tilde{L}$ has decreasing entries.
- The second line is in reverse lexicographic order *when read from right to left*. (Compare it with (6)).

Consider the list $L = (\mathbf{1},\mathbf{3},\mathbf{4})$. From the first line we see that $\text{Rank}(L) = 3$. From the second line we see that $\text{Coank}(\tilde{L}) = 6$. Since $\text{Rank}(L)$ counts the predecessors of L and $\text{Coank}(\tilde{L})$ counts the successors of L, we have $\text{Rank}(L) + \text{Coank}(\tilde{L}) = 3 + 6 = \binom{5}{3} - 1$.

Generalizing this example, we find a relation between Rank and Coank.

Proposition. *Let $\{\ell_1, \dots, \ell_k\} \subseteq [n]$ with $\ell_1 < \ell_2 < \cdots < \ell_k$. Let $L = (\ell_1, \ell_2, \dots, \ell_k)$ and let*

$$\tilde{L} = (n+1-\ell_1, n+1-\ell_2, \dots, n+1-\ell_k)$$

be the reflection of L. Then

$$\text{Rank}(L) + \text{Coank}(\tilde{L}) = \binom{n}{k} - 1.$$

Proof. Let $A = (a_1, a_2, \dots, a_k)$ be a k-subset (presented as an increasing list) such that $A < L$ in the lexicographic order. Then $(n+1-a_1, \dots, n+1-a_k) > (n+1-\ell_1, \dots, n+1-\ell_k)$ in the reverse lexicographic order (verify this statement for yourself). So for every k-subset A which is different from L either $A < L$ or $\tilde{A} < \tilde{L}$ (but not both). Therefore $\text{rank}(L) + \text{corank}(\tilde{L})$ equals the number of k-subsets of $[n]$ other than L itself. □

Putting these two propositions together, we get an alternative ranking formula for k-subsets.

Corollary. *Let $\{\ell_1, \dots, \ell_k\} \subseteq \{1, 2, \dots, n\}$ with $\ell_1 < \ell_2 < \cdots < \ell_k$. Write $L = (\ell_k, \dots, \ell_1)$. Then*

$$\text{Rank}(L) = \binom{n}{k} - 1 - \sum_{i=1}^{k} \binom{n-\ell_i}{k+1-i}.$$

Proof. From the definition of $\tilde{L}$, we have that $\tilde{L} = (m_1, m_2, \dots, m_k)$ where $m_i = n + 1 - \ell_i$. Since $m_1 > m_2 > \cdots > m_k$, we may apply the first proposition using $M = \tilde{L}$. Now the second proposition gives

$$\begin{aligned}\text{Rank}(L) &= \binom{n}{k} - 1 - \text{Coank}((m_1, m_2, \dots, m_k)) \\ &= \binom{n}{k} - 1 - \sum_{i=1}^{k} \binom{(n+1-\ell_i)-1}{k+1-i}\end{aligned}$$

as claimed. □

This gives us a second ranking algorithm for the k-subsets of $\{1, 2, \ldots, n\}$ in lexicographic order:

```
Algorithm: RankkSubset2
input: L (the k-subset as a list in increasing order), k, n
r = 0
for i from 1 to k
  r = r + binom( n-L(i), k+1-i )
output: binom( n, k ) - 1 - r
```

This algorithm is $O(k)$, which is faster than `RankkSubset` when n is significantly larger than k.

Example. Here is the same example as in Section 2.2, this time using `RankkSubset2`. There are significantly fewer calculations here than there were with `RankkSubset`, even though its proof of its correctness was more involved.

`RankkSubset2`

Input: $n = 10, k = 4, L = (3, 5, 6, 9)$

$$\text{Rank}(L) = \binom{n}{k} - 1 - \sum_{i=1}^{k} \binom{n - \ell_i}{k + 1 - i}$$

i	ℓ_i	$\binom{n-\ell_i}{k+1-i}$
1	3	$\binom{7}{4} = 35$
2	5	$\binom{5}{3} = 10$
3	6	$\binom{4}{2} = 6$
4	9	$\binom{1}{1} = 1$
		52

Output: $\text{Rank}(L) = 210 - 1 - 52 = 157$

Exercise. Can you make a similar `UnrankkSubset2`? What is its runtime?