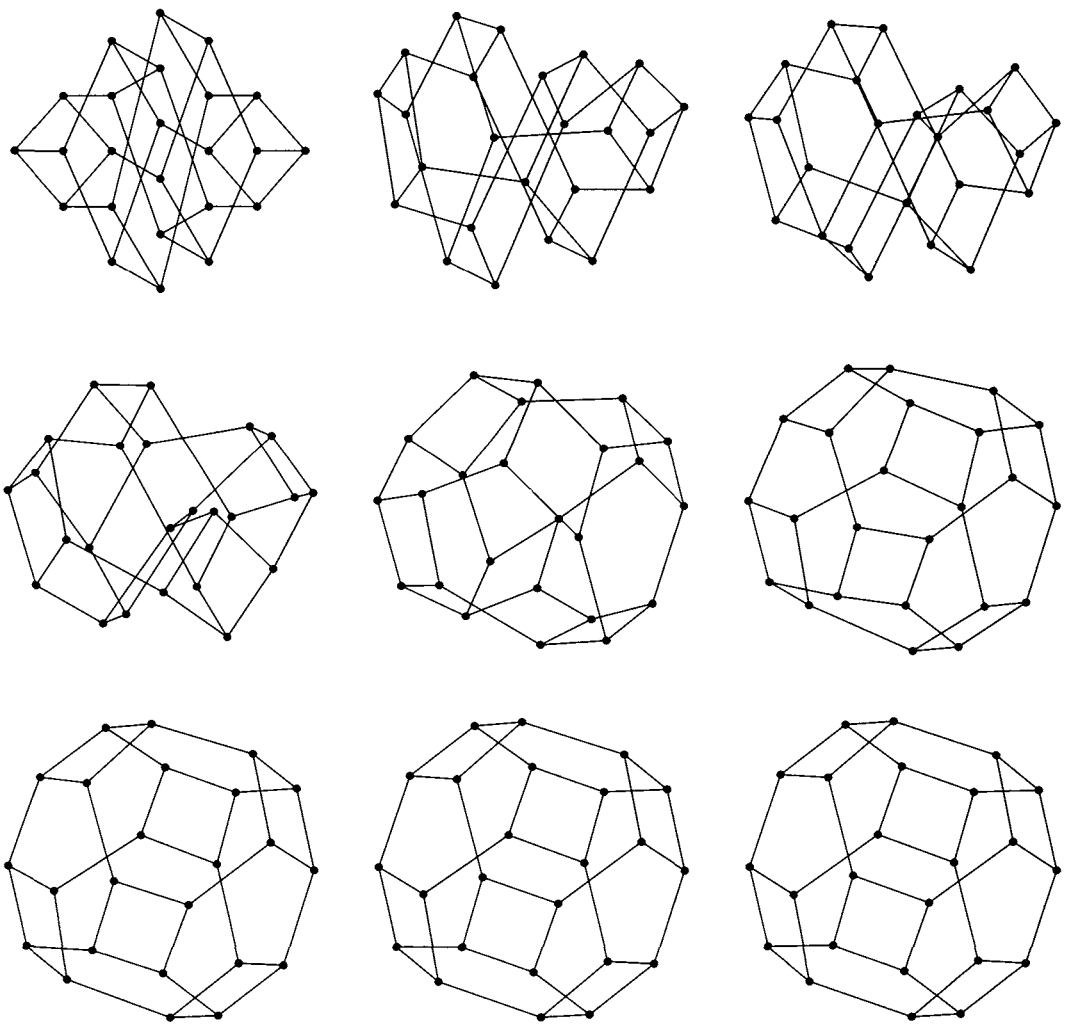
2. Permutations and Combinations



https://doi.org/10.1017/CBO9781139164849.003 Published online by Cambridge University Press

Combinatorics is the study of counting, and over the years mathematicians have counted a variety of different things. More recently, mathematicians have become interested in enumeration and random selection from sets of objects. Tools from computer science are crucial in ensuring that these tasks are done efficiently.

Perhaps the most fundamental combinatorial object is the *permutation*, which we define as an ordering of the integers 1 to n without repetition. Closely related to permutations are *combinations*, that is, distinct subsets of a set whose order does not matter. In this chapter, we provide algorithms for enumerating and selecting permutations and combinations in a variety of ways, each of which has distinct useful properties.

Even though permutations and combinations are simple structures, studying them closely leads to many important concepts in mathematics and computer science. We will encounter recurrence relations, generating functions, dynamic programming, randomized algorithms, and Hamiltonian cycles in this chapter. The programming examples included here provide a proper introduction to the *Mathematica* programming style. We will see that writing efficient functions in *Mathematica* is a different game than in conventional programming languages.

About the illustration overleaf:

An exchange of any two adjacent elements of a permutation yields a different permutation, and by applying adjacent transpositions in a certain order all $n!$ distinct permutations can be constructed. The illustration shows the *adjacent transposition graph* of the set of permutations on four elements, where each vertex represents a permutation and there is an edge whenever two permutations differ by exactly one adjacent transposition. All nine pictures show the same graph; on the top left is a *ranked embedding*, which is then transformed, using a *spring embedding* algorithm, into the *truncated octahedron* shown on the bottom left. By using tools that we will develop, we can construct and display this adjacent transposition graph as follows:

```
atr = MemberQ[Table[p = #1; {p[[i]], p[[i + 1]]} = {p[[i + 1]], p[[i]]}; p, {i, Length[#1]-1}], #2]&;
g = RankedEmbedding[MakeGraph[Permutations[4], atr, Type->Undirected], {1}];
ShowGraphArray[Partition[Table[SpringEmbedding[g, i], {i, 0, 40, 5}], 3]]
```

https://doi.org/10.1017/CBO9781139164849.003 Published online by Cambridge University Press

2.1 Generating Permutations

A permutation can be viewed as a linear arrangement of objects or as a rearrangement operation. When thought of as an operation, a permutation specifies distinct new positions for each of n objects arranged linearly. We define a *size-n permutation* (an *n-permutation*, in short) to be an ordering of the integers 1 to n without repetition. We will use π to denote permutations and use $\pi(i)$ to denote the ith element in π. Our first *Mathematica* function determines if a given object is a permutation.

```
PermutationQ[e_List] := (Sort[e] === Range[Length[e]])
```

Testing a Permutation

Mathematica array notation provides a slick way to permute a set of objects according to a particular permutation, since elements of one list can be used as indices to select elements of another. The same technique can be used to swap two values without an explicit intermediate variable or to select an arbitrary subset of a set.

```
Permute[l_List,p_?PermutationQ] := l [[ p ]]
Permute[l_List,p_List] := Map[ (Permute[l,#])&, p] /; (Apply[And, Map[PermutationQ, p]])
```

Permuting a List

This loads the *Combinatorica* package.

```
In[1]:= <<DiscreteMath`Combinatorica`
```

Our restriction of permutations to the integers from 1 to n is not limiting, since we can easily permute a list of whatever we are interested in.

```
In[2]:= Permute[{a,b,c,d,e},{5,3,2,1,4}]
Out[2]= {e, c, b, a, d}
```

Multiplying permutations should be understood as composing the rearrangement operations. More precisely, if π_1 and π_2 are n-permutations, then $\pi_1 \times \pi_2$ is the rearrangement operation that corresponds to applying π_2 followed by π_1. Thus Permute provides an implementation of permutation multiplication.

`(a,b,c,d,e)` is rearranged by `(1,5,4,2,3)` followed by `(4,5,1,3,2)`.

```
In[3]:= Permute[Permute[{a,b,c,d,e}, {1,5,4,2,3}], {4,5,1,3,2}]
Out[3]= {b, c, a, d, e}
```

This is equivalent to rearranging `(a,b,c,d,e)` by the product `(1,5,4,2,3)` × `(4,5,1,3,2)`.

```
In[4]:= p=Permute[{1,5,4,2,3}, {4,5,1,3,2}]; Permute[{a,b,c,d,e}, p]
Out[4]= {b, c, a, d, e}
```

The multiplication of permutations is not commutative.

```
In[5]:= p == Permute[{4,5,1,3,2},{1,5,4,2,3}]
Out[5]= False
```

https://doi.org/10.1017/CBO9781139164849.003 Published online by Cambridge University Press

But it is associative. Here we see that $(a \times b) \times c = a \times (b \times c)$.

```
In[6]:= Permute[Permute[a = RandomPermutation[20],
                        b = RandomPermutation[20]
                ], c = RandomPermutation[20]
        ] === Permute[a, Permute[b, c]]
Out[6]= True
```

The *identity* is the permutation $(1, 2, \ldots, n)$ that maps every element to itself. For every permutation π there is a unique permutation, denoted π^{-1}, such that $\pi \times \pi^{-1} = \pi^{-1} \times \pi$ equals the identity. π^{-1} is the *multiplicative inverse* or, more simply, the *inverse* of π. Clearly, $\pi^{-1}[j] = k$ if and only if $\pi[k] = j$.

```
InversePermutation[p_?PermutationQ] :=
      Module[{inverse=p},
            Do[ inverse[[ p[[i]] ]] = i, {i,Length[p]} ];
            inverse
      ]
IdentityPermutation[n_Integer] := Range[n]
```

The Inverse of a Permutation and the Identity Permutation

The product of a permutation and its inverse is the identity.

```
In[7]:= Permute[p = RandomPermutation[20], InversePermutation[p]] ==
        IdentityPermutation[20]
Out[7]= True
```

The rest of this section will be devoted to the problems of generating, ranking, and unranking permutations. Their solutions include interesting algorithms that provide an introduction to building combinatorial objects with *Mathematica*.

■ 2.1.1 Lexicographically Ordered Permutations

Constructing all permutations is such an important operation that *Mathematica* includes the function `Permutations` to do it. Algorithms for constructing permutations are surveyed in [Sed77].

The most straightforward way to construct all permutations of the n items of a list l is to successively pull each element out of l and prepend it to each permutation of the $n - 1$ other items. This procedure tells us right away that $n!$ is the number of permutations of n items.

A faster way to do this in *Mathematica* is by iteratively generating the next permutation in lexicographic order. The lexicographic successor of a size-n permutation π is obtained as follows. First find the largest decreasing suffix of π, say $\pi[i+1, \ldots, n]$. Since this suffix is largest, $\pi[i] < \pi[i+1]$. Therefore, $\pi[i+1, \ldots, n]$ contains an element $\pi[j]$ that is the smallest element greater than $\pi[i]$. Exchange $\pi[i]$ and $\pi[j]$ and then reverse $\pi[i+1, \ldots, n]$. For example, let $\pi = (4, 3, 1, 5, 2)$. The largest decreasing suffix is $(5, 2)$ and $\pi[i] = 1$. The smallest element in this suffix larger than 1 is 2. The subsequent exchanging and reversing gives $(4, 3, 2, 1, 5)$. `NextPermutation`, shown below, implements this algorithm.

https://doi.org/10.1017/CBO9781139164849.003 Published online by Cambridge University Press

```
NextPermutation[l_List] := Sort[l] /; (l === Reverse[Sort[l]])

NextPermutation[l_List] :=
      Module[{n = Length[l], i, j, t, nl = l},
            i = n-1; While[ Order[nl[[i]], nl[[i+1]]] == -1, i--];
            j = n; While[ Order[nl[[j]], nl[[i]]] == 1, j--];
            {nl[[i]], nl[[j]]} = {nl[[j]], nl[[i]]};
            Join[ Take[nl,i], Reverse[Drop[nl,i]] ]
      ]
```

Constructing the Next Permutation

A size-12 random permutation and its lexicographic successor.

```
In[8]:= {p = RandomPermutation[12], NextPermutation[p]} //ColumnForm
Out[8]= {7, 12, 4, 1, 3, 10, 5, 6, 8, 11, 2, 9}
        {7, 12, 4, 1, 3, 10, 5, 6, 8, 11, 9, 2}
```

The lexicographic successor of the last permutation is the first.

```
In[9]:= NextPermutation[ Reverse[Range[8]] ]
Out[9]= {1, 2, 3, 4, 5, 6, 7, 8}
```

Enumerating lexicographically ordered permutations is now simply a matter of repeatedly calling `NextPermutation`. The cost of calling a function $n! - 1$ times adds up quickly, so for the sake of speed we use the code from `NextPermutation` explicitly in `LexicographicPermutations`. We also use the *Mathematica* function `NestList` that, given as inputs a function f, an expression x, and an integer n, returns the sequence $(f^0(x), f^1(x), f^2(x), \ldots, f^n(x))$. Here $f^0(x) = x$ and in general, $f^i(x)$ is obtained by starting with x and applying f repeatedly i times. Using such a function not only leads to compact code, but it is also faster than explicitly constructing the list.

However, the most substantial speedup over the earlier version of this function is due to the use of the *Mathematica* function `Compile`. *Mathematica*'s programming language does not force us to specify the types of inputs upfront. However, this imposes a substantial time and space overhead that can be avoided when we know a priori that all input has a specific, simple type. *Mathematica* can compile functions whose arguments are integers, reals, Boolean, or regularly structured lists of these types. This internal code is stored as a `CompiledFunction` object that is executed when the function is called. The compiled `LexicographicPermutations` is roughly ten times faster than the uncompiled version.

```
LP = Compile[{{n, _Integer}},
          Module[{l = Range[n], i, j, t},
                NestList[(i = n-1; While[ #[[i]] > #[[i+1]], i--];
                          j = n; While[ #[[j]] < #[[i]], j--];
                          t = #[[i]]; #[[i]] = #[[j]]; #[[j]] = t;
                          Join[ Take[#,i], Reverse[Drop[#,i]] ])&,
                          l, n!-1
                ]
          ]
    ]
```

https://doi.org/10.1017/CBO9781139164849.003 Published online by Cambridge University Press

```
LexicographicPermutations[0] := {{}}
LexicographicPermutations[1] := {{1}}

LexicographicPermutations[n_Integer?Positive] := LP[n]
LexicographicPermutations[l_List] := Permute[l, LexicographicPermutations[Length[l]] ]
```

Constructing Lexicographically Ordered Permutations

As you can see, the permutations are generated in lexicographic order. The built-in generator `Permutations` also constructs them in lexicographic order.

```
In[10]:= LexicographicPermutations[3]
Out[10]= {{1, 2, 3}, {1, 3, 2}, {2, 1, 3}, {2, 3, 1},
    {3, 1, 2}, {3, 2, 1}}
```

`UnCompiledLexPerms` is identical to `LexicographicPermutations`, except that it is not compiled. The execution time ratios reveal how many times faster the compiled version is than the uncompiled version.

```
In[11]:= <<extraCode/UnCompiledLexPerms;
         Table[Timing[UnCompiledLexPerms[n];][[1, 1]]/
              Timing[LexicographicPermutations[n];][[1, 1]],
              {n, 6, 9}
         ]
Out[12]= {8.5, 11.1818, 11., 8.08787}
```

Using compiled code does not prevent `LexicographicPermutations` from being able to generate permutations of arbitrary objects. It does this in a slightly indirect manner by first generating all size-n permutations for the right n and then applying these one at a time onto the given sequence of objects. Despite this, it is faster to use the compiled version of the function even for sequences of objects that are technically not permutations.

```
In[13]:= Clear[P, Q, R, S];
         LexicographicPermutations[{P, Q, R, S}]
Out[14]= {{P, Q, R, S}, {P, Q, S, R}, {P, R, Q, S},
    {P, R, S, Q}, {P, S, Q, R}, {P, S, R, Q}, {Q, P, R, S},
    {Q, P, S, R}, {Q, R, P, S}, {Q, R, S, P}, {Q, S, P, R},
    {Q, S, R, P}, {R, P, Q, S}, {R, P, S, Q}, {R, Q, P, S},
    {R, Q, S, P}, {R, S, P, Q}, {R, S, Q, P}, {S, P, Q, R},
    {S, P, R, Q}, {S, Q, P, R}, {S, Q, R, P}, {S, R, P, Q},
    {S, R, Q, P}}
```

Here we compare our implementation of permutation enumeration to *Mathematica*'s. Despite being a compiled object, `LexicographicPermutations` is slower than the built-in `Permutations`.

```
In[15]:= Table[Timing[ LexicographicPermutations[i]; ][[1,1]]
         /Timing[ Permutations[Range[i]]; ][[1,1]], {i,7,9}]
Out[15]= {3., 3.5, 2.60479}
```

This results of this timing experiment should not be too surprising. Even if both functions used the same underlying algorithm, one would expect `Permutations` to be faster because it is part of the *Mathematica* kernel and exists in machine code. In contrast, `LexicographicPermutations` exists in internal *Mathematica* code that is supposedly "close" to machine code of a typical computer. In addition to being faster, `Permutations` can also handle multisets correctly, which is something that `LexicographicPermutations` cannot do.

`Permutations` deals with multisets correctly...

```
In[16]:= Permutations[{1, 1, 2}]
Out[16]= {{1, 1, 2}, {1, 2, 1}, {2, 1, 1}}
```

https://doi.org/10.1017/CBO9781139164849.003 Published online by Cambridge University Press

...while LexicographicPermutations does not.

```
In[17]:= LexicographicPermutations[{1, 1, 2}]
Out[17]= {{1, 1, 2}, {1, 2, 1}, {1, 1, 2}, {1, 2, 1},
    {2, 1, 1}, {2, 1, 1}}
```

A natural measure of the amount of work that LexicographicPermutations does is the total number of element exchanges or *transpositions* performed. Let π be a permutation and let $\pi[i+1,\ldots,n]$ denote its longest decreasing suffix. One transposition is needed to introduce $\pi[i]$ into this suffix and $\lfloor (n-i)/2 \rfloor$ transpositions are needed to reverse the suffix. Thus NextPermutation performs $1 + \lfloor (n-i)/2 \rfloor$ transpositions when acting on a permutation π whose longest decreasing suffix starts at position $(i+1)$. How many transpositions does LexicographicPermutations perform in generating all n-permutations?

The average number of transpositions per permutation appears to approach a limit as n increases. But we can only exhaustively test small values of n. Mean is not built-in, but it is defined in the *Mathematica* add-on package Statistics`DescriptiveStatistics`, which is loaded when *Combinatorica* is.

```
In[18]:= Table[ Mean[
           Map[(i=n-1; While[#[[i]] > #[[i+1]],i--]; 1+Floor[(n-i)/2])&,
           Permutations[n]]]//N, {n,1,6}]
Out[18]= {1., 1.5, 1.5, 1.54167, 1.54167, 1.54306}
```

For size-100 permutations, we must rely on random sampling, evaluating 1000 random permutations. The number of transpositions required by NextPermutation is calculated for each, and the average of these is reported. This result approaches the same limit.

```
In[19]:= Mean[
           Map[(i=99; While[#[[i]] > #[[i+1]], i--]; 1+Floor[50-i/2])&,
           Table[RandomPermutation[100], {1000}]] ]//N
Out[19]= 1.55
```

The magic limit turns out to be $Cosh(1)$, where $Cosh(z) \equiv (e^z + e^{-z})/2$ is the hyperbolic cosine function.

```
In[20]:= N[ Cosh[1] ]
Out[20]= 1.54308
```

This result motivates the problem of generating permutations in an order that minimizes the total number of transpositions used. This problem will be solved in Section 2.1.4.

■ 2.1.2 Ranking and Unranking Permutations

Since there are $n!$ distinct permutations of size n, any ordering of permutations defines a bijection with the integers from 0 to $n! - 1$. Such a bijection can be used to give several useful operations. For example, if we seek a permutation with a certain property, it can be more efficient to repeatedly construct and test the ith permutation than to build all $n!$ permutations in advance and search.

A *ranking* function for an object determines its position in the underlying total order. For permutations in lexicographic order, the rank of a permutation may be computed by observing that all permutations sharing the same first element k are ranked $(k-1)(n-1)!$ to $k(n-1)! - 1$. After deleting

https://doi.org/10.1017/CBO9781139164849.003 Published online by Cambridge University Press

the first element and adjusting the rest of the elements so that we have an $(n-1)$-permutation, we can recurse on the smaller permutation to determine the exact rank. For example, the permutation $(2,3,1,5,4)$ has rank between 24 and 47 by virtue of its first element. Once 2 is deleted, the rest of the permutation is adjusted from $(3,1,5,4)$ to $(2,1,4,3)$, and the rank of $(2,1,4,3)$ is obtained. Once the contribution of the first element is noted, the element is stripped and the rest of the permutation is adjusted.

```
RankPermutation[{1}] := 0
RankPermutation[{}] := 0

RankPermutation[p_?PermutationQ] :=
      Block[{$RecursionLimit = Infinity},
           (p[[1]]-1) (Length[Rest[p]]!) +
           RankPermutation[ Map[(If[#>p[[1]], #-1, #])&, Rest[p]]]
      ]
```

Ranking Permutations

An *unranking* algorithm constructs the ith object in the underlying total order for a given rank i. The previous analysis shows that the quotient obtained by dividing i by $(n-1)!$ gives the index of the first element of the permutation. The remainder is used to determine the rest of the permutation.

Applying the unranking algorithm $n!$ times gives an alternate method for constructing all permutations in lexicographic order.

```
In[21]:= Table[UnrankPermutation[n, Range[3]], {n, 0, 5}]
Out[21]= {{1, 2, 3}, {1, 3, 2}, {2, 1, 3}, {2, 3, 1},
    {3, 1, 2}, {3, 2, 1}}
```

The ranking function proves that `Permutations` uses lexicographic sequencing.

```
In[22]:= Map[RankPermutation, Permutations[{1,2,3}]]
Out[22]= {0, 1, 2, 3, 4, 5}
```

■ 2.1.3 Random Permutations

With $n!$ distinct permutations of n items, it is impractical to test a conjecture on all permutations for any substantial value of n. However, experiments on several large, *random* permutations can give confidence that any results we get are not an artifact of how the examples were selected. We have already seen how permutations of size 100 can be randomly sampled to determine the average number of transpositions to take a permutation to its lexicographic successor. We will see many more examples of random sampling of combinatorial objects in this book. Functions for generating random instances of most interesting combinatorial structures are included. Many start by constructing a random permutation.

The fastest algorithm for random permutations starts with an arbitrary permutation and exchanges the ith element with a randomly selected one from the first i elements, for each i from n to 1. The nth element is therefore equally likely to be anything from 1 to n, and it follows by induction that any

https://doi.org/10.1017/CBO9781139164849.003 Published online by Cambridge University Press

permutation is equally likely and is produced with probability $1/n!$. `RandomPermutation` is compiled to make it faster, which is possible because it only manipulates a list of integers.

```
RP = Compile[{{n, _Integer}},
          Module[{p = Range[n],i,x,t},
                Do [x = Random[Integer,{1,i}];
                   t = p[[i]]; p[[i]] = p[[x]]; p[[x]] = t,
                   {i,n,2,-1}
                ];
                p
          ]
      ]

RandomPermutation[n_Integer] := RP[n]
RandomPermutation[l_List] := Permute[l, RP[Length[l]]]
```

Constructing a Random Permutation

If the random permutation is indeed random, then all six permutations of three elements must be generated roughly the same number of times, provided we generate enough of them.

```
In[23]:= Distribution[Table[RandomPermutation[3],{300}]]
Out[23]= {52, 45, 62, 45, 56, 40}
```

`UnCompiledRandPerm` is the uncompiled version of `RandomPermutation`. The execution time ratios clearly show that compilation gives a significant speedup.

```
In[24]:= <<extraCode/UnCompiledRandPerm;
         Table[Timing[UnCompiledRandPerm[n];][[1, 1]]/
         Timing[RandomPermutation[n];][[1, 1]], {n, 1000, 2000, 100}]
Out[25]= {11., 11., 13., 14., 15., 16., 8.5, 9., 9.5, 10.,
    10.5}
```

If the random permutation is indeed random, then the average first element in a size-100 permutation should hover around 50.

```
In[26]:= Mean[Table[First[RandomPermutation[100]], {1000}]] // N
Out[26]= 49.272
```

Care must be taken that random objects are indeed selected with equal probability from the set of possibilities. Nijenhuis and Wilf [NW78, Wil89] considered this problem in depth, and their books are the source of many of the algorithms that we implement. *Markov chain Monte Carlo* methods have been recently applied to generate combinatorial objects such as regular graphs and spanning trees uniformly at random [Bro89, Wil96, JS90].

The plot below, which displays the running times of `RandomPermutation` for permutation sizes from 100 to 10,000, indicates a linear running time. The same function, running on pre-version 4.0 *Mathematica*, exhibits a quadratic running time. This improvement is due to an important new *Mathematica* feature called *packed arrays*. When appropriate, large linear lists or large, regularly structured nested lists of numbers are *automatically* stored as packed arrays of machine-sized integers or real

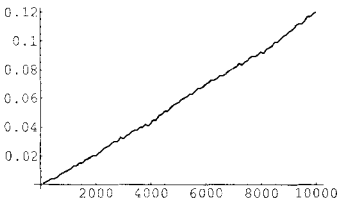
https://doi.org/10.1017/CBO9781139164849.003 Published online by Cambridge University Press

numbers. The implication of this is that writing into such an array becomes an $O(1)$ time operation, in contrast with the linear time that it takes to write into a typical *Mathematica* data structure.

For `RandomPermutation`, this means that exchanging elements in the list `p` takes $O(1)$ time. Thus the overall running time of the function is linear. These two new features – packed arrays and compilation – have been used throughout *Combinatorica*. Along with improved algorithms, they have led to a substantial overall speedup in the package.

This timing plot indicates a linear running time for `RandomPermutation`. The function is extremely fast and here it takes only a small fraction of a second to generate a 10,000-element random permutation!

```
In[27]:= ListPlot[Table[{i, Mean[Table[Timing[RandomPermutation[i];]
         [[1, 1]], {10}]]},{i, 100, 10000, 100}], PlotJoined -> True]
```



■ 2.1.4 Minimum Change Permutations

`LexicographicPermutations` gives just one way of generating permutations. Sedgewick's survey paper [Sed77] describes over 30 permutation generation algorithms published as of 1977! These other permutation generation algorithms typically produce permutations in nonlexicographic orders with special properties. In this section, we study an algorithm that uses a total of $(n! - 1)$ transpositions in generating all permutations. In other words, the permutations are arranged such that every two adjacent permutations differ from each other by exactly one transposition.

Sedgewick's study [Sed77] revealed an algorithm by Heap [Hea63] to be generally the fastest permutation generation algorithm. We implement this algorithm as `MinimumChangePermutation`. Heap's idea is best described using the following recursive framework that can be used to describe many permutation generation algorithms:

Set $\pi[i] = i$ for all $i = 1, 2, \ldots, n$
for $c = 1$ to n do
1. generate all permutations of $\pi[1, \ldots, n-1]$, keeping $\pi[n]$ in place;
(at the end of Step 1, π contains the last generation permutation)
2. exchange $\pi[n]$ with an element in $\pi[1, \ldots, n-1]$ whose position is $f(n, c)$.

https://doi.org/10.1017/CBO9781139164849.003 Published online by Cambridge University Press

$f(n,c) \in \{1,\ldots,n-1\}$ is simply a function of n and c, the number of times permutations of $\pi[1,\ldots,n-1]$ have been generated. Different permutation generation algorithms differ in values for $f(n,c)$. Heap's algorithm uses the somewhat mysterious function:

$$f(n,c) = \begin{cases} 1 & \text{if } n \text{ is odd} \\ c & \text{otherwise.} \end{cases}$$

For example, let $n = 4$ and start with $\pi = (1,2,3,4)$. During the first execution of the for-loop, $c = 1$ and the first step is to generate all permutations of $\pi[1,2,3]$. Heap's algorithm generates permutations in the order $123, 213, 312, 132, 231, 321$.

Generating all size-3 permutations by Heap's algorithm.

```
In[28]:= MinimumChangePermutations[3]
Out[28]= {{1, 2, 3}, {2, 1, 3}, {3, 1, 2}, {1, 3, 2},
    {2, 3, 1}, {3, 2, 1}}
```

The first six size-4 permutations as generated by Step 1 of Heap's algorithm.

```
In[29]:= MinimumChangePermutations[4][[ Range[6] ]]
Out[29]= {{1, 2, 3, 4}, {2, 1, 3, 4}, {3, 1, 2, 4},
    {1, 3, 2, 4}, {2, 3, 1, 4}, {3, 2, 1, 4}}
```

After Step 1 is complete, π has value $(3,2,1,4)$ and then $\pi[4]$ and $\pi[1]$ are exchanged because $f(4,1) = 1$. Thus the second execution of the loop starts with $\pi = (4,2,1,3)$. In Step 1 of the second execution of the for-loop, 3 remains fixed as the last element of π while the first three get permuted as before.

This example shows the remaining sequence of size-4 permutations generated by Heap's algorithm. All consecutive permutations differ by exactly one transposition.

```
In[30]:= MinimumChangePermutations[4][[ Range[7,24] ]]
Out[30]= {{4, 2, 1, 3}, {2, 4, 1, 3}, {1, 4, 2, 3},
    {4, 1, 2, 3}, {2, 1, 4, 3}, {1, 2, 4, 3}, {1, 3, 4, 2},
    {3, 1, 4, 2}, {4, 1, 3, 2}, {1, 4, 3, 2}, {3, 4, 1, 2},
    {4, 3, 1, 2}, {4, 3, 2, 1}, {3, 4, 2, 1}, {2, 4, 3, 1},
    {4, 2, 3, 1}, {3, 2, 4, 1}, {2, 3, 4, 1}}
```

To see that exactly one transposition suffices to take a permutation to its successor in Heap's algorithm, partition the sequence of permutations into blocks, with each block containing permutations with an identical last element. Each such block is produced by a recursive call to generate permutations of size $(n-1)$. In going from the last permutation of a block to the first permutation of the successor, exactly one transposition is used. Within each block, the last element remains fixed and only the first $(n-1)$ elements of the list are moved. By induction we know that each permutation (except the last) in each block can be transformed into its successor by a single transposition.

The 5-permutations as enumerated by Heap's algorithm and partitioned into blocks containing the same last element. The first and last elements of each block are reported. A single transposition suffices to move from the last permutation in a block to the first permutation of the next.

```
In[31]:= Map[{First[#], Last[#]} &,
          Split[MinimumChangePermutations[Range[5]],
          (Last[#1] === Last[#2]) &]]
Out[31]= {{{1, 2, 3, 4, 5}, {2, 3, 4, 1, 5}},
    {{5, 3, 4, 1, 2}, {3, 4, 1, 5, 2}},
    {{2, 4, 1, 5, 3}, {4, 1, 5, 2, 3}},
    {{3, 1, 5, 2, 4}, {1, 5, 2, 3, 4}},
    {{4, 5, 2, 3, 1}, {5, 2, 3, 4, 1}}}
```

Our implementation of Heap's algorithm is iterative. Instead of a single loop control variable *c*, we have an array of loop control variables that keep track of the *n* for-loops. The array c essentially keeps track of each of the *n* for-loop control variables.

```
MinimumChangePermutations[l_List] := LexicographicPermutations[l] /; (Length[l] < 2)
MinimumChangePermutations[l_List] :=
      Module[{i=1,c,p=l,n=Length[l],k},
            c = Table[1,{n}];
            Join[{l},
                Table[While [ c[[i]] >= i, c[[i]] = 1; i++];
                      If[OddQ[i], k=1, k=c[[i]] ];
                      {p[[i]],p[[k]]} = {p[[k]],p[[i]]};
                      c[[i]]++; i = 2; p,
                      {n!-1}
                ]
            ]
      ]
MinimumChangePermutations[n_Integer] := MinimumChangePermutations[Range[n]]
```

Sequencing Permutations in Minimum Change Order

To better illustrate the transposition structure of permutations, we introduce some of the graph-theoretic tools we will develop in the latter part of this book. A *graph* is a collection of pairs of elements, called *edges*, drawn from a finite, nonempty set of elements, called *vertices*. When the edges are ordered pairs, the resulting graphs are called *directed graphs*. If we represent the vertices by points and the edges by curves connecting pairs of points, we get a visual representation of the underlying structure. In a *transposition graph*, the vertices correspond to permutations, with edges connecting pairs of permutations that differ by exactly one transposition.

A *binary relation* on a set *S* is a subset of ordered pairs of elements in *S*. Here we define a binary relation on *n*-permutations.

```
In[32]:= tr=MemberQ[Flatten[Table[p = #1;
              {p[[i]], p[[j]]} = {p[[j]], p[[i]]};
              p, {i, Length[#1] - 1}, {j, i + 1, Length[#1]}], 1], #2]&;
```

Two *n*-permutations are related if and only if one can be obtained from the other by a transposition.

```
In[33]:= {tr[{4,3,1,2},{4,3,2,1}], tr[{4,3,1,2},{1,2,3,4}]}
Out[33]= {True, False}
```

https://doi.org/10.1017/CBO9781139164849.003 Published online by Cambridge University Press

Here, using the relation defined above, we construct the transposition graph for 4-permutations.

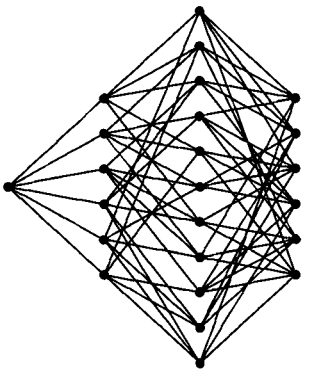
```
In[34]:= g = MakeGraph[Permutations[4], tr, Type -> Undirected];
```

The *degree* of a vertex is the number of edges connecting it to other vertices. Each 4-permutation is one transposition away from six other 4-permutations and so each vertex has degree 6. In general, vertices in the transposition graph of n-permutations have degree $\binom{n}{2}$.

```
In[35]:= DegreeSequence[g]
Out[35]= {6, 6, 6, 6, 6, 6, 6, 6, 6, 6, 6, 6, 6, 6, 6,
    6, 6, 6, 6, 6, 6, 6, 6}
```

This graph is drawn using `RankedEmbedding`, so that the vertices are arranged in columns according to the fewest number of transpositions needed to get to the leftmost vertex.

```
In[36]:= ShowGraph[tg4 = RankedEmbedding[g, {1}]];
```



There are 6 permutations that are one transposition away from the permutation on the left, 11 permutations two away, and 6 permutations three away.

```
In[37]:= Distribution[ RankGraph[g, {1}] ]
Out[37]= {1, 6, 11, 6}
```

This distribution of permutations by transposition "distance" is counted by the *Stirling numbers of the first kind*.

```
In[38]:= Table[StirlingFirst[4,i],{i,4,1,-1}]
Out[38]= {1, 6, 11, 6}
```

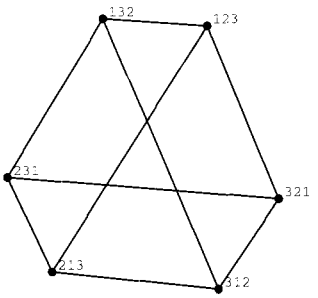
The function `MakeGraph` used above is one of several graph construction functions that *Combinatorica* provides. `MakeGraph` takes as input a set of objects (permutations, in the above example) and a relation defined on these objects and produces a graph that represents the relation. Section 6.5.1 provides details of this and other graph construction functions. The graph construction functions in *Combinatorica* are complemented by a variety of graph drawing functions. `RankedEmbedding` has been used above to arrange the vertices by distance from a specific vertex.

https://doi.org/10.1017/CBO9781139164849.003 Published online by Cambridge University Press

We next show the sequence of permutations produced by `MinimumChangePermutation` as a path in a transposition graph. We show this on a transposition graph of size-3 permutations because this graph is less crowded than the transposition graph of size-4 permutations.

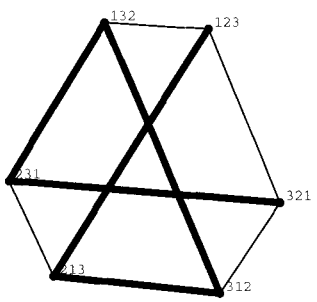
The transposition graph on size-3 permutations. `SpringEmbedding` *tries* (not always successfully) to reveal the symmetry and structure of the graph. Here it does a good job. Spring embeddings and other types of graph drawings are discussed in Section 5.5.

```
In[39]:= ShowGraph[tg3 = SpringEmbedding[MakeGraph[Permutations[3],
           tr, Type -> Undirected, VertexLabel -> True]],
           PlotRange -> 0.15];
```



The sequence of permutations from `MinimumChangePermutations` is shown as a path in this graph. This path is a *Hamiltonian path*, that is, a path that visits every vertex of the graph without repetition. Since the first and the last vertices of the path are adjacent, the path can be extended to a *Hamiltonian cycle*.

```
In[40]:= m = Partition[ Map[RankPermutation[#] + 1 &,
           MinimumChangePermutations[{1, 2, 3}]], 2, 1];
           ShowGraph[Highlight[tg3, {m},
           HighlightedEdgeColors->{Blue}], PlotRange -> 0.15];
```



However, this is not true in general; the first and the last size-4 permutations produced by `MinimumChangePermutations` are not adjacent in the transposition graph.

```
In[42]:= {First[MinimumChangePermutations[4]],
           Last[MinimumChangePermutations[4]]}
Out[42]= {{1, 2, 3, 4}, {2, 3, 4, 1}}
```

https://doi.org/10.1017/CBO9781139164849.003 Published online by Cambridge University Press

Permutations can always be sequenced in *maximum change order*, where neighboring permutations differ in all positions unless $n = 3$. Such a sequence can be constructed by finding a Hamiltonian path on the appropriate adjacency graph [Wil89]. This graph is disconnected for $n = 3$, but [EW85, RS87] show how to enumerate n-permutations in maximum change order for all $n \geq 4$. See Section 3.2.2 for more details.

```
In[43]:= Permutations[{1,2,3,4}] [[ HamiltonianPath[
         MakeGraph[Permutations[{1,2,3,4}], (Count[#1-#2,0] == 0)&]
         ]]]
Out[43]= {{1, 2, 3, 4}, {2, 1, 4, 3}, {1, 3, 2, 4},
    {2, 4, 1, 3}, {1, 3, 4, 2}, {2, 1, 3, 4}, {1, 2, 4, 3},
    {2, 3, 1, 4}, {1, 4, 2, 3}, {2, 3, 4, 1}, {1, 4, 3, 2},
    {3, 1, 2, 4}, {2, 4, 3, 1}, {4, 1, 2, 3}, {3, 2, 1, 4},
    {4, 1, 3, 2}, {3, 2, 4, 1}, {4, 3, 1, 2}, {3, 4, 2, 1},
    {4, 2, 1, 3}, {3, 1, 4, 2}, {4, 2, 3, 1}, {3, 4, 1, 2},
    {4, 3, 2, 1}}
```

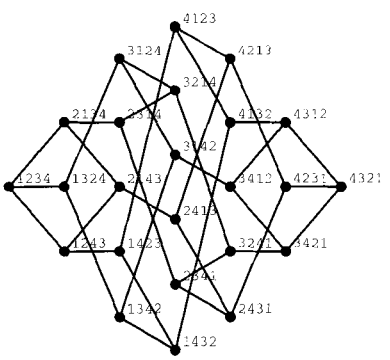
We could get even more ambitious and ask if permutations can be generated in an order in which each permutation is obtained from the previous by a transposition of *adjacent* elements. The *adjacent transposition graph* contains an edge for each pair of permutations that differ by exactly one *adjacent* transposition. The adjacent transposition graph is a *spanning subgraph* of the transposition graph because it contains *all* of the vertices of the transposition graph and some of the edges.

This function relates pairs of permutations that can be obtained from a transposition of adjacent elements.

```
In[44]:= atr = MemberQ[Table[p = #1; {p[[i]], p[[i+1]]} =
         {p[[i+1]], p[[i]]}; p, {i, Length[#1] - 1}], #2]&;
```

This is the *adjacent transposition graph* for 4-permutations. In this drawing, the graph displays a nice symmetry, but the graph can also be drawn in other pleasing ways.

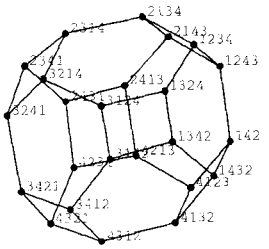
```
In[45]:= ShowGraph[RankedEmbedding[g = MakeGraph[
         Permutations[4], atr, Type -> Undirected,
         VertexLabel -> True], {1}], PlotRange->0.1];
```



https://doi.org/10.1017/CBO9781139164849.003 Published online by Cambridge University Press

Knuth [Knu73b] shows this graph drawn as a *truncated octahedron*, a 14-faced polyhedron that can be obtained by lopping off each of the 6 corners of a regular octahedron. `SpringEmbedding` constructs this embedding for us.

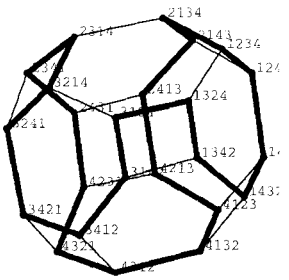
```
In[46]:= ShowGraph[ats4 = SpringEmbedding[g], PlotRange->0.15];
```



A Hamiltonian path of an adjacent transposition graph corresponds to an ordering of permutations in which successive permutations differ by exactly one adjacent transposition. The Johnson-Trotter permutation generation algorithm [Joh63, Tro62] is a proof that every adjacent transposition graph contains a Hamiltonian path. In fact, Chase [Cha73] shows that the transposition graph of a multiset always contains a Hamiltonian path.

This Hamiltonian cycle starts with 1234, 1243, 1423, 1432, ... After 1432 the Johnson-Trotter algorithm would have produced 4132. This is proof that this graph contains more than one Hamiltonian path starting at 1234.

```
In[47]:= ShowGraph[Highlight[ats4,
           {Partition[HamiltonianCycle[ats4], 2, 1]},
           HighlightedEdgeColors->{Blue}]];
```



In fact, it contains many Hamiltonian cycles, 88 to be exact! This inspires the question: As a function of n, how many distinct Hamiltonian cycles does the adjacent transposition graph on n-permutations have?

```
In[48]:= Length[HamiltonianCycle[ats4, All]]
Out[48]= 88
```

In the past two decades, researchers have focused on problems of enumerating combinatorial objects in an order in which successive objects differ in some prespecified small way. The term *combinatorial Gray code* was introduced in 1980 to refer to any method that enumerates combinatorial objects in this manner. Gray codes are known for a large number of combinatorial objects [Sav97].

https://doi.org/10.1017/CBO9781139164849.003 Published online by Cambridge University Press

2.2 Inversions and Inversion Vectors

A pair of elements $(\pi(i), \pi(j))$ in a permutation π represents an *inversion* if $i > j$ and $\pi(i) < \pi(j)$. An inversion is a pair of elements that are out of order, and so they play a prominent role in the analysis of sorting algorithms.

2.2.1 Inversion Vectors

For any n-permutation π, we can define an *inversion vector* v as follows. For each integer i, $1 \le i \le n-1$, the ith element of v is the number of elements in π greater than i to the left of i. The function `ToInversionVector`, shown below, computes the inversion vector of a given permutation. As implemented, it runs in $\Theta(n^2)$ time; however, more sophisticated algorithms exist that can compute this in $O(n \lg n)$ time.

```
ToInversionVector[p_?PermutationQ] :=
      Module[{i,inverse=InversePermutation[p]},
            Table[ Length[ Select[Take[p,inverse[[i]]], (# > i)&] ], {i,Length[p]-1}]
      ] /; (Length[p] > 0)
```

Obtaining the Inversion Vector

The inversion vector contains only $n-1$ elements since the number of inversions of n is always 0. The ith element can range from 0 to $n-i$, so there are indeed $n!$ distinct inversion vectors, one for each permutation.

```
In[49]:= ToInversionVector[{5,9,1,8,2,6,4,7,3}]
Out[49]= {2, 3, 6, 4, 0, 2, 2, 1}
```

Marshall Hall [Tho56] demonstrated that no two permutations have the same inversion vector and also showed how to obtain the corresponding permutation from an inversion vector. Let π be an n-permutation and let v be its inversion vector. Suppose we have the sequence S obtained by ordering elements in $\{i+1, i+2, \ldots, n\}$ according to their order in π. The value v_i tells us the number of elements larger than i that occur to the left of i in π, so we can insert i into S v_i positions from the left. This gives us a correct ordering of the elements in $\{i, i+1, \ldots, n\}$. We start with the list $\{n\}$ and iteratively insert each i from $n-1$ down to 1 in the list, obtaining π. The function `FromInversionVector` is shown below.

```
FromInversionVector[vec_List] :=
      Module[{n=Length[vec]+1,i,p}, p={n}; Do [p = Insert[p, i, vec[[i]]+1], {i,n-1,1,-1}]; p]
```

Inverting the Inversion Vector

https://doi.org/10.1017/CBO9781139164849.003 Published online by Cambridge University Press

A permutation and its inversion vector provide different representations of the same structure.

```
In[50]:= {p=RandomPermutation[10], ToInversionVector[p]}
Out[50]= {{2, 3, 9, 10, 4, 6, 1, 5, 7, 8},
    {6, 0, 0, 2, 3, 2, 2, 2, 0}}
```

Composing these two functions gives an identity operation.

```
In[51]:= FromInversionVector[ToInversionVector[{5,9,1,8,2,6,4,7,3}] ]
Out[51]= {5, 9, 1, 8, 2, 6, 4, 7, 3}
```

Random inversion vectors are easy to construct because each individual element is generated independently. Here we generate random inversion vectors of size 4, convert them into permutations, and demonstrate that all 24 permutations occur with roughly equal frequency.

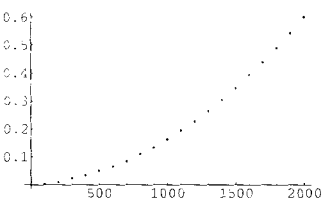
```
In[52]:= Distribution[Map[FromInversionVector, Table[Table[
          Random[Integer, {0, 4-i}], {i, 3}], {10000}]]]
Out[52]= {418, 414, 412, 410, 390, 411, 436, 403, 418, 399,
    437, 445, 446, 405, 397, 378, 404, 397, 419, 412, 464,
    424, 442, 419}
```

This generates five random inversion vectors for different sizes up to 2000 to serve as test data for timings.

```
In[53]:= Map[Length, pt = Table[Table[Random[Integer, {0, n - i}],
          {i, n - 1}], {n, 0, 2000, 100}, {5}] ]
Out[53]= {5, 5, 5, 5, 5, 5, 5, 5, 5, 5, 5, 5, 5, 5, 5, 5,
    5, 5, 5, 5, 5}
```

We timed `FromInversionVector` on each of these random inversion vectors. The quadratic behavior of `FromInversionVector` shows up fairly clearly in this plot.

```
In[54]:= ListPlot[Map[ ({Length[First[#]], Mean[ Table[
             Timing[FromInversionVector[ #[[j]] ];][[1,1]] , {j, 5}] ]})&,
           pt] ]
```



Another method of highlighting the inversions in a permutation uses permutation graphs. The *permutation graph* G_π of a permutation π is a graph whose vertex set is $\{1,2,\ldots,n\}$ and whose edges $\{i,j\}$ correspond exactly to (i,j) being an inversion in permutation π. The structure of permutation graphs permits fast algorithms for certain problems that are intractable for general graphs [AMU88, BK87].

Here is a 15-element permutation h whose structure we will explore.

```
In[55]:= h=RandomPermutation[15]
Out[55]= {11, 10, 13, 3, 5, 15, 2, 4, 1, 8, 9, 7, 12, 14, 6}
```

https://doi.org/10.1017/CBO9781139164849.003 Published online by Cambridge University Press

Here we compute the degrees of vertices in the permutation graph of h. Elements responsible for many inversions have high degree, while vertices of zero degree are in the correct position with smaller elements appearing earlier and larger elements appearing later.

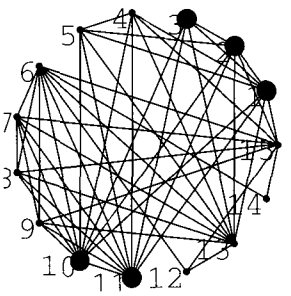
```
In[56]:= Degrees[g=PermutationGraph[h]]
Out[56]= {8, 7, 5, 6, 6, 9, 7, 6, 6, 10, 10, 3, 10, 2, 9}
```

The total number of inversions in a permutation is equal to the number of edges in its permutation graph.

```
In[57]:= {Inversions[h], M[g]}
Out[57]= {52, 52}
```

Here we highlight a maximum-size *clique* in the permutation graph. A clique is a subset of the vertices, such that every pair of vertices in the subset is connected by an edge. Although finding a maximum-size clique is difficult even to approximate [GJ79, Hoc97], the problem is easy for permutation graphs. Every clique in a permutation graph corresponds to a decreasing sequence in the *corresponding permutation*.

```
In[58]:= ShowGraph[Highlight[g, {MaximumClique[g]},
         HighlightedVertexColors -> {Blue}], VertexNumber -> True,
         TextStyle -> {FontSize -> 12}];
```



There is a longest decreasing subsequence in the permutation whose size is the same as the size of a maximum clique in its permutation graph.

```
In[59]:= LongestIncreasingSubsequence[Reverse[h]]
Out[59]= {1, 2, 3, 10, 11}
```

■ 2.2.2 Counting Inversions

The total number of inversions in a permutation is a classic measure of order (or disorder) [Knu73b, Man85a, Ski88] and can be obtained by summing up the inversion vector.

```
Inversions[{}] := 0
Inversions[p_?PermutationQ] := Apply[Plus,ToInversionVector[p]]
```

Counting Inversions

https://doi.org/10.1017/CBO9781139164849.003 Published online by Cambridge University Press

The number of inversions in a permutation is equal to that of its inverse [Knu73b]. The claim follows immediately from the fact that $(\pi(i), \pi(j))$ is an inversion in π if and only if (i, j) is an inversion in π^{-1}.

```
In[60]:= p = RandomPermutation[30];
         {Inversions[p], Inversions[InversePermutation[p]]}
Out[61]= {230, 230}
```

The number of inversions in a permutation ranges from 0 to $\binom{n}{2}$, where the largest number of inversions comes from the reverse of the identity permutation.

```
In[62]:= {Inversions[Reverse[Range[100]]], Binomial[100, 2]}
Out[62]= {4950, 4950}
```

Every number from 0 to $\binom{n}{2}$ is realizable as an inversion total for some permutation.

```
In[63]:= Union[Map[Inversions, Permutations[5]]]
Out[63]= {0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10}
```

A neat proof that there are an average of $n(n-1)/4$ inversions per permutation is that the number of inversions in a permutation and its reverse permutation always total to $\binom{n}{2}$.

```
In[64]:= ( p = RandomPermutation[100];
         Inversions[p] + Inversions[Reverse[p]] )
Out[64]= 4950
```

Thus it is easy to find the number of inversions in a permutation. But the "inverse" question of how many n-permutations have exactly k inversions is harder. The easiest solution uses *generating functions*. Generating functions [Wil90] provide some of the most powerful tools for dealing with combinatorial problems. The *ordinary generating function* of a sequence $(g_0, g_1, g_2, \ldots)$ is a function $G(z)$ defined as

$$G(z) = g_0 + g_1 \cdot z + g_2 \cdot z^2 + g_3 \cdot z^3 + \cdots .$$

Thus $G(z)$ is just a representation of the sequence $(g_0, g_1, g_2, \ldots)$. However, it is also an algebraic object that can be subjected to a variety of algebraic operations that often yield a great deal of information about the sequence.

Let $I_n(k)$ denote the number of n-permutations with k inversions. A more useful way to think about $I_n(k)$ is that it is the number of inversion vectors of n-permutations that sum up to k. So the relevant objects are inversion vectors $(v_1, v_2, \ldots, v_{n-1})$ that sum to k. Note that v_1 can contribute any integer in the range $[0, n-1]$ to the sum k, v_2 can contribute any integer in the range $[0, n-2]$ to the sum k, and so on. Now consider the product

$$(1) \cdot (1 + z) \cdot (1 + z + z^2) \cdots (1 + z + z^2 + \cdots + z^{n-1}).$$

The possible contributions of v_1 are represented by the powers of the terms in the last bracket, the contributions of v_2 are represented by the powers of the terms in the second to last bracket, and so on. Thus the coefficient of z^k in the product is exactly $I_n(k)$. This implies that

$$I(z) = \sum_{k=0} I_n(k) z^k = \prod_{i=1}^{n-1} (1 + z + z^2 + \cdots + z^i) = \frac{1}{(1-z)^n} \prod_{i=1}^{n-1} (1 - z^i).$$

https://doi.org/10.1017/CBO9781139164849.003 Published online by Cambridge University Press

Expanding this polynomial and picking the right coefficient provides an approach to computing $I_n(k)$.

The coefficients of this polynomial are $I_7(k)$ for each k between 0 and 21. Observe that $I_n(k) = I_n(\binom{n}{2} - k)$ for all k. This follows from the fact that any pair of elements that is an inversion in a permutation p is not so in the reverse of p.

```
In[65]:= p = Expand[Product[Cancel[(z^i - 1)/(z - 1)], {i, 7}]]

                                  2      3      4      5
Out[65]= 1 + 6 z + 20 z  + 49 z  + 98 z  + 169 z  +

       6        7        8        9         10         11
  259 z  + 359 z  + 455 z  + 531 z  + 573 z   + 573 z   +

       12         13         14         15         16
  531 z   + 455 z   + 359 z   + 259 z   + 169 z   +

      17       18       19       20    21
  98 z   + 49 z   + 20 z   + 6 z   + z
```

The *Mathematica* function `Coefficient` can be used to pick out coefficients of specific terms of a polynomial.

```
In[66]:= p = Coefficient[p, z, 16]

Out[66]= 169
```

This coefficient gives the number of 7-element permutations with 16 inversions.

```
In[67]:= NumberOfPermutationsByInversions[7,16]

Out[67]= 169
```

The function below implements the above approach of using generating functions to compute the number of n-permutations with k inversions.

```
NumberOfPermutationsByInversions[n_Integer?Positive] :=
     Module[{p = Expand[Product[Cancel[ (z^i -1)/(z-1)], {i, 1, n}]]}, CoefficientList[p, z]]
NumberOfPermutationsByInversions[n_Integer, k_Integer] := 0 /; (k > Binomial[n,2])
NumberOfPermutationsByInversions[n_Integer, 0] := 1
NumberOfPermutationsByInversions[n_Integer, k_Integer?Positive] := NumberOfPermutationsByInversions[n][[k+1]]
```

Computing the Number of Permutations with k Inversions

■ 2.2.3 The Index of a Permutation

The *index* of a permutation π is the sum of all subscripts j such that $\pi(j) > \pi(j+1)$, $1 \le j < n$. MacMahon [Mac60] proved that the number of permutations of size n having index k is the same as the number having exactly k inversions.

```
Index[p_?PermutationQ]:= Module[{i}, Sum[ If [p[[i]] > p[[i+1]], i, 0], {i,Length[p]-1} ]]
```

Computing the Index of a Permutation

These are the six permutations of length 4 with index 3.

```
In[68]:= Select[Permutations[{1,2,3,4}], (Index[#]==3)&]

Out[68]= {{1, 2, 4, 3}, {1, 3, 4, 2}, {2, 3, 4, 1},

    {3, 2, 1, 4}, {4, 2, 1, 3}, {4, 3, 1, 2}}
```

https://doi.org/10.1017/CBO9781139164849.003 Published online by Cambridge University Press

As MacMahon proved, there is an equal number of permutations of length 4 with three inversions. A bijection between the two sets of permutations is not obvious [Knu73b].

```
In[69]:= Select[Permutations[{1,2,3,4}], (Inversions[#]==3)&]
Out[69]= {{1, 4, 3, 2}, {2, 3, 4, 1}, {2, 4, 1, 3},
    {3, 1, 4, 2}, {3, 2, 1, 4}, {4, 1, 2, 3}}
```

Sixty-five years after MacMahon's result was published, Foata and Schützenberger discovered a remarkable extension: The number of permutations with k inversions and index ℓ is equal to the number of permutations with ℓ inversions and index k. This result immediately implies MacMahon's result.

Evidence for the Foata-Schützenberger bijection. For $n = 7$, the number of permutations with four inversions and index 10 equals number of permutations with ten inversions and index 4.

```
In[70]:= p = Permutations[Range[7]];
         {Length[Select[p, ((Index[#]==10) && (Inversions[#]==4))&]],
          Length[Select[p, ((Index[#]==4 ) && (Inversions[#]==10))&]]}
Out[71]= {9, 9}
```

■ 2.2.4 Runs and Eulerian Numbers

Read from left to right, any permutation can be partitioned into a set of ascending sequences or *runs*. A sorted permutation consists of one run, while a reverse permutation consists of n runs, each of one element. For this reason, the number of runs is also used as a measure of the presortedness of a permutation [Man85a]. *Combinatorica* contains a function Runs to partition a permutation into runs.

A random 20-permutation...

```
In[72]:= p = RandomPermutation[20]
Out[72]= {7, 11, 17, 20, 3, 8, 16, 12, 10, 13, 18, 9, 14,
    1, 4, 2, 19, 5, 15, 6}
```

...and its runs.

```
In[73]:= Runs[p]
Out[73]= {{7, 11, 17, 20}, {3, 8, 16}, {12}, {10, 13, 18},
    {9, 14}, {1, 4}, {2, 19}, {5, 15}, {6}}
```

The number of runs in an n-permutation plus the number of runs in its reverse equals $(n + 1)$.

```
In[74]:= Length[Runs[p = RandomPermutation[1000]]] +
         Length[ Runs[ Reverse[p]]]
Out[74]= 1001
```

Interestingly, the average length of the first run is shorter than the average length of the second run [Gas67, Knu73b].

```
In[75]:= Map[N[Mean[#]] &, Transpose[Table[Map[Length,
         Runs[RandomPermutation[100]]][[{1,2}]], {500}]]]
Out[75]= {1.792, 1.844}
```

Why? The first element of the first run is an average element of the permutation, while the first element of the second run must be less than the previous element in the permutation.

```
In[76]:= Map[N[Mean[#]] &, Transpose[Table[Map[First,
         Runs[RandomPermutation[100]]][[{1,2}]], {500}]]]
Out[76]= {50.368, 34.608}
```

https://doi.org/10.1017/CBO9781139164849.003 Published online by Cambridge University Press

The *Eulerian* numbers $\left\langle {n \atop k} \right\rangle$ count the number of permutations π of length n with exactly k "descents," $\pi(j) > \pi(j+1)$. This is the same as the number of permutations with $(k+1)$ runs. A recurrence for the Eulerian numbers follows by noting that a size-n permutation π with $(k+1)$ runs can be built only in the following ways:

(i) by taking an $(n-1)$-permutation with $(k+1)$ runs and appending n to one of the runs;

(ii) by taking a $(n-1)$-permutation with k runs and inserting n into a run, thereby splitting it into two runs.

It is easy to see that every size-n permutation with $(k+1)$ runs can be obtained by (i) or (ii) and no size-n permutation can be obtained via both (i) and (ii). This implies the recurrence

$$\left\langle {n \atop k} \right\rangle = (k+1)\left\langle {n-1 \atop k} \right\rangle + (n-k)\left\langle {n-1 \atop k-1} \right\rangle, \quad n \geq 1,\ k \geq 1,$$

where $\left\langle {n \atop 0} \right\rangle$ counts the number of permutations with a single run and is therefore 1, for all $n \geq 1$.

The function `Eulerian`, shown below, uses this recurrence relation to compute Eulerian numbers. The function provides an example of *dynamic programming* in *Mathematica*. Whenever the value of $\left\langle {n \atop k} \right\rangle$ is computed, it is stored in the symbol `Eulerian1[n, k]`. This means that, whenever this value is needed in the future, a constant-time table lookup is performed instead of an expensive recomputation.

```
Eulerian[n_Integer,k_Integer] := Block[{$RecursionLimit = Infinity}, Eulerian1[n, k]]
Eulerian1[0,k_Integer] := If [k==0, 1, 0]
Eulerian1[n_Integer, k_Integer] := 0 /; (k >= n)
Eulerian1[n_Integer, 0] := 1
Eulerian1[n_Integer,k_Integer] := Eulerian1[n,k] = (k+1) Eulerian1[n-1,k] + (n-k) Eulerian1[n-1,k-1]
```

Counting the Number of Permutations with k Runs

A list of the number of size-10 permutations with i runs, $0 \leq i \leq 9$. The symmetry of this list indicates that $\left\langle {n \atop k} \right\rangle = \left\langle {n \atop n-1-k} \right\rangle$.

```
In[77]:= Table[ Eulerian[10, i], {i, 0, 9}]
Out[77]= {1, 1013, 47840, 455192, 1310354, 1310354, 455192,
    47840, 1013, 1}
```

This follows from the fact that if π has $(k+1)$ runs, then the reverse of π has $(n-k)$ runs.

```
In[78]:= {Length[Runs[p=RandomPermutation[100]]], Length[Runs[Reverse[p]]]}
Out[78]= {44, 57}
```

The symmetry $\left\langle {n \atop k} \right\rangle = \left\langle {n \atop n-1-k} \right\rangle$ implies that the average number of descents in a size-n permutation is $(n-1)/2$.

```
In[79]:= Sum[r = Length[Runs[RandomPermutation[100]]]-1, {200}]/200 //N
Out[79]= 49.655
```

https://doi.org/10.1017/CBO9781139164849.003 Published online by Cambridge University Press

2.3 Combinations

Subsets rank with permutations as the most ubiquitous combinatorial objects. As with permutations, subsets can be generated in a variety of interesting orders. In this section we will examine three such orders.

An important subset of the subsets is those containing the same number of elements. In elementary combinatorics, permutations are always mentioned in the same breath as *combinations*, which are collections of elements of a given size where order does not matter. Thus a combination is exactly a set with k elements or a *k-subset* for some k, and we will use the terms interchangeably.

2.3.1 Subsets via Binary Representation

An element of a set is either in a particular subset or not in it. Thus a sequence of n bits, with one bit for each of the n set elements, can represent any subset. This binary representation provides a method for generating all subsets, for we can iterate through the 2^n distinct binary strings of length n and use each as a descriptor of a subset. This can be done by simply calling the function `Strings` (see Section 2.3.5) to generate all binary strings of length n. The bijection between length-n binary strings and the set of integers $\{0, 1, \ldots, 2^n - 1\}$ given by

$$b_{n-1}b_{n-2}\ldots b_2 b_1 b_0 \iff \sum_{i=0}^{n-1} 2^i b_i$$

is in fact the standard way of representing integers in computers. When interpreted as integers, the length-n binary strings produced by the function `Strings` occur in the order $0, 1, \ldots, 2^n - 1$. This is rather convenient for ranking, unranking, and finding the next subset. For example, the function `Unrank` simply needs to compute the binary representation of the given integer. *Mathematica* has a built-in function called `Digits` to do precisely this.

```
BinarySubsets[l_List] := Map[(l[[Flatten[Position[#, 1], 1]]])&, Strings[{0, 1}, Length[l]]]

BinarySubsets[0] := {{}}
BinarySubsets[n_Integer?Positive] := BinarySubsets[Range[n]]

NextBinarySubset[set_List,subset_List] := UnrankBinarySubset[RankBinarySubset[set,subset]+1, set]

RankBinarySubset[set_List,subset_List] :=
      Module[{i,n=Length[set]},
            Sum[ 2^(n-i) * If[ MemberQ[subset,set[[i]]], 1, 0], {i,n}]
      ]

UnrankBinarySubset[n_Integer, 0] := {}
```

https://doi.org/10.1017/CBO9781139164849.003 Published online by Cambridge University Press

```
UnrankBinarySubset[n_Integer, m_Integer?Positive] := UnrankBinarySubset[Mod[n, 2^m], Range[m]]
UnrankBinarySubset[n_Integer, l_List] :=
      l[[Flatten[Position[IntegerDigits[Mod[n, 2^Length[l]], 2, Length[l]], 1], 1]]]
```

Subsets from Bit Strings

The representation of each subset is obtained by incrementing the binary representation of the previous subset. This corresponds to inserting the last missing element into a subset and then removing all subsequent elements.

```
In[80]:= Clear[a, b, c, d]; BinarySubsets[{a,b,c,d}]
Out[80]= {{}, {d}, {c}, {c, d}, {b}, {b, d}, {b, c},
    {b, c, d}, {a}, {a, d}, {a, c}, {a, c, d}, {a, b},
    {a, b, d}, {a, b, c}, {a, b, c, d}}
```

Here we start with the empty set and generate all subsets by repeatedly calling `NextBinarySubset`. Generating subsets incrementally is efficient if we seek the first subset with a given property, since not every subset need be constructed.

```
In[81]:= NestList[NextBinarySubset[{a, b, c, d}, #] &, {}, 15]
Out[81]= {{}, {d}, {c}, {c, d}, {b}, {b, d}, {b, c},
    {b, c, d}, {a}, {a, d}, {a, c}, {a, c, d}, {a, b},
    {a, b, d}, {a, b, c}, {a, b, c, d}}
```

Because n is taken modulo the number of subsets, any positive or negative integer can be used to specify the rank of a subset.

```
In[82]:= UnrankBinarySubset[-10,{a,b,c,d}]
Out[82]= {b, c}
```

The amount of work done in transforming one subset to another can be measured by the number of insertion and deletion operations needed for this task. For example, the subsets $\{b, c, d\}$ and $\{a\}$ occur consecutively in the output of `BinarySubsets[{a, b, c, d}]`. Four operations – three deletions and one insertion – are needed to transform $\{b, c, d\}$ into $\{a\}$. How many insertion-deletion operations are performed by `BinarySubsets` in generating all 2^n subsets? This is examined in the examples below.

Here we calculate the average number of insertion-deletion operations it takes to generate all subsets of an n-element set, for n varying from 1 to 10. To compute the number of operations needed to go from a set s to its successor, we calculate the largest element i missing from a subset s, noting that $(n - i + 1)$ is the number of operations needed.

```
In[83]:= Table[s = BinarySubsets[Range[n]];
        Mean[Table[n - Max[Complement[Range[n], s[[i]]]] + 1,
        {i, 2^n - 1}]], {n, 10}] // N
Out[83]= {1., 1.33333, 1.57143, 1.73333, 1.83871, 1.90476,
    1.94488, 1.96863, 1.98239, 1.99022}
```

In fact, the total number of insertion-deletion operations performed by `BinarySubsets` in generating all 2^n subsets of an n-element set is $2^{n+1} - (n + 2)$.

```
In[84]:= Table[ N[ (2^(n+1)-(n+2)) / (2^n-1)], {n,10}]
Out[84]= {1., 1.33333, 1.57143, 1.73333, 1.83871, 1.90476,
    1.94488, 1.96863, 1.98239, 1.99022}
```

Ranking combinatorial structures makes it easy to generate random instances of them, for we can pick a random integer between 0 and $M - 1$, where M is the number of structures, and perform an

https://doi.org/10.1017/CBO9781139164849.003 Published online by Cambridge University Press

unranking operation. Picking a random integer between 0 and $2^n - 1$ is equivalent to flipping n coins, each determining whether an element of the set appears in the subset.

```
RandomSubset[set_List] := UnrankSubset[Random[Integer,2^(Length[set])-1],set]

RandomSubset[0] := {}
RandomSubset[n_Integer] := UnrankSubset[Random[Integer,2^(n)-1], Range[n]]
```

Generating Random Subsets

Since the sizes of random subsets obey a binomial distribution, very large or very small subsets are rare compared to subsets with approximately half the elements.

```
In[85]:= Distribution[ Table[ Length[RandomSubset[Range[10]]], {1024}],
         Range[0,10] ]
Out[85]= {1, 14, 34, 124, 217, 230, 226, 122, 44, 11, 1}
```

Here is the actual binomial distribution, for comparison.

```
In[86]:= Table[Binomial[10, i], {i, 0, 10}]
Out[86]= {1, 10, 45, 120, 210, 252, 210, 120, 45, 10, 1}
```

However, each distinct subset occurs with roughly equal frequency since `RandomSubset` picks subsets uniformly at random.

```
In[87]:= Distribution[ Table[ RandomSubset[{1,2,3,4}], {1024}],
         Subsets[{1,2,3,4}] ]
Out[87]= {72, 59, 44, 69, 71, 66, 59, 64, 70, 75, 79, 57,
    62, 54, 57, 66}
```

■ 2.3.2 Gray Codes

We have seen that permutations can be generated to limit the difference between adjacent permutations to one transposition. Combinations can also be generated in "minimum change order." The study of combinatorial Gray codes traces it origins to an elegant enumeration devised by Frank Gray and patented in 1953.

Gray's scheme [Gra53], subsequently referred to as the *standard reflected Gray code*, is an ordering of subsets so that adjacent subsets differ by the insertion or deletion of exactly one element. It can be described as follows. Say our goal is to construct a reflected Gray code G_n of subsets of a size-n set. Suppose that we have a Gray code G_{n-1} of subsets of the last $(n-1)$ elements of the set. Concatenate G_{n-1} with a reversed copy of G_{n-1}. Clearly, all subsets differ by one from their neighbors, except in the center, where they are identical. Adding the first element to the subsets in the bottom half maintains the Gray code property while distinguishing the center two subsets by exactly one element. This gives us G_n.

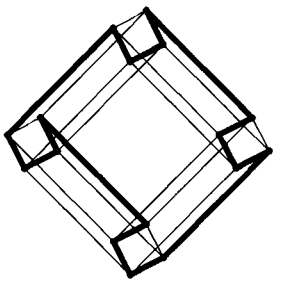
Many different Gray codes are possible. This technique of taking an enumeration of substructures, concatenating it with a reflection of the enumeration, and then extending it to an enumeration of the entire set of structures pervades most Gray code constructions.

https://doi.org/10.1017/CBO9781139164849.003 Published online by Cambridge University Press

When subsets are viewed as bit strings, a Gray code of a size-n set corresponds to a sequence of n-bit strings in which each bit string differs from its neighbors by exactly one bit. This sequence is exactly a Hamiltonian path in a well-known graph, the *n-dimensional hypercube*. The vertex set of an n-dimensional hypercube is the set of all n-bit strings, and its edges connect all pairs of strings that differ by exactly one bit. The existence of a Gray code is proof that an n-dimensional hypercube has a Hamiltonian path.

In the reflected Gray code, the last subset differs from the first by exactly one element, implying the existence of a Hamiltonian cycle in the n-dimensional hypercube. Here a four-dimensional hypercube is constructed using the *Combinatorica* function `Hypercube`. A Hamiltonian cycle in this graph is constructed, giving us a Gray code for subsets of a size-4 set.

```
In[88]:= ShowGraph[Highlight[g=Hypercube[4], {Partition[
         HamiltonianCycle[g], 2, 1]}, HighlightedEdgeColors->{Blue}]]
```



There is nothing unique about a Gray code, since each Hamiltonian cycle in the hypercube corresponds to a distinct Gray code. The exact number of Hamiltonian cycles in an n-dimensional hypercube is not known, not even asymptotically [Wil89].

```
In[89]:= Length[HamiltonianCycle[Hypercube[4], All]]
Out[89]= 2688
```

```
GrayCodeSubsets[n_Integer?Positive] := GrayCodeSubsets[Range[n]]

GrayCodeSubsets[ { } ] := { {} }

GrayCodeSubsets[l_List] :=
      Block[{s, $RecursionLimit = Infinity},
            s = GrayCodeSubsets[Take[l, 1-Length[l]]];
            Join[s,  Map[Prepend[#, First[l]] &, Reverse[s]]]
      ]
```

Constructing Reflected Gray Codes

Each subset differs in exactly one element from its neighbors. Observe that the last eight subsets all contain 1, while none of the first eight do.

```
In[90]:= GrayCodeSubsets[{1,2,3,4}]
Out[90]= {{}, {4}, {3, 4}, {3}, {2, 3}, {2, 3, 4}, {2, 4},
    {2}, {1, 2}, {1, 2, 4}, {1, 2, 3, 4}, {1, 2, 3},
    {1, 3}, {1, 3, 4}, {1, 4}, {1}}
```

https://doi.org/10.1017/CBO9781139164849.003 Published online by Cambridge University Press

GrayCodeSubsets is much faster than BinarySubsets.

```
In[91]:= Table[Timing[ Length[BinarySubsets[ Range[i]]];][[1, 1]]/
          Timing[ Length[GrayCodeSubsets[ Range[i]]];][[1, 1]],
          {i, 10, 15}]
Out[91]= {4.66667, 7.25, 5.45455, 7.94118, 7.22222, 7.4}
```

Reflected Gray codes have many beautiful properties. One of these leads to elegant algorithms for ranking and unranking subsets listed in this order. Let integer $m < 2^n$ have binary representation $b_{n-1}b_{n-2}\ldots b_1b_0$, that is, $m = \sum_{i=0}^{n-1} b_i 2^i$. If $c_{n-1}c_{n-2}\ldots c_1c_0$ is the binary representation of the subset with rank m in the reflected Gray code, then it can be shown that [Wil89]

$$c_i = (b_i + b_{i+1}) \pmod 2 \text{ for all } i, 0 \le i \le n-1.$$

Here we suppose that $b_n = 0$. This immediately leads to an unranking algorithm that computes the binary representation of rank $m = b_{n-1},\ldots,b_2b_1b_0$; uses the above result to compute $C = c_0, c_1,\ldots,c_{n-1}$, the binary representation of the subset; and finally computes the subset itself from C.

The above equation connecting the c_i's and the b_i's can be easily reversed to obtain $b_i = (c_i + c_{i+1} + \cdots + c_{n-1}) \pmod 2$ for all i, $0 \le i \le n-1$. This immediately leads to a ranking algorithm. Other properties of Gray codes are discussed in [Gil58].

```
UnrankGrayCodeSubset[0, {}] := {}

UnrankGrayCodeSubset[m_Integer, s_List] :=
      Module[{c = Table[0, {Length[s]}], n = Length[s], b, nm},
            nm = Mod[m, 2^n];
            b = IntegerDigits[nm, 2, Length[s]];
            c[[ 1 ]] = b[[1]];
            Do[c[[i]] = Mod[b[[i]] + b[[i-1]], 2], {i, 2, n}];
            s[[ Flatten[Position[c, 1], 1] ]]
      ]
RankGrayCodeSubset[l_List, s_List] :=
     Module[{c = Table[If[MemberQ[s, l[[i]]], 1, 0], {i, Length[l]}], b = Table[0, {Length[l]}], n = Length[l]},
            b[[ 1 ]] = c[[ 1 ]];
            Do[b[[i]] = Mod[b[[i - 1]] + c[[i]], 2], {i, 2, n}];
            FromDigits[b, 2]
      ]
```

Ranking and Unranking Binary Reflected Gray Codes

Unranking the sequence 0 through 15 produces subsets in Gray code order.

```
In[92]:= Table[ UnrankGrayCodeSubset[i, {a, b, c, d}], {i, 0, 15}]
Out[92]= {{}, {d}, {c, d}, {c}, {b, c}, {b, c, d}, {b, d},
    {b}, {a, b}, {a, b, d}, {a, b, c, d}, {a, b, c},
    {a, c}, {a, c, d}, {a, d}, {a}}
```

Ranking these inverts the process to get back to the sequence 0 through 15.

```
In[93]:= Map[ RankGrayCodeSubset[{a, b, c, d}, #] &, %]
Out[93]= {0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14,
    15}
```

https://doi.org/10.1017/CBO9781139164849.003 Published online by Cambridge University Press

The algorithms for ranking and unranking subsets in Gray code order immediately give us an algorithm for getting the next subset of a given subset in Gray code order: rank, increment, and unrank. A more interesting and efficient algorithm is hinted at by the following example.

Here we list the subsets of $\{a,b,c,d\}$ in Gray code order along with the element that was inserted or deleted to get the next subset. The output reveals an interesting pattern. For a k-subset, where k is even, the element inserted or deleted is d, the last element. Otherwise, the element operated on occurs immediately before the last element in the subset. For example, the last element in the subset $\{a,b,c\}$ is c, and so b is deleted from it to get the next subset.

```
In[94]:= s = GrayCodeSubsets[{a, b, c, d}];
         Table[{s[[i]], Join[Complement[s[[i]], s[[i + 1]] ],
         Complement[s[[i + 1]], s[[i]] ]]}, {i, 15}] // ColumnForm

Out[95]= {{}, {d}}
         {{d}, {c}}
         {{c, d}, {d}}
         {{c}, {b}}
         {{b, c}, {d}}
         {{b, c, d}, {c}}
         {{b, d}, {d}}
         {{b}, {a}}
         {{a, b}, {d}}
         {{a, b, d}, {c}}
         {{a, b, c, d}, {d}}
         {{a, b, c}, {b}}
         {{a, c}, {d}}
         {{a, c, d}, {c}}
         {{a, d}, {d}}
```

This observation can be proved easily by induction and leads to the following code for `NextGrayCodeSubset`.

```
NextGrayCodeSubset[l_List, s_List] :=
      If[ MemberQ[s,l[[1]]], Rest[s], Prepend[s,l[[1]] ] ] /; EvenQ[Length[s]]

NextGrayCodeSubset[l_List, s_List] :=
      Module[{i = 1},
            While[ ! MemberQ[s, l[[i]] ], i++];
            If[MemberQ[s, l[[i+1]] ], Rest[s], Insert[s, l[[i+1]], 2 ] ]]
```

Computing the Next Subset in Gray Code Order

`NextGrayCodeSubset` provides a compact way of enumerating subsets in reflected Gray code order when used along with the *Mathematica* function `NestList`.

```
In[96]:= NestList[NextGrayCodeSubset[{a, b, c, d}, #] &, {}, 15]

Out[96]= {{}, {a}, {a, b}, {b}, {b, c}, {a, b, c}, {b, c},
    {a, b, c}, {b, c}, {a, b, c}, {b, c}, {a, b, c},
    {b, c}, {a, b, c}, {b, c}, {a, b, c}}
```

■ 2.3.3 Lexicographically Ordered Subsets

We can lexicographically order subsets by viewing them as sequences written in increasing "alphabetical order." Lexicographically ordered subsets can be constructed using a recurrence with a similar flavor to Gray codes. In lexicographic order, all of the subsets containing the first element appear before any of the subsets that do not. Thus we can recursively construct all subsets of the rest of the elements lexicographically and make two copies. The leading copy gets the first element inserted in each subset, while the second one does not.

https://doi.org/10.1017/CBO9781139164849.003 Published online by Cambridge University Press

```
LexicographicSubsets[{}] := {{ }}
LexicographicSubsets[l_List] :=
      Block[{$RecursionLimit = Infinity, s = LexicographicSubsets[Rest[l]]},
            Join[{{}}, Map[Prepend[#, l[[1]]] &, s], Rest[s]]
      ]

LexicographicSubsets[0] := {{ }}
LexicographicSubsets[n_Integer] := LexicographicSubsets[Range[n]]
```

Generating Subsets in Lexicographic Order

In lexicographic order, two subsets are ordered by their smallest elements. Thus the 2^{n-1} subsets containing 1 appear together in the list.

```
In[97]:= LexicographicSubsets[{1,2,3,4}]
Out[97]= {{}, {1}, {1, 2}, {1, 2, 3}, {1, 2, 3, 4},
    {1, 2, 4}, {1, 3}, {1, 3, 4}, {1, 4}, {2}, {2, 3},
    {2, 3, 4}, {2, 4}, {3}, {3, 4}, {4}}
```

The code for `LexicographicSubsets` and `GrayCodeSubsets` is so similar that there is only a slight difference in running time between them.

```
In[98]:= Table[Timing[ LexicographicSubsets[Range[i]];][[1, 1]]/
          Timing[GrayCodeSubsets[Range[i]];][[1, 1]], {i, 10, 15}]
Out[98]= {1.5, 1.25, 1.375, 1.23529, 1.22222, 1.17333}
```

Here we generate 15 lexicographically consecutive subsets of $\{a,b,c,d,e,f\}$ starting with $\{a,b,d,f\}$. This function provides an alternate way of scanning subsets. Suppose we are looking for a superset of a given subset s with a property P. Then it is better to scan subsets in lexicographic order starting from s rather than either of the other orders we have studied.

```
In[99]:= Clear[a,b,c,d,e,f];
          NestList[ NextLexicographicSubset[{a, b, c, d, e, f}, #]&,
          {a, b, d, f}, 15]
Out[100]= {{a, b, d, f}, {a, b, e}, {a, b, e, f}, {a, b, f},
     {a, c}, {a, c, d}, {a, c, d, e}, {a, c, d, e, f},
     {a, c, d, f}, {a, c, e}, {a, c, e, f}, {a, c, f},
     {a, d}, {a, d, e}, {a, d, e, f}, {a, d, f}}
```

Of the three subset generation methods we have seen, generating in Gray code order seems to be the most efficient. So we alias our generic subset generating, ranking, and unranking functions to the corresponding ones for Gray codes.

■ 2.3.4 Generating k-Subsets

A *k-subset* is a subset with exactly k elements in it. The simplest way to construct all k-subsets of a set uses an algorithm to construct all of the subsets and then filters out those of the wrong length. However, since there are 2^n subsets and only $\binom{n}{k}$ k-subsets, this is very wasteful when k is small or large relative to n. A simple recursive construction starts from the observation that each k-subset on n elements either contains the first element of the set or it does not. Prepending the first element to each $(k-1)$-subset of the other $n-1$ elements gives the former, and building all of the k-subsets of the other $n-1$ elements gives the latter. This can also be thought of as a combinatorial proof of the

https://doi.org/10.1017/CBO9781139164849.003 Published online by Cambridge University Press

well-known binomial identity

$$\binom{n}{k} = \binom{n-1}{k-1} + \binom{n-1}{k}.$$

While the running time of this algorithm is proportional to the size of the output, $O(k\binom{n}{k})$, it proves even more efficient to write a function to compute the next k-subset in lexicographical order and then use that repeatedly.

Given a k-subset X of $\{1,2,\ldots,n\}$, the next k-subset of X in lexicographical order can be constructed as follows. Suppose the elements of X are listed in increasing order. Let $i, i+1,\ldots,n$ be the longest suffix of X with consecutive elements. If i is the first element in X, then X is the last k-subset in lexicographic order and its successor is $\{1,2,\ldots,k\}$. Otherwise, the element in X immediately before i is some j, $j < i$. Note that $(j+1)$ is not in X. The lexicographic successor of X is obtained by changing j to $j+1$ and resetting the suffix $i, i+1,\ldots,n$ to $j+2, j+3,\ldots$. This is the algorithm implemented below, taking advantage of *Mathematica*'s `Compile` function.

```
NKS = Compile[{{n, _Integer}, {ss, _Integer, 1}},
            Module[{h = Length[ss], x = n},
                  While[x === ss[[h]], h--; x--];
                  Join[Take[ss, h - 1], Range[ss[[h]]+1, ss[[h]]+Length[ss]-h+1]]
            ]
     ]

NextKSubset[s_List,ss_List] := Take[s,Length[ss]] /; (Take[s,-Length[ss]] === ss)
NextKSubset[s_List,ss_List] :=
       Map[s[[#]] &, NKS[Length[s], Table[Position[s, ss[[i]]][[1, 1]], {i, Length[ss]}]]]
```

Generating the Next Lexicographically Ordered k-Subset

Here are the ten 3-subsets of $\{a,b,c,d,e,f\}$ that immediately follow $\{a,b,e\}$ in lexicographic order.

```
In[101]:= NestList[NextKSubset[{a, b, c, d, e, f}, #] &, {a, b, e}, 10]
Out[101]= {{a, b, e}, {a, b, f}, {a, c, d}, {a, c, e},
      {a, c, f}, {a, d, e}, {a, d, f}, {a, e, f}, {b, c, d},
      {b, c, e}, {b, c, f}}
```

Now enumerating all k-subsets in simply a matter of calling `NextKSubset` repeatedly.

Since the lead element is always positioned first, the k-subsets are generated in lexicographic order.

```
In[102]:= KSubsets[{1,2,3,4,5},3]
Out[102]= {{1, 2, 3}, {1, 2, 4}, {1, 2, 5}, {1, 3, 4},
      {1, 3, 5}, {1, 4, 5}, {2, 3, 4}, {2, 3, 5}, {2, 4, 5},
      {3, 4, 5}}
```

Rewriting `KSubsets` as an iterative function and using compiled code leads to considerable speedup over the corresponding recursive function. The function was implemented recursively in the old package.

```
In[103]:= <<extraCode/RecursiveKSubsets;
          Table[ Timing[ RecursiveKSubsets[Range[20], i];][[1, 1]]/
          Timing[KSubsets[Range[15], i];][[1, 1]], {i, 5, 7}]
Out[104]= {33., 57.0714, 99.5263}
```

https://doi.org/10.1017/CBO9781139164849.003 Published online by Cambridge University Press

To solve the problem of ranking a k-subset in lexicographic order, consider a subset X written in canonical order, that is, $x_1 < x_2 < \cdots < x_k$. The subsets that appear before X in lexicographic order are of two kinds: (i) those that contain an element smaller than x_1 and (ii) those whose smallest element is x_1, but the remaining elements form a set that is lexicographically smaller than $\{x_2, x_3, \ldots, x_k\}$. To count the number of subsets of the first kind, we note that the number of k-subsets of $\{1, 2, \ldots, n\}$ whose smallest element is i is $\binom{n-i}{k-1}$. So the number of subsets of the first kind is $\sum_{i=1}^{x_1-1} \binom{n-i}{k-1}$. The number of subsets of the second kind is calculated by observing that this is simply a smaller ranking problem in which we want to find the rank of $\{x_2, \ldots, x_k\}$ in the lexicographically ordered list of subsets of $\{x_1 + 1, x_1 + 2, \ldots, n\}$. Below we give an implementation of this recurrence.

```
RankKSubset[ss_List, s_List] := 0 /; (Length[ss] === Length[s])
RankKSubset[ss_List, s_List] := Position[s, ss[[1]]][[1, 1]] - 1 /; (Length[ss] === 1)
RankKSubset[ss_List, s_List] :=
      Block[{n = Length[s], k = Length[ss],
            x = Position[s, ss[[1]]][[1, 1]], $RecursionLimit = Infinity},
            Binomial[n, k] - Binomial[n-x+1, k] + RankKSubset[Rest[ss], Drop[s, x]]
      ]
```

Ranking a k-Subset

The 20 3-subsets of $\{a, b, c, d, e, f\}$ are listed in lexicographic order and ranked. Naturally, the output is the integers 0 through 19, in that order.

```
In[105]:= Map[RankKSubset[#, {a, b, c, d, e, f}] &,
            KSubsets[{a, b, c, d, e, f}, 3] ]
Out[105]= {0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14,
      15, 16, 17, 18, 19}
```

We also use these ideas to unrank a k-subset. Suppose a subset $X = \{x_1, x_2, \ldots, x_k\} \subseteq \{1, 2, \ldots, n\}$, $x_1 < x_2 < \cdots < x_n$, has rank m. Then the total number of k-subsets of $\{1, 2, \ldots, n\}$ that contain an element smaller than x_1 is at most m. This translates into the inequality

$$\sum_{i=1}^{x_1-1} \binom{n-i}{k-1} = \binom{n}{k} - \binom{n-x_1+1}{k} \leq m.$$

Also, the total number of k-subsets of $\{1, 2, \ldots, n\}$ that contain an element smaller than *or equal* to x_1 is greater than m. This translates into the inequality

$$\sum_{i=1}^{x_1} \binom{n-i}{k-1} = \binom{n}{k} - \binom{n-x_1}{k} > m.$$

These inequalities give us a way of finding x_1, leading to the code below for `UnrankKSubset`.

```
UnrankKSubset[m_Integer, 1, s_List] := {s[[m + 1]]}
UnrankKSubset[0, k_Integer, s_List] := Take[s, k]
UnrankKSubset[m_Integer, k_Integer, s_List] :=
      Block[{i = 1, n = Length[s], x1, u, $RecursionLimit = Infinity},
```

https://doi.org/10.1017/CBO9781139164849.003 Published online by Cambridge University Press

```
        u = Binomial[n, k]; While[Binomial[i, k] < u - m, i++]; x1 = n - (i - 1);
        Prepend[UnrankKSubset[m-u+Binomial[n-x1+1, k], k-1, Drop[s, x1]], s[[x1]]]
    ]
```

Unranking a k-Subset

The numbers in the sequence 0 through 19 are unranked to give 3-subsets of $\{a,b,c,d,e,f\}$. As expected, we get all 3-subsets in lexicographic order.

```
In[106]:= Map[ UnrankKSubset[#, 3, {a, b, c, d, e, f}] &, Range[0, 19]]
Out[106]= {{a, b, c}, {a, b, d}, {a, b, e}, {a, b, f},
     {a, c, d}, {a, c, e}, {a, c, f}, {a, d, e}, {a, d, f},
     {a, e, f}, {b, c, d}, {b, c, e}, {b, c, f}, {b, d, e},
     {b, d, f}, {b, e, f}, {c, d, e}, {c, d, f}, {c, e, f},
     {d, e, f}}
```

Here is the 100,000th 10-subset in the lexicographic list of 10-subsets of the set 1 through 20.

```
In[107]:= UnrankKSubset[99999, 10, Range[20]]
Out[107]= {2, 3, 4, 6, 10, 11, 13, 15, 17, 18}
```

And here we compute its rank and see that `UnrankKSubset` and `RankKSubset` act as inverses, as they should.

```
In[108]:= RankKSubset[%, Range[20]]
Out[108]= 99999
```

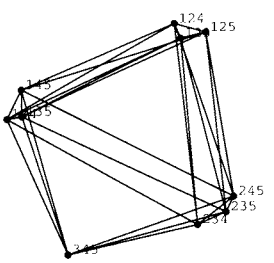
Can k-subsets be ordered in a Gray code order, yielding minimum change from one k-subset to the next? It takes at least two insertion-deletion operations to transform one k-subset into another. So a Gray code order should obtain each k-subset from the previous one with an insertion plus a deletion. We gain confidence that such an order exists from the following examples.

Here is the relation connecting pairs of subsets that are two operations (one deletion and one insertion) apart.

```
In[109]:= sr = ((Length[Complement[#1, #2]] === 1) &&
              (Length[Complement[#2, #1]] === 1))&;
```

This shows a graph whose vertices are 3-subsets of $\{1,2,3,4,5\}$, and each edge in the graph connects a pair of subsets that are two operations apart. The `PlotRange` option has been used here to ensure that the labels are not cut off in the picture. `SpringEmbedding` does a good job revealing the structure in this graph.

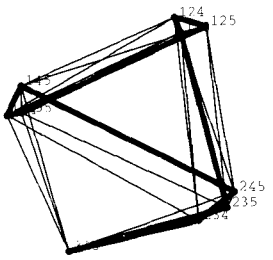
```
In[110]:= ShowGraph[g = SpringEmbedding[ MakeGraph[KSubsets[Range[5],
          3], sr, Type->Undirected, VertexLabel->True]], PlotRange->0.1]
```



https://doi.org/10.1017/CBO9781139164849.003 Published online by Cambridge University Press

A Hamiltonian cycle in the graph tells us that 3-subsets of $\{1,2,3,4,5\}$ can be listed in a Gray code order. The structure of the graph revealed by SpringEmbedding naturally suggests other Hamiltonian cycles.

```
In[111]:= ShowGraph[Highlight[g, {Partition[HamiltonianCycle[g],2,1]},
            HighlightedEdgeColors -> {Blue}], PlotRange -> 0.1]
```



Indeed, the graph contains many different Hamiltonian cycles/Gray codes.

```
In[112]:= Length[HamiltonianCycle[g, All]]
Out[112]= 6432
```

Wilf [Wil89] suggests the following algorithm for generating k-subsets in Gray code order. Let $A(n,k)$ denote the list of k-subsets of $[n]$ in Gray code order, with the first set in the list being $\{1,2,\ldots,n\}$ and the last one being $\{1,2,\ldots,k-1,n\}$. $A(n,k)$ can be constructed recursively by concatenating $A(n-1,k)$ to the list obtained by reversing $A(n-1,k-1)$ and appending n to each subset in $A(n-1,k-1)$. The correctness of this recurrence can be proved easily by induction. GrayCodeKSubsets implements this algorithm.

```
GrayCodeKSubsets[n_Integer?Positive, k_Integer] := GrayCodeKSubsets[Range[n], k]

GrayCodeKSubsets[l_List, 0] := {{}}
GrayCodeKSubsets[l_List, 1] := Partition[l, 1]
GrayCodeKSubsets[l_List, k_Integer?Positive] := {l} /; (k == Length[l])
GrayCodeKSubsets[l_List, k_Integer?Positive] := {} /; (k > Length[l])

GrayCodeKSubsets[l_List, k_Integer] :=
    Block[{$RecursionLimit = Infinity},
        Join[GrayCodeKSubsets[Drop[l, -1], k],
            Map[Append[#, Last[l]]&,
                Reverse[GrayCodeKSubsets[Drop[l,-1], k-1]]
            ]
        ]
    ]
```

Generating k-Subsets in Gray Code Order

https://doi.org/10.1017/CBO9781139164849.003 Published online by Cambridge University Press

Here are 4-subsets of $\{a,b,c,d,e,f\}$ listed in Gray code order. Each subset is obtained from its neighbors by an insertion and a deletion.

```
In[113]:= GrayCodeKSubsets[{a, b, c, d, e, f}, 4]
Out[113]= {{a, b, c, d}, {a, b, d, e}, {b, c, d, e},
     {a, c, d, e}, {a, b, c, e}, {a, b, e, f}, {b, c, e, f},
     {a, c, e, f}, {c, d, e, f}, {b, d, e, f}, {a, d, e, f},
     {a, b, d, f}, {b, c, d, f}, {a, c, d, f}, {a, b, c, f}}
```

To construct a random k-subset, it is sufficient to select the first k elements in a random permutation of the set and then sort them to restore the canonical order. An alternate algorithm is obtained by using `UnrankKSubset`: Generate a random integer between 0 and $\binom{n}{k} - 1$ and unrank it to get a random k-subset of $\{1, 2, \ldots, n\}$. However, `UnrankKSubset` is quite slow, so this is not competitive with the random permutation algorithm.

We can get a random 50,000-subset of a set of 100,000 elements in far less time than it takes to unrank a 250-element subset of a set of 500 elements! Thus it is wasteful to use `UnrankKSubset` to pick a random k-subset.

```
In[114]:= {Timing[UnrankKSubset[Random[Integer, Binomial[500, 250]-1],
            250, Range[500]];],
           Timing[RandomKSubset[100000, 50000];]}
Out[114]= {{11.62 Second, Null}, {1.55 Second, Null}}
```

A more subtle, but no more efficient algorithm appears in [NW78].

```
RandomKSubset[n_Integer,k_Integer] := RandomKSubset[Range[n],k]
RandomKSubset[s_List, k_Integer] := s[[Sort[RandomPermutation[Length[s]]][[Range[k] ]]]]
```

Generating Random k-Subsets

Here we generate random subsets of increasing size.

```
In[115]:= TableForm[ Table[ RandomKSubset[10,i], {i,10}] ]
Out[115]//TableForm= 5
                     1   7
                     4   5   7
                     1   4   5   8
                     1   3   8   9   10
                     1   2   4   7   8   10
                     1   2   3   5   6   7   8
                     1   2   3   4   6   7   9   10
                     1   2   3   4   5   6   8   9   10
                     1   2   3   4   5   6   7   8   9   10
```

The distribution of 3-subsets appears to be uniform.

```
In[116]:= Distribution[ Table[RandomKSubset[5,3], {200}] ]
Out[116]= {15, 25, 30, 20, 19, 21, 14, 22, 16, 18}
```

https://doi.org/10.1017/CBO9781139164849.003 Published online by Cambridge University Press

2.3.5 Strings

There is a another simple combinatorial object that will prove useful in several different contexts. Indeed, we have already seen it used in constructing subsets by binary representation. A *string* of length k on an alphabet M is an arrangement of k not necessarily distinct symbols from M. Such strings should not be confused with the character string, which is a data type for text in *Mathematica*. The m^k distinct strings for an alphabet of size m can be constructed using a simple recursive algorithm that generates all length-$(k-1)$ strings on M and then prepends each of the m symbols to each string.

```
Strings[l_List,0] := { {} }

Strings[l_List, k_Integer] := Strings[Union[l], k] /; (Length[l] =!= Length[Union[l]])
Strings[l_List,k_Integer] := Distribute[Table[l, {k}], List]
```

Constructing all Strings on an Alphabet

All length-2 strings from the alphabet $\{P, Q, R\}$ are constructed. This recursive construction prepends each symbol successively to all strings of length $(k-1)$, thus yielding the strings in lexicographic order.

```
In[117]:= Strings[{P, Q, R}, 2]
Out[117]= {{P, P}, {P, Q}, {P, R}, {Q, P}, {Q, Q}, {Q, R},
     {R, P}, {R, Q}, {R, R}}
```

https://doi.org/10.1017/CBO9781139164849.003 Published online by Cambridge University Press

2.4 Exercises

2.4.1 Thought Exercises

1. Prove that the expected value of the first element of a random n-permutation is $(n+1)/2$.
2. Prove there exists an n-permutation p with exactly k inversions for every integer k, where $0 \leq k \leq \binom{n}{2}$.
3. Prove that the number of runs in an n-permutation plus the number of runs in its reverse equals $(n+1)$.
4. Show that the average number of descents (one less than the number of runs) in a size-n permutation is $(n-1)/2$.
5. Use `NumberOfPermutationsByInversions` to generate a table of $I_n(k)$ values for integers n and k, $1 \leq n \leq 10$ and $1 \leq k \leq 10$. Examine this table to determine a simple recurrence relation for $I_n(k)$, $k < n$. Prove the correctness of the recurrence.
6. How many permutations on n elements have inversion vectors that consist of $n-1$ distinct integers? Can you characterize the permutations with this property?
7. Let O_n denote the number of insertion-deletion operations performed by `BinarySubsets` in generating all 2^n subsets of an n-element set. Prove that for all positive integers n, $O_n = 2O_{n-1} + n$. Solve this recurrence.
8. Show that the number of n-bit strings with exactly k 0's and with no two consecutive 0's is $\binom{n-k+1}{k}$.
9. An undirected graph is *transitively orientable* if each of its edges can be directed so that (a,c) is a (directed) edge whenever (a,b) and (b,c) are, for any vertex b.
   (a) Show that if G is a permutation graph, then both G and its complement are transitively orientable. The *complement* of a graph $G = (V,E)$ is a graph $G^c = (V,E^c)$ in which each pair of distinct vertices a, b is connected by an edge if $\{a,b\}$ is not an edge in G.
   (b) Show that the converse of the above is also true. Based on this proof, derive an algorithm that determines if a given undirected graph G is a permutation graph. If G is a permutation graph, then your algorithm should return a permutation π for which the graph is a permutation graph.
10. Let n be a positive integer and let m be an integer such that $0 \leq m < 2^n$. Suppose that m has binary representation $b_{n-1}b_{n-2}\ldots b_1b_0$, that is, $m = \sum_{i=0}^{n-1} b_i 2^i$. If $c_{n-1}c_{n-2}\cdots c_1c_0$ is the binary representation of the subset with rank m in the reflected Gray code, then show that $c_i = (b_i + b_{i+1}) \pmod 2$ for all $i, 0 \leq i \leq n-1$. Here we suppose that $b_n = 0$.

https://doi.org/10.1017/CBO9781139164849.003 Published online by Cambridge University Press

2.4.2 Programming Exercises

1. As implemented, `RankPermutation` takes $\Theta(n^2)$ time. An alternate algorithm for `RankPermutation` uses inversion vectors.
   (a) Show how to compute the rank of an n-permutation in $\Theta(n)$ time, given its inversion vector.
   (b) Implement a `NewRankPermutation` function using the algorithm implied in (a) and compare its running time with the running time of the current implementation.
2. Heap's algorithm defines a certain total order on permutations. Study this carefully and devise and implement ranking and unranking algorithms for this total order.
3. The recurrence for the Eulerian numbers $\left\langle {n \atop k} \right\rangle$ provides a way of generating permutations with k runs. Write functions to rank, unrank, and generate random size-n permutations with exactly k runs.
4. Write functions `RankString`, `UnrankString`, `RandomString`, and `NextString`.
5. Experiment with the function `GrayCodeKSubsets` to find a pattern in the insertions and deletions in going from one k-subset to the next. Use this observation to write an iterative version of the function. Compile this code and compare its running time with the current recursive function.
6. Write rank and unrank functions for k-subsets listed in Gray code order.
7. Enhance `LexicographicPermutations` so that it deals with multiset permutations correctly. Specifically, given a permutation of a multiset, devise an algorithm to generate the next multiset in lexicographic order. Use this to enhance `NextPermutation` so that it correctly deals with multisets and use the new `NextPermutation` to obtain a new `LexicographicPermutations`.
8. Enhance `RankPermutation`, `UnrankPermutation`, and `RandomPermutation` to deal with multisets correctly.

2.4.3 Experimental Exercises

1. Change `RandomPermutation` so that the random index to swap is drawn from $[1, n]$ intead of $[1, i]$. Are these permutations still random? If not, which permutations are selected most/least often?
2. How much longer, on average, is the first run of a random permutation than the second? What does the expected length of the ith run converge to for large i?
3. Use the *Combinatorica* function `Strings` to generate all n-bit strings and from these select those that do not have two consecutive 0's. Count the number of such strings for various n. Is this number familiar to you?

https://doi.org/10.1017/CBO9781139164849.003 Published online by Cambridge University Press