■ Johnson−TrotterPermutations

The following loads the Combinatorica package. The Combinatorica package is in a single file called "New-Combinatorica.m" which is in the working directory.

```
<< NewCombinatorica.m
```

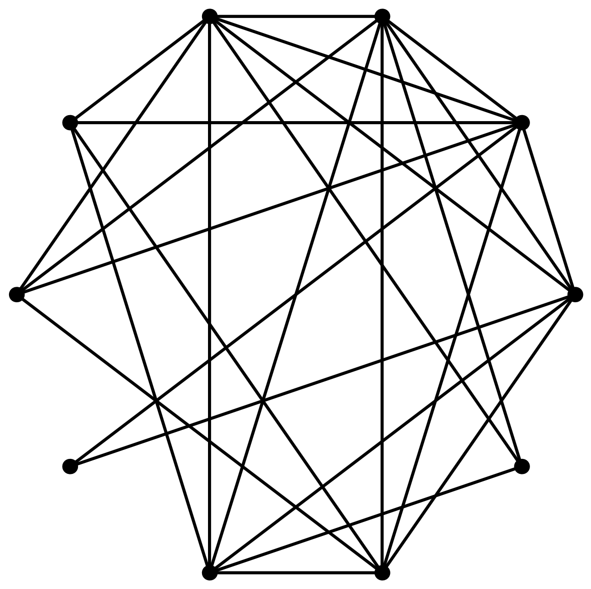
In the next example, we generate the set of 4−permutationsin lexicographic order by calling the Mathematica function Permutations.

```
Permutations[4]
```

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

Graphics and text can both be part of a notebook as the following example shows. I use Combinatorica functions Random-Graph and ShowGraph to construct and display a 10−vertexrandom graph.

```
ShowGraph[RandomGraph[10, .5]]
```



```
- Graphics -
```

■ Attempt 1: A recursive function

Let us now start writing code to generate n−permutationsin Johnson−Trotterorder. The idea is that we first generate all (n−1)-permutations in Johnson−Trotterorder and store this in a variable, say JTL. We then insert n into each permutation in JTL in each of the possible n slots. Specifically, n starts as the last element in JTL[[1]] and moves to the left, one swap at a time. When we reach the beginning of JTL[1], we start with n as the first element in JTL[[2]] and move n to the right one swap at a time and so on until n moves across JTL[[(n−1)!]].

```
JTPermutations[1] := { {1} }

JTPermutations[ n_Integer ] :=
        Module[{JTL = JTPermutations[n-1]},
                Flatten[
                    Table[If[EvenQ[i],
                             Table[Insert[JTL[[i]], n, j], {j, n}],
                             Table[Insert[JTL[[i]], n, j], {j, n, 1, -1}]
                          ], {i, (n-1)!}
                    ] , 1
                ]
        ]
```

Here are the things to note about the above code.

1. The assignment operator ":=" stands for delayed evaluation and this is what should be used for function definition. The help message for ":=" clearly explains this matter.

```
?:=
```

```
lhs := rhs assigns rhs to be the delayed value of lhs. rhs is maintained in an
  unevaluated form. When lhs appears, it is replaced by rhs, evaluated afresh each time.
```

2. JTPermutations is the name of the function and it takes one integer argument. One way to think about "n_Integer" is that it stands for the formal parameter n. The "_" is a pattern object to which anything can match. The "_Integer" stands for a pattern object to which any integer can match. We can prevent matches with anything that is not positive, by using "_Integer?Positive". The help message for "_" explains this a little bit. Look up "Pattern" in the *Mathematica* help browser for more information. Pattern matching is an extremely important technique in *Mathematica* programming.

```
?_
```

```
_ or Blank[ ] is a pattern object that can stand for any Mathematica
  expression. _h or Blank[h] can stand for any expression with head h.
```

3. Module[{ local variables}, body of function] is the structure that most non−trivialprograms have. The usage message for Module is given below. There is a *Mathematica* construct called "Block" that is used sometimes as an alternative to Module. The difference between Module and Block is not important at this time and you can ignore Block for now.

4. The *Mathematica* function Table is used in the code as a looping construct. *Mathematica* has many many looping constructs. For people who want procedural programming there is While, For, and Do. For functional programmers, there is Map, Apply, Table etc. The usage message for Table, given below, is quite clear. Note that Table generates a list and since we have nested calls to Table in our code, we get lists of lists of permutations. The resulting nested lists are flattened out by a call to the Flatten function. Note that the second argument to Flatten is 1, indicating that only the outermost parentheses need to be flattened. Look at the example below.

```
?Table
```

```
Table[expr, {imax}] generates a list of imax copies of expr. Table[expr, {i, imax}] generates a
  list of the values of expr when i runs from 1 to imax. Table[expr, {i, imin, imax}] starts
  with i = imin. Table[expr, {i, imin, imax, di}] uses steps di. Table[expr, {i, imin,
  imax}, {j, jmin, jmax}, ... ] gives a nested list. The list associated with i is outermost.
```

```
Flatten[{ {1, 2, {3, 4}, {5, {{6}}}}}]
```

```
{1, 2, 3, 4, 5, 6}
```

```
Flatten[{ {1, 2, {3, 4}, {5, {{6}}}}}, 1]
```

```
{1, 2, {3, 4}, {5, {{6}}}}
```

5. The If function in Mathematica has 2 or 3 arguments. The 2 argument version corresponds to the if−thenand the 3 argument version corresponds to the if−then−elseIn the example below, the boolean expression "i < 10" evaluates to false and so the second argument is skipped and the third argument is executed. There is also a "Switch" function in *Mathematica* that corresponds to the Switch statement in C.

```
i = 12
```

```
12
```

```
If[i < 10, i + 1, i + 2]
```

```
14
```

6. The Insert function is an example of the many list manipulation functions *Mathematica* provides. Some of the other functions are Delete, Take, Drop, Append, Prepend, Join, etc. Look under "List Construction" in the help browser to see the complete list.

```
? Insert
```

```
Insert[list, elem, n] inserts elem at position n in list. If n
  is negative, the position is counted from the end. Insert[expr, elem, {i,
  j, ... }] inserts elem at position {i, j, ... } in expr. Insert[expr, elem,
  {{i1, j1, ... }, {i2, j2, ... }, ... }] inserts elem at several positions.
```

```
? Module
```

```
Module[{x, y, ... }, expr] specifies that occurrences of the symbols x, y, ...  in expr should
  be treated as local. Module[{x = x0, ... }, expr] defines initial values for x, ... .
```

```
JTPermutations[1] := { {1} }
```

```
JTPermutations[ n_Integer ] := Module[ {l}, l = JTPermutations[n - 1];
  Flatten[Table[ If[EvenQ[i],  Table[Insert[l[[i]], n, j], {j, n}],
     Table[Insert[l[[i]], n, j], {j, n, 1, -1}]], {i, (n - 1) !}] , 1]]
```

```
JTPermutations[1]
```

```
{{1}}
```

```
JTPermutations[2]
```

```
{{1, 2}, {2, 1}}
```

```
JTPermutations[3]
```

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

```
JTPermutations[4]
```

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

■ Attempt 2: An iterative function

Here is second attempt at generating permutations in Johnson−Trotterorder. We avoid recursion this time and the resulting code is not as elegant or compact, but it illustrates a few *Mathematica* functions that programmers in procedural languages will prefer.

The main idea is that we start with the identity permutation and succesively generate subsequent permutations one at a time. The loop that does this is the Table[..] function, whose body is executed (n!−1)times. To help us in finding the successor of a permutation, we keep a "direction" vector d whose entries can be −1or +1. d[[i]] is +1 if and only if we want to move i to the right. d is initialized to all −1'sbecause we want to move all elements to the left, initially. We want to find the largest element in [n] that can be moved. The While[..] function is responsible for this task. A variable i is initialized to n and is decremented repeatedly until we get to a value of i that can be moved in the direction desired by d[[i]]. After the While loop, we have found an i that can be moved and we use the If[..] function to examine d[[i]] and determine if the element s to be moved left or to be moved right.

```
NewJTPermutations[n_] :=
        Module[{p  = Range[n], d = Table[-1, {n}], i , pos},
               Prepend[
                  Table[(*Produce the next permutation from p and assign it to p*)
                        i = n;
                        While[(*i is not movable*)
                              pos = Flatten[Position[p, i]][[1]];
                              (((pos == 1) || (p[[pos-1]] > p[[pos]])) && d[[i]] == -1) ||
                              (((pos == n) || (p[[pos+1]] > p[[pos]])) && d[[i]] == +1),
                              d[[i]] = -1*d[[i]]; i--
                        ];
                        If[d[[i]] == 1,
                           {p[[pos]], p[[pos+1]]} = {p[[pos+1]], p[[pos]]},
                           {p[[pos]], p[[pos-1]]} = {p[[pos-1]], p[[pos]]}
                        ];
                        p,
                        {n!-1}
                  ] ,
                  Range[n]
               ]
        ]
```

Here are some notes on the code.

1. Range[n] generates the set {1, 2, ..., n}. Here are some examples of how Range[..] can be used.

```
Range[10, 12]

{10, 11, 12}

Range[-5, +5]

{-5, -4, -3, -2, -1, 0, 1, 2, 3, 4, 5}

Range[10]

{1, 2, 3, 4, 5, 6, 7, 8, 9, 10}

? Range

Range[imax] generates the list {1, 2, ... , imax}. Range[imin, imax]
  generates the list {imin, ... , imax}. Range[imin, imax, di] uses step di.
```

2. Prepend[..] is a function that prepends an element to a list. There are other related list manipulation functions such as Append, Join, Insert, Delete etc. that you should be familiar with. In the above function, the call to Table[..] generates (n!−1) permutations and the Prepend[..] function is used to prepend the identity permutation to the list.

```
?Prepend
```

```
Prepend[expr, elem] gives expr with elem prepended.
```

```
Prepend[{a, b, c}, 10]
```

```
{10, a, b, c}
```

3. The While[..] function has the syntax While[boolean expression, body]. While[..] is one of those uncommon *Mathematica* functions that returns nothing. It has side effects, but it returns nothing. Here is an example. Note that there is no output corresponding to the following code.

```
i = 10; j = 1; While[i > 0, j++; i--]
```

```
j
```

```
11
```

```
i
```

```
0
```

Here is another example of a While[..] function. Here I want to show how the boolean expression that is the first argument of the While[..] function can be at the end of a sequence of statements separated by commas. The value of the sequence is the value that the last statement (the boolean expression, in this case) evaluates to.

```
i = 10; j = 1; While[j++; i > 0, i--]
```

```
i
```

```
0
```

```
j
```

```
12
```

4. The Position[..] function is a very useful "search" function that can search for patterns in expressions. So it can do much more than search for individual elements in lists. Note the "extra" parentheses in what Position[..] returns.

```
?Position
```

```
Position[expr, pattern] gives a list of the positions at which
  objects matching pattern appear in expr. Position[expr, pattern, levspec]
  finds only objects that appear on levels specified by levspec. Position[
  expr, pattern, levspec, n] gives the positions of the first n objects found.
```

```
Position[a^2 + 5 a b c + 11 a^3, a]
```

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

```
Position[{{10, 1, 3}, {1, 11, 10}}, 10]
```

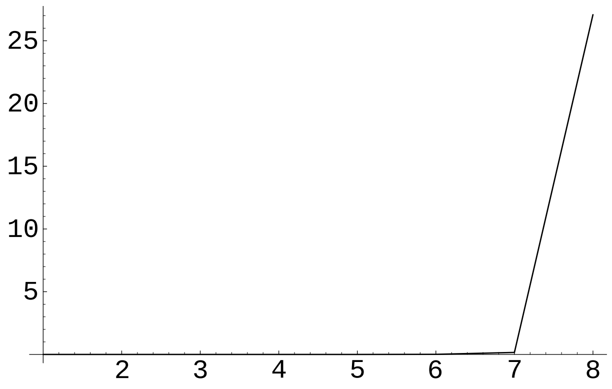
```
{{1, 1}, {2, 3}}
```

```
Position[{1, 2, 3, 4}, 3]

{{3}}

NewJTPermutations[n_] := Module[{p = Range[n], d = Table[-1, {n}], i, pos},
  Prepend[Table[(*Produce the next permutation from p and assign it to p*)i = n;
    While[(*i is not movable*)pos = Flatten[Position[p, i]][[1]];
     (((pos == 1) || (p[[pos - 1]] > p[[pos]])) && d[[i]] == -1) ||
      (((pos == n) || (p[[pos + 1]] > p[[pos]])) && d[[i]] == +1), d[[i]] = -1 * d[[i]]; i--];
    If[d[[i]] == 1, {p[[pos]], p[[pos + 1]]} = {p[[pos + 1]], p[[pos]]},
     {p[[pos]], p[[pos - 1]]} = {p[[pos - 1]], p[[pos]]}]; p, {n ! - 1}], Range[n]]]

NewJTPermutations[4]

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

JTPermutations[4]

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

NewJTPermutations[5] == JTPermutations[5]

True
```

Timing functions in *Mathematica* and plotting timings of functions is very easy. There is a function called Timing[..] that helps in this.

```
Timing[ JTPermutations[6]; ]

{0.01 Second, Null}

Table[ Timing[JTPermutations[i];], {i, 8}]

{{0. Second, Null}, {0. Second, Null}, {0. Second, Null}, {0. Second, Null},
 {0. Second, Null}, {0.01 Second, Null}, {0.16 Second, Null}, {27.08 Second, Null}}

l = %

{{0. Second, Null}, {0. Second, Null}, {0. Second, Null}, {0. Second, Null},
 {0. Second, Null}, {0.01 Second, Null}, {0.16 Second, Null}, {27.08 Second, Null}}
```

```
ListPlot[Map[#[[1, 1]] &, l], PlotJoined → True]
```



```
- Graphics -
```