Permutations

Task

Write a program that generates all permutations of **n** different objects. (Practically numerals!)

Permutations
You are encouraged to solve this task according to the task description, using any language you may know.

Related tasks

- Find the missing permutation
- Permutations/Derangements

The number of samples of size k from n objects.

With combinations and permutations generation tasks.

	Order Unimportant	Order Important
Without replacement	$\binom{n}{k} = {}^{n}\mathrm{C}_k = \frac{n(n-1)\dots(n-k+1)}{k(k-1)\dots 1}$	${}^{n}\mathrm{P}_k = n \cdot (n-1) \cdot (n-2) \cdots (n-k+1)$
	Task: Combinations	Task: Permutations
With replacement	$\binom{n+k-1}{k} = {}^{n+k-1}\mathrm{C}_k = \frac{(n+k-1)!}{(n-1)!k!}$	n^k
	Task: Combinations with repetitions	Task: Permutations with repetitions

11l

```
V a = [1, 2, 3]
L
   print(a)
   I !a.next_permutation()
      L.break
```

Output:

```
[1, 2, 3]
[1, 3, 2]
[2, 1, 3]
[2, 3, 1]
```

```
[3, 1, 2]
[3, 2, 1]
```

360 Assembly

Translation of: Liberty BASIC

```
*        Permutations              26/10/2015
PERMUTE  CSECT
         USING  PERMUTE,R15        set base register
         LA     R9,TMP-A           n=hbound(a)
         SR     R10,R10            nn=0
LOOP     LA     R10,1(R10)         nn=nn+1
         LA     R11,PG             pgi=@pg
         LA     R6,1               i=1
LOOPI1   CR     R6,R9              do i=1 to n
         BH     ELOOPI1
         LA     R2,A-1(R6)         @a(i)
         MVC    0(1,R11),0(R2)     output a(i)
         LA     R11,1(R11)         pgi=pgi+1
         LA     R6,1(R6)           i=i+1
         B      LOOPI1
ELOOPI1  XPRNT  PG,80
         LR     R6,R9              i=n
LOOPUIM  BCTR   R6,0               i=i-1
         LTR    R6,R6              until i=0
         BE     ELOOPUIM
         LA     R2,A-1(R6)         @a(i)
         LA     R3,A(R6)           @a(i+1)
         CLC    0(1,R2),0(R3)      or until a(i)<a(i+1)
         BNL    LOOPUIM
ELOOPUIM LR     R7,R6              j=i
         LA     R7,1(R7)           j=i+1
         LR     R8,R9              k=n
LOOPWJ   CR     R7,R8              do while j<k
         BNL    ELOOPWJ
         LA     R2,A-1(R7)         r2=@a(j)
         LA     R3,A-1(R8)         r3=@a(k)
         MVC    TMP,0(R2)          tmp=a(j)
         MVC    0(1,R2),0(R3)      a(j)=a(k)
         MVC    0(1,R3),TMP        a(k)=tmp
         LA     R7,1(R7)           j=j+1
         BCTR   R8,0               k=k-1
         B      LOOPWJ
ELOOPWJ  LTR    R6,R6              if i>0
         BNP    ILE0
         LR     R7,R6              j=i
         LA     R7,1(R7)           j=i+1
LOOPWA   LA     R2,A-1(R7)         @a(j)
         LA     R3,A-1(R6)         @a(i)
         CLC    0(1,R2),0(R3)      do while a(j)<a(i)
         BNL    AJGEAI
         LA     R7,1(R7)           j=j+1
         B      LOOPWA
AJGEAI   LA     R2,A-1(R7)         r2=@a(j)
         LA     R3,A-1(R6)         r3=@a(i)
         MVC    TMP,0(R2)          tmp=a(j)
         MVC    0(1,R2),0(R3)      a(j)=a(i)
         MVC    0(1,R3),TMP        a(i)=tmp
ILE0     LTR    R6,R6              until i<>0
         BNE    LOOP
         XR     R15,R15            set return code
         BR     R14                return to caller
A        DC     C'ABCD'            <== input
TMP      DS     C                  temp for swap
PG       DC     CL80' '            buffer
         YREGS
         END    PERMUTE
```

Output:

```
ABCD
ABDC
ACBD
ACDB
ADBC
ADCB
BACD
BADC
BCAD
BCDA
BDAC
BDCA
CABD
CADB
CBAD
```

AArch64 Assembly

Works with: as version Raspberry Pi 3B version Buster 64 bits

```
/* ARM assembly AARCH64 Raspberry PI 3B */
/*  program permutation64.s  */

/*********************************************/
/* Constantes file                           */
/*********************************************/
/* for this file see task include a file in language AArch64 assembly */
.include "../includeConstantesARM64.inc"

/*********************************/
/* Initialized data              */
/*********************************/
.data

sMessResult:        .asciz "Value  : @\n"
sMessCounter:       .asciz "Permutations =  @ \n"
szCarriageReturn:   .asciz "\n"

.align 4
TableNumber:       .quad   1,2,3
                   .equ NBELEMENTS, (. - TableNumber) / 8
/*********************************/
/* UnInitialized data            */
/*********************************/
.bss
sZoneConv:            .skip 24
/*********************************/
/*  code section                 */
/*********************************/
.text
.global main
main:                               //entry of program
    ldr x0,qAdrTableNumber          //address number table
    mov x1,NBELEMENTS               //number of élements
    mov x10,0                       //counter
    bl heapIteratif
    mov x0,x10                      //display counter
    ldr x1,qAdrsZoneConv            //
    bl conversion10S                //décimal conversion
    ldr x0,qAdrsMessCounter
    ldr x1,qAdrsZoneConv            //insert conversion
    bl strInsertAtCharInc
    bl affichageMess                //display message

100:                                //standard end of the program
    mov x0,0                        //return code
    mov x8,EXIT                     //request to exit program
    svc 0                           //perform the system call

qAdrszCarriageReturn:     .quad szCarriageReturn
```

```
/*****************************************************************/
/*     permutation by heap iteratif (wikipedia)                  */
/*****************************************************************/
/* x0 contains the address of table */
/* x1 contains the eléments number  */
heapIteratif:
    stp x2,lr,[sp,-16]!             // save  registers
    stp x3,x4,[sp,-16]!             // save  registers
    stp x5,x6,[sp,-16]!             // save  registers
    stp x7,fp,[sp,-16]!             // save  registers
    tst x1,1                        // odd ?
    add x2,x1,1
    csel x2,x2,x1,ne                // the stack must be a multiple of 16
    lsl x7,x2,3                     // 8 bytes by count
    sub sp,sp,x7
    mov fp,sp
    mov x3,#0
    mov x4,#0                       // index
1:                                  // init area counter
    str x4,[fp,x3,lsl 3]
    add x3,x3,#1
    cmp x3,x1
    blt 1b

    bl displayTable
    add x10,x10,#1
    mov x3,#0                       // index
2:
    ldr x4,[fp,x3,lsl 3]            // load count [i]
    cmp x4,x3                       // compare with i
    bge 5f
    tst x3,#1                       // even ?
    bne 3f
    ldr x5,[x0]                     // yes load value A[0]
    ldr x6,[x0,x3,lsl 3]            // and swap with value A[i]
    str x6,[x0]
    str x5,[x0,x3,lsl 3]
    b 4f
3:
    ldr x5,[x0,x4,lsl 3]           // load value A[count[i]]
    ldr x6,[x0,x3,lsl 3]           // and swap with value A[i]
    str x6,[x0,x4,lsl 3]
    str x5,[x0,x3,lsl 3]
4:
    bl displayTable
    add x10,x10,1
    add x4,x4,1                    // increment count i
    str x4,[fp,x3,lsl 3]           // and store on stack
    mov x3,0                       // raz index
    b 2b                           // and loop
5:
    mov x4,0                       // raz count [i]
    str x4,[fp,x3,lsl 3]
    add x3,x3,1                    // increment index
    cmp x3,x1                      // end ?
    blt 2b                         // no -> loop

    add sp,sp,x7                   // stack alignement
100:
    ldp x7,fp,[sp],16              // restaur  2 registers
    ldp x5,x6,[sp],16              // restaur  2 registers
    ldp x3,x4,[sp],16              // restaur  2 registers
    ldp x2,lr,[sp],16              // restaur  2 registers
    ret                            // return to address lr x30
/*****************************************************************/
/*      Display table elements                                   */
/*****************************************************************/
/* x0 contains the address of table */
displayTable:
    stp x1,lr,[sp,-16]!              // save  registers
    stp x2,x3,[sp,-16]!              // save  registers
    mov x2,x0                        // table address
    mov x3,#0
1:                                   // loop display table
    ldr x0,[x2,x3,lsl 3]
```

```
    ldr x1,qAdrsZoneConv             // insert conversion
    bl strInsertAtCharInc
    bl affichageMess                 // display message
    add x3,x3,1
    cmp x3,NBELEMENTS - 1
    ble 1b
    ldr x0,qAdrszCarriageReturn
    bl affichageMess
    mov x0,x2
100:
    ldp x2,x3,[sp],16                // restaur  2 registers
    ldp x1,lr,[sp],16                // restaur  2 registers
    ret                              // return to address lr x30
qAdrsZoneConv:           .quad sZoneConv
/******************************************************/
/*        File Include fonctions                      */
/******************************************************/
/* for this file see task include a file in language AArch64 assembly */
.include "../includeARM64.inc"
```

```
Value  : +1
Value  : +2
Value  : +3

Value  : +2
Value  : +1
Value  : +3

Value  : +3
Value  : +1
Value  : +2

Value  : +1
Value  : +3
Value  : +2

Value  : +2
Value  : +3
Value  : +1

Value  : +3
Value  : +2
Value  : +1

Permutations =  +6
```

ABAP

```
data: lv_flag type c,
      lv_number type i,
      lt_numbers type table of i.

append 1 to lt_numbers.
append 2 to lt_numbers.
append 3 to lt_numbers.

do.
  perform permute using lt_numbers changing lv_flag.
  if lv_flag = 'X'.
    exit.
  endif.
  loop at lt_numbers into lv_number.
    write (1) lv_number no-gap left-justified.
    if sy-tabix <> '3'.
      write ', '.
    endif.
  endloop.
  skip.
enddo.
```

```
            changing ev_last type c.
  data: lv_len     type i,
        lv_first   type i,
        lv_third   type i,
        lv_count   type i,
        lv_temp    type i,
        lv_temp_2  type i,
        lv_second  type i,
        lv_changed type c,
        lv_perm    type i.
  describe table iv_set lines lv_len.

  lv_perm = lv_len - 1.
  lv_changed = ' '.
  " Loop backwards through the table, attempting to find elements which
  " can be permuted. If we find one, break out of the table and set the
  " flag indicating a switch.
  do.
    if lv_perm <= 0.
      exit.
    endif.
    " Read the elements.
    read table iv_set index lv_perm into lv_first.
    add 1 to lv_perm.
    read table iv_set index lv_perm into lv_second.
    subtract 1 from lv_perm.
    if lv_first < lv_second.
      lv_changed = 'X'.
      exit.
    endif.
    subtract 1 from lv_perm.
  enddo.

  " Last permutation.
  if lv_changed <> 'X'.
    ev_last = 'X'.
    exit.
  endif.

  " Swap tail decresing to get a tail increasing.
  lv_count = lv_perm + 1.
  do.
    lv_first = lv_len + lv_perm - lv_count + 1.
    if lv_count >= lv_first.
      exit.
    endif.

    read table iv_set index lv_count into lv_temp.
    read table iv_set index lv_first into lv_temp_2.
    modify iv_set index lv_count from lv_temp_2.
    modify iv_set index lv_first from lv_temp.
    add 1 to lv_count.
  enddo.

  lv_count = lv_len - 1.
  do.
    if lv_count <= lv_perm.
      exit.
    endif.

    read table iv_set index lv_count into lv_first.
    read table iv_set index lv_perm into lv_second.
    read table iv_set index lv_len into lv_third.
    if ( lv_first < lv_third ) and ( lv_first > lv_second ).
      lv_len = lv_count.
    endif.

    subtract 1 from lv_count.
  enddo.

  read table iv_set index lv_perm into lv_temp.
  read table iv_set index lv_len into lv_temp_2.
  modify iv_set index lv_perm from lv_temp_2.
  modify iv_set index lv_len from lv_temp.
endform.
```

```
1,  3,  2

2,  1,  3

2,  3,  1

3,  1,  2

3,  2,  1
```

Action!

```
PROC PrintArray(BYTE ARRAY a BYTE len)
  BYTE i

  FOR i=0 TO len-1
  DO
    PrintB(a(i))
  OD
  Print(" ")
RETURN

BYTE FUNC NextPermutation(BYTE ARRAY a BYTE len)
  BYTE i,j,k,tmp

  i=len-1
  WHILE i>0 AND a(i-1)>a(i)
  DO
    i==-1
  OD

  j=i
  k=len-1
  WHILE j<k
  DO
    tmp=a(j) a(j)=a(k) a(k)=tmp
    j==+1 k==-1
  OD

  IF i=0 THEN
    RETURN (0)
  FI

  j=i
  WHILE a(j)<a(i-1)
  DO
    j==+1
  OD
  tmp=a(i-1) a(i-1)=a(j) a(j)=tmp
RETURN (1)

PROC Main()
  DEFINE len="5"
  BYTE ARRAY a(len)
  BYTE RMARGIN=$53,oldRMARGIN
  BYTE i

  oldRMARGIN=RMARGIN
  RMARGIN=37 ;change right margin on the screen

  FOR i=0 TO len-1
  DO
    a(i)=i
  OD

  DO
    PrintArray(a,len)
  UNTIL NextPermutation(a,len)=0
  OD
```

Output:

Screenshot from Atari 8-bit computer (https://gitlab.com/amarok8bit/action-rosetta-code/-/raw/master/images/Permutations.png)

```
01234 01243 01324 01342 01423 01432
02134 02143 02314 02341 02413 02431
03124 03142 03214 03241 03412 03421
04123 04132 04213 04231 04312 04321
10234 10243 10324 10342 10423 10432
12034 12043 12304 12340 12403 12430
13024 13042 13204 13240 13402 13420
14023 14032 14203 14230 14302 14320
20134 20143 20314 20341 20413 20431
21034 21043 21304 21340 21403 21430
23014 23041 23104 23140 23401 23410
24013 24031 24103 24130 24301 24310
30124 30142 30214 30241 30412 30421
31024 31042 31204 31240 31402 31420
32014 32041 32104 32140 32401 32410
34012 34021 34102 34120 34201 34210
40123 40132 40213 40231 40312 40321
41023 41032 41203 41230 41302 41320
42013 42031 42103 42130 42301 42310
43012 43021 43102 43120 43201 43210
```

Ada

We split the task into two parts: The first part is to represent permutations, to initialize them and to go from one permutation to another one, until the last one has been reached. This can be used elsewhere, e.g., for the Topswaps [[1] (http://rosettacode.org/wiki/Topswops)] task. The second part is to read the N from the command line, and to actually print all permutations over 1 .. N.

The generic package Generic_Perm

When given N, this package defines the Element and Permutation types and exports procedures to set a permutation P to the first one, and to change P into the next one:

```
generic
   N: positive;
package Generic_Perm is
   subtype Element is Positive range 1 .. N;
   type Permutation is array(Element) of Element;

   procedure Set_To_First(P: out Permutation; Is_Last: out Boolean);
   procedure Go_To_Next(P: in out Permutation; Is_Last: out Boolean);
end Generic_Perm;
```

Here is the implementation of the package:

```
package body Generic_Perm is


   procedure Set_To_First(P: out Permutation; Is_Last: out Boolean) is
   begin
      for I in P'Range loop
     P (I) := I;
      end loop;
      Is_Last := P'Length = 1;
      -- if P has a single element, the fist permutation is the last one
```

```
      procedure Swap (A, B : in out Integer) is
         C : Integer := A;
      begin
         A := B;
         B := C;
      end Swap;

      I, J, K : Element;
   begin
      -- find longest tail decreasing sequence
      -- after the loop, this sequence is I+1 .. n,
      -- and the ith element will be exchanged later
      -- with some element of the tail
      Is_Last := True;
      I := N - 1;
      loop
     if P (I) < P (I+1)
     then
        Is_Last := False;
        exit;
     end if;

     -- next instruction will raise an exception if I = 1, so
     -- exit now (this is the last permutation)
     exit when I = 1;
     I := I - 1;
      end loop;

      -- if all the elements of the permutation are in
      -- decreasing order, this is the last one
      if Is_Last then
     return;
      end if;

      -- sort the tail, i.e. reverse it, since it is in decreasing order
      J := I + 1;
      K := N;
      while J < K loop
     Swap (P (J), P (K));
     J := J + 1;
     K := K - 1;
      end loop;

      -- find lowest element in the tail greater than the ith element
      J := N;
      while P (J) > P (I) loop
     J := J - 1;
      end loop;
      J := J + 1;

      -- exchange them
      -- this will give the next permutation in lexicographic order,
      -- since every element from ith to the last is minimum
      Swap (P (I), P (J));
   end Go_To_Next;

end Generic_Perm;
```

The procedure Print_Perms

```
with Ada.Text_IO, Ada.Command_Line, Generic_Perm;

procedure Print_Perms is
   package CML renames Ada.Command_Line;
   package TIO renames Ada.Text_IO;
begin
   declare
      package Perms is new Generic_Perm(Positive'Value(CML.Argument(1)));
      P : Perms.Permutation;
      Done : Boolean := False;
```

```
            TIO.Put (Perms.Element'Image (P (I)));
         end loop;
         TIO.New_Line;
      end Print;
   begin
      Perms.Set_To_First(P, Done);
      loop
         Print(P);
         exit when Done;
         Perms.Go_To_Next(P, Done);
      end loop;
   end;
exception
   when Constraint_Error
     => TIO.Put_Line ("*** Error: enter one numerical argument n with n >= 1");
end Print_Perms;
```

Output:

```
>./print_perms 3
 1 2 3
 1 3 2
 2 1 3
 2 3 1
 3 1 2
 3 2 1
 3 2 1
```

Aime

```
void
f1(record r, ...)
{
    if (~r) {
        for (text s in r) {
            r.delete(s);
            rcall(f1, -2, 0, -1, s);
            r[s] = 0;
        }
    } else {
        ocall(o_, -2, 1, -1, " ", ",");
        o_newline();
    }
}

main(...)
{
    record r;

    ocall(r_put, -2, 1, -1, r, 0);
    f1(r);

    0;
}
```

Output:

```
aime permutations -a Aaa Bb C
 Aaa, Bb, C,
 Aaa, C, Bb,
 Bb, Aaa, C,
 Bb, C, Aaa,
```

```
  C, Aaa, Bb,
  C, Bb, Aaa,
```

ALGOL 68

Works with: ALGOL 68 version Revision 1 - one minor extension to language used - PRAGMA READ, similar to C's #include directive.
Works with: ALGOL 68G version Any - tested with release algol68g-2.6 (http://sourceforge.net/projects/algol68/files/algol68g/algol68g-2.6).

File: prelude_permutations.a68

```
# -*- coding: utf-8 -*- #

COMMENT REQUIRED BY "prelude_permutations.a68"
  MODE PERMDATA = ~;
PROVIDES:
# PERMDATA*=~* #
# perm*=~ list* #
END COMMENT

MODE PERMDATALIST = REF[]PERMDATA;
MODE PERMDATALISTYIELD = PROC(PERMDATALIST)VOID;

# Generate permutations of the input data list of data list #
PROC perm gen permutations = (PERMDATALIST data list, PERMDATALISTYIELD yield)VOID: (
# Warning: this routine does not correctly handle duplicate elements #
  IF LWB data list = UPB data list THEN
    yield(data list)
  ELSE
    FOR elem FROM LWB data list TO UPB data list DO
      PERMDATA first = data list[elem];
      data list[LWB data list+1:elem] := data list[:elem-1];
      data list[LWB data list] := first;
    # FOR PERMDATALIST next data list IN # perm gen permutations(data list[LWB data list+1:] # ) DO #,
    ##   (PERMDATALIST next)VOID:(
        yield(data list)
    # OD #));
      data list[:elem-1] := data list[LWB data list+1:elem];
      data list[elem] := first
    OD
  FI
);

SKIP
```

File: test_permutations.a68

```
#!/usr/bin/a68g --script #
# -*- coding: utf-8 -*- #

CO REQUIRED BY "prelude_permutations.a68" CO
  MODE PERMDATA = INT;
#PROVIDES:#
# PERM*=INT* #
# perm *=int list *#
PR READ "prelude_permutations.a68" PR;

main:(
  FLEX[0]PERMDATA test case := (1, 22, 333, 44444);

  INT upb data list = UPB test case;
  FORMAT
    data fmt := $g(0)$,
    data list fmt := $"("n(upb data list-1)(f(data fmt)", ")f(data fmt)")"$;

# FOR DATALIST permutation IN # perm gen permutations(test case#) DO (#
```

```
)
```

Output:

```
(1, 22, 333, 44444)
(1, 22, 44444, 333)
(1, 333, 22, 44444)
(1, 333, 44444, 22)
(1, 44444, 22, 333)
(1, 44444, 333, 22)
(22, 1, 333, 44444)
(22, 1, 44444, 333)
(22, 333, 1, 44444)
(22, 333, 44444, 1)
(22, 44444, 1, 333)
(22, 44444, 333, 1)
(333, 1, 22, 44444)
(333, 1, 44444, 22)
(333, 22, 1, 44444)
(333, 22, 44444, 1)
(333, 44444, 1, 22)
(333, 44444, 22, 1)
(44444, 1, 22, 333)
(44444, 1, 333, 22)
(44444, 22, 1, 333)
(44444, 22, 333, 1)
(44444, 333, 1, 22)
(44444, 333, 22, 1)
```

Amazing Hopper

Translation of: AWK

```
/* hopper-JAMBO - a flavour of Amazing Hopper! */

#include <jambo.h>
Main
  leng=0
  Void(lista)
  Set("la realidad","escapa","a los sentidos"), Apnd list(lista)
  Length(lista), Move to(leng)
  Toksep(" ")
  Printnl( lista )
  Set(1) Gosub(Permutar)
End-Return

Subrutines

Define( Permutar, pos )
    If ( Sub(leng, pos) Isgeq(1) )
       i=pos
       Loop if( Less( i, leng ) )
          Plusone(pos), Gosub(Permutar)
          Set( pos ),   Gosub(Rotate)
          Printnl( lista )
          ++i
       Back
      Plusone(pos), Gosub(Permutar)
      Set( pos ),   Gosub(Rotate)
   End If
Return

Define ( Rotate, pos )
    c=0, [pos] Get(lista), Move to(c)
    [ Plusone(pos): leng ] Cget(lista)
    [ pos: Minusone(leng) ] Cput(lista)
    Set(c), [ leng ] Cput(lista)
Return
```

Output:

```
la realidad escapa a los sentidos
la realidad a los sentidos escapa
escapa a los sentidos la realidad
escapa la realidad a los sentidos
a los sentidos la realidad escapa
a los sentidos escapa la realidad
```

APL

For Dyalog APL(assumes index origin ⎕IO←1):

```
⍝ Builtin version, takes a vector:
⎕CY'dfns'
perms←{↓ω[pmat ≡⍵]} ⍝ pmat always gives lexicographically ordered permutations.

⍝ Recursive fast implementation, courtesy of dzaima from The APL Orchard:
dpmat←{1=ω:,⊂,0 ⋄ (⊃,/)¨(⍳ω)⌷¨⊂(⊂(!ω-1)⍴ω-1),⍨∇ω-1}
perms2←{↓ω[1+⍋⍋dpmat ≡⍵]}
```

```
      perms 'cat'
┌───┬───┬───┬───┬───┬───┐
│cat│cta│act│atc│tca│tac│
└───┴───┴───┴───┴───┴───┘
      perms2 'cat'
┌───┬───┬───┬───┬───┬───┐
│cta│atc│tac│tca│act│cat│
└───┴───┴───┴───┴───┴───┘
```

AppleScript

Recursive

Translation of: JavaScript

(Functional ES6 version)

Recursively, in terms of concatMap and delete:

```
----------------------- PERMUTATIONS ----------------------

-- permutations :: [a] -> [[a]]
on permutations(xs)
    script go
        on |λ|(xs)
            script h
                on |λ|(x)
                    script ts
                        on |λ|(ys)
                            {{x} & ys}
                        end |λ|
                    end script
                    concatMap(ts, go's |λ|(|delete|(x, xs)))
                end |λ|
            end script

            if {} ≠ xs then
                concatMap(h, xs)
            else
```

```
    end script
    go's |λ|(xs)
end permutations


--------------------------- TEST ---------------------------
on run

    permutations({"aardvarks", "eat", "ants"})

end run


-------------------- GENERIC FUNCTIONS ---------------------

-- concatMap :: (a -> [b]) -> [a] -> [b]
on concatMap(f, xs)
    set lst to {}
    set lng to length of xs
    tell mReturn(f)
        repeat with i from 1 to lng
            set lst to (lst & |λ|(contents of item i of xs, i, xs))
        end repeat
    end tell
    return lst
end concatMap


-- delete :: a -> [a] -> [a]
on |delete|(x, xs)
    if length of xs > 0 then
        set {h, t} to uncons(xs)
        if x = h then
            t
        else
            {h} & |delete|(x, t)
        end if
    else
        {}
    end if
end |delete|


-- Lift 2nd class handler function into 1st class script wrapper
-- mReturn :: Handler -> Script
on mReturn(f)
    if class of f is script then
        f
    else
        script
            property |λ| : f
        end script
    end if
end mReturn


-- uncons :: [a] -> Maybe (a, [a])
on uncons(xs)
    if length of xs > 0 then
        {item 1 of xs, rest of xs}
    else
        missing value
    end if
end uncons
```

Output:

```
{{"aardvarks", "eat", "ants"}, {"aardvarks", "ants", "eat"},
{"eat", "aardvarks", "ants"}, {"eat", "ants", "aardvarks"},
{"ants", "aardvarks", "eat"}, {"ants", "eat", "aardvarks"}}
```

Translation of: Pseudocode

```
to DoPermutations(aList, n)
    --> Heaps's algorithm (Permutation by interchanging pairs)
    if n = 1 then
        tell (a reference to PermList) to copy aList to its end
        -- or: copy aList as text (for concatenated results)
    else
        repeat with i from 1 to n
            DoPermutations(aList, n - 1)
            if n mod 2 = 0 then -- n is even
                tell aList to set [item i, item n] to [item n, item i] -- swaps items i and n of aList
            else
                tell aList to set [item 1, item n] to [item n, item 1] -- swaps items 1 and n of aList
            end if
        end repeat
    end if
    return (a reference to PermList) as list
end DoPermutations

--> Example 1 (list of words)
set [SourceList, PermList] to [{"Good", "Johnny", "Be"}, {}]
DoPermutations(SourceList, SourceList's length)
--> result (value of PermList)
{{"Good", "Johnny", "Be"}, {"Johnny", "Good", "Be"}, {"Be", "Good", "Johnny"}, ¬
    {"Good", "Be", "Johnny"}, {"Johnny", "Be", "Good"}, {"Be", "Johnny", "Good"}}

--> Example 2 (characters with concatenated results)
set [SourceList, PermList] to [{"X", "Y", "Z"}, {}]
DoPermutations(SourceList, SourceList's length)
--> result (value of PermList)
{"XYZ", "YXZ", "ZXY", "XZY", "YZX", "ZYX"}

--> Example 3 (Integers)
set [SourceList, Permlist] to [{1, 2, 3}, {}]
DoPermutations(SourceList, SourceList's length)
--> result (value of Permlist)
{{1, 2, 3}, {2, 1, 3}, {3, 1, 2}, {1, 3, 2}, {2, 3, 1}, {3, 2, 1}}

--> Example 4 (Integers with concatenated results)
set [SourceList, Permlist] to [{1, 2, 3}, {}]
DoPermutations(SourceList, SourceList's length)
--> result (value of Permlist)
{"123", "213", "312", "132", "231", "321"}
```

Non-recursive

As a right fold (which turns out to be significantly faster than recurse + delete):

```
---------------------- PERMUTATIONS ----------------------

-- permutations :: [a] -> [[a]]
on permutations(xs)
    script go
        on |λ|(x, a)
            script
                on |λ|(ys)
                    script infix
                        on |λ|(n)
                            if ys ≠ {} then
                                take(n, ys) & {x} & drop(n, ys)
                            else
                                {x}
                            end if
                        end |λ|
                    end script
                    map(infix, enumFromTo(0, (length of ys)))
                end |λ|
            end script
            concatMap(result, a)
        end |λ|
```

```
--------------------------- TEST ---------------------------
on run

    permutations({1, 2, 3})

    --> {{1, 2, 3}, {2, 1, 3}, {2, 3, 1}, {1, 3, 2}, {3, 1, 2}, {3, 2, 1}}
end run


------------------------- GENERIC --------------------------

-- concatMap :: (a -> [b]) -> [a] -> [b]
on concatMap(f, xs)
    set lng to length of xs
    set acc to {}
    tell mReturn(f)
        repeat with i from 1 to lng
            set acc to acc & |λ|(item i of xs, i, xs)
        end repeat
    end tell
    return acc
end concatMap


-- drop :: Int -> [a] -> [a]
on drop(n, xs)
    if n < length of xs then
        items (1 + n) thru -1 of xs
    else
        {}
    end if
end drop


-- enumFromTo :: Int -> Int -> [Int]
on enumFromTo(m, n)
    if m ≤ n then
        set lst to {}
        repeat with i from m to n
            set end of lst to i
        end repeat
        return lst
    else
        return {}
    end if
end enumFromTo


-- foldr :: (a -> b -> b) -> b -> [a] -> b
on foldr(f, startValue, xs)
    tell mReturn(f)
        set v to startValue
        set lng to length of xs
        repeat with i from lng to 1 by -1
            set v to |λ|(item i of xs, v, i, xs)
        end repeat
        return v
    end tell
end foldr


-- Lift 2nd class handler function into 1st class script wrapper
-- mReturn :: First-class m => (a -> b) -> m (a -> b)
on mReturn(f)
    if class of f is script then
        f
    else
        script
            property |λ| : f
        end script
    end if
end mReturn
```

```
    tell mReturn(f)
        set lng to length of xs
        set lst to {}
        repeat with i from 1 to lng
            set end of lst to |λ|(item i of xs, i, xs)
        end repeat
        return lst
    end tell
end map


-- min :: Ord a => a -> a -> a
on min(x, y)
    if y < x then
        y
    else
        x
    end if
end min


-- take :: Int -> [a] -> [a]
-- take :: Int -> String -> String
on take(n, xs)
    if 0 < n then
        items 1 thru min(n, length of xs) of xs
    else
        {}
    end if
end take
```

Output:

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

Recursive again

This is marginally faster even than the Pseudocode translation above and doesn't demarcate lists with square brackets, which don't officially exist in AppleScript. It returns the 362,880 permutations of a 9-item list in about a second and a half and the 3,628,800 permutations of a 10-item list in about 16 seconds. Don't let Script Editor attempt to display such large results or you'll have to force-quit it!

```
-- Translation of "Improved version of Heap's method (recursive)" found in
-- Robert Sedgewick's PDF document "Permutation Generation Methods"
-- <https://www.cs.princeton.edu/~rs/talks/perms.pdf>

on allPermutations(theList)
    script o
        -- Work list and precalculated indices for its last four items (assuming that many).
        property workList : missing value --(Set to a copy of theList below.)
        property r : (count theList)
        property rMinus1 : r - 1
        property rMinus2 : r - 2
        property rMinus3 : r - 3
        -- Output list and traversal index.
        property output : {}
        property p : 1

        -- Recursive handler.
        on prmt(l)
            -- Is the range length covered by this recursion level even?
            set rangeLenEven to ((r - l) mod 2 = 1)
            -- Tail call elimination repeat. Gives way to hard-coding for the lowest three levels.
            repeat with l from l to rMinus3
                -- Recursively permute items (l + 1) thru r of the work list.
```

```
            -- (if the range l to r is even) or with the rightmost item r - l times
            -- (if the range length is odd). The "recursion" after the last swap will
            -- instead be the next iteration of this tail call elimination repeat.
            if (rangeLenEven) then
                repeat with swapIdx from r to (lPlus1 + 1) by -1
                    tell my workList's item l
                        set my workList's item l to my workList's item swapIdx
                        set my workList's item swapIdx to it
                    end tell
                    prmt(lPlus1)
                end repeat
                set swapIdx to lPlus1
            else
                repeat (r - lPlus1) times
                    tell my workList's item l
                        set my workList's item l to my workList's item r
                        set my workList's item r to it
                    end tell
                    prmt(lPlus1)
                end repeat
                set swapIdx to r
            end if
            tell my workList's item l
                set my workList's item l to my workList's item swapIdx
                set my workList's item swapIdx to it
            end tell
            set rangeLenEven to (not rangeLenEven)
        end repeat
        -- Store a copy of the work list's current state.
        set my output's item p to my workList's items
        -- Then five more with the three rightmost items permuted.
        set v1 to my workList's item rMinus2
        set v2 to my workList's item rMinus1
        set v3 to my workList's end
        set my workList's item rMinus1 to v3
        set my workList's item r to v2
        set my output's item (p + 1) to my workList's items
        set my workList's item rMinus2 to v2
        set my workList's item r to v1
        set my output's item (p + 2) to my workList's items
        set my workList's item rMinus1 to v1
        set my workList's item r to v3
        set my output's item (p + 3) to my workList's items
        set my workList's item rMinus2 to v3
        set my workList's item r to v2
        set my output's item (p + 4) to my workList's items
        set my workList's item rMinus1 to v2
        set my workList's item r to v1
        set my output's item (p + 5) to my workList's items
        set p to p + 6
    end prmt
end script

if (o's r < 3) then
    -- Fewer than three items in the input list.
    copy theList to o's output's beginning
    if (o's r is 2) then set o's output's end to theList's reverse
else
    -- Otherwise prepare a list to hold (factorial of input list length) permutations …
    copy theList to o's workList
    set factorial to 2
    repeat with i from 3 to o's r
        set factorial to factorial * i
    end repeat
    set o's output to makeList(factorial, missing value)
    -- … and call o's recursive handler.
    o's prmt(1)
end if

return o's output
end allPermutations

on makeList(limit, filler)
    if (limit < 1) then return {}
    script o
```

```
    set counter to 1
    repeat until (counter + counter > limit)
        set o's lst to o's lst & o's lst
        set counter to counter + counter
    end repeat
    if (counter < limit) then set o's lst to o's lst & o's lst's items 1 thru (limit - counter)
    return o's lst
end makeList

return allPermutations({1, 2, 3, 4})
```

Output:

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

ARM Assembly

Works with: as version Raspberry Pi

```
/* ARM assembly Raspberry PI  */
/*  program permutation.s  */

 /* REMARK 1 : this program use routines in a include file
   see task Include a file language arm assembly
   for the routine affichageMess conversion10
   see at end of this program the instruction include */
/* for constantes see task include a file in arm assembly */
/************************************/
/* Constantes                       */
/************************************/
.include "../constantes.inc"

/*********************************/
/* Initialized data              */
/*********************************/
.data

sMessResult:        .asciz "Value  : @ \n"
sMessCounter:       .asciz "Permutations =  @  \n"
szCarriageReturn:   .asciz "\n"

.align 4
TableNumber:       .int   1,2,3
                   .equ NBELEMENTS, (. - TableNumber) / 4
/*********************************/
/* UnInitialized data            */
/*********************************/
.bss
sZoneConv:            .skip 24
/*********************************/
/*  code section                 */
/*********************************/
.text
.global main
main:                                           @ entry of program
    ldr r0,iAdrTableNumber                      @ address number table
    mov r1,#NBELEMENTS                          @ number of élements
    mov r10,#0                                  @ counter
    bl heapIteratif
    mov r0,r10                                  @ display counter
    ldr r1,iAdrsZoneConv                        @
    bl conversion10S                            @ décimal conversion
    ldr r0,iAdrsMessCounter
    ldr r1,iAdrsZoneConv                        @ insert conversion
    bl strInsertAtCharInc
    bl affichageMess                            @ display message
```

```
    svc #0                                      @ perform the system call

iAdrszCarriageReturn:     .int szCarriageReturn
iAdrsMessResult:          .int sMessResult
iAdrTableNumber:          .int TableNumber
iAdrsMessCounter:         .int sMessCounter
/******************************************************************/
/*     permutation by heap iteratif (wikipedia)                                  */
/******************************************************************/
/* r0 contains the address of table */
/* r1 contains the eléments number  */
heapIteratif:
    push {r3-r9,lr}               @ save registers
    lsl r9,r1,#2                  @ four bytes by count
    sub sp,sp,r9
    mov fp,sp
    mov r3,#0
    mov r4,#0                     @ index
1:                                @ init area counter
    str r4,[fp,r3,lsl #2]
    add r3,r3,#1
    cmp r3,r1
    blt 1b

    bl displayTable
    add r10,r10,#1
    mov r3,#0                      @ index
2:
    ldr r4,[fp,r3,lsl #2]          @ load count [i]
    cmp r4,r3                      @ compare with i
    bge 5f
    tst r3,#1                      @ even ?
    bne 3f
    ldr r5,[r0]                    @ yes load value A[0]
    ldr r6,[r0,r3,lsl #2]          @ and swap with value A[i]
    str r6,[r0]
    str r5,[r0,r3,lsl #2]
    b 4f
3:
    ldr r5,[r0,r4,lsl #2]         @ load value A[count[i]]
    ldr r6,[r0,r3,lsl #2]         @ and swap with value A[i]
    str r6,[r0,r4,lsl #2]
    str r5,[r0,r3,lsl #2]
4:
    bl displayTable
    add r10,r10,#1
    add r4,r4,#1                   @ increment count i
    str r4,[fp,r3,lsl #2]          @ and store on stack
    mov r3,#0                      @ raz index
    b 2b                           @ and loop
5:
    mov r4,#0                      @ raz count [i]
    str r4,[fp,r3,lsl #2]
    add r3,r3,#1                   @ increment index
    cmp r3,r1                      @ end ?
    blt 2b                         @ no -> loop

    add sp,sp,r9                   @ stack alignement
100:
    pop {r3-r9,lr}
    bx lr                          @ return

/******************************************************************/
/*      Display table elements                                    */
/******************************************************************/
/* r0 contains the address of table */
displayTable:
    push {r0-r3,lr}                                 @ save registers
    mov r2,r0                                       @ table address
    mov r3,#0
1:                                                  @ loop display table
    ldr r0,[r2,r3,lsl #2]
    ldr r1,iAdrsZoneConv                            @
    bl conversion10S                                 @ décimal conversion
    ldr r0,iAdrsMessResult
```

```
    add r3,#1
    cmp r3,#NBELEMENTS - 1
    ble 1b
    ldr r0,iAdrszCarriageReturn
    bl affichageMess
    mov r0,r2
100:
    pop {r0-r3,lr}
    bx lr
iAdrsZoneConv:           .int sZoneConv
/*****************************************************/
/*      ROUTINES INCLUDE                             */
/*****************************************************/
.include "../affichage.inc"
```

```
Value  :          +1
Value  :          +2
Value  :          +3

Value  :          +2
Value  :          +1
Value  :          +3

Value  :          +3
Value  :          +1
Value  :          +2

Value  :          +1
Value  :          +3
Value  :          +2

Value  :          +2
Value  :          +3
Value  :          +1

Value  :          +3
Value  :          +2
Value  :          +1

Permutations =           +6
```

Arturo

```
print permutate [1 2 3]
```

Output:

```
[1 2 3] [1 3 2] [2 1 3] [2 3 1] [3 1 2] [3 2 1]
```

AutoHotkey

from the forum topic http://www.autohotkey.com/forum/viewtopic.php?t=77959

```
#NoEnv
StringCaseSense On

o := str := "Hello"

Loop
{
   str := perm_next(str)
   If !str
```

```
   o.= "`n" . str
}

perm_Next(str){
   p := 0, sLen := StrLen(str)
   Loop % sLen
   {
      If A_Index=1
         continue
      t := SubStr(str, sLen+1-A_Index, 1)
      n := SubStr(str, sLen+2-A_Index, 1)
      If ( t < n )
      {
         p := sLen+1-A_Index, pC := SubStr(str, p, 1)
         break
      }
   }
   If !p
      return false
   Loop
   {
      t := SubStr(str, sLen+1-A_Index, 1)
      If ( t > pC )
      {
         n := sLen+1-A_Index, nC := SubStr(str, n, 1)
         break
      }
   }
   return SubStr(str, 1, p-1) . nC . Reverse(SubStr(str, p+1, n-p-1) . pC .  SubStr(str, n+1))
}

Reverse(s){
   Loop Parse, s
      o := A_LoopField o
   return o
}
```

Output:

```
Hello
Helol
Heoll
Hlelo
Hleol
Hlleo
Hlloe
Hloel
Hlole
Hoell
Holel
Holle
eHllo
eHlol
eHoll
```

Alternate Version

Alternate version to produce numerical permutations of combinations.

```
P(n,k="",opt=0,delim="",str="") { ; generate all n choose k permutations lexicographically
    ;1..n = range, or delimited list, or string to parse
    ;   to process with a different min index, pass a delimited list, e.g. "0`n1`n2"
    ;k = length of result
    ;opt 0 = no repetitions
    ;opt 1 = with repetitions
    ;opt 2 = run for 1..k
    ;opt 3 = run for 1..k with repetitions
```

```
    If !InStr(n,"`n")
        If n in 2,3,4,5,6,7,8,9
            Loop, %n%
                n := A_Index = 1 ? A_Index : n "`n" A_Index
        Else
            Loop, Parse, n, %delim%
                n := A_Index = 1 ? A_LoopField : n "`n" A_LoopField
    If (k = "")
        RegExReplace(n,"`n","",k), k++
    If k is not Digit
        Return "k must be a digit."
    If opt not in 0,1,2,3
        Return "opt invalid."
    If k = 0
        Return str
    Else
        Loop, Parse, n, `n
            If (!InStr(str,A_LoopField) || opt & 1)
                s .= (!i++ ? (opt & 2 ? str "`n" : "") : "`n" )
                    . P(n,k-1,opt,delim,str . A_LoopField . delim)
        Return s
}
```

Output:

```
MsgBox % P(3)
```

```
---------------------------
permute.ahk
---------------------------
123
132
213
231
312
321
---------------------------
OK
---------------------------
```

```
MsgBox % P("Hello",3)
```

```
---------------------------
permute.ahk
---------------------------
Hel
Hel
Heo
Hle
Hlo
Hle
Hlo
Hoe
Hol
Hol
eHl
eHl
```

```
MsgBox % P("2`n3`n4`n5",2,3)
```

```
2
22
23
24
25
3
32
33
34
35
```

```
MsgBox % P("11 a text ] u+z",3,0," ")
```

```
---------------------------
permute.ahk
---------------------------
11 a text
11 a ]
11 a u+z
11 text a
11 text ]
11 text u+z
11 ] a
11 ] text
11 ] u+z
11 u+z a
11 u+z text
11 u+z ]
```

AWK

```
# syntax: GAWK -f PERMUTATIONS.AWK [-v sep=x] [word]
#
# examples:
#   REM all permutations on one line
#   GAWK -f PERMUTATIONS.AWK
#
#   REM all permutations on a separate line
#   GAWK -f PERMUTATIONS.AWK -v sep="\n"
#
#   REM use a different word
#   GAWK -f PERMUTATIONS.AWK Gwen
#
#   REM command used for RosettaCode output
#   GAWK -f PERMUTATIONS.AWK -v sep="\n" Gwen
#
BEGIN {
    sep = (sep == "") ? " " : substr(sep,1,1)
    str = (ARGC == 1) ? "abc" : ARGV[1]
    printf("%s%s",str,sep)
    leng = length(str)
    for (i=1; i<=leng; i++) {
      arr[i-1] = substr(str,i,1)
    }
    ana_permute(0)
    exit(0)
}
function ana_permute(pos,  i,j,str) {
    if (leng - pos < 2) { return }
    for (i=pos; i<leng-1; i++) {
      ana_permute(pos+1)
      ana_rotate(pos)
      for (j=0; j<=leng-1; j++) {
        printf("%s",arr[j])
      }
      printf(sep)
```

```
}
function ana_rotate(pos,  c,i) {
    c = arr[pos]
    for (i=pos; i<leng-1; i++) {
      arr[i] = arr[i+1]
    }
    arr[leng-1] = c
}
```

sample command:

GAWK -f PERMUTATIONS.AWK Gwen

Output:

```
Gwen Gwne Genw Gewn Gnwe Gnew wenG weGn wnGe wneG wGen wGne enGw enwG eGwn eGnw ewnG ewGn nGwe nGew nweG nwGe neGw newG
```

BASIC

Applesoft BASIC

Translation of: Commodore BASIC

Shortened from Commodore BASIC to seven lines. Integer arrays are used instead of floating point. GOTO is used instead of GOSUB to avoid OUT OF MEMORY ERROR due to the call stack being full for values greater than 100.

```
 10  INPUT "HOW MANY? ";N:J = N - 1
 20 S$ = " ":M$ = S$ +  CHR$ (13):T = 0: DIM A%(J),K%(J),I%(J),R%(J): FOR I = 0 TO J:A%(I) = I + 1: NEXT :K%(S) = N:R = S:R%(R)
= 0:S = S + 1
 30  IF K%(R) <  = 1 THEN  FOR I = 0 TO N - 1: PRINT  MID$ (S$,(I = 0) + 1,1)A%(I);: NEXT I:S$ = M$: GOTO 70
 40 K%(S) = K%(R) - 1:R%(S) = 0:R = S:S = S + 1: GOTO 30
 50 J = I%(R) * (1 - (K%(R) -  INT (K%(R) / 2) * 2)):T = A%(J):A%(J) = A%(K%(R) - 1):A%(K%(R) - 1) = T:K%(S) = K%(R) - 1:R%(S)
= 1:R = S:S = S + 1: GOTO 30
 60 I%(R) = (I%(R) + 1) * R%(S): IF I%(R) < K%(R) - 1 GOTO 50
 70 S = S - 1:R = S - 1: IF R >  = 0 GOTO 60
```

Output:

```
HOW MANY? 3
1 2 3
2 1 3
3 1 2
1 3 2
2 3 1
3 2 1
```

```
HOW MANY? 4483

?OUT OF MEMORY ERROR IN 20
```

```
HOW MANY? 4482
BREAK IN 30
]?FRE(0)
1
```

```
arraybase 1
n = 4 : cont = 0
dim a(n)
dim c(n)

for j = 1 to n
    a[j] = j
next j

do
    for i = 1 to n
        print a[i];
    next
    print " ";

    i = n
    cont += 1
    if cont = 12 then
        print
        cont = 0
    else
        print " ";
    end if

    do
        i -= 1
    until (i = 0) or (a[i] < a[i+1])
    j = i + 1
    k = n
    while j < k
        tmp = a[j] : a[j] = a[k] : a[k] = tmp
        j += 1
        k -= 1
    end while
    if i > 0 then
        j = i + 1
        while a[j] < a[i]
            j += 1
        end while
        tmp = a[j] : a[j] = a[i] : a[i] = tmp
    end if
until i = 0
end
```

BBC BASIC

The procedure PROC_NextPermutation() will give the next lexicographic permutation of an integer array.

```
      DIM List%(3)
      List%() = 1, 2, 3, 4
      FOR perm% = 1 TO 24
        FOR i% = 0 TO DIM(List%(),1)
          PRINT List%(i%);
        NEXT
        PRINT
        PROC_NextPermutation(List%())
      NEXT
      END

      DEF PROC_NextPermutation(A%())
      LOCAL first, last, elementcount, pos
      elementcount = DIM(A%(),1)
      IF elementcount < 1 THEN ENDPROC
      pos = elementcount-1
      WHILE A%(pos) >= A%(pos+1)
        pos -= 1
        IF pos < 0 THEN
          PROC_Permutation_Reverse(A%(), 0, elementcount)
          ENDPROC
```

```
      WHILE A%(last) <= A%(pos)
        last -= 1
      ENDWHILE
      SWAP A%(pos), A%(last)
      PROC_Permutation_Reverse(A%(), pos+1, elementcount)
      ENDPROC

      DEF PROC_Permutation_Reverse(A%(), first, last)
      WHILE first < last
        SWAP A%(first), A%(last)
        first += 1
        last -= 1
      ENDWHILE
      ENDPROC
```

Output:

```
         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
```

Commodore BASIC

Heap's algorithm, using a couple extra arrays as stacks to permit recursive calls.

```
100 INPUT "HOW MANY";N
110 DIM A(N-1):REM ARRAY TO PERMUTE
120 DIM K(N-1):REM HOW MANY ITEMS TO PERMUTE (ARRAY AS STACK)
130 DIM I(N-1):REM CURRENT POSITION IN LOOP (ARRAY AS STACK)
140 S=0:REM STACK POINTER
150 FOR I=0 TO N-1
160 : A(I)=I+1: REM INITIALIZE ARRAY TO 1..N
170 NEXT I
180 K(S)=N:S=S+1:GOSUB 200:REM PERMUTE(N)
190 END
200 IF K(S-1)>1 THEN 270
210 REM PRINT OUT THIS PERMUTATION
220 FOR I=0 TO N-1
230 : PRINT A(I);
240 NEXT I
250 PRINT
260 RETURN
270 K(S)=K(S-1)-1:S=S+1:GOSUB 200:S=S-1:REM PERMUTE(K-1)
280 I(S-1)=0:REM FOR I=0 TO K-2
290 IF I(S-1)>=K(S-1)-1 THEN 340
300 J=I(S-1):IF K(S-1) AND 1 THEN J=0:REM ELEMENT TO SWAP BASED ON PARITY OF K
```

```
330 I(S-1)=I(S-1)+1:GOTO 290:REM NEXT I
340 RETURN
```

Output:

```
READY.
RUN
HOW MANY? 3
 1  2  3
 2  1  3
 3  1  2
 1  3  2
 2  3  1
 3  2  1

READY.
```

Craft Basic

```
let n = 3
let i = n + 1

dim a[i]

for i = 1 to n

    let a[i] = i

next i

do

    for i = 1 to n

        print a[i]

    next i

    print

    let i = n

    do

        let i = i - 1
        let b = i + 1

    loopuntil (i = 0) or (a[i] < a[b])

    let j = i + 1
    let k = n

    do

        if j < k then

            let t = a[j]
            let a[j] = a[k]
            let a[k] = t
            let j = j + 1
            let k = k - 1

        endif

    loop j < k

    if i > 0 then

        let i = i + 1
```

```
            if a[j] < a[i] then

                let j = j + 1

            endif

        loop a[j] < a[i]

        let t = a[j]
        let a[j] = a[i]
        let a[i] = t

    endif

loopuntil i = 0
```

Output:

```
1 2 3
1 3 2
2 1 3
2 3 1
3 1 2
3 2 1
```

FreeBASIC

```
' version 07-04-2017
' compile with: fbc -s console

' Heap's algorithm non-recursive
Sub perms(n As Long)

    Dim As ULong i, j, count = 1
    Dim As ULong a(0 To n -1), c(0 To n -1)

    For j = 0 To n -1
        a(j) = j +1
        Print a(j);
    Next
    Print " ";

    i = 0
    While i < n
        If c(i) < i Then
            If (i And 1) = 0 Then
                Swap a(0), a(i)
            Else
                Swap a(c(i)), a(i)
            End If
            For j = 0 To n -1
                Print a(j);
            Next
            count += 1
            If count = 12 Then
                Print
                count = 0
            Else
                Print " ";
            End If
            c(i) += 1
```

```
        End If
    Wend

End Sub

' ------=< MAIN >=------

perms(4)

' empty keyboard buffer
While Inkey <> "" : Wend
Print : Print "hit any key to end program"
Sleep
End
```

Output:

```
1234 2134 3124 1324 2314 3214 4213 2413 1423 4123 2143 1243
1342 3142 4132 1432 3412 4312 4321 3421 2431 4231 3241 2341
```

IS-BASIC

```
100 PROGRAM "Permutat.bas"
110 LET N=4 ! Number of elements
120 NUMERIC T(1 TO N)
130 FOR I=1 TO N
140   LET T(I)=I
150 NEXT
160 LET S=0
170 CALL PERM(N)
180 PRINT "Number of permutations:";S
190 END
200 DEF PERM(I)
210   NUMERIC J,X
220   IF I=1 THEN
230     FOR X=1 TO N
240       PRINT T(X);
250     NEXT
260     PRINT :LET S=S+1
270   ELSE
280     CALL PERM(I-1)
290     FOR J=1 TO I-1
300       LET C=T(J):LET T(J)=T(I):LET T(I)=C
310       CALL PERM(I-1)
320       LET C=T(J):LET T(J)=T(I):LET T(I)=C
330     NEXT
340   END IF
350 END DEF
```

Liberty BASIC

Permuting numerical array (non-recursive):

Translation of: PowerBASIC

```
n=3
dim a(n+1)  '+1 needed due to bug in LB that checks loop condition
    '   until (i=0) or (a(i)<a(i+1))
    'before executing i=i-1 in loop body.
for i=1 to n: a(i)=i: next
do
```

```
    i=i-1
  loop until (i=0) or (a(i)<a(i+1))
  j=i+1
  k=n
  while j<k
    'swap a(j),a(k)
    tmp=a(j): a(j)=a(k): a(k)=tmp
    j=j+1
    k=k-1
  wend
  if i>0 then
    j=i+1
    while a(j)<a(i)
      j=j+1
    wend
    'swap a(i),a(j)
    tmp=a(j): a(j)=a(i): a(i)=tmp
  end if
loop until i=0
```

Output:

```
123
132
213
231
312
321
```

Permuting string (recursive):

```
n = 3

s$=""
for i = 1 to n
    s$=s$;i
next

res$=permutation$("", s$)

Function permutation$(pre$, post$)
    lgth = Len(post$)
    If lgth < 2 Then
        print pre$;post$
    Else
        For i = 1 To lgth
            tmp$=permutation$(pre$+Mid$(post$,i,1),Left$(post$,i-1)+Right$(post$,lgth-i))
        Next i
    End If
End Function
```

Output:

```
123
132
213
231
312
321
```

Microsoft Small Basic

Translation of: vba

```
  For i = 1 To n
    p[i] = i
  EndFor
  count = 0
  Last = "False"
  While Last = "False"
    If printem Then
      For t = 1 To n
        TextWindow.Write(p[t])
      EndFor
      TextWindow.WriteLine("")
    EndIf
    count = count + 1
    Last = "True"
    i = n - 1
    While i > 0
      If p[i] < p[i + 1] Then
        Last = "False"
        Goto exitwhile
      EndIf
      i = i - 1
    EndWhile
    exitwhile:
    j = i + 1
    k = n
    While j < k
      t = p[j]
      p[j] = p[k]
      p[k] = t
      j = j + 1
      k = k - 1
    EndWhile
    j = n
    While p[j] > p[i]
      j = j - 1
    EndWhile
    j = j + 1
    t = p[i]
    p[i] = p[j]
    p[j] = t
  EndWhile
  TextWindow.WriteLine("Number of permutations: "+count)
```

Output:

```
1234
1243
1324
1342
1423
1432
2134
2143
2314
2341
2413
2431
3124
3142
3214
3241
3412
3421
4123
4132
4213
4231
4312
```

```
4321
Number of permutations: 24
```

PowerBASIC

Works with: PowerBASIC version 10.00+

```
#COMPILE EXE
#DIM ALL
GLOBAL a, i, j, k, n  AS INTEGER
GLOBAL d, ns, s AS STRING 'dynamic string
FUNCTION PBMAIN () AS LONG
ns = INPUTBOX$("   n =",, "3") 'input n
n = VAL(ns)
DIM a(1 TO n) AS INTEGER
FOR i = 1 TO n: a(i)= i: NEXT
DO
  s = " "
  FOR i = 1 TO n
    d = STR$(a(i))
    s = BUILD$(s, d) '  s & d concatenate
  NEXT
  ? s  'print and pause
  i = n
  DO
   DECR i
  LOOP UNTIL i = 0 OR a(i) < a(i+1)
  j = i+1
  k = n
  DO WHILE j < k
    SWAP a(j), a(k)
    INCR j
    DECR k
  LOOP
  IF i > 0 THEN
    j = i+1
    DO WHILE a(j) < a(i)
      INCR j
    LOOP
    SWAP a(i), a(j)
  END IF
LOOP UNTIL i = 0
END FUNCTION
```

Output:

```
1 2 3
1 3 2
2 1 3
2 3 1
3 1 2
3 2 1
```

PureBasic

The procedure nextPermutation() takes an array of integers as input and transforms its contents into the next lexicographic permutation of it's elements (i.e. integers). It returns #True if this is possible. It returns #False if there are no more lexicographic permutations left and arranges the elements into the lowest lexicographic permutation. It also returns #False if there is less than 2 elemetns to permute.

The integer elements could be the addresses of objects that are pointed at instead. In this case the addresses will be permuted without respect to what they are pointing to (i.e. strings, or structures) and the lexicographic order will be that of the addresses themselves

```
Macro reverse(firstIndex, lastIndex)
  first = firstIndex
  last = lastIndex
  While first < last
    Swap cur(first), cur(last)
    first + 1
    last - 1
  Wend
EndMacro

Procedure nextPermutation(Array cur(1))
  Protected first, last, elementCount = ArraySize(cur())
  If elementCount < 1
    ProcedureReturn #False ;nothing to permute
  EndIf

  ;Find the lowest position pos such that [pos] < [pos+1]
  Protected pos = elementCount - 1
  While cur(pos) >= cur(pos + 1)
    pos - 1
    If pos < 0
      reverse(0, elementCount)
      ProcedureReturn #False ;no higher lexicographic permutations left, return lowest one instead
    EndIf
  Wend

  ;Swap [pos] with the highest positional value that is larger than [pos]
  last = elementCount
  While cur(last) <= cur(pos)
    last - 1
  Wend
  Swap cur(pos), cur(last)

  ;Reverse the order of the elements in the higher positions
  reverse(pos + 1, elementCount)
  ProcedureReturn #True ;next lexicographic permutation found
EndProcedure

Procedure display(Array a(1))
  Protected i, fin = ArraySize(a())
  For i = 0 To fin
    Print(Str(a(i)))
    If i = fin: Continue: EndIf
    Print(", ")
  Next
  PrintN("")
EndProcedure

If OpenConsole()
  Dim a(2)
  a(0) = 1: a(1) = 2: a(2) =  3
  display(a())
  While nextPermutation(a()): display(a()): Wend

  Print(#CRLF$ + #CRLF$ + "Press ENTER to exit"): Input()
  CloseConsole()
EndIf
```

Output:

```
1, 2, 3
1, 3, 2
2, 1, 3
2, 3, 1
3, 1, 2
3, 2, 1
```

QBasic

Translation of: FreeBASIC

```
SUB perms (n)
    DIM a(0 TO n - 1), c(0 TO n - 1)

    FOR j = 0 TO n - 1
        a(j) = j + 1
        PRINT a(j);
    NEXT j
    PRINT

    i = 0
    WHILE i < n
        IF c(i) < i THEN
            IF (i AND 1) = 0 THEN
                SWAP a(0), a(i)
            ELSE
                SWAP a(c(i)), a(i)
            END IF
            FOR j = 0 TO n - 1
                PRINT a(j);
            NEXT j
            PRINT
            c(i) = c(i) + 1
            i = 0
        ELSE
            c(i) = 0
            i = i + 1
        END IF
    WEND
END SUB

perms(4)
```

Run BASIC

Works with Run BASIC, Liberty BASIC and Just BASIC

```
list$ = "h,e,l,l,o"     ' supply list seperated with comma's

while word$(list$,d+1,",") <> ""  'Count how many in the list
d = d + 1
wend

dim theList$(d)         ' place list in array
for i = 1 to d
  theList$(i) = word$(list$,i,",")
next i

for i = 1 to d          ' print the Permutations
 for j = 2 to d
   perm$ = ""
   for k = 1 to d
    perm$ = perm$ + theList$(k)
   next k
   if instr(perm2$,perm$+",") = 0 then print perm$ ' only list 1 time
   perm2$    = perm2$ + perm$ + ","
   h$        = theList$(j)
   theList$(j)   = theList$(j - 1)
   theList$(j - 1) = h$
  next j
next i
end
```

Output:

```
```

```
ellho
elloh
leloh
lleoh
lloeh
llohe
lolhe
lohle
lohel
olhel
ohlel
ohell
hoell
heoll
helol
```

True BASIC

Translation of: Liberty BASIC

```
SUB SWAP(vb1, vb2)
    LET temp = vb1
    LET vb1 = vb2
    LET vb2 = temp
END SUB

LET n = 4
DIM a(4)
DIM c(4)

FOR i = 1 TO n
    LET a(i) = i
NEXT i
PRINT

DO
   FOR i = 1 TO n
       PRINT a(i);
   NEXT i
   PRINT
   LET i = n
   DO
      LET i = i - 1
   LOOP UNTIL (i = 0) OR (a(i) < a(i + 1))
   LET j = i + 1
   LET k = n
   DO WHILE j < k
      CALL SWAP (a(j), a(k))
      LET j = j + 1
      LET k = k - 1
   LOOP
   IF i > 0 THEN
      LET j = i + 1
      DO WHILE a(j) < a(i)
         LET j = j + 1
      LOOP
      CALL SWAP (a(i), a(j))
   END IF
LOOP UNTIL i = 0
END
```

Yabasic

Translation of: Liberty BASIC

```
n = 4
dim a(n), c(n)

for i = 1 to n : a(i) = i : next i
```

```
  i = n
  repeat
    i = i - 1
  until (i = 0) or (a(i) < a(i+1))
  j = i + 1
  k = n
  while j < k
    tmp = a(j) : a(j) = a(k) : a(k) = tmp
    j = j + 1
    k = k - 1
  wend
  if i > 0 then
    j = i + 1
    while a(j) < a(i)
      j = j + 1
    wend
    tmp = a(j) : a(j) = a(i) : a(i) = tmp
  endif
until i = 0
end
```

Batch File

Recursive permutation generator.

```
@echo off
setlocal enabledelayedexpansion
set arr=ABCD
set /a n=4
:: echo !arr!
call :permu  %n% arr
goto:eof

:permu num  &arr
setlocal
if %1 equ 1 call echo(!%2! & exit /b
set /a "num=%1-1,n2=num-1"
set arr=!%2!
for /L %%c in (0,1,!n2!) do (
   call:permu !num! arr
   set /a  n1="num&1"
   if !n1! equ 0 (call:swapit !num! 0 arr) else (call:swapit !num! %%c arr)
   )
   call:permu !num! arr
endlocal & set %2=%arr%
exit /b

:swapit  from  to  &arr
setlocal
set arr=!%3!
set temp1=!arr:~%~1,1!
set temp2=!arr:~%~2,1!
set arr=!arr:%temp1%=@!
set arr=!arr:%temp2%=%temp1%!
set arr=!arr:@=%temp2%!
:: echo %1 %2 !%~3! !arr!
endlocal & set %3=%arr%
exit /b
```

Output:

```
ABCD
BACD
CABD
ACBD
BCAD
CBAD
DBAC
```

```
BADC
ABDC
ACDB
CADB
DACB
ADCB
CDAB
DCAB
DCBA
CDBA
BDCA
DBCA
CBDA
BCDA
```

Bracmat

```
  ( perm
  =   prefix List result original A Z
    .   !arg:(?.)
      |   !arg:(?prefix.?List:?original)
        & :?result
        &   whl
          ' ( !List:%?A ?Z
            & !result perm$(!prefix !A.!Z):?result
            & !Z !A:~!original:?List
            )
        & !result
  )
& out$(perm$(.a 2 "]" u+z);
```

Output:

```
  (a 2 ] u+z.)
  (a 2 u+z ].)
  (a ] u+z 2.)
  (a ] 2 u+z.)
  (a u+z 2 ].)
  (a u+z ] 2.)
  (2 ] u+z a.)
  (2 ] a u+z.)
  (2 u+z a ].)
  (2 u+z ] a.)
  (2 a ] u+z.)
  (2 a u+z ].)
  (] u+z a 2.)
  (] u+z 2 a.)
  (] a 2 u+z.)
  (] a u+z 2.)
  (] 2 u+z a.)
  (] 2 a u+z.)
  (u+z a 2 ].)
  (u+z a ] 2.)
  (u+z 2 ] a.)
  (u+z 2 a ].)
  (u+z ] a 2.)
  (u+z ] 2 a.)
```

C

version 1

Non-recursive algorithm to generate all permutations. It prints objects in lexicographical order.

```
#include <stdio.h>
int main (int argc, char *argv[]) {
//here we check arguments
    if (argc < 2) {
        printf("Enter an argument. Example 1234 or dcba:\n");
        return 0;
    }
//it calculates an array's length
        int x;
        for (x = 0; argv[1][x] != '\0'; x++);
//buble sort the array
    int f, v, m;
     for(f=0; f < x; f++) {
         for(v = x-1; v > f; v-- ) {
         if (argv[1][v-1] > argv[1][v]) {
    m=argv[1][v-1];
    argv[1][v-1]=argv[1][v];
    argv[1][v]=m;
    }
  }
}

//it calculates a factorial to stop the algorithm
    char a[x];
    int k=0;
    int fact=k+1;
             while (k!=x) {
                   a[k]=argv[1][k];
                   k++;
          fact = k*fact;
                   }
                   a[k]='\0';
//Main part: here we permutate
           int i, j;
           int y=0;
           char c;
          while (y != fact) {
          printf("%s\n", a);
          i=x-2;
          while(a[i] > a[i+1] ) i--;
          j=x-1;
          while(a[j] < a[i] ) j--;
      c=a[j];
      a[j]=a[i];
      a[i]=c;
i++;
for (j = x-1; j > i; i++, j--) {
  c = a[i];
  a[i] = a[j];
  a[j] = c;
      }
y++;
   }
}
```

version 2

Non-recursive algorithm to generate all permutations. It prints them from right to left.

```
#include <stdio.h>
int main() {
        char a[] = "4321";  //array
           int i, j;
           int f=24;        //factorial
           char c;          //buffer
          while (f--) {
          printf("%s\n", a);
          i=1;
          while(a[i] > a[i-1]) i++;
```

```
      a[j]=a[i];
      a[i]=c;
i--;
for (j = 0; j < i; i--, j++) {
  c = a[i];
  a[i] = a[j];
  a[j] = c;
      }
   }
}
```

version 3

See lexicographic generation of permutations.

```
#include <stdio.h>
#include <stdlib.h>

/* print a list of ints */
int show(int *x, int len)
{
    int i;
    for (i = 0; i < len; i++)
        printf("%d%c", x[i], i == len - 1 ? '\n' : ' ');
    return 1;
}

/* next lexicographical permutation */
int next_lex_perm(int *a, int n) {
#   define swap(i, j) {t = a[i]; a[i] = a[j]; a[j] = t;}
    int k, l, t;

    /* 1. Find the largest index k such that a[k] < a[k + 1]. If no such
          index exists, the permutation is the last permutation. */
    for (k = n - 1; k && a[k - 1] >= a[k]; k--);
    if (!k--) return 0;

    /* 2. Find the largest index l such that a[k] < a[l]. Since k + 1 is
       such an index, l is well defined */
    for (l = n - 1; a[l] <= a[k]; l--);

    /* 3. Swap a[k] with a[l] */
    swap(k, l);

    /* 4. Reverse the sequence from a[k + 1] to the end */
    for (k++, l = n - 1; l > k; l--, k++)
        swap(k, l);
    return 1;
#   undef swap
}

void perm1(int *x, int n, int callback(int *, int))
{
    do {
        if (callback) callback(x, n);
    } while (next_lex_perm(x, n));
}

/* Boothroyd method; exactly N! swaps, about as fast as it gets */
void boothroyd(int *x, int n, int nn, int callback(int *, int))
{
    int c = 0, i, t;
    while (1) {
        if (n > 2) boothroyd(x, n - 1, nn, callback);
        if (c >= n - 1) return;

        i = (n & 1) ? 0 : c;
        c++;
        t = x[n - 1], x[n - 1] = x[i], x[i] = t;
        if (callback) callback(x, nn);
```

```c
/* entry for Boothroyd method */
void perm2(int *x, int n, int callback(int*, int))
{
    if (callback) callback(x, n);
    boothroyd(x, n, n, callback);
}

/* same as perm2, but flattened recursions into iterations */
void perm3(int *x, int n, int callback(int*, int))
{
    /* calloc isn't strictly necessary, int c[32] would suffice
       for most practical purposes */
    int d, i, t, *c = calloc(n, sizeof(int));

    /* curiously, with GCC 4.6.1 -O3, removing next line makes
       it ~25% slower */
    if (callback) callback(x, n);
    for (d = 1; ; c[d]++) {
        while (d > 1) c[--d] = 0;
        while (c[d] >= d)
            if (++d >= n) goto done;

        t = x[ i = (d & 1) ? c[d] : 0 ], x[i] = x[d], x[d] = t;
        if (callback) callback(x, n);
    }
done:   free(c);
}

#define N 4

int main()
{
    int i, x[N];
    for (i = 0; i < N; i++) x[i] = i + 1;

    /* three different methods */
    perm1(x, N, show);
    perm2(x, N, show);
    perm3(x, N, show);

    return 0;
}
```

version 4

See lexicographic generation of permutations.

```c
#include <stdio.h>
#include <stdlib.h>

/* print a list of ints */
int show(int *x, int len)
{
    int i;
    for (i = 0; i < len; i++)
        printf("%d%c", x[i], i == len - 1 ? '\n' : ' ');
    return 1;
}

/* next lexicographical permutation */
int next_lex_perm(int *a, int n) {
#   define swap(i, j) {t = a[i]; a[i] = a[j]; a[j] = t;}
    int k, l, t;

    /* 1. Find the largest index k such that a[k] < a[k + 1]. If no such
          index exists, the permutation is the last permutation. */
    for (k = n - 1; k && a[k - 1] >= a[k]; k--);
    if (!k--) return 0;

    /* 2. Find the largest index l such that a[k] < a[l]. Since k + 1 is
```

```
    /* 3. Swap a[k] with a[l] */
    swap(k, l);

    /* 4. Reverse the sequence from a[k + 1] to the end */
    for (k++, l = n - 1; l > k; l--, k++)
        swap(k, l);
    return 1;
#   undef swap
}

void perm1(int *x, int n, int callback(int *, int))
{
    do {
        if (callback) callback(x, n);
    } while (next_lex_perm(x, n));
}

/* Boothroyd method; exactly N! swaps, about as fast as it gets */
void boothroyd(int *x, int n, int nn, int callback(int *, int))
{
    int c = 0, i, t;
    while (1) {
        if (n > 2) boothroyd(x, n - 1, nn, callback);
        if (c >= n - 1) return;

        i = (n & 1) ? 0 : c;
        c++;
        t = x[n - 1], x[n - 1] = x[i], x[i] = t;
        if (callback) callback(x, nn);
    }
}

/* entry for Boothroyd method */
void perm2(int *x, int n, int callback(int*, int))
{
    if (callback) callback(x, n);
    boothroyd(x, n, n, callback);
}

/* same as perm2, but flattened recursions into iterations */
void perm3(int *x, int n, int callback(int*, int))
{
    /* calloc isn't strictly necessary, int c[32] would suffice
       for most practical purposes */
    int d, i, t, *c = calloc(n, sizeof(int));

    /* curiously, with GCC 4.6.1 -O3, removing next line makes
       it ~25% slower */
    if (callback) callback(x, n);
    for (d = 1; ; c[d]++) {
        while (d > 1) c[--d] = 0;
        while (c[d] >= d)
            if (++d >= n) goto done;

        t = x[ i = (d & 1) ? c[d] : 0 ], x[i] = x[d], x[d] = t;
        if (callback) callback(x, n);
    }
done:   free(c);
}

#define N 4

int main()
{
    int i, x[N];
    for (i = 0; i < N; i++) x[i] = i + 1;

    /* three different methods */
    perm1(x, N, show);
    perm2(x, N, show);
    perm3(x, N, show);

    return 0;
}
```

C#

Recursive Linq

Works with: C# version 7

```
public static class Extension
{
    public static IEnumerable<IEnumerable<T>> Permutations<T>(this IEnumerable<T> values) where T : IComparable<T>
    {
        if (values.Count() == 1)
            return new[] { values };
        return values.SelectMany(v => Permutations(values.Where(x => x.CompareTo(v) != 0)), (v, p) => p.Prepend(v));
    }
}
```

Usage

```
Enumerable.Range(0,5).Permutations()
```

A recursive Iterator. Runs under C#2 (VS2005), i.e. no `var`, no lambdas,...

```
public class Permutations<T>
{
    public static System.Collections.Generic.IEnumerable<T[]> AllFor(T[] array)
    {
        if (array == null || array.Length == 0)
        {
            yield return new T[0];
        }
        else
        {
            for (int pick = 0; pick < array.Length; ++pick)
            {
                T item = array[pick];
                int i = -1;
                T[] rest = System.Array.FindAll<T>(
                    array, delegate(T p) { return ++i != pick; }
                );
                foreach (T[] restPermuted in AllFor(rest))
                {
                    i = -1;
                    yield return System.Array.ConvertAll<T, T>(
                        array,
                        delegate(T p) {
                            return ++i == 0 ? item : restPermuted[i - 1];
                        }
                    );
                }
            }
        }
    }
}
```

Usage:

```
namespace Permutations_On_RosettaCode
{
    class Program
    {
        static void Main(string[] args)
        {
            string[] list = "a b c d".Split();
            foreach (string[] permutation in Permutations<string>.AllFor(list))
```

```
        }
    }
}
```

Recursive version

```
using System;
class Permutations
{
  static int n = 4;
  static int [] buf = new int [n];
  static bool [] used = new bool [n];

  static void Main()
  {
    for (int i = 0; i < n; i++) used [i] = false;
    rec(0);
  }

  static void rec(int ind)
  {
    for (int i = 0; i < n; i++)
    {
      if (!used [i])
      {
        used [i] = true;
        buf [ind] = i;
    if (ind + 1 < n) rec(ind + 1);
        else Console.WriteLine(string.Join(",", buf));
    used [i] = false;
      }
    }
  }
}
```

Alternate recursive version

```
using System;
class Permutations
{
  static int n = 4;
  static int [] buf = new int [n];
  static int [] next = new int [n+1];

  static void Main()
  {
    for (int i = 0; i < n; i++) next [i] = i + 1;
    next[n] = 0;
    rec(0);
  }

  static void rec(int ind)
  {
    for (int i = n; next[i] != n; i = next[i])
    {
      buf [ind] = next[i];
      next[i]=next[next[i]];
      if (ind < n - 1) rec(ind + 1);
      else Console.WriteLine(string.Join(",", buf));
      next[i] = buf [ind];
    }
  }
}
```

```
// Always returns the same array which is the one passed to the function
public static IEnumerable<T[]> HeapsPermutations<T>(T[] array)
{
    var state = new int[array.Length];

    yield return array;

    for (var i = 0; i < array.Length;)
    {
        if (state[i] < i)
        {
            var left = i % 2 == 0 ? 0 : state[i];
            var temp = array[left];
            array[left] = array[i];
            array[i] = temp;
            yield return array;
            state[i]++;
            i = 1;
        }
        else
        {
            state[i] = 0;
            i++;
        }
    }
}

// Returns a different array for each permutation
public static IEnumerable<T[]> HeapsPermutationsWrapped<T>(IEnumerable<T> items)
{
    var array = items.ToArray();
    return HeapsPermutations(array).Select(mutating =>
        {
            var arr = new T[array.Length];
            Array.Copy(mutating, arr, array.Length);
            return arr;
        });
}
```

C++

The C++ standard library provides for this in the form of `std::next_permutation` and `std::prev_permutation`.

```
#include <algorithm>
#include <string>
#include <vector>
#include <iostream>

template<class T>
void print(const std::vector<T> &vec)
{
    for (typename std::vector<T>::const_iterator i = vec.begin(); i != vec.end(); ++i)
    {
        std::cout << *i;
        if ((i + 1) != vec.end())
            std::cout << ",";
    }
    std::cout << std::endl;
}

int main()
{
    //Permutations for strings
    std::string example("Hello");
    std::sort(example.begin(), example.end());
    do {
        std::cout << example << '\n';
    } while (std::next_permutation(example.begin(), example.end()));
```

```
    another.push_back(1234);
    another.push_back(4321);
    another.push_back(1234);
    another.push_back(9999);

    std::sort(another.begin(), another.end());
    do {
        print(another);
    } while (std::next_permutation(another.begin(), another.end()));

    return 0;
}
```

Output:

```
Hello
Helol
Heoll
Hlelo
Hleol
Hlleo
Hlloe
Hloel
Hlole
Hoell
Holel
Holle
eHllo
eHlol
eHoll
elHlo
elHol
ellHo
elloH
eloHl
elolH
eoHll
eolHl
eollH
lHelo
lHeol
lHleo
lHloe
lHoel
lHole
leHlo
leHol
lelHo
leloH
leoHl
leolH
llHeo
llHoe
lleHo
lleoH
lloHe
lloeH
loHel
loHle
loeHl
loelH
lolHe
loleH
oHell
oHlel
oHlle
oeHll
oelHl
oellH
olHel
olHle
oleHl
```

```
1234,1234,4321,9999
1234,1234,9999,4321
1234,4321,1234,9999
1234,4321,9999,1234
1234,9999,1234,4321
1234,9999,4321,1234
4321,1234,1234,9999
4321,1234,9999,1234
4321,9999,1234,1234
9999,1234,1234,4321
9999,1234,4321,1234
9999,4321,1234,1234
```

Clojure

Library function

In an REPL:

```
user=> (require 'clojure.contrib.combinatorics)
nil
user=> (clojure.contrib.combinatorics/permutations [1 2 3])
((1 2 3) (1 3 2) (2 1 3) (2 3 1) (3 1 2) (3 2 1))
```

Explicit

Replacing the call to the combinatorics library function by its real implementation.

```
(defn- iter-perm [v]
  (let [len (count v),
    j (loop [i (- len 2)]
        (cond (= i -1) nil
          (< (v i) (v (inc i))) i
          :else (recur (dec i))))]
    (when j
      (let [vj (v j),
        l (loop [i (dec len)]
        (if (< vj (v i)) i (recur (dec i))))]
    (loop [v (assoc v j (v l) l vj), k (inc j), l (dec len)]
      (if (< k l)
        (recur (assoc v k (v l) l (v k)) (inc k) (dec l))
        v))))))


(defn- vec-lex-permutations [v]
  (when v (cons v (lazy-seq (vec-lex-permutations (iter-perm v))))))

(defn lex-permutations
  "Fast lexicographic permutation generator for a sequence of numbers"
  [c]
  (lazy-seq
   (let [vec-sorted (vec (sort c))]
     (if (zero? (count vec-sorted))
       (list [])
       (vec-lex-permutations vec-sorted)))))

(defn permutations
  "All the permutations of items, lexicographic by index"
  [items]
  (let [v (vec items)]
    (map #(map v %) (lex-permutations (range (count v))))))
```

```
(println (permutations [1 2 3]))
```

CoffeeScript

```
# Returns a copy of an array with the element at a specific position
# removed from it.
arrayExcept = (arr, idx) ->
    res = arr[0..]
    res.splice idx, 1
    res

# The actual function which returns the permutations of an array-like
# object (or a proper array).
permute = (arr) ->
    arr = Array::slice.call arr, 0
    return [[]] if arr.length == 0

    permutations = (for value,idx in arr
        [value].concat perm for perm in permute arrayExcept arr, idx)

    # Flatten the array before returning it.
    [].concat permutations...
```

This implementation utilises the fact that the permutations of an array could be defined recursively, with the fixed point being the permutations of an empty array.

Usage:

```
coffee> console.log (permute "123").join "\n"
1,2,3
1,3,2
2,1,3
2,3,1
3,1,2
3,2,1
```

Common Lisp

```
(defun permute (list)
  (if list
    (mapcan #'(lambda (x)
        (mapcar #'(lambda (y) (cons x y))
            (permute (remove x list))))
        list)
    '(())))  ; else

(print (permute '(A B Z)))
```

Output:

```
((A B Z) (A Z B) (B A Z) (B Z A) (Z A B) (Z B A))
```

Lexicographic next permutation:

```
(defun next-perm (vec cmp)  ; modify vector
  (declare (type (simple-array * (*)) vec))
  (macrolet ((el (i) `(aref vec ,i))
             (cmp (i j) `(funcall cmp (el ,i) (el ,j))))
```

```
         (loop for k from len downto i
            when (cmp i k) do
              (rotatef (el i) (el k))
              (setf k (1+ len))
              (loop while (< (incf i) (decf k)) do
                   (rotatef (el i) (el k)))
              (return-from next-perm vec)))))

;;; test code
(loop for a = "1234" then (next-perm a #'char<) while a do
     (write-line a))
```

Recursive implementation of Heap's algorithm:

```
(defun heap-permutations (seq)
  (let ((permutations nil))
    (labels ((permute (seq k)
           (if (= k 1)
           (push seq permutations)
           (progn
             (permute seq (1- k))
             (loop for i from 0 below (1- k) do
              (if (evenp k)
                  (rotatef (elt seq i) (elt seq (1- k)))
                  (rotatef (elt seq 0) (elt seq (1- k))))
              (permute seq (1- k)))))))
      (permute seq (length seq))
      permutations)))
```

Crystal

```
puts [1, 2, 3].permutations
```

Output:

```
[[1, 2, 3], [1, 3, 2], [2, 1, 3], [2, 3, 1], [3, 1, 2], [3, 2, 1]]
```

Curry

```
insert :: a -> [a] -> [a]
insert x xs  = x : xs
insert x (y:ys) = y : insert x ys

permutation :: [a] -> [a]
permutation []     = []
permutation (x:xs) = insert x $ permutation xs
```

D

Simple Eager version

Compile with -version=permutations1_main to see the output.

```
T[][] permutations(T)(T[] items) pure nothrow {
    T[][] result;
```

```
                perms(s[0 .. i] ~ s[i+1 .. $], prefix ~ c);
        else
            result ~= prefix;
    }

    perms(items);
    return result;
}

version (permutations1_main) {
    void main() {
        import std.stdio;
        writefln("%(%s\n%)", [1, 2, 3].permutations);
    }
}
```

Output:

```
[1, 2, 3]
[1, 3, 2]
[2, 1, 3]
[2, 3, 1]
[3, 1, 2]
[3, 2, 1]
```

Fast Lazy Version

Compiled with `-version=permutations2_main` produces its output.

```
import std.algorithm, std.conv, std.traits;

struct Permutations(bool doCopy=true, T) if (isMutable!T) {
    private immutable size_t num;
    private T[] items;
    private uint[31] indexes;
    private ulong tot;

    this (T[] items) pure nothrow @safe @nogc
    in {
        static enum string L = indexes.length.text;
        assert(items.length >= 0 && items.length <= indexes.length,
               "Permutations: items.length must be >= 0 && < " ~ L);
    } body {
        static ulong factorial(in size_t n) pure nothrow @safe @nogc {
            ulong result = 1;
            foreach (immutable i; 2 .. n + 1)
                result *= i;
            return result;
        }

        this.num = items.length;
        this.items = items;
        foreach (immutable i; 0 .. cast(typeof(indexes[0]))this.num)
            this.indexes[i] = i;
        this.tot = factorial(this.num);
    }

    @property T[] front() pure nothrow @safe {
        static if (doCopy) {
            return items.dup;
        } else
            return items;
    }

    @property bool empty() const pure nothrow @safe @nogc {
        return tot == 0;
    }
```

```
        foreach (immutable x; 1 .. items.length + 1)
            result *= x;
        return result;
    }

    void popFront() pure nothrow @safe @nogc {
        tot--;
        if (tot > 0) {
            size_t j = num - 2;

            while (indexes[j] > indexes[j + 1])
                j--;
            size_t k = num - 1;
            while (indexes[j] > indexes[k])
                k--;
            swap(indexes[k], indexes[j]);
            swap(items[k], items[j]);

            size_t r = num - 1;
            size_t s = j + 1;
            while (r > s) {
                swap(indexes[s], indexes[r]);
                swap(items[s], items[r]);
                r--;
                s++;
            }
        }
    }
}

Permutations!(doCopy,T) permutations(bool doCopy=true, T)
                                    (T[] items)
pure nothrow if (isMutable!T) {
    return Permutations!(doCopy, T)(items);
}

version (permutations2_main) {
    void main() {
        import std.stdio, std.bigint;
        alias B = BigInt;
        foreach (p; [B(1), B(2), B(3)].permutations)
            assert((p[0] + 1) > 0);
        [1, 2, 3].permutations!false.writeln;
        [B(1), B(2), B(3)].permutations!false.writeln;
    }
}
```

Standard Version

```
void main() {
    import std.stdio, std.algorithm;

    auto items = [1, 2, 3];
    do
        items.writeln;
    while (items.nextPermutation);
}
```

Delphi

```
program TestPermutations;

{$APPTYPE CONSOLE}

type
  TItem = Integer;                // declare ordinal type for array item
  TArray = array[0..3] of TItem;
```

```
procedure Permutation(K: Integer; var A: TArray);
var
  I, J: Integer;
  Tmp: TItem;

begin
  for I:= Low(A) + 1 to High(A) + 1 do begin
    J:= K mod I;
    Tmp:= A[J];
    A[J]:= A[I - 1];
    A[I - 1]:= Tmp;
    K:= K div I;
  end;
end;

var
  A: TArray;
  I, K, Count: Integer;
  S, S1, S2: ShortString;

begin
  Count:= 1;
  I:= Length(A);
  while I > 1 do begin
    Count:= Count * I;
    Dec(I);
  end;

  S:= '';
  for K:= 0 to Count - 1 do begin
    A:= Source;
    Permutation(K, A);
    S1:= '';
    for I:= Low(A) to High(A) do begin
      Str(A[I]:1, S2);
      S1:= S1 + S2;
    end;
    S:= S + '  ' + S1;
    if Length(S) > 40 then begin
      Writeln(S);
      S:= '';
    end;
  end;

  if Length(S) > 0 then Writeln(S);
  Readln;
end.
```

Output:

```
  4123  4213  4312  4321  4132  4231  3421
  3412  2413  1423  2431  1432  3142  3241
  2341  1342  2143  1243  3124  3214  2314
  1324  2134  1234
```

EasyLang

```
proc permlist k . list[] .
   if k = len list[]
      print list[]
      return
   .
   for i = k to len list[]
      swap list[i] list[k]
      permlist k + 1 list[]
      swap list[k] list[i]
   .
.
```

```
l[] = [ 1 2 3 ]
permlist 1 l[]
```

Ecstasy

```
/**
 * Implements permutations without repetition.
 */
module Permutations {
    static Int[][] permut(Int items) {
        if (items <= 1) {
            // with one item, there is a single permutation; otherwise there are no permutations
            return items == 1 ? [[0]] : [];
        }

        // the "pattern" for all values but the first value in each permutation is
        // derived from the permutations of the next smaller number of items
        Int[][] pattern = permut(items - 1);

        // build the list of all permutations for the specified number of items by iterating only
        // the first digit
        Int[][] result = new Int[][];
        for (Int prefix : 0 ..< items) {
            for (Int[] suffix : pattern) {
                result.add(new Int[items](i -> i == 0 ? prefix : (prefix + suffix[i-1] + 1) % items));
            }
        }
        return result;
    }

    void run() {
        @Inject Console console;
        console.print($"permut(3) = {permut(3)}");
    }
}
```

Output:

```
permut(3) = [[0, 1, 2], [0, 2, 1], [1, 2, 0], [1, 0, 2], [2, 0, 1], [2, 1, 0]]
```

EDSAC order code

Uses two subroutines which respectively (1) Generate the first permutation in lexicographic order; (2) Return the next permutation in lexicographic order, or set a flag to indicate there are no more permutations. The algorithm for (2) is the same as in the Wikipedia article "Permutation".

```
[Permutations task for Rosetta Code.]
[EDSAC program, Initial Orders 2.]

    T51K P200F [G parameter: start address of subroutines]
    T47K P100F [M parameter: start address of main routine]

[====================== G parameter: Subroutines ======================]
            E25K TG GK
[Constants used in the subroutines]
      [0]   AF  [add to address to make A order for that address]
      [1]   SF  [add to address to make S order for that address]
      [2]   UF  [(1) add to address to make U order for that address]
                [(2) subtract from S order to make T order, same address]
      [3]   OF  [add to A order to make T order, same address]

[------------------------------------------------------------
Subroutine to initialize an array of n short (17-bit) words
```

```
      [4]   A3F     [plant return link as usual]
            T19@
            A4F     [address of array]
            A2@     [make U order for that address]
            T1F     [store U order in 1F]
            A5F     [load n = number of elements (in address field)]
            S2F     [make n-1]
[Start of loop; works backwards, n-1 to 0]
     [11]   UF      [store array element in 0F]
            A1F     [make order to store element in array]
            T15@    [plant that order in code]
            AF      [pick up element fron 0F]
     [15]   UF      [(planted) store element in array]
            S2F     [dec to next element]
            E11@    [loop if still >= 0]
            TF      [clear acc. before return]
     [19]   ZF      [overwritten by jump back to caller]

[--------------------------------------------------------------------
Subroutine to get next permutation in lexicographic order.
Uses same 4-step algorithm as Wikipedia article "Permutations",
but notation in comments differs from that in Wikipedia.
Parameters: 4F = address of array; 5F = n = length of array.
            0F is returned as 0 for success, < 0 if passed-in
               permutation is the last.
Workspace: 0F, 1F.]
     [20]   A3F     [plant return link as usual]
            T103@

[Step 1: Find the largest index k such that a{k} > a{k-1}.
 If no such index exists, the passed-in permutation is the last.]
            A4F     [load address of a{0}]
            A@      [make A order for a{0}]
            U1F     [store as test for end of loop]
            A5F     [make A order for a{n}]
            U96@    [plant in code below]
            S2F     [make A order for a{n-1}]
            T43@    [plant in code below]
            A4F     [load address of a{0}]
            A5F     [make address of a{n}]
            A1@     [make S order for a{n}]
            T44@    [plant in code below]
[Start of loop for comparing a{k} with a{k-1}]
     [33]   TF      [clear acc]
            A43@    [load A order for a{k}]
            S2F     [make A order for a{k-1}]
            S1F     [tested all yet?]
            G102@   [if yes, jump to failed (no more permutations)]
            A1F     [restore accumulator after test]
            T43@    [plant updated A order]
            A44@    [dec address in S order]
            S2F
            T44@
     [43]   SF      [(planted) load a{k-1}]
     [44]   AF      [(planted) subtract a{k}]
            E33@    [loop back if a{k-1} > a{k}]

[Step 2: Find the largest index j >= k such that a{j} > a{k-1}.
 Such an index j exists, because j = k is an instance.]
            TF      [clear acc]
            A4F     [load address of a{0}]
            A5F     [make address of a{n}]
            A1@     [make S order for a{n}]
            T1F     [save as test for end of loop]
            A44@    [load S order for a{k}]
            T64@    [plant in code below]
            A43@    [load A order for a{k-1}]
            T63@    [plant in code below]
[Start of loop]
     [55]   TF      [clear acc]
            A64@    [load S order for a{j} (initially j = k)]
            U75@    [plant in code below]
            A2F     [inc address (in effect inc j)]
            S1F     [test for end of array]
            E66@    [jump out if so]
```

```
     [64]   SF      [(planted) subtract a{j}]
            G55@    [loop back if a{j} still > a{k-1}]
     [66]
[Step 3: Swap a{k-1} and a{j}]
            TF      [clear acc]
            A63@    [load A order for a{k-1}]
            U77@    [plant in code below, 2 places]
            U94@
            A3@     [make T order for a{k-1}]
            T80@    [plant in code below]
            A75@    [load S order for a{j}]
            S2@     [make T order for a{j}]
            T78@    [plant in code below]
     [75]   SF      [(planted) load -a{j}]
            TF      [park -a{j} in 0F]
     [77]   AF      [(planted) load a{k-1}]
     [78]   TF      [(planted) store a{j}]
            SF      [load a{j} by subtracting -a{j}]
     [80]   TF      [(planted) store in a{k-1}]

[Step 4: Now a{k}, ..., a{n-1} are in decreasing order.
 Change to increasing order by repeated swapping.]
     [81]   A96@    [counting down from a{n} (exclusive end of array)]
            S2F     [make A order for a{n-1}]
            U96@    [plant in code]
            A3@     [make T order for a{n-1}]
            T99@    [plant]
            A94@    [counting up from a{k-1} (exclusive)]
            A2F     [make A order for a{k}]
            U94@    [plant]
            A3@     [make T order for a{k}]
            U97@    [plant]
            S99@    [swapped all yet?]
            E101@   [if yes, jump to exit from subroutine]
[Swapping two array elements, initially a{k} and a{n-1}]
            TF      [clear acc]
     [94]   AF      [(planted) load 1st element]
            TF      [park in 0F]
     [96]   AF      [(planted) load 2nd element]
     [97]   TF      [(planted) copy to 1st element]
            AF      [load old 1st element]
     [99]   TF      [(planted) copy to 2nd element]
            E81@    [always loop back]
    [101]   TF      [done, return 0 in location 0F]
    [102]   TF      [return status to caller in 0F; also clears acc]
    [103]   ZF      [(planted) jump back to caller]

[==================== M parameter: Main routine ==================]
[Prints all 120 permutations of the letters in 'EDSAC'.]
            E25K TM GK
[Constants used in the main routine]
      [0]   P900F   [address of permutation array]
      [1]   P5F     [number of elements in permutation (in address field)]
[Array of letters in 'EDSAC', in alphabetical order]
      [2]   AF CF DF EF SF
      [7]   O2@     [add to index to make O order for letter in array]
      [8]   P12F    [permutations per printed line (in address field)]
      [9]   AF      [add to address to make A order for that address]
[Teleprinter characters]
     [10]   K2048F  [set letters mode]
     [11]   !F      [space]
     [12]   @F      [carriage return]
     [13]   &F      [line feed]
     [14]   K4096F  [null]

[Entry point, with acc = 0.]
     [15]   O10@    [set teleprinter to letters]
            S8@     [intialize -ve count of permutations per line]
            T7F     [keep count in 7F]
            A@      [pass address of permutation array in 4F]
            T4F
            A1@     [pass number of elements in 5F]
            T5F
     [22]   A22@    [call subroutine to initialize permutation array]
            G4G
```

```
            T29@    [plant in code]
            S5F     [initialize -ve count of array elements]
     [28]   T6F     [keep count in 6F]
     [29]   AF      [(planted) load permutation element]
            A7@     [make order to print letter from table]
            T32@    [plant in code]
     [32]   OF      [(planted) print letter from table]
            A29@    [inc address in permutation array]
            A2F
            T29@
            A6F     [inc -ve count of array elements]
            A2F
            G28@    [loop till count becomes 0]
            A7F     [inc -ve count of perms per line]
            A2F
            E44@    [jump if end of line]
            O11@    [else print a space]
            G47@    [join common code]
     [44]   O12@    [print CR]
            O13@    [print LF]
            S8@
     [47]   T7F     [update -ve count of permutations in line]
     [48]   A48@    [call subroutine for next permutation (if any)]
            G20G
            AF      [test 0F: got a new permutation?]
            E24@    [if so, loop to print it]
            O14@    [no more, output null to flush teleprinter buffer]
            ZF      [halt program]
            E15Z    [define entry point]
            PF      [enter with acc = 0]
[end]
```

Output:

```
ACDES ACDSE ACEDS ACESD ACSDE ACSED ADCES ADCSE ADECS ADESC ADSCE ADSEC
AECDS AECSD AEDCS AEDSC AESCD AESDC ASCDE ASCED ASDCE ASDEC ASECD ASEDC
CADES CADSE CAEDS CAESD CASDE CASED CDAES CDASE CDEAS CDESA CDSAE CDSEA
CEADS CEASD CEDAS CEDSA CESAD CESDA CSADE CSAED CSDAE CSDEA CSEAD CSEDA
DACES DACSE DAECS DAESC DASCE DASEC DCAES DCASE DCEAS DCESA DCSAE DCSEA
DEACS DEASC DECAS DECSA DESAC DESCA DSACE DSAEC DSCAE DSCEA DSEAC DSECA
EACDS EACSD EADCS EADSC EASCD EASDC ECADS ECASD ECDAS ECDSA ECSAD ECSDA
EDACS EDASC EDCAS EDCSA EDSAC EDSCA ESACD ESADC ESCAD ESCDA ESDAC ESDCA
SACDE SACED SADCE SADEC SAECD SAEDC SCADE SCAED SCDAE SCDEA SCEAD SCEDA
SDACE SDAEC SDCAE SDCEA SDEAC SDECA SEACD SEADC SECAD SECDA SEDAC SEDCA
```

Eiffel

```
class
    APPLICATION

create
    make

feature {NONE}

    make
        do
            test := <<2, 5, 1>>
            permute (test, 1)
        end

    test: ARRAY [INTEGER]

    permute (a: ARRAY [INTEGER]; k: INTEGER)
            -- All permutations of 'a'.
        require
            count_positive: a.count > 0
            k_valid_index: k > 0
        local
```

```
                across
                    a as ar
                loop
                    io.put_integer (ar.item)
                end
                io.new_line
            else
                across
                    k |..| a.count as c
                loop
                    t := a [k]
                    a [k] := a [c.item]
                    a [c.item] := t
                    permute (a, k + 1)
                    t := a [k]
                    a [k] := a [c.item]
                    a [c.item] := t
                end
            end
        end

end
```

Output:

```
251
215
521
512
152
125
```

Elixir

Translation of: Erlang

```
defmodule RC do
  def permute([]), do: [[]]
  def permute(list) do
    for x <- list, y <- permute(list -- [x]), do: [x|y]
  end
end

IO.inspect RC.permute([1, 2, 3])
```

Output:

```
[[1, 2, 3], [1, 3, 2], [2, 1, 3], [2, 3, 1], [3, 1, 2], [3, 2, 1]]
```

Erlang

Shortest form:

```
-module(permute).
-export([permute/1]).

permute([]) -> [[]];
permute(L) -> [[X|Y] || X<-L, Y<-permute(L--[X])].
```

Y-combinator (for shell):

```
F = fun(L) -> G = fun(_, []) -> [[]]; (F, L) -> [[X|Y] || X<-L, Y<-F(F, L--[X])] end, G(G, L) end.
```

More efficient zipper implementation:

```
-module(permute).

-export([permute/1]).

permute([]) -> [[]];
permute(L) -> zipper(L, [], []).

% Use zipper to pick up first element of permutation
zipper([], _, Acc) -> lists:reverse(Acc);
zipper([H|T], R, Acc) ->
  % place current member in front of all permutations
  % of rest of set - both sides of zipper
  prepend(H, permute(lists:reverse(R, T)),
    % pass zipper state for continuation
    T, [H|R], Acc).

prepend(_, [], T, R, Acc) -> zipper(T, R, Acc); % continue in zipper
prepend(X, [H|T], ZT, ZR, Acc) -> prepend(X, T, ZT, ZR, [[X|H]|Acc]).
```

Demonstration (escript):

```
main(_) -> io:fwrite("~p~n", [permute:permute([1,2,3])]).
```

Output:

```
[[1,2,3],[1,3,2],[2,1,3],[2,3,1],[3,1,2],[3,2,1]]
```

Euphoria

Translation of: PureBasic

```
function reverse(sequence s, integer first, integer last)
    object x
    while first < last do
        x = s[first]
        s[first] = s[last]
        s[last] = x
        first += 1
        last -= 1
    end while
    return s
end function

function nextPermutation(sequence s)
    integer pos, last
    object x
    if length(s) < 1 then
        return 0
    end if

    pos = length(s)-1
    while compare(s[pos], s[pos+1]) >= 0 do
        pos -= 1
        if pos < 1 then
            return -1
        end if
    end while

    last = length(s)
```

```
    x = s[pos]
    s[pos] = s[last]
    s[last] = x

    return reverse(s, pos+1, length(s))
end function

object s
s = "abcd"
puts(1, s & '\t')
while 1 do
    s = nextPermutation(s)
    if atom(s) then
        exit
    end if
    puts(1, s & '\t')
end while
```

Output:

```
abcd    abdc    acbd    acdb    adbc    adcb    bacd    badc    bcad    bcda
bdac    bdca    cabd    cadb    cbad    cbda    cdab    cdba    dabc    dacb
dbac    dbca    dcab    dcba
```

F#

```
let rec insert left x right = seq {
    match right with
    | [] -> yield left @ [x]
    | head :: tail ->
        yield left @ [x] @ right
        yield! insert (left @ [head]) x tail
    }

let rec perms permute =
    seq {
        match permute with
        | [] -> yield []
        | head :: tail -> yield! Seq.collect (insert [] head) (perms tail)
    }

[<EntryPoint>]
let main argv =
    perms (Seq.toList argv)
    |> Seq.iter (fun x -> printf "%A\n" x)
    0
```

```
>RosettaPermutations 1 2 3
["1"; "2"; "3"]
["2"; "1"; "3"]
["2"; "3"; "1"]
["1"; "3"; "2"]
["3"; "1"; "2"]
["3"; "2"; "1"]
```

Translation of Haskell "insertion-based approach" (last version)

```
let permutations xs =
    let rec insert x = function
        | [] -> [[x]]
        | head :: tail -> (x :: (head :: tail)) :: (List.map (fun l -> head :: l) (insert x tail))
    List.fold (fun s e -> List.collect (insert e) s) [[]] xs
```

The all-permutations word is part of factor's standard library. See http://docs.factorcode.org/content/word-all-permutations,math.combinatorics.html

Fortran

```
program permutations

  implicit none
  integer, parameter :: value_min = 1
  integer, parameter :: value_max = 3
  integer, parameter :: position_min = value_min
  integer, parameter :: position_max = value_max
  integer, dimension (position_min : position_max) :: permutation

  call generate (position_min)

contains

  recursive subroutine generate (position)

    implicit none
    integer, intent (in) :: position
    integer :: value

    if (position > position_max) then
      write (*, *) permutation
    else
      do value = value_min, value_max
        if (.not. any (permutation (: position - 1) == value)) then
          permutation (position) = value
          call generate (position + 1)
        end if
      end do
    end if

  end subroutine generate

end program permutations
```

Output:

```
           1           2           3
           1           3           2
           2           1           3
           2           3           1
           3           1           2
           3           2           1
```

Alternate solution

Instead of looking up unused values, this program starts from [1, ..., n] and does only swaps, hence the array always represents a valid permutation. The values need to be "swapped back" after the recursive call.

```
program allperm
    implicit none
    integer :: n, i
    integer, allocatable :: a(:)
    read *, n
    allocate(a(n))
    a = [ (i, i = 1, n) ]
    call perm(1)
    deallocate(a)
contains
    recursive subroutine perm(i)
```

```
        else
            do j = i, n
                t = a(i)
                a(i) = a(j)
                a(j) = t
                call perm(i + 1)
                t = a(i)
                a(i) = a(j)
                a(j) = t
            end do
        end if
    end subroutine
end program
```

Fortran Speed Test

So ... what is the fastest algorithm?

Here below is the speed test for a couple of algorithms of permutation. We can add more algorithms into this frame-work. When they work in the same circumstance, we can see which is the fastest one.

```
   program testing_permutation_algorithms

   implicit none
   integer :: nmax
   integer, dimension(:),allocatable :: ida
   logical :: mtc
   logical :: even
   integer :: i
   integer(8) :: ic
   integer :: clock_rate, clock_max, t1, t2
   real(8) :: dt
   integer :: pos_min, pos_max
!
!
!  Beginning:
!
   write(*,*) 'INPUT N:'
   read *, nmax
   write(*,*) 'N =', nmax
   allocate ( ida(1:nmax) )
!
!
!  (1) Starting:
!
   do i  =  1, nmax
      ida(i) = i
   enddo
!
   ic = 0
   call system_clock ( t1, clock_rate, clock_max )
!
   mtc = .false.
!
   do
      call subnexper ( nmax, ida, mtc, even )
!
!     1) counting the number of permutatations
!
      ic = ic + 1
!
!     2) writing out the result:
!
!     do i  =  1, nmax
!        write (100,"(i3,',')",advance = "no") ida(i)
!     enddo
```

```
!
      if (mtc) then
         cycle
      else
         exit
      endif
!
   enddo
!
   call system_clock ( t2, clock_rate, clock_max )
   dt =  ( dble(t2) - dble(t1) )/ dble(clock_rate)
!
!  Finishing (1)
!
   write(*,*) "1) subnexper:"
   write(*,*) 'Total permutations :', ic
   write(*,*) 'Total time elapsed :', dt
!
!
!  (2) Starting:
!
   do i  =  1, nmax
      ida(i) = i
   enddo
!
   pos_min = 1
   pos_max = nmax
!
   ic = 0
   call system_clock ( t1, clock_rate, clock_max )
!
   call generate ( pos_min )
!
   call system_clock ( t2, clock_rate, clock_max )
   dt =  ( dble(t2) - dble(t1) )/ dble(clock_rate)
!
!  Finishing (2)
!
   write(*,*) "2) generate:"
   write(*,*) 'Total permutations :', ic
   write(*,*) 'Total time elapsed :', dt
!
!
!  (3) Starting:
!
   do i  =  1, nmax
      ida(i) = i
   enddo
!
   ic = 0
   call system_clock ( t1, clock_rate, clock_max )
!
   i = 1
   call perm ( i )
!
   call system_clock ( t2, clock_rate, clock_max )
   dt =  ( dble(t2) - dble(t1) )/ dble(clock_rate)
!
!  Finishing (3)
!
   write(*,*) "3) perm:"
   write(*,*) 'Total permutations :', ic
   write(*,*) 'Total time elapsed :', dt
!
!
!  (4) Starting:
!
   do i  =  1, nmax
      ida(i) = i
   enddo
!
   ic = 0
   call system_clock ( t1, clock_rate, clock_max )
!
   do
```

```
      ic = ic + 1
!
!     2) writing out the result:
!
!     do i  =  1, nmax
!        write (100,"(i3,',')",advance = "no") ida(i)
!     enddo
!     write(100,*)
!
!     repeat if not being finished yet, otherwise exit.
!
      if ( nextp(nmax,ida) ) then
         cycle
      else
         exit
      endif
!
   enddo
!
   call system_clock ( t2, clock_rate, clock_max )
   dt =  ( dble(t2) - dble(t1) )/ dble(clock_rate)
!
!  Finishing (4)
!
   write(*,*) "4) nextp:"
   write(*,*) 'Total permutations :', ic
   write(*,*) 'Total time elapsed :', dt
!
!
!  What's else?
!  ...
!
!==
   deallocate(ida)
!
   stop
!==
   contains
!==
!     Modified version of SUBROUTINE NEXPER from the book of
!     Albert Nijenhuis and Herbert S. Wilf, "Combinatorial
!     Algorithms For Computers and Calculators", 2nd Ed, p.59.
!
      subroutine subnexper ( n, a, mtc, even )
      implicit none
      integer,intent(in)    ::  n
      integer,dimension(n),intent(inout)  :: a
      logical,intent(inout) :: mtc, even
!
!     local varialbes:
!
      integer,save :: nm3
      integer :: ia, i, s, d, i1, l, j, m
!
      if (mtc) goto 10

      nm3 = n-3

      do i = 1,n
         a(i) = i
      enddo

      mtc  = .true.
5     even = .true.

      if ( n .eq. 1 ) goto 8

6     if ( a(n) .ne. 1 .or. a(1) .ne. 2+mod(n,2) ) return

      if ( n .le. 3 ) goto 8

      do i = 1,nm3
         if( a(i+1) .ne. a(i)+1 ) return
      enddo
```

```
10    if ( n .eq. 1 ) goto 27

      if( .not. even ) goto 20

      ia   = a(1)
      a(1) = a(2)
      a(2) = ia
      even = .false.

      goto 6

20    s = 0

      do i1 = 2,n
         ia = a(i1)
         i = i1-1
         d = 0
         do j = 1,i
            if ( a(j) .gt. ia ) d = d+1
         enddo
         s = d+s
         if ( d .ne. i*mod(s,2) ) goto 35
      enddo

27    a(1) = 0

      goto 8

35    m = mod(s+1,2)*(n+1)

      do j = 1,i
         if(isign(1,a(j)-ia) .eq. isign(1,a(j)-m)) cycle
         m = a(j)
         l = j
      enddo

      a(l) = ia
      a(i1) = m
      even = .true.

      return
      end subroutine
!=====
!
!     http://rosettacode.org/wiki/Permutations#Fortran
!
      recursive subroutine generate (pos)

      implicit none
      integer,intent(in) :: pos
      integer :: val

      if (pos > pos_max) then
!
!        1) counting the number of permutatations
!
         ic = ic + 1
!
!        2) writing out the result:
!
!        write (*,*) permutation
!
      else
         do val = 1, nmax
            if (.not. any (ida( : pos-1) == val)) then
               ida(pos) = val
               call generate (pos + 1)
            endif
         enddo
      endif

      end subroutine
!=====
!
```

```
      implicit none
      integer,intent(inout) :: i
!
      integer :: j, t, ip1
!
      if (i == nmax) then
!
!        1) couting the number of permutatations
!
         ic = ic + 1
!
!        2) writing out the result:
!
!        write (*,*) a
!
      else
         ip1 = i+1
         do j = i, nmax
            t = ida(i)
            ida(i) = ida(j)
            ida(j) = t
            call perm ( ip1 )
            t = ida(i)
            ida(i) = ida(j)
            ida(j) = t
         enddo
      endif
      return
      end subroutine
!=====
!
!     http://rosettacode.org/wiki/Permutations#Fortran
!
      function nextp ( n, a )
      logical :: nextp
      integer,intent(in) :: n
      integer,dimension(n),intent(inout) :: a
!
!     local variables:
!
      integer i,j,k,t
!
      i = n-1
   10 if ( a(i) .lt. a(i+1) ) goto 20
      i = i-1
      if ( i .eq. 0 ) goto 20
      goto 10
   20 j = i+1
      k = n
   30 t = a(j)
      a(j) = a(k)
      a(k) = t
      j = j+1
      k = k-1
      if ( j .lt. k ) goto 30
      j = i
      if (j .ne. 0 ) goto 40
!
      nextp = .false.
!
      return
!
   40 j = j+1
      if ( a(j) .lt. a(i) ) goto 40
      t = a(i)
      a(i) = a(j)
      a(j) = t
!
      nextp = .true.
!
      return
      end function
!=====
!
!     What's else ?
```

```
!=====
   end program
```

An example of performance:

1) Compiled with GNU fortran compiler:

gfortran -O3 testing_permutation_algorithms.f90 ; ./a.out

```
INPUT N:
```

10

```
N =          10
1) subnexper:
Total permutations :              3628800
Total time elapsed :   4.9000000000000002E-002
2) generate:
Total permutations :              3628800
Total time elapsed :  0.84299999999999997
3) perm:
Total permutations :              3628800
Total time elapsed :   5.6000000000000001E-002
4) nextp:
Total permutations :              3628800
Total time elapsed :   2.9999999999999999E-002
```

b) Compiled with Intel compiler:

ifort -O3 testing_permutation_algorithms.f90 ; ./a.out

INPUT N: 10

```
N =          10
1) subnexper:
Total permutations :               3628800
Total time elapsed :  8.2400000000000000E-002
2) generate:
Total permutations :               3628800
Total time elapsed :  0.616200000000000
3) perm:
Total permutations :               3628800
Total time elapsed :  5.7600000000000000E-002
4) nextp:
Total permutations :               3628800
Total time elapsed :  3.6000000000000000E-002
```

So far, we have conclusion from the above performance: 1) subnexper is the 3rd fast with ifort and the 2nd with gfortran. 2) generate is the slowest one with not only ifort but gfortran. 3) perm is the 2nd fast one with ifort and the 3rd one with gfortran. 4) nextp is the fastest one with both ifort and gfortran (the winner in this test).

Note: It is worth mentioning that the performance of this test is dependent not only on algorithm, but also on computer where the test runs. Therefore we should run the test on our own computer and make conclusion by ourselves.

Fortran 77

Translation of: Ada

```
      program nptest
      integer n,i,a
      logical nextp
      external nextp
      parameter(n=4)
      dimension a(n)
      do i=1,n
      a(i)=i
      enddo
   10 print *,(a(i),i=1,n)
      if(nextp(n,a)) go to 10
      end

      function nextp(n,a)
      integer n,a,i,j,k,t
      logical nextp
      dimension a(n)
      i=n-1
   10 if(a(i).lt.a(i+1)) go to 20
      i=i-1
      if(i.eq.0) go to 20
      go to 10
   20 j=i+1
      k=n
   30 t=a(j)
      a(j)=a(k)
      a(k)=t
      j=j+1
      k=k-1
      if(j.lt.k) go to 30
      j=i
      if(j.ne.0) go to 40
      nextp=.false.
      return
   40 j=j+1
      if(a(j).lt.a(i)) go to 40
      t=a(i)
      a(i)=a(j)
      a(j)=t
      nextp=.true.
      end
```

Ratfor 77

See RATFOR.

Frink

Frink's array class has built-in methods `permute[]` and `lexicographicPermute[]` which permute the elements of an array in reflected Gray code order and lexicographic order respectively.

```
a = [1,2,3,4]
println[formatTable[a.lexicographicPermute[]]]
```

Output:

```
1 2 3 4
1 2 4 3
1 3 2 4
1 3 4 2
1 4 2 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
```

FutureBasic

With recursion

Here's a sweet and short solution adapted from Robert Sedgewick's 'Algorithms' (1989, p. 628). It generates its own array of integers.

```
void local fn perm( k as Short)
static Short w( 4 ), i = -1
  Short j
  i ++ : w( k ) = i
  if i = 4
    for j = 1 to 4 : print w( j ),
    next : print
  else
    for j = 1 to 4 : if w( j ) = 0 then fn perm( j )
    next
  end if
  i -- : w( k ) = 0
end fn

fn perm(0)

handleevents
```

With iteration

We can also do it by brute force:

```
void local fn perm( w as CFStringRef )
  Short a, b, c, d
  for a = 0 to 3 : for b = 0 to 3 : for c = 0 to 3 : for d = 0 to 3
    if a != b and a != c and a != d and b != c and b != d and c != d
      print mid(w,a,1); mid(w,b,1); mid(w,c,1); mid(w,d,1)
    end if
  next : next : next : next
end fn

fn perm (@"abel")

handleevents
```

GAP

GAP can handle permutations and groups. Here is a straightforward implementation : for each permutation p in S(n) (symmetric group), compute the images of 1 .. n by p. As an alternative, List(SymmetricGroup(n)) would yield the permutations as GAP *Permutation* objects, which would probably be more manageable in later computations.

```
gap>List(SymmetricGroup(4), p -> Permuted([1 .. 4], p));
perms(4);
[ [ 1, 2, 3, 4 ], [ 4, 2, 3, 1 ], [ 2, 4, 3, 1 ], [ 3, 2, 4, 1 ], [ 1, 4, 3, 2 ], [ 4, 1, 3, 2 ], [ 2, 1, 3, 4 ],
  [ 3, 1, 4, 2 ], [ 1, 3, 4, 2 ], [ 4, 3, 1, 2 ], [ 2, 3, 1, 4 ], [ 3, 4, 1, 2 ], [ 1, 2, 4, 3 ], [ 4, 2, 1, 3 ],
  [ 2, 4, 1, 3 ], [ 3, 2, 1, 4 ], [ 1, 4, 2, 3 ], [ 4, 1, 2, 3 ], [ 2, 1, 4, 3 ], [ 3, 1, 2, 4 ], [ 1, 3, 2, 4 ],
  [ 4, 3, 2, 1 ], [ 2, 3, 4, 1 ], [ 3, 4, 2, 1 ] ]
```

GAP has also built-in functions to get permutations

```
# All arrangements of 4 elements in 1 .. 4
Arrangements([1 .. 4], 4);
# All permutations of 1 .. 4
PermutationsList([1 .. 4]);
```

Here is an implementation using a function to compute next permutation in lexicographic order:

```
NextPermutation := function(a)
   local i, j, k, n, t;
   n := Length(a);
   i := n - 1;
   while i > 0 and a[i] > a[i + 1] do
      i := i - 1;
   od;
   j := i + 1;
   k := n;
   while j < k do
      t := a[j];
      a[j] := a[k];
      a[k] := t;
      j := j + 1;
      k := k - 1;
   od;
   if i = 0 then
      return false;
   else
      j := i + 1;
      while a[j] < a[i] do
         j := j + 1;
      od;
      t := a[i];
      a[i] := a[j];
      a[j] := t;
      return true;
   fi;
end;

Permutations := function(n)
   local a, L;
   a := List([1 .. n], x -> x);
   L := [ ];
   repeat
      Add(L, ShallowCopy(a));
   until not NextPermutation(a);
   return L;
end;

Permutations(3);
[ [ 1, 2, 3 ], [ 1, 3, 2 ],
```

```
  [ 2, 1, 3 ], [ 2, 3, 1 ],
  [ 3, 1, 2 ], [ 3, 2, 1 ] ]
```

Glee

```
$$ n !! k    dyadic: Permutations for k out of n elements (in this case k = n)
$$ #s        monadic: number of elements in s
$$ ,,        monadic: expose with space-lf separators
$$ s[n]      index n of s

'Hello' 123 7.9 '•'=>s;
s[s# !! (s#)],,
```

Result:

```
Hello 123 7.9 •
Hello 123 • 7.9
Hello 7.9 123 •
Hello 7.9 • 123
Hello • 123 7.9
Hello • 7.9 123
123 Hello 7.9 •
123 Hello • 7.9
123 7.9 Hello •
123 7.9 • Hello
123 • Hello 7.9
123 • 7.9 Hello
7.9 Hello 123 •
7.9 Hello • 123
7.9 123 Hello •
7.9 123 • Hello
7.9 • Hello 123
7.9 • 123 Hello
• Hello 123 7.9
• Hello 7.9 123
• 123 Hello 7.9
• 123 7.9 Hello
• 7.9 Hello 123
• 7.9 123 Hello
```

GNU make

Recursive on unique elements

```
#delimiter should not occur inside elements
delimiter=;
#convert list to delimiter separated string
implode=$(subst $() $(),$(delimiter),$(strip $1))
#convert delimiter separated string to list
explode=$(strip $(subst $(delimiter), ,$1))
#enumerate all permutations and subpermutations
permutations0=$(if $1,$(foreach x,$1,$x $(addprefix $x$(delimiter),$(call permutations0,$(filter-out $x,$1)))),)
#remove subpermutations from permutations0 output
permutations=$(strip $(foreach x,$(call permutations0,$1),$(if $(filter $(words $1),$(words $(call explode,$x))),$(call
implode,$(call explode,$x)),)))

delimiter_separated_output=$(call permutations,a b c d)
$(info $(delimiter_separated_output))
```

Output:

```
a;b;c;d a;b;d;c a;c;b;d a;c;d;b a;d;b;c a;d;c;b b;a;c;d b;a;d;c b;c;a;d b;c;d;a b;d;a;c b;d;c;a c;a;b;d c;a;d;b c;b;a;d c;b;d;a
c;d;a;b c;d;b;a d;a;b;c d;a;c;b d;b;a;c d;b;c;a d;c;a;b d;c;b;a
```

Go

recursive

```go
package main

import "fmt"

func main() {
    demoPerm(3)
}

func demoPerm(n int) {
    // create a set to permute.  for demo, use the integers 1..n.
    s := make([]int, n)
    for i := range s {
        s[i] = i + 1
    }
    // permute them, calling a function for each permutation.
    // for demo, function just prints the permutation.
    permute(s, func(p []int) { fmt.Println(p) })
}

// permute function.  takes a set to permute and a function
// to call for each generated permutation.
func permute(s []int, emit func([]int)) {
    if len(s) == 0 {
        emit(s)
        return
    }
    // Steinhaus, implemented with a recursive closure.
    // arg is number of positions left to permute.
    // pass in len(s) to start generation.
    // on each call, weave element at pp through the elements 0..np-2,
    // then restore array to the way it was.
    var rc func(int)
    rc = func(np int) {
        if np == 1 {
            emit(s)
            return
        }
        np1 := np - 1
        pp := len(s) - np1
        // weave
        rc(np1)
        for i := pp; i > 0; i-- {
            s[i], s[i-1] = s[i-1], s[i]
            rc(np1)
        }
        // restore
        w := s[0]
        copy(s, s[1:pp+1])
        s[pp] = w
    }
    rc(len(s))
}
```

Output:

```
[1 2 3]
[1 3 2]
[3 1 2]
[2 1 3]
```

```
[2 3 1]
[3 2 1]
```

non-recursive, lexicographical order

```go
package main

import "fmt"

func main() {
        var a = []int{1, 2, 3}
        fmt.Println(a)
        var n = len(a) - 1
        var i, j int
        for c := 1; c < 6; c++ { // 3! = 6:
                i = n - 1
                j = n
                for a[i] > a[i+1] {
                        i--
                }
                for a[j] < a[i] {
                        j--
                }
                a[i], a[j] = a[j], a[i]
                j = n
                i += 1
                for i < j {
                        a[i], a[j] = a[j], a[i]
                        i++
                        j--
                }
                fmt.Println(a)
        }
}
```

Output:

```
[1 2 3]
[1 3 2]
[2 1 3]
[2 3 1]
[3 1 2]
[3 2 1]
```

Groovy

Solution:

```groovy
def makePermutations = { l -> l.permutations() }
```

Test:

```groovy
def list = ['Crosby', 'Stills', 'Nash', 'Young']
def permutations = makePermutations(list)
assert permutations.size() == (1..<(list.size()+1)).inject(1) { prod, i -> prod*i }
permutations.each { println it }
```

Output:

```
[Stills, Nash, Crosby, Young]
[Young, Stills, Crosby, Nash]
[Stills, Crosby, Nash, Young]
[Stills, Crosby, Young, Nash]
[Stills, Young, Nash, Crosby]
[Nash, Stills, Young, Crosby]
[Crosby, Young, Nash, Stills]
[Crosby, Nash, Young, Stills]
```

Haskell

```haskell
import Data.List (permutations)

main = mapM_ print (permutations [1,2,3])
```

A simple implementation, that assumes elements are unique and support equality:

```haskell
import Data.List (delete)

permutations :: Eq a => [a] -> [[a]]
permutations [] = [[]]
permutations xs = [ x:ys | x <- xs, ys <- permutations (delete x xs)]
```

A slightly more efficient implementation that doesn't have the above restrictions:

```haskell
permutations :: [a] -> [[a]]
permutations [] = [[]]
permutations xs = [ y:zs | (y,ys) <- select xs, zs <- permutations ys]
  where select []     = []
        select (x:xs) = (x,xs) : [ (y,x:ys) | (y,ys) <- select xs ]
```

The above are all selection-based approaches. The following is an insertion-based approach:

```haskell
permutations :: [a] -> [[a]]
permutations = foldr (concatMap . insertEverywhere) [[]]
  where insertEverywhere :: a -> [a] -> [[a]]
        insertEverywhere x [] = [[x]]
        insertEverywhere x l@(y:ys) = (x:l) : map (y:) (insertEverywhere x ys)
```

A serialized version:

Translation of: Mathematica

```haskell
import Data.Bifunctor (second)

permutations :: [a] -> [[a]]
permutations =
  let ins x xs n = uncurry (<>) $ second (x :) (splitAt n xs)
  in foldr
    ( \x a ->
        a >>= (fmap . ins x)
          <*> (enumFromTo 0 . length)
    )
    [[]]

main :: IO ()
main = print $ permutations [1, 2, 3]
```

Output:

```
[[1,2,3],[2,3,1],[3,1,2],[2,1,3],[1,3,2],[3,2,1]]
```

Icon and Unicon

```
procedure main(A)
    every p := permute(A) do every writes((!p||" ")|"\n")
end

procedure permute(A)
    if *A <= 1 then return A
    suspend [(A[1]<->A[i := 1 to *A])] ||| permute(A[2:0])
end
```

Output:

```
->permute Aardvarks eat ants
Aardvarks eat ants
Aardvarks ants eat
eat Aardvarks ants
eat ants Aardvarks
ants eat Aardvarks
ants Aardvarks eat
->
```

J

```
perms=: A.&i.~ !
```

Example use:

```
   perms 2
0 1
1 0
   ({~ perms@#)&.;: 'some random text'
some random text
some text random
random some text
random text some
text some random
text random some
```

Java

Using the code of Michael Gilleland.

```
public class PermutationGenerator {
    private int[] array;
    private int firstNum;
    private boolean firstReady = false;

    public PermutationGenerator(int n, int firstNum_) {
        if (n < 1) {
            throw new IllegalArgumentException("The n must be min. 1");
        }
        firstNum = firstNum_;
        array = new int[n];
        reset();
```

```
        for (int i = 0; i < array.length; i++) {
            array[i] = i + firstNum;
        }
        firstReady = false;
    }

    public boolean hasMore() {
        boolean end = firstReady;
        for (int i = 1; i < array.length; i++) {
            end = end && array[i] < array[i-1];
        }
        return !end;
    }

    public int[] getNext() {

        if (!firstReady) {
            firstReady = true;
            return array;
        }

        int temp;
        int j = array.length - 2;
        int k = array.length - 1;

        // Find largest index j with a[j] < a[j+1]

        for (;array[j] > array[j+1]; j--);

        // Find index k such that a[k] is smallest integer
        // greater than a[j] to the right of a[j]

        for (;array[j] > array[k]; k--);

        // Interchange a[j] and a[k]

        temp = array[k];
        array[k] = array[j];
        array[j] = temp;

        // Put tail end of permutation after jth position in increasing order

        int r = array.length - 1;
        int s = j + 1;

        while (r > s) {
            temp = array[s];
            array[s++] = array[r];
            array[r--] = temp;
        }

        return array;
    } // getNext()

    // For testing of the PermutationGenerator class
    public static void main(String[] args) {
        PermutationGenerator pg = new PermutationGenerator(3, 1);

        while (pg.hasMore()) {
            int[] temp =  pg.getNext();
            for (int i = 0; i < temp.length; i++) {
                System.out.print(temp[i] + " ");
            }
            System.out.println();
        }
    }

} // class
```

Output:

```
1 2 3
```

```
3 1 2
3 2 1
```

optimized

Following needs: Utils.java

```
public class Permutations {
    public static void main(String[] args) {
        System.out.println(Utils.Permutations(Utils.mRange(1, 3)));
    }
}
```

Output:

```
[[1, 2, 3], [1, 3, 2], [2, 1, 3], [2, 3, 1], [3, 1, 2], [3, 2, 1]]
```

JavaScript

ES5

Iteration

Copy the following as an HTML file and load in a browser.

```
<html><head><title>Permutations</title></head>
<body><pre id="result"></pre>
<script type="text/javascript">
var d = document.getElementById('result');

function perm(list, ret)
{
    if (list.length == 0) {
        var row = document.createTextNode(ret.join(' ') + '\n');
        d.appendChild(row);
        return;
    }
    for (var i = 0; i < list.length; i++) {
        var x = list.splice(i, 1);
        ret.push(x);
        perm(list, ret);
        ret.pop();
        list.splice(i, 0, x);
    }
}

perm([1, 2, 'A', 4], []);
</script></body></html>
```

Alternatively: 'Genuine' js code, assuming no duplicate.

```
function perm(a) {
    if (a.length < 2) return [a];
    var c, d, b = [];
    for (c = 0; c < a.length; c++) {
        var e = a.splice(c, 1),
            f = perm(a);
        for (d = 0; d < f.length; d++) b.push([e].concat(f[d]));
```

```
console.log(perm(['Aardvarks', 'eat', 'ants']).join("\n"));
```

Output:

```
Aardvarks,eat,ants
Aardvarks,ants,eat
eat,Aardvarks,ants
eat,ants,Aardvarks
ants,Aardvarks,eat
ants,eat,Aardvarks
```

Functional composition
Translation of: Haskell

(Simple version – assuming a unique list of objects comparable by the JS === operator)

```
(function () {
    'use strict';

    // permutations :: [a] -> [[a]]
    var permutations = function (xs) {
        return xs.length ? concatMap(function (x) {
            return concatMap(function (ys) {
                return [[x].concat(ys)];
            }, permutations(delete_(x, xs)));
        }, xs) : [[]];
    };

    // GENERIC FUNCTIONS

    // concatMap :: (a -> [b]) -> [a] -> [b]
    var concatMap = function (f, xs) {
        return [].concat.apply([], xs.map(f));
    };

    // delete :: Eq a => a -> [a] -> [a]
    var delete_ = function (x, xs) {
        return deleteBy(function (a, b) {
            return a === b;
        }, x, xs);
    };

    // deleteBy :: (a -> a -> Bool) -> a -> [a] -> [a]
    var deleteBy = function (f, x, xs) {
        return xs.length > 0 ? f(x, xs[0]) ? xs.slice(1) :
        [xs[0]].concat(deleteBy(f, x, xs.slice(1))) : [];
    };

    // TEST
    return permutations(['Aardvarks', 'eat', 'ants']);
})();
```

Output:

```
[["Aardvarks", "eat", "ants"], ["Aardvarks", "ants", "eat"],
 ["eat", "Aardvarks", "ants"], ["eat", "ants", "Aardvarks"],
["ants", "Aardvarks", "eat"], ["ants", "eat", "Aardvarks"]]
```

ES6

Recursively, in terms of concatMap and delete:

```
(() => {
    'use strict';

    // permutations :: [a] -> [[a]]
    const permutations = xs => {
        const go = xs => xs.length ? (
            concatMap(
                x => concatMap(
                    ys => [[x].concat(ys)],
                    go(delete_(x, xs))), xs
                )
        ) : [[]];
        return go(xs);
    };

    // GENERIC FUNCTIONS ----------------------------------

    // concatMap :: (a -> [b]) -> [a] -> [b]
    const concatMap = (f, xs) =>
        xs.reduce((a, x) => a.concat(f(x)), []);


    // delete :: Eq a => a -> [a] -> [a]
    const delete_ = (x, xs) => {
        const go = xs => {
            return 0 < xs.length ? (
                (x === xs[0]) ? (
                    xs.slice(1)
                ) : [xs[0]].concat(go(xs.slice(1)))
            ) : [];
        }
        return go(xs);
    };

    // TEST
    return JSON.stringify(
        permutations(['Aardvarks', 'eat', 'ants'])
    );
})();
```

Output:

```
[["Aardvarks", "eat", "ants"], ["Aardvarks", "ants", "eat"],
 ["eat", "Aardvarks", "ants"], ["eat", "ants", "Aardvarks"],
["ants", "Aardvarks", "eat"], ["ants", "eat", "Aardvarks"]]
```

Or, without recursion, in terms of concatMap and reduce:

```
(() => {
    'use strict';

    // permutations :: [a] -> [[a]]
    const permutations = xs =>
        xs.reduceRight(
            (a, x) => concatMap(
                xs => enumFromTo(0, xs.length)
                .map(n => xs.slice(0, n)
                    .concat(x)
                    .concat(xs.slice(n))
                ),
                a
            ),
            [[]]
        );

    // GENERIC FUNCTIONS ----------------------------------
```

```
    // ft :: Int -> Int -> [Int]
    const enumFromTo = (m, n) =>
        Array.from({
            length: 1 + n - m
        }, (_, i) => m + i);

    // showLog :: a -> IO ()
    const showLog = (...args) =>
        console.log(
            args
            .map(JSON.stringify)
            .join(' -> ')
        );

    // TEST ----------------------------------------------
    showLog(
        permutations([1, 2, 3])
    );
})();
```

Output:

```
[[1,2,3],[2,1,3],[2,3,1],[1,3,2],[3,1,2],[3,2,1]]
```

jq

"permutations" generates a stream of the permutations of the input array.

```
def permutations:
  if length == 0 then []
  else
    range(0;length) as $i
    | [.[$i]] + (del(.[$i])|permutations)
  end ;
```

Example 1: list them

```
[range(0;3)] | permutations
[0,1,2]
[0,2,1]
[1,0,2]
[1,2,0]
[2,0,1]
[2,1,0]
```

Example 2: count them

```
[[range(0;3)] | permutations] | length
6
```

Or more efficiently:

```
def count(s): reduce s as $i (0;.+1);
```

```
[range(0;3)] | count(permutations)
6
```

```
[range(0;10)] | count(permutations)
3628800
```

Julia

```
julia> perms(l) = isempty(l) ? [l] : [[x; y] for x in l for y in perms(setdiff(l, x))]
```

Output:

```
julia> perms([1,2,3])
6-element Vector{Vector{Int64}}:
 [1, 2, 3]
 [1, 3, 2]
 ⋮
 [3, 1, 2]
 [3, 2, 1]
```

Further support for permutation creation and processing is available in the `Combinatorics.jl` package. `permutations(v)` creates an iterator over all permutations of v. Julia 0.7 and 1.0+ require the line global i inside the for to update the i variable.

```
using Combinatorics

term = "RCode"
i = 0
pcnt = factorial(length(term))
print("All the permutations of ", term, " (", pcnt, "):\n    ")
for p in permutations(split(term, ""))
    global i
    print(join(p), " ")
    i += 1
    i %= 12
    i != 0 || print("\n    ")
end
println()
```

Output:

```
All the permutations of RCode (120):
    RCode RCoed RCdoe RCdeo RCeod RCedo RoCde RoCed RodCe RodeC RoeCd RoedC
    RdCoe RdCeo RdoCe RdoeC RdeCo RdeoC ReCod ReCdo ReoCd ReodC RedCo RedoC
    CRode CRoed CRdoe CRdeo CReod CRedo CoRde CoRed CodRe CodeR CoeRd CoedR
    CdRoe CdReo CdoRe CdoeR CdeRo CdeoR CeRod CeRdo CeoRd CeodR CedRo CedoR
    oRCde oRCed oRdCe oRdeC oReCd oRedC oCRde oCRed oCdRe oCdeR oCeRd oCedR
    odRCe odReC odCRe odCeR odeRC odeCR oeRCd oeRdC oeCRd oeCdR oedRC oedCR
    dRCoe dRCeo dRoCe dRoeC dReCo dReoC dCRoe dCReo dCoRe dCoeR dCeRo dCeoR
    doRCe doReC doCRe doCeR doeRC doeCR deRCo deRoC deCRo deCoR deoRC deoCR
    eRCod eRCdo eRoCd eRodC eRdCo eRdoC eCRod eCRdo eCoRd eCodR eCdRo eCdoR
    eoRCd eoRdC eoCRd eoCdR eodRC eodCR edRCo edRoC edCRo edCoR edoRC edoCR
```

```
# Generate all permutations of size t from an array a with possibly duplicated elements.
collect(Combinatorics.multiset_permutations([1,1,0,0,0],3))
```

Output:

```
7-element Array{Array{Int64,1},1}:
 [1, 1, 0]
 [1, 0, 1]
```

```
  [0, 0, 1]
  [0, 0, 0]
```

K

Translation of: J

```
   perm:{:[1<x;,/(>:'(x,x)#1,x#0)[;0,'1+_f x-1];,!x]}
   perm 2
(0 1
 1 0)

   `0:{1_,/" ",/:x}'r@perm@#r:("some";"random";"text")
some random text
some text random
random some text
random text some
text some random
text random some
```

Alternative:

```
   perm:{x@m@&n=(#?:)'m:!n#n:#x}

   perm[!3]
(0 1 2
 0 2 1
 1 0 2
 1 2 0
 2 0 1
 2 1 0)

   perm "abc"
("abc"
 "acb"
 "bac"
 "bca"
 "cab"
 "cba")

   `0:{1_,/" ",/: $x}' perm `$" "\"some random text"
some random text
some text random
random some text
random text some
text some random
text random some
```

Works with: ngn/k

```
 prm:{$[0=x;,!0;,/(prm x-1){?[1+x;y;0]}/:\:!x]}
perm:{x[prm[#x]]}

(("some";"random";"text")
 ("random";"some";"text")
 ("random";"text";"some")
 ("some";"text";"random")
 ("text";"some";"random")
 ("text";"random";"some"))
```

Note, however that K is heavily optimized for "long horizontal columns and short vertical rows". Thus, a different approach drastically improves performance:

```
prm:{$[x~*x;;:x@o@#x];(x-1){,/'((,(#*x)##x),x)m*(!l)+&\m:~=l:1+#x}/0}
```

```
(("text";"text";"random";"some";"random";"some")
 ("random";"some";"text";"text";"some";"random")
 ("some";"random";"some";"random";"text";"text"))
```

Kotlin

Translation of C# recursive 'insert' solution in Wikipedia article on Permutations:

```
// version 1.1.2

fun <T> permute(input: List<T>): List<List<T>> {
    if (input.size == 1) return listOf(input)
    val perms = mutableListOf<List<T>>()
    val toInsert = input[0]
    for (perm in permute(input.drop(1))) {
        for (i in 0..perm.size) {
            val newPerm = perm.toMutableList()
            newPerm.add(i, toInsert)
            perms.add(newPerm)
        }
    }
    return perms
}

fun main(args: Array<String>) {
    val input = listOf('a', 'b', 'c', 'd')
    val perms = permute(input)
    println("There are ${perms.size} permutations of $input, namely:\n")
    for (perm in perms) println(perm)
}
```

Output:

```
There are 24 permutations of [a, b, c, d], namely:

[a, b, c, d]
[b, a, c, d]
[b, c, a, d]
[b, c, d, a]
[a, c, b, d]
[c, a, b, d]
[c, b, a, d]
[c, b, d, a]
[a, c, d, b]
[c, a, d, b]
[c, d, a, b]
[c, d, b, a]
[a, b, d, c]
[b, a, d, c]
[b, d, a, c]
[b, d, c, a]
[a, d, b, c]
[d, a, b, c]
[d, b, a, c]
[d, b, c, a]
[a, d, c, b]
[d, a, c, b]
[d, c, a, b]
[d, c, b, a]
```

Using rotate

```
fun <T> List<T>.rotateLeft(n: Int) = drop(n) + take(n)
```

```
        else -> {
            permute(input.drop(1))
                .map { it + input.first() }
                .flatMap { subPerm -> List(subPerm.size) { i -> subPerm.rotateLeft(i) } }
        }
    }

fun main(args: Array<String>) {
    permute(listOf(1, 2, 3)).also {
        println("""There are ${it.size} permutations:
            |${it.joinToString(separator = "\n")}""".trimMargin())
    }
}
```

Output:

```
There are 6 permutations:
[3, 2, 1]
[2, 1, 3]
[1, 3, 2]
[2, 3, 1]
[3, 1, 2]
[1, 2, 3]
```

Lambdatalk

```
{def inject
 {lambda {:x :a}
  {if {A.empty? :a}
   then {A.new {A.new :x}}
   else {let { {:c {{lambda {:a :b} {A.cons {A.first :a} :b}} :a}}
               {:d {inject :x {A.rest :a}}}
               {:e {A.cons :x :a}}
             } {A.cons :e {A.map :c :d}}}}}}
-> inject

{def permut
 {lambda {:a}
  {if {A.empty? :a}
   then {A.new :a}
   else {let { {:c {{lambda {:a :b} {inject {A.first :a} :b}} :a}}
               {:d {permut {A.rest :a}}}
             } {A.reduce A.concat {A.map :c :d}}}}}}
-> permut

{permut {A.new 1 2 3}}
-> [[1,2,3],[2,1,3],[2,3,1],[1,3,2],[3,1,2],[3,2,1]]

{permut {A.new 1 2 3 4}}
->
[[1,2,3,4],[2,1,3,4],[2,3,1,4],[2,3,4,1],[1,3,2,4],[3,1,2,4],[3,2,1,4],[3,2,4,1],[1,3,4,2],[3,1,4,2],[3,4,1,2],[3,4,2,1],
[1,2,4,3],[2,1,4,3],[2,4,1,3],[2,4,3,1],[1,4,2,3],[4,1,2,3],[4,2,1,3],[4,2,3,1],[1,4,3,2],[4,1,3,2],[4,3,1,2],[4,3,2,1]]
```

And this is an illustration of the way lambdatalk builds an interface for javascript functions (the first one is given in this page):

```
1) permutations on sentences

{script
  var S_perm = function(a) {
    if (a.length < 2) return [a];
    var b = [];
    for (var c = 0; c < a.length; c++) {
        var e = a.splice(c, 1), f = S_perm(a);
        for (var d = 0; d < f.length; d++)
           b.push( e.concat( f[d] ) );
```

```
  }

  LAMBDATALK.DICT['S.perm'] = function() {  // {S.perm 1 2 3}
    return S_perm( arguments[0].trim()
                                .split(" ") )
                                .join(" ")
                                .replace(/\s/g,"{br}")
  };
}

{S.perm 1 2 3}
->
1,2,3
1,3,2
2,1,3
2,3,1
3,1,2
3,2,1

{S.perm hello brave world}
->
hello,brave,world
hello,world,brave
brave,hello,world
brave,world,hello
world,hello,brave
world,brave,hello

2) permutations on words

{script
  var W_perm = function(word) {
    if (word.length === 1) return [word]
    var results = [];
    for (var i = 0; i < word.length; i++) {
      var buti = W_perm( word.substring(0, i) + word.substring(i + 1) );
      for (var j = 0; j < buti.length; j++)
        results.push(word[i] + buti[j]);
    }
    return results;
  };

  LAMBDATALK.DICT['W.perm'] = function() {  // {W.perm 123}
    return W_perm( arguments[0].trim() ).join("{br}")
  };

}

{W.perm 123}
->
123
132
213
231
312
321
```

langur

Translation of: Go

This follows the Go language non-recursive example, but is not limited to integers, or even to numbers.

```
val .factorial = fn .x: if(.x < 2: 1; .x * self(.x - 1))

val .permute = fn(.list) {
    if .list is not list: throw "expected list"

    val .limit = 10
    if len(.list) > .limit: throw "permutation limit exceeded (currently {{.limit}})"
```

```
    val .n = len(.ordinals)
    var .i, .j

    for[.p=[.list]] of .factorial(len .list)-1 {
        .i = .n - 1
        .j = .n
        while .ordinals[.i] > .ordinals[.i+1] {
            .i -= 1
        }
        while .ordinals[.j] < .ordinals[.i] {
            .j -= 1
        }

        .ordinals[.i], .ordinals[.j] = .ordinals[.j], .ordinals[.i]
        .elements[.i], .elements[.j] = .elements[.j], .elements[.i]

        .i += 1
        for .j = .n; .i < .j ; .i, .j = .i+1, .j-1 {
            .ordinals[.i], .ordinals[.j] = .ordinals[.j], .ordinals[.i]
            .elements[.i], .elements[.j] = .elements[.j], .elements[.i]
        }
        .p = more .p, .elements
    }
}

for .e in .permute([1, 3.14, 7]) {
    writeln .e
}
```

Output:

```
[1, 3.14, 7]
[1, 7, 3.14]
[3.14, 1, 7]
[3.14, 7, 1]
[7, 1, 3.14]
[7, 3.14, 1]
```

LFE

```
(defun permute
  (('())
    '(()))
  ((l)
    (lc ((<- x l)
         (<- y (permute (-- l `(,x)))))
        (cons x y))))
```

REPL usage:

```
> (permute '(1 2 3))
((1 2 3) (1 3 2) (2 1 3) (2 3 1) (3 1 2) (3 2 1))
```

Lobster

```
// Lobster implementation of the (very fast) Go example
// http://rosettacode.org/wiki/Permutations#Go
// implementing the plain changes (bell ringers) algorithm, using a recursive function
// https://en.wikipedia.org/wiki/Steinhaus-Johnson-Trotter_algorithm

def permr(s, f):
    if s.length == 0:
```

```
        if np == 1:
            f(s)
            return
        let np1 = np - 1
        let pp = s.length - np1
        rc(np1) // recurs prior swaps
        var i = pp
        while i > 0:
            // swap s[i], s[i-1]
            let t = s[i]
            s[i] = s[i-1]
            s[i-1] = t
            rc(np1) // recurs swap
            i -= 1
        let w = s[0]
        for(pp): s[_] = s[_+1]
        s[pp] = w
    rc(s.length)

// Heap's recursive method https://en.wikipedia.org/wiki/Heap%27s_algorithm

def permh(s, f):
    def rc(k: int):
        if k <= 1:
            f(s)
        else:
            // Generate permutations with kth unaltered
            // Initially k == length(s)
            rc(k-1)
            // Generate permutations for kth swapped with each k-1 initial
            for(k-1) i:
                // Swap choice dependent on parity of k (even or odd)
                 // zero-indexed, the kth is at k-1
                if (k & 1) == 0:
                    let t = s[i]
                    s[i] = s[k-1]
                    s[k-1] = t
                else:
                    let t = s[0]
                    s[0] = s[k-1]
                    s[k-1] = t
                rc(k-1)
    rc(s.length)

// iterative Boothroyd method

import std

def permi(xs, f):
    var d = 1
    let c = map(xs.length): 0
    f(xs)
    while true:
        while d > 1:
            d -= 1
            c[d] = 0
        while c[d] >= d:
            d += 1
            if d >= xs.length:
                return
        let i = if (d & 1) == 1: c[d] else: 0
        let t = xs[i]
        xs[i] = xs[d]
        xs[d] = t
        f(xs)
        c[d] = c[d] + 1

// next lexicographical permutation
// to get all permutations the initial input `a` must be in sorted order
// returns false when input `a` is in reverse sorted order

def next_lex_perm(a):
    def swap(i, j):
        let t = a[i]
        a[i] = a[j]
```

```
          index exists, the permutation is the last permutation. */
    var k = n - 1
    while k > 0 and a[k-1] >= a[k]: k--
    if k == 0: return false
    k -= 1
    /* 2. Find the largest index l such that a[k] < a[l]. Since k + 1 is
       such an index, l is well defined */
    var l = n - 1
    while a[l] <= a[k]: l--
    /* 3. Swap a[k] with a[l] */
    swap(k, l)
    /* 4. Reverse the sequence from a[k + 1] to the end */
    k += 1
    l = n - 1
    while l > k:
        swap(k, l)
        l -= 1
        k += 1
    return true

var se = [0, 1, 2, 3] //, 4, 5, 6, 7, 8, 9, 10]

print "Iterative lexicographical permuter"

print se
while next_lex_perm(se): print se

print "Recursive plain changes iterator"

se = [0, 1, 2, 3]

permr(se): print(_)

print "Recursive Heap\'s iterator"

se = [0, 1, 2, 3]

permh(se): print(_)

print "Iterative Boothroyd iterator"

se = [0, 1, 2, 3]

permi(se): print(_)
```

Output:

```
Iterative lexicographical permuter
[0, 1, 2, 3]
[0, 1, 3, 2]
[0, 2, 1, 3]
[0, 2, 3, 1]
[0, 3, 1, 2]
[0, 3, 2, 1]
[1, 0, 2, 3]
[1, 0, 3, 2]
[1, 2, 0, 3]
[1, 2, 3, 0]
[1, 3, 0, 2]
[1, 3, 2, 0]
[2, 0, 1, 3]
[2, 0, 3, 1]
[2, 1, 0, 3]
[2, 1, 3, 0]
[2, 3, 0, 1]
[2, 3, 1, 0]
[3, 0, 1, 2]
[3, 0, 2, 1]
[3, 1, 0, 2]
[3, 1, 2, 0]
[3, 2, 0, 1]
[3, 2, 1, 0]
```

```
[0, 3, 1, 2]
[3, 0, 1, 2]
[0, 2, 1, 3]
[0, 2, 3, 1]
[0, 3, 2, 1]
[3, 0, 2, 1]
[2, 0, 1, 3]
[2, 0, 3, 1]
[2, 3, 0, 1]
[3, 2, 0, 1]
[1, 0, 2, 3]
[1, 0, 3, 2]
[1, 3, 0, 2]
[3, 1, 0, 2]
[1, 2, 0, 3]
[1, 2, 3, 0]
[1, 3, 2, 0]
[3, 1, 2, 0]
[2, 1, 0, 3]
[2, 1, 3, 0]
[2, 3, 1, 0]
[3, 2, 1, 0]
Recursive Heap's iterator
[0, 1, 2, 3]
[1, 0, 2, 3]
[2, 0, 1, 3]
[0, 2, 1, 3]
[1, 2, 0, 3]
[2, 1, 0, 3]
[3, 1, 0, 2]
[1, 3, 0, 2]
[0, 3, 1, 2]
[3, 0, 1, 2]
[1, 0, 3, 2]
[0, 1, 3, 2]
[0, 2, 3, 1]
[2, 0, 3, 1]
[3, 0, 2, 1]
[0, 3, 2, 1]
[2, 3, 0, 1]
[3, 2, 0, 1]
[3, 2, 1, 0]
[2, 3, 1, 0]
[1, 3, 2, 0]
[3, 1, 2, 0]
[2, 1, 3, 0]
[1, 2, 3, 0]
Iterative Boothroyd iterator
[0, 1, 2, 3]
[1, 0, 2, 3]
[2, 0, 1, 3]
[0, 2, 1, 3]
[1, 2, 0, 3]
[2, 1, 0, 3]
[3, 1, 0, 2]
[1, 3, 0, 2]
[0, 3, 1, 2]
[3, 0, 1, 2]
[1, 0, 3, 2]
[0, 1, 3, 2]
[0, 2, 3, 1]
[2, 0, 3, 1]
[3, 0, 2, 1]
[0, 3, 2, 1]
[2, 3, 0, 1]
[3, 2, 0, 1]
[3, 2, 1, 0]
[2, 3, 1, 0]
[1, 3, 2, 0]
[3, 1, 2, 0]
[2, 1, 3, 0]
[1, 2, 3, 0]
```

```
:- object(list).

    :- public(permutation/2).

    permutation(List, Permutation) :-
        same_length(List, Permutation),
        permutation2(List, Permutation).

    permutation2([], []).
    permutation2(List, [Head| Tail]) :-
        select(Head, List, Remaining),
        permutation2(Remaining, Tail).

    same_length([], []).
    same_length([_| Tail1], [_| Tail2]) :-
        same_length(Tail1, Tail2).

    select(Head, [Head| Tail], Tail).
    select(Head, [Head2| Tail], [Head2| Tail2]) :-
        select(Head, Tail, Tail2).

:- end_object.
```

Usage example:

```
| ?- forall(list::permutation([1, 2, 3], Permutation), (write(Permutation), nl)).

[1,2,3]
[1,3,2]
[2,1,3]
[2,3,1]
[3,1,2]
[3,2,1]
yes
```

Lua

```
local function permutation(a, n, cb)
    if n == 0 then
        cb(a)
    else
        for i = 1, n do
            a[i], a[n] = a[n], a[i]
            permutation(a, n - 1, cb)
            a[i], a[n] = a[n], a[i]
        end
    end
end

--Usage
local function callback(a)
    print('{'..table.concat(a, ', ')..'}')
end
permutation({1,2,3}, 3, callback)
```

Output:

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

```
    local taken = {} local slots = {}
    for i=1,a do slots[i]=0 end
    for i=1,b do taken[i]=false end
    local index = 1
    while index > 0 do repeat
        repeat slots[index] = slots[index] + 1
        until slots[index] > b or not taken[slots[index]]
        if slots[index] > b then
            slots[index] = 0
            index = index - 1
            if index > 0 then
                taken[slots[index]] = false
            end
            break
        else
            taken[slots[index]] = true
        end
        if index == a then
            for i=1,a do io.write(slots[i]) io.write(" ") end
            io.write("\n")
            taken[slots[index]] = false
            break
        end
        index = index + 1
    until true end
end

ipermutations(3, 3)
```

```
1 2 3
1 3 2
2 1 3
2 3 1
3 1 2
3 2 1
```

fast, iterative with coroutine to use as a generator

```
#!/usr/bin/env luajit
-- Iterative version
local function ipermgen(a,b)
    if a==0 then return end
    local taken = {} local slots = {}
    for i=1,a do slots[i]=0 end
    for i=1,b do taken[i]=false end
    local index = 1
    while index > 0 do repeat
        repeat slots[index] = slots[index] + 1
        until slots[index] > b or not taken[slots[index]]
        if slots[index] > b then
            slots[index] = 0
            index = index - 1
            if index > 0 then
                taken[slots[index]] = false
            end
            break
        else
            taken[slots[index]] = true
        end
        if index == a then
            coroutine.yield(slots)
            taken[slots[index]] = false
            break
        end
        index = index + 1
    until true end
end
local function iperm(a)
    local co=coroutine.create(function() ipermgen(a,a) end)
    return function()
```

```
end

local a=arg[1] and tonumber(arg[1]) or 3
for p in iperm(a) do
    print(table.concat(p, " "))
end
```

Output:

```
> ./perm_iter_coroutine.lua 3
1 2 3
1 3 2
2 1 3
2 3 1
3 1 2
3 2 1
```

M2000 Interpreter

All permutations in one module

```
Module Checkit {
      Global a$
      Document a$
      Module Permutations (s){
            Module Level (n, s, h)   {
                  If n=1 then {
                        while Len(s) {
                              m1=each(h)
                              while m1 {
                                    Print Array$(m1);" ";
                              }
                               Print Array$(S)
                               ToClipBoard()
                               s=cdr(s)
                         }
                  } Else {
                        for i=1 to len(s) {
                              call Level n-1, cdr(s),  cons(h, car(s))
                              s=cons(cdr(s), car(s))
                        }
                  }
                  Sub ToClipBoard()
                        local m=each(h)
                        Local b$=""
                        While m {
                              b$+=If$(Len(b$)<>0->" ","")+Array$(m)+" "
                        }
                        b$+=If$(Len(b$)<>0->" ","")+Array$(s,0)+" "+{
                        }
                        a$<=b$   ' assign to global need <=
                  End Sub
            }
            If len(s)=0 then Error
            Head=(,)
            Call Level Len(s),  s, Head
      }
      Clear a$
      Permutations (1,2,3,4)
      Permutations (100, 200, 500)
      Permutations ("A", "B", "C","D")
      Permutations ("DOG", "CAT", "BAT")
      ClipBoard a$
```

```
}
Checkit
```

Step by step Generator

```
Module StepByStep {
      Function PermutationStep (a) {
            c1=lambda (&f, a) ->{
                  =car(a)
                  f=true
            }
            m=len(a)
            c=c1
            while m>1 {
                  c1=lambda c2=c,p, m=(,) (&f, a) ->{
                        if len(m)=0 then m=a
                        =cons(car(m),c2(&f, cdr(m)))
                        if f then f=false:p++:  m=cons(cdr(m), car(m)) : if p=len(m) then p=0 : m=(,):: f=true
                  }
                  c=c1
                  m--
            }
            =lambda c, a (&f) -> {
                  =c(&f, a)
            }
      }
      k=false
      StepA=PermutationStep((1,2,3,4))
      while not k {
                  Print StepA(&k)
      }
      k=false
      StepA=PermutationStep((100,200,300))
      while not k {
                  Print StepA(&k)
      }
      k=false
      StepA=PermutationStep(("A", "B", "C", "D"))
      while not k {
                  Print StepA(&k)
      }
      k=false
      StepA=PermutationStep(("DOG", "CAT", "BAT"))
      while not k {
                  Print StepA(&k)
      }
}
StepByStep
```

Output:

```
1  2  3  4
1  2  4  3
1  3  4  2
1  3  2  4
1  4  2  3
1  4  3  2
2  3  4  1
2  3  1  4
2  4  1  3
2  4  3  1
2  1  3  4
```

```
divert(-1)

# 1-based indexing of a string's characters.
define(`get',`substr(`$1',decr(`$2'),1)')
define(`set',`substr(`$1',0,decr(`$2'))`'$3`'substr(`$1',`$2')')
define(`swap',
`pushdef(`_u',`get(`$1',`$2')')`'dnl
pushdef(`_v',`get(`$1',`$3')')`'dnl
set(set(`$1',`$2',_v),`$3',_u)`'dnl
popdef(`_u',`_v')')

# $1-fold repetition of $2.
define(`repeat',`ifelse($1,0,`',`$2`'$0(decr($1),`$2')')')

#
# Heap's algorithm. Algorithm 2 in Robert Sedgewick, 1977. Permutation
# generation methods. ACM Comput. Surv. 9, 2 (June 1977), 137-164.
#
# This implementation permutes the characters in a string of length no
# more than 9. On longer strings, it may strain the resources of a
# very old implementation of m4.
#
define(`permutations',
    `ifelse($2,`',`$1
$0(`$1',repeat(len(`$1'),1),2)',
        `ifelse(eval(($3) <= len(`$1')),1,
             `ifelse(eval(get($2,$3) < $3),1,
                      `swap(`$1',_$0($2,$3),$3)
$0(swap(`$1',_$0($2,$3),$3),set($2,$3,incr(get($2,$3))),2)',
                      `$0(`$1',set($2,$3,1),incr($3))')')')')
define(`_permutations',`eval((($2) % 2) + ((1 - (($2) % 2)) * get($1,$2)))')

divert`'dnl
permutations(`123')
permutations(`abcd')
```

Output:

$ m4 permutations.m4

```
123
213
312
132
231
321

abcd
bacd
cabd
acbd
bcad
cbad
dbac
bdac
adbc
dabc
badc
abdc
acdb
cadb
dacb
adcb
cdab
dcab
dcba
cdba
bdca
dbca
```

```
cbda
bcda
```

Maple

```
combinat:-permute(3);
   [[1, 2, 3], [1, 3, 2], [2, 1, 3], [2, 3, 1], [3, 1, 2], [3, 2, 1]]

combinat:-permute([a,b,c]);
   [[a, b, c], [a, c, b], [b, a, c], [b, c, a], [c, a, b], [c, b, a]]
```

An implementation based on Mathematica solution:

```
fold:=(f,a,v)->`if`(nops(v)=0,a,fold(f,f(a,op(1,v)),[op(2...,v)])):
insert:=(v,a,n)->`if`(n>nops(v),[op(v),a],subsop(n=(a,v[n]),v)):
perm:=s->fold((a,b)->map(u->seq(insert(u,b,k+1),k=0..nops(u)),a),[[]],s):
perm([$1..3]);
   [[3, 2, 1], [2, 3, 1], [2, 1, 3], [3, 1, 2], [1, 3, 2], [1, 2, 3]]
```

Mathematica/Wolfram Language

Note: The built-in version will have better performance.

Version from scratch

```
(***Standard list functions:*)
fold[f_, x_, {}] := x
fold[f_, x_, {h_, t___}] := fold[f, f[x, h], {t}]
insert[L_, x_, n_] := Join[L[[;; n - 1]], {x}, L[[n ;;]]]

(***Generate all permutations of a list S:*)

permutations[S_] :=
 fold[Join @@ (Function[{L},
       Table[insert[L, #2, k + 1], {k, 0, Length[L]}]] /@ #1) &, {{}},
   S]
```

Output:

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

Built-in version

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

Output:

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

MATLAB / Octave

```
perms([1,2,3,4])
```

Output:

```
4321
4312
4231
4213
4123
4132
3421
3412
3241
3214
3124
3142
2341
2314
2431
2413
2143
2134
1324
1342
1234
1243
1423
1432
```

Maxima

```
next_permutation(v) := block([n, i, j, k, t],
   n: length(v), i: 0,
   for k: n - 1 thru 1 step -1 do (if v[k] < v[k + 1] then (i: k, return())),
   j: i + 1, k: n,
   while j < k do (t: v[j], v[j]: v[k], v[k]: t, j: j + 1, k: k - 1),
   if i = 0 then return(false),
   j: i + 1,
   while v[j] < v[i] do j: j + 1,
   t: v[j], v[j]: v[i], v[i]: t,
   true
)$

print_perm(n) := block([v: makelist(i, i, 1, n)],
   disp(v),
   while next_permutation(v) do disp(v)
)$

print_perm(3);
/* [1, 2, 3]
   [1, 3, 2]
   [2, 1, 3]
   [2, 3, 1]
```

```
    [3, 1, 2]
    [3, 2, 1] */
```

Builtin version

```
(%i1) permutations([1, 2, 3]);
(%o1) {[1, 2, 3], [1, 3, 2], [2, 1, 3], [2, 3, 1], [3, 1, 2], [3, 2, 1]}
```

Mercury

```
:- module permutations2.
:- interface.

:- import_module io.

:- pred main(io::di, io::uo) is det.


:- import_module list.
:- import_module set_ordlist.
:- import_module set.
:- import_module solutions.

%% permutationSet(List, Set) is true if List is a permutation of Set:
:- pred permutationSet(list(A)::out,set(A)::in) is nondet.

%% Two ways to compute all permutations of a given list (using backtracking):
:- func all_permutations1(list(int))=set_ordlist.set_ordlist(list(int)).
:- func all_permutations2(list(int))=set_ordlist.set_ordlist(list(int)).

:- implementation.


permutationSet([],set.init).
permutationSet([H|T], S) :- set.member(H,S), permutationSet(T,set.delete(S,H)).

all_permutations1(L) =
    solutions_set(pred(X::out) is nondet:-permutationSet(X,set.from_list(L))).

%%Alternatively, using the imported list.perm predicate:
all_permutations2(L) =
    solutions_set(pred(X::out) is nondet:-perm(L,X)).

main(!IO) :-
    print(all_permutations1([1,2,3,4]),!IO),
    nl(!IO),
    print(all_permutations2([1,2,3,4]),!IO).
```

Output:

```
>./permutations2

sol([[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]])
sol([[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]])
```

Modula-2

Works with: ADW Modula-2 (1.6.291)

```
FROM    Terminal
IMPORT  Read, Write, WriteLn;

FROM    Terminal2
IMPORT  WriteString;

CONST   MAXIDX = 6;
    MINIDX = 1;

TYPE    TInpCh = ['a'..'z'];
    TChr   = SET OF TInpCh;

VAR n,
    nl: INTEGER;
    ch: CHAR;
    a:  ARRAY[MINIDX..MAXIDX] OF CHAR;
    kt:     TChr = TChr{'a'..'f'};

PROCEDURE output;
VAR i:  INTEGER;
BEGIN
    FOR i := MINIDX TO n DO Write(a[i]) END;
    WriteString(" | ");
END output;

PROCEDURE exchange(VAR x, y : CHAR);
VAR z:  CHAR;
BEGIN z := x; x := y; y := z
END exchange;

PROCEDURE permute(k: INTEGER);
VAR i:  INTEGER;
BEGIN
    IF k = 1 THEN
        output;
        INC(nl);
        IF (nl MOD 8 = 1) THEN WriteLn END;
    ELSE
        permute(k-1);
        FOR i := MINIDX TO k-1 DO
            exchange(a[i], a[k]);
            permute(k-1);
            exchange(a[i], a[k]);
        END
    END
END permute;

BEGIN
    n := 0; nl := 1; WriteString("Input {a,b,c,d,e,f} >");
    REPEAT
        Read(ch);
        IF ch IN kt THEN INC(n); a[n] := ch; Write(ch) END
    UNTIL (ch <= " ") OR (n > MAXIDX);

    WriteLn;
    IF n > 0 THEN permute(n) END;
    (*Wait*)
END Permute.
```

Modula-3

Simple version

This implementation merely prints out the orbit of the list (1, 2, ..., n) under the action of S_n. It shows off Modula-3's built-in `Set` type and uses the standard `IntSeq` library module.

```
MODULE Permutations EXPORTS Main;
```

```
CONST n = 3;

TYPE Domain = SET OF [ 1.. n ];

VAR

  chosen: IntSeq.T;
  values := Domain { };

PROCEDURE GeneratePermutations(VAR chosen: IntSeq.T; remaining: Domain) =
(*
  Recursively generates all the permutations of elements
  in the union of "chosen" and "values".
  Values in "chosen" have already been chosen;
  values in "remaining" can still be chosen.
  If "remaining" is empty, it prints the sequence and returns.
  Otherwise, it picks each element in "remaining", removes it,
  adds it to "chosen", recursively calls itself,
  then removes the last element of "chosen" and adds it back to "remaining".
*)
BEGIN
  FOR i := 1 TO n DO
    (* check if each element is in "remaining" *)
    IF i IN remaining THEN
      (* if so, remove from "remaining" and add to "chosen" *)
      remaining := remaining - Domain { i };
      chosen.addhi(i);
      IF remaining # Domain { } THEN
        (* still something to process? do it *)
        GeneratePermutations(chosen, remaining);
      ELSE
        (* otherwise, print what we've chosen *)
        FOR j := 0 TO chosen.size() - 2 DO
          IO.PutInt(chosen.get(j)); IO.Put(", ");
        END;
        IO.PutInt(chosen.gethi());
        IO.PutChar('\n');
      END;
      (* add "i" back to "remaining" and remove from "chosen" *)
      remaining := remaining + Domain { i };
      EVAL chosen.remhi();
    END;
  END;
END GeneratePermutations;

BEGIN

  (* initial setup *)
  chosen := NEW(IntSeq.T).init(n);
  FOR i := 1 TO n DO values := values + Domain { i }; END;

  GeneratePermutations(chosen, values);

END Permutations.
```

Output:

For reasons of space, we show only the elements of S_3, but we have tested it with higher.

```
1, 2, 3
1, 3, 2
2, 1, 3
2, 3, 1
3, 1, 2
3, 2, 1
```

Generic version

This version works on any type, and requires the library's `Set` and `Sequence`. As usual in Modula-3, the generic instance will need to be instantiated for whatever type you want to use, and you will also need to instantiate a set of, sequence of, and sequence of sequences of the domain elements. This will have to be taken care of by the `m3makefile`.

interface

Suppose that `D` is the domain of elements to be permuted. This module requires a `DomainSeq` (`Sequence` of `D`), a `DomainSet` (`Set` of `D`), and a `DomainSeqSeq` (`Sequence` of `Sequences` of `Domain`).

```
GENERIC INTERFACE GenericPermutations(DomainSeq, DomainSet, DomainSeqSeq);

(*
  "Domain" is where the things to permute come from (unused in interface).
  "DomainSeq" is a "Sequence" of "Domain".
  "DomainSet" is a "Set" of "Domain".
  "DomainSeqSeq" is a "Sequence" of "DomainSeq".
*)

PROCEDURE GeneratePermutations(
  READONLY chosen: DomainSeq.T;
  READONLY remaining: DomainSet.T;
  READONLY result: DomainSeqSeq.T
);
(*
  Recursively generates all the permutations of elements
  in the union of "chosen" and "remaining".
  Values in "chosen" have already been chosen;
  values in "remaining" can still be chosen.
  If "remaining" is empty, it adds the permutation to "result".
  Otherwise, it picks each element in "remaining", removes it,
  adds it to "chosen", recursively calls itself,
  then removes the last element of "chosen" and adds it back to "remaining".
  Although the parameters are modified, we can describe them as "READONLY"
  because we do not re-assign them.
*)

END GenericPermutations.
```

implementation

In addition to the interface's specifications, this requires a generic `Domain`. Some implementations of a set are not safe to iterate over while modifying (e.g., a tree), so this copies the values and iterates over them.

```
GENERIC MODULE GenericPermutations(Domain, DomainSeq, DomainSet, DomainSeqSeq);

(*
  "Domain" is where the things to permute come from.
  "DomainSeq" is a "Sequence" of "Domain".
  "DomainSet" is a "Set" of "Domain".
  "DomainSeqSeq" is a "Sequence" of "DomainSeq".
*)

PROCEDURE GeneratePermutations(
  READONLY chosen: DomainSeq.T;
  READONLY remaining: DomainSet.T;
  READONLY result: DomainSeqSeq.T
) =

(*
  Recursively generates all the permutations of elements
  in the union of "chosen" and "remaining".
  Values in "chosen" have already been chosen;
  values in "remaining" can still be chosen.
  If "remaining" is empty, it adds the permutation to "result".
  Otherwise, it picks each element in "remaining", removes it,
  adds it to "chosen", recursively calls itself,
```

```
VAR

  r: Domain.T; (* element added to permutation *)

  iterator := remaining.iterate(); (* to iterate through remaining elements *)

  values := NEW(DomainSeq.T).init(remaining.size());
  (* used to store values for iteration *)

BEGIN

  (* cannot safely modify a set while iterating, so we'll store the values *)
  WHILE iterator.next(r) DO values.addhi(r); END;

  (* now loop through the stored values *)
  FOR i := 0 TO values.size() - 1 DO

    (* remove from "remaining" and add to "chosen" *)
    r := values.get(i);
    EVAL remaining.delete(r);
    chosen.addhi(r);

    (* if this is not the last remaining elements, call recursively *)
    IF remaining.size() # 0 THEN
      GeneratePermutations(chosen, remaining, result);
    ELSE
      (* we have a new permutation; add a copy to the set *)
      VAR newPerm := NEW(DomainSeq.T).init(chosen.size());
      BEGIN
        FOR i := 0 TO chosen.size() - 1 DO
          newPerm.addhi(chosen.get(i));
        END;
        result.addhi(newPerm);
      END;
    END;

    (* move r back from chosen *)
    EVAL remaining.insert(chosen.remhi());

  END;

END GeneratePermutations;

BEGIN
END GenericPermutations.
```

Sample Usage

Here the domain is `Integer`, but the interface doesn't require that, so we "merely" need `IntSeq` (a `Sequence` of `Integer`), `IntSetTree` (a set type I use, but you could use `SetDef` or `SetList` if you prefer; I've tested it and it works), `IntSeqSeq` (a `Sequence` of `Sequences` of `Integer`), and `IntPermutations`, which is `GenericPermutations` instantiated for `Integer`.

```
MODULE GPermutations EXPORTS Main;

IMPORT IO, IntSeq, IntSetTree, IntSeqSeq, IntPermutations;

CONST

  n = 7;

VAR

  chosen: IntSeq.T;
  remaining: IntSetTree.T;
  result: IntSeqSeq.T;

PROCEDURE Factorial(n: CARDINAL): CARDINAL =
VAR result := 1;
BEGIN
```

```
END Factorial;

BEGIN

  (* initial setup *)
  chosen := NEW(IntSeq.T).init(n);
  remaining := NEW(IntSetTree.T).init();
  result := NEW(IntSeqSeq.T).init(Factorial(n));
  FOR i := 1 TO n DO EVAL remaining.insert(i); END;

  IntPermutations.GeneratePermutations(chosen, remaining, result);

  IO.Put("Printing "); IO.PutInt(result.size());
  IO.Put(" permutations of "); IO.PutInt(n); IO.Put(" elements \n");
  FOR i := 0 TO result.size() - 1 DO
    FOR j := 0 TO result.get(i).size() - 1 DO
      IO.PutInt(result.get(i).get(j)); IO.PutChar(' ');
    END;
    IO.PutChar('\n');
  END;

END GPermutations.
```

Output:

(somewhat edited!)

```
Printing 5040 permutations of 7 elements
1 2 3 4 5 6 7
1 2 3 4 5 7 6
1 2 3 4 6 5 7
...
7 6 5 4 2 3 1
7 6 5 4 3 1 2
7 6 5 4 3 2 1
```

NetRexx

```
/* NetRexx */
options replace format comments java crossref symbols nobinary

import java.util.List
import java.util.ArrayList

-- =============================================================================
/**
 * Permutation Iterator
 * <br />
 * <br />
 * Algorithm by E. W. Dijkstra, "A Discipline of Programming", Prentice-Hall, 1976, p.71
 */
class RPermutationIterator implements Iterator

  -- -------------------------------------------------------------------------
  properties indirect
    perms = List
    permOrders = int[]
    maxN
    currentN
    first = boolean

  -- -------------------------------------------------------------------------
  properties constant
    isTrue  = boolean (1 == 1)
    isFalse = boolean (1 \= 1)

  -- -------------------------------------------------------------------------
  method RPermutationIterator(initial = List) public
```

```
-- ------------------------------------------------------------------------
method RPermutationIterator(initial = Object[]) public
  init = ArrayList(initial.length)
  loop elmt over initial
    init.add(elmt)
    end elmt
  setUp(init)
  return

-- ------------------------------------------------------------------------
method RPermutationIterator(initial = Rexx[]) public
  init = ArrayList(initial.length)
  loop elmt over initial
    init.add(elmt)
    end elmt
  setUp(init)
  return

-- ------------------------------------------------------------------------
method setUp(initial = List) private
  setFirst(isTrue)
  setPerms(initial)
  setPermOrders(int[getPerms().size()])
  setMaxN(getPermOrders().length)
  setCurrentN(0)
  po = getPermOrders()
  loop i_ = 0 while i_ < po.length
    po[i_] = i_
    end i_
  return

-- ------------------------------------------------------------------------
method hasNext() public returns boolean
  status = isTrue
  if getCurrentN() == factorial(getMaxN()) then status = isFalse
  setCurrentN(getCurrentN() + 1)
  return status

-- ------------------------------------------------------------------------
method next() public returns Object
  if isFirst() then setFirst(isFalse)
  else do
    po = getPermOrders()
    i_ = getMaxN() - 1
    loop while po[i_ - 1] >= po[i_]
      i_ = i_ - 1
      end

    j_ = getMaxN()
    loop while po[j_ - 1] <= po[i_ - 1]
      j_ = j_ - 1
      end

    swap(i_ - 1, j_ - 1)

    i_ = i_ + 1
    j_ = getMaxN()
    loop while i_ < j_
      swap(i_ - 1, j_ - 1)
      i_ = i_ + 1
      j_ = j_ - 1
      end
    end
  return reorder()

-- ------------------------------------------------------------------------
method remove() public signals UnsupportedOperationException
  signal UnsupportedOperationException()

-- ------------------------------------------------------------------------
method swap(i_, j_) private
  po = getPermOrders()
  save   = po[i_]
  po[i_] = po[j_]
  po[j_] = save
```

```
  method reorder() private returns List
    result = ArrayList(getPerms().size())
    loop ix over getPermOrders()
      result.add(getPerms().get(ix))
      end ix
    return result

  -- ------------------------------------------------------------------------
  /**
   * Calculate n factorial: {@code n! = 1 * 2 * 3 .. * n}
   * @param n
   * @return n!
   */
  method factorial(n) public static
    fact = 1
    if n > 1 then loop i = 1 while i <= n
      fact = fact * i
      end i
    return fact

  -- ------------------------------------------------------------------------
  method main(args = String[]) public static
    thing02 = RPermutationIterator(['alpha', 'omega'])
    thing03 = RPermutationIterator([String 'one', 'two', 'three'])
    thing04 = RPermutationIterator(Arrays.asList([Integer(1), Integer(2), Integer(3), Integer(4)]))
    things = [thing02, thing03, thing04]
    loop thing over things
      N = thing.getMaxN()
      say 'Permutations:' N'! =' factorial(N)
      loop lineCount = 1 while thing.hasNext()
        prm = thing.next()
        say lineCount.right(8)':' prm.toString()
        end lineCount
      say 'Permutations:' N'! =' factorial(N)
      say
      end thing
    return
```

Output:

```
Permutations: 2! = 2
       1: [alpha, omega]
       2: [omega, alpha]
Permutations: 2! = 2

Permutations: 3! = 6
       1: [one, two, three]
       2: [one, three, two]
       3: [two, one, three]
       4: [two, three, one]
       5: [three, one, two]
       6: [three, two, one]
Permutations: 3! = 6

Permutations: 4! = 24
       1: [1, 2, 3, 4]
       2: [1, 2, 4, 3]
       3: [1, 3, 2, 4]
       4: [1, 3, 4, 2]
       5: [1, 4, 2, 3]
       6: [1, 4, 3, 2]
```

Nim

Using the standard library

```
while v.nextPermutation():
  echo v
```

Output:

```
[1, 2, 3]
[1, 3, 2]
[2, 1, 3]
[2, 3, 1]
[3, 1, 2]
[3, 2, 1]
```

Single yield iterator

```
iterator inplacePermutations[T](xs: var seq[T]): var seq[T] =
    assert xs.len <= 24, "permutation of array longer than 24 is not supported"

    let n = xs.len - 1
    var
      c: array[24, int8]
      i: int = 0

    for i in 0 .. n: c[i] = int8(i+1)

    while true:
      yield xs
      if i >= n: break

      c[i] -= 1
      let j = if (i and 1) == 1: 0 else: int(c[i])
      swap(xs[i+1], xs[j])

      i = 0
      while c[i] == 0:
        let t = i+1
        c[i] = int8(t)
        i = t
```

verification

```
import intsets
from math import fac
block:
  # test all permutations of length from 0 to 9
  for l in 0..9:

    # prepare data
    var xs = newSeq[int](l)
    for i in 0..<l: xs[i] = i
    var s = initIntSet()

    for cs in inplacePermutations(xs):

      # each permutation must be of length l
      assert len(cs) == l

      # each permutation must contain digits from 0 to l-1 exactly once
      var ds = newSeq[bool](l)
      for c in cs:
        assert not ds[c]
        ds[c] = true

      # generate a unique number for each permutation
      var h = 0
      for e in cs:
        h = l * h + e
```

```
    # check exactly l! unique number of permutations
    assert len(s) == fac(l)
```

Translation of C

Translation of: C

```
# iterative Boothroyd method
iterator permutations[T](ys: openarray[T]): seq[T] =
  var
    d = 1
    c = newSeq[int](ys.len)
    xs = newSeq[T](ys.len)

  for i, y in ys: xs[i] = y
  yield xs

  block outer:
    while true:
      while d > 1:
        dec d
        c[d] = 0
      while c[d] >= d:
        inc d
        if d >= ys.len: break outer
      let i = if (d and 1) == 1: c[d] else: 0
      swap xs[i], xs[d]
      yield xs
      inc c[d]

var x = @[1,2,3]

for i in permutations(x):
  echo i
```

Output:

```
@[1, 2, 3]
@[2, 1, 3]
@[3, 1, 2]
@[1, 3, 2]
@[2, 3, 1]
@[3, 2, 1]
```

Translation of Go

Translation of: Go

```
# Nim implementation of the (very fast) Go example.
# http://rosettacode.org/wiki/Permutations#Go
# implementing a recursive https://en.wikipedia.org/wiki/Steinhaus–Johnson–Trotter_algorithm

import algorithm

proc perm(s: openArray[int]; emit: proc(emit: openArray[int])) =
  var s = @s
  if s.len == 0:
    emit(s)
    return

  proc rc(np: int) =
    if np == 1:
      emit(s)
      return
    var
      np1 = np - 1
```

```
    for i in countDown(pp, 1):
      swap s[i], s[i-1]
      rc(np1) # Recurse swap.

    s.rotateLeft(0..pp, 1)

  rc(s.len)

var se = @[0, 1, 2, 3] #, 4, 5, 6, 7, 8, 9, 10]

perm(se, proc(s: openArray[int])= echo s)
```

OCaml

```
(* Iterative, though loops are implemented as auxiliary recursive functions.
   Translation of Ada version. *)
let next_perm p =
    let n = Array.length p in
    let i = let rec aux i =
        if (i < 0) || (p.(i) < p.(i+1)) then i
        else aux (i - 1) in aux (n - 2) in
    let rec aux j k = if j < k then
        let t = p.(j) in
            p.(j) <- p.(k);
            p.(k) <- t;
            aux (j + 1) (k - 1)
    else () in aux (i + 1) (n - 1);
    if i < 0 then false else
        let j = let rec aux j =
            if p.(j) > p.(i) then j
            else aux (j + 1) in aux (i + 1) in
        let t = p.(i) in
            p.(i) <- p.(j);
            p.(j) <- t;
            true;;

let print_perm p =
    let n = Array.length p in
    for i = 0 to n - 2 do
        print_int p.(i);
        print_string " "
    done;
    print_int p.(n - 1);
    print_newline ();;

let print_all_perm n =
    let p = Array.init n (function i -> i + 1) in
    print_perm p;
    while next_perm p do
        print_perm p
    done;;

print_all_perm 3;;
(* 1 2 3
   1 3 2
   2 1 3
   2 3 1
   3 1 2
   3 2 1 *)
```

Permutations can also be defined on lists recursively:

```
let rec permutations l =
   let n = List.length l in
   if n = 1 then [l] else
   let rec sub e = function
      | [] -> failwith "sub"
      | h :: t -> if h = e then t else h :: sub e t in
```

```
      let t = List.map (fun a -> e::a) subperms in
      if k < n-1 then List.rev_append t (aux (k+1)) else t in
   aux 0;;

let print l = List.iter (Printf.printf " %d") l; print_newline() in
List.iter print (permutations [1;2;3;4])
```

or permutations indexed independently:

```
let rec pr_perm k n l =
   let a, b = let c = k/n in c, k-(n*c) in
   let e = List.nth l b in
   let rec sub e = function
      | [] -> failwith "sub"
      | h :: t -> if h = e then t else h :: sub e t in
   (Printf.printf " %d" e; if n > 1 then pr_perm a (n-1) (sub e l))

let show_perms l =
   let n = List.length l in
   let rec fact n = if n < 3 then n else n * fact (n-1) in
   for i = 0 to (fact n)-1 do
      pr_perm i n l;
      print_newline()
   done

let () = show_perms [1;2;3;4]
```

ooRexx

Essentially derived fom the program shown under rexx. This program works also with Regina (and other REXX implementations?)

```
/* REXX Compute bunch permutations of things elements */
Parse Arg bunch things
If bunch='?' Then
  Call help
If bunch=='' Then bunch=3
If datatype(bunch)<>'NUM' Then Call help 'bunch ('bunch') must be numeric'
thing.=''
Select
  When things='' Then things=bunch
  When datatype(things)='NUM' Then Nop
  Otherwise Do
    data=things
    things=words(things)
    Do i=1 To things
      Parse Var data thing.i data
      End
    End
  End
If things<bunch Then Call help 'things ('things') must be >= bunch ('bunch')'

perms =0
Call time 'R'
Call permSets things, bunch
Say  perms  'Permutations'
Say time('E') 'seconds'
Exit

/*-------------------------------------------------------------------------------------*/
first_word: return word(Arg(1),1)
/*-------------------------------------------------------------------------------------*/
permSets: Procedure Expose perms thing.
  Parse Arg things,bunch
  aa.=''
  sep=''
  perm_elements='123456789ABCDEF'
```

```
    End
  Call .permSet 1
  Return

.permSet: Procedure Expose dd. aa. things bunch perms thing.
  Parse Arg iteration
  If iteration>bunch  Then do
    perm= aa.1
    Do j=2 For bunch-1
      perm= perm aa.j
      End
    perms+=1
    If thing.1<>'' Then Do
      ol=''
      Do pi=1 To words(perm)
        z=word(perm,pi)
        If datatype(z)<>'NUM' Then
          z=9+pos(z,'ABCDEF')
        ol=ol thing.z
        End
      Say strip(ol)
      End
    Else
      Say perm
    End
  Else Do
    Do q=1 for things
      Do k=1 for iteration-1
        If aa.k==dd.q  Then
          iterate q
        End
      aa.iteration= dd.q
      Call .permSet iteration+1
      End
    End
  Return

help:
  Parse Arg msg
  If msg<>'' Then Do
    Say 'ERROR:' msg
    Say ''
    End
  Say 'rexx perm            -> Permutations of 1 2 3                '
  Say 'rexx perm 2          -> Permutations of 1 2                  '
  Say 'rexx perm 2 4        -> Permutations of 1 2 3 4 in 2 positions'
  Say 'rexx perm 2 a b c d  -> Permutations of a b c d in 2 positions'
  Exit
```

Output:

```
H:\>rexx perm 2 U V W X
U V
U W
U X
V U
V W
V X
W U
W V
W X
X U
X V
X W
12 Permutations
0.006000 seconds

H:\>rexx perm ?
rexx perm            -> Permutations of 1 2 3
rexx perm 2          -> Permutations of 1 2
```

```
rexx perm 2 4        -> Permutations of 1 2 3 4 in 2 positions
rexx perm 2 a b c d  -> Permutations of a b c d in 2 positions
```

OpenEdge/Progress

```
DEFINE VARIABLE charArray AS CHARACTER EXTENT 3 INITIAL ["A","B","C"].
DEFINE VARIABLE sizeofArray AS INTEGER.

sizeOfArray = EXTENT(charArray).

RUN GetPermutations(1).

PROCEDURE GetPermutations:
    DEFINE INPUT PARAMETER n AS INTEGER.

    DEFINE VARIABLE i AS INTEGER.
    DEFINE VARIABLE j AS INTEGER.
    DEFINE VARIABLE currentPermutation AS CHARACTER.

    REPEAT i = n TO sizeOfArray:
        RUN swapValues(i,n).
        RUN GetPermutations(n + 1).
        RUN swapValues(i,n).
    END.
    IF n = sizeOfArray THEN DO:
        DO j = 1 TO EXTENT(charArray):
            currentPermutation = currentPermutation + charArray[j].
        END.
        DISPLAY currentPermutation WITH FRAME A DOWN.
    END.
END PROCEDURE.

PROCEDURE swapValues:
    DEFINE INPUT PARAMETER a AS INTEGER.
    DEFINE INPUT PARAMETER b AS INTEGER.
    DEFINE VARIABLE temp AS CHARACTER.
    temp = charArray[a].
    charArray[a] = charArray[b].
    charArray[b] = temp.
END PROCEDURE.
```

Output:

```
ABC
ACB
BAC
BCA
CAB
CBA
```

PARI/GP

```
vector(n!,k,numtoperm(n,k))
```

Pascal

```
program perm;

var
    p: array[1 .. 12] of integer;
    is_last: boolean;
    n: integer;
```

```
begin
is_last := true;
i := n - 1;
while i > 0 do
    begin
    if p[i] < p[i + 1] then
        begin
        is_last := false;
        break;
        end;
    i := i - 1;
    end;

if not is_last then
    begin
    j := i + 1;
    k := n;
    while j < k do
        begin
        t := p[j];
        p[j] := p[k];
        p[k] := t;
        j := j + 1;
        k := k - 1;
        end;

    j := n;
    while p[j] > p[i] do j := j - 1;
    j := j + 1;

    t := p[i];
    p[i] := p[j];
    p[j] := t;
    end;
end;

procedure print;
var i: integer;
begin
for i := 1 to n do write(p[i], ' ');
writeln;
end;

procedure init;
var i: integer;
begin
n := 0;
while (n < 1) or (n > 10) do
    begin
    write('Enter n (1 <= n <= 10): ');
    readln(n);
    end;
for i := 1 to n do p[i] := i;
end;

begin
init;
repeat
    print;
    next;
until is_last;
end.
```

alternative

a little bit more speed.I take n = 12. The above version takes more than 5 secs.My permlex takes 2.8s, but in the depth of my harddisk I found a version, creating all permutations using k places out of n.The cpu loves it! 1.33 s. But you have to use the integers [1..n] directly or as Index to your data. 1 to n are in lexicographic order.

```
{$ELSE}
  {$APPTYPE CONSOLE}
{$ENDIF}
uses
  sysutils;
type
  tPermfield  =  array[0..15] of Nativeint;
var
  permcnt: NativeUint;

procedure DoSomething(k: NativeInt;var x:tPermfield);
var
  i:integer;
  kk:string;
begin
  kk:='';
  for i:=1 to k do kk:=kk+inttostr(x[i])+' ';
    writeln(kk);
end;

procedure PermKoutOfN(k,n: nativeInt);
var
  x,y:tPermfield;
  i,yi,tmp:NativeInt;
begin
  //initialise
  permcnt:= 1;
  if k>n then
    k:=n;
  if k=n then
    k:=k-1;
  for i:=1 to n do x[i]:=i;
  for i:=1 to k do y[i]:=i;

//  DoSomething(k,x);
  i := k;
  repeat
    yi:=y[i];
    if yi <n then
    begin
      inc(permcnt);
      inc(yi);
      y[i]:=yi;
      tmp:=x[i];x[i]:=x[yi];x[yi]:=tmp;
      i:=k;
//      DoSomething(k,x);
    end
    else
    begin
      repeat
        tmp:=x[i];x[i]:=x[yi];x[yi]:=tmp;
        dec(yi);
      until yi<=i;
      y[i]:=yi;
      dec(i);
    end;
  until (i=0);
end;

var
  t1,t0 : TDateTime;
Begin
  permcnt:= 0;
  T0 := now;
  PermKoutOfN(12,12);
  T1 := now;
  writeln(permcnt);
  writeln(FormatDateTime('HH:NN:SS.zzz',T1-T0));
end.
```

Output:

{fpc 2.64/3.0 32Bit or 3.1 64 Bit i4330 3.5 Ghz same timings. //PermKoutOfN(12,12);

```
479001600 //= 12!
00:00:01.328
```

Permutations from integers

A console application in Free Pascal, created with the Lazarus IDE.

```
program Permutations;
(*
Demonstrates four closely related ways of establishing a bijection between
permutations of 0..(n-1) and integers 0..(n! - 1).
Each integer in that range is represented by mixed-base digits d[0..n-1],
where each d[j] satisfies 0 <= d[j] <=j.
The integer represented by d[0..n-1] is
  d[n-1]*(n-1)! + d[n-2]*(n-2)! + ... + d[1]*1! + d[0]*0!
where the last term can be omitted in practice because d[0] is always 0.
See the section "Numbering permutations" in the Wikipedia article
"Permutation" (NB their digit array d is 1-based).
*)
uses SysUtils, TypInfo;
type TPermIntMapping = (map_I, map_J, map_K, map_L);
type TPermutation = array of integer;

// Function to map an integer to a permutation.
function IntToPerm( map : TPermIntMapping;
                    nrItems, z : integer) : TPermutation;
var
  d, lookup : array of integer;
  x, y : integer;
  h, j, k, m : integer;
begin
  SetLength( result, nrItems);
  SetLength( lookup, nrItems);
  SetLength( d, nrItems);
  m := nrItems - 1;
  // Convert z to digits d[0..m] (see comment at head of program).
  d[0] := 0;
  y := z;
  for j := 1 to m - 1 do begin
    x := y div (j + 1);
    d[j] := y - x*(j + 1);
    y := x;
  end;
  d[m] := y;

  // Set up the permutation elements
  case map of
    map_I, map_L: for j := 0 to m do lookup[j] := j;
    map_J, map_K: for j := 0 to m do lookup[j] := m - j;
  end;
  for j := m downto 0 do begin
    k := d[j];
    case map of
      map_I: result[lookup[k]] := m - j;
      map_J: result[j] := lookup[k];
      map_K: result[lookup[k]] := j;
      map_L: result[m - j] := lookup[k];
    end;
    // When lookup[k] has been used, it's removed from the lookup table
    //   and the elements above it are moved down one place.
    for h := k to j - 1 do lookup[h] := lookup[h + 1];
  end;
end;

// Function to map a permutation to an integer; inverse of the above.
// Put in for completeness, not required for Rosetta Code task.
function PermToInt( map : TPermIntMapping;
                    p : TPermutation) : integer;
```

```
begin
  m := High(p); // number of items in permutation is m + 1
  SetLength( d, m + 1);
  for k := 0 to m do d[k] := 0; // initialize all digits to 0

  // Looking for inversions
  for i := 0 to m - 1 do begin
    for j := i + 1 to m do begin
      if p[j] < p[i] then begin
        case map of
          map_I : inc( d[m - p[j]]);
          map_J : inc( d[j]);
          map_K : inc( d[p[i]]);
          map_L : inc( d[m - i]);
        end;
      end;
    end;
  end;
  // Get result from its digits (see comment at head of program).
  result := d[m];
  for j := m downto 2 do result := result*j + d[j - 1];
end;

// Main routine to generate permutations of the integers 0..(n-1),
// where n is passed as a command-line parameter, e.g. Permutations 4
var
  n, n_fac, z, j : integer;
  nrErrors : integer;
  perm : TPermutation;
  map : TPermIntMapping;
  lineOut : string;
  pinfo : TypInfo.PTypeInfo;
begin
  n := SysUtils.StrToInt( ParamStr(1));
  n_fac := 1;
  for j := 2 to n do n_fac := n_fac*j;
  pinfo := System.TypeInfo( TPermIntMapping);
  lineOut := 'integer';
  for map := Low( TPermIntMapping) to High( TPermIntMapping) do begin
    lineOut := lineOut + '   ' + TypInfo.GetEnumName( pinfo, ord(map));
  end;
  WriteLn( lineOut);
  for z := 0 to n_fac - 1 do begin
    lineOut := SysUtils.Format( '%7d', [z]);
    for map := Low( TPermIntMapping) to High( TPermIntMapping) do begin
      perm := IntToPerm( map, n, z);
      // Check the inverse mapping (not required for Rosetta Code task)
      Assert( z = PermToInt( map, perm));
      lineOut := lineOut + '    ';
      for j := 0 to n - 1 do
        lineOut := lineOut + SysUtils.Format( '%d', [perm[j]]);
    end;
    WriteLn( lineOut);
  end;
end.
```

Output:

```
integer   map_I   map_J   map_K   map_L
      0    0123    0123    0123    0123
      1    0132    1023    1023    0132
      2    0213    0213    0213    0213
      3    0312    2013    1203    0231
      4    0231    1203    2013    0312
      5    0321    2103    2103    0321
      6    1023    0132    0132    1023
      7    1032    1032    1032    1032
      8    2013    0312    0231    1203
      9    3012    3012    1230    1230
     10    2031    1302    2031    1302
     11    3021    3102    2130    1320
     12    1203    0231    0312    2013
```

```
     16    2301    2301    2301    2301
     17    3201    3201    2310    2310
     18    1230    1230    3012    3012
     19    1320    2130    3102    3021
     20    2130    1320    3021    3102
     21    3120    3120    3120    3120
     22    2310    2310    3201    3201
     23    3210    3210    3210    3210
```

Perl

A simple recursive implementation.

```
sub permutation {
    my ($perm,@set) = @_;
    print "$perm\n" || return unless (@set);
    permutation($perm.$set[$_],@set[0..$_-1],@set[$_+1..$#set]) foreach (0..$#set);
}
my @input = (qw/a b c d/);
permutation('',@input);
```

Output:

```
abcd
abdc
acbd
acdb
adbc
adcb
bacd
badc
bcad
bcda
bdac
bdca
cabd
cadb
cbad
cbda
cdab
cdba
dabc
dacb
dbac
dbca
dcab
dcba
```

For better performance, use a module like `ntheory` or `Algorithm::Permute`.

Library: ntheory

```
use ntheory qw/forperm/;
my @tasks = (qw/party sleep study/);
forperm {
  print "@tasks[@_]\n";
} @tasks;
```

Output:

```
party sleep study
party study sleep
```

```
study party sleep
study sleep party
```

Phix

```
with javascript_semantics
requires("1.0.2")
?shorten(permutes("abcd"),"elements",5)
```

Output:

```
{"abcd","abdc","acbd","acdb","adbc","...","dacb","dbac","dbca","dcab","dcba"," (24 elements)"}
```

The elements can be any type. There is also a permute() function which accepts an integer between 1 and factorial(length(s)) and returns the permutations in lexicographical position order. It is just as fast to generate the (n!)th permutation as the first, so some applications may benefit by storing an integer key rather than duplicating all the elements of the given set.

Phixmonti

```
include ..\Utilitys.pmt

def save
    over over chain ps> swap 0 put >ps
enddef

def permute /# l l -- #/
    len 2 > if
        len for drop
            pop swap rot swap 1 put swap permute
        endfor
    else
        save rotate save rotate
    endif
    swap len if
        pop rot rot 0 put
    else
        drop drop
    endif
enddef

( ) >ps
( ) ( 1 2 3 4 ) permute
ps> sort print
```

Picat

Picat has built-in support for permutations:

- `permutation(L)`: Generates all permutations for a list L.
- `permutation(L,P)`: Generates (via backtracking) all permutations for a list L.

Recursion

Use `findall/2` to find all permutations. See example below.

```
permutation_rec1([X|Y],Z) :-
  permutation_rec1(Y,W),
  select(X,Z,W).
permutation_rec1([],[]).

permutation_rec2([], []).
permutation_rec2([X], [X]) :-!.
permutation_rec2([T|H], X) :-
  permutation_rec2(H, H1),
  append(L1, L2, H1),
  append(L1, [T], X1),
  append(X1, L2, X).
```

Constraint modelling

Constraint modelling only handles integers, and here generates all permutations of a list 1..N for a given N.

`permutation_cp_list(L)` permutes a list via `permutation_cp2/1`.

```
import cp.

% Returns all permutations
permutation_cp1(N) = solve_all(X) =>
   X = new_list(N),
   X :: 1..N,
   all_different(X).

% Find next permutation on backtracking
permutation_cp2(N,X) =>
   X = new_list(N),
   X :: 1..N,
   all_different(X),
   solve(X).

% Use the cp approach on a list L.
permutation_cp_list(L) = Perms =>
  Perms = [ [L[I] : I in P] : P in permutation_cp1(L.len)].
```

Tests

Here is a test of the different approaches, including the two built-ins.

```
import util, cp.
main =>
  N = 3,
  println(permutations=permutations(1..N)), % built in
  println(permutation=findall(P,permutation([a,b,c],P))), % built-in
  println(permutation_rec1=findall(P,permutation_rec1(1..N,P))),
  println(permutation_rec2=findall(P,permutation_rec2(1..N,P))),
  println(permutation_cp1=permutation_cp1(N)),
  println(permutation_cp2=findall(P,permutation_cp2(N,P))),
  println(permutation_cp_list=permutation_cp_list("abc")).
```

Output:

```
permutations = [[1,2,3],[2,1,3],[2,3,1],[1,3,2],[3,1,2],[3,2,1]]
permutation = [abc,acb,bac,bca,cab,cba]
permutation_rec1 = [[1,2,3],[2,1,3],[2,3,1],[1,3,2],[3,1,2],[3,2,1]]
permutation_rec2 = [[1,2,3],[2,1,3],[2,3,1],[1,3,2],[3,1,2],[3,2,1]]
permutation_cp1 = [[1,2,3],[1,3,2],[2,1,3],[2,3,1],[3,1,2],[3,2,1]]
permutation_cp2 = [[1,2,3],[1,3,2],[2,1,3],[2,3,1],[3,1,2],[3,2,1]]
```

PicoLisp

```
(load "@lib/simul.l")

(permute (1 2 3))
```

Output:

```
-> ((1 2 3) (1 3 2) (2 1 3) (2 3 1) (3 1 2) (3 2 1))
```

PowerShell

```
function permutation ($array) {
    function generate($n, $array, $A) {
        if($n -eq 1) {
            $array[$A] -join ' '
        }
        else{
            for( $i = 0; $i -lt ($n - 1); $i += 1) {
                generate ($n - 1) $array $A
                if($n % 2 -eq 0){
                    $i1, $i2 = $i, ($n-1)
                    $A[$i1], $A[$i2] = $A[$i2], $A[$i1]
                }
                else{
                    $i1, $i2 = 0, ($n-1)
                    $A[$i1], $A[$i2] = $A[$i2], $A[$i1]
                }
            }
            generate ($n - 1) $array $A
        }
    }
    $n = $array.Count
    if($n -gt 0) {
        (generate $n $array (0..($n-1)))
    } else {$array}
}
permutation @('A','B','C')
```

Output:

```
A B C
B A C
C A B
A C B
B C A
C B A
```

Prolog

Works with SWI-Prolog and library clpfd,

```
:- use_module(library(clpfd)).

permut_clpfd(L, N) :-
    length(L, N),
```

Output:

```
?- permut_clpfd(L, 3), writeln(L), fail.
[1,2,3]
[1,3,2]
[2,1,3]
[2,3,1]
[3,1,2]
[3,2,1]
false.
```

A declarative way of fetching permutations:

```
% permut_Prolog(P, L)
% P is a permutation of L

permut_Prolog([], []).
permut_Prolog([H | T], NL) :-
    select(H, NL, NL1),
    permut_Prolog(T, NL1).
```

Output:

```
 ?- permut_Prolog(P, [ab, cd, ef]), writeln(P), fail.
[ab,cd,ef]
[ab,ef,cd]
[cd,ab,ef]
[cd,ef,ab]
[ef,ab,cd]
[ef,cd,ab]
false.
```

Translation of: Curry

```
insert(X, L, [X|L]).
insert(X, [Y|Ys], [Y|L2]) :- insert(X, Ys, L2).

permutation([], []).
permutation([X|Xs], P) :- permutation(Xs, L), insert(X, L, P).
```

Output:

```
?- permutation([a,b,c],X).
X = [a, b, c] ;
X = [b, a, c] ;
X = [b, c, a] ;
X = [a, c, b] ;
X = [c, a, b] ;
X = [c, b, a] ;
false.
```

Python

Standard library function

Works with: Python version 2.6+

```
import itertools
for values in itertools.permutations([1,2,3]):
```

Output:

```
(1, 2, 3)
(1, 3, 2)
(2, 1, 3)
(2, 3, 1)
(3, 1, 2)
(3, 2, 1)
```

Recursive implementation

The follwing functions start from a list [0 ... n-1] and exchange elements to always have a valid permutation. This is done recursively: first exchange a[0] with all the other elements, then a[1] with a[2] ... a[n-1], etc. thus yielding all permutations.

```
def perm1(n):
    a = list(range(n))
    def sub(i):
        if i == n - 1:
            yield tuple(a)
        else:
            for k in range(i, n):
                a[i], a[k] = a[k], a[i]
                yield from sub(i + 1)
                a[i], a[k] = a[k], a[i]
    yield from sub(0)

def perm2(n):
    a = list(range(n))
    def sub(i):
        if i == n - 1:
            yield tuple(a)
        else:
            for k in range(i, n):
                a[i], a[k] = a[k], a[i]
                yield from sub(i + 1)
            x = a[i]
            for k in range(i + 1, n):
                a[k - 1] = a[k]
            a[n - 1] = x
    yield from sub(0)
```

These two solutions make use of a generator, and "yield from" introduced in PEP-380 (https://www.python.org/dev/peps/pep-0380/). They are slightly different: the latter produces permutations in lexicographic order, because the "remaining" part of a (that is, a[i+1:]) is always sorted, whereas the former always reverses the exchange just after the recursive call.

On three elements, the difference can be seen on the last two permutations:

```
for u in perm1(3): print(u)
(0, 1, 2)
(0, 2, 1)
(1, 0, 2)
(1, 2, 0)
(2, 1, 0)
(2, 0, 1)

for u in perm2(3): print(u)
(0, 1, 2)
(0, 2, 1)
(1, 0, 2)
(1, 2, 0)
```

```
(2, 0, 1)
(2, 1, 0)
```

Iterative implementation

Given a permutation, one can easily compute the *next* permutation in some order, for example lexicographic order, here. Then to get all permutations, it's enough to start from [0, 1, ... n-1], and store the next permutation until [n-1, n-2, ... 0], which is the last in lexicographic order.

```
def nextperm(a):
    n = len(a)
    i = n - 1
    while i > 0 and a[i - 1] > a[i]:
        i -= 1
    j = i
    k = n - 1
    while j < k:
        a[j], a[k] = a[k], a[j]
        j += 1
        k -= 1
    if i == 0:
        return False
    else:
        j = i
        while a[j] < a[i - 1]:
            j += 1
        a[i - 1], a[j] = a[j], a[i - 1]
        return True

def perm3(n):
    if type(n) is int:
        if n < 1:
            return []
        a = list(range(n))
    else:
        a = sorted(n)
    u = [tuple(a)]
    while nextperm(a):
        u.append(tuple(a))
    return u

for p in perm3(3): print(p)
(0, 1, 2)
(0, 2, 1)
(1, 0, 2)
(1, 2, 0)
(2, 0, 1)
(2, 1, 0)
```

Implementation using destructive list updates

```
def permutations(xs):
    ac = [[]]
    for x in xs:
        ac_new = []
        for ts in ac:
            for n in range(0,ts.__len__()+1):
                new_ts = ts[:]  #(shallow) copy of ts
                new_ts.insert(n,x)
                ac_new.append(new_ts)
        ac=ac_new
    return ac
```

```
print(permutations([1,2,3,4]))
```

Functional :: type-preserving

The **itertools.permutations** function is polymorphic in its inputs but not in its outputs – it discards the type of input lists and strings, coercing all inputs to tuples.

In this type-preserving variant, permutation is defined (without the need for mutating name-bindings) in terms of two universal abstractions: **reduce** and **concatMap**:

Works with: Python version 3.7

```
'''Permutations of a list, string or tuple'''

from functools import (reduce)
from itertools import (chain)


# permutations :: [a] -> [[a]]
def permutations(xs):
    '''Type-preserving permutations of xs.
    '''
    ps = reduce(
        lambda a, x: concatMap(
            lambda xs: (
                xs[n:] + [x] + xs[0:n] for n in range(0, 1 + len(xs)))
        )(a),
        xs, [[]]
    )
    t = type(xs)
    return ps if list == t else (
        [''.join(x) for x in ps] if str == t else [
            t(x) for x in ps
        ]
    )


# TEST ----------------------------------------------------

# main :: IO ()
def main():
    '''Permutations of lists, strings and tuples.'''

    print(
        fTable(__doc__ + ':\n')(repr)(showList)(
            permutations
        )([
            [1, 2, 3],
            'abc',
            (1, 2, 3),
        ])
    )


# GENERIC -------------------------------------------------

# concatMap :: (a -> [b]) -> [a] -> [b]
def concatMap(f):
    '''A concatenated list over which a function has been mapped.
       The list monad can be derived by using a function f which
       wraps its output in a list,
       (using an empty list to represent computational failure).'''
    return lambda xs: list(
        chain.from_iterable(map(f, xs))
    )
```

```
#                     (b -> String) -> (a -> b) -> [a] -> String
def fTable(s):
    '''Heading -> x display function -> fx display function ->
                    f -> xs -> tabular string.
    '''
    def go(xShow, fxShow, f, xs):
        ys = [xShow(x) for x in xs]
        w = max(map(len, ys))
        return s + '\n' + '\n'.join(map(
            lambda x, y: y.rjust(w, ' ') + ' -> ' + fxShow(f(x)),
            xs, ys
        ))
    return lambda xShow: lambda fxShow: lambda f: lambda xs: go(
        xShow, fxShow, f, xs
    )


# showList :: [a] -> String
def showList(xs):
    '''Stringification of a list.'''
    return '[' + ','.join(showList(x) for x in xs) + ']' if (
        isinstance(xs, list)
    ) else repr(xs)


# MAIN ---
if __name__ == '__main__':
    main()
```

Output:

```
[1, 2, 3] -> [[1,2,3],[2,3,1],[3,1,2],[2,1,3],[1,3,2],[3,2,1]]
    'abc' -> ['abc','bca','cab','bac','acb','cba']
(1, 2, 3) -> [(1, 2, 3),(2, 3, 1),(3, 1, 2),(2, 1, 3),(1, 3, 2),(3, 2, 1)]
```

Qi

Translation of: Erlang

```
(define insert
  L      0 E -> [E|L]
  [L|Ls] N E -> [L|(insert Ls (- N 1) E)])

(define seq
  Start Start -> [Start]
  Start End   -> [Start|(seq (+ Start 1) End)])

(define append-lists
  []    -> []
  [A|B] -> (append A (append-lists B)))

(define permutate
  []    -> [[]]
  [H|T] -> (append-lists (map (/. P
                                (map (/. N
                                       (insert P N H))
                                     (seq 0 (length P))))
                              (permute T))))
```

Quackery

General Solution

The word *perms* solves a more general task; generate permutations of between *a* and *b* items (inclusive) from the specified nest.

```
  [ stack ]                    is perms.min    (       --> [ )

  [ stack ]                    is perms.max    (       --> [ )

                        forward is (perms)

  [ over size
    perms.min share > if
      [ over temp take
        swap nested join
        temp put ]
    over size
    perms.max share < if
      [ dup size times
         [ 2dup i^ pluck
           rot swap nested join
           swap (perms) ] ]
    2drop ]               resolves (perms)      (   [ [ -->    )

  [ perms.max put
    1 - perms.min put
    [] temp put
    [] swap (perms)
    temp take
    perms.min release
    perms.max release ]         is perms        ( [ a b --> [ )

  [ dup size dup perms ]        is permutations (     [ --> [ )

  ' [ 1 2 3 ] permutations echo cr
  $ "quack" permutations 60 wrap$
  $ "quack" 3 4 perms 46 wrap$
```

Output:

```
[ [ 1 2 3 ] [ 1 3 2 ] [ 2 1 3 ] [ 2 3 1 ] [ 3 1 2 ] [ 3 2 1 ] ]

quack quakc qucak qucka qukac qukca qauck qaukc qacuk qacku
qakuc qakcu qcuak qcuka qcauk qcaku qckua qckau qkuac qkuca
qkauc qkacu qkcua qkcau uqack uqakc uqcak uqcka uqkac uqkca
uaqck uaqkc uacqk uackq uakqc uakcq ucqak ucqka ucaqk ucakq
uckqa uckaq ukqac ukqca ukaqc ukacq ukcqa ukcaq aquck aqukc
aqcuk aqcku aqkuc aqkcu auqck auqkc aucqk auckq aukqc aukcq
acquk acqku acuqk acukq ackqu ackuq akquc akqcu akuqc akucq
akcqu akcuq cquak cquka cqauk cqaku cqkua cqkau cuqak cuqka
cuaqk cuakq cukqa cukaq caquk caqku cauqk caukq cakqu cakuq
ckqua ckqau ckuqa ckuaq ckaqu ckauq kquac kquca kqauc kqacu
kqcua kqcau kuqac kuqca kuaqc kuacq kucqa kucaq kaquc kaqcu
kauqc kaucq kacqu kacuq kcqua kcqau kcuqa kcuaq kcaqu kcauq

qua quac quak quc quca quck quk quka qukc qau
qauc qauk qac qacu qack qak qaku qakc qcu qcua
qcuk qca qcau qcak qck qcku qcka qku qkua qkuc
qka qkau qkac qkc qkcu qkca uqa uqac uqak uqc
uqca uqck uqk uqka uqkc uaq uaqc uaqk uac uacq
uack uak uakq uakc ucq ucqa ucqk uca ucaq ucak
uck uckq ucka ukq ukqa ukqc uka ukaq ukac ukc
ukcq ukca aqu aquc aquk aqc aqcu aqck aqk aqku
aqkc auq auqc auqk auc aucq auck auk aukq aukc
acq acqu acqk acu acuq acuk ack ackq acku akq
akqu akqc aku akuq akuc akc akcq akcu cqu cqua
cquk cqa cqau cqak cqk cqku cqka cuq cuqa cuqk
cua cuaq cuak cuk cukq cuka caq caqu caqk cau
cauq cauk cak cakq caku ckq ckqu ckqa cku ckuq
ckua cka ckaq ckau kqu kqua kquc kqa kqau kqac
kqc kqcu kqca kuq kuqa kuqc kua kuaq kuac kuc
```

```
kucq kuca kaq kaqu kaqc kau kauq kauc kac kacq
kacu kcq kcqu kcqa kcu kcuq kcua kca kcaq kcau
```

An Uncommon Ordering

Edit: I *think* this process is called "iterative deepening". Would love to have this confirmed or corrected.

The central idea is that given a list of the permutations of say 3 items, each permutation can be used to generate 4 of the permutations of 4 items, so for example, from `[ 3 1 2 ]` we can generate

```
[ 0 3 1 2 ]
[ 3 0 1 2 ]
[ 3 1 0 2 ]
[ 3 1 2 0 ]
```

by stuffing the 0 into each of the 4 possible positions that it could go.

The code start with a nest of all the permutations of 0 items `[ [ ] ]`, and each time though the outer `times` loop (i.e. 4 times in the example) it takes each of the permutations generated so far (this is the `witheach` loop) and applies the central idea described above (that is the inner `times` loop.)

Some aids to reading the code.

Quackery is a stack based language. If you are unfamiliar the with words `swap`, `rot`, `dup`, `2dup`, `dip`, `unrot` or `drop` they can be skimmed over as "noise" to get a gist of the process.

`[]` creates an empty nest `[ ]`.

`times` indicates that the word or nest following it is to be repeated a specified number of times. (The specified number is on the top of the stack, so `4 times [ ... ]`repeats some arbitrary code 4 times.)

`i` returns the number of times a `times` loop has left to repeat. It counts down to zero.

`i^` returns the number of times a `times` loop has been repeated. It counts up from zero.

`size` returns the number of items (words, numbers, nests) in a nest.

`witheach` indicates that the word or nest following it is to be repeated once for each item in a specified nest, with successive items from the nest available on the top of stack on each repetition.

`999 ' [ 10 11 12 13 ] 3 stuff` will return `[ 10 11 12 999 13 ]`by stuffing the number 999 into the 3rd position in the nest. (The start of a nest is the zeroth position, the end of this nest is the 5th position.)

`nested join` adds a nest to the end of a nest as its last item.

```
  [ ' [ [ ] ] swap times
      [ [] i rot witheach
        [ dup size 1+ times
          [ 2dup i^ stuff
            dip rot nested join
            unrot ] drop ] drop ] ] is perms ( n --> [ )

  4 perms witheach [ echo cr ]
```

Output:

```
[ 0 1 2 3 ]
[ 1 0 2 3 ]
[ 1 2 0 3 ]
[ 1 2 3 0 ]
[ 0 2 1 3 ]
[ 2 0 1 3 ]
[ 2 1 0 3 ]
[ 2 1 3 0 ]
[ 0 2 3 1 ]
[ 2 0 3 1 ]
[ 2 3 0 1 ]
[ 2 3 1 0 ]
[ 0 1 3 2 ]
[ 1 0 3 2 ]
[ 1 3 0 2 ]
[ 1 3 2 0 ]
[ 0 3 1 2 ]
[ 3 0 1 2 ]
[ 3 1 0 2 ]
[ 3 1 2 0 ]
[ 0 3 2 1 ]
[ 3 0 2 1 ]
[ 3 2 0 1 ]
[ 3 2 1 0 ]
```

R

Iterative version

```
next.perm <- function(a) {
  n <- length(a)
  i <- n
  while (i > 1 && a[i - 1] >= a[i]) i <- i - 1
  if (i == 1) {
    NULL
  } else {
    j <- i
    k <- n
    while (j < k) {
      s <- a[j]
      a[j] <- a[k]
      a[k] <- s
      j <- j + 1
      k <- k - 1
    }
    s <- a[i - 1]
    j <- i
    while (a[j] <= s) j <- j + 1
    a[i - 1] <- a[j]
    a[j] <- s
    a
  }
}

perm <- function(n) {
  e <- NULL
  a <- 1:n
  repeat {
    e <- cbind(e, a)
    a <- next.perm(a)
    if (is.null(a)) break
  }
  unname(e)
}
```

Example

```
[1,]    1    1    2    2    3    3
[2,]    2    3    1    3    1    2
[3,]    3    2    3    1    2    1
```

Recursive version

```
# list of the vectors by inserting x in s at position 0...end.
linsert <- function(x,s) lapply(0:length(s), function(k) append(s,x,k))

# list of all permutations of 1:n
perm <- function(n){
    if (n == 1) list(1)
    else unlist(lapply(perm(n-1), function(s) linsert(n,s)),
                recursive = F)}

# permutations of a vector s
permutation <- function(s) lapply(perm(length(s)), function(i) s[i])
```

Output:

```
> permutation(letters[1:3])
[[1]]
[1] "c" "b" "a"

[[2]]
[1] "b" "c" "a"

[[3]]
[1] "b" "a" "c"

[[4]]
[1] "c" "a" "b"

[[5]]
[1] "a" "c" "b"

[[6]]
[1] "a" "b" "c"
```

Racket

```
#lang racket

;; using a builtin
(permutations '(A B C))
;; -> '((A B C) (B A C) (A C B) (C A B) (B C A) (C B A))

;; a random simple version (which is actually pretty good for a simple version)
(define (perms l)
  (let loop ([l l] [tail '()])
    (if (null? l) (list tail)
        (append-map (λ(x) (loop (remq x l) (cons x tail))) l))))
(perms '(A B C))
;; -> '((C B A) (B C A) (C A B) (A C B) (B A C) (A B C))

;; permutations in lexicographic order
(define (lperms s)
  (cond [(empty? s) '()]
        [(empty? (cdr s)) (list s)]
        [else
         (let splice ([l '()][m (car s)][r (cdr s)])
           (append
            (map (lambda (x) (cons m x)) (lperms (append l r)))
            (if (empty? r) '()
                (splice (append l (list m)) (car r) (cdr r)))))]))
```

```
;; permutations in lexicographical order using generators
(require racket/generator)
(define (splice s)
  (generator ()
             (let outer-loop ([l '()][m (car s)][r (cdr s)])
               (let ([permuter (lperm (append l r))])
                 (let inner-loop ([p (permuter)])
                   (when (not (void? p))
                     (let ([q (cons m p)])
                       (yield q)
                       (inner-loop (permuter))))))
               (if (not (empty? r))
                   (outer-loop (append l (list m)) (car r) (cdr r))
                   (void)))))
(define (lperm s)
  (generator ()
             (cond [(empty? s) (yield '())]
                   [(empty? (cdr s)) (yield s)]
                   [else
                    (let ([splicer (splice s)])
                      (let loop ([q (splicer)])
                        (when (not (void? q))
                          (begin
                            (yield q)
                            (loop (splicer))))))])
             (void)))
(let ([permuter (lperm '(A B C))])
  (let next-perm ([p (permuter)])
    (when (not (void? p))
      (begin
        (display p)
        (next-perm (permuter))))))
;; -> (A B C)(A C B)(B A C)(B C A)(C A B)(C B A)
```

Raku

(formerly Perl 6)

Works with: rakudo version 2018.10

First, you can just use the built-in method on any list type.

```
.say for <a b c>.permutations
```

Output:

```
a b c
a c b
b a c
b c a
c a b
c b a
```

Here is some generic code that works with any ordered type. To force lexicographic ordering, change `after` to `gt`. To force numeric order, replace it with `>`.

```
sub next_perm ( @a is copy ) {
    my $j = @a.end - 1;
    return Nil if --$j < 0 while @a[$j] after @a[$j+1];

    my $aj = @a[$j];
    my $k  = @a.end;
    $k-- while $aj after @a[$k];
```

```
    @a[ $r--, $s++ ] .= reverse while $r > $s;
    return @a;
}

.say for [<a b c>], &next_perm ...^ !*;
```

Output:

```
a b c
a c b
b a c
b c a
c a b
c b a
```

Here is another non-recursive implementation, which returns a lazy list. It also works with any type.

```
sub permute(+@items) {
   my @seq := 1..+@items;
   gather for (^[*] @seq) -> $n is copy {
      my @order;
      for @seq {
         unshift @order, $n mod $_;
         $n div= $_;
      }
      my @i-copy = @items;
      take map { |@i-copy.splice($_, 1) }, @order;
   }
}
.say for permute( 'a'..'c' )
```

Output:

```
(a b c)
(a c b)
(b a c)
(b c a)
(c a b)
(c b a)
```

Finally, if you just want zero-based numbers, you can call the built-in function:

```
.say for permutations(3);
```

Output:

```
0 1 2
0 2 1
1 0 2
1 2 0
2 0 1
2 1 0
```

RATFOR

For translation to FORTRAN 77 with the public domain ratfor77 preprocessor.

```
# Heap's algorithm for generating permutations. Algorithm 2 in
```

```
define(n, 3)
define(n_minus_1, 2)

implicit none

integer a(1:n)

integer c(1:n)
integer i, k
integer tmp

10000 format ('(', I1, n_minus_1(' ', I1), ')')

# Initialize the data to be permuted.
do i = 1, n {
   a(i) = i
}

# What follows is a non-recursive Heap’s algorithm as presented by
# Sedgewick. Sedgewick neglects to fully initialize c, so I have
# corrected for that. Also I compute k without branching, by instead
# doing a little arithmetic.
do i = 1, n {
   c(i) = 1
}
i = 2
write (*, 10000) a
while (i <= n) {
   if (c(i) < i) {
      k = mod (i, 2) + ((1 - mod (i, 2)) * c(i))
      tmp = a(i)
      a(i) = a(k)
      a(k) = tmp
      c(i) = c(i) + 1
      i = 2
      write (*, 10000) a
   } else {
      c(i) = 1
      i = i + 1
   }
}

end
```

Here is what the generated FORTRAN 77 code looks like:

```
C Output from Public domain Ratfor, version 1.0
      implicit none
      integer a(1: 3)
      integer c(1: 3)
      integer i, k
      integer tmp
10000 format ('(', i1,  2(' ', i1), ')')
      do23000 i = 1,  3
      a(i) = i
23000 continue
23001 continue
      do23002 i = 1,  3
      c(i) = 1
23002 continue
23003 continue
      i = 2
      write (*, 10000) a
23004 if(i .le.  3)then
      if(c(i) .lt. i)then
      k = mod (i, 2) + ((1 - mod (i, 2)) * c(i))
      tmp = a(i)
      a(i) = a(k)
      a(k) = tmp
      c(i) = c(i) + 1
      i = 2
      write (*, 10000) a
```

```
      endif
      goto 23004
      endif
23005 continue
      end
```

Output:

$ ratfor77 permutations.r > permutations.f && f2c permutations.f && cc -o permutations permutations.c -lf2c && ./permutations

```
(1 2 3)
(2 1 3)
(3 1 2)
(1 3 2)
(2 3 1)
(3 2 1)
```

REXX

This program could be simplified quite a bit if the "things" were just restricted to numbers (numerals), but that would make it specific to numbers and not "things" or objects.

```
/*REXX pgm generates/displays all permutations of N different objects taken M at a time.*/
parse arg things bunch inbetweenChars names      /*obtain optional arguments from the CL*/
if things=='' | things==","  then things= 3      /*Not specified?  Then use the default.*/
if  bunch=='' |  bunch==","  then  bunch= things /* "      "         "   "   "     "    */
                 /* ╔═══════════════════════════════════════════════════════════════════╗ */
                 /* ║  inBetweenChars  (optional)   defaults to a  [null].              ║ */
                 /* ║            names  (optional)   defaults to digits (and letters).  ║ */
                 /* ╚═══════════════════════════════════════════════════════════════════╝ */
call permSets things, bunch, inBetweenChars, names
exit                                             /*stick a fork in it,  we're all done. */
/*──────────────────────────────────────────────────────────────────────────────────────*/
p:        return word( arg(1), 1)                /*P  function (Pick first arg of many).*/
/*──────────────────────────────────────────────────────────────────────────────────────*/
permSets: procedure; parse arg x,y,between,uSyms /*X    things taken    Y    at a time. */
          @.=;       sep=                        /*X  can't be  >  length(@0abcs).      */
          @abc  = 'abcdefghijklmnopqrstuvwxyz';     @abcU=  @abc;          upper @abcU
          @abcS = @abcU || @abc;                    @0abcS= 123456789  ||  @abcS

            do k=1  for x                        /*build a list of permutation symbols. */
            _= p(word(uSyms, k)  p(substr(@0abcS, k, 1) k) )    /*get/generate a symbol.*/
            if length(_)\==1  then sep= '_'      /*if not 1st character,  then use sep. */
            $.k= _                               /*append the character to symbol list. */
            end   /*k*/

          if between==''  then between= sep      /*use the appropriate separator chars. */
          call .permSet 1                        /*start with the  first  permutation.  */
          return                                 /* [↓]  this is a recursive subroutine.*/
.permSet: procedure expose $. @. between x y;       parse arg ?
          if ?>y  then do;  _= @.1;      do j=2  for y-1
                                         _= _  ||  between  ||  @.j
                                         end   /*j*/
                            say _
                       end
                  else do q=1  for x             /*build the  permutation  recursively. */
                         do k=1 for ?-1;  if @.k==$.q  then iterate q
                         end   /*k*/
                       @.?= $.q;         call .permSet ?+1
                       end     /*q*/
          return
```

output when the following was used for input: 3 3

```
213
231
312
321
```

output when the following was used for input: `4 4 --- A B C D`

```
A---B---C---D
A---B---D---C
A---C---B---D
A---C---D---B
A---D---B---C
A---D---C---B
B---A---C---D
B---A---D---C
B---C---A---D
B---C---D---A
B---D---A---C
B---D---C---A
C---A---B---D
C---A---D---B
C---B---A---D
C---B---D---A
C---D---A---B
C---D---B---A
D---A---B---C
D---A---C---B
D---B---A---C
D---B---C---A
D---C---A---B
D---C---B---A
```

output when the following was used for input: `4 3 ~ aardvark gnu stegosaurus platypus`

```
aardvark~gnu~stegosaurus
aardvark~gnu~platypus
aardvark~stegosaurus~gnu
aardvark~stegosaurus~platypus
aardvark~platypus~gnu
aardvark~platypus~stegosaurus
gnu~aardvark~stegosaurus
gnu~aardvark~platypus
gnu~stegosaurus~aardvark
gnu~stegosaurus~platypus
gnu~platypus~aardvark
gnu~platypus~stegosaurus
stegosaurus~aardvark~gnu
stegosaurus~aardvark~platypus
stegosaurus~gnu~aardvark
stegosaurus~gnu~platypus
stegosaurus~platypus~aardvark
stegosaurus~platypus~gnu
platypus~aardvark~gnu
platypus~aardvark~stegosaurus
platypus~gnu~aardvark
platypus~gnu~stegosaurus
platypus~stegosaurus~aardvark
platypus~stegosaurus~gnu
```

Ring

```
load "stdlib.ring"

list = 1:4
lenList = len(list)
```

```
        add(permut,list[i])
    next
    perm(list)
next
for n = 1 to len(permut)/lenList
    for m = (n-1)*lenList+1 to n*lenList
        see "" + permut[m]
        if m < n*lenList
           see ","
        ok
    next
    see nl
next

func perm a
     elementcount = len(a)
     if elementcount < 1 then return ok
     pos = elementcount-1
     while a[pos] >= a[pos+1]
           pos -= 1
           if pos <= 0 permutationReverse(a, 1, elementcount)
              return ok
     end
     last = elementcount
     while a[last] <= a[pos]
           last -= 1
     end
     temp = a[pos]
     a[pos] = a[last]
     a[last] = temp
     permReverse(a, pos+1, elementcount)

 func permReverse a, first, last
      while first < last
            temp = a[first]
            a[first] = a[last]
            a[last] = temp
            first += 1
            last -= 1
      end
```

Output:

```
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
```

Ring

```
Another Solution

// Permutations -- Bert Mariani  2020-07-12
// Ask User for number of digits to permutate

   ? "Enter permutations number : "  Give n
   n = number(n)
   x = 1:n                            // array

    ? "Permutations are : "
   count = 0

    nPermutation(1,n)                  //===>>>  START

    ? " "                              // ? = print
    ? "Exiting of the program... "
    ? "Enter to Exit : "  Give m       // To Exit CMD window

//======================
// Returns true only if uniq number on row

Func Place(k,i)

     for j=1 to k-1
          if x[j] = i                  // Two numbers in same row
             return 0
          ok
     next

return 1

//======================
Func nPermutation(k, n)

     for i = 1 to n
          if( Place(k,i))              //===>>> Call
               x[k] = i
               if(k=n)
                  See nl
                     for i= 1 to n
                  See " "+ x[i]
                     next
                  See "    "+ (count++)
               else
                    nPermutation(k+1,n)   //===>>>  Call RECURSION
               ok
          ok
     next
return
```

Output:

```
Enter permutations number :
4
Permutations are :

 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 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
Exiting of the program...
Enter to Exit :
```

Ruby

```
p [1,2,3].permutation.to_a
```

Output:

```
[[1, 2, 3], [1, 3, 2], [2, 1, 3], [2, 3, 1], [3, 1, 2], [3, 2, 1]]
```

Rust

Iterative

Uses Heap's algorithm. An in-place version is possible but is incompatible with `Iterator`.

```
pub fn permutations(size: usize) -> Permutations {
    Permutations { idxs: (0..size).collect(), swaps: vec![0; size], i: 0 }
}

pub struct Permutations {
    idxs: Vec<usize>,
    swaps: Vec<usize>,
    i: usize,
}

impl Iterator for Permutations {
    type Item = Vec<usize>;

    fn next(&mut self) -> Option<Self::Item> {
        if self.i > 0 {
            loop {
                if self.i >= self.swaps.len() { return None; }
                if self.swaps[self.i] < self.i { break; }
                self.swaps[self.i] = 0;
                self.i += 1;
            }
            self.idxs.swap(self.i, (self.i & 1) * self.swaps[self.i]);
            self.swaps[self.i] += 1;
        }
        self.i = 1;
        Some(self.idxs.clone())
    }
}

fn main() {
    let perms = permutations(3).collect::<Vec< >>();
```

```
        vec![2, 0, 1],
        vec![0, 2, 1],
        vec![1, 2, 0],
        vec![2, 1, 0],
    ]);
}
```

Recursive

```
use std::collections::VecDeque;

fn permute<T, F: Fn(&[T])>(used: &mut Vec<T>, unused: &mut VecDeque<T>, action: &F) {
    if unused.is_empty() {
        action(used);
    } else {
        for _ in 0..unused.len() {
            used.push(unused.pop_front().unwrap());
            permute(used, unused, action);
            unused.push_back(used.pop().unwrap());
        }
    }
}

fn main() {
    let mut queue = (1..4).collect::<VecDeque<_>>();
    permute(&mut Vec::new(), &mut queue, &|perm| println!("{:?}", perm));
}
```

SAS

```
/* Store permutations in a SAS dataset. Translation of Fortran 77 */
data perm;
  n=6;
  array a{6} p1-p6;
  do i=1 to n;
    a(i)=i;
  end;
L1:
  output;
  link L2;
  if next then goto L1;
  stop;
L2:
  next=0;
  i=n-1;
L10:
  if a(i)<a(i+1) then goto L20;
  i=i-1;
  if i=0 then goto L20;
  goto L10;
  L20:
  j=i+1;
  k=n;
L30:
  t=a(j);
  a(j)=a(k);
  a(k)=t;
  j=j+1;
  k=k-1;
  if j<k then goto L30;
  j=i;
  if j=0 then return;
L40:
  j=j+1;
  if a(j)<a(i) then goto L40;
  t=a(i);
  a(i)=a(j);
```

```
  keep p1-p6;
run;
```

Scala

There is a built-in function in the Scala collections library, that is part of the language's standard library. The permutation function is available on any sequential collection. It could be used as follows given a list of numbers:

```
List(1, 2, 3).permutations.foreach(println)
```

Output:

```
List(1, 2, 3)
List(1, 3, 2)
List(2, 1, 3)
List(2, 3, 1)
List(3, 1, 2)
List(3, 2, 1)
```

The following function returns all the permutations of a list:

```
  def permutations[T]: List[T] => Traversable[List[T]] = {
    case Nil => List(Nil)
    case xs => {
      for {
        (x, i) <- xs.zipWithIndex
        ys <- permutations(xs.take(i) ++ xs.drop(1 + i))
      } yield {
        x :: ys
      }
    }
  }
```

If you need the unique permutations, use `distinct` or `toSet` on either the result or on the input.

Scheme

Translation of: Erlang

```
(define (insert l n e)
  (if (= 0 n)
      (cons e l)
      (cons (car l)
            (insert (cdr l) (- n 1) e))))

(define (seq start end)
  (if (= start end)
      (list end)
      (cons start (seq (+ start 1) end))))

(define (permute l)
  (if (null? l)
      '(())
      (apply append (map (lambda (p)
                           (map (lambda (n)
                                  (insert p n (car l)))
                                (seq 0 (length p))))
                         (permute (cdr l))))))
```

```
; translation of ocaml : mostly iterative, with auxiliary recursive functions for some loops
(define (vector-swap! v i j)
(let ((tmp (vector-ref v i)))
(vector-set! v i (vector-ref v j))
(vector-set! v j tmp)))

(define (next-perm p)
(let* ((n (vector-length p))
    (i (let aux ((i (- n 2)))
    (if (or (< i 0) (< (vector-ref p i) (vector-ref p (+ i 1))))
        i (aux (- i 1))))))
(let aux ((j (+ i 1)) (k (- n 1)))
    (if (< j k) (begin (vector-swap! p j k) (aux (+ j 1) (- k 1)))))
(if (< i 0) #f (begin
    (vector-swap! p i (let aux ((j (+ i 1)))
        (if (> (vector-ref p j) (vector-ref p i)) j (aux (+ j 1)))))
    #t))))

(define (print-perm p)
(let ((n (vector-length p)))
(do ((i 0 (+ i 1))) ((= i n)) (display (vector-ref p i)) (display " "))
(newline)))

(define (print-all-perm n)
(let ((p (make-vector n)))
(do ((i 0 (+ i 1))) ((= i n)) (vector-set! p i i))
(print-perm p)
(do ( ) ((not (next-perm p))) (print-perm p))))

(print-all-perm 3)
; 0 1 2
; 0 2 1
; 1 0 2
; 1 2 0
; 2 0 1
; 2 1 0

;a more recursive implementation
(define (permute p i)
(let ((n (vector-length p)))
(if (= i (- n 1)) (print-perm p)
(begin
    (do ((j i (+ j 1))) ((= j n))
        (vector-swap! p i j)
        (permute p (+ i 1)))
    (do ((j (- n 1) (- j 1))) ((< j i))
        (vector-swap! p i j))))))


(define (print-all-perm-rec n)
(let ((p (make-vector n)))
(do ((i 0 (+ i 1))) ((= i n)) (vector-set! p i i))
(permute p 0)))

(print-all-perm-rec 3)
; 0 1 2
; 0 2 1
; 1 0 2
; 1 2 0
; 2 0 1
; 2 1 0
```

Completely recursive on lists:

```
(define (perm s)
  (cond ((null? s) '())
    ((null? (cdr s)) (list s))
    (else ;; extract each item in list in turn and perm the rest
      (let splice ((l '()) (m (car s)) (r (cdr s)))
        (append
          (map (lambda (x) (cons m x)) (perm (append l r)))
          (if (null? r) '()
```

```
(display (perm '(1 2 3)))
```

Seed7

```
$ include "seed7_05.s7i";

const type: permutations is array array integer;

const func permutations: permutations (in array integer: items) is func
  result
    var permutations: permsList is 0 times 0 times 0;
  local
    const proc: perms (in array integer: sequence, in array integer: prefix) is func
      local
        var integer: element is 0;
        var integer: index is 0;
      begin
        if length(sequence) <> 0 then
          for element key index range sequence do
            perms(sequence[.. pred(index)] & sequence[succ(index) ..], prefix & [] (element));
          end for;
        else
          permsList &:= prefix;
        end if;
      end func;
  begin
    perms(items, 0 times 0);
  end func;

const proc: main is func
  local
    var array integer: perm is 0 times 0;
    var integer: element is 0;
  begin
    for perm range permutations([] (1, 2, 3)) do
      for element range perm do
        write(element <& " ");
      end for;
      writeln;
    end for;
  end func;
```

Output:

```
1 2 3
1 3 2
2 1 3
2 3 1
3 1 2
3 2 1
```

Shen

```
(define permute
[] -> []
[X] -> [[X]]
X -> (permute-helper [] X))

(define permute-helper
_ [] -> []
Done [X|Rest] -> (append (prepend-all X (permute (append Done Rest))) (permute-helper [X|Done] Rest))
)

(define prepend-all
```

```
(set *maximum-print-sequence-size* 50)

(permute [a b c d])
```

Output:

```
[[a b c d] [a b d c] [a c b d] [a c d b] [a d c b] [a d b c] [b a c d] [b a d c] [b c a d] [b c d a] [b d c a] [b d a c] [c b a
d] [c b d a] [c a b d] [c a d b] [c d a b] [c d b a] [d c b a] [d c a b] [d b c a] [d b a c] [d a b c] [d a c b]]
```

For lexical order, make a small change:

```
(define permute-helper
_ [] -> []
Done [X|Rest] -> (append (prepend-all X (permute (append Done Rest))) (permute-helper (append Done [X]) Rest))
)
```

Sidef

Built-in

```
[0,1,2].permutations { |*a|
    say a
}
```

Iterative

```
func forperm(callback, n) {
    var idx = @^n

    loop {
        callback(idx...)

        var p = n-1
        while (idx[p-1] > idx[p]) {--p}
        p == 0 && return()

        var d = p
        idx += idx.splice(p).reverse

        while (idx[p-1] > idx[d]) {++d}
        idx.swap(p-1, d)
    }

    return()
}

forperm({|*p| say p }, 3)
```

Recursive

```
func permutations(callback, set, perm=[]) {
    set || callback(perm)
    for i in ^set {
        __FUNC__(callback, [
            set[^i, i+1 ..^ set.len]
```

```
}

permutations({|p| say p }, [0,1,2])
```

Output:

```
[0, 1, 2]
[0, 2, 1]
[1, 0, 2]
[1, 2, 0]
[2, 0, 1]
[2, 1, 0]
```

Smalltalk

Works with: Squeak
Works with: Pharo

```
(1 to: 4) permutationsDo: [ :x |
    Transcript show: x printString; cr ].
```

Works with: GNU Smalltalk

```
ArrayedCollection extend [

    permuteAndDo: aBlock
        ["Permute receiver in-place, and call aBlock.
        Requires integer keys."
        self permuteUpto: self size andDo: aBlock]

    permuteUpto: n andDo: aBlock
        [n = 0 ifTrue: [^aBlock value].
        1 to: n do:
            [:i |
            self swap: i with: n.
            self permuteUpto: n-1 andDo: aBlock.
            self swap: i with: n]]
]

SequenceableCollection extend [

    permutations
        ["Answer a ReadStream of permuted shallow copies of receiver."
        | c |
        c := MappedCollection
            collection: self
            map: self keys asArray.
        ^Generator on:
            [:g |
            c map permuteAndDo: [g yield: (c copyFrom: 1 to: c size)]]]
```

Use example:

```
st> 'Abc' permutations contents
('bcA' 'cbA' 'cAb' 'Acb' 'bAc' 'Abc' )
```

Standard ML

```
fun interleave x [] = [[x]]
  | interleave x (y::ys) = (x::y::ys) :: (List.map (fn a => y::a) (interleave x ys))
```

```
fun perms [] = [[]]
  | perms (x::xs) = List.concat (List.map (interleave x) (perms xs))
```

Stata

Program to build a dataset containing all permutations of 1...n. Each permutation is stored as an observation.

For instance:

```
perm 4
```

Program

```
program perm
    local n=`1'
    local r=1
    forv i=1/`n' {
        local r=`r'*`i'
    }
    clear
    qui set obs `r'
    forv i=1/`n' {
        gen p`i'=0
    }
    mata: genperm()
end

mata
void genperm() {
    real scalar n, i, j, k, s, p
    real rowvector u
    st_view(a=., ., .)
    n = cols(a)
    u = 1..n
    p = 1
    do {
        a[p++, .] = u
        for (i = n; i > 1; i--) {
            if (u[i-1] < u[i]) break
        }
        if (i > 1) {
            j = i
            k = n
            while (j < k) u[(j++, k--)] = u[(k, j)]

            s = u[i-1]
            for (j = i; u[j] < s; j++) {
            }
            u[i-1] = u[j]
            u[j] = s
        }
    } while (i > 1)
}
end
```

Swift

```
func perms<T>(var ar: [T]) -> [[T]] {
  return heaps(&ar, ar.count)
}

func heaps<T>(inout ar: [T], n: Int) -> [[T]] {
  return n == 1 ? [ar] :
```

```
      return shuffles
  }
}

perms([1, 2, 3]) // [[1, 2, 3], [2, 1, 3], [3, 1, 2], [1, 3, 2], [2, 3, 1], [3, 2, 1]]
```

Tailspin

This solution seems to be the same as the Kotlin solution. Permutations flow independently without being collected until the end.

```
templates permutations
  when <=1> do [1] !
  otherwise
    def n: $;
    templates expand
      def p: $;
      1..$n -> \(def k: $;
        [$p(1..$k-1)..., $n, $p($k..last)...] !\) !
    end expand
    $n - 1 -> permutations -> expand !
end permutations

def alpha: ['ABCD'...];
[ $alpha::length -> permutations -> '$alpha($)...;' ] -> !OUT::write
```

Output:

```
[DCBA, CDBA, CBDA, CBAD, DBCA, BDCA, BCDA, BCAD, DBAC, BDAC, BADC, BACD, DCAB, CDAB, CADB, CABD, DACB, ADCB, ACDB, ACBD, DABC,
ADBC, ABDC, ABCD]
```

If we collect all the permutations of the next size down, we can output permutations in lexical order

```
templates lexicalPermutations
  when <=1> do [1] !
  otherwise
    def n: $;
    def p: [ $n - 1 -> lexicalPermutations ];
    1..$n -> \(def k: $;
      $p... -> [ $k, $... -> \(when <$k..> do $+1! otherwise $!\)] !\) !
end lexicalPermutations

def alpha: ['ABCD'...];
[ $alpha::length -> lexicalPermutations -> '$alpha($)...;' ] -> !OUT::write
```

Output:

```
[ABCD, ABDC, ACBD, ACDB, ADBC, ADCB, BACD, BADC, BCAD, BCDA, BDAC, BDCA, CABD, CADB, CBAD, CBDA, CDAB, CDBA, DABC, DACB, DBAC,
DBCA, DCAB, DCBA]
```

That algorithm can also be written from the bottom up to produce an infinite stream of sets of larger and larger permutations, until we stop

```
templates lexicalPermutations2
  def N: $;
  [[1]] -> #
  when <[<[]($N)>]> do $... !
  otherwise
    def tails: $;
```

```
    \)] -> #
end lexicalPermutations2

def alpha: ['ABCD'...];
[ $alpha::length -> lexicalPermutations2 -> '$alpha($)...;' ] -> !OUT::write
```

Output:

```
[ABCD, ABDC, ACBD, ACDB, ADBC, ADCB, BACD, BADC, BCAD, BCDA, BDAC, BDCA, CABD, CADB, CBAD, CBDA, CDAB, CDBA, DABC, DACB, DBAC,
DBCA, DCAB, DCBA]
```

The solutions above create a lot of new arrays at various stages. We can also use mutable state and just emit a copy for each generated solution.

```
templates perms
  templates findPerms
    when <$@perms::length..> do $@perms !
    otherwise
      def index: $;
      $index..$@perms::length
      -> \(
          @perms([$, $index]): $@perms([$index, $])...;
          $index + 1 -> findPerms !
      \) !
      @perms([last, $index..last-1]): $@perms($index..last)...;
  end findPerms
  @: [1..$];
  1 -> findPerms !
end perms

def alpha: ['ABCD'...];
[4 -> perms -> '$alpha($)...;' ] -> !OUT::write
```

Output:

```
[ABCD, ABDC, ACBD, ACDB, ADBC, ADCB, BACD, BADC, BCAD, BCDA, BDAC, BDCA, CABD, CADB, CBAD, CBDA, CDAB, CDBA, DABC, DACB, DBAC,
DBCA, DCAB, DCBA]
```

Tcl

Library: Tcllib (**Package:** struct::list)

```
package require struct::list

# Make the sequence of digits to be permuted
set n [lindex $argv 0]
for {set i 1} {$i <= $n} {incr i} {lappend sequence $i}

# Iterate over the permutations, printing as we go
struct::list foreachperm p $sequence {
    puts $p
}
```

Testing with `tclsh listPerms.tcl 3` produces this output:

```
1 2 3
1 3 2
2 1 3
2 3 1
```

```
3 1 2
3 2 1
```

UNIX Shell

Works with: Bourne Again SHell
Works with: Korn Shell

Straightforward implementation of Heap's algorithm operating in-place on an array local to the `permute` function.

```
function permute {
  if (( $# == 1 )); then
    set -- $(seq $1)
  fi
  local A=("$@")
  permuteAn "$#"
}

function permuteAn {
  # print all permutations of first n elements of the array A, with remaining
  # elements unchanged.
  local -i n=$1 i
  shift
  if (( n == 1 )); then
    printf '%s\n' "${A[*]}"
  else
    permuteAn $(( n-1 ))
    for (( i=0; i<n-1; ++i )); do
      local -i k
      (( k=n%2 ? 0: i ))
      local t=${A[k]}
      A[k]=${A[n-1]}
      A[n-1]=$t
      permuteAn $(( n-1 ))
    done
  fi
}
```

For Zsh the array indices need to be bumped by 1 inside the `permuteAn` function:

Works with: Z Shell

```
function permuteAn {
  # print all permutations of first n elements of the array A, with remaining
  # elements unchanged.
  local -i n=$1 i
  shift
  if (( n == 1 )); then
    printf '%s\n' "${A[*]}"
  else
    permuteAn $(( n-1 ))
    for (( i=1; i<n; ++i )); do
      local -i k
      (( k=n%2 ? 1 : i ))
      local t=$A[k]
      A[k]=$A[n]
      A[n]=$t
      permuteAn $(( n-1 ))
    done
  fi
}
```

Output:

```
$ permute 4
permute 4
1 2 3 4
2 1 3 4
3 1 2 4
1 3 2 4
2 3 1 4
3 2 1 4
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
```

Ursala

In practice there's no need to write this because it's in the standard library.

```
#import std

permutations =

~&itB^?a(                     # are both the input argument list and its tail non-empty?
   @ahPfatPRD *= refer ^C(      # yes, recursively generate all permutations of the tail, and for each one
      ~&a,                        # insert the head at the first position
      ~&ar&& ~&arh2falrtPXPRD),   # if the rest is non-empty, recursively insert at all subsequent positions
   ~&aNC)                       # no, return the singleton list of the argument
```

test program:

```
#cast %nLL

test = permutations <1,2,3>
```

Output:

```
<
   <1,2,3>,
   <2,1,3>,
   <2,3,1>,
   <1,3,2>,
   <3,1,2>,
   <3,2,1>>
```

VBA

Translation of: Pascal

```
Dim P() As Integer
Dim t As Integer, i As Integer, j As Integer, k As Integer
Dim count As Long
Dim Last As Boolean

If n <= 1 Then

  Debug.Print "Please give a number greater than 1"
  Exit Sub

End If

'Initialize
ReDim P(n)

For i = 1 To n
  P(i) = i
Next

count = 0
Last = False

Do While Not Last
   'print?
   If printem Then

      For t = 1 To n
        Debug.Print P(t);
      Next

      Debug.Print

   End If

count = count + 1

Last = True
i = n - 1

   Do While i > 0

     If P(i) < P(i + 1) Then

       Last = False
       Exit Do

     End If

     i = i - 1
   Loop

  j = i + 1
  k = n

  While j < k
    ' Swap p(j) and p(k)
    t = P(j)
    P(j) = P(k)
    P(k) = t
    j = j + 1
    k = k - 1
  Wend

  j = n

  While P(j) > P(i)
    j = j - 1
  Wend

  j = j + 1
  'Swap p(i) and p(j)
  t = P(i)
  P(i) = P(j)
  P(j) = t
Loop 'While not last
```

```
End Sub
```

Sample dialogue:

```
permute 1
give a number greater than 1!
permute 2
 1  2
 2  1
Number of permutations:  2
permute 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
Number of permutations:  24
permute 10,False
Number of permutations:  3628800
```

VBScript

A recursive implementation. Arrays can contain anything, I stayed with with simple variables. (Elements could be arrays but then the printing routine should be recursive...)

```
'permutation ,recursive
a=array("Hello",1,True,3.141592)
cnt=0
perm a,0
wscript.echo vbcrlf &"Count " & cnt

sub print(a)
  s=""
  for i=0  to ubound(a):
    s=s &" " & a(i):
  next:
  wscript.echo s :
  cnt=cnt+1 :
end sub
sub swap(a,b) t=a: a=b :b=t:  end sub

sub perm(byval a,i)
   if i=ubound(a) then print a: exit sub
   for j= i to ubound(a)
      swap a(i),a(j)
      perm a,i+1
     swap a(i),a(j)
```

Output

```
Hello 1 Verdadero 3.141592
Hello 1 3.141592 Verdadero
Hello Verdadero 1 3.141592
Hello Verdadero 3.141592 1
Hello 3.141592 Verdadero 1
Hello 3.141592 1 Verdadero
1 Hello Verdadero 3.141592
1 Hello 3.141592 Verdadero
1 Verdadero Hello 3.141592
1 Verdadero 3.141592 Hello
1 3.141592 Verdadero Hello
1 3.141592 Hello Verdadero
Verdadero 1 Hello 3.141592
Verdadero 1 3.141592 Hello
Verdadero Hello 1 3.141592
Verdadero Hello 3.141592 1
Verdadero 3.141592 Hello 1
Verdadero 3.141592 1 Hello
3.141592 1 Verdadero Hello
3.141592 1 Hello Verdadero
3.141592 Verdadero 1 Hello
3.141592 Verdadero Hello 1
3.141592 Hello Verdadero 1
3.141592 Hello 1 Verdadero

Count 24
```

Wren

Recursive

Translation of: Kotlin

```
var permute // recursive
permute = Fn.new { |input|
    if (input.count == 1) return [input]
    var perms = []
    var toInsert = input[0]
    for (perm in permute.call(input[1..-1])) {
        for (i in 0..perm.count) {
            var newPerm = perm.toList
            newPerm.insert(i, toInsert)
            perms.add(newPerm)
        }
    }
    return perms
}

var input = [1, 2, 3]
var perms = permute.call(input)
System.print("There are %(perms.count) permutations of %(input), namely:\n")
perms.each { |perm| System.print(perm) }
```

Output:

```
There are 6 permutations of [1, 2, 3], namely:

[1, 2, 3]
[2, 1, 3]
[2, 3, 1]
[1, 3, 2]
```

```
[3, 1, 2]
[3, 2, 1]
```

Iterative, lexicographical order

Translation of: Go
Library: Wren-math

Output modified to follow the pattern of the recursive version.

```
import "./math" for Int

var input = [1, 2, 3]
var perms = [input]
var a = input.toList
var n = a.count - 1
for (c in 1...Int.factorial(n+1)) {
    var i = n - 1
    var j = n
    while (a[i] > a[i+1]) i = i - 1
    while (a[j] < a[i])   j = j - 1
    var t = a[i]
    a[i] = a[j]
    a[j] = t
    j = n
    i = i + 1
    while (i < j) {
        t = a[i]
        a[i] = a[j]
        a[j] = t
        i = i + 1
        j = j - 1
    }
    perms.add(a.toList)
}
System.print("There are %(perms.count) permutations of %(input), namely:\n")
perms.each { |perm| System.print(perm) }
```

Output:

```
There are 6 permutations of [1, 2, 3], namely:

[1, 2, 3]
[1, 3, 2]
[2, 1, 3]
[2, 3, 1]
[3, 1, 2]
[3, 2, 1]
```

Library based

Library: Wren-perm

```
import "./perm" for Perm

var a = [1, 2, 3]
System.print(Perm.list(a))    // not lexicographic
System.print()
System.print(Perm.listLex(a)) // lexicographic
```

Output:

```
[[1, 2, 3], [1, 3, 2], [2, 1, 3], [2, 3, 1], [3, 1, 2], [3, 2, 1]]
```

XPLo

```
code ChOut=8, CrLf=9;
def  N=4;                       \number of objects (letters)
char S0, S1(N);

proc Permute(D);                \Display all permutations of letters in S0
int D;                          \depth of recursion
int I, J;
[if D=N then
        [for I:= 0 to N-1 do ChOut(0, S1(I));
        CrLf(0);
        return;
        ];
for I:= 0 to N-1 do
        [for J:= 0 to D-1 do    \check if object (letter) already used
                if S1(J) = S0(I) then J:=100;
        if J<100 then
                [S1(D):= S0(I); \object (letter) not used so append it
                Permute(D+1);   \recurse next level deeper
                ];
        ];
];

[S0:= "rose ";                  \N different objects (letters)
Permute(0);                     \(space char avoids MSb termination)
]
```

Output:

```
rose
roes
rsoe
rseo
reos
reso
orse
ores
osre
oser
oers
oesr
sroe
sreo
sore
soer
sero
seor
eros
erso
eors
eosr
esro
esor
```

zkl

Using the solution from task Permutations by swapping#zkl:

```
zkl: Utils.Helpers.permute("rose").apply("concat")
L("rose","roes","reos","eros","erso","reso","rseo","rsoe","sroe","sreo",...)
```

```
zkl: Utils.Helpers.permute(T(1,2,3,4))
L(L(1,2,3,4),L(1,2,4,3),L(1,4,2,3),L(4,1,2,3),L(4,1,3,2),L(1,4,3,2),L(1,3,4,2),L(1,3,2,4),...)
```