Combinations

Task

Given non-negative integers **m** and **n**, generate all size **m** combinations (http://mathworld.wolfram.com/Combination.html) of the integers from **0** (zero) to **n-1** in sorted order (each combination is sorted and the entire table is sorted).

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

Example

3 comb **5** is:

```
0 1 2
0 1 3
0 1 4
0 2 3
0 2 4
0 3 4
1 2 3
1 2 4
1 3 4
2 3 4
```

If it is more "natural" in your language to start counting from **1** (unity) instead of **0** (zero), the combinations can be of the integers from **1** to **n**.

See also

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

Translation of: D

```
F comb(arr, k)
   I k == 0
      R [[Int]()]

   [[Int]] result
   L(x) arr
      V i = L.index
      L(suffix) comb(arr[i+1..], k-1)
         result [+]= x [+] suffix

   R result

print(comb([0, 1, 2, 3, 4], 3))
```

Output:

```
[[0, 1, 2], [0, 1, 3], [0, 1, 4], [0, 2, 3], [0, 2, 4], [0, 3, 4], [1, 2, 3], [1, 2, 4], [1, 3, 4], [2, 3, 4]]
```

360 Assembly

Translation of: C

Nice algorithm without recursion borrowed from C. Recursion is elegant but iteration is efficient. For maximum compatibility, this program uses only the basic instruction set (S/360) and two ASSIST macros (XDECO, XPRNT) to keep the code as short as possible.

```
*        Combinations              26/05/2016
COMBINE  CSECT
         USING  COMBINE,R13        base register
         B      72(R15)            skip savearea
         DC     17F'0'             savearea
         STM    R14,R12,12(R13)    prolog
         ST     R13,4(R15)         "
         ST     R15,8(R13)         "
         LR     R13,R15            "
         SR     R3,R3              clear
         LA     R7,C               @c(1)
         LH     R8,N               v=n
LOOPI1   STC    R8,0(R7)           do i=1 to n; c(i)=n-i+1
         LA     R7,1(R7)             @c(i)++
         BCT    R8,LOOPI1          next i
LOOPBIG  LA     R10,PG             big loop {------------------
         LH     R1,N               n
         LA     R7,C-1(R1)         @c(i)
         LH     R6,N               i=n
LOOPI2   IC     R3,0(R7)           do i=n to 1 by -1; r2=c(i)
         XDECO  R3,PG+80             edit c(i)
         MVC    0(2,R10),PG+90       output c(i)
         LA     R10,3(R10)           @pgi=@pgi+3
         BCTR   R7,0                 @c(i)--
         BCT    R6,LOOPI2          next i
         XPRNT  PG,80              print buffer
         LA     R7,C               @c(1)
         LH     R8,M               v=m
         LA     R6,1               i=1
LOOPI3   LR     R1,R6              do i=1 by 1; r1=i
```

```
         BNL    ELOOPBIG            exit loop
         BCTR   R8,0                v=v-1
         LA     R7,1(R7)            @c(i)++
         LA     R6,1(R6)            i=i+1
         B      LOOPI3            next i
ELOOPI3  LR     R1,R6             i
         LA     R4,C-1(R1)        @c(i)
         IC     R3,0(R4)          c(i)
         LA     R3,1(R3)          c(i)+1
         STC    R3,0(R4)          c(i)=c(i)+1
         BCTR   R7,0              @c(i)--
LOOPI4   CH     R6,=H'2'          do i=i to 2 by -1
         BL     ELOOPI4           leave i
         IC     R3,1(R7)            c(i)
         LA     R3,1(R3)            c(i)+1
         STC    R3,0(R7)            c(i-1)=c(i)+1
         BCTR   R7,0                @c(i)--
         BCTR   R6,0                i=i-1
         B      LOOPI4            next i
ELOOPI4  B      LOOPBIG           big loop }------------------
ELOOPBIG L      R13,4(0,R13)      epilog
         LM     R14,R12,12(R13)   "
         XR     R15,R15           "
         BR     R14               exit
M        DC     H'5'              <=input
N        DC     H'3'              <=input
C        DS     64X               array of 8 bit integers
PG       DC     CL92' '           buffer
         YREGS
         END    COMBINE
```

Output:

```
 1  2  3
 1  2  4
 1  2  5
 1  3  4
 1  3  5
 1  4  5
 2  3  4
 2  3  5
 2  4  5
 3  4  5
```

Acornsoft Lisp

Translation of: Emacs Lisp

```
(defun comb (m n (i . 0))
  (cond ((zerop m) '(()))
        ((eq i n) '())
        (t (append
             (mapc '(lambda (rest) (cons i rest))
                   (comb (sub1 m) n (add1 i)))
             (comb m n (add1 i))))))

(defun append (a b)
  (cond ((null a) b)
        (t (cons (car a) (append (cdr a) b)))))

(map print (comb 3 5))
```

Output:

```
(0 1 2)
(0 1 3)
```

```
(1 2 3)
(1 2 4)
(1 3 4)
(2 3 4)
```

Action!

```
PROC PrintComb(BYTE ARRAY c BYTE len)
  BYTE i

  Put('()
  FOR i=0 TO len-1
  DO
    IF i>0 THEN Put(',) FI
    PrintB(c(i))
  OD
  Put(')) PutE()
RETURN

BYTE FUNC Increasing(BYTE ARRAY c BYTE len)
  BYTE i

  IF len<2 THEN RETURN (1) FI

  FOR i=0 TO len-2
  DO
    IF c(i)>=c(i+1) THEN
      RETURN (0)
    FI
  OD
RETURN (1)

BYTE FUNC NextComb(BYTE ARRAY c BYTE n,k)
  INT pos,i

  DO
    pos=k-1
    DO
      c(pos)==+1
      IF c(pos)<n THEN
        EXIT
      ELSE
        pos==-1
        IF pos<0 THEN RETURN (0) FI
      FI
      FOR i=pos+1 TO k-1
      DO
        c(i)=c(pos)
      OD
    OD
  UNTIL Increasing(c,k)
  OD
RETURN (1)

PROC Comb(BYTE n,k)
  BYTE ARRAY c(10)
  BYTE i

  IF k>n THEN
    Print("Error! k is greater than n.")
    Break()
  FI

  FOR i=0 TO k-1
  DO
    c(i)=i
  OD

  DO
    PrintComb(c,k)
  UNTIL NextComb(c,n,k)=0
```

```
PROC Main()
  Comb(5,3)
RETURN
```

Output:

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

```
(0,1,2)
(0,1,3)
(0,1,4)
(0,2,3)
(0,2,4)
(0,3,4)
(1,2,3)
(1,2,4)
(1,3,4)
(2,3,4)
```

Ada

```
with Ada.Text_IO;  use Ada.Text_IO;

procedure Test_Combinations is
   generic
      type Integers is range <>;
   package Combinations is
      type Combination is array (Positive range <>) of Integers;
      procedure First (X : in out Combination);
      procedure Next (X : in out Combination);
      procedure Put (X : Combination);
   end Combinations;

   package body Combinations is
      procedure First (X : in out Combination) is
      begin
         X (1) := Integers'First;
         for I in 2..X'Last loop
            X (I) := X (I - 1) + 1;
         end loop;
      end First;
      procedure Next (X : in out Combination) is
      begin
         for I in reverse X'Range loop
            if X (I) < Integers'Val (Integers'Pos (Integers'Last) - X'Last + I) then
               X (I) := X (I) + 1;
               for J in I + 1..X'Last loop
                  X (J) := X (J - 1) + 1;
               end loop;
               return;
            end if;
         end loop;
         raise Constraint_Error;
      end Next;
      procedure Put (X : Combination) is
      begin
         for I in X'Range loop
            Put (Integers'Image (X (I)));
         end loop;
      end Put;
   end Combinations;

   type Five is range 0..4;
   package Fives is new Combinations (Five);
   use Fives;

   X : Combination (1..3);
```

```
      Put (X); New_Line;
      Next (X);
   end loop;
exception
   when Constraint_Error =>
      null;
end Test_Combinations;
```

The solution is generic the formal parameter is the integer type to make combinations of. The type range determines *n*. In the example it is

```
type Five is range 0..4;
```

The parameter *m* is the object's constraint. When $n < m$ the procedure First (selects the first combination) will propagate Constraint_Error. The procedure Next selects the next combination. Constraint_Error is propagated when it is the last one.

Output:

```
 0 1 2
 0 1 3
 0 1 4
 0 2 3
 0 2 4
 0 3 4
 1 2 3
 1 2 4
 1 3 4
 2 3 4
```

ALGOL 68

Translation of: Python
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_combinations.a68

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

COMMENT REQUIRED BY "prelude_combinations_generative.a68"
  MODE COMBDATA = ~;
PROVIDES:
# COMBDATA*=~* #
# comb*=~ list* #
END COMMENT

MODE COMBDATALIST = REF[]COMBDATA;
MODE COMBDATALISTYIELD = PROC(COMBDATALIST)VOID;

PROC comb gen combinations = (INT m, COMBDATALIST list, COMBDATALISTYIELD yield)VOID:(
  CASE m IN
  # case 1: transpose list #
    FOR i TO UPB list DO yield(list[i]) OD
  OUT
    [m + LWB list - 1]COMBDATA out;
    INT index out := 1;
    FOR i TO UPB list DO
      COMBDATA first = list[i];
    # FOR COMBDATALIST sub recombination IN # comb gen combinations(m - 1, list[i+1:] #) DO (#
```

```
        yield(out)
    # OD #))
    OD
  ESAC
);

SKIP
```

File: test_combinations.a68

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

CO REQUIRED BY "prelude_combinations.a68" CO
  MODE COMBDATA = INT;
#PROVIDES:#
# COMBDATA~=INT~ #
# comb ~=int list ~#
PR READ "prelude_combinations.a68" PR;

FORMAT data fmt = $g(0)$;

main:(
  INT m = 3;
  FORMAT list fmt = $"("n(m-1)(f(data fmt)",")f(data fmt)")"$;
  FLEX[0]COMBDATA test data list := (1,2,3,4,5);
# FOR COMBDATALIST recombination data IN # comb gen combinations(m, test data list #) DO (#,
##    (COMBDATALIST recombination)VOID:(
    printf ((list fmt, recombination, $l$))
# OD # ))
)
```

Output:

```
(1,2,3)
(1,2,4)
(1,2,5)
(1,3,4)
(1,3,5)
(1,4,5)
(2,3,4)
(2,3,5)
(2,4,5)
(3,4,5)
```

AppleScript

Iteration

```
on comb(n, k)
    set c to {}
    repeat with i from 1 to k
        set end of c to i's contents
    end repeat
    set r to {c's contents}
    repeat while my next_comb(c, k, n)
        set end of r to c's contents
    end repeat
    return r
end comb

on next_comb(c, k, n)
    set i to k
    set c's item i to (c's item i) + 1
```

```
    end repeat
    if (c's item 1 > n - k + 1) then return false
    repeat with i from i + 1 to k
        set c's item i to (c's item (i - 1)) + 1
    end repeat
    return true
end next_comb

return comb(5, 3)
```

Output:

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

Functional composition

Translation of: JavaScript

```
---------------------- COMBINATIONS ---------------------

-- comb :: Int -> [a] -> [[a]]
on comb(n, lst)
    if 1 > n then
        {{}}
    else
        if not isNull(lst) then
            set {h, xs} to uncons(lst)

            map(cons(h), ¬
                comb(n - 1, xs)) & comb(n, xs)
        else
            {}
        end if
    end if
end comb

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

    intercalate(linefeed, ¬
        map(unwords, comb(3, enumFromTo(0, 4))))

end run

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

-- cons :: a -> [a] -> [a]
on cons(x)
    script
        on |λ|(xs)
            {x} & xs
        end |λ|
    end script
end cons

-- 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
        lst
    else
        {}
    end if
end enumFromTo

-- intercalate :: Text -> [Text] -> Text
```

```
    set my text item delimiters to dlm
    return strJoined
end intercalate

-- isNull :: [a] -> Bool
on isNull(xs)
    if class of xs is string then
        xs = ""
    else
        xs = {}
    end if
end isNull

-- map :: (a -> b) -> [a] -> [b]
on map(f, xs)
    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

-- 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)
    set lng to length of xs
    if lng > 0 then
        if class of xs is string then
            set cs to text items of xs
            {item 1 of cs, rest of cs}
        else
            {item 1 of xs, rest of xs}
        end if
    else
        missing value
    end if
end uncons

-- unwords :: [String] -> String
on unwords(xs)
    intercalate(space, xs)
end unwords
```

Output:

```
0 1 2
0 1 3
0 1 4
0 2 3
0 2 4
0 3 4
1 2 3
1 2 4
```

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

Arturo

```
print.lines combine.by:3 @0..4
```

Output:

```
[0 1 2]
[0 1 3]
[0 1 4]
[0 2 3]
[0 2 4]
[0 3 4]
[1 2 3]
[1 2 4]
[1 3 4]
[2 3 4]
```

AutoHotkey

contributed by Laszlo on the ahk forum (http://www.autohotkey.com/forum/post-276224.html#276224)

```
MsgBox % Comb(1,1)
MsgBox % Comb(3,3)
MsgBox % Comb(3,2)
MsgBox % Comb(2,3)
MsgBox % Comb(5,3)

Comb(n,t) { ; Generate all n choose t combinations of 1..n, lexicographically
   IfLess n,%t%, Return
   Loop %t%
      c%A_Index% := A_Index
   i := t+1, c%i% := n+1

   Loop {
      Loop %t%
         i := t+1-A_Index, c .= c%i% " "
      c .= "`n"     ; combinations in new lines
      j := 1, i := 2
      Loop
         If (c%j%+1 = c%i%)
            c%j% := j, ++j, ++i
         Else Break
      If (j > t)
         Return c
      c%j% += 1
   }
}
```

AWK

```
BEGIN {
    ## Default values for r and n (Choose 3 from pool of 5).  Can
    ## alternatively be set on the command line:-
    ## awk -v r=<number of items being chosen> -v n=<how many to choose from> -f <scriptname>
    if (length(r) == 0) r = 3
    if (length(n) == 0) n = 5

    for (i=1; i<=r; i++) { ## First combination of items:
```

```
    ## While 1st item is less than its maximum permitted value...
    while (A[1] < n - r + 1) {
        ## loop backwards through all items in the previous
        ## combination of items until an item is found that is
        ## less than its maximum permitted value:
        for (i = r; i >= 1; i--) {
            ## If the equivalently positioned item in the
            ## previous combination of items is less than its
            ## maximum permitted value...
            if (A[i] < n - r + i) {
                ## increment the current item by 1:
                A[i]++
                ## Save the current position-index for use
                ## outside this "for" loop:
                p = i
                break}}
        ## Put consecutive numbers in the remainder of the array,
        ## counting up from position-index p.
        for (i = p + 1; i <= r; i++) A[i] = A[i - 1] + 1

        ## Print the current combination of items:
        for (i=1; i <= r; i++) {
            if (i < r) printf A[i] OFS
            else print A[i]}}
    exit}
```

Usage:

```
awk -v r=3 -v n=5 -f combn.awk
```

Output:

```
1 2 3
1 2 4
1 2 5
1 3 4
1 3 5
1 4 5
2 3 4
2 3 5
2 4 5
3 4 5
```

BASIC

BASIC256

```
input "Enter n comb m. ", n
input m

outstr$ = ""
call iterate (outstr$, 0, m-1, n-1)
end

subroutine iterate (curr$, start, stp, depth)
    for i = start to stp
        if depth = 0 then print curr$ + " " + string(i)
        call iterate (curr$ + " " + string(i), i+1, stp, depth-1)
    next i
end subroutine
```

Output:

```
Enter n comb m. 3
5
 0 1 2
 0 1 3
 0 1 4
 0 2 3
 0 2 4
 0 3 4
 1 2 3
 1 2 4
 1 3 4
 2 3 4
```

IS-BASIC

```
100 PROGRAM "Combinat.bas"
110 LET MMAX=3:LET NMAX=5
120 NUMERIC COMB(0 TO MMAX)
130 CALL GENERATE(1)
140 DEF GENERATE(M)
150   NUMERIC N,I
160   IF M>MMAX THEN
170     FOR I=1 TO MMAX
180       PRINT COMB(I);
190     NEXT
200     PRINT
220   ELSE
230     FOR N=0 TO NMAX-1
240       IF M=1 OR N>COMB(M-1) THEN
250         LET COMB(M)=N
260         CALL GENERATE(M+1)
270       END IF
280     NEXT
290   END IF
300 END DEF
```

QBasic

Works with: QBasic version 1.1
Works with: QuickBasic version 4.5

```
SUB iterate (curr$, start, stp, depth)
  FOR i = start TO stp
    IF depth = 0 THEN PRINT curr$ + " " + STR$(i)
    CALL iterate(curr$ + " " + STR$(i), i + 1, stp, depth - 1)
  NEXT i
END SUB

INPUT "Enter n comb m.  ", n, m

outstr$ = ""
CALL iterate(outstr$, 0, m - 1, n - 1)
END
```

Output:

```
Same as FreeBASIC entry.
```

Run BASIC

```
    call iterate curr$ + " " + str$(i), i+1, stp, depth-1
  next i
end sub

input "Enter n comb m. "; n, m
outstr$ = ""
call iterate outstr$, 0, m-1, n-1
end
```

XBasic

Works with: Windows XBasic

```
PROGRAM "Combinations"
VERSION "0.0000"

DECLARE FUNCTION Entry ()
DECLARE FUNCTION iterate (curr$, start, stp, depth)

FUNCTION Entry ()
n = 3
m = 5
outstr$ = ""
iterate(outstr$, 0, m - 1, n - 1)

END FUNCTION

FUNCTION iterate (curr$, start, stp, depth)
  FOR i = start TO stp
  IF depth = 0 THEN PRINT curr$ + " " + STR$(i)
  iterate(curr$ + " " + STR$(i), i + 1, stp, depth - 1)
  NEXT i
  RETURN
END FUNCTION
END PROGRAM
```

BBC BASIC

Works with: BBC BASIC for Windows

```
      INSTALL @lib$+"SORTLIB"
      sort% = FN_sortinit(0,0)

      M% = 3
      N% = 5

      C% = FNfact(N%)/(FNfact(M%)*FNfact(N%-M%))
      DIM s$(C%)
      PROCcomb(M%, N%, s$())

      CALL sort%, s$(0)
      FOR I% = 0 TO C%-1
        PRINT s$(I%)
      NEXT
      END

      DEF PROCcomb(C%, N%, s$())
      LOCAL I%, U%
      FOR U% = 0 TO 2^N%-1
        IF FNbits(U%) = C% THEN
          s$(I%) = FNlist(U%)
          I% += 1
        ENDIF
      NEXT
      ENDPROC

      DEF FNbits(U%)
      LOCAL N%
```

```
      ENDWHILE
      = N%

      DEF FNlist(U%)
      LOCAL N%, s$
      WHILE U%
        IF U% AND 1 s$ += STR$(N%) + " "
        N% += 1
        U% = U% >> 1
      ENDWHILE
      = s$

      DEF FNfact(N%)
      IF N%<=1 THEN = 1 ELSE = N%*FNfact(N%-1)
```

BQN

```
Cmat←{⊑w∊0‿_x?≍↕w;0-∘∾˘-∘∾´1+(w-1‿_0)𝕊¨x-1} # Recursive
Cmat1←{k←◶↕d←x¬w⋄∾◶{k∾˘¨∾¯`1+x}⍟wd↑↓1‿_0↩0} # Roger Hui
```

```
┌─
  0 1 2
  0 1 3
  0 1 4
  0 2 3
  0 2 4
  0 3 4
  1 2 3
  1 2 4
  1 3 4
  2 3 4
      ┘
```

Bracmat

The program first constructs a pattern with `m` variables and an expression that evaluates `m` variables into a combination. Then the program constructs a list of the integers `0 ... n-1`. The real work is done in the expression `!list:!pat`. When a combination is found, it is added to the list of combinations. Then we force the program to backtrack and find the next combination by evaluating the always failing `~`. When all combinations are found, the pattern fails and we are in the rhs of the last `|` operator.

```
(comb=
  bvar combination combinations list m n pat pvar var
.     !arg:(?m.?n)
    & ( pat
      =   ?
        & !combinations (.!combination):?combinations
        & ~
      )
    & :?list:?combination:?combinations
    &   whl
      ' ( !m+-1:~<0:?m
        & chu$(utf$a+!m):?var
        & glf$('(%@?.$var)):(=?pvar)
        & '(? ()$pvar ()$pat):(=?pat)
        & glf$('(!.$var)):(=?bvar)
        & (   '$combination:(=)
            & '$bvar:(=?combination)
          | '($bvar ()$combination):(=?combination)
          )
        )
    &   whl
      ' (!n+-1:~<0:?n&!n !list:?list)
```

```
comb$(3.5)
```

```
(.0 1 2)
(.0 1 3)
(.0 1 4)
(.0 2 3)
(.0 2 4)
(.0 3 4)
(.1 2 3)
(.1 2 4)
(.1 3 4)
(.2 3 4)
```

C

```c
#include <stdio.h>

/* Type marker stick: using bits to indicate what's chosen.  The stick can't
 * handle more than 32 items, but the idea is there; at worst, use array instead */
typedef unsigned long marker;
marker one = 1;

void comb(int pool, int need, marker chosen, int at)
{
    if (pool < need + at) return; /* not enough bits left */

    if (!need) {
        /* got all we needed; print the thing.  if other actions are
         * desired, we could have passed in a callback function. */
        for (at = 0; at < pool; at++)
            if (chosen & (one << at)) printf("%d ", at);
        printf("\n");
        return;
    }
    /* if we choose the current item, "or" (|) the bit to mark it so. */
    comb(pool, need - 1, chosen | (one << at), at + 1);
    comb(pool, need, chosen, at + 1);  /* or don't choose it, go to next */
}

int main()
{
    comb(5, 3, 0, 0);
    return 0;
}
```

Lexicographic ordered generation

Without recursions, generate all combinations in sequence. Basic logic: put n items in the first n of m slots; each step, if right most slot can be moved one slot further right, do so; otherwise find right most item that can be moved, move it one step and put all items already to its right next to it.

```c
#include <stdio.h>

void comb(int m, int n, unsigned char *c)
{
    int i;
    for (i = 0; i < n; i++) c[i] = n - i;

    while (1) {
        for (i = n; i--;)
            printf("%d%c", c[i], i ? ' ': '\n');

        /* this check is not strictly necessary, but if m is not close to n,
```

```
        for (; c[i] >= m - i;) if (++i >= n) return;
        for (c[i]++; i; i--) c[i-1] = c[i] + 1;
    }
}

int main()
{
    unsigned char buf[100];
    comb(5, 3, buf);
    return 0;
}
```

C#

```
using System;
using System.Collections.Generic;

public class Program
{
    public static IEnumerable<int[]> Combinations(int m, int n)
    {
            int[] result = new int[m];
            Stack<int> stack = new Stack<int>();
            stack.Push(0);

            while (stack.Count > 0)
           {
                int index = stack.Count - 1;
                int value = stack.Pop();

                while (value < n)
               {
                    result[index++] = ++value;
                    stack.Push(value);

                    if (index == m)
                    {
                        yield return result;
                        break;
                    }
                }
            }
    }

    static void Main()
    {
        foreach (int[] c in Combinations(3, 5))
        {
            Console.WriteLine(string.Join(",", c));
            Console.WriteLine();
        }
    }
}
```

Here is another implementation that uses recursion, intead of an explicit stack:

```
using System;
using System.Collections.Generic;

public class Program
{
  public static IEnumerable<int[]> FindCombosRec(int[] buffer, int done, int begin, int end)
  {
    for (int i = begin; i < end; i++)
    {
      buffer[done] = i;

      if (done == buffer.Length - 1)
```

```
            yield return child;
        }
    }

    public static IEnumerable<int[]> FindCombinations(int m, int n)
    {
        return FindCombosRec(new int[m], 0, 0, n);
    }

    static void Main()
    {
        foreach (int[] c in FindCombinations(3, 5))
        {
            for (int i = 0; i < c.Length; i++)
            {
                Console.Write(c[i] + " ");
            }
            Console.WriteLine();
        }
    }
}
```

Recursive version

```
using System;
class Combinations
{
    static int k = 3, n = 5;
    static int [] buf = new int [k];

    static void Main()
    {
        rec(0, 0);
    }

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

C++

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

void comb(int N, int K)
{
    std::string bitmask(K, 1); // K leading 1's
    bitmask.resize(N, 0); // N-K trailing 0's

    // print integers and permute bitmask
    do {
        for (int i = 0; i < N; ++i) // [0..N-1] integers
        {
            if (bitmask[i]) std::cout << " " << i;
        }
        std::cout << std::endl;
    } while (std::prev_permutation(bitmask.begin(), bitmask.end()));
```

```
{
    comb(5, 3);
}
```

Output:

```
 0 1 2
 0 1 3
 0 1 4
 0 2 3
 0 2 4
 0 3 4
 1 2 3
 1 2 4
 1 3 4
 2 3 4
```

Clojure

```
(defn combinations
  "If m=1, generate a nested list of numbers [0,n)
   If m>1, for each x in [0,n), and for each list in the recursion on [x+1,n), cons the two"
  [m n]
  (letfn [(comb-aux
       [m start]
       (if (= 1 m)
         (for [x (range start n)]
           (list x))
         (for [x (range start n)
           xs (comb-aux (dec m) (inc x))]
           (cons x xs))))]
    (comb-aux m 0)))

(defn print-combinations
  [m n]
  (doseq [line (combinations m n)]
    (doseq [n line]
      (printf "%s " n))
    (printf "%n")))
```

The below code do not comply to the task described above. However, the combinations of n elements taken from m elements might be more natural to be expressed as a set of unordered sets of elements in Clojure using its Set data structure.

```
(defn combinations
  "Generate the combinations of n elements from a list of [0..m)"
  [m n]
  (let [xs (range m)]
    (loop [i (int 0) res #{#{}}]
      (if (== i n)
        res
        (recur (+ 1 i)
               (set (for [x xs r res
                          :when (not-any? #{x} r)]
                      (conj r x))))))))
```

CLU

```
% generate the size-M combinations from 0 to n-1
combinations = iter (m, n: int) yields (sequence[int])
    if m<=n then
        state: array[int] := array[int]$predict(1, m)
```

```
        i: int := m
        while i>0 do
            yield (sequence[int]$a2s(state))
            i := m
            while i>0 do
                state[i] := state[i] + 1
                for j: int in int$from_to(i,m-1) do
                    state[j+1] := state[j] + 1
                end
                if state[i] < n-(m-i) then break end
                i := i - 1
            end
        end
    end
end combinations

% print a combination
print_comb = proc (s: stream, comb: sequence[int])
    for i: int in sequence[int]$elements(comb) do
        stream$puts(s, int$unparse(i) || " ")
    end
end print_comb

start_up = proc ()
    po: stream := stream$primary_output()
    for comb: sequence[int] in combinations(3, 5) do
        print_comb(po, comb)
        stream$putl(po, "")
    end
end start_up
```

Output:

```
0 1 2
0 1 3
0 1 4
0 2 3
0 2 4
0 3 4
1 2 3
1 2 4
1 3 4
2 3 4
```

CoffeeScript

Basic backtracking solution.

```
combinations = (n, p) ->
  return [ [] ] if p == 0
  i = 0
  combos = []
  combo = []
  while combo.length < p
    if i < n
      combo.push i
      i += 1
    else
      break if combo.length == 0
      i = combo.pop() + 1

    if combo.length == p
      combos.push clone combo
      i = combo.pop() + 1
  combos

clone = (arr) -> (n for n in arr)
```

```
  console.log "------ #{N} #{i}"
  for combo in combinations N, i
    console.log combo
```

Output:

```
> coffee combo.coffee
------ 5 0
[]
------ 5 1
[ 0 ]
[ 1 ]
[ 2 ]
[ 3 ]
[ 4 ]
------ 5 2
[ 0, 1 ]
[ 0, 2 ]
[ 0, 3 ]
[ 0, 4 ]
[ 1, 2 ]
[ 1, 3 ]
[ 1, 4 ]
[ 2, 3 ]
[ 2, 4 ]
[ 3, 4 ]
------ 5 3
[ 0, 1, 2 ]
[ 0, 1, 3 ]
[ 0, 1, 4 ]
[ 0, 2, 3 ]
[ 0, 2, 4 ]
[ 0, 3, 4 ]
[ 1, 2, 3 ]
[ 1, 2, 4 ]
[ 1, 3, 4 ]
[ 2, 3, 4 ]
------ 5 4
[ 0, 1, 2, 3 ]
[ 0, 1, 2, 4 ]
[ 0, 1, 3, 4 ]
[ 0, 2, 3, 4 ]
[ 1, 2, 3, 4 ]
------ 5 5
[ 0, 1, 2, 3, 4 ]
```

Common Lisp

```
(defun map-combinations (m n fn)
  "Call fn with each m combination of the integers from 0 to n-1 as a list. The list may be destroyed after fn returns."
  (let ((combination (make-list m)))
    (labels ((up-from (low)
               (let ((start (1- low)))
                 (lambda () (incf start))))
             (mc (curr left needed comb-tail)
               (cond
                ((zerop needed)
                 (funcall fn combination))
                ((= left needed)
                 (map-into comb-tail (up-from curr))
                 (funcall fn combination))
                (t
                 (setf (first comb-tail) curr)
                 (mc (1+ curr) (1- left) (1- needed) (rest comb-tail))
                 (mc (1+ curr) (1- left) needed comb-tail)))))
      (mc 0 n m combination))))
```

Example use

```
> (map-combinations 3 5 'print)

(0 1 2)
(0 1 3)
(0 1 4)
(0 2 3)
(0 2 4)
(0 3 4)
(1 2 3)
(1 2 4)
(1 3 4)
(2 3 4)
(2 3 4)
```

Recursive method

```
(defun comb (m list fn)
  (labels ((comb1 (l c m)
           (when (>= (length l) m)
             (if (zerop m) (return-from comb1 (funcall fn c)))
             (comb1 (cdr l) c m)
             (comb1 (cdr l) (cons (first l) c) (1- m)))))
    (comb1 list nil m)))

(comb 3 '(0 1 2 3 4 5) #'print)
```

Alternate, iterative method

```
(defun next-combination (n a)
    (let ((k (length a)) m)
    (loop for i from 1 do
        (when (> i k) (return nil))
        (when (< (aref a (- k i)) (- n i))
            (setf m (aref a (- k i)))
            (loop for j from i downto 1 do
                (incf m)
                (setf (aref a (- k j)) m))
            (return t)))))

(defun all-combinations (n k)
    (if (or (< k 0) (< n k)) '()
        (let ((a (make-array k)))
            (loop for i below k do (setf (aref a i) i))
            (loop collect (coerce a 'list) while (next-combination n a)))))

(defun map-combinations (n k fun)
    (if (and (>= k 0) (>= n k))
        (let ((a (make-array k)))
            (loop for i below k do (setf (aref a i) i))
            (loop do (funcall fun (coerce a 'list)) while (next-combination n a)))))

; all-combinations returns a list of lists

> (all-combinations 4 3)
((0 1 2) (0 1 3) (0 2 3) (1 2 3))

; map-combinations applies a function to each combination

> (map-combinations 6 4 #'print)
(0 1 2 3)
(0 1 2 4)
(0 1 2 5)
(0 1 3 4)
(0 1 3 5)
```

```
(0 2 4 5)
(0 3 4 5)
(1 2 3 4)
(1 2 3 5)
(1 2 4 5)
(1 3 4 5)
(2 3 4 5)
```

Crystal

```ruby
def comb(m, n)
    (0...n).to_a.each_combination(m) { |p| puts(p) }
end
```

```
[0, 1, 2]
[0, 1, 3]
[0, 1, 4]
[0, 2, 3]
[0, 2, 4]
[0, 3, 4]
[1, 2, 3]
[1, 2, 4]
[1, 3, 4]
[2, 3, 4]
```

D

Slow Recursive Version

Translation of: Python

```d
T[][] comb(T)(in T[] arr, in int k) pure nothrow {
    if (k == 0) return [[]];
    typeof(return) result;
    foreach (immutable i, immutable x; arr)
        foreach (suffix; arr[i + 1 .. $].comb(k - 1))
            result ~= x ~ suffix;
    return result;
}

void main() {
    import std.stdio;
    [0, 1, 2, 3].comb(2).writeln;
}
```

Output:

```
[[0, 1], [0, 2], [0, 3], [1, 2], [1, 3], [2, 3]]
```

More Functional Recursive Version

Translation of: Haskell

Same output.

```d
import std.stdio, std.algorithm, std.range;

immutable(int)[][] comb(immutable int[] s, in int m) pure nothrow @safe {
```

```
}

void main() {
    4.iota.array.comb(2).writeln;
}
```

Lazy Version

```
module combinations3;

import std.traits: Unqual;

struct Combinations(T, bool copy=true) {
    Unqual!T[] pool, front;
    size_t r, n;
    bool empty = false;
    size_t[] indices;
    size_t len;
    bool lenComputed = false;

    this(T[] pool_, in size_t r_) pure nothrow @safe {
        this.pool = pool_.dup;
        this.r = r_;
        this.n = pool.length;
        if (r > n)
            empty = true;
        indices.length = r;
        foreach (immutable i, ref ini; indices)
            ini = i;
        front.length = r;
        foreach (immutable i, immutable idx; indices)
            front[i] = pool[idx];
    }

    @property size_t length() /*logic_const*/ pure nothrow @nogc {
        static size_t binomial(size_t n, size_t k) pure nothrow @safe @nogc
        in {
            assert(n > 0, "binomial: n must be > 0.");
        } body {
            if (k < 0 || k > n)
                return 0;
            if (k > (n / 2))
                k = n - k;
            size_t result = 1;
            foreach (size_t d; 1 .. k + 1) {
                result *= n;
                n--;
                result /= d;
            }
            return result;
        }

        if (!lenComputed) {
            // Set cache.
            len = binomial(n, r);
            lenComputed = true;
        }
        return len;
    }

    void popFront() pure nothrow @safe {
        if (!empty) {
            bool broken = false;
            size_t pos = 0;
            foreach_reverse (immutable i; 0 .. r) {
                pos = i;
                if (indices[i] != i + n - r) {
                    broken = true;
                    break;
                }
            }
```

```
            }
            indices[pos]++;
            foreach (immutable j; pos + 1 .. r)
                indices[j] = indices[j - 1] + 1;
            static if (copy)
                front = new Unqual!T[front.length];
            foreach (immutable i, immutable idx; indices)
                front[i] = pool[idx];
        }
    }
}

Combinations!(T, copy) combinations(bool copy=true, T)
                                   (T[] items, in size_t k)
in {
    assert(items.length, "combinations: items can't be empty.");
} body {
    return typeof(return)(items, k);
}

// Compile with -version=combinations3_main to run main.
version(combinations3_main)
void main() {
    import std.stdio, std.array, std.algorithm;
    [1, 2, 3, 4].combinations!false(2).array.writeln;
    [1, 2, 3, 4].combinations!true(2).array.writeln;
    [1, 2, 3, 4].combinations(2).map!(x => x).writeln;
}
```

Lazy Lexicographical Combinations

Includes an algorithm to find mth Lexicographical Element of a Combination (http://msdn.microsoft.com/en-us/library/aa289166.aspx#mth_lexicograp_topic3).

```
module combinations4;
import std.stdio, std.algorithm, std.conv;

ulong choose(int n, int k) nothrow
in {
    assert(n >= 0 && k >= 0, "choose: no negative input.");
} body {
    static ulong[][] cache;

    if (n < k)
        return 0;
    else if (n == k)
        return 1;
    while (n >= cache.length)
        cache ~= [1UL]; // = choose(m, 0);
    auto kmax  = min(k, n - k);
    while(kmax >= cache[n].length) {
        immutable h = cache[n].length;
        cache[n] ~= choose(n - 1, h - 1) + choose(n - 1, h);
    }

    return cache[n][kmax];
}

int largestV(in int p, in int q, in long r) nothrow
in {
    assert(p > 0 && q >= 0 && r >= 0, "largestV: no negative input.");
} body {
    auto v = p - 1;
    while (choose(v, q) > r)
        v--;
    return v;
}

struct Comb {
    immutable int n, m;
```

```
    }

    int[] opIndex(in size_t idx) const {
        if (m < 0 || n < 0)
            return [];
        if (idx >= length)
            throw new Exception("Out of bound");
        ulong x = choose(n, m) - 1 - idx;
        int a = n, b = m;
        auto res = new int[m];
        foreach (i; 0 .. m) {
            a = largestV(a, b, x);
            x = x - choose(a, b);
            b = b - 1;
            res[i] = n - 1 - a;
        }
        return res;
    }

    int opApply(int delegate(ref int[]) dg) const {
        int[] yield;

        foreach (i; 0 .. length) {
            yield = this[i];
            if (dg(yield))
                break;
        }

        return 0;
    }

    static auto On(T)(in T[] arr, in int m) {
        auto comb = Comb(arr.length, m);

        return new class {
            @property size_t length() const /*nothrow*/ {
                return comb.length;
            }

            int opApply(int delegate(ref T[]) dg) const {
                auto yield = new T[m];

                foreach (c; comb) {
                    foreach (idx; 0 .. m)
                        yield[idx] = arr[c[idx]];
                    if (dg(yield))
                        break;
                }

                return 0;
            }
        };
    }
}


version(combinations4_main)
    void main() {
        foreach (c; Comb.On([1, 2, 3], 2))
            writeln(c);
    }
```

Delphi

See Pascal (https://rosettacode.org/wiki/Combinations#Pascal).

E

```
def combinations(m, range) {
```

```
        for i in range {
          for suffix in combinations(m.previous(), range & (int > i)) {
            f(null, [i] + suffix)
          }
        }
      }
    }
  }
}
```

```
? for x in combinations(3, 0..4) { println(x) }
```

EasyLang

Translation of: Julia

```
n = 5
m = 3
len result[] m
#
proc combinations pos val . .
   if pos <= m
      for i = val to n - m
         result[pos] = pos + i
         combinations pos + 1 i
      .
   else
      print result[]
   .
.
combinations 1 0
```

Output:

```
[ 1 2 3 ]
[ 1 2 4 ]
[ 1 2 5 ]
[ 1 3 4 ]
[ 1 3 5 ]
[ 1 4 5 ]
[ 2 3 4 ]
[ 2 3 5 ]
[ 2 4 5 ]
[ 3 4 5 ]
```

EchoLisp

```
;;
;; using the native (combinations) function
(lib 'list)
(combinations (iota 5) 3)
→ ((0 1 2) (0 1 3) (0 1 4) (0 2 3) (0 2 4) (0 3 4) (1 2 3) (1 2 4) (1 3 4) (2 3 4))
;;
;; using an iterator
;;
(lib 'sequences)
(take (combinator (iota 5) 3) #:all)
→ ((0 1 2) (0 1 3) (0 1 4) (0 2 3) (0 2 4) (0 3 4) (1 2 3) (1 2 4) (1 3 4) (2 3 4))
;;
;; defining a function
;;
(define (combine lst p) (cond
    [(null? lst) null]
    [(< (length lst) p) null]
```

```
        (map cons (circular-list (first lst)) (combine (rest lst) (1- p)))
        (combine (rest lst) p))]))

(combine (iota 5) 3)
    → ((0 1 2) (0 1 3) (0 1 4) (0 2 3) (0 2 4) (0 3 4) (1 2 3) (1 2 4) (1 3 4) (2 3 4))
```

Egison

```
(define $comb
  (lambda [$n $xs]
    (match-all xs (list integer)
      [(loop $i [1 ,n] <join _ <cons $a_i ...>> _) a])))

(test (comb 3 (between 0 4)))
```

Output:

```
{[|0 1 2|] [|0 1 3|] [|0 2 3|] [|1 2 3|] [|0 1 4|] [|0 2 4|] [|0 3 4|] [|1 2 4|] [|1 3 4|] [|2 3 4|]}
```

Eiffel

The core of the program is the recursive feature solve, which returns all possible strings of length n with k "ones" and n-k "zeros". The strings are then evaluated, each resulting in k corresponding integers for the digits where ones are found.

```
class
    COMBINATIONS

create
    make

feature

    make (n, k: INTEGER)
        require
            n_positive: n > 0
            k_positive: k > 0
            k_smaller_equal: k <= n
        do
            create set.make
            set.extend ("")
            create sol.make
            sol := solve (set, k, n - k)
            sol := convert_solution (n, sol)
        ensure
            correct_num_of_sol: num_of_comb (n, k) = sol.count
        end

    sol: LINKED_LIST [STRING]

feature {None}

    set: LINKED_LIST [STRING]

    convert_solution (n: INTEGER; solution: LINKED_LIST [STRING]): LINKED_LIST [STRING]
            -- strings of 'k' digits between 1 and 'n'
        local
            i, j: INTEGER
            temp: STRING
        do
            create temp.make (n)
            from
                i := 1
```

```
                from
                    j := 1
                until
                    j > n
                loop
                    if solution [i].at (j) = '1' then
                        temp.append (j.out)
                    end
                    j := j + 1
                end
                solution [i].deep_copy (temp)
                temp.wipe_out
                i := i + 1
            end
            Result := solution
        end

    solve (seta: LINKED_LIST [STRING]; one, zero: INTEGER): LINKED_LIST [STRING]
            -- list of strings with a number of 'one' 1s and 'zero' 0, standig for wether the corresponing digit is taken or
not.
        local
            new_P1, new_P0: LINKED_LIST [STRING]
        do
            create new_P1.make
            create new_P0.make
            if one > 0 then
                new_P1.deep_copy (seta)
                across
                    new_P1 as P1
                loop
                    new_P1.item.append ("1")
                end
                new_P1 := solve (new_P1, one - 1, zero)
            end
            if zero > 0 then
                new_P0.deep_copy (seta)
                across
                    new_P0 as P0
                loop
                    new_P0.item.append ("0")
                end
                new_P0 := solve (new_P0, one, zero - 1)
            end
            if one = 0 and zero = 0 then
                Result := seta
            else
                create Result.make
                Result.fill (new_p0)
                Result.fill (new_p1)
            end
        end

    num_of_comb (n, k: INTEGER): INTEGER
            -- number of 'k' sized combinations out of 'n'.
        local
            upper, lower, m, l: INTEGER
        do
            upper := 1
            lower := 1
            m := n
            l := k
            from
            until
                m < n - k + 1
            loop
                upper := m * upper
                lower := l * lower
                m := m - 1
                l := l - 1
            end
            Result := upper // lower
        end

end
```

```
class
    APPLICATION

create
    make

feature

    make
        do
            create comb.make (5, 3)
            across
                comb.sol as ar
            loop
                io.put_string (ar.item.out + "%T")
            end
        end

    comb: COMBINATIONS

end
```

Output:

```
345 245 235 234 145 135 134 125 124 123
```

Elena

ELENA 6.x :

```
import system'routines;
import extensions;
import extensions'routines;

const int M = 3;
const int N = 5;

Numbers(n)
{
    ^ Array.allocate(n).populate::(int n => n)
}

public program()
{
    var numbers := Numbers(N);
    Combinator.new(M, numbers).forEach::(row)
    {
        console.printLine(row.toString())
    };

    console.readChar()
}
```

Output:

```
0,1,2
0,1,3
0,1,4
0,2,3
0,2,4
0,3,4
1,2,3
1,2,4
```

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

Elixir

Translation of: Erlang

```elixir
defmodule RC do
  def comb(0, _), do: [[]]
  def comb(_, []), do: []
  def comb(m, [h|t]) do
    (for l <- comb(m-1, t), do: [h|l]) ++ comb(m, t)
  end
end

{m, n} = {3, 5}
list = for i <- 1..n, do: i
Enum.each(RC.comb(m, list), fn x -> IO.inspect x end)
```

Output:

```
[1, 2, 3]
[1, 2, 4]
[1, 2, 5]
[1, 3, 4]
[1, 3, 5]
[1, 4, 5]
[2, 3, 4]
[2, 3, 5]
[2, 4, 5]
[3, 4, 5]
```

Emacs Lisp

Translation of: Haskell

```elisp
(defun comb-recurse (m n n-max)
  (cond ((zerop m) '(()))
        ((= n-max n) '())
        (t (append (mapcar #'(lambda (rest) (cons n rest))
                           (comb-recurse (1- m) (1+ n) n-max))
                   (comb-recurse m (1+ n) n-max)))))

(defun comb (m n)
  (comb-recurse m 0 n))

(comb 3 5)
```

Output:

```
((0 1 2) (0 1 3) (0 1 4) (0 2 3) (0 2 4) (0 3 4) (1 2 3) (1 2 4) (1 3 4) (2 3 4))
```

Erlang

```erlang
-module(comb).
-compile(export_all).

comb(0,_) ->
    [[]];
```

```
comb(N,[H|T]) ->
    [[H|L] || L <- comb(N-1,T)]++comb(N,T).
```

Dynamic Programming

Translation of: Haskell

Could be optimized with a custom `zipwith/3` function instead of using `lists:sublist/2`.

```
-module(comb).
-export([combinations/2]).

combinations(K, List) ->
    lists:last(all_combinations(K, List)).

all_combinations(K, List) ->
    lists:foldr(
      fun(X, Next) ->
              Sub = lists:sublist(Next, length(Next) - 1),
              Step = [[]] ++ [[[X|S] || S <- L] || L <- Sub],
              lists:zipwith(fun lists:append/2, Step, Next)
      end, [[[]]] ++ lists:duplicate(K, []), List).
```

ERRE

```
PROGRAM COMBINATIONS

CONST M_MAX=3,N_MAX=5

DIM COMBINATION[M_MAX],STACK[100,1]

PROCEDURE GENERATE(M)
   LOCAL I
   IF (M>M_MAX) THEN
    FOR I=1 TO M_MAX DO
     PRINT(COMBINATION[I];" ";)
    END FOR
    PRINT
   ELSE
    FOR N=1 TO N_MAX DO
      IF ((M=1) OR (N>COMBINATION[M-1])) THEN
        COMBINATION[M]=N
        ! --- PUSH STACK -----------
        STACK[SP,0]=M  STACK[SP,1]=N
        SP=SP+1
        ! --------------------------

        GENERATE(M+1)

        ! --- POP STACK ------------
        SP=SP-1
        M=STACK[SP,0] N=STACK[SP,1]
        ! --------------------------
      END IF
    END FOR
   END IF
END PROCEDURE

BEGIN
 GENERATE(1)
END PROGRAM
```

Output:

```
1 2 3
```

```
1 3 5
1 4 5
2 3 4
2 3 5
2 4 5
3 4 5
```

F#

```
let choose m n =
    let rec fC prefix m from = seq {
        let rec loopFor f = seq {
            match f with
            | [] -> ()
            | x::xs ->
                yield (x, fC [] (m-1) xs)
                yield! loopFor xs
        }
        if m = 0 then yield prefix
        else
            for (i, s) in loopFor from do
                for x in s do
                    yield prefix@[i]@x
    }
    fC [] m [0..(n-1)]

[<EntryPoint>]
let main argv =
    choose 3 5
    |> Seq.iter (printfn "%A")
    0
```

Output:

```
[0; 1; 2]
[0; 1; 3]
[0; 1; 4]
[0; 2; 3]
[0; 2; 4]
[0; 3; 4]
[1; 2; 3]
[1; 2; 4]
[1; 3; 4]
[2; 3; 4]
```

Factor

```
USING: math.combinatorics prettyprint ;

5 iota 3 all-combinations .
```

```
{
    { 0 1 2 }
    { 0 1 3 }
    { 0 1 4 }
    { 0 2 3 }
    { 0 2 4 }
    { 0 3 4 }
    { 1 2 3 }
    { 1 2 4 }
    { 1 3 4 }
    { 2 3 4 }
}
```

```
{ "a" "b" "c" } 2 all-combinations .
```

```
{ { "a" "b" } { "a" "c" } { "b" "c" } }
```

Fortran

```
program Combinations
  use iso_fortran_env
  implicit none

  type comb_result
     integer, dimension(:), allocatable :: combs
  end type comb_result

  type(comb_result), dimension(:), pointer :: r
  integer :: i, j

  call comb(5, 3, r)
  do i = 0, choose(5, 3) - 1
     do j = 2, 0, -1
        write(*, "(I4, ' ')", advance="no") r(i)%combs(j)
     end do
     deallocate(r(i)%combs)
     write(*,*) ""
  end do
  deallocate(r)

contains

  function choose(n, k, err)
    integer :: choose
    integer, intent(in) :: n, k
    integer, optional, intent(out) :: err

    integer :: imax, i, imin, ie

    ie = 0
    if ( (n < 0 ) .or. (k < 0 ) ) then
       write(ERROR_UNIT, *) "negative in choose"
       choose = 0
       ie = 1
    else
       if ( n < k ) then
          choose = 0
       else if ( n == k ) then
          choose = 1
       else
          imax = max(k, n-k)
          imin = min(k, n-k)
          choose = 1
          do i = imax+1, n
             choose = choose * i
          end do
          do i = 2, imin
             choose = choose / i
          end do
       end if
    end if
    if ( present(err) ) err = ie
  end function choose

  subroutine comb(n, k, co)
    integer, intent(in) :: n, k
    type(comb_result), dimension(:), pointer, intent(out) :: co

    integer :: i, j, s, ix, kx, hm, t
    integer :: err

    hm = choose(n, k, err)
```

```
    end if

    allocate(co(0:hm-1))
    do i = 0, hm-1
       allocate(co(i)%combs(0:k-1))
    end do
    do i = 0, hm-1
       ix = i; kx = k
       do s = 0, n-1
          if ( kx == 0 ) exit
          t = choose(n-(s+1), kx-1)
          if ( ix < t ) then
             co(i)%combs(kx-1) = s
             kx = kx - 1
          else
             ix = ix - t
          end if
       end do
    end do

  end subroutine comb

end program Combinations
```

Alternatively:

```
program combinations

  implicit none
  integer, parameter :: m_max = 3
  integer, parameter :: n_max = 5
  integer, dimension (m_max) :: comb
  character (*), parameter :: fmt = '(i0' // repeat (', 1x, i0', m_max - 1) // ')'

  call gen (1)

contains

  recursive subroutine gen (m)

    implicit none
    integer, intent (in) :: m
    integer :: n

    if (m > m_max) then
      write (*, fmt) comb
    else
      do n = 1, n_max
        if ((m == 1) .or. (n > comb (m - 1))) then
          comb (m) = n
          call gen (m + 1)
        end if
      end do
    end if

  end subroutine gen

end program combinations
```

Output:

```
1 2 3
1 2 4
1 2 5
1 3 4
1 3 5
1 4 5
2 3 4
2 3 5
```

```
2 4 5
3 4 5
```

FreeBASIC

This is remarkably compact and elegant.

```
sub iterate( byval curr as string, byval start as uinteger,_
             byval stp as uinteger, byval depth as uinteger )
    dim as uinteger i
    for i = start to stp
        if depth = 0 then
            print curr + " " + str(i)
        end if
        iterate( curr+" "+str(i), i+1, stp, depth-1 )
    next i
    return
end sub

dim as uinteger m, n
input "Enter n comb m.  ", n, m
dim as string outstr = ""
iterate outstr, 0, m-1, n-1
```

Output:

```
Enter n comb m.  3,5
 0 1 2
 0 1 3
 0 1 4
 0 2 3
 0 2 4
 0 3 4
 1 2 3
 1 2 4
 1 3 4
 2 3 4
```

GAP

```
# Built-in
Combinations([1 .. n], m);

Combinations([1 .. 5], 3);
# [ [ 1, 2, 3 ], [ 1, 2, 4 ], [ 1, 2, 5 ], [ 1, 3, 4 ], [ 1, 3, 5 ],
#   [ 1, 4, 5 ], [ 2, 3, 4 ], [ 2, 3, 5 ], [ 2, 4, 5 ], [ 3, 4, 5 ] ]
```

Glee

```
5!3 >>> ,,\

$$(5!3) give all combinations of 3 out of 5
$$(>>>) sorted up,
$$(,,\) printed with crlf delimiters.
```

Result:

```
Result:
```

```
1 3 4
1 3 5
1 4 5
2 3 4
2 3 5
2 4 5
3 4 5
```

Go

```
package main

import (
    "fmt"
)

func main() {
    comb(5, 3, func(c []int) {
        fmt.Println(c)
    })
}

func comb(n, m int, emit func([]int)) {
    s := make([]int, m)
    last := m - 1
    var rc func(int, int)
    rc = func(i, next int) {
        for j := next; j < n; j++ {
            s[i] = j
            if i == last {
                emit(s)
            } else {
                rc(i+1, j+1)
            }
        }
        return
    }
    rc(0, 0)
}
```

Output:

```
[0 1 2]
[0 1 3]
[0 1 4]
[0 2 3]
[0 2 4]
[0 3 4]
[1 2 3]
[1 2 4]
[1 3 4]
[2 3 4]
```

Groovy

Following the spirit of the Haskell solution.

In General

A recursive closure must be *pre-declared*.

```
    m == 0 ?
        [[]] :
        (0..(n-m)).inject([]) { newlist, k ->
            def sublist = (k+1 == n) ? [] : list[(k+1)..<n]
            newlist += comb(m-1, sublist).collect { [list[k]] + it }
        }
}
```

Test program:

```
def csny = [ "Crosby", "Stills", "Nash", "Young" ]
println "Choose from ${csny}"
(0..(csny.size())).each { i -> println "Choose ${i}:"; comb(i, csny).each { println it }; println() }
```

Output:

```
Choose from [Crosby, Stills, Nash, Young]
Choose 0:
[]

Choose 1:
[Crosby]
[Stills]
```

Zero-based Integers

```
def comb0 = { m, n -> comb(m, (0..<n)) }
```

Test program:

```
println "Choose out of 5 (zero-based):"
(0..3).each { i -> println "Choose ${i}:"; comb0(i, 5).each { println it }; println() }
```

Output:

```
Choose out of 5 (zero-based):
Choose 0:
[]

Choose 1:
[0]
[1]
```

One-based Integers

```
def comb1 = { m, n -> comb(m, (1..n)) }
```

Test program:

```
println "Choose out of 5 (one-based):"
(0..3).each { i -> println "Choose ${i}:"; comb1(i, 5).each { println it }; println() }
```

```
Choose out of 5 (one-based):
Choose 0:
[]

Choose 1:
[1]
[2]
```

Haskell

It's more natural to extend the task to all (ordered) sublists of size *m* of a list.

Straightforward, unoptimized implementation with divide-and-conquer:

```
comb :: Int -> [a] -> [[a]]
comb 0 _      = [[]]
comb _ []     = []
comb m (x:xs) = map (x:) (comb (m-1) xs) ++ comb m xs
```

In the induction step, either *x* is not in the result and the recursion proceeds with the rest of the list *xs*, or it is in the result and then we only need *m-1* elements.

Shorter version of the above:

```
import Data.List (tails)

comb :: Int -> [a] -> [[a]]
comb 0 _      = [[]]
comb m l = [x:ys | x:xs <- tails l, ys <- comb (m-1) xs]
```

To generate combinations of integers between 0 and *n-1*, use

```
comb0 m n = comb m [0..n-1]
```

Similar, for integers between 1 and *n*, use

```
comb1 m n = comb m [1..n]
```

Another method is to use the built in *Data.List.subsequences* function, filter for subsequences of length *m* and then sort:

```
import Data.List (sort, subsequences)
comb m n = sort . filter ((==m) . length) $ subsequences [0..n-1]
```

And yet another way is to use the list monad to generate all possible subsets:

```
comb m n = filter ((==m . length) $ filterM (const [True, False]) [0..n-1]
```

Dynamic Programming

The first solution is inefficient because it repeatedly calculates the same subproblem in different branches of recursion. For example, `comb m (x1:x2:xs)` involves computing `comb (m-1) (x2:xs)` and `comb m (x2:xs)`, both of which (separately) compute `comb (m-1) xs`. To avoid repeated computation, we can use dynamic programming:

```
comb :: Int -> [a] -> [[a]]
comb m xs = combsBySize xs !! m
  where
    combsBySize = foldr f ([[]] : repeat [])
    f x next =
      zipWith
        (<>)
        (fmap (x :) <$> ([] : next))
        next

main :: IO ()
main = print $ comb 3 [0 .. 4]
```

Output:

```
[[0,1,2],[0,1,3],[0,1,4],[0,2,3],[0,2,4],[0,3,4],[1,2,3],[1,2,4],[1,3,4],[2,3,4]]
```

Icon and Unicon

```
procedure main()
return combinations(3,5,0)
end

procedure combinations(m,n,z)                    # demonstrate combinations
/z := 1

write(m," combinations of ",n," integers starting from ",z)
every put(L := [], z to n - 1 + z by 1)          # generate list of n items from z
write("Intial list\n",list2string(L))
write("Combinations:")
every write(list2string(lcomb(L,m)))
end

procedure list2string(L)                         # helper function
every (s := "[") ||:= " " || (!L|"]")
return s
end

link lists
```

The

Library: Icon Programming Library

provides the core procedure lcomb in lists (http://www.cs.arizona.edu/icon/library/src/procs/lists.icn) written by Ralph E. Griswold and Richard L. Goerwitz.

```
procedure lcomb(L,i)            #: list combinations
   local j

   if i < 1 then fail
   suspend if i = 1 then [!L]
      else [L[j := 1 to *L - i + 1]] ||| lcomb(L[j + 1:0],i - 1)

end
```

```
3 combinations of 5 integers starting from 0
Intial list
[ 0 1 2 3 4 ]
Combinations:
[ 0 1 2 ]
[ 0 1 3 ]
[ 0 1 4 ]
[ 0 2 3 ]
[ 0 2 4 ]
[ 0 3 4 ]
[ 1 2 3 ]
[ 1 2 4 ]
[ 1 3 4 ]
[ 2 3 4 ]
```

J

Library

```
require'stats'
```

Example use:

```
   3 comb 5
0 1 2
0 1 3
0 1 4
0 2 3
0 2 4
0 3 4
1 2 3
1 2 4
1 3 4
2 3 4
```

All implementations here give that same result if given the same arguments.

Iteration

```
comb1=: dyad define
  c=. 1 {.~ - d=. 1+y-x
  z=. i.1 0
  for_j. (d-1+y)+/&i.d do. z=. (c#j) ,. z{~;(-c){.&.><i.{.c=. +/\.c end.
)
```

another iteration version

```
comb2=: dyad define
  d =. 1 + y - x
  k =. >: |. i. d
  z =. < \. |. i. d
  for. i.x-1 do.
     z=. , each /\. k ,. each z
     k =. 1 + k
  end.
```

```
  ;{.z
)
```

Recursion

```
combr=: dyad define M.
  if. (x>:y)+.0=x do. i.(x<:y),x else. (0,.x combr&.<: y),1+x combr y-1 end.
)
```

The `M.` uses memoization (caching) which greatly reduces the running time. As a result, this is probably the fastest of the implementations here.

A less efficient but easier to understand recursion (similar to Python and Haskell).

```
 combr=: dyad define
  if.(x=#y) +. x=1 do.
    y
  else.
    (({.y) ,. (x-1) combr (}.y)) , (x combr }.y)
  end.
)
```

You need to supply the "list" for example `i.5`

```
 3 combr i.5
```

Brute Force

We can also generate all permutations and exclude those which are not properly sorted combinations. This is inefficient, but efficiency is not always important.

```
combb=: (#~ ((-:/:~)>/:~-:\:~)"1)@(# #: [: i. ^~)
```

Java

Translation of: JavaScript
Works with: Java version 1.5+

```
import java.util.Collections;
import java.util.LinkedList;

public class Comb{

        public static void main(String[] args){
                System.out.println(comb(3,5));
        }

        public static String bitprint(int u){
                String s= "";
                for(int n= 0;u > 0;++n, u>>= 1)
                        if((u & 1) > 0) s+= n + " ";
                return s;
        }

        public static int bitcount(int u){
```

```
        }

        public static LinkedList<String> comb(int c, int n){
                LinkedList<String> s= new LinkedList<String>();
                for(int u= 0;u < 1 << n;u++)
                        if(bitcount(u) == c) s.push(bitprint(u));
                Collections.sort(s);
                return s;
        }
}
```

JavaScript

Imperative

```
function bitprint(u) {
  var s="";
  for (var n=0; u; ++n, u>>=1)
    if (u&1) s+=n+" ";
  return s;
}
function bitcount(u) {
  for (var n=0; u; ++n, u=u&(u-1));
  return n;
}
function comb(c,n) {
  var s=[];
  for (var u=0; u<1<<n; u++)
    if (bitcount(u)==c)
      s.push(bitprint(u))
  return s.sort();
}
comb(3,5)
```

Alternative recursive version using and an array of values instead of length:

Translation of: Python

```
function combinations(arr, k){
    var i,
    subI,
    ret = [],
    sub,
    next;
    for(i = 0; i < arr.length; i++){
        if(k === 1){
            ret.push( [ arr[i] ] );
        }else{
            sub = combinations(arr.slice(i+1, arr.length), k-1);
            for(subI = 0; subI < sub.length; subI++ ){
                next = sub[subI];
                next.unshift(arr[i]);
                ret.push( next );
            }
        }
    }
    return ret;
}
combinations([0,1,2,3,4], 3);
// produces: [[0, 1, 2], [0, 1, 3], [0, 1, 4], [0, 2, 3], [0, 2, 4], [0, 3, 4], [1, 2, 3], [1, 2, 4], [1, 3, 4], [2, 3, 4]]

combinations(["Crosby", "Stills", "Nash", "Young"], 3);
```

```
// produces: [["Crosby", "Stills", "Nash"], ["Crosby", "Stills", "Young"], ["Crosby", "Nash", "Young"], ["Stills", "Nash",
"Young"]]
```

Functional

ES5

Simple recursion:

```
(function () {

  function comb(n, lst) {
    if (!n) return [[]];
    if (!lst.length) return [];

    var x = lst[0],
        xs = lst.slice(1);

    return comb(n - 1, xs).map(function (t) {
      return [x].concat(t);
    }).concat(comb(n, xs));
  }


  // [m..n]
  function range(m, n) {
    return Array.apply(null, Array(n - m + 1)).map(function (x, i) {
      return m + i;
    });
  }

  return comb(3, range(0, 4))

    .map(function (x) {
      return x.join(' ');
    }).join('\n');

})();
```

We can significantly improve on the performance of the simple recursive function by deriving a memoized version of it, which stores intermediate results for repeated use.

```
(function (n) {

  // n -> [a] -> [[a]]
  function comb(n, lst) {
    if (!n) return [[]];
    if (!lst.length) return [];

    var x = lst[0],
      xs = lst.slice(1);

    return comb(n - 1, xs).map(function (t) {
      return [x].concat(t);
    }).concat(comb(n, xs));
  }

  // f -> f
  function memoized(fn) {
    m = {};
    return function (x) {
      var args = [].slice.call(arguments),
        strKey = args.join('-');

      v = m[strKey];
```

```
    }
  }

  // [m..n]
  function range(m, n) {
    return Array.apply(null, Array(n - m + 1)).map(function (x, i) {
      return m + i;
    });
  }

  var fnMemoized = memoized(comb),
    lstRange = range(0, 4);

  return fnMemoized(n, lstRange)

  .map(function (x) {
    return x.join(' ');
  }).join('\n');

})(3);
```

Output:

```
0 1 2
0 1 3
0 1 4
0 2 3
0 2 4
0 3 4
1 2 3
1 2 4
1 3 4
2 3 4
```

ES6

Defined in terms of a recursive helper function:

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

    // ------------------ COMBINATIONS -------------------

    // combinations :: Int -> [a] -> [[a]]
    const combinations = n =>
        xs => {
            const comb = n => xs => {
                return 1 > n ? [
                    []
                ] : 0 === xs.length ? (
                    []
                ) : (() => {
                    const
                        h = xs[0],
                        tail = xs.slice(1);
                    return comb(n - 1)(tail)
                        .map(cons(h))
                        .concat(comb(n)(tail));
                })()
            };
            return comb(n)(xs);
        };
```

```
            combinations(3)(
                enumFromTo(0)(4)
            )
        );


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

    // cons :: a -> [a] -> [a]
    const cons = x =>
        // A list constructed from the item x,
        // followed by the existing list xs.
        xs => [x].concat(xs);


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


    // show :: a -> String
    const show = (...x) =>
        JSON.stringify.apply(
            null, x.length > 1 ? [x[0], null, x[1]] : x
        );

    // MAIN ---
    return main();
})();
```

Output:

```
[[0, 1, 2], [0, 1, 3], [0, 1, 4], [0, 2, 3], [0, 2, 4],
[0, 3, 4], [1, 2, 3], [1, 2, 4], [1, 3, 4], [2, 3, 4]]
```

Or, defining combinations in terms of a more general subsequences function:

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

    // ------------------ COMBINATIONS -------------------

    // comb :: Int -> Int -> [[Int]]
    const comb = m =>
        n => combinations(m)(
            enumFromTo(0)(n - 1)
        );

    // combinations :: Int -> [a] -> [[a]]
    const combinations = k =>
        xs => sort(
            filter(xs => k === xs.length)(
                subsequences(xs)
            )
        );

    // ---------------------- TEST ----------------------
    const main = () =>
        show(
            comb(3)(5)
        );

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

    // cons :: a -> [a] -> [a]
    const cons = x =>
```

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


    // filter :: (a -> Bool) -> [a] -> [a]
    const filter = p =>
        // The elements of xs which match
        // the predicate p.
        xs => [...xs].filter(p);


    // list :: StringOrArrayLike b => b -> [a]
    const list = xs =>
        // xs itself, if it is an Array,
        // or an Array derived from xs.
        Array.isArray(xs) ? (
            xs
        ) : Array.from(xs || []);


    // show :: a -> String
    const show = x =>
        // JSON stringification of a JS value.
        JSON.stringify(x)


    // sort :: Ord a => [a] -> [a]
    const sort = xs => list(xs).slice()
        .sort((a, b) => a < b ? -1 : (a > b ? 1 : 0));


    // subsequences :: [a] -> [[a]]
    // subsequences :: String -> [String]
    const subsequences = xs => {
        const
            // nonEmptySubsequences :: [a] -> [[a]]
            nonEmptySubsequences = xxs => {
                if (xxs.length < 1) return [];
                const [x, xs] = [xxs[0], xxs.slice(1)];
                const f = (r, ys) => cons(ys)(cons(cons(x)(ys))(r));
                return cons([x])(nonEmptySubsequences(xs)
                    .reduceRight(f, []));
            };
        return ('string' === typeof xs) ? (
            cons('')(nonEmptySubsequences(xs.split(''))
                .map(x => ''.concat.apply('', x)))
        ) : cons([])(nonEmptySubsequences(xs));
    };

    // MAIN ---
    return main();
})();
```

Output:

```
[[0,1,2],[0,1,3],[0,1,4],[0,2,3],[0,2,4],[0,3,4],[1,2,3],[1,2,4],[1,3,4],[2,3,4]]
```

With recursions:

```
function combinations(k, arr, prefix = []) {
    if (prefix.length == 0) arr = [...Array(arr).keys()];
    if (k == 0) return [prefix];
    return arr.flatMap((v, i) =>
```

```
    );
}
```

jq

combination(r) generates a stream of combinations of the input array. The stream can be captured in an array as shown in the second example.

```
def combination(r):
  if r > length or r < 0 then empty
  elif r == length then .
  else  ( [.[0]] + (.[1:]|combination(r-1))),
        ( .[1:]|combination(r))
  end;

# select r integers from the set (0 .. n-1)
def combinations(n;r): [range(0;n)] | combination(r);
```

Example 1

```
combinations(5;3)
```

Output:

```
[0,1,2]
[0,1,3]
[0,1,4]
[0,2,3]
[0,2,4]
[0,3,4]
[1,2,3]
[1,2,4]
[1,3,4]
[2,3,4]
```

Example 2

```
["a", "b", "c", "d", "e"] | combination(3) ] | length
```

Output:

```
10
```

Julia

The `combinations` function in the `Combinatorics.jl` package generates an iterable sequence of the combinations that you can loop over. (Note that the combinations are computed on the fly during the loop iteration, and are not pre-computed or stored since there many be a very large number of them.)

```
using Combinatorics
n = 4
m = 3
for i in combinations(0:n,m)
    println(i')
```

Output:

```
[0 1 2]
[0 1 3]
[0 1 4]
[0 2 3]
[0 2 4]
[0 3 4]
[1 2 3]
[1 2 4]
[1 3 4]
[2 3 4]
```

Recursive solution without the library

The previous solution is the best: it is most elegant, production stile solution.

If, on the other hand we wanted to show how it could be done in Julia, this recursive solution shows some potentials of Julia lang.

```
##############################
# COMBINATIONS OF 3 OUT OF 5 #
##############################

# Set n and m
m = 5
n = 3

# Prepare the boundary of the calculation. Only m - n numbers are changing in each position.
max_n = m - n

#Prepare an array for result
result = zeros(Int64, n)

function combinations(pos, val)            # n, max_n and result are visible in the function
    for i = val:max_n                      # from current value to the boundary
        result[pos] = pos + i              # fill the position of result
        if pos < n                         # if combination isn't complete,
           combinations(pos+1, i)         # go to the next position
        else
            println(result)                # combination is complete, print it
        end
   end
end

combinations(1, 0)
end
```

Output:

```
[1, 2, 3]
[1, 2, 4]
[1, 2, 5]
[1, 3, 4]
[1, 3, 5]
[1, 4, 5]
[2, 3, 4]
[2, 3, 5]
[2, 4, 5]
[3, 4, 5]
```

Iterator Solution

```
using Base.Iterators

function bitmask(u, max_size)
    res = BitArray(undef, max_size)
    res.chunks[1] = u%UInt64
    res
end

function combinations(input_collection::Vector{T}, choice_size::Int)::Vector{Vector{T}} where T
    num_elements = length(input_collection)
    size_filter(x) = Iterators.filter(y -> count_ones(y) == choice_size, x)
    bitmask_map(x) = Iterators.map(y -> bitmask(y, num_elements), x)
    getindex_map(x) = Iterators.map(y -> input_collection[y], x)

    UnitRange(0, (2^num_elements)-1) |>
        size_filter  |>
        bitmask_map  |>
        getindex_map |>
        collect
end
```

Output:

```
julia> show(combinations([1,2,3,4,5], 3))
[[1, 2, 3], [1, 2, 4], [1, 3, 4], [2, 3, 4], [1, 2, 5], [1, 3, 5], [2, 3, 5], [1, 4, 5], [2, 4, 5], [3, 4, 5]]
```

end

K

Recursive implementation:

```
comb:{[n;k]
    f:{:[k=#x; :,x; :,/_f' x,'(1+*|x) _ !n]}
    :,/f' !n
}
```

Lambdatalk

Translation from Emacs-lisp

```
{def comb
 {def comb.r
  {lambda {:m :n :N}
   {if {= :m 0}
    then {A.new {A.new}}
    else {if {= :n :N}
    then {A.new}
    else {A.concat
          {A.map {{lambda {:n :rest} {A.addfirst! :n :rest}} :n}
                 {comb.r {- :m 1} {+ :n 1} :N}}
          {comb.r :m {+ :n 1} :N}}}}}}
 {lambda {:m :n}
  {comb.r :m 0 :n}}}
-> comb

{comb 3 5}
```

```
-> [[0,1,2],[0,1,3],[0,1,4],[0,2,3],[0,2,4],[0,3,4],
   [1,2,3],[1,2,4],[1,3,4],[2,3,4]]
```

Kotlin

Recursion

Translation of: Pascal

```
class Combinations(val m: Int, val n: Int) {
    private val combination = IntArray(m)

    init {
        generate(0)
    }

    private fun generate(k: Int) {
        if (k >= m) {
            for (i in 0 until m) print("${combination[i]} ")
            println()
        }
        else {
            for (j in 0 until n)
                if (k == 0 || j > combination[k - 1]) {
                    combination[k] = j
                    generate(k + 1)
                }
        }
    }
}

fun main(args: Array<String>) {
    Combinations(3, 5)
}
```

Output:

```
0 1 2
0 1 3
0 1 4
0 2 3
0 2 4
0 3 4
1 2 3
1 2 4
1 3 4
2 3 4
```

Lazy

Translation of: C#

```
import java.util.LinkedList

inline fun <reified T> combinations(arr: Array<T>, m: Int) = sequence {
    val n = arr.size
    val result = Array(m) { arr[0] }
    val stack = LinkedList<Int>()
    stack.push(0)
    while (stack.isNotEmpty()) {
        var resIndex = stack.size - 1;
        var arrIndex = stack.pop()

        while (arrIndex < n) {
```

```
            if (resIndex == m) {
                yield(result.toList())
                break
            }
        }
    }
}

fun main() {
    val n = 5
    val m = 3
    combinations((1..n).toList().toTypedArray(), m).forEach { println(it.joinToString(separator = " ")) }
}
```

Output:

```
1 2 3
1 2 4
1 2 5
1 3 4
1 3 5
1 4 5
2 3 4
2 3 5
2 4 5
3 4 5
```

Lobster

Translation of: Nim

```
import std

// combi is an itertor that solves the Combinations problem for iota arrays as stated

def combi(m, n, f):
    let c = map(n): _

    while true:
        f(c)
        var i = n-1
        c[i] = c[i] + 1
        if c[i] > m - 1:
            while c[i] >= m - n + i:
                i -= 1
                if i < 0: return
            c[i] = c[i] + 1
            while i < n-1:
                c[i+1] = c[i] + 1
                i += 1

combi(5, 3): print(_)
```

Output:

```
[0, 1, 2]
[0, 1, 3]
[0, 1, 4]
[0, 2, 3]
[0, 2, 4]
[0, 3, 4]
[1, 2, 3]
[1, 2, 4]
[1, 3, 4]
[2, 3, 4]
```

```
import std

// comba solves the general problem for any values in an input array

def comba<T>(arr: [T], k) -> [[T]]:
    let ret = []
    for(arr.length) i:
        if k == 1:
            ret.push([arr[i]])
        else:
            let sub = comba(arr.slice(i+1, -1), k-1)
            for(sub) next:
                next.insert(0, arr[i])
                ret.push(next)
    return ret

print comba([0,1,2,3,4], 3)
print comba(["Crosby", "Stills", "Nash", "Young"], 3)
// Of course once could use combi to index the input array instead
var s = ""
combi(4, 3): s += (map(_) i: ["Crosby", "Stills", "Nash", "Young"][i]) + " "
print s
```

Output:

```
[[0, 1, 2], [0, 1, 3], [0, 1, 4], [0, 2, 3], [0, 2, 4], [0, 3, 4], [1, 2, 3], [1, 2, 4], [1, 3, 4], [2, 3, 4]]
[["Crosby", "Stills", "Nash"], ["Crosby", "Stills", "Young"], ["Crosby", "Nash", "Young"], ["Stills", "Nash", "Young"]]
["Crosby", "Stills", "Nash"] ["Crosby", "Stills", "Young"] ["Crosby", "Nash", "Young"] ["Stills", "Nash", "Young"]
```

Logo

```
to comb :n :list
  if :n = 0 [output [[]]]
  if empty? :list [output []]
  output sentence map [sentence first :list ?] comb :n-1 bf :list ~
                  comb :n bf :list
end
print comb 3 [0 1 2 3 4]
```

Lua

```
function map(f, a, ...) if a then return f(a), map(f, ...) end end
function incr(k) return function(a) return k > a and a or a+1 end end
function combs(m, n)
  if m * n == 0 then return {{}} end
  local ret, old = {}, combs(m-1, n-1)
  for i = 1, n do
    for k, v in ipairs(old) do ret[#ret+1] = {i, map(incr(i), unpack(v))} end
  end
  return ret
end

for k, v in ipairs(combs(3, 5)) do print(unpack(v)) end
```

M2000 Interpreter

Including a helper sub to export result to clipboard through a global variable (a temporary global variable)

```
Module Checkit {
      Global a$
```

```
                  If n=1 then {
                        while Len(s) {
                               Print h, car(s)
                               ToClipBoard()
                               s=cdr(s)
                         }
                  } Else {
                        While len(s) {
                              call Level n-1, cdr(s),  cons(h, car(s))
                              s=cdr(s)
                        }
                  }
                  Sub ToClipBoard()
                        local m=each(h)
                        Local b$=""
                        While m {
                              b$+=If$(Len(b$)<>0->" ","")+Format$("{0::-10}",Array(m))
                        }
                        b$+=If$(Len(b$)<>0->" ","")+Format$("{0::-10}",Array(s,0))+{
                        }
                        a$<=b$   ' assign to global need <=
                  End Sub
            }
            If m<1 or n<1 then Error
            s=(,)
            for i=0 to n-1 {
                  s=cons(s, (i,))
            }
            Head=(,)
            Call Level m,  s, Head
      }
      Clear a$
      Combinations 3, 5
      ClipBoard a$
}
Checkit
```

Output:

```
         0          1          2
         0          1          3
         0          1          4
         0          2          3
         0          2          4
         0          3          4
         1          2          3
         1          2          4
         1          3          4
         2          3          4
```

Step by Step

```
Module StepByStep {
      Function CombinationsStep (a, nn) {
            c1=lambda (&f, &a) ->{
                  =car(a) : a=cdr(a) : f=len(a)=0
            }
            m=len(a)
            c=c1
            n=m-nn+1
            p=2
            while m>n {
                  c1=lambda c2=c,n=p, z=(,) (&f, &m) ->{
                        if len(z)=0 then z=cdr(m)
                        =cons(car(m),c2(&f, &z))
                        if f then z=(,) : m=cdr(m) : f=len(m)+len(z)<n
                   }
                  c=c1
                  n++
```

```
                =c(&f, &a)
            }
      }
      k=false
      StepA=CombinationsStep((1, 2, 3, 4,5), 3)
      while not k {
                Print StepA(&k)
      }
      k=false
      StepA=CombinationsStep((0, 1, 2, 3, 4), 3)
      while not k {
                Print StepA(&k)
      }
      k=false
      StepA=CombinationsStep(("A", "B", "C", "D","E"), 3)
      while not k {
                Print StepA(&k)
      }
      k=false
      StepA=CombinationsStep(("CAT", "DOG", "BAT"), 2)
      while not k {
                Print StepA(&k)
      }
}
StepByStep
```

M4

```
divert(-1)
define(`set',`define(`$1[$2]',`$3')')
define(`get',`defn(`$1[$2]')')
define(`setrange',`ifelse(`$3',`',$2,`define($1[$2],$3)``setrange($1,
   incr($2),shift(shift(shift($@))))')')
define(`for',
   `ifelse($#,0,``$0'',
   `ifelse(eval($2<=$3),1,
   `pushdef(`$1',$2)$4`'popdef(`$1')$0(`$1',incr($2),$3,`$4')')')')
define(`show',
   `for(`k',0,decr($1),`get(a,k) ')')

define(`chklim',
   `ifelse(get(`a',$3),eval($2-($1-$3)),
      `chklim($1,$2,decr($3))',
      `set(`a',$3,incr(get(`a',$3)))``for(`k',incr($3),decr($2),
         `set(`a',k,incr(get(`a',decr(k))))')``nextcomb($1,$2)')')
define(`nextcomb',
   `show($1)
ifelse(eval(get(`a',0)<$2-$1),1,
      `chklim($1,$2,decr($1))')')
define(`comb',
   `for(`j',0,decr($1),`set(`a',j,j)')``nextcomb($1,$2)')
divert

comb(3,5)
```

Maple

This is built-in in Maple:

```
> combinat:-choose( 5, 3 );
[[1, 2, 3], [1, 2, 4], [1, 2, 5], [1, 3, 4], [1, 3, 5], [1, 4, 5], [2, 3, 4], [2, 3, 5],
```

```
    [2, 4, 5], [3, 4, 5]]
```

Mathematica/Wolfram Language

```
combinations[n_Integer, m_Integer]/;m>= 0:=Union[Sort /@ Permutations[Range[0, n - 1], {m}]]
```

built-in function example

```
Subsets[Range[5], {2}]
```

MATLAB

This a built-in function in MATLAB called "nchoosek(n,k)". The argument "n" is a vector of values from which the combinations are made, and "k" is a scalar representing the amount of values to include in each combination.

Task Solution:

```
>> nchoosek((0:4),3)

ans =

     0     1     2
     0     1     3
     0     1     4
     0     2     3
     0     2     4
     0     3     4
     1     2     3
     1     2     4
     1     3     4
     2     3     4
```

Maxima

```
next_comb(n, p, a) := block(
   [a: copylist(a), i: p],
   if a[1] + p = n + 1 then return(und),
   while a[i] - i >= n - p do i: i - 1,
   a[i]: a[i] + 1,
   for j from i + 1 thru p do a[j]: a[j - 1] + 1,
   a
)$

combinations(n, p) := block(
   [a: makelist(i, i, 1, p), v: [ ]],
   while a # 'und do (v: endcons(a, v), a: next_comb(n, p, a)),
   v
)$

combinations(5, 3);
/* [[1, 2, 3],
    [1, 2, 4],
    [1, 2, 5],
    [1, 3, 4],
    [1, 3, 5],
    [1, 4, 5],
    [2, 3, 4],
```

```
    [2, 4, 5],
    [3, 4, 5]] */
```

Modula-2

Translation of: Pascal
Works with: ADW Modula-2 version any (Compile with the linker option *Console Application*).

```
MODULE Combinations;
FROM STextIO IMPORT
  WriteString, WriteLn;
FROM SWholeIO IMPORT
  WriteInt;

CONST
  MMax = 3;
  NMax = 5;

VAR
  Combination: ARRAY [0 .. MMax] OF CARDINAL;

PROCEDURE Generate(M: CARDINAL);
VAR
  N, I: CARDINAL;
BEGIN
  IF (M > MMax) THEN
    FOR I := 1 TO MMax DO
      WriteInt(Combination[I], 1);
      WriteString(' ');
    END;
    WriteLn;
  ELSE
    FOR N := 1 TO NMax DO
      IF (M = 1) OR (N > Combination[M - 1]) THEN
        Combination[M] := N;
        Generate(M + 1);
      END
    END
  END
END Generate;

BEGIN
  Generate(1);
END Combinations.
```

Output:

```
1 2 3
1 2 4
1 2 5
1 3 4
1 3 5
1 4 5
2 3 4
2 3 5
2 4 5
3 4 5
```

Nim

```
iterator comb(m, n: int): seq[int] =
  var c = newSeq[int](n)
  for i in 0 ..< n: c[i] = i

  block outer:
```

```
      var i = n - 1
      inc c[i]
      if c[i] <= m - 1: continue

      while c[i] >= m - n + i:
        dec i
        if i < 0: break outer
      inc c[i]
      while i < n-1:
        c[i+1] = c[i] + 1
        inc i

for i in comb(5, 3):
  echo i
```

Output:

```
@[0, 1, 2]
@[0, 1, 3]
@[0, 1, 4]
@[0, 2, 3]
@[0, 2, 4]
@[0, 3, 4]
@[1, 2, 3]
@[1, 2, 4]
@[1, 3, 4]
@[2, 3, 4]
```

Another way, using a stack. Adapted from C#:

```
iterator combinations(m: int, n: int): seq[int] =

  var result = newSeq[int](n)
  var stack = newSeq[int]()
  stack.add 0

  while stack.len > 0:
    var index = stack.high
    var value = stack.pop()

    while value < m:
      result[index] = value
      inc value
      inc index
      stack.add value

      if index == n:
        yield result
        break

for i in combinations(5, 3):
  echo i
```

OCaml

```
let combinations m n =
  let rec c = function
    | (0,_) -> [[]]
    | (_,0) -> []
    | (p,q) -> List.append
               (List.map (List.cons (n-q)) (c (p-1, q-1)))
               (c (p , q-1))
  in c (m , n)
```

```
    | hd :: tl -> print_int hd ; print_string " "; print_list tl
  in List.iter print_list (combinations 3 5)
```

Octave

```
nchoosek([0:4], 3)
```

OpenEdge/Progress

Translation of: Julia

```
define variable r as integer no-undo extent 3.
define variable m as integer no-undo initial 5.
define variable n as integer no-undo initial 3.
define variable max_n as integer no-undo.

max_n = m - n.

function combinations returns logical (input pos as integer, input val as integer):
  define variable i as integer no-undo.
  do i = val to max_n:
    r[pos] = pos + i.
    if pos lt n then
        combinations(pos + 1, i).
    else
        message r[1] - 1 r[2] - 1 r[3] - 1.
  end.
end function.

combinations(1, 0).
```

Output:

0 1 2
0 1 3
0 1 4
0 2 3
0 2 4
0 3 4
1 2 3
1 2 4
1 3 4
2 3 4

Oz

This can be implemented as a trivial application of finite set constraints:

```
declare
  fun {Comb M N}
     proc {CombScript Comb}
        %% Comb is a subset of [0..N-1]
        Comb = {FS.var.upperBound {List.number 0 N-1 1}}
        %% Comb has cardinality M
        {FS.card Comb M}
        %% enumerate all possibilities
        {FS.distribute naive [Comb]}
     end
```

```
  end
in
  {Inspect {Comb 3 5}}
```

PARI/GP

```
Crv ( k, v, d ) = {
    if( d == k,
        print ( vecextract( v , "2..-2" ) )
    ,
        for( i = v[ d + 1 ] + 1, #v,
            v[ d + 2 ] = i;
            Crv( k, v, d + 1 ) ));
}

combRV( n, k ) = Crv ( k, vector( n, X, X-1), 0 );
```

```
Cr ( c, z, b, n, k ) = {
    if( z < b,  print1( c, " " );
        if( n>0, Cr( c+1, z  , b*  k  \n, n-1, k - 1 ))
    ,
        if( n>0, Cr( c+1, z-b, b*(n-k)\n, n-1, k     ))
    );
}

combR( n, k ) = {
    local(
        bnk = binomial( n,   k ),
        b11 = bnk * k \ n );          \\binomial( n-1, k-1 )
    for( z = 0,  bnk - 1,
        Cr( 1, z, b11, n-1, k-1 );
        print
    );
}
```

```
Ci( z, b, n, k ) = { local( c = 1 );
    n--; k--;
    while( k >= 0 ,
        if( z < b,
            print1(c, " ");
            c++;
            if( n > 0,
                b = b*k \ n);
            n--; k--;
        ,
            c++;
            z -= b;
            b = b*(n-k)\n;
            n--
        )
    );
    print;
}

combI( n, k ) = {
    local(  bnk = binomial( n, k ),
            b11 = bnk * k \ n );            \\ binomial( n-1, k-1 )
    for( z = 0, bnk - 1,
        Ci(z,   b11,   n, k ) );
}
```

Pascal

```
Program Combinations;
```

```
var
 combination: array [0..m_max] of integer;

 procedure generate(m: integer);
  var
   n, i: integer;
  begin
   if (m > m_max) then
    begin
    for i := 1 to m_max do
     write (combination[i], ' ');
    writeln;
    end
   else
    for n := 1 to n_max do
     if ((m = 1) or (n > combination[m-1])) then
      begin
       combination[m] := n;
       generate(m + 1);
      end;
   end;

begin
 generate(1);
end.
```

Output:

```
1 2 3
1 2 4
1 2 5
1 3 4
1 3 5
1 4 5
2 3 4
2 3 5
2 4 5
3 4 5
```

Perl

The ntheory (https://metacpan.org/pod/ntheory) module has a combinations iterator that runs in lexicographic order.

Library: ntheory

```
use ntheory qw/forcomb/;
forcomb { print "@_\n" } 5,3
```

Output:

```
0 1 2
0 1 3
0 1 4
0 2 3
0 2 4
0 3 4
1 2 3
1 2 4
1 3 4
2 3 4
```

Algorithm::Combinatorics (https://metacpan.org/pod/Algorithm::Combinatorics) also does lexicographic

Library: Algorithm::CombinatoricsCombinatorics

```
use Algorithm::Combinatorics qw/combinations/;
my @c = combinations( [0..4], 3 );
print "@$_\n" for @c;
```

```
use Algorithm::Combinatorics qw/combinations/;
my $iter = combinations([0..4],3);
while (my $c = $iter->next) {
  print "@$c\n";
}
```

Math::Combinatorics (https://metacpan.org/pod/Math::Combinatorics) is another option but results will not be in lexicographic order as specified by the task.

Perl5i

Use a recursive solution, derived from the Raku (Haskell) solution

- If we run out of eligable characters, we've gone too far, and won't find a solution along this path.
- If we are looking for a single character, each character in @set is elegible, so return each as the single element of an array.
- We have not yet reached the last character, so there are two possibilities:
  1. push the first element of the set onto the front of an N-1 length combination from the remainder of the set.
  2. skip the current element, and generate an N-length combination from the remainder

The major *Perl5i* -isms are the implicit "autoboxing" of the intermediate resulting array into an array object, with the use of unshift() as a method, and the "func" keyword and signature. Note that Perl can construct ranges of numbers or of letters, so it is natural to identify the characters as 'a' .. 'e'.

```
use perl5i::2;

# ----------------------------------------
# generate combinations of length $n consisting of characters
# from the sorted set @set, using each character once in a
# combination, with sorted strings in sorted order.
#
# Returns a list of array references, each containing one combination.
#
func combine($n, @set) {
  return unless @set;
  return map { [ $_ ] } @set if $n == 1;

  my ($head) = shift @set;
  my @result = combine( $n-1, @set );
  for my $subarray ( @result ) {
   $subarray->unshift( $head );
  }
  return ( @result, combine( $n, @set ) );
}

say @$_ for combine( 3, ('a'..'e') );
```

Output:

```
abc
abd
```

```
ade
bcd
bce
bde
cde
```

Phix

It does not get much simpler or easier than this. See Sudoku for a practical application of this algorithm

```
with javascript_semantics
procedure comb(integer pool, needed, done=0, sequence chosen={})
    if needed=0 then    -- got a full set
        ?chosen         -- (or use a routine_id, result arg, or whatever)
        return
    end if
    if done+needed>pool then return end if -- cannot fulfil
    -- get all combinations with and without the next item:
    done += 1
    comb(pool,needed-1,done,append(deep_copy(chosen),done))
    comb(pool,needed,done,chosen)
end procedure

comb(5,3)
```

Output:

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

As of 1.0.2 there is a builtin combinations() function. Using a string here for simplicity and neater output, but it works with any sequence:

```
?join(combinations("12345",3),',')
```

Output:

```
"123,124,125,134,135,145,234,235,245,345"
```

PHP

non-recursive

Full non-recursive algorithm generating all combinations without repetions. Taken from here: [1] (https://habrahabr.ru/post/311934/)

Much slower than normal algorithm.

```php
<?php

$a=array(1,2,3,4,5);
$k=3;
$n=5;
$c=array_splice($a, $k);
$b=array_splice($a, 0, $k);
$j=$k-1;
print_r($b);

        while (1) {

            $m=array_search($b[$j]+1,$c);
             if ($m!==false) {
            $c[$m]-=1;
            $b[$j]=$b[$j]+1;
                print_r($b);
        }
        if ($b[$k-1]==$n) {
     $i=$k-1;
     while ($i >= 0) {

            if ($i == 0 && $b[$i] == $n-$k+1) break 2;

              $m=array_search($b[$i]+1,$c);
          if ($m!==false) {
              $c[$m]=$c[$m]-1;
              $b[$i]=$b[$i]+1;

            $g=$i;
        while ($g != $k-1) {
            array_unshift ($c, $b[$g+1]);
            $b[$g+1]=$b[$g]+1;
            $g++;
            }
            $c=array_diff($c,$b);
            print_r($b);
                 break;
                 }
        $i--;

        }

    }


}

?>
```

Output:

```
Array
(
    [0] => 1
    [1] => 2
)
Array
(
    [0] => 1
    [1] => 3
)
Array
(
    [0] => 2
```

```
    [1] => 3
)
```

recursive

```
<?php

function combinations_set($set = [], $size = 0) {
    if ($size == 0) {
        return [[]];
    }

    if ($set == []) {
        return [];
    }


    $prefix = [array_shift($set)];

    $result = [];

    foreach (combinations_set($set, $size-1) as $suffix) {
        $result[] = array_merge($prefix, $suffix);
    }

    foreach (combinations_set($set, $size) as $next) {
        $result[] = $next;
    }

    return $result;
}

function combination_integer($n, $m) {
    return combinations_set(range(0, $n-1), $m);
}

assert(combination_integer(5, 3) == [
    [0, 1, 2],
    [0, 1, 3],
    [0, 1, 4],
    [0, 2, 3],
    [0, 2, 4],
    [0, 3, 4],
    [1, 2, 3],
    [1, 2, 4],
    [1, 3, 4],
    [2, 3, 4]
]);

echo "3 comb 5:\n";
foreach (combination_integer(5, 3) as $combination) {
    echo implode(", ", $combination), "\n";
}
```

Outputs:

```
3 comb 5:
0, 1, 2
0, 1, 3
0, 1, 4
0, 2, 3
0, 2, 4
0, 3, 4
1, 2, 3
1, 2, 4
```

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

Picat

Recursion

```
go =>
   % Integers 1..K
   N = 3,
   K = 5,
   printf("comb1(3,5): %w\n", comb1(N,K)),
   nl.

% Recursive (numbers)
comb1(M,N) = comb1_(M, 1..N).
comb1_(0, _X)     = [[]].
comb1_(_M, [])    = [].
comb1_(M, [X|Xs]) = [ [X] ++ Xs2 : Xs2 in comb1_(M-1, Xs) ] ++ comb1_(M, Xs).
```

Output:

```
comb1(3,5): [[1,2,3],[1,2,4],[1,2,5],[1,3,4],[1,3,5],[1,4,5],[2,3,4],[2,3,5],[2,4,5],[3,4,5]]
```

Using built-in power_set

```
comb2(K, N) = sort([[J : J in I] : I in power_set(1..N), I.length == K]).
```

Combinations from a list

```
go3 =>
   L = "abcde",
   printf("comb3(%d,%w): %w\n",3,L,comb3(3,L)).

comb3(M, List) = [ [List[P[I]] : I in 1..P.length] : P in comb1(M,List.length)].
```

Output:

```
comb3(3,abcde): [abc,abd,abe,acd,ace,ade,bcd,bce,bde,cde]
```

PicoLisp

Translation of: Scheme

```
(de comb (M Lst)
   (cond
      ((=0 M) '(NIL))
      ((not Lst))
      (T
         (conc
            (mapcar
               '((Y) (cons (car Lst) Y))
               (comb (dec M) (cdr Lst)) )
```

```
(comb 3 (1 2 3 4 5))
```

Pop11

Natural recursive solution: first we choose first number i and then we recursively generate all combinations of m - 1 numbers between i + 1 and n - 1. Main work is done in the internal 'do_combs' function, the outer 'comb' just sets up variable to accumulate results and reverses the final result.

The 'el_lst' parameter to 'do_combs' contains partial combination (list of numbers which were chosen in previous steps) in reverse order.

```
define comb(n, m);
    lvars ress = [];
    define do_combs(l, m, el_lst);
        lvars i;
        if m = 0 then
            cons(rev(el_lst), ress) -> ress;
        else
            for i from l to n - m do
                do_combs(i + 1, m - 1, cons(i, el_lst));
            endfor;
        endif;
    enddefine;
    do_combs(0, m, []);
    rev(ress);
enddefine;

comb(5, 3) ==>
```

PowerShell

An example of how PowerShell itself can translate C# code:

```
$source = @'
    using System;
    using System.Collections.Generic;

    namespace Powershell
    {
        public class CSharp
        {
            public static IEnumerable<int[]> Combinations(int m, int n)
            {
                int[] result = new int[m];
                Stack<int> stack = new Stack<int>();
                stack.Push(0);

                while (stack.Count > 0) {
                    int index = stack.Count - 1;
                    int value = stack.Pop();

                    while (value < n) {
                        result[index++] = value++;
                        stack.Push(value);
                        if (index == m) {
                            yield return result;
                            break;
                        }
                    }
                }
            }
        }
    }
```

```
[Powershell.CSharp]::Combinations(3,5) | Format-Wide {$_} -Column 3 -Force
```

Output:

```
0                               1                               2
0                               1                               3
0                               1                               4
0                               2                               3
0                               2                               4
0                               3                               4
1                               2                               3
1                               2                               4
1                               3                               4
2                               3                               4
```

Prolog

The solutions work with SWI-Prolog
Solution with library clpfd : we first create a list of M elements, we say that the members of the list are numbers between 1 and N and there are in ascending order, finally we ask for a solution.

```
:- use_module(library(clpfd)).

comb_clpfd(L, M, N) :-
    length(L, M),
    L ins 1..N,
    chain(L, #<),
    label(L).
```

Output:

```
 ?- comb_clpfd(L, 3, 5), writeln(L), fail.
[1,2,3]
[1,2,4]
[1,2,5]
[1,3,4]
[1,3,5]
[1,4,5]
[2,3,4]
[2,3,5]
[2,4,5]
[3,4,5]
false.
```

Another solution :

```
comb_Prolog(L, M, N) :-
    length(L, M),
    fill(L, 1, N).

fill([], _, _).

fill([H | T], Min, Max) :-
    between(Min, Max, H),
    H1 is H + 1,
    fill(T, H1, Max).
```

with the same output.

List comprehension

Works with SWI-Prolog, library **clpfd** from **Markus Triska**, and list comprehension (see List comprehensions).

```
:- use_module(library(clpfd)).
comb_lstcomp(N, M, V) :-
    V <- {L & length(L, N), L ins 1..M & all_distinct(L), chain(L, #<), label(L)}.
```

Output:

```
2?- comb_lstcomp(3, 5, V).
V = [[1,2,3],[1,2,4],[1,2,5],[1,3,4],[1,3,5],[1,4,5],[2,3,4],[2,3,5],[2,4,5],[3,4,5]] ;
false.
```

Pure

```
comb m n = comb m (0..n-1) with
  comb 0 _ = [[]];
  comb _ [] = [];
  comb m (x:xs) = [x:xs | xs = comb (m-1) xs] + comb m xs;
end;

comb 3 5;
```

PureBasic

```
Procedure.s Combinations(amount, choose)
  NewList comb.s()
  ; all possible combinations with {amount} Bits
  For a = 0 To 1 << amount
    count = 0
    ; count set bits
    For x = 0 To amount
      If (1 << x)&a
        count + 1
      EndIf
    Next
    ; if set bits are equal to combination length
    ; we generate a String representing our combination and add it to list
    If count = choose
      string$ = ""
      For x = 0 To amount
        If (a >> x)&1
          ; replace x by x+1 to start counting with 1
          String$ + Str(x) + " "
        EndIf
      Next
      AddElement(comb())
      comb() = string$
    EndIf
  Next
  ; now we sort our list and format it for output as string
  SortList(comb(), #PB_Sort_Ascending)
  ForEach comb()
    out$ + ", [ " + comb() + "]"
  Next
  ProcedureReturn Mid(out$, 3)
```

```
Debug Combinations(5, 3)
```

Pyret

```
fun combos<a>(lst :: List<a>, size :: Number) -> List<List<a>>:
  # return all subsets of lst of a certain size,
  # maintaining the original ordering of the list

  # Let's handle a bunch of degenerate cases up front
  # to be defensive...
  if lst.length() < size:
    # return an empty list if size is too big
    [list:]
  else if lst.length() == size:
    # combos([list: 1,2,3,4]) == list[list: 1,2,3,4]]
    [list: lst]
  else if size == 1:
    # combos(list: 5, 9]) == list[[list: 5], [list: 9]]
    lst.map(lam(elem): [list: elem] end)
  else:
    # The main resursive step here is to consider
    # all the combinations of the list that have the
    # first element (aka head) and then those that don't
    # don't.
    cases(List) lst:
      | empty => [list:]
      | link(head, rest) =>
        # All the subsets of our list either include the
        # first element of the list (aka head) or they don't.
        with-head-combos = combos(rest, size - 1).map(
          lam(combo):
          link(head, combo) end
          )
        without-head-combos = combos(rest, size)
        with-head-combos._plus(without-head-combos)
    end
  end
where:
  # define semantics for the degenerate cases, although
  # maybe we should just make some of these raise errors
  combos([list:], 0) is [list: [list:]]
  combos([list:], 1) is [list:]
  combos([list: "foo"], 1) is [list: [list: "foo"]]
  combos([list: "foo"], 2) is [list:]

  # test the normal stuff
  lst = [list: 1, 2, 3]
  combos(lst, 1) is [list:
    [list: 1],
    [list: 2],
    [list: 3]
  ]
  combos(lst, 2) is [list:
    [list: 1, 2],
    [list: 1, 3],
    [list: 2, 3]
  ]
  combos(lst, 3) is [list:
    [list: 1, 2, 3]
  ]

  # remember the 10th row of Pascal's Triangle? :)
  lst10 = [list: 1,2,3,4,5,6,7,8,9,10]
  combos(lst10, 3).length() is 120
  combos(lst10, 4).length() is 210
  combos(lst10, 5).length() is 252
  combos(lst10, 6).length() is 210
  combos(lst10, 7).length() is 120

  # more sanity checks...
```

```
  for each(sublst from combos(lst10, 9)):
    sublst.length() is 9
  end
end

fun int-combos(n :: Number, m :: Number) -> List<List<Number>>:
  doc: "return all lists of size m containing distinct, ordered nonnegative ints < n"
  lst = range(0, n)
  combos(lst, m)
where:
  int-combos(5, 5) is [list: [list: 0,1,2,3,4]]
  int-combos(3, 2) is [list:
    [list: 0, 1],
    [list: 0, 2],
    [list: 1, 2]
  ]
end

fun display-3-comb-5-for-rosetta-code():
  # The very concrete nature of this function is driven
  # by the web page from Rosetta Code.  We want to display
  # output similar to the top of this page:
  #
  # https://rosettacode.org/wiki/Combinations
  results = int-combos(5, 3)
  for each(lst from results):
    print(lst.join-str(" "))
  end
end

display-3-comb-5-for-rosetta-code()
```

Python

Starting from Python 2.6 and 3.0 you have a pre-defined function that returns an iterator. Here we turn the result into a list for easy printing:

```
>>> from itertools import combinations
>>> list(combinations(range(5),3))
[(0, 1, 2), (0, 1, 3), (0, 1, 4), (0, 2, 3), (0, 2, 4), (0, 3, 4), (1, 2, 3), (1, 2, 4), (1, 3, 4), (2, 3, 4)]
```

Earlier versions could use functions like the following:

Translation of: E

```
def comb(m, lst):
    if m == 0: return [[]]
    return [[x] + suffix for i, x in enumerate(lst)
            for suffix in comb(m - 1, lst[i + 1:])]
```

Example:

```
>>> comb(3, range(5))
[[0, 1, 2], [0, 1, 3], [0, 1, 4], [0, 2, 3], [0, 2, 4], [0, 3, 4], [1, 2, 3], [1, 2, 4], [1, 3, 4], [2, 3, 4]]
```

Translation of: Haskell

```
def comb(m, s):
    if m == 0: return [[]]
    if s == []: return []
    return [s[:1] + a for a in comb(m-1, s[1:])] + comb(m, s[1:])
```

A slightly different recursion version

```
def comb(m, s):
    if m == 1: return [[x] for x in s]
    if m == len(s): return [s]
    return [s[:1] + a for a in comb(m-1, s[1:])] + comb(m, s[1:])
```

Quackery

Bit Bashing

```
  [ 0 swap
    [ dup 0 != while
      dup 1 & if
        [ dip 1+ ]
      1 >> again ]
    drop ]                      is bits      (   n --> n )

  [ [] unrot
    bit times
      [ i bits
        over = if
          [ dip
              [ i join ] ] ]
    drop ]                      is combnums  ( n n --> [ )

  [ [] 0 rot
    [ dup 0 != while
      dup 1 & if
        [ dip
            [ dup dip join ] ]
       dip 1+
       1 >>
       again ]
    2drop ]                     is makecomb  (   n --> [ )

  [ over 0 = iff
      [ 2drop [] ] done
    combnums
    [] swap witheach
       [ makecomb
         nested join ] ]        is comb      ( n n --> [ )

  [ behead swap witheach max ] is largest   (   [ --> n )

  [ 0 rot witheach
      [ [ dip [ over * ] ] + ]
    nip ]                       is comborder ( [ n --> n )

  [ dup [] != while
    sortwith
      [ 2dup join
        largest 1+ dup dip
          [ comborder swap ]
        comborder < ] ]         is sortcombs (   [ --> [ )

  3 5 comb
  sortcombs
  witheach [ witheach [ echo sp ] cr ]
```

Output:

```
0 1 2
0 1 3
0 1 4
```

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

Iterative

```
  [ stack ]                             is comb.stack
  [ stack ]                             is comb.items
  [ stack ]                             is comb.required
  [ stack ]                             is comb.result

  [ 1 - comb.items put
    1+ comb.required put
    0 comb.stack put
    [] comb.result put
    [ comb.required share
      comb.stack size = if
        [ comb.result take
          comb.stack behead
          drop nested join
          comb.result put ]
      comb.stack take
      dup comb.items share
      = iff
          [ drop
            comb.stack size 1 > iff
              [ 1 comb.stack tally ] ]
            else
              [ dup comb.stack put
                1+ comb.stack put ]
             comb.stack size 1 = until ]
    comb.items release
    comb.required release
    comb.result take ]                  is comb ( n n --> )

  3 5 comb
  witheach [ witheach [ echo sp ] cr ]
```

Output:

```
0 1 2
0 1 3
0 1 4
0 2 3
0 2 4
0 3 4
1 2 3
1 2 4
1 3 4
2 3 4
```

… and a handy tool

Can be used with `comb`, and is general purpose.

```
  [ dup size dip
      [ witheach
          [ over swap peek swap ] ]
      nip pack ]                    is arrange ( [ [ --> [ )

  ' [ 10 20 30 40 50 ]
  3 5 comb
  witheach
```

```
  drop
  cr
  $ "zero one two three four" nest$
  ' [ 4 3 1 0 1 4 3 ]  arrange
  witheach [ echo$ sp ]
```

Output:

```
10 20 30
10 20 40
10 20 50
10 30 40
10 30 50
10 40 50
20 30 40
20 30 50
20 40 50
30 40 50

four three one zero one four three
```

R

```
print(combn(0:4, 3))
```

Combinations are organized per column, so to provide an output similar to the one in the task text, we need the following:

```
r <- combn(0:4, 3)
for(i in 1:choose(5,3)) print(r[,i])
```

Racket

Translation of: Haskell

```
(define sublists
  (match-lambda**
   [(0 _)           '(())]
   [(_ '())         '()]
   [(m (cons x xs)) (append (map (curry cons x) (sublists (- m 1) xs))
                            (sublists m xs))]))

(define (combinations n m)
  (sublists n (range m)))
```

Output:

```
> (combinations 3 5)
'((0 1 2)
  (0 1 3)
  (0 1 4)
  (0 2 3)
  (0 2 4)
  (0 3 4)
  (1 2 3)
  (1 2 4)
```

```
  (1 3 4)
  (2 3 4))
```

Raku

(formerly Perl 6)

Works with: rakudo version 2015.12

There actually is a builtin:

```
.say for combinations(5,3);
```

Output:

```
(0 1 2)
(0 1 3)
(0 1 4)
(0 2 3)
(0 2 4)
(0 3 4)
(1 2 3)
(1 2 4)
(1 3 4)
(2 3 4)
```

Here is an iterative routine with the same output:

```
sub combinations(Int $n, Int $k) {
    return ([],) unless $k;
    return if $k > $n || $n <= 0;
    my @c = ^$k;
    gather loop {
      take [@c];
      next if @c[$k-1]++ < $n-1;
      my $i = $k-2;
      $i-- while $i >= 0 && @c[$i] >= $n-($k-$i);
      last if $i < 0;
      @c[$i]++;
      while ++$i < $k { @c[$i] = @c[$i-1] + 1; }
    }
}
.say for combinations(5,3);
```

REXX

Version 1

This REXX program supports up to 100 symbols (one symbol for each "thing").

If *things taken at a time* is negative, the combinations aren't listed, only a count is shown.

The symbol list could be extended by added any unique viewable symbol (character).

```
/*REXX program displays combination sets for X things taken Y at a time.         */
Parse Arg things size chars .  /* get optional arguments from the command line   */
```

```
  Say 'example rexx combi , , xyzuvw'
  Say 'size<0 shows only the number of possible combinations'
  Exit
  End
If things==''|things==","  Then things=5  /* No things specified? Then use default*/
If size==''  |size==","    Then size=3    /* No size   specified? Then use default*/
If chars==''|chars==","  Then             /* No chars  specified? Then Use default*/
  chars='123456789abcdefghijklmnopqrstuvwxyz'||,
                'ABCDEFGHIJKLMNOPQRSTUVWXYZ'||,
  "~!@#chars%^&*()_+`{}|[]\:;<>?,./¦++++±==~·" /*some extended chars             */

show_details=sign(size)                 /* -1: Don't show details               */
size=abs(size)
If things<size Then
  Call exit 'Not enough things ('things') for size ('size').'
Say '------------' things 'things taken' size 'times at a time:'
Say '------------' combn(things,size) 'combinations.'
Exit                                        /* stick a fork in it,  we're all */
/*--------------------------------------------------------------------------------*/
combn: Procedure Expose chars show_details
  Parse Arg things,size
  thingsp=things+1
  thingsm=thingsp-size
  index.=0
  If things=0|size=0 Then
    Return 'no'
  Do i=1 For size
    index.i=i
    End
  done=0
  Do combi=1 By 1 Until done
    combination=''
    Do d=1 To size
      combination=combination substr(chars,index.d,1)
      End
    If show_details=1 Then
      Say combination
    index.size=index.size+1
    If index.size==thingsp Then
      done=.combn(size-1)
    End
  Return combi
/*--------------------------------------------------------------------------------*/
.combn: Procedure Expose index. size thingsm
  Parse Arg d
--Say '.combn' d thingsm show()
  If d==0 Then
    Return 1
  p=index.d
  Do u=d To size
    index.u=p+1
    If index.u==thingsm+u Then
      Return .combn(u-1)
    p=index.u
    End
  Return 0

show:
  list=''
  Do k=1 To size
    list=list index.k
    End
  Return list

exit:
  Say '*****error*****' arg(1)
  Exit 13
```

output when using the input of: 5 3 01234

```
------------ 5  things taken  3  at a time:
 0 1 2
```

```
 0 2 4
 0 3 4
 1 2 3
 1 2 4
 1 3 4
 2 3 4
------------ 10  combinations.
```

output when using the input of: `5 3 abcde`

```
------------ 5  things taken  3  at a time:
 a b c
 a b d
 a b e
 a c d
 a c e
 a d e
 b c d
 b c e
 b d e
 c d e
------------ 10  combinations.
```

output when using the input of: `44 0`

```
------------ 44  things taken  0  at a time:
------------ no  combinations.
```

output when using the input of: `52 -5`

```
------------ 52  things taken  5  at a time:
------------ 2598960  combinations.
```

output when using the input of: `5 -8`

```
*****error***** Not enough things (5) for size (8).
```

Version 2

Translation of: Java

```
/*REXX program displays combination sets for X things taken Y at a time.         */
Parse Arg things size characters
If things='?' Then Do
  Say 'rexx combi2 things size characters'
  Say ' defaults:  5      3    123456789...'
  Say 'example rexx combi2 , , xyzuvw'
  Say 'size<0 shows only the number of possible combinations'
  Exit
  End
If things==''|things==","  Then things=5  /* No things specified? Then use default*/
If size==''  |size==","    Then size=3    /* No size   specified? Then use default*/
Numeric Digits 20
show=sign(size)
size=abs(size)
If things<size Then
  Call exit 'Not enough things ('things') for size ('size').'    Say '----------' things 'things taken' size 'at a time:'
n=2**things-1
nc=0
Do u=1 to n
  nc=nc+combinations(u)
```

```
combinations: Procedure Expose things size characters show
  Parse Arg u
  nc=0
  bu=x2b(d2x(u))
  bu1=space(translate(bu,' ',0),0)
  If length(bu1)=size Then Do
    ub=reverse(bu)
    res=''
    Do i=1 To things
      If characters<>'' then
        c=substr(characters,i,1)
      Else
        c=i
      If substr(ub,i,1)=1 Then res=res c
      End
    If show=1 then
      Say res
    Return 1
    End
  Else
    Return 0
exit:
  Say '*****error*****' arg(1)
  Exit 13
```

Ring

```
# Project : Combinations

n = 5
k = 3
temp = []
comb = []
num = com(n, k)
while true
         temp = []
         for n = 1 to 3
               tm = random(4) + 1
               add(temp, tm)
         next
         bool1 = (temp[1] = temp[2]) and (temp[1] = temp[3]) and (temp[2] = temp[3])
         bool2 = (temp[1] < temp[2]) and (temp[2] < temp[3])
         if not bool1 and bool2
            add(comb, temp)
         ok
         for p = 1  to len(comb) - 1
               for q = p + 1 to len(comb)
                     if (comb[p][1] = comb[q][1]) and (comb[p][2] = comb[q][2]) and (comb[p][3] = comb[q][3])
                        del(comb, p)
                     ok
                next
         next
         if len(comb) = num
            exit
         ok
end
comb = sortfirst(comb, 1)
see showarray(comb) + nl

func com(n, k)
        res1 = 1
        for n1 = n - k + 1 to n
             res1 = res1 * n1
        next
        res2 = 1
        for n2 = 1 to k
              res2 = res2 * n2
        next
        res3 = res1/res2
        return res3
```

```
              svect = "[" + vect[nrs][1] + " " + vect[nrs][2] + " " + vect[nrs][3] + "]" + nl
              see svect
        next

Func sortfirst(alist, ind)
        aList = sort(aList,ind)
        for n = 1 to len(alist)-1
             for m= n + 1 to len(aList)
                  if alist[n][1] = alist[m][1] and alist[m][2] < alist[n][2]
                     temp = alist[n]
                     alist[n] = alist[m]
                     alist[m] = temp
                   ok
             next
        next
        for n = 1 to len(alist)-1
             for m= n + 1 to len(aList)
                  if alist[n][1] = alist[m][1] and alist[n][2] = alist[m][2] and alist[m][3] < alist[n][3]
                     temp = alist[n]
                     alist[n] = alist[m]
                     alist[m] = temp
                   ok
             next
       next
       return aList
```

Output:

```
[1 2 3]
[1 2 4]
[1 2 5]
[1 3 4]
[1 3 5]
[1 4 5]
[2 3 4]
[2 3 5]
[2 4 5]
[3 4 5]
```

RPL

Translation of: BASIC
Works with: HP version 48SX

```
≪ → currcomb start stop depth
  ≪ WHILE start stop ≤ REPEAT
      currcomb start +
      1 'start' STO+
      IF depth THEN
         start stop depth 1 - GENCOMB END
    END
≫ ≫ 'GENCOMB' STO

≪
  { } 0 4 ROLL 1 - 4 ROLL 1 - GENCOMB
≫ 'COMBS' STO
```

```
5 3 COMBS
```

Output:

```
10: { 0 1 2 }
9:  { 0 1 3 }
8:  { 0 1 4 }
```

```
4:  { 1 2 3 }
3:  { 1 2 4 }
2:  { 1 3 4 }
1:  { 2 3 4 }
```

Ruby

Works with: Ruby version 1.8.7+

```
def comb(m, n)
  (0...n).to_a.combination(m).to_a
end

comb(3, 5)  # => [[0, 1, 2], [0, 1, 3], [0, 1, 4], [0, 2, 3], [0, 2, 4], [0, 3, 4], [1, 2, 3], [1, 2, 4], [1, 3, 4], [2, 3, 4]]
```

Rust

Works with: Rust version 0.9

```
fn comb<T: std::fmt::Default>(arr: &[T], n: uint) {
  let mut incl_arr: ~[bool] = std::vec::from_elem(arr.len(), false);
  comb_intern(arr, n, incl_arr, 0);
}

fn comb_intern<T: std::fmt::Default>(arr: &[T], n: uint, incl_arr: &mut [bool], index: uint) {
  if (arr.len() < n + index) { return; }
  if (n == 0) {
    let mut it = arr.iter().zip(incl_arr.iter()).filter_map(|(val, incl)|
      if (*incl) { Some(val) } else { None }
    );
    for val in it { print!("{} ", *val); }
    print("\n");
    return;
  }

  incl_arr[index] = true;
  comb_intern(arr, n-1, incl_arr, index+1);
  incl_arr[index] = false;

  comb_intern(arr, n, incl_arr, index+1);
}

fn main() {
  let arr1 = ~[1, 2, 3, 4, 5];
  comb(arr1, 3);

  let arr2 = ~["A", "B", "C", "D", "E"];
  comb(arr2, 3);
}
```

Works with: Rust version 1.26

```
struct Combo<T> {
    data_len: usize,
    chunk_len: usize,
    min: usize,
    mask: usize,
    data: Vec<T>,
}

impl<T: Clone> Combo<T> {
    fn new(chunk_len: i32, data: Vec<T>) -> Self {
        let d_len = data.len();
        let min = 2usize.pow(chunk_len as u32) - 1;
        let max = 2usize.pow(d_len as u32) - 2usize.pow((d_len - chunk_len as usize) as u32);
```

```
            min: min,
            mask: max,
            data: data,
        }
    }

    fn get_chunk(&self) -> Vec<T> {
        let b = format!("{:01$b}", self.mask, self.data_len);
        b
           .chars()
           .enumerate()
           .filter(|&(_, e)| e == '1')
           .map(|(i, _)| self.data[i].clone())
           .collect()
    }
}

impl<T: Clone> Iterator for Combo<T> {
    type Item = Vec<T>;
    fn next(&mut self) -> Option<Self::Item> {
        while self.mask >= self.min {
            if self.mask.count_ones() == self.chunk_len as u32 {
                let res = self.get_chunk();
                self.mask -= 1;
                return Some(res);
            }
            self.mask -= 1;
        }
        None
    }
}

fn main() {
    let v1 = vec![1, 2, 3, 4, 5];
    let combo = Combo::new(3, v1);
    for c in combo.into_iter() {
        println!("{:?}", c);
    }

    let v2 = vec!("A", "B", "C", "D", "E");
    let combo = Combo::new(3, v2);
    for c in combo.into_iter() {
        println!("{:?}", c);
    }
}
```

Works with: Rust version 1.47

```
fn comb<T>(slice: &[T], k: usize) -> Vec<Vec<T>>
where
    T: Copy,
{
    // If k == 1, return a vector containing a vector for each element of the slice.
    if k == 1 {
        return slice.iter().map(|x| vec![*x]).collect::<Vec<Vec<T>>>();
    }
    // If k is exactly the slice length, return the slice inside a vector.
    if k == slice.len() {
        return vec![slice.to_vec()];
    }

    // Make a vector from the first element + all combinations of k - 1 elements of the rest of the slice.
    let mut result = comb(&slice[1..], k - 1)
        .into_iter()
        .map(|x| [&slice[..1], x.as_slice()].concat())
        .collect::<Vec<Vec<T>>>();

    // Extend this last vector with the all the combinations of k elements after from index 1 onward.
    result.extend(comb(&slice[1..], k));
    // Return final vector.
```

```
    return result;
}
```

Scala

```
implicit def toComb(m: Int) = new AnyRef {
  def comb(n: Int) = recurse(m, List.range(0, n))
  private def recurse(m: Int, l: List[Int]): List[List[Int]] = (m, l) match {
    case (0, _)   => List(Nil)
    case (_, Nil) => Nil
    case _        => (recurse(m - 1, l.tail) map (l.head :: _)) ::: recurse(m, l.tail)
  }
}
```

Usage:

```
scala> 3 comb 5
res170: List[List[Int]] = List(List(0, 1, 2), List(0, 1, 3), List(0, 1, 4), List(0, 2, 3), List(0, 2, 4), List(0, 3, 4),
 List(1, 2, 3), List(1, 2, 4), List(1, 3, 4), List(2, 3, 4))
```

Lazy version using iterators:

```
  def combs[A](n: Int, l: List[A]): Iterator[List[A]] = n match {
    case _ if n < 0 || l.lengthCompare(n) < 0 => Iterator.empty
    case 0 => Iterator(List.empty)
    case n => l.tails.flatMap({
      case Nil => Nil
      case x :: xs => combs(n - 1, xs).map(x :: _)
    })
  }
```

Usage:

```
scala> combs(3, (0 to 4).toList).toList
res0: List[List[Int]] = List(List(0, 1, 2), List(0, 1, 3), List(0, 1, 4), List(0, 2, 3), List(0, 2, 4), List(0, 3, 4), List(1,
2, 3), List(1, 2, 4), List(1, 3, 4), List(2, 3, 4))
```

Dynamic programming

Adapted from Haskell version:

```
  def combs[A](n: Int, xs: List[A]): Stream[List[A]] =
    combsBySize(xs)(n)

  def combsBySize[A](xs: List[A]): Stream[Stream[List[A]]] = {
    val z: Stream[Stream[List[A]]] = Stream(Stream(List())) ++ Stream.continually(Stream.empty)
    xs.toStream.foldRight(z)((a, b) => zipWith[Stream[List[A]]](_ ++ _, f(a, b), b))
  }

  def zipWith[A](f: (A, A) => A, as: Stream[A], bs: Stream[A]): Stream[A] = (as, bs) match {
    case (Stream.Empty, _) => Stream.Empty
    case (_, Stream.Empty) => Stream.Empty
    case (a #:: as, b #:: bs) => f(a, b) #:: zipWith(f, as, bs)
  }

  def f[A](x: A, xsss: Stream[Stream[List[A]]]): Stream[Stream[List[A]]] =
    Stream.empty #:: xsss.map(_.map(x :: _))
```

```
combs(3, (0 to 4).toList).toList
res0: List[List[Int]] = List(List(0, 1, 2), List(0, 1, 3), List(0, 1, 4), List(0, 2, 3), List(0, 2, 4), List(0, 3, 4), List(1,
2, 3), List(1, 2, 4), List(1, 3, 4), List(2, 3, 4))
```

Using Scala Standard Runtime Library

Scala REPL

```
scala>(0 to 4).combinations(3).toList
res0: List[scala.collection.immutable.IndexedSeq[Int]] = List(Vector(0, 1, 2), Vector(0, 1, 3), Vector(0, 1, 4), Vector(0, 2,
3), Vector(0, 2, 4), Vector(0, 3, 4), Vector(1, 2, 3), Vector(1, 2, 4), Vector(1, 3, 4), Vector(2, 3, 4))
```

Other environments

Output:

See it running in your browser by ScalaFiddle (JavaScript, non JVM) (https://scalafiddle.io/sf/DH34cqq/0) or by Scastie (JVM) (https://scastie.scala-lang.org/bwADub2XR8eu6bVVDbQw7g).

Scheme

Like the Haskell code:

```
(define (comb m lst)
  (cond ((= m 0) '(()))
        ((null? lst) '())
        (else (append (map (lambda (y) (cons (car lst) y))
                           (comb (- m 1) (cdr lst)))
                      (comb m (cdr lst))))))

(comb 3 '(0 1 2 3 4))
```

Seed7

```
$ include "seed7_05.s7i";

const type: combinations is array array integer;

const func combinations: comb (in array integer: arr, in integer: k) is func
  result
    var combinations: combResult is combinations.value;
  local
    var integer: x is 0;
    var integer: i is 0;
    var array integer: suffix is 0 times 0;
  begin
    if k = 0 then
      combResult := 1 times 0 times 0;
    else
      for x key i range arr do
        for suffix range comb(arr[succ(i) ..], pred(k)) do
          combResult &:= [] (x) & suffix;
        end for;
      end for;
    end if;
  end func;
```

```
    var integer: element is 0;
  begin
    for aCombination range comb([] (0, 1, 2, 3, 4), 3) do
      for element range aCombination do
        write(element lpad 3);
      end for;
      writeln;
    end for;
  end func;
```

Output:

```
  0  1  2
  0  1  3
  0  1  4
  0  2  3
  0  2  4
  0  3  4
  1  2  3
  1  2  4
  1  3  4
  2  3  4
```

SETL

```
print({0..4} npow 3);
```

Sidef

Built-in

```
combinations(5, 3, {|*c| say c })
```

Recursive

Translation of: Perl5i

```
func combine(n, set) {

    set.len || return []
    n == 1  && return set.map{[_]}

    var (head, result)
    head   = set.shift
    result = combine(n-1, [set...])

    for subarray in result {
        subarray.prepend(head)
    }

    result + combine(n, set)
}

combine(3, @^5).each {|c| say c }
```

Iterative

```
    if (k == 0) {
        callback([])
        return()
    }

    if (k<0 || k>n || n==0) {
        return()
    }

    var c = @^k

    loop {
        callback([c...])
        c[k-1]++ < n-1 && next
        var i = k-2
        while (i>=0 && c[i]>=(n-(k-i))) {
            --i
        }
        i < 0 && break
        c[i]++
        while (++i < k) {
            c[i] = c[i-1]+1
        }
    }

    return()
}

forcomb({|c| say c }, 5, 3)
```

Output:

```
[0, 1, 2]
[0, 1, 3]
[0, 1, 4]
[0, 2, 3]
[0, 2, 4]
[0, 3, 4]
[1, 2, 3]
[1, 2, 4]
[1, 3, 4]
[2, 3, 4]
```

Smalltalk

Works with: Pharo
Works with: Squeak

```
(0 to: 4) combinations: 3 atATimeDo: [ :x | Transcript cr; show: x printString].

"output on Transcript:
#(0 1 2)
#(0 1 3)
#(0 1 4)
#(0 2 3)
#(0 2 4)
#(0 3 4)
#(1 2 3)
#(1 2 4)
#(1 3 4)
#(2 3 4)"
```

SPAD

Works with: FriCAS
Works with: OpenAxiom

```
[reverse subSet(5,3,i)$SGCF for i in 0..binomial(5,3)-1]

   [[0,1,2], [0,1,3], [0,2,3], [1,2,3], [0,1,4], [0,2,4], [1,2,4], [0,3,4],
    [1,3,4], [2,3,4]]
                                                    Type: List(List(Integer))
```

SGCF (http://fricas.github.io/api/SymmetricGroupCombinatoricFunctions.html?highlight=choose) ==> SymmetricGroupCombinatoricFunctions

SparForte

As a structured script.

```
#!/usr/local/bin/spar
pragma annotate( summary, "combinations" )
       @( description, "Given non-negative integers m and n, generate all size m" )
       @( description, "combinations of the integers from 0 to n-1 in sorted" )
       @( description, "order (each combination is sorted and the entire table" )
       @( description, "is sorted" )
       @( see_also, "http://rosettacode.org/wiki/Combinations" )
       @( author, "Ken O. Burtch" );

pragma restriction( no_external_commands );

procedure combinations is
  number_of_items : constant natural := 3;
  max_item_value  : constant natural := 5;

  -- get_first_combination
  -- return the first combination (e.g. 0,1,2 for 3 items)

  function get_first_combination return string is
    c : string;
  begin
    for i in 1..number_of_items loop
      c := @ & strings.image( natural( i-1 ) );
    end loop;
    return c;
  end get_first_combination;

  -- get_last_combination
  -- return the highest value (e.g. 4,4,4 for 3 items
  -- with a maximum value of 5).

  function get_last_combination return string is
    c : string;
  begin
    for i in 1..number_of_items loop
      c := @ & strings.image( max_item_value-1 );
    end loop;
    return c;
  end get_last_combination;

  combination : string := get_first_combination;
  last_combination : constant string := get_last_combination;

  item : natural; -- a number from the combination
  bad : boolean; -- true if we know a value is too big
  s : string;    -- a temp string for deleting leading space

begin
  put_line( combination );
  while combination /= last_combination loop

    -- the combination is 3 numbers with leading spaces
    -- so the field positions start at 2 (1 is a null string)
```

```
            item := @+1;
            s := strings.image( item );
            s := strings.delete( s, 1, 1 );
            strings.replace( combination, i+1, s, ' ' );
            bad := false;
            for j in i+1..number_of_items loop
               item := numerics.value( strings.field( combination, j, ' ') );
               if item < max_item_value-1 then
                  item := @+1;
                  s := strings.image( item );
                  s := strings.delete( s, 1, 1 );
                  strings.replace( combination, j+1, s, ' ' );
               else
                  bad;
                   end if;
            end loop;
            exit;
         end if;
     end loop;
         if not bad then
            put_line( combination );
     end if;
   end loop;
end combinations;
```

Standard ML

```
fun comb (0, _    ) = [[]]
  | comb (_, []   ) = []
  | comb (m, x::xs) = map (fn y => x :: y) (comb (m-1, xs)) @
                  comb (m, xs)
;
comb (3, [0,1,2,3,4]);
```

Stata

```
program combin
tempfile cp
tempvar k
gen `k'=1
quietly save "`cp'"
rename `1' `1'1
forv i=2/`2' {
    joinby `k' using "`cp'"
    rename `1' `1'`i'
    quietly drop if `1'`i'<=`1'`=`i'-1'
}
sort `1'*
end
```

Example

```
. set obs 5
. gen a=_n
. combin a 3
. list

     +--------------+
     | a1   a2   a3 |
     |--------------|
  1. |  1    2    3 |
  2. |  1    2    4 |
  3. |  1    2    5 |
  4. |  1    3    4 |
  5. |  1    3    5 |
```

```
  8. |  2    3    5 |
  9. |  2    4    5 |
 10. |  3    4    5 |
     +--------------+
```

Mata

```
function combinations(n,k) {
    a = J(comb(n,k),k,.)
    u = 1..k
    for (i=1; 1; i++) {
        a[i,.] = u
        for (j=k; j>0; j--) {
            if (u[j]-j<n-k) break
        }
        if (j<1) return(a)
        u[j..k] = u[j]+1..u[j]+1+k-j
    }
}

combinations(5,3)
```

Output

```
        1   2   3
     +-------------+
   1 |  1   2   3  |
   2 |  1   2   4  |
   3 |  1   2   5  |
   4 |  1   3   4  |
   5 |  1   3   5  |
   6 |  1   4   5  |
   7 |  2   3   4  |
   8 |  2   3   5  |
   9 |  2   4   5  |
  10 |  3   4   5  |
     +-------------+
```

Swift

```
func addCombo(prevCombo: [Int], var pivotList: [Int]) -> [([Int], [Int])] {

  return (0..<pivotList.count)
    .map {
      _ -> ([Int], [Int]) in
      (prevCombo + [pivotList.removeAtIndex(0)], pivotList)
    }
}
func combosOfLength(n: Int, m: Int) -> [[Int]] {

  return [Int](1...m)
    .reduce([([Int](), [Int](0..<n))]) {
      (accum, _) in
      accum.flatMap(addCombo)
    }.map {
      $0.0
    }
}

println(combosOfLength(5, 3))
```

Output:

```
[[0, 1, 2], [0, 1, 3], [0, 1, 4], [0, 2, 3], [0, 2, 4], [0, 3, 4], [1, 2, 3], [1, 2, 4], [1, 3, 4], [2, 3, 4]]
```

Tcl

ref[2] (http://wiki.tcl.tk/2553)

```
proc comb {m n} {
    set set [list]
    for {set i 0} {$i < $n} {incr i} {lappend set $i}
    return [combinations $set $m]
}
proc combinations {list size} {
    if {$size == 0} {
        return [list [list]]
    }
    set retval {}
    for {set i 0} {($i + $size) <= [llength $list]} {incr i} {
        set firstElement [lindex $list $i]
        set remainingElements [lrange $list [expr {$i + 1}] end]
        foreach subset [combinations $remainingElements [expr {$size - 1}]] {
            lappend retval [linsert $subset 0 $firstElement]
        }
    }
    return $retval
}

comb 3 5 ;# ==> {0 1 2} {0 1 3} {0 1 4} {0 2 3} {0 2 4} {0 3 4} {1 2 3} {1 2 4} {1 3 4} {2 3 4}
```

TXR

TXR has repeating and non-repeating permutation and combination functions that produce lazy lists. They are generic over lists, strings and vectors. In addition, the combinations function also works over hashes.

Combinations and permutations are produced in lexicographic order (except in the case of hashes).

```
(defun comb-n-m (n m)
  (comb (range* 0 n) m))

(put-line `3 comb 5 = @(comb-n-m 5 3)`)
```

Run:

```
$ txr combinations.tl
3 comb 5 = ((0 1 2) (0 1 3) (0 1 4) (0 2 3) (0 2 4) (0 3 4) (1 2 4) (1 3 4) (2 3 4))
```

uBasic/4tH

Translation of: C

```
o = 1
Proc _Comb(5, 3, 0, 0)
End

_Comb
  Param (4)

  If a@ < b@ + d@ Then Return
```

```
    Next
    Print : Return
  EndIf

  Proc _Comb(a@, b@ - 1, OR(c@, SHL(o, d@)), d@ + 1)
  Proc _Comb(a@, b@, c@, d@ + 1)
Return
```

Output:

```
0 1 2
0 1 3
0 1 4
0 2 3
0 2 4
0 3 4
1 2 3
1 2 4
1 3 4
2 3 4

0 OK, 0:33
```

Ursala

Most of the work is done by the standard library function `choices`, whose implementation is shown here for the sake of comparison with other solutions,

```
choices = ^(iota@r,~&l); leql@a^& ~&al?\&! ~&arh2fabt2RDfalrtPXPRT
```

where `leql` is the predicate that compares list lengths. The main body of the algorithm (`~&arh2fabt2RDfalrtPXPRT`) concatenates the results of two recursive calls, one of which finds all combinations of the required size from the tail of the list, and the other of which finds all combinations of one less size from the tail, and then inserts the head into each. `choices` generates combinations of an arbitrary set but not necessarily in sorted order, which can be done like this.

```
#import std
#import nat

combinations = @rlX choices^|(iota,~&); -< @p nleq+ ==-~rh
```

- The sort combinator (`-<`) takes a binary predicate to a function that sorts a list in order of that predicate.
- The predicate in this case begins by zipping its two arguments together with `@p`.
- The prefiltering operator `-~` scans a list from the beginning until it finds the first item to falsify a predicate (in this case equality, `==`) and returns a pair of lists with the scanned items satisfying the predicate on the left and the remaining items on the right.
- The `rh` suffix on the `-~` operator causes it to return only the head of the right list as its result, which in this case will be the first pair of unequal items in the list.
- The `nleq` function then tests whether the left side of this pair is less than or equal to the right.
- The overall effect of using everything starting from the `@p` as the predicate to a sort combinator is therefore to sort a list of lists of natural numbers according to the order of the numbers in the first position where they differ.

test program:

```
example = combinations(3,5)
```

Output:

```
<
   <0,1,2>,
   <0,1,3>,
   <0,1,4>,
   <0,2,3>,
   <0,2,4>,
   <0,3,4>,
   <1,2,3>,
   <1,2,4>,
   <1,3,4>,
   <2,3,4>>
```

V

like scheme (using variables)

```
[comb [m lst] let
   [ [m zero?] [[[]]]
     [lst null?] [[]]
     [true] [m pred lst rest comb [lst first swap cons]  map
            m lst rest comb concat]
   ] when].
```

Using destructuring view and stack not *pure at all

```
[comb
   [ [pop zero?] [pop pop [[]]]
     [null?] [pop pop []]
     [true] [ [m lst : [m pred lst rest comb [lst first swap cons]  map
            m lst rest comb concat]] view i ]
   ] when].
```

Pure concatenative version

```
[comb
   [2dup [a b : a b a b] view].
   [2pop pop pop].

   [ [pop zero?] [2pop [[]]]
     [null?] [2pop []]
     [true] [2dup [pred] dip uncons swapd comb [cons] map popd rollup rest comb concat]
   ] when].
```

Using it

```
|3 [0 1 2 3 4] comb
=[[0 1 2] [0 1 3] [0 1 4] [0 2 3] [0 2 4] [0 3 4] [1 2 3] [1 2 4] [1 3 4] [2 3 4]]
```

VBA

```
Option Explicit
Option Base 0
```

```
Sub test()
    'compute
    Main_Combine 5, 3

    'return
    Dim j As Long, i As Long, temp As String
    For i = LBound(ArrResult, 1) To UBound(ArrResult, 1)
        temp = vbNullString
        For j = LBound(ArrResult, 2) To UBound(ArrResult, 2)
            temp = temp & " " & ArrResult(i, j)
        Next
        Debug.Print temp
    Next
    Erase ArrResult
End Sub

Private Sub Main_Combine(M As Long, N As Long)
Dim MyArr, i As Long
    ReDim MyArr(M - 1)
    If LBound(MyArr) > 0 Then ReDim MyArr(M) 'Case Option Base 1
    For i = LBound(MyArr) To UBound(MyArr)
        MyArr(i) = i
    Next i
    i = IIf(LBound(MyArr) > 0, N, N - 1)
    ReDim ArrResult(i, LBound(MyArr))
    Combine MyArr, N, LBound(MyArr), LBound(MyArr)
    ReDim Preserve ArrResult(UBound(ArrResult, 1), UBound(ArrResult, 2) - 1)
    'In VBA Excel we can use Application.Transpose instead of personal Function Transposition
    ArrResult = Transposition(ArrResult)
End Sub

Private Sub Combine(MyArr As Variant, Nb As Long, Deb As Long, Ind As Long)
Dim i As Long, j As Long, N As Long
    For i = Deb To UBound(MyArr, 1)
        ArrResult(Ind, UBound(ArrResult, 2)) = MyArr(i)
        N = IIf(LBound(ArrResult, 1) = 0, Nb - 1, Nb)
        If Ind = N Then
            ReDim Preserve ArrResult(UBound(ArrResult, 1), UBound(ArrResult, 2) + 1)
            For j = LBound(ArrResult, 1) To UBound(ArrResult, 1)
                ArrResult(j, UBound(ArrResult, 2)) = ArrResult(j, UBound(ArrResult, 2) - 1)
            Next j
        Else
            Call Combine(MyArr, Nb, i + 1, Ind + 1)
        End If
    Next i
End Sub

Private Function Transposition(ByRef MyArr As Variant) As Variant
Dim T, i As Long, j As Long
    ReDim T(LBound(MyArr, 2) To UBound(MyArr, 2), LBound(MyArr, 1) To UBound(MyArr, 1))
    For i = LBound(MyArr, 1) To UBound(MyArr, 1)
        For j = LBound(MyArr, 2) To UBound(MyArr, 2)
            T(j, i) = MyArr(i, j)
        Next j
    Next i
    Transposition = T
    Erase T
End Function
```

Output:

If Option Base 0 :

```
 0 1 2
 0 1 3
 0 1 4
 0 2 3
 0 2 4
 0 3 4
 1 2 3
 1 2 4
```

If Option Base 1 :

```
 1 2 3
 1 2 4
 1 2 5
 1 3 4
 1 3 5
 1 4 5
 2 3 4
 2 3 5
 2 4 5
 3 4 5
```

Translation of: Phix

```
Private Sub comb(ByVal pool As Integer, ByVal needed As Integer, Optional ByVal done As Integer = 0, Optional ByVal chosen As
Variant)
    If needed = 0 Then  '-- got a full set
        For Each x In chosen: Debug.Print x;: Next x
        Debug.Print
        Exit Sub
    End If
    If done + needed > pool Then Exit Sub '-- cannot fulfil
    '-- get all combinations with and without the next item:
    done = done + 1
    Dim tmp As Variant
    tmp = chosen
    If IsMissing(chosen) Then
        ReDim tmp(1)
    Else
        ReDim Preserve tmp(UBound(chosen) + 1)
    End If
    tmp(UBound(tmp)) = done
    comb pool, needed - 1, done, tmp
    comb pool, needed, done, chosen
End Sub

Public Sub main()
    comb 5, 3
End Sub
```

VBScript

```
Function Dec2Bin(n)
    q = n
    Dec2Bin = ""
    Do Until q = 0
        Dec2Bin = CStr(q Mod 2) & Dec2Bin
        q = Int(q / 2)
    Loop
    Dec2Bin = Right("00000" & Dec2Bin,6)
End Function

Sub Combination(n,k)
    Dim arr()
    ReDim arr(n-1)
    For h = 0 To n-1
        arr(h) = h + 1
    Next
    Set list = CreateObject("System.Collections.Arraylist")
    For i = 1 To 2^n
        bin = Dec2Bin(i)
        c = 0
        tmp_combo = ""
        If Len(Replace(bin,"0","")) = k Then
            For j = Len(bin) To 1 Step -1
                If CInt(Mid(bin,j,1)) = 1 Then
                    tmp_combo = tmp_combo & arr(c) & ","
```

```
            list.Add Mid(tmp_combo,1,(k*2)-1)
        End If
    Next
    list.Sort
    For l = 0 To list.Count-1
        WScript.StdOut.Write list(l)
        WScript.StdOut.WriteLine
    Next
End Sub

'Testing with n = 5 / k = 3
Call Combination(5,3)
```

Output:

```
1,2,3
1,2,4
1,2,5
1,3,4
1,3,5
1,4,5
2,3,4
2,3,5
2,4,5
3,4,5
```

Wren

Library: Wren-perm

```
import "./perm" for Comb

var fib = Fiber.new { Comb.generate((0..4).toList, 3) }
while (true) {
    var c = fib.call()
    if (!c) return
    System.print(c)
}
```

Output:

```
[0, 1, 2]
[0, 1, 3]
[0, 1, 4]
[0, 2, 3]
[0, 2, 4]
[0, 3, 4]
[1, 2, 3]
[1, 2, 4]
[1, 3, 4]
[2, 3, 4]
```

XPLo

```
code ChOut=8, CrLf=9, IntOut=11;
def M=3, N=5;
int A(N-1);

proc Combos(D, S);      \Display all size M combinations of N in sorted order
int  D, S;              \depth of recursion, starting value of N
int  I;
[if D<M then            \depth < size
      for I:= S to N-1 do
```

```
else [for I:= 0 to M-1 do
        [IntOut(0, A(I));  ChOut(0, ^ )];
     CrLf(0);
     ];
];

Combos(0, 0)
```

Output:

```
0 1 2
0 1 3
0 1 4
0 2 3
0 2 4
0 3 4
1 2 3
1 2 4
1 3 4
2 3 4
```

zkl

Translation of: OCaml

```
fcn comb(k,seq){    // no repeats, seq is finite
   seq=seq.makeReadOnly();  // because I append to parts of seq
   fcn(k,seq){
      if(k<=0)    return(T(T));
      if(not seq) return(T);
      self.fcn(k-1,seq[1,*]).pump(List,seq[0,1].extend)
       .extend(self.fcn(k,seq[1,*]));
   }(k,seq);
}
```

```
comb(3,"abcde".split("")).apply("concat")
```

Output:

```
L("abc","abd","abe","acd","ace","ade","bcd","bce","bde","cde")
```

Retrieved from "https://rosettacode.org/wiki/Combinations?oldid=348688"