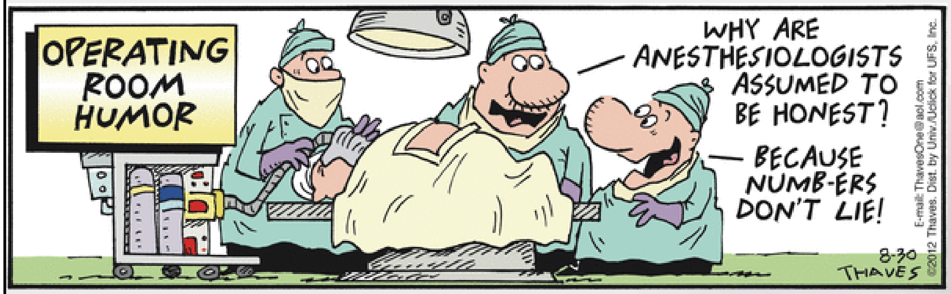
MA/CSSE 473
Day 13

Finish
Topological Sort

Permutation
Generation

MA/CSSE 473 Day 13

- Student Questions
- Finish Topological Sort
- Permutation generation



Recap: Topologically sort a DAG

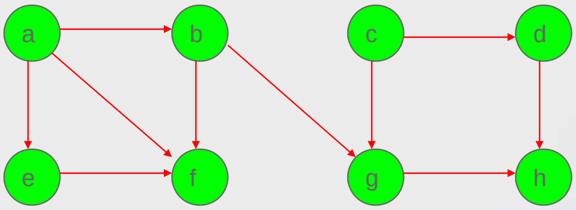
- DAG = Directed Aclyclic Graph
- Linearly order the vertices of the DAG so that for every edge e, e's tail vertex precedes its head vertex in the ordering.

DFS-based Algorithm

DFS-based algorithm for topological sorting

- Perform DFS traversal, noting the order vertices are popped off the traversal stack
- Reversing order solves topological sorting problem
- Back edges encountered?→ NOT a dag!

Example:

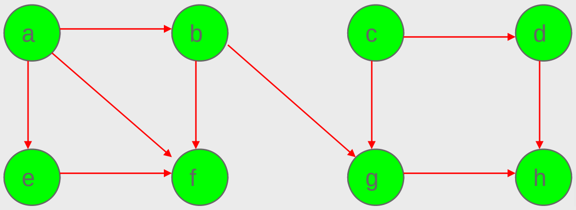


Efficiency:

Source Removal Algorithm

Repeatedly identify and remove a *source* (a vertex with no incoming edges) and all the edges incident to it until either no vertex is left (problem is solved) or there is no source among remaining vertices (not a dag)

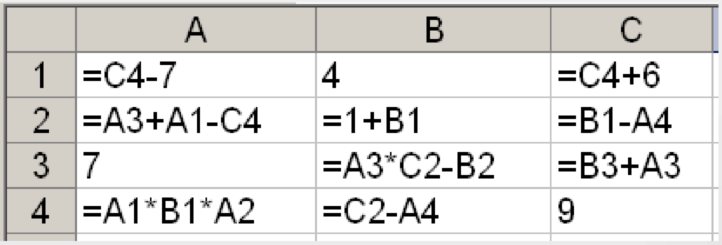
Example:



Efficiency: same as efficiency of the DFS-based algorithm

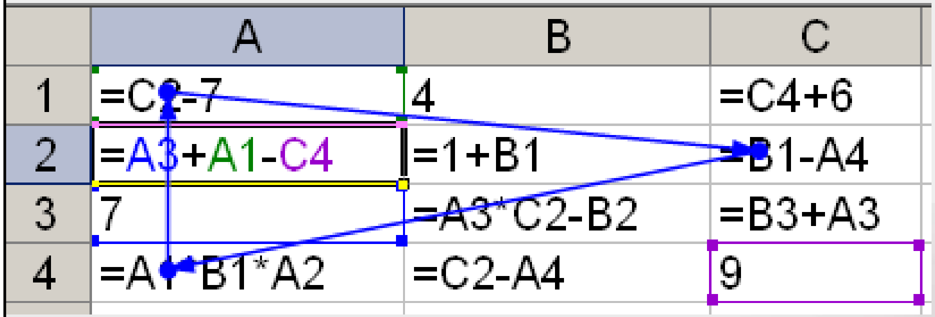
Application: Spreadsheet program

- What is an allowable order of computation of the cells' values?



	A	B	C
1	=C4-7	4	=C4+6
2	=A3+A1-C4	=1+B1	=B1-A4
3	7	=A3*C2-B2	=B3+A3
4	=A1*B1*A2	=C2-A4	9

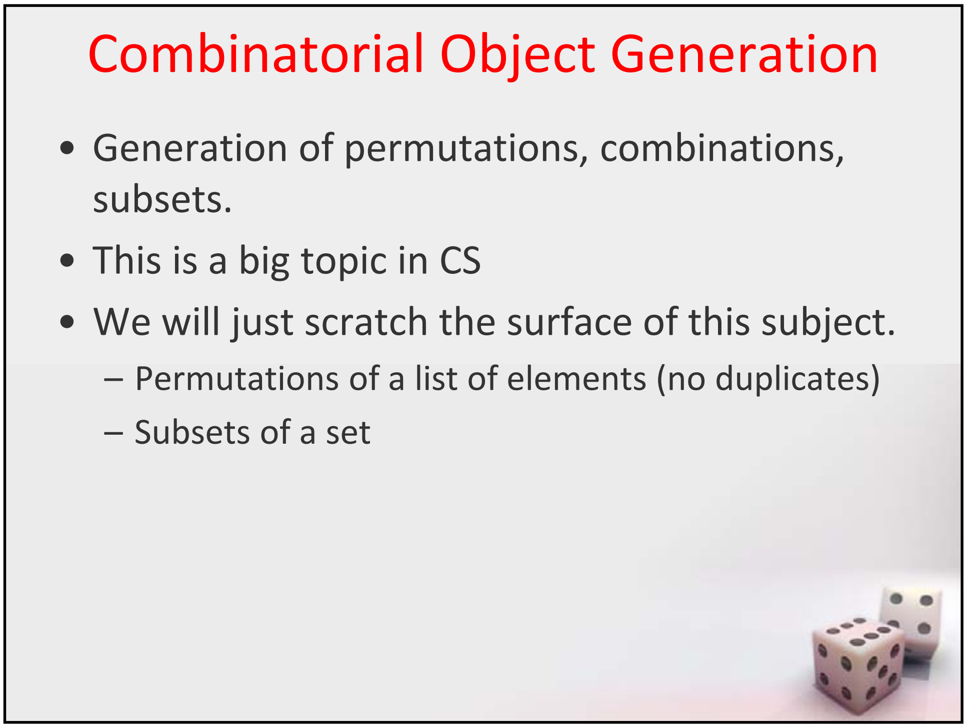
Cycles cause a problem!



	A	B	C
1	=C2-7	4	=C4+6
2	=A3+A1-C4	=1+B1	=B1-A4
3	7	=A3*C2-B2	=B3+A3
4	=A1+B1*A2	=C2-A4	9

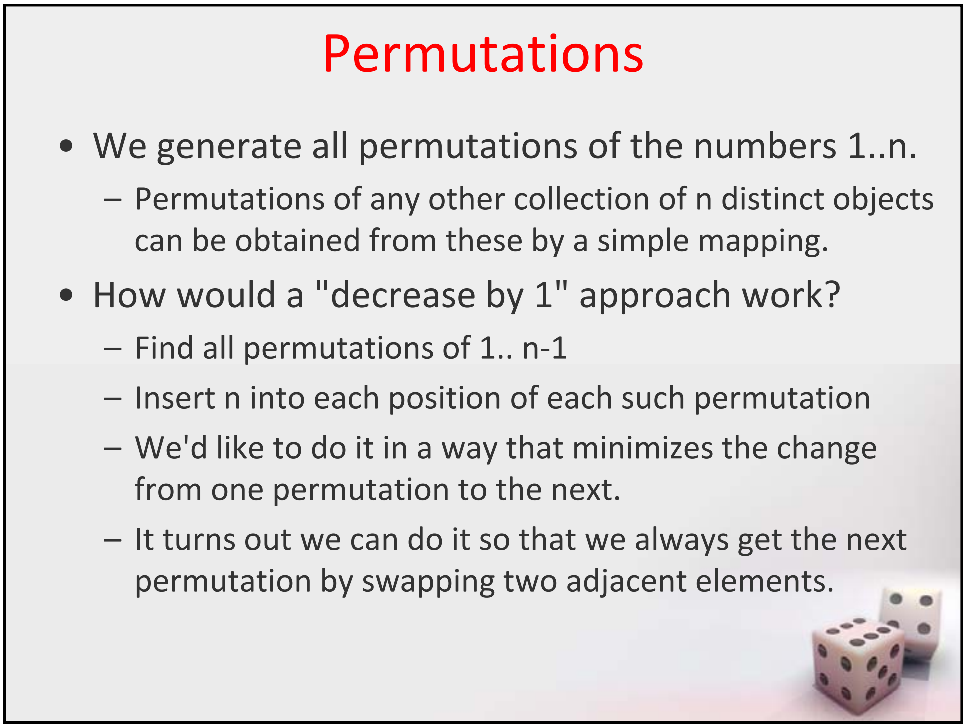
Permutations
Subsets

COMBINATORIAL OBJECT GENERATION



Combinatorial Object Generation

- Generation of permutations, combinations, subsets.
- This is a big topic in CS
- We will just scratch the surface of this subject.
  - Permutations of a list of elements (no duplicates)
  - Subsets of a set



Permutations

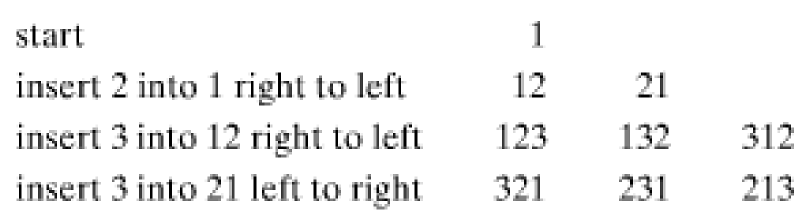
- We generate all permutations of the numbers 1..n.
  - Permutations of any other collection of n distinct objects can be obtained from these by a simple mapping.
- How would a "decrease by 1" approach work?
  - Find all permutations of 1.. n-1
  - Insert n into each position of each such permutation
  - We'd like to do it in a way that minimizes the change from one permutation to the next.
  - It turns out we can do it so that we always get the next permutation by swapping two adjacent elements.

First approach we might think of

- for each permutation of 1..n-1
  - for i=0..n-1
    - insert n in position i
- That is, we do the insertion of n into each smaller permutation from left to right each time
- However, to get "minimal change", we alternate:
  - Insert n L-to-R in one permutation of 1..n-1
  - Insert n R-to-L in the next permutation of 1..n-1
  - Etc.

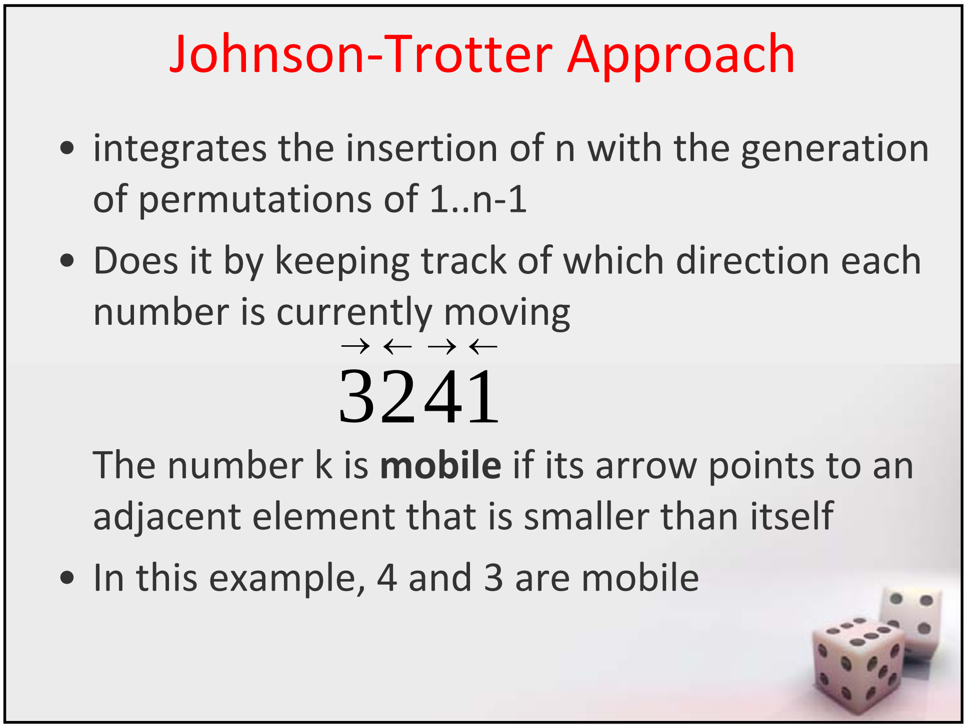
Example

- Bottom-up generation of permutations of 123



start	1		
insert 2 into 1 right to left	12	21	
insert 3 into 12 right to left	123	132	312
insert 3 into 21 left to right	321	231	213

- Example: Do the first few permutations for n=4

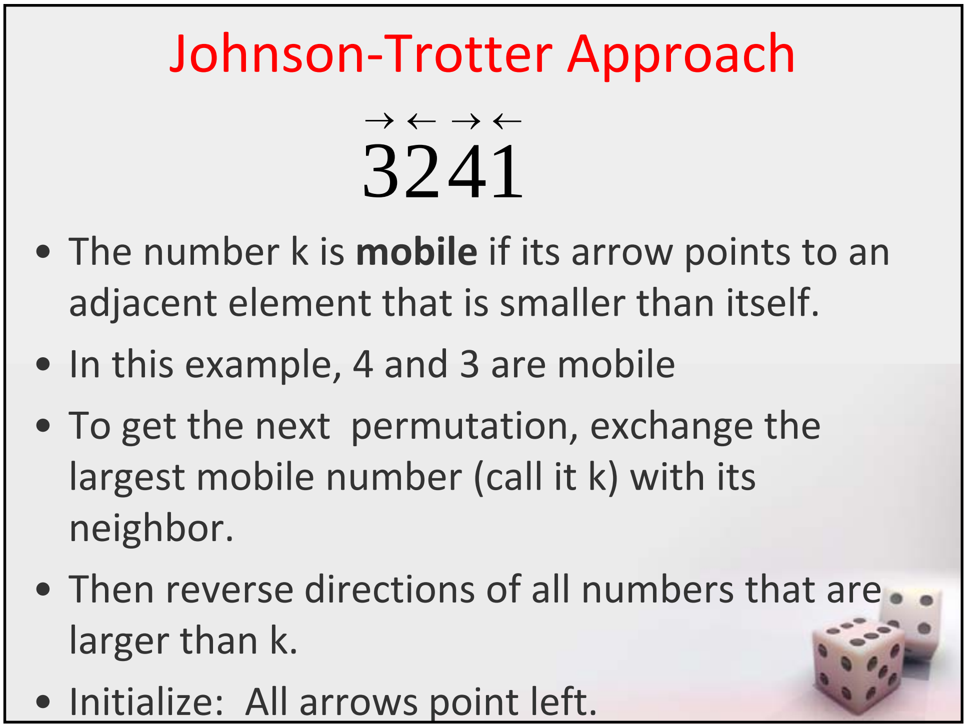


Johnson-Trotter Approach

- integrates the insertion of n with the generation of permutations of 1..n-1
- Does it by keeping track of which direction each number is currently moving

$$\overset{\rightarrow}{3}\overset{\leftarrow}{2}\overset{\rightarrow}{4}\overset{\leftarrow}{1}$$

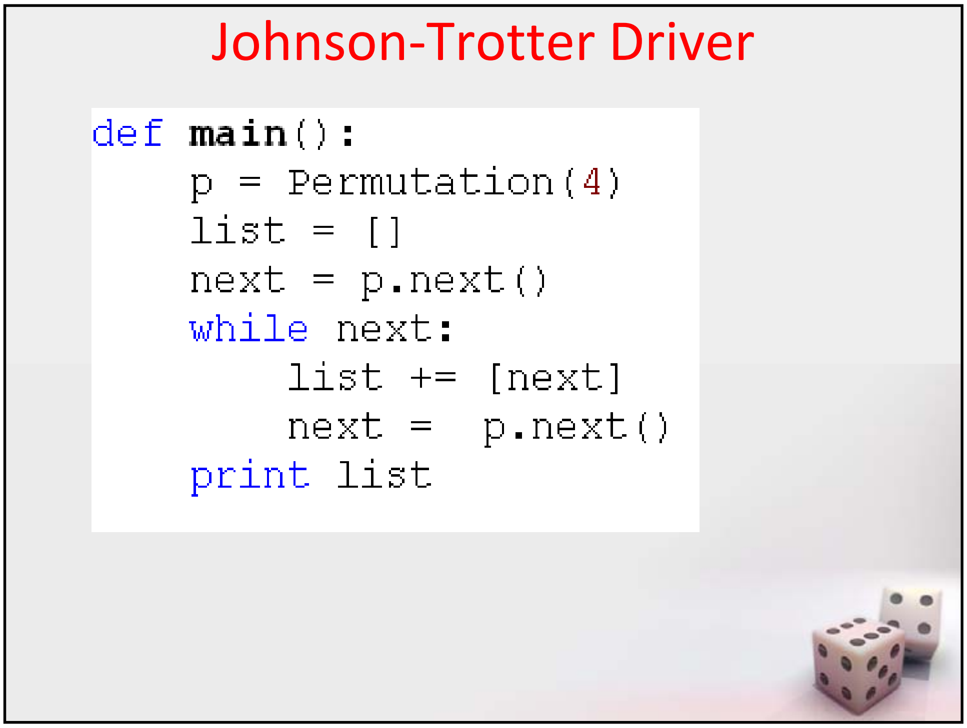
  The number k is **mobile** if its arrow points to an adjacent element that is smaller than itself
- In this example, 4 and 3 are mobile



Johnson-Trotter Approach

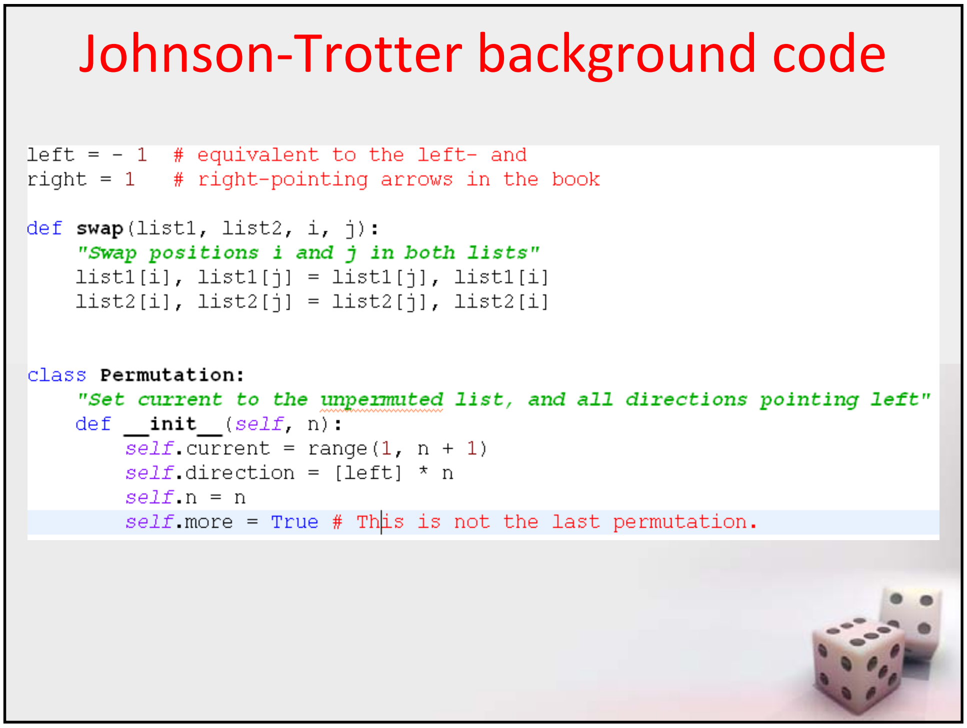
$$\overset{\rightarrow}{3}\overset{\leftarrow}{2}\overset{\rightarrow}{4}\overset{\leftarrow}{1}$$

- The number k is **mobile** if its arrow points to an adjacent element that is smaller than itself.
- In this example, 4 and 3 are mobile
- To get the next permutation, exchange the largest mobile number (call it k) with its neighbor.
- Then reverse directions of all numbers that are larger than k.
- Initialize: All arrows point left.



Johnson-Trotter Driver

```
def main():
    p = Permutation(4)
    list = []
    next = p.next()
    while next:
        list += [next]
        next =  p.next()
    print list
```



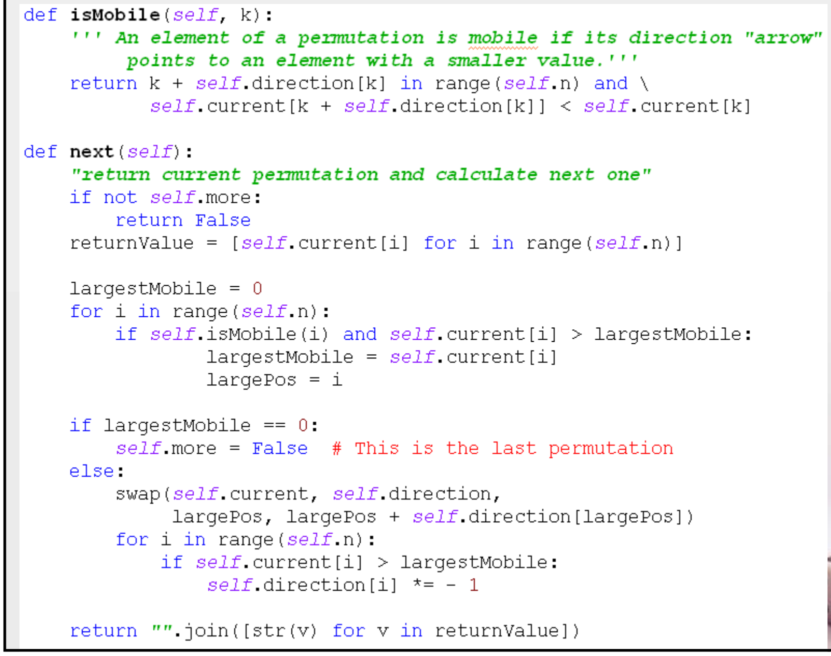
Johnson-Trotter background code

```
left = - 1  # equivalent to the left- and
right = 1   # right-pointing arrows in the book

def swap(list1, list2, i, j):
    "Swap positions i and j in both lists"
    list1[i], list1[j] = list1[j], list1[i]
    list2[i], list2[j] = list2[j], list2[i]


class Permutation:
    "Set current to the unpermuted list, and all directions pointing left"
    def __init__(self, n):
        self.current = range(1, n + 1)
        self.direction = [left] * n
        self.n = n
        self.more = True # This is not the last permutation.
```

Johnson-Trotter major methods



```
def isMobile(self, k):
    ''' An element of a permutation is mobile if its direction "arrow"
         points to an element with a smaller value.'''
    return k + self.direction[k] in range(self.n) and \
           self.current[k + self.direction[k]] < self.current[k]

def next(self):
    "return current permutation and calculate next one"
    if not self.more:
        return False
    returnValue = [self.current[i] for i in range(self.n)]

    largestMobile = 0
    for i in range(self.n):
        if self.isMobile(i) and self.current[i] > largestMobile:
                largestMobile = self.current[i]
                largePos = i

    if largestMobile == 0:
        self.more = False  # This is the last permutation
    else:
        swap(self.current, self.direction,
             largePos, largePos + self.direction[largePos])
        for i in range(self.n):
            if self.current[i] > largestMobile:
                self.direction[i] *= - 1

    return "".join([str(v) for v in returnValue])
```

Lexicographic Permutation Generation

- Generate the permutations of 1..n in "natural" order.
- Let's do it recursively.

Lexicographic Permutation Code

```
def permuterecursive(prefix, remaining):
    """ Generate all lists that begin with prefix and
        end with a permutation of remaining"""
    if remaining == []: # this is where the recursion ends
        return [prefix]
    result = [] # accumlate the list of generated prefixes
    for n in remaining:
        copy = [e for e in remaining] # need to remove a different
        copy.remove(n)          # number for each suffix we generate.
        result += permuterecursive(prefix + [n], copy)
    return result

def permute(n):
    return permuterecursive([], range(1, n+1))

print (permute(4))
```

Permutations and order

number	permutation	number	permutation
0	0123	12	2013
1	0132	13	2031
2	0213	14	2103
3	0231	15	2130
4	0312	16	2301
5	0321	17	2310
6	1023	18	3012
7	1032	19	3021
8	1203	20	3102
9	1230	21	3120
10	1302	22	3201
11	1320	23	3210

- Given a permutation of 0, 1, …, n-1, can we directly find the next permutation in the lexicographic sequence?
- Given a permutation of 0..n-1, can we determine its permutation sequence number?

- Given n and i, can we directly generate the ith permutation of 0, …, n-1?

Discovery time (with two partners)

- Which permutation follows each of these in lexicographic order?
  - 183647520 471638520
  - Try to write an algorithm for generating the next permutation, with only the current permutation as input.
- If the lexicographic permutations of the numbers [0, 1, 2, 3, 4, 5] are numbered starting with 0, what is the number of the permutation 14032?
  - General form? How to calculate efficiency?
- In the lexicographic ordering of permutations of [0, 1, 2, 3, 4, 5], which permutation is number 541?
  - How to calculate efficiently?