Permutations / Combinations

1. Combinations : $\{1, \dots, n\}$, k — Visit every combination of k objects out of n.

# of things in output: $\binom{n}{k}$.   Ex: $n = 4$ $k = 2$

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

Setup: Boolean Array $A[1..n]$.   Initial: count ← 0

```
Visit (A) { // A will have exactly k true values.
            // "Visiting" the subset corresponding to the true values.
    Ex:  for i ← 1 to n do
             if A[i] then
                 Print i
    Count ++
```

```
Combination (A, n, k)            // Precondition: A[1..n] = false
    if k > n then
        return
    else if k = 0 then
        Visit(A)
    else      // choose A[n] ?
        // case 1: n ~~A[n]~~ is not selected
        Combination (A, n-1, k)
        // case 2: n ~~A[n]~~ is selected
        A[n] ← true
        Combination (A, n-1, k-1)
        A[n] ← false      // clean up
```

RT = $O\left(\binom{n}{k} \cdot \text{Time to visit one combination}\right)$

Permutations : Take 1 (Naive algorithm)

// Output all $n!$ permutations. of $1..n$

```
Permute (A, i)
                                        // Initial condition:
                                           A[i] = 0
// Precondition:
   i+1, i+2, ... n have already been
      placed in A.
   if i = 0 then Visit (A).
   else  // Place i into A in some empty spot
      for k ← 1 to n do
         if A[k] = 0 then
            A[k] ← i
            Permute (A, i-1)
            A[k] ← 0
```

RT= $O(n \cdot n! \cdot \text{Time to visit a permutation})$.

Initial call: Permute (A, A.length)

Permutations : Take 2

```
                          // Precondition: A[i] = i
Permute (A, i)            // A[i+1 .. n] are frozen.
   if i = 0 then Visit (A)
   else
      for j ← 1 to i do
         swap A[j] ↔ A[i]
         Permute (A, i-1)
         swap ~~(~~A[j] ↔ A[i]   //clean up
```

RT= $O(n! \ast \text{Time to visit one permutation})$.

Permutations : Take 3: Heap's algorithm.

start with 1 2 3 ... n

In each step, exchange 2 elements that leads to the next permutation. — RT= 1 swap per permutation

$n!$ + Time to visit each permutation

Generating permutations of non distinct numbers (elements) - Knuth's L Algorithm.

- in lexicographic order.

Ex: $-\infty\ 1\ 2\ 2\ 3\ \underset{j}{\textcircled{3}}\ \underset{l}{\textcircled{4}}$

$j =$ rightmost index such that $A[j] < A[j+1]$

$l = \overset{\text{max}}{\text{index}}$ such that $A[j] < A[l]$

$1\ 2\ 2\ \underset{j}{\textcircled{3}}\ \underset{l}{\textcircled{4}}\ 3$

$1\ 2\ \underset{j}{\textcircled{2}}\ 4\ 3\ \underset{l}{\textcircled{3}}$

$1\ 2\ 3\ 2\ \underset{j}{\textcircled{3}}\ \textcircled{4}\ l$

$1\ 2\ 3\ \underset{j}{\textcircled{2}}\ 4\ \underset{l}{\textcircled{3}}$

$1\ \underset{\cancel{j}}{\textcircled{2}}\ 3\ 3\ \underset{j}{\textcircled{2}}\ \underset{l}{\textcircled{4}}$

$1\ 2\ 3\ \underset{j}{\textcircled{3}}\ \underset{l}{\textcircled{4}}\ 2$

$1\ 2\ 3\ 4\ \underset{j}{\textcircled{2}}\ \underset{l}{\textcircled{3}}$ $\quad$ $1\ 3\ 2\ 2\ 3\ 4$

$\vdots$

$1\ 2\ \underset{j}{\textcircled{3}}\ \underset{l}{\textcircled{4}}\ 3\ 2$

$1\ 2\ 4\ \underset{j}{\textcircled{2}}\ 3\ \underset{l}{\textcircled{3}}$

$1\ 2\ 4\ 3\ \underset{j}{\textcircled{2}}\ \underset{l}{\textcircled{3}}$

$1\ \underset{j}{\textcircled{2}}\ 4\ 3\ \underset{l}{\textcircled{3}}\ 2$

Knuth's algorithm

```
Input:  $a_0 < a_1 \le a_2 \le \dots \le a_n$    ($a_0$ = sentinel, not part of input)

  Visit $a_1 \cdots a_n$
Step2: Find max value of $j$ such that $a_j < a_{j+1}$
   if $j = 0$ then stop.
   Find max value of $l$ such that $a_j < a_l$.
   Exchange $a_j \leftrightarrow a_l$     // After this step,
                                           $a_{j+1} \cdots a_n$ in descending order
   Reverse $a_{j+1} \cdots a_n$
   Visit permutation and go to step 2
```

$$RT = O(n * \#\text{ of } \overset{\text{distinct}}{\text{permutations}} * \text{Time to visit a permutation})$$

↑ k distinct values, $n_1, n_2 \dots n_k$ copies of each

$$\frac{n!}{n_1!\, n_2! \dots n_k!}$$

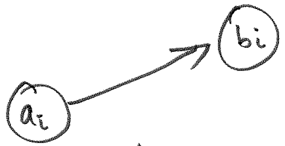
Enumerating Topological orderings of DAGs

Input: a set of precedences (a_1, b_1) (a_2, b_2) … (a_k, b_k)

Output: Permutations of 1..n in which

a_i precedes b_i for $i = 1..k$.

Construct a graph from the list of precedences.



If graph has a cycle ⟶ # of permutations = 0

$\boxed{\text{DAG}}$ — List all topological orders of DAG.

Think about it ⟶ in next SP.