Efficiently Generating Classical and Vincular Pattern Avoiding Permutations Based on Permutation Decision Diagrams

Yuma Inoue[1], Takahisa Toda[2], Shin-ichi Minato[1,2]

Hokkaido University[1], JST ERATO project[2]

Main result

- Provide efficient algorithms for generating all permutations which avoid classical pattern σ.
  - use permutation decision diagrams (πDD)
  - fast and low memory usage on computational experiment
- Extend the algorithm to generate vincular pattern avoiding permutations.

Outline

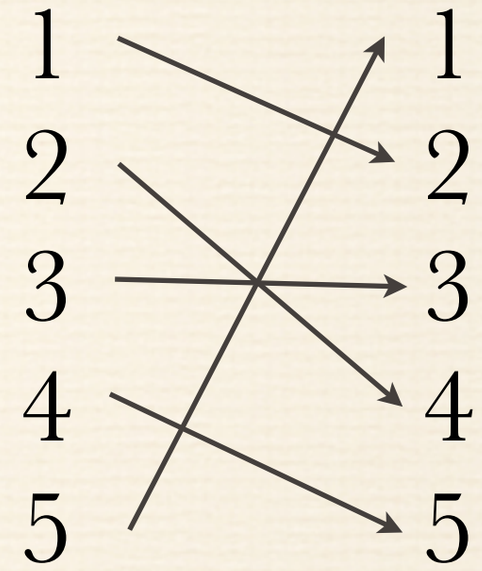
- Preparation
  - Permutation Pattern
    - πDD (Permutation Decision Diagrams)
- Our Algorithm
- Experimental Result
- Conclusion

Permutation Pattern

Permutation

- Permutation: bijection on {1, 2, ... n}

- Example: $\pi = \begin{pmatrix} 1 & 2 & 3 & 4 & 5 \\ 2 & 4 & 3 & 5 & 1 \end{pmatrix}$



- We use one-line notation, e.g., $\pi = (2\ 4\ 3\ 5\ 1)$.

Classical Pattern Avoiding

$\pi = (\pi_1\ \pi_2 \ldots \pi_n)$: permutation

$\sigma = (\sigma_1\text{-}\sigma_2\text{-}\ldots\text{-}\sigma_k)$: classical pattern

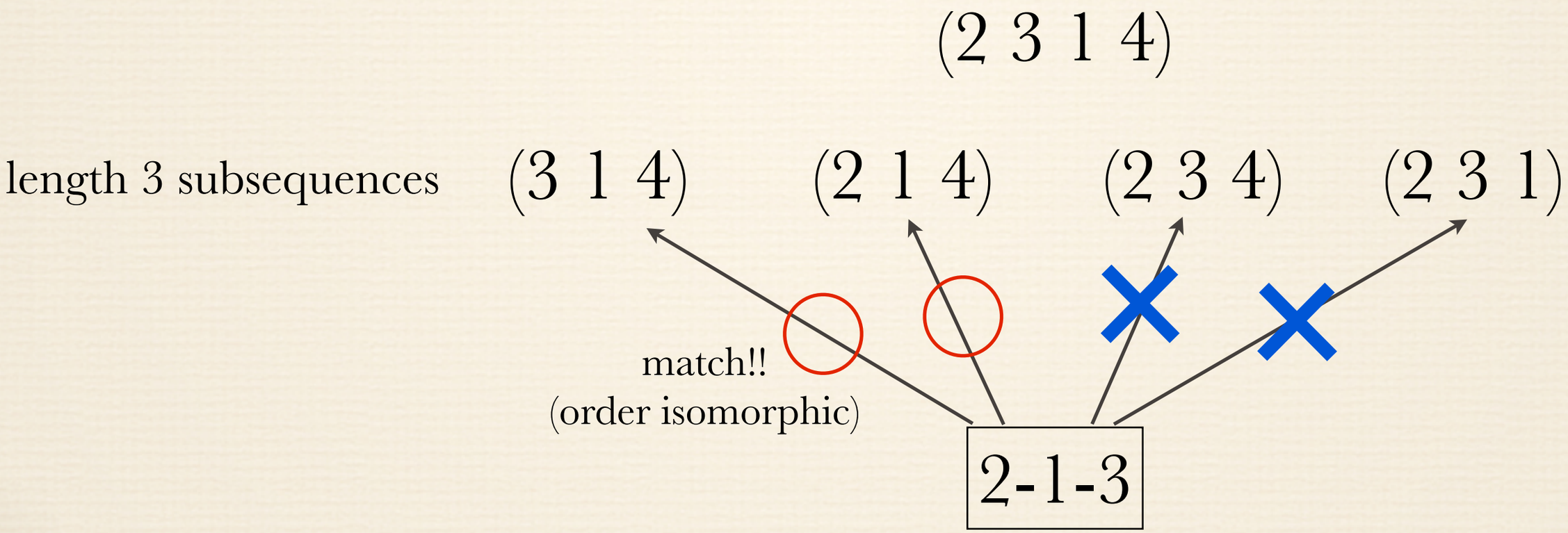
Definition:

- π **contains** σ if there is length k subsequence of π, $(\pi_{i_1}\ \pi_{i_2} \ldots \pi_{i_k})$ with $1 \leqq i_1 < i_2 < \ldots < i_k \leqq n$, such that $\pi_{i_x} < \pi_{i_y}$ iff $\sigma_x < \sigma_y$.
- Otherwise, π **avoids** σ.

- Note: "-" means "not have to be adjacent".

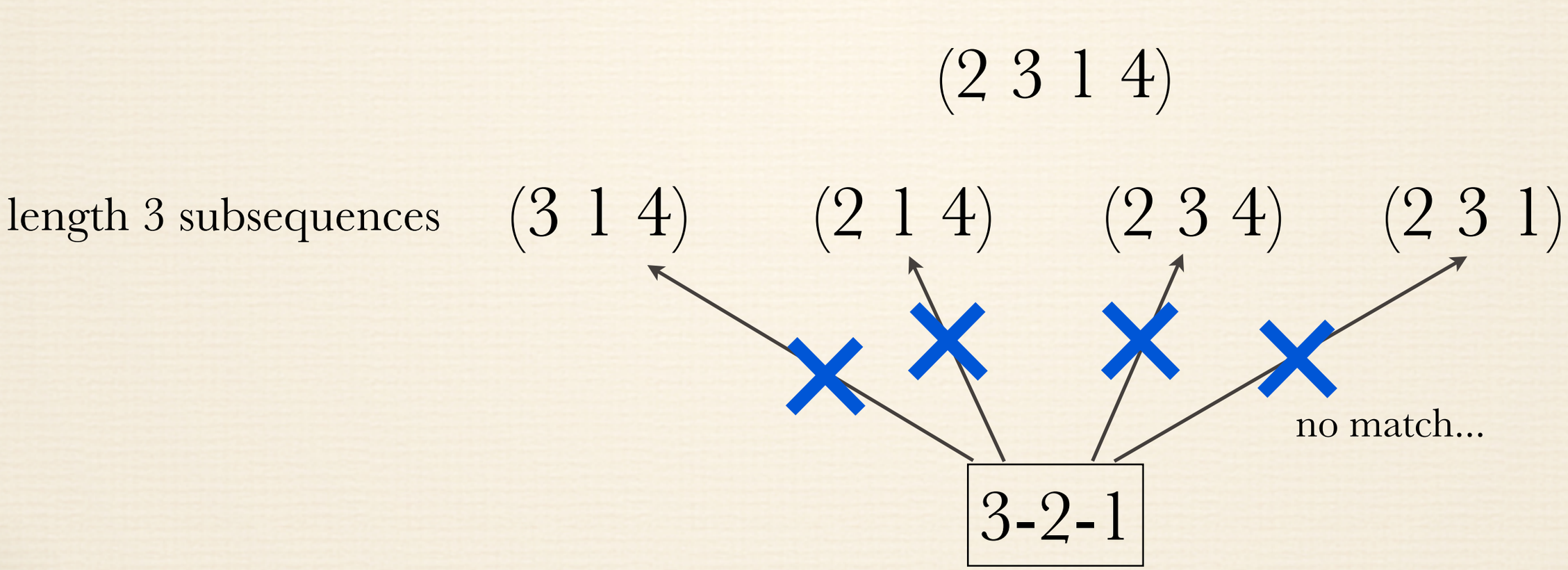
Classical Pattern Avoiding

- Example:
  - (2 3 1 4) contains (2-1-3), but avoids (3-2-1).



Classical Pattern Avoiding

- Example:
  - (2 3 1 4) contains (2-1-3), but avoids (3-2-1).



Problem

- Let $\mathbf{A_n(\sigma)}$ be the set of n-permutations which avoid σ.

Problem:

Generate $A_n(\sigma)$ for given positive integer n and pattern σ.

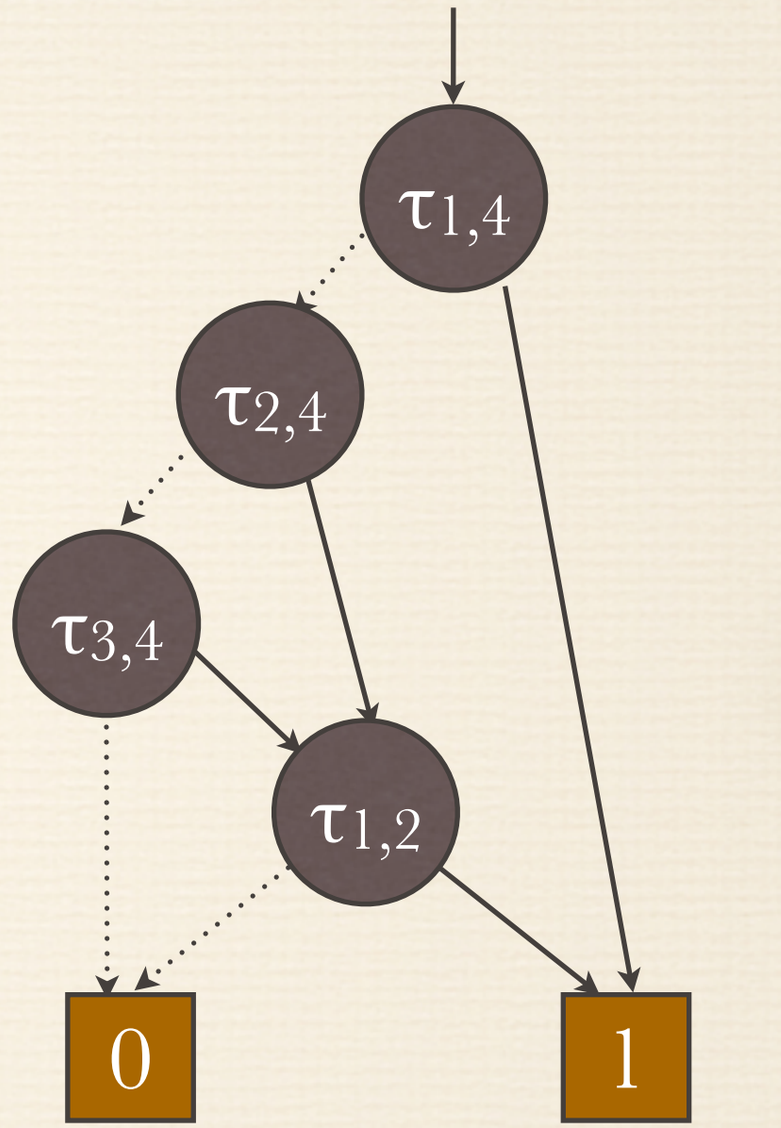
Example:

- input: $n = 4$, $\sigma = (2\text{-}3\text{-}1)$
- output: {(1 2 3 4), (1 2 4 3), (1 3 2 4), (1 4 2 3),
  (1 4 3 2), (2 1 3 4), (2 1 4 3), (3 1 2 4), (3 2 1 4),
  (4 1 2 3), (4 1 3 2), (4 2 1 3), (4 3 1 2), (4 3 2 1) }

πDD
(Permutation Decision Diagrams)

πDD
(permutation decision diagrams)

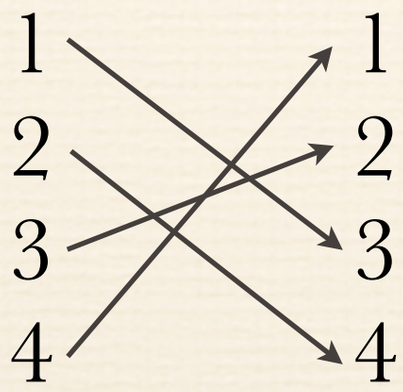
- **πDD**: decision diagrams which represent set of permutations
- one πDD correspond to one unique set of permutations
- based on decomposition of permutation by transposition



$\{(2\ 1\ 4\ 3), (4\ 1\ 3\ 2), (4\ 2\ 3\ 1)\}$
$= \{\tau_{1,2} \circ \tau_{3,4},\ \tau_{1,2} \circ \tau_{2,4},\ \tau_{1,4}\}$

Decomposition of permutation

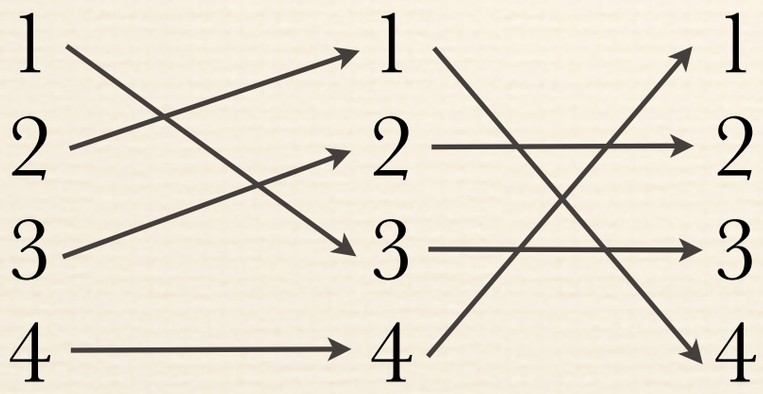
- Transposition $\tau_{x,y}$: exchange only two elements x and y.
- Any n-permutation π can be uniquely decomposed into at most n-1 transpositions.
  - Until π is not in ascending order, repeat the followings:
  - Let x be maximum element i such that $\pi_i \neq i$.
  - Exchange x and π_x, and add $\tau_{x,\pi x}$ to its decomposition.



$= (3\ 4\ 2\ 1)$

Decomposition of permutation

- Transposition $\tau_{x,y}$: exchange only two elements x and y.
- Any n-permutation π can be uniquely decomposed into at most n-1 transpositions.
  - Until π is not in ascending order, repeat the followings:
  - Let x be maximum element i such that $\pi_i \neq i$.
  - Exchange x and π_x, and add $\tau_{x,\pi x}$ to its decomposition.

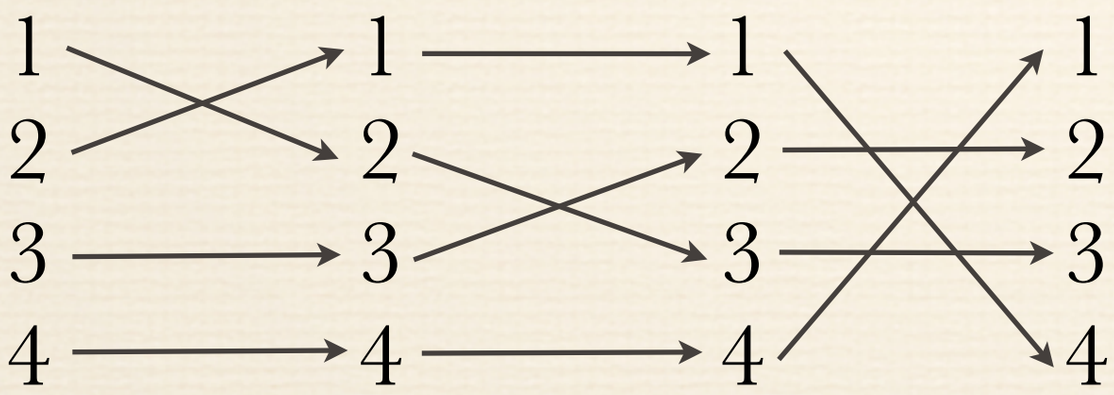


$= (3\ 1\ 2\ 4) \circ \tau_{1,4} = (3\ 4\ 2\ 1)$

"$\circ$" means composition of perm.

Decomposition of permutation

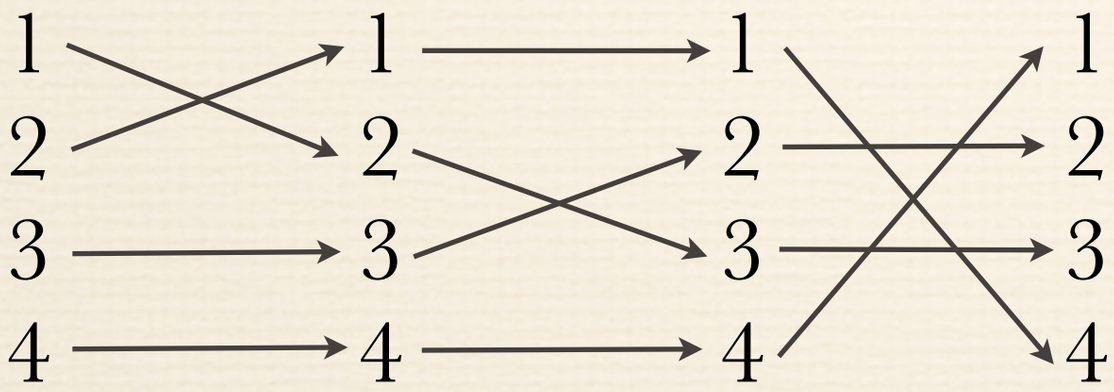
- Transposition $\tau_{x,y}$: exchange only two elements x and y.
- Any n-permutation π can be uniquely decomposed into at most n-1 transpositions.
  - Until π is not in ascending order, repeat the followings:
    - Let x be maximum element i such that $\pi_i \neq i$.
    - Exchange x and π_x, and add $\tau_{x,\pi x}$ to its decomposition.



$= (2\ 1\ 3\ 4) \circ \tau_{2,3} \circ \tau_{1,4} = (3\ 4\ 2\ 1)$

Decomposition of permutation

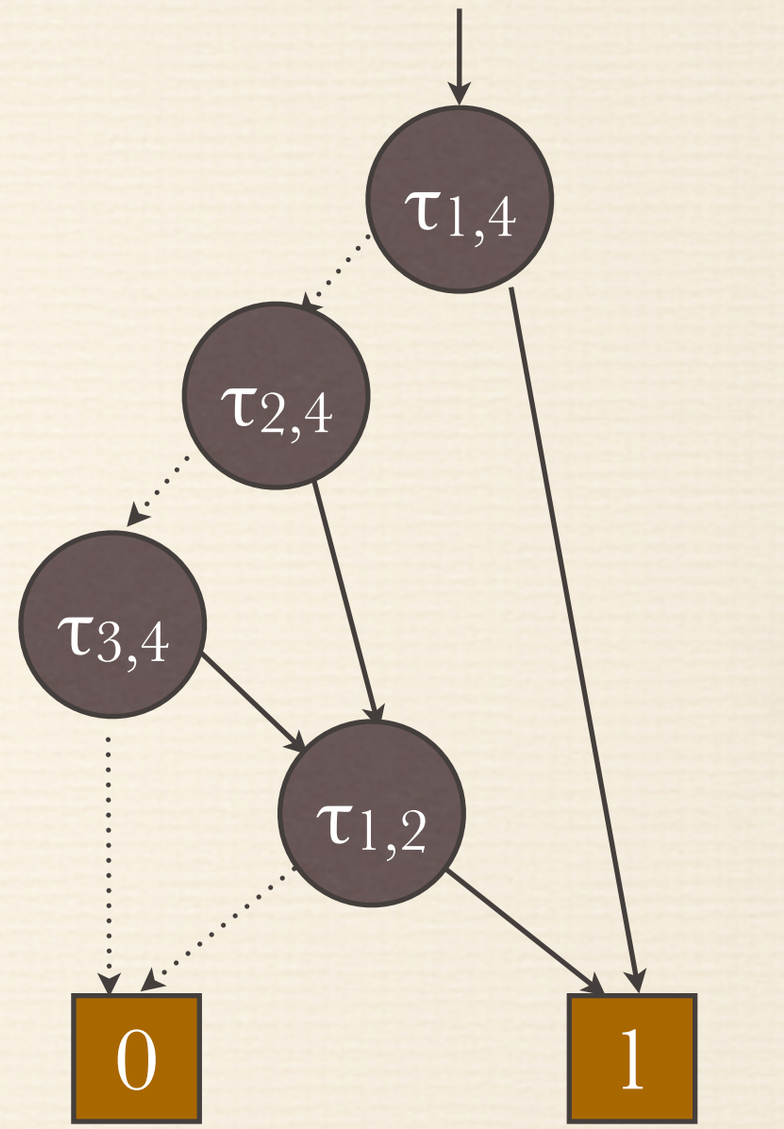
- Transposition $\tau_{x,y}$: exchange only two elements x and y.
- Any n-permutation π can be uniquely decomposed into at most n-1 transpositions.
  - Until π is not in ascending order, repeat the followings:
    - Let x be maximum element i such that $\pi_i \neq i$.
    - Exchange x and π_x, and add $\tau_{x,\pi x}$ to its decomposition.



$= \tau_{1,2} \circ \tau_{2,3} \circ \tau_{1,4} = (3\ 4\ 2\ 1)$

πDD

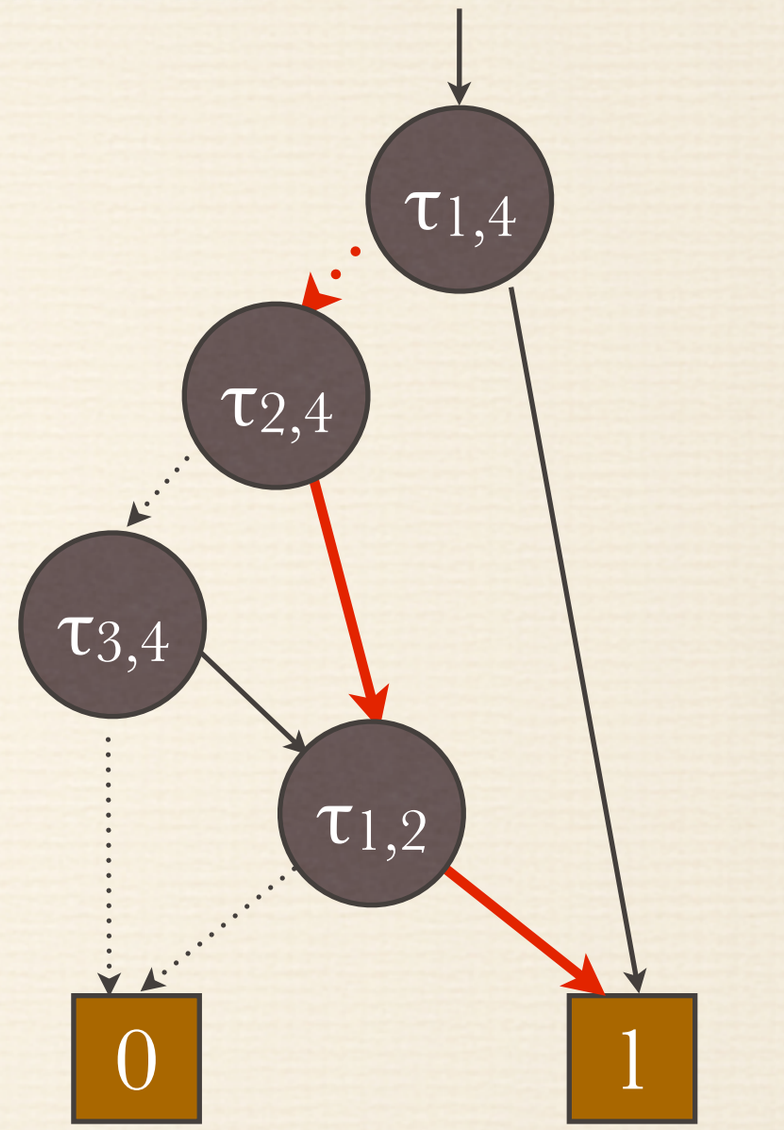
- πDD: reduced binary decision diagrams which represent the set of permutations
  - node: transposition
  - path to 1-terminal node: permutation
  - each nodes have only two edges:
    - 1-edge (means “use”)
    - 0-edge (means “ignore”).



$\{(2\ 1\ 4\ 3), (4\ 1\ 3\ 2), (4\ 2\ 3\ 1)\}$
$= \{\tau_{1,2} \circ \tau_{3,4},\ \tau_{1,2} \circ \tau_{2,4},\ \tau_{1,4}\}$

πDD

- πDD: reduced binary decision diagrams which represent the set of permutations
  - node: transposition
  - path to 1-terminal node: permutation
  - each nodes have only two edges:
    - 1-edge (means “include”)
    - 0-edge (means “exclude”).

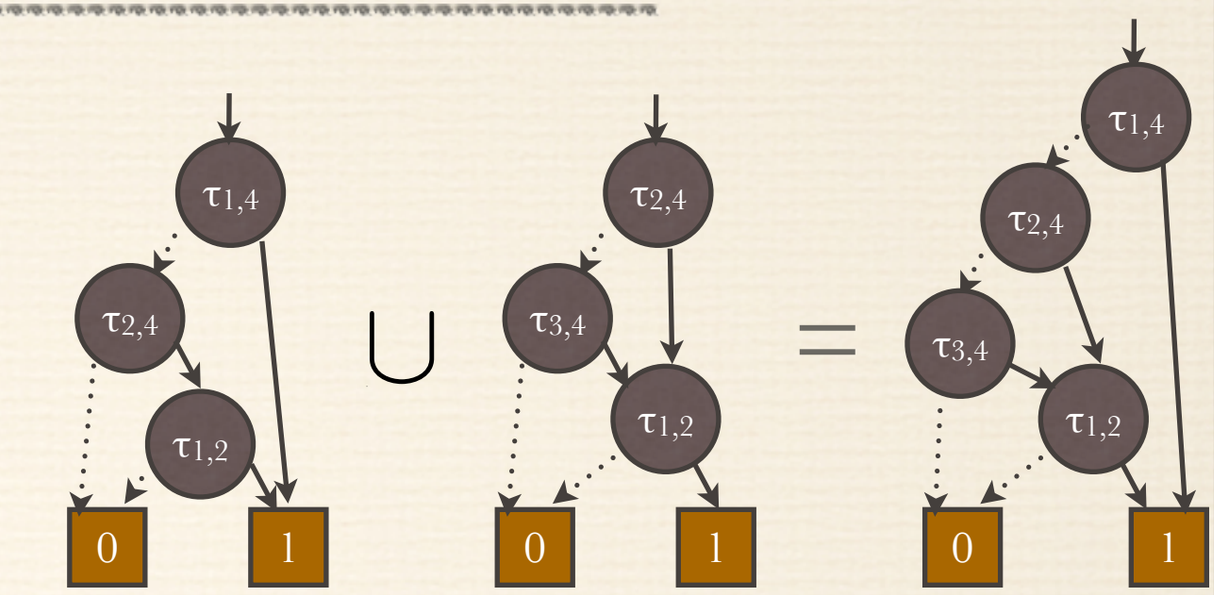


$\{(2\ 1\ 4\ 3), (4\ 1\ 3\ 2), (4\ 2\ 3\ 1)\}$

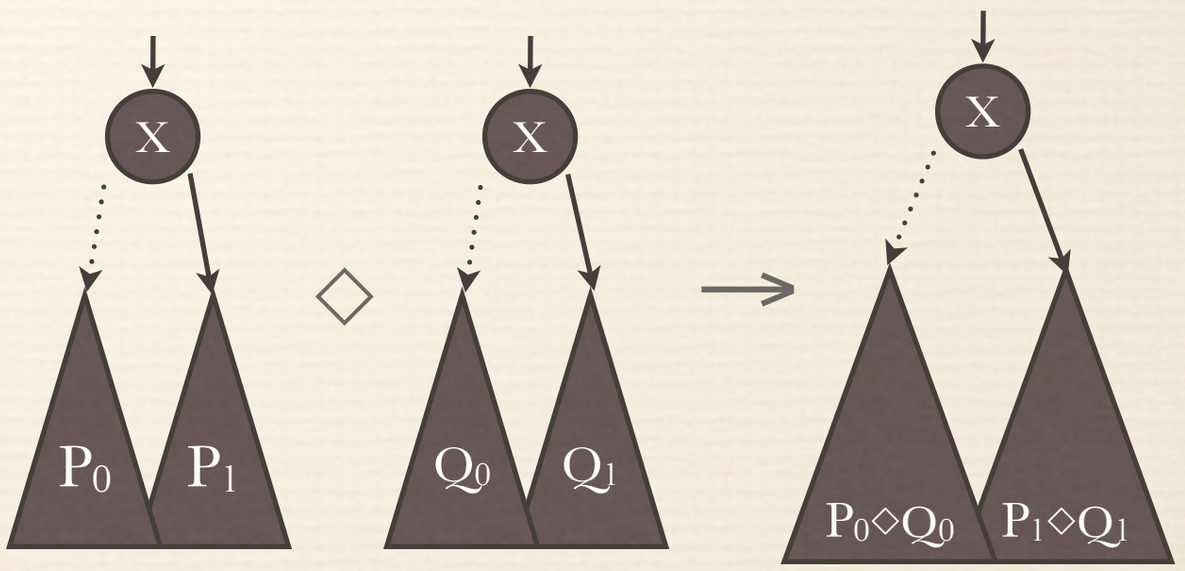
$= \{\tau_{1,2} \circ \tau_{3,4},\ \boldsymbol{\tau_{1,2} \circ \tau_{2,4}},\ \tau_{1,4}\}$

Binary Operation

- πDD has binary operation, such as union, intersection, difference and so on, without restoring.



- These computation time depend on the number of nodes, not permutations.
- Because they are based on recursive method.



Cartesian Product

- πDD also has cartesian product operation "×":
  - $P \times Q = \{p \circ q \mid p \in P, q \in Q\}$
- Example:
  - $\{ (4\ 3\ 2\ 1), (2\ 3\ 4\ 1) \} \times \{ (4\ 3\ 1\ 2), (4\ 2\ 1\ 3) \}$

$= \{ (4\ 3\ 2\ 1) \circ (4\ 3\ 1\ 2), (4\ 3\ 2\ 1) \circ (4\ 2\ 1\ 3),$
$(2\ 3\ 4\ 1) \circ (4\ 3\ 1\ 2), (2\ 3\ 4\ 1) \circ (4\ 2\ 1\ 3) \}$

Cartesian Product

- πDD also has cartesian product operation "×":
  - $P \times Q = \{p \circ q \mid p \in P, q \in Q\}$
- Example:
  - $\{ (4\ 3\ 2\ 1), (2\ 3\ 4\ 1) \} \times \{ (4\ 3\ 1\ 2), (4\ 2\ 1\ 3) \}$

$= \{ (4\ 3\ 2\ 1) \circ (4\ 3\ 1\ 2), (4\ 3\ 2\ 1) \circ (4\ 2\ 1\ 3),$
$(2\ 3\ 4\ 1) \circ (4\ 3\ 1\ 2), (2\ 3\ 4\ 1) \circ (4\ 2\ 1\ 3) \}$

Cartesian Product

- πDD also has cartesian product operation "×":
  - $P \times Q = \{p \circ q \mid p \in P, q \in Q\}$
- Example:
  - $\{ (4\ 3\ 2\ 1), (2\ 3\ 4\ 1) \} \times \{ (4\ 3\ 1\ 2), (4\ 2\ 1\ 3) \}$

$= \{ (4\ 3\ 2\ 1) \circ (4\ 3\ 1\ 2), (4\ 3\ 2\ 1) \circ (4\ 2\ 1\ 3),$
$(2\ 3\ 4\ 1) \circ (4\ 3\ 1\ 2), (2\ 3\ 4\ 1) \circ (4\ 2\ 1\ 3) \}$

Cartesian Product

- πDD also has cartesian product operation "×":
  - $P \times Q = \{p \circ q \mid p \in P, q \in Q\}$
- Example:
  - $\{ (4\ 3\ 2\ 1), (2\ 3\ 4\ 1) \} \times \{ (4\ 3\ 1\ 2), (4\ 2\ 1\ 3) \}$

$= \{ (4\ 3\ 2\ 1) \circ (4\ 3\ 1\ 2), (4\ 3\ 2\ 1) \circ (4\ 2\ 1\ 3),$
$(2\ 3\ 4\ 1) \circ (4\ 3\ 1\ 2), (2\ 3\ 4\ 1) \circ (4\ 2\ 1\ 3) \}$

Cartesian Product

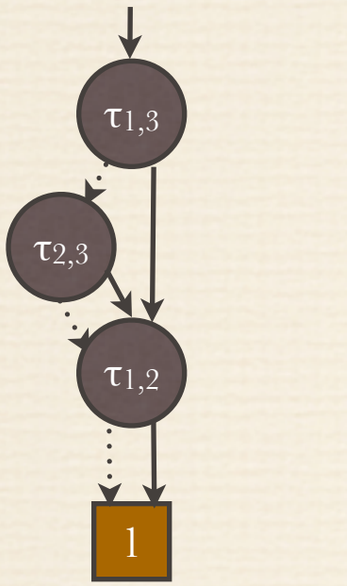
- πDD also has cartesian product operation "×":
  - $P \times Q = \{p \circ q \mid p \in P, q \in Q\}$
- Example:
  - $\{ (4\ 3\ 2\ 1), (2\ 3\ 4\ 1) \} \times \{ (4\ 3\ 1\ 2), (4\ 2\ 1\ 3) \}$

$= \{ (4\ 3\ 2\ 1) \circ (4\ 3\ 1\ 2), (4\ 3\ 2\ 1) \circ (4\ 2\ 1\ 3),$
$(2\ 3\ 4\ 1) \circ (4\ 3\ 1\ 2), (2\ 3\ 4\ 1) \circ (4\ 2\ 1\ 3) \}$

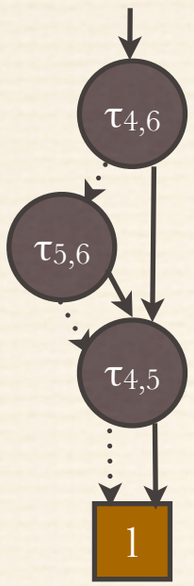
$= \{ (2\ 1\ 3\ 4), (3\ 1\ 2\ 4), (3\ 1\ 2\ 4), (2\ 1\ 3\ 4) \}$

$= \{ (2\ 1\ 3\ 4), (3\ 1\ 2\ 4), (2\ 1\ 3\ 4) \}$

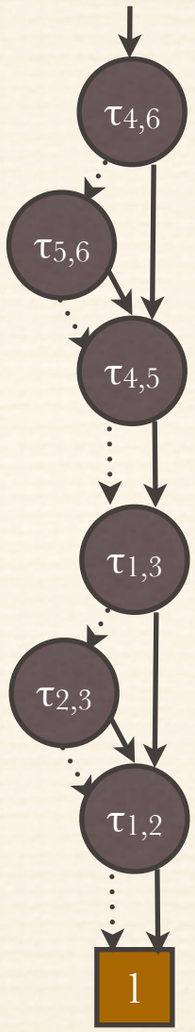
Cartesian Product



×



=



{(1 2 3), (1 3 2),
(2 1 3),(2 3 1),
(3 1 2), (3 2 1)}

{(1 2 3 4 5 6), (1 2 3 4 6 5),
(1 2 3 5 4 6), (1 2 3 5 6 4),
(1 2 3 6 4 5), (1 2 3 6 5 4)}

{(1 2 3 4 5 6), (1 2 3 4 6 5), (1 2 3 5 4 6),
(1 2 3 5 6 4), (1 2 3 6 4 5), (1 2 3 6 5 4),
(1 3 2 4 5 6), (1 3 2 4 6 5), (1 3 2 5 4 6),
(1 3 2 5 6 4), (1 3 2 6 4 5), (1 3 2 6 5 4),
(2 1 3 4 5 6), (2 1 3 4 6 5), (2 1 3 5 4 6),
(2 1 3 5 6 4), (2 1 3 6 4 5), (2 1 3 6 5 4),
(2 3 1 4 5 6), (2 3 1 4 6 5), (2 3 1 5 4 6),
(2 3 1 5 6 4), (2 3 1 6 4 5), (2 3 1 6 5 4),
(3 1 2 4 5 6), (3 1 2 4 6 5), (3 1 2 5 4 6),
(3 1 2 5 6 4), (3 1 2 6 4 5), (3 1 2 6 5 4),
(3 2 1 4 5 6), (3 2 1 4 6 5), (3 2 1 5 4 6),
(3 2 1 5 6 4), (3 2 1 6 4 5), (3 2 1 6 5 4) }

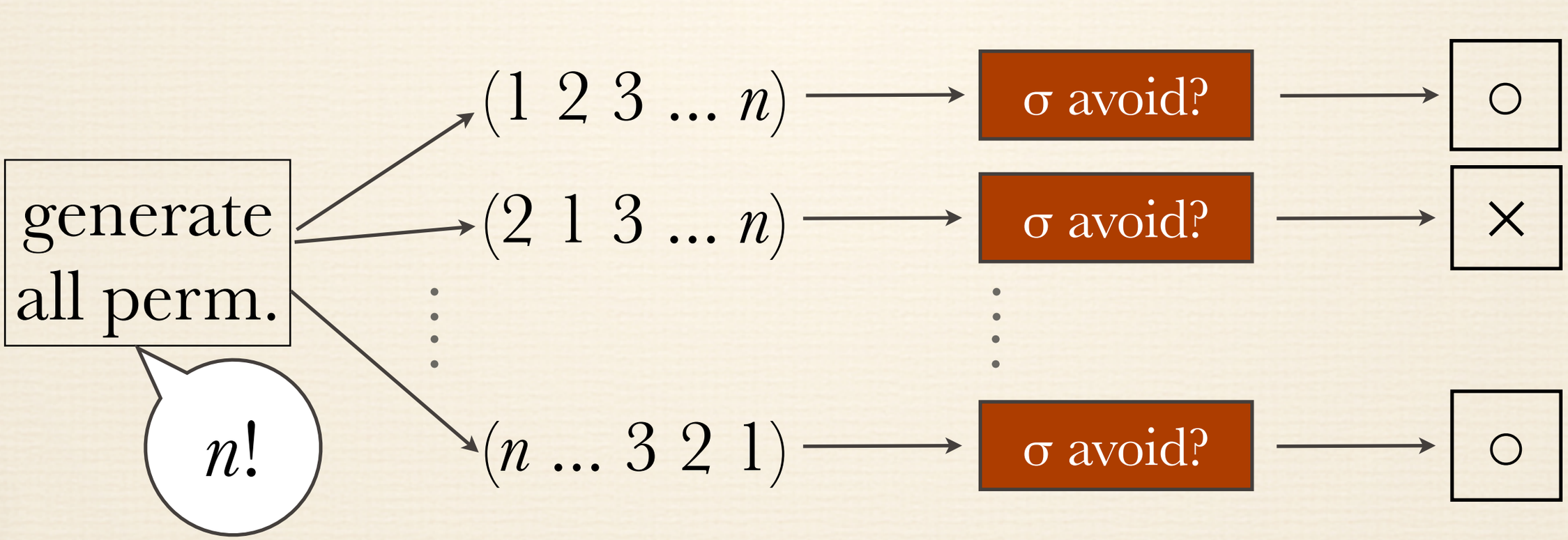
cardinality: Product
# of node: Sum

Our Algorithm

Naive Method

General generating approach:

- generate all n-permutations.
- check the avoidance for each permutations.



Basic idea

Our approach: use πDD's operations.

1. Obtain πDD for S_n, which is the set of n-permutations.
2. Obtain πDD for $C_n(\sigma)$, which is the set of n-permutations containing σ .
3. Calculate the set difference $S_n \backslash C_n(\sigma)$, i.e, obtain πDD for $A_n(\sigma)$ (= the set of n-permutations avoiding σ.)

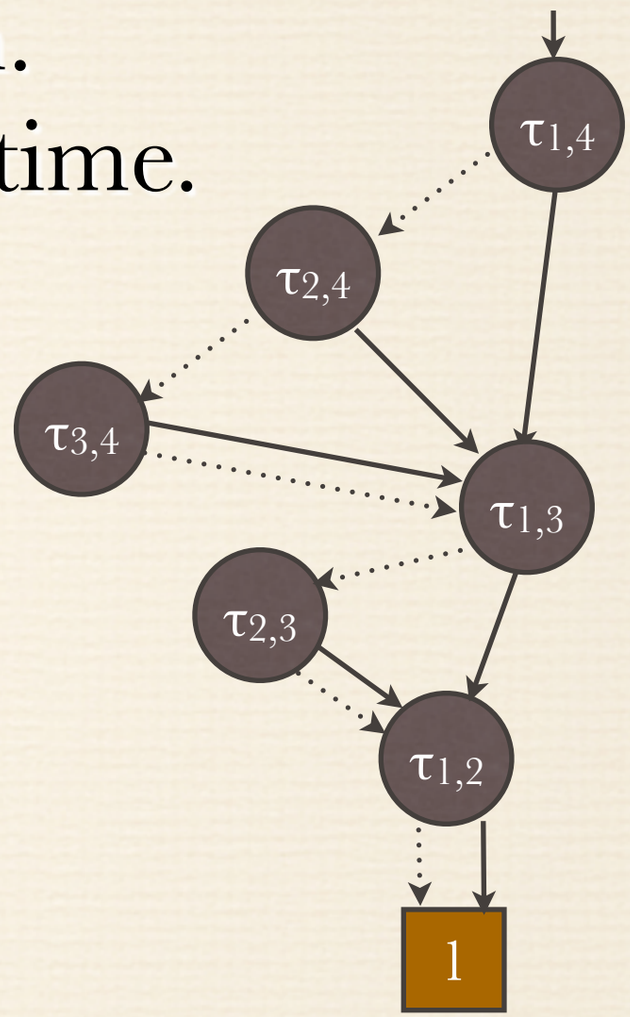
Basic idea

Our approach: use πDD's operations.

1. Obtain πDD for S_n, which is the set of n-permutations.
2. Obtain πDD for $C_n(\sigma)$, which is the set of n-permutations containing σ .
3. Calculate the set difference $S_n \backslash C_n(\sigma)$, i.e, obtain πDD for $A_n(\sigma)$ (= the set of n-permutations avoiding σ.)

Generate S_n

- It is very compact and easy to obtain.
  Direct construction takes only $O(n^2)$ time.



{(1 2 3 4), (1 2 4 3), (1 3 2 4), (1 3 4 2),
(1 4 2 3), (1 4 3 2), (2 1 3 4), (2 1 4 3),
(2 3 1 4), (2 3 4 1), (2 4 1 3), (2 4 3 1),
(3 1 2 4), (3 1 4 2), (3 2 1 4), (3 2 4 1),
(3 4 1 2), (3 4 2 1), (4 1 2 3), (4 1 3 2),
(4 2 1 3), (4 2 3 1), (4 3 1 2), (4 3 2 1) }

$n!$

$$\frac{n(n-1)}{2}+1$$

Basic idea

Our approach: use πDD's operations.

1. Obtain πDD for S_n, which is the set of n-permutations.
2. Obtain πDD for $C_n(\sigma)$, which is the set of n-permutations containing σ .
3. Calculate the set difference $S_n \backslash C_n(\sigma)$, i.e, obtain πDD for $A_n(\sigma)$ (= the set of length n-permutations avoiding σ.)

Generate $C_n(\sigma)$ - part.A

- Let k be length of σ.
- Generate all permutations whose k-prefix is in ascending order.
- Example: $n = 5$, $\sigma = (2\text{-}3\text{-}1)$, $k = 3$

  { (1 2 3 4 5), (1 2 4 3 5), ... , (2 4 5 1 3), ... , (3 4 5 1 2),
  (1 2 3 5 4), (1 2 4 5 3), ... , (2 4 5 3 1), ... , (3 4 5 2 1) }

Generate $C_n(\sigma)$ - part.A

- Let k be length of σ.
- Generate all permutations whose k-prefix is in ascending order.
- Example: $n = 5$, $\sigma = (2\text{-}3\text{-}1)$, $k = 3$

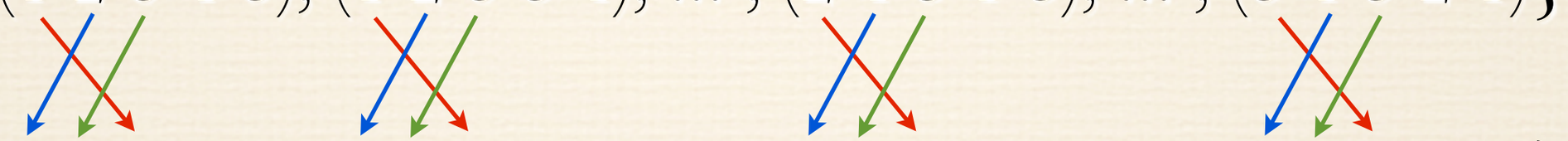
{ (1 2 3 4 5), (1 2 4 3 5), ... , (2 4 5 1 3), ... , (3 4 5 1 2),
(1 2 3 5 4), (1 2 4 5 3), ... , (2 4 5 3 1), ... , (3 4 5 2 1) }

all $\binom{n}{k}$

Generate $C_n(\sigma)$ - part.B

- Rearrange each k-prefixes into the order which is order isomorphic to σ.
- Example: $n = 5$, $\sigma = (2\text{-}3\text{-}1)$, $k = 3$

$\{ (1\ 2\ 3\ 4\ 5), (1\ 2\ 3\ 5\ 4), \dots , (2\ 4\ 5\ 1\ 3), \dots , (3\ 4\ 5\ 2\ 1)\}$



$\{ (2\ 3\ 1\ 4\ 5), (2\ 3\ 1\ 5\ 4), \dots , (4\ 5\ 2\ 1\ 3), \dots , (4\ 5\ 3\ 2\ 1)\}$

Generate $C_n(\sigma)$ - part.C

- Rearrange each k-prefixes into all possible $\binom{n}{k}$ positions.

Example: $n = 5$, $\sigma = (2\text{-}3\text{-}1)$, $k = 3$

{ (2 3 1 4 5), (2 3 1 5 4), ... , (4 5 3 2 1)}

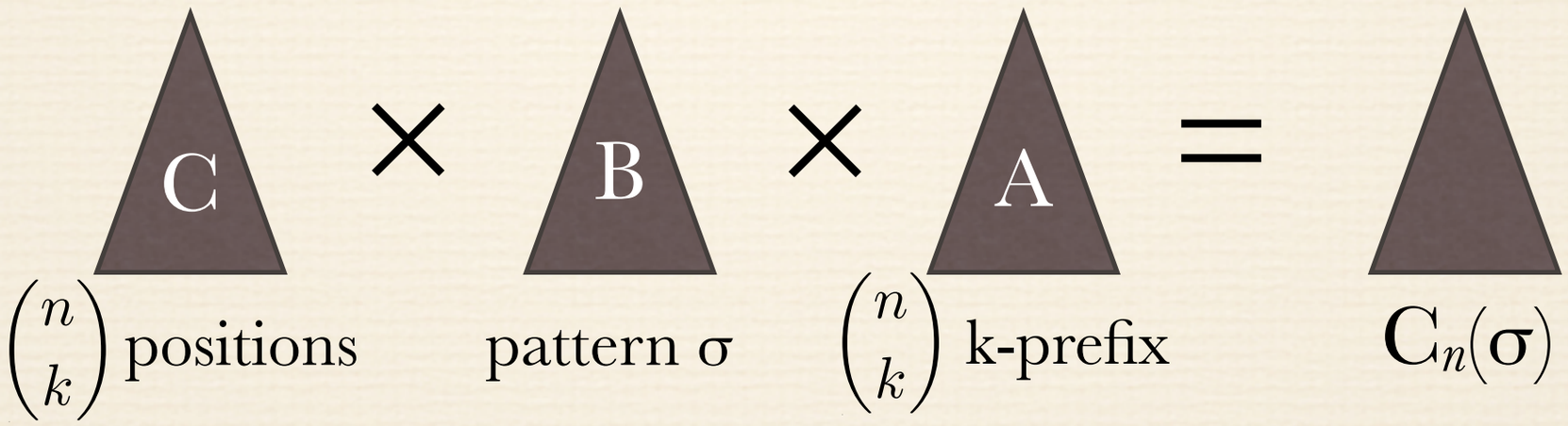
{ (2 3 1 4 5), (2 3 4 1 5), ... , (2 4 5 3 1), ... , (4 5 2 3 1),
(2 3 1 5 4), (2 3 5 1 4), ... , (2 5 4 3 1), ... , (5 4 2 3 1),

...

(4 5 3 2 1), (4 5 2 3 1), ... , (4 2 1 5 3), ... , (2 1 4 5 3) }

Generate $C_n(\sigma)$

- All 3 parts of generating $C_n(\sigma)$ are "rearrangement".
- Rearrangement is realized by permutation composition.
  - Example: $(4\ 3\ 2\ 1)\circ(2\ 4\ 1\ 3) = (3\ 1\ 4\ 2)$
- πDD's cartesian product can rearrange permutations into multiple order at once.



Basic idea

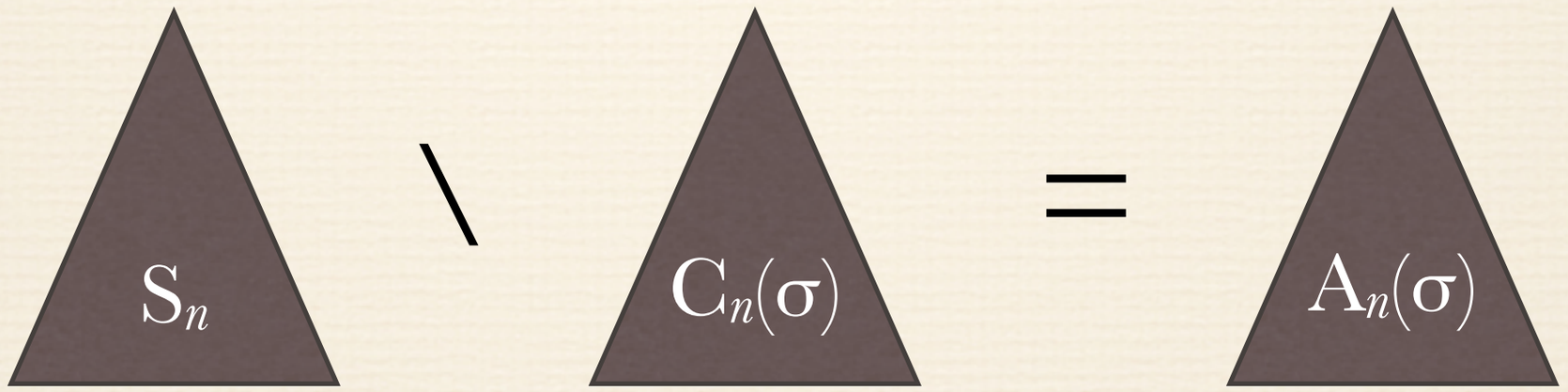
Our approach: use πDD's operations.

1. Obtain πDD for S_n, which is the set of n-permutations.
2. Obtain πDD for $C_n(\sigma)$, which is the set of n-permutations containing σ .
3. Calculate the set difference $S_n \backslash C_n(\sigma)$, i.e, obtain πDD for $A_n(\sigma)$ (= the set of n-permutations avoiding σ.)

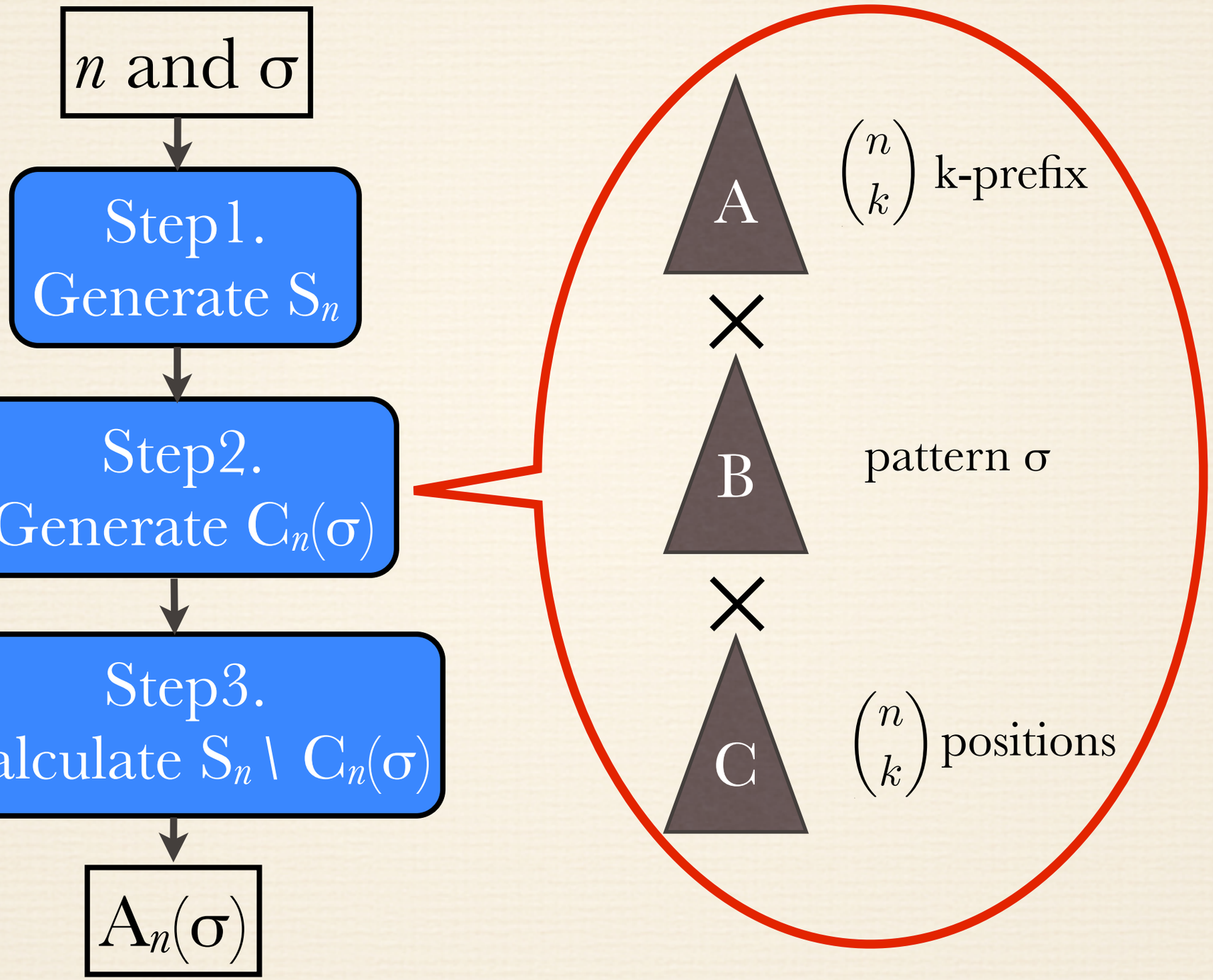
Basic idea

Our approach: use πDD's operations.

1. Obtain πDD for S_n, which is the set of n-permutations.
2. Obtain πDD for $C_n(\sigma)$, which is the set of n-permutations containing σ .
3. Calculate the set difference $S_n \setminus C_n(\sigma)$, i.e, obtain πDD for $A_n(\sigma)$ (= the set of n-permutations avoiding σ.)



Algorithm summary



Extended algorithm

Vincular pattern

- Our algorithm is easy to be extended for vincular pattern avoiding permutation.
- Vincular pattern is pattern whose some element must be adjacent.
- If σ_i and σ_{i+1} must be adjacent, we omit "-".
  - Example: $\sigma = (2\ 3\text{-}1)$
    - $(3\ 4\ 1\ 2)$ contains σ, but $(3\ 1\ 4\ 2)$ avoids σ.

Generating Vincular pattern avoiding perm.

- In the same way as classical pattern.
- But step2-C change a little bit.
- Example: $n = 5$, $\sigma = (2\ 3\text{-}1)$

{ (2 3 1 4 5), (2 3 1 5 4), ... , (4 5 3 2 1)}

{ (2 3 1 4 5), (2 3 4 1 5), ... , (2 4 5 3 1), ... , (4 5 2 3 1),
(2 3 1 5 4), (2 3 5 1 4), ... , (2 5 4 3 1), ... , (5 4 2 3 1),
...
(4 5 3 2 1), (4 5 2 3 1), ... , (4 2 1 5 3), ... , (2 1 4 5 3) }

not adjacent...

Generating Vincular pattern avoid perm.

- In the same way as classical pattern.
- But step2-C change a little bit.
- Example: $n = 5$, $\sigma = (2\ 3\text{-}1)$

{ (2 3 1 4 5), (2 3 1 5 4), ... , (4 5 3 2 1)}

{ (2 3 1 4 5), (2 3 4 1 5), ... , (4 5 2 3 1),
(2 3 1 5 4), (2 3 5 1 4), ... , (5 4 2 3 1),
...
(4 5 3 2 1), (4 5 2 3 1), ... , (2 1 4 5 3) }

Experimental Results

Experiment for classical pattern

- We carried out computational experiment for classical pattern avoiding.

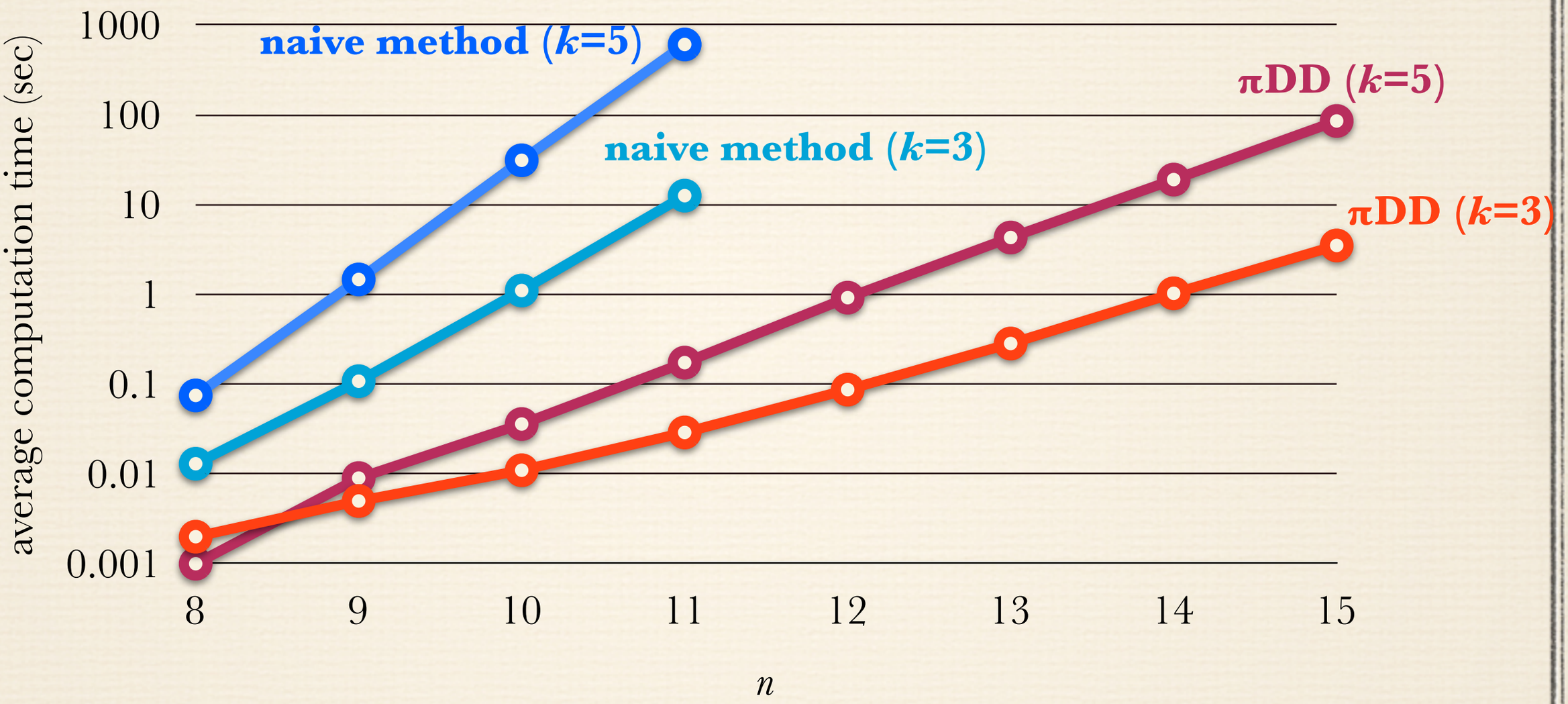
Comparison with naive method:

> Generate all n-permutations and check to avoid the given classical pattern or not for each.

Experiment:

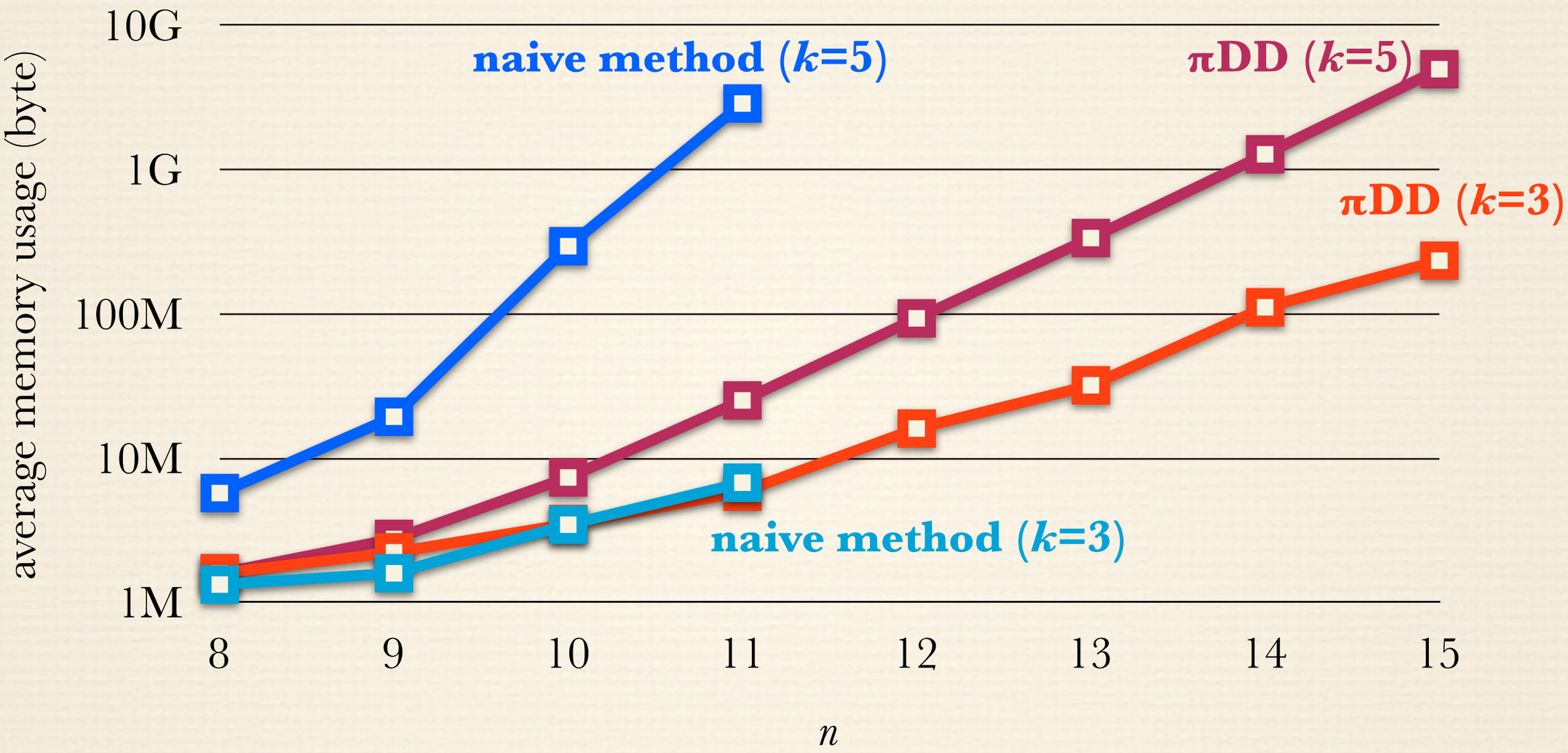
> Generate $A_n(\sigma)$, where σ is all patterns whose length is k for each k.

Computation time comparison



- πDD is faster than naive method.

Memory usage comparison



- Note that naive method store $A_n(\sigma)$ in arrays.
- πDD use about 1% memory of arrays when n=11 and k=5.

Experiment of vincular pattern

- We also carried out computational experiments for vincular pattens.

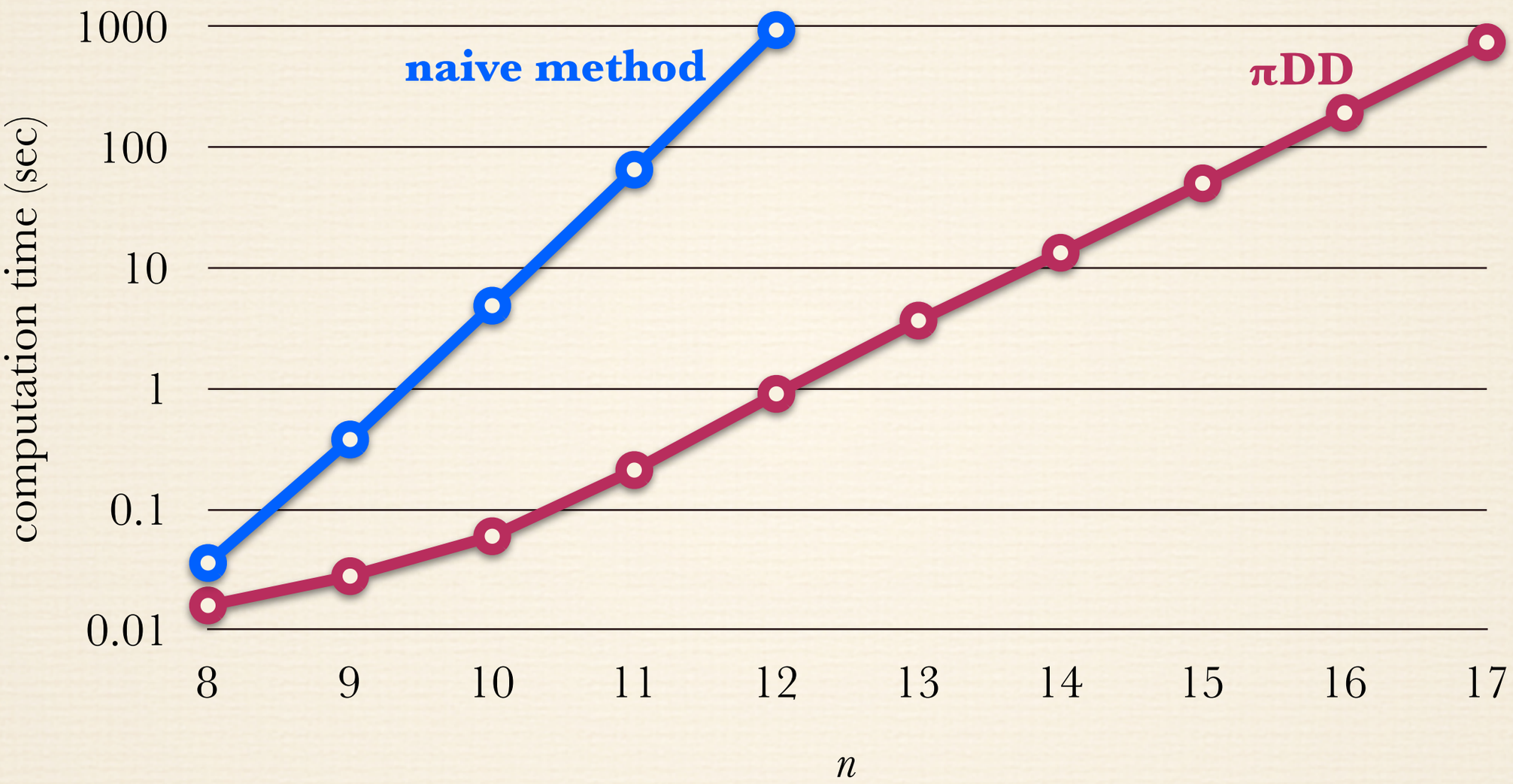
Comparison with naive method:

> Generate all n-permutations and check to avoid the given vincular pattern or not for each.

Experiment:

> Generate Baxter permutations, which avoid the two vincular patterns, (3-1 4-2) and (2-4 1-3).

Computation time comparison



- πDD method is also fast for vincular pattern.

Memory usage comparison

n	# of Baxter	πDD(KB)	Byte/# of Bax	naive(KB)	πDD/naive
8	10754	2760	256.65	2752	1.00
9	58202	4164	71.54	5676	0.73
10	326240	12984	39.80	37864	0.34
11	1882960	26828	14.25	181112	0.15
12	11140560	98924	8.88	1442904	0.07
13	67329992	383272	5.69	---	---
14	414499438	824732	1.99	---	---
15	2593341586	3151164	1.22	---	---
16	16458756586	12403488	0.75	---	---
17	105791986682	40788188	0.39	---	---

- The larger n is, the higher compression ratio πDD has.

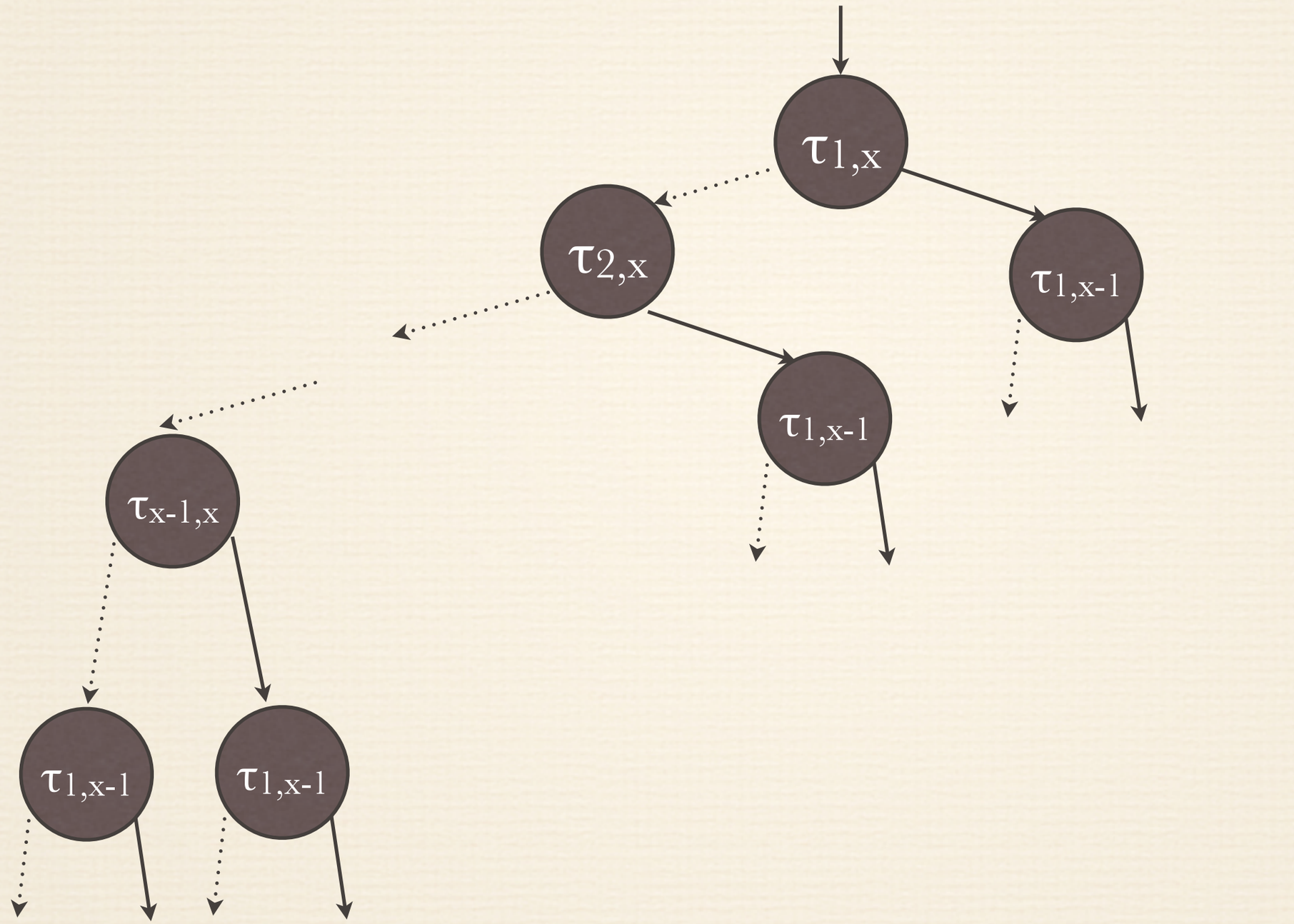
Conclusion

Conclusion

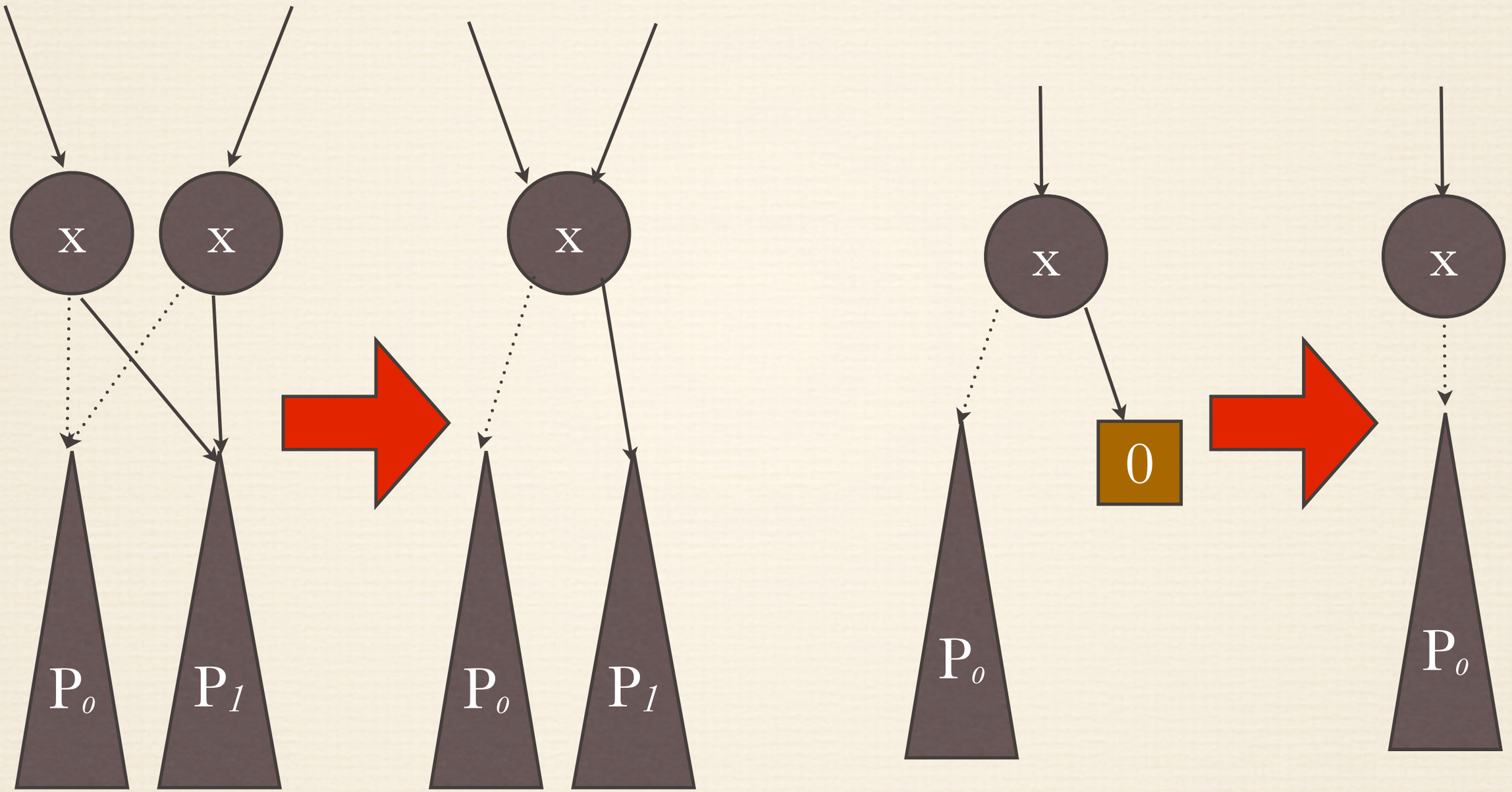
- We propose algorithms that generate classical and vincular pattern avoiding permutations using πDD.
  - compact representation
  - brand-new and simple approach
- Experimental result looks good.
  - In almost cases, our algorithms are fast and use low memory to compare naive method.

Thank you!

Variable order

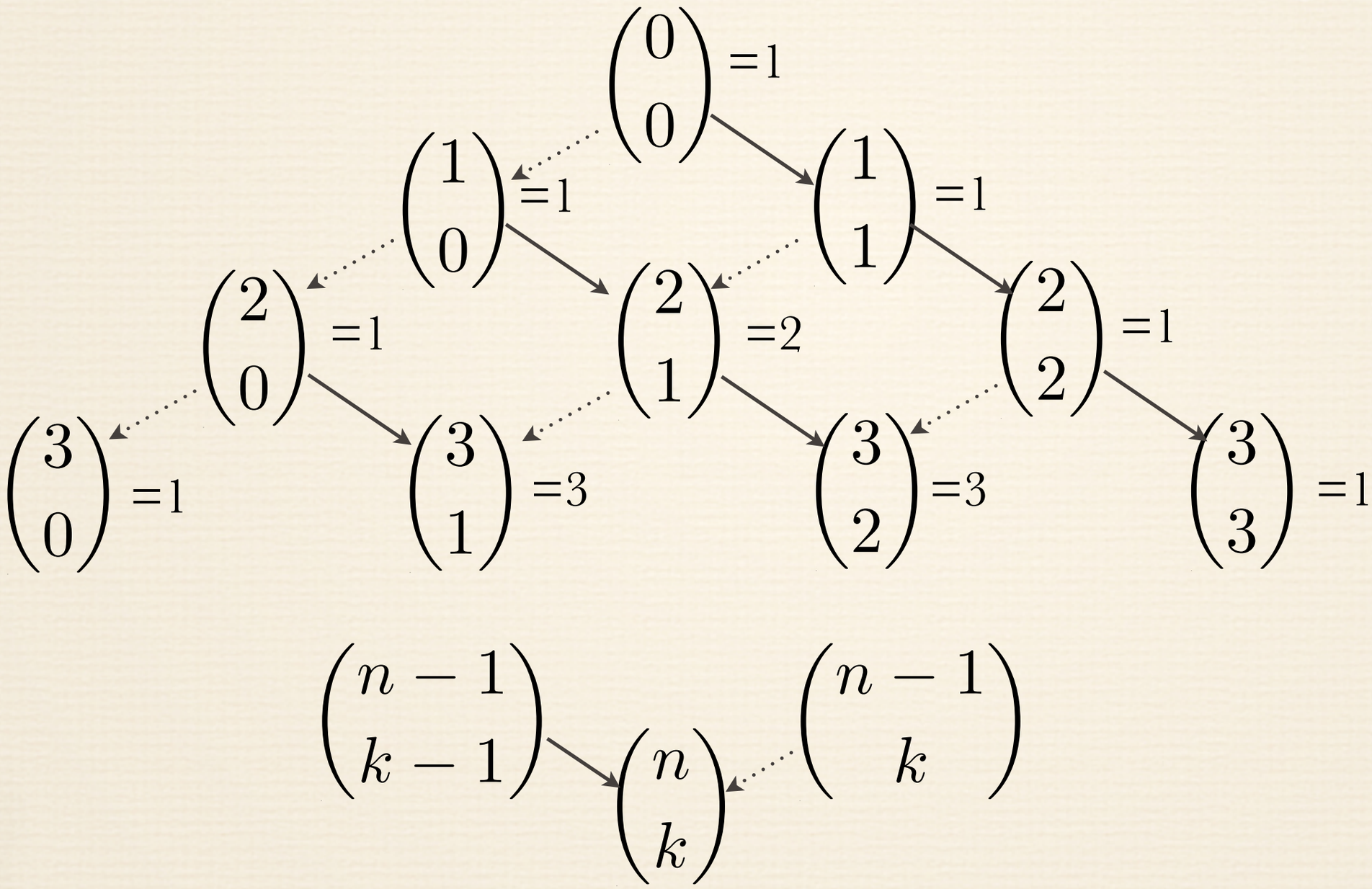


Compression Rules



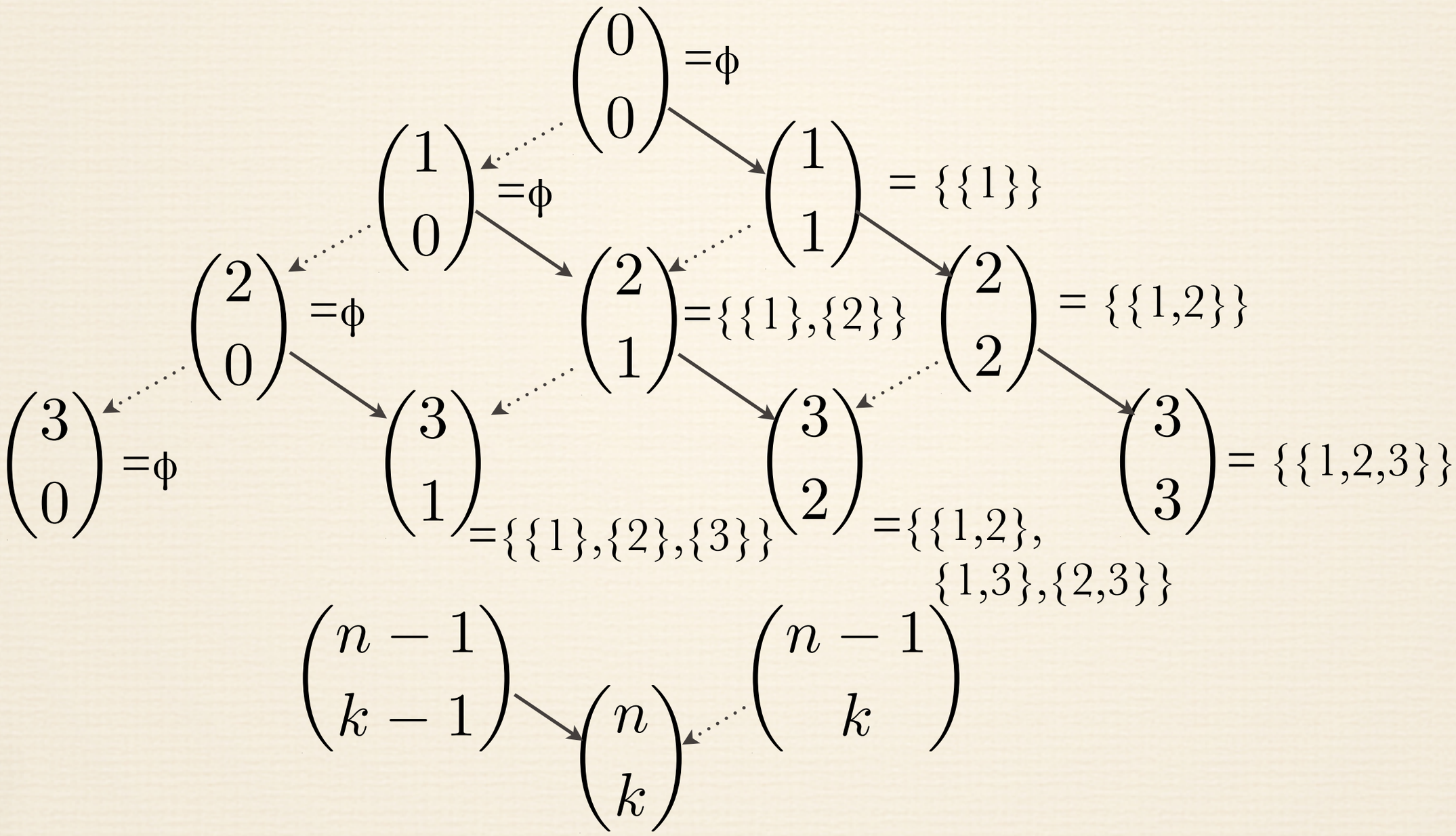
Generate Combination

- The idea is based on Pascal's Triangle



Generate Combination

- The idea is based on Pascal's Triangle



Generate Combination

- The idea is based on Pascal's Triangle