FULLY HOMOMORPHIC ENCRYPTION WITH POLYLOG OVERHEAD

Craig Gentry and Shai Halevi
IBM Watson

Nigel Smart
Univ. Of Bristol

Homomorphic Encryption

- Usual procedures (KeyGen,Enc,Dec)
  - Say, encrypting bits
- Usual semantic-security requirement
  - $(pk, Enc_{pk}(0)) \sim (pk, Enc_{pk}(1))$
- Additional Eval procedure
  - Evaluate arithmetic circuits on ciphertexts
  - Result decrypted to the evaluation of the same circuit on the underlying plaintext bits
  - Ciphertext does not grow with circuit complexity
- This work: asymptotically efficient Eval

Contemporary HE Schemes

- The [Gentry'09] approach
  - Ciphertext is noisy (to get security)
  - Noise grows with homomorphic evaluation
  - Until ciphertext is too noisy to decrypt
- Ciphertext is inherently large
  - Need to leave lots of room for noise to grow
  - It takes $\Omega(\lambda)$-bit ciphertext to encrypt a single bit
    - λ is the security parameter
- Implementing each binary arithmetic gate takes at least $\Omega(\lambda)$ time
  - $\Omega(\lambda)$ time just to read the input ciphertexts

Our Result

- Homomorphic evaluation of T-gate binary arithmetic circuits of average width $\Omega(\lambda)$ in time **$T \cdot \mathrm{polylog}(\lambda)$**
- More Generally, a T-gate, W-average-width circuit can be evaluated homomorphically in time $O(\lceil W/\lambda \rceil \cdot \lambda \cdot T/W)$

time per level # of levels

Our Approach

- Use HE over polynomial rings
- Pack an array of bits in each ciphertext
- Use ring-automorphisms to move bits around in the arrays
- Efficient data-movement schemes
  - Using Beneš/Waksman networks and extensions

BACKGROUND

- Homomorphic Encryption over Polynomials Rings
- Polynomial-CRT representation, plaintext slots
- Homomorphic SIMD operations

Hom.Enc. Over Polynomial Rings

- Used, e.g., in [BGV'12], [LTV'12], [B'12]
- Native plaintext space is $R\downarrow 2 = Z\downarrow 2\ [X]/\Phi\downarrow m\ (X)$
  - Binary polynomials modulo $\Phi\downarrow m\ (X)$
  - $\Phi\downarrow m\ (X)$ is m'th cyclotomic polynomial, deg=$\phi(m)$
  - For our purposes m is an odd number
- Ciphertexts, secret-keys are vectors over $R\downarrow q = Z\downarrow q\ [X]/\Phi\downarrow m\ (X)$ (for some odd q)
- Decryption formula is $pt = [[ct \bullet sk]\downarrow q\ ]\downarrow 2$

Plaintext Algebra (1)

- $\Phi\downarrow m\,(X)$ irreducible over Z, but not mod 2
  - $\Phi\downarrow m\,(X) \equiv \prod\downarrow j=1\uparrow \ell\ F\downarrow j\,(X)$ (mod 2)
  - F_j's are irreducible, all have the same degree d
    - degree d is the order of 2 in $Z\downarrow m\uparrow *$
  - For some m's we can get $\ell = \phi(m)/d = \Omega(\mathrm{m}/\log \mathrm{m})$
- $GF(2\uparrow d)$ has a primitive m-th root of unity ζ
- $\Phi\downarrow m\,(X)$ splits over $GF(2\uparrow d)$ into linear terms, $\Phi\downarrow m\,(X) \equiv \prod\downarrow j\in Z\downarrow m\uparrow * \uparrow (X-\zeta\uparrow j)$ (mod 2)

Plaintext Algebra (2)

- The $F_j(X)$'s are products of conjugates
  - For each $F_j(X)$ there is $t_j \in Z_m^*$ such that $F_j(X) = (X - \zeta^{t_j})(X - \zeta^{2t_j})(X - \zeta^{4t_j}) \cdots (X - \zeta^{2^{d-1} t_j})$
- $T = \{t_1, \ldots, t_\ell\}$ represents the quotient group $Z_m^* / \langle 2 \rangle$

E.g., Z_{63}^* =

32T:	32	34	44	31	29	19
16T:	16	17	22	47	46	41
8T:	8	40	11	55	23	52
4T:	4	20	37	59	43	26
2T:	2	10	50	61	53	13
T:	**1**	**5**	**25**	**62**	**58**	**38**

Plaintext Slots

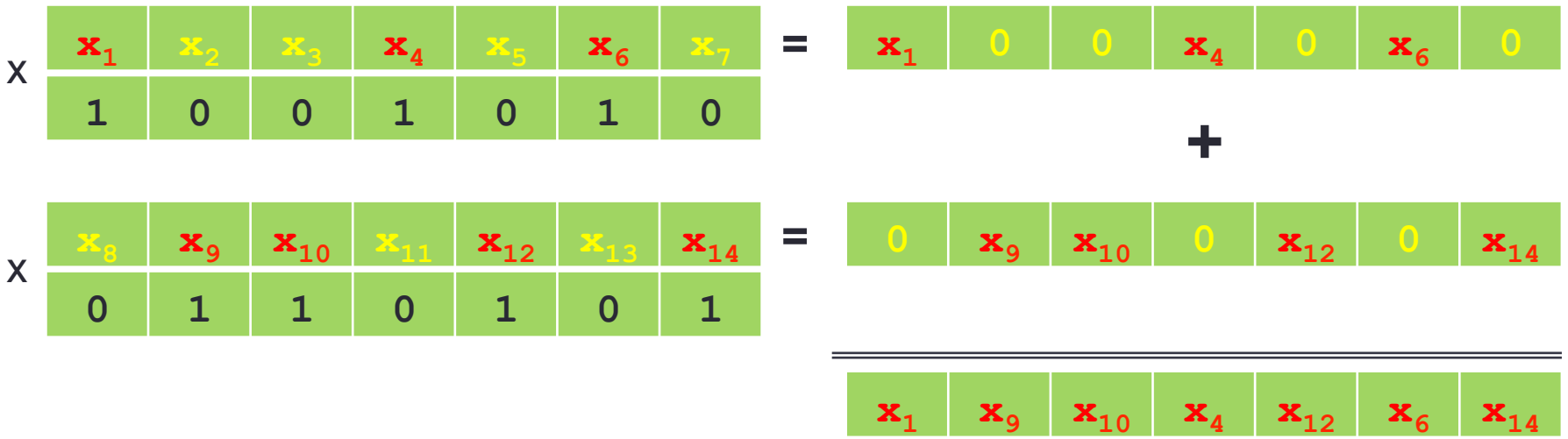
- Plaintext polynomial $a \in R\downarrow 2$ encodes ℓ values
  - $a \cong [\alpha\downarrow 1, ..., \alpha\downarrow \ell], \ \alpha\downarrow j = a(\zeta\uparrow t\downarrow j)$
    - Evaluation of a in roots of $\Phi\downarrow m(X)$
  - 1-1 mapping $a \in R\downarrow 2 \leftrightarrow [\alpha\downarrow 1, ..., \alpha\downarrow \ell] \in GF(2\uparrow d)\uparrow \ell$
- When working with plaintext bits, use a's for which each $\alpha\downarrow j$ is a bit
- Ops +,× work independently on the slots
  - ℓ-ADD: $a + a' \cong [\alpha\downarrow 1 + \alpha\downarrow 1', ..., \alpha\downarrow \ell + \alpha\downarrow \ell']$
  - ℓ-MUL: $a \times a' \cong [\alpha\downarrow 1 \times \alpha\downarrow 1', ..., \alpha\downarrow \ell \times \alpha\downarrow \ell']$

Homomorphic SIMD [SV'11]

SIMD = **S**ingle **I**nstruction **M**ultiple **D**ata

- Computing the same function on ℓ inputs at the price of one computation
- Pack the inputs into the slots
  - Bit-slice, inputs to *j*'th instance go in *j*'th slots
- Compute the function once
- After decryption, decode the ℓ output bits from the output plaintext polynomial

Aside: an ℓ-SELECT Operation

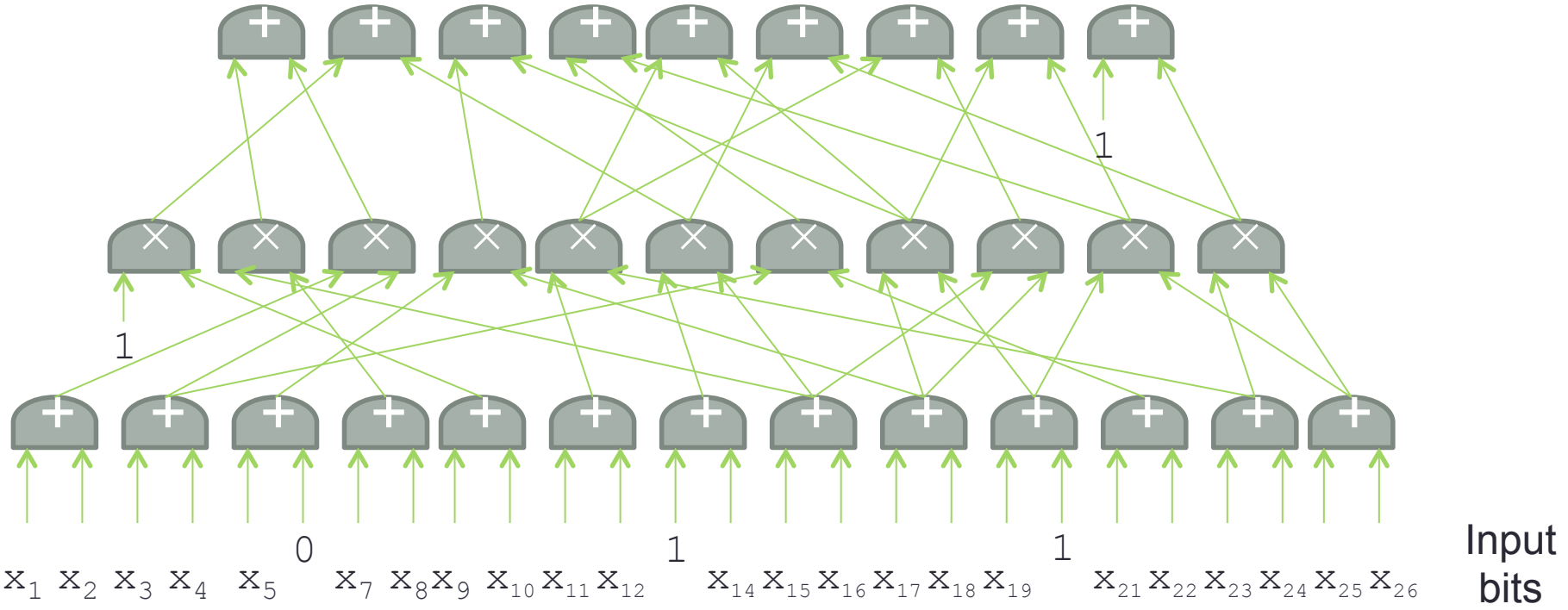


- We will use this later

COMPUTING ON DATA ARRAYS

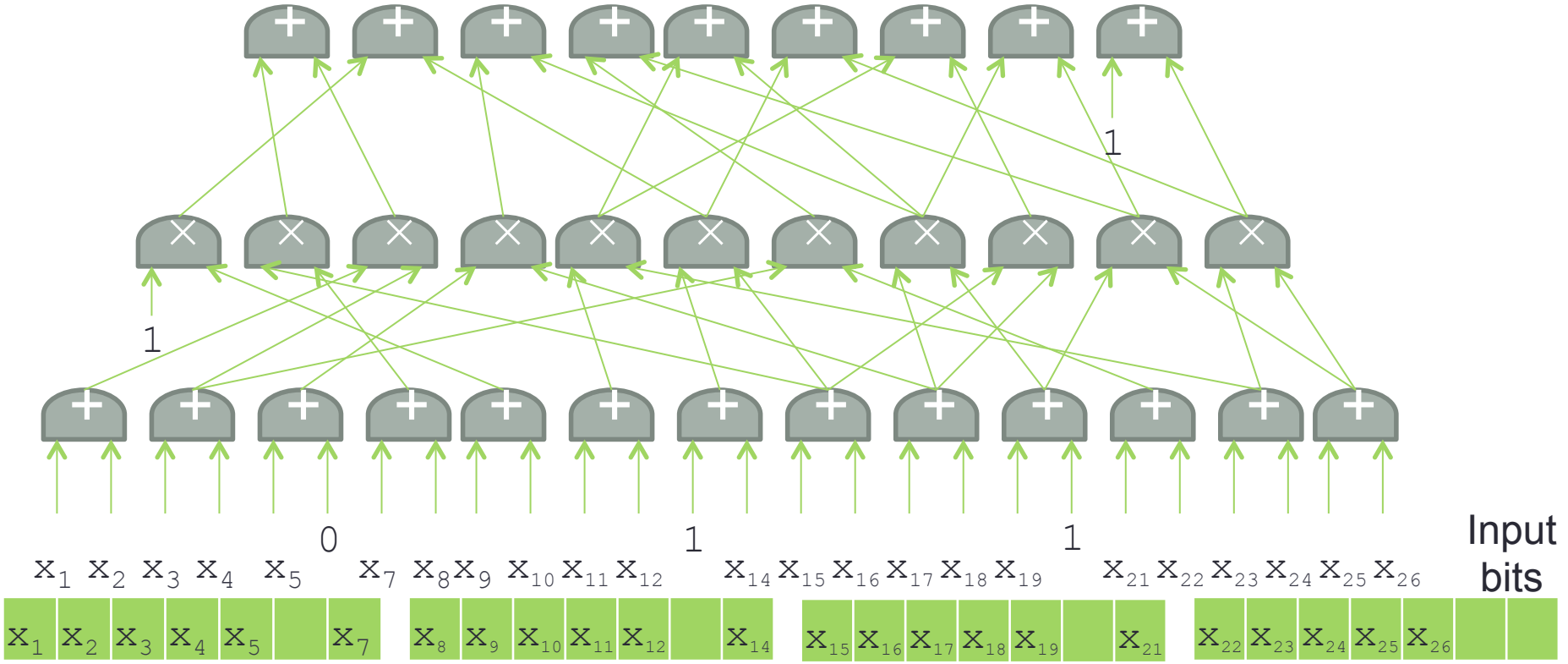
Forget about encryption for the moment…

So you want to compute some function…



ADD and MUL are a *complete* set of operations.

So you want to compute some function using SIMD…

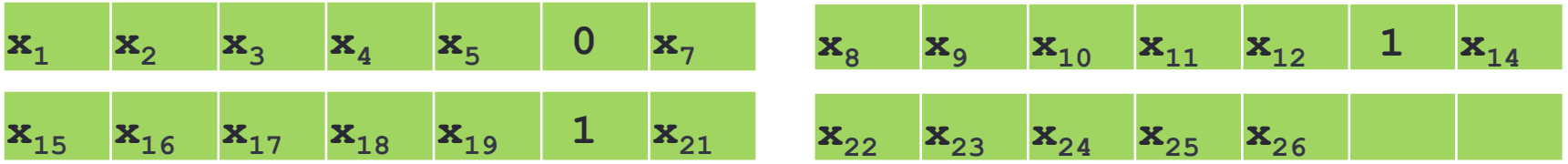


ℓ-ADD and ℓ-MUL are not a complete set of operations!!!

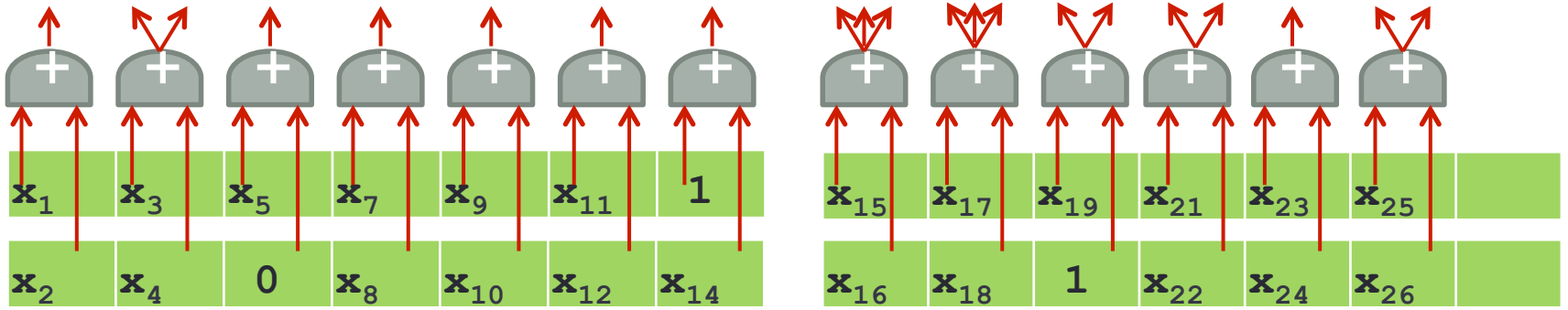
… unless, of course, we use ℓ=1… ☹

Routing Values Between Levels

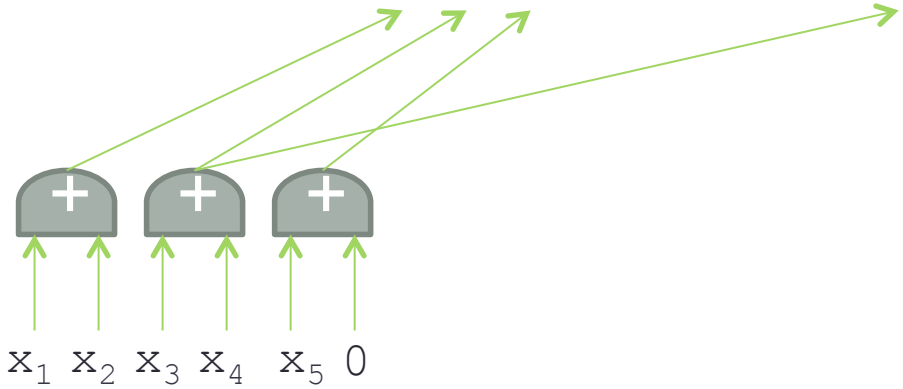
- We need to map this

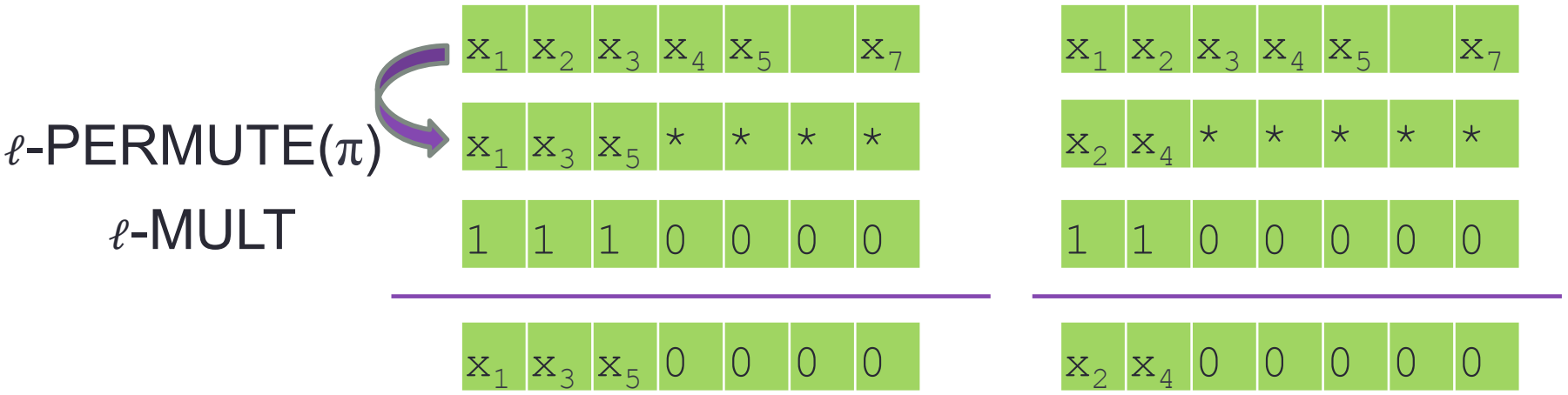


- Into that … so we can use ℓ-add

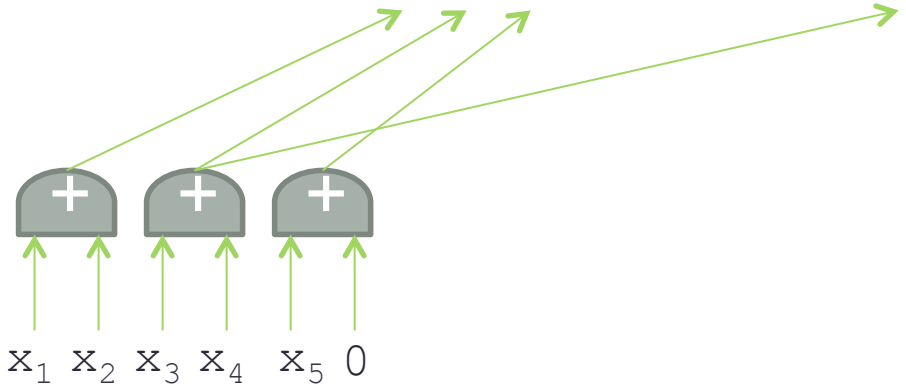


ℓ-ADD, ℓ-MUL, ℓ-PERMUTE:
a complete set of SIMD ops





ℓ-ADD, ℓ-MUL, ℓ-PERMUTE:
a complete set of SIMD ops

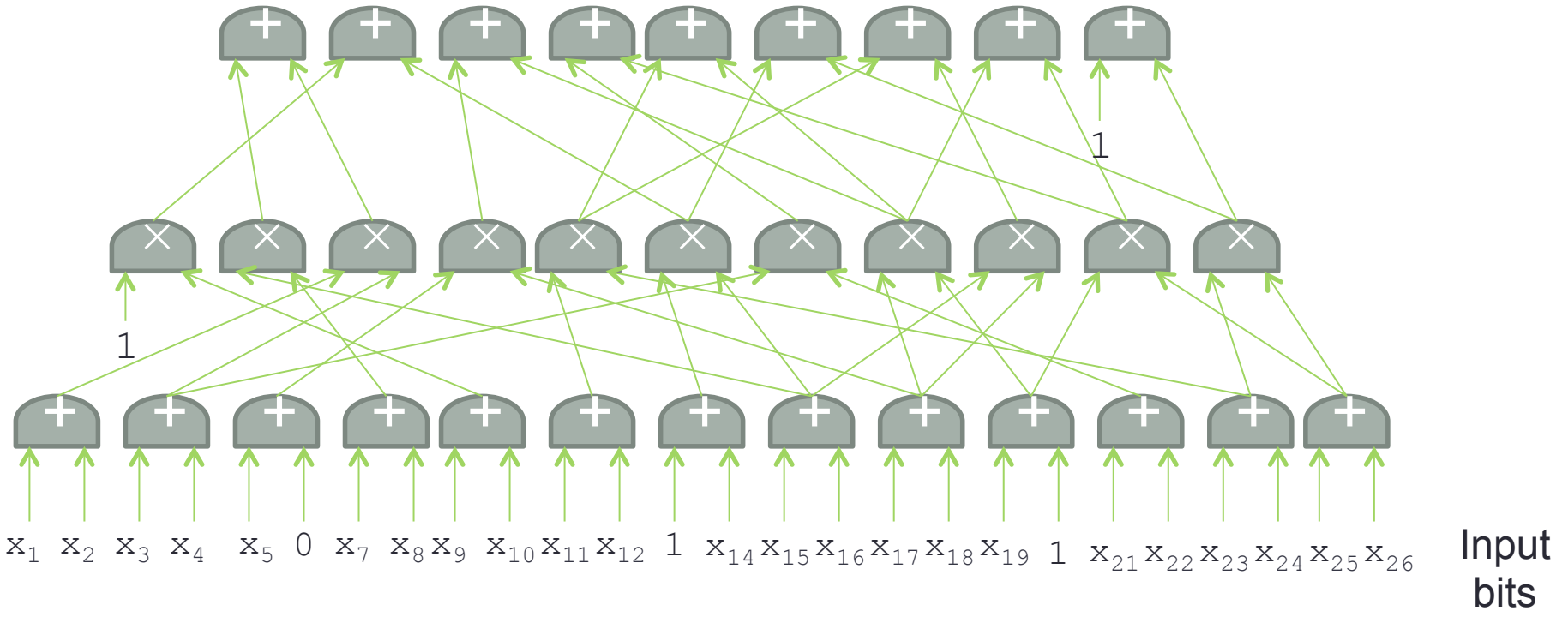


ℓ-ADD

x_1 + x_2	x_3 + x_4	x_5	0	0	0	0

x_2	x_4	0	0	0	0	0

ℓ-ADD, ℓ-MUL, ℓ-PERMUTE:
a complete set of SIMD ops



Use ℓ-PERMUTE for routing between circuit levels

- Not quite obvious

Routing Values Between Levels: Three Problems to Solve

1. How to implement ℓ-permute?
   - $a \in R \downarrow 2$ encodes ℓ-array using polynomial-CRT
   - We are given an encryption of a
2. Fan-out: need to **clone** values from high fan-out gates before routing to next level
3. **Big permutation**: For a width-W level, we need a permutation over 2W values
   - Implemented using ℓ-permute on ℓ-arrays
   - Even when $W \gg \ell$

1. Implementing ℓ-Permute

- Recall: native plaintext is binary polynomial modulo $\Phi\downarrow m\,(X)$, $a\in R\downarrow 2 = Z\downarrow 2\,[X]/\Phi\downarrow m\,(X)$
- $a\cong[\alpha\downarrow 1\,, ..., \alpha\downarrow \ell\,]$, $\alpha\downarrow j = a(\zeta\uparrow t\downarrow j\,)$
  - $a+a\uparrow' \cong[\alpha\downarrow 1 + \alpha\downarrow 1\uparrow'\,, ..., \alpha\downarrow \ell + \alpha\downarrow \ell\uparrow'\,]$
  - $a\times a\uparrow' \cong[\alpha\downarrow 1 \times \alpha\downarrow 1\uparrow'\,, ..., \alpha\downarrow \ell \times \alpha\downarrow \ell\uparrow'\,]$
- Is there a natural operation on polynomials that moves values between slots?

Moving Values Between Slots

- [BGV12] use automorphisms $\kappa\downarrow j : a(X) \rightarrow a(X^j)$
  - Similar technique in [LPR'10]
- For the example $m=63$, $T=\{1,5,5\uparrow 2, ..., 5\uparrow 5\}$
  - Consider $b=\kappa\downarrow 5$ $(a)=a(X\uparrow 5)$
  - Then $b(\zeta\uparrow 5\uparrow j)=a((\zeta\uparrow 5\uparrow j)\uparrow 5)=a(\zeta\uparrow 5\uparrow j+1)$
  - If $a\cong[\alpha\downarrow 1, \alpha\downarrow 2, ..., \alpha\downarrow \ell-1, \alpha\downarrow \ell]$ then b$\cong[\alpha\downarrow 2, \alpha\downarrow 3 ..., \alpha\downarrow \ell, \alpha\downarrow 1]$
- Can be used to shift by any amount
  - The general case a little more complicated

Homomorphic Automorphisms

- Roughly, applying $\kappa\downarrow j$ to the ciphertext c=Enc(a) yields an encryption of $\kappa\downarrow j\,(a)$
  - With respect to a different secret key
    - $\kappa\downarrow j\,(s)$ rather than s
  - But this can be fixed with key-switching
- So we can implement circular shifts
- But we need arbitrary permutations
  - In order to do intra-circuit routing

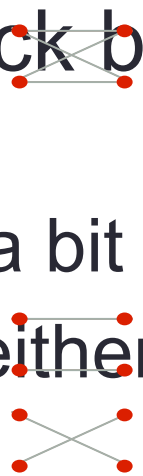
From Shifts to Arbitrary Permutations

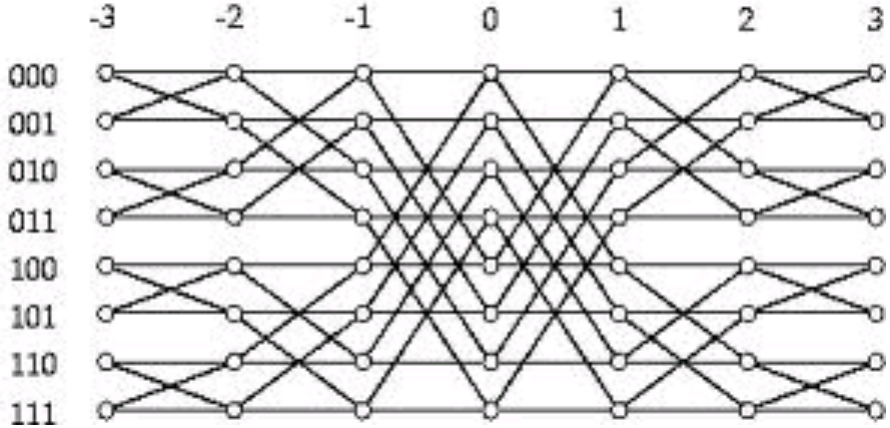
- For every j we can implement rotate-by-j
  - How to implement arbitrary given permutation π?
- A naïve solution:
  - For every i, let $a{\downarrow}i = [a$ rotated by $\pi(i) - i]$
  - Use a big SELECT on all the $a{\downarrow}i$'s
    - Pick the slot $\pi(i)$ from $a{\downarrow}i$
- Implements π, but takes $\Theta(\ell)$ ops
  - Inefficient, we might as well not use SIMD

From Shifts to Arbitrary Permutations

Use Beneš/Walksman Permutation Networks:

- Two back-to-back but
  - Every exchange is controlled by a bit
  - Values sent on either straight edges or cross edges

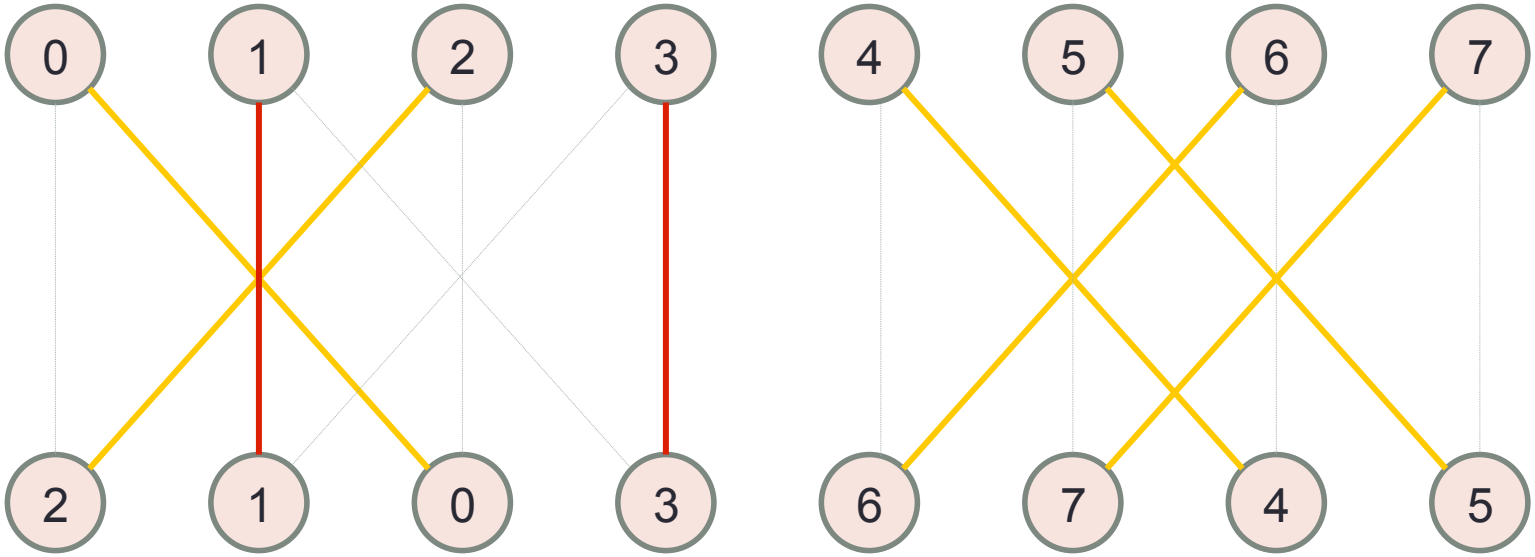




- Every permutation can be realized by appropriate setting of the control bits

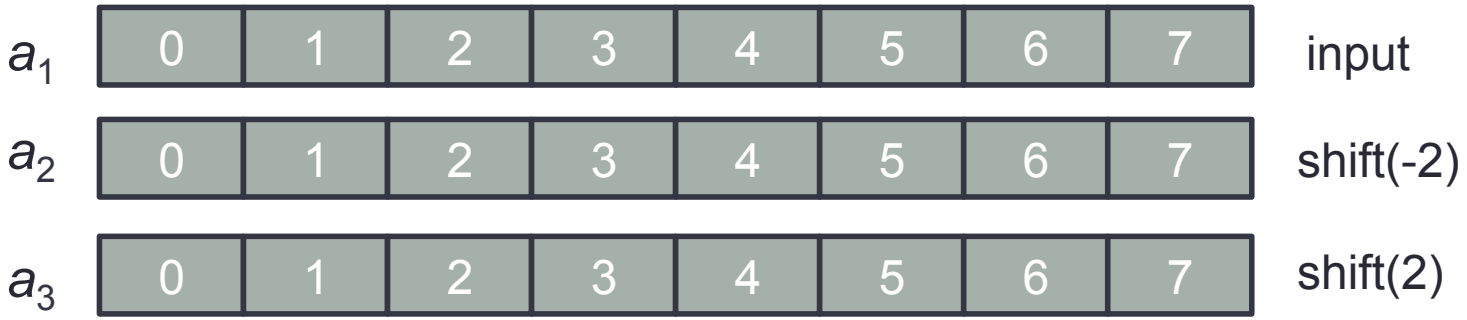
Realizing Permutation Networks

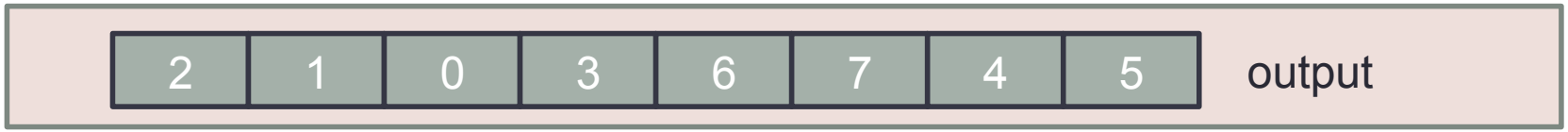
- Claim: every butterfly level can be realized by two shifts and two SELECTs
- Example:



Control bits: **1 0 1 1**

Realizing Permutation Networks





Realizing Permutation Networks

Realizing Permutation Networks

Claim: every level of the Benes network can be realized by two shifts and two SELECTs

Proof : In every level, all the exchanges are between nodes at the same distance

- Distance 2^i for some i

Can implement all these exchanges using shift(2^i), shift(-2^i), and two SELECTs

Realizing Permutation Networks

- Every level takes 2 shifts and 2 SELECTs
- There are 2log(ℓ) levels

$\Rightarrow$Any permutation on ℓ-arrays can be realized using 4log(ℓ) shifts and 4log(ℓ) SELECTs

- Some more complications when ℓ is not a power of two
  - But still only O(log ℓ) operations

Routing Values Between Levels

✓Implementing ℓ-permute

- Using $X \mapsto X \uparrow j$ to get simple shifts
- Benes network to get arbitrary permutation
- Takes O(log ℓ) operations

see paper

2. Cloning values from high fan-out gates
3. Permutations over $W \gg \ell$ elements
   - How to handle large permutations?

Handling Large Permutations

- A width-W level has $n = \lceil W/\ell \rceil$ ℓ-arrays
- How to permute these W values?
  - Using ℓ-ADD, ℓ-MUL, ℓ-PERMUTE

Theorem (Lev, Pippenger, Valiant '84): Any permutation π over $n \cdot \ell$ values (viewed as a rectangle) can be decomposed as $\pi = \pi_3 \circ \pi_2 \circ \pi_1$, where:

  - π_1 only permutes within the columns
  - π_2 only permutes within the rows
  - π_3 only permutes within the columns

Decomposing Permutations

13	18	14	16	12
15	3	4	5	6
7	8	9	20	19
2	17	1	11	10

1	2	3	4	5
6	7	8	9	10
11	12	13	14	15
16	17	18	19	20

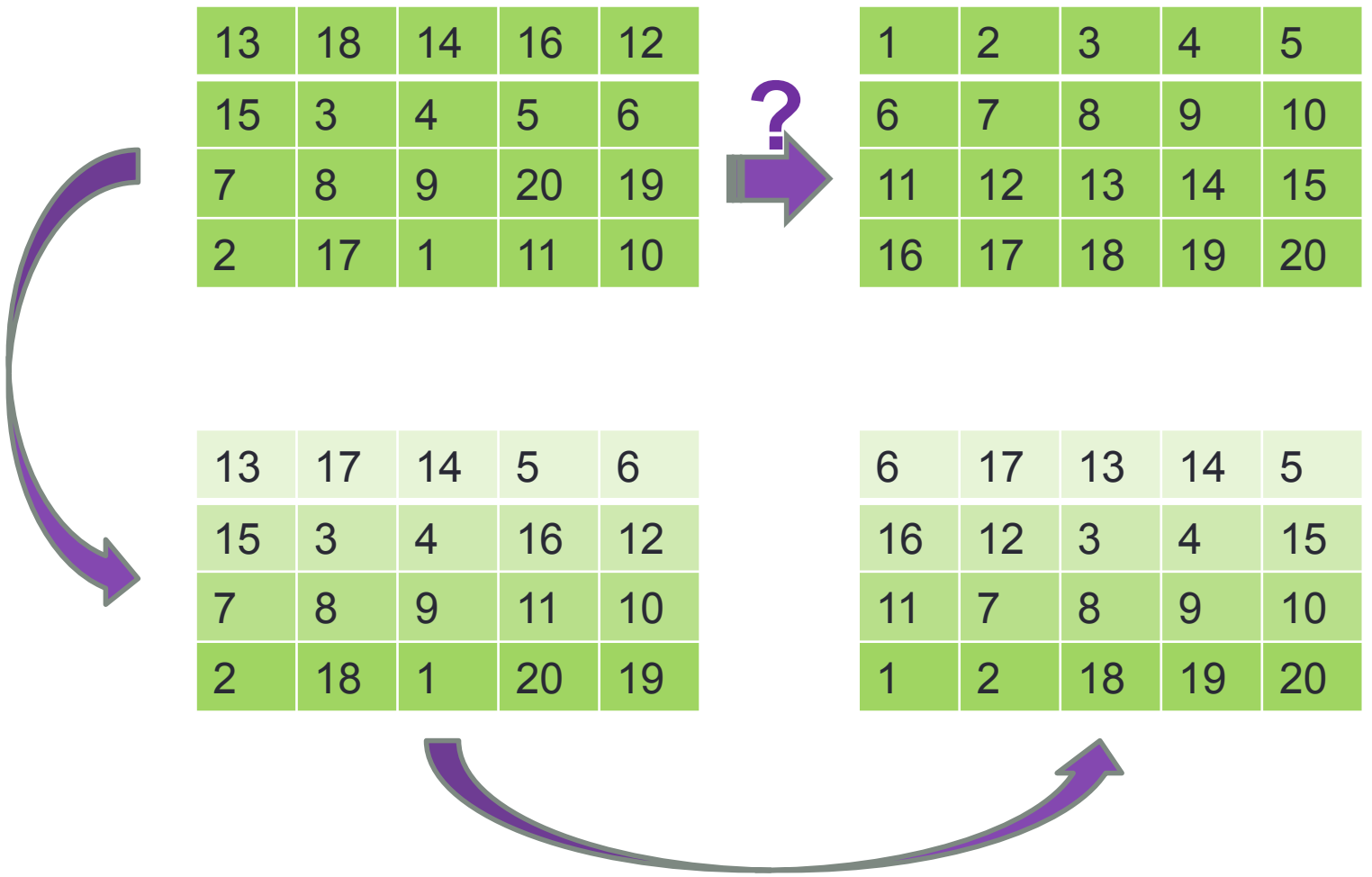
Decomposing Permutations

13	18	14	16	12
15	3	4	5	6
7	8	9	20	19
2	17	1	11	10

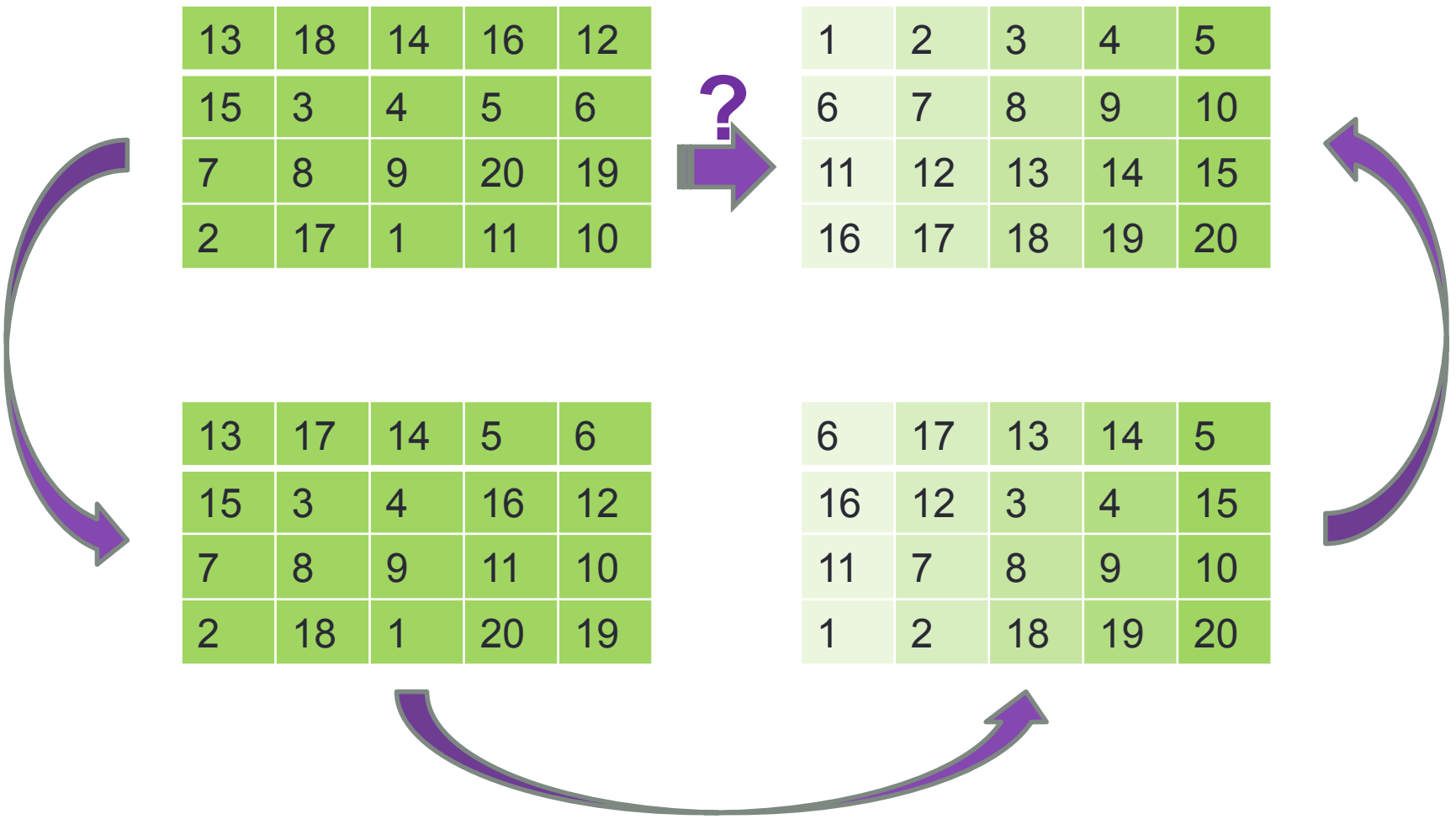
1	2	3	4	5
6	7	8	9	10
11	12	13	14	15
16	17	18	19	20

13	17	14	5	6
15	3	4	16	12
7	8	9	11	10
2	18	1	20	19

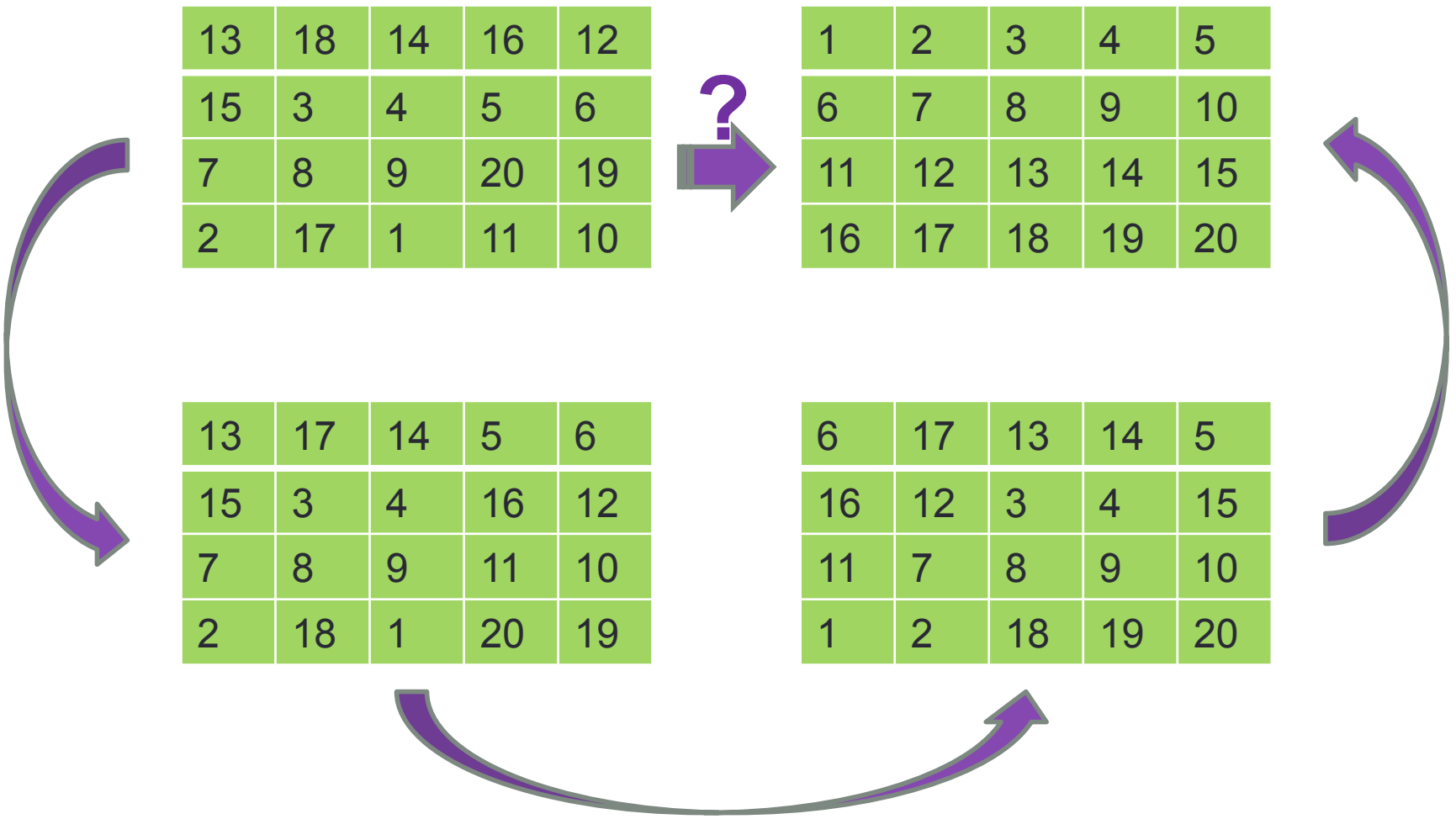
Decomposing Permutations



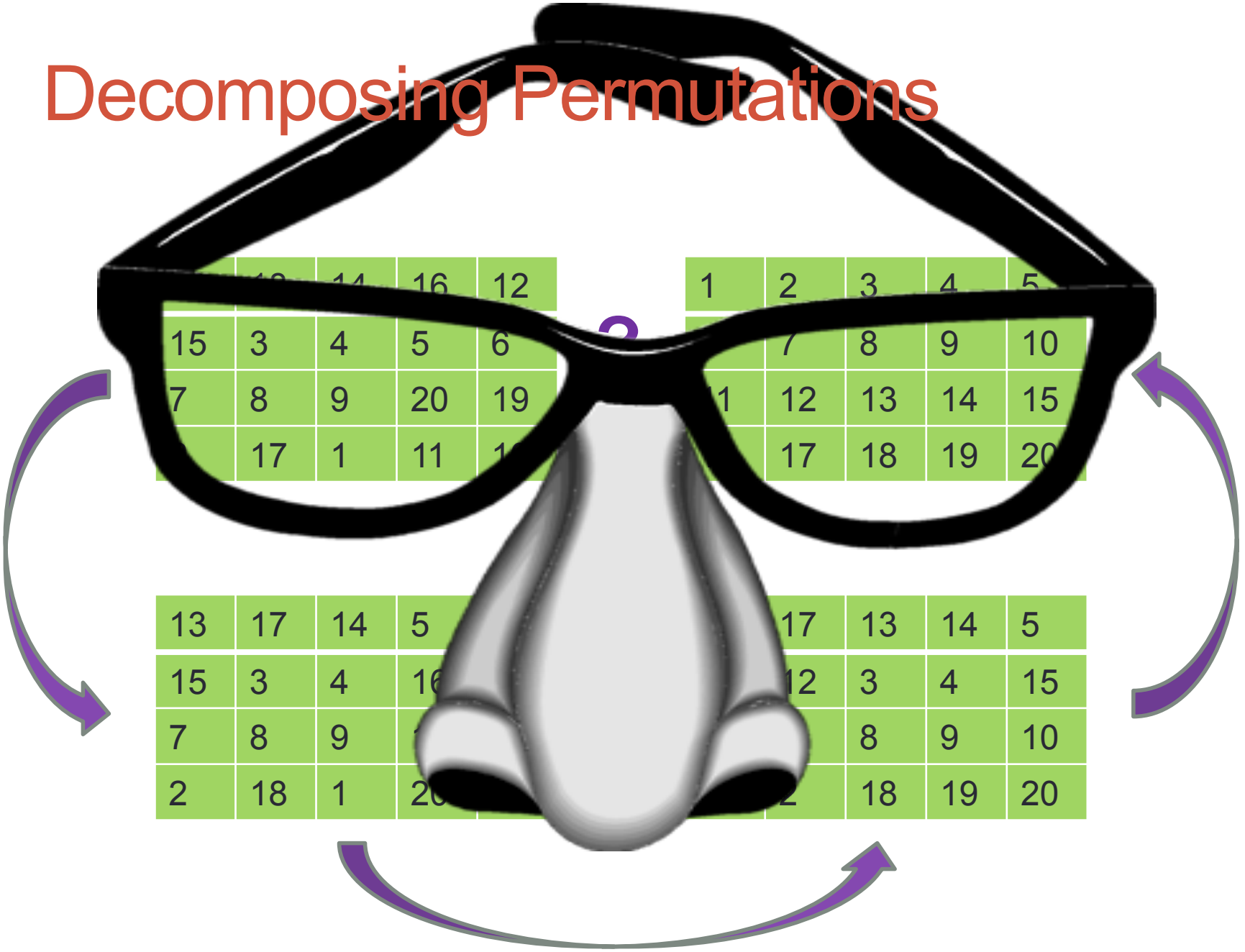
Decomposing Permutations



Decomposing Permutations



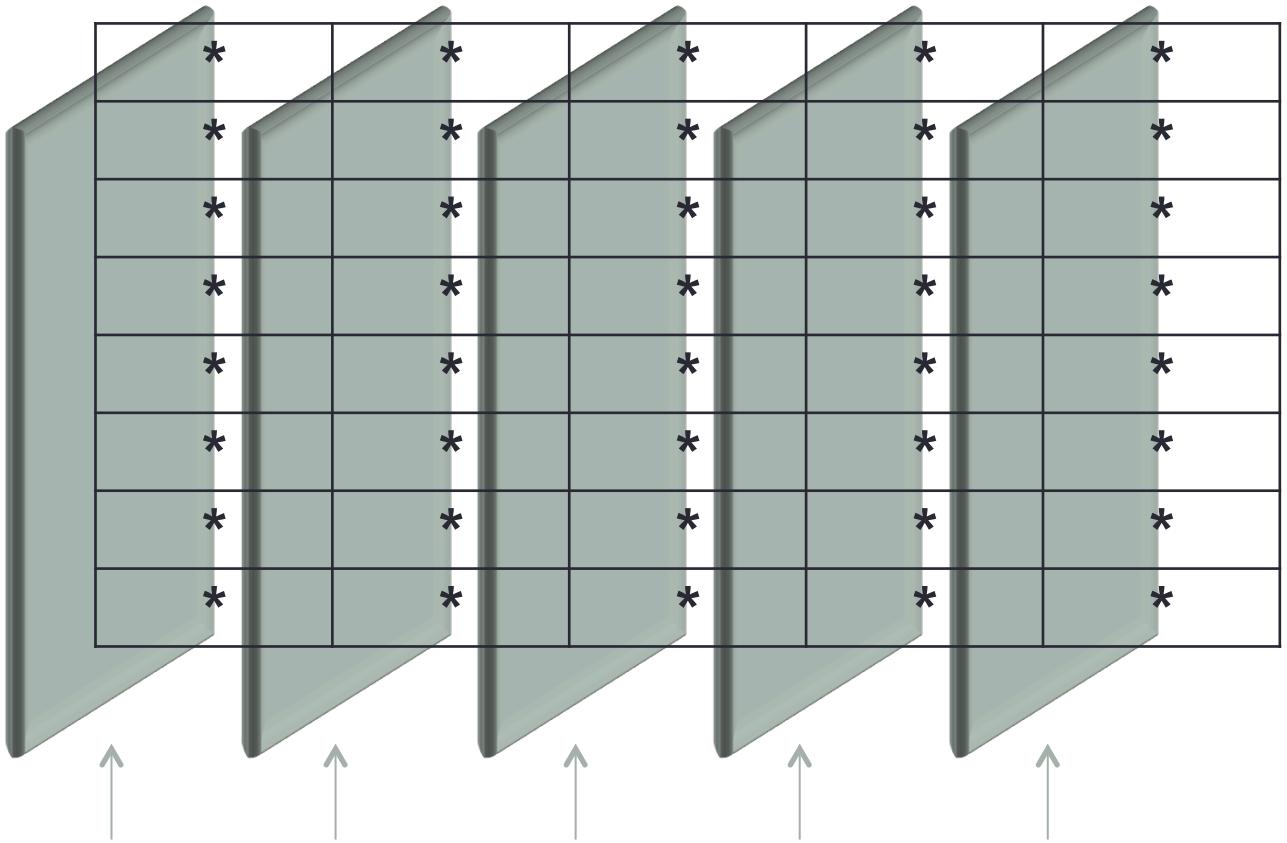
Decomposing Permutations



Handling Large Permutations

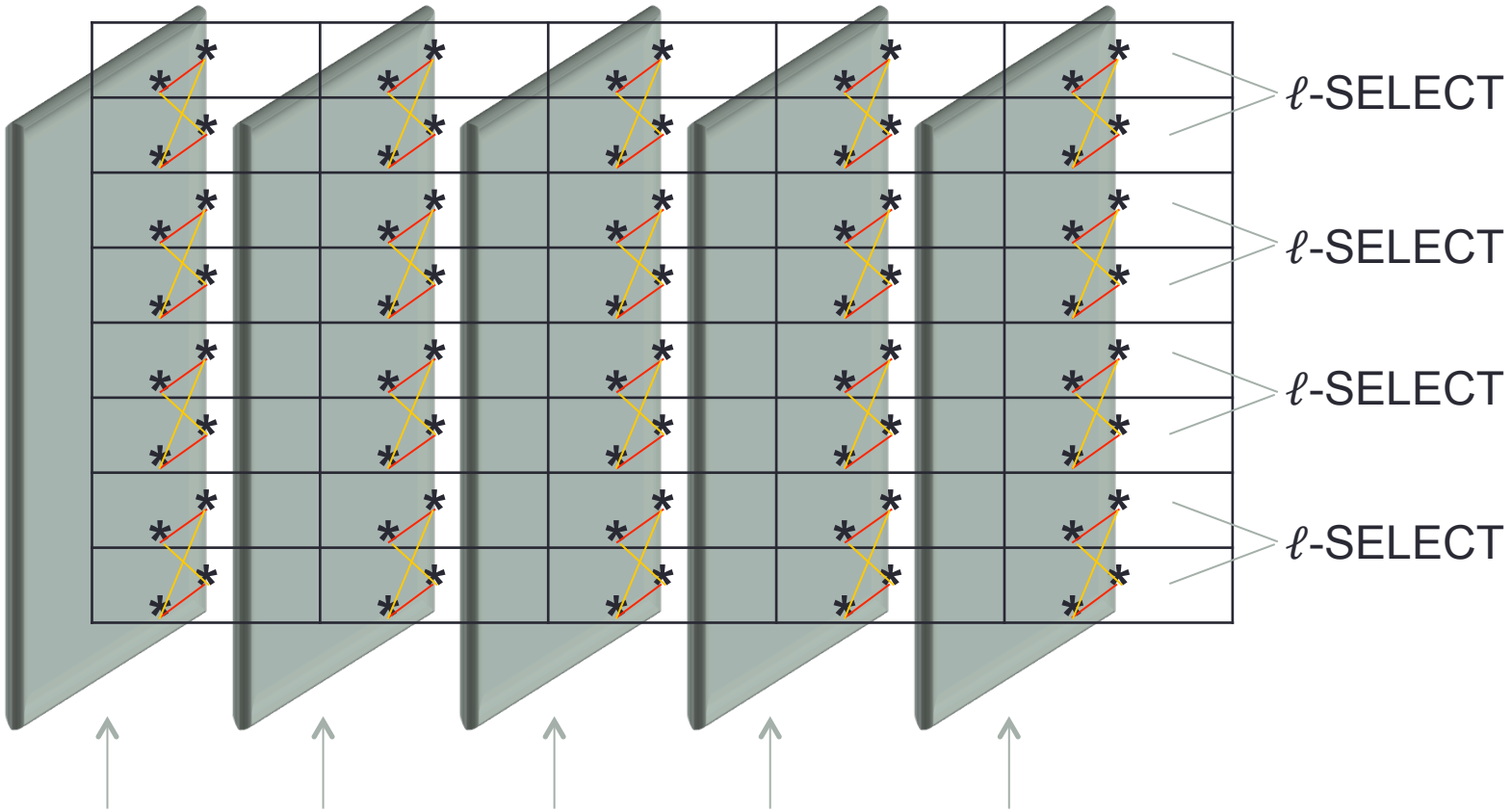
- Input: n such ℓ-arrays (each is a ciphertext)
  - Consider each ℓ-array as a row in $n \times \ell$ matrix
  - We have a permutation π over the $n \cdot \ell$ slots
- Decompose $\pi = \pi_3 \circ \pi_2 \circ \pi_1$
- Row perm's π_2 implemented with ℓ-PERMUTE
  - Permuting each row separately
- Column perm's π_1, π_3 implemented with ℓ-SELECT
  - Permuting all the columns in parallel
  - For each column a different permutation

Permuting The Columns

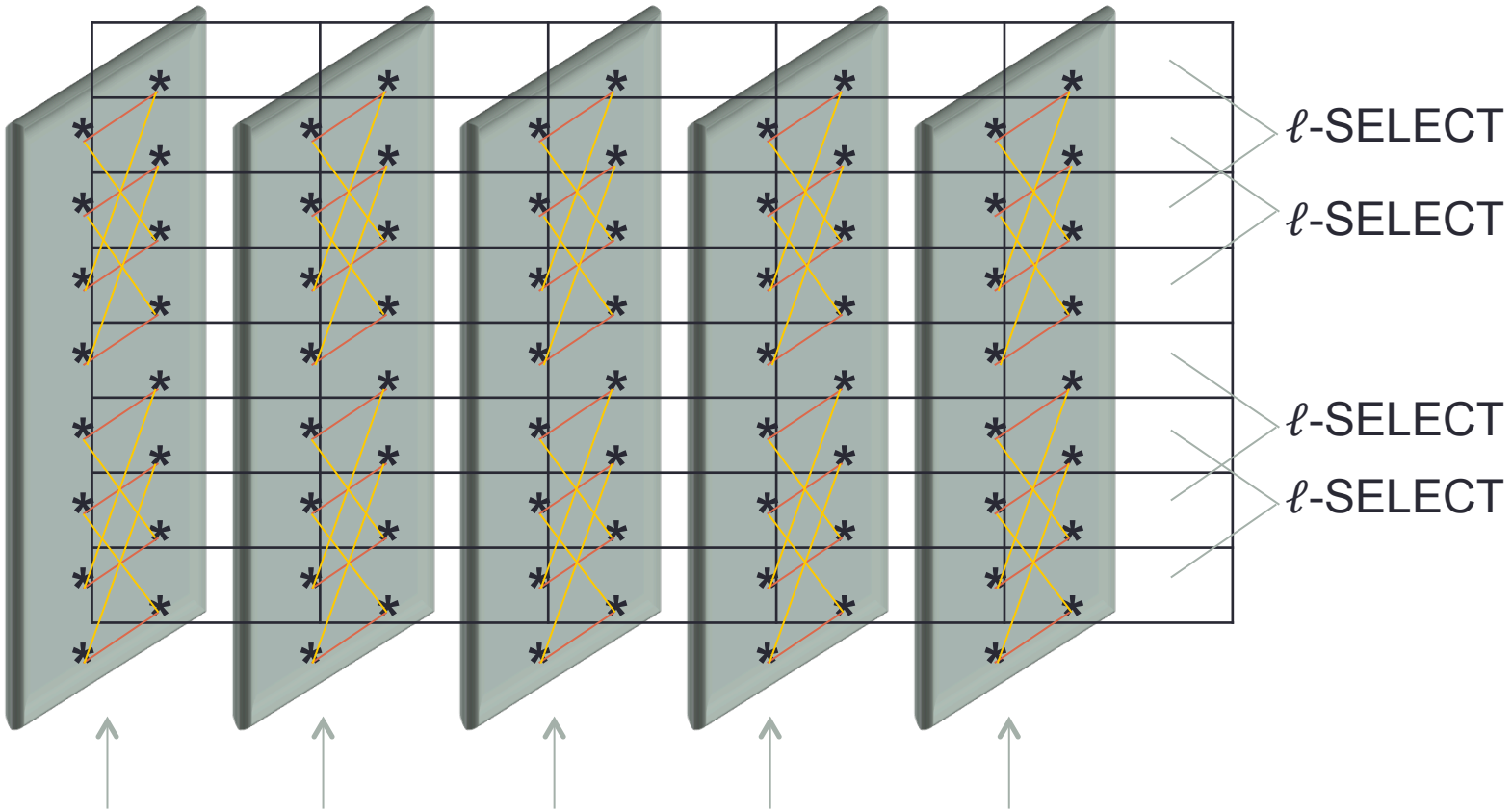


Implement a Beneš network over each column

Permuting The Columns



Permuting The Columns



Handling Large Permutations

- The two column permutations, π_1,π_3:
  - Each level takes $n/2$ ℓ-SELECT operations
    - Level applied to all the networks in parallel
  - $\log(n)$ levels ➔ $n\log(n)$ operations for each π_i
- The row permutation, π_2:
  - n separate applications of ℓ-PERMUTE
    - Each takes $O(\log(\ell))$ operations
  - $O(n\log(\ell))$ operations overall
- Total number of operations is $O(n\log(n) + n\log(\ell)) = O(n\log(n\ell)) = O(W/\ell \log(W))$

Routing Values Between Levels

✓Implementing ℓ-permute

- Using $X \mapsto X\uparrow j$ to get simple shifts
- Benes network to get arbitrary permutation
- Takes O(log ℓ) operations

2. Cloning values from high fan-out gates

see paper

✓Permutations over $W \gg \ell$ elements

➔ **Intra-level routing takes** O(W/ℓ log(W)) **ops**

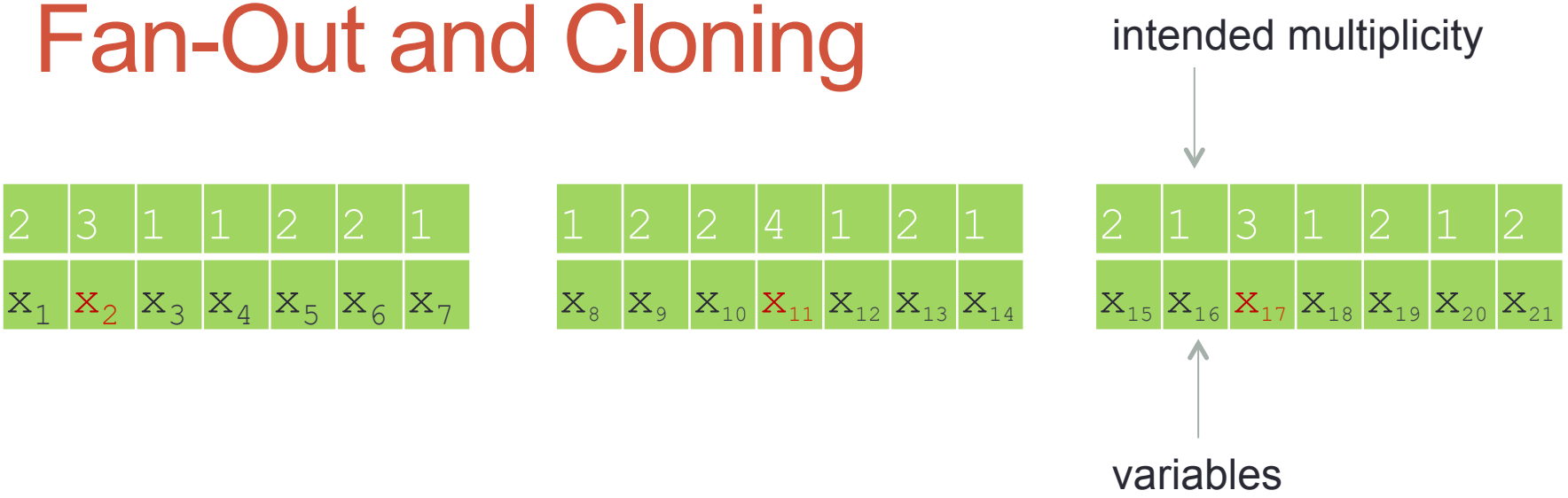
- For a width-W level

Low Overhead Homomorphic Encryption

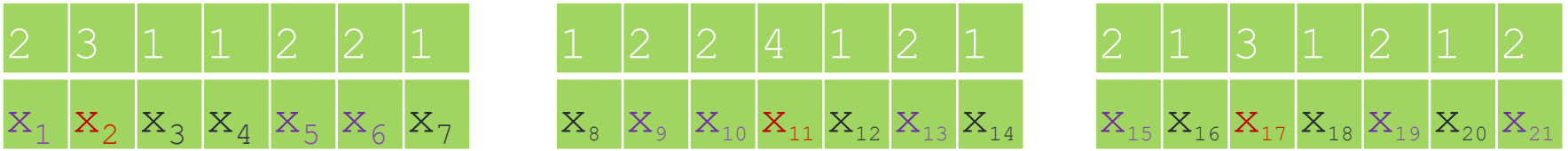
- Pack inputs into ℓ-arrays
  - ℓ can made as large as $\Omega(\lambda)$
- SIMD operations to implement each level
- Route values to their place for next level
- Each level takes $O(\lceil W/\lambda \rceil \cdot \lambda)$ work
- Total work for size-T width-W circuit is $O(\lceil W/\lambda \rceil \cdot \lambda \cdot T/W)$

QUESTIONS?

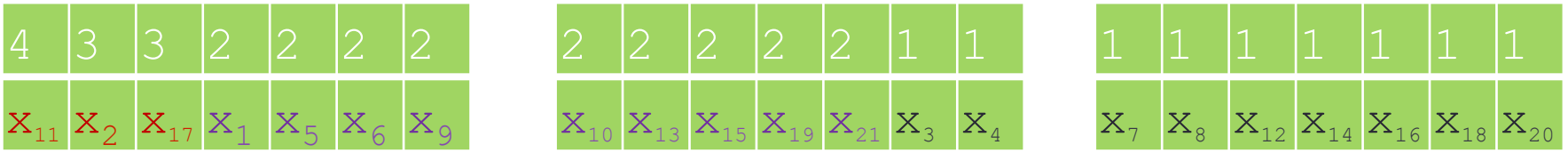
Fan-Out and Cloning



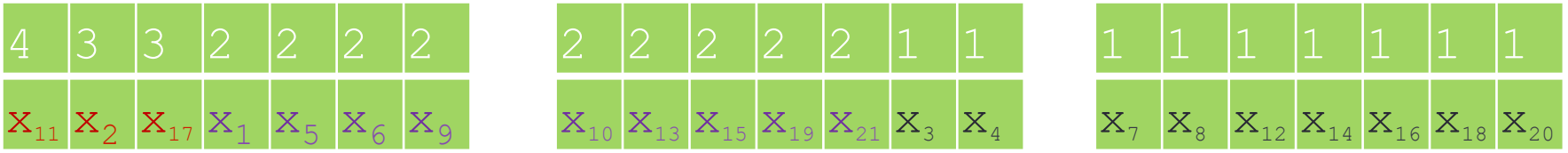
Fan-Out and Cloning



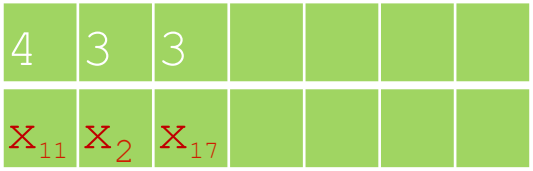
Sort by intended multiplicity:



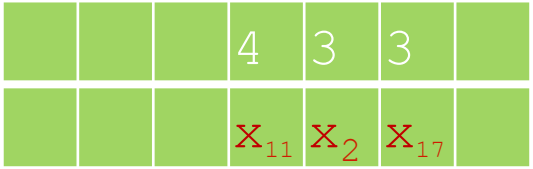
Fan-Out and Cloning



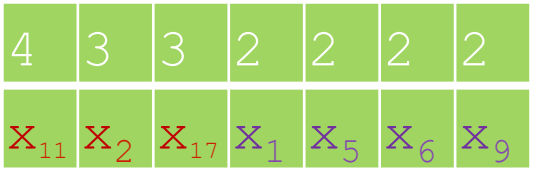
Replicate

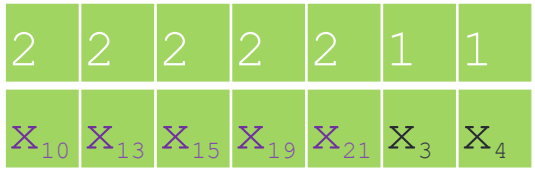


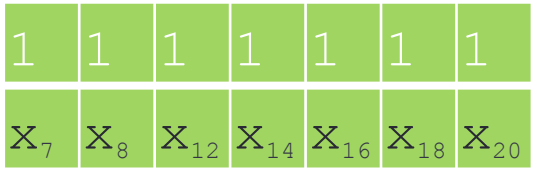
Replicate and shift



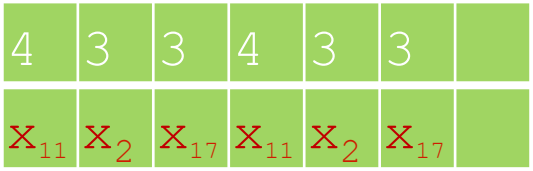
Fan-Out and Cloning



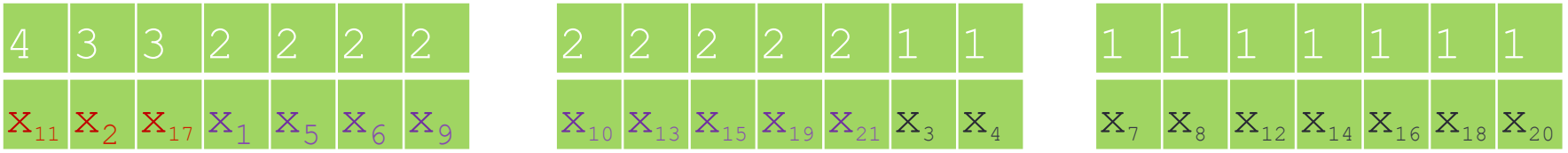




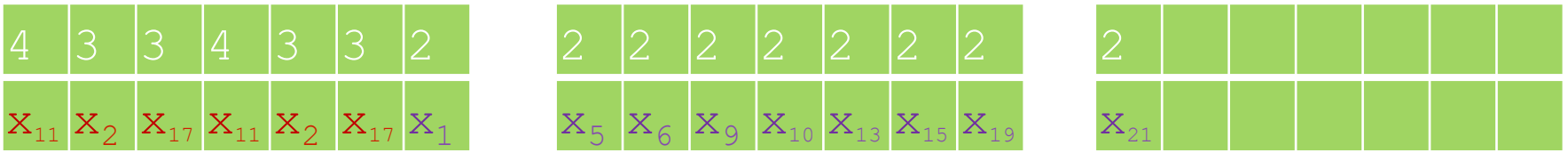
Merge



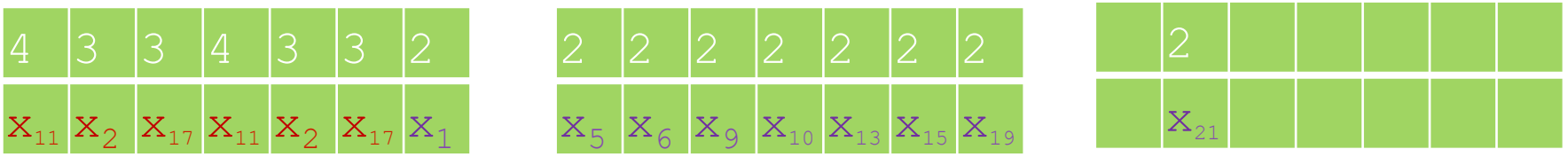
Fan-Out and Cloning



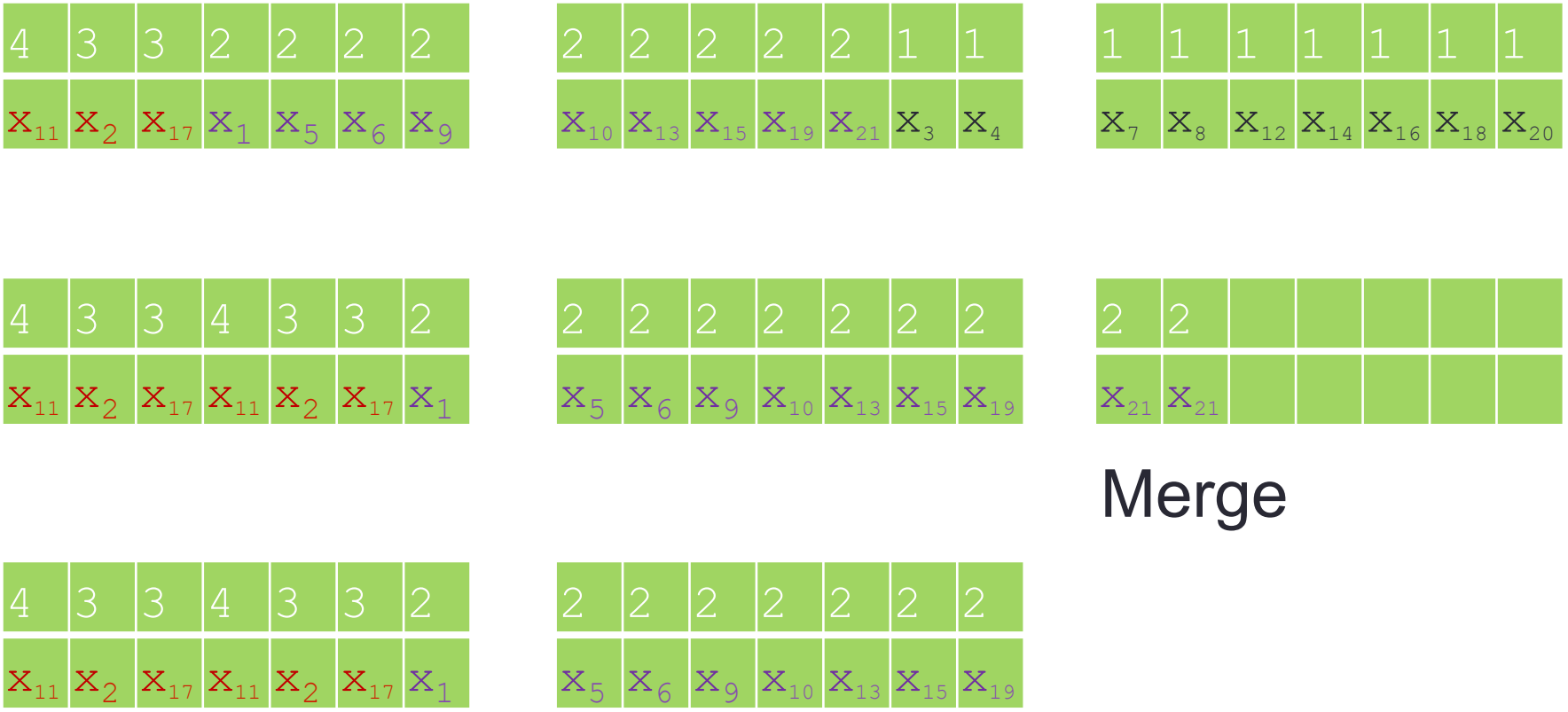
Replicate, shift, merge



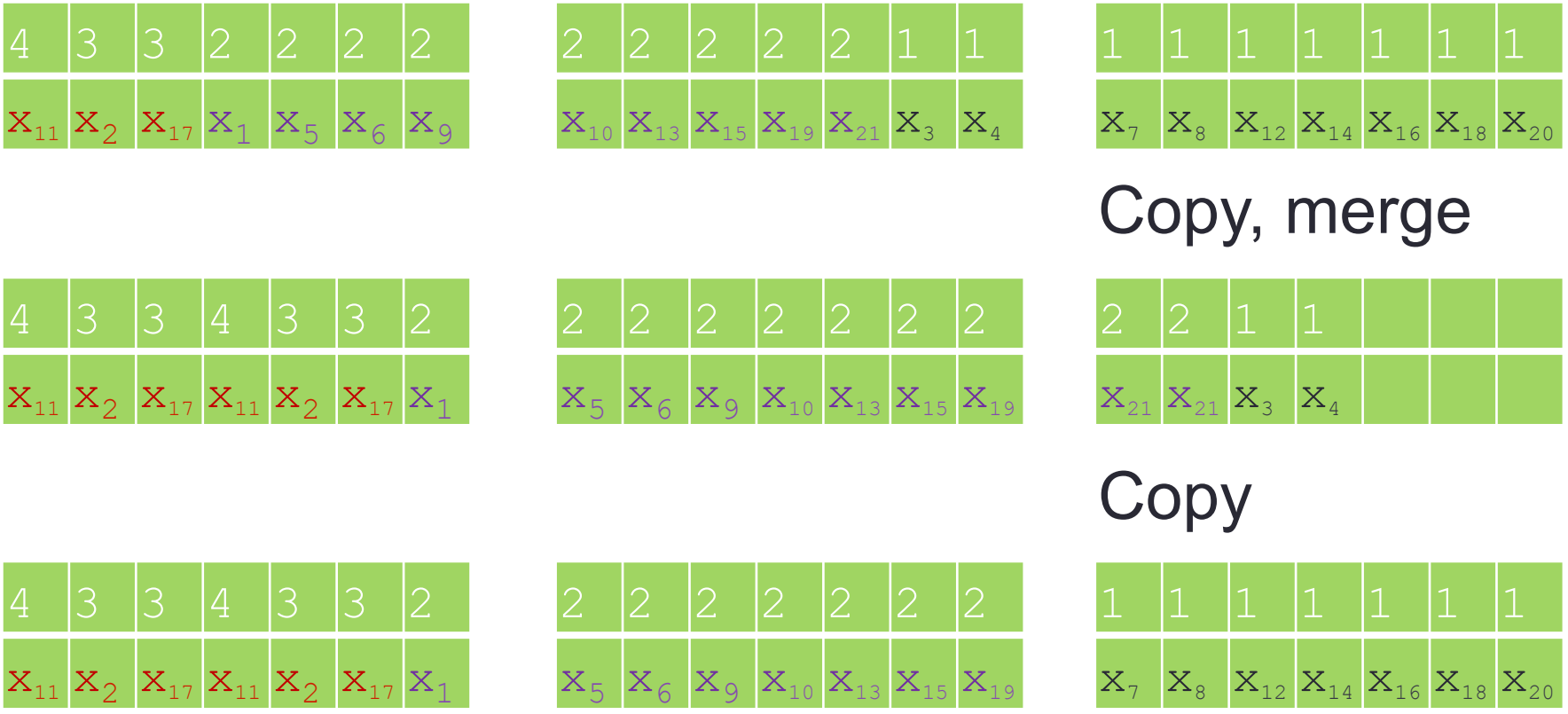
Replicate, shift



Fan-Out and Cloning



Fan-Out and Cloning



Each variable appears at least as much as needed