Analysis of Algorithms: Data Cleanup Algorithms

Garrett Dancik, PhD

Fall 2021

Course Notes: https://gdancik.github.io

What do we mean by Data Cleanup?

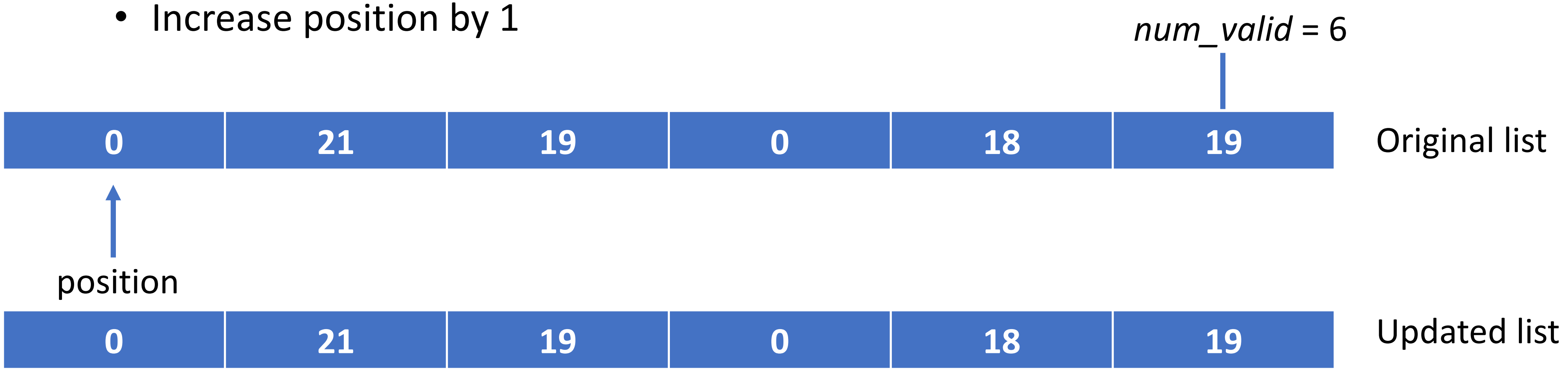
- If data contains invalid or missing values, those invalid values should be removed.
  - In a survey, a student does not enter their age (or enters an invalid one)
  - In a survey, a student does not enter their GPA (or enters an invalid one)
- We will assume that missing / invalid values are recorded as 0
- Example data:

0	**21**	**19**	**0**	**18**	**19**

- In this case, we want a list containing only the numbers: 21, 19, 18, and 19

Shuffle-left algorithm

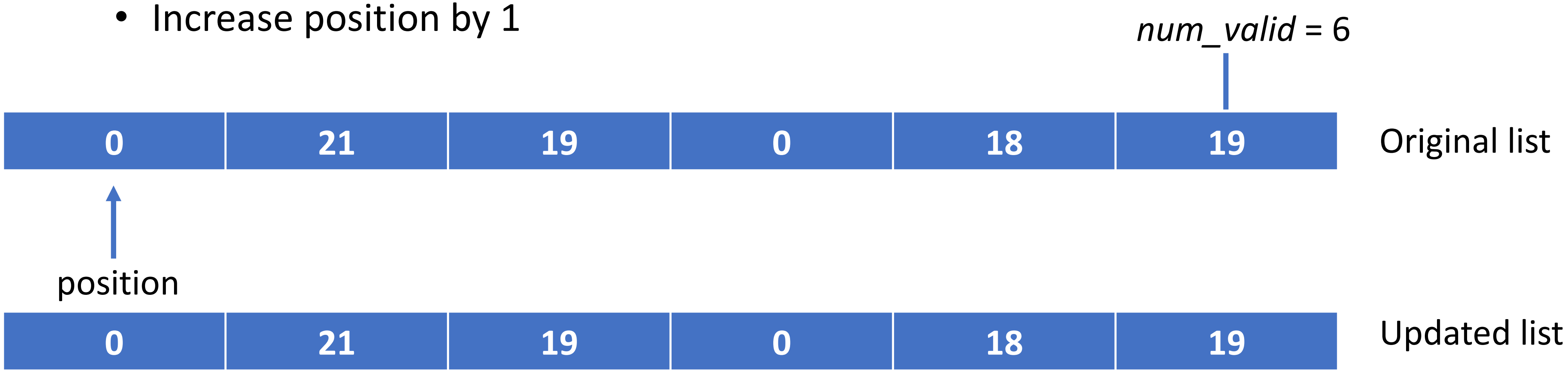
- While *position* <= *num_valid* :
  - If *num*[*position*] is invalid, e.g., 0 :
    - All valid numbers to the right of *num* are shifted 1 position to the left
    - Decrease *num_valid* by 1
  - Else:
    - Increase position by 1



Since the first number is 0, we shift all other numbers one position to the left

Shuffle-left algorithm

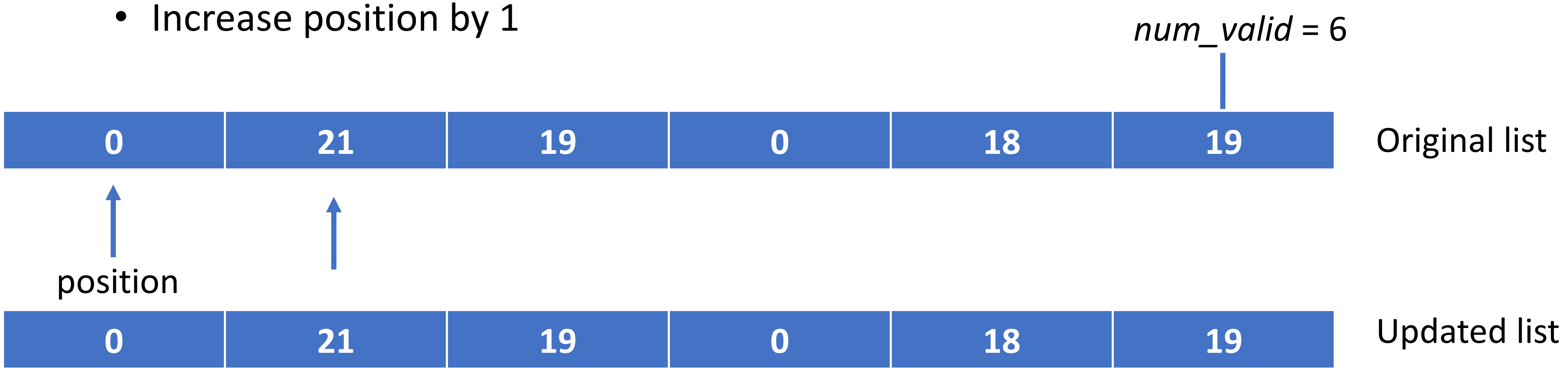
- While *position* <= *num_valid* :
  - If *num*[*position*] is invalid, e.g., 0 :
    - All valid numbers to the right of *num* are shifted 1 position to the left
    - Decrease *num_valid* by 1
  - Else:
    - Increase position by 1



Since the first number is 0, we shift all other numbers one position to the left

Shuffle-left algorithm

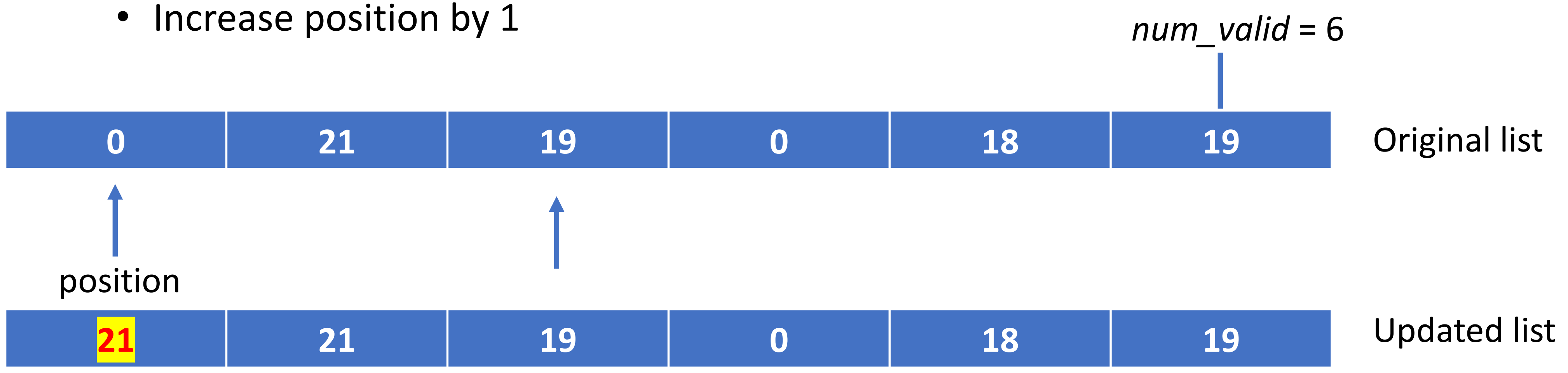
- While *position* <= *num_valid* :
  - If *num*[*position*] is invalid, e.g., 0 :
    - All valid numbers to the right of *num* are shifted 1 position to the left
    - Decrease *num_valid* by 1
  - Else:
    - Increase position by 1



Since the first number is 0, we shift all other numbers one position to the left

Shuffle-left algorithm

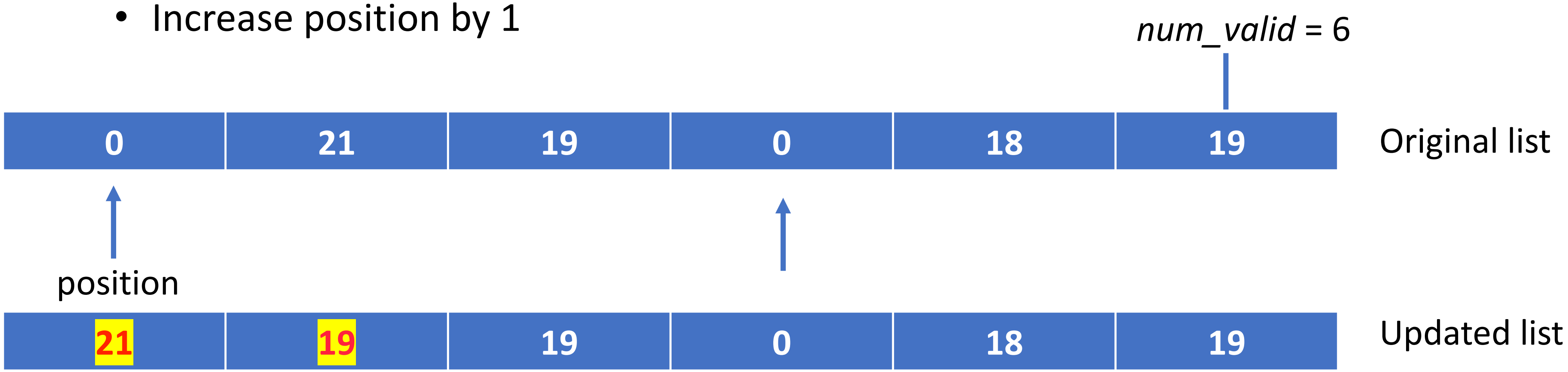
- While *position* <= *num_valid* :
  - If *num*[*position*] is invalid, e.g., -1 :
    - All valid numbers to the right of *num* are shifted 1 position to the left
    - Decrease *num_valid* by 1
  - Else:
    - Increase position by 1



Since the first number is 0, we shift all other numbers one position to the left

Shuffle-left algorithm

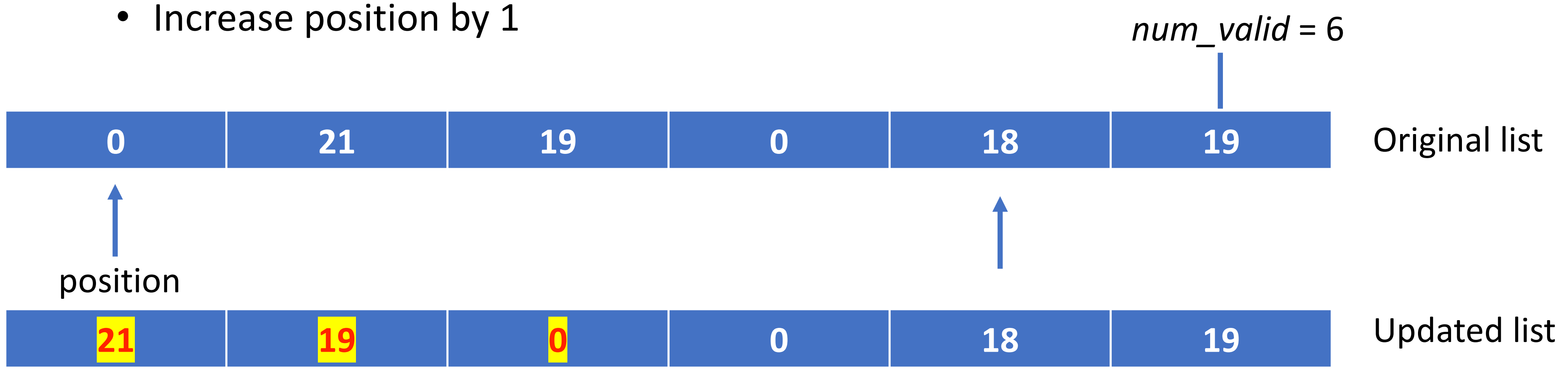
- While *position* <= *num_valid* :
  - If *num*[*position*] is invalid, e.g., 0 :
    - All valid numbers to the right of *num* are shifted 1 position to the left
    - Decrease *num_valid* by 1
  - Else:
    - Increase position by 1



Since the first number is 0, we shift all other numbers one position to the left

Shuffle-left algorithm

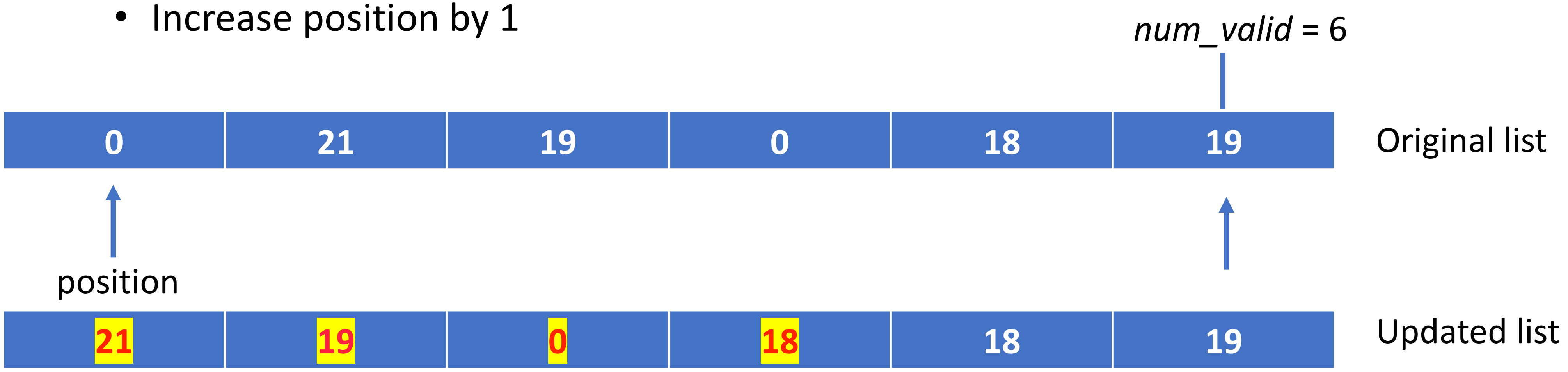
- While *position* <= *num_valid* :
  - If *num*[*position*] is invalid, e.g., 0 :
    - All valid numbers to the right of *num* are shifted 1 position to the left
    - Decrease *num_valid* by 1
  - Else:
    - Increase position by 1



Since the first number is 0, we shift all other numbers one position to the left

Shuffle-left algorithm

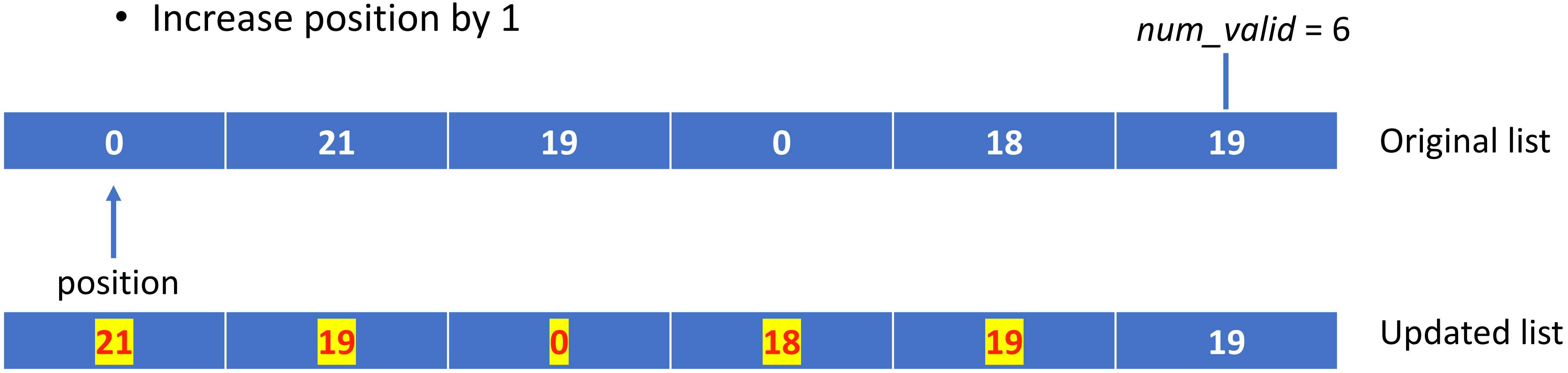
- While *position* <= *num_valid* :
  - If *num*[*position*] is invalid, e.g., 0 :
    - All valid numbers to the right of *num* are shifted 1 position to the left
    - Decrease *num_valid* by 1
  - Else:
    - Increase position by 1



Since the first number is 0, we shift all other numbers one position to the left

Shuffle-left algorithm

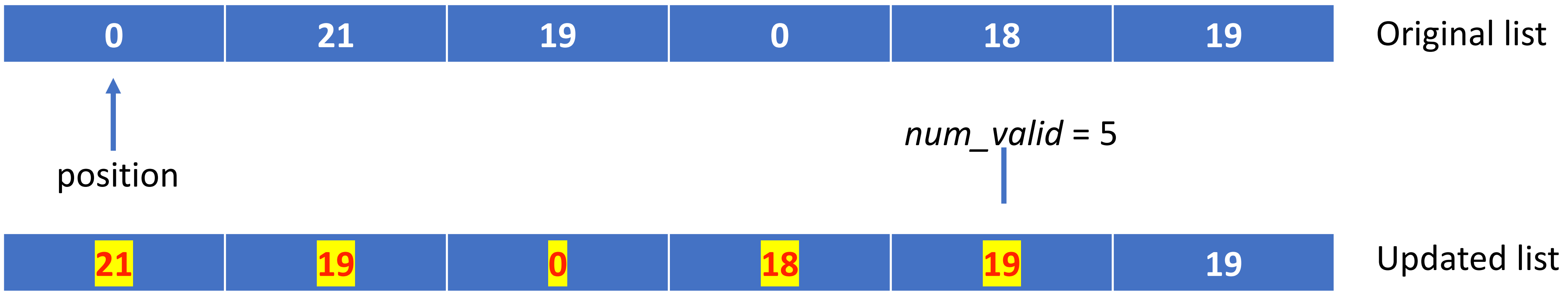
- While *position* <= *num_valid* :
  - If *num*[*position*] is invalid, e.g., 0 :
    - All valid numbers to the right of *num* are shifted 1 position to the left
    - Decrease *num_valid* by 1
  - Else:
    - Increase position by 1



Since the first number is 0, we shift all other numbers one position to the left

Shuffle-left algorithm

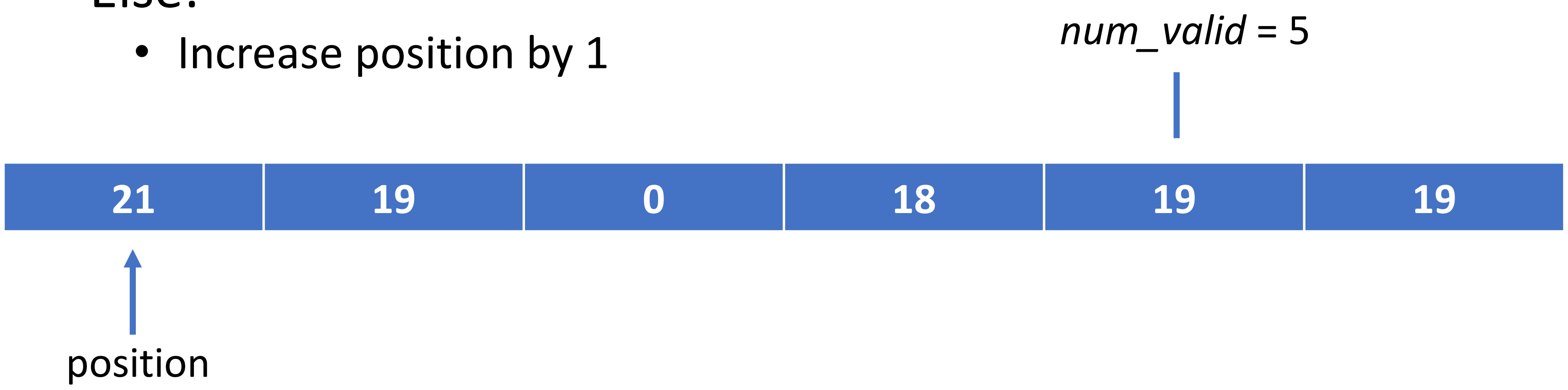
- While *position* <= *num_valid* :
  - If *num*[*position*] is invalid, e.g., 0 :
    - All valid numbers to the right of *num* are shifted 1 position to the left
    - Decrease *num_valid* by 1
  - Else:
    - Increase position by 1



Since the first number is 0, we shift all other numbers one position to the left

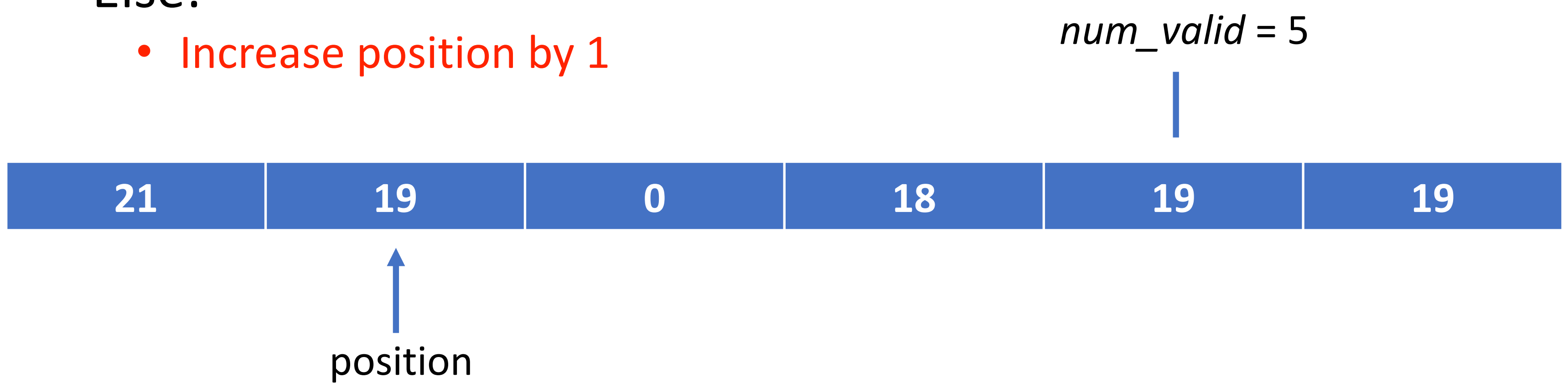
Shuffle-left algorithm

- While *position* <= *num_valid* :
  - If *num*[*position*] is invalid, e.g., 0 :
    - All valid numbers to the right of *num* are shifted 1 position to the left
    - Decrease *num_valid* by 1
  - Else:
    - Increase position by 1



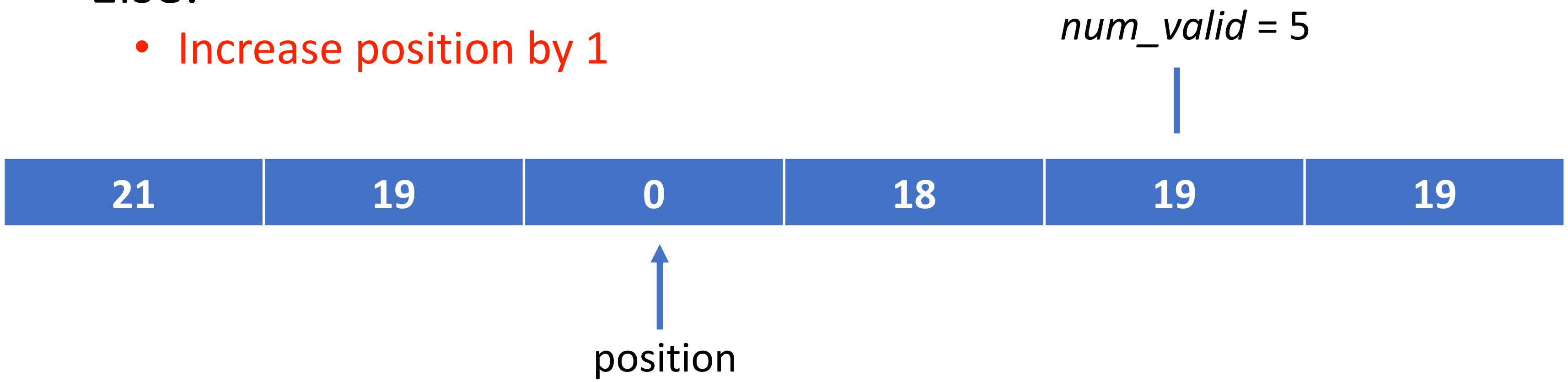
Shuffle-left algorithm

- While *position* <= *num_valid* :
  - If *num*[*position*] is invalid, e.g., 0 :
    - All valid numbers to the right of *num* are shifted 1 position to the left
    - Decrease *num_valid* by 1
  - Else:
    - Increase position by 1



Shuffle-left algorithm

- While *position* <= *num_valid* :
  - If *num*[*position*] is invalid, e.g., 0 :
    - All <u>valid</u> numbers to the right of *num* are shifted 1 position to the left
    - Decrease *num_valid* by 1
  - Else:
    - Increase position by 1

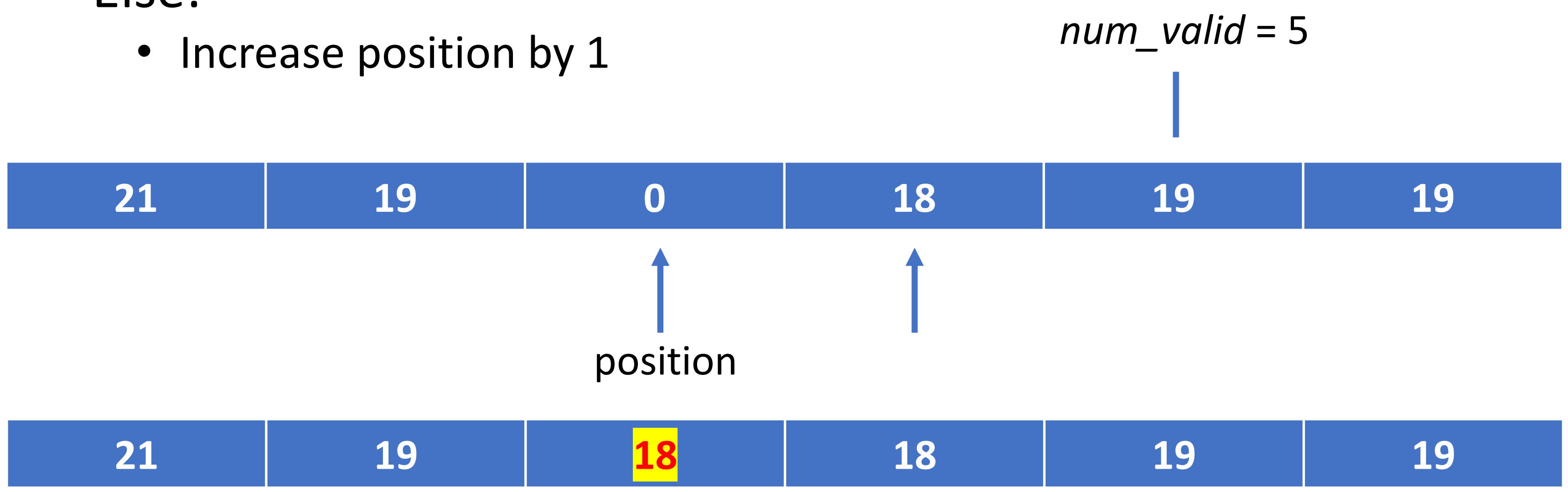


Shuffle-left algorithm

- While *position* <= *num_valid* :
  - If *num*[*position*] is invalid, e.g., 0 :
    - All valid numbers to the right of *num* are shifted 1 position to the left
    - Decrease *num_valid* by 1
  - Else:
    - Increase position by 1

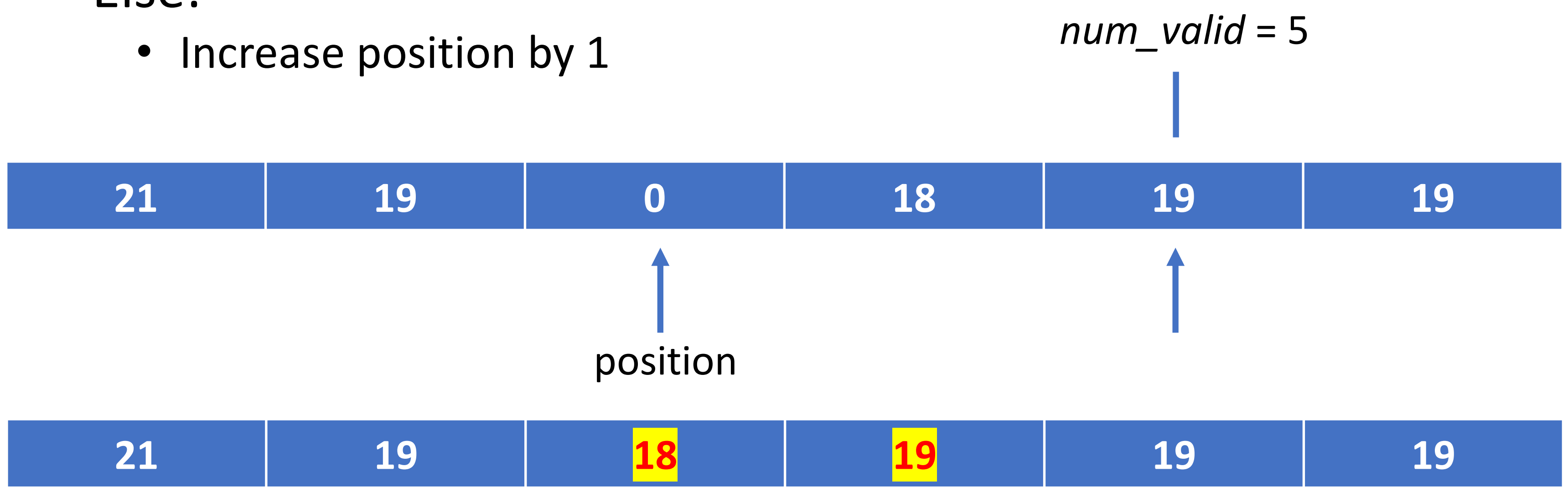
Shuffle-left algorithm

- While *position* <= *num_valid* :
  - If *num*[*position*] is invalid, e.g., 0 :
    - All valid numbers to the right of *num* are shifted 1 position to the left
    - Decrease *num_valid* by 1
  - Else:
    - Increase position by 1



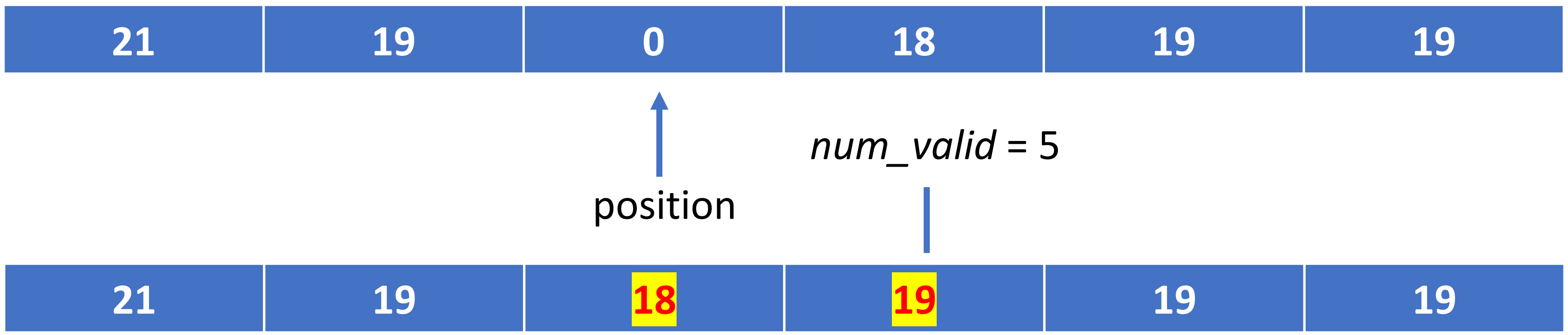
Shuffle-left algorithm

- While *position* <= *num_valid* :
  - If *num*[*position*] is invalid, e.g., 0 :
    - All valid numbers to the right of *num* are shifted 1 position to the left
    - Decrease *num_valid* by 1
  - Else:
    - Increase position by 1



Shuffle-left algorithm

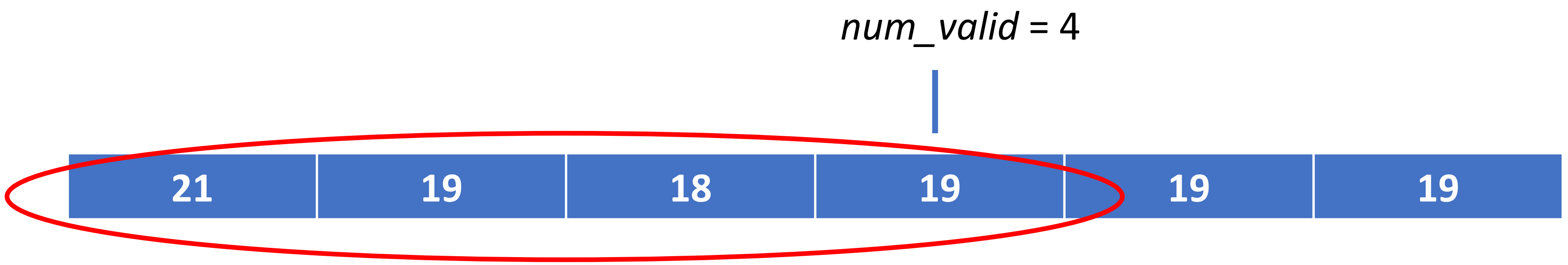
- While *position* <= *num_valid* :
  - If *num*[*position*] is invalid, e.g., 0 :
    - All valid numbers to the right of *num* are shifted 1 position to the left
    - Decrease *num_valid* by 1
  - Else:
    - Increase position by 1



Shuffle-left algorithm

- While *position* <= *num_valid* :
    - If *num*[*position*] is invalid, e.g., 0 :
        - All valid numbers to the right of *num* are shifted 1 position to the left
        - Decrease *num_valid* by 1
    - Else:
        - Increase position by 1

- The final list, containing 4 valid items, is below:



Shuffle-left algorithm:

- While *position* <= *num_valid* :
  - If *num*[*position*] is invalid, e.g., 0 :
    - All valid numbers to the right of *num* are shifted 1 position to the left
    - Decrease *num_valid* by 1
  - Else:
    - Increase position by 1
- Running time (best case)
  - If *no* numbers are invalid, then the *while* loop is executed *n* times, where *n* is the initial size of the list, and the only other operations are the comparison in the *if* statement, and *position* is increased by 1. The running time is $\theta(n)$. This is the best case.

Shuffle-left algorithm:

- While *position* <= *num_valid* :
  - If *num*[*position*] is invalid, e.g., 0 :
    - All valid numbers to the right of *num* are shifted 1 position to the left
    - Decrease *num_valid* by 1
  - Else:
    - Increase position by 1
- Running time (worst case):
  - If *all* the numbers are invalid, then for all *n* passes through the list, n - 1 copies (shifts) are made. This is a worst case.
  - The total number of operations in the loop is (ignoring comparisons):
    - For the first position: n + 1 operations: n – 1 copies, plus 2 to increase *num_valid* and *position*
    - For the second position: n operations, n – 2 copies, plus 2 to increase *num_valid* and *position*
  - The total number of operations is the sum of 1 through n + 1 which equals
    - n(n+1)/2 + 1 → $\theta(n^2)$

Shuffle-left algorithm:

- While *position* <= *num_valid* :
  - If *num*[*position*] is invalid, e.g., 0 :
    - All <u>valid</u> numbers to the right of *num* are shifted 1 position to the left
    - Decrease *num_valid* by 1
  - Else:
    - Increase position by 1
- Running time:
  - Best case (all entries are valid) is $\theta(n)$
  - Worst case (all entries are invalid) is $\theta(n^2)$
  - Average case is also $\theta(n^2)$
- Space:
  - *n* (all cases – best, worst, and average) (*n* is required for the original list, plus a few additional variables)

Copy-over algorithm

- Find the total number of valid elements in the list, and store in *num_valid*
- Create an empty list, called *copyNum*, of length *num_valid*
- Set *index* to 0
- For each *num* in the original list:
  - If *num* is valid
    - Assign *num* to *copyNum[index]*
    - Increase *index* by 1

0	**21**	**19**	**0**	**18**	**19**

Copy-over algorithm

- Find the total number of valid elements in the list, and store in *num_valid*
- Create an empty list, called *copyNum*, of length *num_valid*
- Set *index* to 0
- For each *num* in the original list:
  - If *num* is valid
    - Assign *num* to *copyNum[index]*
    - Increase *index* by 1

0	**21**	**19**	**0**	**18**	**19**

-	**-**	**-**	**-**

num_valid = 4

Copy-over algorithm

- Find the total number of valid elements in the list, and store in *num_valid*
- Create an empty list, called *copyNum*, of length *num_valid*
- Set *index* to 0
- For each *num* in the original list:
  - If *num* is valid
    - Assign *num* to *copyNum[index]*
    - Increase *index* by 1

0	**21**	**19**	**0**	**18**	**19**

X

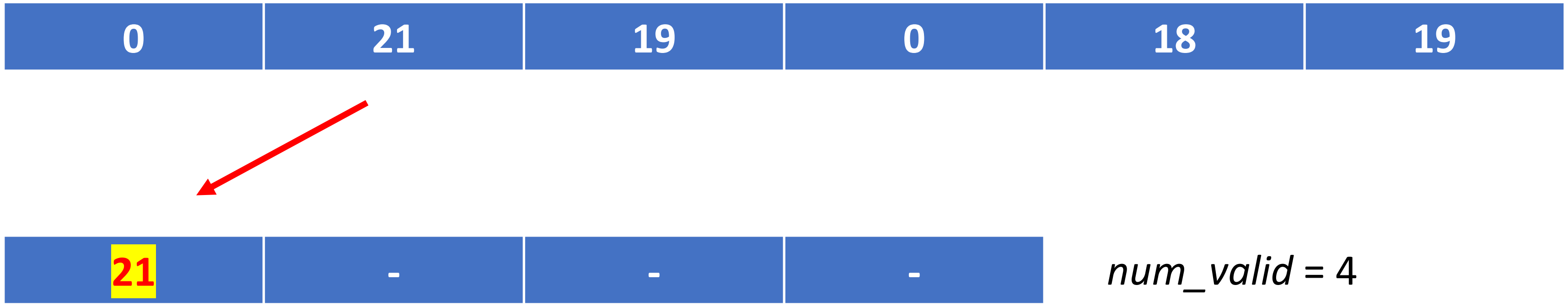
-	**-**	**-**	**-**

num_valid = 4

Index = 0

Copy-over algorithm

- Find the total number of valid elements in the list, and store in *num_valid*
- Create an empty list, called *copyNum*, of length *num_valid*
- Set *index* to 0
- For each *num* in the original list:
  - If *num* is valid
    - Assign *num* to *copyNum[index]*
    - Increase *index* by 1

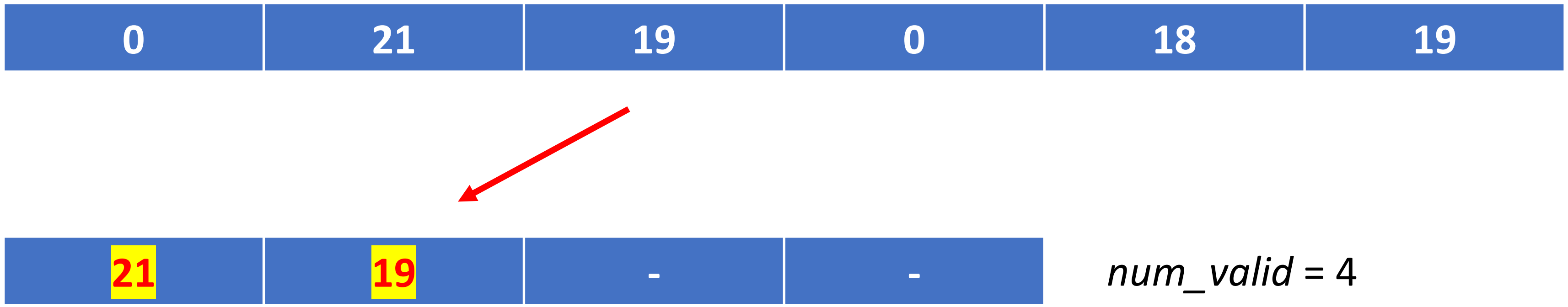


num_valid = 4

Index = 1

Copy-over algorithm

- Find the total number of valid elements in the list, and store in *num_valid*
- Create an empty list, called *copyNum*, of length *num_valid*
- Set *index* to 0
- For each *num* in the original list:
  - If *num* is valid
    - Assign *num* to *copyNum[index]*
    - Increase *index* by 1



num_valid = 4

Index = 2

Copy-over algorithm

- Find the total number of valid elements in the list, and store in *num_valid*
- Create an empty list, called *copyNum*, of length *num_valid*
- Set *index* to 0
- For each *num* in the original list:
  - If *num* is valid
    - Assign *num* to *copyNum[index]*
    - Increase *index* by 1

0	**21**	**19**	**0**	**18**	**19**
			X		

21	**19**	**-**	**-**

num_valid = 4

Index = 2

Copy-over algorithm

- Find the total number of valid elements in the list, and store in *num_valid*
- Create an empty list, called *copyNum*, of length *num_valid*
- Set *index* to 0
- For each *num* in the original list:
  - If *num* is valid
    - Assign *num* to *copyNum[index]*
    - Increase *index* by 1

0	**21**	**19**	**0**	**18**	**19**

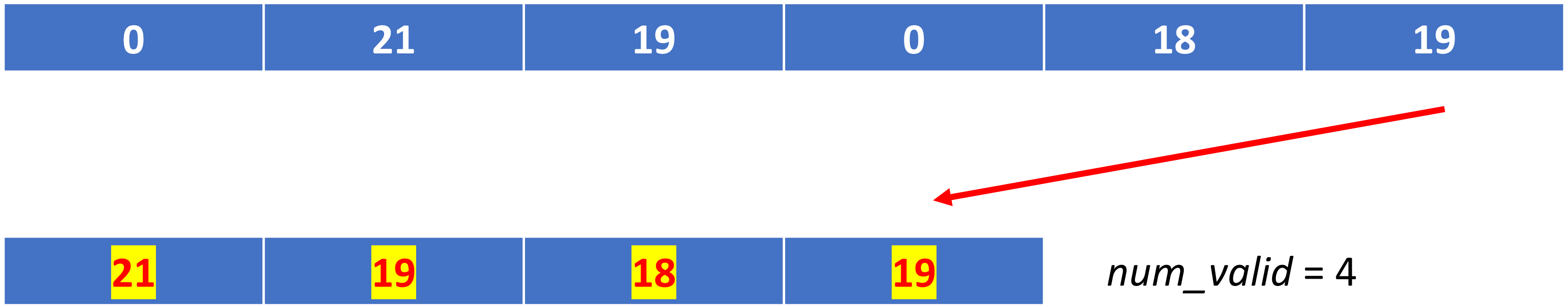
21	**19**	**18**	**-**

num_valid = 4

Index = 3

Copy-over algorithm

- Find the total number of valid elements in the list, and store in *num_valid*
- Create an empty list, called *copyNum*, of length *num_valid*
- Set *index* to 0
- For each *num* in the original list:
    - If *num* is valid
        - Assign *num* to *copyNum[index]*
        - Increase *index* by 1



num_valid = 4

Index = 4

Copy-over algorithm

- Find the total number of valid elements in the list, and store in *num_valid*
- Create an empty list, called *copyNum*, of length *num_valid*
- Set *index* to 0
- For each *num* in the original list:
  - If *num* is valid
    - Assign *num* to *copyNum[index]*
    - Increase *index* by 1

- Running time:
  - The first step is order *n*, since we need to iterate through all elements in the list to count the number of valid elements. For each element, there is a constant number of operations. (More details for this step are required, but this likely would use a *for* loop).
  - The main work then occurs in the *for* loop on the 4th line, which is also order *n*. For each element, we either copy it or not, and this is also a constant number of operations for each of the *n* elements.
  - The running time is $\theta(n)$, in the best, worst, and average cases.

Copy-over algorithm

- Find the total number of valid elements in the list, and store in *num_valid*
- Create an empty list, called *copyNum*, of length *num_valid*
- Set *index* to 0
- For each *num* in the original list:
  - If *num* is valid
    - Assign *num* to *copyNum[index]*
    - Increase *index* by 1
- Space (depends on the number of valid elements):
  - Best case: if there are *no* valid elements, then the space only requires the original list, which is *n* (we ignore a few additional variables)
  - Worst case: if *all* the elements are valid, we create an additional copy of the original list. The space requirements are 2*n.*
  - Average case: this depends on the expected number of valid/invalid items, and will be between *n* and 2*n*. If the number of valid items is equally likely to be between 0, 1, 2, …*n*, then the average space requirement is 1.5*n*.

Converging pointers algorithm

- We keep a *left* and *right* index
  - Set *left* to 0 and *right* to *n – 1* (index of the last element)
- Set *num_valid* to the length of the *numbers* list
- While *left* < *right*
  - If *number[left]* is valid :
    - Increase *left* by 1
  - Else (number[left] is not valid) :
    - Copy number[right] to number[left]
    - Decrease *num_valid* by 1
    - Decrease *right* by 1
- If number[left] is not valid : ← Correction: this is *after* the *while* loop.
  - Decrease *num_valid* by 1

Converging pointers example

num_valid = 6

Item at *left* is 0, so we copy from *right* to *left*, and decrease *right* and *num_valid by 1.*

Converging pointers example

num_valid = 5

Item at *left* is not 0, so we increase *left* by 1

Converging pointers example

num_valid = 5

Item at *left* is not 0, so we increase *left* by 1

Converging pointers example

num_valid = 5

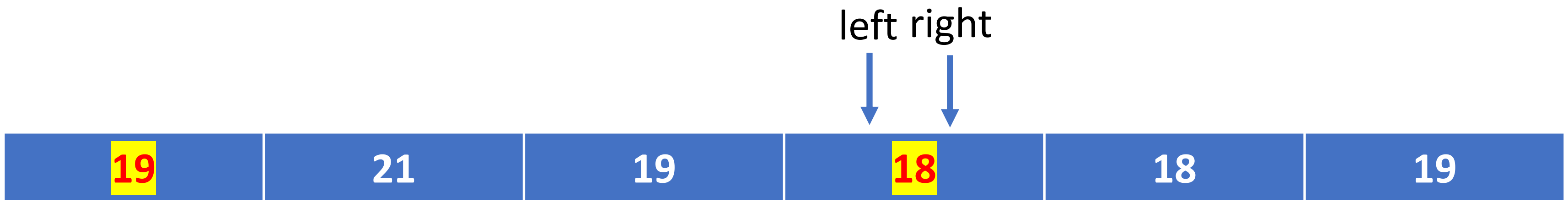
Item at *left* is not 0, so we increase *left* by 1

Converging pointers example

num_valid = 5

Item at *left* is 0, so we copy from *right* to *left*, and decrease *right* and *num_valid by 1.*

Converging pointers example



num_valid = 4

Item at *left* is not 0 (if it was, we would decrease *num_valid*).

Once *left* is equal to *right*, we are done

Converging pointers algorithm

- While *left* < *right*
  - If *number[left]* is valid :
    - Increase *left* by 1
  - Else (number[left] is not valid) :
    - Copy number[right] to number[left]
    - Decrease *num_valid* by 1
    - Decrease *right* by 1
- If number[left] is not valid :
    - Decrease *num_valid* by 1

← Correction: this is *after* the *while* loop.

- Running time:
  - The main work occurs in the *while* loop. The loop always increases *left* or decreases right, until *left* and *right* are the same. This can only happen *n* times. All other operations inside the loop are constant, so the running time is $\theta(n)$, which is true for the best, worst, and average cases.
- Space: *n* (we need space only for the original list, as well as a few additional terms). This is the most space efficient algorithm

Data Cleanup Algorithms

	Shuffle-left		Copy over		Converging Pointers	
	Time	Space	Time	Space	Time	Space
Best	$\theta(n)$	n	$\theta(n)$	n	$\theta(n)$	n
Worst	$\theta(n^2)$	n	$\theta(n)$	$2n$	$\theta(n)$	n
Average	$\theta(n^2)$	n	$\theta(n)$	$(n, 2n)$	$\theta(n)$	n

- Which algorithm is the *best*?