WIA2005 Algorithm Design and Analysis

Lecture 3: Sorting Algorithm

Dr. Asmiza A. Sani

Semester 2, Session 2024/25

Learning objectives

- The student will know the following algorithm:
  - Bubble sort
  - Counting sort
  - Radix sort
  - Bucket sort
  - Shell sort

Sorting Algorithms

- Sorting refers to the arranging of data in a particular format.
- Sorting algorithms contains a set of instruction to arrange data into a particular order.
- Input data may be stored in an array or list.
- Common sort operation is performed in numerical or lexicographical order.

Why sorting?

Many computer scientists consider sorting to be the most fundamental problem in the study of algorithms for the following reasons:

- Sometimes an application inherently needs to sort information.
  - Optimised searching of data.
- Sorting allows data to be presented in a more readable format.

Classification of Sorting Algorithm

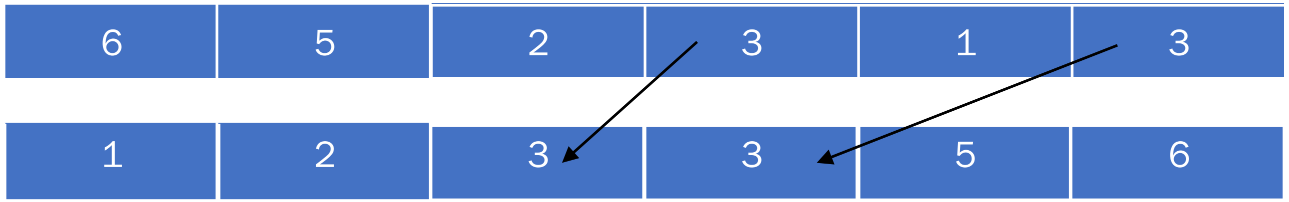
- Before we investigate sorting algorithms, there are a few classifications of sorting that we need to know:
  - In-place vs. Not-In-Place Sorting
  - Stable vs. Not Stable Sorting
  - Adaptive vs. Non-Adaptive Sorting
  - Online vs. Offline
- Other important terms related to sorting:
  - Increasing order vs. Non- decreasing order
  - Decreasing order vs. Non-increasing order

In-place vs. Not-In-Place Sorting

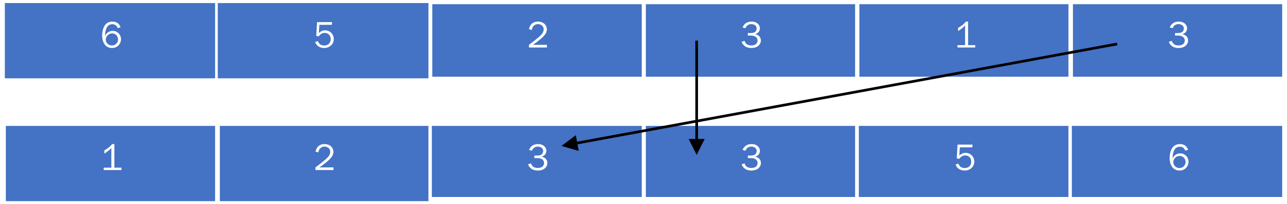
- Some sorting algorithms may require additional space for comparing or temporary storage of data.
  - This is called not-in-place sorting.
  - Merge sort is an example of a not-in-place sorting algorithm.
- The sorting algorithm that arranges without any additional storage (e.g., within the array itself) is called in-place sorting.
  - Insertion sort is an example of an in-place sorting.

Stable vs. Not Stable Sorting

- A stable sorting algorithm preserve the sequence of input after the sort has been perform.



- If the sequence of input is not preserved after sorting, it is called the not-stable sorting algorithm.



- Stability matters when we want to maintain the sequence of original input, such as tuples.

Adaptive vs. Non-Adaptive Sorting

- An adaptive sorting algorithm can take advantage of a pre-sorted input, while non-adaptive ones do not.
- Usually, an adaptive algorithm is an improvement of an existing sorting algorithm.

- Example of adaptive sorting: Insertion Sort

Online vs. Offline

- An online sorting algorithm can process as the data is being fed and does not require the whole input to be available initially.
  - An example of an online algorithm is insertion sort.
- Conversely, an offline sorting algorithm needs all input to be available before it can start processing the output.
  - An example of an offline algorithm is the selection sort.

Increasing order vs. Non- decreasing order

- Increasing order is any sequence where the element is greater than the previous element.

1	2	3	4	5	6

- Non-decreasing order is any sequence where the element is greater than or equal to the previous element.

1	2	3	3	5	6

Decreasing order vs. Non-increasing order

- Decreasing order is any sequence where the element is smaller than the previous element.

6	5	4	3	2	1

- Non-increasing order is any sequence where the element is smaller than or equal to the previous element.

6	5	3	3	2	1

How do you choose which sorting algorithm to use?

- To choose the most appropriate sorting algorithm, you must know what the input data is going to look like:
  - The size of input?
  - Nearly sorted/Random/Reversed/Duplicate?
  - Best worst-case?
  - Good average-case?
  - The universe (range) of input?

Sorting algorithms

Classify the Sorting Algorithm

Algorithm	In-place?	Stable?	Adaptive?	Online?
Bubble sort				
Counting sort				
Radix Sort				
Bucket sort				
Shell sort				

Bubble sort

- Bubble sort is a popular but inefficient sorting algorithm. It works by repeatedly swapping adjacent elements that are out of order.

```
BUBBLESORT(A)
for i = 1 to A.length − 1
    for j = A.length downto i + 1
        if A[j] < A[j − 1]
            exchange A[j] with A[j − 1]
```

Bubble Sort Example

5	1	12	-5	16	unsorted
5	1	12	-5	16	5 > 1, swap
1	5	12	-5	16	5 < 12, ok
1	5	12	-5	16	12 > -5, swap
1	5	-5	12	16	12 < 16, ok
1	5	-5	12	16	1 < 5, ok
1	5	-5	12	16	5 > -5, swap
1	-5	5	12	16	5 < 12, ok
1	-5	5	12	16	1 > -5, swap
-5	1	5	12	16	1 < 5, ok
-5	1	5	12	16	-5 < 1, ok
-5	1	5	12	16	sorted

Time complexity..

- What is the time complexity for Bubble sort?

Time complexity..

- What is the time complexity for Bubble sort?

$$T(n) = \Theta(n^2)$$

Counting sort

- *The counting sort algorithm sorts* elements based on numeric keys between a specific range.
- No comparison is done during sorting.
- Used as a subroutine in another sorting algorithm.

```
COUNTING-SORT(A, B, k)
let C[0..k] be a new array
for i = 0 to k
    C[i] = 0
for j = 1 to A.length
    C[A[j]] = C[A[j]] + 1
// C[i] now contains the number of elements equal to i.
for i = 1 to k
    C[i] = C[i] + C[i − 1]
// C[i] now contains the number of elements less than or equal to i.
for j = A.length downto 1
    B[C[A[j]]] = A[j]
    C[A[j]] = C[A[j]] − 1
```

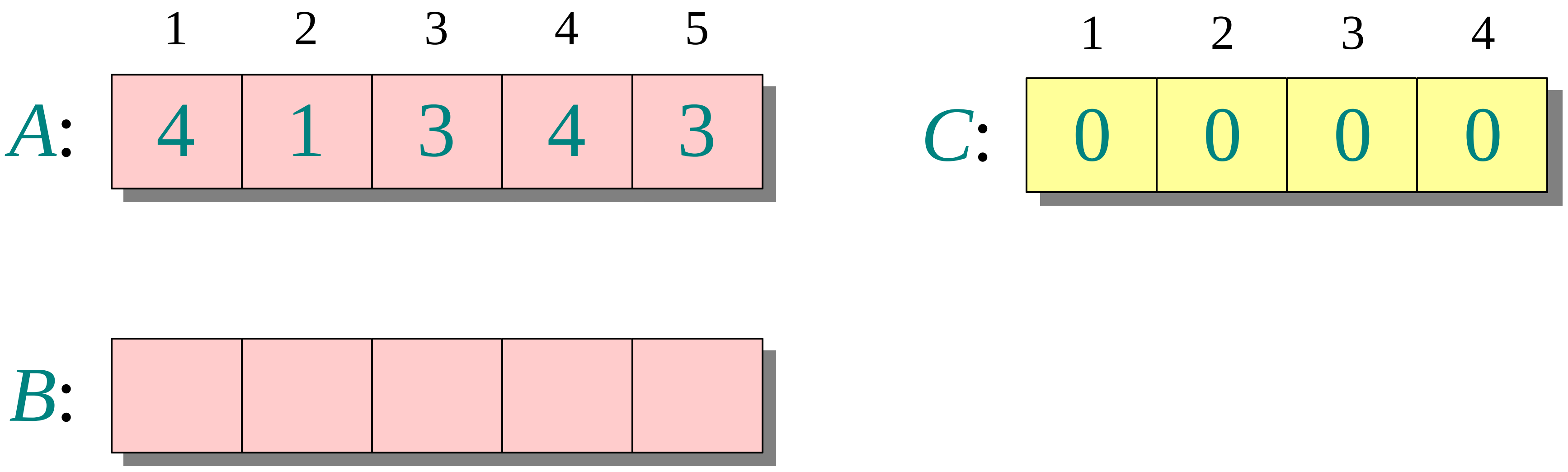
Counting-sort example

	1	2	3	4	5
A:	4	1	3	4	3

	1	2	3	4
C:				

B:					

Loop 1: initialization




```
let C[0..k] be a new array
for i = 0 to k
    C[i] = 0
```

Loop 2: count

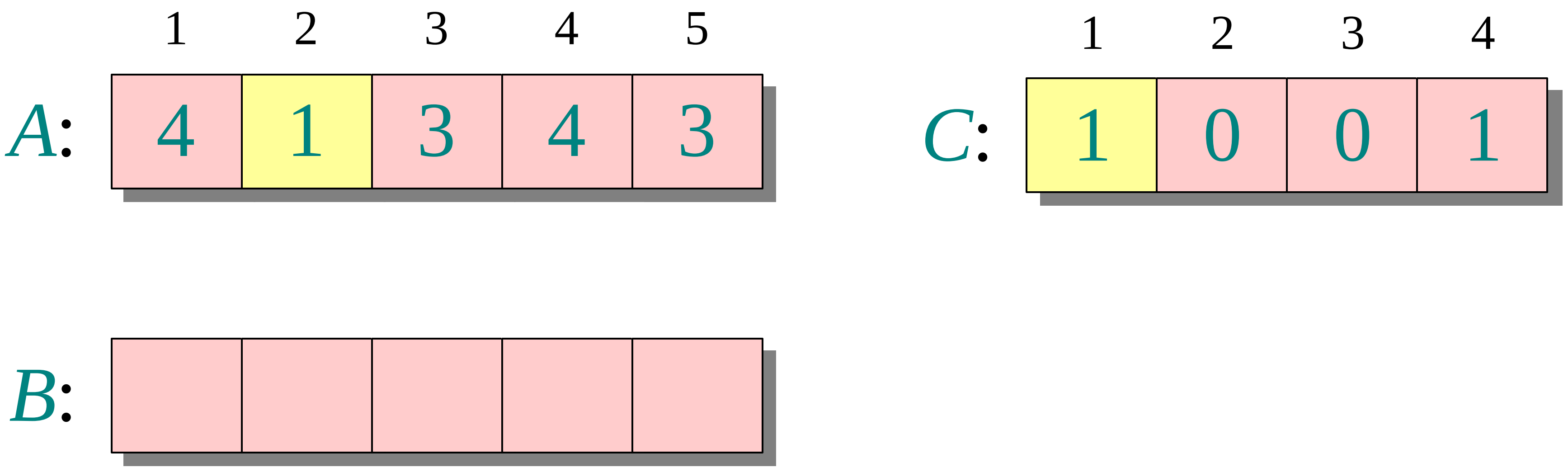
	1	2	3	4	5
A:	4	1	3	4	3

	1	2	3	4
C:	0	0	0	1

B:					

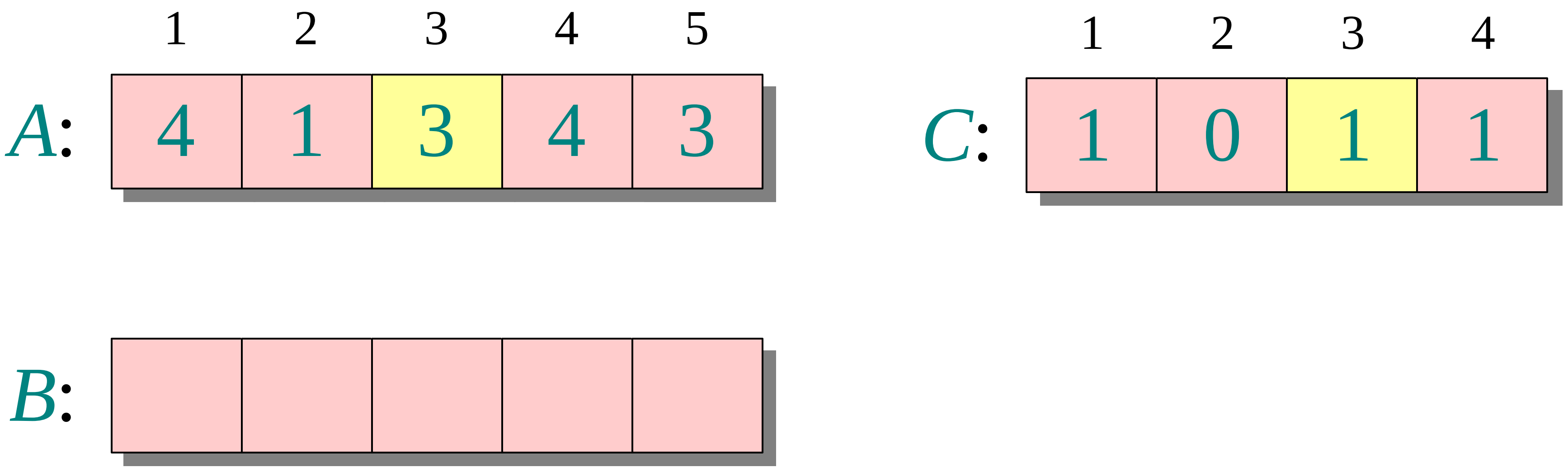
```
for j = 1 to A.length
    C[A[j]] = C[A[j]] + 1
```

Loop 2: count



for $j = 1$ **to** $A.length$
$\quad C[A[j]] = C[A[j]] + 1$

Loop 2: count




```
for j = 1 to A.length
    C[A[j]] = C[A[j]] + 1
```

Loop 2: count

	1	2	3	4	5
A:	4	1	3	4	3

	1	2	3	4
C:	1	0	1	2

B:					

```
for j = 1 to A.length
    C[A[j]] = C[A[j]] + 1
```

Loop 2: count

	1	2	3	4	5
A:	4	1	3	4	3
B:					

	1	2	3	4
C:	1	0	2	2

```
for j = 1 to A.length
    C[A[j]] = C[A[j]] + 1
```

Loop 3: compute running sum

	1	2	3	4	5
A:	4	1	3	4	3
B:					

	1	2	3	4
C:	1	0	2	2
C':	1	1	2	2

```
// C[i] now contains the number of elements equal to i.
for i = 1 to k
    C[i] = C[i] + C[i - 1]
```

Loop 3: compute running sum

	1	2	3	4	5
A:	4	1	3	4	3
B:					

	1	2	3	4
C:	1	0	2	2
C':	1	1	3	2

```
// C[i] now contains the number of elements equal to i.
for i = 1 to k
    C[i] = C[i] + C[i – 1]
```

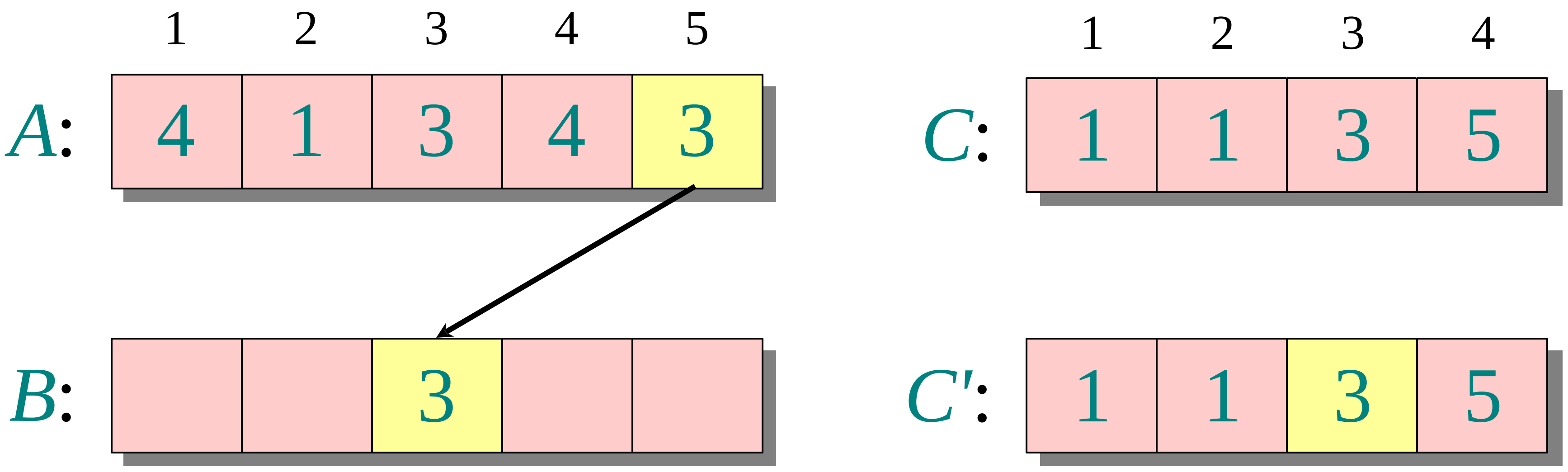
Loop 3: compute running sum

	1	2	3	4	5
A:	4	1	3	4	3
B:					

	1	2	3	4
C:	1	0	2	2
C':	1	1	3	5

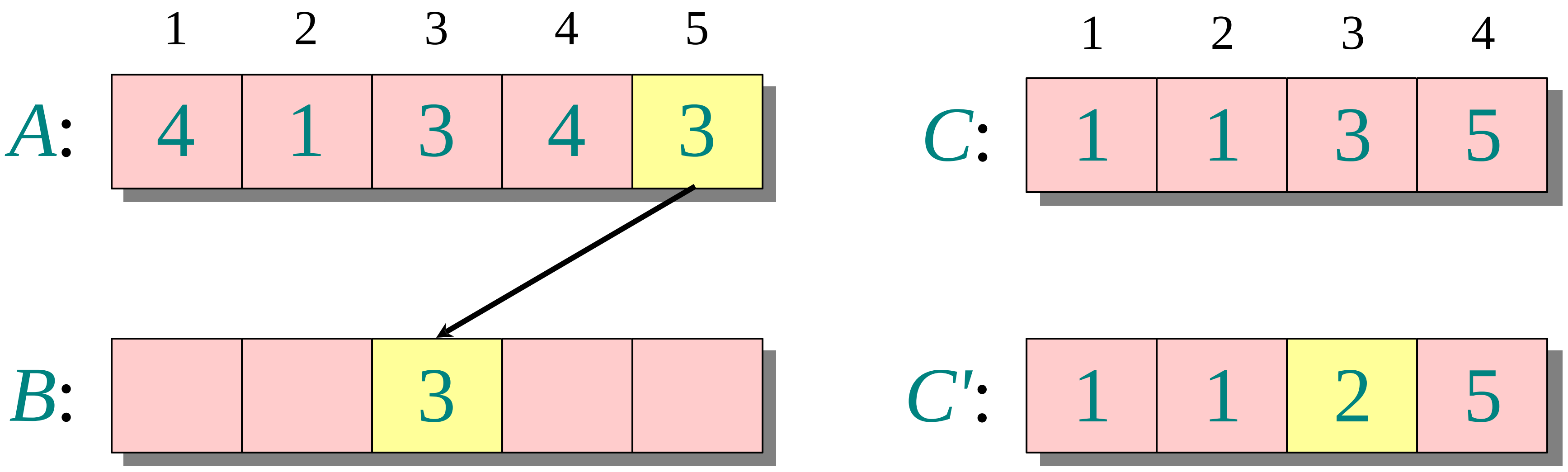
```
// C[i] now contains the number of elements equal to i.
for i = 1 to k
    C[i] = C[i] + C[i - 1]
```

Loop 4: re-arrange



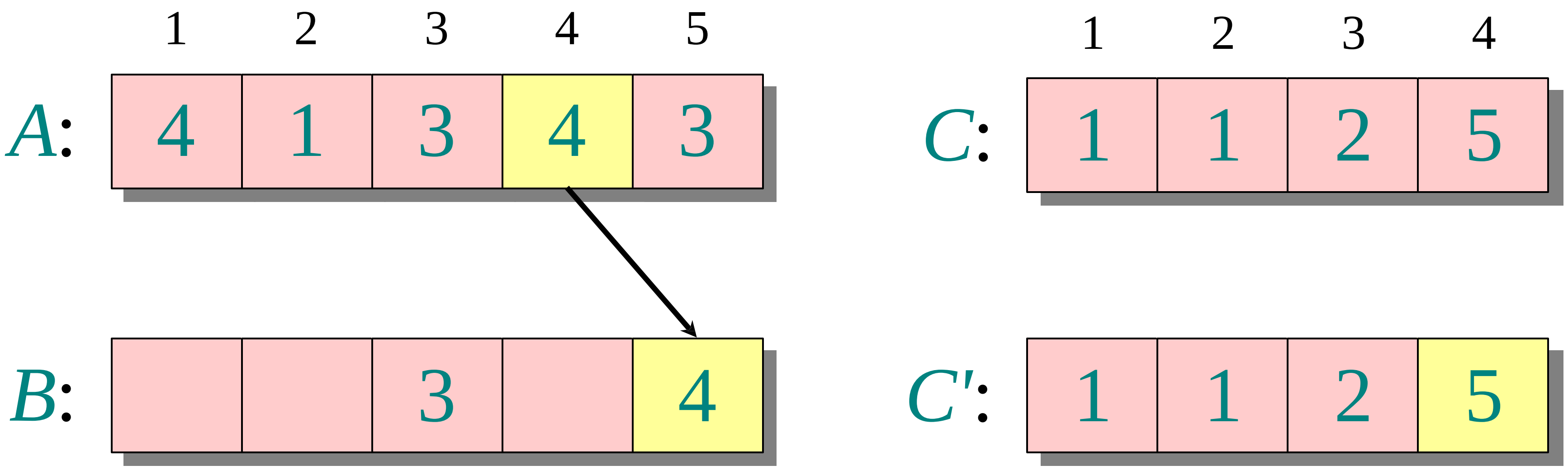

```
// C[i] now contains the number of elements less than or equal to i.
for j = A.length downto 1
    B[C[A[j]]] = A[j]
    C[A[j]] = C[A[j]] - 1
```

Loop 4: re-arrange



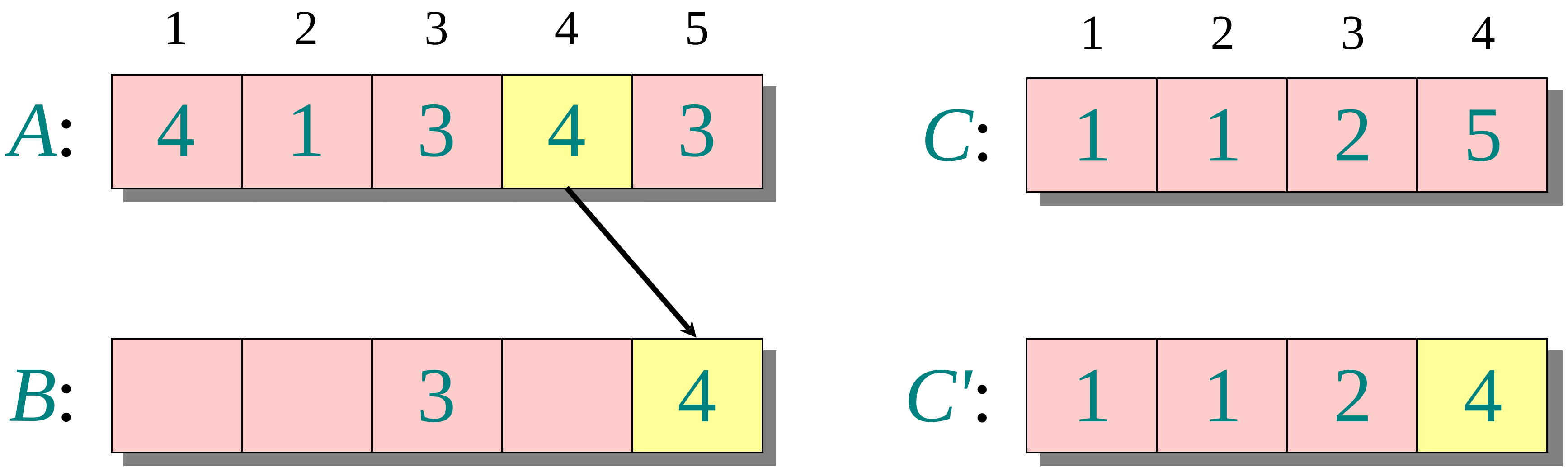

```
// C[i] now contains the number of elements less than or equal to i.
for j = A.length downto 1
    B[C[A[j]]] = A[j]
    C[A[j]] = C[A[j]] − 1
```

Loop 4: re-arrange



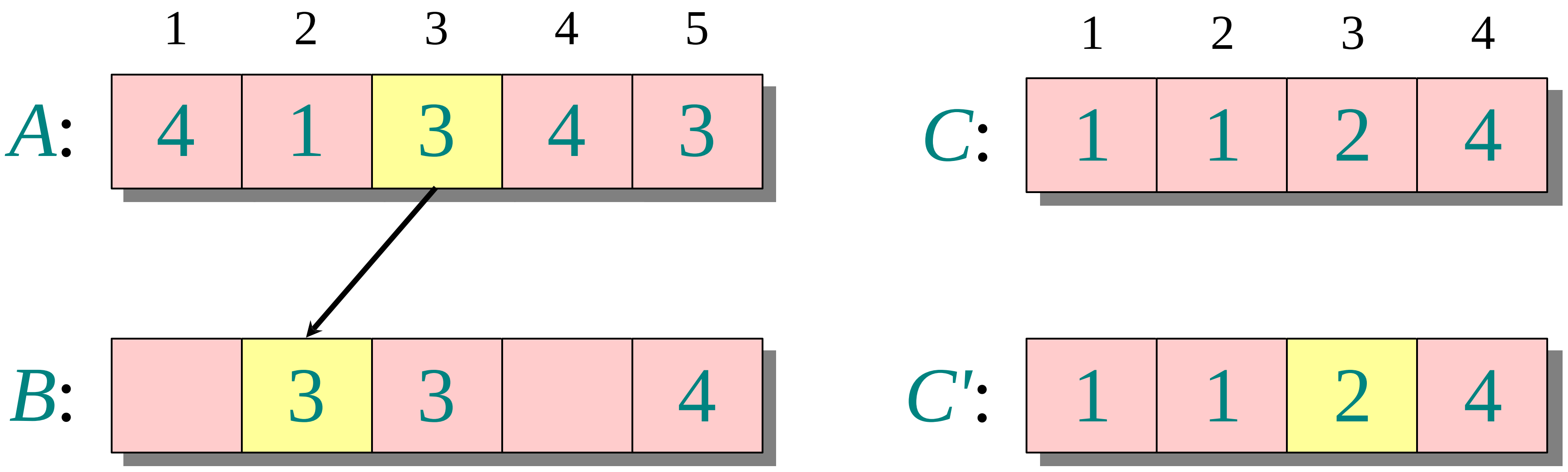

```
// C[i] now contains the number of elements less than or equal to i.
for j = A.length downto 1
    B[C[A[j]]] = A[j]
    C[A[j]] = C[A[j]] - 1
```

Loop 4: re-arrange



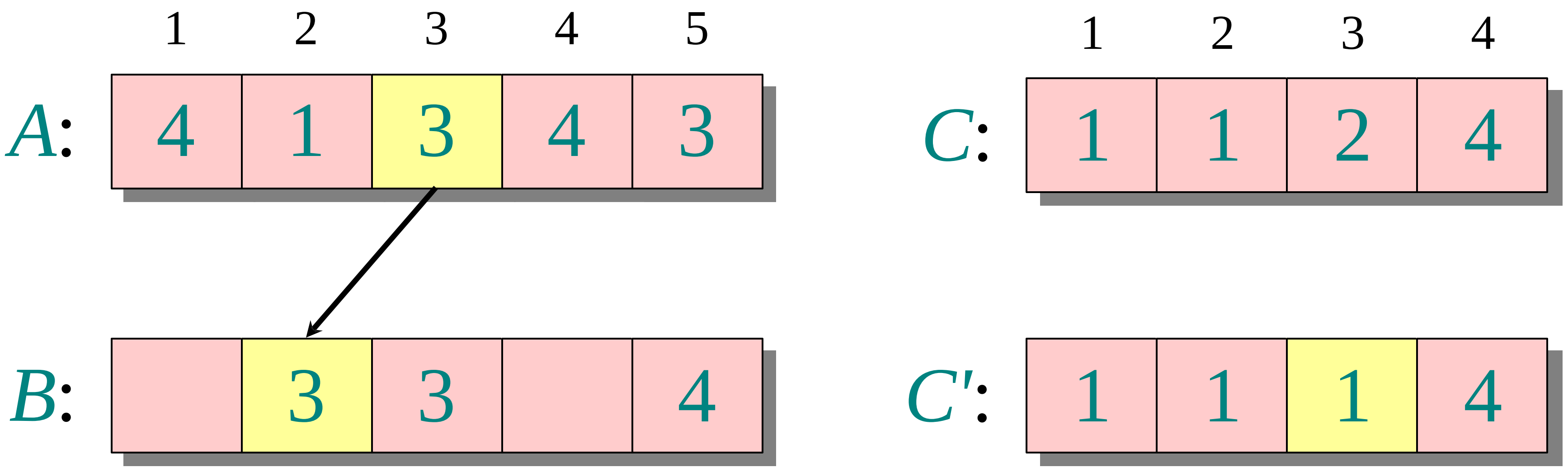

```
// C[i] now contains the number of elements less than or equal to i.
for j = A.length downto 1
    B[C[A[j]]] = A[j]
    C[A[j]] = C[A[j]] - 1
```

Loop 4: re-arrange



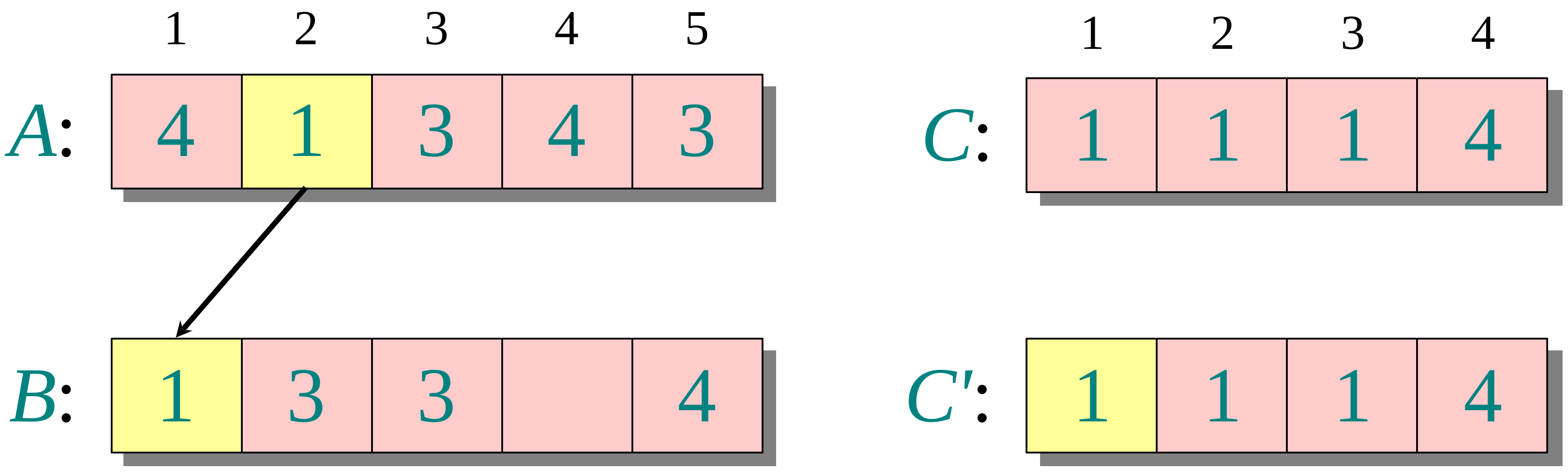

```
// C[i] now contains the number of elements less than or equal to i.
for j = A.length downto 1
    B[C[A[j]]] = A[j]
    C[A[j]] = C[A[j]] - 1
```

Loop 4: re-arrange



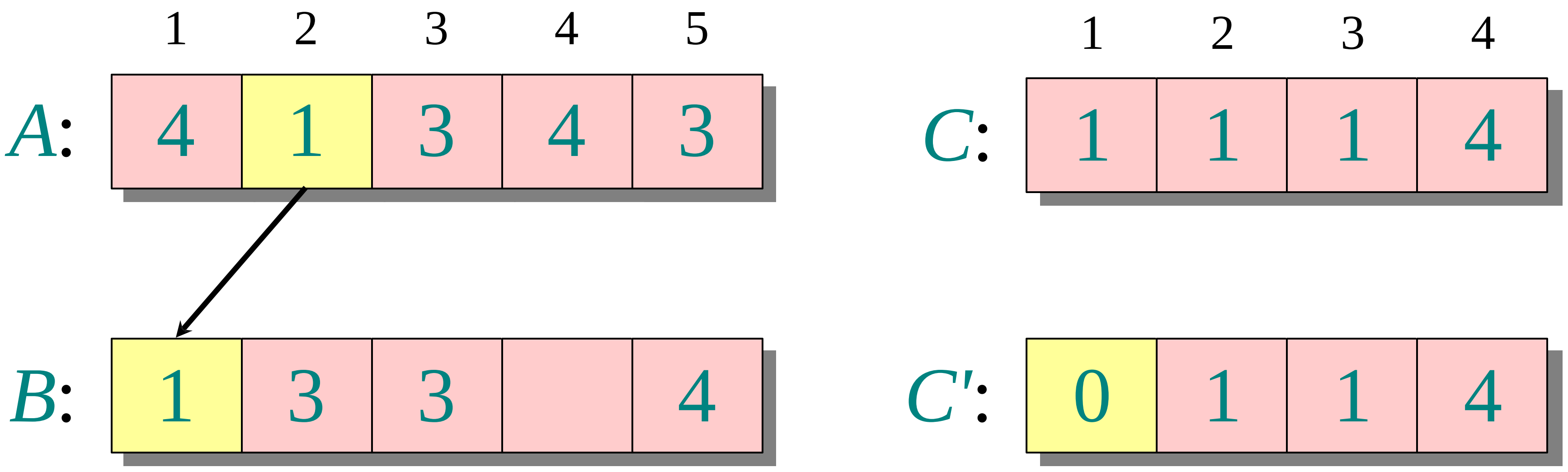
// $C[i]$ now contains the number of elements less than or equal to i.
for $j = A.length$ **downto** 1
    $B[C[A[j]]] = A[j]$
    $C[A[j]] = C[A[j]] - 1$

Loop 4: re-arrange



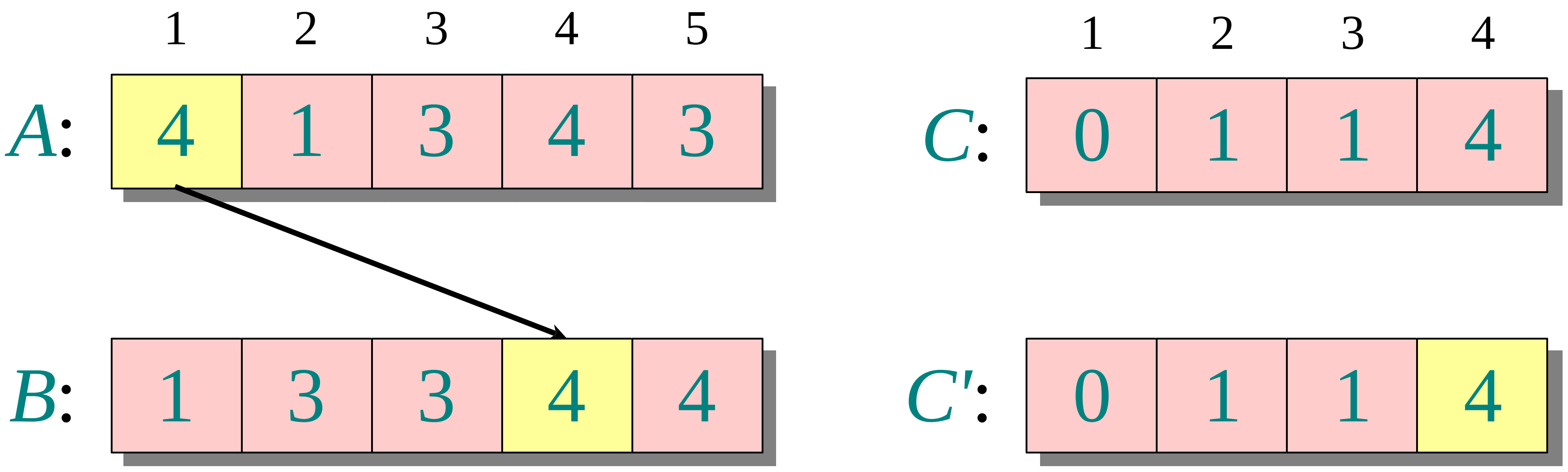

```
// C[i] now contains the number of elements less than or equal to i.
for j = A.length downto 1
    B[C[A[j]]] = A[j]
    C[A[j]] = C[A[j]] - 1
```

Loop 4: re-arrange



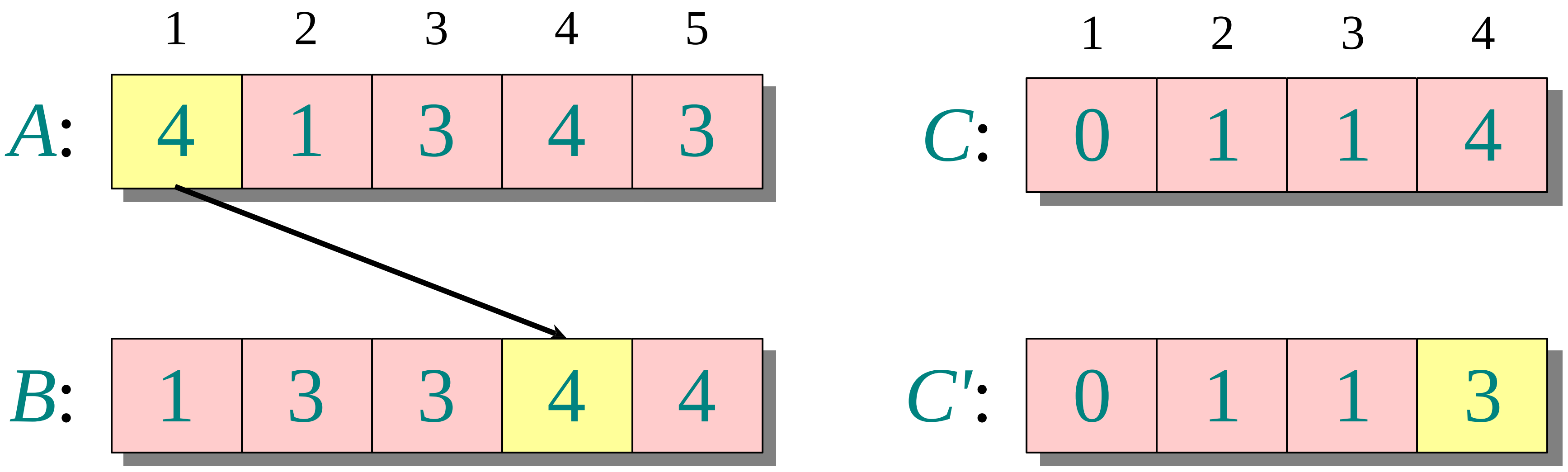

```
// C[i] now contains the number of elements less than or equal to i.
for j = A.length downto 1
    B[C[A[j]]] = A[j]
    C[A[j]] = C[A[j]] - 1
```

Loop 4: re-arrange



// $C[i]$ now contains the number of elements less than or equal to i.
for $j = A.length$ **downto** 1
    $B[C[A[j]]] = A[j]$
    $C[A[j]] = C[A[j]] - 1$

Loop 4: re-arrange




```
// C[i] now contains the number of elements less than or equal to i.
for j = A.length downto 1
    B[C[A[j]]] = A[j]
    C[A[j]] = C[A[j]] - 1
```

Time complexity..

- What is the time complexity for Counting sort?

```
COUNTING-SORT(A, B, k)
let C[0..k] be a new array
for i = 0 to k
    C[i] = 0
for j = 1 to A.length
    C[A[j]] = C[A[j]] + 1
// C[i] now contains the number of elements equal to i.
for i = 1 to k
    C[i] = C[i] + C[i - 1]
// C[i] now contains the number of elements less than or equal to i.
for j = A.length downto 1
    B[C[A[j]]] = A[j]
    C[A[j]] = C[A[j]] - 1
```

Time complexity..

- What is the time complexity for Counting sort?

```
COUNTING-SORT(A, B, k)
let C[0..k] be a new array
for i = 0 to k
    C[i] = 0
for j = 1 to A.length
    C[A[j]] = C[A[j]] + 1
// C[i] now contains the number of elements equal to i.
for i = 1 to k
    C[i] = C[i] + C[i − 1]
// C[i] now contains the number of elements less than or equal to i.
for j = A.length downto 1
    B[C[A[j]]] = A[j]
    C[A[j]] = C[A[j]] − 1
```

$T(n) = \Theta(n+k)$

Radix sort

- Radix sort is a non-comparative integer sorting algorithm that sorts data with integer keys by grouping keys by the individual digits which share the same significant position and value.
- It does this by using counting sort (but not limited to this – any stable sorting method can be used to do this) to sort the n integers by digits, starting from the least significant digit (i.e., one's digit for integers) to the most significant digit.

RADIX-SORT(A, d)

for $i = 1$ **to** d
    use a stable sort to sort array A on digit i

Radix sort

- Treat each digit as a key.
- Start from the least significant bit.
- Group the keys based on the digit while keeping the original order.

Most significant $\downarrow$ Least significant $\downarrow$

19809910912351818359 9
34019954038012811529 5
38470010159453961469 6
38240836020103925853 8
61438650762868132893 6

Radix Sort Example (LSD)

*Do not change the order!

- Original list (d = 3) :

171	46	76	491	803	4	26	67

  - Sorting 1st least significant digit:

171	491	803	004	046	076	026	067

  - Sorting 2nd least significant digit:

803	004	026	046	067	171	076	091

  - Sorting 3rd least significant digit:

004	026	046	067	076	091	171	803

Time complexity..

- What is the time complexity for Radix sort?

```
RADIX-SORT(A, d)
1  for i = 1 to d
2      use a stable sort to sort array A on digit i
```

Time complexity..

- What is the time complexity for Radix sort?

```
RADIX-SORT(A, d)
for i = 1 to d
    use a stable sort to sort array A on digit i
```

$$T(n) = \Theta\ (d(n+k))$$

Bucket sort

- *Bucket sort* algorithm creates buckets and put elements into them.
- Then using some sorting algorithm (e.g., Insertion sort) to sort elements in each bucket.
- Then the elements are taken out and joined to get the sorted result.

```
BUCKET-SORT(A)
let B[0..n − 1] be a new array
n = A.length
for i = 0 to n − 1
    make B[i] an empty list
for i = 1 to n
    insert A[i] into list B[⌊nA[i]⌋]
for i = 0 to n − 1
    sort list B[i] with insertion sort
concatenate the lists B[0], B[1], ..., B[n − 1] together in order
```

Bucket sort

- Assumption: uniform distribution
  - Input numbers are uniformly distributed in [0,1).
  - Suppose the input size is n.
- Idea:
  - Divide [0,1) into n equal-sized subintervals (buckets).
  - Distribute n numbers into buckets
  - Expect that each bucket contains a few numbers.
  - Sort numbers in each bucket (insertion sort as default).
  - Then, go through buckets in order, listing elements.

Bucket Sort Example

- Sort:

28	24	4	48	8	36	22	43

Step 1: Insert data in bucket accordingly.

Bucket (0-9)	4, 8
Bucket (10-19)	
Bucket (20-29)	28, 24, 22
Bucket (30-39)	36
Bucket (40-49)	48, 43

Step 2: Sort data in bucket.

Bucket (0-9)	4, 8
Bucket (10-19)	
Bucket (20-29)	22, 24, 28
Bucket (30-39)	36
Bucket (40-49)	43, 48

Step 3: Merge data in all bucket sequentially.

4	8	22	24	28	36	43	48

Example of BUCKET-SORT

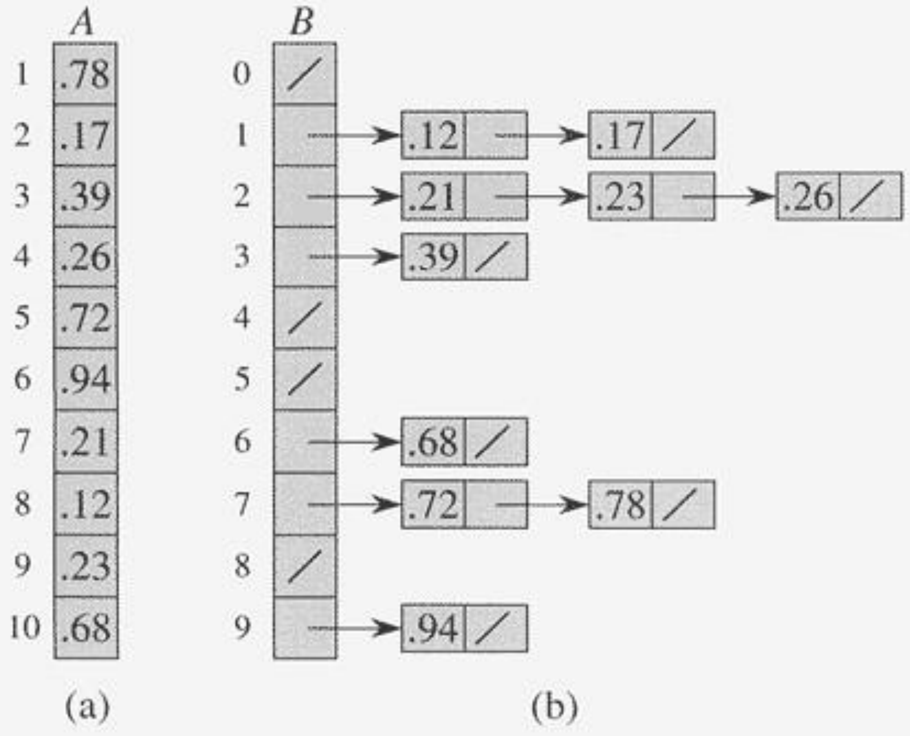


Figure 8.4 The operation of BUCKET-SORT. **(a)** The input array $A[1..10]$. **(b)** The array $B[0..9]$ of sorted lists (buckets) after line 5 of the algorithm. Bucket i holds values in the half-open interval $[i/10, (i+1)/10)$. The sorted output consists of a concatenation in order of the lists $B[0], B[1], \ldots, B[9]$.

Time complexity..

- What is the time complexity of Bucket sort?

```
BUCKET-SORT(A)
let B[0..n − 1] be a new array
n = A.length
for i = 0 to n − 1
    make B[i] an empty list
for i = 1 to n
    insert A[i] into list B[⌊n A[i]⌋]
for i = 0 to n − 1
    sort list B[i] with insertion sort
concatenate the lists B[0], B[1], ..., B[n − 1] together in order
```

Time complexity..

- What is the time complexity of Bucket sort?

```
BUCKET-SORT(A)
let B[0..n − 1] be a new array
n = A.length
for i = 0 to n − 1
    make B[i] an empty list
for i = 1 to n
    insert A[i] into list B[⌊n A[i]⌋]
for i = 0 to n − 1
    sort list B[i] with insertion sort
concatenate the lists B[0], B[1], ..., B[n − 1] together in order
```

$$T(n) = \Theta(n)$$

Shell sort

- A generalisation of the Insertion sort
  - sorting by comparing elements that are distant apart rather than adjacent.
- If we start comparing N elements that are at a certain distance apart
  - value gap < N.
- The value of the gap is reduced in each pass until the last pass, where gap = 1.
- In the last pass, the sort is like an insertion sort.

Algorithm:

```
gaps = [x, y, z, …]
for each (G in gaps)
        create sub-arrays with elements having gap as G
        for each(S in sub-arrays)
                insertion_sort(s)
        merge all sub-arrays
```

Shell Sort Example

Sort:

18	32	12	5	38	33	16	2

8 Numbers to be sorted, Shell's increment will be floor(n/2)

Step 1: When gap floor(8/2) = 4

1	2	3	4	1	2	3	4
18	32	12	5	38	33	16	2

18	32	12	2	38	33	16	5

* Do insertion sort operation with all elements with the same colour

Shell Sort Example

Step 2: When gap floor(4/2) = 2

1	2	1	2	1	2	1	2
18	32	12	2	38	33	16	5

12	2	16	5	18	32	38	33

* Do insertion sort operation with all elements with the same colour

Shell Sort Example

Step 3: When gap floor(2/2) = 1

1	1	1	1	1	1	1	1
12	2	16	5	18	32	38	33

2	5	12	16	18	32	33	38

* Do insertion sort operation with all elements with the same colour

Time complexity..

- What is the time complexity of Shell sort?

Algorithm:

```
gaps = [x, y, z, …]
for each (G in gaps)
        create sub-arrays with elements having gap as G
        for each(S in sub-arrays)
                insertion_sort(s)
        merge all sub-arrays
```

Time complexity..

- What is the time complexity of Shell sort?

```
Algorithm:
gaps = [x, y, z, ...]
for each (G in gaps)
        create sub-arrays with elements having gap as G
        for each(S in sub-arrays)
                insertion_sort(s)
        merge all sub-arrays
```

Several variants, ranging from slightly worse than $\Theta(n \log n)$ to $\Theta(n^2)$ – Depending on the gap sequence.

Time complexity..

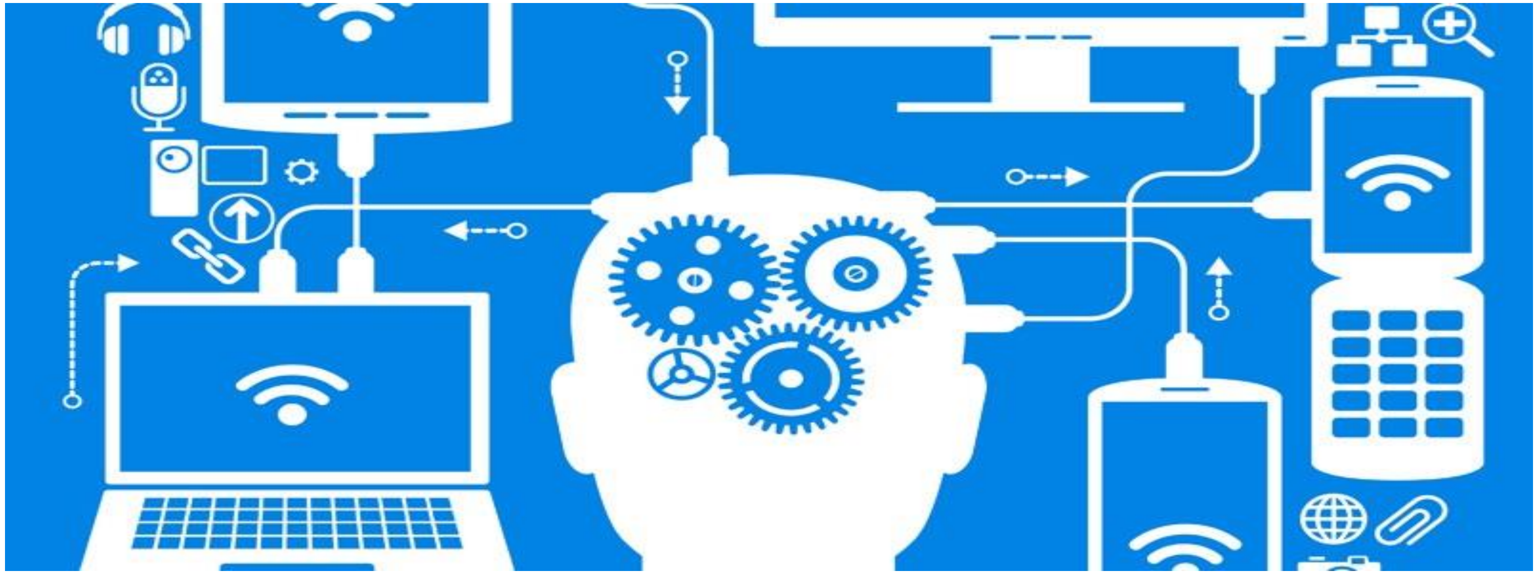
OEIS	General term ($k \geq 1$)	Concrete gaps	Worst-case time complexity	Author and year of publication
	$\left\lfloor \frac{N}{2^k} \right\rfloor$	$\left\lfloor \frac{N}{2} \right\rfloor, \left\lfloor \frac{N}{4} \right\rfloor, \ldots, 1$	$\Theta\left(N^2\right)$ [e.g. when $N = 2^p$]	Shell, 1959[4]
	$2\left\lfloor \frac{N}{2^{k+1}} \right\rfloor + 1$	$2\left\lfloor \frac{N}{4} \right\rfloor + 1, \ldots, 3, 1$	$\Theta\left(N^{\frac{3}{2}}\right)$	Frank & Lazarus, 1960[8]
A000225	$2^k - 1$	$1, 3, 7, 15, 31, 63, \ldots$	$\Theta\left(N^{\frac{3}{2}}\right)$	Hibbard, 1963[9]
A083318	$2^k + 1$, prefixed with 1	$1, 3, 5, 9, 17, 33, 65, \ldots$	$\Theta\left(N^{\frac{3}{2}}\right)$	Papernov & Stasevich, 1965[10]
A003586	Successive numbers of the form $2^p 3^q$ (3-smooth numbers)	$1, 2, 3, 4, 6, 8, 9, 12, \ldots$	$\Theta\left(N \log^2 N\right)$	Pratt, 1971[1]
A003462	$\frac{3^k - 1}{2}$, not greater than $\left\lceil \frac{N}{3} \right\rceil$	$1, 4, 13, 40, 121, \ldots$	$\Theta\left(N^{\frac{3}{2}}\right)$	Knuth, 1973[3], based on Pratt, 1971[1]
A036569	$\prod_I a_q$, where $a_q = \min\left\{ n \in \mathbb{N} : n \geq \left(\frac{5}{2}\right)^{q+1}, \forall p : 0 \leq p < q \Rightarrow \gcd(a_p, n) = 1 \right\}$ $I = \left\{ 0 \leq q < r \mid q \neq \frac{1}{2}\left(r^2 + r\right) - k \right\}$ $r = \left\lfloor \sqrt{2k + \sqrt{2k}} \right\rfloor$	$1, 3, 7, 21, 48, 112, \ldots$	$O\left(N^{1+\sqrt{\frac{8 \ln(5/2)}{\ln(N)}}}\right)$	Incerpi & Sedgewick, 1985,[11] Knuth[3]
A036562	$4^k + 3 \cdot 2^{k-1} + 1$, prefixed with 1	$1, 8, 23, 77, 281, \ldots$	$O\left(N^{\frac{4}{3}}\right)$	Sedgewick, 1982[6]
A033622	$\begin{cases} 9\left(2^k - 2^{\frac{k}{2}}\right) + 1 & k \text{ even}, \\ 8 \cdot 2^k - 6 \cdot 2^{(k+1)/2} + 1 & k \text{ odd} \end{cases}$	$1, 5, 19, 41, 109, \ldots$	$O\left(N^{\frac{4}{3}}\right)$	Sedgewick, 1986[12]
	$h_k = \max\left\{ \left\lfloor \frac{5h_{k-1}}{11} \right\rfloor, 1 \right\}, h_0 = N$	$\left\lfloor \frac{5N}{11} \right\rfloor, \left\lfloor \frac{5}{11} \left\lfloor \frac{5N}{11} \right\rfloor \right\rfloor, \ldots, 1$	Unknown	Gonnet & Baeza-Yates, 1991[13]
A108870	$\left\lceil \frac{1}{5}\left(9 \cdot \left(\frac{9}{4}\right)^{k-1} - 4\right) \right\rceil$	$1, 4, 9, 20, 46, 103, \ldots$	Unknown	Tokuda, 1992[14]
A102549	Unknown (experimentally derived)	$1, 4, 10, 23, 57, 132, 301, 701$	Unknown	Ciura, 2001[15]

Wikipedia

Classify the Sorting Algorithm

Algorithm	In-place?	Stable?	Adaptive?	Online?
Bubble sort				
Counting sort				
Radix Sort				
Bucket sort				
Shell sort				

In the next lecture..



Lecture 4: String Matching Algorithm

References

- Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest and Clifford Stein. 2009. Introduction to Algorithms, 3rd edition. MIT Press.
- Robert Sedgewick and Kevin Wayne. 2011. Algorithm. 5th Edition. Addison- Wesley.