CIS 313:
Intermediate Data Structure

eighth slide

theme

- Sorting
- Order statistics

merge sort

1. break list of n elements into two halves (time O(1))
2. recursively sort each half (time T(n/2)+T(n/2))
3. merge the two sorted lists (time O(n))

- total time T(n) = 2T(n/2) + O(n)
- this can be shown to be T(n) = O(n *lg*n)
- master method (powerful) can solve recurrence relations

The Master Method

more general and complicated version in text

If $T(n) \leq aT\left(\frac{n}{b}\right) + O(n^d)$

$a > 0,\ b > 1,\ d \geq 0$

then

$$T(n) = \begin{cases} O(n^d \log n) & \text{if } a = b^d \quad \text{(Case 1)} \\ O(n^d) & \text{if } a < b^d \quad \text{(Case 2)} \\ O(n^{\log_b a}) & \text{if } a > b^d \quad \text{(Case 3)} \end{cases}$$

quicksort: ”dual” version of merge sort

1. ”unmerge” the array
    1. use Partition method
    2. break into “halves” of small elements and large elements
    3. of course, they may not be same size, just hope so
2. sort each side recursively
3. put the two sides together (no work involved here)

- Partition is O(n)
- total is O(n^2) worst case
- but O(n *lg*n) on average

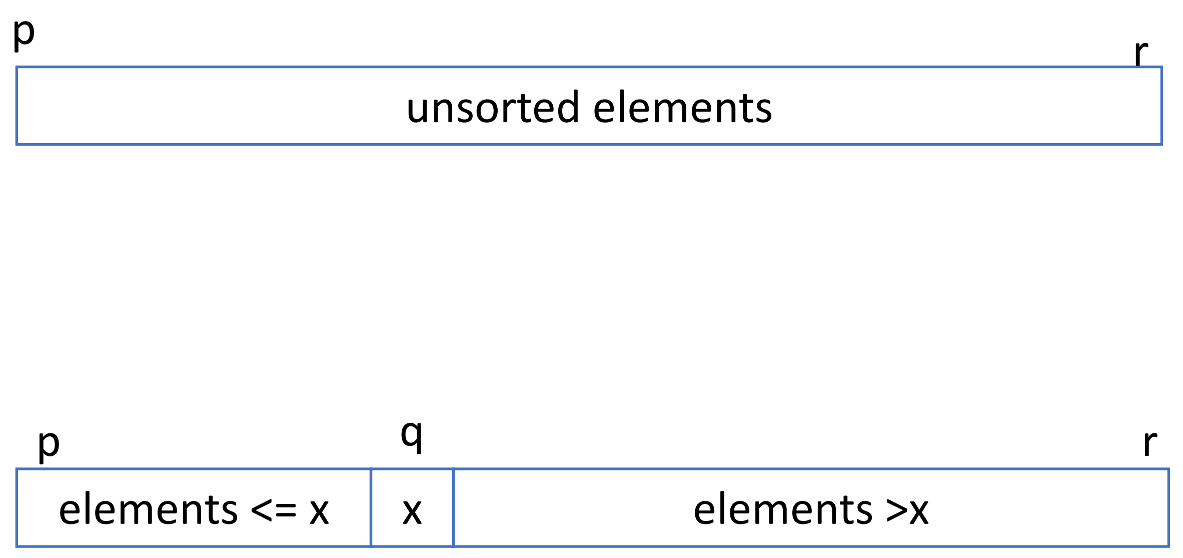
Partition: the key method

```
PARTITION($array\ A, int\ p, int\ r$)
$x \leftarrow A[r]$                          ▷ Choose **pivot**
$i \leftarrow p - 1$
**for** $j \leftarrow p$ **to** $r - 1$
        **do if** $(A[j] \leq x)$
                **then** $i \leftarrow i + 1$
                        exchange $A[i] \leftrightarrow A[j]$
exchange $A[i+1] \leftrightarrow A[r]$
**return** $i + 1$
```

Partition: observation

- in place operation
  - merge requires workspace
  - then need to copy items back into array
- single pass
- can choose other elements for pivot
  - median of first, last, middle elements
  - choose random location in [p,r], use element there as pivot
    - that’s Randomized-Partition
    - highly recommended

what partition does



the sizes of the left and right sides depend on the value of x

quicksort easy with partition

```
QuickSort(p, r)

if p<r then
     q = Partition(p,r)
     QuickSort(p,q-1)
     QuickSort(q+1,r)
```

quicksort worst case

- suppose partition uses time cn
- let T(n) be the time for quicksort on n elements
- in the worst case, the pivot element is always the largest (or smallest) remaining element
- so $T(n) = cn + T(n-1)$
- expanding $T(n) = cn + T(n-1)$

  $= cn + c(n-1) + T(n-2)$

  $= cn + c(n-1) + c(n-2) + T(n-3) \ldots$

  $= c(n + (n-1) + (n-2) + \ldots + 1) + T(0)$

  $= cn(n+1)/2 + T(0)$

  $= O(n^2)$

quicksort average case

- in the call to Partition, suppose that each returned value of q is equally likely
- there are n possibilities: q = 1, 2, 3, …, n
- the recursive calls to QuickSort have sizes q-1 and n-q
- so the average case time is $T(n) = cn + \frac{1}{n}\sum_{q=1}^{n}(T(q-1) + T(n-q))$
- note that in the sum, each of T(0), T(1), …, T(n-1) appear twice, so we can rewrite it as
- $T(n) = cn + \frac{1}{n}\sum_{k=0}^{n-1} 2T(k) = cn + \frac{2}{n}(T(0) + T(1) + \cdots + T(n-1))$

average case (2)

- rewrite
- $n \cdot T(n) = cn^2 + 2(T(0) + T(1) + \cdots + T(n-1))$
- substitute n-1 for n
- $(n-1) \cdot T(n-1) = c(n-1)^2 + 2(T(0) + T(1) + \cdots + T(n-2))$
- subtract one from the other, and notice that T(0),…,T(n-2) cancel
- $nT(n) - (n-1)T(n-1) = cn^2 - c(n-1)^2 + 2T(n-1)$
- simplify, collect terms
- $nT(n) = c(2n-1) + (n+1)T(n-1) \leq 2cn + (n+1)T(n-1)$

average case (3)

- grind away, use ”standard” tricks
- $T(n) \leq 2c + \frac{(n+1)}{n} T(n-1)$
- $\frac{T(n)}{n+1} \leq \frac{2c}{n+1} + \frac{T(n-1)}{n}$
- let $f(n) = \frac{T(n)}{n+1}$ and rewrite the previous line as
- $f(n) \leq \frac{2c}{n+1} + f(n-1)$
- expand (again)
- $f(n) \leq \frac{2c}{n+1} + f(n-1) = \frac{2c}{n+1} + \frac{2c}{n} + f(n-2) \leq \frac{2c}{n+1} + \frac{2c}{n} + \frac{2c}{n-1} + f(n-3) \leq \ \ldots$

average case (4)

- … $f(n) \leq 2\text{c}\left(\frac{1}{n+1} + \frac{1}{n} + \frac{1}{n-1} + \cdots + \frac{1}{2}\right) + f(0) = 2c(H_{n+1} - 1) + f(0) = 2c \cdot H_{n+1} + \theta(1)$
- recall the Harmonic series $H_n = \sum_{i=1}^{n} \frac{1}{i} = \ln n + \theta(1)$
- so $f(n) = 2c \cdot \ln(n+1) + \theta(1) \cong 2.885c \cdot \log_2(n+1) + \theta(1)$
- remembering that $f(n) = \frac{T(n)}{n+1}$ we get
- $\frac{T(n)}{n+1} = \theta(\lg(n+1))$ or $T(n) = \theta((n \lg(n+1))$
- DONE!!

lower bounds

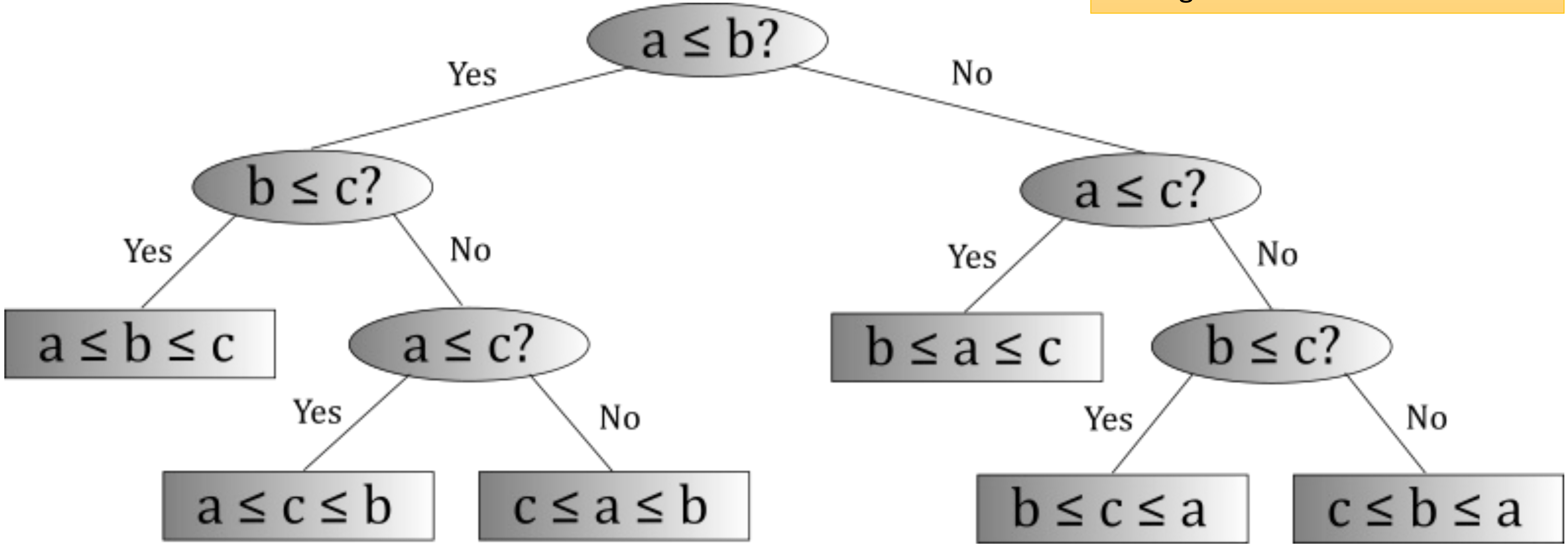
- O(n*lg*n) seems like a common time bound for sorting
- there is a reason for this
- look at the *comparison-based* model
- access to data items are only through comparisons of two items
  - is $a \leq b$?
- the more common sorts are comparison based: merge-sort, heap-sort, quick-sort, bubble-sort, etc.
- counting sort is O(n) times (sort of), but it is not comparison-based
  - it uses data item as array index

comparison-based sorts require n*lg*n time

- the $\Omega(n \cdot lgn)$ lower bound is on the number of comparisons required to sort n items
- use the *decision tree* model
- a decision tree is a full binary tree
    - internal node represents a comparison between two items
    - left/right branches indicate yes/no outcomes
    - external nodes are the outcomes
- any comparison based sorting algorithm on n elements corresponds to a (BIG!!) decision tree
- height of the tree is the worst case number of comparisons

decision tree example

this tree has 3!=6 external nodes, one for each possible outcome of sorting 3 elements



observations about decision trees

- any decision tree for sorting n items must have at least n! external nodes (outcomes): #outcomes $\geq n!$
- a decision tree of (internal) height h has at most 2^h external nodes
  - height h means h+1 comparisons
- combining: $n! \leq \#\text{outcomes} \leq 2^h$
- take logs of both sides: $\lg n! \leq \lg 2^h = h$
- Stirling's Approx: $\lg n! = \lg \sqrt{2\pi n} \cdot \left(\frac{n}{e}\right)^n \cdot (1 + \Theta\left(\frac{1}{n}\right)) = \Theta(n \lg n)$

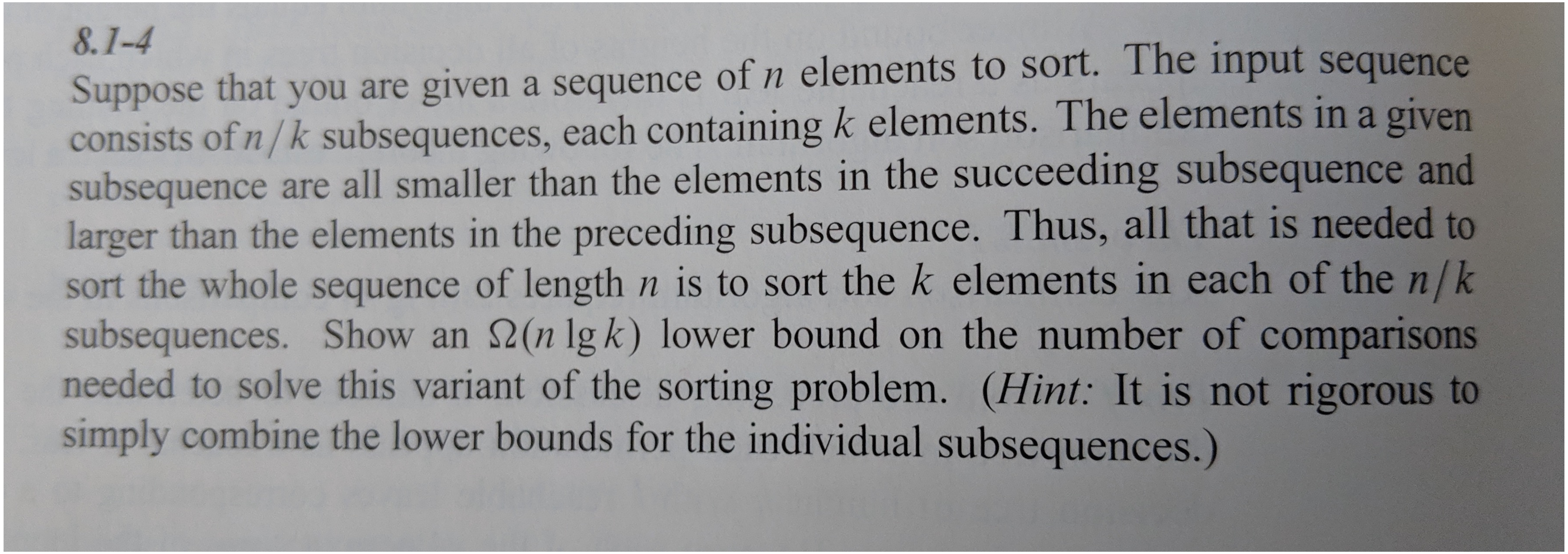
aside: avoid Stirling's Approx

- $n! = n(n-1)(n-2)\cdots 2\cdot 1 \geq n(n-1)\cdots(\frac{n}{2}+1)(\frac{n}{2}) \geq (\frac{n}{2})^{\frac{n}{2}}$
- so from previous page…
- $h = \lg 2^h \ \geq \lg n! \geq \lg(\frac{n}{2})^{\frac{n}{2}} \ = (\frac{n}{2})(\lg n - 1)$
- conclude: $h = \Omega(n \lg n)$

lower bound: conclusion

- theorem 8.1: Any comparison sort algorithm requires $\Omega(n \cdot lgn)$ comparisons in the worst case
- this general technique is called an *information theoretic* lower bound
- general idea: find number of outcomes, take logarithm

- application: merge 2 sorted lists of n elements, requires n comparisons exercise 8.1-4 from text



8.1-4

Suppose that you are given a sequence of n elements to sort. The input sequence consists of n/k subsequences, each containing k elements. The elements in a given subsequence are all smaller than the elements in the succeeding subsequence and larger than the elements in the preceding subsequence. Thus, all that is needed to sort the whole sequence of length n is to sort the k elements in each of the n/k subsequences. Show an $\Omega(n \lg k)$ lower bound on the number of comparisons needed to solve this variant of the sorting problem. (*Hint:* It is not rigorous to simply combine the lower bounds for the individual subsequences.)

linear time sorts

- Counting sort, Radix sort
- for Counting sort, we sort n elements, each in the range 0 to k (k fixed)
  - sometimes k=n
  - use element as array index
- simple version: count the number of items with value i, for $0 \leq i \leq k$
- use i as index to an array C
- then for each i print C[i] copies of i
- problem: i may be a key to larger element, need associated info

```
COUNTING-SORT$(A, B, k)$
1  let $C[0..k]$ be a new array
2  for $i = 0$ to $k$
3      $C[i] = 0$
4  for $j = 1$ to $A.length$
5      $C[A[j]] = C[A[j]] + 1$
6  // $C[i]$ now contains the number of elements equal to $i$.
7  for $i = 1$ to $k$
8      $C[i] = C[i] + C[i-1]$
9  // $C[i]$ now contains the number of elements less than or equal to $i$.
10 for $j = A.length$ downto 1
11     $B[C[A[j]]] = A[j]$
12     $C[A[j]] = C[A[j]] - 1$
```

example run: n=8, k=5

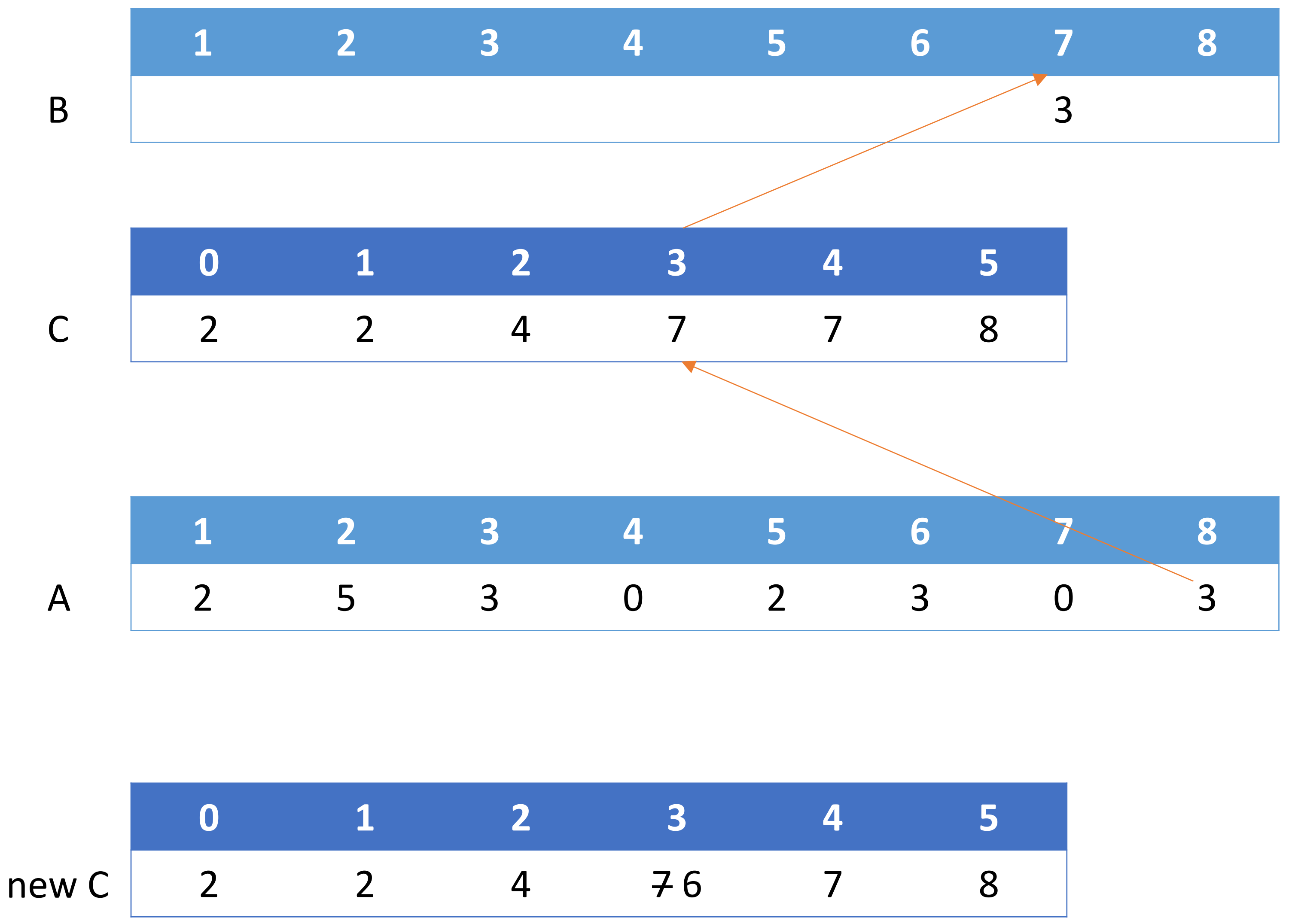
	1	2	3	4	5	6	7	8
A	2	5	3	0	2	3	0	3

	0	1	2	3	4	5
C	2	0	2	3	0	1

C[i] contains the number of elements i

	0	1	2	3	4	5
C	2	2	4	7	7	8

C[i] now contains the number of elements <= i



	1	2	3	4	5	6	7	8
B							3	

	0	1	2	3	4	5
C	2	2	4	7	7	8

	1	2	3	4	5	6	7	8
A	2	5	3	0	2	3	0	3

	0	1	2	3	4	5
new C	2	2	4	~~7~~ 6	7	8

B[C[A[8]]=A[8]

next 3 goes into location 6