Sorting

Data Structures and Algorithms (094224)

Yuval Emek

Winter 2022/23

The sorting problem

- Input: an array A of elements coming from a totally ordered set
  - Predicates $x < y$, $x \leq y$, $x = y$ defined for every two elements x, y
  - For simplicity: compare x to y rather than $x.key$ to $y.key$
  - Denote $n = A.length$
- Output: reorder the elements in A so that $A[i] \leq A[i+1]$ for every $1 \leq i \leq n-1$
- We've already encountered the insertion sort algorithm
  - Run-time $\Theta(n^2)$
- In this lecture:
  - Faster sorting algorithms
  - A lower bound!
- Criteria:
  - Run-time
    - Also in practice (beyond asymptotic analysis)
  - Sort in place
    - All operations occur "inside A", no need for additional data structures
  - Comparison based sorting
    - Do not assume anything on the elements except total order

The merge sort algorithm

- The high-level idea:
  1. Divide $A[1 \ldots n]$ to subarrays $A[1 \ldots n/2]$ and $A[n/2+1 \ldots n]$
  2. Sort each subarray recursively
  3. Merge the sorted subarrays
- An example for a *divide and conquer* ⟨הפרד ומשול⟩ algorithm
- The heart of the algorithm: the merging procedure

Procedure Merge

Merge the sorted subarrays $A[p \ldots q]$ and $A[q+1 \ldots r]$ into a sorted subarray $A[p \ldots r]$

`Merge`(A, p, q, r)

1: $(L, R) =$ `Merge_Init`(A, p, q, r)
2: $i = 1$
3: $j = 1$
4: **for** $k = p, \ldots, r$ **do**
5: **if** $L[i] \leq R[j]$ **then**
6: $A[k] = L[i]$
7: $i = i + 1$
8: **else** $A[k] = R[j]$
9: $j = j + 1$

`Merge_Init`(A, p, q, r)

1: $n_1 = q - p + 1$
2: $n_2 = r - q$
3: new array $L[1 \ldots n_1 + 1]$
4: new array $R[1 \ldots n_2 + 1]$
5: **for** $i = 1, \ldots, n_1$ **do**
6: $L[i] = A[p + i - 1]$
7: **for** $j = 1, \ldots, n_2$ **do**
8: $R[j] = A[q + j]$
9: $L[n_1 + 1] = \infty$ ▷ sentinel
10: $R[n_2 + 1] = \infty$ ▷ sentinel
11: return (L, R)

Example

$A =$ | 1 | 4 | 5 | 8 | 2 | 3 | 6 | 7 | $p = 1 \quad q = 4 \quad r = 8$

$A =$ | | | | | | | | |

$L =$ | 1 | 4 | 5 | 8 | ∞ | $R =$ | 2 | 3 | 6 | 7 | ∞ |

$A =$ | 1 | | | | | | | |

$L =$ | 1 | 4 | 5 | 8 | ∞ | $R =$ | 2 | 3 | 6 | 7 | ∞ |

$A =$ | 1 | 2 | | | | | | |

$L =$ | 1 | 4 | 5 | 8 | ∞ | $R =$ | 2 | 3 | 6 | 7 | ∞ |

$A =$ | 1 | 2 | 3 | | | | | |

$L =$ | 1 | 4 | 5 | 8 | ∞ | $R =$ | 2 | 3 | 6 | 7 | ∞ |

Example — cont.

Correctness of the merging procedure

Lemma

When iteration k of the for loop (lines 4–9) of `Merge`(A, p, q, r) *starts, subarray $A[p \ldots k-1]$ contains the $k-p$ smallest elements of L and R in sorted order. Moreover, $L[i]$ and $R[j]$ are the smallest elements of L and R, respectively, that have not been copied to A yet.*

- By induction on $k = p, \ldots, r$
- Base: when iteration $k = p$ starts
  - $A[p \ldots k-1]$ is empty, hence sorted
  - L and R are sorted, hence $L[i]$ and $R[j]$ are the smallest elements of L and R not copied yet
- Inductive hypothesis: when iteration $k \geq p$ starts
  - $A[p \ldots k-1]$ contains the $k-p$ smallest elements
  - $L[i]$ and $R[j]$ are the smallest elements of L and R not copied yet

The proof continues

- Step:
  - Assume w.l.o.g. that $L[i] \leq R[j]$
  - $\Longrightarrow$ $L[i]$ is the smallest element of L and R not copied yet
  - Following line 6, $A[p \ldots k]$ contains the $k - p + 1$ smallest elements
  - Following line 7, $L[i]$ and $R[j]$ are the smallest elements of L and R not copied yet
  - $\Longrightarrow$ assertion holds when iteration $k + 1$ starts ■

Corollary

When `Merge`(A, p, q, r) *terminates, subarray* $A[p \ldots r]$ *is sorted.*

Run-time of the merging procedure

- Set $n = r - p + 1$ (#elements in the input to `Merge`(A, p, q, r))
- `Merge_Init` takes $O(n)$ time
- The for loop of lines 4–9 performs n iterations, each takes $O(1)$ time
- $\Longrightarrow$ $O(n)$ time in total

Algorithm Merge_Sort(A, p, r)

Sort the subarray $A[p \ldots r]$

Merge_Sort(A, p, r)

1: **if** $p < r$ **then**
2: $\quad q = \lfloor (p + r)/2 \rfloor$
3: $\quad$ Merge_Sort(A, p, q)
4: $\quad$ Merge_Sort$(A, q + 1, r)$
5: $\quad$ Merge(A, p, q, r)

How do we sort the whole array A?

Correctness of the merge sort algorithm

Theorem

`Merge_Sort`(A, p, r) *sorts the the subarray* $A[p \ldots r]$.

- By induction on the size $n = r - p + 1$ of the subarray
- Base: when $n = 1$, subarray $A[p \ldots r]$ is already sorted
- Step: Assume that assertion holds for subarrays of size $< n$ and consider subarray of size n
- By ind. hyp., subarrays $A[p \ldots q]$ and $A[q + 1 \ldots r]$ are sorted when we reach line 5
- By correctness of `Merge`(A, p, q, r), subarray $A[p \ldots r]$ is sorted when line 5 is completed ■

1. Merge sort
   - Run-time analysis of recursive algorithms
2. Heap sort
3. Lower bound
4. Counting sort
5. Radix sort

Recurrences

- How do we analyze the tun-time of a recursive algorithm `Alg`?
- Sometimes, we can find *explicit* ⟨**מפורשים**⟩ (tight) bounds on the run-time of `Alg` via combinatorial arguments
  - E.g., DFS
- The more common process:
  1. Express the run-time of `Alg` as a *recurrence* ⟨**משוואת נסיגה**⟩
  2. Solve the recurrence and provide explicit bounds on the run-time
     - Suffices to obtain asymptotic bounds
- Several techniques for solving recurrences

The run-time of merge sort

- Assume that $n = r - p + 1$ is a power of 2
  - Simplifies the analysis (will see why soon)
  - W.l.o.g. as we can extend the input with dummy elements
    - A (slightly) different algorithm
- If $n = 1$, then the algorithm takes $T(1) = O(1)$ time
- Assume that $n > 1$
- Lines 1–2 take $O(1)$ time
- Line 5 takes $O(n)$ time
- What about lines 3 and 4?
- Key observation: calls in lines 3 and 4 take $T(n/2)$ time each
  - Using the fact that if $n > 1$ is a power of 2, then $n/2$ is an integer
- The run-time function $T(n)$ of `Merge_Sort` satisfies

$$T(n) \leq \begin{cases} O(1), & n = 1 \\ 2T(n/2) + O(n), & n > 1 \end{cases}$$

Solving the recurrence — the iterative method

- Safer to write the recurrence with explicit constants

$$T(n) \leq \begin{cases} c, & n = 1 \\ 2T(n/2) + cn, & n > 1 \end{cases}$$

  - How come we use the same constant c?
- Develop

$$\begin{aligned} T(n) &\leq 2T(n/2) + cn \leq 2\left(2T(n/4) + cn/2\right) + cn \\ &= 4T(n/4) + 2cn \leq 4\left(2T(n/8) + cn/4\right) + 2cn \\ &= 8T(n/8) + 3cn \\ &\vdots \\ &= 2^j T\left(n/2^j\right) + jcn \end{aligned}$$

  - Holds since n is a power of 2

The iterative method — cont.

- Setting $j = \lg n$, we get

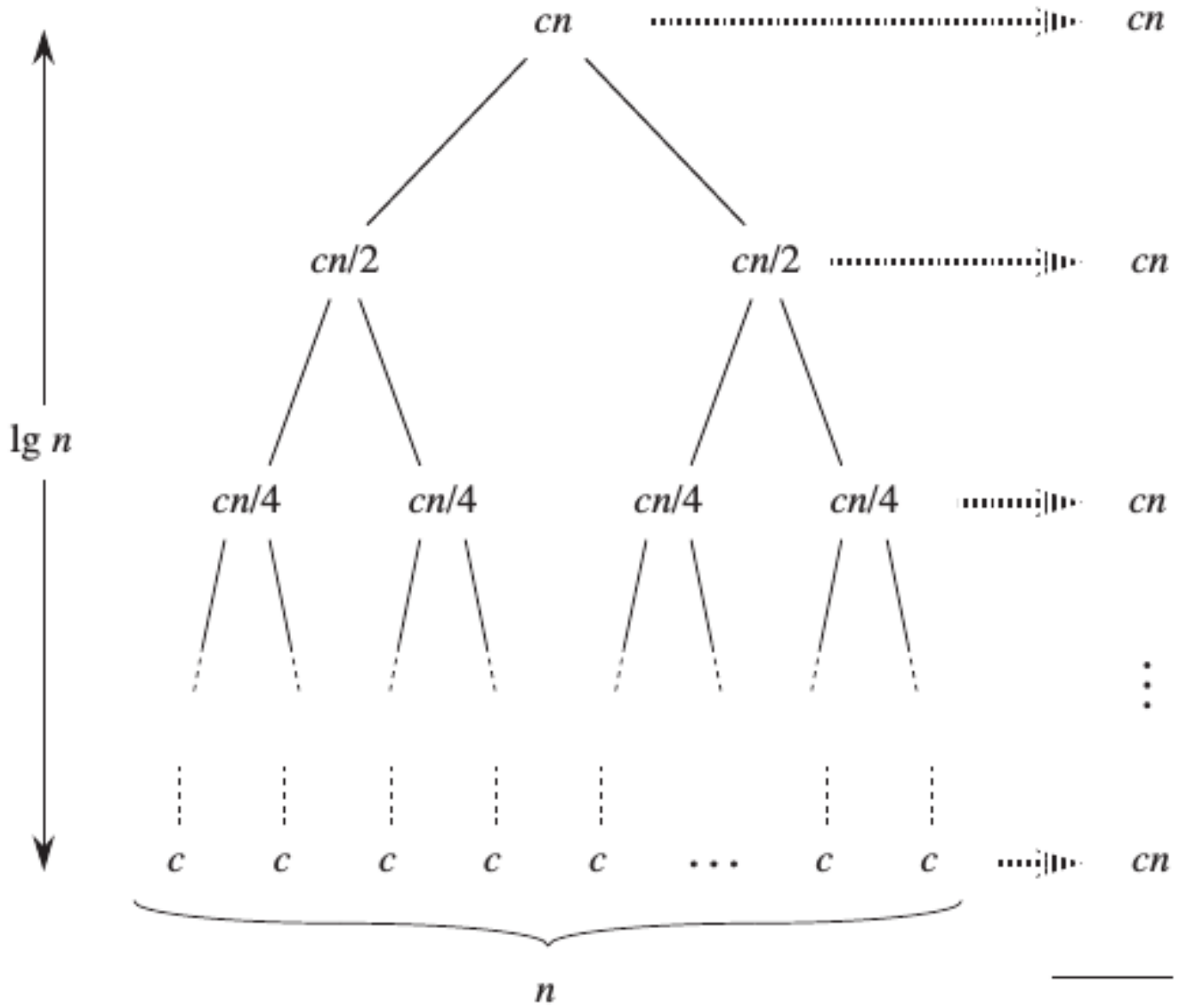
$$T(n) \leq 2^{\lg n} T(n/2^{\lg n}) + cn \lg n = nT(1) + cn \lg n \leq cn + cn \lg n$$

- Therefore $T(n) \leq cn(\lg n + 1)$
- Verify by induction:
  - Base: $T(1) \leq c = c \cdot 1(\lg(1) + 1)$
  - Step:

$$\begin{aligned} T(n) &\leq 2T(n/2) + cn \leq 2c(n/2)(\lg(n/2) + 1) + cn \\ &= cn(\lg n - 1 + 1) + cn = cn(\lg n + 1) \end{aligned}$$

- With some experience: guess the solution directly
  - A.k.a. the substitution method
- We conclude that $T(n) = O(n \log n)$
  - Much faster than insertion sort!

Solving the recurrence — the recursion tree method



Using efficient data structures

- Merge sort is fast: run-time $O(n \log n)$
- Can you think of another way to sort an array in $O(n \log n)$ time?
- Idea:
  1. Construct a balanced tree by inserting the array elements one-by-one
     - n `Insert` operations
  2. Traverse the tree elements in order:
     - 1 call to `Minimum` and then $n-1$ calls to `Successor`;
     - n calls to `Minimum` + `Delete`; or
     - Traverse the tree directly
- Run-time:
  - $O(n \log n)$ time for the construction
  - $O(n \log n)$ time for the traversal
    - $O(n)$ time when using direct traversal
- May be inefficient in practice
  - Relatively large hidden constants
- Sorting not in place

Sorting using a binary heap

- A different approach: use a binary heap instead of a balanced tree
  - Efficient in practice
  - Sorting in place (thanks to the array representation)

`Heap_Sort`(A)

1: `Build_Max_Heap`(A) the root have the max and bigger than children
2: **for** $i = A.length$ down to 2 **do**
3: swap $A[1]$ with $A[i]$
4: $A.heap\text{-}size = A.heap\text{-}size - 1$
5: `Max_Heapify`$(A, 1)$

- Correctness:
  - By induction on i:
    at the end of iteration i, $A[i \ldots n]$ consists of the $n - i + 1$ largest elements in sorted order and $A[1 \ldots i - 1]$ is a maximum heap
- Run-time:
  - $O(n)$ time for `Build_Max_Heap`
  - $O(\log n)$ time for each call to `Max_Heapify`
  - $O(n \log n)$ time in total

Comparison based algorithms

- In many scenarios, the input elements belong to a totally ordered set
  - Supported queries: $x < y$, $x \leq y$, $x = y$, $x \geq y$, $x > y$
- A *comparison based* $\langle$מבוסס השוואות$\rangle$ algorithm doesn't use any other queries on the elements
- W.l.o.g. only the query $x \leq y$ is used
  - The other queries can be implemented with $O(1)$ calls to $x \leq y$
- Examples for queries/operations beyond comparison based:
  - Does the 3rd bit in x equal 1?
  - $z \leftarrow x \oplus y$
  - Is x longer than y?

Comparison based sorting

Theorem

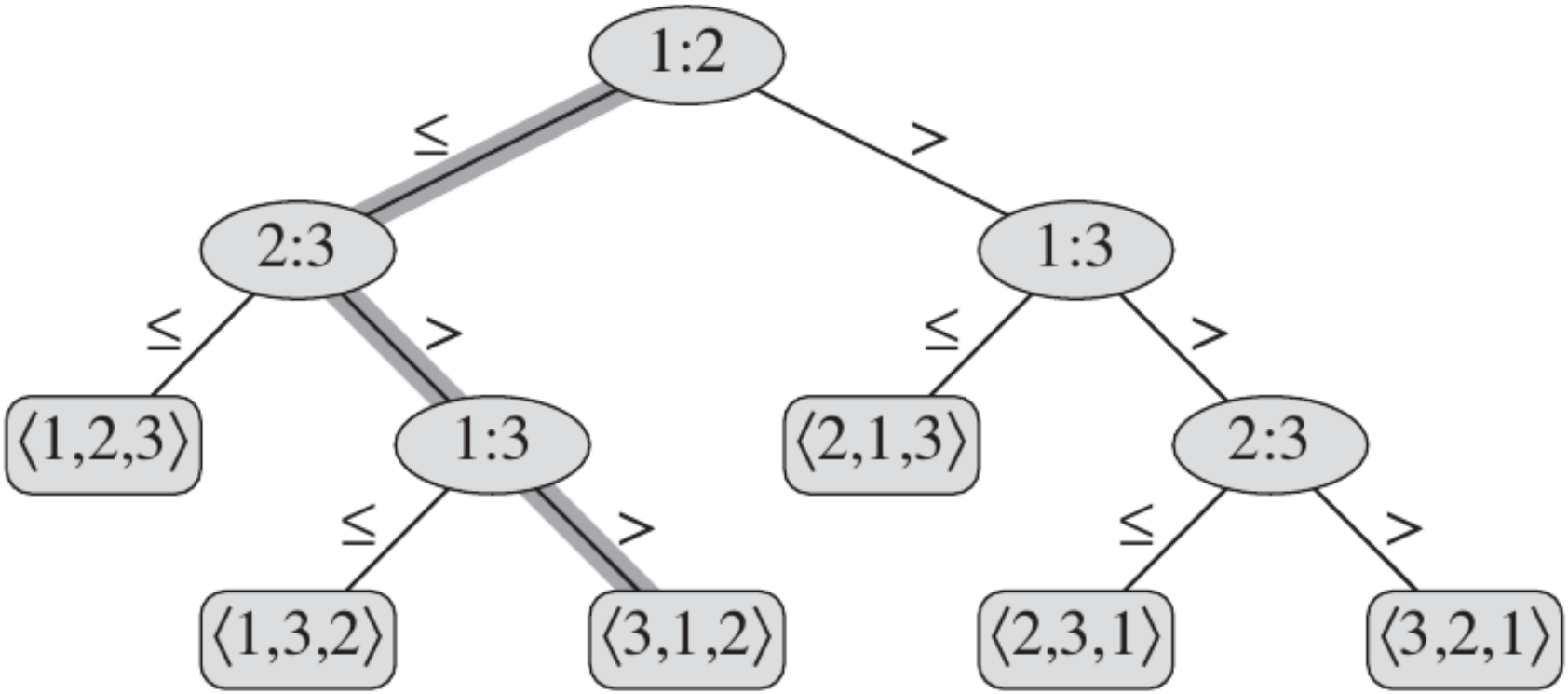
The run-time of a comparison based sorting algorithm must be $\Omega(n \log n)$.

- Establishing run-time lower bounds:
  - An instance: direct analysis
  - An algorithm: $\exists$ bad instance
  - A problem: $\forall$ algorithm, $\exists$ bad instance
    - Typically, much more challenging!
- Preparing for the proof:
  - Consider an arbitrary comparison based sorting algorithm `Alg`
  - Fix some n (can be arbitrarily large)
  - Input array $A[1 \ldots n]$
  - Distinct elements: $A[i] \neq A[j]$ for every $1 \leq i < j \leq n$

Decision trees

- Operation of `Alg` can be described via a *decision tree* $\langle$**עץ החלטה**$\rangle$:
  - A rooted full binary tree
    - Binary tree: the degree of each vertex ≤ 2
    - Full binary tree: the degree of each internal vertex $= 2$
    - Left and right children
  - Each internal vertex is associated with an index pair $(i, j) \in [n] \times [n]$
- From internal vertex (i, j), the execution progresses according to the result of query $A[i] \leq A[j]$
  - Positive answer: continue to left child
    Negative answer: continue to right child
  - Elements $A[i]$ and $A[j]$ in the original input array
- Captures only the comparison queries
  - Data movements, other program control operations, etc. are hidden
    - Occur "in between" tree vertices
- Reaching a leaf $=$ no more comparison queries
  - The sorting permutation for A has been revealed

The decision tree of insertion sort



Highlighted path corresponds to the execution on $A = [6, 8, 5]$

The decision tree of Alg

- Consider the decision tree D_n of Alg on inputs of size n
- Alg is a general algorithm — we make no assumptions on D_n
- Observation 1: $T_{\texttt{Alg}}(n) \geq \text{height}(D_n)$
  - Even if we ignore all operations other than comparison queries

alg time

- Observation 2: Each sorting permutation is associated with one leaf
  - To accommodate every possible input
- $\Longrightarrow$ D_n has $n!$ leaves
- Observation 3: A binary tree of height h has $\leq 2^h$ leaves
  - By induction on h
- $\Longrightarrow$ The height of a binary tree with ℓ leaves $\geq \lg \ell$
- The height of D_n satisfies

$$\begin{aligned} \text{height}(D_n) &\geq \lg(n!) = \lg(1 \cdot 2 \cdots n) \\ &= \lg(1) + \lg(2) + \cdots + \lg(n) \geq \Omega(n \log n) \end{aligned}$$ ■

Sorting integers from a bounded range

- Suppose that the input elements are integers from the range $[0, k]$
- The counting sort algorithm sorts these elements in $O(n + k)$ time
- $\Longrightarrow$ a linear time sorting algorithm if $k = O(n)$
  - Does it contradict the $\Omega(n \log n)$ lower bound?
- The high-level idea:
  1. For each input element x, count the number c_x of elements $\leq x$
  2. Place x in $A[c_x]$
- Challenges:
  - How do we compute c_x for all x (concurrently) in time $O(n + k)$?
  - Modify the algorithm to handle multiple elements with the same value

Algorithm Counting_Sort(A, n, k)

Sort the array $A[1 \ldots n]$ given that $A[i] \in [0, k]$ for every $1 \leq i \leq n$

Counting_Sort(A, n, k)

1: new array $B[1 \ldots n]$
2: new array $C[0 \ldots k] = [0, \ldots, 0]$
3: **for** $j = 1, \ldots, n$ **do**
4: $\quad C[A[j]] = C[A[j]] + 1$ ▷ $C[i]$ holds #elements $= i$
5: **for** $i = 1, \ldots, k$ **do**
6: $\quad C[i] = C[i] + C[i-1]$ ▷ $C[i]$ holds #elements $\leq i$
7: **for** j = n, . . . , 2, 1 **do**
8: $\quad B[C[A[j]]] = A[j]$ ▷ place $A[j]$ in its right location in B
9: $\quad C[A[j]] = C[A[j]] - 1$ ▷ update $C[A[j]]$ as if $A \leftarrow A[1 \ldots j-1]$
10: $A = B$ ▷ entry-wise copy

Example

$A =$ | 2 | 1 | 3 | 2 | 0 | 2 | 1 | 3 |

$C =$ | 1 | 3 | 6 | 8 |

$B =$ | | | | | | | | 3 |

$C =$ | 1 | 3 | 6 | 7 |

$B =$ | | | 1 | | | | | 3 |

$C =$ | 1 | 2 | 6 | 7 |

$B =$ | | | 1 | | | 2 | | 3 |

$C =$ | 1 | 2 | 5 | 7 |

$B =$ | 0 | | 1 | | | 2 | | 3 |

$C =$ | 0 | 2 | 5 | 7 |

Example — cont.

$A =$

2	1	3	2	0	2	1	3

$B =$

0		1		2	2		3

$C =$

0	2	4	7

$B =$

0		1		2	2	3	3

$C =$

0	2	4	6

$B =$

0	1	1		2	2	3	3

$C =$

0	1	4	6

$B =$

0	1	1	2	2	2	3	3

$C =$

0	1	3	6

Analysis

- Run-time: $O(n+k)$
  - Immediate
- Correctness:
  - Invariant: At the beginning of iteration j of the for loop of lines 7–9,

$$C[i] = |\{1 \leq \ell \leq n \mid A[\ell] \leq i\}| - |\{j+1 \leq \ell \leq n \mid A[\ell] = i\}|$$

Stable sorting

- A sorting algorithm is called *stable* ⟨יציב⟩ if elements with the same value appear in the output in the same relative order as in the input
  - I.e., breaking ties so that the element that appears first in the input will also appear first in the output
- Merge sort is stable
  - Exercise: verify
- Heap sort is not stable
  - Exercise: why?
- Counting sort is stable
  - Thanks to the reverse for loop in line 7

Sorting digit by digit

- Suppose that the elements in the input array $A[1 \ldots n]$ are d-digit non-negative integers
  - Index digits so that 1 is least significant and d is most significant
- The radix sort algorithm sorts A digit by digit

`Radix_Sort`(A, d)

1: **for** $i = 1, \ldots, d$ **do**
2: invoke a stable sorting algorithm on A according to digit i

- Correctness:
  - Consider elements x and y and suppose that most significant digit in which x and y differ $= i$
  - Relative order of x and y becomes correct in iteration i
  - Does not change in any iteration $i' > i$
    - Thanks to stable sorting

Implementing radix sort based on counting sort

- If each digit is an integer in the range $[0, k]$, then we can use counting sort in line 2
  - Run-time $= O(d(n + k))$
- Exercise: when is it better than merge sort?
- Exercise: when is it better than (ordinary) counting sort?