# Topic 2: Sorting

# Agenda

- Introduction: Insertion Sort and Merge Sort
- Heap Sort
- Quick Sort
- Radix Sort

# Introduction to Sorting

*Arrange all entries in the input array in an increasing order.*

**Input:** A array of $n$ numbers, $A = \{a_1, \cdots, a_n\}$

**Output:** A permutation (reordering) $A' = \{a'_1, \cdots, a'_n\}$ of the input sequence such that $a'_1 \leq a'_2 \leq \cdots \leq a'_n$.

# Insertion Sort

INSERTION-SORT($A$)

```
1  for  j = 2 to A.length
2       key = A[j]
3       //  insert A[j] into the sorted sequence A[1 : j − 1].
4       i = j − 1
5       while  i > 0 and A[i] > key
6            A[i + 1] = A[i]
7            i = i − 1
8       A[i + 1] = key
```

| Analysis Type | Performance |
|---|---|
| Best-case | $O(n)$ |
| Worst-case | $O(n^2)$ |
| Average | $O(n^2)$ |
| Space | $O(1)$ |

# Merge Sort

```
MERGE-SORT(A, p, r)
1  if p < r
2        q = ⌊(p + r)/2⌋
3        MERGE-SORT(A, p, q)
4        MERGE-SORT(A, q + 1, r)
5        MERGE(A, p, q, r)
```

| Analysis Type | Performance |
|---|---|
| Best-case | $O(n \log n)$ |
| Worst-case | $O(n \log n)$ |
| Average | $O(n \log n)$ |
| Space | $O(n)$ |

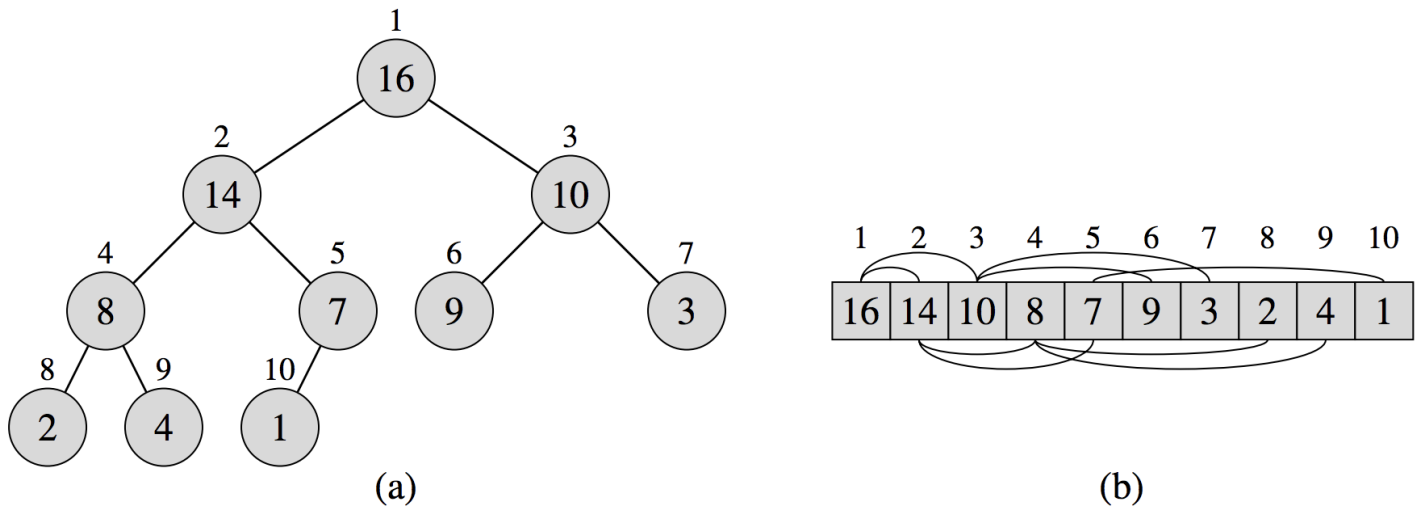
```
MERGE(A, p, q, r)
 1  n_1 = q − p + 1
 2  n_2 = r − q
 3  for  i = 1 to n_1
 4       L[i] = A[p + i − 1]
 5  for  j = 1 to n_2
 6       R[j] = A[q + j]
 7  L[n_1 + 1] = ∞
 8  R[n_2 + 1] = ∞
 9  i = 1
10  j = 1
11  for  k = p to r
12       if L[i] ≤ R[j]
13            A[k] = L[i]
14            i = i + 1
15       else
16            A[k] = R[j]
17            j = j + 1
```

# Can we do any better?

# Heap Sort

- Space Complexity of INSERTION-SORT: $O(1)$
- Time Complexity of MERGE-SORT: $O(n \log n)$

**Heap Data Structure:** (Nearly complete) Binary trees




Source: Figure 6.1 in the textbook.

PARENT$(i)$

1 **return** $\lfloor i/2 \rfloor$

LEFT$(i)$

1 **return** $2i$

RIGHT$(i)$

1 **return** $2i + 1$

# Max-Heap Property

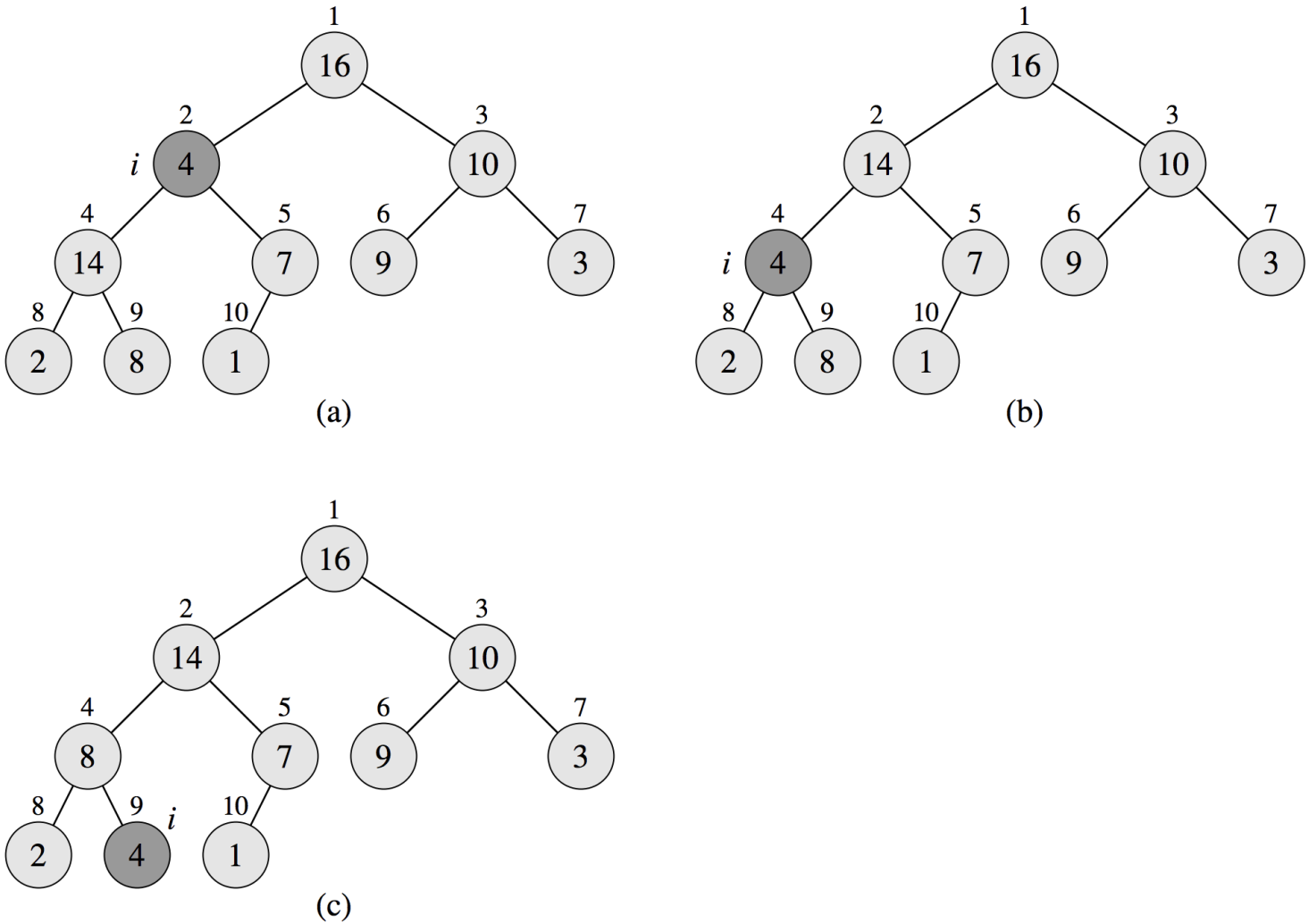
$$A[\textsc{Parent}(i)] \geq A[i].$$

$\textsc{Max-Heapify}(A, i)$

```
 1  l = LEFT(i)
 2  r = RIGHT(i)
 3  if l ≤ A.heap-size and A[l] > A[i]
 4        largest = l
 5  else largest = i
 6  if r ≤ A.heap.size and A[r] > A[largest]
 7        largest = r
 8  if largest ≠ i
 9        swap A[i] and A[largest]
10        MAX-HEAPIFY(A, largest)
```

**Time Complexity:** $O(\log n)$

# Example: MAX-HEAPIFY$(A, 2)$




Source: Figure 6.2 in the textbook.

# Building a Max-Heap

Convert $A[1:n]$ into a max-heap **bottom-up**:

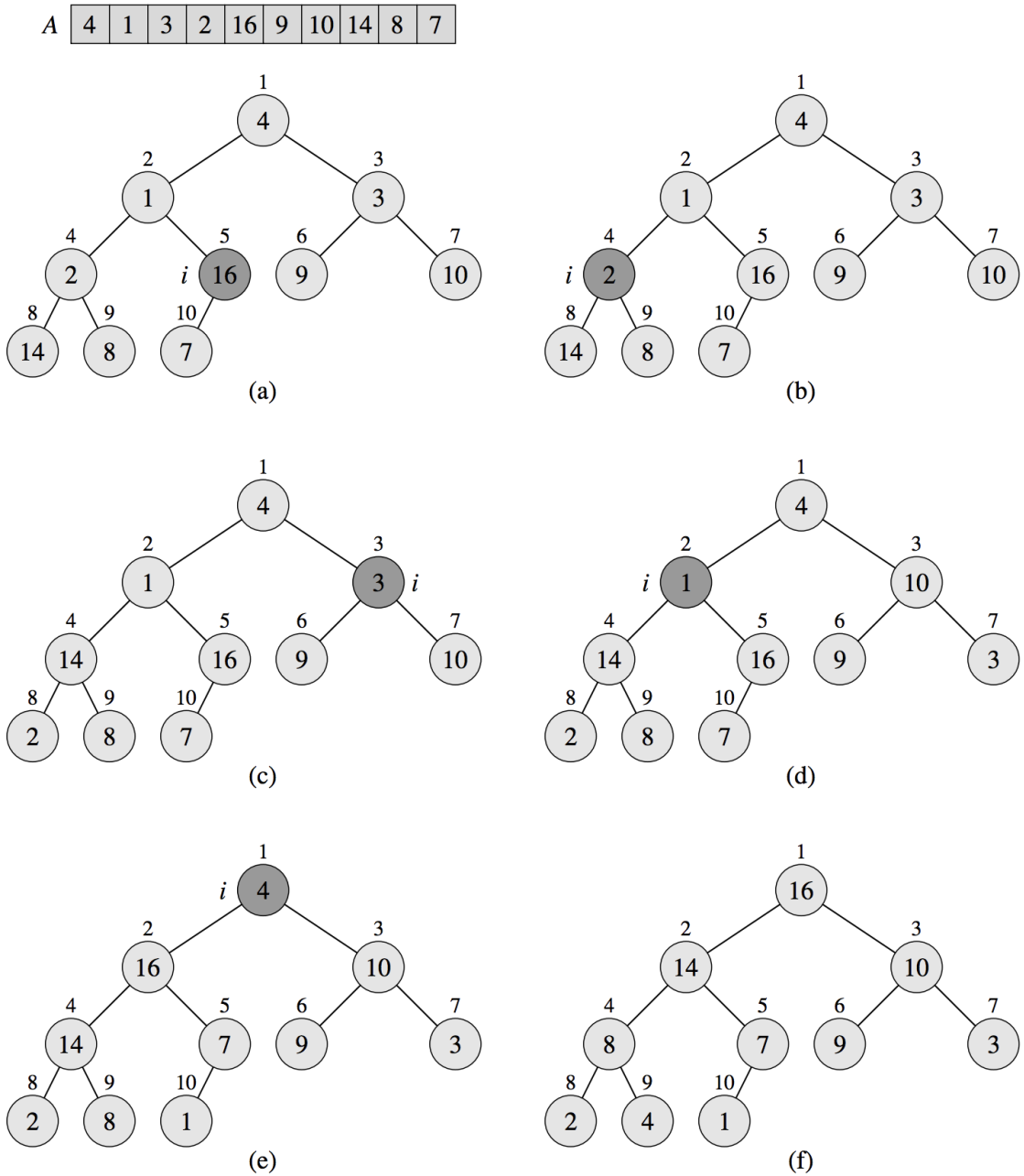
BUILD-MAX-HEAP$(A)$

1 $A.heap\text{-}size = A.size$
2 **for** $i = \lfloor A.length/2 \rfloor$ **downto** $1$
3 MAX-HEAPIFY$(A, i)$

**Run-time:** $O(n)$

**Proof of Correctness:**

- Loop-Invariant: Every node $i+1, \cdots, n$ is the root of a max-heap.




Source: Figure 6.4 in the textbook.

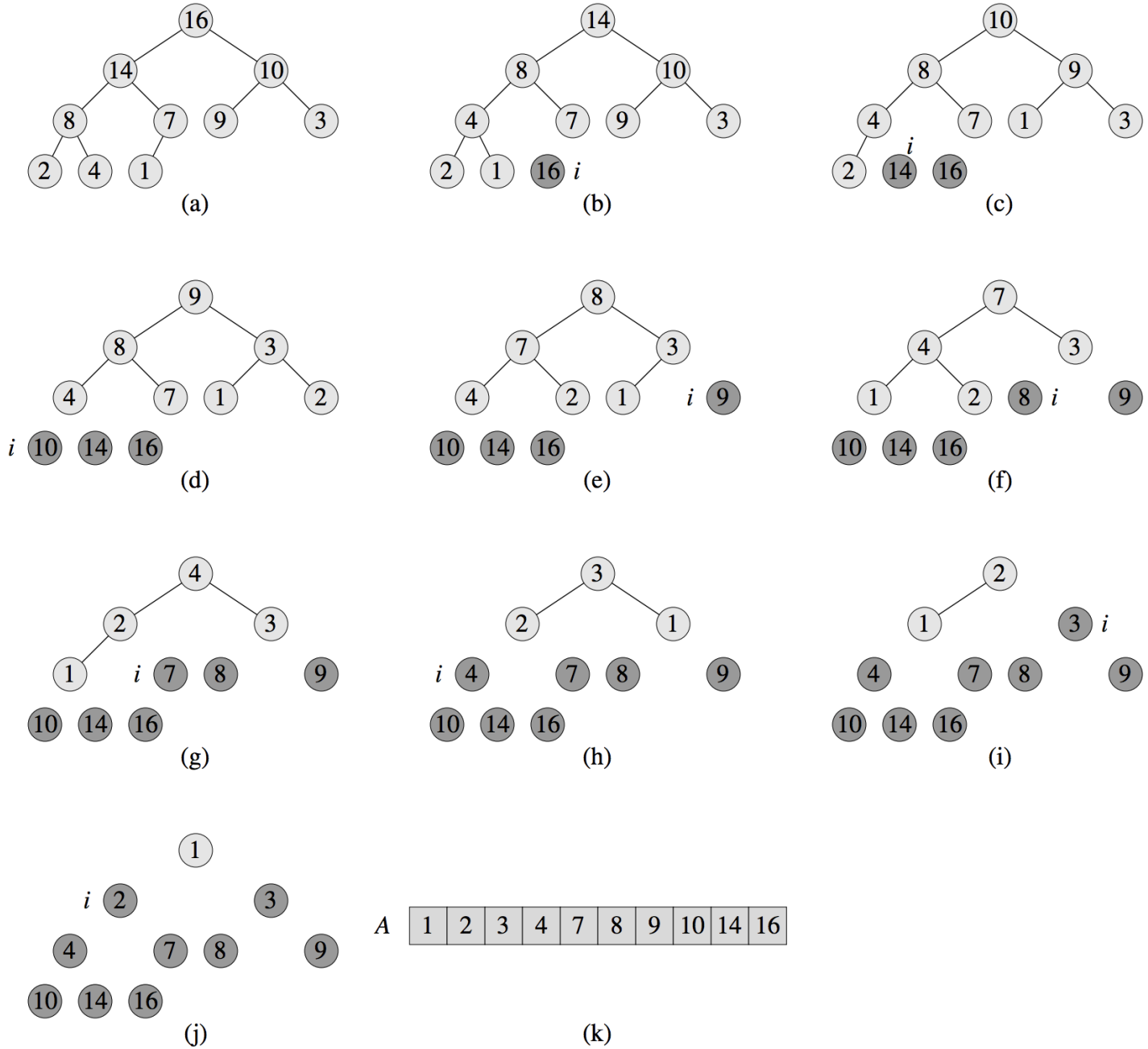
# Heap Sort

HeapSort($A$)

```
1  Build-Max-Heap(A)
2  for i = ⌊A.length⌋ downto 2
3        Swap A[1] and A[i]
4        A.heap-size = A.heap-size − 1
5        Max-Heapify(A, 1)
```

**Time Complexity:** $O(n \log n)$

- Build-Max-Heap($A$): $O(n)$
- Max-Heapify($A, i$): $O(\log n) \rightarrow (n-1)$ calls.




Source: Figure 6.5 in the textbook.

# Quick Sort

**Divide and Conquer** Approach

– *very similar to* MERGE-SORT

- Worst case run-time: $O(n^2)$
- Average run-time: $O(n \log n)$

Partition $A[p : r]$ such that $A[p : q-1] \leq A[q] \leq A[q+1 : r]$:

QUICKSORT$(A, p, r)$

```
1  if p < r
2        q = PARTITION(A, p, r)
3        QUICKSORT(A, p, q − 1)
4        QUICKSORT(A, q + 1, r)
```

# Partitioning the Array

How to partition $A[p:r]$ such that

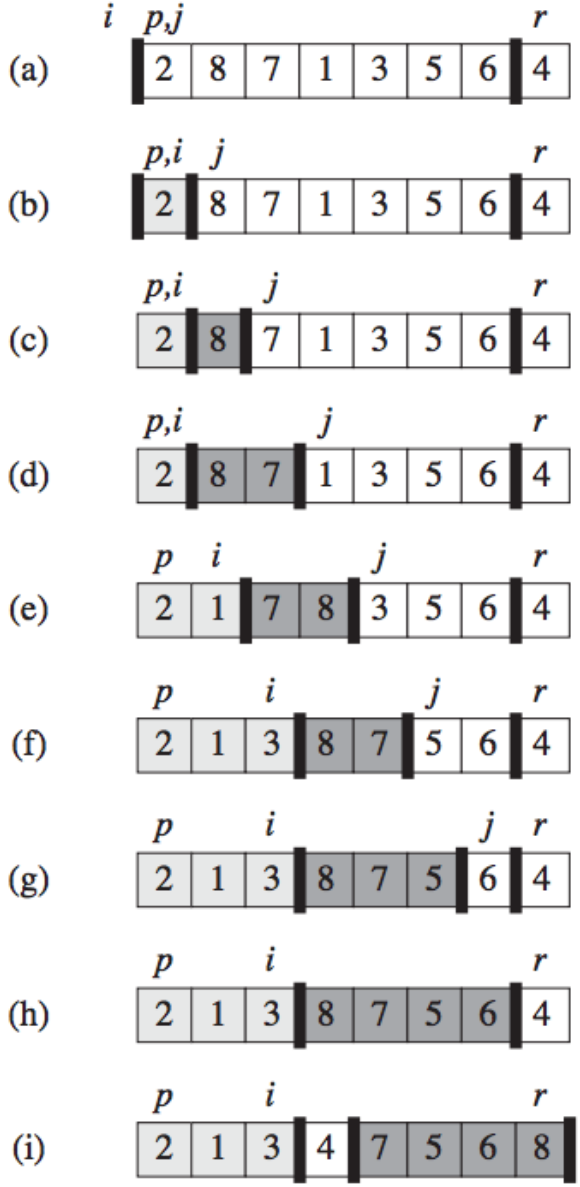
$$A[p:q-1] \leq A[q] \leq A[q+1:r]?$$

PARTITION$(A, p, r)$

```
1  x = A[r]
2  i = p − 1
3  for j = p to r − 1
4        if A[j] ≤ x
5              i = i + 1
6              exchange A[i] with A[j]
7  exchange A[i + 1] with A[r]
8  return i + 1
```

**Run-time Performance:** $\Theta(n)$

# Example: Partitioning the Array



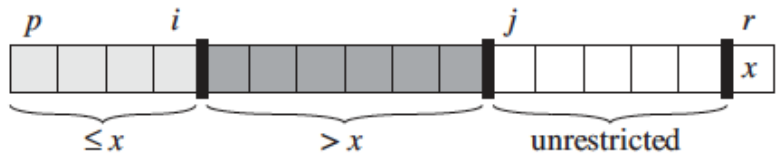

Source: Figure 7.1 in the textbook.

# Proof of Correctness for PARTITION

```
PARTITION(A, p, r)
1  x = A[r]
2  i = p − 1
3  for j = p to r − 1
4       if A[j] ≤ x
5            i = i + 1
6            exchange A[i] with A[j]
7  exchange A[i + 1] with A[r]
8  return i + 1
```

**Loop Invariant:**

1. If $p \leq k \leq i$, then $A[k] \leq x$
2. If $i + 1 \leq k \leq j - 1$, then $A[k] > x$
3. If $k = r$, $A[k] = x$




Source: Figure 7.2 in the textbook.

**PROOF:**

**Initialization:** Prior to first iteration, we have $i = p - 1$ and $j = p$. Note that no $k$ exists between $p$ and $i$, and similarly, no $k$ exists between $i + 1$ and $j - 1$. However, the assignment $x = A[r]$ satisfies the third condition.

# Proof of Correctness for PARTITION (cont.)

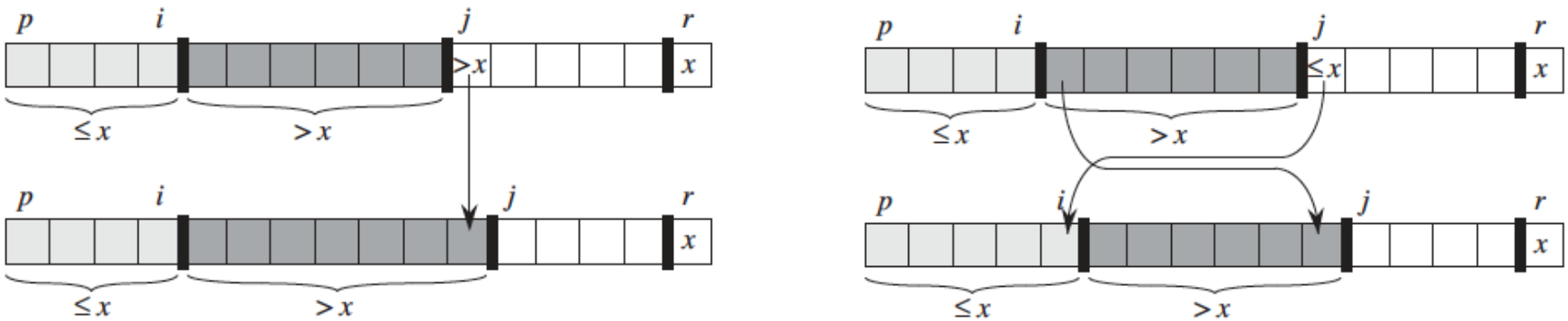
PARTITION$(A, p, r)$

```
1  x = A[r]
2  i = p − 1
3  for j = p to r − 1
4        if A[j] ≤ x
5              i = i + 1
6              exchange A[i] with A[j]
7  exchange A[i + 1] with A[r]
8  return i + 1
```

**Loop Invariant:**

1. If $p \leq k \leq i$, then $A[k] \leq x$
2. If $i + 1 \leq k \leq j - 1$, then $A[k] > x$
3. If $k = r$, $A[k] = x$

**Maintenance:** Two cases arise.




Source: Figure 7.3 in the textbook.

**Termination:** Entire array satisfies the loop invariant.

# Partitioning Effects on Quick Sort

**Worst-case Partitioning:** $T(n) = \Theta(n^2)$, since

$$\begin{aligned} T(n) &= T(n-1) + T(0) + \Theta(n) \\ &= T(n-1) + \Theta(n) \end{aligned}$$

**Best-case Partitioning:** $T(n) = \Theta(n \log n)$, since

$$T(n) = 2T(n/2) + \Theta(n)$$

# Run-time Performance for Quick Sort

More formally, the worst case run-time performance is given by

$$T(n) = \underset{1 \leq q \leq n}{\text{maximize}} \ (T(q-1) + T(n-q)) + \Theta(n)$$

Let us use substitution method, as we saw the worst partitioning leads to $\Theta(n^2)$. Therefore, we have

$$\begin{aligned} T(n) &= \underset{1 \leq q \leq n}{\text{maximize}} \ \left(\Theta\left[(q-1)^2\right] + \Theta\left[(n-q)^2\right]\right) + \Theta(n) \\ &\leq c_2 \cdot \underset{1 \leq q \leq n}{\text{maximize}} \ \left((q-1)^2 + (n-q)^2\right) + \Theta(n) \end{aligned}$$

Note that $(q-1)^2 + (n-q)^2 \leq (n-1)^2$ (Use second derivative test to show that LHS is convex.) Consequently,

$$T(n) \leq c_2(n-1)^2 + \Theta(n) = O(n^2).$$

# Run-time Performance for Quick Sort

Similarly, we have

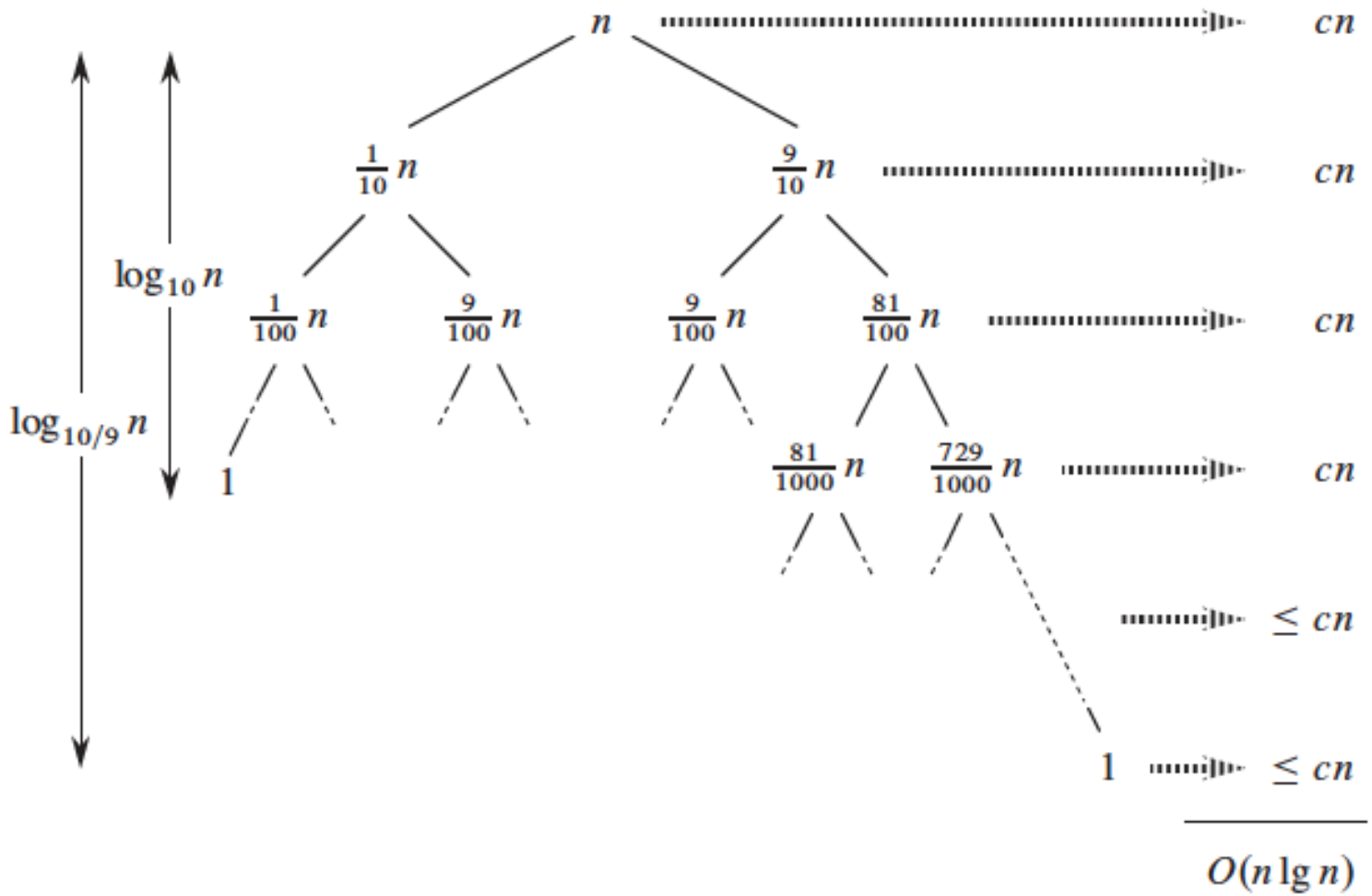
$$\begin{aligned} T(n) &\geq \underset{1 \leq q \leq n}{\text{maximize}} \left( c_1(q-1)^2 + c_1(n-q)^2 \right) + \Theta(n) \\ &= c_1 \cdot \underset{1 \leq q \leq n}{\text{maximize}} \left( (q-1)^2 + (n-q)^2 \right) + \Theta(n) \\ &\geq c_1(n-1)^2 + \Theta(n) = \Omega(n^2). \end{aligned}$$

In other words, we have $T(n) = \Theta(n^2)$.

# Unbalanced Partitioning

**Example:** Always partition as a 9-to-1 split:

$$T(n) = T(9n/10) + T(n/10) + cn$$

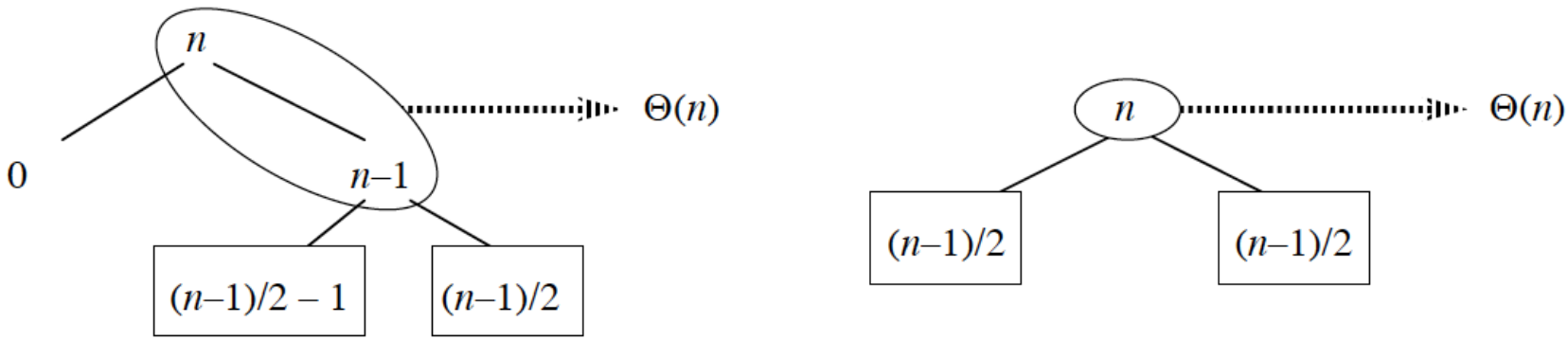



Source: Figure 7.4 in the textbook.

# Average Performance on Quick Sort

Random Input $\Rightarrow$ Mix of balanced and unbalanced partitions.

This has no effect on average performance. Why?




Source: Figure 7.5 in the textbook.

- The extra level in the left figure only adds a constant in $\Theta(n)$.
- Same number of sub-arrays to be sorted $\Rightarrow$ Both result in $O(n \log n)$.

**Average Performance closer to best case performance.**

# How fast can we sort?

**Comparison Sorting:** The only operation used to find the sorting order is comparison between a pair of elements.

- Insertion Sort, Merge Sort, Heap Sort, Quick Sort are all comparison sorts.
- **Note:** It takes $\Omega(n)$ to examine all the array elements.
- So far, all algorithms exhibit $\Omega(n \log n)$.

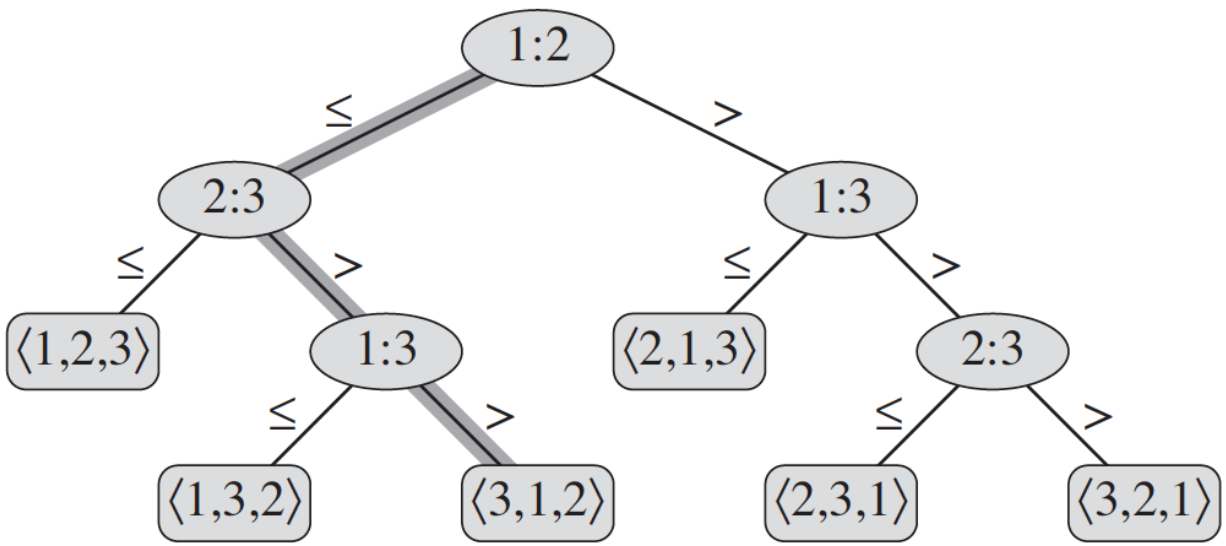
*Is $\Omega(n \log n)$ the lower bound for all comparison sorts?*

# Decision Trees

## Abstraction of Comparison Sorting

- Represents comparisons made by a specific sorting algorithm
- Abstracts out control and data movement.

**Example:** Insertion Sorting on an array with three elements.




Source: Figure 8.1 in the textbook.

# Lower Bounds for Sorting

- Total number of leaves $l \geq n!$ (each permutations of a given array appears at least once).
- Length of the longest path depends on the algorithm
  - Insertion sort: $O(n^2)$
  - Merge sort: $O(n \log n)$

**Lemma**

Any binary tree of height $h$ has atmost $2^h$ leaves.

Proof.

Prove by induction. □

**Theorem**

Any decision tree that sorts $n$ elements has height $\Omega(n \log n)$.

Proof.

- We have $n! \leq l \leq 2^h$, i.e. $2^h \geq n!$, or, $h \geq \log n!$ (Similar to your HW 1 problem)
- From Sterling's formula, we have $n! > (n/e)^n$.
- In other words, we have

$$\begin{aligned} h &\geq \log(n/e)^n \\ &= n \log n - n \log e \\ &= \Omega(n \log n) \end{aligned}$$

□

# Asymptotically Optimal Comparison Sorting

**Corollary**

Heap Sort and Merge Sort are asymptotically optimal comparison sorts.

**Can we beat comparison sorts by adopting a different approach?**

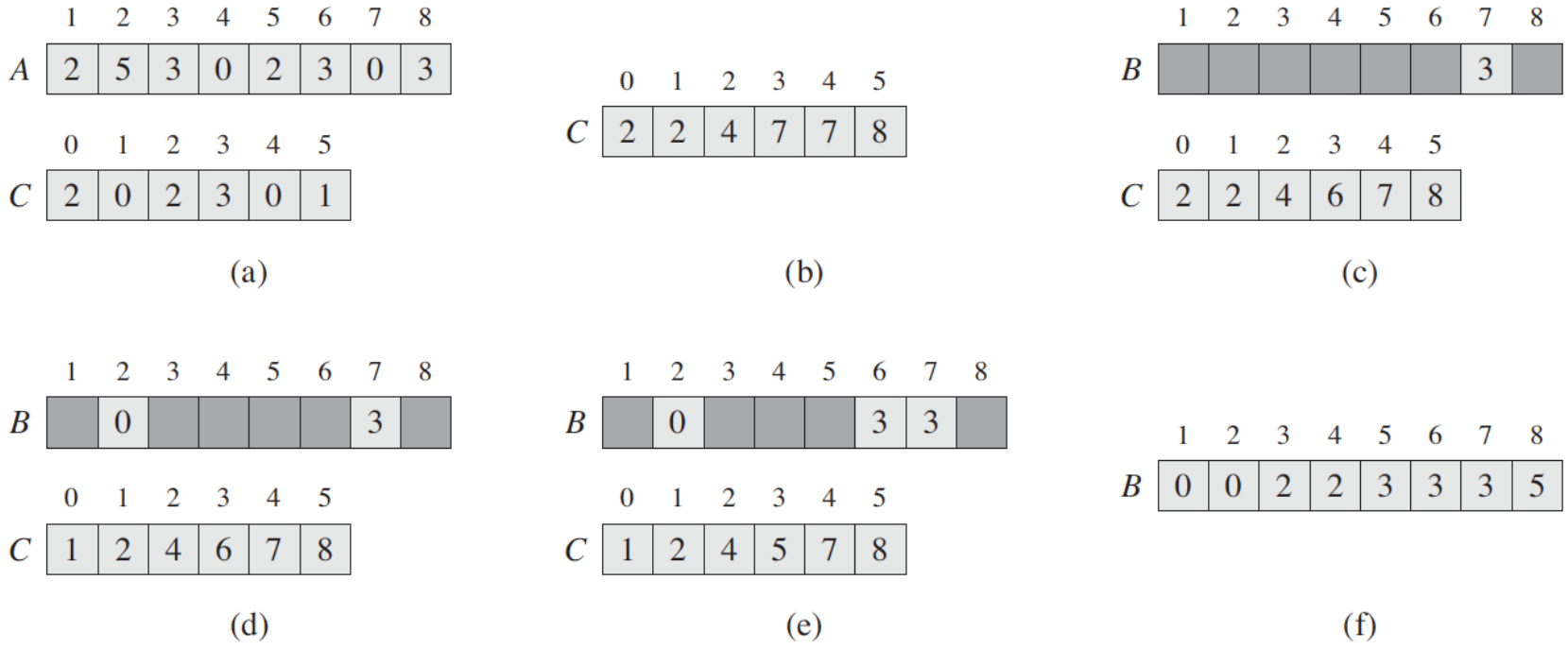
# Counting Sort

- **Input:** $A[1:n]$, where $A[j] \in \{0, 1, \cdots, k\}$.
- **Output:** $B[1:n]$. (Note: $B$ is already allocated)
- **Auxiliary Storage:** $C[0:k]$

COUNTING-SORT$(A, B, k)$

```
1   // Let C[1 : 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   // Now, C[i] 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] contains the number of elements ≤ i
10  for j = A.length to 1
11      B[C[A[j]]] = A[j]
12      C[A[j]] = C[A[j]] − 1
```

# Counting Sort




Source: Figure 8.2 in the textbook.

**Run-time:** $\Theta(n + k)$

- First loop: $\Theta(k)$, Second loop: $\Theta(k)$, Third loop: $\Theta(n)$

Counting sort is **stable**, i.e.

- Keys with same value appear in same order in output as in input.

**What if, there is no bound on the array entries?**

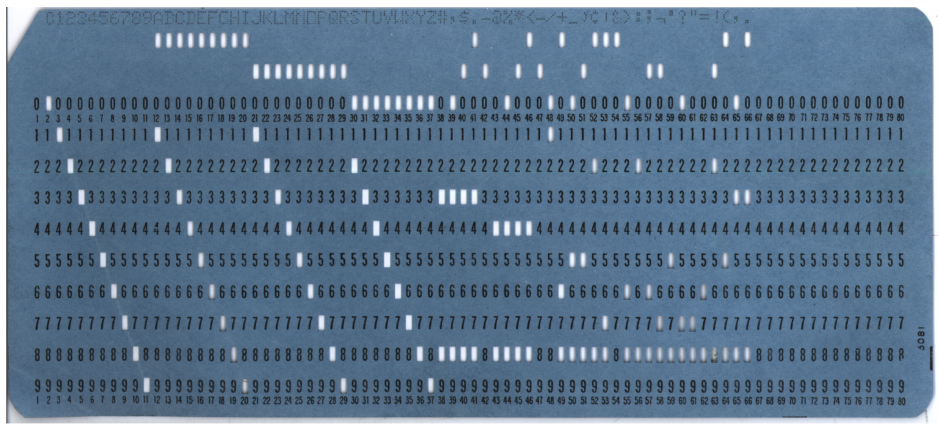
- How about $b$-bit input?
- Note: $k = 2^b$ – Exponentially growing in $b$

**Solution: Radix Sort**

- Uses counting sort as a sub-routine.
- Here, k $= 9$ (a constant).

# History of Radix Sort – Birth of IBM

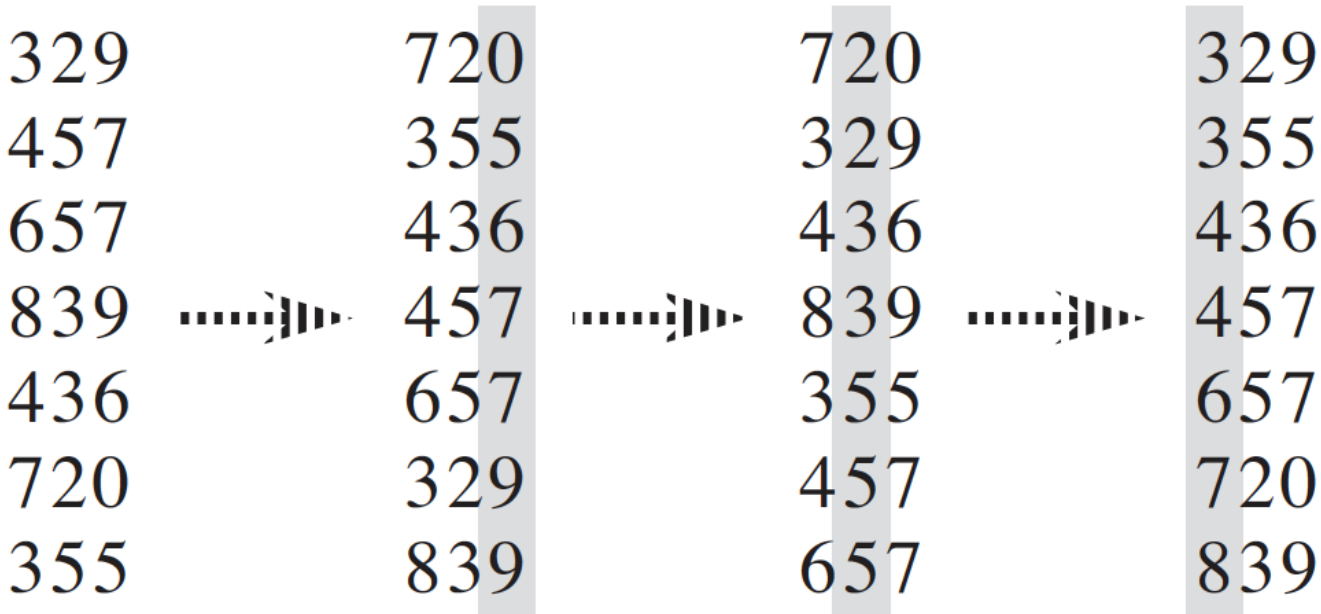
- The Hollerith Machine: 1888 Census Competition
- IBM's punch-card sorters for tabulating census data.
- Sort one column in the punch-card at a time.
- Human operator – a part of this algorithm.



Source: Wikipedia

# Radix Sort

- First sort the least significant digit column
- Proceed column-by-column in the increasing order of significance of the digits.
- Finally, sort the most significant digit column.
- To sort each column, use COUNTING-SORT routine.




Source: Figure 8.3 in the textbook.

# Analysis of Radix Sort

RADIX-SORT$(A, d)$

1 **for** $i = 1$ **to** $d$

2 Use COUNTING-SORT to sort array A on digit-$i$.

**Run-time:** $\Theta(d(n + 10))$

*How does* RADIX-SORT *perform for other number representations?*

**Lemma**

Given $n$ $b$-bit numbers and any positive integer $r \leq b$, RADIX-SORT correctly sorts these numbers represented using $r$-digits in $\Theta\left[(b/r)(n + 2^r)\right]$ time.

# Then, why do we need comparison sorts?

- Consider quick sort, for example.
- Note the constants inside the $\Theta$ notation in both Radix and Quick Sorts.
- Although radix sort takes less passes $(d)$, each pass takes significantly longer time than quicksort.
- Depends on the setting...
  - Quicksort often uses hardware caches more effectively than Radix sort
  - Radix sort requires more memory.
  - Also depends on the input data.

# Summary: Sorting Performance

| Algorithm | Run-Time | | | Memory |
|---|---|---|---|---|
| | Best-Case | Worst-Case | Average | |
| Insertion Sort | $O(n)$ | $O(n^2)$ | $O(n^2)$ | $O(1)$ |
| Merge Sort | $O(n \log n)$ | $O(n \log n)$ | $O(n \log n)$ | $O(n)$ |
| Heap Sort | $O(n \log n)$ | $O(n \log n)$ | $O(n \log n)$ | $O(1)$ |
| Quick Sort | $O(n \log n)$ | $O(n^2)$ | $O(n \log n)$ | $O(1)$ |
| Radix Sort | $O(d(n+k))$ | $O(d(n+k))$ | $O(d(n+k))$ | $O(n+k)$ |

*Which sorting algorithm would you choose?*

# Sorting in Practical Applications

- Before we can sort data, note the data structure used to store the input data.
- Example 1: We can only access the top of a *stack*.
- Example 2: We can only access adjacent elements in a *linked list*.

**How do we sort elements if the input data is provided in a specific data structure?**

# Basic Data Structures and Operations

Operations allowed in a

- **Stack:** Push and pop.
- **Linked List:** Append, Prepend, Insert and Delete.
- **Array:** Insert, Delete and Search.
- **Heap/Tree:** Insert, Delete, Parent, Child and Heapify.
- **Graph:** Insert_Vertex, Delete_Vertex, Insert_Edge, Delete_Edge, Neighbors and Adjacent.

# Sorting on Stacks

Operations allowed in a stack are PUSH and POP.

- Can observe only the top of the stack at any given time.
- Can **never** sort a stack *in-place*.
- Therefore, we always need an additional data structure (e.g. array, tree) to store all the data stored in the stack.

SORT-STACK($A$)

1 Pop out all the entries in $A$ and save them in an array $B$.
2 Sort $B$ using an appropriate sorting algorithm.
3 Push in all the entries in $B$ into $A$.

# Sorting on Linked Lists

- Can always view the next element in a singly linked list
- Can view both adjacent elements in a doubly linked list.
- *In-place* sorting feasible.

**Singly-Linked List:** Can only view elements in the forward direction. Else, start from root again.

SORT-SINGLYLINKEDLIST($A$)

1 SELECTION-SORT($A$)

**Doubly-Linked List:** Can view elements in both forward and backward directions.

SORT-DOUBLYLINKEDLIST($A$)

1 INSERTION-SORT($A$)

# Sorting on Arrays

**1-D arrays:**

- Use MERGE-SORT if additional memory is available.
- Use QUICK-SORT for *in-place* sorting.

**n-D arrays:**

- Examples: matrices.
  - Use COLUMN-SORT – allows parallel implementations
  - Example: RADIX-SORT

SORT-MATRIX($A$)

1 COLUMN-SORT($A$)

# Sorting on Graphs

**Binary Trees and Heaps:**

Sort-Graph($G$)

1 Heap-Sort(G)

**Undirected Graphs:**

- Node Centrality in Social Networks

Sort-Graph($G$)

1 Store node centralities as a priority value
2 Build a heap on the set of vertices.
3 Heap-Sort(V)

**Directed Acyclic Graphs:**

- Example: Prerequisites
- Topological Sorting – will be covered later in this course.

# Something to Think About...

*This lecture only provides one possible design approach for sorting per data structure. Students are encouraged to explore other design approaches.*

Also, can we gain by

- Convert the input's data structure into another form,
- Sort the input data on the new data structure,
- Convert the sorted input into its original data structure?

**Example:**

- Convert singly linked-list into array,
- Sort the input data as an array using Quick-Sort,
- Convert back the sorted array into a singly linked list.