# Linear-Time Sorting

Summer 2018 • Lecture 07/05

# Announcements

- Homework 1
  - `hw1.zip` is live!
  - It's due next Tuesday 7/10.
  - Two slight updates published on 7/4: (1) n >= 3 in Jupyter notebook comment and (2) **N** > 2 for Exercise 3b.
  - After today, you will have learned all of the required material.
- Tutorial 2
  - Friday, 7/6 3:30-4:50 p.m. in STLC 115.
  - RSVP, so I can print enough copies for everyone: https://goo.gl/forms/MSGUGEVBnXaS21kR2 (requires Stanford email).

# Course Overview

- Algorithmic Analysis
- **Divide and Conquer**
- Randomized Algorithms
- Tree Algorithms
- Graph Algorithms
- Dynamic Programming
- Greedy Algorithms
- Advanced Algorithms

# Today's Outline

- Finish Divide and Conquer II properly
  - Linear-time selection
  - Proving runtime with substitution method
- Linear-Time Sorting
  - Comparison-based sorting lower bounds
  - *Algorithms: Counting sort, bucket sort, and radix sort*
  - Reading: CLRS: 8.1-8.2

# Linear-Time Selection

Summer 2018 • Lecture 07/05

# Analyzing Runtime

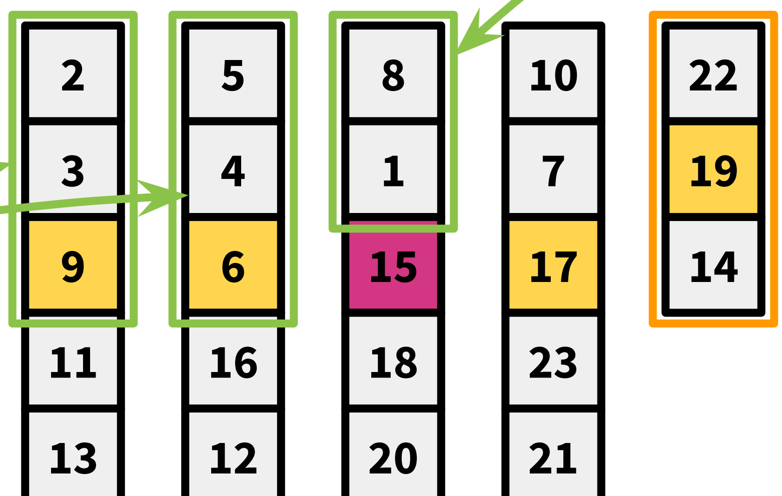
- As a function of n (the size of the original list), how many elements are guaranteed to be **smaller** than the median of medians?
  - Let $g = \lceil n/5 \rceil$ represent the number of groups.
  - **At least** $3 \cdot (\lceil g/2 \rceil - 1 - 1) + 2$ elements.

To exclude the list with the **median of medians**.

To exclude the list with the **leftovers**.

**2** elements from the small group containing the **median of medians**.

**3** elements from each small group with a **median** smaller than the **median of medians**.



# Analyzing Runtime

- What's the recurrence relation?
  - $T(n) = n\log(n)$ when $n \le 100$
  - $T(n) \le T(n/5) + T(7n/10) + O(n)$
  - We can't use **Master Theorem**!
  - We use **substitution method**!

```
def select(A, k, c=100):
  if len(A) <= c:
    return naive_select(A, k)
  pivot = median_of_medians(A)
  left, right = partition_about_pivot(A, pivot)
  if len(left) == k: return pivot
  elif len(left) > k: return select(left, k, c)
  else: return select(right, k-len(left)-1, c)
```

# Substitution Method

$$T(n) = n\log(n) \text{ when } n \le 100$$

$$T(n) \le T(n/5) + T(7n/10) + O(n)$$

1. **Guess what the answer is.**
   - **Linear-time** select
   - Compared to mergesort recurrence, less than n log(n)

→ **Guess O(n)**

# Substitution Method

$T(n) = n\log(n)$ when $n \leq 100$

$T(n) \leq T(n/5) + T(7n/10) + O(n)$

## 2. Formally prove that's what the answer is.

- **Inductive hypothesis** $T(k) \leq Ck$ for all $1 \leq k < n$.
- **Base case** $T(k) \leq Ck$ for all $k \leq 100$.
- **Inductive step**
  - $T(n) = T(n/5) + T(7n/10) + dn$
    $\leq C(n/5) + C(7n/10) + dn$
    $= (C/5)n + (7C/10)n + dn$
    $\leq Cn$

C is some constant we'll have to fill in later!

C must be $\geq \log(n)$ for $n \leq 100$, so $C \geq 7$.

Solve for C to satisfy the inequality. $C \geq 10d$ works.

- **Conclusion** There exists some $C = \max\{7, 10d\}$ such that for all $n > 1$, $T(n) \leq Cn$. Therefore, $T(n) = O(n)$.

# Substitution Method

$T(n) = n\log(n)$ when $n \le 100$

$T(n) \le T(n/5) + T(7n/10) + O(n)$

## 2. Formally prove that's what the answer is.

- **Inductive hypothesis** $T(k) \le \max\{7, 10d\}k$ for all $1 \le k < n$.
- **Base case** $T(k) \le \max\{7, 10d\}k$ for all $k \le 100$.
- **Inductive step**
  - $$\begin{aligned} T(n) &= T(n/5) + T(7n/10) + dn \\ &\le \max\{7, 10d\}(n/5) + \max\{7, 10d\}(7n/10) + dn \\ &= (\max\{7, 10d\}/5)n + (7\max\{7, 10d\}/10)n + dn \\ &\le \max\{7, 10d\}n \end{aligned}$$
- **Conclusion** There exists some $C = \max\{7, 10d\}$ such that for all $n > 1$, $T(n) \le \max\{7, 10d\}n$. Therefore, $T(n) = O(n)$.

# Today's Outline

- Finish Divide and Conquer II properly
  - ~~Linear-time selection~~ **Done!**
  - ~~Proving runtime with substitution method~~ **Done!**
- Linear-Time Sorting
  - Comparison-based sorting lower bounds
  - *Algorithms: Counting sort, bucket sort, and radix sort*
  - Reading: CLRS: 8.1-8.2

# Linear-Time Sorting

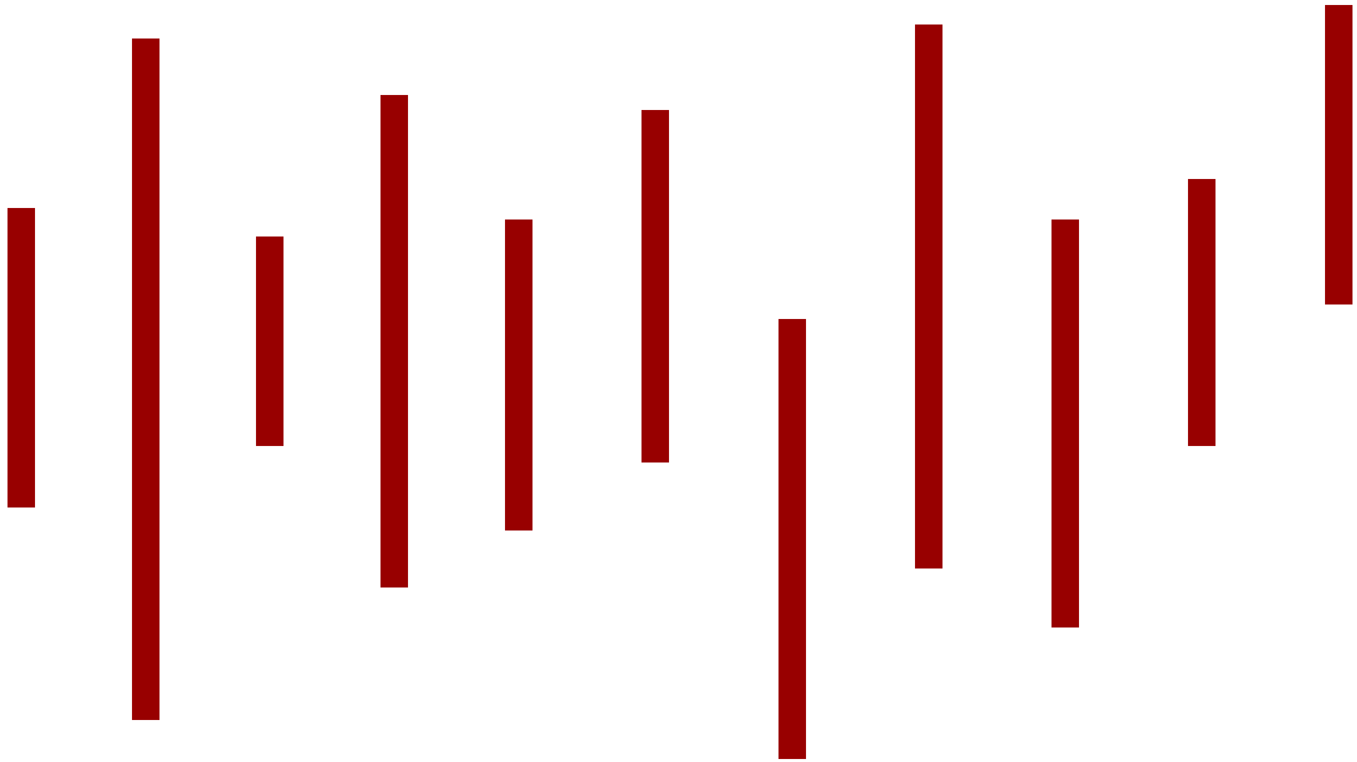
Summer 2018 • Lecture 07/05

# Sorting

- We've seen a few sorting algorithms
  - Insertion sort is worst-case $\Theta(n^2)$-time.
  - Mergesort is worst-case $\Theta(n \log(n))$-time.
- Can we do better?

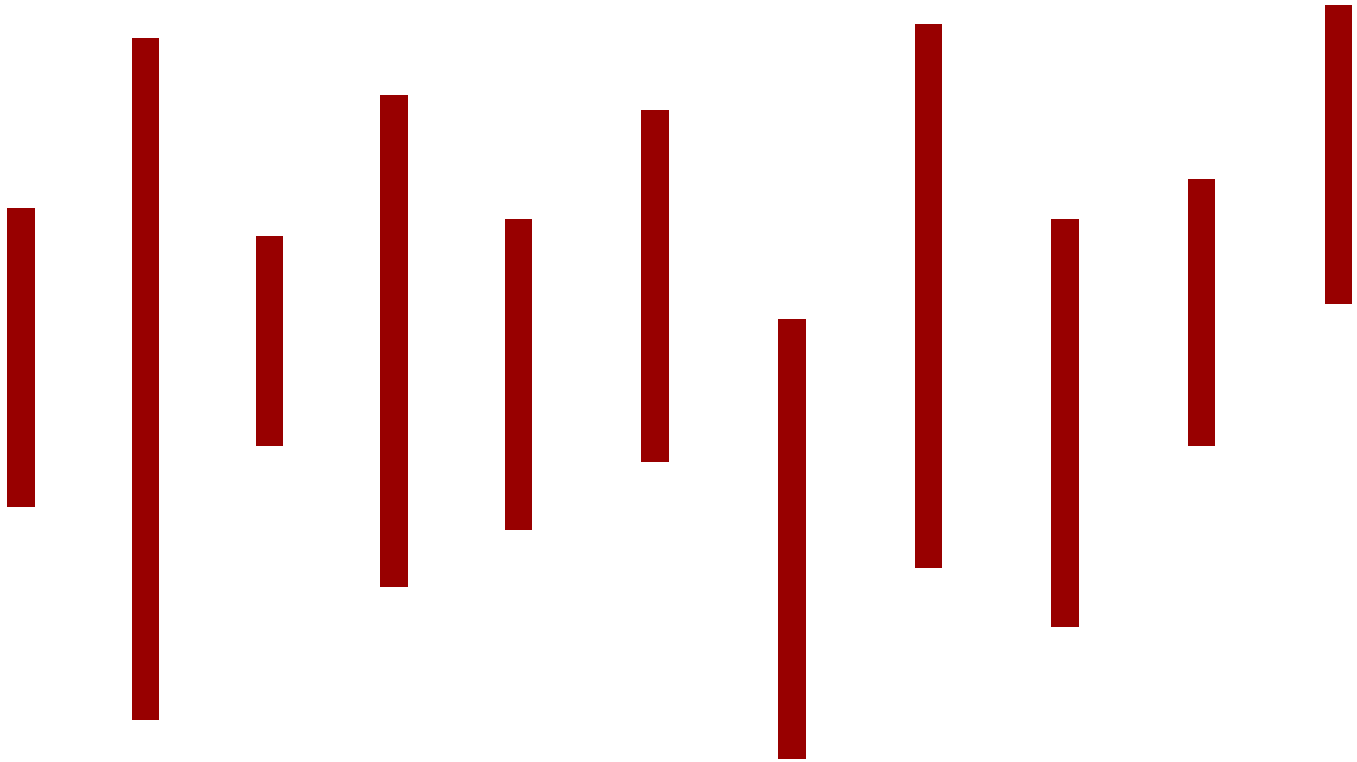
# Sorting

- Problem: sort these n sticks by length.

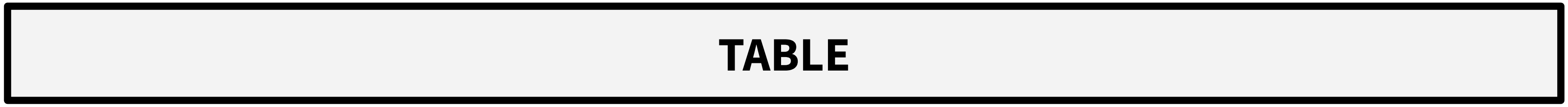


# Sorting

- Problem: sort these n sticks by length.



Algorithm: Drop them on the table.

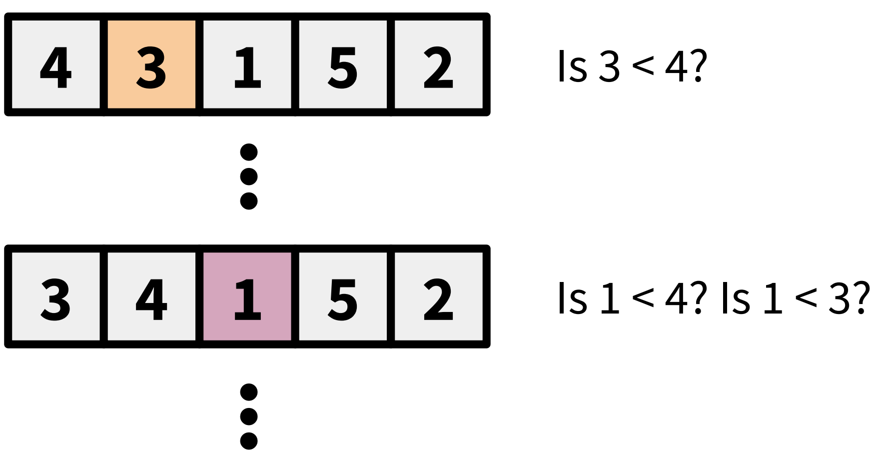


# Sorting

- That might have been unsatisfying, but this **`stick_sort`** does raise some important questions.
  - What is our model of computation?
    - **Input:** list
      **Output:** sorted list
      **Operations allowed:** comparisons
    - **Input:** sticks
      **Output:** sorted sticks in vertical order
      **Operations allowed:** dropping on tables
  - What are reasonable models of computation?

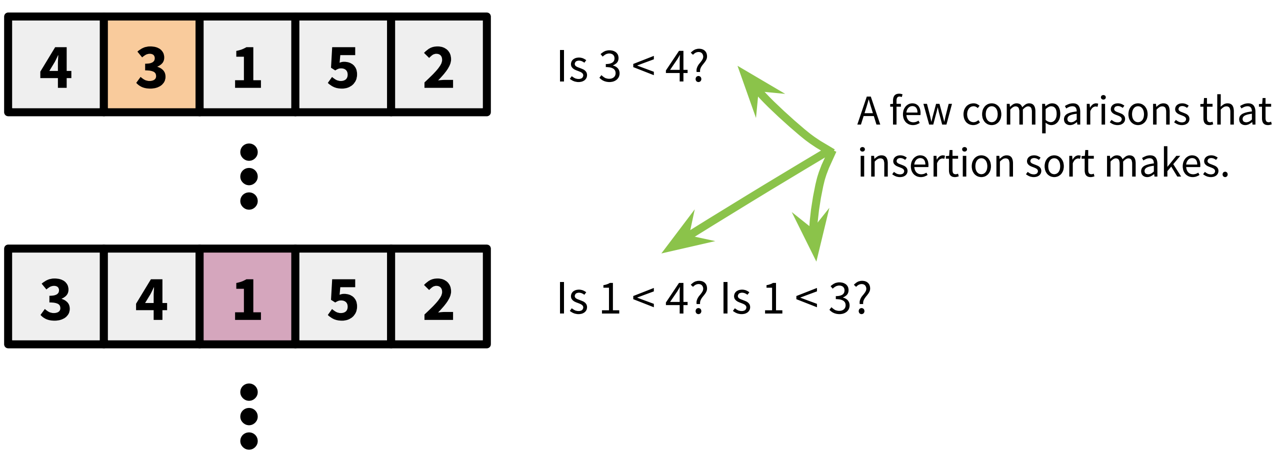
# Comparison-Based Sorting

- Comparison-based algorithms use “comparisons” to achieve their output.
  - Insertion sort and mergesort are comparison-based sorting algorithms.
  - Linear-time selection is a comparison-based algorithm.
  - Next week, we’ll see a randomized comparison-based sorting algorithm called quicksort.



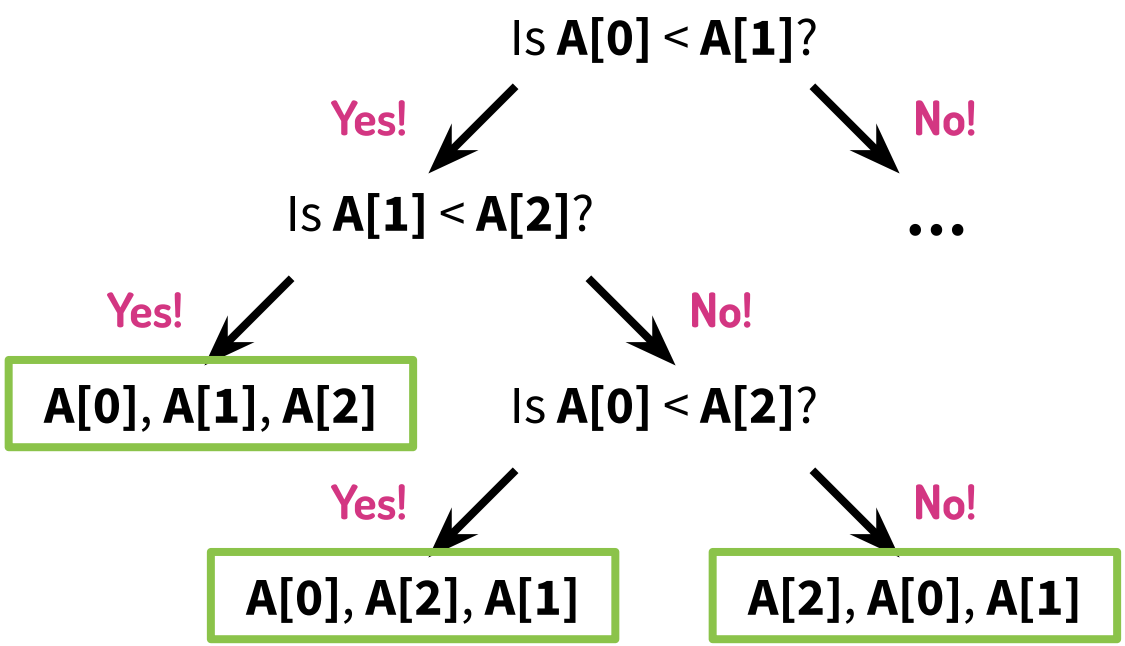
# Comparison-Based Sorting

- Comparison-based algorithms use “comparisons” to achieve their output.
  - Insertion sort and mergesort are comparison-based sorting algorithms.
  - Linear-time selection is a comparison-based algorithm.
  - Next week, we’ll see a randomized comparison-based sorting algorithm called quicksort.



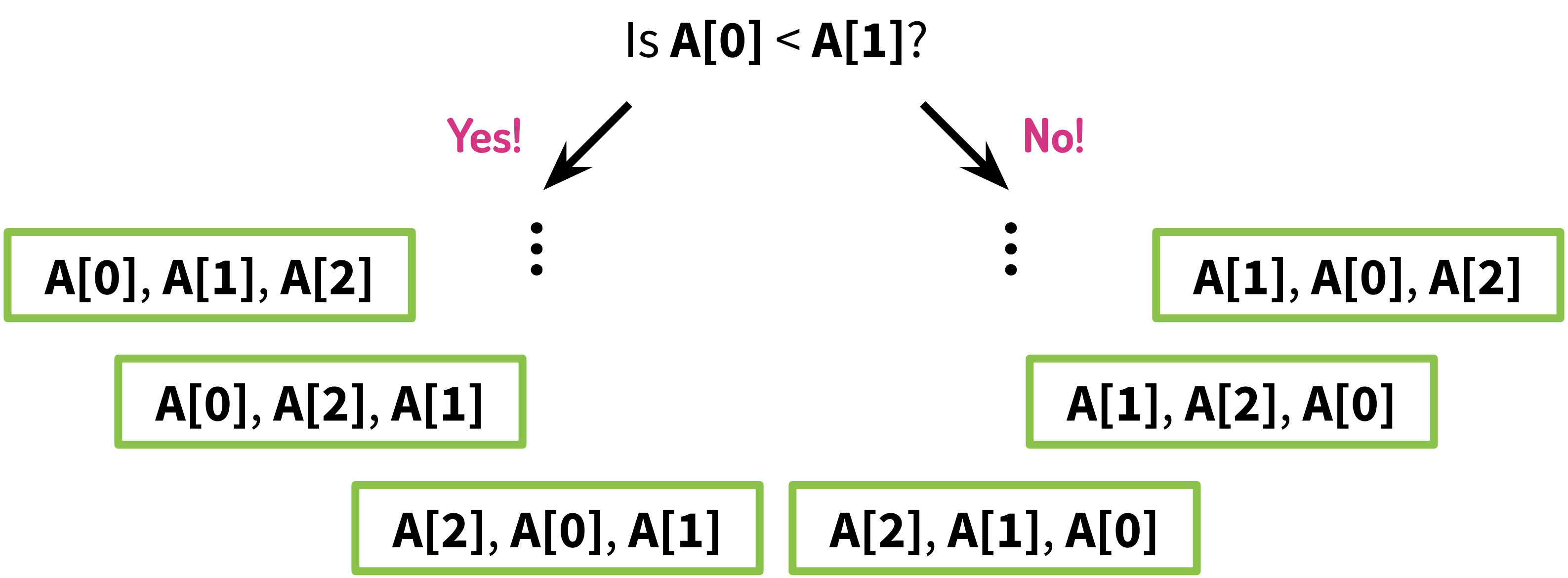
# Comparison-Based Sorting

- Any deterministic comparison-based sorting algorithm requires $\Omega(n \log(n))$-time.
  - Suppose we want to sort three items in a list A.
  - We can represent the comparisons made by a comparison-based sorting algorithm as a decision tree, where the leaves are all possible orderings.



# Comparison-Based Sorting

- Any deterministic comparison-based sorting algorithm requires $\Omega(n \log(n))$-time.
  - Suppose we want to sort three items in a list A.
  - We can represent the comparisons made by a comparison-based sorting algorithm as a decision tree, where the leaves are all possible orderings.
  - The worst-case runtime must be at least $\Omega$(length of longest path).



# Comparison-Based Sorting

- How long is the longest path?
    - At least how many leaves must this decision tree have?
    - What is the depth of the shallowest tree with this many leaves?

# Comparison-Based Sorting

- How long is the longest path?
  - At least how many leaves must this decision tree have? **n!**
  - What is the depth of the shallowest tree with this many leaves? **log(n!)**
  - The longest path is at least **log(n!)**, so the worst-case runtime must be at least $\Omega(n!) = \Omega(n \log(n))$.

# Comparison-Based Sorting

- Any deterministic comparison-based sorting algorithm requires $\Omega(n \log(n))$-time.
    - Any deterministic comparison-based sorting algorithm can be represented as a decision tree with n! leaves.
    - The worst-case runtime is at least the depth of the decision tree.
    - All decision trees with n! leaves have depth $\Omega(n \log(n))$.
    - Therefore, any deterministic comparison-based sorting algorithm requires $\Omega(n \log(n))$-time.

# Is Linear-Time Sorting Nonsense?

- If any deterministic comparison-based sorting algorithm requires $\Omega(n \log(n))$-time, then what's this nonsense about linear-time sorting algorithms?
    - We can achieve $O(n)$ worst-case runtime if we make assumptions about the input.
    - e.g. They are integers ranging from 0 to k-1.

# Counting Sort

```python
def counting_sort(A, k):
  # A consists of n integers ranging from 0 to k-1
```

# Counting Sort

```
def counting_sort(A, k):
  # A consists of n integers ranging from 0 to k-1
  counts = [0] * k
  for i in range(len(A)):
    counts[A[i]] += 1
```

# Counting Sort

```
def counting_sort(A, k):
  # A consists of n integers ranging from 0 to k-1
  counts = [0] * k
  for i in range(len(A)):
    counts[A[i]] += 1
  result = []
  for i in range(k):
    # Extends result by counts[i] i's
    result.extend([i] * counts[i])
```

# Counting Sort

```
def counting_sort(A, k):
  # A consists of n integers ranging from 0 to k-1
  counts = [0] * k
  for i in range(len(A)):
    counts[A[i]] += 1
  result = []
  for i in range(k):
    # Extends result by counts[i] i's
    result.extend([i] * counts[i])
  return result
```

# Counting Sort

```python
def counting_sort(A, k):
  # A consists of n integers ranging from 0 to k-1
  counts = [0] * k
  for i in range(len(A)):
    counts[A[i]] += 1
  result = []
  for i in range(k):
    # Extends result by counts[i] i's
    result.extend([i] * counts[i])
  return result
```

**<u>Worst-case</u> runtime** $\Theta(n+k)$

# Counting Sort

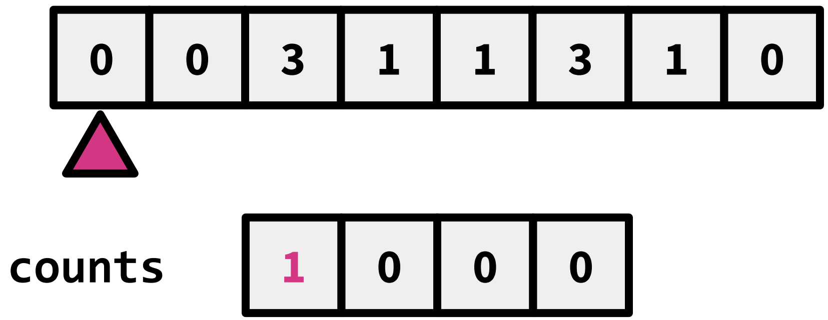
- Suppose **A** consists of 8 integers ranging from 0 to 3

| | | | | | | | |
|---|---|---|---|---|---|---|---|
| 0 | 0 | 3 | 1 | 1 | 3 | 1 | 0 |

| | | | | |
|---|---|---|---|---|
| `counts` | 0 | 0 | 0 | 0 |

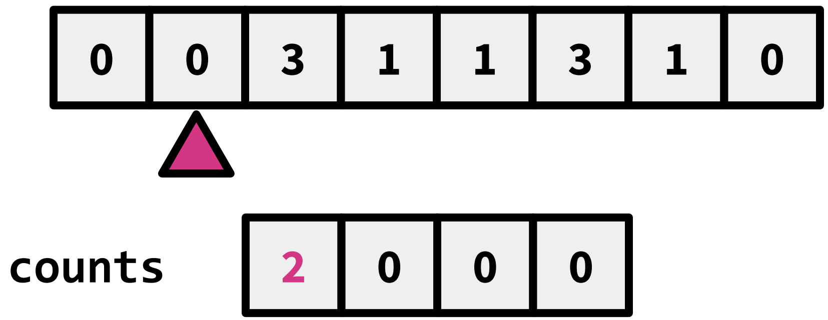
# Counting Sort

- Suppose **A** consists of 8 integers ranging from 0 to 3



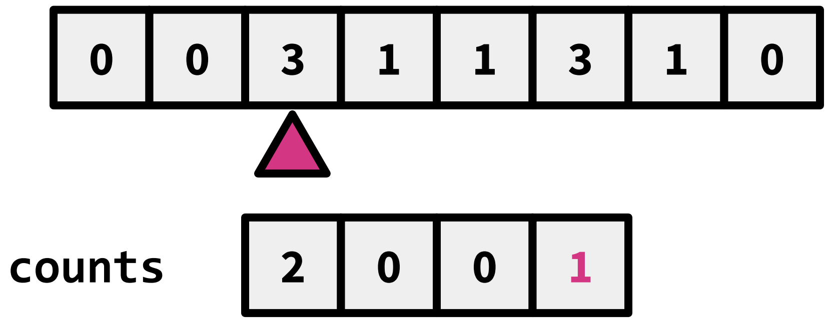
# Counting Sort

- Suppose **A** consists of 8 integers ranging from 0 to 3



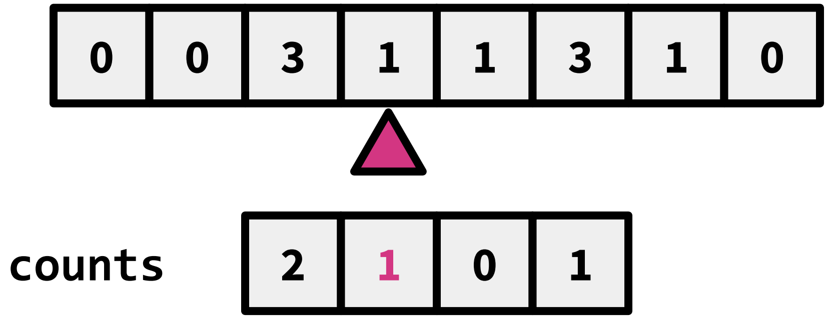
# Counting Sort

- Suppose **A** consists of 8 integers ranging from 0 to 3



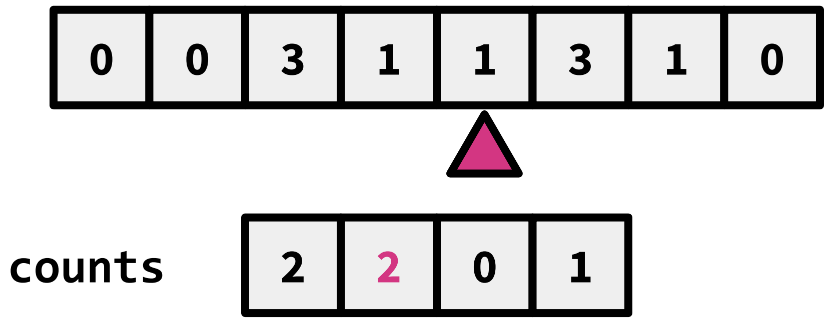
# Counting Sort

- Suppose **A** consists of 8 integers ranging from 0 to 3



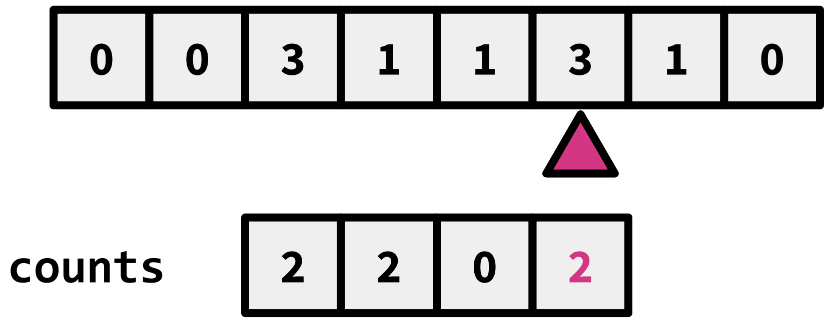
# Counting Sort

- Suppose **A** consists of 8 integers ranging from 0 to 3



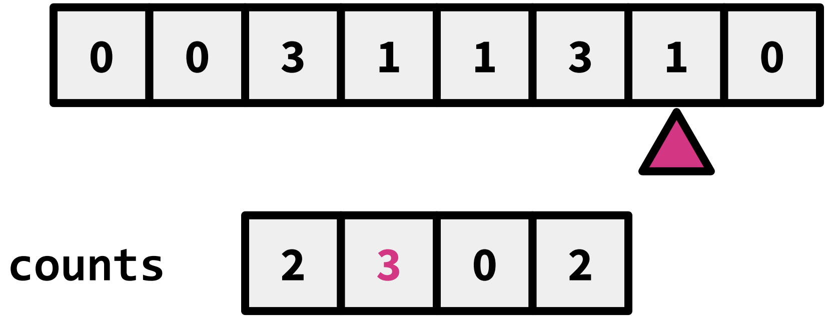
# Counting Sort

- Suppose **A** consists of 8 integers ranging from 0 to 3



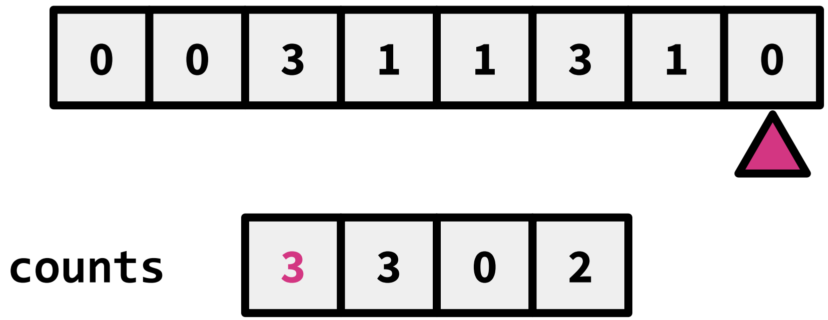
# Counting Sort

- Suppose **A** consists of 8 integers ranging from 0 to 3



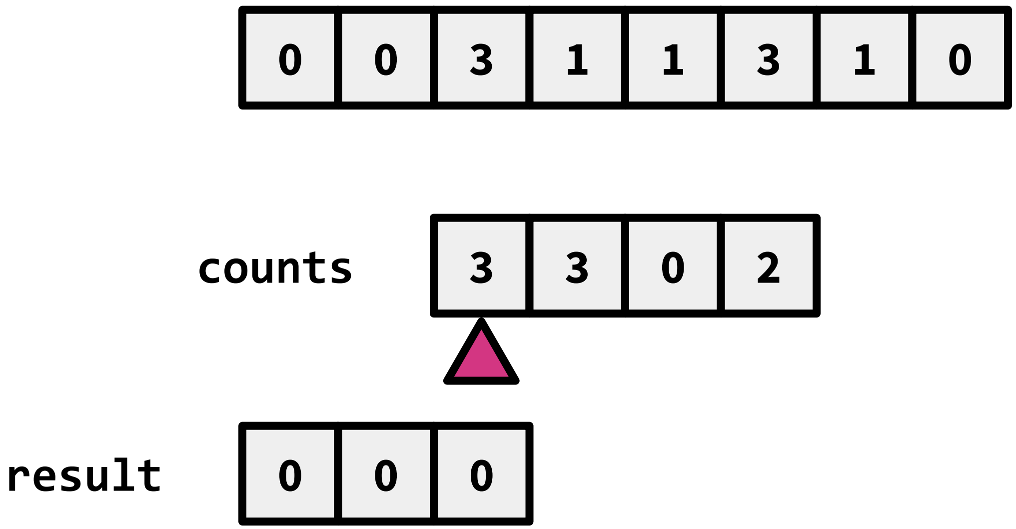
# Counting Sort

- Suppose **A** consists of 8 integers ranging from 0 to 3



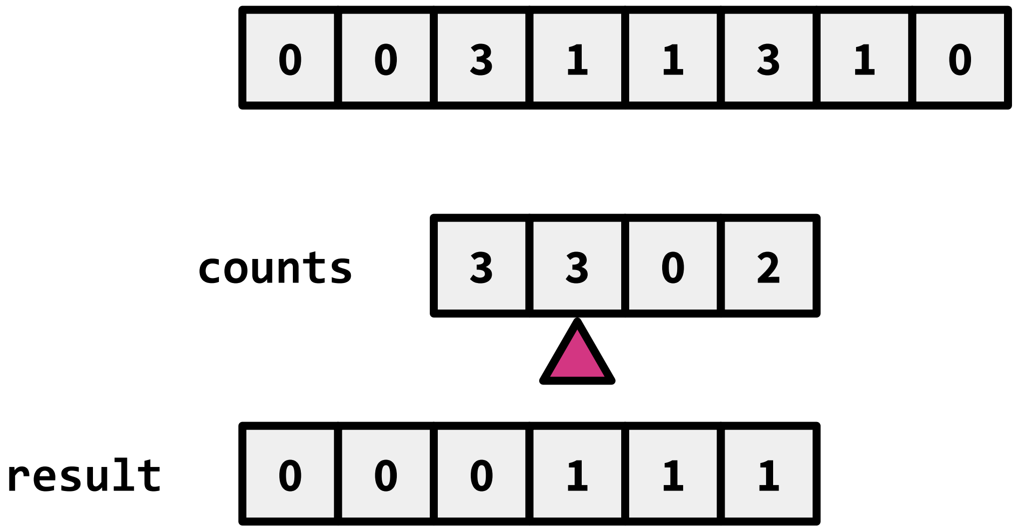
# Counting Sort

- Suppose **A** consists of 8 integers ranging from 0 to 3



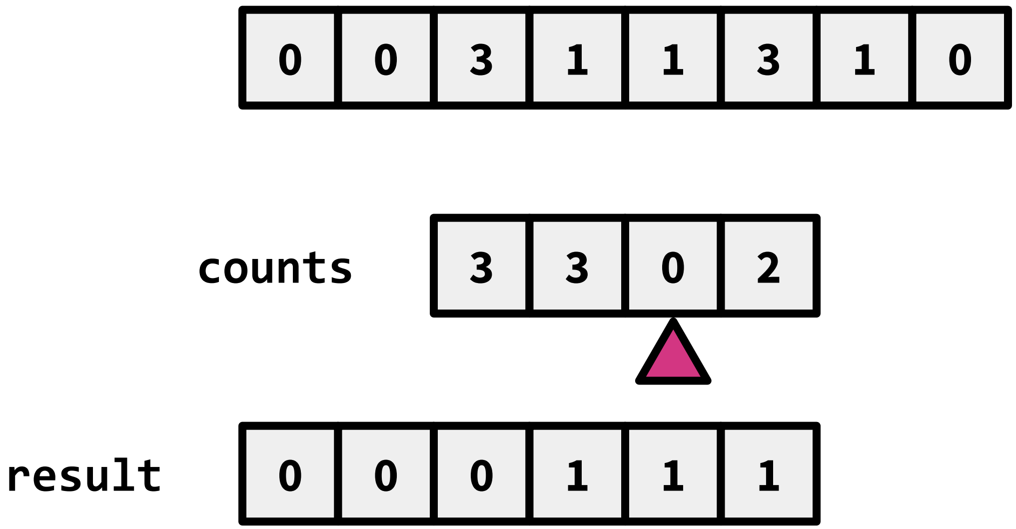
# Counting Sort

- Suppose **A** consists of 8 integers ranging from 0 to 3



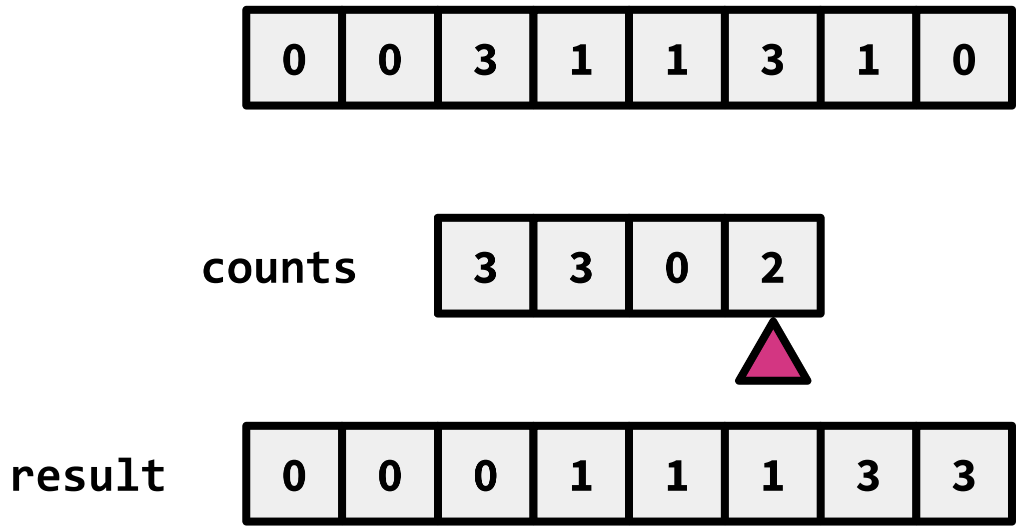
# Counting Sort

- Suppose **A** consists of 8 integers ranging from 0 to 3



# Counting Sort

- Suppose **A** consists of 8 integers ranging from 0 to 3



# Bucket Sort

```
def bucket_sort(A, k, num_buckets):
  # A consists of n integers ranging from 0 to k-1.
  # Pointless to have more buckets than integers.
```

# Bucket Sort

```
def bucket_sort(A, k, num_buckets):
  # A consists of n integers ranging from 0 to k-1.
  # Pointless to have more buckets than integers.
  num_buckets = min(num_buckets, k)
```

# Bucket Sort

```python
def bucket_sort(A, k, num_buckets):
  # A consists of n integers ranging from 0 to k-1.
  # Pointless to have more buckets than integers.
  num_buckets = min(num_buckets, k)
  buckets = [[] for i in range(num_buckets)]
```

# Bucket Sort

```
def bucket_sort(A, k, num_buckets):
  # A consists of n integers ranging from 0 to k-1.
  # Pointless to have more buckets than integers.
  num_buckets = min(num_buckets, k)
  buckets = [[] for i in range(num_buckets)]
  for i in range(len(A)):
    b = get_bucket(A[i], k, num_buckets)
    buckets[b].append(A[i])
```

# Bucket Sort

```
def bucket_sort(A, k, num_buckets):
  # A consists of n integers ranging from 0 to k-1.
  # Pointless to have more buckets than integers.
  num_buckets = min(num_buckets, k)
  buckets = [[] for i in range(num_buckets)]
  for i in range(len(A)):
    b = get_bucket(A[i], k, num_buckets)
    buckets[b].append(A[i])
  result = []
  if num_buckets < k:
    for j in range(num_buckets):
      result.extend(stable_sort(buckets[j]))
  else:
    for j in range(num_buckets):
      result.extend(buckets[j])
```

# Bucket Sort

```
def bucket_sort(A, k, num_buckets):
  # A consists of n integers ranging from 0 to k-1.
  # Pointless to have more buckets than integers.
  num_buckets = min(num_buckets, k)
  buckets = [[] for i in range(num_buckets)]
  for i in range(len(A)):
    b = get_bucket(A[i], k, num_buckets)
    buckets[b].append(A[i])
  result = []
  if num_buckets < k:
    for j in range(num_buckets):
      result.extend(stable_sort(buckets[j]))
  else:
    for j in range(num_buckets):
      result.extend(buckets[j])
  return result
```

# Bucket Sort

```python
def bucket_sort(A, k, num_buckets):
  # A consists of n integers ranging from 0 to k-1.
  # Pointless to have more buckets than integers.
  num_buckets = min(num_buckets, k)
  buckets = [[] for i in range(num_buckets)]
  for i in range(len(A)):
    b = get_bucket(A[i], k, num_buckets)
    buckets[b].append(A[i])
  result = []
  if num_buckets < k:
    for j in range(num_buckets):
      result.extend(stable_sort(buckets[j]))
  else:
    for j in range(num_buckets):
      result.extend(buckets[j])
  return result
```

**<u>Worst-case</u> runtime** $\Theta(\max\{n\log(n), n+k\})$

# Bucket Sort

```
def bucket_sort(A, k, num_buckets):
  # A consists of n integers ranging from 0 to k-1.
  # Pointless to have more buckets than integers.
  num_buckets = min(num_buckets, k)
  buckets = [[] for i in range(num_buckets)]
  for i in range(len(A)):
    b = get_bucket(A[i], k, num_buckets)
    buckets[b].append(A[i])
  result = []
  if num_buckets < k:
    for j in range(num_buckets):
      result.extend(stable_sort(buckets[j]))
  else:
    for j in range(num_buckets):
      result.extend(buckets[j])
  return result
```

A stable sort keeps equal elements in the same order:
$[1^{(a)}, 3, 2, 1^{(b)}] \Rightarrow [1^{(a)}, 1^{(b)}, 2, 3]$

**Worst-case runtime** $\Theta(\max\{n \log(n), n+k\})$

# Bucket Sort

```
def bucket_sort(A, k, num_buckets):
  # A consists of n integers ranging from 0 to k-1.
  # Pointless to have more buckets than integers.
  num_buckets = min(num_buckets, k)
  buckets = [[] for i in range(num_buckets)]
  for i in range(len(A)):
    b = get_bucket(A[i], k, num_buckets)
    buckets[b].append(A[i])
  result = []
  if num_buckets < k:
    for j in range(num_buckets):
      result.extend(stable_sort(buckets[j]))
  else:
    for j in range(num_buckets):
      result.extend(buckets[j])
  return result
```

A stable sort keeps equal elements in the same order:
$[1^{(a)}, 3, 2, 1^{(b)}] \Rightarrow [1^{(a)}, 1^{(b)}, 2, 3]$

**<u>Worst-case</u> runtime** $\Theta(\max\{n \log(n), n+k\})$

Happens if k > num_buckets and all of the values end up in the same bucket.

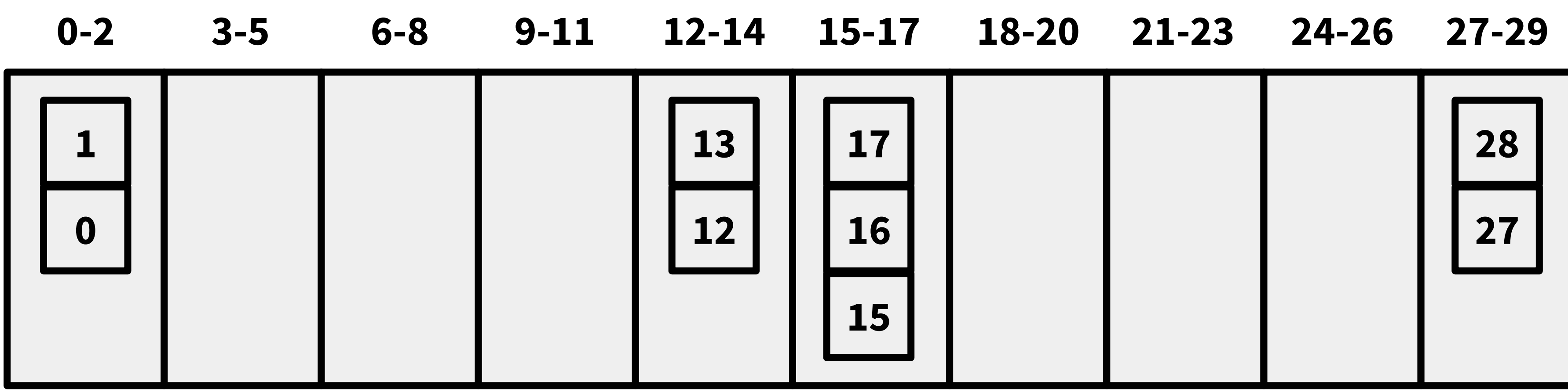
# Bucket Sort

```python
def get_bucket(value, k, num_buckets):
  # The implementation of this function varies
  # depending on the predefined bucketing scheme;
  # the following examples rely on use of int
  # division.
  return value / math.ceil(k / num_buckets)
```

**Worst-case runtime** $\Theta(1)$

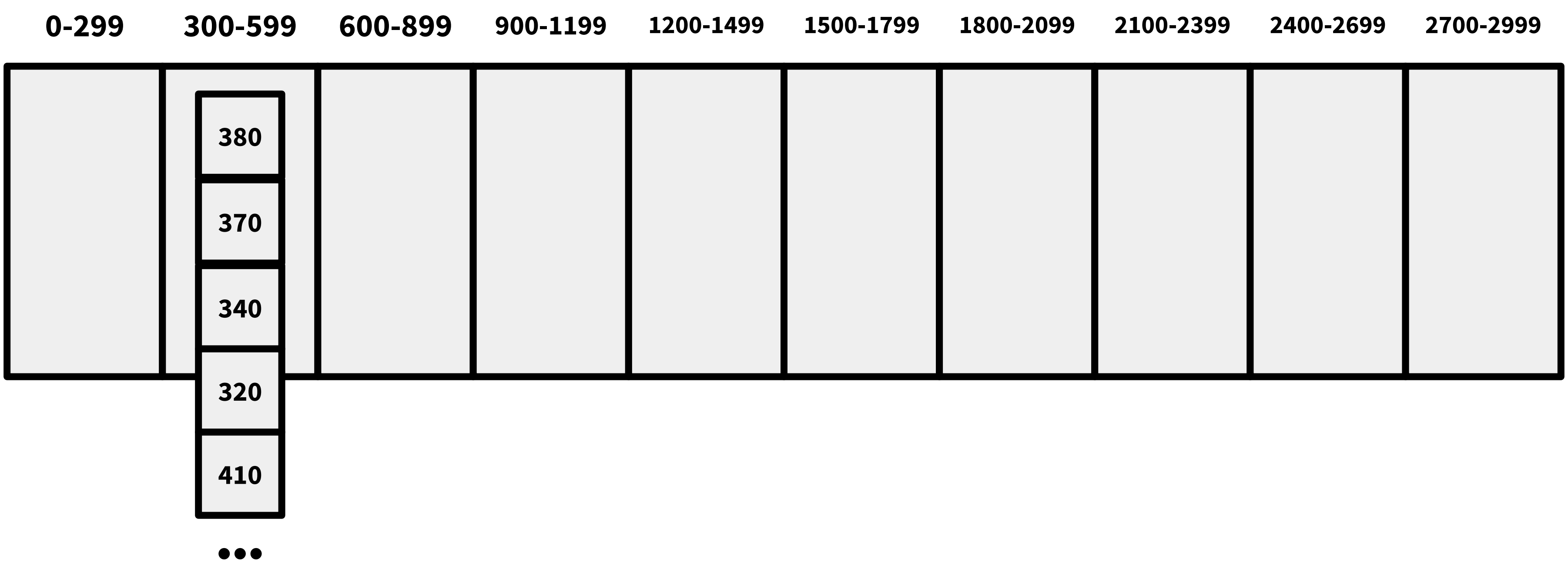
# Bucket Sort

- Two cases for num_buckets and k:
  - **k ≤ num_buckets** At most one key per bucket, so buckets don't need another stable_sort to be sorted (similar to counting_sort).
  - **k > num_buckets** Might be multiple keys per bucket, so buckets need another stable_sort to be sorted.
- Suppose k = 30 and num_buckets = 10. Then we group keys 0 to 2 in the same bucket, 3 to 5 in the same bucket, etc.
  - A = [17, 13, 16, 12, 15, 1, 28, 0, 27] produces:



# Bucket Sort

- In an extreme case, a bucket might receive all of the inserted keys.
- Suppose k = 3000 and num_buckets = 10.
  - A = [380, 370, 340, 320, 410, …] would need to stable_sort all of the elements in the original list since they all fall in the same bucket.



# Radix Sort

```
def radix_sort(A, d, k):
  # A consists of n d-digit integers
  # with digits ranging from 0 to k-1.
```

# Radix Sort

```python
def radix_sort(A, d, k):
  # A consists of n d-digit integers
  # with digits ranging from 0 to k-1.
  for i in range(d):
    # Creates list of (value's digit i, value) pairs
    # For i = 0: [23, 4, 51, 76, 8] =>
    # [(3, 23), (4, 4), (1, 51), (6, 76), (8, 8)]
    A_pairs = make_pairs(A, k, i)
```

# Radix Sort

```
def radix_sort(A, d, k):
  # A consists of n d-digit integers
  # with digits ranging from 0 to k-1.
  for i in range(d):
    # Creates list of (value's digit i, value) pairs
    # For i = 0: [23, 4, 51, 76, 8] =>
    # [(3, 23), (4, 4), (1, 51), (6, 76), (8, 8)]
    A_pairs = make_pairs(A, k, i)

    # Bucket sorts according to first element of
    # pair and returns a list of values
    A_new = bucket_sort_with_pairs(A_pairs, k, k)
    A = A_new
```

# Radix Sort

```
def radix_sort(A, d, k):
  # A consists of n d-digit integers
  # with digits ranging from 0 to k-1.
  for i in range(d):
    # Creates list of (value's digit i, value) pairs
    # For i = 0: [23, 4, 51, 76, 8] =>
    # [(3, 23), (4, 4), (1, 51), (6, 76), (8, 8)]
    A_pairs = make_pairs(A, k, i)

    # Bucket sorts according to first element of
    # pair and returns a list of values
    A_new = bucket_sort_with_pairs(A_pairs, k, k)
    A = A_new
  return result
```

# Radix Sort

```
def radix_sort(A, d, k):
  # A consists of n d-digit integers
  # with digits ranging from 0 to k-1.
  for i in range(d):
    # Creates list of (value's digit i, value) pairs
    # For i = 0: [23, 4, 51, 76, 8] =>
    # [(3, 23), (4, 4), (1, 51), (6, 76), (8, 8)]
    A_pairs = make_pairs(A, k, i)

    # Bucket sorts according to first element of
    # pair and returns a list of values
    A_new = bucket_sort_with_pairs(A_pairs, k, k)
    A = A_new
  return result
```

**Worst-case** runtime $\Theta(d(n+k))$

# Radix Sort

```python
def make_pairs(A, k, i):
  result = []
  for a in A:
    # e.g. a=1023, k=10, i=1: (1023/(10**1))%10 = 2
    key = (a / (k ** i)) % k
    result.append((key, a))
  return result
```

**Worst-case runtime** $\Theta(n)$

# Radix Sort

- Suppose **A** consists of eight 3-digit integers, with digits ranging from 0 to 9. Calling `radix_sort(A, 3, 10)`:

| | | | | | | | | |
|---|---|---|---|---|---|---|---|---|
| **A** | 31 | 5 | 210 | 14 | 95 | 477 | 555 | 125 |

| | |
|---|---|
| **i** | 0 |

# Radix Sort

- Suppose **A** consists of eight 3-digit integers, with digits ranging from 0 to 9. Calling **`radix_sort(A, 3, 10)`**:

**A**

| | | | | | | | |
|---|---|---|---|---|---|---|---|
| 31 | 5 | 210 | 14 | 95 | 477 | 555 | 125 |

**i**

| |
|---|
| 0 |

**A_pairs**

| | | | | | |
|---|---|---|---|---|---|
| (1, 031) | (5, 005) | (0, 210) | (4, 014) | ... | (5, 125) |

**A_new**

| | | | | | | | |
|---|---|---|---|---|---|---|---|
| 210 | 031 | 014 | 005 | 095 | 555 | 125 | 477 |

# Radix Sort

- Suppose **`A`** consists of eight 3-digit integers, with digits ranging from 0 to 9. Calling **`radix_sort(A, 3, 10)`**:

| | | | | | | | | |
|---|---|---|---|---|---|---|---|---|
| **A** | 210 | 031 | 014 | 005 | 095 | 555 | 125 | 477 |

| | |
|---|---|
| **i** | 1 |

# Radix Sort

- Suppose **A** consists of eight 3-digit integers, with digits ranging from 0 to 9. Calling **`radix_sort(A, 3, 10)`**:

**A**

| | | | | | | | |
|---|---|---|---|---|---|---|---|
| 210 | 031 | 014 | 005 | 095 | 555 | 125 | 477 |

**i**

| |
|---|
| 1 |

**A_pairs**

| | | | | | |
|---|---|---|---|---|---|
| (1, 210) | (3, 031) | (1, 014) | (0, 005) | ... | (7, 477) |

**A_new**

| | | | | | | | |
|---|---|---|---|---|---|---|---|
| 005 | 210 | 014 | 125 | 031 | 555 | 477 | 095 |

# Radix Sort

- Suppose **A** consists of eight 3-digit integers, with digits ranging from 0 to 9. Calling `radix_sort(A, 3, 10)`:

| | | | | | | | | |
|---|---|---|---|---|---|---|---|---|
| **A** | 005 | 210 | 014 | 125 | 031 | 555 | 477 | 095 |

| | |
|---|---|
| **i** | 2 |

# Radix Sort

- Suppose **A** consists of eight 3-digit integers, with digits ranging from 0 to 9. Calling `radix_sort(A, 3, 10)`:

| | | | | | | | | |
|---|---|---|---|---|---|---|---|---|
| **A** | 005 | 210 | 014 | 125 | 031 | 555 | 477 | 095 |

| | |
|---|---|
| **i** | 2 |

| | | | | | | |
|---|---|---|---|---|---|---|
| **A_pairs** | (0, 005) | (2, 210) | (0, 014) | (1, 125) | ... | (0, 095) |

| | | | | | | | | |
|---|---|---|---|---|---|---|---|---|
| **A_new** | 005 | 014 | 031 | 095 | 125 | 210 | 477 | 555 |

# Radix Sort

- Proof of correctness
  - **Inductive hypothesis:** At the start of iteration i, **A** is sorted by its i least-significant digits.
  - **Base case:** At the start of the first iteration of the loop, **A** is trivially sorted by its 0 least-significant digits.
  - **Inductive step:** Since **`bucket_sort`** is stable, the elements within each bucket are still sorted by their i least-significant digits. **`bucket_sort`** sorts **A** by the i+1 digit of the elements, so the elements are sorted by their i+1 least-significant digits.
  - **Conclusion:** The loop terminates at the start of iteration d. The collection of d-digit integers in **A** are sorted by their d least-significant digits, which implies that **A** is sorted.

# Comparison-Based vs. Linear-Time Sorting

- Why would we ever use a comparison-based sorting algorithm?
  - **It has lots of precision...**

| | | | | | | |
|---|---|---|---|---|---|---|
| $\pi$ | $\frac{1234}{9876}$ | $e$ | 37! | 6.0221409 | $n^n$ | 42 |

# Comparison-Based vs. Linear-Time Sorting

- Why would we ever use a comparison-based sorting algorithm?
  - **It has lots of precision…**

| | | | | | | |
|---|---|---|---|---|---|---|
| $\pi$ | $\frac{1234}{9876}$ | $e$ | 37! | 6.0221409 | $n^n$ | 42 |

We can compare these pretty quickly (just look at their most significant digit):

- $\pi = 3.14159\ldots$
- $e = 2.71818\ldots$

But **`radix_sort`** requires us to look at all digits, which is problematic—both have infinitely many!

# Comparison-Based vs. Linear-Time Sorting

- Why would we ever use a comparison-based sorting algorithm?
  - **It has lots of precision…**

| | | | | | | |
|---|---|---|---|---|---|---|
| **π** | $\frac{1234}{9876}$ | **e** | **37!** | **6.0221409** | $n^n$ | **42** |

We can compare these pretty quickly (just look at their most significant digit):

- π = 3.14159…
- e = 2.71818…

But **`radix_sort`** requires us to look at all digits, which is problematic—both have infinitely many!

Even with integers, if it's really big, **`radix_sort`** is slow.

# Comparison-Based vs. Linear-Time Sorting

- Why would we ever use a comparison-based sorting algorithm?
  - **It has lots of precision…**

| | | | | | | |
|---|---|---|---|---|---|---|
| $\pi$ | $\frac{1234}{9876}$ | $e$ | 37! | 6.0221409 | $n^n$ | 42 |

We can compare these pretty quickly (just look at their most significant digit):

- $\pi = 3.14159\ldots$
- $e = 2.71818\ldots$

But **`radix_sort`** requires us to look at all digits, which is problematic—both have infinitely many!

Even with integers, if it's really big, **`radix_sort`** is slow.

  - **`radix_sort`** needs extra memory for the buckets (not in-place).
  - Need to know ordering and buckets ahead of time for linear-time sorting.

# Overview

- The difficulty of a problem depends on the model of computation.
  - In a comparison-based sorting model, algorithms must use at least $\Omega(n \log(n))$ operations.
  - But if we're sorting small integers (or other reasonable data), **`bucket_sort`** and **`radix_sort`** both run in linear-time, given predefined buckets and ordering.

# Today's Outline

- Finish Divide and Conquer II properly
  - ~~Linear-time selection~~ **Done!**
  - ~~Proving runtime with substitution method~~ **Done!**
- Linear-Time Sorting
  - ~~Comparison-based sorting lower bounds~~ **Done!**
  - ~~*Algorithms: Counting sort, bucket sort, and radix sort*~~ **Done!**
  - Reading: CLRS: 8.1-8.2