# Lecture 6: Sorting Lower Bound and "Linear-Time" Sorting

Jessica Sorrell

September 11, 2025
601.433/633 Introduction to Algorithms
Slides by Michael Dinitz

# Introduction

Lots of ways of sorting in $\boldsymbol{O(n \log n)}$ time: mergesort, heapsort, randomized quicksort, deterministic quicksort with BPFRT pivot selection, . . .

Is it possible to do better?

# Introduction

Lots of ways of sorting in $\boldsymbol{O(n \log n)}$ time: mergesort, heapsort, randomized quicksort, deterministic quicksort with BPFRT pivot selection, ...

Is it possible to do better? No!

# Introduction

Lots of ways of sorting in $\boldsymbol{O(n \log n)}$ time: mergesort, heapsort, randomized quicksort, deterministic quicksort with BPFRT pivot selection, ...

Is it possible to do better? No! And yes!

# Introduction

Lots of ways of sorting in $O(n \log n)$ time: mergesort, heapsort, randomized quicksort, deterministic quicksort with BPFRT pivot selection, ...

Is it possible to do better? No! And yes!

*Comparison Model*: we are given a constant-time algorithm which can compare any two elements. No other information about elements.

- All algorithms we've seen so far have been in this model

# Introduction

Lots of ways of sorting in $O(n \log n)$ time: mergesort, heapsort, randomized quicksort, deterministic quicksort with BPFRT pivot selection, ...

Is it possible to do better? No! And yes!

*Comparison Model*: we are given a constant-time algorithm which can compare any two elements. No other information about elements.

- All algorithms we've seen so far have been in this model

No: every algorithm in the comparison model must have worst-case running time $\Omega(n \log n)$.

Yes: If we assume extra structure for the elements, can do sorting in $O(n)$ time*

# Sorting Lower Bound

# Statement

**Theorem**

*Any sorting algorithm in the comparison model must make at least* $\log(n!) = \Theta(n \log n)$ *comparisons (in the worst case).*

Lower bound on the number of comparisons – running time could be even worse!

Allows algorithm to reorder elements, copy them, move them, etc. for free.

# Statement

**Theorem**

*Any sorting algorithm in the comparison model must make at least* $\log(n!) = \Theta(n \log n)$ *comparisons (in the worst case).*

Lower bound on the number of comparisons – running time could be even worse!
Allows algorithm to reorder elements, copy them, move them, etc. for free.

Why is this hard?

- Lower bound needs to hold for *all* algorithms
- How can we simultaneously reason about algorithms as different as mergesort, quicksort, heapsort, ...?

# Sorting as Permutations

Think of an array $A$ as a *permutation*: $A[i]$ is the $\pi(i)$'th smallest element

$$A = [23, 14, 2, 5, 76]$$

Corresponds to $\pi = (3, 2, 0, 1, 4)$:

$$\pi(0) = 3 \qquad \pi(1) = 2 \qquad \pi(2) = 0 \qquad \pi(3) = 1 \qquad \pi(4) = 4$$

# Sorting as Permutations

Think of an array $A$ as a *permutation*: $A[i]$ is the $\pi(i)$'th smallest element

$$A = [23, 14, 2, 5, 76]$$

Corresponds to $\pi = (3, 2, 0, 1, 4)$:

$$\pi(0) = 3 \qquad \pi(1) = 2 \qquad \pi(2) = 0 \qquad \pi(3) = 1 \qquad \pi(4) = 4$$

**Lemma**

*Given $A$ with $|A| = n$, if can sort in $T(n)$ comparisons then can find $\pi$ in $T(n)$ comparisons*

# Sorting As Permutations (cont'd)

## Lemma

*Given* $A$ *with* $|A| = n$*, if can sort in* $T(n)$ *comparisons then can find* $\pi$ *in* $T(n)$ *comparisons*

## Proof Sketch.

- "Tag" each element of $A$ with index:
  $[23, 14, 2, 5, 76] \to [(23, 0), (14, 1), (2, 2), (5, 3), (76, 4)]$
- Sort tagged $A$ into tagged $B$ with $T(n)$ comparisons:
  $[(2, 2), (5, 3), (14, 1), (23, 0), (76, 4)]$
- Iterate through to get $\pi$: $\pi(2) = 0, \pi(3) = 1, \pi(1) = 2, \pi(0) = 3, \pi(4) = 4$ □

# Sorting As Permutations (cont'd)

## Lemma

*Given $A$ with $|A| = n$, if can sort in $T(n)$ comparisons then can find $\pi$ in $T(n)$ comparisons*

## Proof Sketch.

- "Tag" each element of $A$ with index:
  $[23, 14, 2, 5, 76] \to [(23, 0), (14, 1), (2, 2), (5, 3), (76, 4)]$
- Sort tagged $A$ into tagged $B$ with $T(n)$ comparisons:
  $[(2, 2), (5, 3), (14, 1), (23, 0), (76, 4)]$
- Iterate through to get $\pi$: $\pi(2) = 0, \pi(3) = 1, \pi(1) = 2, \pi(0) = 3, \pi(4) = 4$ □

## Corollary

*If need at least $T(n)$ comparisons to find $\pi$, need at least $T(n)$ comparisons to sort!*

# Generic Algorithm

Want to show that it takes $\Omega(n \log n)$ comparisons to find $\pi$ in comparison model.

- Only comparisons cost us anything!

# Generic Algorithm

Want to show that it takes $\Omega(n \log n)$ comparisons to find $\pi$ in comparison model.

- Only comparisons cost us anything!

Arbitrary algorithm:

- Starts with some comparison (e.g., compares $A[0]$ to $A[1]$)
- Rules out some possible permutations!
  - If $A[0] < A[1]$ then $\pi(0) < \pi(1)$
  - If $A[0] > A[1]$ then $\pi(1) > \pi(0)$
- Depending on outcome, choose next comparison to make.
- Continue until only one possible permutation.

# Generic Algorithm

Want to show that it takes $\Omega(n \log n)$ comparisons to find $\pi$ in comparison model.

- Only comparisons cost us anything!

Arbitrary algorithm:

- Starts with some comparison (e.g., compares $A[0]$ to $A[1]$)
- Rules out some possible permutations!
  - If $A[0] < A[1]$ then $\pi(0) < \pi(1)$
  - If $A[0] > A[1]$ then $\pi(1) > \pi(0)$
- Depending on outcome, choose next comparison to make.
- Continue until only one possible permutation.

Remind you of anything?

# Decision Trees

Model any algorithm as a *binary decision tree*

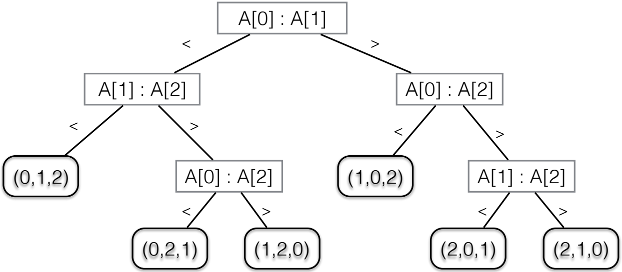
- Internal nodes: comparisons
- Leaves: permutations

# Decision Trees

Model any algorithm as a *binary decision tree*

- Internal nodes: comparisons
- Leaves: permutations

Example: $n = 3$. Six possible permutations.

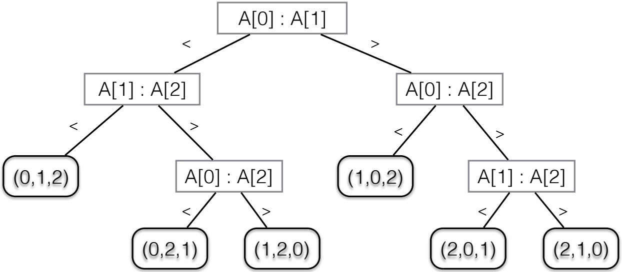


# Decision Trees

Model any algorithm as a *binary decision tree*

- Internal nodes: comparisons
- Leaves: permutations

Example: $n = 3$. Six possible permutations.



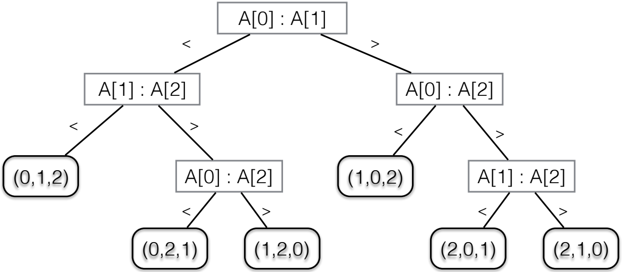

Max # comparisons:

# Decision Trees

Model any algorithm as a *binary decision tree*

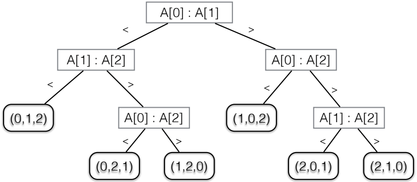
- Internal nodes: comparisons
- Leaves: permutations

Example: $n = 3$. Six possible permutations.



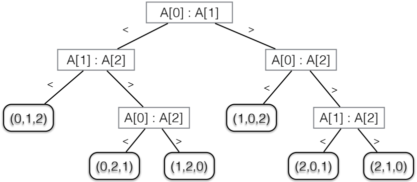

Max # comparisons: **3**

# Finishing Up




Scale to general $n$. Consider arbitrary decision tree.

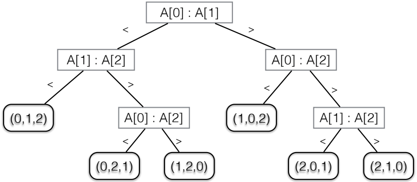
# Finishing Up




Scale to general $n$. Consider arbitrary decision tree.

Max # comparisons

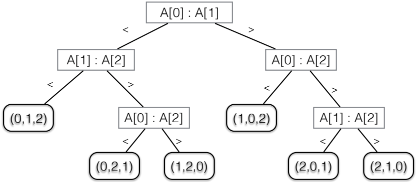
# Finishing Up




Scale to general $n$. Consider arbitrary decision tree.

Max # comparisons = depth of tree

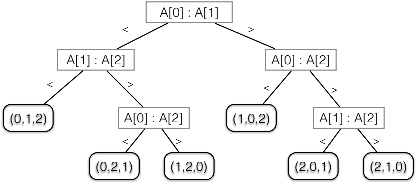
# Finishing Up




Scale to general $n$. Consider arbitrary decision tree.

$$\text{Max } \# \text{ comparisons} = \text{depth of tree}$$
$$\geq \log_2(\# \text{ leaves})$$

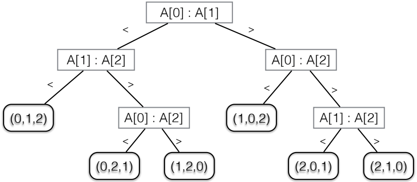
# Finishing Up




Scale to general $n$. Consider arbitrary decision tree.

$$
\begin{aligned}
\text{Max \# comparisons} &= \text{depth of tree} \\
&\geq \log_2(\#\ \text{leaves}) \\
&= \log_2(n!)
\end{aligned}
$$

# Finishing Up




Scale to general $n$. Consider arbitrary decision tree.

$$
\begin{aligned}
\text{Max \# comparisons} &= \text{depth of tree} \\
&\geq \log_2(\#\text{ leaves}) \\
&= \log_2(n!) \\
&= \Theta(n \log n)
\end{aligned}
$$

# Sorting Lower Bound Summary

**Theorem**

*Every sorting algorithm in the comparison model must make at least* $\log(n!) = \Theta(n \log n)$ *comparisons (in the worst case).*

**Proof Sketch.**

1. Lower bound on finding permutation $\pi \implies$ lower bound on sorting
2. Any algorithm for finding $\pi$ is a binary decision tree with $n!$ leaves.
3. Any binary decision tree with $n!$ leaves has depth $\geq \log(n!) = \Theta(n \log n)$

$\implies$ Every algorithm has worst case number of comparisons at least $\Theta(n \log n)$. □

# “Linear-Time” Sorting

# Bypassing the Lower Bound

What if we're *not* in the comparison model?

- Can do more than just compare elements.

Main example: *integers*.

- What is the $\mathbf{3}$rd bit of $\boldsymbol{A}[\mathbf{0}]$?
- Is $\boldsymbol{A}[\mathbf{0}]$ even?

Same ideas apply to letters, strings, etc.

# Counting Sort

Suppose $A$ consists of $n$ integers, all in $\{0, 1, \ldots, k-1\}$.

# Counting Sort

Suppose $A$ consists of $n$ integers, all in $\{0, 1, \ldots, k-1\}$.

*Counting Sort:*

- Maintain an array $B$ of length $k$ initialized to all $0$
- Scan through $A$ and increment $B[A[i]]$.
- Scan through $B$, output $i$ exactly $B[i]$ times.

# Counting Sort

Suppose $A$ consists of $n$ integers, all in $\{0, 1, \ldots, k-1\}$.

*Counting Sort:*

- Maintain an array $B$ of length $k$ initialized to all $0$
- Scan through $A$ and increment $B[A[i]]$.
- Scan through $B$, output $i$ exactly $B[i]$ times.

Correctness: Obvious

# Counting Sort

Suppose $A$ consists of $n$ integers, all in $\{0, 1, \ldots, k-1\}$.

*Counting Sort:*

- Maintain an array $B$ of length $k$ initialized to all $0$
- Scan through $A$ and increment $B[A[i]]$.
- Scan through $B$, output $i$ exactly $B[i]$ times.

Correctness: Obvious

Running time:

# Counting Sort

Suppose $A$ consists of $n$ integers, all in $\{0, 1, \ldots, k-1\}$.

*Counting Sort:*

- Maintain an array $B$ of length $k$ initialized to all $0$
- Scan through $A$ and increment $B[A[i]]$.
- Scan through $B$, output $i$ exactly $B[i]$ times.

Correctness: Obvious

Running time: $O(n+k)$

# Bucket Sort: Counting Sort++

Often want to sort *objects* based on *keys*:

- Each object has a key: integer in $\{0, 1, \ldots, k-1\}$
- $A$ consists of $n$ objects

# Bucket Sort: Counting Sort++

Often want to sort *objects* based on *keys*:

- Each object has a key: integer in $\{0, 1, \dots, k-1\}$
- $A$ consists of $n$ objects

*Bucket Sort*:

- Same idea as counting sort, but $B[i]$ is bucket of objects with key $i$
- Bucket is a linked list with pointers to beginning and end
- Insert at *end* of list, using end pointer.
- For output, go through each bucket in order.

# Bucket Sort: Counting Sort++

Often want to sort *objects* based on *keys*:

- Each object has a key: integer in $\{0, 1, \ldots, k-1\}$
- $A$ consists of $n$ objects

*Bucket Sort*:

- Same idea as counting sort, but $B[i]$ is bucket of objects with key $i$
- Bucket is a linked list with pointers to beginning and end
- Insert at *end* of list, using end pointer.
- For output, go through each bucket in order.

Running time:

# Bucket Sort: Counting Sort++

Often want to sort *objects* based on *keys*:

- Each object has a key: integer in $\{0, 1, \ldots, k-1\}$
- $A$ consists of $n$ objects

*Bucket Sort*:

- Same idea as counting sort, but $B[i]$ is bucket of objects with key $i$
- Bucket is a linked list with pointers to beginning and end
- Insert at *end* of list, using end pointer.
- For output, go through each bucket in order.

Running time: $O(n+k)$

# Bucket Sort: Counting Sort++

Often want to sort *objects* based on *keys*:

- Each object has a key: integer in $\{0, 1, \dots, k-1\}$
- $A$ consists of $n$ objects

*Bucket Sort*:

- Same idea as counting sort, but $B[i]$ is bucket of objects with key $i$
- Bucket is a linked list with pointers to beginning and end
- Insert at *end* of list, using end pointer.
- For output, go through each bucket in order.

Running time: $O(n + k)$

*Stable*: if two objects have same key, order between them after sorting is same as before.

# Radix Sort: Setup

What if $k$ is much larger than $n$, e.g., $k = \Theta(n^2)$?

# Radix Sort: Setup

What if $\boldsymbol{k}$ is much larger than $\boldsymbol{n}$, e.g., $\boldsymbol{k} = \boldsymbol{\Theta}(\boldsymbol{n}^2)$?

*Radix sort*: $\boldsymbol{O}(\boldsymbol{n})$ time* for this case

# Radix Sort: Setup

What if $k$ is much larger than $n$, e.g., $k = \Theta(n^2)$?

*Radix sort*: $O(n)$ time* for this case

Setup:

- Numbers represented base 10 for historical reasons (all works fine in binary)
- Assume all numbers have exactly $d$ digits (for simplicity)

# Radix Sort: Setup

What if $\boldsymbol{k}$ is much larger than $\boldsymbol{n}$, e.g., $\boldsymbol{k} = \boldsymbol{\Theta}(\boldsymbol{n}^2)$?

*Radix sort*: $\boldsymbol{O}(\boldsymbol{n})$ time* for this case

Setup:

- Numbers represented base 10 for historical reasons (all works fine in binary)
- Assume all numbers have exactly $\boldsymbol{d}$ digits (for simplicity)

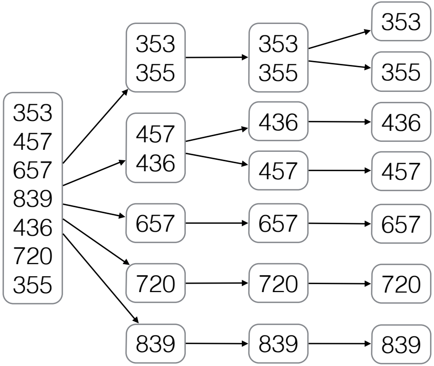
If you were sorting cards, with a number on each card, what might you do?

# Radix Sort: Algorithm

Divide into **10** buckets by first digit, recurse on each bucket by second-digit, etc.

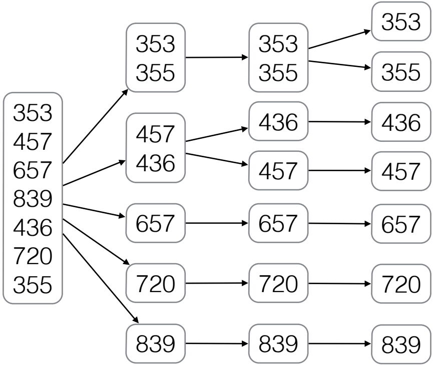
# Radix Sort: Algorithm

Divide into **10** buckets by first digit, recurse on each bucket by second-digit, etc.



# Radix Sort: Algorithm

Divide into **10** buckets by first digit, recurse on each bucket by second-digit, etc.



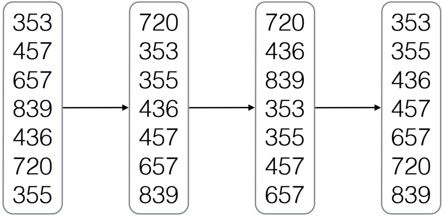

Works, but clunky

# Radix-Sort: Algorithm (II)

More elegant (and surprising): one bucket, sorting from *least* significant digit to *most*!

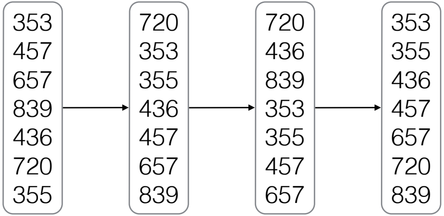
# Radix-Sort: Algorithm (II)

More elegant (and surprising): one bucket, sorting from *least* significant digit to *most*!



# Radix-Sort: Algorithm (II)

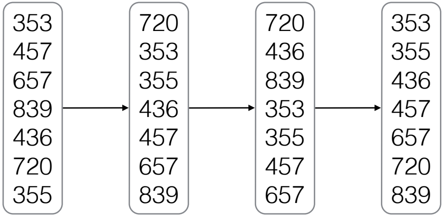
More elegant (and surprising): one bucket, sorting from *least* significant digit to *most*!




For iteration $\boldsymbol{i}$, use bucket sort where key is $\boldsymbol{i}$'th digit and object is number.

# Radix-Sort: Algorithm (II)

More elegant (and surprising): one bucket, sorting from *least* significant digit to *most*!




For iteration $i$, use bucket sort where key is $i$'th digit and object is number.

**Theorem**

*Radix sort from least significant to most significant is correct if the sort used on each digit is stable.*

# Least-Significant Radix Sort: Correctness

**Proof.**

Claim: After $i$'th iteration, correctly sorted by last $i$ digits (interpreted as # in $[0, 10^i - 1]$).

# Least-Significant Radix Sort: Correctness

**Proof.**

Claim: After $i$'th iteration, correctly sorted by last $i$ digits (interpreted as # in $[0, 10^i - 1]$).
Induction on $i$.

# Least-Significant Radix Sort: Correctness

**Proof.**

Claim: After $i$'th iteration, correctly sorted by last $i$ digits (interpreted as # in $[0, 10^i - 1]$).
Induction on $i$.

Base case: After first iteration, correctly sorted by last digit

# Least-Significant Radix Sort: Correctness

## Proof.

Claim: After $i$'th iteration, correctly sorted by last $i$ digits (interpreted as # in $[0, 10^i - 1]$).
Induction on $i$.

Base case: After first iteration, correctly sorted by last digit

Induction:

- Suppose correct for $i$
- After $i + 1$ sort:

# Least-Significant Radix Sort: Correctness

**Proof.**

Claim: After $i$'th iteration, correctly sorted by last $i$ digits (interpreted as # in $[0, 10^i - 1]$).
Induction on $i$.

Base case: After first iteration, correctly sorted by last digit

Induction:

- Suppose correct for $i$
- After $i + 1$ sort:
  - If two numbers have different $i + 1$ digits, now correct.
  - If two number have same $i + 1$ digit, were correct and still correct by stability.

□

# Least-Significant Radix Sort: Running Time

Recall have $\boldsymbol{n}$ numbers, all numbers have $\boldsymbol{d}$ digits.

# Least-Significant Radix Sort: Running Time

Recall have $\boldsymbol{n}$ numbers, all numbers have $\boldsymbol{d}$ digits.

# bucket sorts:

# Least-Significant Radix Sort: Running Time

Recall have $\boldsymbol{n}$ numbers, all numbers have $\boldsymbol{d}$ digits.

\# bucket sorts: $\boldsymbol{d}$

# Least-Significant Radix Sort: Running Time

Recall have $n$ numbers, all numbers have $d$ digits.

# bucket sorts: $d$
Time per bucket sort:

# Least-Significant Radix Sort: Running Time

Recall have $\boldsymbol{n}$ numbers, all numbers have $\boldsymbol{d}$ digits.

# bucket sorts: $\boldsymbol{d}$
Time per bucket sort: $\boldsymbol{O(n+k) = O(n+10) = O(n)}$.

# Least-Significant Radix Sort: Running Time

Recall have $n$ numbers, all numbers have $d$ digits.

# bucket sorts: $d$
Time per bucket sort: $O(n + k) = O(n + 10) = O(n)$.
Total time: $O(dn)$

# Least-Significant Radix Sort: Running Time

Recall have $n$ numbers, all numbers have $d$ digits.

# bucket sorts: $d$
Time per bucket sort: $O(n+k) = O(n+10) = O(n)$.
Total time: $O(dn)$

Is this good? Bad? In between?
If all numbers distinct, $d \geq \log_{10} n \implies$ total time $O(n \log n)$

# Least-Significant Radix Sort: Running Time

Recall have $n$ numbers, all numbers have $d$ digits.

# bucket sorts: $d$
Time per bucket sort: $O(n + k) = O(n + 10) = O(n)$.
Total time: $O(dn)$

Is this good? Bad? In between?
If all numbers distinct, $d \geq \log_{10} n \implies$ total time $O(n \log n)$

Bad: not $O(n)$ time!
Good: "Size of input" is $N = nd$, so linear in size of input!

# Least-Significant Radix Sort: Running Time

Recall have $n$ numbers, all numbers have $d$ digits.

# bucket sorts: $d$
Time per bucket sort: $O(n+k) = O(n+10) = O(n)$.
Total time: $O(dn)$

Is this good? Bad? In between?
If all numbers distinct, $d \geq \log_{10} n \implies$ total time $O(n \log n)$

Bad: not $O(n)$ time!
Good: "Size of input" is $N = nd$, so linear in size of input!

Improve to $O(n)$?

# Fast Radix Sort

Change to go $b$ digits at a time instead of just $\mathbf{1}$.

- Kind of cheating: look at $b$ digits in constant time.
- Necessary if we want time better than $nd$

# Fast Radix Sort

Change to go $b$ digits at a time instead of just $1$.

- Kind of cheating: look at $b$ digits in constant time.
- Necessary if we want time better than $nd$

# bucket sorts:

# Fast Radix Sort

Change to go $\boldsymbol{b}$ digits at a time instead of just $\mathbf{1}$.

- Kind of cheating: look at $\boldsymbol{b}$ digits in constant time.
- Necessary if we want time better than $\boldsymbol{nd}$

# bucket sorts: $\boldsymbol{d/b}$

# Fast Radix Sort

Change to go $b$ digits at a time instead of just $1$.

- Kind of cheating: look at $b$ digits in constant time.
- Necessary if we want time better than $nd$

# bucket sorts: $d/b$
Time per bucket sort:

# Fast Radix Sort

Change to go $b$ digits at a time instead of just $1$.

- Kind of cheating: look at $b$ digits in constant time.
- Necessary if we want time better than $nd$

# bucket sorts: $d/b$
Time per bucket sort: $O(n + k) = O(n + 10^b)$

# Fast Radix Sort

Change to go $b$ digits at a time instead of just $1$.

- Kind of cheating: look at $b$ digits in constant time.
- Necessary if we want time better than $nd$

# bucket sorts: $d/b$
Time per bucket sort: $O(n+k) = O(n+10^b)$
Total time: $O\left(\frac{d}{b}\left(n+10^b\right)\right)$

# Fast Radix Sort

Change to go $b$ digits at a time instead of just $1$.

- Kind of cheating: look at $b$ digits in constant time.
- Necessary if we want time better than $nd$

# bucket sorts: $d/b$
Time per bucket sort: $O(n+k) = O(n+10^b)$
Total time: $O\left(\frac{d}{b}\left(n+10^b\right)\right)$

Set $b = \log_{10} n$. If $d = O(\log n)$, then time

$$O\left(\frac{d}{\log_{10} n}(n+n)\right) = O(n)$$

# Fast Radix Sort

Change to go $b$ digits at a time instead of just $1$.

- Kind of cheating: look at $b$ digits in constant time.
- Necessary if we want time better than $nd$

\# bucket sorts: $d/b$
Time per bucket sort: $O(n+k) = O(n+10^b)$
Total time: $O\left(\frac{d}{b}\left(n+10^b\right)\right)$

Set $b = \log_{10} n$. If $d = O(\log n)$, then time

$$O\left(\frac{d}{\log_{10} n}(n+n)\right) = O(n)$$

Example: sorting integers between $0$ and $n^{10}$. Then $d$ should be about $\log_{10} n^{10} = 10 \log_{10} n$, as required.