Sorting

15-121 Fall 2020

Margaret Reid-Miller

Today

Today

- Bucket and Radix sorts
- Sorting properties
- Java sorts

Which sort algorithm is always slow?

Which sort is fast when the data is already sorted in non-decreasing order?

Which sort is always O(n log n)?

Which sort is fast on randomly ordered data?

Randomized quicksort is fast

- Fact: Quicksort has **expected** runtime of O(n log n) averaged over **all n! input orderings**.
- **Randomized quicksort:** For every partition, pick a pivot at **random** from the partition.
- Fact: **Randomized quicksort** has **expected** runtime of O(n log n) for **any** input ordering.
- Although it is possible for randomized quicksort to have $O(n^2)$ runtime (bad random pivots), it is highly unlikely.
  - If you run it again on the same data, the expected runtime will be O(n log n).

Comparison-based Sorts

- All of the sorts we've seen so far are comparison sorts.
  - The order of the elements is determined by comparing two elements at a time.
- It has been proven that the worst-case complexity for comparison sorts is $\Omega(n \log n)$.
  - O gives an asymptotically **upper** bound.
  - Ω gives an asymptotically **lower** bound;
    - no comparison-based sort can be faster.
- But there are sorts that can sort in O(n) time!
  ... they just don't use pair-wise comparisons

Sorting playing cards

- Given a deck of n playing cards, give an algorithm to put all the red cards before all the black cards.
  1. Deal all the cards into two piles, a red pile and a black pile
  2. Put each red card one at a time and then put each black card one at a time into a single pile.
- What is the run time of this algorithm?
  1. O(n) to deal the card into 2 piles.
  2. O(n) to collect the cards from the piles.

  Overall, O(n)

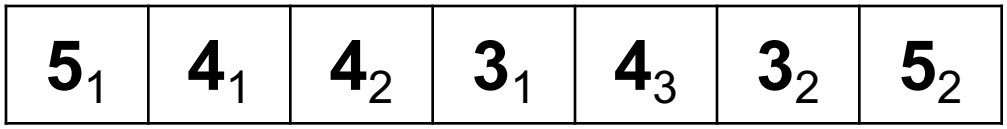
Sorting playing cards

- Can we do the same to sort n playing cards by suit?
- How about sorting by rank?
- Could we use the same idea to sort n values in the range 1 to 100?
- Given a value, how can we add it to a pile in O(1) time?

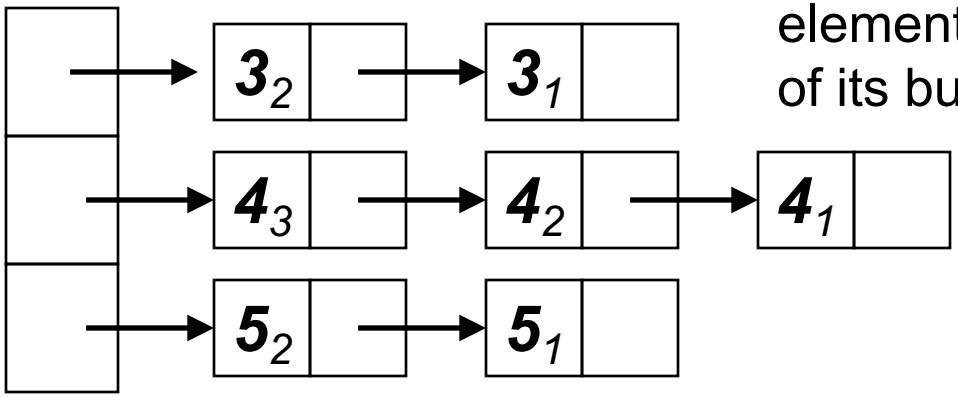
Bucket Sort

- Given an array of n elements that contain **only b unique** values (b < n). Let's call all them n_1, n_2, ..., n_b such that $n_1 < n_2 < ... < n_b$.
- Create an array of k "buckets", one for each unique value.
- For each value in the array, move it into its corresponding bucket.
- Copy the data values from each bucket, n_1 to n_b, back into the array to sort the data.

Bucket Sort Example



buckets are linked lists



insert each element at head of its bucket

remove each element from head of its bucket

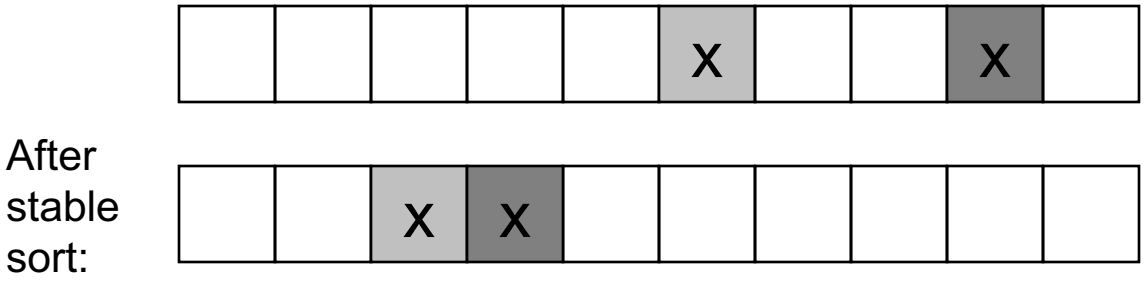
Bucket Sort is fast but limited use

- If we have n values and b buckets
  - We can put them into buckets in O(n) time.
  - We can collect them back in O(n+b) time;

    (We have to loop over all b buckets)
- However, we usually only use bucket sort when n >> b, so the runtime is O(n).
- Limitations?
  - Finite number of possible values. (Not a limitation of comparison-based sorts.)
  - Given a value, must be able to determine its bucket index in O(1) time.

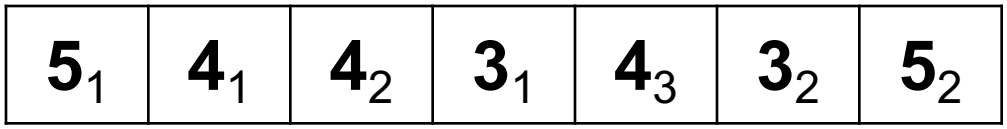
Stable Sorts

- A sort is stable if two elements with the same value maintain their same relative order before and after the sort is performed.



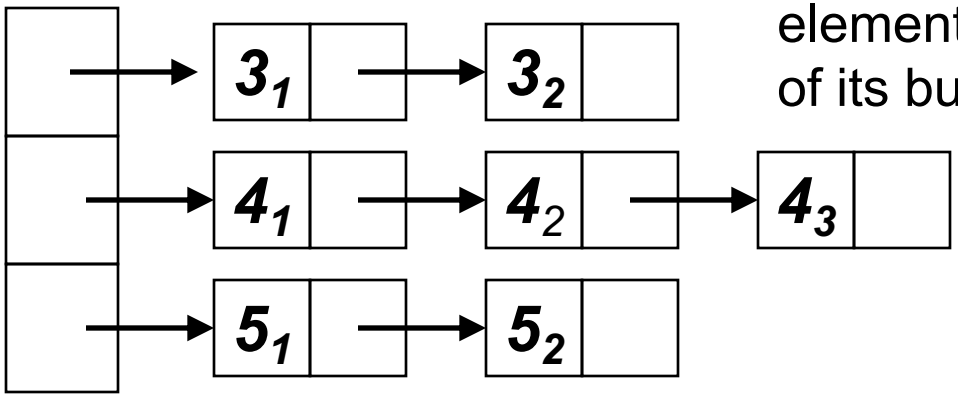
- Benefit: If you sort students by name and then sort again by sections, you get a list that is sorted by section, but alphabetical within each section.

Is Bucket Sort Stable?



insert each element at head of its bucket

buckets are linked lists



remove each element from head of its bucket

Sorting 3-digit integers

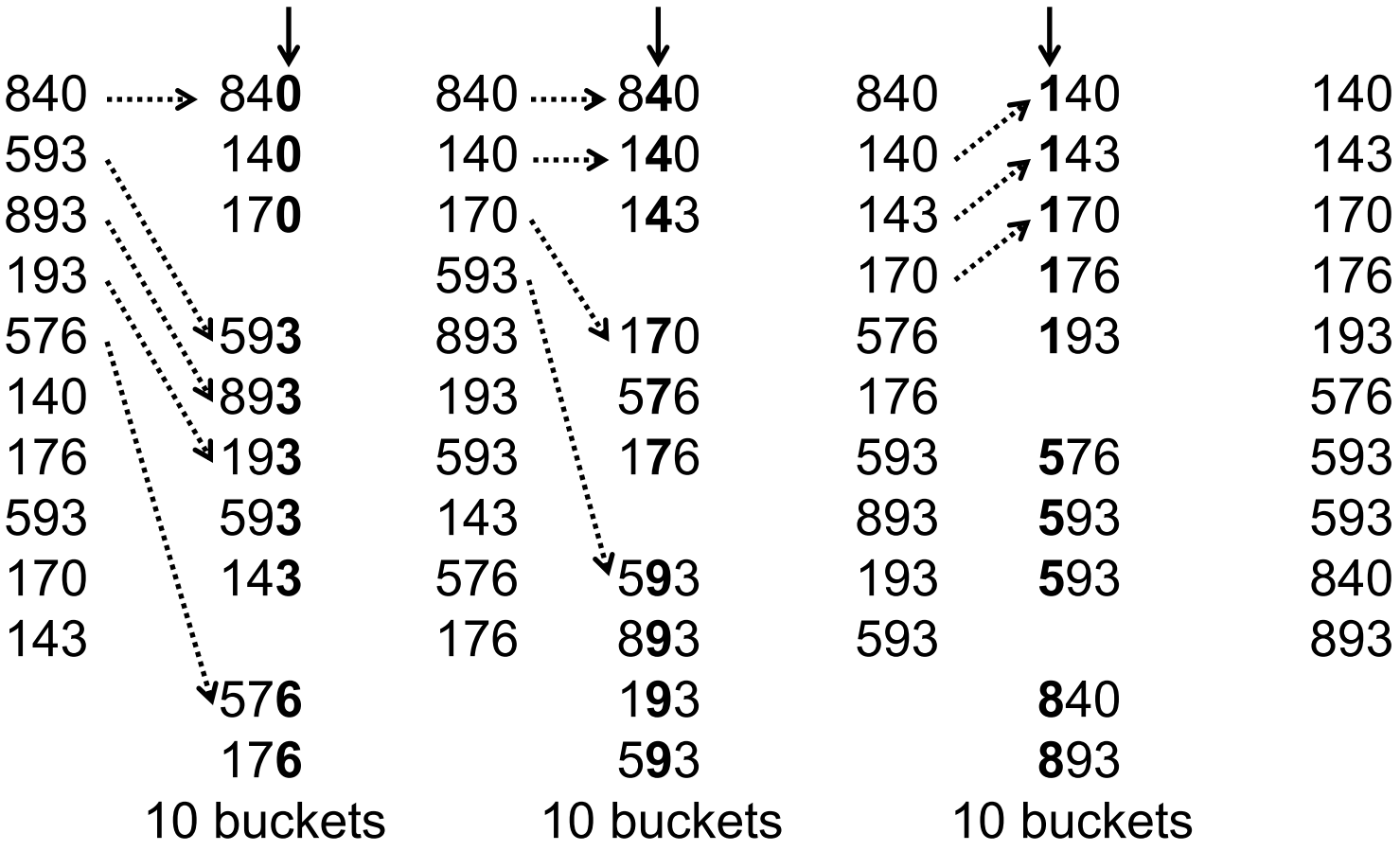
- Suppose we wanted to sort 3-digit integers.
- How many buckets would we need?
  - 1000 buckets?
- Do we really need 1000 buckets?
  - Could we sort one digit at a time?
  - Sort by the hundreds digit and within each hundreds sort the tens digit and within those the ones digit.
  - End up with 1000 mini sorts, though
- Radix sort is a variant of this idea.

Radix Sort Algorithm

- For integers use 10 buckets (0-9);
- Sort integers by **least significant** digit (ones digit) using bucket sort.
- Then sort by the next least significant digit (tens digit) using bucket sort.
- And so on until run out of digits.

- We never use more buckets than we have digits or symbols

Radix Sort example



Radix sort is O(n) as long as k & b are small, which limits its use

- If radix sort uses bucket sort, what must be true about bucket sort?
  - Bucket sort must be stable.

- What is the runtime of radix sort?
  - Each pass requires O(n+b) time, b buckets
  - We make one pass for each of k digits
  - Overall runtime is O(k * (n+b))

Complexity Summary

Sort	Worst	Average	Best
Selection			
Insertion			
Tree			
Merge			
Quick			
Bucket			
Radix			

Properties Summary

Sort	In-place	Adaptive	Stable
Selection			
Insertion			
Tree			
Merge			
Quick			
Bucket			
Radix			

Java sorts in "natural order"

- In `Arrays` class:

```
public static void sort(Object[] items)
```

  - All objects must **`Comparable`** (compareTo).
  - Implemented with a modified merge sort in O(n log n)
    - Adapted from sort used in Python (Tim's sort)
  - Sort is stable

- In `Collections` class:

```
public static <T extends Comparable<T>> void
                                    sort(List<T> list)
```

  - Same conditions as above
  - Copies elements into an array and uses `Arrays.sort`

Java sorts with other orderings

Also in `Arrays` class:

```
public static <T> void
    sort(T[] items, Comparator<? super T> comp)
```

- Another version allows a sort using a **Comparator** so ordering can be done on some other property other than the items' natural ordering.
- For example: You might order strings not alphabetically, but instead by string length.
- `comp` must be an object of type T or a subclass of type `T` where T implements the `Comparator` interface.

Example: Sort with `Comparator`

```
public class StringLengthCmptr
                implements Comparator<String> {
  public int compare(String s1, String s2) {
     return s1.length() - s2.length();
  }
}
```

Example:

Assume `s` is an array of strings.

```
Arrays.sort(s);
Arrays.sort(s, new StringLengthCmptr());
```