CMPT 225: Data Structures & Programming – Unit 28 – Bucket Sort, Radix Sort, and Sorting Overview

Dr. Jack Thomas
Simon Fraser University

Spring 2021

Today’s Topics

- Bucket-Sort
- Radix-Sort
- Stable Sorting
- In-Place Sorting
- Comparison Between Sorts

Non-Comparison-Based Sorting

- **Comparison-based** sorting methods have a lower bound for efficiency of **O(n log n)** – it's just not possible to get it any lower.
- Most practical applications of sorting, however, have some **limits to the expected inputs** that we can take advantage of to improve our sorting efficiency to **O(n)**
- The two we'll consider today are **Bucket-Sort** and **Radix-Sort**.

Restricting the Range of Inputs

- One set of inputs that comes up a lot is a **bounded range of numbers**, either on their own or as keys for entries.
- For example, the **ranks of an ordered set of three entries** could give us the sequence (1, 2, 3).
- More formally, this type of input is defined as:
  – A sequence S of n entries whose keys are integers in the range [0, N-1], for some integer N >= 2.

Bucket-Sort

- **Bucket-Sort** is not comparison-based, it uses the **keys as indices into a bucket array** with cells 0 to N-1.
- An entry with key k matches with the bucket B[k], with **identical keys landing in the same bucket** (this should remind you of Hash Tables).
- Once the bucket array is filled, **emptying it in order** from B[0] to B[N-1] will give you the entries ordered by their keys.

```
Algorithm bucketSort(S):
Input: Sequence S of entries with integer keys in the range [0, N-1]
Output: Sequence S sorted in nondecreasing order of the keys
Let B be an array of N sequences, each of which is initially empty
For each entry e in S do
        k <- e.getKey()
        remove e from S and insert it at the end bucket (sequence) B[k]
For I <- 0 to N – 1 do
        for each entry e in sequence B[i] do
                remove e from B[i] and insert it at the end of S
```

Radix-Sort

- One issue with Bucket-Sort is that it can **only sort according to one term** – it can't handle some of the more sophisticated multi-part keys you can normally compare with a Comparator rule (for example, comparing dates made of a day, month, and year).
- **Radix-Sort** augments Bucket-Sort to handle these situations by **layering the Bucket-Sorts in reverse-order** of the key's parts.

Radix Example Part One

- Imagine **a two-part key made of two different integers**, which we would like to sort according to a **lexicographical order**.

S	(3,3)	(1,5)	(2,5)	(1,2)	(2,3)	(1,7)	(3,2)	(2,2)

- This is essentially what **alphabetical order** is, so thinking of these as two-letter keys may help.

Radix Example Part 2

- Sorting our example set by our first key will leave some entries out of order.

S_1 (1,5) (1,2) (1,7) (2,5) (2,3) (2,2) (3,3) (3,2)

- Sorting them again by the second key will fix that ordering, but break it for the first.

S_{12} (1,2) (2,2) (3,2) (2,3) (3,3) (1,5) (2,5) (1,7)

Radix Example, The Thrilling Conclusion

- On the other hand, if we sort them according to the second key first…

S_2	(1,2)	(3,2)	(2,2)	(3,3)	(2,3)	(1,5)	(2,5)	(1,7)

- And then sort them by the first…

S_{21}	(1,2)	(1,5)	(1,7)	(2,2)	(2,3)	(2,5)	(3,2)	(3,3)

- The reversed process of Radix-Sort ensures that entries with equal keys are **already ordered** by their second (or third, or fourth…) key components.

Sorting Stability

- Bucket and Radix-Sort raise the issue of **Stable Sorting**, which is the question of how to handle entries with **equal keys**.
- Stable sorting methods are ones where entries with equal keys **retain their relative positions to one another** in the sorted sequence.

Unstable Sorting

- **Quick-Sort** is an example of an unstable sorting method – the set of values equal to the pivot value is not built in a way that preserves their relative ordering.

(2, 5, 4, 3, **4**, 3, 2, 6, 4) -> (2, 3, 3, 2) (**4**, 4, 4) (5, 6)

Consequences of Sorting Stability

- The stability of different sorting algorithms can influence their **efficiency** (why spend time shuffling around equal entries?).
- It can also create **unintended side-effects** – recall that Java used to use Merge-Sort to sort arrays of objects, but Quick-Sort for arrays of primitive data types.
- This was because **two integers are completely identical**, so an unstable sort has no effect, **but two objects with the same key are not identical** and an unstable sort might randomly swap their positions each time it runs.

Sorting “In Place”

- Another consideration for sorting methods aside from run-time efficiency is **how much memory space they take up** while they’re running.
- Ideally, sorting can be done **In-Place**, meaning that there’s no need for creating new, temporary copies of the data being sorted during the running of the algorithm.
- If a sort can’t be done in-place, how much extra memory space it requires becomes **another metric of performance** to worry about.

Examples for Sorting In-Place

- **Insertion-Sort** can be done in-place, simply by moving the entries around within the sequence being sorted. This is how the version we introduced in section 4 works.

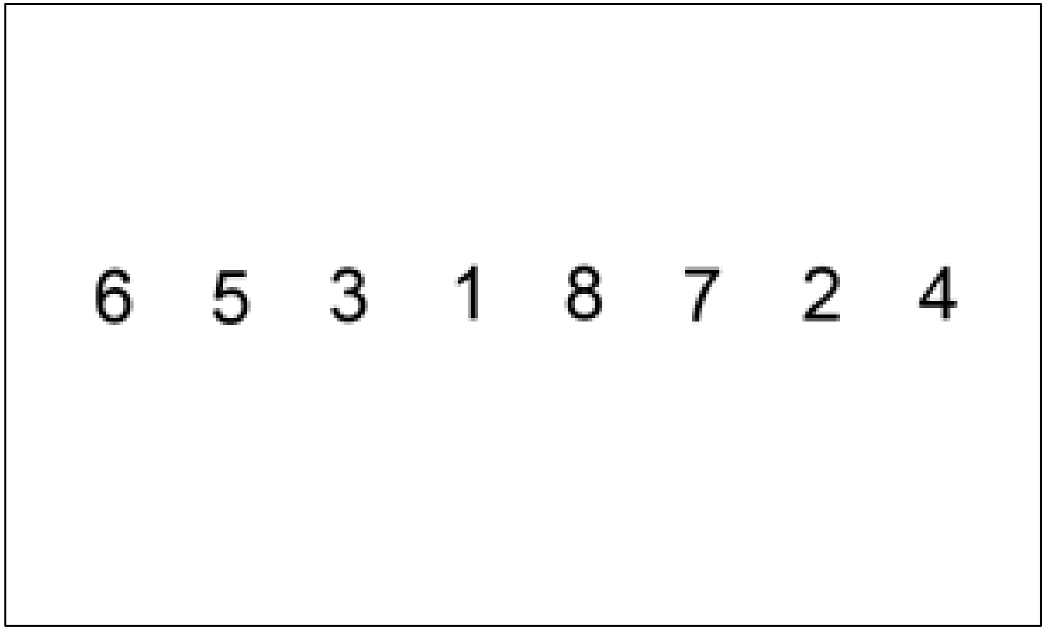

Image credit: https://upload.wikimedia.org/wikipedia/commons/0/0f/Insertion-sort-example-300px.gif

When a Sort Can't be In-Place

- **Merge-Sort** is a good example of a sorting algorithm that **can't be done in-place**, specifically thanks to the merge step where the two sorted sets are emptied into the new, combined set.
- This requires the creation of new subsets during the division step, meaning **each layer of Merge-Sort's decision tree** creates a **whole new copy of the input data**, divided across twice as many sets as the previous layer – that's expensive!

Oh Hey Remember Heap-Sort

- While we're wrapping up, I should remind you of **Heap-Sort**, introduced alongside **Heaps**.
- Take a set of input data and build a Heap (a Complete Binary Tree), then **keep removing the root** to build a new **in-order sequence**.
- Like the other comparison-based sorts, Heap-Sort can run **in O(n log n).** It can also be done **in-place**, but is **unstable**.

Summary of Sorting Algorithms

- Let’s do a quick run-down of our options:
  1. **Insertion-Sort** ($O(n^2)$, stable, in-place)
  2. **Merge-Sort** (O(n log n), stable, not in-place)
  3. **Quick-Sort** (O(n log n)*, unstable, in-place)
  4. **Heap-Sort** (O(n log n), unstable, in-place)
  5. **Bucket-Sort/Radix-Sort**(O(n+N)/O(d(n+N)), stable, not in-place).

When to Use the Different Sorts

- **Insertion-Sort**’s actual run-time is O(n+m), where m is the number of inversions (the number of pairs of elements out of order).
- **Small sequences** (i.e. fewer than 50) necessarily have fewer inversions, as do cases where the inputs are already **nearly sorted**.
- In those limited instances, Insertion-Sort can actually be a pretty **efficient, easy-to-program, stable, in-place** option.

When to Use the Different Sorts

- **Merge-Sort** is the best comparison-based search if **stability** is required, but since it isn't in-place, the **memory usage is often too high**.
- **Quick-Sort** is, on-average, the **fastest of the comparison-sorts**, but because it **can't truly guarantee O(n log n)** it's not always appropriate.
- **Heap-Sort** is actually seen as the **best choice of the comparison-sorts where consistent performance is needed**, since it guarantees O(n log n) and can be implemented in-place.

When to Use the Different Sorts

- **Bucket/Radix-Sort** is a good pick when **sorting a limited range of integers**, or d-tuple keys made up of integers.
- Keep an eye out for **cases where N = n**, like sorting entries according to their ranking, since those can be sorted in O(n) instead of the comparison-based algorithms' O(n log n).

Recap – Summing Up This Sort Of Thing

- **Bucket-Sort** and **Radix-Sort** make linear-time sorting possible for inputs with limited ranges.
- **Stable sorting** ensures entries with equal keys retain their relative positions in the sorted set.
- **In-place sorting** is when algorithms can sort within the structure being sorted, without needing to take up additional memory.
- Remember **Heap-Sort**? Sorting by making a Heap? Don’t forget Heap-Sort!
- **Each sorting method has benefits and drawbacks** that influence when it should be used.