# *BucketSort (a.k.a. BinSort)*

- If all values to be sorted are known to be integers between 1 and *K* (or any small range),
  - Create an array of size *K,* and put each element in its proper bucket (a.ka. bin)
  - *If* data is only integers, no need to store more than a *count* of how many times that bucket has been used
- Output result via linear pass through array of buckets

| `count` array | |
|---|---|
| 1 | |
| 2 | |
| 3 | |
| 4 | |
| 5 | |

- Example:

  K=5

  Input: (5,1,3,4,3,2,1,1,5,4,5)

  output:

# *BucketSort (a.k.a. BinSort)*

- If all values to be sorted are known to be integers between 1 and *K* (or any small range),
  - Create an array of size *K,* and put each element in its proper bucket (a.ka. bin)
  - *If* data is only integers, no need to store more than a *count* of how many times that bucket has been used
- Output result via linear pass through array of buckets

| **`count`** array | |
|---|---|
| 1 | 3 |
| 2 | 1 |
| 3 | 2 |
| 4 | 2 |
| 5 | 3 |

- Example:

  K=5

  input (5,1,3,4,3,2,1,1,5,4,5)

  output: 1,1,1,2,3,3,4,4,5,5,5

What is the running time?

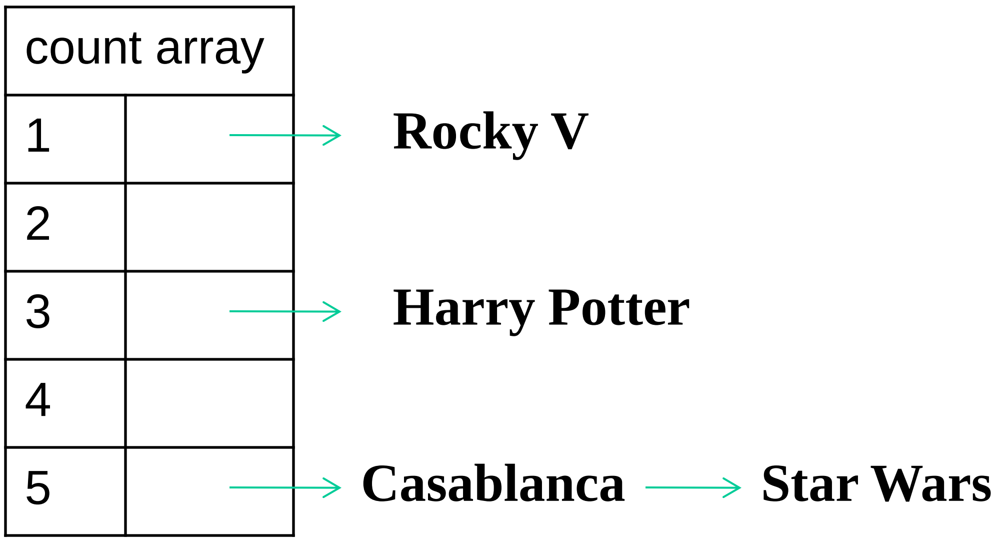
# *Analyzing bucket sort*

- Overall: $O(n+K)$
  - Linear in $n$, but also linear in $K$
  - $\Omega(n \log n)$ lower bound does not apply because **this is not a comparison sort**
- Good when range, $K$, is smaller (or not much larger) than $n$
  - (We don't spend time doing lots of comparisons of duplicates!)
- Bad when $K$ is much larger than $n$
  - Wasted space; wasted time during final linear $O(K)$ pass
- For data in addition to integer keys, use list at each bucket

# *Bucket Sort with Data*

Bucket sort illustrates a more general trick: How might you implement a heap for a small range of integer priorities in a similar manner…

- Most real lists aren't just #'s; we have data
- Each bucket is a list (say, linked list)
- To add to a bucket, place at end O(1) (keep pointer to last element)




- Example: Movie ratings: 1=bad,… 5=excellent
- Input=

  5: Casablanca

  3: Harry Potter movies

  1: Rocky V

  5: Star Wars

**Result**: 1: Rocky V, 3: Harry Potter, 5: Casablanca, 5: Star Wars
This result is stable; Casablanca still before Star Wars

# *Radix sort*

- Radix = "the base of a number system"
  - Examples will use 10 because we are used to that
  - In implementations use larger numbers
    - For example, for ASCII strings, might use 128
- Idea:
  - Bucket sort on one digit at a time
    - Number of buckets = radix
    - Starting with *least* significant digit, sort with Bucket Sort
    - Keeping sort *stable*
  - Do one pass per digit
- **Invariant**: After *k* passes, the last *k* digits are sorted

- Aside: Origins go back to the 1890 U.S. census

# *Example*

Radix = 10

| 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 |
|---|---|---|---|---|---|---|---|---|---|
| | 721 | | 3<br>143 | | | | 537<br>67 | 478<br>38 | 9 |

**Input:** 478
537
9
721
3
38
143
67

**First pass:**

1. bucket sort by ones digit
2. Iterate thru and collect into a list

- List is sorted by first digit

**Order now:** 721
3
143
537
67
478
38
9

*Example*

| 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 |
|---|---|---|---|---|---|---|---|---|---|
| | 721 | | 3<br>143 | | | | 537<br>67 | 478<br>38 | 9 |

Radix = 10

| 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 |
|---|---|---|---|---|---|---|---|---|---|
| 3<br>9 | | 721 | 537<br>38 | 143 | | 67 | 478 | | |

Order was: 721
3
143
537
67
478
38
9

Second pass:

stable bucket sort by tens digit

If we chop off the 100's place, these #s are sorted

Order now: 3
9
721
537
38
143
67
478

*Example*

| 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 |
|---|---|---|---|---|---|---|---|---|---|
| 3<br>9 | | 721 | 537<br>38 | 143 | | 67 | 478 | | |

Radix = 10

| 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 |
|---|---|---|---|---|---|---|---|---|---|
| 3<br>9<br>38<br>67 | 143 | | | 478 | 537 | | 721 | | |

Order was:

3
9
721
537
38
143
67
478

Third pass:

stable bucket sort by 100s digit

Only 3 digits: We’re done!

Order now:

3
9
38
67
143
478
537
721

**Student Activity**

# *RadixSort*

- Input:126, 328, 636, 341, 416, 131, 328

**BucketSort on lsd:**

| | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|
| | | | | | | | | | |
| 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 |

**BucketSort on next-higher digit:**

| | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|
| | | | | | | | | | |
| 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 |

**BucketSort on msd:**

| | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|
| | | | | | | | | | |
| 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 |

# *Analysis of Radix Sort*

Performance depends on:

- Input size: *n*
- Number of buckets = Radix: *B*
  - e.g. Base 10 #: 10; binary #: 2; Alpha-numeric char: 62
- Number of passes = “Digits”: *P*
  - e.g. Ages of people: 3; Phone #: 10; Person’s name: ?

- Work per pass is 1 bucket sort: ____________
  - Each pass is a Bucket Sort
- Total work is ______________
  - We do ‘P’ passes, each of which is a Bucket Sort

# *Analysis of Radix Sort*

Performance depends on:

- Input size: $n$
- Number of buckets = Radix: $B$
  - e.g. Base 10 #: 10; binary #: 2; Alpha-numeric char: 62
- Number of passes = "Digits": $P$
  - e.g. Ages of people: 3; Phone #: 10; Person's name: ?

- Work per pass is 1 bucket sort: $O(B+n)$
  - Each pass is a Bucket Sort
- Total work is $O(P(B+n))$
  - We do 'P' passes, each of which is a Bucket Sort

# *Comparison to Comparison Sorts*

Compared to comparison sorts, sometimes a win, but often not

- Example: Strings of English letters up to length 15
  - Approximate run-time: 15*(52 + *n*)
  - This is less than *n* log n only if *n* > 33,000
  - Of course, cross-over point depends on constant factors of the implementations plus *P* and *B*
    - And radix sort can have poor locality properties
- Not really practical for many classes of keys
  - Strings: Lots of buckets

# *Recap: Features of Sorting Algorithms*

**In-place**

– Sorted items occupy the same space as the original items. (No copying required, only O(1) extra space if any.)

**Stable**

– Items in input with the same value end up in the same order as when they began.

Examples:

- Merge Sort - not in place, stable
- Quick Sort - in place, not stable

# *Sorting massive data: External Sorting*

Need sorting algorithms that **minimize disk/tape access** time:

- Quicksort and Heapsort both jump all over the array, leading to expensive random disk accesses
- Mergesort scans linearly through arrays, leading to (relatively) efficient sequential disk access

Basic Idea:

- Load chunk of data into Memory, sort, store this “run” on disk/tape
- Use the Merge routine from Mergesort to merge runs
- Repeat until you have only one run (one sorted chunk)

- Mergesort can leverage multiple disks
- Weiss gives some examples

# *Sorting Summary*

- Simple $O(n^2)$ sorts can be fastest for small $n$
  - selection sort, insertion sort (latter linear for mostly-sorted)
  - good for “below a cut-off” to help divide-and-conquer sorts
- $O(n \log n)$ sorts
  - heap sort, in-place but not stable nor parallelizable
  - merge sort, not in place but stable and works as external sort
  - quick sort, in place but not stable and $O(n^2)$ in worst-case
    - often fastest, but depends on costs of comparisons/copies
- $\Omega(n \log n)$ is worst-case and average lower-bound for sorting by comparisons
- Non-comparison sorts
  - Bucket sort good for small number of key values
  - Radix sort uses fewer buckets and more phases
- Best way to sort? It depends!