Sorting Algorithms

Introduction

- ***sorting***: an algorithm that arranges the elements in a list in a particular order (ascending or descending).

- Why bother studying sorting algorithms?
  - demonstrates creative methods to problem solving
    - these methods can be applied to many other problems
  - good for practicing fundamental programming techniques
    - good use of selection statements, loops, methods, and arrays
  - great examples for demonstrating the need for good algorithm efficiency

- How to study sorting algorithms?
  - hand trace them on paper to see how they act on a list of data.
  - implement the algorithms and see how they compare to one another when they have to deal with large amounts of data.

Sorting Classification

- **Number of Comparisons**
  - classifed based on number of comparisons that occur during the sort.
  - best case for these is O(nlogn) and worst case is $O(n^2)$.
  - comparison based algorithms work by comparing elements to eachother and deciding where they should fall in the final result. they need at least (nlogn) comparisons.

- **Number of Swaps**
  - sorting algorithms are categorized by the number of swaps (interchanging the place of two values.)

- **Memory Usage**
  - Some algorithms are "in place" (they don't need extra memory locations) and they need O(1) or O(logn) memory to create temporary memory locations.
  -

- **Recursion**
  - Some algorithms are either recursive or non recursive and even some that use both recursion and non-recursion.

Sorting Classification

- **Stability**
  - A ***stable*** sorting algorithm is an algorithm that keeps the same relative ordering of items with the same value.
    - Example: If you have a list of values $\{5_a, 3, 7, 2_a, 4_a, 2_b, 5_b, 4_b, 6\}$ then the result would be $\{2_a, 2_b, 3, 4_a, 4_b, 5_a, 5_b, 6, 7\}$
- **Adaptability**
  - with some algorithms the complexity changes based on pre-sortedness.
- **Internal Sort**:
  - algorithms that use only the main memory to perform the sorting.
  - usually for when the amount of data is relatively small.
- **External Sort:**
  - Sorting algorithms use external memory such as the hard drive.
  - the algorithm will involve read / write operations.
  - usually for when the amount of data is really large.

Pseudocode Note

- **NOTE:** For all following pseudocode, the bounds are inclusive.

Bubble Sort

Bubble Sort Properties

- Named for the way the larger elements "bubble" to the top.

- Runtime:
  - $O(n^2)$ in the worst case.
  - $O(n)$ if partially or fully sorted or optimized

- Is a stable sort: elements which are equal are not swapped.

- General Algorithm:
  - Check adjacent pairs of elements and swap them if they are not in place.
  - Repeat from the beginning of the list until sorted.

- Is a comparison sort.

- Is the simplest sorting algorithm but is too slow and impractical for most problems.
  - can be practical if the input is usually in sorted order but occasionally has some out of order elements
  - can detect if a list is already sorted.

Bubble Sort – Pseudocode (Unoptimized)

```
bubble_sort(list)
   for i from 1 to list.length - 1:
      for j from 0 to list.length – 2:
         if list[j] > list[j + 1]
            swap list[j] with list[j + 1])
```

Bubble Sort Pseudocode (Optimized)

```
bubble_sort2(list)
   for i from 1 to list.length - 1
      swapped = false
         for j from 0 to list.length - 2
            if a[j] > a[j + 1]
               swap( a[j], a[j + 1] )
               swapped = true
         if !swapped
           break
```

- Here if the list is already sorted (or partially sorted), we can get a best case complexity of O(n)

Selection Sort

Selection Sort Properties

- Named for the way it repeatedly selects the smallest element.
- Runtime: $O(n^2)$
- an in-place sorting algorithm (requires no additional storage).
- works well for small amounts of data.
- easy to implement
- General Algorithm:
  - Find the minimum value in the list.
  - Swap the minimum with the value at the current position
  - Repeat for all elements until list is sorted.

Selection Sort Pseudocode

```
selection_sort(list):
   for i from 0 to list.length - 1:
      min = i
      for j from i + 1 to list.length - 1:
         if list[j] < list[min]
            min = j;
      swap list[i] with list[min]
```

Insertion Sort

Insertion Sort Properties

- Named for the fact that it chooses and element from the list and instead of swapping, inserts the element into the correct position.

- Runtime: $O(n^2)$
  - Despite its runtime, it's usually more efficient (on average) than selection or bubble sort.

- Easy and simple implementation.

- Efficient for a small amount of data.

- Adaptive: if the list is presorted (or partially sorted) then the sorting takes $O(n + d)$, where d is the number of inversions.

- Stable: maintains the relative order of input data if the keys are the same.

- In-place: requires only constant amount $O(1)$ of additional memory space.

- General Algorithm:
  - Choose an item from the unsorted portion of the list.
  - Insert the item into the sorted portion of the list in the correct insertion point.
  - Continue until list is sorted.

Insertion Sort Pseudocode

```
insertion_sort(list):
   for i from 1 to list.length - 1:
      temp = list[i]
      j = i
      while j > 0 and temp < list[j - 1]:
         list[j] = list[j - 1]
         j--
      list[j] = temp
```

Heap Sort

Heap Sort Properties

- Heap Sort is named after the fact that it uses a heap as its underlying mechanism.
- Runtime: O(nlogn)
  - Slower in practice on most machines than a good implementation of Quick Sort.
- Simple to implement assuming you can implement a working heap (also very easy).
  - Review heaps from last week's lecture.
- General Algorithm:
  - Throw all the elements of your list into a heap.
  - Delete each element from the heap one by one until the heap is empty.
  - Each element will be returned in the correct sorted order.
- A variation of this could be done with a binary search tree:
  - add all elements to the BST
  - Use an in order traversal to get all of the elements in sorted order.

Heap Sort Pseudocode

```
heap_sort(list):
    heap = create an empty heap

    for i from 0 to list.length - 1:
        heap.add(list[i])

    for i from 0 to list.length - 1:
        list[i] = heap.delete()
```

Merge Sort

Merge Sort Properties

- A divide and conquer sort which is named after the fact that it divides the data into sublists and then merges them back together.
- Runtime: O(nlogn)
- Notes:
  - ***merging***: process of combining two sorted lists to make a bigger list.
  - ***selection***: process of dividing a list into two parts: *k* smallest elements and n - k largest elements.
  - selection and merging are opposites:
    - selection splits the list into two lists
    - merging combines the lists into one list
- Good for sorting linked lists.
- Does not care about initial ordering of input.
- General Algorithm:
  - Divide the unsorted list into n sublists, each containing 1 element (a list of 1 element is considered sorted).
  - Repeatedly merge sublists to produce new sorted sublists until there is only 1 sublist remaining. This will be the sorted list.

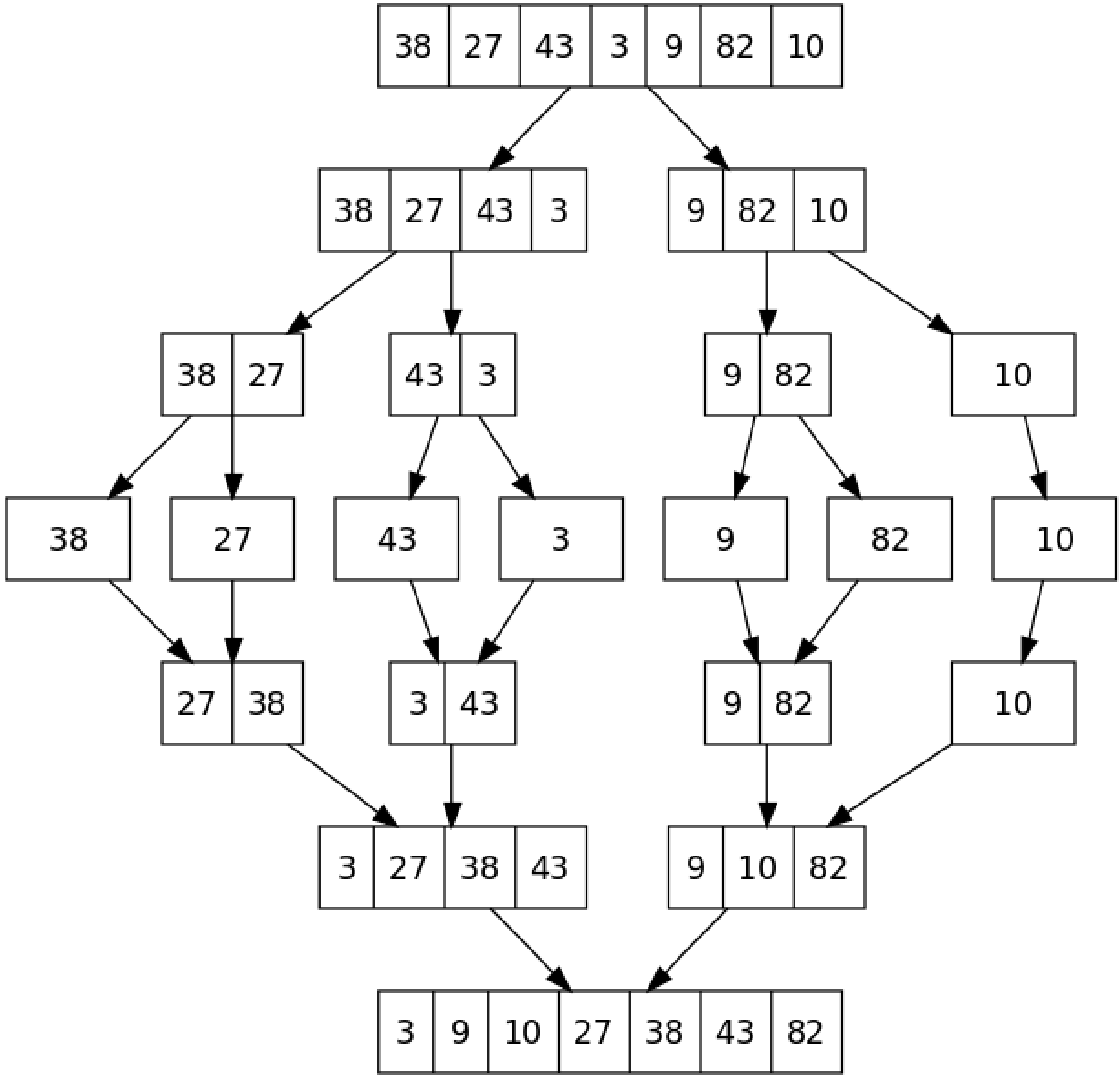
Merge Sort - Merge Function Pseudocode

```
merge(list1, list2, result):
   i = 0, j = 0, k = 0

   while i < list1.length && j < list2.length:
      if list1[i] < list2[j]:
         result[k] = list1[i]
         i++
      else
         result[k] = list2[j]
         j++
      k++

   while i < list1.length:
      result[k] = list1[i]
      i++
      k++


   while j < list2.length:
      result[k] = list2[j]
      j++
      k++
```

Merge Sort Pseudocode

```
merge_sort(list):
   if list.length > 1:
      mid = (list.length - 1) / 2

      left = list from 0 to mid
      merge_sort(left)


      right = list from (mid + 1) to list.length - 1
      merge_sort(right)

      merge(left, right, list)
```

Merge Sort Example



Quick Sort

Quick Sort Properties

- Another divide and conquer algorithm.

- Runtime O(nlogn)

- General Algorithm:
  - If there are one or no elements in the array to be sorted return.
  - Pick an element in the array to serve as the "pivot" point. (This is usually the right-most element in the array, but can vary depending on how optimized Quick Sort is.)
  - Split the array into two parts - one with elements larger then the pivot and the other with elements smaller than the pivot.
  - Recursively repeat the algorithm for both halves of the array.

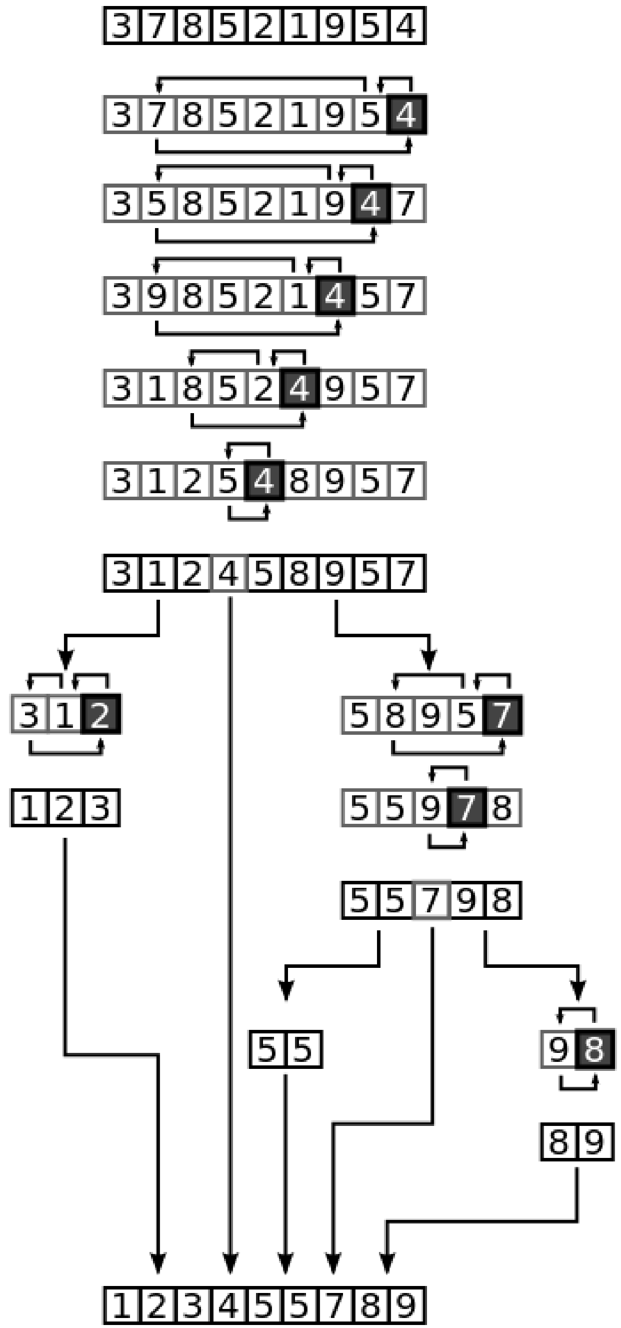
Quick Sort - Partition Pseudocode

```
partition(list, low, high):
   pivot = list[high]
   i = low - 1

   for j from low to high - 1:
      if list[j] <= pivot:
         i = i + 1
         swap list[i] with list[j]

   swap list[i+1] with list[high]
   return i + 1
```

Quick Sort Pseudocode

```
quick_sort(list):
    quick_sort(list, 0, list.length - 1):

quick_sort(list, low, high):
    if low < high:
        p = partition(list, low, high):
        quicksort(list, low, p - 1)
        quicksort(list, p + 1, high)
```

Quick Sort Example



Counting Sort

Counting Sort

- A sorting algorithm that is **not** a comparison algorithm:
  - It can be mathematically proven that no comparison algorithm can achieve a runtime of better than O(nlogn).
    - You will see this in CS-3112.

- Runtime: O(n)

- A sorting algorithm that only works with integers:
  - determine for each integer x the number of elements less than x which can tell us where x should end up
    - example: if there are 10 elements less than x, then we know that x belongs in the 11^{th} position.

Counting Sort Pseudocode

```
counting_sort(list, k): //k is a number such that all
                         //keys are in the range 0..k-1
   result = empty list with same size as list
   counts = empty list with size of k

   for i from 0 to k-1:
      counts[i] = 0;

   for i from 0 to list.length - 1:
      counts[list[i]]++

   for i from 1 to k-1:
      counts[i] += counts[i - 1]

   for i from list.length - 1 to 0:
      result[counts[list[i]] - 1] = list[i]
      counts[list[i]]--

   return result
```

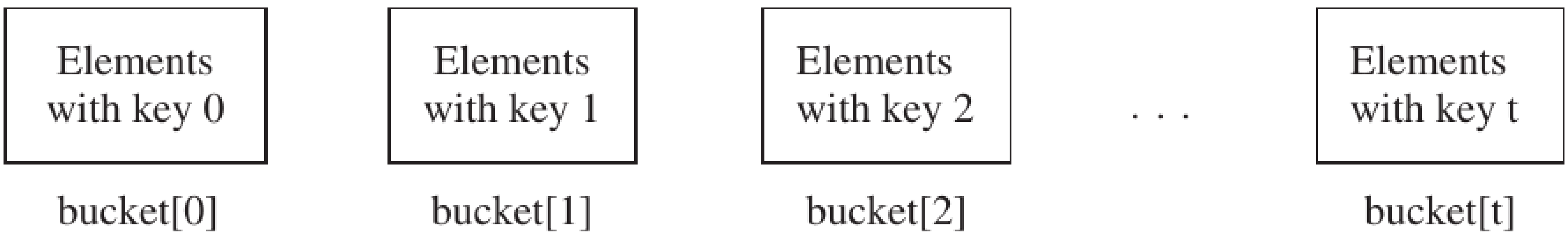
Radix Sort

Radix Sort

- Similar to Counting Sort
  - assumes all input values are from the base *d* number system.
  - this means that all numbers are *d*-digit numbers
  - if you want to sort normal integers then this is the base 10 system which means you have digits radix 10 with digits 0 - 9
- Runtime: O(nd)
  - if d is small then it is O(n)
- General Algorithm:
  - Take the least significant digit of each element.
  - Sort the list of elements based on that digit, but keep the order of elements with the same digit (have to use a stable sorting algorithm here).
  - Repeat with each more significant digit until all digits have been exhausted.

Radix Sort

- In this algorithm, you have a data structure with a bunch of "buckets".
  - each bucket can hold a smaller list of items.
  - this can be implemented in Java as an array of lists with size 10 (assuming you are using base 10).



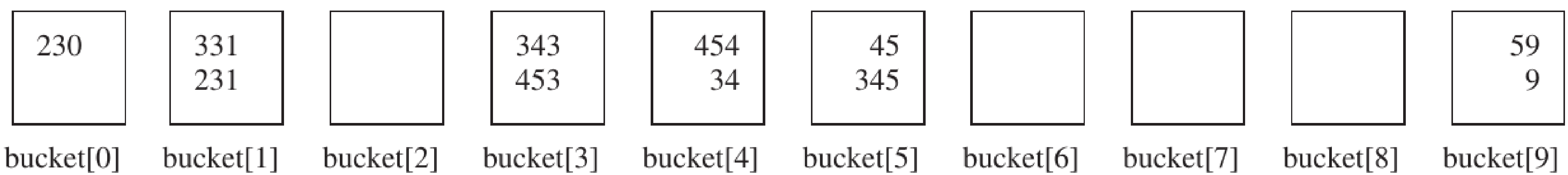
Radix Sort Pseudocode

```
radix_sort(list): //assumes base 10
   buckets = array of lists of integers

   initialize all of the buckets

   for i from 1 to number of digits in max value in list:
      for j from 0 to list.length - 1:
         key = get the ith least significant digit of list[j]
         buckets[key].add(list[j])

      k = 0;

      for j from 0 to buckets.length - 1:
         for each element x in buckets[j]:
            list[k++] = x;
         clear list at buckets[j];
```

Radix Sort Example

- Example: Apply Radix Sort to:
  331, 454, 230, 34, 343, 45, 59, 453, 345, 231, 9

- Assign each element to buckets based on the least significant digit.

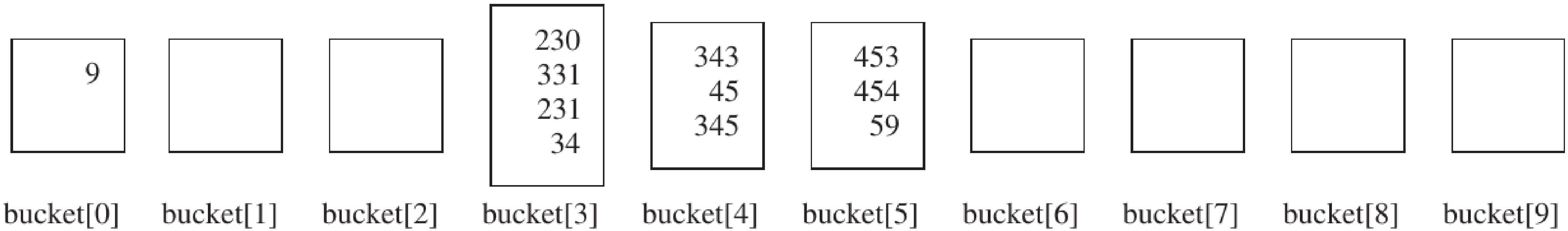


- Remove elements from the buckets in order to get:

  23**0**, 33**1**, 23**1**, 34**3**, 45**3**, 45**4**, 3**4**, 4**5**, 34**5**, 5**9**, **9**

Radix Sort Example

- Assign elements from the previous result to buckets based on the next most significant digit.



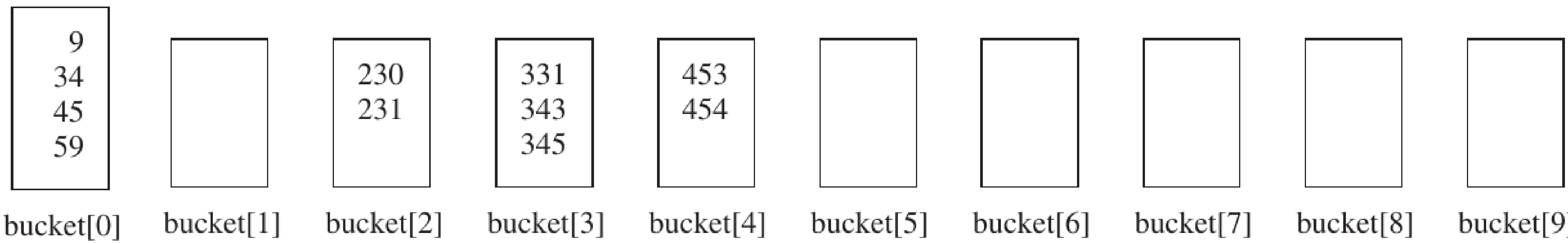
- Remove elements from the buckets to get:

  9, 2**3**0, 3**3**1, 2**3**1, **3**4, 3**4**3, **4**5, 3**4**5, 4**5**3, 4**5**4, **5**9

  - NOTE: 9 is 009

Radix Sort Example

- Assign elements from the previous result to buckets based on the next most significant digit (which is this case is the most significant digit)



- Remove elements from the buckets to get:

  9, 34, 45, 59, **2**30, **2**31, **3**31, **3**43, **3**45, **4**53, **4**54

- Everything is now sorted.

External Sorting

External Sorting

- a general term for algorithms that can handle sorting massive amounts of data.

- useful when files are too large to fit into main memory.

- there are many algorithms for external sorting, one such algorithm is external merge sort.

External Merge Sort Algorithm

- In this scenario assume that we need to sort 900 megabytes of data but we can only use 100 megabytes of RAM.

- 1. Read 100MB of the data into main memory and sort by some conventional method.
  - This could be any fast sorting algorithm (Quick Sort, Merge Sort, etc).
- 2. Write the sorted data to disk.
- 3. Repeat steps 1 and 2 until till of the data is sorted in chunks of 100MB.
  - (The final step is to merge the data into a single sorted output file.)
- 4. Read the first 10MB of each sorted chunk in main memory (these are the input buffers) and perform a 9-way merge between all chunks.
  - 10MB is reserved for an output buffer.
- 5. Continue until all data is merged into one file.

Generalized Calculations

- Assume the amount of data to be sorted exceeds the available memory by a factor of K.
- K chunks of data need to be sorted and a K-way merge is performed.
- Assume X is the amount of main memory available, there will be K input buffers and 1 output buffer of size X/(K + 1) each.
- The output buffer size can be increased for better performance depending on how fast the hard drive is.