Sorting

Sorting

- We seek algorithms to arrange items, a_i such that:

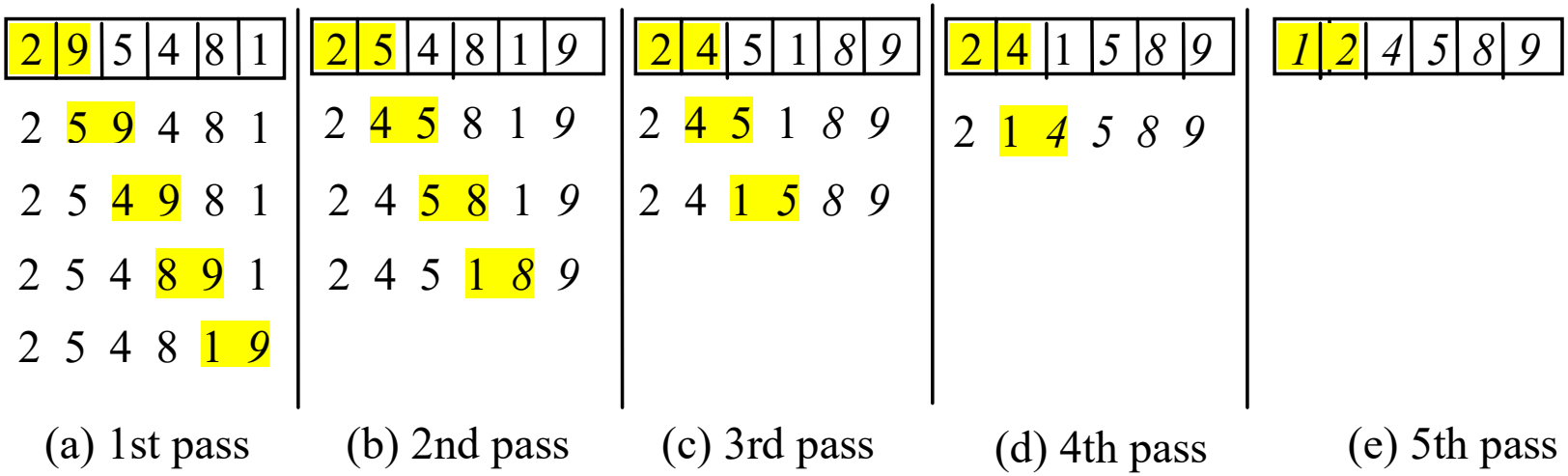
  entry 1 ≤ entry 2 ≤ . . . ≤ entry n

- Sorting an array is usually easier than sorting other data structures.
- Efficiency of a sorting algorithm is significant
- The data to be sorted might be integers, doubles, characters, or objects.

Bubble Sort

- Bubble sort
  - Strategy
    - Compare adjacent elements and exchange them if they are out of order
      - Comparing the first two elements, the second and third elements, and so on, will move the largest (or smallest) elements to the end of the array
      - Repeating this process will eventually sort the array into ascending (or descending) order

Bubble Sort



(a) 1st pass (b) 2nd pass (c) 3rd pass (d) 4th pass (e) 5th pass

BubbleSort

Selection Sort

Steps for selection sort.

- Scan the array and find the minimum item.
- Swap the item with the first location.
- Now the first location number is in it’s correct position.
- Scan the rest of the array and repeat steps one and two.
- Repeat the steps above to get a fully sorted array.

Insertion Sort

Insertion sort is relatively faster than bubble sort and selection sort. Here are the steps.

- Have a sorted and unsorted region in your list.
- Initially the sorted region will contain the first item and the unsorted region will contain the rest of the array.
- Compare the first in the unsorted region with the item in the sorted region. Place in correct location.
- Increase the sorted region by one.
- Repeat the above process for rest of the items in the unsorted region.

Insertion Sort

The insertion sort algorithm sorts a list of values by repeatedly inserting an unsorted element into a sorted sublist until the whole list is sorted.

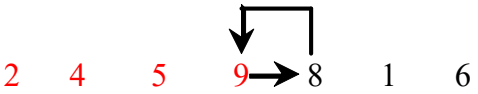
int[] myList = {2, 9, 5, 4, 8, 1, 6}; // Unsorted

Step 1: Initially, the sorted sublist contains the first element in the list. Insert 9 into the sublist.

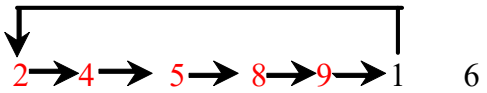
Step2: The sorted sublist is {2, 9}. Insert 5 into the sublist.

Step 3: The sorted sublist is {2, 5, 9}. Insert 4 into the sublist.

Step 4: The sorted sublist is {2, 4, 5, 9}. Insert 8 into the sublist.



Step 5: The sorted sublist is {2, 4, 5, 8, 9}. Insert 1 into the sublist.



Step 6: The sorted sublist is {1, 2, 4, 5, 8, 9}. Insert 6 into the sublist.

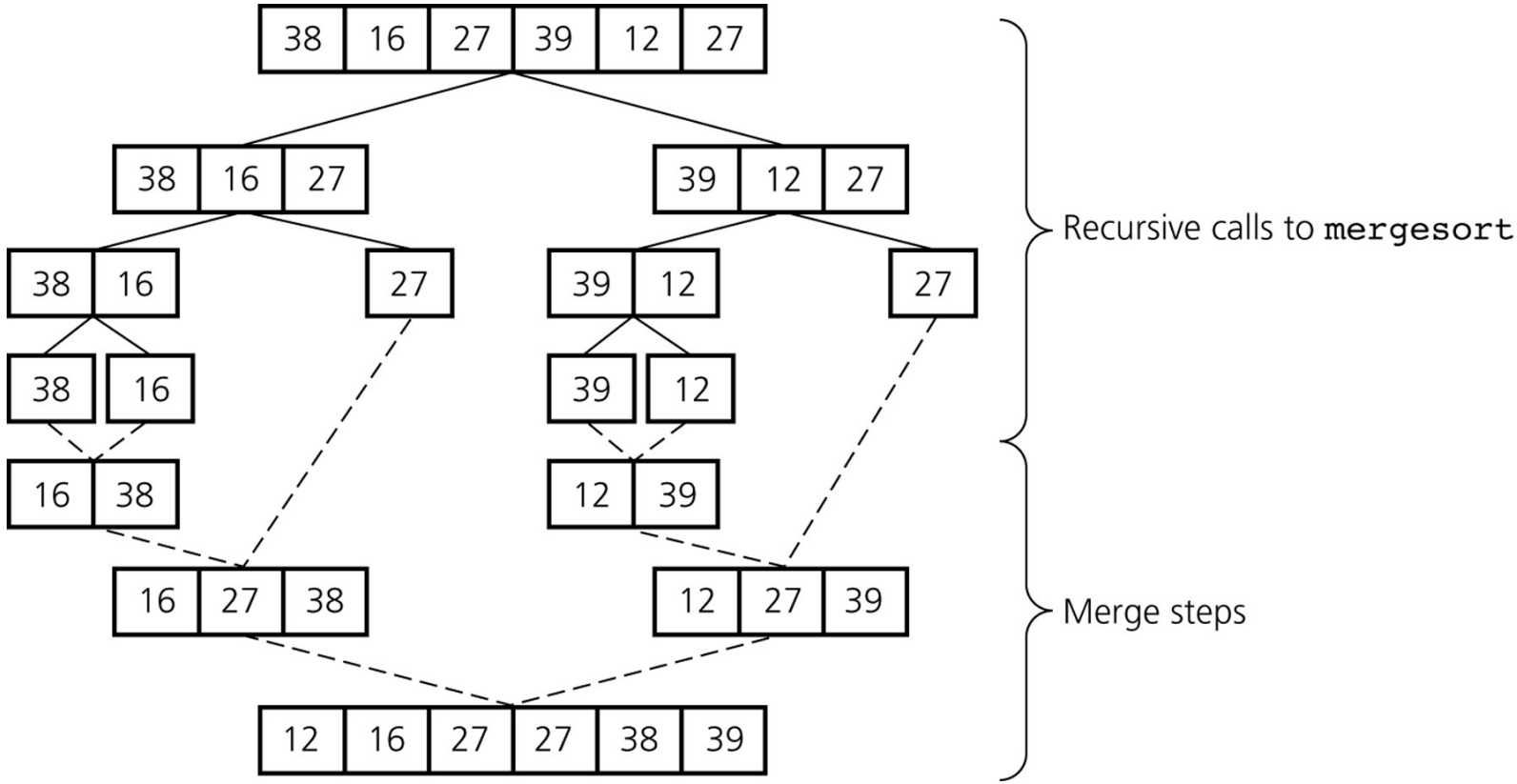
Step 7: The entire list is now sorted.

1 2 4 5 6 8 9

Mergesort

- Important divide-and-conquer sorting algorithms
  - Mergesort
  - Quicksort
- Mergesort
  - A recursive sorting algorithm
  - Gives the same performance, regardless of the initial order of the array items
  - Strategy
    - Divide an array into halves
    - Sort each half
    - Merge the sorted halves into one sorted array

Mergesort



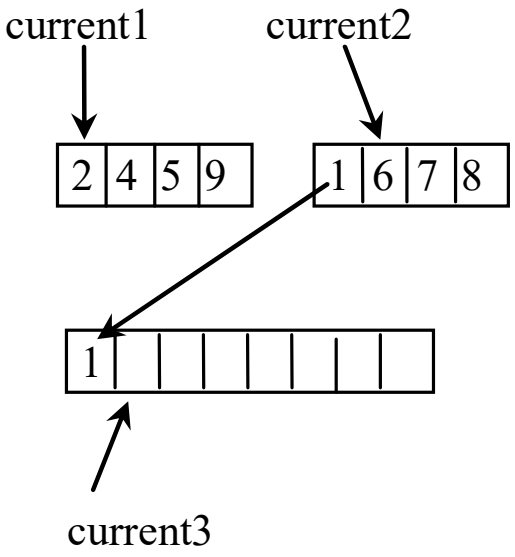
A mergesort of an array of six integers

Recursive Merge Sort

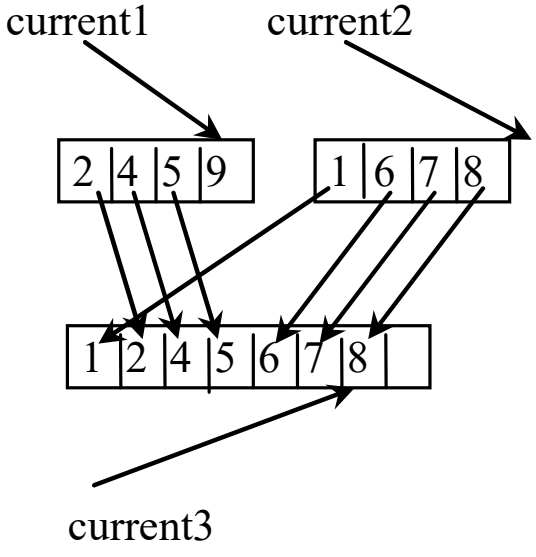
- Recursive algorithm for merge sort.

```
Algorithm  mergeSort(list) {
if (list.length > 1)
{
      // mergeSort the first half
      // mergeSort the second half
      merge(firstHalf,secondHalf,list)
}
}
```

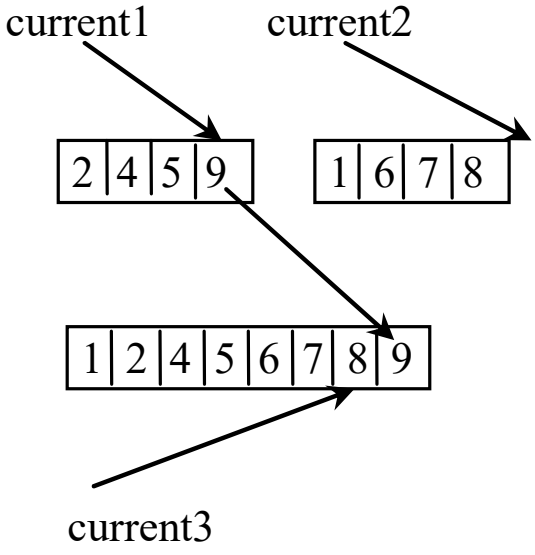
Merge Two Sorted Lists



(a) After moving 1 to temp



(b) After moving all the elements in list2 to temp



(c) After moving 9 to temp

Animation for Merging Two Sorted Lists

Analysis

- Dividing an array in half requires logN time.
- Merging requires N comparisons
- Efficiency of mergesort is O(N* logN)

Mergesort

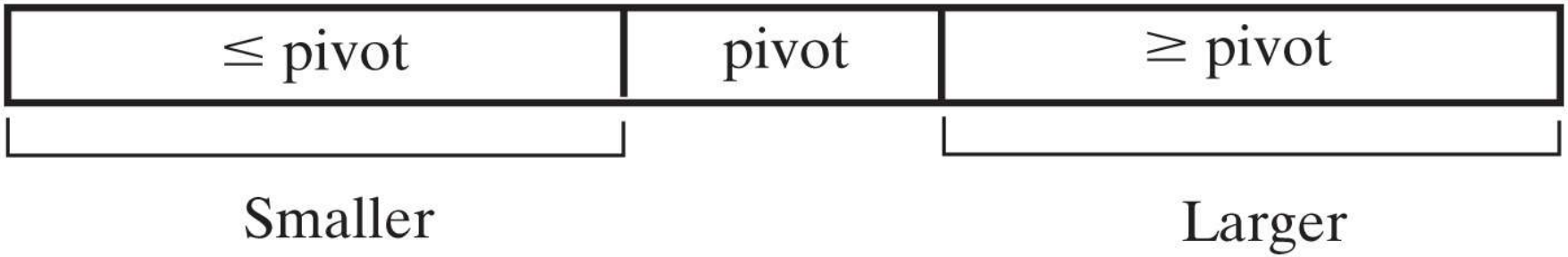
- Analysis
  - Worst case: $O(n * \log_2 n)$
  - Average case: $O(n * \log_2 n)$
  - Advantage
    - It is an extremely efficient algorithm with respect to time
  - Drawback
    - It requires a second array as large as the original array

Quick Sort

- Divides an array into two pieces
  - Pieces are not necessarily halves of the array
  - Chooses one entry in the array—called the pivot
    - Pivot is in position that it will occupy in final sorted array
    - Entries in positions before pivot are less than or equal to pivot
    - Entries in positions after pivot are greater than or equal to pivot
  - Recursively apply the quick sort algorithm to the first part and then the second part.

```
Algorithm quickSort(list, first, last)
// Sorts the array entries list[first..last] recursively.
  if (last > first)
  {
        Choose a pivot
        Partition the array about the pivot
        pivotIndex = index of pivot
        quickSort(list, first, pivotIndex − 1) // Sort Smaller
        quickSort(list, pivotIndex + 1, last) //Sort Larger
  }
```

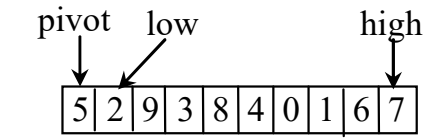
Quick Sort



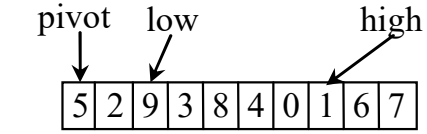
Partition

Animation for partition

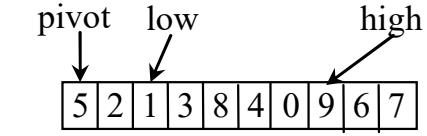
Quick Sort



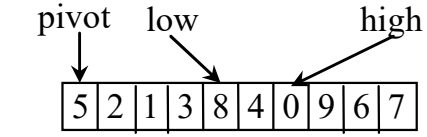
(a) Initialize pivot, low, and high



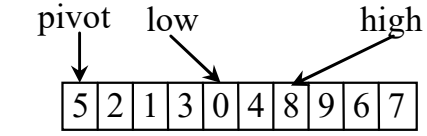
(b) Search forward and backward



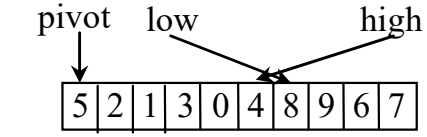
(c) 9 is swapped with 1



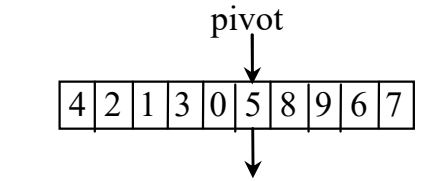
(d) Continue search



(e) 8 is swapped with 0



(f) when high < low, search is over



(g) pivot is in the right place

Analysis

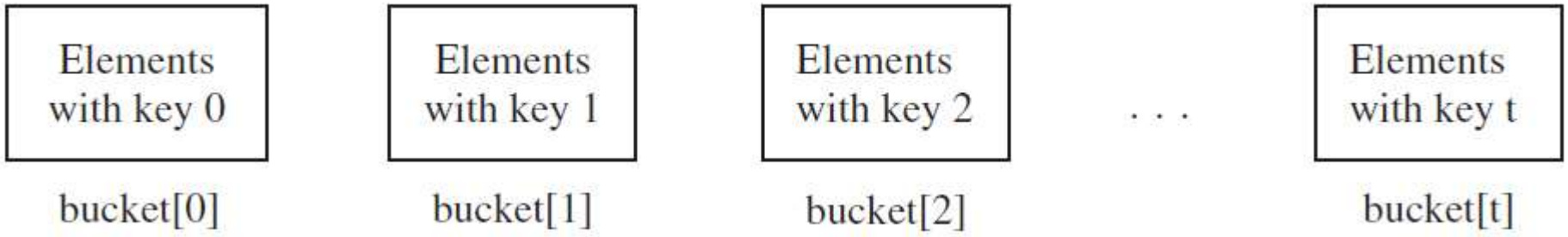
- A single partition runs in O(N) time.
- Dividing an array in half requires logN time.
- Efficiency of quicksort is O(N* logN)

Bucket Sort and Radix Sort

All sort algorithms discussed so far are general sorting algorithms that work for any types of keys (e.g., integers, strings, and any comparable objects). These algorithms sort the elements by comparing their keys. The lower bound for general sorting algorithms is O(nlogn). So, no sorting algorithms based on comparisons can perform better than O(nlogn). However, if the keys are small integers, you can use bucket sort without having to compare the keys.

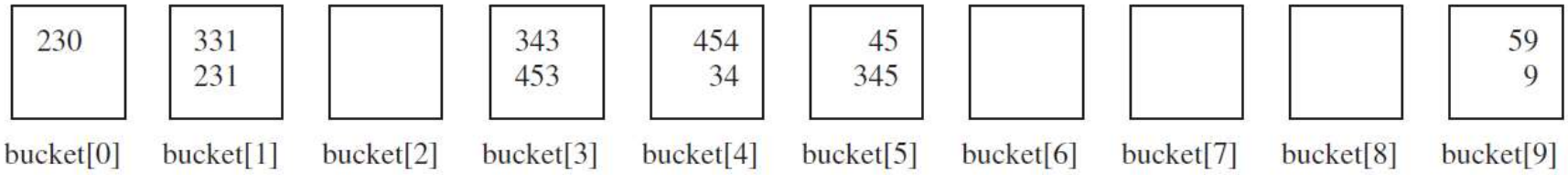
Bucket Sort

The bucket sort algorithm works as follows. Assume the keys are in the range from 0 to N-1. We need N buckets labeled 0, 1, ..., and N-1. If an element's key is i, the element is put into the bucket i. Each bucket holds the elements with the same key value. You can use an ArrayList to implement a bucket.

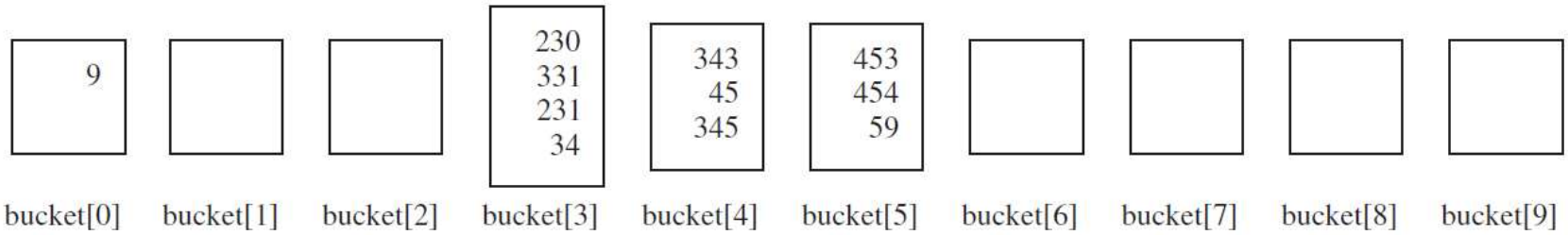


Radix Sort

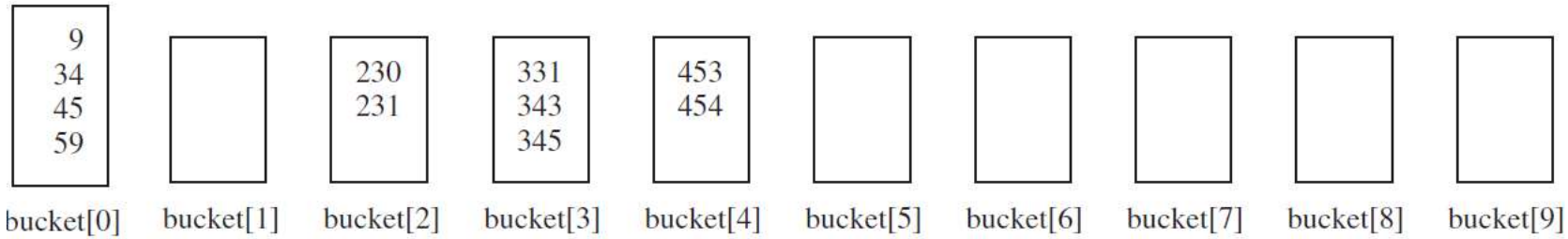
Sort 331, 454, 230, 34, 343, 45, 59, 453, 345, 231, 9



230, 331, 231, 343, 453, 454, 34, 45, 345, 59, 9



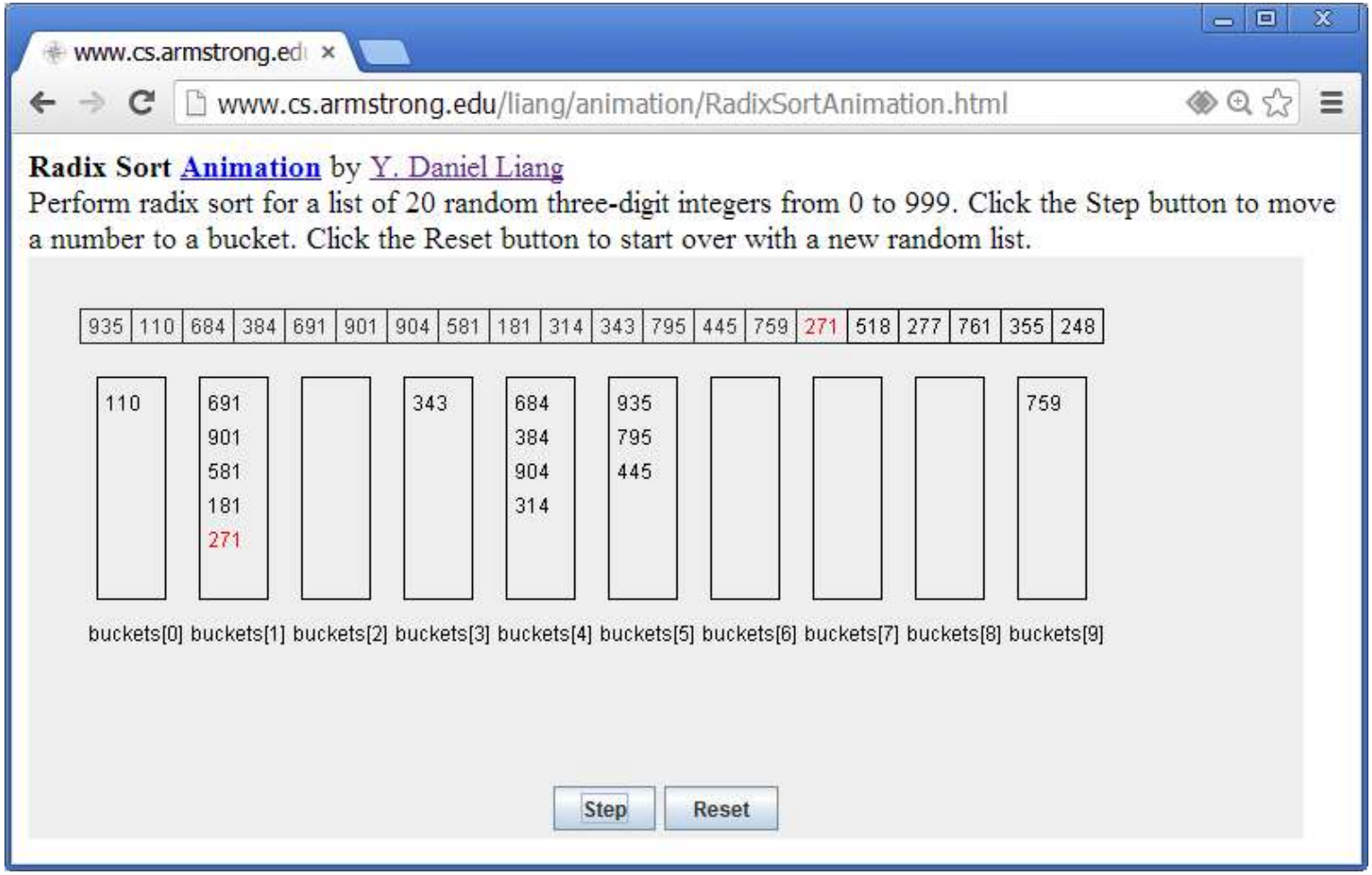
9, 230, 331, 231, 34, 343, 45, 345, 453, 454, 59



9, 34, 45, 59, 230, 231, 331, 343, 345, 453, 454

Radix Sort Animation

https://liveexample.pearsoncmg.com/dsanimation/RadixSorteBook.html



A Comparison of Sorting Algorithms

	Worst case	Average case
Selection sort	n^2	n^2
Bubble sort	n^2	n^2
Insertion sort	n^2	n^2
Mergesort	n * log n	n * log n
Quicksort	n^2	n * log n
Radix sort	n	n
Treesort	n^2	n * log n
Heapsort	n * log n	n * log n

Approximate growth rates of time required for eight sorting algorithms