COMP 355
Advanced Algorithms

Sorting and Selection Review

Sorting

Given n elements, rearrange in ascending order.

Obvious sorting applications.
- List files in a directory.
- Organize an MP3 library.
- List names in a phone book.
- Display Google PageRank results.

Problems become easier once sorted.
- Find the median.
- Find the closest pair.
- Binary search in a database.
- Identify statistical outliers.
- Find duplicates in a mailing list.

Non-obvious sorting applications.
- Data compression.
- Computer graphics.
- Interval scheduling.
- Computational biology.
- Minimum spanning tree.
- Supply chain management.
- Simulate a system of particles.
- Book recommendations on Amazon.
- Load balancing on a parallel computer.
- . . .

Sorting Algorithms

Usually divided into two classes,

- *internal sorting algorithms*, (assume that data is stored in an array in main memory
- *external sorting algorithm* (assume that data is stored on disk or some other device that is best accessed sequentially.

We will only consider internal sorting.

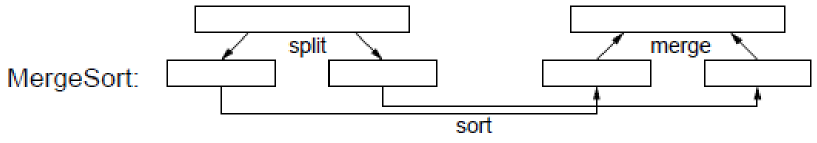
Two Important Properties

- **In-place:** The algorithm uses no additional array storage, and hence (other than perhaps the system's recursion stack) it is possible to sort very large lists without the need to allocate additional working storage.
- **Stable:** A sorting algorithm is stable if two elements that are equal remain in the same relative position after sorting is completed. This is of interest, since in some sorting applications you sort first on one key and then on another. It is nice to know that two items that are equal on the second key, remain sorted on the first key.

MergeSort

Classic divide-and-conquer algorithm

- Recursively sort each half.
- Divide array into two halves.
- Merge two halves to make sorted whole.



A	L	G	O	R	I	T	H	M	S		
A	L	G	O	R	I	T	H	M	S	divide	O(1)
A	G	L	O	R	H	I	M	S	T	sort	2T(n/2)
A	G	H	I	L	M	O	R	S	T	merge	O(n)

- Advantage: *stable*
- Disadvantage: *not in-place*.

Selection

- Related to sorting
- Given an array A of n numbers (not sorted) and an integer k, where $1 \leq k \leq n$, return the kth smallest value of A.
- Easy algorithm:
  - Sort the array (Θ (n log n))
  - Return kth element
- Harder algorithm
  - O(n)
  - Variant of QuickSort

Lower Bounds for Comparison-Based Sorting

- O(n log n) sorting algorithms have been the fastest algorithms for many years.
- Can we sort faster?

- **Theorem:** Any comparison-based sorting algorithm has worst-case running time Ω(*n* log *n*).

Linear-Time Sorting

- The Ω(*n* log *n*) lower bound implies that if we hope to sort numbers faster than in *O*(*n* log *n*) time, we cannot do it by making comparisons alone.
- **Counting Sort**: assumes each integer in range from 1 to k.
- **Radix Sort**: only practical for very small ranges of integers.
- **BucketSort**: works for floating-point numbers, but should only be used if numbers are roughly uniformly distributed over some range.

Counting Sort

```
COUNTING-SORT$(A, B, n, k)$
    let $C[0..k]$ be a new array
    for $i = 0$ to $k$
        $C[i] = 0$
    for $j = 1$ to $n$
        $C[A[j]] = C[A[j]] + 1$
    for $i = 1$ to $k$
        $C[i] = C[i] + C[i-1]$
    for $j = n$ downto 1
        $B[C[A[j]]] = A[j]$
        $C[A[j]] = C[A[j]] - 1$
```

Example

	1	2	3	4	5	6	7	8
A	2	5	3	0	2	3	0	3

A = array you need to sort

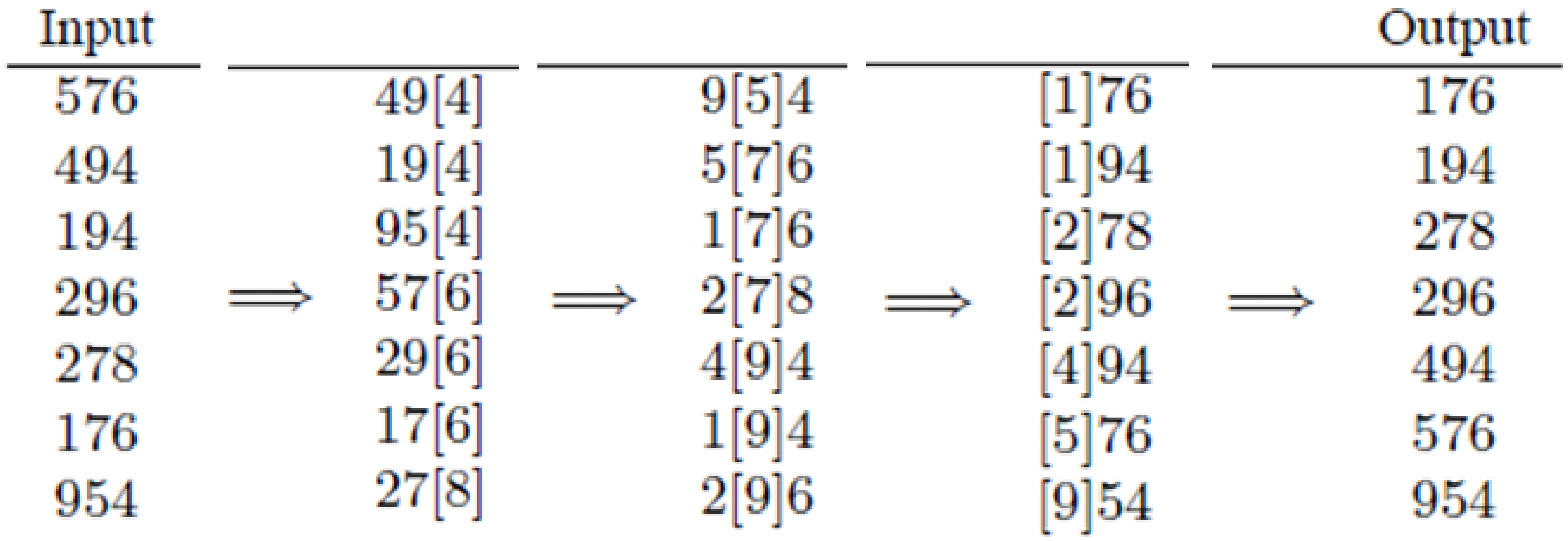
B = empty array of size n

n = len(A)

k = max value in A

Radix Sort

```
RADIX_SORT(A, d):
        for i = 1 to d:
                use stable sort to sort array A on digit i
```



Input								Output
576		49[4]		9[5]4		[1]76		176
494		19[4]		5[7]6		[1]94		194
194		95[4]		1[7]6		[2]78		278
296	$\Longrightarrow$	57[6]	$\Longrightarrow$	2[7]8	$\Longrightarrow$	[2]96	$\Longrightarrow$	296
278		29[6]		4[9]4		[4]94		494
176		17[6]		1[9]4		[5]76		576
954		27[8]		2[9]6		[9]54		954