香港中文大學（深圳）
The Chinese University of Hong Kong

CSC3100 Data Structures
Lecture 12: More sorting algorithms

Li Jiang
School of Data Science (SDS)
The Chinese University of Hong Kong, Shenzhen

Outline

- Comparison-based sorting algorithms
  - ShellSort
- Non-comparison-based sorting algorithms
  - CountingSort
  - BucketSort
  - RadixSort
- A summary of 10 classic sorting algorithms

ShellSort

- Break the quadratic time barrier by comparing elements that are distant

- The distance between comparisons decreases as the algorithm runs until the last phase, in which adjacent elements are compared (*diminishing increment sort*)

- An increment sequence $h_1, h_2, h_3, \ldots, h_t$, used in reverse order with $h_1=1$

ShellSort

- After a phase, with an increment h_k, A[i] <= A[i + h_k]
- All elements spaced h_k apart are sorted (insertion sort)

- Example
  - Consider an increment sequence: 1, 3, 5
  - For each h_k, we need to sort h_k subsequences

Original	**81**	**94**	**11**	**96**	**12**	**35**	**17**	**95**	**28**	**58**	**41**	**75**	**15**
5-sorted	**35**	**17**	**11**	**28**	**12**	**41**	**75**	**15**	**96**	**58**	**81**	**94**	**95**
3-sorted	**28**	**12**	**11**	**35**	**15**	**41**	**58**	**17**	**94**	**75**	**81**	**96**	**95**
1-sorted	**11**	**12**	**15**	**17**	**28**	**35**	**41**	**58**	**75**	**81**	**94**	**95**	**96**

Standard insertion sort

ShellSort with {1,2,4,8,...,n/2}

```
public static void shellSort(int[ ] a) {
    int j;
    for (int gap = a.length/2; gap > 0; gap /=2)
        for (int i = gap; i < a.length; i++) {
            int tmp = a[i];
            for (j = i; j >= gap && tmp < a[j-gap]; j-= gap)
                a[j] = a[j-gap];
            a[j] = tmp;
        }
}
```

ShellSort

- Analysis of Shellsort
  - Very hard (average-case is a long-standing open problem)
  - Depend on the selection of an increment sequence
    - Difference sequences lead to different time cost!

  - **Theorem**: the worst-case running time of Shellsort, using some increment, is $\Theta(N^2)$
    - Put the largest N/2 numbers in the even positions

      e.g., 4,12,1,10,3,11,2,9
    - Use the increments {…, 8,4,2,1}
    - Before the last sort, the N/2 largest numbers are still in the even positions, e.g., 1,9,2,10,3,11,4,12
    - The numbers of inversions is 1+2+…+(N-1)/2 = $\Theta(N^2)$

CountingSort

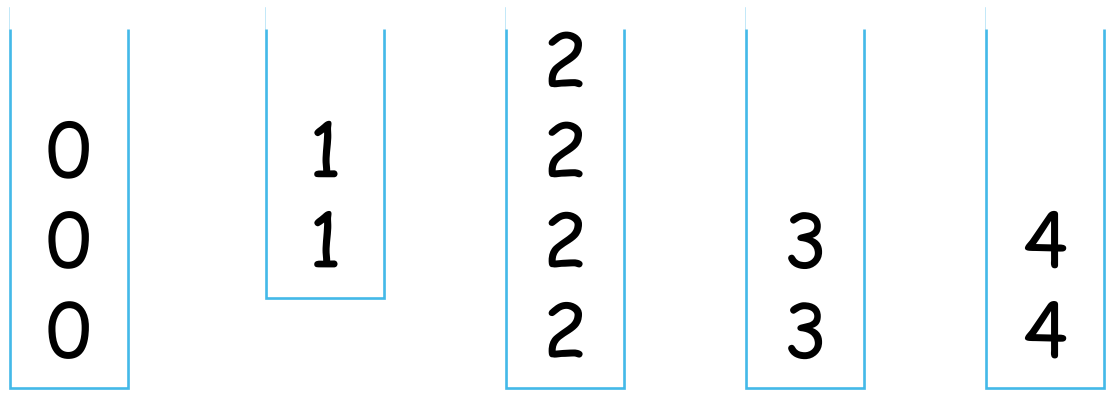
- Assumption: all the values are integers in [0, m-1]

- Steps
  - Start with m empty buckets numbered 0 to m-1
  - Scan the list and place element s[i] in bucket s[i]
  - Output the buckets in order

- Will need an array of buckets, and the values to be sorted will be the indexes to the buckets
  - No comparisons will be necessary

CountingSort

4	2	1	2	0	3	2	1	4	0	2	3	0



0	0	0	1	1	2	2	2	2	3	3	4	4

CountingSort

Algorithm CountingSort(s[ ])
(values in s[ ] are between 0 and m-1)

```
for j ← 0 to m-1 do      // initialize m buckets
   b[j] ← 0
for i ← 0 to n-1 do      // place elements in their appropriate buckets
   b[s[i]] ← b[s[i]]  + 1
i ← 0
for j ← 0 to m-1 do      // place elements in buckets
   for r ← 1 to b[j] do // back in s
         S[i] ← j
         i ← i + 1
```

Exercise

- Use CountingSort to sort the following sequence of integer values

  4,3,2,1,5,2,4,1,2,0,4,2,0

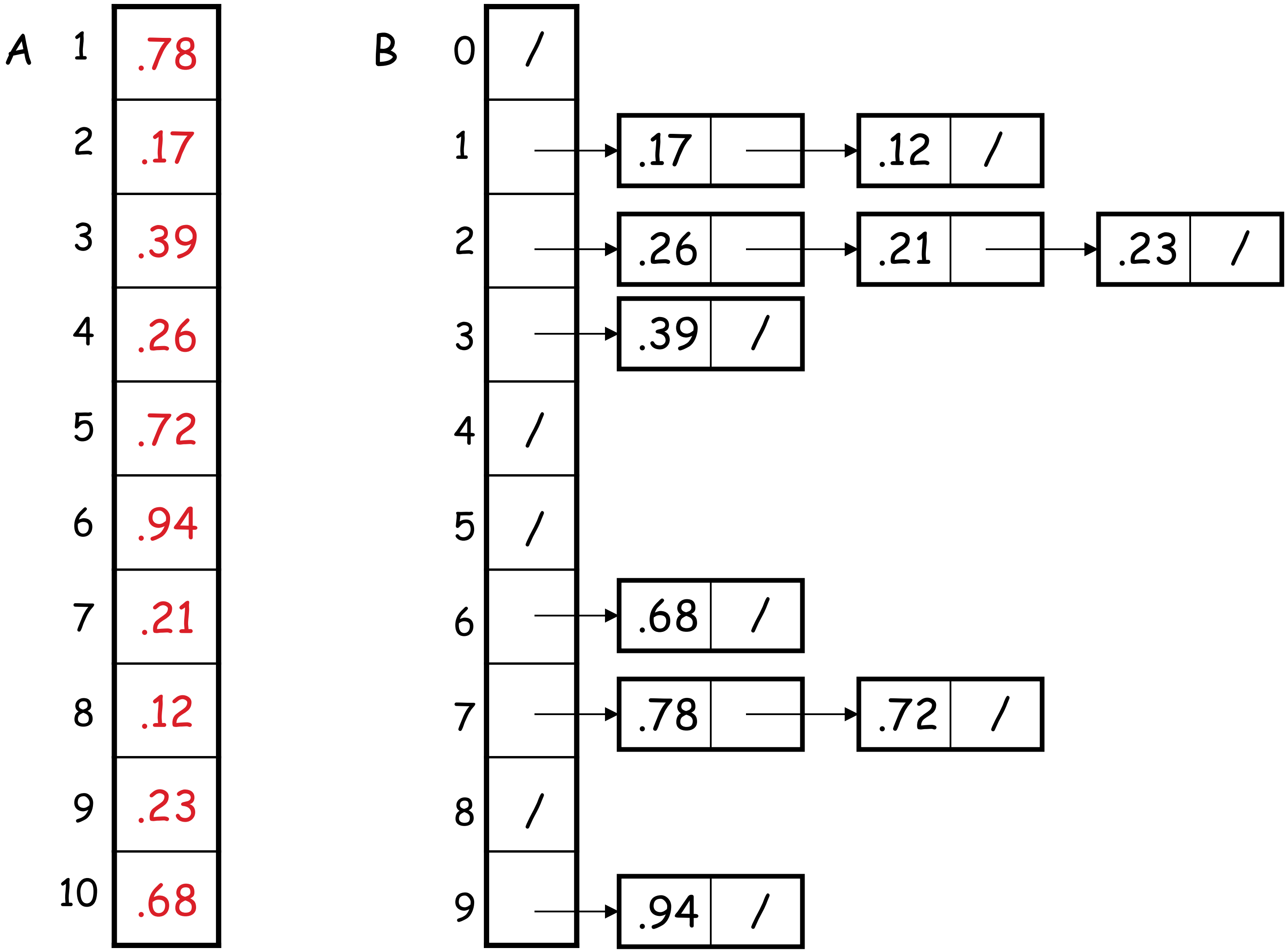
- How to process the case that the minimum value in the input sequence of integers is very large?

BucketSort

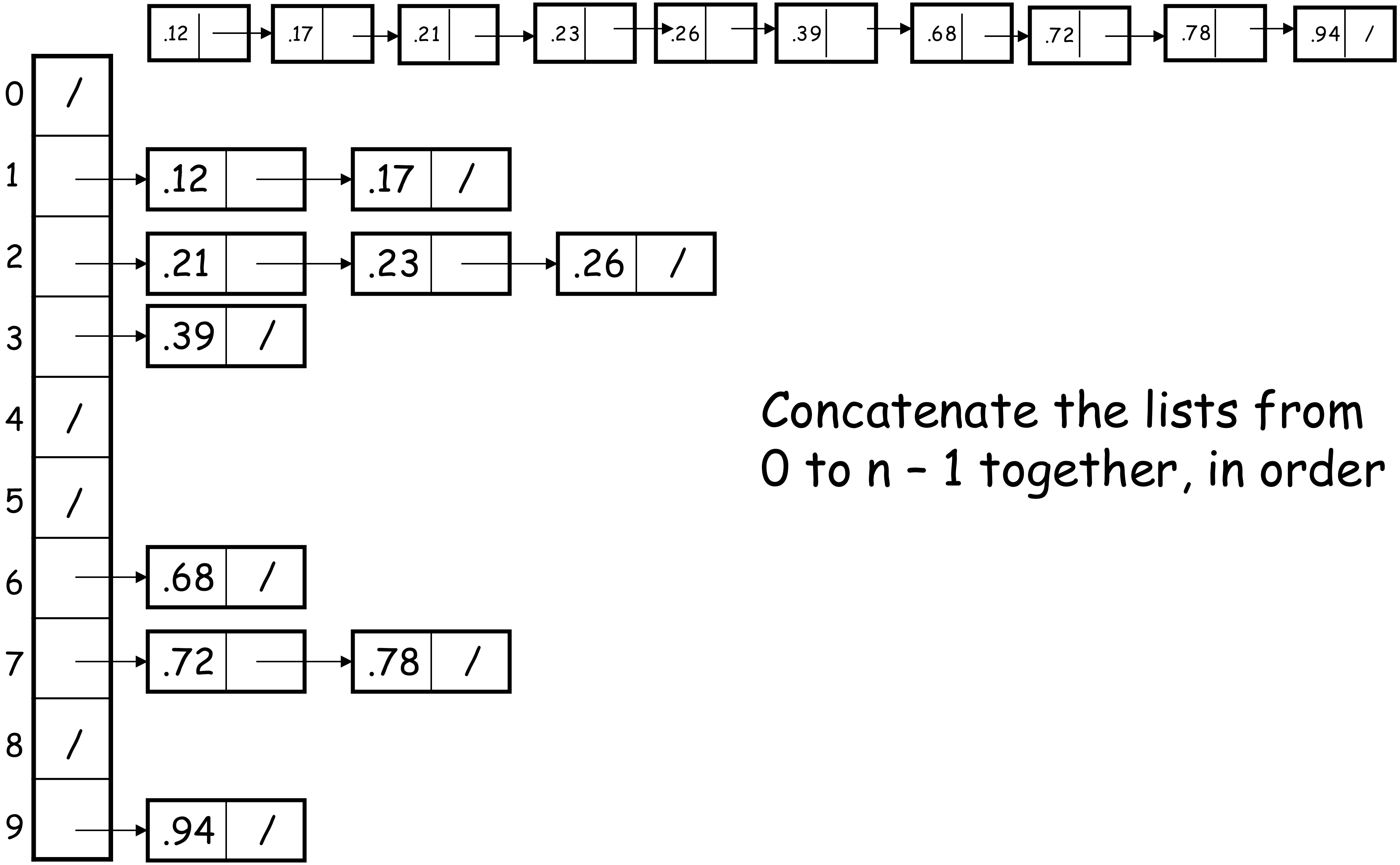
- Assumption:
  - The input is generated by a random process that distributes elements uniformly over [0, 1)

- Key steps:
  - Divide [0, 1) into n equal-sized buckets
  - Distribute the n input values into the buckets
  - Sort each bucket (e.g., using QuickSort)
  - Go through buckets in order, listing elements in each one
  - Extra array: B[0 . . n - 1] of linked lists, each of which is initially empty

BucketSort


A
1 .78
2 .17
3 .39
4 .26
5 .72
6 .94
7 .21
8 .12
9 .23
10 .68
B
0 /
1 .17 .12 /
2 .26 .21 .23 /
3 .39 /
4 /
5 /
6 .68 /
7 .78 .72 /
8 /
9 .94 /

BucketSort



Concatenate the lists from 0 to n – 1 together, in order

BucketSort

BUCKET-SORT(A, n)

for $i \leftarrow 1$ **to** n $\quad O(n)$

  do insert $A[i]$ into list $B[\lfloor nA[i] \rfloor]$

for $i \leftarrow 0$ **to** $n - 1$ $\quad O(n)$

  do sort list $B[i]$ with QuickSort

concatenate lists $B[0], B[1], \ldots, B[n-1]$ $\quad O(n)$

together in order

return the concatenated lists

$O(n)$

Exercise

- Use BucketSort to sort the following sequence of real values

  0.81, 0.65, 0.91, 0.61, 0.55, 0.71

RadixSort

- CountingSort is costly in both time and space, if m is large
- The idea of RadixSort:
  - Apply BucketSort on each digit (from Least Significant Digit to Most Significant Digit)
- A complication:
  - Just keeping the count is not enough
  - Need to keep the actual elements
  - Use a queue for each digit

RadixSort: example (1/3)

- Input: 170, 045, 075, 090, 002, 024, 802, 066
- The **first** pass
  - Consider **the least significant digits** as keys and move the keys into their buckets

0	170, 090
1	
2	002, 802
3	
4	024
5	045, 075
6	066
7	
8	
9	

  - Output: 170, 090, 002, 802, 024, 045, 075, 066

RadixSort: example (2/3)

- The **second** pass
  - Input: 170, 090, 002, 802, 024, 045, 075, 066
  - Consider **the second least significant digits** as keys and move the keys into their buckets

0	002, 802
1	
2	024
3	
4	045
5	
6	066
7	170, 075
8	
9	090

  - Output: 002, 802, 024, 045, 066, 170, 075, 090

RadixSort: example (3/3)

- The **third** pass
  - Input: 002, 802, 024, 045, 066, 170, 075, 090
  - Consider **the third least significant digits** as keys and move the keys into their buckets

0	002, 024, 045, 066, 075, 090
1	170
2	
3	
4	
5	
6	
7	
8	802
9	

  - Output: 002, 024, 045, 066, 075, 090, 170, 802 (Sorted)

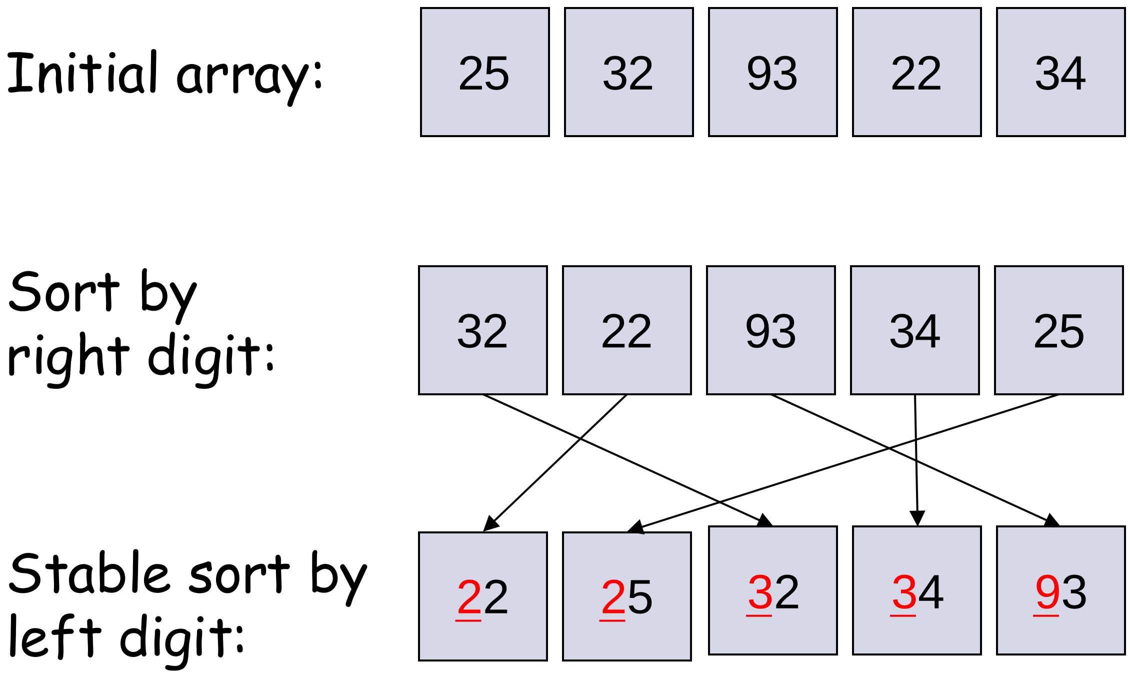
Another example

- Suppose we sort some 2-digit integers
- **Phase 1: Sort by the right digit (the least significant digit)**

Initial array:	25	32	93	22	34
Sort by right digit:	3<u>2</u>	2<u>2</u>	9<u>3</u>	3<u>4</u>	2<u>5</u>

Another example (cont.)

- **Phase 2: Sort by the left digit (the second least significant digit)**



Codes (1/2)

```
// items to be sorted are in {0,…,10^d-1},
// i.e., the type of d-digit integers
void radixsort(int A[], int n, int d)
{
   int i;
   for (i=0; i<d; i++)
      bucketsort(A, n, i);
}

// To extract d-th digit of x
int digit(int x, int d)
{
   int i;
   for (i=0; i<d; i++)
      x /= 10; // integer division
   return x%10;
}
```

Codes (2/2)

```
void bucketsort(int A[], int n, int d)
// stable-sort according to d-th digit
{
   int i, j;
   Queue *C = new Queue[10];
   for (i=0; i<10; i++) C[i].makeEmpty();
   for (i=0; i<n; i++)
      C[digit(A[i],d)].EnQueue(A[i]);
   for (i=0, j=0; i<10; i++)
      while (!C[i].empty())
      { // copy values from queues to A[]
         A[j] = C[i].DeQueue();
         j++;
      }
}
```

Inductive proof that RadixSort works

- Keys: k-digit numbers, base B
  - (that wasn't hard!)

- Hypothesis: after i^{th} RadixSort, the least significant i digits are sorted
  - Base case: $i = 0$, implying 0 digits are sorted
  - Inductive step: Assume for i, prove for $i+1$

    Consider two numbers: X, Y. Say X_i is i^{th} digit of X:
    - $X_{i+1} < Y_{i+1}$ then $i+1^{th}$ RadixSort will put them in order
    - $X_{i+1} > Y_{i+1}$, same thing
    - $X_{i+1} = Y_{i+1}$, order depends on last i digits. Induction hypothesis says already sorted for these digits because RadixSort is **stable**

Worst-case time complexity

- Assume k digits, each digit comes from {0,…,M-1}

- For each digit,
  - O(M) time to initialize M queues
  - O(n) time to distribute n numbers into M queues

- Total time = O(k(M+n))
- When k is constant and M = O(n), we can make RadixSort run in linear time, i.e., O(n)

Exercises

- Use RadixSort to sort the following sequence

  123, 251, 369, 278, 451, 222

- Can we start from the most significant digit?

  Now let sort three 3-digit numbers?

  478, 430, 356

  1st digit:

  4, 4, 3 => 3, 4, 4 => 356, 478, 430

  2nd digit:

  5, 7, 3 => 3, 5, 7 => 430, 356, 478

  3rd digit:

  0, 6, 8 => 0, 6, 8 => 430, 356, 478

Exercises

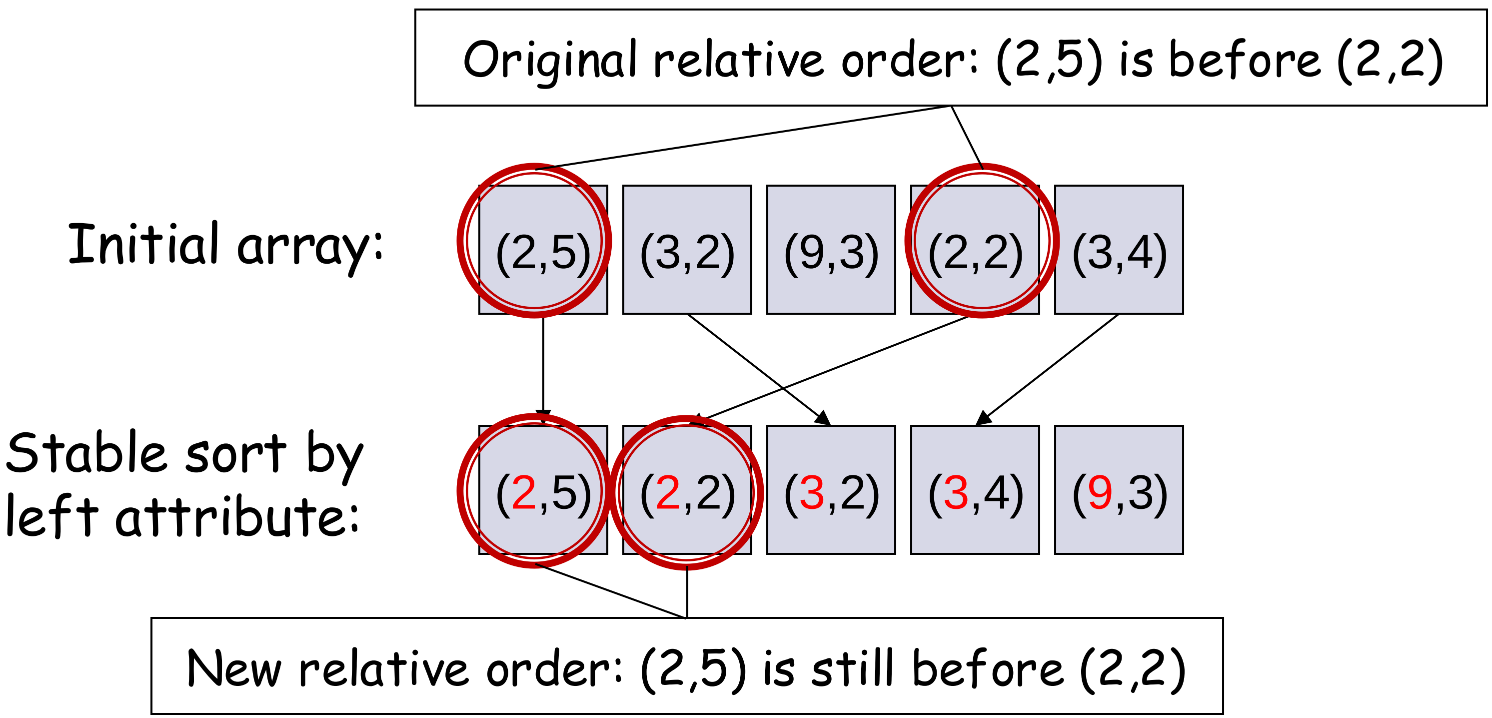
- Since RadixSort is faster than QuickSort, why is QuickSort still preferable in many cases?
  - Although RadixSort runs in $\Theta(n)$ while QuickSort $\Theta(n \lg n)$, QuickSort has a much smaller constant factor c
  - RadixSort requires extra memory, whereas QuickSort works in place

10 classic sorting algorithms

Sorting algorithm	Stability	Time cost			Extra space cost
		Best	Average	Worst	
Bubble sort	√	$O(n)$	$O(n^2)$	$O(n^2)$	$O(1)$
Insertion sort	√	$O(n)$	$O(n^2)$	$O(n^2)$	$O(1)$
Selection sort	×	$O(n^2)$	$O(n^2)$	$O(n^2)$	$O(1)$
MergeSort	√	$O(nlogn)$	$O(nlogn)$	$O(nlogn)$	$O(n)$
HeapSort	×	$O(nlogn)$	$O(nlogn)$	$O(nlogn)$	$O(1)$
QuickSort	×	$O(nlogn)$	$O(nlogn)$	$O(n^2)$	$O(1)$
ShellSort	×	$O(n)$	$O(n^{1.3})$	$O(n^2)$	$O(1)$
CountingSort	√	$O(n+k)$	$O(n+k)$	$O(n+k)$	$O(k)$
BucketSort	√	$O(n)$	$O(n+k)$	$O(n^2)$	$O(k)$
RadixSort	√	$O(nk)$	$O(nk)$	$O(nk)$	$O(n)$

Concept: stable sorting algorithm

- Definition: A *stable* sorting algorithm is one that preserves the original relative order of elements with equal key
  - E.g., suppose the left attribute is the key attribute



Recommended reading

- Reading this week
  - Chapter 8, textbook

- Next lecture
  - Tree data structure: Chapter 12