



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

CSC3100 Data Structures

Lecture 13: Sorting algorithms

Yixiang Fang
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

- ▶ External sorting



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 $h_1, h_2, h_3, \dots, h_t$, used in **reverse order** with **$h_1=1$**



ShellSort

- After a phase, with an increment h_k , $A[i] \leq A[i + h_k]$. All elements spaced h_k apart are sorted (insertion sort)
- Example (1,3,5)

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 $\{1, 2, 4, 8, \dots, n/2\}$

```
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
 - It is very hard (average-case is a long-standing open problem);
 - Results depend on the selection of an increment sequence;
 - Theorem:
 - The worst-case running time of Shellsort, using some increment, is $\Theta(N^2)$



ShellSort

► Theorem:

- The worst-case running time of Shellsort, using some increments, is $\Theta(N^2)$

► Proof (for some cases):

- Put the largest $N/2$ numbers in the even positions.
e.g., 4,12,1,10,3,11,2,9
- Using 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 $1+2+\dots+(N-1)/2 = \Theta(N^2)$



ShellSort

- ▶ Hibbard's increment $1, 3, 7, \dots, 2^k - 1$
(consecutive increments have no common factors)
 - Worst-case running time: $\Theta(N^{3/2})$
 - Average-case running time: $O(N^{5/4})$
- ▶ Sedgewick's increment $1, 5, 19, 41, 109, \dots$
(each term is either $9 \cdot 4^i - 9 \cdot 2^{i+1} + 1$ or $4^i - 3 \cdot 2^{i+1} + 1$)
 - Worst-case running time: $O(N^{4/3})$
 - Average-case running time: $O(N^{7/6})$



Non-comparison-based sorting algorithms



CountingSort

- ▶ Idea: suppose 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 in the list 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	1	2	3	4
0	1	2	3	4
0		2		
0		2		
		2		
		2		



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



CountingSort

Algorithm BucketSort(S)
(values in S are between 0 and $m-1$)

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



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?
- ▶ How to process the case that the values in the sequence vary greatly (i.e., 1, 10, 101, 1000, 100001) ?



BucketSort

▶ Assumption:

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

▶ Idea:

- 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 the buckets in order, listing elements in each one

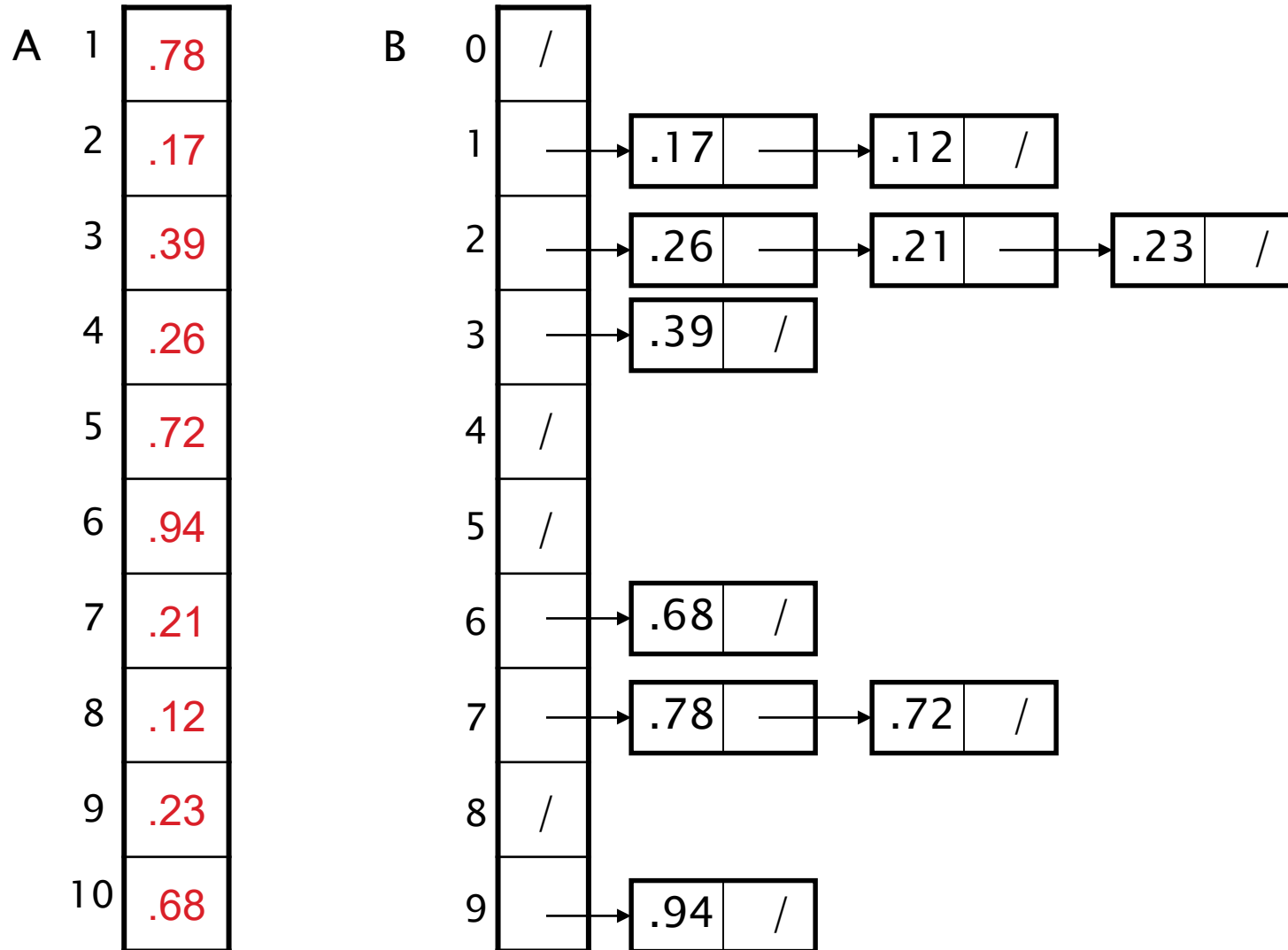
▶ **Input:** $A[1 \dots n]$, where $0 \leq A[i] < 1$ for all i

▶ **Output:** elements $A[i]$ sorted

▶ **Extra array:** $B[0 \dots n - 1]$ of linked lists, each list initially empty

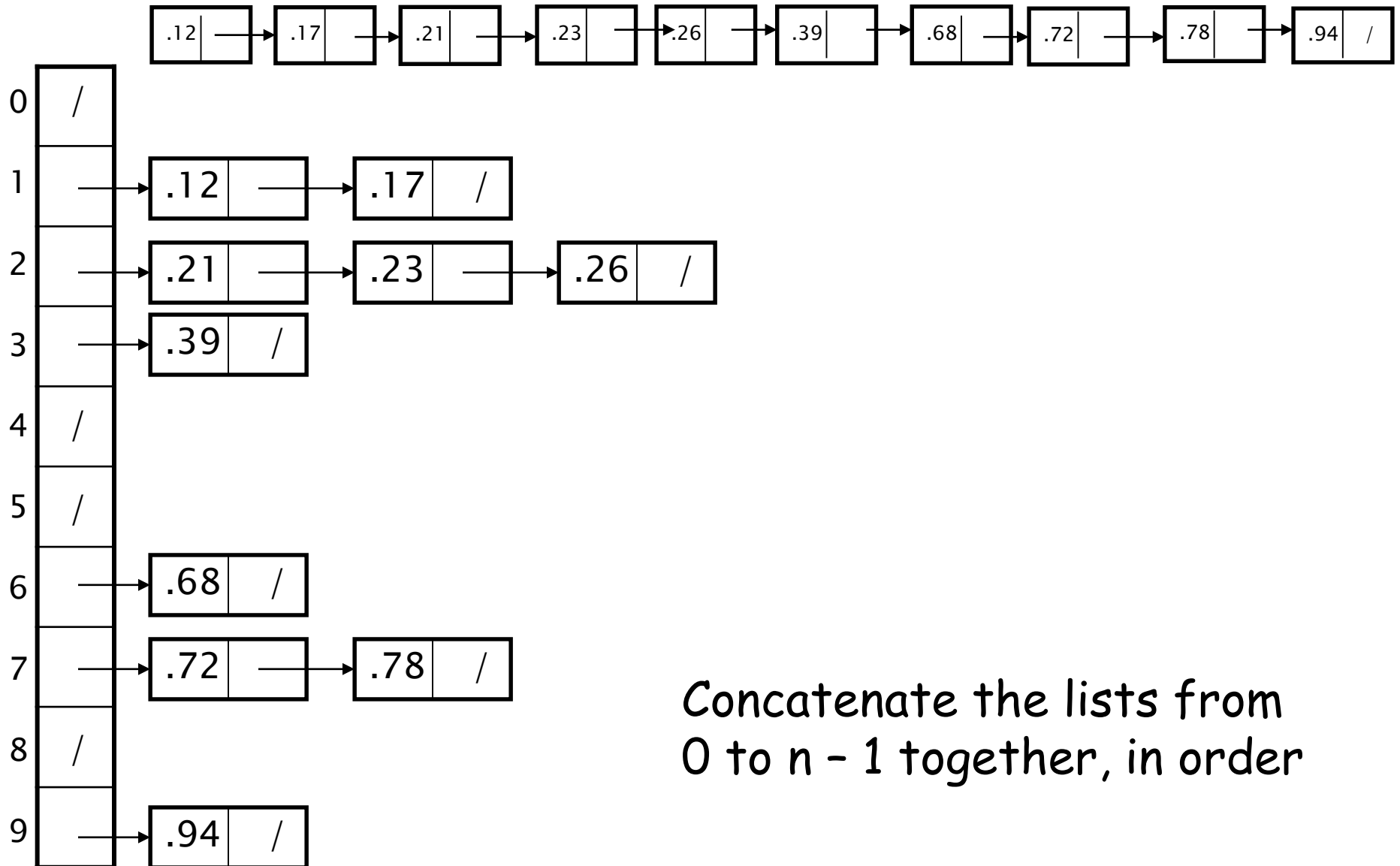


BucketSort





BucketSort



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



BucketSort

BUCKET-SORT(A, n)

for $i \leftarrow 1$ **to** n

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

for $i \leftarrow 0$ **to** $n - 1$

do sort list $B[i]$ with quicksort sort

concatenate lists $B[0], B[1], \dots, B[n - 1]$

together in order

return the concatenated lists

$O(n)$

$\Theta(n)$

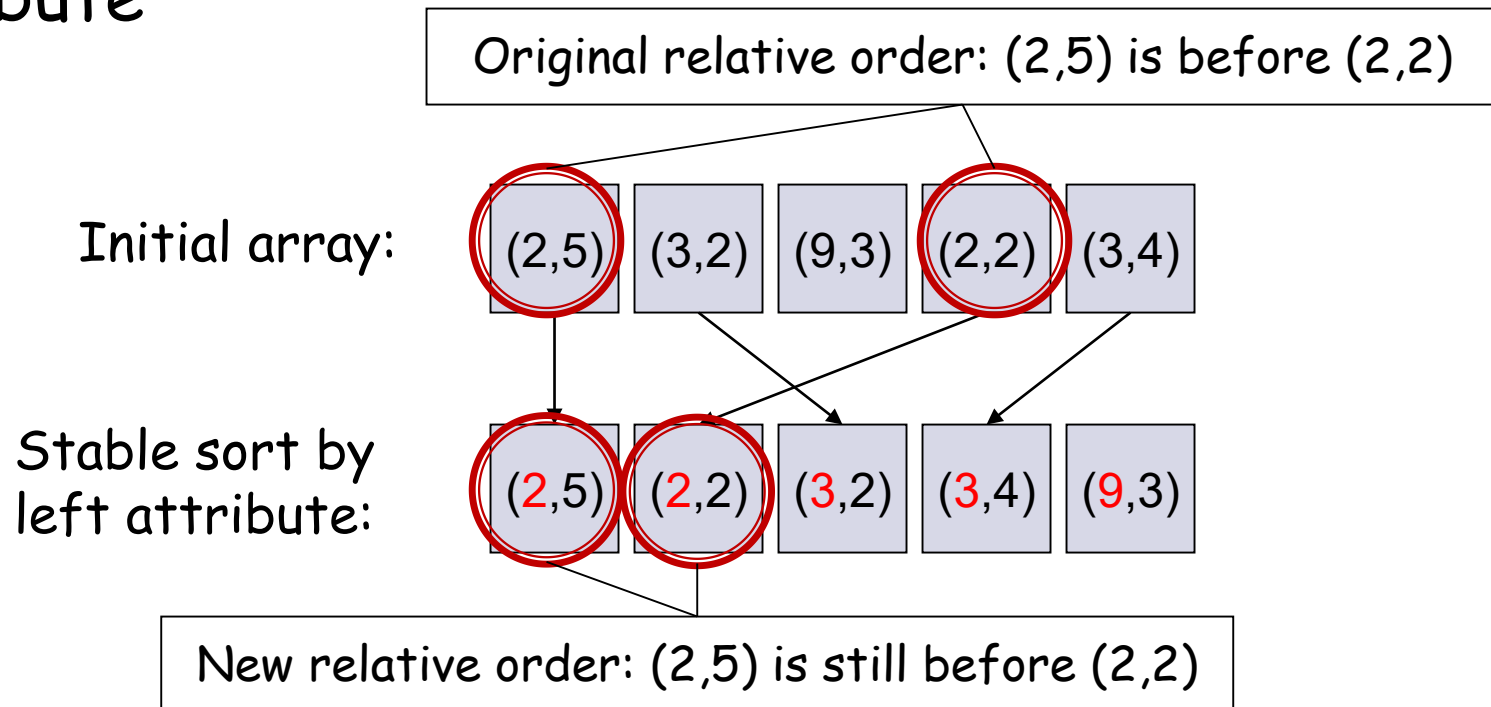
$O(n)$

$\Theta(n)$



Stable Sort

- ▶ 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





Using Stable Sort (1/4)

- ▶ Suppose we sort some 2-digit integers
- ▶ **Phase 1: Stable 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>
------------	------------	------------	------------	------------



Using Stable Sort (2/4)

- ▶ Suppose we sort some 2-digit integers
- ▶ **Phase 2: Stable sort by the left digit (the second least significant digit)**

Initial array:

25	32	93	22	34
----	----	----	----	----

Sort by
right digit:

32	22	93	34	25
----	----	----	----	----

Stable sort by
left digit:

<u>2</u> 2	<u>2</u> 5			
------------	------------	--	--	--



Using Stable Sort (3/4)

- ▶ Suppose we sort some 2-digit integers
- ▶ **Phase 2: Stable sort by the left digit (the second least significant digit)**

Initial array:

25	32	93	22	34
----	----	----	----	----

Sort by
right digit:

32	22	93	34	25
----	----	----	----	----

Stable sort by
left digit:

<u>2</u> 2	<u>2</u> 5	<u>3</u> 2	<u>3</u> 4	
------------	------------	------------	------------	--



Using Stable Sort (4/4)

- ▶ Suppose we sort some 2-digit integers
- ▶ **Phase 2: Stable sort by the left digit (the second least significant digit)**

Initial array:

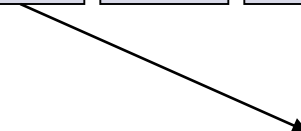
25	32	93	22	34
----	----	----	----	----

Sort by
right digit:

32	22	93	34	25
----	----	----	----	----

Stable sort by
left digit:

<u>2</u> 2	<u>2</u> 5	<u>3</u> 2	<u>3</u> 4	<u>9</u> 3
------------	------------	------------	------------	------------





RadixSort

- ▶ BucketSort is not efficient if m is large
- ▶ The idea of radix sort:
 - Apply stable bucket sort 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	17 <u>0</u> , 09 <u>0</u>
1	
2	00 <u>2</u> , 80 <u>2</u>
3	
4	02 <u>4</u>
5	04 <u>5</u> , 07 <u>5</u>
6	06 <u>6</u>
7	
8	
9	

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



Radix Sort (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	0 <u>0</u> 2, 8 <u>0</u> 2
1	
2	0 <u>2</u> 4
3	
4	0 <u>4</u> 5
5	
6	0 <u>6</u> 6
7	1 <u>7</u> 0, 0 <u>7</u> 5
8	
9	0 <u>9</u> 0

- 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	<u>0</u> 02, <u>0</u> 24, <u>0</u> 45, <u>0</u> 66, <u>0</u> 75, <u>0</u> 90
1	<u>1</u> 70
2	
3	
4	
5	
6	
7	
8	<u>8</u> 02
9	

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



Code (1/2)

```
// item is the type:  $\{0, \dots, 10^d - 1\}$ ,  
// i.e., the type of d-digit integers  
void radixsort(item 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(item x, int d)  
{  
    int i;  
    for (i=0; i<d; i++)  
        x /= 10; // integer division  
    return x%10;  
}
```



Code (2/2)

```
void bucketsort(item 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[]
            C[i].DeQueue(A[j]);
            j++;
        }
}
```



Inductive Proof that RadixSort Works

- ▶ Keys: k -digit numbers, base B
 - (that wasn't hard!)
- ▶ Claim: after i^{th} BucketSort, the least significant i digits are sorted
 - Base case: $i=0$. 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 $i+1^{\text{th}}$ BucketSort 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 BucketSort is **stable**



Worst-case Time Complexity

- ▶ Assume k digits, each digit comes from $\{0, \dots, M-1\}$
- ▶ 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., $O(n)$



Question

- ▶ Can we start from the most significant digit?

Now let sort three 3-digit numbers?

478, **4**30, **3**56

1st digit:

4, **4**, **3** => **3**, **4**, **4** => **3**56, **4**78, **4**30

2nd digit:

5, **7**, **3** => **3**, **5**, **7** => 4**3**0, 3**5**6, 4**7**8

3rd digit:

0, **6**, **8** => **0**, **6**, **8** => 43**0**, 35**6**, 47**8**



Questions

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



Exercise

- ▶ Show how RadixSort sorts the following sequence of integer values

140, 121, 356, 801, 911, 109, 056, 631



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	×	$O(n)$	$O(n^2)$	$O(n^2)$	$O(1)$
MergeSort	✓	$O(n \log n)$	$O(n \log n)$	$O(n \log n)$	$O(n)$
HeapSort	×	$O(n \log n)$	$O(n \log n)$	$O(n \log n)$	$O(1)$
QuickSort	×	$O(n \log n)$	$O(n \log n)$	$O(n^2)$	$O(\log n)$
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	✓	$O(nk)$	$O(nk)$	$O(nk)$	$O(n)$



External sort

- ▶ For the input on some sequential-access media like tapes and it is too large to be all incorporated into the memory
- ▶ Computing time becomes insignificant when compared with access time
- ▶ Merge routine from MergeSort is used

MERGE-SORT(A, p, r)

if $p < r$

 then $q \leftarrow \lfloor (p + r)/2 \rfloor$

 MERGE-SORT(A, p, q)

 MERGE-SORT($A, q + 1, r$)

 MERGE(A, p, q, r)

▷ Check for base case

▷ Divide

▷ Conquer

▷ Conquer

▷ Combine



External sort

- ▶ Assume we have four tape drives and four tapes
- ▶ The data in Tape T_{a1} are to be sorted
- ▶ The memory can read at most $M=3$

T_{a1}	81 94 11 96 12 35 17 99 28 58 41 75 15
T_{a2}	
T_{b1}	
T_{b2}	



External sort

- ▶ Read 3 data at a time from T_{a1} and write the sorted results into T_{b1} and T_{b2} alternatively


T_{a1}	81 94 11 96 12 35 17 99 28 58 41 75 15
T_{a2}	
T_{b1}	<u>11 81 94</u> <u>17 28 99</u> <u>15</u>
T_{b2}	<u>12 35 96</u> <u>41 58 75</u>



External sort

- ▶ Merge sorted groups from T_{b1} and T_{b2} and write to T_{a1} and T_{a2} alternatively, and repeat this step

T_{a1}	<u>11</u> <u>12</u> <u>35</u> <u>81</u> <u>94</u> <u>96</u> <u>15</u>
T_{a2}	<u>17</u> <u>28</u> <u>41</u> <u>58</u> <u>75</u> <u>99</u>
T_{b1}	<u>11</u> <u>81</u> <u>94</u> <u>17</u> <u>28</u> <u>99</u> <u>15</u>
T_{b2}	<u>12</u> <u>35</u> <u>96</u> <u>41</u> <u>58</u> <u>75</u>





External sort

T_{a1}	<u>11 12 35 81 94 96 15</u>
T_{a2}	<u>17 28 41 58 75 99</u>
T_{b1}	<u>11 12 17 28 35 41 58 75 81 94 96 99</u>
T_{b2}	<u>15</u>

T_{a1}	<u>11 12 15 17 28 35 41 58 75 81 94 96 99</u>
T_{a2}	
T_{b1}	<u>11 12 17 28 35 41 58 75 81 94 96 99</u>
T_{b2}	<u>15</u>

Finished!!



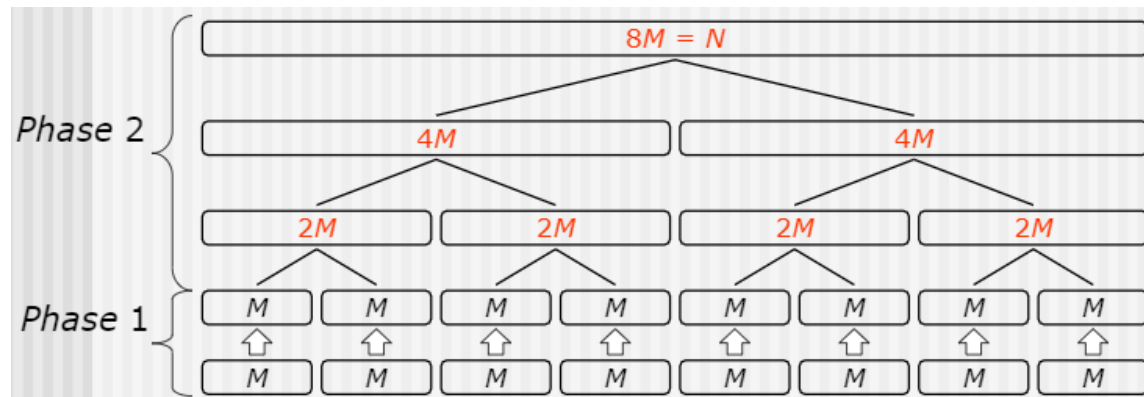
Recommended reading

- ▶ Reading this week
 - Chapters 7&8, textbook
- ▶ Next lecture:
 - Hashing, chapter 11



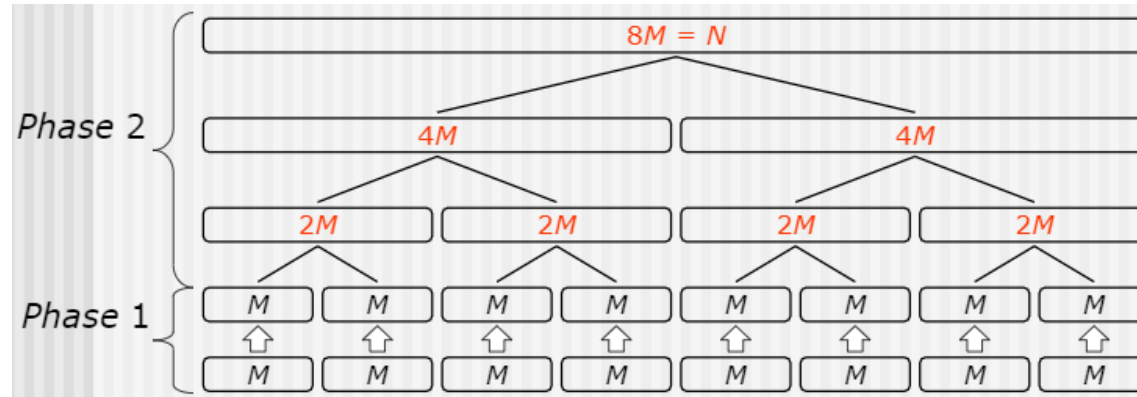
Complexity analysis

- ▶ Assumptions and notations
 - We use **disk** to do the external sorting
 - Disk page size: **B** data elements
 - Data file size: **N** data elements, $n=N/B$ disk pages
 - Available main memory: **M** elements, $m=M/B$ pages





Complexity analysis



► Analysis

- Phase 1:
 - Read file, write file: $2n = O(n)$ I/Os
- Phase 2:
 - One iteration: read file, write file, taking $2n = O(n)$ I/Os
 - Number of iterations: $\log_2 N/M = \log_2 n/m$
- Total running time: $O(n \log_2 n/m)$

► Can we do better?

- Phase 1: uses all available memory
- Phase 2: uses multi-way merge!