

IS 709/809: Computational Methods in IS Research

Algorithm Analysis (Sorting)

Nirmalya Roy

Department of Information Systems

University of Maryland Baltimore County

Sorting Problem

- Given an array $A[0 \dots N - 1]$, modify A such that $A[i] \leq A[i + 1]$ for $0 \leq i \leq N - 1$
- Internal vs. external sorting
 - Main memory and disk access
- Stable vs. unstable sorting
 - Equal elements retain original order
 - Keep elements with equal keys in the same relative order in the output as they were in the input
 - Input: $1, 5_x, 3, 5_y, 2, 4$
 - Output: $1, 2, 3, 4, 5_x, 5_y$

Sorting Problem

- In-place sorting
 - Transform input using a data structure with a constant amount of extra storage space; $O(1)$ extra memory
 - Constant additional storage for the auxiliary variables (i and temp)
 - Input is overwritten by the output as the algorithm executes
 - Example: Bubble sort, Selection sort, Insertion sort, Heap sort, Shell sort etc.
- Comparison sorting vs. non-comparison sorting
 - Besides assignment operator; “<” and “>” operators are allowed on the input data
 - Function template `sort` with comparator `cmp`
 - `void sort(Iterator begin, Iterator end, Comparator cmp)`

Sorting Algorithms

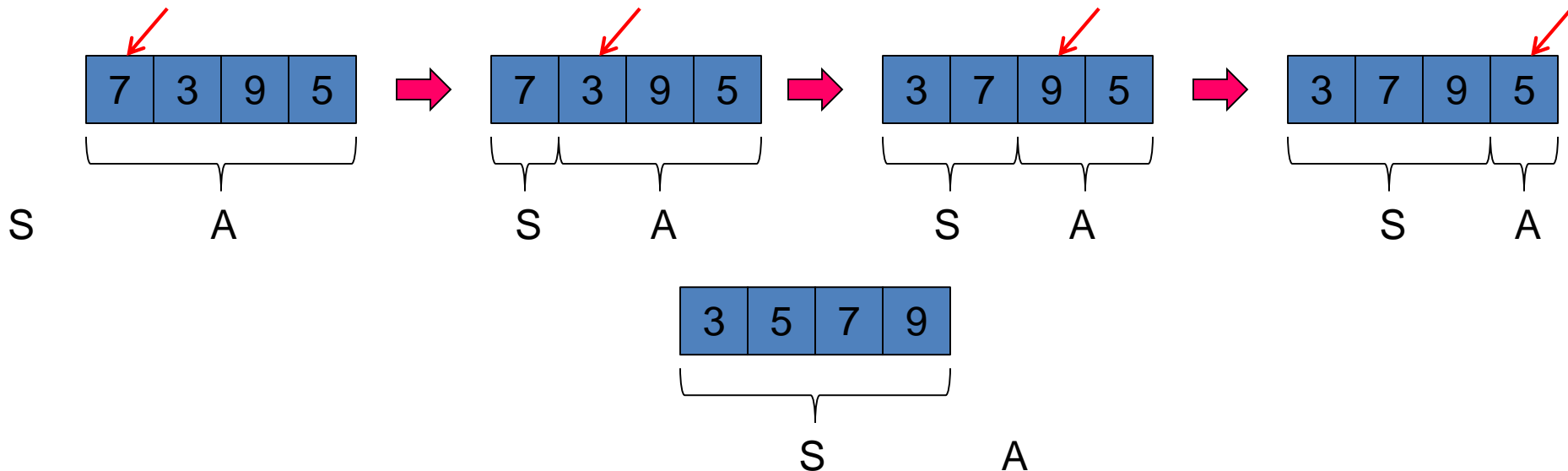
- Insertion sort
- Selection sort
- Shell sort
- Heap sort
- Merge sort
- Quick sort
- ...
- Simple data structure; focus on analysis

Insertion Sort

■ Algorithm:

- Start with empty list S and **unsorted** list A of N items
- For each item x in A
 - Insert x into S, in sorted order

■ Example:



Insertion Sort (cont'd)

- In-place
- Stable
- Best-case?
 - $O(N)$
- Worst-case?
 - $O(N^2)$
- Average-case?
 - $O(N^2)$

```
InsertionSort(A) {  
    for p = 1 to N - 1 {  
        tmp = A[p]  
        j = p  
        while (j > 0) and (tmp < A[j - 1]) {  
            A[j] = A[j - 1]  
            j = j - 1  
        }  
        A[j] = tmp  
    }  
}
```

- Consists of $N-1$ **passes**
- For pass $p = 1$ to $N-1$
 - Position 0 thru p are in sorted order
 - Move the element in position p left until its correct place; among first $p+1$ elements

Insertion Sort Example

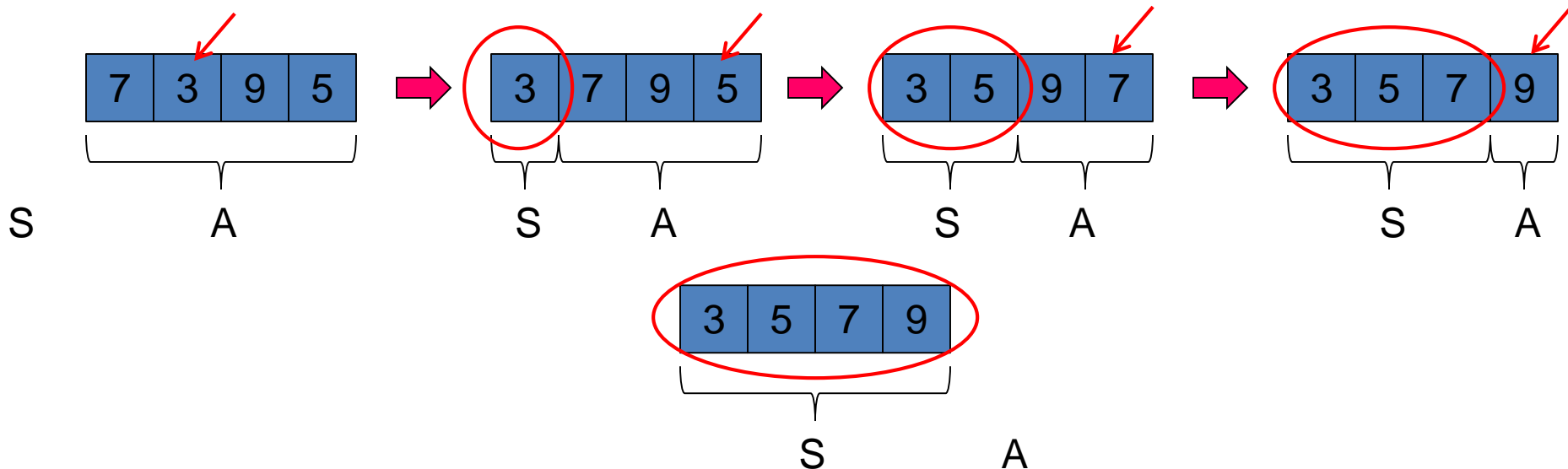
Original	34	8	64	51	32	21	Positions Moved
After $p = 1$	8	34	64	51	32	21	1
After $p = 2$	8	34	64	51	32	21	0
After $p = 3$	8	34	51	64	32	21	1
After $p = 4$	8	32	34	51	64	21	3
After $p = 5$	8	21	32	34	51	64	4

- Consists of $N-1$ **passes**
- For pass $p = 1$ to $N-1$
 - Position 0 thru p are in sorted order
 - Move the element in position p left until its correct place is found; among first $p+1$ elements

Selection Sort

■ Algorithm:

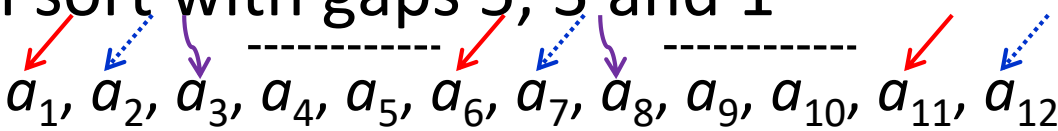
- Start with empty list S and unsorted list A of N items
- for ($i = 0$; $i < N$; $i++$)
 - $x \leftarrow$ item in A with smallest key
 - Remove x from A
 - Append x to end of S



Selection Sort (cont'd)

- In-place
- Unstable
- Best-case: $O(N^2)$
- Worst-case: $O(N^2)$
- Average-case: $O(N^2)$

Shell Sort

- Shell sort is a multi-pass algorithm
 - Each pass is an **insertion sort** of the sequences consisting of every **h -th** element for a fixed gap h , known as the **increment**
 - This is referred to as **h -sorting**
- Consider shell sort with gaps 5, 3 and 1
 - Input array: $a_1, a_2, a_3, a_4, a_5, a_6, a_7, a_8, a_9, a_{10}, a_{11}, a_{12}$ 
 - First pass, **5-sorting**, performs insertion sort on separate sub-arrays (a_1, a_6, a_{11}) , (a_2, a_7, a_{12}) , (a_3, a_8) , (a_4, a_9) , (a_5, a_{10})
 - Next pass, **3-sorting**, performs insertion sort on the sub-arrays (a_1, a_4, a_7, a_{10}) , (a_2, a_5, a_8, a_{11}) , (a_3, a_6, a_9, a_{12})
 - Last pass, **1-sorting**, is an ordinary insertion sort of the entire array (a_1, \dots, a_{12})

Shell Sort (cont'd)

13 14 94 33 82 25 59 94 65 23 45 27 73 25 39 10

- Works by comparing the elements that are distant
- The distance between comparisons decreases as algorithm runs until its last phase

13 14 94 33 82
25 59 94 65 23
45 27 73 25 39
10

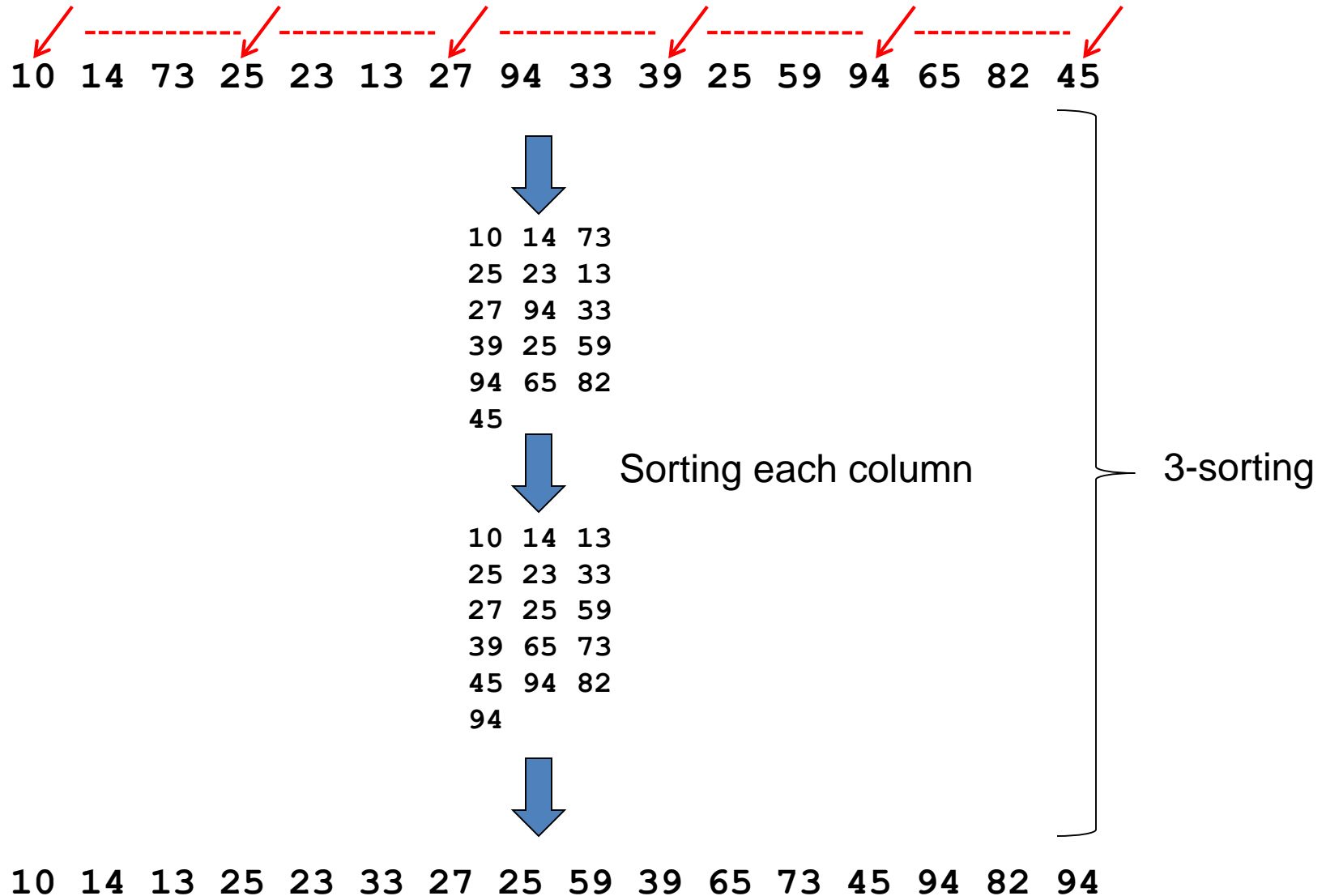
Sorting each column
(Insertion Sort)

10 14 73 25 23
13 27 94 33 39
25 59 94 65 82
45

10 14 73 25 23 13 27 94 33 39 25 59 94 65 82 45

5-sorting

Shell Sort (cont'd)



Shell Sort (cont'd)

10 14 13 25 23 33 27 25 59 39 65 73 45 94 82 94 **Unsorted A**



1-sorting/ Insertion Sort

10 13 14 23 25 25 27 33 39 45 59 65 73 82 94 94 **Sorted S**

■ Insertion Sort:

- Start with empty list S and unsorted list A of N items
- For each item x in A
 - Insert x into S, in sorted order

Shell Sort (cont'd)

- In-place
- Unstable
- Best-case
 - Sorted: $\Theta(N \log_2 N)$
- Worst-case
 - Shell's increments (by 2^k): $\Theta(N^2)$
 - Hibbard's increments (by $2^k - 1$): $\Theta(N^{3/2})$
- Average-case: $\Theta(N^{7/6})$
- Later sorts do not undo the work done in previous sorts
 - If an array is 5-sorted and then 3-sorted, the array is now not only 3-sorted, but both 5- and 3-sorted

```
ShellSort(A) {  
    gap = N  
    while (gap > 0) {  
        gap = gap / 2  
        B = <A[0], A[gap], A[2*gap], ...>  
        InsertionSort(B)  
    }  
}
```

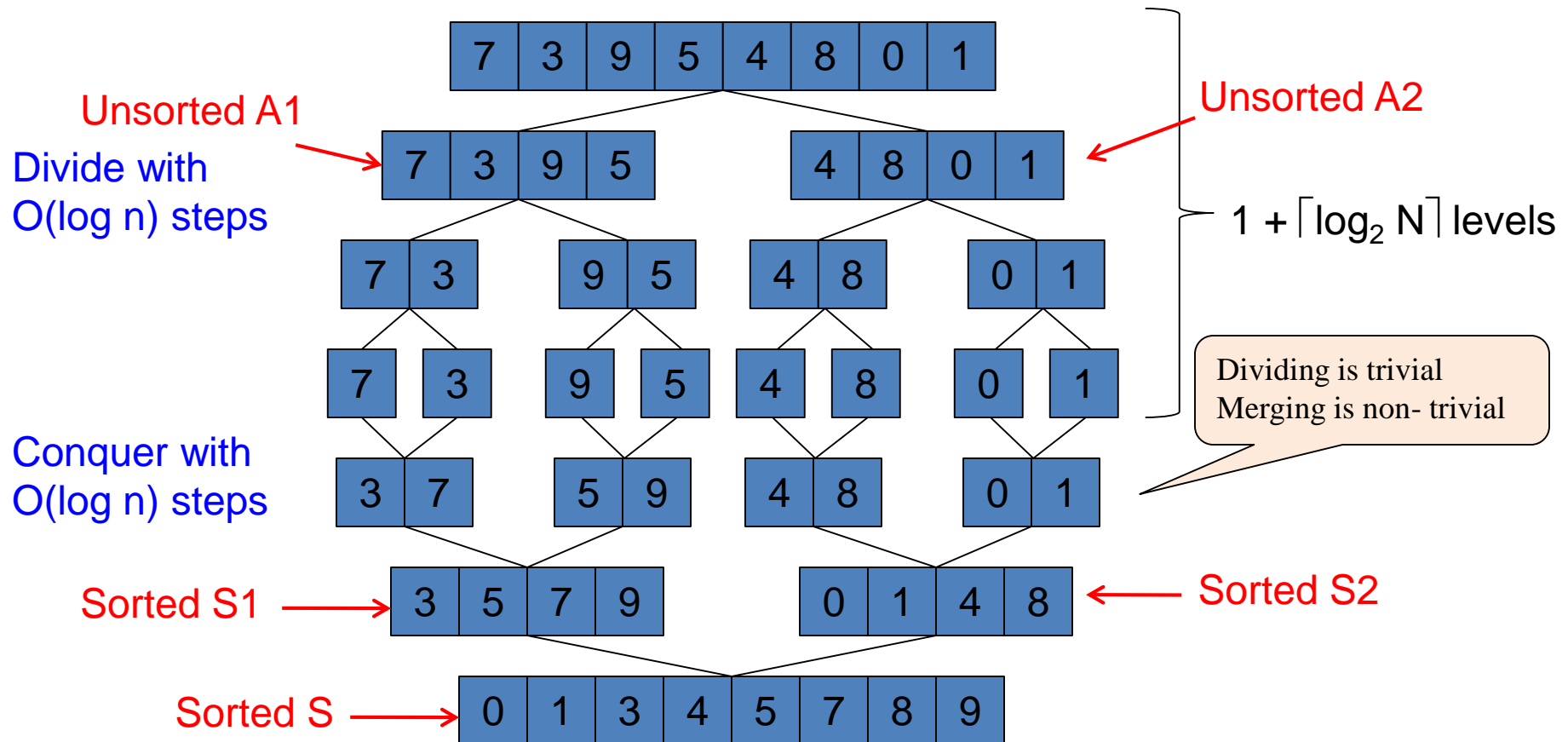
Merge Sort

- Idea: We can merge 2 sorted lists into 1 sorted list in linear time
- Let Q1 and Q2 be 2 sorted queues
- Let Q be empty queue
- Algorithm for merging Q1 and Q2 into Q:
 - While (neither Q1 nor Q2 is empty)
 - item1 = Q1.front()
 - item2 = Q2.front()
 - Move smaller of item1, item2 from present queue to end of Q
 - Concatenate remaining non-empty queue (Q1 or Q2) to end of Q

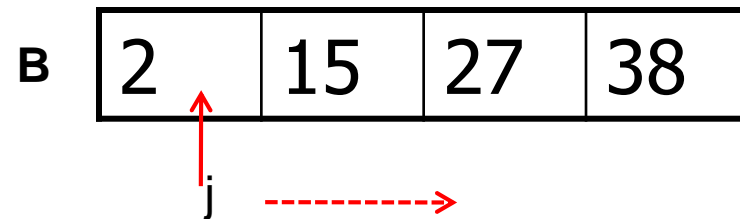
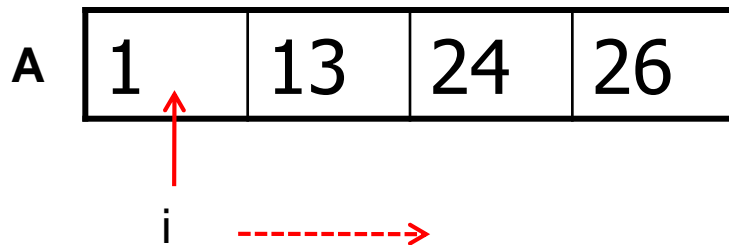
Merge Sort (cont'd)

- Recursive divide-and-conquer algorithm
- Algorithm:
 - Start with unsorted list A of N items
 - Break A into halves A1 and A2, having $\lceil N/2 \rceil$ and $\lfloor N/2 \rfloor$ items
 - Sort A1 recursively, yielding S1
 - Sort A2 recursively, yielding S2
 - Merge S1 and S2 into one sorted list S

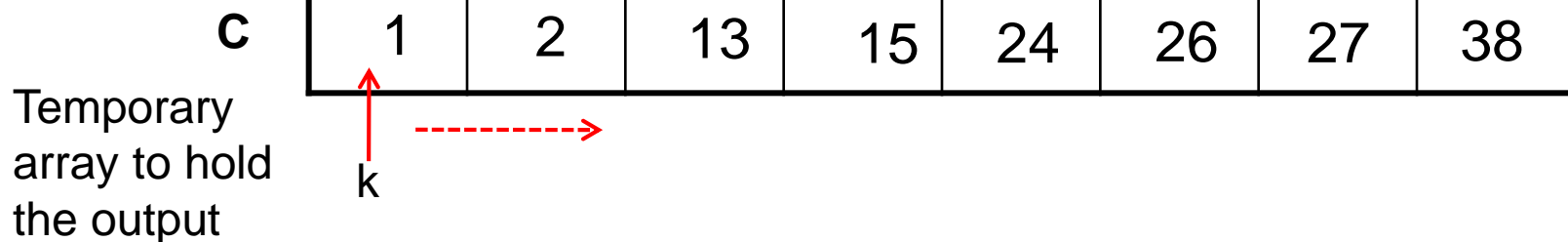
Merge Sort (cont'd)



Merging Two Sorted Arrays



- ↓
1. $C[k++] = \text{Populate } \min\{ A[i], B[j] \}$
 2. Advance the minimum contributing pointer



$\Theta(N)$ time

Merge Sort (cont'd)

- Not in-place
- Stable
- Analysis: All cases
 - $T(1) = \Theta(1)$
 - $T(N) = 2T(N/2) + \Theta(N)$
 - $T(N) = \Theta(N \log_2 N)$
 - See whiteboard

```
MergeSort(A)
    MergeSort2(A, 0, N - 1)

MergeSort2(A, i, j)
    if (i < j)
        k = (i + j) / 2
        MergeSort2(A, i, k)
        MergeSort2(A, k + 1, j)
        Merge(A, i, k + 1, j)

Merge(A, i, k, j)
    Create auxiliary array B
    Copy elements of sorted A[i...k] and
    sorted A[k+1...j] into B (in order)
    A = B
```

Quick Sort

- Like merge sort, quick sort is a divide-and-conquer algorithm, except
 - Don't divide the array in half
 - Partition the array based on elements being less than or greater than some element of the array (the pivot)
- In-place, unstable
- Worst-case running time: $O(N^2)$
- Average-case running time: $O(N \log_2 N)$
- Fastest generic sorting algorithm in practice

Quick Sort (cont'd)

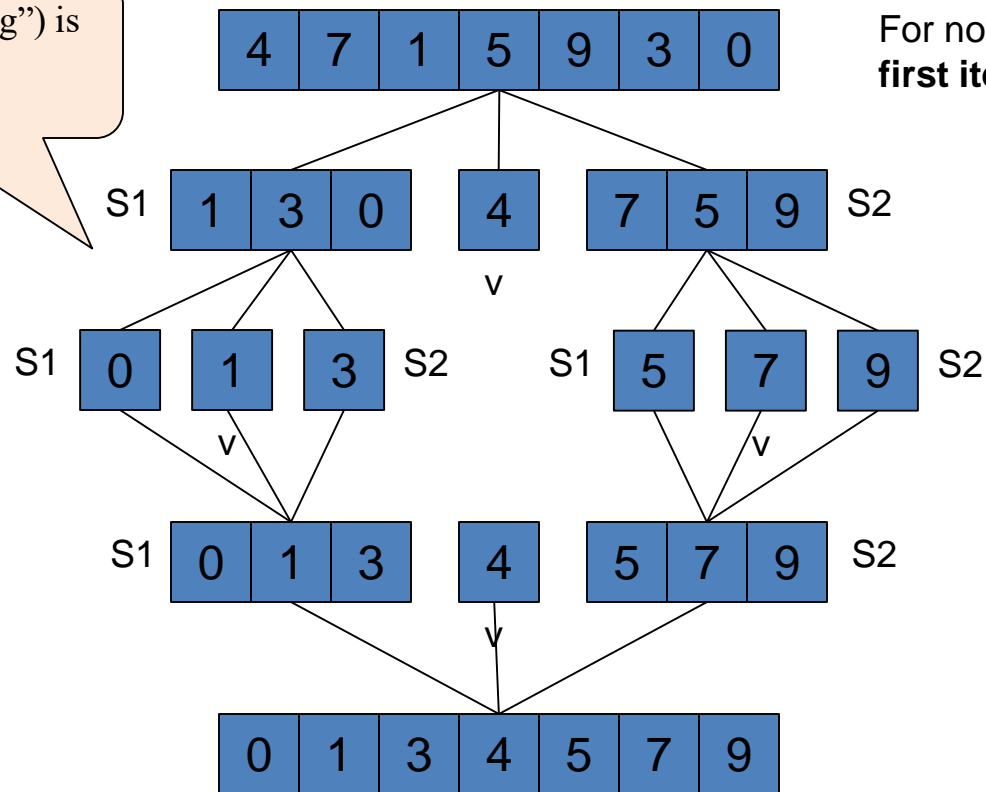
■ Algorithm:

- Start with list A of N items
- Choose pivot item v from A —————→

How to choose pivot?
- Partition A into 2 unsorted lists A1 and A2
 - A1: All keys smaller than v's key
 - A2: All keys larger than v's key
 - Items with same key as v can go into either list
 - The pivot v does not go into either list
- Sort A1 recursively, yielding sorted list S1
- Sort A2 recursively, yielding sorted list S2
- Concatenate S1, v, and S2, yielding sorted list S

Quick Sort (cont'd)

Dividing (“Partitioning”) is
non-trivial
Merging is trivial



For now, let the pivot v be the
first item in the list.

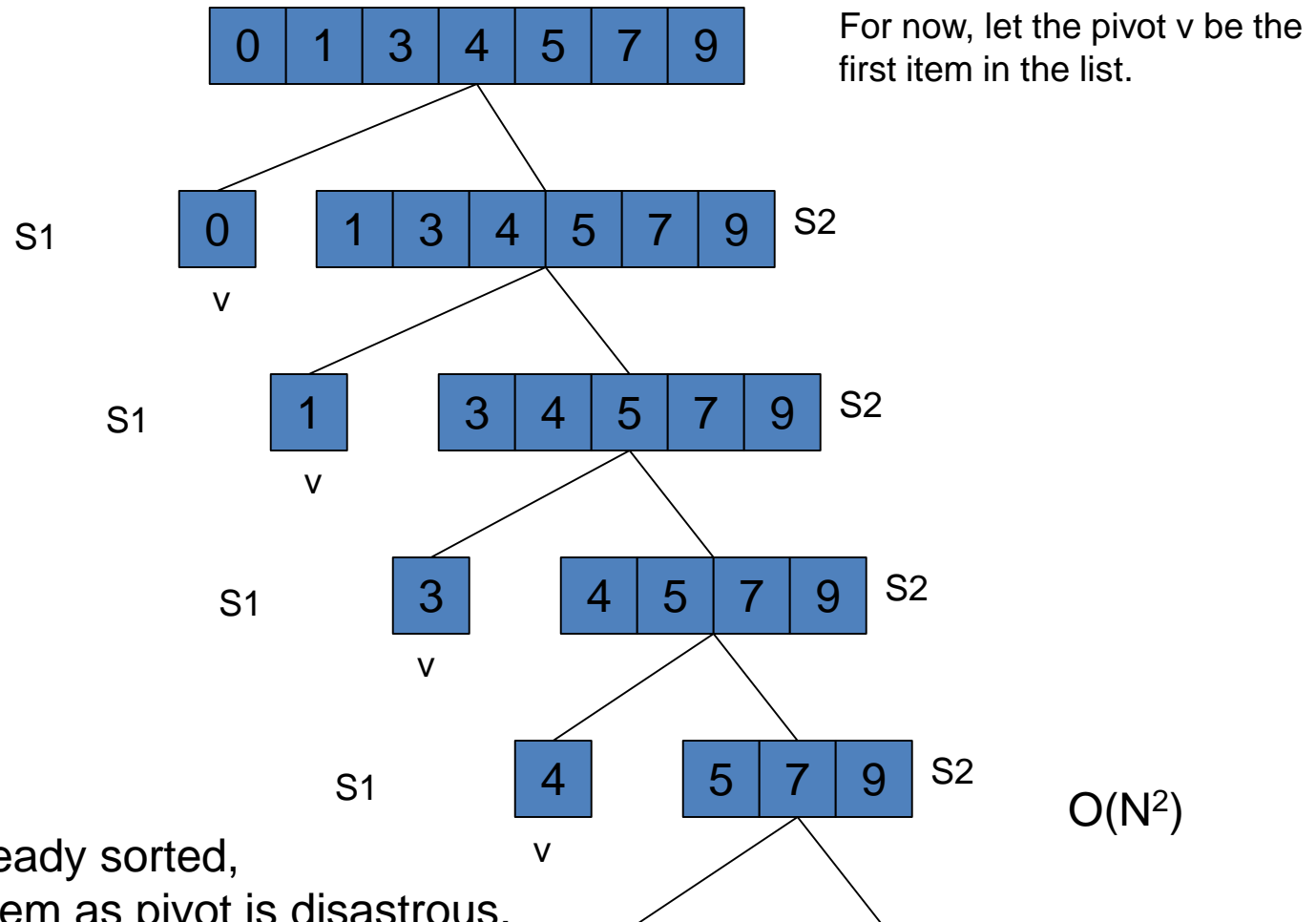
$O(N \log_2 N)$

Quick Sort Algorithm

- quicksort (array: S)
 1. If size of S is 0 or 1, return
 2. Pivot = Pick an element v in S
 3. Partition $S - \{v\}$ into two disjoint groups
$$S1 = \{x \in (S - \{v\}) \mid x < v\}$$
$$S2 = \{x \in (S - \{v\}) \mid x > v\}$$
 4. Return {quicksort(S1), followed by v, followed by quicksort(S2)}

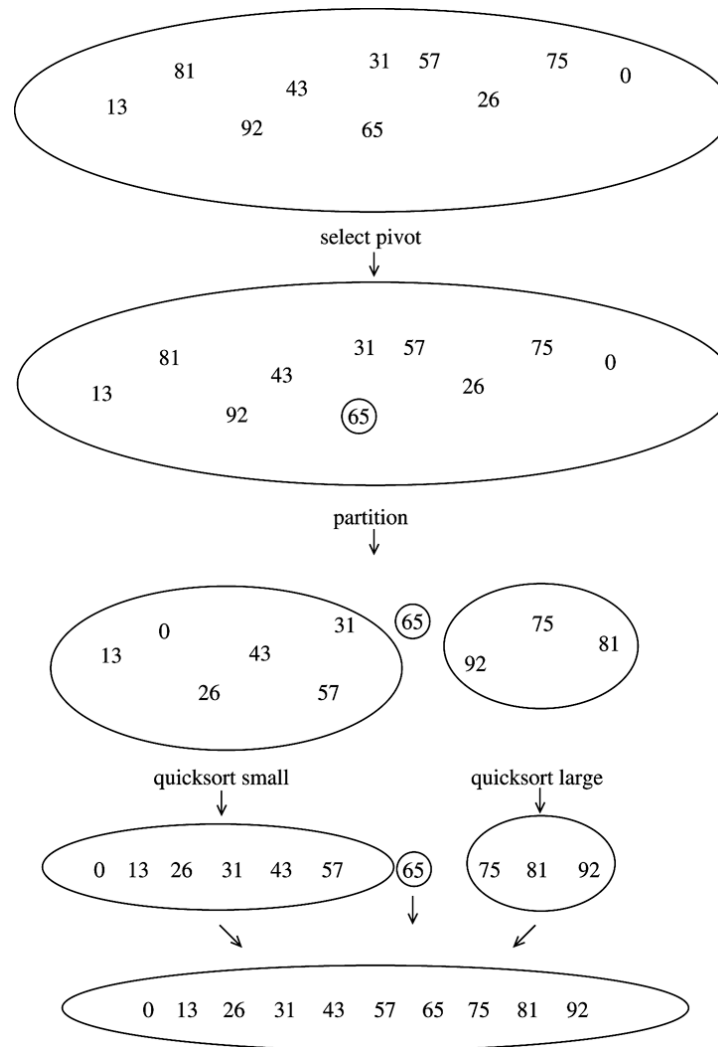
Quick Sort (cont'd)

What if the list is already sorted?



Quick Sort (cont'd)

We need a better
pivot-choosing strategy.



Quick Sort (cont'd)

- Merge sort always divides array in half
 - Quick sort might divide array into sub problems of size 1 and $N - 1$
 - When?
 - Leading to $O(N^2)$ performance
 - Need to choose pivot wisely (but efficiently)
- Merge sort requires temporary array for merge step
 - Quick sort can partition the array in place
 - This more than makes up for bad pivot choices

Quick Sort (cont'd)

- Choosing the pivot
 - Choosing the first element
 - What if array already or nearly sorted?
 - Good for random array
 - Choose random pivot
 - Good in practice if truly random
 - Still possible to get some bad choices
 - Requires execution of random number generator
 - On average, generates $\frac{1}{4}$, $\frac{3}{4}$ split

Quick Sort (cont'd)

- Choosing the pivot
 - Best choice of pivot?
 - Median of array
 - Median is expensive to calculate
 - Estimate median as the median of three elements (called the **median-of-three strategy**)
 - Choose first, middle, and last elements
 - E.g., <8, 1, 4, 9, 6, 3, 5, 2, 7, 0>
 - Has been shown to reduce running time (comparisons) by 14%

Quick Sort (cont'd)

■ Partitioning strategy

- Partitioning is conceptually straightforward, but easy to do inefficiently
- Good strategy
 - Swap pivot with last element $A[\text{right}]$
 - Set $i = \text{left}$
 - Set $j = (\text{right} - 1)$
 - While $(i < j)$
 - Increment i until $A[i] > \text{pivot}$
 - Decrement j until $A[j] < \text{pivot}$
 - If $(i < j)$ then swap $A[i]$ and $A[j]$
 - Swap pivot and $A[i]$

Partitioning Example

8 1 4 9 6 3 5 2 7 0

Initial array

8 1 4 9 0 3 5 2 7 6

Swap pivot; initialize i and j

i

j

8 1 4 9 0 3 5 2 7 6

Position i and j

i

j

2 1 4 9 0 3 5 8 7 6

After first swap

i

j

2 1 4 9 0 3 5 8 7 6

Before second swap

i

j

2 1 4 5 0 3 9 8 7 6

After second swap

i

j

2 1 4 5 0 3 9 8 7 6

Before third swap

j i

2 1 4 5 0 3

6

8 7 9

After swap with pivot

i

Quick Sort (cont'd)

- Partitioning strategy
 - How to handle duplicates?
 - Consider the case where all elements are equal.
 - Current approach: Skip over elements equal to pivot
 - No swaps (good)
 - But then $i = (\text{right} - 1)$ and array partitioned into $N - 1$ and 1 elements
 - Worst-case performance: $O(N^2)$

Quick Sort (cont'd)

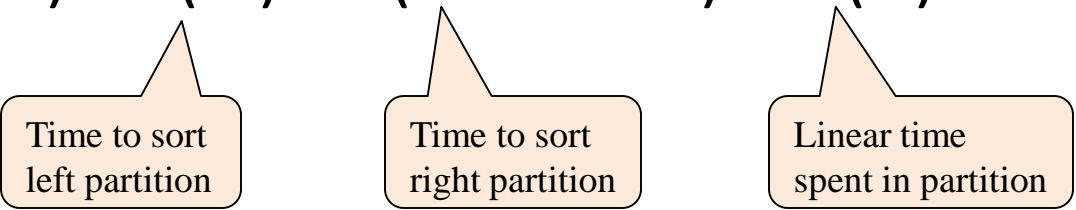
- Partitioning strategy
 - How to handle duplicates?
 - Alternate approach
 - Don't skip elements equal to pivot
 - Increment i **while** $A[i] < \text{pivot}$
 - Decrement j **while** $A[j] > \text{pivot}$
 - Adds some unnecessary swaps
 - But results in perfect partitioning for the array of identical elements
 - Unlikely for input array, but more likely for recursive calls to quick sort

Which Sort to Use?

- When array A is small, generating lots of recursive calls on small sub-arrays is expensive
- General strategy
 - When $N < \text{threshold}$, use a sort more efficient for small arrays (e.g. insertion sort)
 - Good thresholds range from 5 to 20
 - Also avoids issue with finding median-of-three pivot for array of size 2 or less
 - Has been shown to reduce running time by 15%

Analysis of Quick Sort

- Let m be the number of elements sent to the left partition
- Compute running time $T(N)$ for array of size N
- $T(0) = T(1) = O(1)$
- $T(N) = T(m) + T(N - m - 1) + O(N)$



Time to sort
left partition

Time to sort
right partition

Linear time
spent in partition

- Pivot selection takes constant time

Analysis of Quick Sort (cont'd)

- Recurrence formula:

- $T(N) = T(m) + T(N - m - 1) + O(N)$

- Worst-case analysis

- Pivot is the smallest element ($m = 0$)

$$T(N) = T(0) + T(N - 1) + O(N)$$

$$T(N) = O(1) + T(N - 1) + O(N)$$

$$T(N) = T(N - 1) + O(N); \text{ since } T(N - 1) = T(N - 2) + O(N - 1);$$

$$T(N) = T(N - 2) + O(N - 1) + O(N)$$

$$T(N) = T(N - 3) + O(N - 2) + O(N - 1) + O(N)$$

...

$$T(N) = \sum_{i=1}^N O(i) = O(N^2)$$

Analysis of Quick Sort (cont'd)

- Recurrence formula:

- $T(N) = T(m) + T(N - m - 1) + O(N)$

- Best-case analysis

- Pivot is in the middle ($m = N / 2$)

$$T(N) = T(N / 2) + T(N / 2) + O(N)$$

$$T(N) = 2T(N / 2) + O(N)$$

$$T(N) = O(N \log N)$$

- Average-case analysis

- Assuming each partition equally likely

- $T(N) = O(N \log N)$

Comparison of Sorting Algorithms

Sort	Worst Case	Average Case	Best Case	Comments
Bubble Sort	$\Theta(N^2)$	$\Theta(N^2)$	$\Theta(N)$	Best Case is linear
Selection Sort	$\Theta(N^2)$	$\Theta(N^2)$	$\Theta(N^2)$	Best Case is quadratic
InsertionSort	$\Theta(N^2)$	$\Theta(N^2)$	$\Theta(N)$	Fast for small N
ShellSort	$\Theta(N^{3/2})$	$\Theta(N^{7/6})$?	$\Theta(N \log N)$	Increment sequence?
HeapSort	$\Theta(N \log N)$	$\Theta(N \log N)$	$\Theta(N \log N)$	Large constants
MergeSort	$\Theta(N \log N)$	$\Theta(N \log N)$	$\Theta(N \log N)$	Requires memory
QuickSort	$\Theta(N^2)$	$\Theta(N \log N)$	$\Theta(N \log N)$	Small constants

Comparison of Sorting Algorithms (cont'd)

N	Insertion Sort $O(N^2)$	Shellsort $O(N^{7/6})(?)$	Heapsort $O(N \log N)$	Quicksort $O(N \log N)$	Quicksort (opt.) $O(N \log N)$
10	0.000001	0.000002	0.000003	0.000002	0.000002
100	0.000106	0.000039	0.000052	0.000025	0.000023
1000	0.011240	0.000678	0.000750	0.000365	0.000316
10000	1.047	0.009782	0.010215	0.004612	0.004129
100000	110.492	0.13438	0.139542	0.058481	0.052790
1000000	NA	1.6777	1.7967	0.6842	0.6154

All times are in seconds

~3 hours

Good sorting applets:

- <http://www.sorting-algorithms.com>
- <http://math.hws.edu/TMCM/java/xSortLab/>

Lower Bound on Sorting

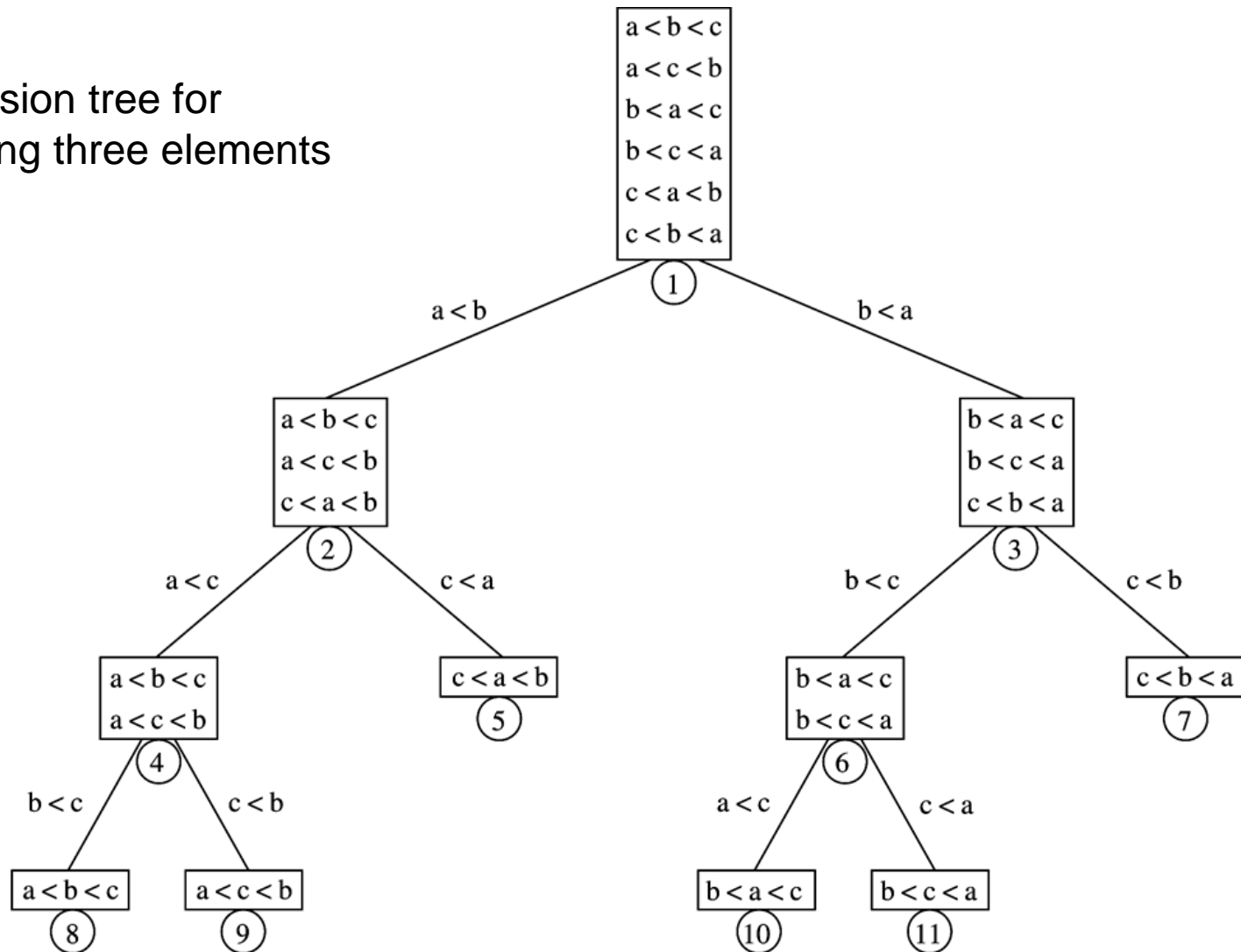
- Best worst-case sorting algorithm (so far) is $O(N \log N)$
- Can we do better?
- Can we prove a lower bound on the sorting problem?
- Preview
 - For comparison-based sorting, we can't do better
 - We can show lower bound of $\Omega(N \log N)$

Decision Trees

- A decision tree is a binary tree
 - Each node represents a set of possible orderings of the array elements
 - Each branch represents an outcome of a particular comparison
- Each leaf of the decision tree represents a particular ordering of the original array elements

Decision Trees (cont'd)

Decision tree for
sorting three elements



Decision Trees (cont'd)

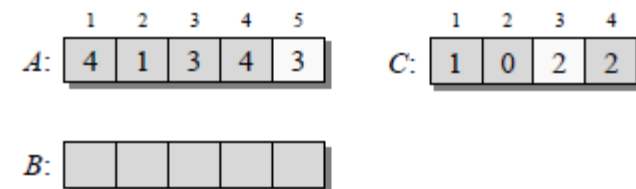
- The logic of every sorting algorithm that uses comparisons can be represented by a decision tree
- In the worst case, the number of comparisons used by the algorithm equals the depth of the deepest leaf
- In the average case, the number of comparisons is the average of the depth of all leaves
- There are $N!$ different orderings of N elements

Lower Bound for Comparison-based Sorting

- Lemma 7.1: A binary tree of depth d has at most 2^d leaves
- Lemma 7.2: A binary tree with L leaves must have depth at least $\lceil \log L \rceil$
- Theorem 7.6: Any comparison-based sorting requires at least $\lceil \log (N!) \rceil$ comparison in the worst case
- Theorem 7.7: Any comparison-based sorting requires $\Omega(N \log N)$ comparisons

Linear Sorting

- Some constraints on input array allow faster than $\Theta(N \log N)$ sorting (no comparisons)
- Counting Sort¹
 - Given array A of N integer elements, each less than M
 - Create array C of size M, where C[i] is the number of i's in A
 - Use C to place elements into new sorted array B
 - Running time $\Theta(N + M) = \Theta(N)$ if $M = \Theta(N)$

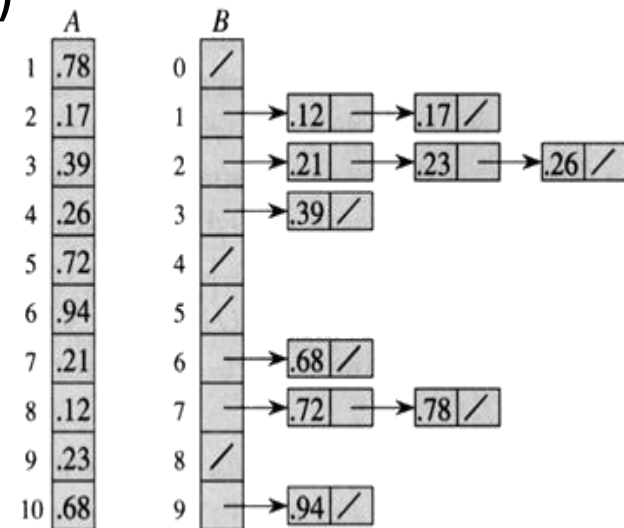


¹Weiss incorrectly calls this Bucket Sort.

Linear Sorting (cont'd)

■ Bucket Sort

- Assume N elements of A uniformly distributed over the range $[0, 1)$
- Create N equal-size buckets over $[0, 1)$
- Add each element of A into appropriate bucket
- Sort each bucket (e.g. with insertion sort)
- Return concatenation of buckets
- Average-case running time $\Theta(N)$
 - Assumes each bucket will contain $\Theta(1)$ elements



External Sorting

- What if the number of elements N we wish to sort do not fit in main memory?
- Obviously, our existing sorting algorithms are inefficient
 - Each comparison potentially requires a disk access
- Once again, we want to minimize disk accesses

External Merge Sort

- N = number of elements in array A to be sorted
- M = number of elements that fit in main memory
- $K = \lceil N / M \rceil$
- Approach
 - Read in M amount of A , sort it using quick sort, and write it back to disks: $O(M \log M)$
 - Repeat above K times until all of A processed
 - Create K input buffers and 1 output buffer, each of size $M / (K + 1)$
 - Perform a K -way merge: $O(N)$
 - Update input buffers one disk-page at a time
 - Write output buffer one disk-page at a time

Multiway Merge (3-way) Example

Ta1 81 94 11 96 12 35 17 99 28 58 41 75 15 Read from input tape

Ta1

Ta2

Ta3

Sort in internal memory that can hold M records

Tb1 11 81 94 41 58 75 Write each of these *runs* into output tape

Tb2 12 35 96 15

Tb3 17 28 99

Ta1 11 12 17 28 35 81 94 96 99 Take the first *run* from each tape and merge

Ta2 15 41 58 75

Ta3

Tb1

Tb2

Tb3

Ta1

Ta2

Ta3

Tb1 11 12 15 17 28 35 41 58 75 81 94 96 99

Tb2

Tb3

Sorting: Summary

- Need for sorting is ubiquitous in software
- Optimizing the sorting algorithm to the domain is essential
- Good general-purpose algorithms available
 - Quick sort
- Optimizations continue...
 - Sort benchmark
 - <http://sortbenchmark.org/>