

Sorting Techniques

Design and Analysis of Algorithms Lecture Set 2

Outline

- The problem
- Importance and Application
- Bubble Sort
- Insertion Sort
- Selection Sort
- Heap Sort

The problem

- Given a set of n numbers that may be in any order, the goal is to output the numbers in sorted order
 - A set of elements S are sorted in ascending order when $S_i < S_{i+1}$ for all $1 \leq i < n$
 - A set of elements S are sorted in descending order when $S_i > S_{i+1}$ for all $1 \leq i < n$
- Can sort just the key or entire records

Importance of Sorting

- Basic block around which many other algorithms are built on
- Interesting ideas in design of algorithms appear in the context of sorting
 - Divide and conquer, data structures, randomized algorithms
- Remains the most ubiquitous combinatorial algorithmic problem in practice
- Loads of study on this problem

Applications of Sorting

- Searching
 - Whether linear or binary, it is easier to search for an element if the list is sorted
- Closest Pair
 - Given a set of n numbers, find the pair of numbers that have the smallest difference between them
 - Closest pair lie next to each other somewhere in the sorted order

Applications of Sorting

- Finding duplicates among n elements in a list
 - In sorted list, it is easy to find this by scanning adjacent elements
- Frequency Distribution
 - Find the number of occurrences of different elements in a list
 - Identical items together in a sorted list
 - Just scan list once
- Selection
 - Find the k th largest element in a list

Sorting Algorithms

- Comparison Sort
- Bucket Sort
- Counting Sort
- Radix Sort
- Heap Sort

Bubble Sort

- A basic sorting algorithm where corresponding elements are checked and swapped
 - The smallest elements bubble up the way



http://t2.gstatic.com/images?q=tbn:ANd9GcRnvLV-rVerCKzEZ7sN7WwBIUL09M_FqIkgX52w5mSLElDgtFVRX-FgHtSsWg

Pseudocode

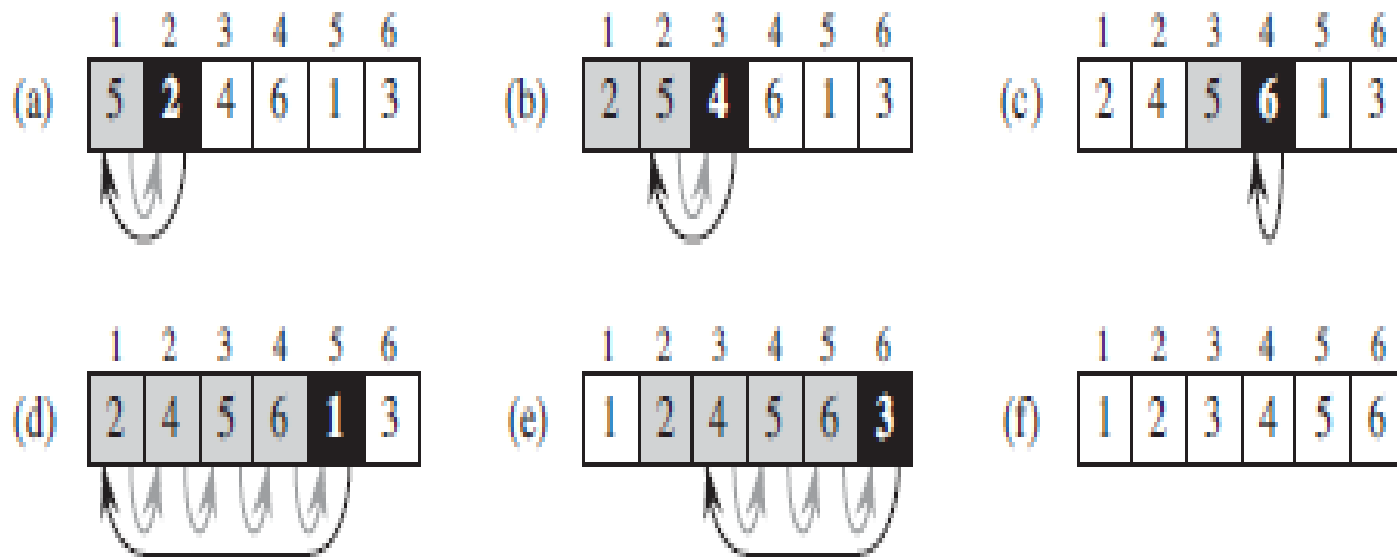
- BUBBLESORT(A)
 1. **for** $i = 1$ **to** $A.length - 1$
 2. **for** $j = A.length$ **downto** $i - 1$
 3. **if** $A[j] < A[j-1]$
 4. exchange $A[j]$ with $A[j - 1]$

Bubble Sort: Analysis

- Best Case
 - $O(n)$
 - List already sorted, hence zero swaps after first iteration
- Worst case
 - $O(n^2)$
- On average
 - $(n-1)*n/2$ comparisons
 - $O(n^2)$

Insertion Sort

- At each iteration an element is removed from the list and inserted into the right place
 - At any iteration the first i items are in place



Src: CLR – Ch2- Pg22

Insertion Sort: Pseudocode

- INSERTION-SORT(A)
 1. **for** $j = 2$ **to** $A.length$
 2. $key = A[j]$ // Insert $A[j]$ into the sorted sequence $A[1 : j-1]$
 3. $i = j-1$
 4. **while** $i > 0$ **and** $A[i] > key$
 5. $A[i+1] = A[i]$
 6. $i = i-1$
 7. $A[i+1] = key$

Src: CLR – Ch2

Insertion Sort : Analysis

- At each iteration
 - Element compared and/or swapped with atmost i elements
 - i varies from 1 to n
- n such elements inserted
- Average case and worst case- $O(n^2)$
 - Worst case when list sorted in reverse order
- Best Case – $O(n)$
 - When list is already sorted
 - No swaps needed

Selection Sort

- At each iteration the minimum element is chosen and inserted in the top of the list

S E L E C T I O N S O R T
C E L E S T I O N S O R T
C E L E S T I O N S O R T
C E E L S T I O N S O R T
C E E I S T L O N S O R T
C E E I L T S O N S O R T
C E E I L N S O T S O R T
C E E I L N O S T S O R T
C E E I L N O O T S S R T
C E E I L N O O R S S T T
C E E I L N O O R S S T T
C E E I L N O O R S S T T
C E E I L N O O R S S T T
C E E I L N O O R S S T T

Src: Steven S. Skiena, “The Algorithm Design Manual”, Second Edition

Selection Sort: Pseudocode

- ```
selection_sort(int s[], int n){
 int i,j; /* counters */
 for (i=0; i<n; i++) {
 min=i; /* index of minimum */
 for (j=i+1; j<n; j++)
 if (s[j] < s[min]) min=j;
 swap(s[i],s[min]);
 }
}
```

Src: Steven S. Skiena, “The Algorithm Design Manual”, Second Edition

# Analysis of Selection Sort

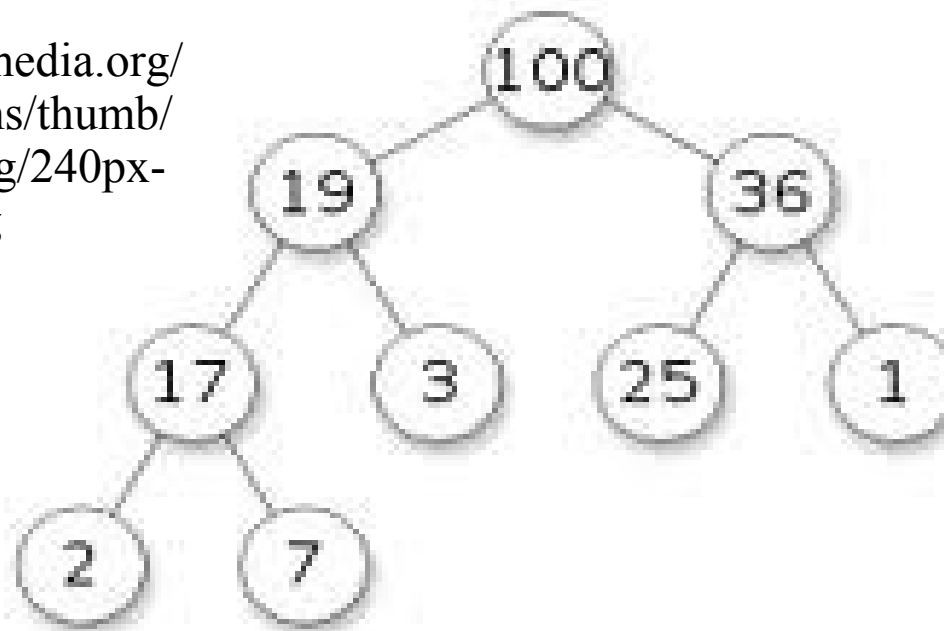
- Complexity depends on cost of finding minimum element in remaining list
- Best Case –  $O(n^2)$ 
  - Almost sorted still requires cost of finding min
- Average Case –  $O(n^2)$ 
  - Cost of finding minimum element is  $i$  per iteration  
 $= n(n-1)/2$
- Worst Case –  $O(n^2)$ 
  - When list is in reverse sorted order
  - The minimum is always at the end



# Review : Heaps

- Priority queue
  - Elements in sorted order
- Stores elements in a binary tree
  - insertions and deletions logarithmic time

<http://upload.wikimedia.org/wikipedia/commons/thumb/3/38/Max-Heap.svg/240px-Max-Heap.svg.png>



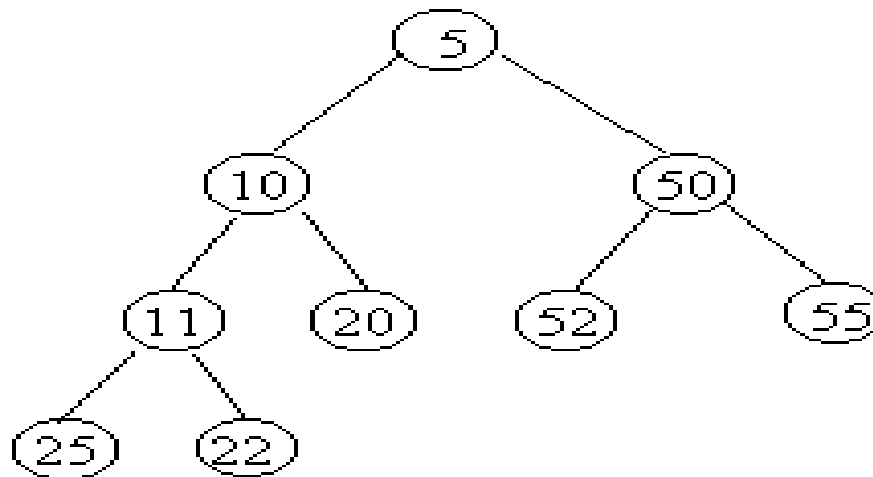
# Heap: Properties

- Heap-Order Property
  - For every node  $v$  other than the root, the key stored at  $v$  is greater than or equal to the key stored at  $v$ 's parent
- Complete Binary tree
  - A binary tree with height  $h$  is complete if the levels  $0, 1, 2, \dots, h-1$  have the maximum number of nodes possible and
  - All internal nodes are to the left of the external nodes
  - Helps keep the height of the heap small



# Heap Implementation

- Implemented using vector representation
- The last node is the rightmost node in the last level



|   |    |    |    |    |    |    |    |    |
|---|----|----|----|----|----|----|----|----|
| 5 | 10 | 50 | 11 | 20 | 52 | 55 | 25 | 22 |
| 0 | 1  | 2  | 3  | 4  | 5  | 6  | 7  | 8  |

# Heap Sort

- A priority queue based sort
- Step1: Create the heap
  - Inserting elements takes  $O(\log k)$  where  $k$  is the number of elements in the heap at that time
  - Using bottom-up approach cost is  $O(n)$
- Step 2: Sorting
  - Remove the minimum element in each iteration and store in external array
  - Complexity:  $O(\log k)$  where  $k$  is the number of elements in the heap at that time
- Total Cost:  $O(n \log n)$



# In-Place Heap Sort

- Uses the vector representation
  - Use the left portion of the vector  $S$  to store elements in heap upto  $i-1$  rank
    - These elements are sorted
  - Right portion of  $S$  to store the other elements in sequence
- Start with empty heap and move the boundary between heap and sequence from left to right
  - In step  $i$ , expand heap by adding element at  $i-1$
- Start with empty sequence and move the boundary
  - In step  $i$ , remove maximum from heap and store at rank  $n-i$
- We rearrange instead of using extra memory

# Heap-Sort

BUILD-MAX-HEAP(A)

for  $i = A.length$  downto 2

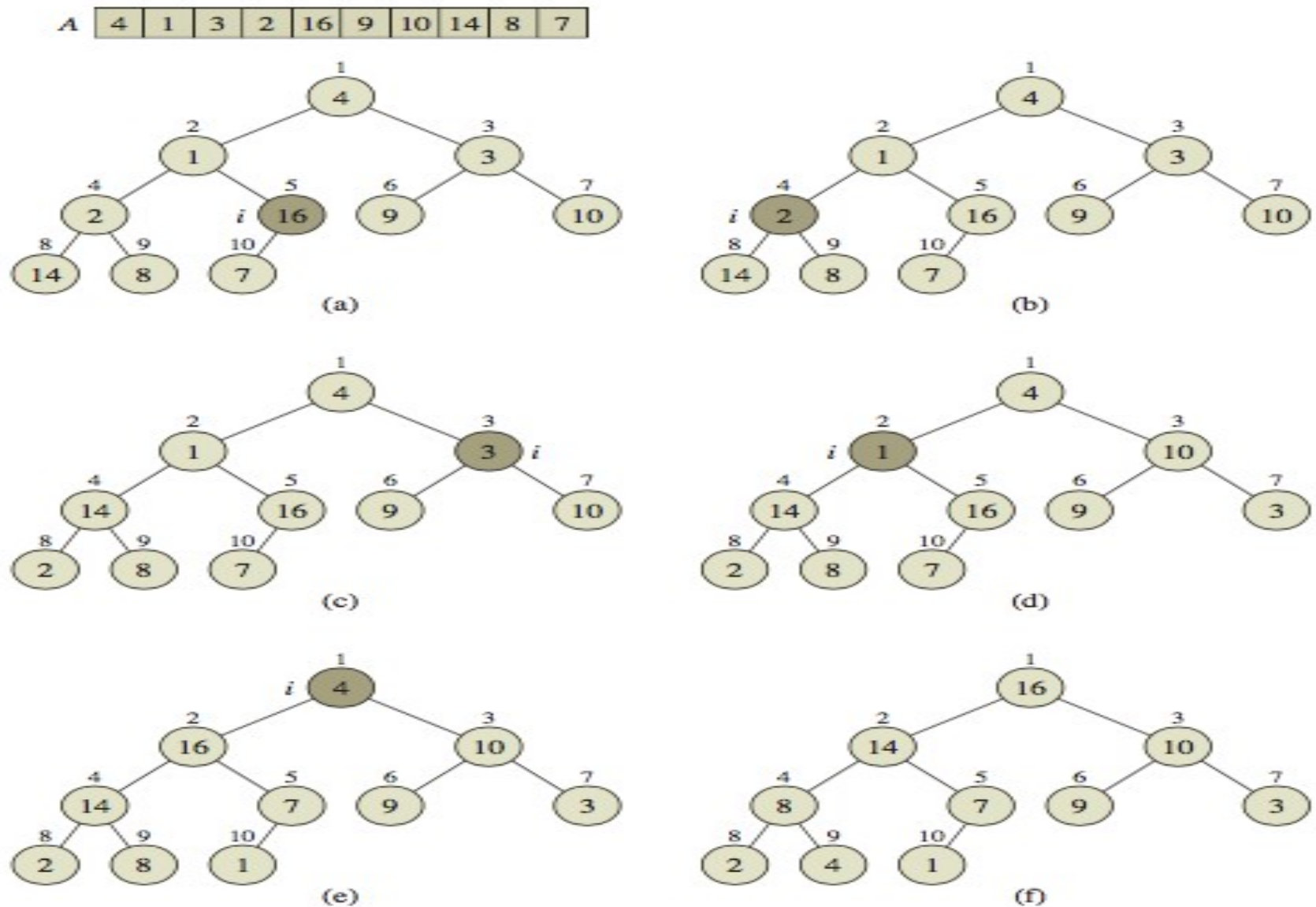
    exchange  $A[1]$  with  $A[i]$

$A[\text{heap-size}] = A[\text{heap-size}-1]$

    MAX-HEAPIFY(A, 1)



# In-place Heap Sort



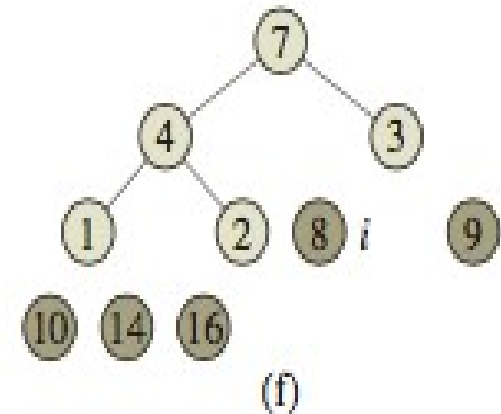
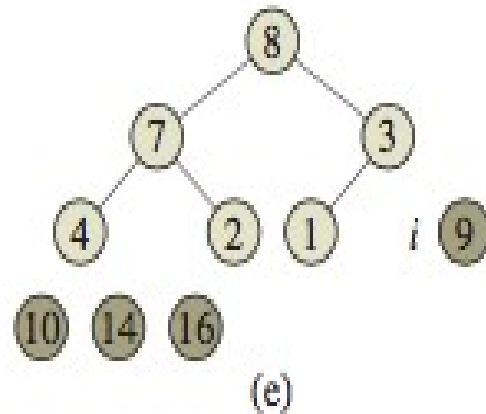
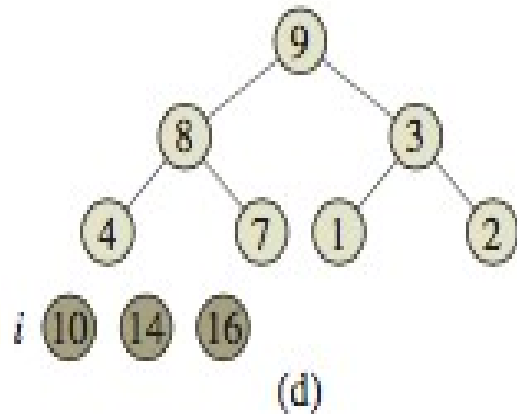
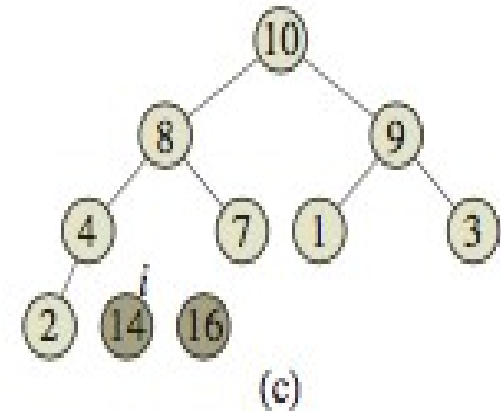
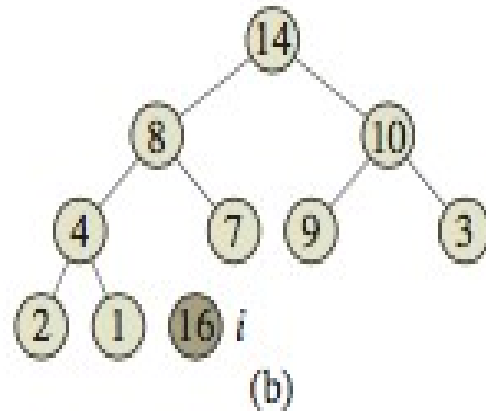
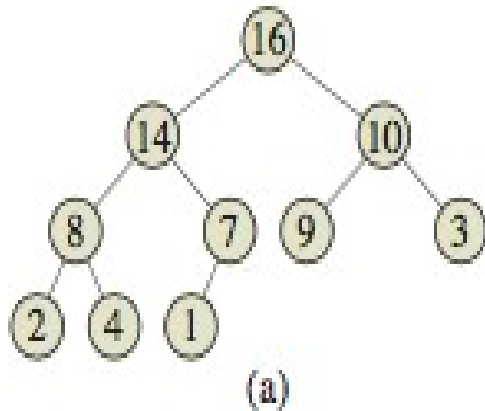
Src: Introduction to Algorithms: CLRS

CSE211: Design and Algorithms  
of Algorithms

Amrita School of Engineering  
Amrita Vswaha Vidyapeetham

Vidhya Balasubramanian

# In-place Heap Sort



Src: Introduction to Algorithms: CLRS

CSE211: Design and Algorithms  
of Algorithms

Amrita School of Engineering  
Amrita Vswaha Vidyapeetham

Vidhya Balasubramanian



# Merge-Sort

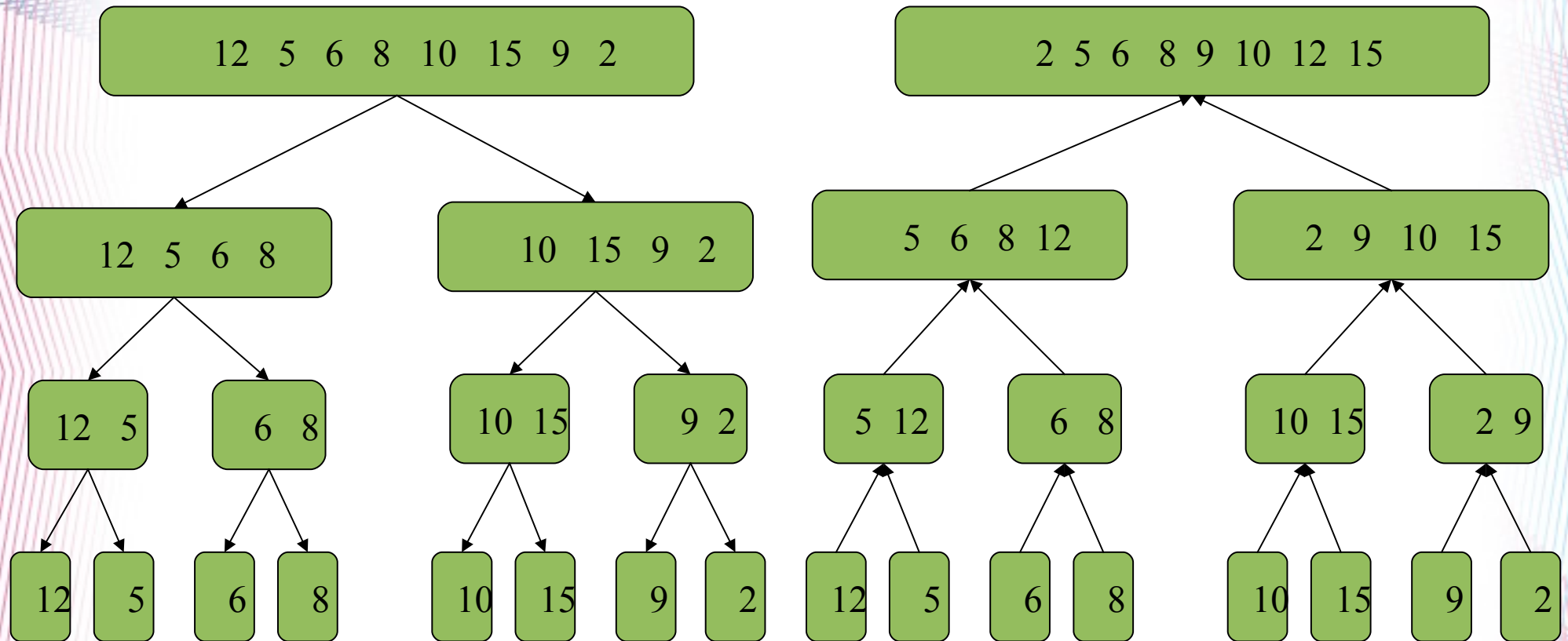
- Uses divide and conquer strategy to sort a set of numbers
- Divide:
  - If  $S$  has zero or one element, return  $S$
  - Divide  $S$  into two sequences  $S_1$  and  $S_2$  each containing half of the elements of  $S$
- Recur
  - Recursively apply merge sort to  $S_1$  and  $S_2$
- Conquer
  - Merge  $S_1$  and  $S_2$  into a sorted sequence

# Merging two sorted sequences

- Iteratively remove smallest element from one of the two sequences  $S_1$  and  $S_2$  and add it to end of output sequence  $S$



# Merge Sort

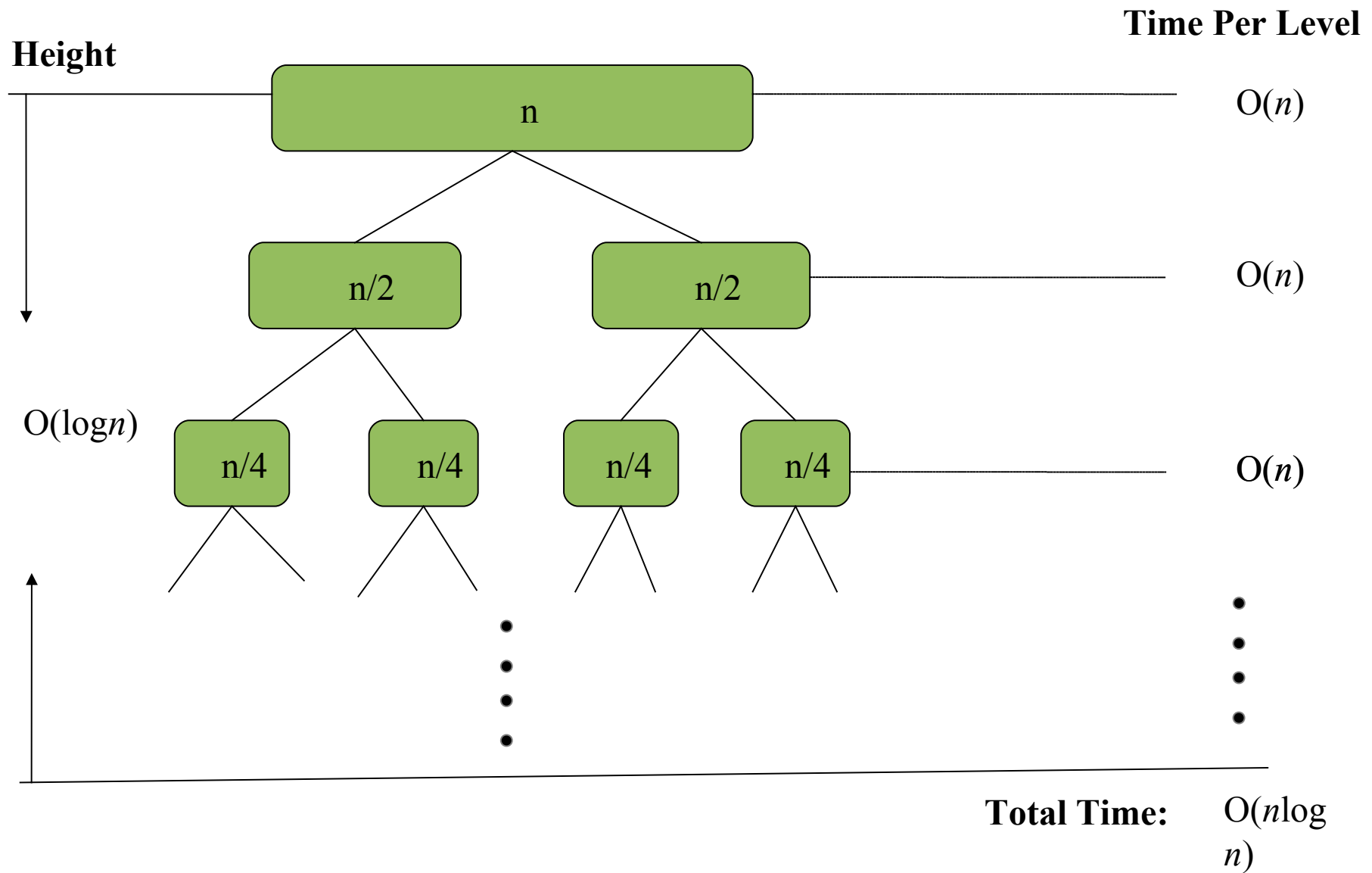


# Merge Sort

```
mergesort(item_type s[], int low, int high) {
 int i; /* counter */
 int middle; /* index of middle element */
 if (low < high) {
 middle = (low+high)/2;
 mergesort(s,low,middle);
 mergesort(s,middle+1,high);
 merge(s, low, middle, high);
 }
}
```



# Analysis of Merge Sort



# Merge Sort: Analysis

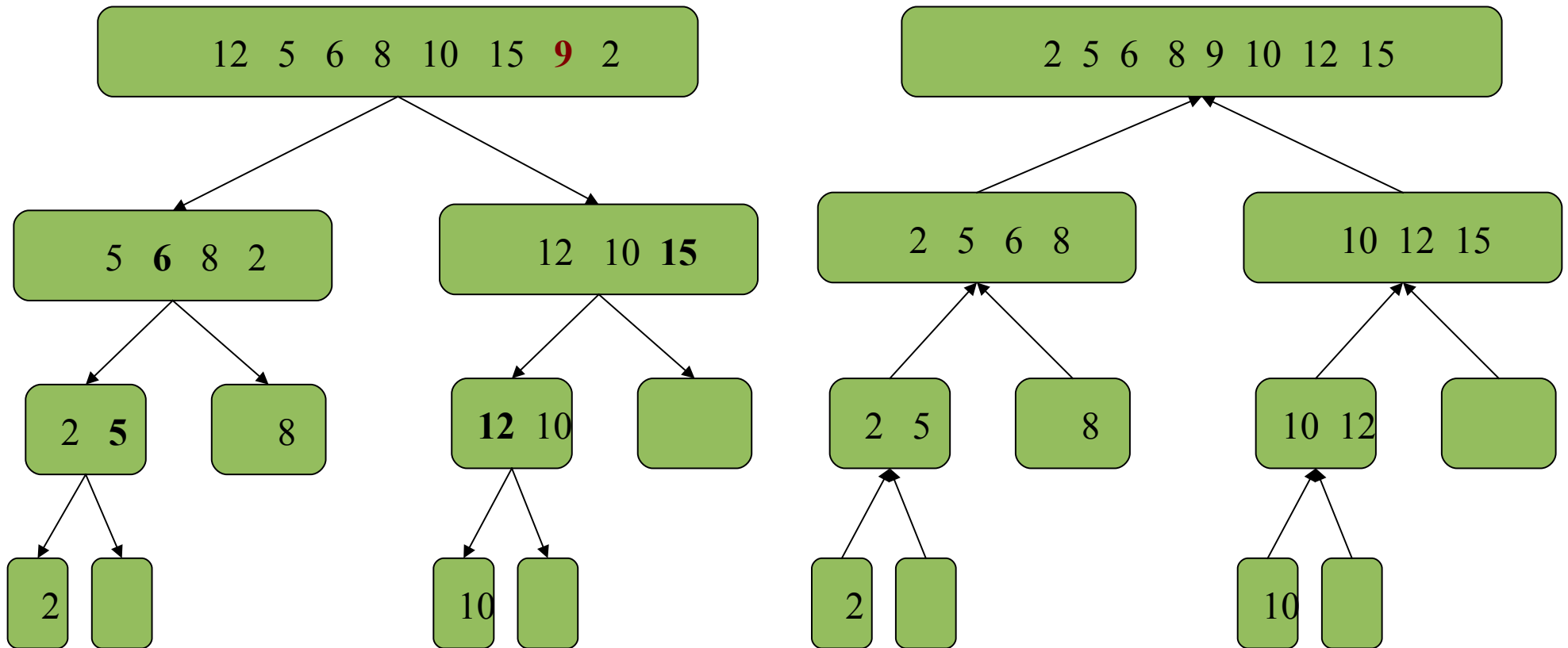
- work done on the  $k$ th level involves merging  $2^k$  pairs sorted list, each of size  $n/2^{k+1}$ 
  - A total of at most  $n \cdot 2^k$  comparisons
- Linear time for merging at each level
  - Each of the  $n$  elements appear exactly in one of the subproblems at each level
- Requires extra memory



# Quick Sort

- A divide and conquer strategy which also uses randomization
- Divide
  - Select a random pivot  $p$ . Divide  $S$  into two subarrays, where one contains elements  $< p$  and the other which are  $> p$
- Recur
  - Sort subarrays by recursively applying quicksort on each subarray
- Conquer
  - Since the subarrays are already sorted, no work is needed in merge part

# Quick Sort





# Quick Sort

```
■ quickSort(Element[] E, int first, int last)
 if (first < last)
 Element pivot = E[first];
 int splitPoint = partition(E,pivot,first,last);
 E[splitPoint] = pivot;
 quickSort(E, first,splitpoint-1);
 quickSort(E, splitpoint+1, last);
```

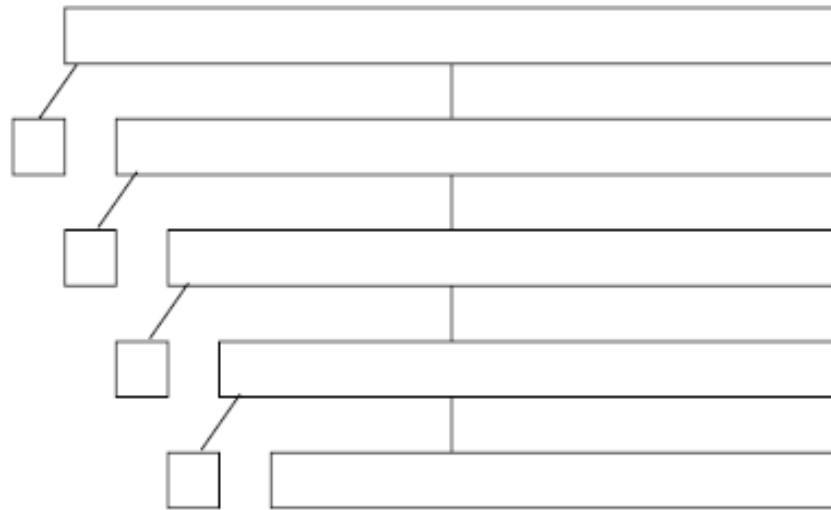
# Quick Sort

- Selection of pivot
  - Can be first or last element (here)
  - Median element
  - Random element



# Quick Sort Analysis

- Worst Case Running Time
  - Occurs when pivot is always the largest element
  - Selecting the first element or last element as pivot causes this problem when list is already sorted
  - Running time proportional to  $n+(n-1)+(n-2)+\dots+1$
  - $O(n^2)$



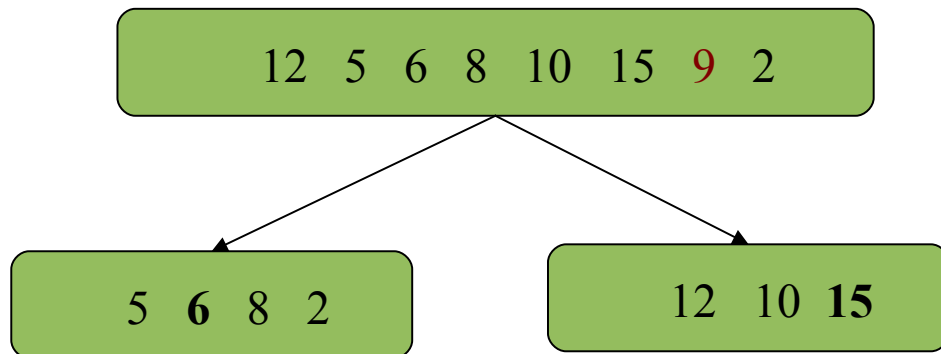
Src: Skiena, Algorithm Design Manual, Chapter 4.6

# Average Case Analysis

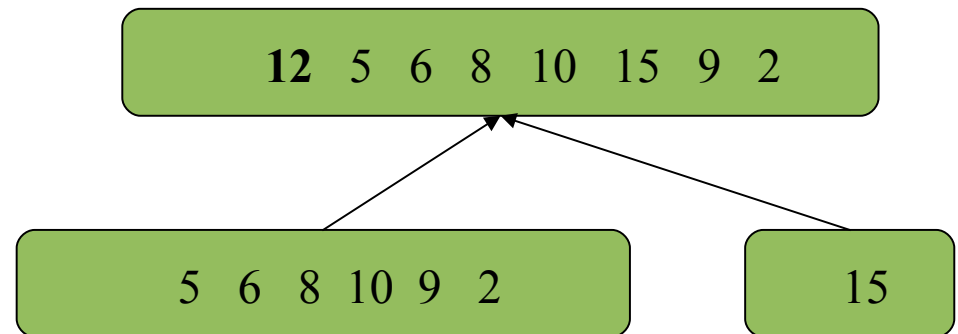
- The partition in a quick sort of size  $s$  can be
  - Good : if the sizes of partitions are each less than  $3s/4$ 
    - A partition is good with the probability  $\frac{1}{2}$
    - $\frac{1}{2}$  the pivots cause good partitions

2 5 6 8 9 10 12 15

- Bad : if one of the partitions has size greater than  $3s/4$



Good



Bad



# Average Case Analysis

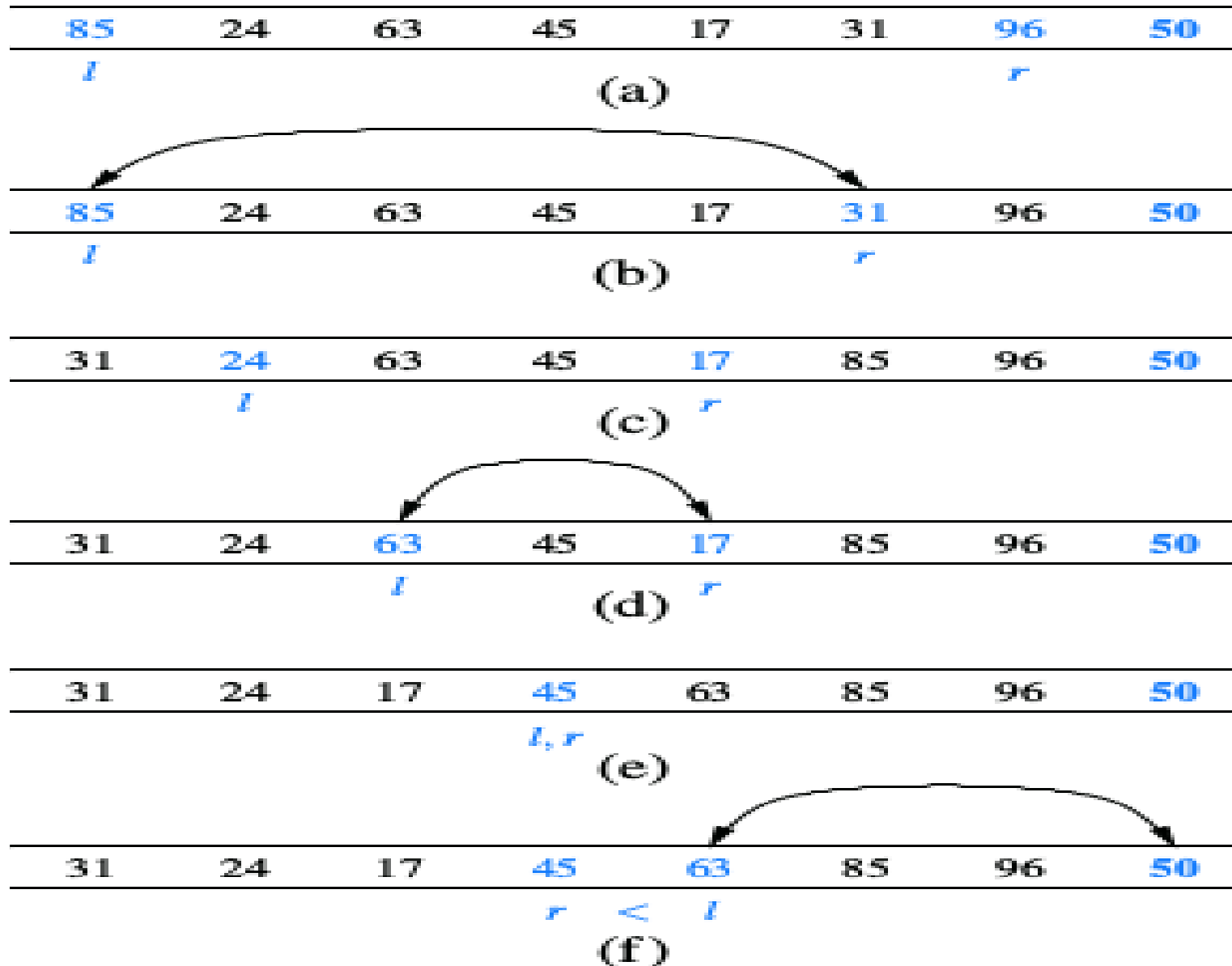
- For a node of depth  $i$ , we expect
  - $i/2$  ancestors are good calls, they are result of good pivots
  - The size of the input sequence for the current call is at most  $(3/4)^{i/2}n$
- For a node of depth  $2\log_{4/3}n$ ,
  - the expected input size is one
  - The expected height of the quick-sort tree is  $O(\log n)$
  - The amount of work done at the nodes of the same depth is  $O(n)$
- Expected running time of quicksort is  $O(n\log n)$

# In-Place Quick Sort

- Quick sort can be implemented without extra memory
- Replace operation used in partition step
  - Elements less than pivot have rank less than  $p$  of pivot
  - Elements greater than pivot have rank greater than  $p$



# In-Place Quick Sort



# Quick Sort: Partitioning

- PARTITION(A,p, r) //inplace partitioning
  - $x = A[r]$
  - $i = p-1$
  - **for**  $j = p$  **to**  $r-1$ 
    - **if**  $A[j] \leq x$ 
      - $i = i + 1$
      - exchange  $A[i]$  with  $A[j]$
  - exchange  $A[i+1]$  with  $A[r]$
- return  $i+1$

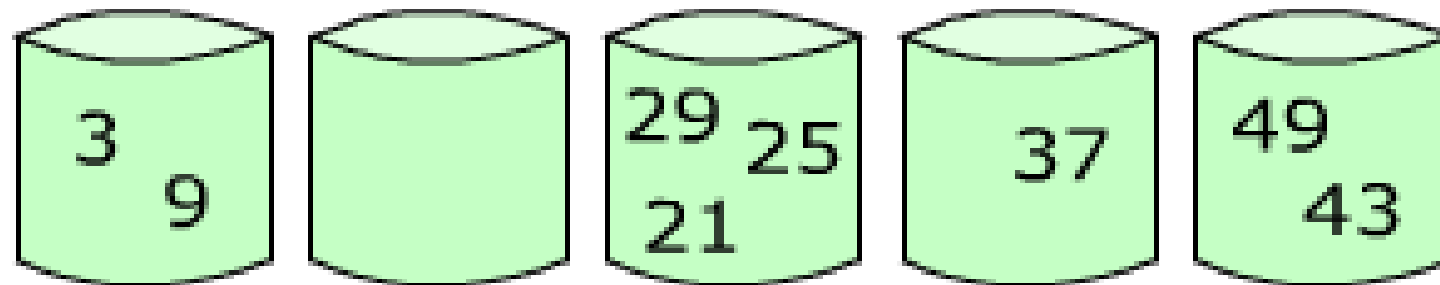
# External Sort: Bucket Sort

- Assumes elements are uniformly distributed
- Distribution sort
  - Array is partitioned into  $n$  equal sized subintervals/buckets
  - Each interval is a bucket
    - e.g each bucket can contain element starting with a particular alphabet
  - Each bucket sorted separately
    - Can use bucket sort again
    - Use other sorting techniques
- Elements from the buckets merged to form the sorted list



# Bucket Sort

29 25 3 49 9 37 21 43



0-9

10-19

20-29

30-39

40-49

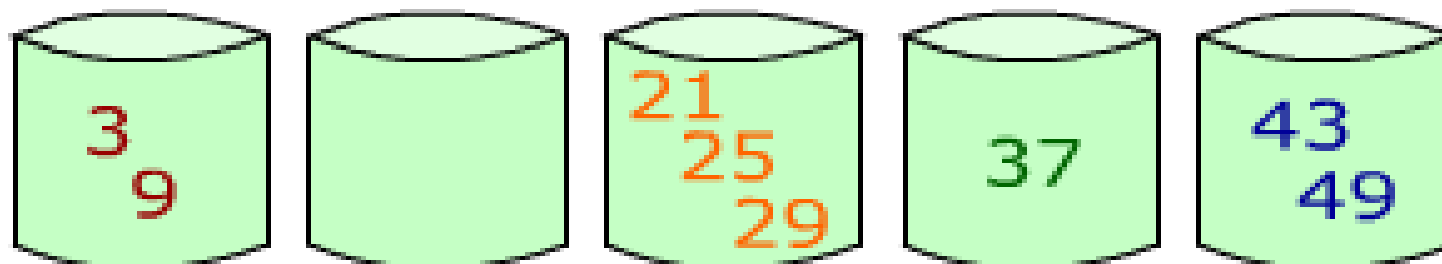
0-9

10-19

20-29

30-39

40-49



3 9 21 25 29 37 43 49

[http://en.wikipedia.org/wiki/File:Bucket\\_sort\\_2.png](http://en.wikipedia.org/wiki/File:Bucket_sort_2.png)

# Bucket Sort: Properties

- Let  $S$  be a sequence of  $n$  (key, element) items with keys in the range  $[0, N - 1]$ 
  - Bucket-sort uses the keys as indices into an auxiliary array  $B$  of sequences (buckets)
  - Key-type Property
    - The keys are used as indices into an array and cannot be arbitrary objects
    - No external comparator
- Stable Sort Property
  - The relative order of any two items with the same key is preserved after the execution of the algorithm



# Methods for Bucketing

- Integer keys in the range  $[a, b]$ 
  - Put item  $(k, o)$  into bucket  $B[k - a]$
- String keys from a set  $D$  of possible strings, where  $D$  has constant size (e.g., names of the Indian states)
  - Sort  $D$  and compute the rank  $r(k)$  of each string  $k$  of  $D$  in the sorted sequence
  - Put item  $(k, o)$  into bucket  $B[r(k)]$



# Bucket Sort: Pseudocode

## BUCKET-SORT(A)

let  $B[0 : n - 1]$  be a new array

$n = A.length$

**for**  $i = 0$  **to**  $n - 1$

    make  $B[i]$  an empty list

**for**  $i = 1$  **to**  $n$

    insert  $A[i]$  into list  $B[\text{floor}(n A[i])]$

**for**  $i = 0$  **to**  $n - 1$

    sort list  $B[i]$  with insertion sort

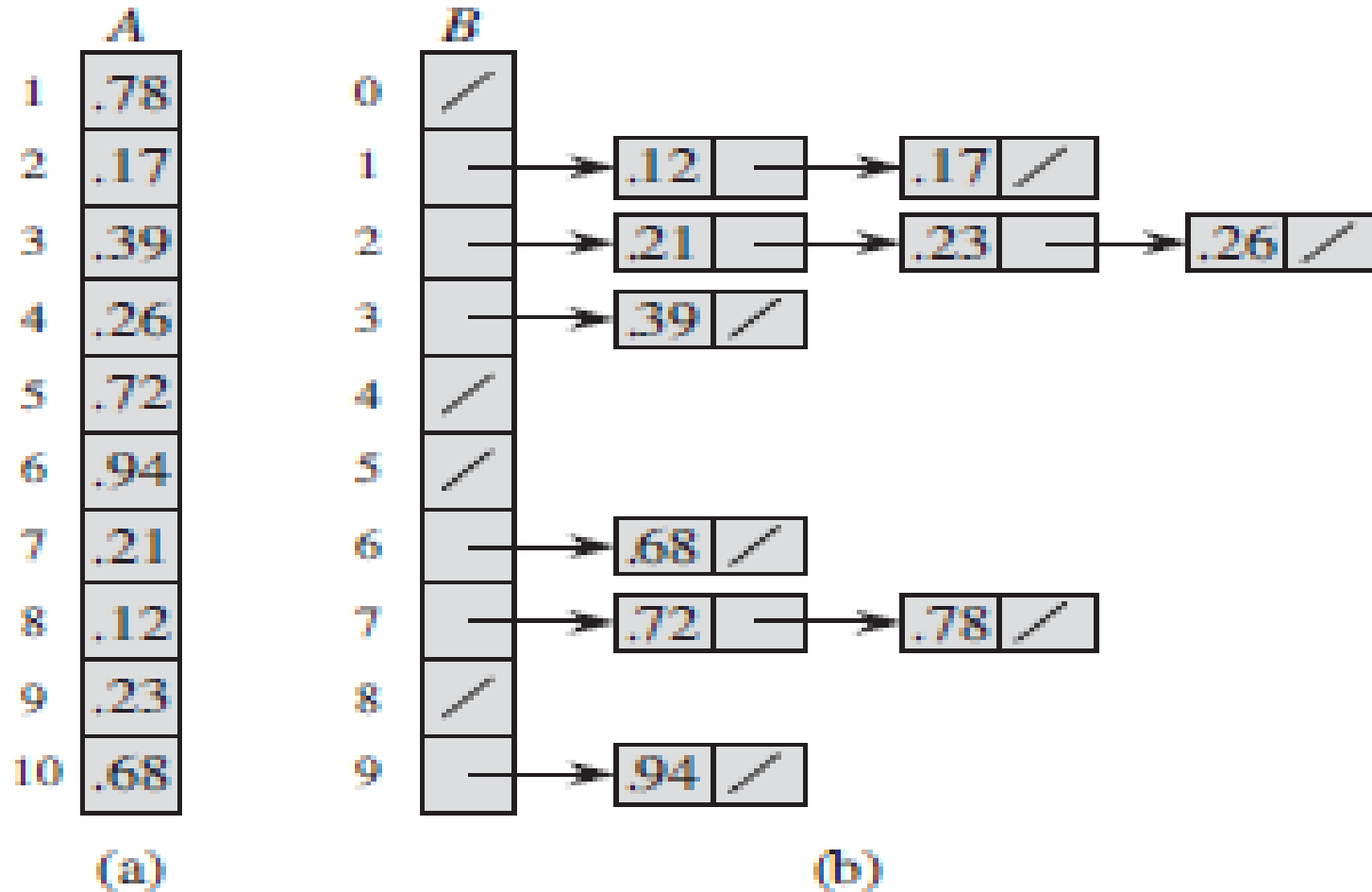
concatenate the lists  $B[0], B[1], \dots, B[n]$  together  
in order

# Analysis of Bucket Sort

- Expected time is  $O(n)$ 
  - Each step takes  $O(n)$  time except for sorting each bucket
  - Expected time to sort bucket  $B_i$  is  $E[O(n_i^2)] = O(E[n_i^2])$ 
    - Depends on distribution of each random variable  $n$
  - $$\sum_{i=1}^n O\left(E\left(n_i^2\right)\right)$$
  - Given  $n$  elements and  $n$  buckets, probability that a given element falls in a bucket  $B[i]$  is  $1/n$ 
    - Probability follows Binomial distribution
      - Mean  $E[n_i] = np = 1$ , Variance  $\text{Var}[n_i] = np(1-p) = 1-1/n$
    - $E[n_i^2] = \text{Var}[n_i] + (E[n_i])^2 = 1 - 1/n + 1^2 = \theta(1)$
    - Substituting in above equation expected time is  $O(n)$
- Worst case complexity –  $O(n^2)$  (skewed distribution)



# Bucket Sort: Example



Src: Corman, Rivest etal, “Introduction to Algorithms”, 8.4



# Problem

- Using the previous example as a model, illustrate the operation of BUCKET-SORT on the array A (.79, .13, .16, .64, .39, .20, .89, .53, .71, .42).
- Use bucket sort to sort the following elements
  - (233, 456, 567, 777, 554, 343, 267, 466, 786, 900, 600, 545, 531, 879, 821, 289, 192, 945)
  - How are the buckets chosen?

# Lexicographic Order

- A d-tuple is a sequence of d keys  $(k_1, k_2, \dots, k_d)$ , where key  $k_i$  is said to be the i-th dimension of the tuple
  - Example: coordinates in 3d space
- The lexicographic order of two d-tuples is recursively defined as follows
  - $(x_1, x_2, \dots, x_d) < (y_1, y_2, \dots, y_d)$  if
  - $x_1 < y_1 \vee x_1 = y_1 \wedge (x_2, \dots, x_d) < (y_2, \dots, y_d)$ 
    - i.e., the tuples are compared by the first dimension, then by the second dimension etc



# Radix Sort

- Specialization of lexicographic sort
  - Lexicographic-sort sorts a sequence of d-tuples in lexicographic order by executing a stable sort algorithm d times
  - Radix sort uses bucket sort for each dimension
  - Applicable where keys in each dimension are integers
- Example:
  - (7, 4, 6) (5, 1, 5) (2, 4, 6) (2, 1, 4) (3, 2, 4)
  - (2, 1, 4) (3, 2, 4) (5, 1, 5) (7, 4, 6) (2, 4, 6)
  - (2, 1, 4) (5, 1, 5) (3, 2, 4) (7, 4, 6) (2, 4, 6)
  - (2, 1, 4) (2, 4, 6) (3, 2, 4) (5, 1, 5) (7, 4, 6)



# Radix Sort

- **RADIX-SORT(A,d)**
  - for  $i = 1$  to  $d$ 
    - use a stable sort to sort array A on digit  $i$
- Given  $n$   $d$ -digit numbers in which each digit can take on up to  $k$  possible values, RADIX-SORT correctly sorts these numbers in  $\Theta(d \cdot (n+k))$  time if the stable sort it uses takes  $\Theta(n+k)$  time.

|     |     |     |     |
|-----|-----|-----|-----|
| 329 | 720 | 720 | 329 |
| 457 | 355 | 329 | 355 |
| 657 | 436 | 436 | 436 |
| 839 | 457 | 839 | 457 |
| 436 | 657 | 355 | 657 |
| 720 | 329 | 457 | 720 |
| 355 | 839 | 657 | 839 |