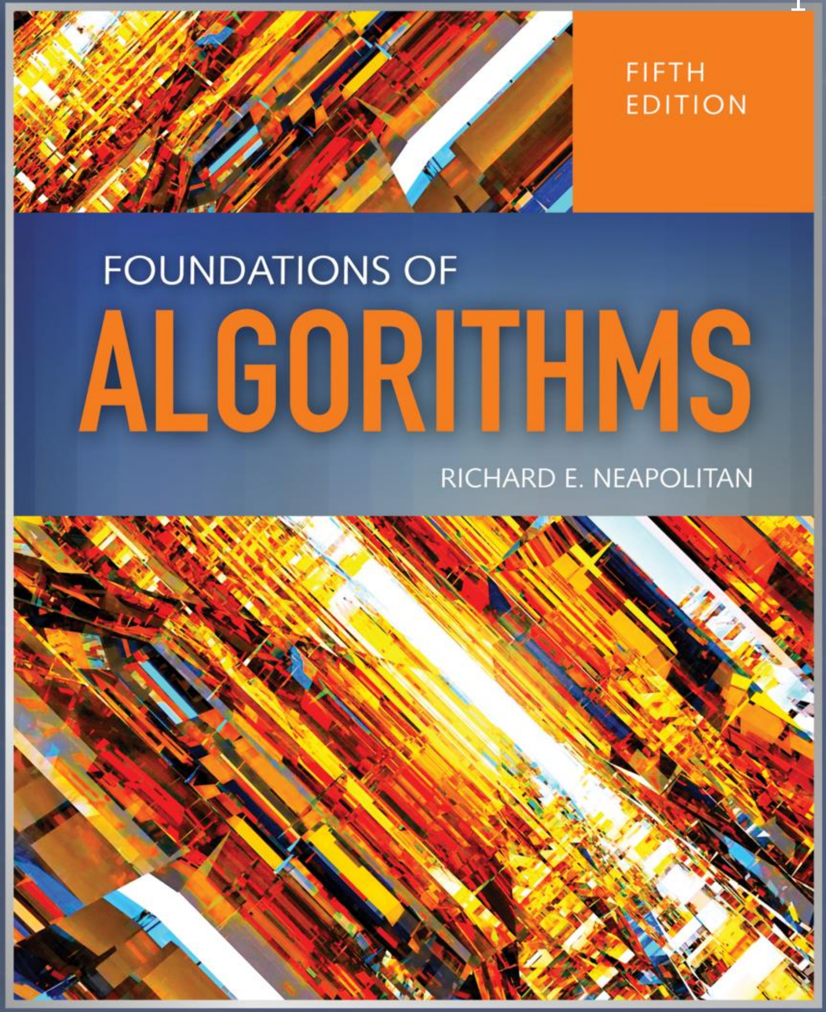
Introduction to Computational Complexity: The Sorting Problem

Chapter 7



Objectives

- Use computational complexity analysis to determine a lower bounds on sorting algorithms
- Analyze algorithms that sort only by comparison of keys
- Use computational complexity analysis to determine a lower bounds of quadratic time on a the class of sorting algorithms that remove one inversion per comparison
- Analyze the class of θ(nlgn) sorting algorithms

Objectives

- Prove a lower bound on algorithms that sort only by comparing keys
- Discuss Radix Sort

Computational Complexity

- Study of all possible algorithms that solve a given problem
- Determine a lower bound on efficiency of all algorithms for a given problem
- Problem analysis as opposed to algorithm analysis

e.g. Matrix Multiplication

- Computational complexity analysis has determined a lower bound on the efficiency as $\Omega(n^2)$
- Does not mean it is possible to create an algorithm $\theta(n^2)$
- It means it is impossible to create an algorithm better than $\theta(n^2)$
- Best algorithm to date: $\theta(n^{2.38})$

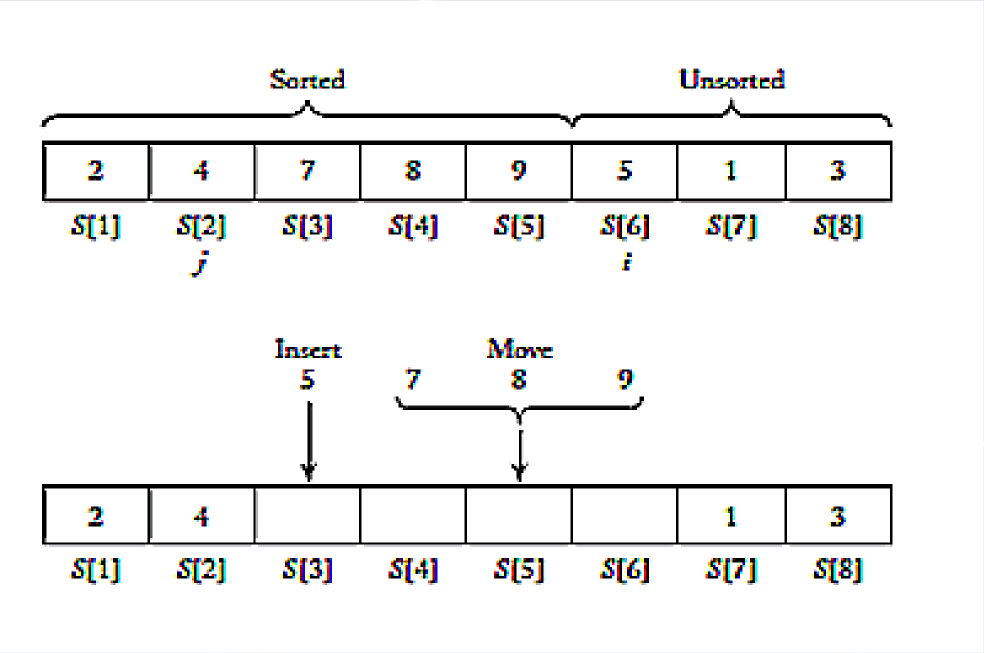
e.g Matrix Multiplication

- Proceed? Two directions:
  - Try to find a more efficient algorithm
  - Use computational complexity analysis to find a higher lower bound

Sort

- Re-arrange records according to a key field
- Algorithms that sort by comparison of keys can compare 2 keys to determine which is larger and can copy keys
- Cannot do other operations on them

Figure 7.1


Sorted
Unsorted
2
4
7
8
9
5
1
3
S[1]
S[2]
S[3]
S[4]
S[5]
S[6]
S[7]
S[8]
j
i
Insert
5
Move
7
8
9
2
4
1
3
S[1]
S[2]
S[3]
S[4]
S[5]
S[6]
S[7]
S[8]

Insertion Sort – Algorithm 7.1

- Insert records in an existing sorted array – arranging cards being dealt one at a time
- Assume keys in first i-1 array slots are sorted
- Let x be the key in the ith slot
- Compare x with A[i-1], A[i-2], . . . Until A[j]< x
- Move A[j+1] through A[i-1] to A[j+2] through A[i]
- Set A[j+1] to x
- Repeat for i = 2 to n

Worst-case Time Complexity Analysis of Number of Comparisons of Keys

- Basic Operation: Comparison of S[j] with x
- Input size: n, the number of keys to be sorted
- Assume short-circuit evaluation
- Prior to loop execution, j set to i-1
- j decremented at each loop iteration
- j > 0 clause of the while expression becomes FALSE after i-1 iterations of the while loop
- While loop executes from i = 2 to n

Total number of comparisons is at most

$$\sum_{i=2}^{n}(i-1) = \frac{n(n-1)}{2}$$

Worst-case behavior of Insertion Sort

- Keys in original array are in non-increasing order
- At position i+1, S[i+1]<S[j] for 1 <= j <=I
- Positions S[1] . . . S[i] sorted
- While loop will exit after i iterations because S[j]> x will always be true

Space usage of Insertion Sort

- In-place sort

Table 7.1 analysis summary for exchange, insertion, and selection sorts

Algorithm	Comparisons of Keys	Assignments of Records	Extra Space Usage
Exchange Sort	$T(n) = \frac{n^2}{2}$	$W(n) = \frac{3n^2}{2}$	In-place
		$A(n) = \frac{3n^2}{4}$	
Insertion Sort	$W(n) = \frac{n^2}{2}$	$W(n) = \frac{n^2}{2}$	In-place
	$A(n) = \frac{n^2}{4}$	$A(n) = \frac{n^2}{4}$	
Selection Sort	$T(n) = \frac{n^2}{2}$	$T(n) = 3n$	In-place

*Entries are approximate.

Lower Bounds Sort by Comparison of Keys

- Insertion Sort, Exchange Sort, Selection Sort
- Worst case input of size n contains n distinct keys
- n! different orderings
- Permutation: [k1, k2, . . . ,kn] ki is the integer at the ith position
  - E.g. [3, 2, 1] k1 = 3, k2 = 2, k3 = 1
- Inversion (ki,kj) pair such that i < j and ki > kj
- [3, 2, 4, 1, 6, 5]
  - Inversions: (3,2), (3,1), (2,1), (6,5), (4,1)

Theorem 7.1

Any algorithm that sorts n distinct keys only by comparison of keys and removes at most one inversion after each comparison must, I the worst case, do at least n(n-1)/2 comparisons of keys and on the average n(n-1)/4 comparisons of keys

Proof Theorem 7.1

- Show there is a permutation with n(n-1)/2 inversions
- At most one inversion is removed with each comparison => n(n-1)/2 comparisons
- Show [n, n-1, n-2, . . . , 3, 2, 1] is such a permutation
- n-1 inversions with n
- n-2 inversions with n-1

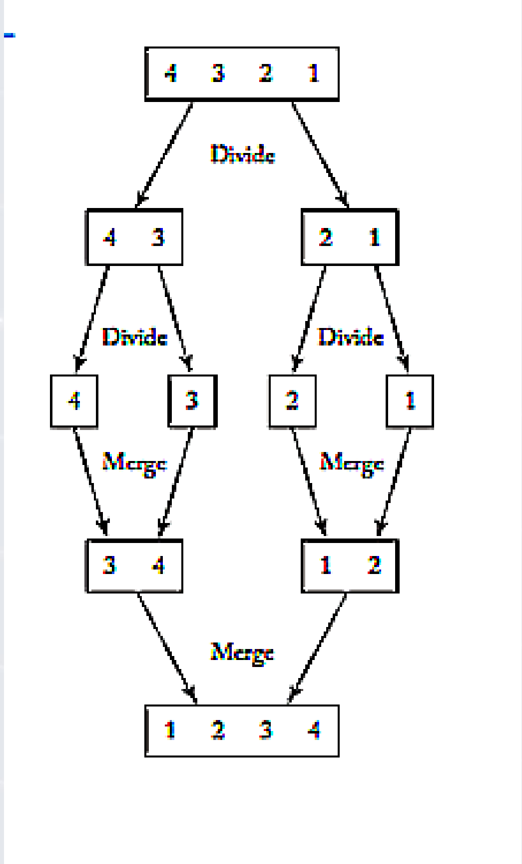
Algorithms addressed by Theorem 7.1

- Insertion Sort
- Selection Sort
- Exchange Sort

Mergesort

- Input in reverse order: S = [4,3,2,1]
- 3 and 1 are compared and 1 placed in output array
  - Inversions (3,1) and (4,1) removed
- 3 and 2 are compared and 2 placed in output array
  - Inversions (3,2) and (4,2) removed
- $W(n) = n \lg n - (n - 1) \in \theta(n \lg n)$

Figure 7.2


4 3 2 1
Divide
4 3
2 1
Divide
Divide
4
3
2
1
Merge
Merge
3 4
1 2
Merge
1 2 3 4

Extra space usage

- Stack grows to a depth of $\lceil \lg n \rceil$
- Space for additional array of records dominates
- Every-case extra space usage is $\theta(n)$

Improvements to Mergesort

- Dynamic Programming version
- Linked List version

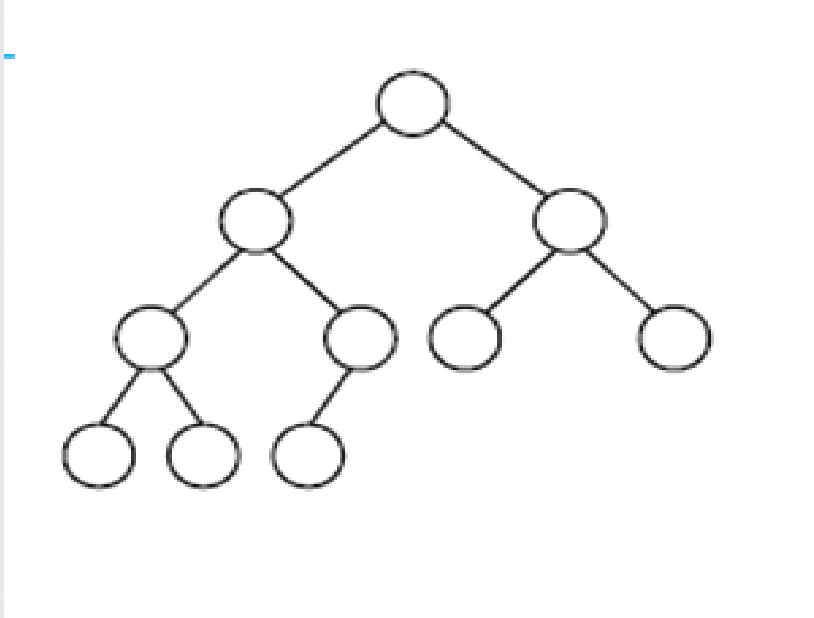
Definitions

- Tree: connected, acyclic graph
- Depth of a node: number of edges in the unique path from the root to that node
- Depth d of a tree is the maximum depth of all nodes in the tree
- Leaf is any node with no children
- Internal node is any node that has at least one child

Binary Tree

- Complete Binary Tree
  - All internal nodes have two children
  - All leaves have depth d
- Essentially Complete Binary Tree
  - A complete binary tree down to depth of d-1
  - Nodes with depth d are as far to the left as possible

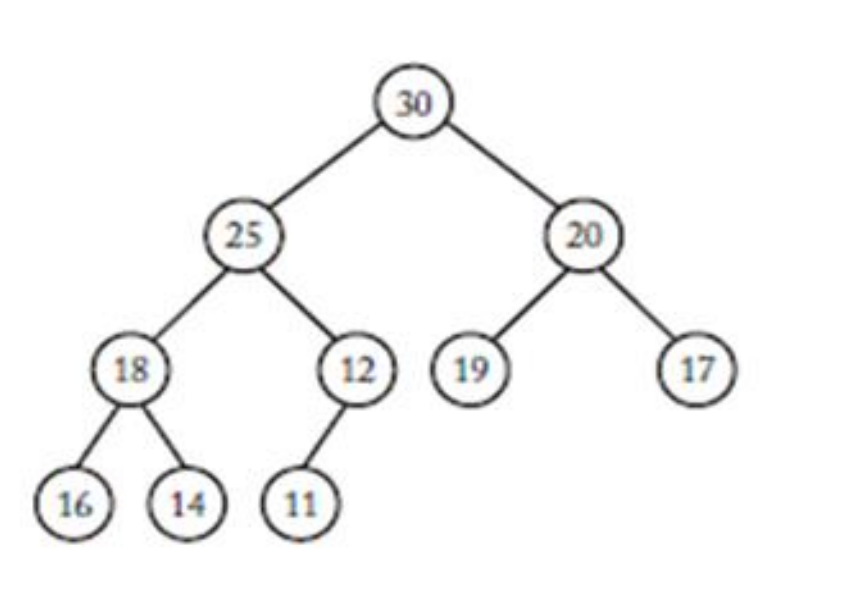
Figure 7.4



Heap: Essentially complete binary tree such that:

- Values stored at the nodes come from an ordered set
- Heap Property: value stored at each node is >= the values stored at its children

Figure 7.5



Heapsort

- In-place sort
- $\Theta(n \lg n)$
- Main idea:
  - Arrange keys to be sorted in a heap
  - Repeatedly remove the key stored at the root while maintaining the heap property
  - Removes keys in non-decreasing order
  - As keys removed, placed in array starting in nth entry down to the first position (reverse order)
  - Array will be sorted in non-decreasing order

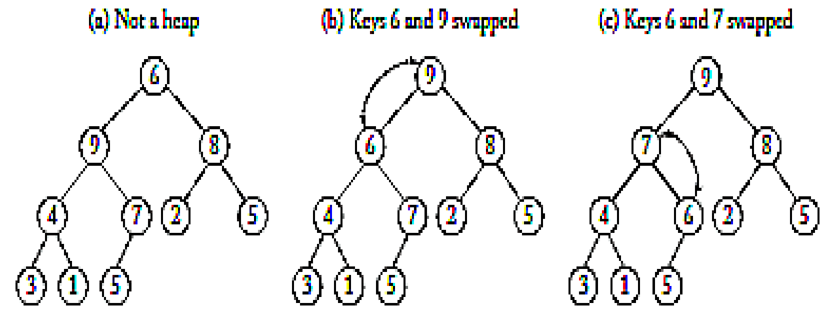
Restoring the heap property:

- Remove key at root
- Replace key at root with key stored at the bottom node (far right leaf) and deleting bottom node (decrement heap size)
- Sift new root down the heap until heap property restored
  - Compare key at root with larger of the keys of the root's children
  - If key at root is smaller, exchange keys

Restoring the heap property

- Repeat process down the tree until the key at node is not smaller than the larger of the children

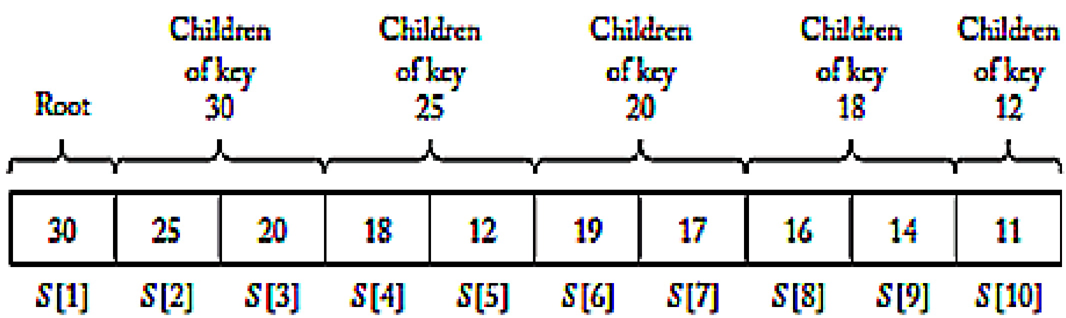
Figure 7.6


(a) Not a heap
(b) Keys 6 and 9 swapped
(c) Keys 6 and 7 swapped
6
9
8
4
7
2
5
3
1
5
9
6
8
4
7
2
5
3
1
5
9
7
8
4
6
2
5
3
1
5

Array implementation of heap

- Root stored at A[1]
- Let I be the index of a given node m
  - 2i = index of m’s left child
  - 2i+1 = index of m’s right child

Figure 7.8


Root
Children of key 30
Children of key 25
Children of key 20
Children of key 18
Children of key 12
30
25
20
18
12
19
17
16
14
11
S[1]
S[2]
S[3]
S[4]
S[5]
S[6]
S[7]
S[8]
S[9]
S[10]

Depth	# nodes with this depth	> # nodes that a key would be sifted
0	2^0	d - 1
1	2^1	d - 2
2	2^2	d - 3
…	…	…
j	2^j	d – j - 1
…	…	…
d - 1	2^{d-1}	0

Heapsort – Algorithm 7.5 Worst-Case Time Complexity Analysis of Number of Comparisons of Keys

- Basic instruction: Comparison of Keys in procedure siftdown
- Input size: n
- makeheap:
  - Upper bound on total # nodes all keys will be sifted through n – 1
  - For each pass of while loop in siftdown, 2 comparisons of keys

Heapsort-Algorithm 7.5 Worst-Case Time Complexity Analysis of Number of Comparisons of Keys

  - Number of comparisons of keys done by makeheap is at most $2(n – 1)$
- Analysis of remove keys $2n \lg n – 4n + 4$
- Combine analysis of makeheap and remove keys: $2(n-1)+2n \lg n – 4n + 4 = 2(n \lg n – n + 1) \approx 2n \lg n$

Extra space for heapsort

- In-place sort – no extra space
- Θ(1)

Lower Bounds for Sorting only by comparison of keys

- Mergesort and heapsort: $\theta(n \lg n)$
- Substantially better than $\theta(n^2)$
- Can it be improved?
- Show that sorting by comparison a faster algorithm cannot be developed

```
void sortthree(keytype S[]) //S indexed
from 1 to 3
{
            keytype a, b, c;
a = S[1]; b = S[2]; c = S[3];
if          (a < b)
                        if (b < c)
                                    S = a, b, c;
//means S[1]=a; S[2]=c;S[3]=c;
                        else if (a < c)
                                    S = a, c, b;
                        else
                                    S = c, a, b;
                        else if (b < c)
                                    if (a < c)
                                                S = b,
a, c;
                                    else
                                                S = b,
c, a;
                        else
                                    S = c, b, a;
}
```

Figure 7.11

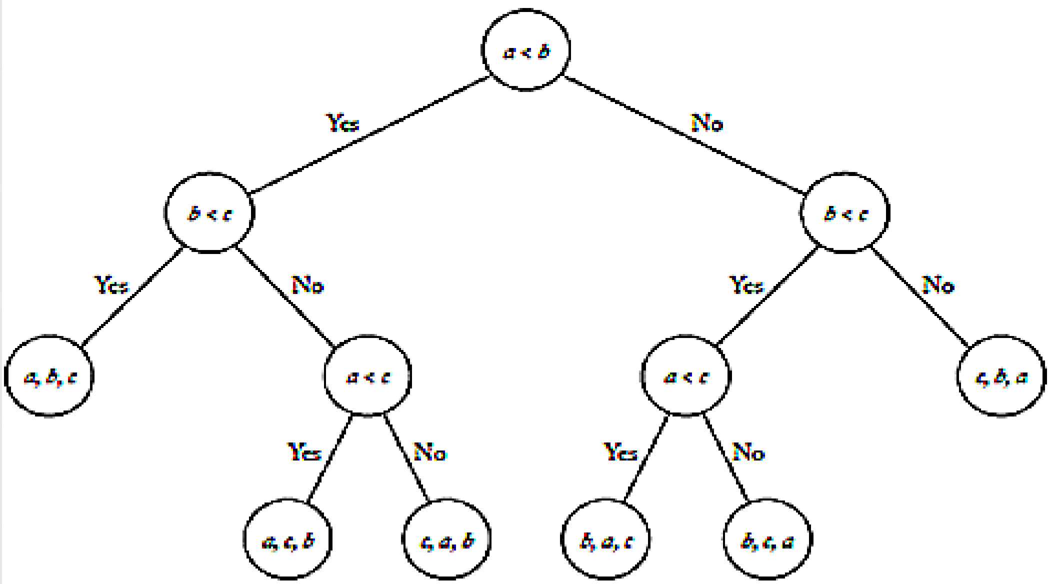
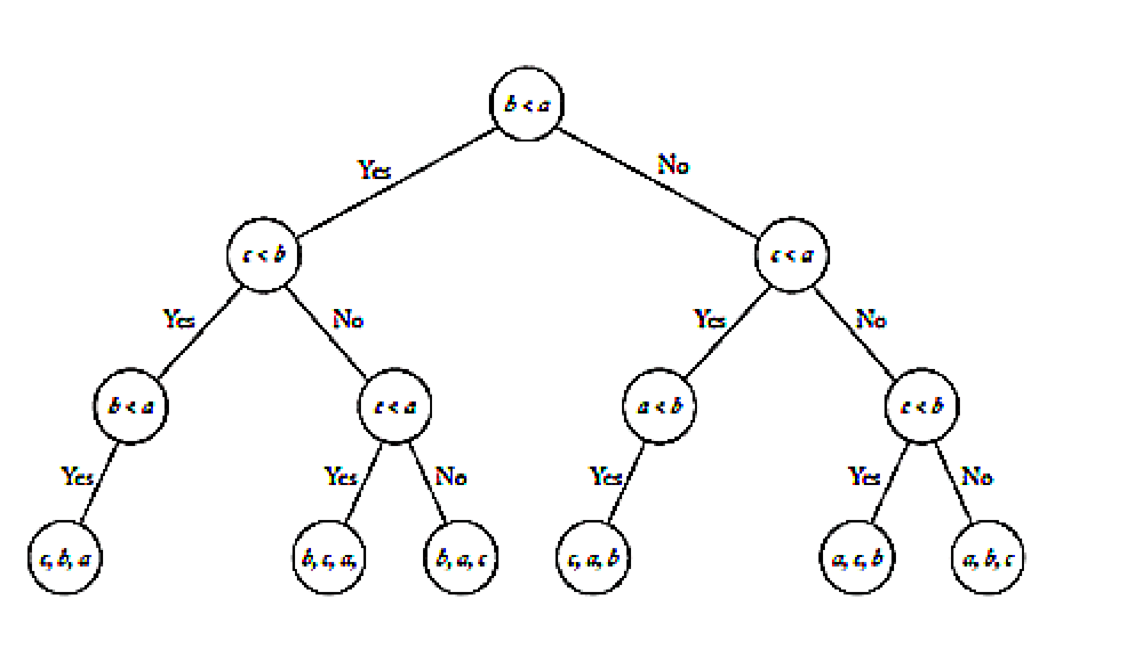

a < b
Yes
No
b < c
b < c
Yes
No
Yes
No
a, b, c
a < c
a < c
c, b, a
Yes
No
Yes
No
a, c, b
c, a, b
b, a, c
b, c, a

Figure 7.12


b < a
Yes
No
c < b
c < a
Yes
No
Yes
No
b < a
c < a
a < b
c < b
Yes
Yes
No
Yes
Yes
No
c, b, a
b, c, a,
b, a, c
c, a, b
a, c, b
a, b, c

Lemma 7.1

- To every deterministic algorithm for sorting n distinct keys there corresponds a pruned, valid, binary decision tree containing exactly n! leaves

Proof Outline:

- All keys distinct: result of a comparison is < or >
- Each node has at most two children – binary tree
- n! leaves
  - n! different inputs that contain n distinct keys
  - Decision tree is valid only if it has a leaf for every input
  - Decision tree has n! leaves
- Unique path in the tree for each of the n! different inputs

Proof Outline

- Every leaf in a pruned decision tree must be reachable
- Tree can have no more than n! leaves. Therefore, the tree has exactly n! leaves

Lemma 7.2

- Worst-case number of comparisons done by a decision tree is equal to the depth
- Proof Outline
  - Number of comparisons done by a decision tree is the number of internal nodes on the path followed for the input
  - Number of internal nodes is the same as the length of the path
  - Worst case number of comparisons done by the decision tree is the length of the longest path to a leaf (depth of the decision tree)

Lemma 7.3

- If m is the number of leaves in a binary tree and d is the depth: $d >= \lceil \lg m \rceil$
- Proof by Induction
  - Induction Base: complete binary tree depth 0: $2^0 = 1$
  - Induction Hypothesis: Assume for the complete binary tree with depth d: $2^d = m$
- Induction step: show that for the complete binary tree with depth d+1, $2^{d+1} = m'$ where m' is the umber of leaves

Theorem 7.2

- Any deterministic algorithm that sorts n distinct keys only by comparisons of keys must in the worst case do at least $\lceil \lg(n!) \rceil$ comparison of keys
- Proof:
  - Lemma 7.1
  - Lemma 7.3
  - Lemma 7.2

Lemma 7.4

- For any positive integer n, lg(n!) >= n lg n – 1.45n

Theorem 7.3

- Any deterministic algorithm that sorts n distinct keys only by comparison of keys must in the worst case do at least
  - $\lceil n \lg n - 1.45n \rceil$ comparisons of keys
- Proof follows from Theorem 7.2 and Lemma 7.4

Sorting by Distribution

- Keys non-negative integers
- Keys represented in some base
- All keys have the same number of digits
- Radix Sort – based on old card sorting machines
- Radix – any number of alphabet base

Distribute the keys into piles

- Number of piles equals the number base (radix)
- Inspect keys from right to left (lsb -> msb)
- Place a key into a pile corresponding to the digit currently being inspected
- Each pass: if 2 keys are to be placed in the same pile, the key coming from the left-most pile (previous pass) is placed to the left of the other key

Distribute the keys into piles

- Implementation:
  - Piles represented by a linked list
  - After each pass, keys removed from each list pile and merged into single linked list
  - Next pass, single linked list traversed and keys placed in appropriate piles based on the digit being sorted

Figures 7.14

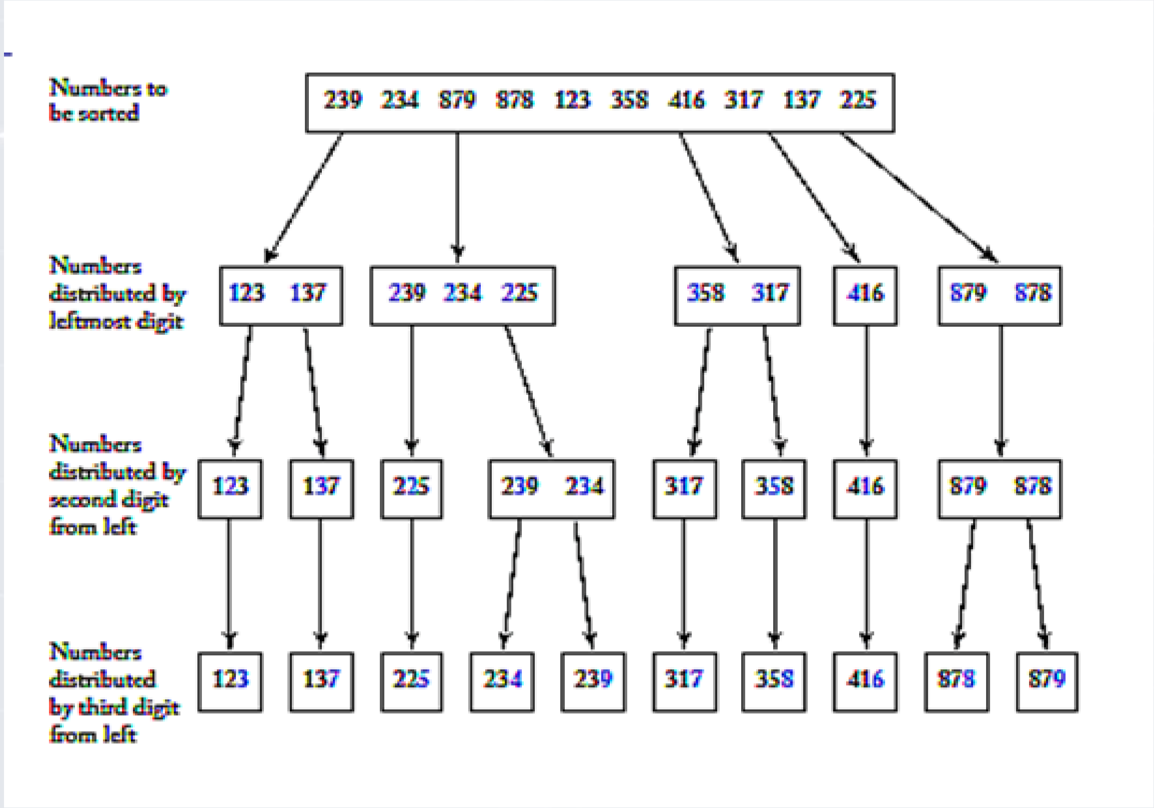
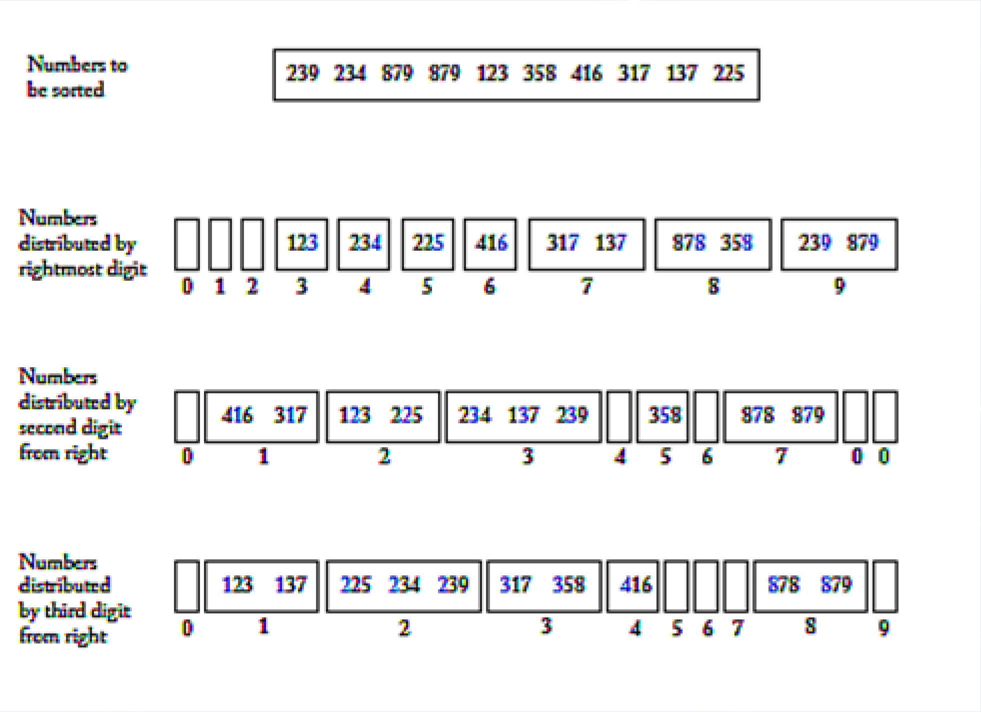

Numbers to be sorted
239 234 879 878 123 358 416 317 137 225
Numbers distributed by leftmost digit
123 137
239 234 225
358 317
416
879 878
Numbers distributed by second digit from left
123
137
225
239 234
317
358
416
879 878
Numbers distributed by third digit from left
123
137
225
234
239
317
358
416
878
879

Figure 7.15


Numbers to be sorted
239 234 879 879 123 358 416 317 137 225
Numbers distributed by rightmost digit
123 234 225 416 317 137 878 358 239 879
0 1 2 3 4 5 6 7 8 9
Numbers distributed by second digit from right
416 317 123 225 234 137 239 358 878 879
0 1 2 3 4 5 6 7 0 0
Numbers distributed by third digit from right
123 137 225 234 239 317 358 416 878 879
0 1 2 3 4 5 6 7 8 9