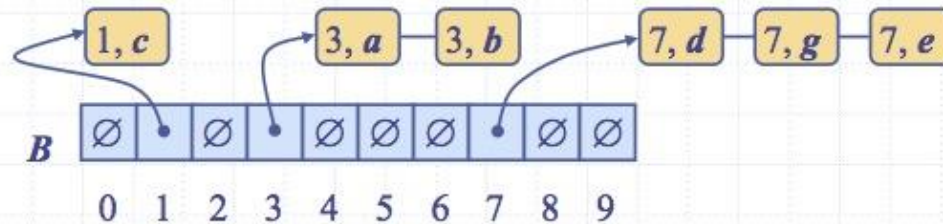# Lesson 6

Lower Bound on Comparison-Based Algorithms:
Discovering the Range of Natural Law

Wholeness of the Lesson

Using the technique of decision trees, one establishes the following lower bound on comparison-based sorting algorithms: Every comparison-based sorting algorithm has at least one worst case for which running time is $\Omega(n \log n)$. Bucket Sort and its relatives, under suitable conditions, run in linear time in the worst case, but are not comparison-based algorithms. Each level of existence has its own laws of nature. The laws of nature that operate at one level of existence may not apply to other levels of existence.

A Lower Bound on Comparison-Based Sorting Algorithms

- ◆ All sorting algorithms discussed so far have used comparison as a central operation

- ◆ So far, all sorting algorithms discussed have a worst-case running time of $\Omega(n\log n)$

- ◆ Using the technique of *decision trees* we can show that nlog n is the best that can be done for comparison-based sorting algorithms

Sample Decision Tree

◆ Every comparison-based sorting algorithm, applied to a given array, can be represented by a decision tree.

◆ To illustrate the technique, we use a simple but typical example: sorting a 3-element array of distinct values *a,b,c.*

Decision Tree for Sorting 3 Elements
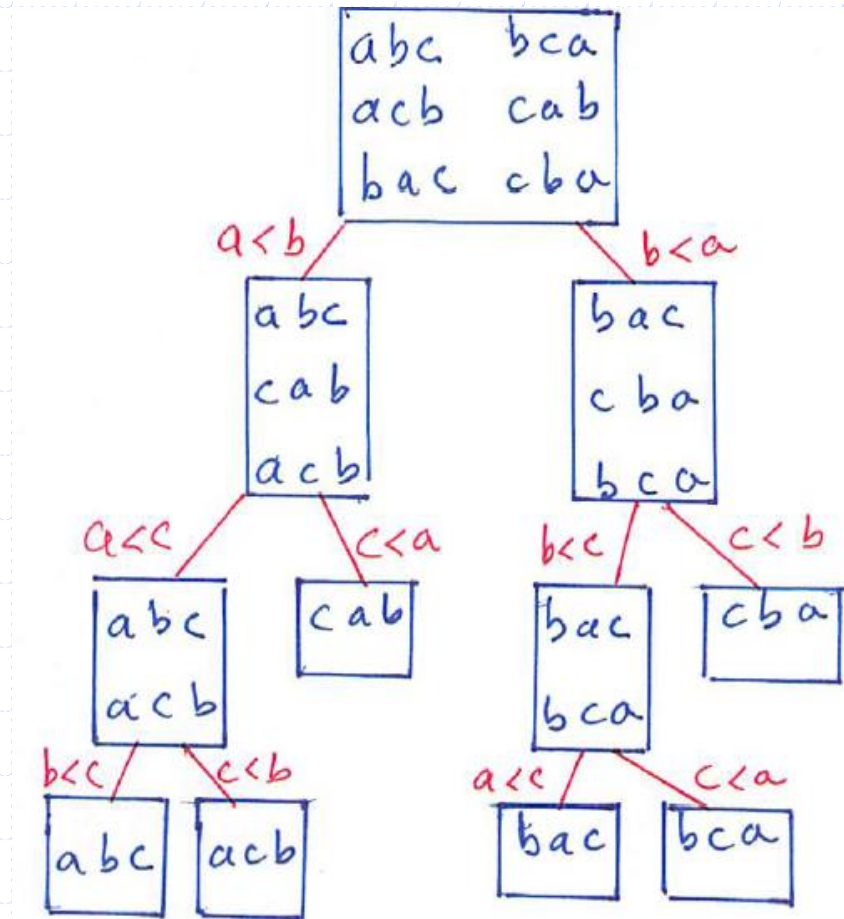
Anatomy of the Decision Tree

◆ Each node of the tree represents all possible sorting outcomes that have not been eliminated by the comparisons done so far.

◆ The labels on the links of the tree represent comparison steps as the algorithm runs. Different algorithms will perform some of the comparison steps in a different sequence.

◆ A leaf in the decision tree represents a possible sorting outcome. The different paths to the leaves represent all possible ways n distinct items could be put in sorted order; therefore, the leaves represent all possible arrangements of n distinct elements.

Decision Tree for MergeSort on 4 Elements
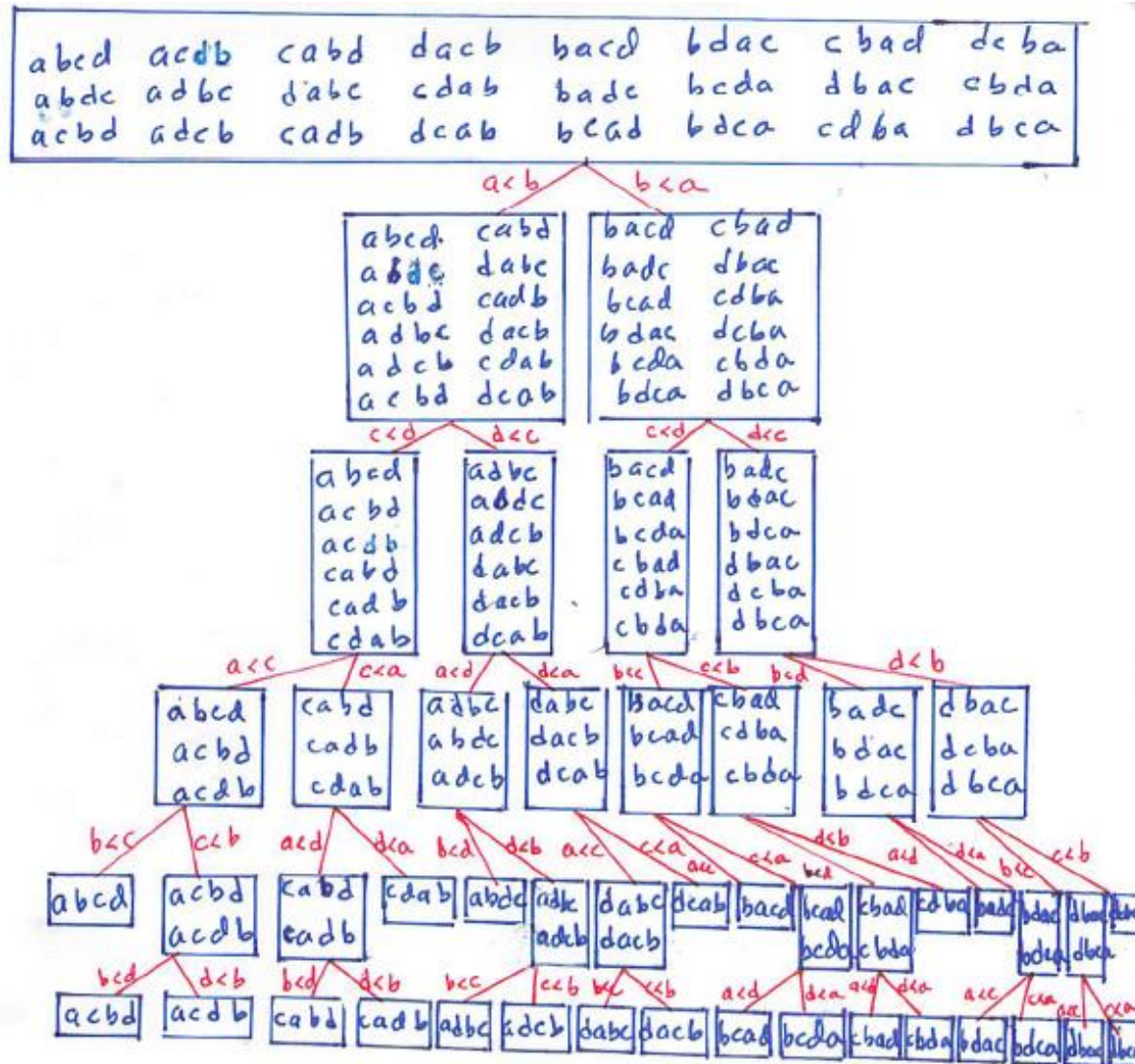
Strategy for Computing # Comparisons

◆ *Observation:* The number of comparisons performed in order to arrive at an arrangement found in a leaf node equals the depth of that leaf node in the decision tree.

◆ Therefore, to count # comparisons performed in the worst case, we determine the depth of the deepest node in the decision tree.

(And the depth of deepest node = height of the tree)

Mathematical Observations

Suppose T is a binary tree with L leaves and height h.

❖ Lemma 1. $L \leq 2^h$. (T has at most 2^h leaves).

Use Lemma 1 to prove the next important fact:

◆ Lemma 2. $h \geq \lceil \log L \rceil$. In particular, a binary tree T with L leaves has a leaf with depth d at least $\lceil \log L \rceil$.

◆ Lemma 3. A decision tree T representing a comparison-based sorting algorithm for an n-element array has n! leaves. [Proof: Follows because leaves represent all possible permutations of the n elements of the array]

Proof of the Lower Bound

Lemmas 2 and 3 tell us that a decision tree for a comparison-based algorithm for sorting an n-element list has a leaf with depth at least log n!

We determine the complexity class of log n!

The Computation of log n!

$$\log n! = \log(1 \cdot 2 \cdot 3 \cdot \ldots \cdot n)$$

$$= \log n + \log(n-1) + \log(n-2) + \ldots + \log 2 + \log 1$$

$$\geq \log n + \log(n-1) + \ldots + \log \frac{n}{2}$$

$$\geq \sum_{i=1}^{n/2} \log \frac{n}{2}$$

$$= \frac{n}{2} \cdot \log \frac{n}{2}$$

$$= \frac{n}{2} \cdot (\log n - 1)$$

$$= \frac{n}{2} \cdot \log n - \frac{n}{2}$$

$$= \Omega(n \log n)$$

Proof of Lemma 1

Lemma 1. If the height of T is h and L is the number of leaves, then L ≤ 2^h .

A binary tree is *completely filled* if every nonempty level has the maximum possible size. In particular:

- every node that has a child has two children, AND
- for any k, if a node at level k has a child, then every node at level k has a child

- Fact: Every completely filled binary tree of height h has 2^h leaves. (Proof by induction on height h.)

- Proof of Lemma 1. Depth d of deepest leaf in T is the height of T. The completely filled binary tree of height d has 2^d leaves. Therefore, T must have ≤ 2^d leaves.

Proof of Lemma 2

Lemma 2. A binary tree T with L leaves and height h

satisfies h ≥ log L; in particular, h ≥ ⌈log L⌉

Proof: By Lemma 1, $L \leq 2^h$. Applying log on both sides (recalling log is increasing) yields the first result. Since h is an integer, the second part also follows.

Specialized Sorting: Bucket Sort

◆ The Context: Suppose we have an array A of n distinct integers a_1, a_2, ..., a_n, all lying in the range 0..m-1.

◆ Sorting strategy:

1. Create an array bucket[] of size m, entries initialized to 0. Create an output array B of size n

2. Scan A. When a_i is encountered, increment the value bucket a_i

3. Scan bucket[]. For each j < m for which bucket[j] > 0, copy j into next available slot in B.

Analysis of Bucket Sort

- Step 1 requires $O(m+n)$
- Step 2 requires $O(n)$
- Step 3 requires $O(m)$

** Therefore, BucketSort runs in $O(m+n)$. When m is $O(n)$, BucketSort runs in $O(n)$

- *Example:* When m is $O(n)$, BucketSort runs in $O(n)$. Example: Sort 1000 integers all lying in the range 0..1999.

- *Handling Duplicates (simple case):* Same algorithm, except in loading array B at the end, if bucket[j] = k, insert k copies of a_j into B.
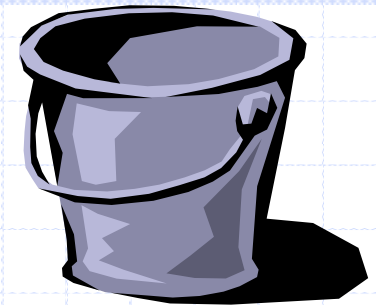
Stability of Bucket-Sort

- Given n integers $a_1, a_2, ..., a_n$ in the range $[0, m-1]$, possibly with duplicates, and n matched objects o1, o2, …, on. We sort array A whose elements are pairs $(a_1, o_1), (a_2, o_2), …, (a_n, o_n)$.

- To handle duplicates in this case, array bucket[] now consist of m *lists* rather than integers. Output array B will consist of pairs (a,o).

  Phase 1: Scan A. When (a_i, o_i) is encountered, copy it to the end of the list in bucket $B[a_i]$

  Phase 2: For $j = 0, ..., m-1$, copy the items of bucket $bucket[j]$ to the end of sequence B.

- As before, running time is O(n+m). Notice the algorithm is stable because we always add newly scanned pairs to end of the list in the appropriate bucket

---

Algorithm $bucketSort(A, m)$

    Input array A of (key, element) items with keys in the range $[0, m-1]$

    Output array B sorted by increasing keys

    $bucket \leftarrow$ array of N empty lists

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

        $(k, o) \leftarrow A[i]$

        $bucket[k].insertLast((k, o))$
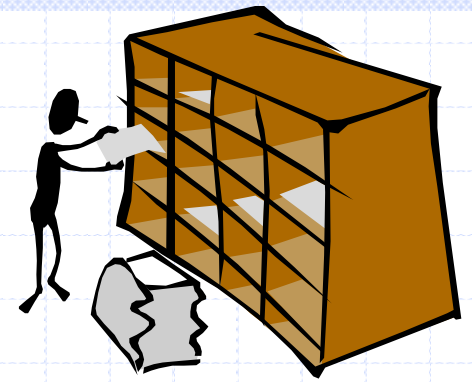
    for $j \leftarrow 0$ **to** $m-1$

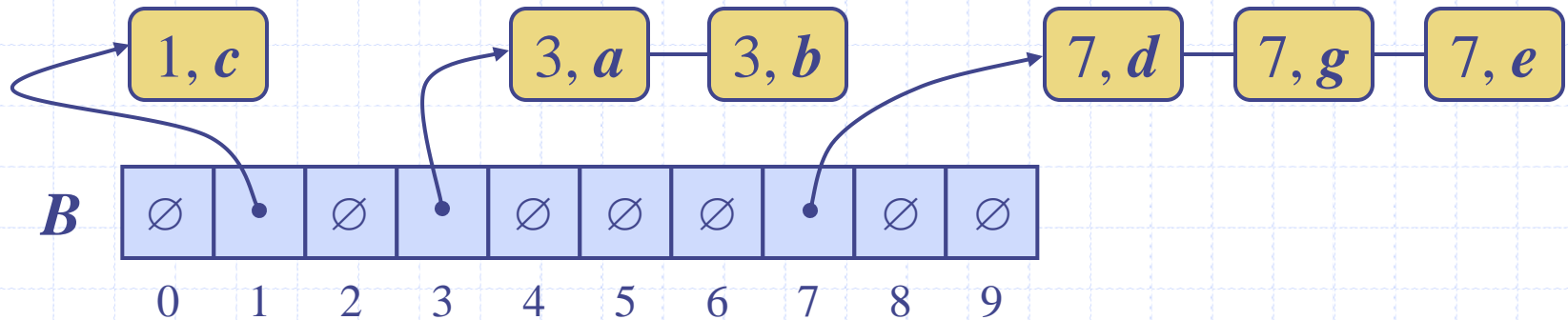   **while** $\neg bucket[j].isEmpty()$

            $(k, o) \leftarrow bucket[j].removeFirst()$

            $B.insertLast((k, o))$

Example

◆ Key range [0, 9]

| 7, *d* | — | 1, *c* | — | 3, *a* | — | 7, *g* | — | 3, *b* | — | 7, *e* |

⬇

| 1, *c* | | 3, *a* | — | 3, *b* | | 7, *d* | — | 7, *g* | — | 7, *e* |

B | ∅ | • | ∅ | • | ∅ | ∅ | ∅ | • | ∅ | ∅ |
    | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 |

⬇

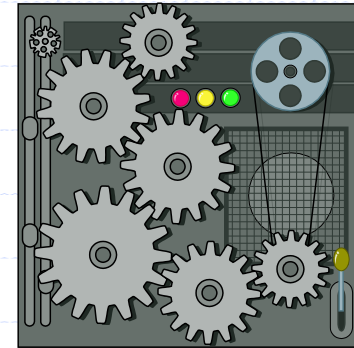| 1, *c* | — | 3, *a* | — | 3, *b* | — | 7, *d* | — | 7, *g* | — | 7, *e* |

Bucket-Sort and Radix-Sort

16

Expanding the Context

- Negative integers can be included if lower bound on these integers is known in advance. Similarly, integers in any range [a,b] can be sorted; worthwhile if b-a is O(n)

- If m is too big (i.e., not O(n)), BucketSort is not useful because scanning the bucket array is too costly

Radix-Sort

- BucketSort doesn't work because range is too big – would run in $\Omega(n^2)$

- Radix-sort is a generalization of BucketSort that uses multiple bucket arrays

- Example: Sort 48, 1, 6, 23, 37, 19 ,21

Radix Sort Example

◆ Strategy is to use 2 bucket arrays, each of size 7 (7 is the *radix*)

◆ Based on observation that every k in [0,48] can be written:
$$k = 7q + r \text{ where } 0 <= q < 7, \ 0 <= r < 7$$

◆ <u>Procedure:</u>

- Pass #1: Scan initial array and place values the "remainders" bucket r[] – put x in r[i] if x % 7 = i. (Need to assume bucket array consists of lists)

- Pass #2: Scan r[], reading from front of each list to back, and place values in the "quotients" bucket q[] – put x in q[i] if x/7 = i.

- Output: Scan q[], again reading lists front to back

Analysis of RadixSort

- Running time is $O(d(n+r))$ where
  - d = # bucket arrays (in example d=2)
  - n = size of initial array (example: 7)
  - r = size of each bucket array (ex: 7)
- RadixSort runs in $O(n)$ whenever the number of bucket arrays is $O(1)$, and size of each bucket array is $O(n)$.

Optional: Using 3 Bucket Arrays

◆ Generalizing previous example, we observe that whenever k belongs to [0, 342] (note $342 = 7^3 - 1$), then we can write

$$k = 49q_1 + 7q_2 + r$$

where $0 \leq q_1 < 7$, $0 \leq q_2 < 7$, $0 \leq r < 7$.

Example: $340 = 49*6 + 7*6 + 4$

Therefore to handle sorting approx 7 elements in the range [0, 342], create bucket arrays r[], q_1[], q_2[], and compute,for each x in input array, values x%7, (x/7)%7, x/49, respectively

What Was That Lower Bound Again?

◆ We established a lower bound of $\Omega(n\log n)$ on comparison-based sorting algorithms. Do BucketSort and RadixSort disprove our earlier findings?

What Was That Lower Bound Again?

- We established a lower bound of $\Omega(n\log n)$ on comparison-based sorting algorithms. Do BucketSort and RadixSort disprove our earlier findings?

- *Resolution:* Neither of these is comparison-based.

Main Point

A decision-tree argument shows that comparison-based sorting algorithms can perform no better than $\Theta(n\log n)$. However, BucketSort is an example of a sorting algorithm that runs in $O(n)$. This is possible only because BucketSort does not rely primarily on comparisons in order to perform sorting. This phenomenon illustrates two points from SCI. First, to solve a problem, often the best approach is to bring a new element to the situation (in this case, bucket arrays); this is the Principle of the Second Element. The second point is that different laws of nature are applicable at different levels of creation. Deeper levels are governed by more comprehensive and unified laws of nature.

Connecting the Parts of Knowledge
With The Wholeness of Knowledge

Transcending The Lower Bound On Comparison-Based Algorithms

1. Comparison-based sorting algorithms can achieve a worst-case running time of $\Theta(n \log n)$, but can do no better.

2. Under certain conditions on the input, Bucket Sort and Radix Sort can sort in $O(n)$ steps, even in the worst case. The $n \log n$ bound does not apply because these algorithms are not comparison-based.

3. *Transcendental Consciousness* is the field of all possibilities and of pure orderliness. Contact with this field brings to light new possibilities and leads to spontaneous orderliness in all aspects of life.

4. *Impulses Within The Transcendental Field*. The organizing power of pure knowledge is the lively expression of the Transcendent, giving rise to all expressions of intelligence.

5. *Wholeness Moving Within Itself*. In Unity Consciousness, the organizing dynamics at the source of creation are appreciated as an expression of one's own Self.