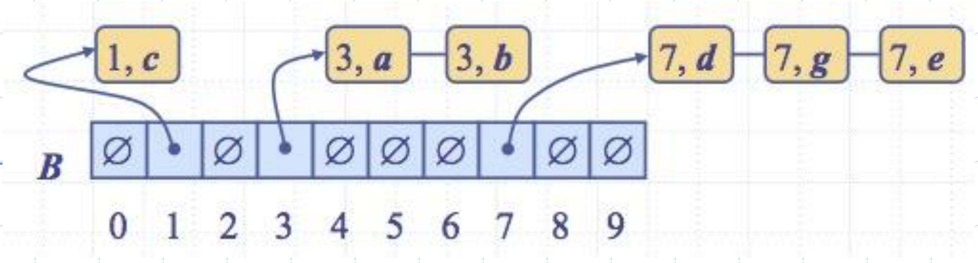
Lesson 7

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.

Road Map for Lower Bound on Comparison Based Sorting

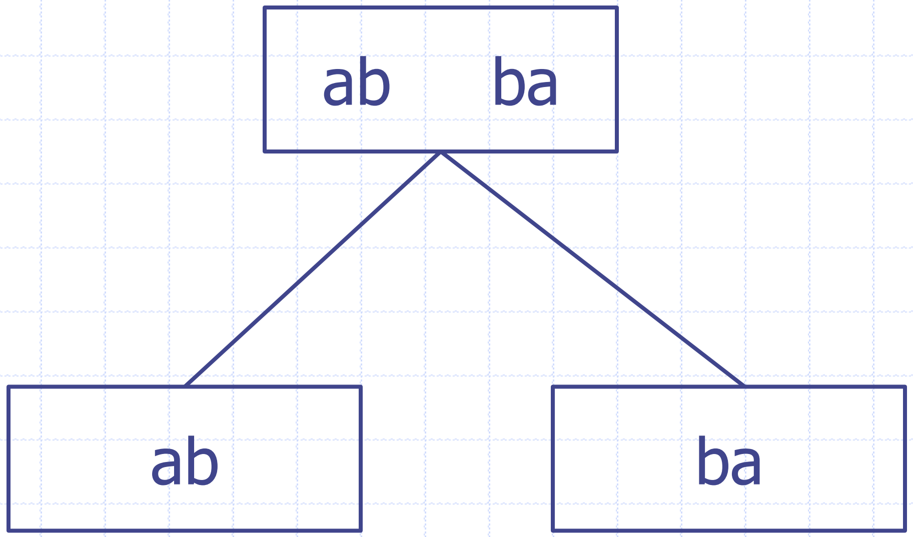
1. Given n, there are n! permutations.
2. Therefore any decision tree to sort n items will have n! leaves.
3. Since the decision tree is a binary tree, it must at least have log (n!) height.
4. Since the height of the decision tree is the same as the number of comparisons, any comparison based sorting algorithm must perform log (n!) comparisons.
5. log (n!) is theta(n log n).
6. Thus lower bound for the comparison based sorting algorithms is big-Omega(n log n)

Sample Decision Tree

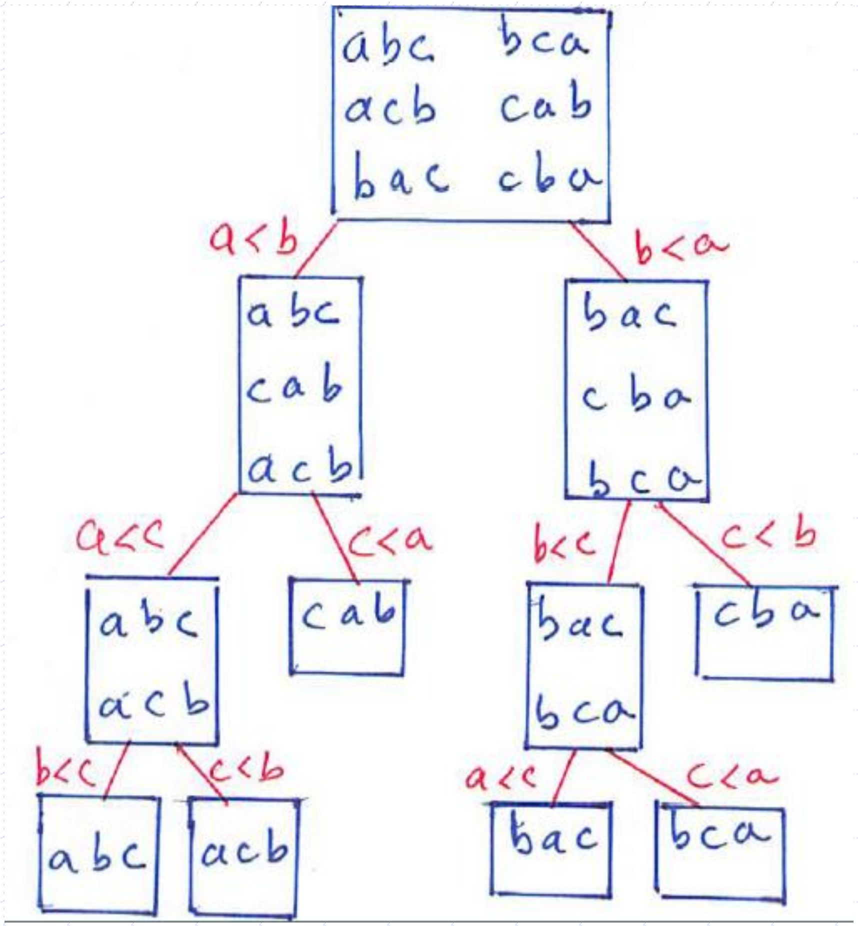
- Every comparison-based sorting algorithm, applied to a given array, can be represented by a decision tree
- To illustrate the technique, we use a simple but typical example: sorting 2, 3 and 4 element array of distinct values *a,b,c,d.*

Decision Tree for Sorting 2 Elements

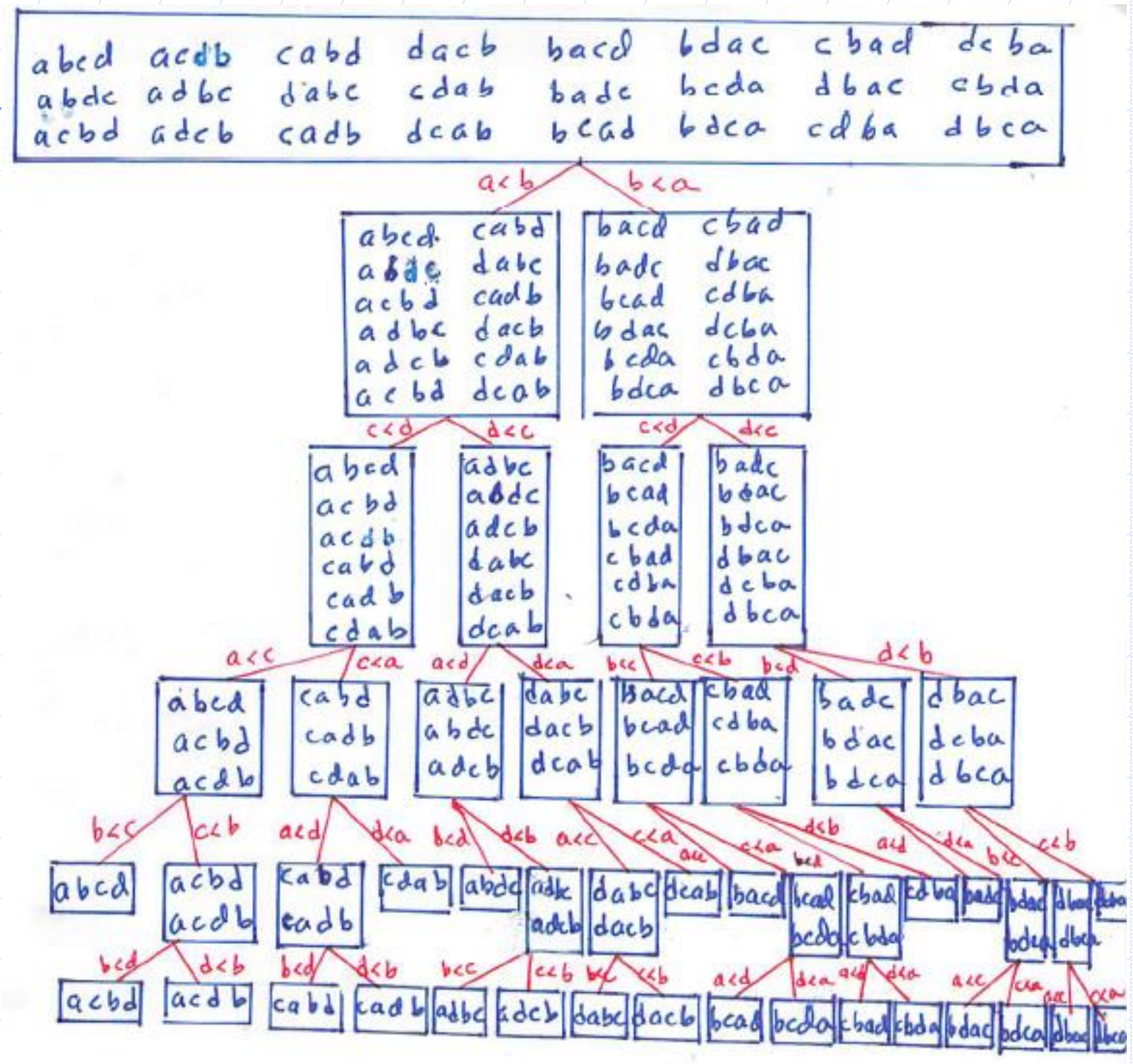
```
Read a, b
if (a < b)
    print(a, b)
else
    print(b, a)
```



Decision Tree for Sorting 3 Elements



Decision Tree for Sorting 4 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 three (as in slide 4 or four as in slide 5) distinct items could be put in sorted order; therefore, the leaves represent all possible arrangements of three (or four) distinct 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

Mathematical Observations

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

- Lemma 1. $L \leq 2^h$.

Use Lemma 1 to prove the next important fact:

- Lemma 2. $h \geq \lceil \log L \rceil$.

- 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]

Establishing the Lower Bound

Summary: Suppose we have a decision tree T for a sorting algorithm running on input of size n. Then T has n! leaves. So the height of T is at least $\lceil \log n! \rceil$ so some leaf in T has depth at least $\lceil \log n! \rceil$. Therefore, in the worst case, at least $\lceil \log n! \rceil$ comparisons are performed, so running time is $\Omega(\log n!)$. We determine the complexity class of log n!

The Computation of log n!

$$\begin{aligned}
\log n! &= \log(1 \cdot 2 \cdot 3 \cdot \ldots \cdot n) \\
&= \log n + \log(n-1) + \log(n-2) + \ldots + \log 2 + \log 1 \\
&\geq \log n + \log(n-1) + \ldots + \log \frac{n}{2} \\
&\geq \sum_{i=1}^{n/2} \log \frac{n}{2} \\
&= \frac{n}{2} \cdot \log \frac{n}{2} \\
&= \frac{n}{2} \cdot (\log n - 1) \\
&= \frac{n}{2} \cdot \log n - \frac{n}{2} \\
&= \Omega(n \log n)
\end{aligned}$$

Proof of Lemma 1

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

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 $\leq 2^d$ leaves.

Proof of Lemma 2

Lemma 2. A binary tree T with L leaves and height h satisfies $h \geq \log L$; in particular, $h \geq \lceil \log L \rceil$

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 a_j into next available slot in B.

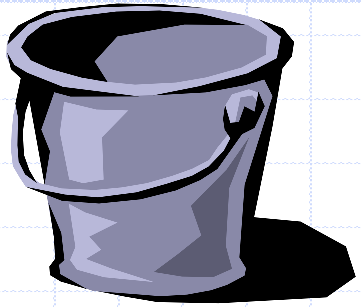
Analysis of Bucket Sort

- 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



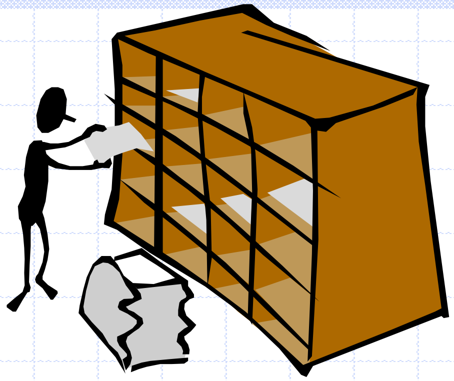
- Given n integers $a_1, a_2, \ldots, a_n$ in the range [0, m-1], possibly with duplicates, and n matched objects $o_1, o_2, \ldots, o_n$. We sort array A whose elements are pairs $(a_1,o_1), (a_2,o_2),\ldots,(a_n,o_n)$.
- To handle duplicates in this case, array bucket[] now consist of m *lists* rather 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 $B[a_i]$

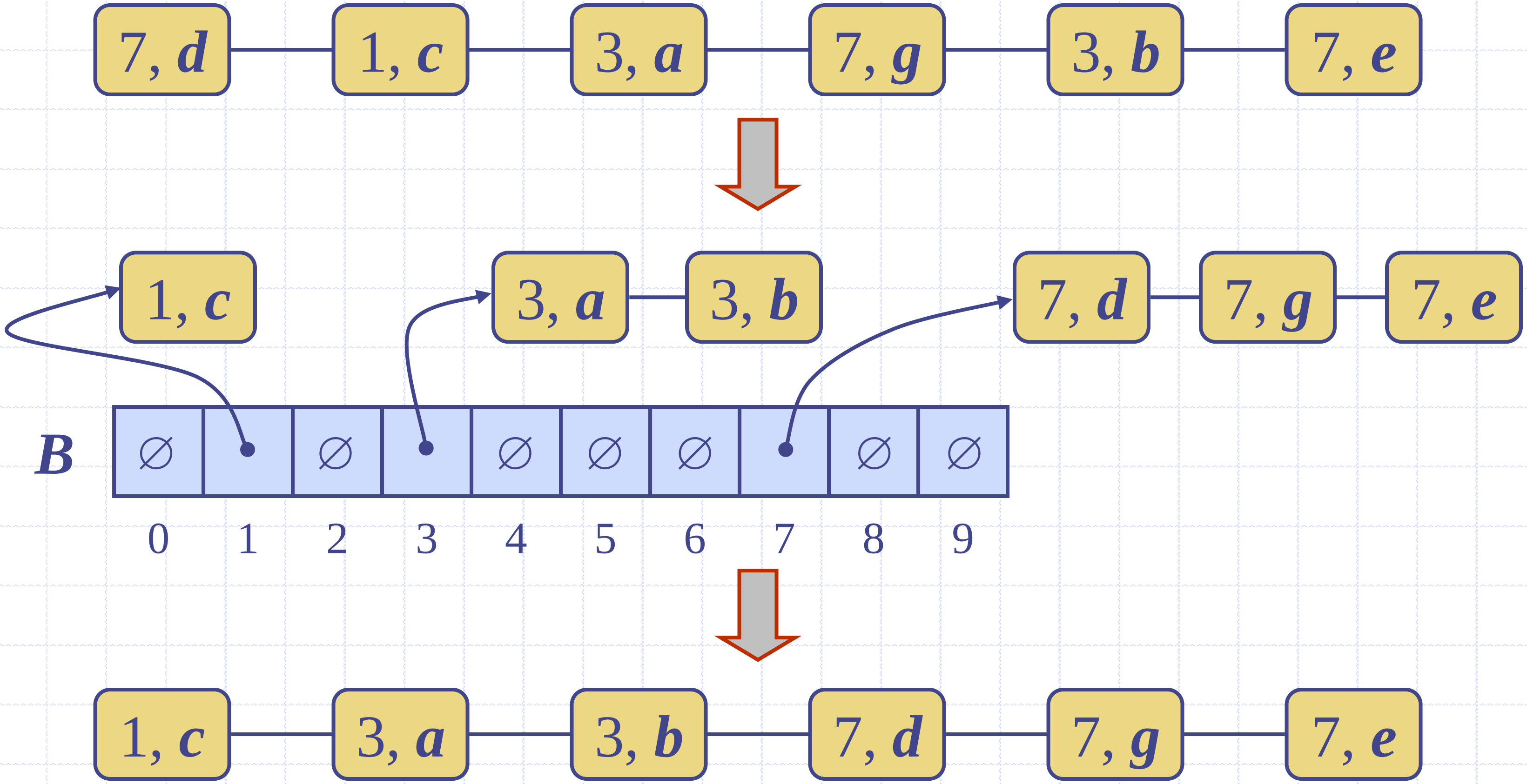
  Phase 2: For $j = 0, \ldots, m-1$, copy the items of bucket $bucket[j]$ to the end of sequence B.
- As before, running time is O(n+m). Notice the algorithm is stable because we always add newly scanned pairs to end of the list in the appropriate bucket

```
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 ← array of N empty lists
    for i ← 0 to n – 1
        (k, o) ← A[i]
        bucket[k].insertLast((k, o))
    for j ← 0 to m – 1
        while ¬bucket[j].isEmpty()
            (k, o) ← bucket[j].removeFirst()
            B.insertLast((k, o))
```

Example



- Key range [0, 9]

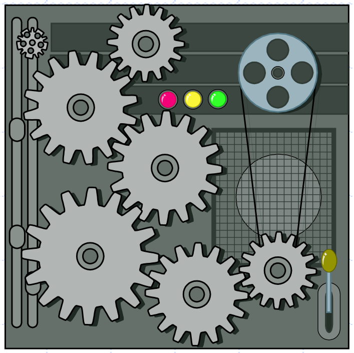


Expanding the Context

- 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



- Radix-sort is a generalization of BucketSort that uses multiple bucket arrays
- Example: Sort 48, 1, 6, 23, 37, 19 ,21
- BucketSort doesn't work because range is too big – would run in $\Omega(n^2)$

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:
  $k = 7q + r$ where $0 \leq q < 7$, $0 \leq r < 7$
- <u>Procedure:</u>
  - Pass #1: Scan initial array and place values in 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

Radix Sort Example

Keys: 48 1 6 23 37 19 21

Key % 7

0 -> 21 1 -> 1 2 -> 23 -> 37

5 -> 19 6 -> 48 -> 6

Key / 7

0 -> 1 -> 6 2 -> 19 3 -> 21 -> 23

5 -> 37 6 -> 48

Sorted order

1 6 19 21 23 37 48

Radix Sort Example

Keys: 300, 215, 110, 115, 210, 315, 310, 200, 100

Key % 7

0 -> 210 -> 315 2 -> 310 -> 100 3 -> 115

4 -> 200 5 -> 215 -> 110 6 -> 300

(Key / 7) % 7

0 -> 100 -> 200 -> 300 1 -> 110

2 -> 210 -> 310 -> 115 -> 215 3 -> 315

Key / 49 OR ((Key / 7) / 7)

2 -> 100 -> 110 -> 115 4 -> 200 -> 210 -> 215

6 -> 300 -> 310 -> 315

Sorted order

100 110 115 200 210 215 300 310 315

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.