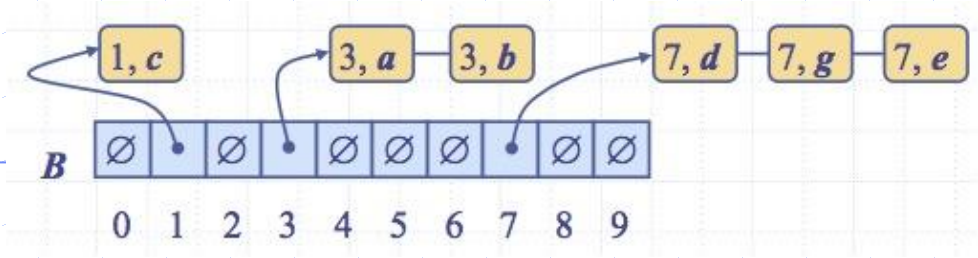
Lesson 7
Sorting in Linear Time

: *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.

Sorting So Far

- Insertion sort:
  - Easy to code
  - Fast on small inputs (less than ~50 elements)
  - Fast on nearly-sorted inputs
  - $O(n^2)$ worst case
  - $O(n^2)$ average (equally-likely inputs) case
  - $O(n^2)$ reverse-sorted case

Sorting So Far

- Merge sort:
  - Divide-and-conquer:
    - Split array in half
    - Recursively sort subarrays
    - Linear-time merge step
  - O(n lg n) worst case
  - Doesn’t sort in place

Sorting So Far

- Heap sort (will cover later):
  - Uses the very useful heap data structure
    - Complete binary tree
    - Heap property: parent key > children's keys
  - O(n lg n) worst case
  - Sorts in place
  - Fair amount of shuffling memory around

Sorting So Far

- Quick sort:
  - Divide-and-conquer:
    - Partition array into two subarrays, recursively sort
    - All of first subarray < all of second subarray
    - No merge step needed!
  - O(n lg n) average case
  - Fast in practice (2-3 times faster than Merge sort)
  - $O(n^2)$ worst case – but probability of this case is very low (randomization)

How Fast Can We Sort?

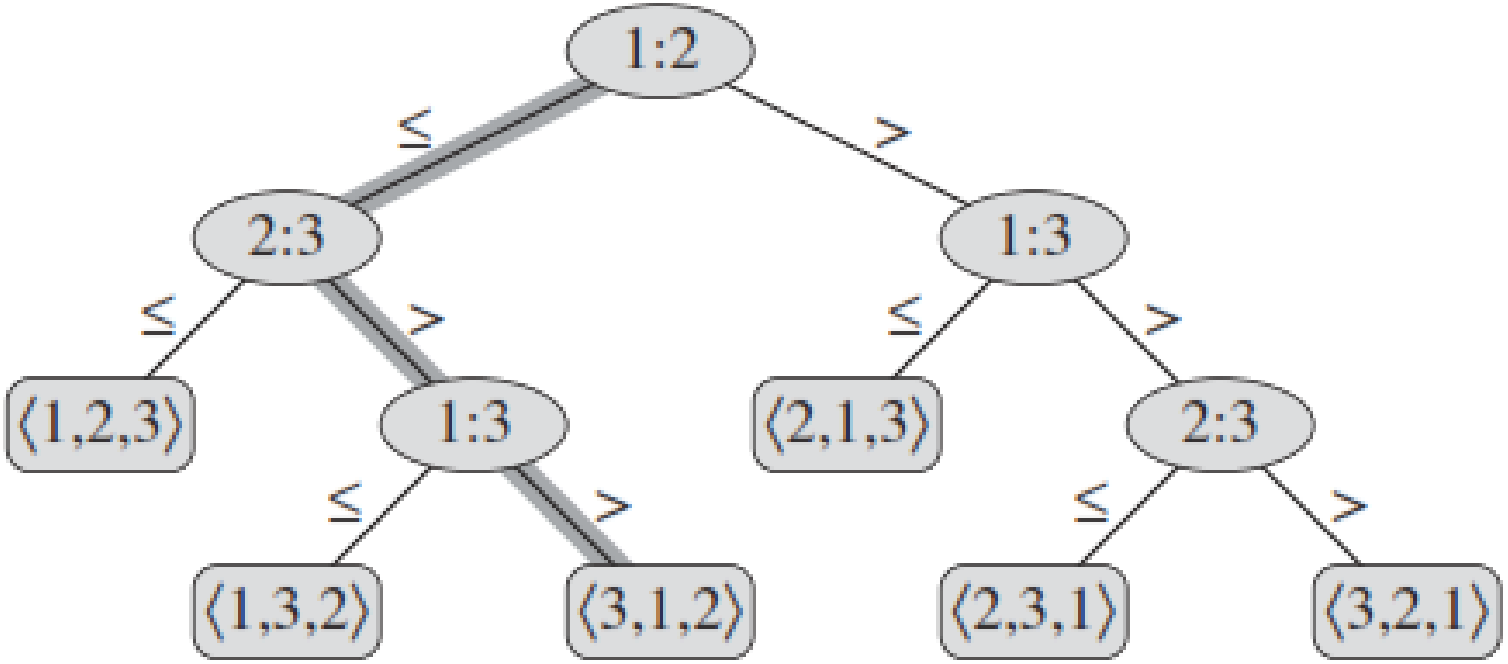
- We will provide a lower bound, then beat it
  - *How do you suppose we'll beat it?*
- First, an observation: all of the sorting algorithms so far are *comparison sorts*
  - The only operation used to gain ordering information about a sequence is the pairwise comparison of two elements
  - Theorem: all comparison sorts are $\Omega(n \lg n)$
    - Note - A comparison sort must do O(n) comparisons

Decision Trees

- *Decision trees* provide an abstraction of comparison sorts
  - A decision tree represents the comparisons made by a comparison sort. Every thing else ignored

    >> Diagram in NEXT PAGE
- *What do the leaves represent?*
- *How many leaves must there be?*

Decision Trees



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 (in this case) distinct items could be put in sorted order; therefore, the leaves represent all possible arrangements of 3 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

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 Ω(nlog n) [*we are excluding O(n^2) Insertion, Bubble and Selection sort; these are good for small size array*]
- Using the technique of *decision trees* we can show that nlog n is the best that can be done for comparison-based sorting algorithms

Lower Bound For Comparison Sorting

- Theorem: Any decision tree that sorts n elements has height $\Omega(n \lg n)$ [as mentioned]
- *What's the minimum # of leaves?*
- *What's the maximum # of leaves of a binary tree of height h?*
- Clearly the minimum # of leaves is less than or equal to the maximum # of leaves

Lower Bound For Comparison Sorting

- We know there are n! leaves. We also know that a tree with height h has max 2^h leaves. So we have… $n! \leq 2^h$
- Taking logarithms: $\lg(n!) \leq h$
- Stirling's approximation tells us: $n! > \left(\frac{n}{e}\right)^n$
- Thus: $h \geq \lg\left(\frac{n}{e}\right)^n$

Lower Bound For Comparison Sorting

- So we have

$$h \geq \lg\left(\frac{n}{e}\right)^n$$

$$= n \lg n - n \lg e$$

$$= \Omega(n \lg n)$$

- Thus the minimum height of a decision tree is $\Omega(n \lg n)$

Sorting In Linear Time

- But the name of this lecture is "Sorting in linear time"!
  - *How can we do better than* $\Omega(n \lg n)$*?*

Sorting In Linear Time

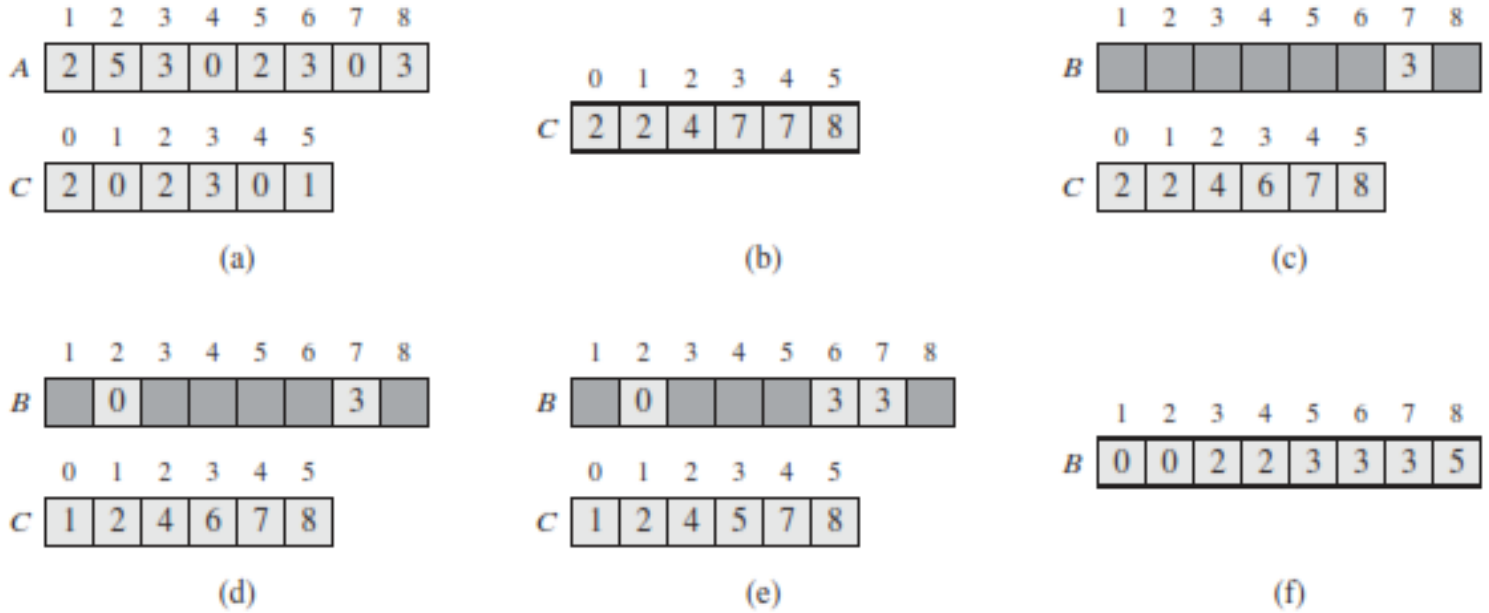
- Counting sort
  - No comparisons between elements!
  - ***But****…*depends on assumption about the numbers being sorted
    - We assume numbers are in the range *1.. k*
  - The algorithm:
    - Input: A[1..*n*], where A[j] $\in$ {1, 2, 3, …, *k*}
    - Output: B[1..*n*], sorted (notice: not sorting in place)
    - Also: Array C[1..*k*] for auxiliary storage

Counting Sort

```
1       CountingSort(A, B, k)
2               for i=1 to k
3                       C[i]= 0;
4               for j=1 to n
5                       C[A[j]] += 1;
6               for i=1 to k
7                       C[i] = C[i] + C[i-1];
8               for j=n down to 1
9                       B[C[A[j]]] = A[j];
10                      C[A[j]] -= 1;
```

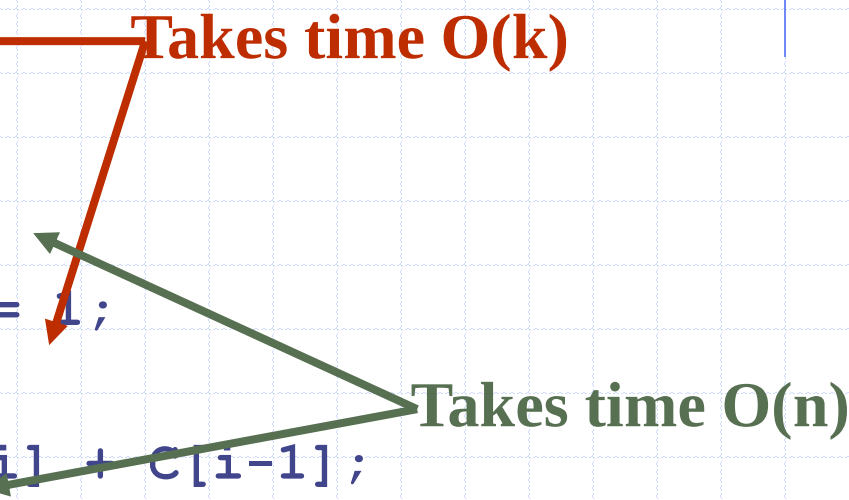
Counting Sort - Example

1



Counting Sort

```
1     CountingSort(A, B, k)
2           for i=1 to k
3                 C[i]= 0;
4           for j=1 to n
5                 C[A[j]] += 1;
6           for i=1 to k
7                 C[i] = C[i] + C[i-1];
8           for j=n downto 1
9                 B[C[A[j]]] = A[j];
10                C[A[j]] -= 1;
```



What will be the running time?

Counting Sort

- Total time: $O(n + k)$
  - Usually, $k = O(n)$
  - Thus counting sort runs in $O(n)$ time
- But sorting is $\Omega(n \lg n)$!
  - No contradiction--this is not a comparison sort (in fact, there are *no* comparisons at all!)
  - Notice that this algorithm is *stable* i.e. preserve order when there is a tie (Decrementing C[i] in Line 10)

Counting Sort

- Cool! *Why don't we always use counting sort?*
- Because it depends on range *k* of elements
- *Could we use counting sort to sort 32 bit integers? Why or why not?*
- Answer: no, *k* too large (2^{32} = 4,294,967,296)

[not in place sorting!]

Counting Sort

- *How did IBM get rich originally?*
- Answer: punched card readers for census tabulation in early 1900's.
  - In particular, a *card sorter* that could sort cards into different bins
    - Each column can be punched in 12 places
    - Decimal digits use 10 places
  - Problem: only one column can be sorted on at a time

Radix Sort

- Intuitively, you might sort on the most significant digit, then the second msd, etc.
- Problem: lots of intermediate piles of cards (read: scratch arrays) to keep track of
- Key idea: sort the *least* significant digit first

Radix Sort

- Key idea: sort the *least* significant digit first

```
RadixSort(A, d)
    for i=1 to d
        StableSort(A) on digit i
```

  - Example: Fig 9.3

Radix Sort

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

```
RADIX-SORT(A, d)
1  for i = 1 to d
2          use a stable sort to sort array A on digit i
```

Radix Sort

- *What sort will we use to sort on digits?*
- Counting sort is obvious choice:
  - Sort n numbers on digits that range from $1..k$
  - Time: $O(n + k)$
- Each pass over n numbers with d digits takes time $O(n+k)$, so total time $O(dn+dk)$
  - When d is constant and k=$O(n)$, takes $O(n)$ time

Radix Sort

- Problem: sort 1 million 64-bit numbers
  - Treat as four-digit radix 2^{16} numbers
  - Can sort in just four passes with radix sort!
- Compares well with typical O(n lg n) comparison sort
  - Requires approx lg n = 20 operations per number being sorted
- *Does it sort in place with Counting Sort?*

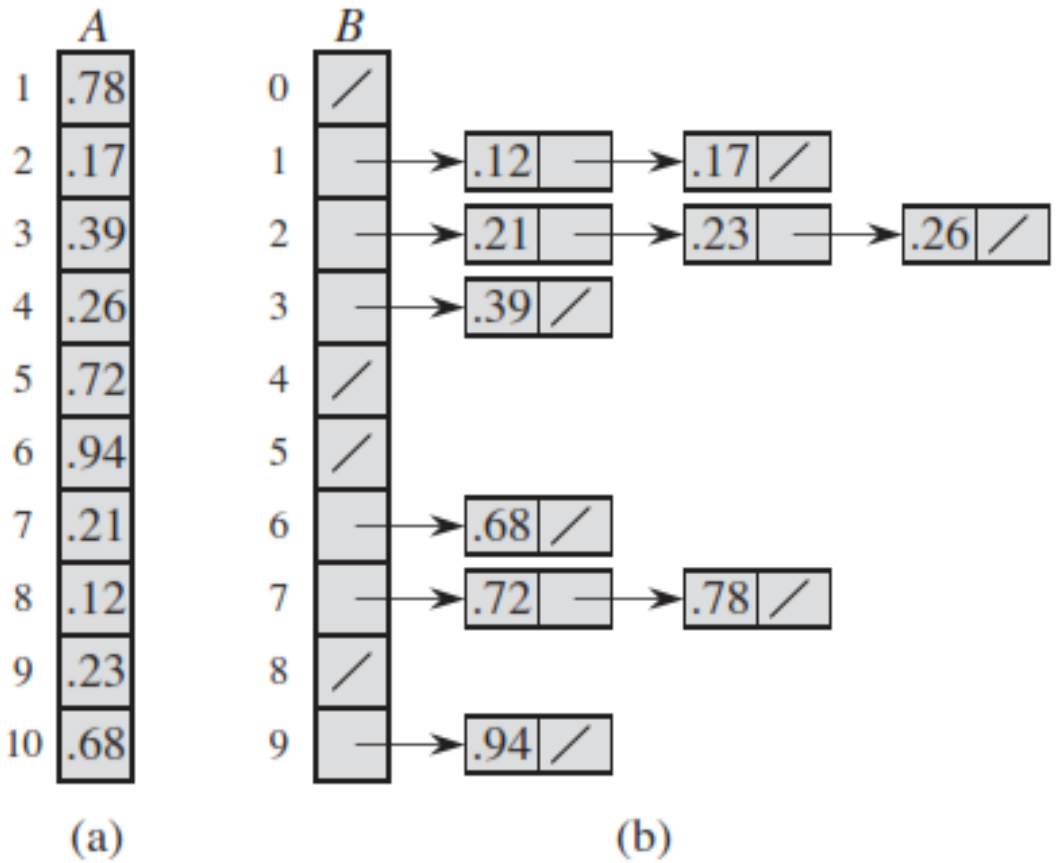
Radix Sort

- In general, radix sort based on counting sort is
  - Fast
  - Asymptotically fast (i.e., O(*n*))
  - Simple to code
  - A good choice

Bucket Sort

- Key Idea
  - Divide into n equal sized buckets
  - Distribute the n input numbers into the buckets
  - Then sort the numbers in each bucket
  - Example – Next page

Bucket Sort


A
1 .78
2 .17
3 .39
4 .26
5 .72
6 .94
7 .21
8 .12
9 .23
10 .68
(a)
B
0
1 .12 .17
2 .21 .23 .26
3 .39
4
5
6 .68
7 .72 .78
8
9 .94
(b)

Bucket Sort

```
BUCKET-SORT(A)
1  let B[0..n − 1] be a new array
2  n = A.length
3  for i = 0 to n − 1
4       make B[i] an empty list
5  for i = 1 to n
6       insert A[i] into list B[⌊n A[i]⌋]
7  for i = 0 to n − 1
8       sort list B[i] with insertion sort
9  concatenate the lists B[0], B[1], . . . , B[n − 1] together in order
```

Bucket Sort

- Running time

(the 2nd term is for the Insertion sort)

$$T(n) = \Theta(n) + \sum_{i=0}^{n-1} O(n_i^2) .$$

Bucket Sort

- Running time – 2nd term

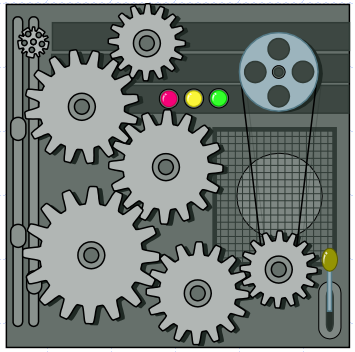
$$
\begin{aligned}
\mathrm{E}[T(n)] &= \mathrm{E}\left[\Theta(n) + \sum_{i=0}^{n-1} O(n_i^2)\right] \\
&= \Theta(n) + \sum_{i=0}^{n-1} \mathrm{E}\left[O(n_i^2)\right] \quad \text{(by linearity of expectation)} \\
&= \Theta(n) + \sum_{i=0}^{n-1} O\left(\mathrm{E}\left[n_i^2\right]\right) \quad \text{(by equation (C.22))} \; . \qquad (8.1)
\end{aligned}
$$

We claim that

$$\mathrm{E}\left[n_i^2\right] = 2 - 1/n \qquad (8.2)$$

>> Hence Linear

Radix-Sort - Revisited



- Radix-sort can be considered as a generalization of BucketSort that uses multiple bucket arrays
  - E.g. use of 2 Bucket arrays in Lab 6
- 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 <= q < 7, 0 <= r < 7
- Procedure:
  - 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

Main Point

A decision-tree argument shows that comparison-based sorting algorithms can perform no better than Θ(nlog 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.