Lecture 6: Conclusions on Sorting

Knowledge Has Organizing Power

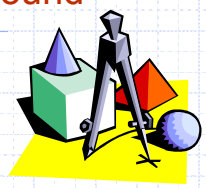
Wholeness Statement

We can prove that the lower bound on sorting by key comparisons in the best and worst cases is O(n log n). However, we can do better, i.e. linear time, through knowledge of the structure and distribution of keys. Knowledge has organizing power; pure knowledge has infinite organizing power.

Outline

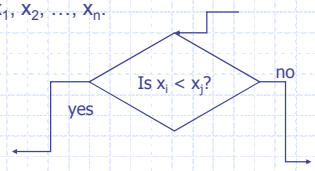
- Lower Bound on Comparison-Based Sorting (§4.4)
- Linear Time Sorting Algorithms (§4.5)
  - Bucket Sort
    - Text version of bucket sort
    - Usual bucket sort (Monday)
  - Radix Sort
- Divide-and-Conquer Analysis(§5.2)
  - Master Theorem for solving recurrence relations

Sorting Lower Bound



Comparison-Based Sorting (§ 4.4)

- Many sorting algorithms are comparison based.
  - They sort by making comparisons between pairs of objects
  - Examples: bubble-sort, selection-sort, insertion-sort, heap-sort, merge-sort, quick-sort, shell sort, ...
- Let us therefore derive a lower bound on the running time of any algorithm that uses comparisons to sort n elements, $x_1, x_2, \ldots, x_n$.

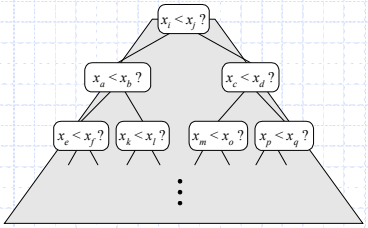


Definition of a Decision Tree

- Internal nodes correspond to key comparisons
  - Thus number of comparisons corresponds to the number of internal nodes
- Leaf nodes correspond to the resulting sorted sequence
- Left subtree shows the next comparison when $x < y$
- Right subtree shows the next comparison when $x \geq y$
- Make the tree as efficient as possible by
  - Removing nodes with single children
  - Removing any paths not followed

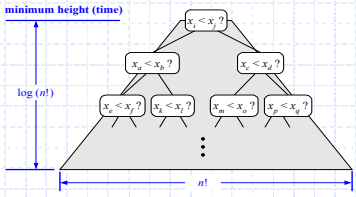
Counting Comparisons

- Each possible run of the algorithm corresponds to a root-to-leaf path in a **decision tree**



Decision Tree Height

- The height of the decision tree is a lower bound on the running time
- Every possible input permutation must lead to a separate leaf output.
  - If not, some input …4…5… would have same output ordering as …5…4…, which would be wrong.
- Since there are n!=1*2*…*n leaves, the height is at least log (n!)



Worst Case Lower Bound

- Number of nodes on the longest path
- i.e., the height h of the decision tree

$2^h \geq n!$ (number of leaf nodes)
$\log 2^h \geq \log n!$
$h \geq \log n!$
$h \geq \log n! \geq \log (n/2)^{n/2}$
$= n/2 \log n/2$
$= n/2 (\log n - \log 2)$
$= 1/2 (n \log n - n)$

- Thus h is $\Omega(n \log n)$

Average Behavior Lower Bound

- Assuming all inputs are equally likely, the average path length can also be shown to be $\Omega(n \log n)$
- Thus the average comparisons to sort by comparison of keys is $\Omega(n \log n)$
- Therefore, no sorting algorithm that sorts by comparison of keys can do substantially better than Heapsort, Quicksort, or Merge-sort

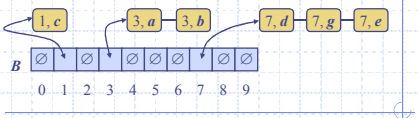
Main Point

1. Any algorithm that sorts n items by comparison of keys must do $\Omega(n \log n)$ comparisons in the worst and average case. Quicksort and Merge-sort come very close to realizing this lower bound; thus $\Omega(n \log n)$ is close to being a maximal lower bound. In enlightenment, one realizes the Absolute in the relative for maximal power to fulfill one's goals.

Linear Time Sorting Algorithms

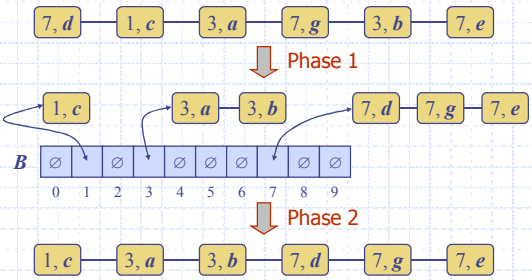
Pure Knowledge Has Infinite Organizing Power

Bucket-Sort and Radix-Sort



Example

- Key range [0, 9]



Bucket Sort Algorithm in Text

- Note: the following algorithm uses Lists instead of Sequences (used in text) to emphasize that the *remove*() method has to run in O(1) time

Bucket-Sort (§ 4.5.1)

- Let S be a list containing 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 lists (buckets)
  - Phase 1: Empty list L by moving each item (k, o) into its bucket $B[k]$
  - Phase 2: For $i = 0, \ldots, N-1$, move the items of bucket $B[i]$ to the end of list L
- Analysis:
  - Phase 1 takes $O(n)$ time
  - Phase 2 takes $O(n + N)$ time

Bucket-sort takes $O(n + N)$ time

Algorithm *bucketSort*(L, N)
Input List L of (key, element) items with keys in the range $[0, N-1]$
Output List L sorted by increasing keys

```
B ← array of N empty lists
while ¬L.isEmpty() do
    (k, o) ← L.remove(L.first())
    B[k].insertLast((k, o))
for i ← 0 to N − 1 do
    while ¬B[i].isEmpty() do
        f ← B[i].first()
        (k, o) ← B[i].remove(f)
        L.insertLast((k, o))
```


Properties and Extensions

- Key-type Property
  - The keys are used as indices into an array and cannot be arbitrary objects
  - No external comparator

Extensions

- 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 50 U.S. 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 $B[r(k)]$

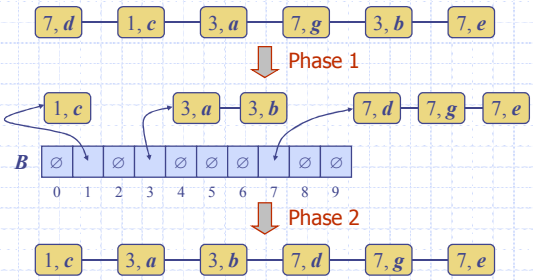
Lexicographic Sort

Stable Sorting

- **Stable** Sort Property
  - The relative order of any two items with the same key is preserved after the execution of the algorithm
- Not all sorting algorithms preserve this property

Example

- Key range [0, 9]



Lexicographic Order

- A d-tuple is a sequence of d keys $(k_1, k_2, \ldots, k_d)$, where key k_i is said to be the i-th dimension of the tuple
- Example:
  - The Cartesian coordinates of a point in space are a 3-tuple
- The lexicographic order of two d-tuples is recursively defined as follows

$$(x_1, x_2, \ldots, x_d) < (y_1, y_2, \ldots, y_d)$$
$$\Leftrightarrow$$
$$x_1 < y_1 \vee (x_1 = y_1 \wedge (x_2, \ldots, x_d) < (y_2, \ldots, y_d))$$

I.e., the tuples are compared by the first dimension, then by the second dimension, etc.

Lexicographic-Sort

- Let C_i be the comparator that compares two tuples by their i-th dimension
- Let *stableSort*(S, C) be a stable sorting algorithm that uses comparator C
- Lexicographic-sort sorts a sequence of d-tuples in lexicographic order by executing d times algorithm *stableSort*, one per dimension
- Lexicographic-sort runs in $O(dT(n))$ time, where $T(n)$ is the running time of *stableSort*

Algorithm *lexicographicSort*(S)
Input sequence S of d-tuples
Output sequence S sorted in lexicographic order

for $i \leftarrow d$ **downto** 1
  stableSort(S, C_i)

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)

- This kind of ordering is sometimes used when records are keyed by multiple fields
- Question:
  - What is the meaning of “radix”?

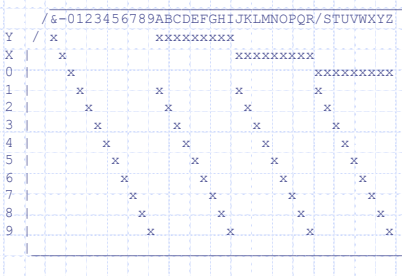
Radix

- The base of a number system,
  - e.g., base 10 in decimal numbers or base 2 in binary numbers
  - aka radix 10 or radix 2

Radix Sort

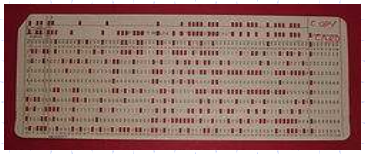
Radix Sort

- The algorithm used by card sorting machines (now found only in museums)
  - Cards were organized into 80 columns such that a hole could be punched in 12 possible slots per column
  - The sorter was mechanically "programmed" to examine a given column of each card and distribute the card into one of 12 bins
- What if we need to sort more than one column?
  - Radix sort solves the problem
  - Requires a stable sorter (defined below)



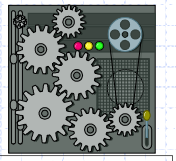
Punch Cards

From a Fortran program: Z(1) = Y + W(1)



Binary punch card

Radix-Sort (§ 4.5.2)



- Radix-sort is a specialization of lexicographic-sort that uses bucket-sort as the stable sorting algorithm in each dimension
- Radix-sort is applicable to tuples where the keys in each dimension i are integers in the range $[0, N-1]$
- Radix-sort runs in time $O(d(n+N))$

Algorithm *radixSort*(S, N)
Input sequence S of d-tuples such that $(0, \ldots, 0) \le (x_1, \ldots, x_d)$ and $(x_1, \ldots, x_d) \le (N-1, \ldots, N-1)$ for each tuple $(x_1, \ldots, x_d)$ in S
Output sequence S sorted in lexicographic order

for $i \leftarrow d$ **downto** 1
  replace the key k of each item (k, x) of S with dimension x_i of x
  bucketSort(S, N)

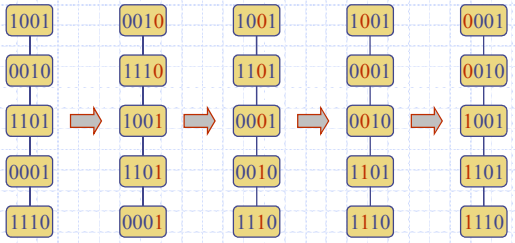
Radix-Sort for Binary Numbers

- Consider a sequence of n b-bit integers
  $x = x_{b-1} \ldots x_1 x_0$
- We represent each element as a b-tuple of integers in the range $[0, 1]$ and apply radix-sort with $N = 2$
- This application of the radix-sort algorithm runs in $O(bn)$ time
- For example, we can sort a sequence of 32-bit integers in linear time

Algorithm *binaryRadixSort*(S)
Input sequence S of b-bit integers
Output sequence S sorted
  replace each element x of S with the item $(0, x)$
for $i \leftarrow 0$ **to** $b-1$
  replace the key k of each item (k, x) of S with bit x_i of x
  bucketSort$(S, 2)$

Example

- Sorting a sequence of 4-bit integers



Main Point

2. A radix-sort does successive bucket sorts, one for each "digit" in the key beginning with the least significant digit going up to the most significant; it has linear running time.
   The nature of life is to grow and progress; Natural Law unfolds in perfectly orderly sequence that gives rise to the universe, all of manifest creation.

Summary of Sorting Algorithms (§4.6)

Algorithm	Time	Notes (pros & cons)
insertion-sort		
merge-sort		
quick-sort		
heap-sort		
bucket-sort radix-sort		

Summary of Sorting Algorithms (§4.6)

Algorithm	Time	Notes (pros & cons)
insertion-sort	$O(n^2)$ or $O(n+k)$	excellent for small inputs fast for 'almost' sorted inputs
merge-sort	$O(n \log n)$	excels in sequential access for huge data sets
quick-sort	$O(n \log n)$ expected	in-place, randomized excellent generalized sort
heap-sort	$O(n \log n)$	in-place fastest for in-memory
bucket-sort radix-sort	$O(n+N)$ $O(d(n+N))$	if integer keys & keys known faster than quick-sort

Recurrence Equations (continued)

Self-Referral

Recurrence Equations

- An equation or inequality that describes a function in terms of its value on smaller inputs
- AKA Recurrence Relations
- Why do we care about solving recurrence equations?
- Four ways for solving
  - Iterative Substitution method
  - Recursion-tree method
  - Guess-and-test method
  - Master method

Outline and Reading

- Recurrence Equations (§5.2.1)
  - The master method

Master Method

- Many divide-and-conquer recurrence equations have the form:

$$T(n) = \begin{cases} c & \text{if } n < d \\ aT(n/b) + f(n) & \text{if } n \geq d \end{cases}$$

- The Master Theorem: for some $\varepsilon > 0$
  1. if $f(n)$ is $O(n^{\log_b a - \varepsilon})$, then $T(n)$ is $\Theta(n^{\log_b a})$
  2. if $f(n)$ is $\Theta(n^{\log_b a} \log^k n)$, then $T(n)$ is $\Theta(n^{\log_b a} \log^{k+1} n)$
  3. if $f(n)$ is $\Omega(n^{\log_b a + \varepsilon})$, then $T(n)$ is $\Theta(f(n))$, provided $af(n/b) \leq \delta f(n)$ for some $\delta < 1$.

Iterative "Proof" of the Master Theorem

- Using iterative substitution, let us see if we can find a pattern:

$$\begin{aligned} T(n) &= aT(n/b) + f(n) \\ &= a(aT(n/b^2) + f(n/b)) + f(n) \\ &= a^2T(n/b^2) + af(n/b) + f(n) \\ &= a^3T(n/b^3) + a^2f(n/b^2) + af(n/b) + f(n) \\ &= \ldots \\ &= a^{\log_b n}T(1) + \sum_{i=0}^{(\log_b n)-1} a^i f(n/b^i) \\ &= n^{\log_b a}T(1) + \sum_{i=0}^{(\log_b n)-1} a^i f(n/b^i) \end{aligned}$$

Iterative "Proof" of the Master Theorem

- The Master Theorem distinguishes the three cases as
  - The first term is dominant
  - Each part of the summation is equally dominant
  - The summation is a geometric series

Master Method, Example 1

- The form: $T(n) = \begin{cases} c & \text{if } n < d \\ aT(n/b) + f(n) & \text{if } n \geq d \end{cases}$
- The Master Theorem:
  1. if $f(n)$ is $O(n^{\log_b a - \varepsilon})$, then $T(n)$ is $\Theta(n^{\log_b a})$
  2. if $f(n)$ is $\Theta(n^{\log_b a} \log^k n)$, then $T(n)$ is $\Theta(n^{\log_b a} \log^{k+1} n)$
  3. if $f(n)$ is $\Omega(n^{\log_b a + \varepsilon})$, then $T(n)$ is $\Theta(f(n))$, provided $af(n/b) \leq \delta f(n)$ for some $\delta < 1$.
- Example:

$$T(n) = 4T(n/2) + n$$

Solution: $\log_b a = 2$, so case 1 says T(n) is $\Theta(n^2)$.

Master Method, Example 2

- The form: $T(n) = \begin{cases} c & \text{if } n < d \\ aT(n/b) + f(n) & \text{if } n \geq d \end{cases}$
- The Master Theorem:
  1. if $f(n)$ is $O(n^{\log_b a - \varepsilon})$, then $T(n)$ is $\Theta(n^{\log_b a})$
  2. if $f(n)$ is $\Theta(n^{\log_b a} \log^k n)$, then $T(n)$ is $\Theta(n^{\log_b a} \log^{k+1} n)$
  3. if $f(n)$ is $\Omega(n^{\log_b a + \varepsilon})$, then $T(n)$ is $\Theta(f(n))$, provided $af(n/b) \leq \delta f(n)$ for some $\delta < 1$.
- Example:

$$T(n) = 2T(n/2) + n \log n$$

Solution: $\log_b a = 1$, so case 2 says T(n) is $\Theta(n \log^2 n)$.

Master Method, Example 3

- The form: $T(n) = \begin{cases} c & \text{if } n < d \\ aT(n/b) + f(n) & \text{if } n \geq d \end{cases}$
- The Master Theorem:
  1. if $f(n)$ is $O(n^{\log_b a - \varepsilon})$, then $T(n)$ is $\Theta(n^{\log_b a})$
  2. if $f(n)$ is $\Theta(n^{\log_b a} \log^k n)$, then $T(n)$ is $\Theta(n^{\log_b a} \log^{k+1} n)$
  3. if $f(n)$ is $\Omega(n^{\log_b a + \varepsilon})$, then $T(n)$ is $\Theta(f(n))$, provided $af(n/b) \leq \delta f(n)$ for some $\delta < 1$.
- Example:

$$T(n) = T(n/3) + n \log n$$

Solution: $\log_b a = 0$, so case 3 says T(n) is $\Theta(n \log n)$.

Master Method, Example 4

- The form: $T(n) = \begin{cases} c & \text{if } n < d \\ aT(n/b) + f(n) & \text{if } n \geq d \end{cases}$
- The Master Theorem:
  1. if $f(n)$ is $O(n^{\log_b a - \varepsilon})$, then $T(n)$ is $\Theta(n^{\log_b a})$
  2. if $f(n)$ is $\Theta(n^{\log_b a} \log^k n)$, then $T(n)$ is $\Theta(n^{\log_b a} \log^{k+1} n)$
  3. if $f(n)$ is $\Omega(n^{\log_b a + \varepsilon})$, then $T(n)$ is $\Theta(f(n))$, provided $af(n/b) \leq \delta f(n)$ for some $\delta < 1$.
- Example:

$$T(n) = 8T(n/2) + n^2$$

Solution: $\log_b a = 3$, so case 1 says T(n) is $\Theta(n^3)$.

Master Method, Example 5

- The form: $T(n) = \begin{cases} c & \text{if } n < d \\ aT(n/b) + f(n) & \text{if } n \geq d \end{cases}$
- The Master Theorem:
  1. if $f(n)$ is $O(n^{\log_b a - \varepsilon})$, then $T(n)$ is $\Theta(n^{\log_b a})$
  2. if $f(n)$ is $\Theta(n^{\log_b a} \log^k n)$, then $T(n)$ is $\Theta(n^{\log_b a} \log^{k+1} n)$
  3. if $f(n)$ is $\Omega(n^{\log_b a + \varepsilon})$, then $T(n)$ is $\Theta(f(n))$, provided $af(n/b) \leq \delta f(n)$ for some $\delta < 1$.
- Example:

$$T(n) = 9T(n/3) + n^3$$

Solution: $\log_b a = 2$, so case 3 says T(n) is $\Theta(n^3)$.

Master Method, Example 6

- The form: $T(n) = \begin{cases} c & \text{if } n < d \\ aT(n/b) + f(n) & \text{if } n \geq d \end{cases}$
- The Master Theorem:
  1. if $f(n)$ is $O(n^{\log_b a - \varepsilon})$, then $T(n)$ is $\Theta(n^{\log_b a})$
  2. if $f(n)$ is $\Theta(n^{\log_b a} \log^k n)$, then $T(n)$ is $\Theta(n^{\log_b a} \log^{k+1} n)$
  3. if $f(n)$ is $\Omega(n^{\log_b a + \varepsilon})$, then $T(n)$ is $\Theta(f(n))$, provided $af(n/b) \leq \delta f(n)$ for some $\delta < 1$.
- Example:

$$T(n) = T(n/2) + 1$$ (binary search)

Solution: $\log_b a = 0$, so case 2 says T(n) is $\Theta(\log n)$.

Master Method, Example 7

- The form: $T(n) = \begin{cases} c & \text{if } n < d \\ aT(n/b) + f(n) & \text{if } n \geq d \end{cases}$
- The Master Theorem:
  1. if $f(n)$ is $O(n^{\log_b a - \varepsilon})$, then $T(n)$ is $\Theta(n^{\log_b a})$
  2. if $f(n)$ is $\Theta(n^{\log_b a} \log^k n)$, then $T(n)$ is $\Theta(n^{\log_b a} \log^{k+1} n)$
  3. if $f(n)$ is $\Omega(n^{\log_b a + \varepsilon})$, then $T(n)$ is $\Theta(f(n))$, provided $af(n/b) \leq \delta f(n)$ for some $\delta < 1$.
- Example:

$$T(n) = T(3n/4) + 2bn$$ (quick select)

Solution: $\log_b a = 0$, so case 3 says T(n) is Θ(n).

Master Method, Example 8

- The form: $T(n) = \begin{cases} c & \text{if } n < d \\ aT(n/b) + f(n) & \text{if } n \geq d \end{cases}$
- The Master Theorem:
  1. if $f(n)$ is $O(n^{\log_b a - \varepsilon})$, then $T(n)$ is $\Theta(n^{\log_b a})$
  2. if $f(n)$ is $\Theta(n^{\log_b a} \log^k n)$, then $T(n)$ is $\Theta(n^{\log_b a} \log^{k+1} n)$
  3. if $f(n)$ is $\Omega(n^{\log_b a + \varepsilon})$, then $T(n)$ is $\Theta(f(n))$, provided $af(n/b) \leq \delta f(n)$ for some $\delta < 1$.
- Example:

$$T(n) = T(n/2) + \log n$$

Solution: $\log_b a = 0$, so case 2 says T(n) is $\Theta(\log^2 n)$.

Master Method, Example 9

- The form: $T(n) = \begin{cases} c & \text{if } n < d \\ aT(n/b) + f(n) & \text{if } n \geq d \end{cases}$
- The Master Theorem:
  1. if $f(n)$ is $O(n^{\log_b a - \varepsilon})$, then $T(n)$ is $\Theta(n^{\log_b a})$
  2. if $f(n)$ is $\Theta(n^{\log_b a} \log^k n)$, then $T(n)$ is $\Theta(n^{\log_b a} \log^{k+1} n)$
  3. if $f(n)$ is $\Omega(n^{\log_b a + \varepsilon})$, then $T(n)$ is $\Theta(f(n))$, provided $af(n/b) \leq \delta f(n)$ for some $\delta < 1$.
- Example:

$$T(n) = 2T(n/2) + \log n$$ (heap construction)

Solution: $\log_b a = 1$, so case 1 says T(n) is Θ(n).

Exercise:
Prove that $a^{\log x} = x^{\log a}$

Main Point

3. Divide-and-conquer algorithms can be directly translated into a recurrence relation; this can then be translated directly into a precise estimate of the algorithm's time complexity. Mathematics forms the basis of these analytic techniques (e.g., the Master Theorem). Their proofs of validity give us confidence in their correctness.
   Maharishi's Science and Technology of Consciousness provides systematic techniques for experiencing total knowledge of the Universe to enhance individual life.

Connecting the Parts of Knowledge with the Wholeness of Knowledge

1. Using comparison of keys only, the best sorting algorithm can only achieve a running time of O(n log n) on the average.
2. Through further knowledge of the structure and distribution of keys, a bucket sort and a radix sort can achieve O(n) running time.

3. **Transcendental Consciousness**, when directly experienced, is the basis for fully understanding the unified field located by Physics.
4. **Impulses within Transcendental Consciousness**: The dynamic natural laws within this field create and maintain the order and balance in creation. We verify this through regular practice and finding the nourishing influence of the Absolute in all areas of our life.
5. **Wholeness moving within itself:** In Unity Consciousness, knowledge is on the move; the fullness of pure consciousness is flowing onto the outer fullness of relative experience. Here there is nothing but knowledge; the knowledge is self-validating.

55