# Sorting Lower Bound Radix Sort

CSE 326
Data Structures
Lecture 16

## Reading

- Reading
  - Sections 7.8-7.11



## A Simple Lower Bound

- How do you tell if value X is in an unsorted array?
- Compare X to every value of array. Takes N comparisons.
- Can we do better?
- Doesn't seem like it, and we can prove it easily.



## A Simple Lower Bound

- Theorem: All find algorithms on unsorted arrays need at least N comparisons.
- Proof: Say algorithm F uses at most N-1 comparisons. Say F returns the correct answer on input (A, x). Let A'={ A[x] if F looked at x, max A +1 otherwise}. Then F has same behavior on (A', x), but returns wrong answer.



## How fast can we sort?

- Heapsort, Mergesort, and Quicksort all run in O(N log N) best case running time
- Can we do any better?
- No, if the basic action is a comparison.



## Sorting Model

- Recall our basic assumption: we can only compare two elements at a time
  - we can only reduce the possible solution space by half each time we make a comparison
- Suppose you are given N elements
  - Assume no duplicates
- How many possible orderings can you get?
  - Example: a, b, c (N = 3)

## Decision Trees

- Example of doctor's diagnosis.
- Possible outcomes = {healthy, flu, strep, migraine}
- Questions: Fever? Eyes hurt? Bacteria culture?

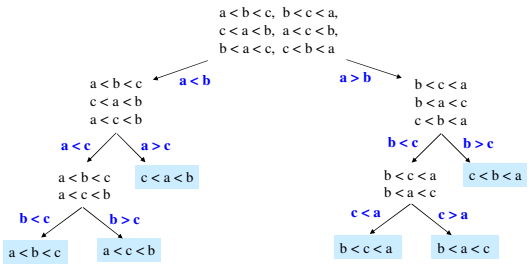


## Decision Trees

- Formally: Nodes represent possible hypothesis, leaves have only 1 hypothesis.
- Edges represent outcomes of some test.



## Decision Tree




The leaves contain all the possible orderings of a, b, c



## Decision Trees

- A Decision Tree is a Binary Tree such that:
  - Each node = a set of orderings
    - ie, the remaining solution space
  - Each edge = 1 comparison
  - Each leaf = 1 unique ordering
  - How many leaves for N distinct elements?
    - N!, ie, a leaf for each possible ordering
- Only 1 leaf has the ordering that is the desired correctly sorted arrangement

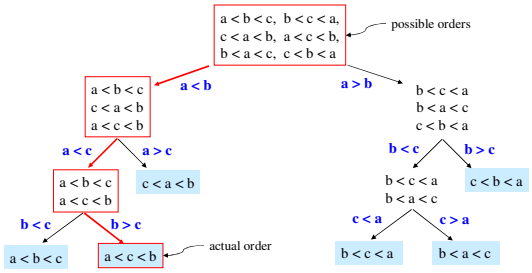


## Decision Trees and Sorting

- Every sorting algorithm corresponds to a decision tree
  - Finds correct leaf by choosing edges to follow
    - ie, by making comparisons
  - Each decision reduces the possible solution space by one half
- Run time is ≥ maximum no. of comparisons
  - maximum number of comparisons is the length of the longest path in the decision tree, i.e. the height of the tree

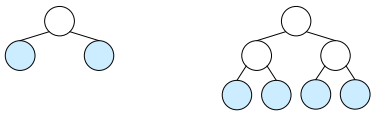


## Decision Tree Example



## How many leaves on a tree?

- Suppose you have a binary tree of height d . How many leaves can the tree have?
  - d = 1 → at most 2 leaves,
  - d = 2 → at most 4 leaves, etc.





## Permutations

- How many possible orderings can you get?
  - Example: a, b, c (N = 3)
  - (a b c), (a c b), (b a c), (b c a), (c a b), (c b a)
  - 6 orderings = 3·2·1 = 3! (ie, "3 factorial")
  - All the possible permutations of a set of 3 elements
- For N elements
  - N choices for the first position, (N-1) choices for the second position, …, (2) choices, 1 choice
  - N(N-1)(N-2)□(2)(1)= N! possible orderings



## Lower bound on Height

- A binary tree of height d has at most $2^d$ leaves
  - depth d = 1 → 2 leaves, d = 2 → 4 leaves, etc.
  - Can prove by induction
- Number of leaves, $L \leq 2^d$
- Height $d \geq \log_2 L$
- The decision tree has N! leaves
- So the decision tree has height $d \geq \log_2(N!)$



## log(*N*!) is Ω(*N*log*N*)

$$\log(N!) = \log\left(N \cdot (N-1) \cdot (N-2) \cdots (2) \cdot (1)\right)$$

"select just the first N/2 terms"

$$= \log N + \log(N-1) + \log(N-2) + \cdots + \log 2 + \log 1$$

$$\geq \log N + \log(N-1) + \log(N-2) + \cdots + \log \frac{N}{2}$$

"each of the selected terms is ≥ logN/2"

$$\geq \frac{N}{2} \log \frac{N}{2}$$

$$\geq \frac{N}{2}(\log N - \log 2) = \frac{N}{2} \log N - \frac{N}{2}$$

$$= \Omega(N \log N)$$



## Ω(N log N)

- Run time of any comparison-based sorting algorithm is **Ω(N log N)**
- Can we do better if we don't use comparisons?



## Bucket Sort

- If we know that keys are in some range [1…M], then make a bucket for each value, and drop items in bin.
- *Do example for M=16.*
- How long does this take?
  - O(N+M)

# Radix Sort

- If numbers are 32 bit integers, this doesn't work.
- Or: If keys are strings.
- Radix Sort: Like bucket sort, but get away with fewer buckets by making more passes



# Radix Sort Example

Input data

478
537
9
721
3
38
123
67

Bucket sort by 1's digit

| 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 |
|---|---|---|---|---|---|---|---|---|---|
| | 721 | | 3<br>123 | | | | 537<br>67 | 478<br>38 | 9 |

After 1st pass

721
3
123
537
67
478
38
9

This example uses B=10 and base 10 digits for simplicity of demonstration. Larger bucket counts should be used in an actual implementation.



# Radix Sort Example

After 1st pass

721
3
123
537
67
478
38
9

Bucket sort by 10's digit

| 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 |
|---|---|---|---|---|---|---|---|---|---|
| 03<br>09 | | 721<br>123 | 537<br>38 | | | 67 | 478 | | |

After 2nd pass

3
9
721
123
537
38
67
478



# Radix Sort Example

After 2nd pass

3
9
721
123
537
38
67
478

Bucket sort by 100's digit

| 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 |
|---|---|---|---|---|---|---|---|---|---|
| 003<br>009<br>038<br>067 | 123 | | | 478 | 537 | | 721 | | |

After 3rd pass

3
9
38
67
123
478
537
721

Invariant: after k passes the low order k digits are sorted.



# Radix Sort: Sorting integers

- Historically goes back to the 1890 census.
- Radix sort = multi-pass bucket sort of integers in the range 0 to $B^P$-1
- Bucket-sort from least significant to most significant "digit" (base B)
- Requires P(B+N) operations where P is the number of passes (the number of base B digits in the largest possible input number).
- If P and B are constants then O(N) time to sort!



# Radix sort with Strings

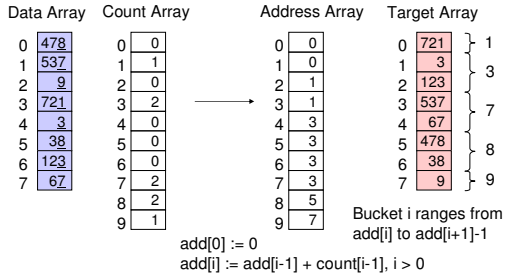
- Think of strings as just base 27 numbers.
- Pad with blank at end of string
- *Do example: art, ate, at.*

## Implementation Options

- List
  - › List of data, bucket array of lists.
  - › Concatenate lists for each pass.
- Array / List
  - › Array of data, bucket array of lists.
- Array / Array
  - › Array of data, array for all buckets.
  - › Requires counting.



## Array / Array






## Array / Array

- Pass 1 (over A)
  - › Calculate counts and addresses for 1st "digit"
- Pass 2 (over T)
  - › Move data from A to T
  - › Calculate counts and addresses for 2nd "digit"
- Pass 3 (over A)
  - › Move data from T to A
  - › Calculate counts and addresses for 3nd "digit"
- …
- In the end an additional copy may be needed.



## Choosing Parameters for Radix Sort

- N number of integers – given
- m bit numbers - given
- B number of buckets
  - › $B = 2^r$ – calculations can be done by shifting.
  - › N/B not too small, otherwise too many empty buckets.
  - › P = m/r should be small.
- Example – 1 million 64 bit numbers. Choose $B = 2^{16} = 65{,}536$. 1 Million / B ≈ 15 numbers per bucket. P = 64/16 = 4 passes.



## Properties of Radix Sort

- Not in-place
  - › needs lots of auxiliary storage.
- Stable
  - › equal keys always end up in same bucket in the same order.
- Fast
  - › $B = 2^r$ buckets on m bit numbers

$$O(\frac{m}{r}(n+2^r)) \quad \text{time}$$



## Internal versus External Sorting

- So far assumed that accessing A[i] is fast – Array A is stored in internal memory (RAM)
  - › Algorithms so far are good for internal sorting
- What if A is so large that it doesn't fit in internal memory?
  - › Data on disk or tape
  - › Delay in accessing A[i] – e.g. need to spin disk and move head

## Internal versus External Sorting

- Need sorting algorithms that minimize disk/tape access time
  - › External sorting – Basic Idea:
    - Load chunk of data into RAM, sort, store this "run" on disk/tape
    - Use the Merge routine from Mergesort to merge runs
    - Repeat until you have only one run (one sorted chunk)
    - Text gives some examples



## Summary of Sorting

- Sorting choices:
  - › $O(N^2)$ – Bubblesort, Insertion Sort
  - › O(N log N) average case running time:
    - Heapsort: In-place, not stable (read about it).
    - Mergesort: O(N) extra space, stable.
    - Quicksort: claimed fastest in practice but, $O(N^2)$ worst case. Needs extra storage for recursion. Not stable.
  - › O(N) – Radix Sort: fast and stable. Not comparison based. Not in-place.