

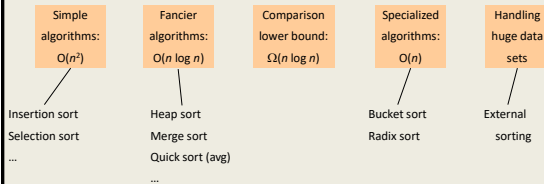
# CSE 332: Data Structures and Parallelism

Spring 2022  
 Richard Ruth Anderson  
 Lecture 14: Sorting II

# Announcements

- Exam Friday
  - today's material will not be on the midterm

## Sorting: The Big Picture



## Quicksort

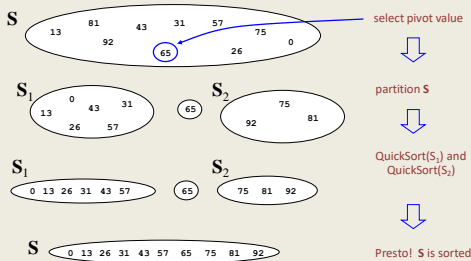
Quicksort uses a divide and conquer strategy, but does not require the  $O(N)$  extra space that MergeSort does.

Here's the idea for sorting array  $S$ :

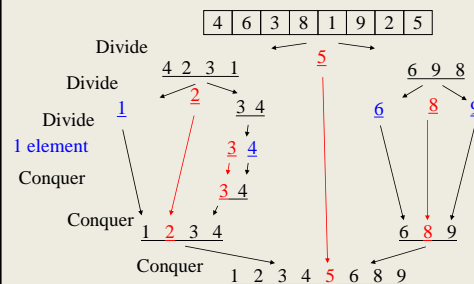
1. Pick an element  $v$  in  $S$ . This is the **pivot** value.
2. Partition  $S - \{v\}$  into two disjoint subsets,  $S_1$  and  $S_2$  such that:
  - elements in  $S_1$  are all  $\leq v$
  - elements in  $S_2$  are all  $\geq v$
3. Return concatenation of  $\text{QuickSort}(S_1), v, \text{QuickSort}(S_2)$

Recursion ends if  $\text{QuickSort}()$  receives an array of length 0 or 1.

## The steps of Quicksort



## Quicksort Example



## Pivot Picking and Partitioning

The tricky parts are:

- **Picking the pivot**
  - Goal: pick a pivot value so that  $|S_1|$  and  $|S_2|$  are roughly equal in size.
- **Partitioning**
  - Preferably in-place
  - Dealing with duplicates

4/27/2022

CSE 332

7

## Picking the pivot

- Choose the first element in the subarray
- Choose a value that might be close to the middle
  - Median of three
- Choose a random element

4/27/2022

CSE 332

8

## Quicksort Partitioning

- Partition the array into left and right sub-arrays such that:
  - elements in left sub-array are  $\leq$  pivot
  - elements in right sub-array are  $\geq$  pivot
- Can be done in-place with another “two pointer method”
  - Sounds like mergesort, but here we are *partitioning*, not sorting...
  - ...and we can do it in-place.
- Lots of work has been invested in engineering quicksort

4/27/2022

CSE 332

9

## Quicksort Pseudocode

Putting the pieces together:

```
Quicksort(A[], left, right) {
  if (left < right) {
    medianOf3Pivot(A, left, right);
    pivotIndex = Partition(A, left+1, right-1);

    Quicksort(A, left, pivotIndex - 1);
    Quicksort(A, pivotIndex + 1, right);
  }
}
```

4/27/2022

CSE 332

10

## Important Tweak

Insertion sort is actually better than quicksort on small arrays. Thus, a better version of quicksort:

```
Quicksort(A[], left, right) {
  if (right - left >= CUTOFF) {
    medianOf3Pivot(A, left, right);
    pivotIndex = Partition(A, left+1, right-1);

    Quicksort(A, left, pivotIndex - 1);
    Quicksort(A, pivotIndex + 1, right);
  } else {
    InsertionSort(A, left, right);
  }
}
```

CUTOFF = 16 is reasonable.

4/27/2022

CSE 332

11

## Quicksort run time

- What is the best case behavior?

4/27/2022

CSE 332

12

## Worst case run time

- What is the bad case for partitioning?
- Design a bad case input (assume first element is chosen as pivot)

4/27/2022

CSE 332

13

## Average case performance

- Assume all permutations of the data are equally likely
  - Or equivalently, a random pivot is chosen
- The math gets messy, but doable

4/27/2022

CSE 332

14

## Properties of Quicksort

- $O(N^2)$  worst case performance, but  $O(N \log N)$  average case performance.
- Pure quicksort not good for small arrays.
- No iterative version (without using a stack).
- “In-place,” but uses auxiliary storage because of recursive calls.
- Used by Java for sorting arrays of primitive types.

4/27/2022

CSE 332

15

## How fast can we sort?

Heapsort and Mergesort have  $O(N \log N)$  **worst** case running time.

These algorithms, along with Quicksort, also have  $O(N \log N)$  **average** case running time.

Can we do any better?

4/27/2022

CSE 332

16

## Permutations

- Suppose you are given  $N$  elements
  - Assume no duplicates
- How many possible orderings can you get?
  - Example: a, b, c ( $N = 3$ )

4/27/2022

CSE 332

17

## 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 =  $3 \cdot 2 \cdot 1 = 3!$  (i.e., “3 factorial”)
- For  $N$  elements
  - $N$  choices for the first position,  $(N-1)$  choices for the second position, ..., (2) choices, 1 choice
  - $N(N-1)(N-2) \cdots (2)(1) = \underline{N!}$  possible orderings

4/27/2022

CSE 332

18

## Sorting Model

Recall our basic sorting assumption:

**We can only compare two elements at a time.**

These comparisons prune the space of possible orderings.

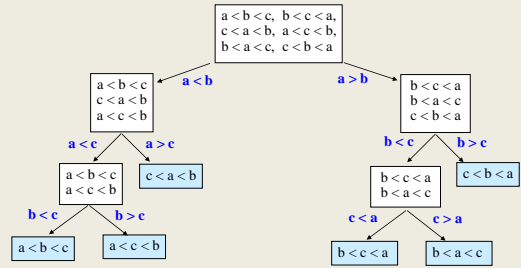
We can represent these concepts in a...

4/27/2022

CSE 332

19

## Decision Tree



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

4/27/2022

CSE 332

20

## Decision Trees

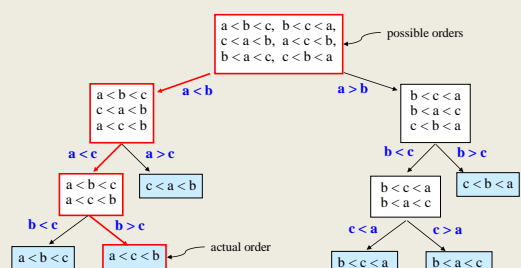
- A Decision Tree is a Binary Tree such that:
  - Each node = a set of orderings
    - i.e., the remaining solution space
  - Each edge = 1 comparison
  - Each leaf = 1 unique ordering
  - How many leaves for  $N$  distinct elements?
- Only 1 leaf has the ordering that is the desired correctly sorted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

4/27/2022

CSE 332

21

## Decision Tree Example



4/27/2022

CSE 332

22

## Decision Trees and Sorting

- Every comparison based sorting algorithm corresponds to a decision tree
  - Finds correct leaf by choosing edges to follow
    - i.e., by making comparisons
- We will focus on worst case run time
- Observations:
  - Worst case run time  $\geq$  max number of comparisons
  - Max number of comparisons = length of the longest path in the decision tree = tree height

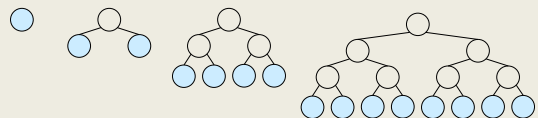
4/27/2022

CSE 332

23

## How many leaves on a tree?

Suppose you have a binary tree of height  $h$ . How many leaves in a perfect tree?



We can prune a perfect tree to make any binary tree of same height. Can # of leaves increase?

4/27/2022

CSE 332

24

## Lower bound on Height

- A binary tree of height  $h$  has at most  $2^h$  leaves
  - Can prove by induction
- A decision tree has  $N!$  leaves. What is its minimum height?

4/27/2022

CSE 332

25

## Lower bound on $\log(n!)$

$$\begin{aligned} n! &= n \cdot (n-1) \cdot (n-2) \cdots 4 \cdot 3 \cdot 2 \cdot 1 \\ &\geq n \cdot (n-1) \cdot (n-2) \cdots \frac{n}{2} \\ &\geq \frac{n}{2} \cdot \frac{n}{2} \cdot \frac{n}{2} \cdots \frac{n}{2} \\ &\geq \left(\frac{n}{2}\right)^{n/2} \end{aligned}$$

$$\log n! \geq \log \left(\frac{n}{2}\right)^{n/2} = \frac{n}{2} \log \frac{n}{2}$$

4/27/2022

CSE 332

26

## $\Omega(N \log N)$

**Worst case** run time of any comparison-based sorting algorithm is  $\Omega(N \log N)$ .

Can also show that **average case** run time is also  $\Omega(N \log N)$ .

Can we do better if we don't use comparisons? (Huh?)

4/27/2022

CSE 332

27

## Can we sort in $O(n)$ ?

- Suppose keys are integers between 0 and 1000

4/27/2022

CSE 332

28

## BucketSort (aka BinSort)

If all values to be sorted are integers between 1 and  $B$ , create an array **count** of size  $B$ , **increment** counts while traversing the input, and finally output the result.

**Example**  $B=5$ . Input = (5,1,3,4,3,2,1,1,5,4,5)

count array	
1	
2	
3	
4	
5	



Running time to sort  $n$  items?

4/27/2022

CSE 332

29

## What about our $\Omega(n \log n)$ bound?

4/27/2022

CSE 332

30

## Dependence on $B$

What if  $B$  is very large (e.g.,  $2^{64}$ )?

4/27/2022

CSE 332

31

## Fixing impracticality: RadixSort

- RadixSort: generalization of BucketSort for large integer keys
- Origins go back to the 1890 census.
- Radix = “The base of a number system”
  - We’ll use 10 for convenience, but could be anything
- Idea:
  - BucketSort on one digit at a time
  - After  $k^{\text{th}}$  sort, the last  $k$  digits are sorted
  - Set number of buckets:  $B = \text{radix}$ .

4/27/2022

CSE 332

32

## Radix Sort Example

Input: 478, 537, 9, 721, 3, 38, 123, 67

BucketSort  
on 1's

0	1	2	3	4	5	6	7	8	9

BucketSort  
on 10's

0	1	2	3	4	5	6	7	8	9

BucketSort  
on 100's

0	1	2	3	4	5	6	7	8	9

Output:

4/27/2022

CSE 332

33

## Radix Sort Example (1<sup>st</sup> pass)

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				537	478	9

After 1<sup>st</sup> 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.

4/27/2022

CSE 332

34

## Radix Sort Example (2<sup>nd</sup> pass)

After 1<sup>st</sup> 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	09	721	537	123	38		67	478	

After 2<sup>nd</sup> pass

3  
9  
721  
123  
537  
38  
67  
478

4/27/2022

CSE 332

35

## Radix Sort Example (3<sup>rd</sup> pass)

After 2<sup>nd</sup> 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	009	123		478	537		721		

After 3<sup>rd</sup> pass

3  
9  
38  
67  
123  
478  
537  
721

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

4/27/2022

CSE 332

36

## Radixsort: Complexity

In our examples, we had:

- Input size,  $N$
- Number of buckets,  $B = 10$
- Maximum value,  $M < 10^3$
- Number of passes,  $P =$

How much work per pass?

Total time?

4/27/2022

CSE 332

37

## Choosing the Radix

Run time is roughly proportional to:

$$P(B+N) = \log_b M(B+N)$$

Can show that this is minimized when:

$$B \log_e B \approx N$$

In theory, then, the best base (radix) depends only on  $N$ .

For fast computation, prefer  $B = 2^p$ . Then best  $b$  is:

$$b + \log_2 b \approx \log_2 N$$

Example:

- $N = 1$  million (i.e.,  $\sim 2^{20}$ ) 64 bit numbers,  $M = 2^{64}$
- $\log_2 N \approx 20 \rightarrow b = 16$
- $B = 2^{16} = 65,536$  and  $P = \log_{(2^{16})} 2^{64} = 4$ .

In practice, memory word sizes, space, other architectural considerations, are important in choosing the radix.

4/27/2022

CSE 332

38

## Sorting Summary

$O(N^2)$  average, worst case:

- **Selection Sort, Bubblesort, Insertion Sort**

$O(N \log N)$  average case:

- **Heapsort:** In-place, not stable.
- **BST Sort:**  $O(N)$  extra space (including tree pointers, possibly poor memory locality), stable.
- **Mergesort:**  $O(N)$  extra space, stable.
- **Quicksort:** claimed fastest in practice, but  $O(N^2)$  worst case. Recursion/stack requirement. Not stable.

$\Omega(N \log N)$  worst and average case:

- **Any comparison-based sorting algorithm**

$O(N)$

- **Radix Sort:** fast and stable. Not comparison based. Not in-place. Poor memory locality can undercut performance.

4/27/2022

CSE 332

39