

### Lower Bounds for Comparison-Based Sorting Algorithms (Ch. 8)

We have seen several sorting algorithms that run in  $\Omega(n \lg n)$  time in the worst case (meaning there is some input on which the algorithms run in at least  $\Omega(n \lg n)$  time).

- mergesort
- heapsort
- quicksort

In all comparison-based sorting algorithms, the sorted order results only from comparisons between input elements.

Is it possible for any comparison-based sorting algorithm to do better?

### Lower Bounds for Sorting Algorithms

Theorem: Any comparison-based sort must make  $\Omega(n \lg n)$  comparisons in the worst case to sort a sequence of  $n$  elements. (Across all comparison-based sorting algorithms, no worst case runs faster than  $n \lg n$  time.)

But how do we prove this?

We'll use the *decision tree model* to represent any sorting algorithm and then argue that no matter the algorithm, there is some input that will cause it to run in  $\Omega(n \lg n)$  time.

Question: How many ways are there to order  $n$  elements?

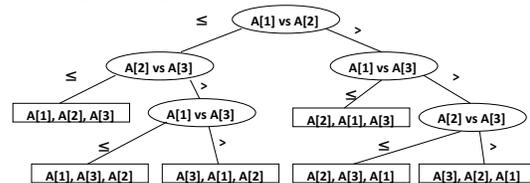
### The Decision Tree Model

Given any comparison-based sorting algorithm, we can represent its behavior on an input of size  $n$  by a decision tree. Note: we need only consider the *comparisons* in the algorithm (the other operations only make the algorithm take longer).

- each internal node in the decision tree corresponds to one of the comparisons in the algorithm.
  - start at the root and do first comparison (e.g.,  $x > y$ )
    - if  $x \leq y$  take left branch, if  $x > y$  take right branch, etc.
  - each leaf represents one possible ordering of the input
- ⇒ One decision tree exists for each algorithm and input size

### The Decision Tree Model

Example: decision tree with  $n = 3$ , with elements  $A[1..3]$  has  $3! = 6$  leaves (sorting in ascending order)



Note: The length of the longest root to leaf path in this tree is  $h$   
 = worst-case number of comparisons  
 $\leq$  worst-case number of operations of algorithm

### The $\Omega(n \lg n)$ Lower Bound

Theorem: Any decision tree for sorting  $n$  elements has height  $\Omega(n \lg n)$  (therefore, any comparison-based sorting algorithm requires  $\Omega(n \lg n)$  comparisons in worst case).

- Proof: Let  $h$  be the height of the tree. Then we know
- tree has at least  $(\geq) n!$  leaves
  - tree is binary, so it has at most  $(\leq) 2^h$  leaves

$$2^h \geq \text{number of leaves} \geq n!$$

$$2^h \geq n!$$

$$\lg(2^h) \geq \lg(n!)$$

$$h \geq \Omega(n \lg n) \text{ (Eq. 3.18)} \quad \square$$

### Optimal Sorting Algorithms

- This lower bound proof tells us that heap-sort and merge-sort are asymptotically optimal comparison-based sorting algorithms.
- Randomized-Quick-Sort is asymptotically optimal with high probability.
- We also know that insertion-sort, selection-sort, and bubble-sort are not asymptotically optimal comparison-based algorithms.

## Beating the lower bound ... non-comparison-based sorts

**Idea:** Algorithms that are NOT comparison-based might be faster.

There are three such algorithms presented in Chapter 8:

- counting sort
- radix sort
- bucket sort

These algorithms

- run in  $O(n)$  time (under certain conditions)
- either use information about the values to be sorted (counting sort, bucket sort), or
- operate on "pieces" of the input elements (radix sort)

## Counting Sort

**Requirement:** input elements are integers in *known range*  $[0..k]$  for some constant  $k$ .

**Idea:** for each input element  $x$ , find the number of elements  $\leq x$  (say this number =  $m$ ) and put  $x$  in the  $(m+1)$ st spot in the output array.

```
Counting-Sort(A, k)
// A[1..n] is input array, C[0..k] is initially all 0's, B[1..n] is output array
// (initially all 0's)
1. for i = 1 to A.length
2.   C[A[i]] = C[A[i]] + 1 // Count number of times each value appears in A
3. for i = 1 to k
4.   C[i] = C[i] + C[i-1] // Make C into a "prefix sum" array, where C[i]
// contains number of elements <= i
5. for j = A.length downto 1
6.   B[C[A[j]]] = A[j]
7.   C[A[j]] = C[A[j]] - 1
```

Running time of Counting-Sort?

## Running Time of Counting Sort

for loop in lines 1-2 takes  $\theta(n)$  time. Overall time is  $\theta(k + n)$ .  
for loop in lines 3-4 takes  $\theta(k)$  time.  
for loop in lines 5-7 takes  $\theta(n)$  time.

In practice, use counting sort when we have  $k = \theta(n)$ , so running time is  $\theta(n)$ .

This version of counting sort has the important property of **stability**. A sorting algorithm is **stable** when numbers with equal values appear in the output array in the same order as they do in the input array.

Important when satellite data is stored with elements being sorted and when counting sort is used as a subroutine for radix sort, the next NCB algorithm we'll look at.

## Radix Sort

Let  $d$  be the number of digits in each input number.

```
Radix-Sort(A, d)
1. for i = 1 to d
2.   use stable sort to sort array A on digit i
```

Note:

- radix sort sorts the *least* significant digit first.
- correctness can be shown by induction on the digit being sorted.
- counting sort is often used as the stable sort in step 2.

Running time of Radix-Sort?

## Radix Sort

Let  $d$  be the number of digits in each input number.

```
Radix-Sort(A, d)
1. for i = 1 to d
2.   use stable sort to sort array A on digit i
```

Running time of radix sort:  $O(dT_{cs}(n))$

- $T_{cs}$  is the time for the internal sort. Counting sort gives  $T_{cs}(n) = O(k + n)$ , so  $O(dT_{cs}(n)) = O(d(k + n))$ , which is  $O(n)$  if  $d = O(1)$  and  $k = O(n)$ .

## Bucket Sort

**Assumption:** input elements distributed uniformly over some known range, e.g.,  $[0, 1)$ . (Appendix C.2 has definition of uniform distribution)

```
Bucket-Sort(A, x, y)
1. divide interval [x, y] into n equal-sized subintervals (buckets)
2. distribute the n input keys into the buckets
3. sort the numbers in each bucket (e.g., with insertion sort)
4. scan the (sorted) buckets in order and produce output array
```

If a bucket has  $> 1$  key they are joined into a linked list.

Running time of Bucket-Sort?

## Bucket Sort

Bucket-Sort( $A, x, y$ )

1. divide interval  $[x, y]$  into  $n$  equal-sized subintervals (buckets)
2. distribute the  $n$  input keys into the buckets
3. sort the numbers in each bucket (e.g., with insertion sort) as they are inserted in the bucket
4. scan the (sorted) buckets in order and produce output array

Running time of bucket sort:  $O(n)$  expected time

*Step 1:*  $O(1)$  for each interval =  $O(n)$  time total.

*Step 2:*  $O(n)$  time.

*Step 3:* The expected number of elements in each bucket is  $O(1)$  (see book for formal argument, section 8.4), so total is  $O(n)$

*Step 4:*  $O(n)$  time to scan the  $n$  buckets containing a total of  $n$  input elements

A bucket is really a linked list.

## Summary NCB Sorts

### Non-Comparison-Based Sorts

	Running Time			
	worst-case	average-case	best-case	in place
Counting Sort	$O(n + k)$	$O(n + k)$	$O(n + k)$	no
Radix Sort	$O(d(n + k))$	$O(d(n + k))$	$O(d(n + k))$	no
Bucket Sort		$O(n)$		no

Counting sort requires known range of data  $[0, 1, 2, \dots, k]$  and uses array indexing to count the number of occurrences of each value.

Radix sort requires that each integer consists of  $d$  digits, and each digit is in range  $[1, 2, \dots, k]$ .

Bucket sort requires advance knowledge of input distribution (sorts  $n$  numbers uniformly distributed in range in  $O(n)$  time).