Sorting In Linear Time

Sorting

- Sort n integers in $O(n\ logn)$
  - Merge Sort, Heap Sort in the worst case
  - Quick Sort on average

- Compare elements to determine the sorted order
  - Called comparison sorts

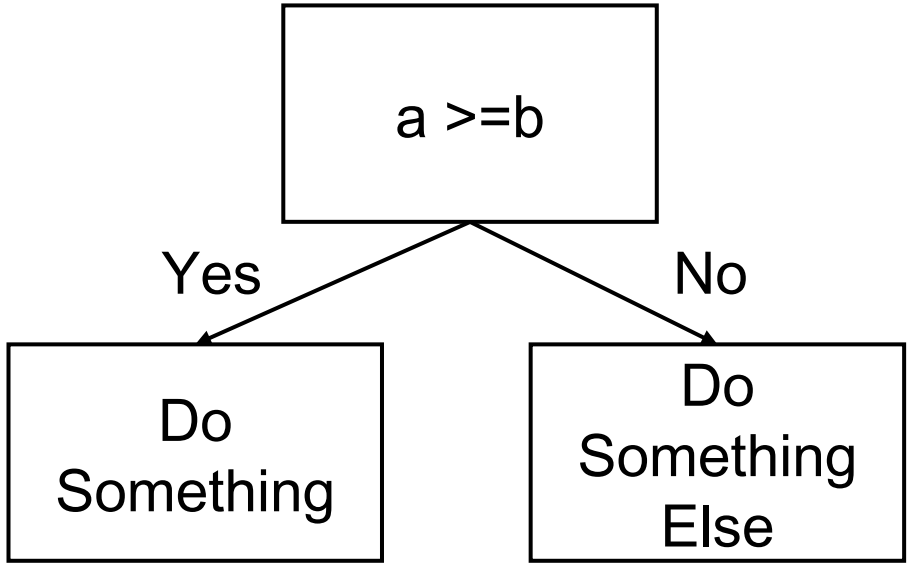
Lower Bounds for Sorting

- Comparison sorts gains order information from comparisons
  - Doesn't inspect values
  - Given a_i and a_j, test $a_i < a_j$, $a_i \leq a_j$, $a_i = a_j$, $a_i \geq a_j$, or $a_i > a_j$

- For lower bound analysis,
  - All elements are distinct

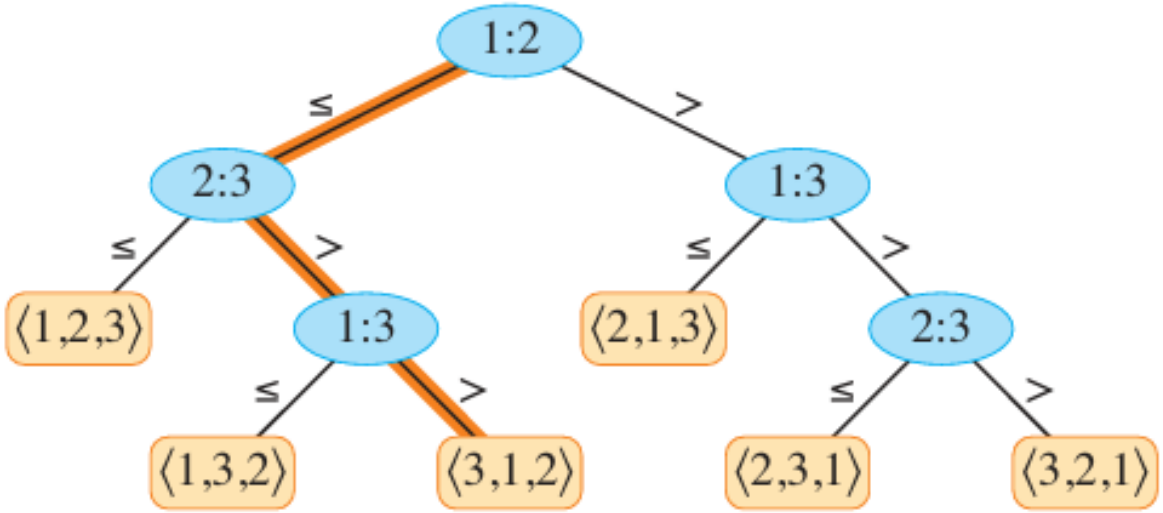
Lower Bounds for Sorting

- The decision tree model
  - A full binary-tree
  - Node is either a leaf or has both children
  - Node represents the comparisons between two elements



Lower Bounds for Sorting

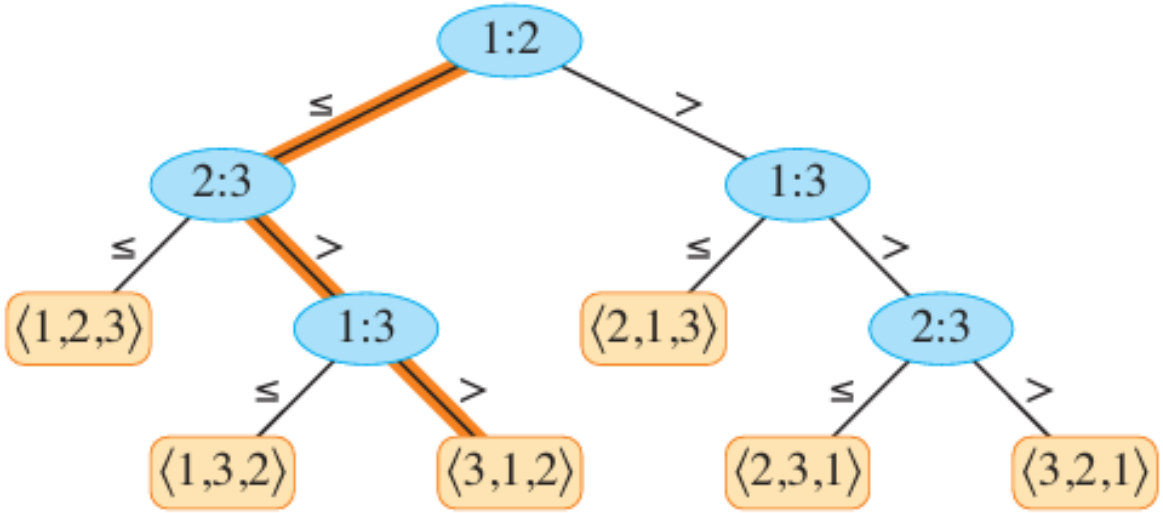
- The decision tree model
  - Leaf indicates a permutation of n elements
  - Internal nodes are annotated as $i:j$
  - Internal node $i:j$ indicates a comparison $a_i \geq a_j$



Lower Bounds for Sorting

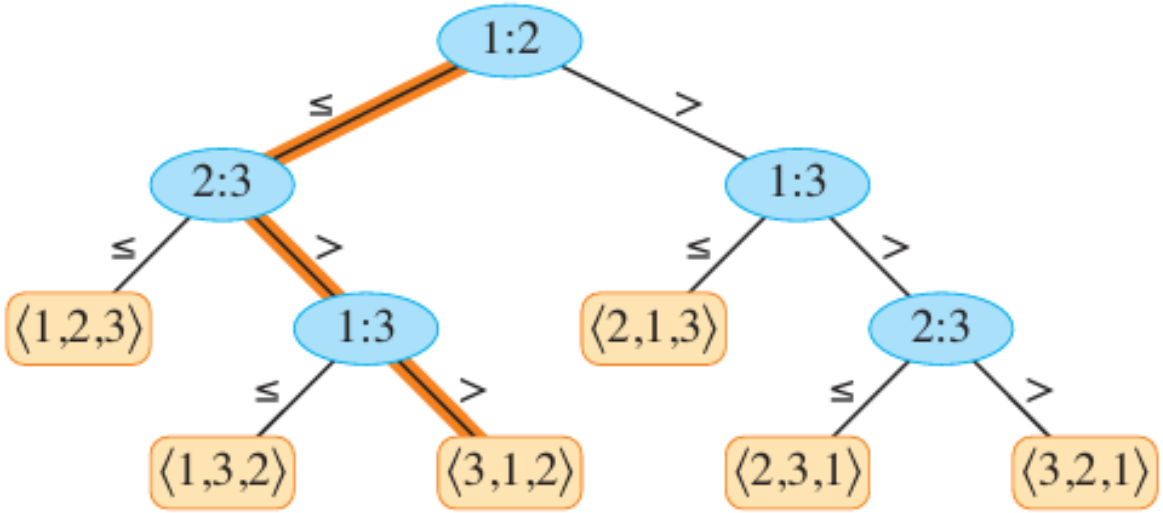
- Any correct sorting algorithm must be able to produce each permutation of its input

- Each of the $n!$ permutations on n elements must appear as at least one of the leaves



Lower Bounds for Sorting

- Worst-case comparisons
  - Longest simple path length from the root to any of its reachable leaves
  - Equals the height of the decision tree
- Can we estimate a bound of the height of such decision trees?



Lower Bounds for Sorting

- Any comparison sort algorithm requires $\Omega(n \log n)$ comparisons in the worst case.
- Proof:
  - A binary tree of height h has no more than 2^h leaves

$$n! \leq 2^h$$

$$\Theta(n \log n) = \log(n!) \leq h$$

- Merge sort and Heapsort are asymptotically optimal algorithms

Counting Sort

- Assumes that inputs are integers in the range 0 to k
- For an input x,
  - Determine the number of elements less than or equal to x
  - Place element x directly into its position in the output array

- Runs in $\Theta(n + k)$

- How to handle duplicates?

Counting Sort

```
COUNTING-SORT(A, n, k)
 1  let B[1 : n] and C[0 : k] be new arrays
 2  for i = 0 to k
 3      C[i] = 0
 4  for j = 1 to n
 5      C[A[j]] = C[A[j]] + 1
 6  // C[i] now contains the number of elements equal to i.
 7  for i = 1 to k
 8      C[i] = C[i] + C[i − 1]
 9  // C[i] now contains the number of elements less than or equal to i.
10  // Copy A to B, starting from the end of A.
11  for j = n downto 1
12      B[C[A[j]]] = A[j]
13      C[A[j]] = C[A[j]] − 1    // to handle duplicate values
14  return B
```

Counting Sort

```
COUNTING-SORT(A, n, k)
 1  let B[1 : n] and C[0 : k] be new arrays
 2  for i = 0 to k
 3      C[i] = 0
 4  for j = 1 to n
 5      C[A[j]] = C[A[j]] + 1
 6  // C[i] now contains the number of elements equal to i.
 7  for i = 1 to k
 8      C[i] = C[i] + C[i − 1]
 9  // C[i] now contains the number of elements less than or equal to i.
10  // Copy A to B, starting from the end of A.
11  for j = n downto 1
12      B[C[A[j]]] = A[j]
13      C[A[j]] = C[A[j]] − 1    // to handle duplicate values
14  return B
```

A							
2	5	3	0	2	3	0	3

C					
2	0	2	3	0	1

Counting Sort

```
COUNTING-SORT(A, n, k)
 1  let B[1 : n] and C[0 : k] be new arrays
 2  for i = 0 to k
 3      C[i] = 0
 4  for j = 1 to n
 5      C[A[j]] = C[A[j]] + 1
 6  // C[i] now contains the number of elements equal to i.
 7  for i = 1 to k
 8      C[i] = C[i] + C[i − 1]
 9  // C[i] now contains the number of elements less than or equal to i.
10  // Copy A to B, starting from the end of A.
11  for j = n downto 1
12      B[C[A[j]]] = A[j]
13      C[A[j]] = C[A[j]] − 1    // to handle duplicate values
14  return B
```

A							
2	5	3	0	2	3	0	3

C (old)					
2	0	2	3	0	1

C (current state)					
2	2	4	7	7	8

Counting Sort

```
COUNTING-SORT(A, n, k)
 1  let B[1 : n] and C[0 : k] be new arrays
 2  for i = 0 to k
 3      C[i] = 0
 4  for j = 1 to n
 5      C[A[j]] = C[A[j]] + 1
 6  // C[i] now contains the number of elements equal to i.
 7  for i = 1 to k
 8      C[i] = C[i] + C[i − 1]
 9  // C[i] now contains the number of elements less than or equal to i.
10  // Copy A to B, starting from the end of A.
11  for j = n downto 1
12      B[C[A[j]]] = A[j]
13      C[A[j]] = C[A[j]] − 1    // to handle duplicate values
14  return B
```

A							
2	5	3	0	2	3	0	3

C (current state)					
2	2	4	7	7	8

B							

Counting Sort

```
COUNTING-SORT(A, n, k)
 1  let B[1 : n] and C[0 : k] be new arrays
 2  for i = 0 to k
 3      C[i] = 0
 4  for j = 1 to n
 5      C[A[j]] = C[A[j]] + 1
 6  // C[i] now contains the number of elements equal to i.
 7  for i = 1 to k
 8      C[i] = C[i] + C[i − 1]
 9  // C[i] now contains the number of elements less than or equal to i.
10  // Copy A to B, starting from the end of A.
11  for j = n downto 1
12      B[C[A[j]]] = A[j]
13      C[A[j]] = C[A[j]] − 1    // to handle duplicate values
14  return B
```

A							
2	5	3	0	2	3	0	3

C (current state)					
2	2	4	7	7	8

B							
						3	

Counting Sort

```
COUNTING-SORT(A, n, k)
1   let B[1 : n] and C[0 : k] be new arrays
2   for i = 0 to k
3       C[i] = 0
4   for j = 1 to n
5       C[A[j]] = C[A[j]] + 1
6   // C[i] now contains the number of elements equal to i.
7   for i = 1 to k
8       C[i] = C[i] + C[i − 1]
9   // C[i] now contains the number of elements less than or equal to i.
10  // Copy A to B, starting from the end of A.
11  for j = n downto 1
12      B[C[A[j]]] = A[j]
13      C[A[j]] = C[A[j]] − 1    // to handle duplicate values
14  return B
```

A							
2	5	3	0	2	3	0	3

C (current state)					
2	2	4	6	7	8

B							
						3	

Counting Sort

```
COUNTING-SORT(A, n, k)
1   let B[1 : n] and C[0 : k] be new arrays
2   for i = 0 to k
3       C[i] = 0
4   for j = 1 to n
5       C[A[j]] = C[A[j]] + 1
6   // C[i] now contains the number of elements equal to i.
7   for i = 1 to k
8       C[i] = C[i] + C[i − 1]
9   // C[i] now contains the number of elements less than or equal to i.
10  // Copy A to B, starting from the end of A.
11  for j = n downto 1
12      B[C[A[j]]] = A[j]
13      C[A[j]] = C[A[j]] − 1    // to handle duplicate values
14  return B
```

A							
2	5	3	0	2	3	0	3

C (current state)					
2	2	4	6	7	8

B							
	0					3	

Counting Sort

```
COUNTING-SORT(A, n, k)
1   let B[1 : n] and C[0 : k] be new arrays
2   for i = 0 to k
3       C[i] = 0
4   for j = 1 to n
5       C[A[j]] = C[A[j]] + 1
6   // C[i] now contains the number of elements equal to i.
7   for i = 1 to k
8       C[i] = C[i] + C[i − 1]
9   // C[i] now contains the number of elements less than or equal to i.
10  // Copy A to B, starting from the end of A.
11  for j = n downto 1
12      B[C[A[j]]] = A[j]
13      C[A[j]] = C[A[j]] − 1    // to handle duplicate values
14  return B
```

A							
2	5	3	0	2	3	0	3

C (current state)					
1	2	4	6	7	8

B							
	0					3	

Counting Sort

```
COUNTING-SORT(A, n, k)
1   let B[1 : n] and C[0 : k] be new arrays
2   for i = 0 to k
3       C[i] = 0
4   for j = 1 to n
5       C[A[j]] = C[A[j]] + 1
6   // C[i] now contains the number of elements equal to i.
7   for i = 1 to k
8       C[i] = C[i] + C[i − 1]
9   // C[i] now contains the number of elements less than or equal to i.
10  // Copy A to B, starting from the end of A.
11  for j = n downto 1
12      B[C[A[j]]] = A[j]
13      C[A[j]] = C[A[j]] − 1    // to handle duplicate values
14  return B
```

A							
2	5	3	0	2	3	0	3

C (current state)					
1	2	4	6	7	8

B							
	0				3	3	

Counting Sort

```
COUNTING-SORT(A, n, k)
 1  let B[1 : n] and C[0 : k] be new arrays
 2  for i = 0 to k
 3      C[i] = 0
 4  for j = 1 to n
 5      C[A[j]] = C[A[j]] + 1
 6  // C[i] now contains the number of elements equal to i.
 7  for i = 1 to k
 8      C[i] = C[i] + C[i − 1]
 9  // C[i] now contains the number of elements less than or equal to i.
10  // Copy A to B, starting from the end of A.
11  for j = n downto 1
12      B[C[A[j]]] = A[j]
13      C[A[j]] = C[A[j]] − 1    // to handle duplicate values
14  return B
```

A							
2	5	3	0	2	3	0	3

C (current state)					
0	2	2	4	7	7

B							
0	0	2	2	3	3	3	5

Counting Sort

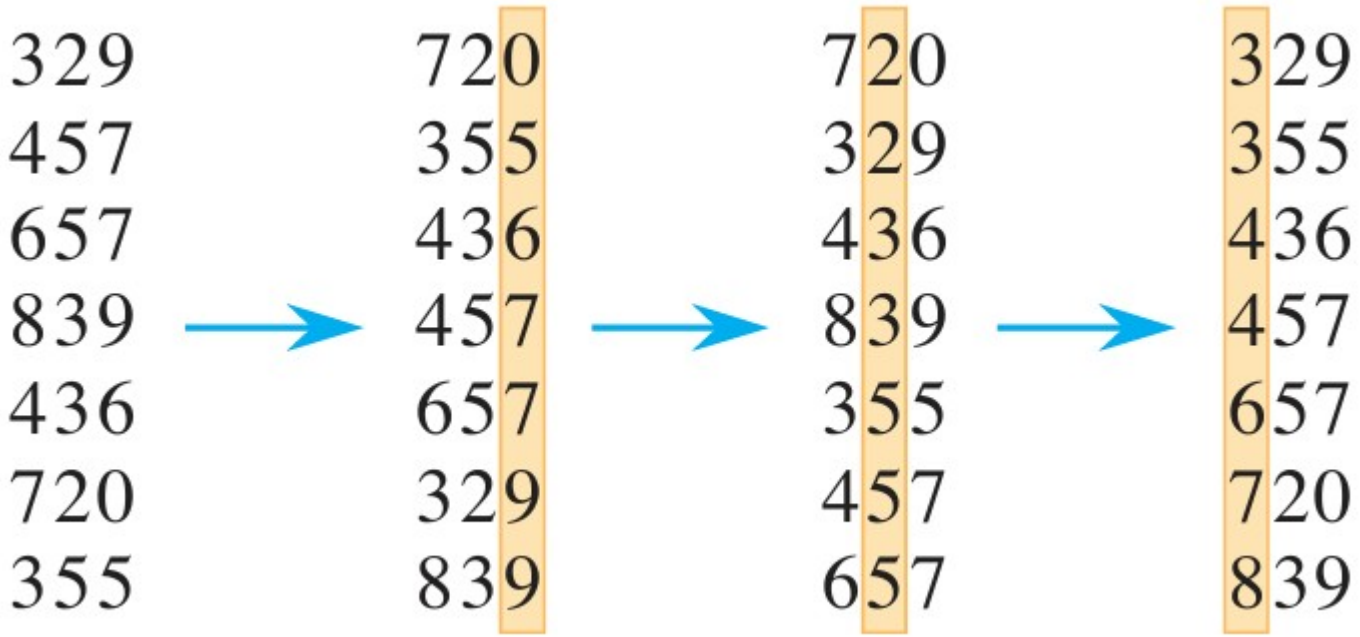
- Counting sort is stable
  - Elements with the same value appear in the output array in the same order as they do in the input array.

Radix Sort

- We can sort the elements
  - By most significant (leftmost) digit
  - Then sort the resulting bins recursively
  - Finally combine the bins in order.
  - Generates many intermediate bins to track

Radix Sort

- Let’s start by the least significant bit
- For example,



Radix Sort

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

Radix-sort sorts n d-digit number in $O(d(n+k))$

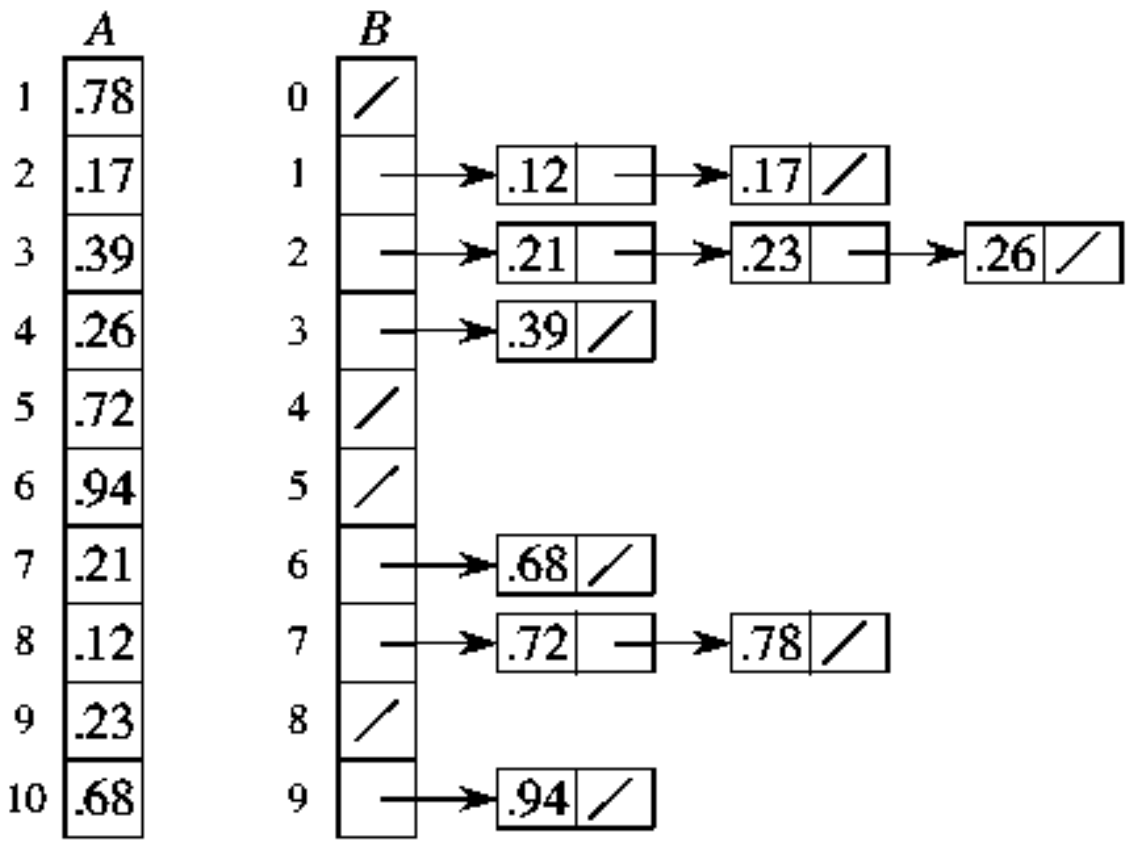
Bucket Sort

- Assumes that the input is uniformly distributed in a range
- Divides the range into n equal-sized subintervals, or buckets
- Distributes the inputs into the buckets
- The inputs are uniformly distributed
- Expected number of input per bucket is low
- Sort the numbers in each bucket
- Go through the buckets in order, listing the elements in each

Bucket Sort

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

Bucket Sort



- Runs in $O(n)$

Reference

- Sorting in Linear Time
  - CLRS 4th Ed. Chapter 8