Sorting in Linear Time

Outline

1. Lower Bounds and Decision Trees

2. The $\Omega(n \log n)$ Lower Bound for Comparison-Based Sorting

3. Linear-time Sorting
   - Counting Sort
   - Radix Sort

4. Sorting Reprise

Running time of sorting algorithms

Do you still remember what these statements mean?

- Sorting algorithm $\mathcal{A}$ runs in $O(n \log n)$ time.
- Sorting algorithm $\mathcal{A}$ runs in $\Omega(n \log n)$ time.

So far, all sorting algorithms have running time $\Omega(n \log n)$. Merge Sort and Heapsort also have running time $O(n \log n)$.

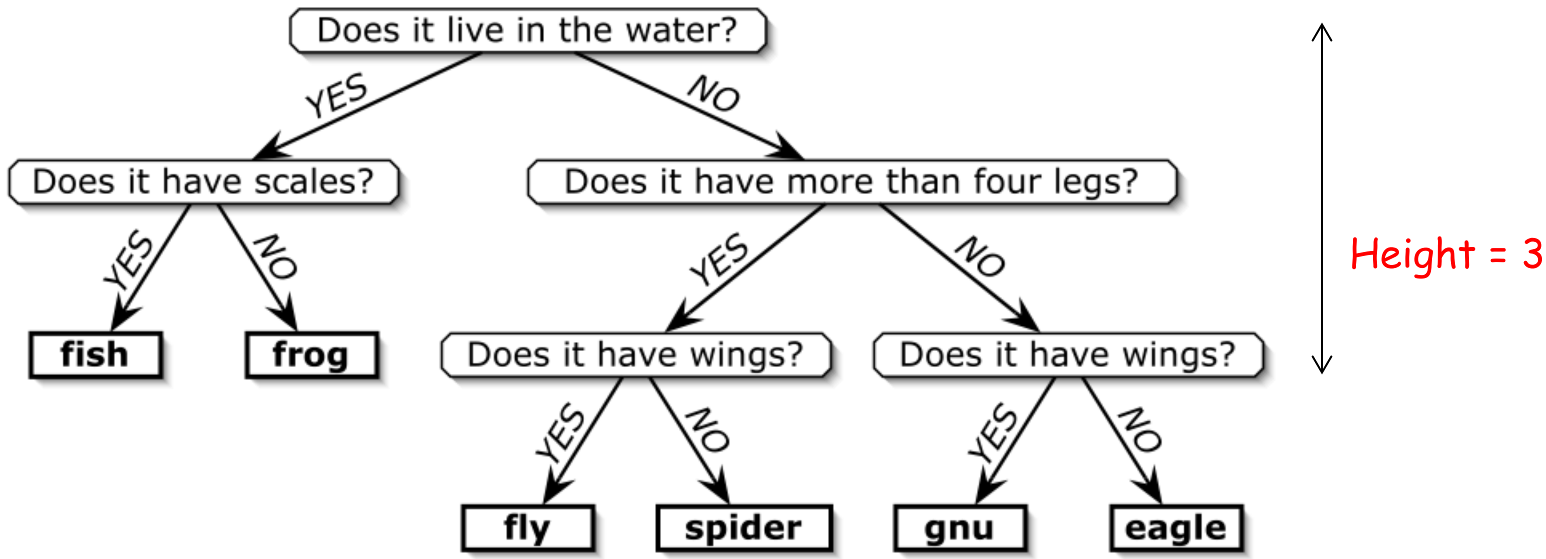
Q: Is it possible to design an algorithm that runs faster than $\Omega(n \log n)$?

A: No, if the algorithm is **comparison-based**. We will use the decision-tree model of computation to show that any **comparison-based** sorting algorithm requires $\Omega(n \log n)$ time.

Remark: A comparison-based sorting algorithm is any algorithm that makes decisions **only** by using comparisons between items (and not by referencing their actual values).

Thus, Merge Sort and Heapsort are asymptotically optimal **comparison-based** algorithms.

The decision-tree model (i)

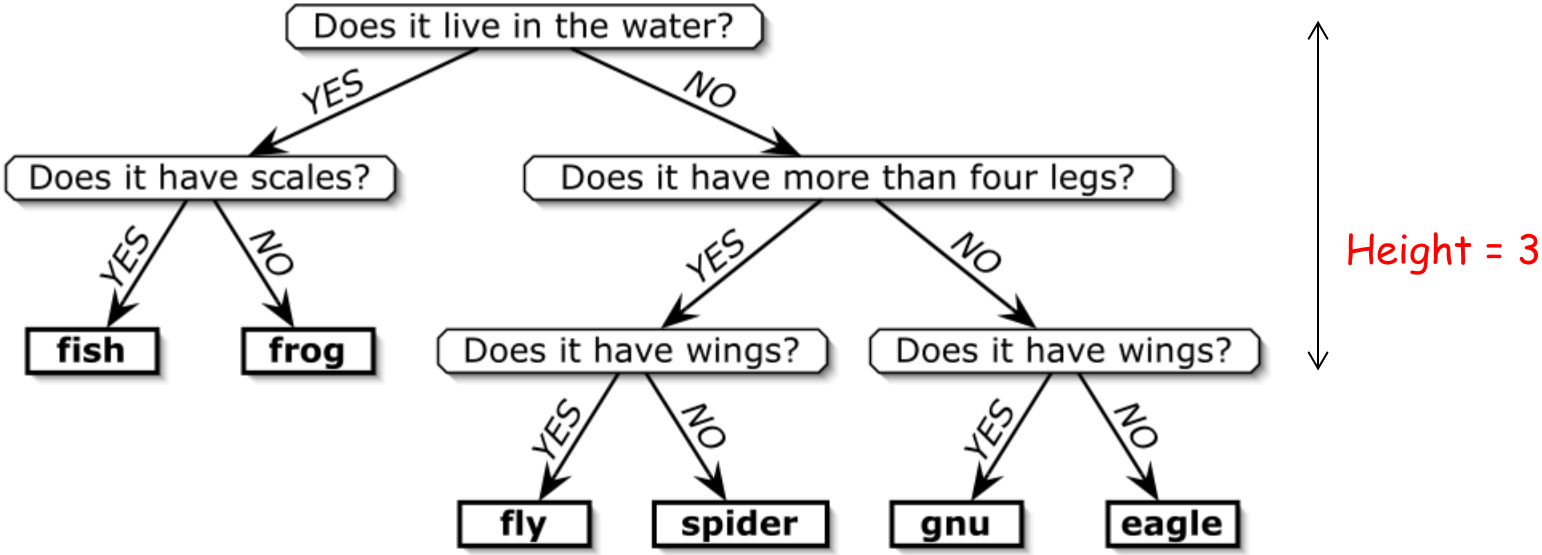


A decision tree to choose one of six animals.

An algorithm in the decision-tree model

- Solves the problem by asking questions with binary (Yes/No) answers
  - For sorting algorithms instead of binary questions we have comparisons with 2 outcomes ($\leq$,$>$).
- The worst-case running time is the height of the decision tree.
  - Height is length of longest path from root to a leaf

The decision-tree model (ii)



A decision tree to choose one of six animals.

An algorithm in the decision-tree model

- Solves the problem by asking questions with binary (Yes/No) answers
- The worst-case running time is the height of the decision tree.
- FACT: A binary tree with n leaves must have height $\Omega(\log n)$.

- => An algorithm in the decision tree model that has n different outputs has running time $\Omega(\log n)$.

Heights of Binary Trees

FACT: A binary tree with n leaves must have height $\Omega(\log n)$.

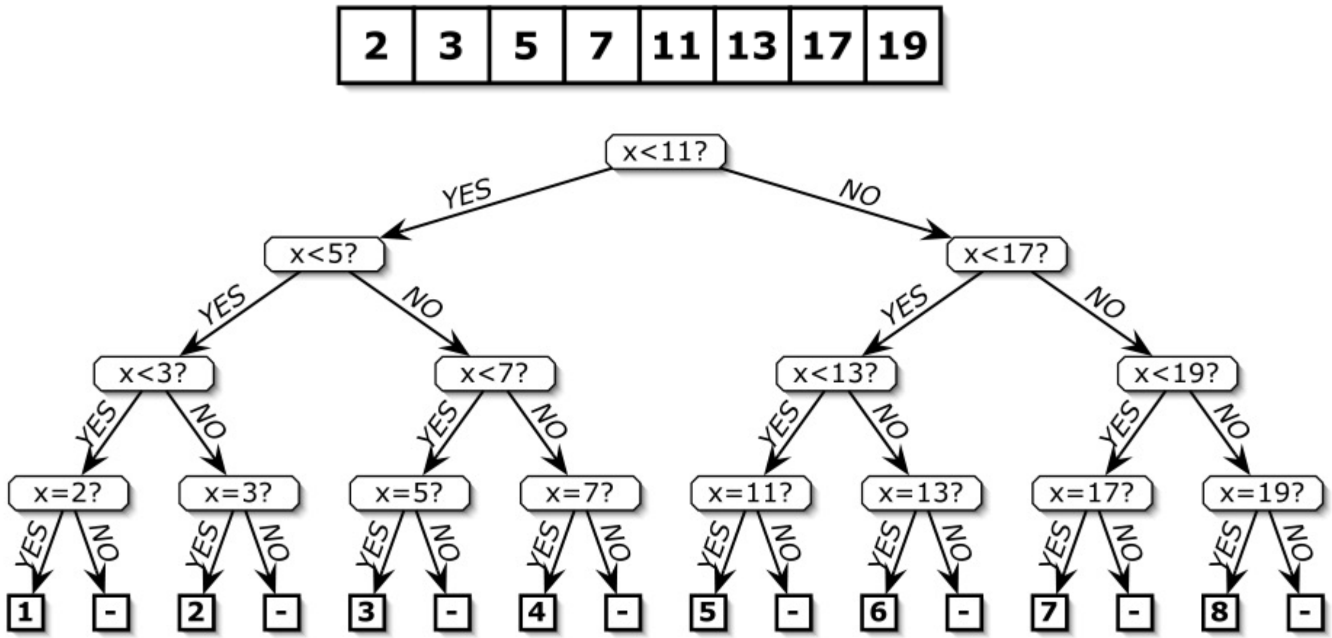
Proof: Consider a binary tree with n leaves and height h

(i) A binary tree of height *h* has at most 2^h leaves
(Known Fact: easy to prove by induction on *h*)

(ii) $\Rightarrow n \leq 2^h$

(iii) $\Rightarrow h \geq \log_2 n$

The decision-tree for binary search



Theorem: Any algorithm for finding location of given element in a sorted array of size n must have running time $\Omega(\log n)$ in the decision-tree model.

Proof:

- The algorithm must have at least $n+1$ different outputs.
  One for each element array & one to report search failure
- => The decision-tree has at least $n+1$ leaves.
- Any binary tree with $n+1$ leaves must have height $\Omega(\log(n+1)) = \Omega(\log n)$.
- => Algorithm has running time $\Omega(\log n)$.

Exercise on Coins

1. You are given a set of n coins, and are told that at most one (possibly none) of the n coins is either too heavy or too light (but you do not know which). You must determine which of the n coins is too heavy or too light, or report that none is defective. To do this test you have a scale; you place some of the coins on the left side of the scale and some of the coins on the right side. The scale indicates either (1) the left side is heavier, (2) the right side is heavier, or (3) both subsets have the same weight. It does not indicate how much heavier or lighter.

2. Use a decision tree argument to prove that the minimum number of scale comparisons is $\lceil \log_3(1 + 2n) \rceil$

3. Present a method to determine the defective coin using at most $c \log_3 n$ scale measurements, where c is a constant (independent of n). Assume that n is a power of 3.

Exercise on Coins - Number of Comparisons

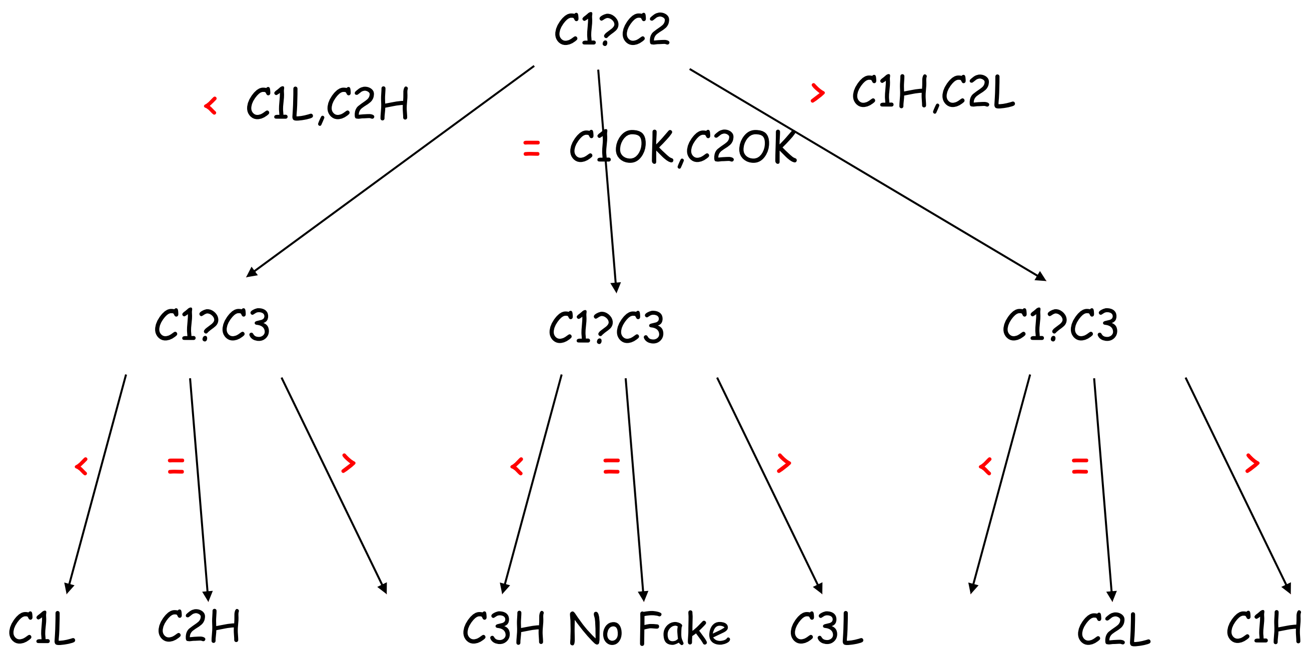
- The decision tree comes from the outcomes of using the scale.
- The total number of possible outcomes is $1 + 2n$:
  - If there is no defective coin, there is one possible outcome.
  - Otherwise, there are n coins, among which one is defective. There are n choices of the defective one. Since the defective coin can be either heavier or lighter, there are in all $2n$ possibilities for the case that there is one defective coin.
- Thus, the number of leaf nodes in the decision tree is $1 + 2n$.
- Since there are three possible outcomes for comparison (balanced, heavier left and heavier right) the decision tree is ternary. A ternary tree with $1 + 2n$ leaf nodes has height at least $\lceil \log_3(1 + 2n) \rceil$. Thus, we need at least $\lceil \log_3(1 + 2n) \rceil$ comparisons.

Exercise on Coins - Algorithm

- Present a method to determine the defective coin using at most $c \log_3 n$ scale measurements, where c is a constant (independent of n). Assume that n is a power of 3.
- Split the coins into three equal sets C1, C2, C3. With 2 comparisons you can determine 1 of the 7 possible outcomes: C1 H, C1 L, C2 H, C2 L, C3 H, C3 L, and no defective coin (on board).
- If no defective coin, then stop. Otherwise, repeat recursively step 1 with the set that contains the defective coin.
- Analysis. Since with 2 comparisons we reduce the size of the problem by 1/3, we have: $T(n) = T(n/3) + 2 = 2\log_3 n$.

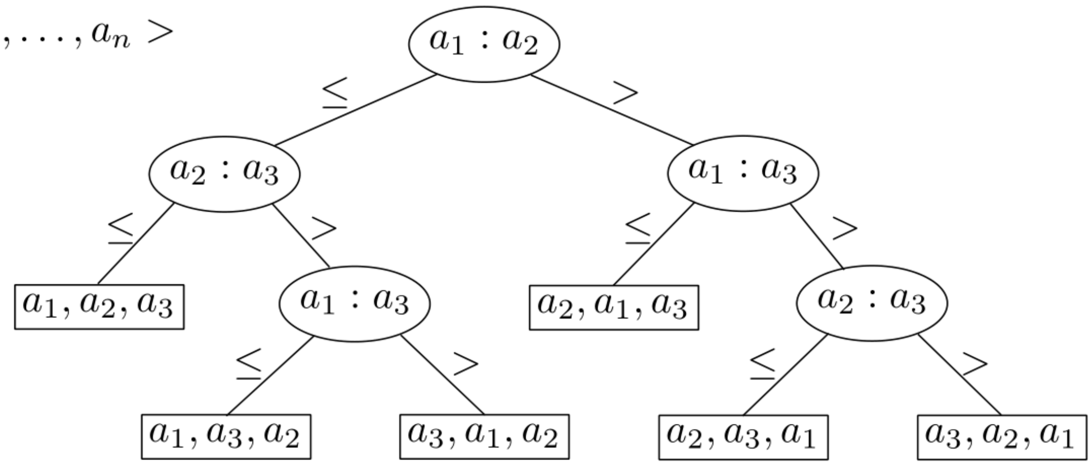
Exercise on Coins - Algorithm

- Present a method to determine the defective coin using at most $c \log_3 n$ scale measurements, where c is a constant (independent of n). Assume that n is a power of 3.



The decision-tree for sorting

Sort $< a_1, a_2, \ldots, a_n >$



Theorem: Any algorithm for sorting n elements must have running time $\Omega(n \log n)$ in the decision-tree model.

Proof: A sorting algorithm has at least $n!$ different outputs (one for each possible permutation on n items) => Decision-tree has at least $n!$ leaves =>

the height of the decision tree is at least $\log(n!) = \Theta(n \log n)$

(Stirling's approximation - exercise solved in Lecture 2)

=> Sorting algorithm requires at least $\Omega(n \log n)$ time

Can we do better?

Implication of the lower bound

- Anything "better" must be a ***non*** *comparison-based* algorithm.

Integer sorting

- Assumes that the elements are **integers** from 1 to k
- Running time is expressed as function of BOTH n and k.
- Both $n < k$ are $n > k$ possible.
- Functions of two variables might not be comparable to each other

We will now see **Counting Sort**.

It sorts n integers in the range $[1, k]$ in $\Theta(n+k)$ time.

Counting-sort(A, B, k)

Input: $A[1..n]$, where $A[j] \in \{1,2,\ldots,k\}$
Output: $B[1..n]$, sorted

```
for i ← 1 to k do                // counters C[1..k]
    C[i] ← 0;
end

for j ← 1 to n do
    C[A[j]] ← C[A[j]] + 1;       // C[i] = |{key = i}|
end

for i ← 2 to k do
    C[i] ← C[i] + C[i − 1];      // C[i] = |{key ≤ i}|
end

for j ← n to 1 do                // Move items into proper location
    B[C[A[j]]] ← A[j];
    C[A[j]] ← C[A[j]] − 1;
end
```

	1	2	3	4	5
A	4	2	1	4	2

Range of Integers [1,4]

B	1	2	2	4	4

We will work through the algorithm, showing that initial array $A[1..5]$ gets sorted to $B[1..5]$.
Pay attention to the fact that the algorithm will move the red entries on top into the red entries on bottom and the blue entries on top into the blue items on bottom.

A sorting algorithm is **STABLE** if two items with the same value appear in the same order in the sorted array as they did in the initial array.

Counting Sort is stable
Radix Sort (our next sorting algorithm), will depend upon the stability of counting sort

	1	2	3	4	5
A	4	2	1	4	2

	1	2	3	4
C				

B					

1^{st} Loop: Counter Initialization

	1	2	3	4	5
A	4	2	1	4	2

	1	2	3	4
C	0	0	0	0

B					

for $i \leftarrow 1$ **to** k **do**

$C[i] \leftarrow 0;$

end

2nd Loop: Count # of each item

	1	2	3	4	5
A	4	2	1	4	2

	1	2	3	4
C	0	0	0	1

B					

for $j \leftarrow 1$ **to** n **do**
  $C[A[j]] \leftarrow C[A[j]] + 1$; // $C[i] = |\{\text{key} = i\}|$
end

2nd Loop: Count # of each item

	1	2	3	4	5
A	4	2	1	4	2

	1	2	3	4
C	0	1	0	1

B					

for $j \leftarrow 1$ **to** n **do**

  $C[A[j]] \leftarrow C[A[j]] + 1$; // $C[i] = |\{\text{key} = i\}|$

end

2nd Loop: Count # of each item

	1	2	3	4	5
A	4	2	1	4	2

	1	2	3	4
C	1	1	0	1

B					

for $j \leftarrow 1$ **to** n **do**

  $C[A[j]] \leftarrow C[A[j]] + 1$; // $C[i] = |\{\text{key} = i\}|$

end

2nd Loop: Count # of each item

	1	2	3	4	5
A	4	2	1	4	2

	1	2	3	4
C	1	1	0	2

B					

for $j \leftarrow 1$ **to** n **do**

$\quad C[A[j]] \leftarrow C[A[j]] + 1$; // $C[i] = |\{\text{key} = i\}|$

end

2nd Loop: Count # of each item

	1	2	3	4	5
A	4	2	1	4	2

	1	2	3	4
C	1	2	0	2

B					

for $j \leftarrow 1$ **to** n **do**

$C[A[j]] \leftarrow C[A[j]] + 1$; // $C[i] = |\{\text{key} = i\}|$

end

3rd Loop: Aggregation

A

1	2	3	4	5
4	2	1	4	2

C

1	2	3	4
1	2	0	2

B

C'

1	3	0	2

for $i \leftarrow 2$ **to** k **do**

  $C[i] \leftarrow C[i] + C[i-1]$; // $C[i] = |\{\text{key} \leq i\}|$

end

3rd Loop: Aggregation

	1	2	3	4	5
A	4	2	1	4	2
B					

	1	2	3	4
C	1	2	0	2
C'	1	3	3	2

for $i \leftarrow 2$ **to** k **do**

   $C[i] \leftarrow C[i] + C[i-1]$; // $C[i] = |\{\text{key} \leq i\}|$

end

3rd Loop: Aggregation

	1	2	3	4	5
A	4	2	1	4	2
B					

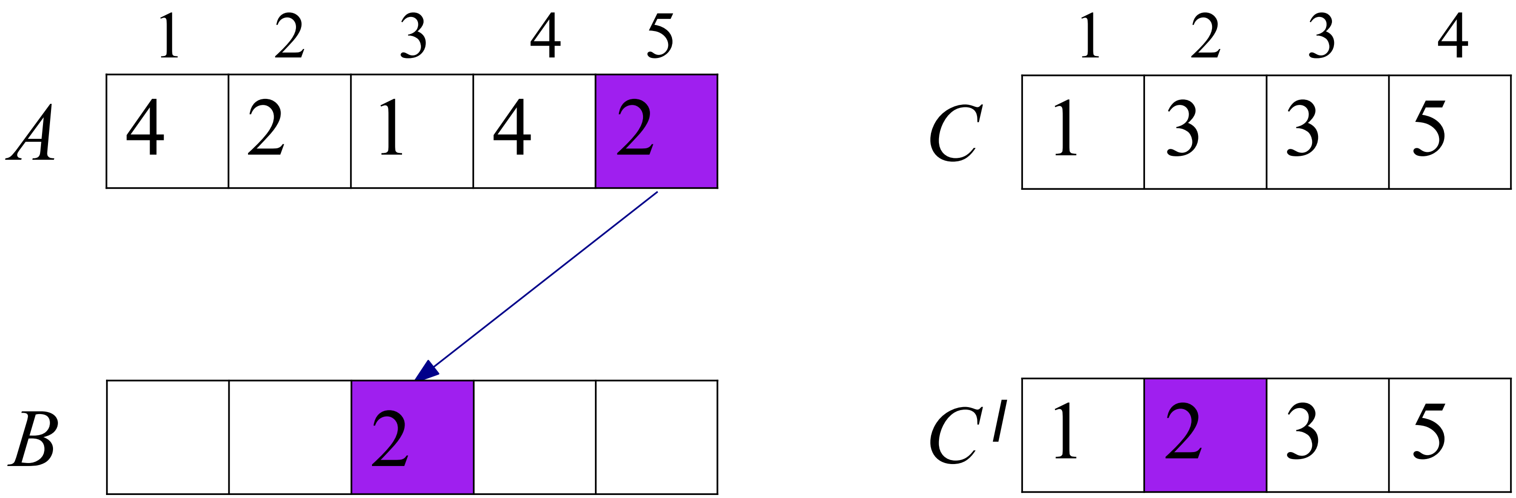
	1	2	3	4
C	1	2	0	2
C'	1	3	3	5

for $i \leftarrow 2$ **to** k **do**

  $C[i] \leftarrow C[i] + C[i-1]$; // $C[i] = |\{\text{key} \le i\}|$

end

4th Loop: Place items properly



for $j \leftarrow n$ **to 1 do**

$B[C[A[j]]] \leftarrow A[j]$;

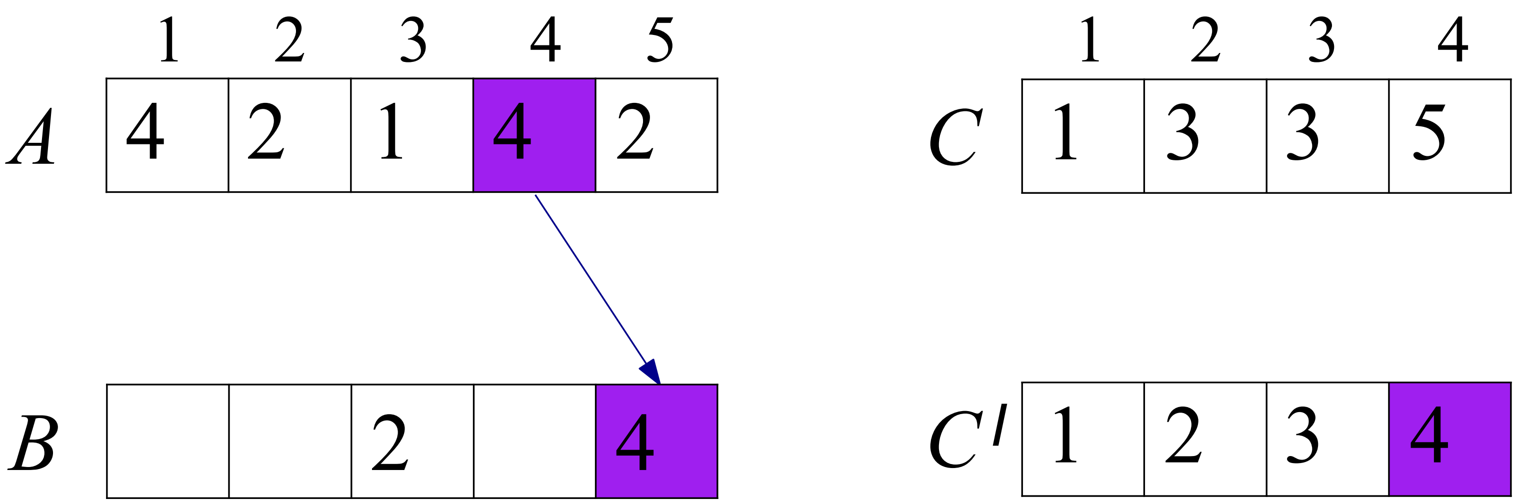
$C[A[j]] \leftarrow C[A[j]] - 1$;

end

Walk through A
from end to front.

At step j, i.e.,
when scanning x=A[j],
C'[x] will hold appropriate location in B of the rightmost currently unplaced x

4th Loop: Place items properly



for $j \leftarrow n$ **to 1 do**

$B[C[A[j]]] \leftarrow A[j]$;

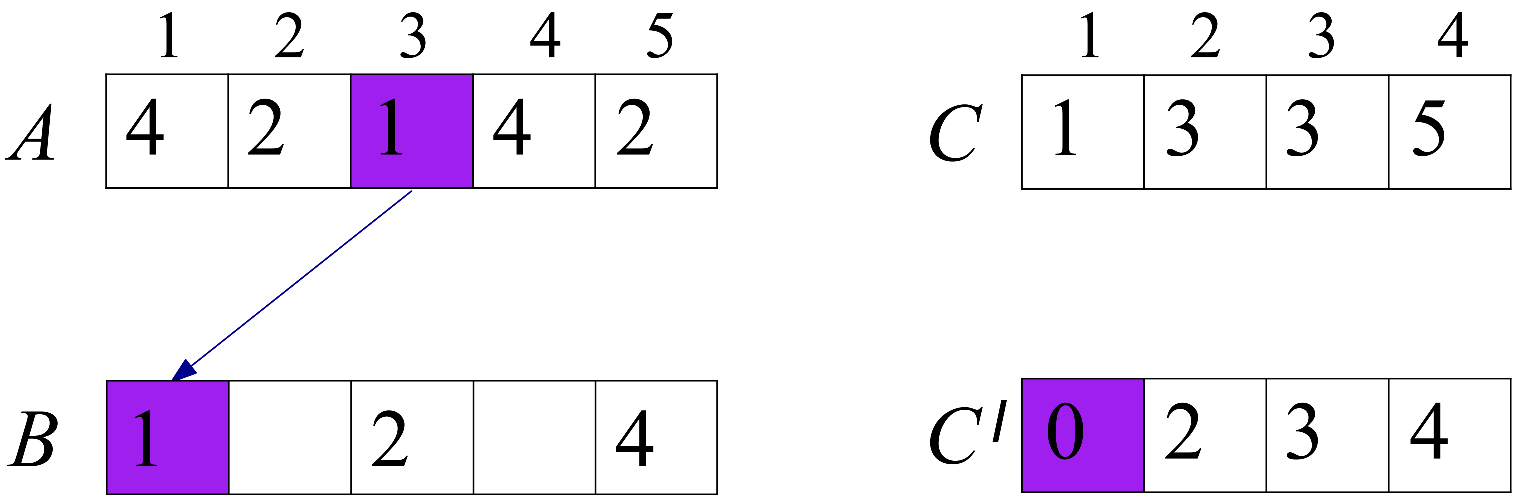
$C[A[j]] \leftarrow C[A[j]] - 1$;

end

Walk through A
from end to front.

At step j, i.e.,
when scanning x=A[j],
C′[x] will hold appropriate location in B of the rightmost currently unplaced x

4th Loop: Place items properly



for $j \leftarrow n$ **to** 1 **do**

$B[C[A[j]]] \leftarrow A[j]$;

$C[A[j]] \leftarrow C[A[j]] - 1$;

end

Walk through A
from end to front.

At step j, i.e.,
when scanning x=A[j],
C'[x] will hold appropriate
location in B of the rightmost
currently unplaced x

4th Loop: Place items properly

	1	2	3	4	5
A	4	2	1	4	2
B	1	2	2		4

	1	2	3	4
C	1	3	3	5
C'	0	1	3	4

for $j \leftarrow n$ **to 1 do**

$B[C[A[j]]] \leftarrow A[j];$

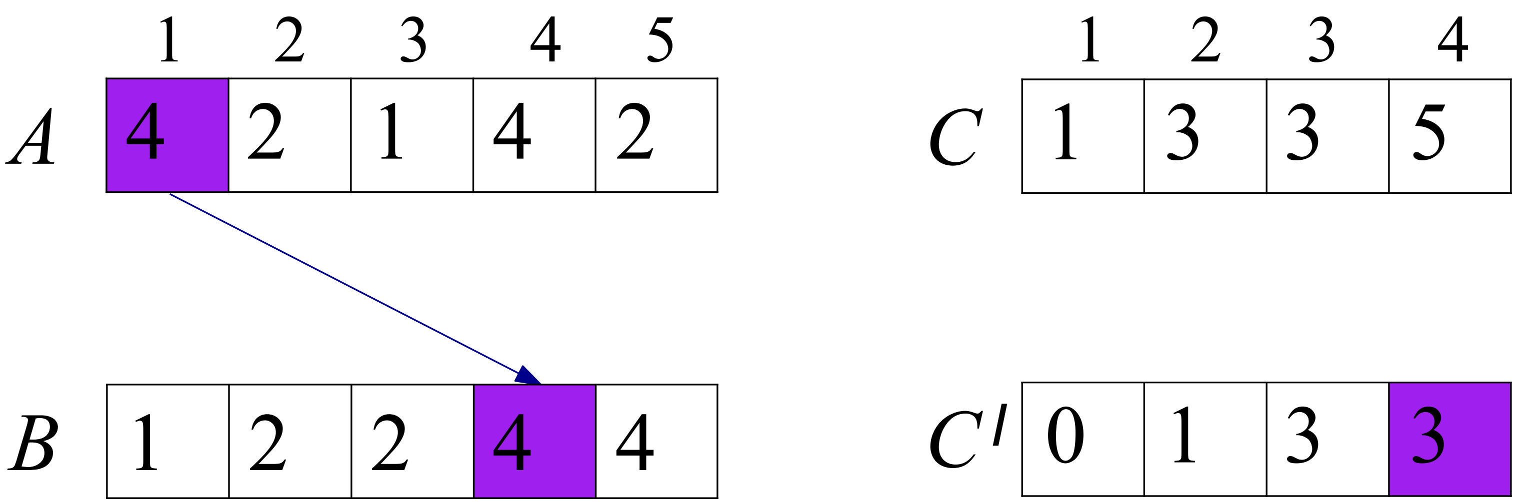
$C[A[j]] \leftarrow C[A[j]] - 1;$

end

Walk through A
from end to front.

At step j, i.e.,
when scanning x=A[j],
C′[x] will hold appropriate location in B of the rightmost currently unplaced x

4th Loop: Place items properly



for $j \leftarrow n$ **to** 1 **do**

$B[C[A[j]]] \leftarrow A[j];$

$C[A[j]] \leftarrow C[A[j]] - 1;$

end

Walk through A
from end to front.

At step j, i.e.,
when scanning x=A[j],
C′[x] will hold appropriate location in B of the rightmost currently unplaced x

Counting sort

```
Counting-Sort(A, B):
let C[0..k] be a new array
for i ← 0 to k
    C[i] ← 0
for j ← 1 to n
    C[A[j]] ← C[A[j]] + 1
    // C[i] now contains the number of i's
for i ← 1 to k
    C[i] ← C[i] + C[i − 1]
    // C[i] now contains the number of elements ≤ i
for j ← n downto 1
    B[C[A[j]]] ← A[j]
    C[A[j]] ← C[A[j]] − 1
```

Running time: $\Theta(n+k)$

Working space: $\Theta(n+k)$

Counting Sort is a stable sorting algorithm.

Exercise on Range Queries

- You are given an array A of n integers in the range $[1, k]$.
- What is the straightforward way to process the query: "how many elements are in the range $(a, b]$", where $1 \leq a < b \leq k$.
- Describe a pre-processing algorithm that generates a new array C. Using C you can answer any range query of the form "how many elements are in the range $(a, b]$" in constant time?
- Solution: You generate the same array $C[1..k]$ as in counting sort, i.e., the element $C[i]$ is the total number of elements smaller or equal to i. The answer to the query is: $C[b] - C[a]$.

Radix sort

2 3 2 9
5 4 5 7
3 6 5 7
5 8 3 9
3 4 3 6
2 7 2 0
5 3 5 5

```
Radix-Sort(A, d):
for i ← 1 to d
    use counting sort to sort array A on digit i
```

Input: Array of n numbers.
Each number has d digits
Each digit is in $[0, k-1]$

Output:
A sorted array

Radix sort

2	3	2	9		2	7	2	0
5	4	5	7		5	3	5	5
3	6	5	7		3	4	3	6
5	8	3	9	$\rightarrow$	5	4	5	7
3	4	3	6		3	6	5	7
2	7	2	0		2	3	2	9
5	3	5	5		5	8	3	9

Radix-Sort(A, d):
for $i \leftarrow 1$ to d
    use counting sort to sort array A on digit i

Input: Array of n numbers.
Each number has d digits
Each digit is in $[0, k-1]$

Output:
A sorted array

Radix sort

2329		2720		2720
5457		5355		2329
3657		3436		3436
5839	→	5457	→	5839
3436		3657		5355
2720		2329		5457
5355		5839		3657

Radix-Sort(A, d):

for $i \leftarrow 1$ to d

    use counting sort to sort array A on digit i

Input: Array of n numbers.
Each number has d digits
Each digit is in $[0, k-1]$

Output:
A sorted array

Radix sort

2 3 2 9		2 7 2 0		2 7 2 0		2 3 2 9
5 4 5 7		5 3 5 5		2 3 2 9		5 3 5 5
3 6 5 7		3 4 3 6		3 4 3 6		3 4 3 6
5 8 3 9	→	5 4 5 7	→	5 8 3 9	→	5 4 5 7
3 4 3 6		3 6 5 7		5 3 5 5		3 6 5 7
2 7 2 0		2 3 2 9		5 4 5 7		2 7 2 0
5 3 5 5		5 8 3 9		3 6 5 7		5 8 3 9

Radix-Sort(A, d):

for $i \leftarrow 1$ **to** d

  use counting sort to sort array A **on digit** i

Input: Array of n numbers.
Each number has d digits
Each digit is in $[0, k-1]$

Output:
A sorted array

Radix sort

2 3 2 9	2 7 2 0	2 7 2 0	2 3 2 9	2 3 2 9
5 4 5 7	5 3 5 5	2 3 2 9	5 3 5 5	2 7 2 0
3 6 5 7	3 4 3 6	3 4 3 6	3 4 3 6	3 4 3 6
5 8 3 9	5 4 5 7	5 8 3 9	5 4 5 7	3 6 5 7
3 4 3 6	3 6 5 7	5 3 5 5	3 6 5 7	5 3 5 5
2 7 2 0	2 3 2 9	5 4 5 7	2 7 2 0	5 4 5 7
5 3 5 5	5 8 3 9	3 6 5 7	5 8 3 9	5 8 3 9

Radix-Sort(A, d):

for $i \leftarrow 1$ to d

    use counting sort to sort array A on digit i

Input: Array of n numbers.
Each number has d digits
Each digit is in $[0, k-1]$

Output:
A sorted array

Radix sort: Correctness

Proof: (induction on digit position)

- Assume that the numbers are sorted by their low-order $i-1$
Digits

- Sort on digit i
  After the sort

2	7	2	0		2	3	2	9
2	3	2	9		5	3	5	5
3	4	3	6		3	4	3	6
5	8	3	9	$\rightarrow$	5	4	5	7
5	3	5	5		3	6	5	7
5	4	5	7		2	7	2	0
3	6	5	7		5	8	3	9

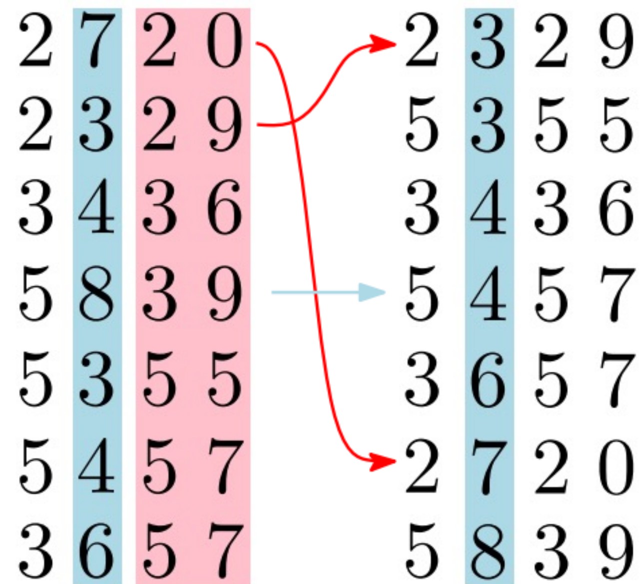
Radix sort: Correctness

Proof: (induction on digit position)

- Assume that the numbers are sorted by their low-order $i-1$ Digits

- Sort on digit i
  After the sort

  - Two numbers that differ on digit i are correctly sorted by their low-order i digits



Radix sort: Correctness

Proof: (induction on digit position)

- Assume that the numbers are sorted by their low-order $i-1$ digits

- Sort on digit i
  After the sort

  - Two numbers that differ on digit i are correctly sorted by their low-order i digits

  - Because of STABILITY of counting sort, two numbers that have same i-th digit are in the same order in output as they were in input
    $\Rightarrow$ they are correctly sorted by their low-order i digits

2 7 2 0	2 3 2 9
2 3 2 9	5 3 5 5
3 4 3 6	3 4 3 6
5 8 3 9	5 4 5 7
5 3 5 5	3 6 5 7
5 4 5 7	2 7 2 0
3 6 5 7	5 8 3 9

Radix sort: Running time analysis

Q: What is running time of Radix Sort?

n = # integers, k = # of values each digit can take, d= # digits

Analysis:

- Counting sort takes $\Theta(n+k)$ time.
- Counting sort is run d times
  => total run time is $\Theta(d\,(n+k))$

Example:

- n 16-bit binary words
- Bit takes only two values so $k=2$.
- Algorithm takes $\Theta\big(16(n+2)\big)$

Summary of sorting algorithms

	Insertion sort	Merge sort	Quicksort	Heapsort	Radix sort
Running time	$\Theta(n^2)$	$\Theta(n \log n)$	$\Theta(n \log n)$	$\Theta(n \log n)$	$\Theta\big(d(n+k)\big)$
Randomized	No	No	Yes	No	No
Working space	$\Theta(1)$	$\Theta(n)$	$\Theta(\log n)$	$\Theta(1)$	$\Theta(n+k)$
Comparison-based	Yes	Yes	Yes	Yes	No
Stable	Yes	Yes	No	No	Yes
Other Properties					
Cache performance	Good	Good	Good	Bad	Bad
Parallelization	No	Excellent	Good	No	No

Note: **Our** versions of Insertion Sort and Mergesort are stable. Possible to write unstable versions. Also, our version of Quicksort is unstable. If allowed extra memory space, it's possible to write a stable version of Quicksort.

Exercise on List Merging

You are given n/k lists such that:

(i) Each list contain k real numbers

(ii) for $i = 1$ to n/k, the elements in list $i - 1$ are all less than all the elements in list i.

The obvious algorithm to fully sort these items is to sort each list separately and then concatenate the sorted lists together.

- What is the running time of this approach (in terms of comparisons).
- Sorting each list requires $k \log k$ comparisons. Since, there n/k lists, the running time is $\frac{n}{k} k \log k = n \log k$.
- Use the decision tree model to show that this approach is asymptotically optimal.

Exercise on List Merging (cont)

I have to compute the number of leafs of the corresponding decision tree.

- How many possible arrangements (permutations) exist for each list?
- $k!$
- How many possible arrangements (permutations) exist in total?
- $(k!)^{n/k}$
- What is the height for a binary tree to accommodate all possible arrangements?
- $\log\left((k!)^{n/k}\right) = (n/k)\log(k!) = (n/k)\times\Theta(k\log k) = \Theta(n\log k)$