

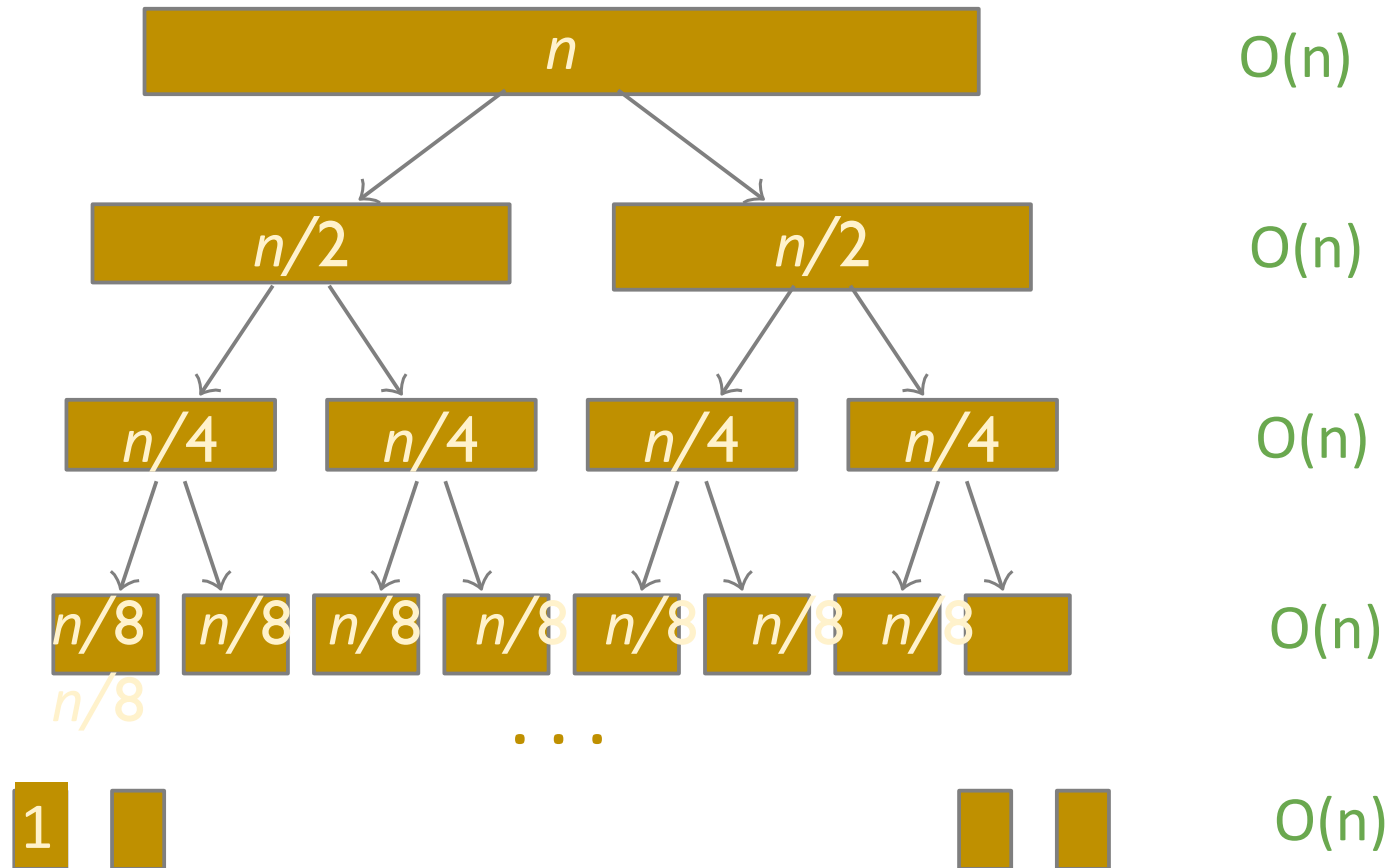
Lower-bound on Comparison-Based Sorting

Lecture 19

Can we sort **without** comparing objects?

Merge sort: recursion tree

Work at each level: $O(n)$



Total: $O(n) * \log n = O(n \log n)$

Algorithm mergeSort ($A[1..n]$)

if $n = 1$: return A

$m \leftarrow \lfloor n/2 \rfloor$

$B \leftarrow \text{mergeSort}(A[1 \dots m])$

$C \leftarrow \text{mergeSort}(A[m + 1 \dots n])$

$A' \leftarrow \text{merge}(B, C)$

return A'

The running time of $\text{mergeSort}(A[1 \dots n])$ is $O(n \log n)$.

Can we do better?

Lower bound for Comparison-based sorting

Definition

A *comparison-based sorting algorithm* sorts objects by comparing pairs of them.

Example:

Selection sort and merge sort are comparison based.

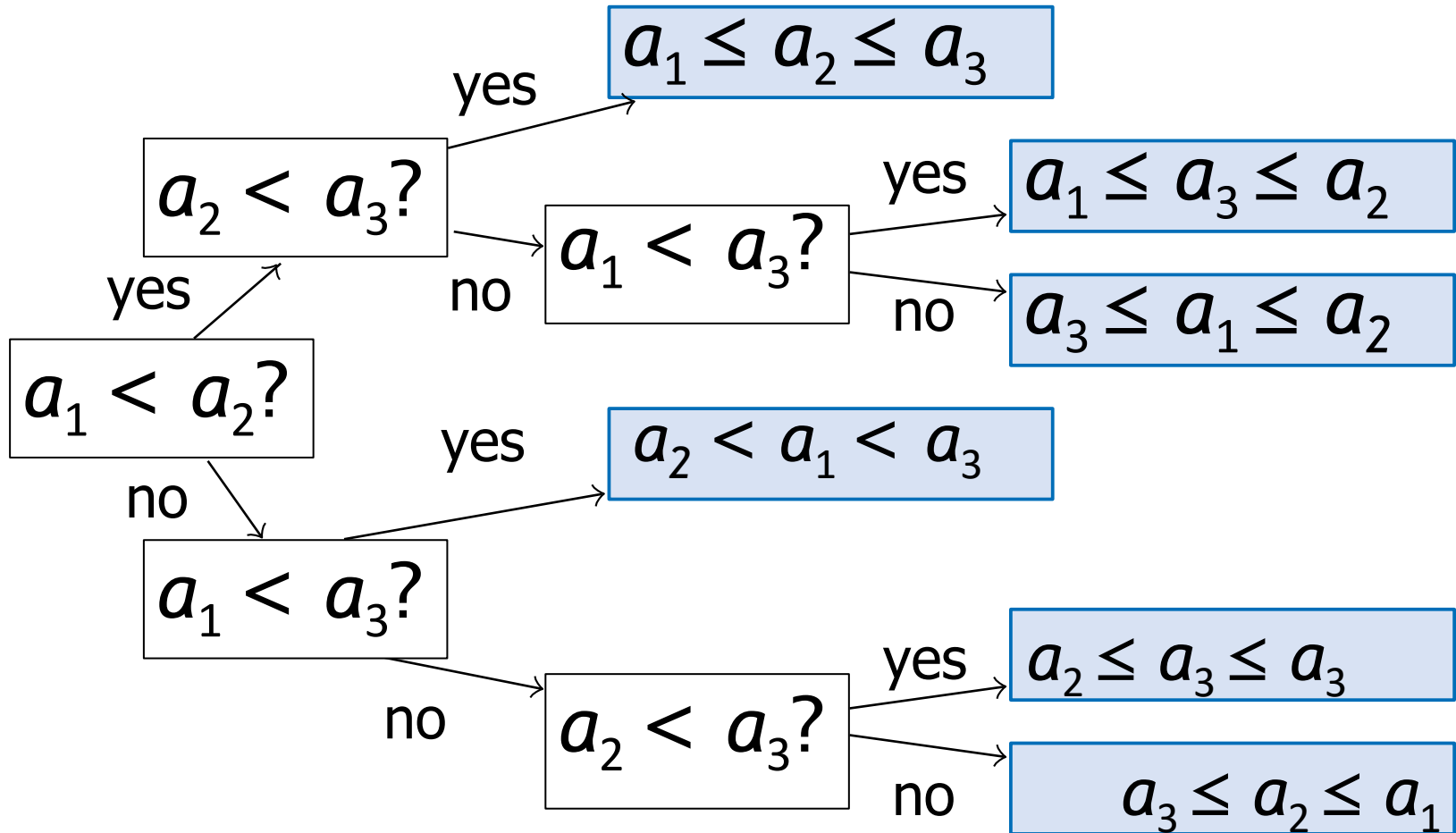
Lemma

Any comparison-based sorting algorithm performs $\Omega(n \log n)$ comparisons in the worst case to sort n objects.

In other words

For any comparison-based sorting algorithm, there exists an input array $A[1 \dots n]$ such that the algorithm performs **at least** $\Omega(n \log n)$ comparisons to sort A .

Decision Tree for deciding the order of 3 objects



Estimating max leaf depth

- The number of leaves ℓ in the tree must be $n!$ (the total number of permutations of n array elements)
- For the worst-case input the number of comparisons made is equal to the maximum depth d of this tree
- The max depth of any node in a binary tree with ℓ leaves is at least $O(\log \ell)$: the minimum happens when the binary tree is complete. In all other incomplete binary trees the max depth will be $> \log \ell$.

$$d \geq \log_2 \ell \text{ (or, equivalently, } 2^d \geq \ell \text{)}$$

- The number of leaves ℓ in our decision tree is $n!$
- Let's show that:

$$\log_2(n!) = \Omega(n \log n)$$

Lemma

At least!

$$\log_2(n!) = \Omega(n \log n)$$

Proof

$$\begin{aligned}\log_2(n!) &= \log_2(1 \cdot 2 \cdot \dots \cdot n) \\ &= \log_2 1 + \log_2 2 + \dots + \log_2 n \\ &\geq \log_2(n/2) + \dots + \log_2 n \\ &\geq (n/2) \log_2(n/2) = \Omega(n \log n)\end{aligned}$$

Consider only
the smallest
element of the
sum

Consider only
the second half
of the sum



Corollary

Any **comparison-based sorting** algorithm performs (at least) $\Omega(n \log n)$ comparisons on the worst case input of size n .

Merge Sort

The running time of `MergeSort(A[1 . . . n])` is $O(n \log n)$.

This running time is *optimal* if we consider **sorting based on comparing pairs of numbers**

Sorting not based on comparison:
can be faster

Example 1: sorting small integers

	0	1	2	3	4	5	6	7	8	9	10	11
A	2	3	2	1	3	2	2	3	2	2	2	1

Non-comparison based sorting?

Sorting small integers

	0	1	2	3	4	5	6	7	8	9	10	11
A	2	3	2	1	3	2	2	3	2	2	2	1

	1	2	3
Count	2	7	3

Non-comparison based sorting

Sorting small integers

	0	1	2	3	4	5	6	7	8	9	10	11
A	2	3	2	1	3	2	2	3	2	2	2	1

	1	2	3
Count	2	7	3

[illegible]

Sorting small integers

we have sorted these numbers
without actually comparing them!

[illegible]

Count Sort

- Assume that all elements of $A[0 \dots N-1]$ are integers from 1 to M .
- By a single scan of the array A , count the number of occurrences of each $1 \leq k \leq M$ in the array A and store it in $Count[k]$.
- Using this information, fill in the sorted array A' .

CountSort($A[0 \dots N-1]$)

$Count[1 \dots M] \leftarrow [0, \dots, 0]$ # to store counts of $A[i]$

for i from 0 to $N-1$:

$Count[A[i]] \leftarrow Count[A[i]] + 1$

number k appears $Count[k]$ times in A

$Pos[1 \dots M] \leftarrow [0, \dots, 0]$

$Pos[1] \leftarrow 0$

for j from 2 to M :

$Pos[j] \leftarrow Pos[j-1] + Count[j-1]$

$A'[0 \dots n-1] \leftarrow [0, \dots, 0]$ # to store sorted values of A

number k will occupy range $[Pos[k] \dots Pos[k+1] - 1]$

for i from 0 to $N-1$:

$A'[Pos[A[i]]] \leftarrow A[i]$

$Pos[A[i]] \leftarrow Pos[A[i]] + 1$

CountSort($A[0 \dots n-1]$)

$Count[1 \dots M] \leftarrow [0, \dots, 0]$ # to store counts of $A[i]$

for i from 0 to $N-1$:

$Count[A[i]] \leftarrow Count[A[i]] + 1$

...

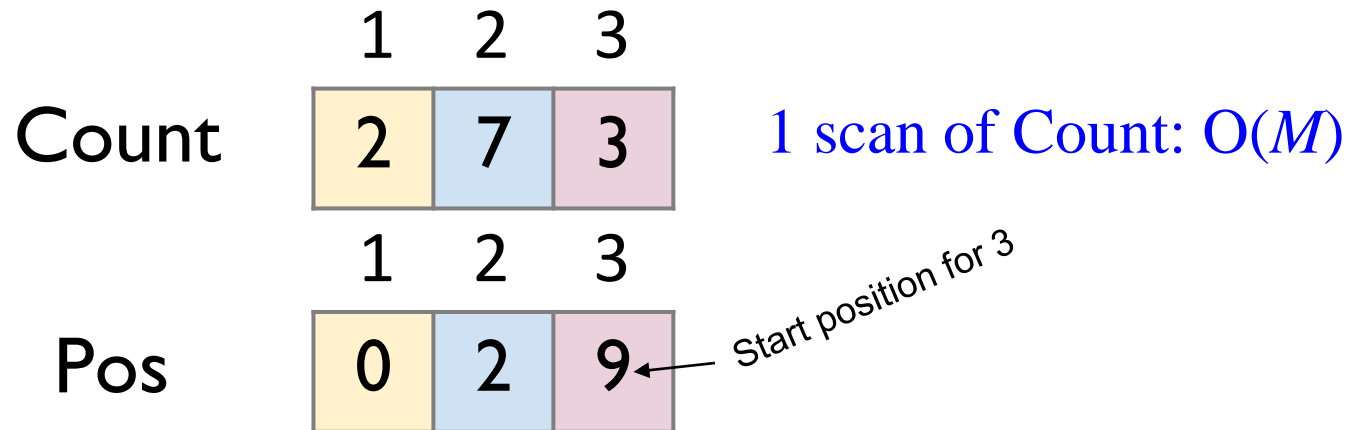
	0	1	2	3	4	5	6	7	8	9	10	11
A	2	3	2	1	3	2	2	3	2	2	2	1

1 scan of A: $O(N)$

	1	2	3
Count	2	7	3

CountSort($A[0 \dots n-1]$)

...



number k appears $Count[k]$ times in A

$Pos[1 \dots M] \leftarrow [0, \dots, 0]$

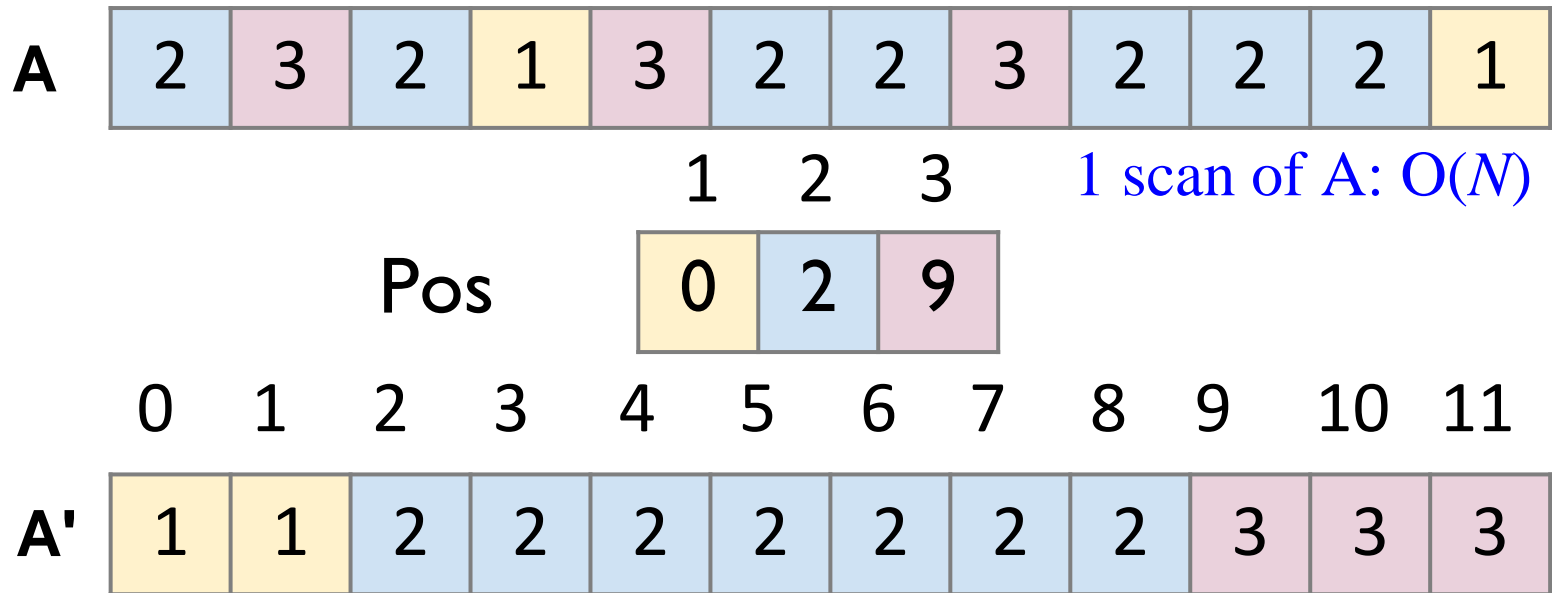
$Pos[1] \leftarrow 0$

for j from 2 to M :

$Pos[j] \leftarrow Pos[j - 1] + Count[j - 1]$

CountSort($A[0 \dots n-1]$)

...



number k will occupy range $[Pos[k] \dots Pos[k + 1] - 1]$

for i from 0 to $N-1$:

$A'[Pos[A[i]]] \leftarrow A[i]$

$Pos[A[i]] \leftarrow Pos[A[i]] + 1$

Lemma

Provided that all elements of $A[1 \dots n]$ are integers from 1 to M , `countSort(A)` sorts A in time $O(N + M)$.

Corollary

If $M = O(N)$, then the running time of Count Sort is $O(N)$.

Non-comparison-based sorting: another approach

Example 2: sorting strings

	0	1	2	3	4	5	6	7
A	beds	dust	cabs	cubs	bass	abba	stud	best

- Consider an array of Strings
- We can use a comparison-based sort to sort these, utilizing the *compareTo()* method of String class

Example 2: sorting strings (char arrays)

A

	0	1	2	3	4	5	6	7	← Index in A
0	b	d	c	c	b	a	s	b	
1	e	u	a	u	a	b	t	e	
2	d	s	b	b	s	b	u	s	
→ 3	s	t	s	s	s	a	d	t	

Position of a character

- What if we think of a String as a char array
- Consider the positions in each String, from rightmost to leftmost, and the character value at that position
- Instead of comparing these characters to one another, we will use each as an index to a "bin" (actually a Queue) of Strings
- We will have an array of Queues indexed on the ASCII characters

Assign to buckets based on $A[i][3]$

A

	0	1	2	3	4	5	6	7
0	b	d	c	c	b	a	s	b
1	e	u	a	u	a	b	t	e
2	d	s	b	b	s	b	u	s
3	s	t	s	s	s	a	d	t

Buckets indexed by each character value

'a'	abb a
'b'	
'c'	
'd'	stud d
'e'	
...	
's'	bed s , cab s , cub s , bass s ,
't'	dust t , best t
'u'	
...	

Each bucket
is a Queue
of Strings

Transfer back to A in this order

A

	0	1	2	3	4	5	6	7
0	a	s	b	c	c	b	d	b
1	b	t	e	a	u	a	u	e
2	b	u	d	b	b	s	s	s
3	a	d	s	s	s	s	t	t

We copy the data in order from the Queues back into the array

Buckets indexed by each character value

'a'	abba
'b'	
'c'	
'd'	stud
'e'	
...	
's'	beds, cabs, cubs, bass,
't'	dust, best
'u'	
...	

Assign to buckets based on $A[i][2]$

A

	0	1	2	3	4	5	6	7
0	a	s	b	c	c	b	d	b
1	b	t	e	a	u	a	u	e
2	b	u	d	b	b	s	s	s
3	a	d	s	s	s	s	t	t

Buckets indexed by each character value

'a'	
'b'	abba, cab s , cub s
'c'	
'd'	bed s
'e'	
...	
's'	bass, dust, best
't'	
'u'	stud
...	

Transfer back to A in this order

A

	0	1	2	3	4	5	6	7
0	a	c	c	b	b	d	b	s
1	b	a	u	e	a	u	e	t
2	b	b	b	d	s	s	s	u
3	a	s	s	s	s	t	t	d

Buckets indexed by each character value

'a'	
'b'	abba, cabs, cubs
'c'	
'd'	beds
'e'	
...	
's'	bass, dust, best
't'	
'u'	stud
...	

Assign to buckets based on $A[i][1]$

A

	0	1	2	3	4	5	6	7
0	a	c	c	b	b	d	b	s
1	b	a	u	e	a	u	e	t
2	b	b	b	d	s	s	s	u
3	a	s	s	s	s	t	t	d

Buckets indexed by each character value

'a'	ca b s, ba s s
'b'	a b ba
'c'	
'd'	
'e'	be d s, be s t
...	
's'	
't'	st u d
'u'	cu b s, du s t
...	

Transfer back to A in this order

A

	0	1	2	3	4	5	6	7
0	c	b	a	b	b	s	c	d
1	a	a	b	e	e	t	u	u
2	b	s	b	d	s	u	b	s
3	s	s	a	s	t	d	s	t

Buckets indexed by each character value

'a'	cabs, bass
'b'	abba
'c'	
'd'	
'e'	beds, best
...	
's'	
't'	stud
'u'	cubs, dust
...	

Assign to buckets based on $A[i][0]$

A

	0	1	2	3	4	5	6	7
0	c	b	a	b	b	s	c	d
1	a	a	b	e	e	t	u	u
2	b	s	b	d	s	u	b	s
3	s	s	a	s	t	d	s	t

Buckets indexed by each character value

'a'	abba
'b'	bass, beds, best
'c'	cabs, cubs
'd'	dust
'e'	
...	
's'	stud
't'	
'u'	
...	

Transfer back to A in this order

A								
	0	1	2	3	4	5	6	7
0	a	b	b	b	c	c	d	s
1	b	a	e	e	a	u	u	t
2	b	s	d	s	b	b	s	u
3	a	s	s	t	s	s	t	d


Note that this is the final order

Buckets indexed by each character value

'a'	abba
'b'	bass, beds, best
'c'	cabs, cubs
'd'	dust
'e'	
...	
's'	stud
't'	
'u'	
...	

We sorted strings without comparing them

	0	1	2	3	4	5	6	7
A	beds	dust	cabs	cubs	bass	abba	stud	best



	0	1	2	3	4	5	6	7
A	abba	bass	beds	best	cabs	cubs	dust	stud

- Why did this work?
- Each time we put the data into bins we are **sorting based on that character**
- Strings that are the same in characters 0 to K for some K will be already distinguished (ordered) by character K+1 and that order will not change when considering characters from K down to 0

The algorithm is called *Radix Sort*

- Radix = “The base of a number system” (Webster’s dictionary)
- History: used already in 1890 U.S. census by Hollerith, became popular in 1920s with sorting data on punch cards
- Idea: Bin Sort on each digit, bottom up.

Strings of different lengths

- Note that direct comparison of Strings goes from left to right but in Radix Sort we go from right to left
- What if the Strings are of different lengths?
- Example:
 - A[0] = HELP
 - A[1] = HELPS
 - A[2] = HELPED
- Note that A[0].length() == 4, A[1].length() == 5 and A[2].length() == 6
- What can we do to handle this situation?

Solution: padding

- We can "pad" the smaller Strings to make them all the same length
- Alphabetically, we would expect
HELP < HELPED < HELPS
- On which side should we add padding (left or right)?
- The prefix of all 3 Strings is "HELP"
- The suffix (right side) is what distinguishes them
- We can get the sort to work correctly if we pad on the right of smaller Strings with characters that are less than any valid characters in a word

A[0] = **HELP@@**

A[1] = **HELPS@**

A[2] = **HELPED**

Running time of Radix Sort

- We must iterate through each position in a String (at most M such positions)
- For each position we must iterate through all of the Strings, putting each into a bucket (N such strings)
- We must then remove them from the buckets and put them back into the array (N strings)
- If the max String length is M , and the length of the array is N , this will yield a run-time of $O(MN)$
- If we consider M to be a constant then this run-time will be $O(N)$

Radix Sort of integers

- Input array:
126, 328, 636, 341, 416, 131, 328
- Bin Sort on lower digit:
341, 131, 126, 636, 416, 328, 328
- Bin Sort result on next-higher digit:
416, 126, 328, 328, 131, 636, 341
- Bin Sort that result on highest digit:
126, 131, 328, 328, 341, 416, 636

Radix Sort: performance notes

- Considerable overhead:
 - Space overhead for the bins ($O(N)$)
 - Time overhead for the copying (**not in place**)
 - Also, overhead in extracting the individual values
 - For String this is not a problem
 - For int isolating each digit requires some math (i.e. overhead)
- Also, even though this is MN vs. $N \log N$ for comparison based sort, the value of M may be larger than $\log N$ for small or medium sized arrays
 - Ex: Sorting 1000 Strings of maximum length 15 requires $15 \times N$ work for Radix Sort while in this case $\log N$ is only ~ 10

Applicability of Radix Sort

- Radix Sort is not a generally applicable sorting algorithm
 - We must be able to break our key into separate values that can be binned into a limited-size array
- Comparison-based sorts allow for arbitrary algorithms to be used for the comparison – perhaps even utilizing multiple data values
- In some situations Radix Sort can be effective
- It also enables us to look at sorting in a different way
 - Later and also in CS 1501 you will look at other algos which take an approach similar to Radix Sort (Hashing, Tries)

Summary on sorting so far

- **Merge sort** uses the divide-and-conquer strategy to sort an N -element array in time $O(N \log N)$
- No comparison-based algorithm can do this (asymptotically) faster
- One **can** do faster if something special is known about the input in advance (e.g., each element of an array is a small integer or a limited-length sequence)