Design & Analysis of Algorithms
COMP 482 / ELEC 420

John Greiner

Sorting

Should already know some sorting algorithms:
E.g., Insertion sort, Selection sort, Quicksort, Mergesort, Heapsort

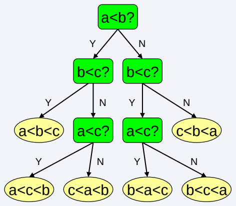
We'll concentrate on ideas not seen in previous courses:

- Lower bounds on general-purpose sorting
- Quicksort probabilistic analysis
- Special-purpose linear-time sorting

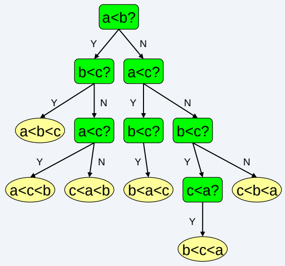
To do:
[CLRS] 7-8
#3

Comparison Trees

Comparisons used to determine order of 3 distinct elements.
Leaves correspond to all possible permutations.



Minimal



Not Minimal

How do Comparison Trees Relate to Sorting?

- Any sorting algorithm must be able to reorder any permutation.
- Sorting algorithm's behavior corresponds to some comparison tree.

Lower Bounds on Sorting

? How many leaves in a comparison tree? ?

? How many levels? ?

So, any general sorting algorithm must make $\Omega(n \lg n)$ comparisons on at least some inputs.

Quicksort – Immutable

```
qsort(A):
  if |A| ≤ 1
      return A
  else
      pivot = some element of A
      L = [x in A : x<pivot]
      G = [x in A : x>pivot]
      return qsort(L) + pivot + qsort(G)
```

Quicksort – Mutable

Algorithm in [CLRS] 7.

Advantages:

- Constant factor less time.
- Constant factor less space.

Disadvantage:

- More difficult to understand.

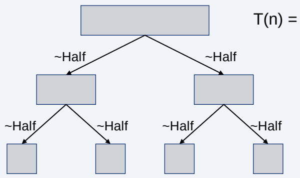
Quicksort Pivot Best Case

Time depends on a good (lucky?) choice of pivot.

? Best case? ?

? What is resulting recurrence & bound? ?

$T(n) = 2T(n/2) + \Theta(n) = \Theta(n \lg n)$



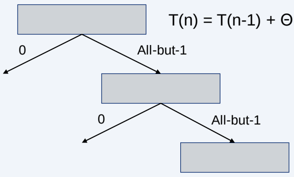
Quicksort Pivot Worst Case

Time depends on a good (lucky?) choice of pivot.

? Worst case? ?

? What is resulting recurrence & bound? ?

$T(n) = T(n-1) + \Theta(n) = \Theta(n^2)$



Quicksort Worst Case

For deterministic pivot-choosing algorithms:

O(1): Can be unlucky → $O(n^2)$ quicksort worst case

O(n): Can find median (as we'll see soon) → O(n log n) quicksort

We took shortcut by assuming the 0 & all-but-1 split is worst.

Intuitively obvious, but could prove this, e.g.,

$T(n) = \max_{q=0..n-1} (T(q) + T(n-q-1)) + \Theta(n)$

...which can be solved to...

$T(n) = \Theta(n^2)$

Quicksort Average Case Intuition

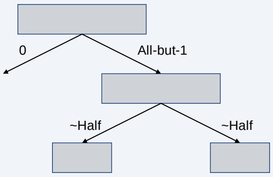
Average case is more like the best case than the worst case.

Two interesting cases for intuition:

1. Any sequence of partitions with the same ratios, such as 1/2::1/2 (the best case), 1/3::2/3, or even 1/100::99/100.
   - As have previously seen, the recursion tree depth is still logarithmic, which leads to the same bound.
   - Thus, the "good" cases don't have to be that good.

Quicksort Average Case Intuition

2. Sequence of alternating worst case and best case partitions.
   - Each pair of these partitions behaves like a best case partition, except with higher overhead.



- Thus, can tolerate having some bad partitions.

Quicksort Average Case Overview

Already have Ω(n log n) bound.

Want to obtain O(n log n).

Can overestimate in analysis.

Always look for ways to simplify!

Quicksort Average Case: Partitioning

Observe: Partitioning dominates Quicksort's work.

- **Partitioning includes the comparisons – the interesting work.**
- Every Quicksort call partitions – except the O(1)-time base cases.
- Partitioning more expensive than joining.

? How many partitions are done in the sort? ?

n-1 = O(n)

Observe: Comparisons dominate partitioning work.

- Each partition's time ∝ that partition's #comparisons.

So, concentrate on time spent comparing.

Quicksort Average Case: Comparisons

# of comparisons in partitions in n-element Quicksort

$C(n) = \quad \ldots$

# of comps. in this partition? $n-1$

# of comps. in two recursive calls? $C(i)+C(n-i-1)$

Average this over all partition sizes. $\frac{1}{n}\sum_{i=0}^{n-1}\left(C(i)+C(n-i-1)\right)$

$$C(n) = n-1+\frac{1}{n}\sum_{i=0}^{n-1}\left(C(i)+C(n-i-1)\right)$$

$$= O(n \ln n)$$

Quicksort Average Case: Comparisons

Rather than analyzing the time for each partition, and then summing, instead directly analyze the total number of comparisons performed over the whole sort.

Quicksort's behavior depends on only values' ranks, not values themselves.

Z	= set of values in array input A
z_i	= i^{th}-ranked value in Z
Z_{ij}	= set of values $\{z_i,\ldots,z_j\}$

Quicksort Average Case: Comparisons

Let X_{ij} = I{z_i is compared to z_j}

Total comparisons $X = \sum_{i=1}^{n-1}\sum_{j=i+1}^{n} X_{ij}$

Each pivot is selected at most once, so each z_i,z_j pair is compared at most once.

$$E[X] = E\left[\sum_{i=1}^{n-1}\sum_{j=i+1}^{n} X_{ij}\right]$$

$$= \sum_{i=1}^{n-1}\sum_{j=i+1}^{n} E[X_{ij}] \quad \text{by linearity of expectation}$$

$$= \sum_{i=1}^{n-1}\sum_{j=i+1}^{n} \Pr\{z_i \text{ is compared to } z_j\} \quad \text{by definition}$$

Quicksort Average Case: Comparisons

What is this probability?

Consider arbitrary i,j and corresponding Z_{ij}.

Z_{ij} need not correspond to a partition executed during the sort.

Claim: z_i and z_j are compared ↔ either is the first element in Z_{ij} to be chosen as a pivot.

Proof: Which is first element in Z_{ij} to be chosen as pivot?

- If z_i, then that partition must start with at least all the elements in Z_{ij}. Then z_i compared with all the elements in that partition (except itself), including z_j.
- If z_j, similar argument.
- If something else, the resulting partition puts z_i and z_j into separate sets (without comparing them), so that no future Quicksort or partition call will consider both of them.

Quicksort Average Case: Comparisons

Now, compute the probability:

Pr{z_i is compared to z_j}

= Pr{z_i or z_j is first pivot chosen from Z_{ij}} just explained

= Pr{z_i is first pivot chosen from Z_{ij}} + Pr{z_j is first pivot chosen from Z_{ij}} mutually exclusive possibilities

= 1/(j-i+1) + 1/(j-i+1)
= 2/(j-i+1)

Quicksort Average Case: Total

Plug this back into the sum:

$$E[X] = \sum_{i=1}^{n-1} \sum_{j=i+1}^{n} \frac{2}{j-i+1}$$

$$= \sum_{i=1}^{n-1} \sum_{k=2}^{n-i+1} \frac{2}{k}$$ Simplify with a change of variable, k=j-i+1.

$$< \sum_{i=1}^{n-1} \sum_{k=1}^{n} \frac{2}{k}$$ Simplify and overestimate by adding terms.

$$= \sum_{i=1}^{n-1} O(\log n)$$

$$= O(n \log n)$$

Quicksort Input Distribution

? Are all inputs equally likely **in practice**? ?

Linear-Time Sorting

In limited circumstances, can avoid comparison-based sorting, & thus do better than previous lower bound!

Must rely on some restriction on inputs.

Counting Sort

Limit data to a small discrete range: e.g., 0,…,5.
Let m = size of range.

	Input=	2	5	0	1	2	3	0
Count instances of each possible element:	Count=	2	1	2	1	0	1	
Produce Count[i] instances of each i:	Output=	0	0	1	2	2	3	5

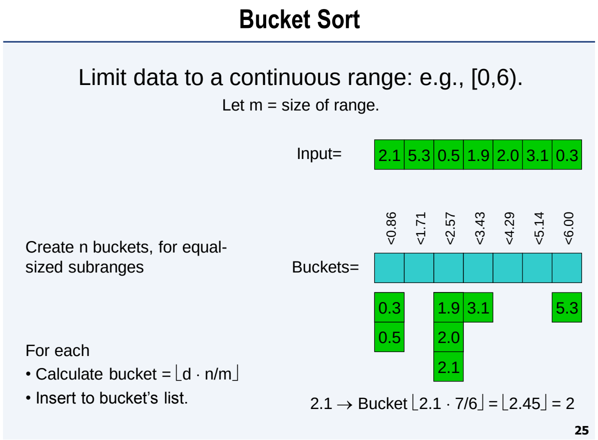
Limited usefulness, but the simplest example of a non-comparison-based sorting algorithm.

Counting Sort

```
csort(A,n):
    /* Count number of instances of each possible element. */
    Count[0…m-1] = 0                                  Θ(m)
    For index = 0 to n-1
        Count[A[index]] += 1                          Θ(n)

    /* Produce Count[i] copies of each i. */
    index = 0                                    ⎫
    For i = 0 to m-1                             ⎪
        For copies = 0 to Count[A[i]]            ⎬ Θ(m+n)
            A[index] = i                         ⎪
            index += 1                           ⎭
```

Θ(m+n) = Θ(n) time, when m taken to be a constant.



Bucket Sort

Limit data to a continuous range: e.g., [0,6).

Let m = size of range.

Input=

2.1	5.3	0.5	1.9	2.0	3.1	0.3

Create n buckets, for equal-sized subranges

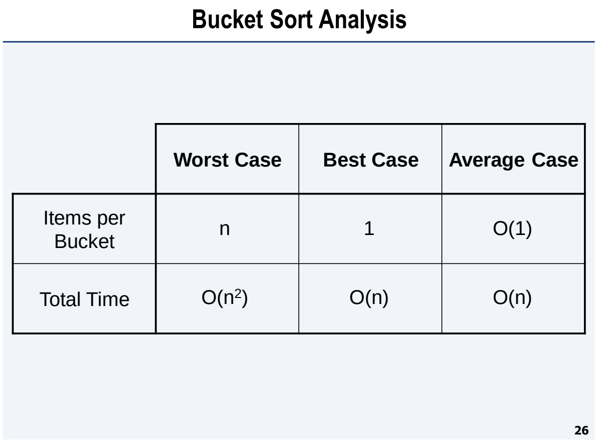
Buckets=

<0.86	<1.71	<2.57	<3.43	<4.29	<5.14	<6.00
0.3		1.9	3.1			5.3
0.5		2.0				
		2.1				

For each

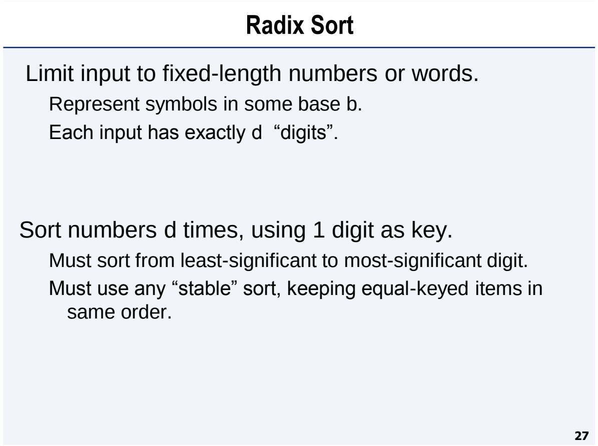
- Calculate bucket = $\lfloor d \cdot n/m \rfloor$
- Insert to bucket's list.

2.1 → Bucket $\lfloor 2.1 \cdot 7/6 \rfloor = \lfloor 2.45 \rfloor = 2$



Bucket Sort Analysis

	Worst Case	Best Case	Average Case
Items per Bucket	n	1	O(1)
Total Time	$O(n^2)$	O(n)	O(n)



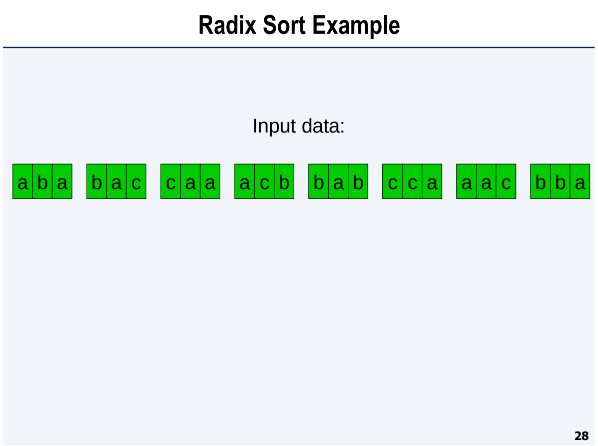
Radix Sort

Limit input to fixed-length numbers or words.

- Represent symbols in some base b.
- Each input has exactly d "digits".

Sort numbers d times, using 1 digit as key.

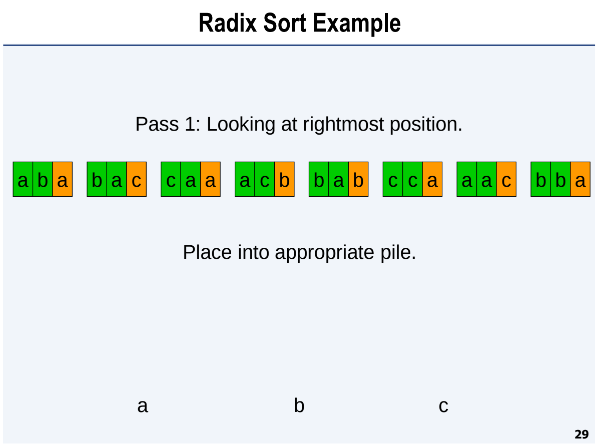
- Must sort from least-significant to most-significant digit.
- Must use any "stable" sort, keeping equal-keyed items in same order.



Radix Sort Example

Input data:

a b a   b a c   c a a   a c b   b a b   c c a   a a c   b b a

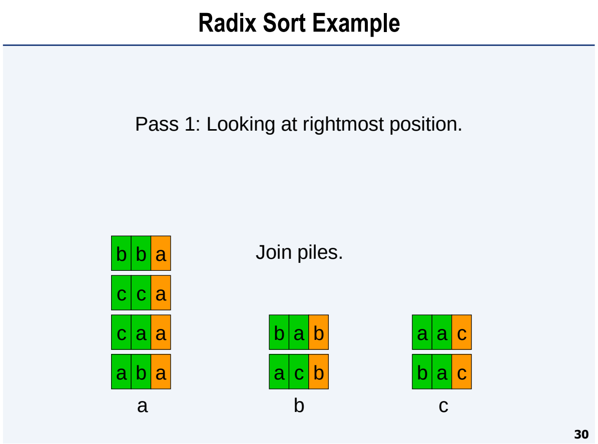


Radix Sort Example

Pass 1: Looking at rightmost position.

a b a   b a c   c a a   a c b   b a b   c c a   a a c   b b a

Place into appropriate pile.



Radix Sort Example

Pass 1: Looking at rightmost position.

Join piles.

a	b	c
b b a		
c c a		
c a a	b a b	a a c
a b a	a c b	b a c

Radix Sort Example

Pass 2: Looking at next position.

aba caa cca bba acb bab bac aac

Place into appropriate pile.

a b c

Radix Sort Example

Pass 2: Looking at next position.

Join piles.

a	b	c
aac		
bac		
bab	bba	acb
caa	aba	cca

Radix Sort Example

Pass 3: Looking at last position.

caa bab bac aac aba bba cca acb

Place into appropriate pile.

a b c

Radix Sort Example

Pass 3: Looking at last position.

Join piles.

a	b	c
acb	bba	
aba	bac	cca
aac	bab	caa

Radix Sort Example

Result is sorted.

aac aba acb bab bac bba caa cca

Radix Sort Algorithm

```
rsort(A,n):
  For j = 0 to d-1
      /* Stable sort A, using digit position j as the key. */
      For i = 0 to n-1
            Add A[i] to end of list ((A[i]>>j) mod b)

      A = Join lists 0…b-1
```

Θ(dn) time, where d is taken to be a constant.