

# Sorting

Dirty tricks to sort faster than  $O(n \log n)$

Terence Parr

MSDS program

**University of San Francisco**

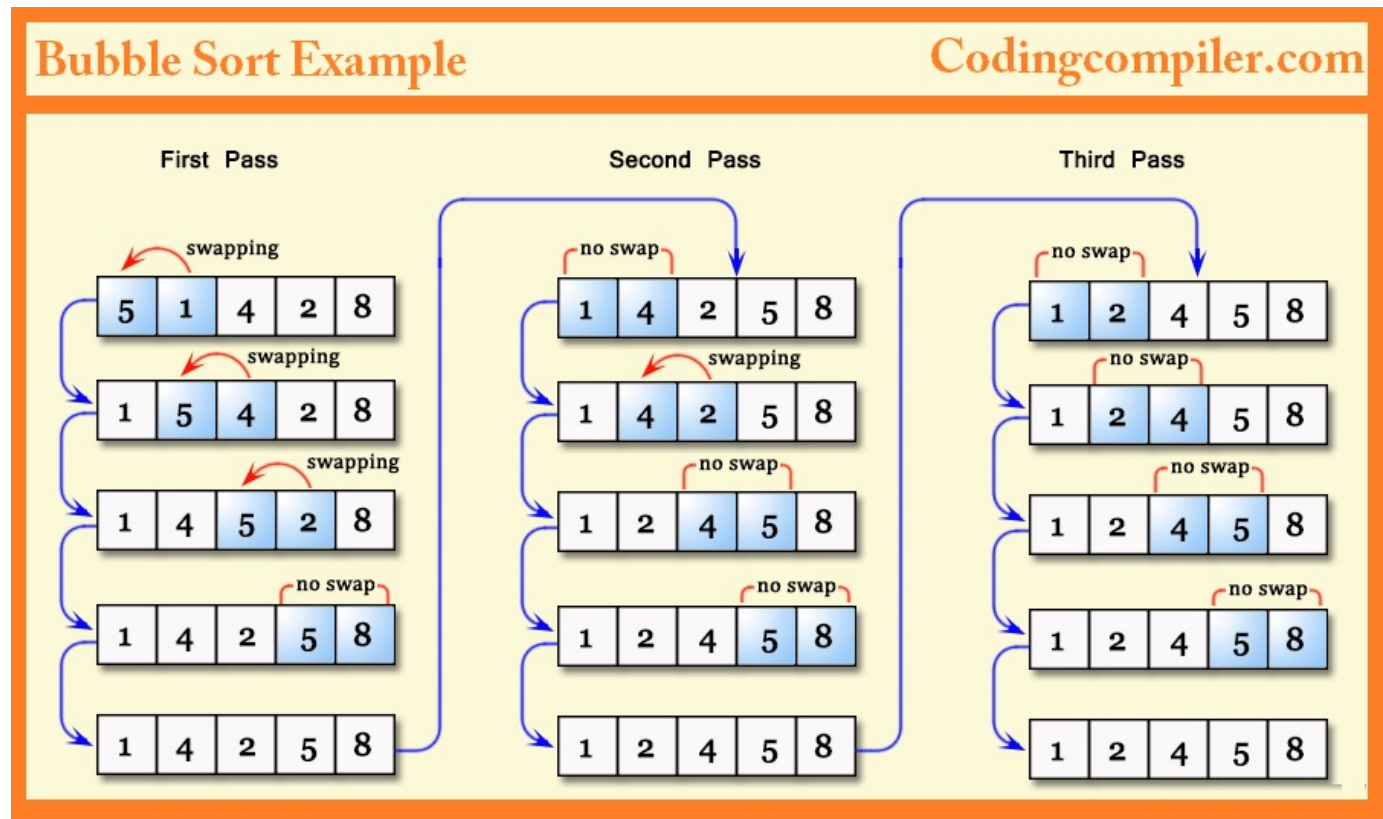
# Sorting

- We can sort any kind of element for which we have a similarity or distance measure between any two elements (subject to triangle inequality property\*)
- Traditional sorting algorithms: bubble sort, merge sort, quicksort
- Dirty tricks: pigeonhole sort, bucket sort can often sort in  $O(n)$
- Really dirty trick: nested bucket sort
- What's the fastest we could ever sort  $n$  numbers?
  - It depends on whether we're stuck using comparisons only
- Sorting notebook  
<https://github.com/parrt/msds689/blob/master/notes/sorting.ipynb>

\*[https://en.wikipedia.org/wiki/Triangle\\_inequality](https://en.wikipedia.org/wiki/Triangle_inequality)

# Bubble sort

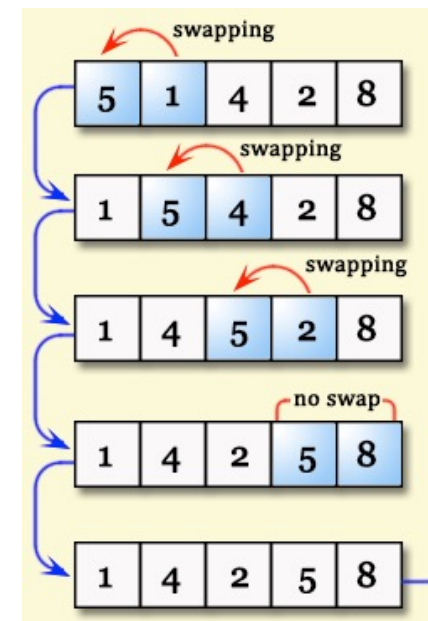
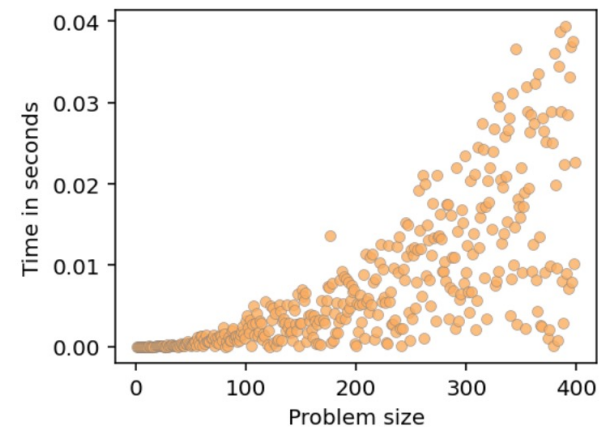
- $O(n^2)$
- *Stable*: order of equal elements doesn't change
- **Idea**: look for out-of-order elements and then keep swapping until nothing changes



# Bubble sort in Python

```
changed=True
second_to_last_idx = len(A)-2
while changed:
    changed=False
    for i in range(second_to_last_idx+1):
        if A[i] > A[i+1]:
            A[i], A[i+1] = A[i+1], A[i]
            changed=True
```

Why is this  $O(n^2)$ ?  
(hint: What is worst case order in array?)



# Merge sort (review)

- Faster than bubblesort:  $O(n \log n)$
- Simpler too, if you are comfortable with recursion
- It's stable
- Not in-place, uses lots of extra storage (sort halves)
- **Idea:** split currently active region in half, sorting both the left and right subregions, then merge two sorted subregions
- Eventually, the regions are so small we can sort in constant time; i.e., sorting 2 nums is easy
- Merging two sorted lists can be done in linear time

# Quicksort, another divide and conquer sort

- $O(n^2)$  worst-case behavior but  $O(n \log n)$  typical behavior
- **Idea:** pick pivot, partition so elements left of pivot are less than pivot and elements right are greater (not sorting here); recursively partition the left and right until small enough to sort trivially
- Picks a pivot element, rather than just split in half like mergesort
- Faster than bubble because it moves elements more than just one spot in the array
- Quicksort is in-place whereas merge sort makes lots of temporary arrays, which can get expensive
- Quicksort is mostly faster than merge sort due to the constant in front of the complexity (memory allocation, hardware efficiencies, ...)

In-place quicksort in notebook <https://github.com/parrr/msds689/blob/master/notes/sorting.ipynb>

# Quicksort algorithm

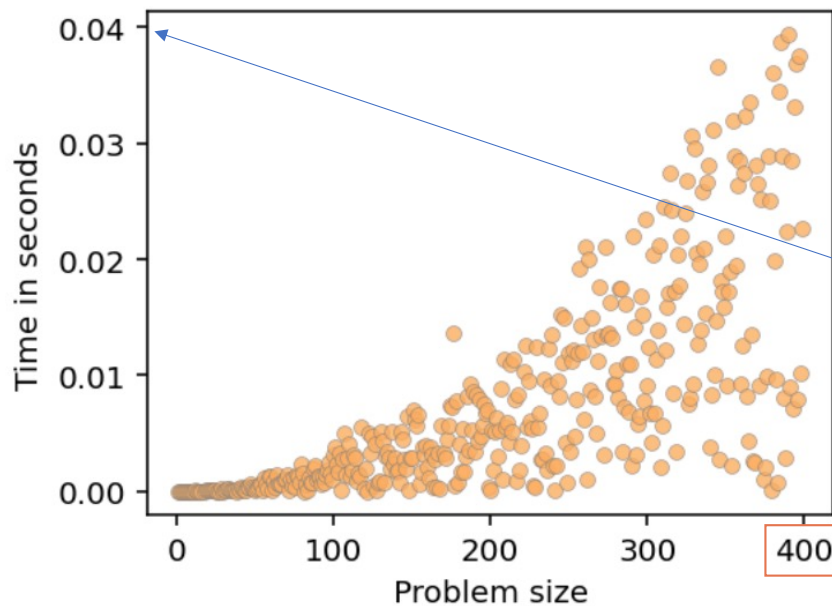
```
def qsort(A, lo=0, hi=len(A)-1):  
    if lo >= hi:  
        return  
    pivot_idx = partition(A, lo, hi)  
    qsort(A, lo, pivot_idx-1)  
    qsort(A, pivot_idx+1, hi)
```

```
# many ways to do this; here's a slow O(n) one  
# breaks idea of in-place for qsort  
def partition(A, lo, hi):  
    pivot = A[hi] # pick last element as pivot  
    left = [a for a in A if a < pivot]  
    right = [a for a in A if a > pivot]  
    A[lo:hi+1] = left + [pivot] + right # copy back  
    return len(left) # return index of pivot
```

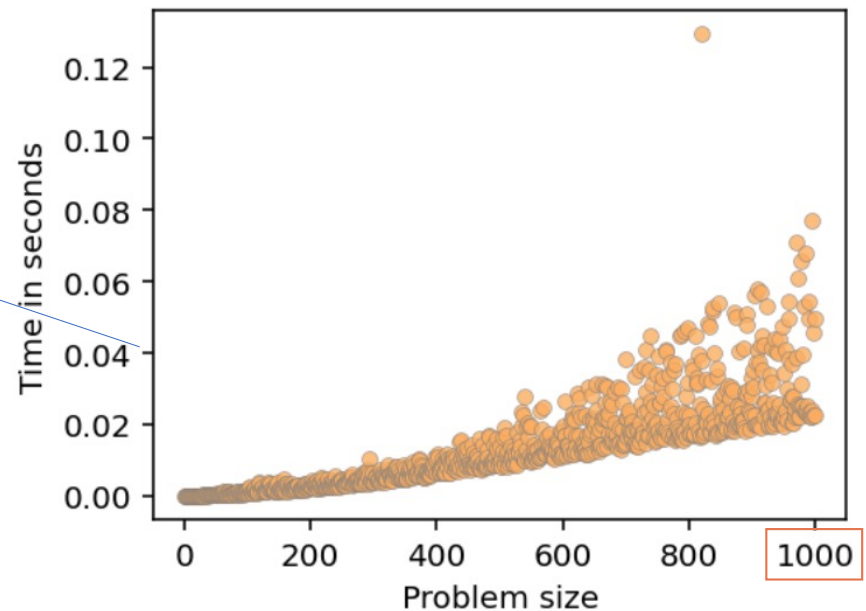
Video on partitioning: [https://www.youtube.com/watch?v=MZaf\\_9IZCrc](https://www.youtube.com/watch?v=MZaf_9IZCrc)

# Compare bubble, quicksort

Bubble sort



Quicksort



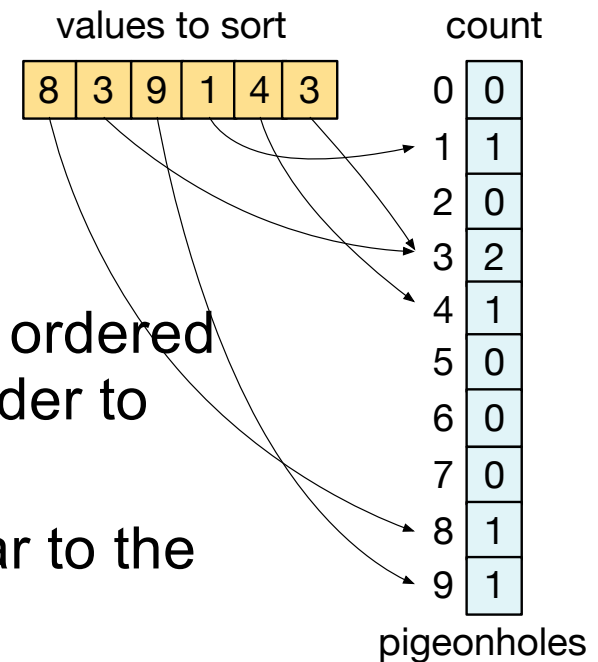


# So much for traditional sorts

- Theory says we can't beat  $O(n \log n)$ ...
- ...for generic elements and doing comparisons
- But, what if we know the elements are ints or strings or floats?
- What if we know something about the values?
- E.g., what if we know the elements are ints in range 0..99?
- How can we sort those numbers in less than  $O(n \log n)$ ?

# Pigeonhole sort

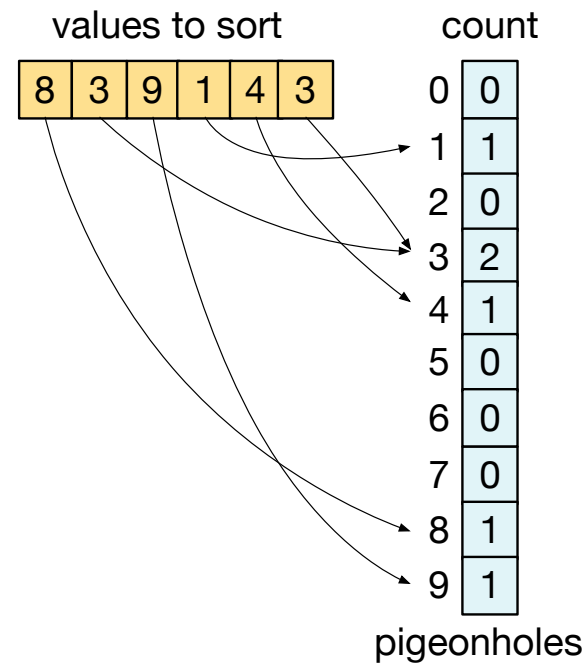
- **Idea:** Map each key to unique pigeonhole in an ordered range of holes; then just walk pigeonholes in order to get sorted elements
- Works best when the range of keys,  $m$ , is similar to the number of elements,  $n$ ; why is that?
- $T(n,m) = n + m$
- This should smack of perfect hashing to you!



# Pigeonhole sort algorithm

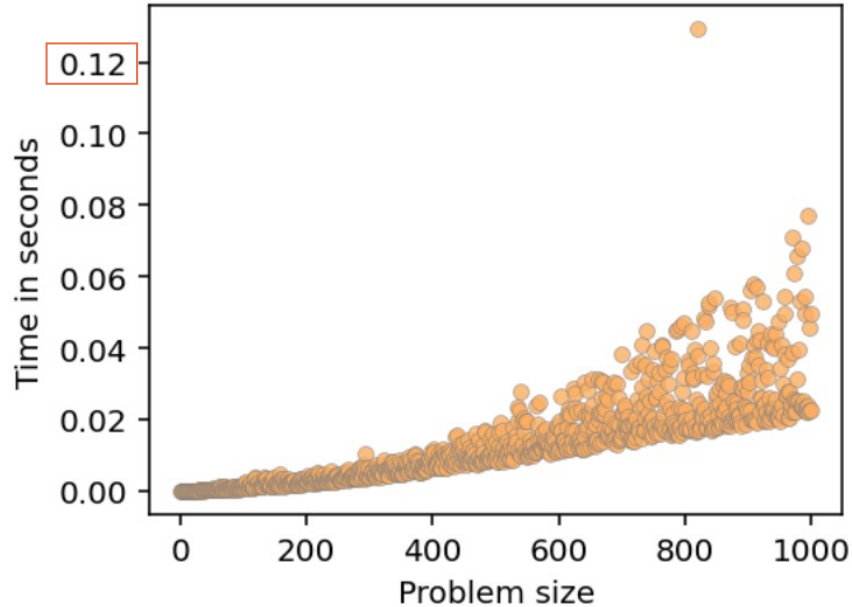
```
# fill holes
size = max(A) + 1
holes = [0] * size
for a in A:
    holes[a] += 1

# pull out in order
A_ = []
for i in range(0, size):
    A_.extend([i] * holes[i])
```

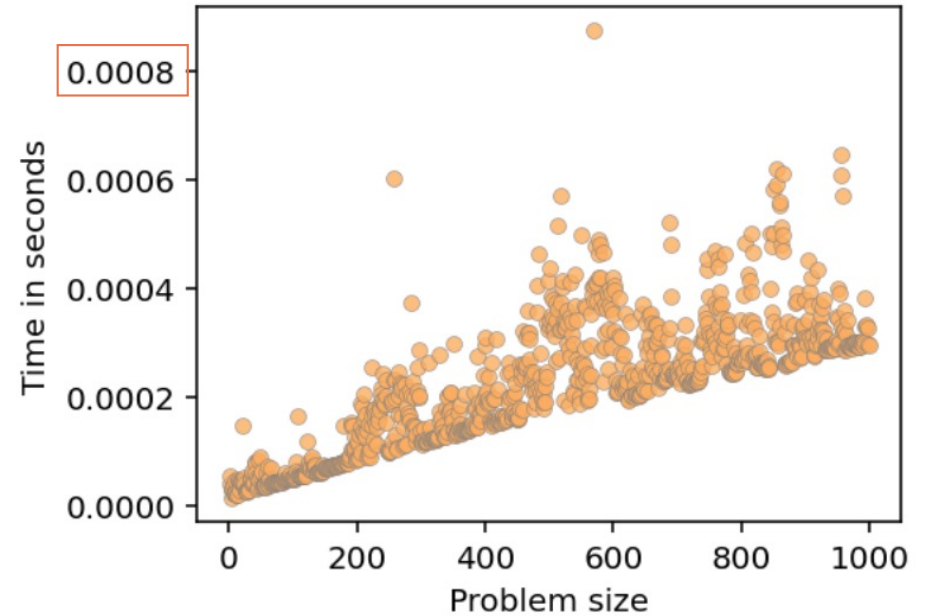


# Compare quicksort, pigeonhole

Quicksort



Pigeonhole

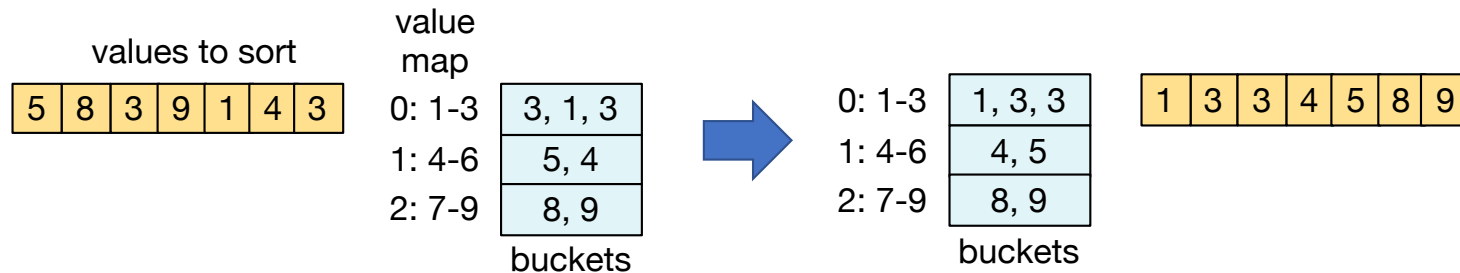


# Issue with pigeonhole sort

- Super fast and simple but...
- What do we do when  $m \gg n$ ? E.g., sort 2 numbers, 5 and 5 million. Takes  $T(n,m) = n + m = 5 + 5,000,000$
- How can we handle this case & generalize to work for floats too?
- Hint: compress  $m$  to some fixed number of buckets instead of range of numbers
- Now we have hash table but with special hash function

# Bucket sort (also called bin sort)

- Idea: distribute  $n$  elements across  $m$  buckets, sort elements within buckets, then concatenate elements from buckets in order



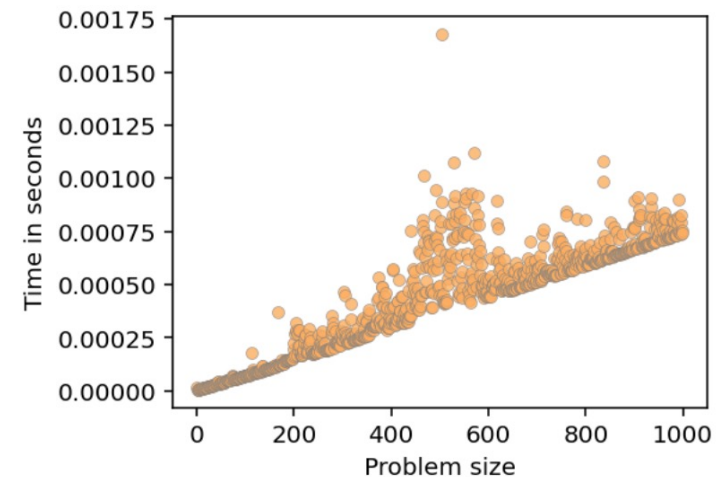
- Hash must preserve order of values!
- Similar to pigeonhole sort but pigeonhole has 1 key per bucket
- Best when there is even distribution of values like hash table
- Works for floats not just ints; see notebook for implementation

# Key bits of bucket sort algorithm

```
mx = max(A)
...
for a in A:
    a_normalized = a / mx # get into 0..1
    # spread across buckets
    i = int(a_normalized * (nbuckets-1))
    buckets[i].append(a)
...
for i in range(nbuckets):
    A_.extend( sorted(buckets[i]) )
```

# Bucket sort worst-case analysis

- What is  $T(n,m)$  worst-case?
- What if all values are the same? All go into 1 bucket!
- Sorting one bucket at best costs us  $O(k \log k)$  for bucket size  $k$
- Bubblesort might be faster for small buckets but that's  $O(k^2)$  worst-case in theory
- Can use insertion sort is  $O(k^2)$  for adding to bucket or leave unsorted and sort later



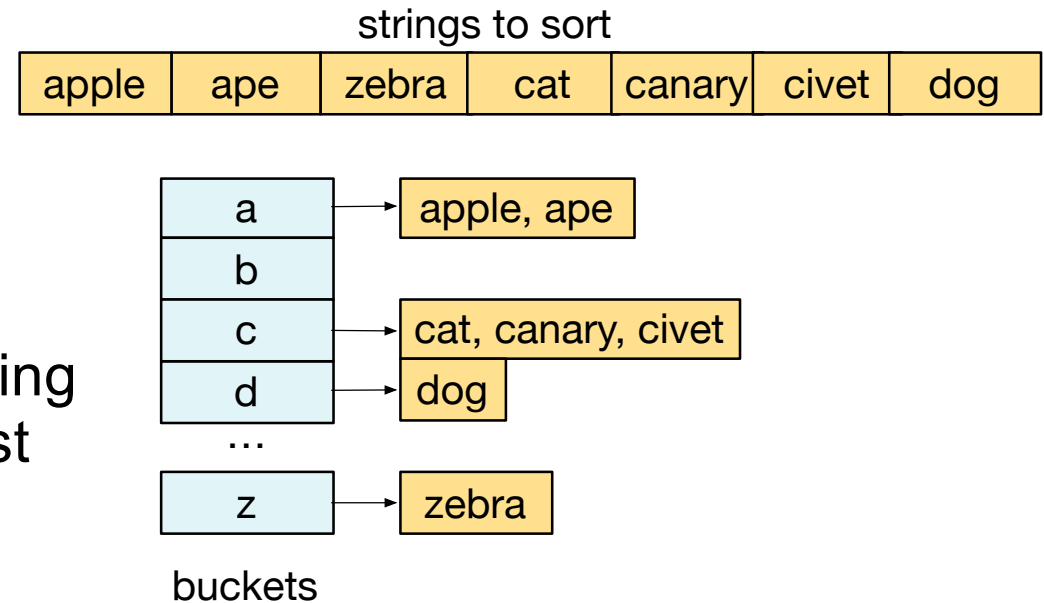


# Bucket sort best-case analysis

- What does the best case or average case look like?
- Assume even distribution of elements across  $m$  buckets
- Choose  $m$  always so  $k=n/m$  is some small fixed constant size  $k$
- Sort  $k$  elements  $m$  times (bubblesort  $O(k^2)$ ), merge  $m$  sorted lists
- $T(n,m,k=n/m) = m * k^2 + n$
- Replace  $m=n/k$ :  
 $T(n,k) = n/k * k^2 + n = n*k + n = n(k+1)$  (choose small  $k$ )
- That gives us  $O(n)$

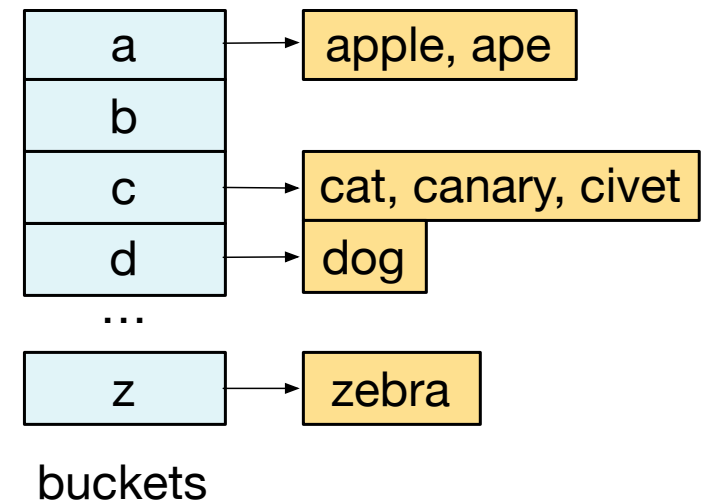
# Bucket sort on strings

- Use first letter as bucket key
- Add strings to buckets
- Sort within bucket
- Walk a..z buckets, concatenating those sorted lists into single list
- See sorting notebook for implementation



# Key bits of string bucket sort

```
for s in A:
    i = ord(s[0]) - ord('a')
    holes[i].append(s)
...
for i in range(ord('z') - ord('a') + 1):
    A_.extend( sorted(holes[i]) )
```

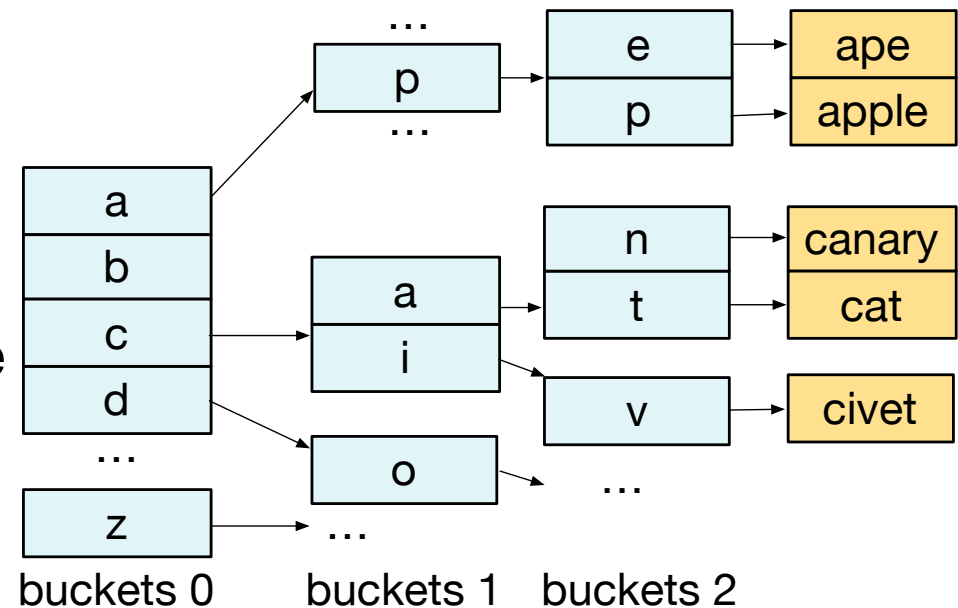


**Exercise:** What if all words start with same letter?

# Nested or recursive string bucket sort

(Called *TRIEs* and we'll see again)

- Nested indexes based upon  $s[i]$
- With nesting  $k$  deep, words are sorted uniquely to first  $k$  letters, giving nested bucket sort
- Nested dynamically to full len of string gives nested pigeonhole sort
- Walk all edges in alpha order to collect words in leaves
- What is rough complexity?  
 $O(n*k)$  for  $k$  = max length of any string



# Summary

- If asked, sorting is  $O(n \log n)$  (via comparisons)
- Divide and conquer, merge and quicksort, are primary algorithms
  - Mergesort merges two sorted halves recursively; takes extra memory
  - Quicksort partitions instead of sorting halves; works in-place (usually better)
- But, we can do better with pigeonhole sort, mapping each element to unique bucket based on the key;  $O(n)$
- If mapping to unique bucket is hard, as with floating-point numbers, use bin/bucket sort like a hash table;  $O(n)$  if reasonably evenly distributed and enough buckets
- Use `ord(char)` for strings to bucket sort
- Use all letters in strings to get nested bucket sort (called a TRIE)