Sorting

Dirty tricks to sort faster than *O(n log n)*

Terence Parr
MSDS program
University of San Francisco

UNIVERSITY OF SAN FRANCISCO

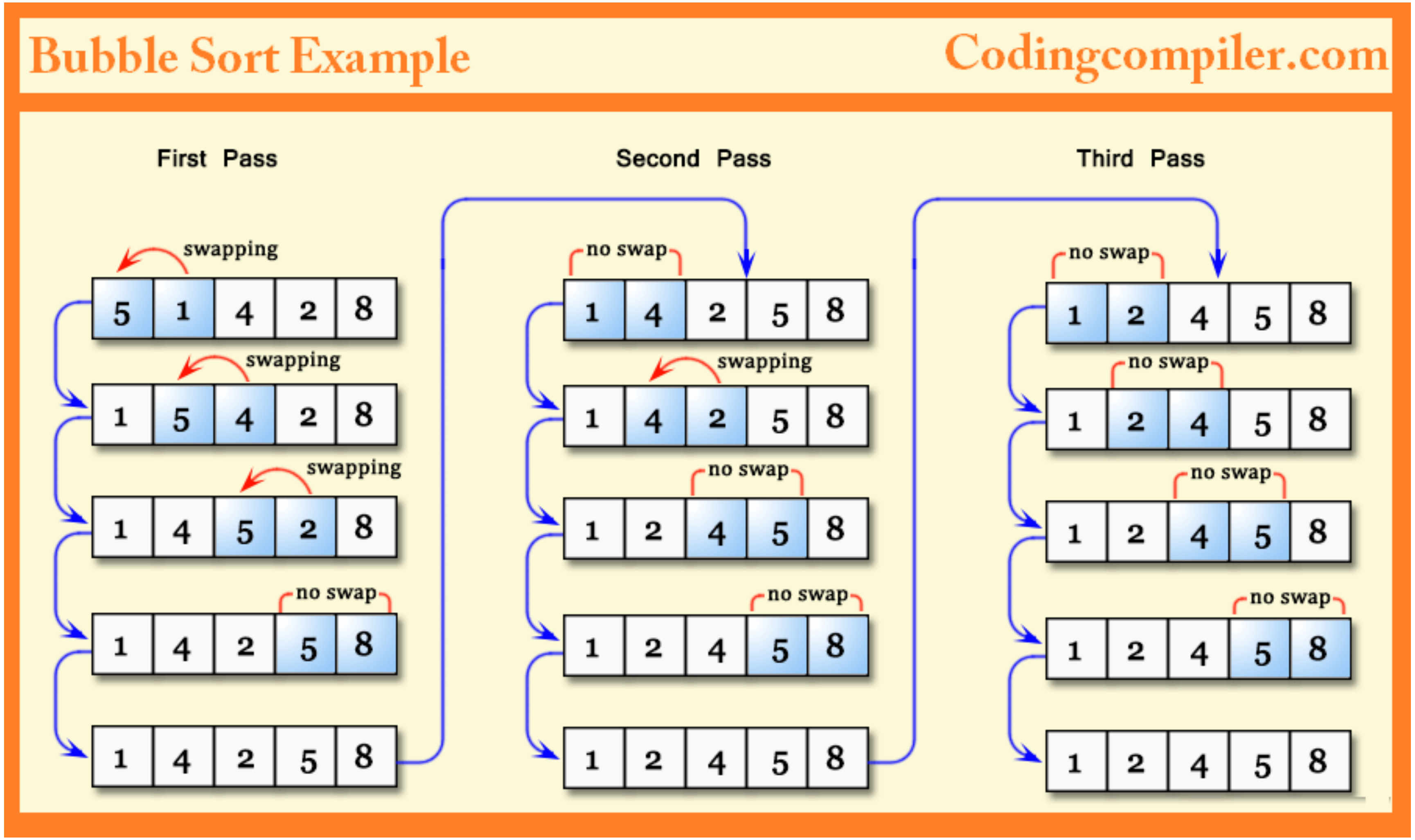
Sorting

- 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?

*https://en.wikipedia.org/wiki/Triangle_inequality

Bubble sort

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

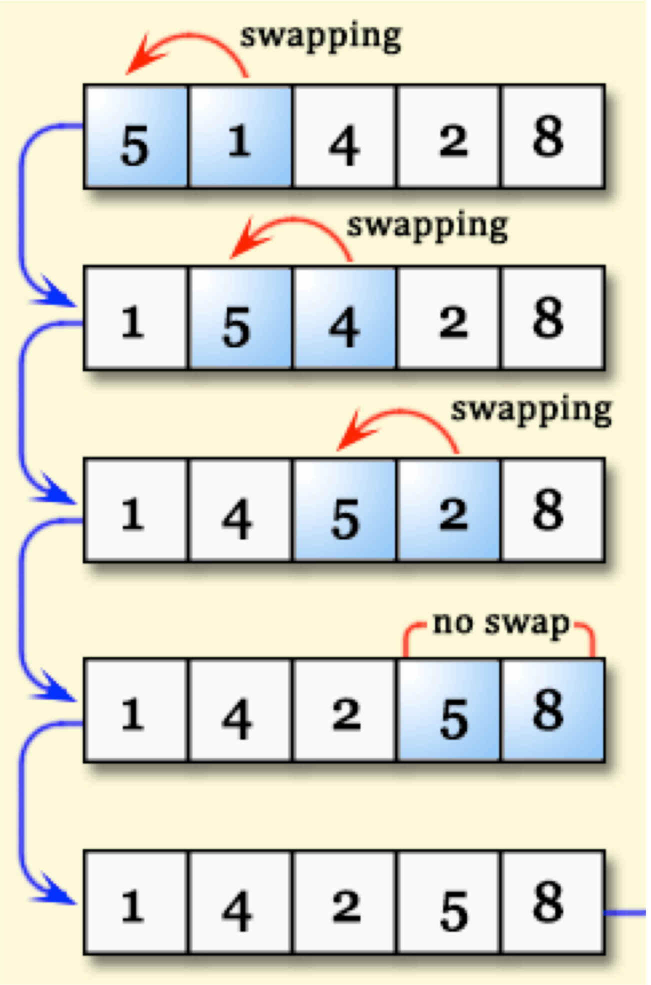


https://codingcompiler.com/bubble-sort-program-in-c-using-function/

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)?



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
- **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 due to the constant in front of the complexity (memory allocation, hardware efficiencies, …)

Quicksort algorithm

```
def qsort(A, lo, hi):
    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[:] = left+[pivot]+right # copy back to A
    return len(left) # return index of pivot
```

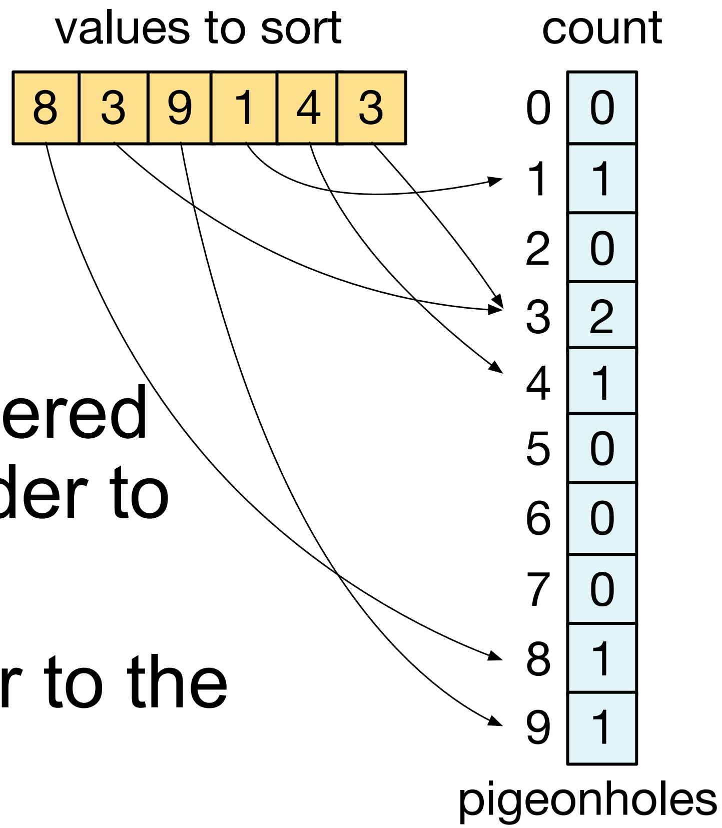
Partitioning important for decision and isolation trees

Video on partitioning:
https://www.youtube.com/watch?v=MZaf_9IZCrc

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



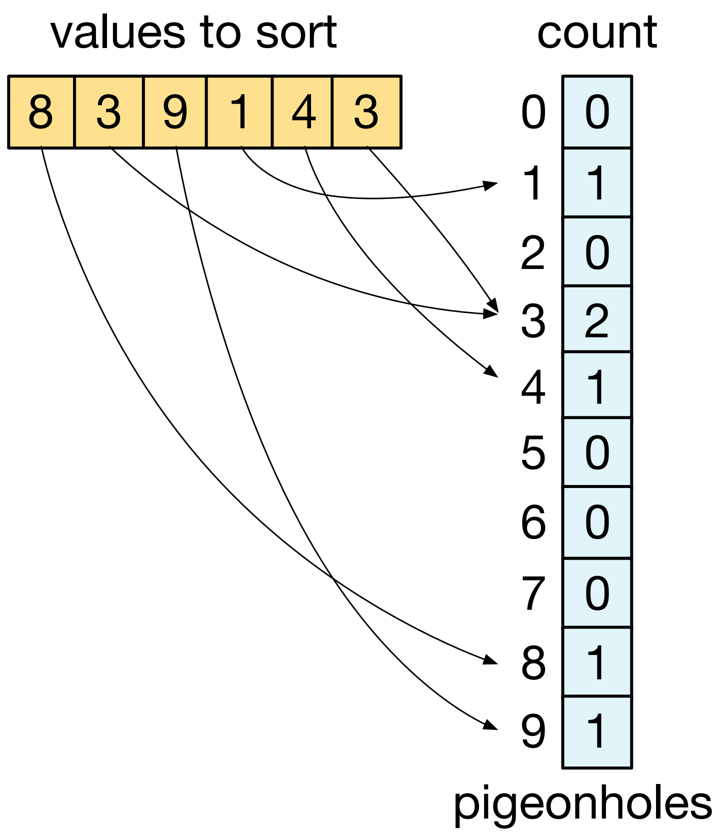
- **Idea**: Map each key to unique pigeonhole in ordered range of holes; then just walk pigeonholes in order to get sorted elements
- Works best when the range of keys, m, is similar to the number of elements, n; why is that?
- $T(n,m) = n + m$
- This should smack of perfect hashing to you!

https://en.wikipedia.org/wiki/Pigeonhole_sort

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):
    for j in range(holes[i]):
        A_.append(i)
```



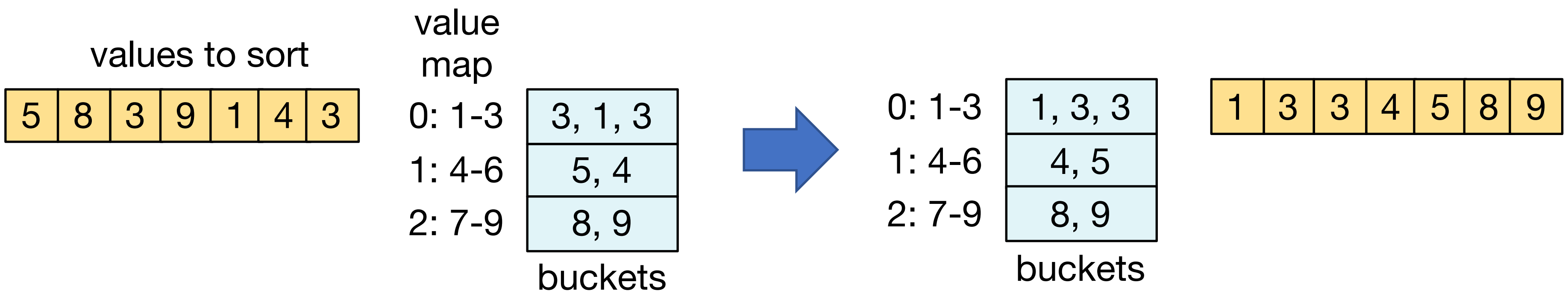
See sorting notebook

Issue with pigeonhole sort

- Super fast and simple but…
- What do we do when m >> 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, number of buckets, to some fixed number 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 ordered buckets, sort elements within each bucket, then concatenate elements from sorted buckets in order



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

https://en.wikipedia.org/wiki/Bucket_sort See sorting notebook

Bucket sort worst-case analysis

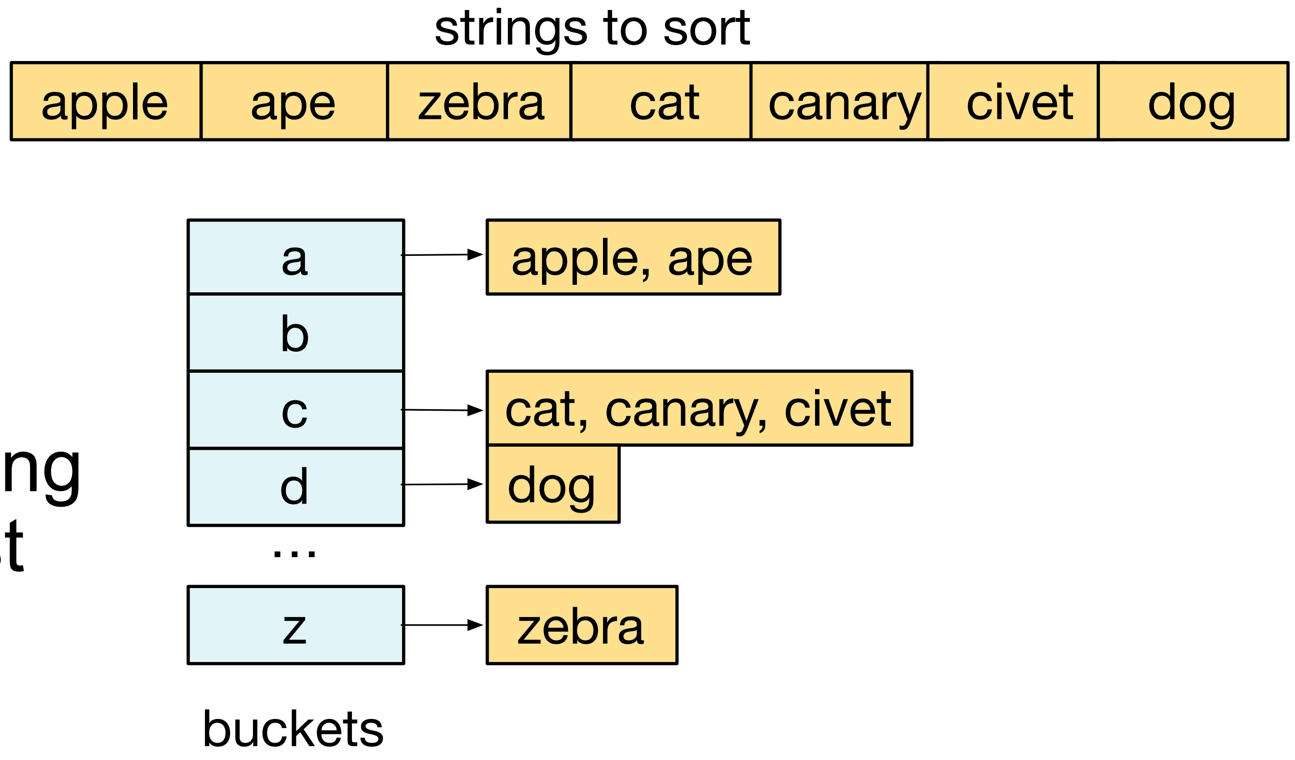
- What is T(n,m) worst-case?
- Assume all values are the same, putting everything into one bucket
- Sorting one bucket at best costs us n log n
- There are m buckets we must walk
- T(n,m) = n log n + m, yielding O(n log n)
- Bubblesort might be faster for small buckets of size k=n/m but that's O(n^2) worst-case in theory

Bucket sort best-case analysis

- What's the best case or average case look like?
- Assume even distribution of elements across m buckets
- Choose m always so n/m is some small fixed constant size k
- Sort k elements m times (bubblesort), merge m sorted lists
- T(n,m,k) = m * k^2 + n
- T(n,k) = n/k * k^2 + n = n/k + n   (choose k close to n)
- 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
- **Exercise**: What if all words start with same letter?



Nested or recursive string bucket sort

- 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