Comparison Sorting Lower Bounds

What is the theoretically fastest possible sort speed?

Comparison sorting algorithms sort arrays by comparing elements (with no extra information)

Examples: Insertion sort | Heapsort | Quicksort

H. Steinhaus considered a sorting puzzle - ranking all players in tennis tournaments (1939) 

L. Ford and S. Johnson introduced merge-insertion sort with decision tree analysis (1959)  

Runtime complexity results for comparison sorts

Lower bound for worst input: $\rightarrow \Omega(n \log n)$

Lower bound for average input: $\rightarrow \Omega(n \log n)$

$\Longrightarrow O(n \log n)$ sorts such as heapsort are asymptotically optimal

Note: **non-comparison** sorts (e.g. Radix sort) can do better

References/Notes/Image credits:
H. Steinhaus, "Mathematical Snapshots" (1939)
(Image of Steinhaus) https://en.wikipedia.org/wiki/Hugo_Steinhaus#/media/File:Hugo_Steinhaus.jpg
(Bumps photo) https://www.flickr.com/photos/stanbury/42283400832/in/photostream/

L. Ford and S. Johnson, "A tournament problem", The American Mathematical Monthly (1959)
(Image of Ford) https://en.wikipedia.org/wiki/Lester_R._Ford#/media/File:Lester_R._Ford.gif
(Image of Johnson) https://alchetron.com/Selmer-M-Johnson

Comparison Sorting Algorithms and Decision Trees

A comparison sort ingests an array $[a_0, \ldots, a_{n-1}]$

It only gains information by comparing pairs:

"is $a_i < a_j$"?

It outputs a permutation that orders the items

For our analysis, assume all elements are distinct

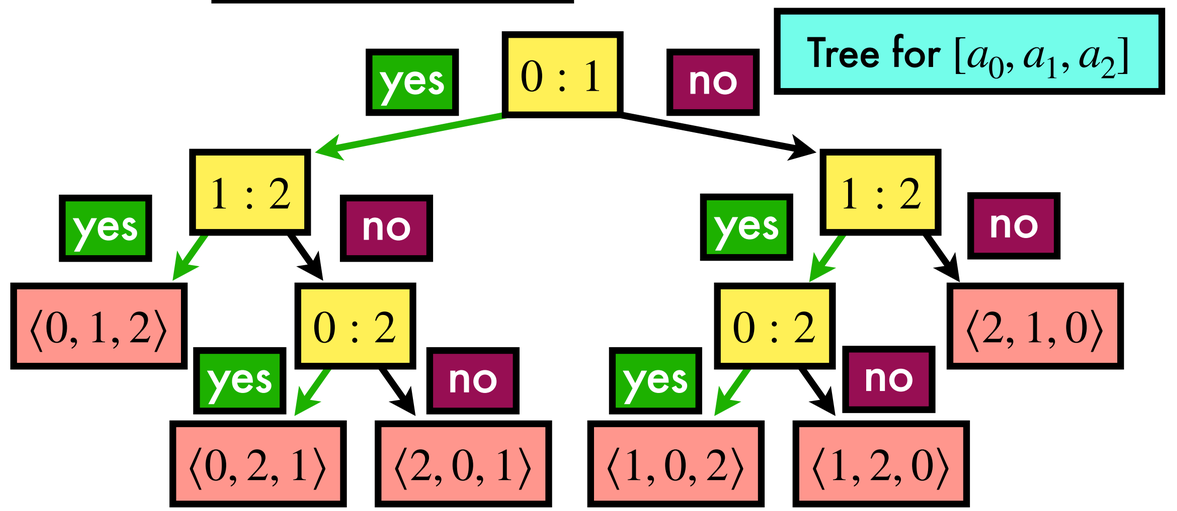
(repeated elements won't affect lower bounds)

Decision trees (or "Comparison trees")

Full binary trees where each internal node

$i : j$ represents a comparison $a_i < a_j$

Each leaf $\langle \pi(0), \ldots, \pi(n-1) \rangle$ is an array permutation



A comparison sort follows a path from root to leaf

References:

A. Blum and M. Blum, "Comparison-based Lower Bounds for Sorting", https://www.cs.cmu.edu/~avrim/451f11/lectures/lect0913.pdf

D. E. Knuth, "The art of computer programming, vol. 3: sorting and searching", Chap 5.3 (1998)

T. Cormen et al., "Introduction to algorithms", Chap 8, MIT press (2022)

J. Erickson, "Lower Bounds", https://jeffe.cs.illinois.edu/teaching/algorithms/notes/12-lowerbounds.pdf (2018)

Note: in a "full" binary tree, every node has either 0 or 2 children.

Lower Bound On The Worst Case

There are $n!$ possible permutations (without repetitions) i.e. ways to sort the input

A correct comparison sorting algorithm must be able to produce all of these

$\Longrightarrow$ a comparison sort corresponds to a full binary tree with $\geq n!$ leaves

Worst case number of comparisons - length of longest simple path from root to leaf

What is the maximum height of a decision tree?

A binary tree with height h has at most 2^h leaves, so $n! \leq 2^h$

Stirling's approximation: $n! = \left(\frac{n}{e}\right)^n \sqrt{2\pi n}\left(1 + \Theta\left(\frac{1}{n}\right)\right) \geq \left(\frac{n}{e}\right)^n$

$$h \geq \log(n!) \geq \log\left(\frac{n}{e}\right)^n = n\log n - n\log e = \Omega(n\log n)$$

No comparison sort can be faster than this on its worst case input

References:

A. Blum and M. Blum, "Comparison-based Lower Bounds for Sorting", https://www.cs.cmu.edu/~avrim/451f11/lectures/lect0913.pdf

D. E. Knuth, "The art of computer programming, vol. 3: sorting and searching", Chap 5.3 (1998)

T. Cormen et al., "Introduction to algorithms", Chap 8, MIT press (2022)

J. Erickson, "Lower Bounds", https://jeffe.cs.illinois.edu/teaching/algorithms/notes/12-lowerbounds.pdf (2018)

Lower Bound On The Average Case

What is the smallest possible average height of a decision tree?

The average height is smallest when the tree is completely balanced

Completely balanced means no leaf heights differ by more than 1

If the tree is completely balanced, each leaf has depth $\lceil \log(n!) \rceil$ or $\lfloor \log(n!) \rfloor$

$\Longrightarrow$ smallest possible average height is $\Omega(n \log n)$

No comparison sort can be faster than this on average

References:
A. Blum and M. Blum, "Comparison-based Lower Bounds for Sorting", https://www.cs.cmu.edu/~avrim/451f11/lectures/lect0913.pdf
D. E. Knuth, "The art of computer programming, vol. 3: sorting and searching", Chap 5.3 (1998)

Counting Sort

Counting Sort

Fast sorting algorithm that assumes keys are integers in a fixed range

Introduced by H. Seward (1954)  

Not a comparison sort (uses key values)

Stable - preserves order of repeated elements

Counting sort complexity

For n items with integer keys (in 0 to $k-1$):

Running time for any input: $\rightarrow \Theta(n+k)$

Note: counting sort avoids the $\Omega(n \log n)$ lower bound because it is not a comparison sort

Storage: $\Theta(n+k)$ in addition to $\Theta(n)$ for input

Uses of counting sort:

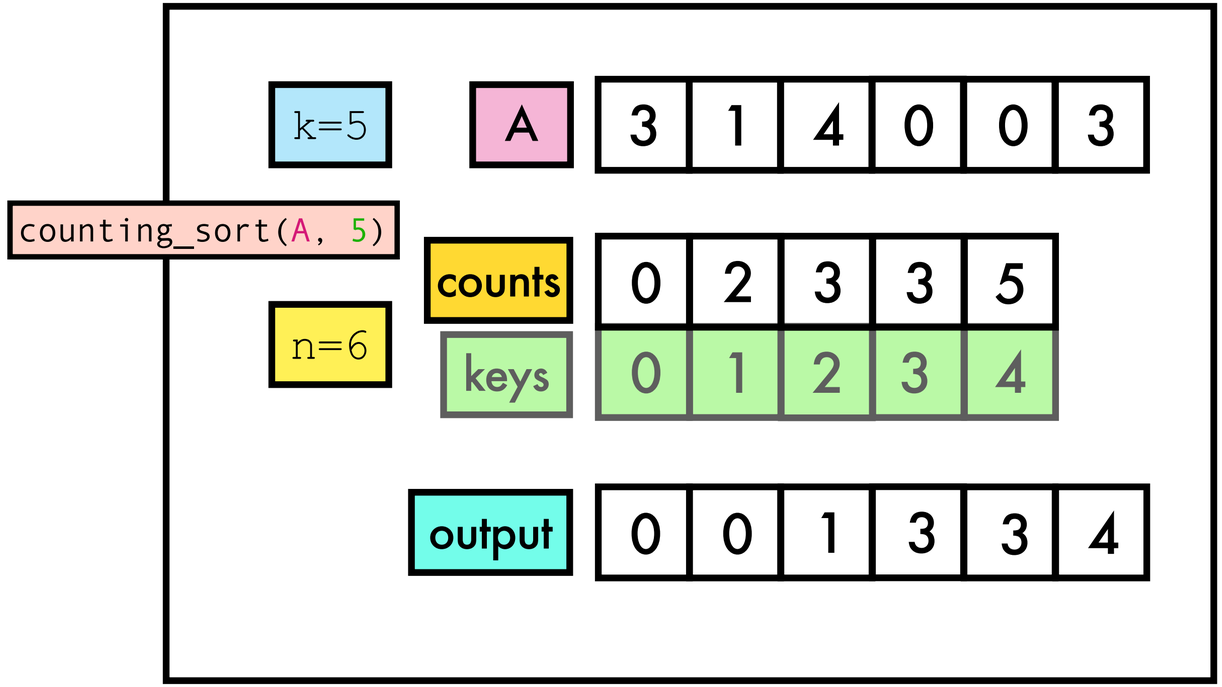
As a building block for radix sort

Very fast when $k = O(n) \rightarrow$ run time is $\Theta(n)$

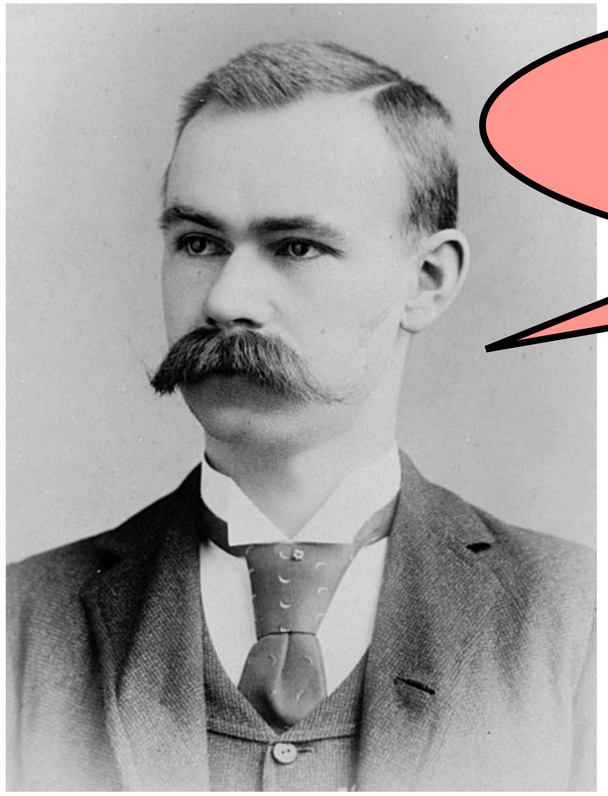
References/Notes/Image credits:
H. Seward, "Information sorting in the application of electronic digital computers to business operations", Dissertation MIT (1954)
(Image of H. Seward) https://www.legacy.com/us/obituaries/bostonglobe/name/harold-seward-obituary?id=20744826
(Image of thesis) http://bitsavers.trailing-edge.com/pdf/mit/whirlwind/R-series/R-232_Information_Sorting_in_the_Application_of_Electronic_Digital_Computers_to_Business_Operations_May54.pdf
D. E. Knuth, "The art of computer programming, vol. 3: sorting and searching", Chap 5.2 (1998)
T. Cormen et al., "Introduction to algorithms", Chap 8, MIT press (2022)

Counting Sort Implementation

```
def counting_sort(A, k): # keys from 0 to k-1
    n = len(A)
    counts = [0 for _ in range(k)]
    output = [None for _ in range(n)]
    for key in A:
        counts[key] += 1
    for i in range(1, k):
        counts[i] += counts[i-1]
    for key in reversed(A):
        output[counts[key]-1] = key
        counts[key] = counts[key] - 1
    return output
```

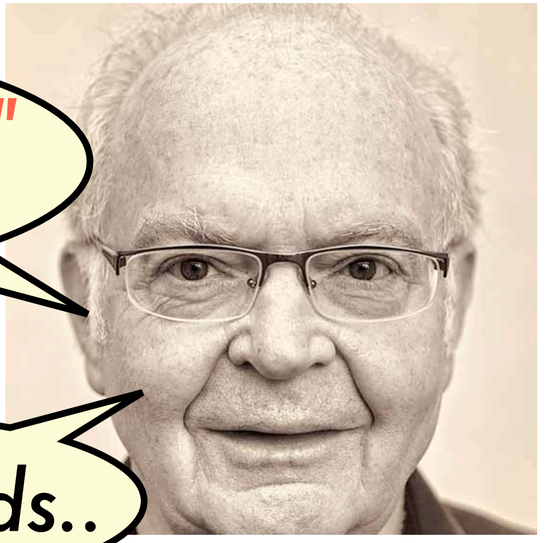


References:
https://en.wikipedia.org/wiki/Counting_sort
T. Cormen et al., "Introduction to algorithms", Chap 8, MIT press (2022)



What is the best way to sort data with machines?

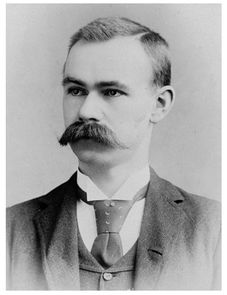
There is no "best" way to sort.



It depends..

Radix Sort

valuable for certain kinds of keys



History

Runtime complexity

$$\Theta(d(n+k))$$

Properties

```
def radix_sort(A, d, k):
    ...
```

Python Implementation

Radix Sort

Radix Sort

Non-comparative sorting algorithm

It orders keys by values, not by comparisons

H. Hollerith card sorting (1901)  

L. J. Comrie published LSD variant (1929) 

H. Seward improved storage (1954) 

Radix sort applies to data that can be sorted lexicographically (integers, words, cards etc.)

Radix sort runtime complexity

Two radix sort variants:

Least Significant Digit (LSD) | Most Significant Digit (MSD)

For n items with d-digit integer keys (in 0 to $k-1$):

LSD Radix Sort

Running time for any input: $\rightarrow \Theta(d(n+k))$ when using counting sort for inner loop

Storage: $\Theta(n+k)$ in addition to $\Theta(dn)$ for input

MSD Radix Sort *(worst case runtime matches LSD)*

Average runtime: $\Theta(n \log_k n)$ on random inputs

Storage: $\Theta(n+pk)$ in addition to $\Theta(dn)$ for input, where p is length of longest key prefix match

References/Notes/Image credits:
(Early history) https://www.ibm.com/ibm/history/ibm100/us/en/icons/tabulator/
D. E. Knuth, "The art of computer programming, vol. 3: sorting and searching", Chap 5 (1998)
(Hollerith image) https://en.wikipedia.org/wiki/Herman_Hollerith#/media/File:Hollerith.jpg
(Hollerith machine) https://americanhistory.si.edu/collections/search/object/nmah_694412
(CTR logo) https://en.wikipedia.org/wiki/Computing-Tabulating-Recording_Company#/media/File:CTR_Company_Logo.png
(IBM logo) https://en.wikipedia.org/wiki/File:IBM_logo.svg
L. J. Comrie, "The Hollerith and Powers Tabulating Machines", Transactions of the Office Machinery Users Assoc. (1929)
H. Seward, "Information sorting in the application of electronic digital computers to business operations", Dissertation MIT (1954)
T. Cormen et al., "Introduction to algorithms", Chap 8, MIT press (2022)
R. Sedgewick, "MSD Radix Sort", https://es.coursera.org/lecture/algorithms-part2/msd-radix-sort-gFxwG (2007)

Least And Most Significant Digit Radix Sorts

Should we radix sort digits from right to left or from left to right?

Least Significant Digit - LSD (right to left)

Most Significant Digit - MSD (left to right)

LSD radix sort

Assumes all keys have same length

Approach used in card-sorting machines

Requires a stable sort in inner loop

Iteratively sorts d-digit data in d passes

MSD radix sort

Supports keys with different lengths

Sorts by recursion (exits early if < 2 keys share digit)

Runtime depends heavily on the input data distribution

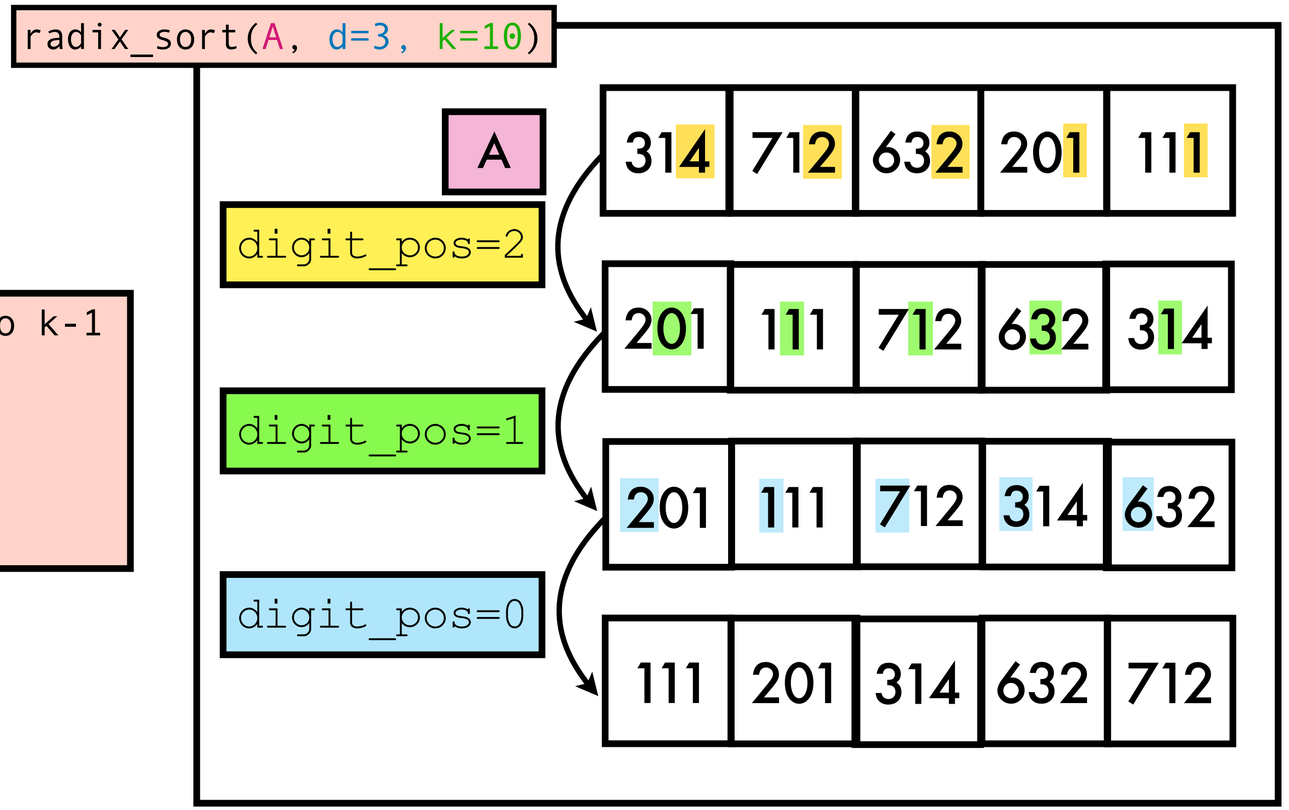
Can create lots of (small and inefficient) subarray sorts

This can be addressed with cutoff to insertion sort

Reference:
R. Sedgewick et al., "Radix Sorts", https://www.cs.princeton.edu/courses/archive/spr07/cos226/lectures/11RadixSort.pdf (2007)

LSD Radix Sort Implementation

```
def radix_sort(A, d, k): # keys w. d digits from 0 to k-1
    # loop from least to most significant digit
    for digit_pos in range(d-1,-1,-1):
        # typically use counting sort as stable sort
        stable_sort(A, k, digit_pos)
```



Reference:
T. Cormen et al., "Introduction to algorithms", Chap 8, MIT press (2022)

Digit Size Selection

The influence of digit size

For n numbers with b-bits and $r \leq b$ (digit size)

LSD radix sort runtime is $\Theta\left(\frac{b(n+2^r)}{r}\right)$

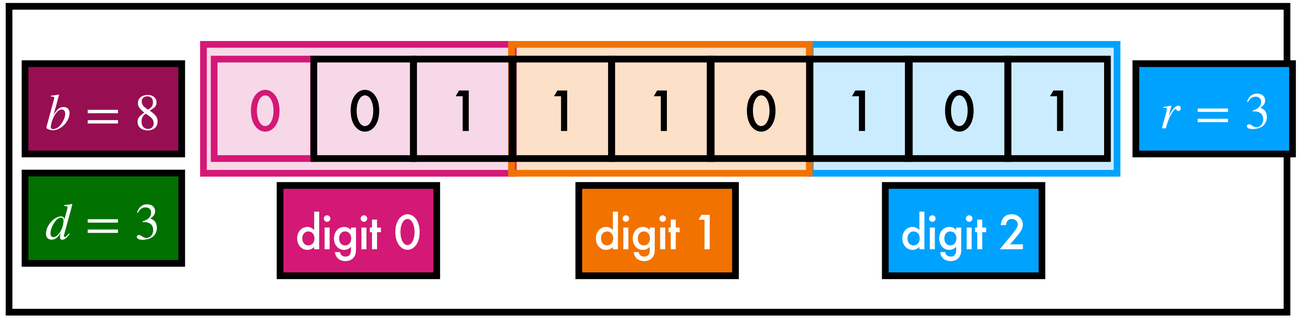
(assumes counting sort in inner loop)

Proof: interpret keys as $d = \lceil b/r \rceil$ digits

where each digit has r bits

Use counting sort with $k = 2^r$

Runtime is $\Theta(d(n+k)) = \Theta\left(\frac{b(n+2^r)}{r}\right)$



How should we choose digit size (r bits)?

Goal: minimise $\frac{b(n+2^r)}{r}$

If $b < \lfloor \log n \rfloor$: pick $r = b \implies \frac{b(n+2^b)}{b} = \Theta(n)$ (optimal)

If $b \geq \lfloor \log n \rfloor$: pick $r = \lfloor \log n \rfloor$

$\implies \frac{b(n+2^{\lfloor \log n \rfloor})}{\lfloor \log n \rfloor} = \Theta\left(\frac{n}{\log n}\right)$ (best we can do)

References:
T. Cormen et al., "Introduction to algorithms", Chap 8, MIT press (2022)

Comparison To Quicksort

LSD radix sort vs quicksort

Quicksort runtime is typically $\Theta(n \log n)$

LSD radix sort $\Theta(dn)$ when $k = O(n)$

Quicksort - typically more cache-friendly & in-place

LSD radix sort - less cache-friendly & not in-place

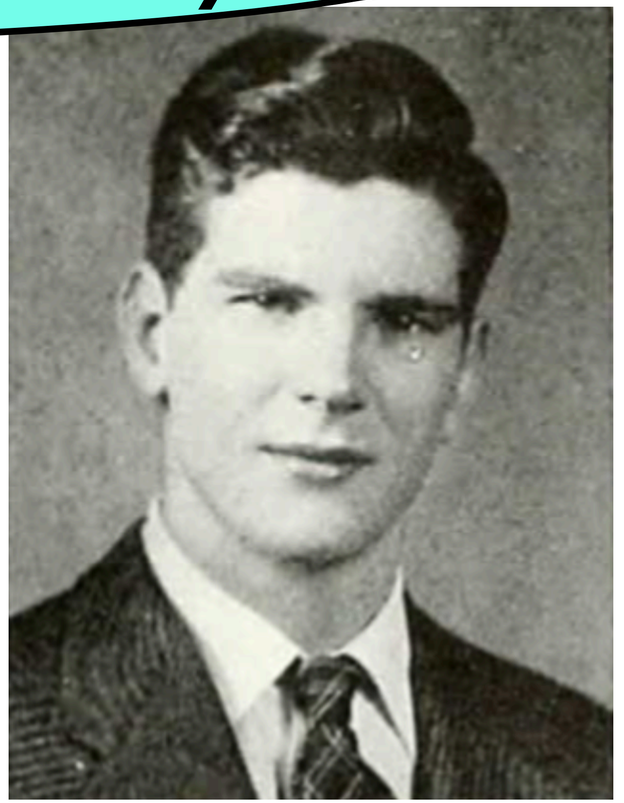
Best choice of sorting algorithm will depend on

- data distribution
- storage requirements
- machine details

References:
R. Sedgewick et al., "Radix Sorts", https://www.cs.princeton.edu/courses/archive/spr07/cos226/lectures/11RadixSort.pdf (2007)
T. Cormen et al., "Introduction to algorithms", Chap 8, MIT press (2022)
D. E. Knuth, "The art of computer programming, vol. 3: sorting and searching", Chap 5 (1998)

How can we *sort data* efficiently?

(when we know the *data distribution*)



Bucket Sort

useful when data distribution is known

Runtime complexity

Average case $\Theta(n)$

Properties

```
def bucket_sort(A):
    ...
```

Python Implementation

Bucket Sort

Bucket Sort

Distribution sort algorithm with three stages:

1. **Scatter** distribute keys to buckets
2. **Sort** sort keys within buckets
3. **Gather** gather sorted keys (in order)

Effective with knowledge of key distribution

Key idea - E. Isaac & R. Singleton (1955) 

Bucket sort complexity

Assume uniformly distributed keys

Runtime for n keys and b buckets:

Average-case: $\rightarrow \Theta\left(n + \frac{n^2}{b} + b\right)$

Worst–case: $\rightarrow \Theta(n^2 + b)$ (all keys in same bucket)

Storage: $\Theta(n)$ for items, sort requires: $\Theta(n + b)$

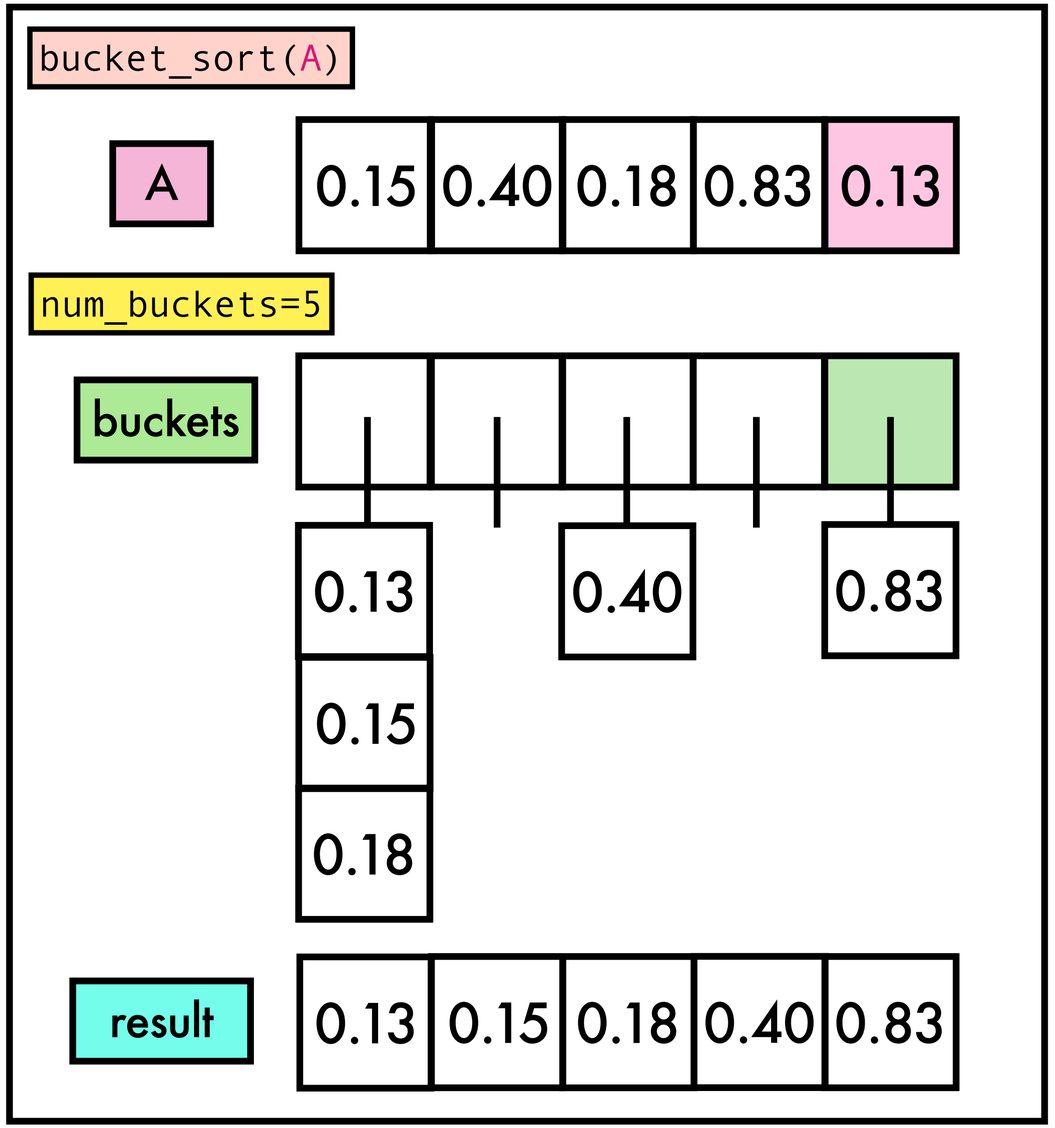
Typically, we choose $b \approx n$:

Average-case: $\rightarrow \Theta(n)$

References/Notes/Image credits:
E. J. Isaac and R. C. Singleton, "Sorting by address calculation", *Journal of the ACM* (1956)
(Earl Isaac image source) https://www.finnotes.org/people/earl-isaac
(R. Singleton image source) Massachusetts Institute of Technology Yearbook, Vol. 65 (1949)
https://en.wikipedia.org/wiki/Bucket_sort

Bucket Sort Python Implementation

```
def bucket_sort(A: list): # all keys fall in [0, 1)
    num_buckets = len(A)
    buckets = [[] for _ in range(num_buckets)]
    for key in A:   # scatter
        buckets[int(num_buckets * key)].append(key)
    for bucket in buckets:
        insertion_sort(bucket)
    return [x for bucket in buckets for x in bucket]  # gather
```



Reference:
https://en.wikipedia.org/wiki/Bucket_sort
T. Cormen et al., "Introduction to algorithms", Chap 8, MIT press (2022)

Bucket Sort Runtime Complexity

Runtime for average case when num. keys, n equals num. buckets, b

The scatter and gather operations involve simple for loops - each are $\Theta(n)$

Rest of cost is from calling `insertion_sort` on n buckets

Let m_i denote the number of keys in bucket i

$$\sum_{i=0}^{n-1} m_i = n$$

m_i has binomial distribution $Bin(n, p)$

Probability of falling in bucket i: $p = 1/n$

$$\mathbb{E}[m_i] = n \cdot 1/n = 1 \qquad \mathrm{Var}[m_i] = 1 - 1/n$$

Cost of n `insertion_sort` calls: $\sum_{i=0}^{n-1} O(m_i^2)$

Take expectation w.r.t key distribution $\mathbb{E}\left[\sum_{i=0}^{n-1} O(m_i^2)\right] = \sum_{i=0}^{n-1} O(\mathbb{E}[m_i^2])$

$$\mathrm{Var}[m_i] = \mathbb{E}[m_i^2] - \mathbb{E}^2[m_i]$$

$$\mathbb{E}[m_i^2] = \mathrm{Var}[m_i] + \mathbb{E}^2[m_i] = (1 - 1/n) + 1 = 2 - 1/n$$

Expected cost of n `insertion_sort` calls: $\sum_{i=0}^{n-1} O(2 - 1/n) = O(n)$

Average case bucket sort is $\Theta(n)$

References:
(CLRS) T. Cormen et al., "Introduction to algorithms", Chap 8, MIT press (2022)
For the general case when $n \neq b$, see https://en.wikipedia.org/wiki/Bucket_sort

Worst case (all in 1 bucket) - degrades to insertion_sort: $\Theta(n^2)$

A Variant Of Bucket Sort

a "surprisingly easy improvement"

Switching the execution order

Conventional bucket sort: Scatter | Sort | Gather

Variation of bucket sort: Scatter | Gather | Sort

This can be faster (due to greater cache-friendliness)

Note: whether this helps depends on implementation/hardware details

References:
E. Corwin et al., "Sorting in linear time - variations on the bucket sort", Journal of Computing Sciences in Colleges (2004)