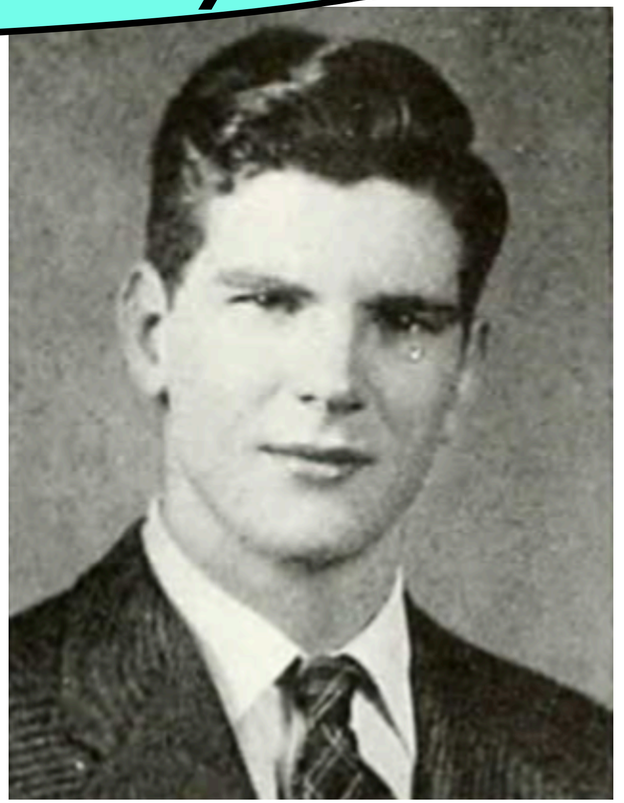
How can we *sort data* efficiently?

(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: $\Theta(n + b)$

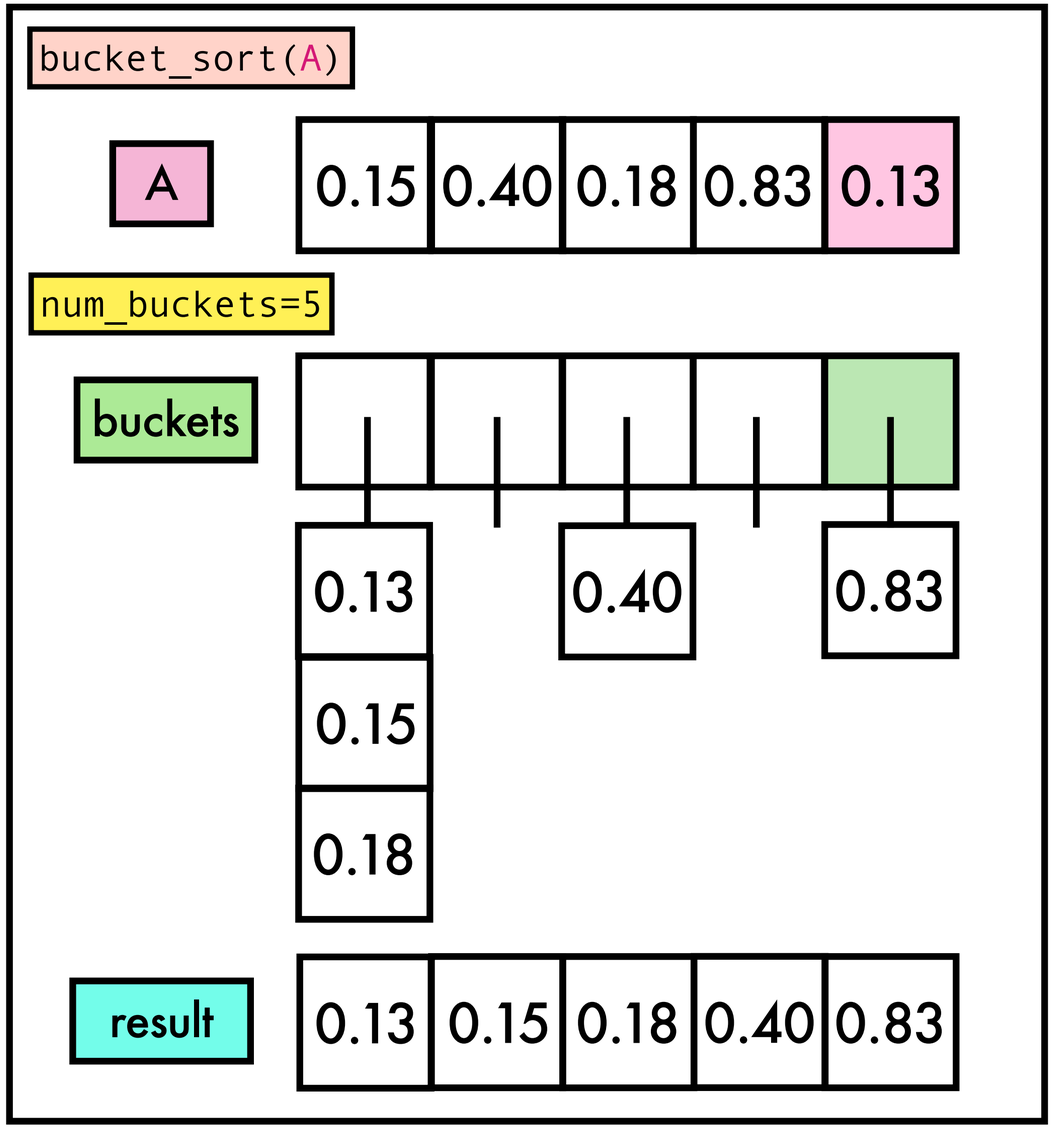
Typically, we choose $b \approx n$:

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)