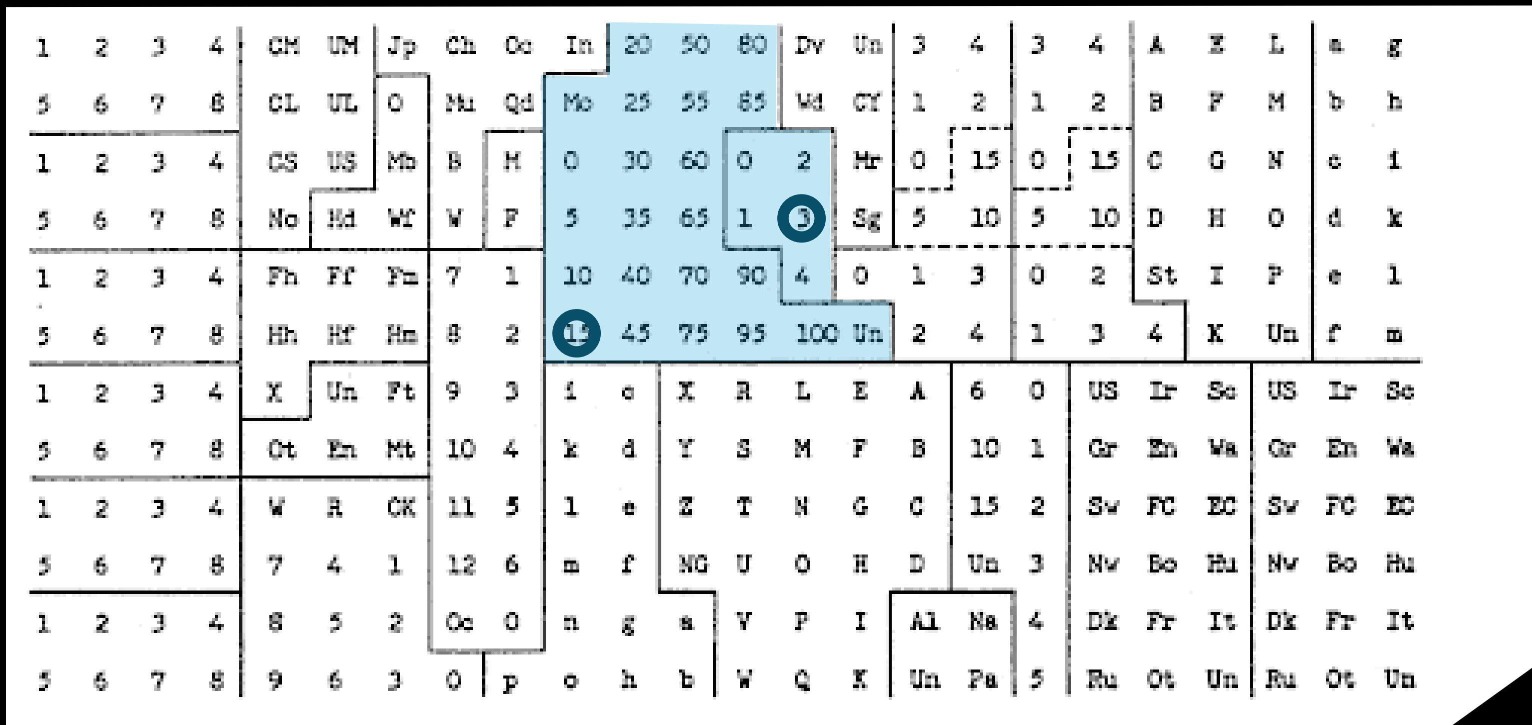
IBM 83 Card Sorter (1955) sorts 1000 cards per minute

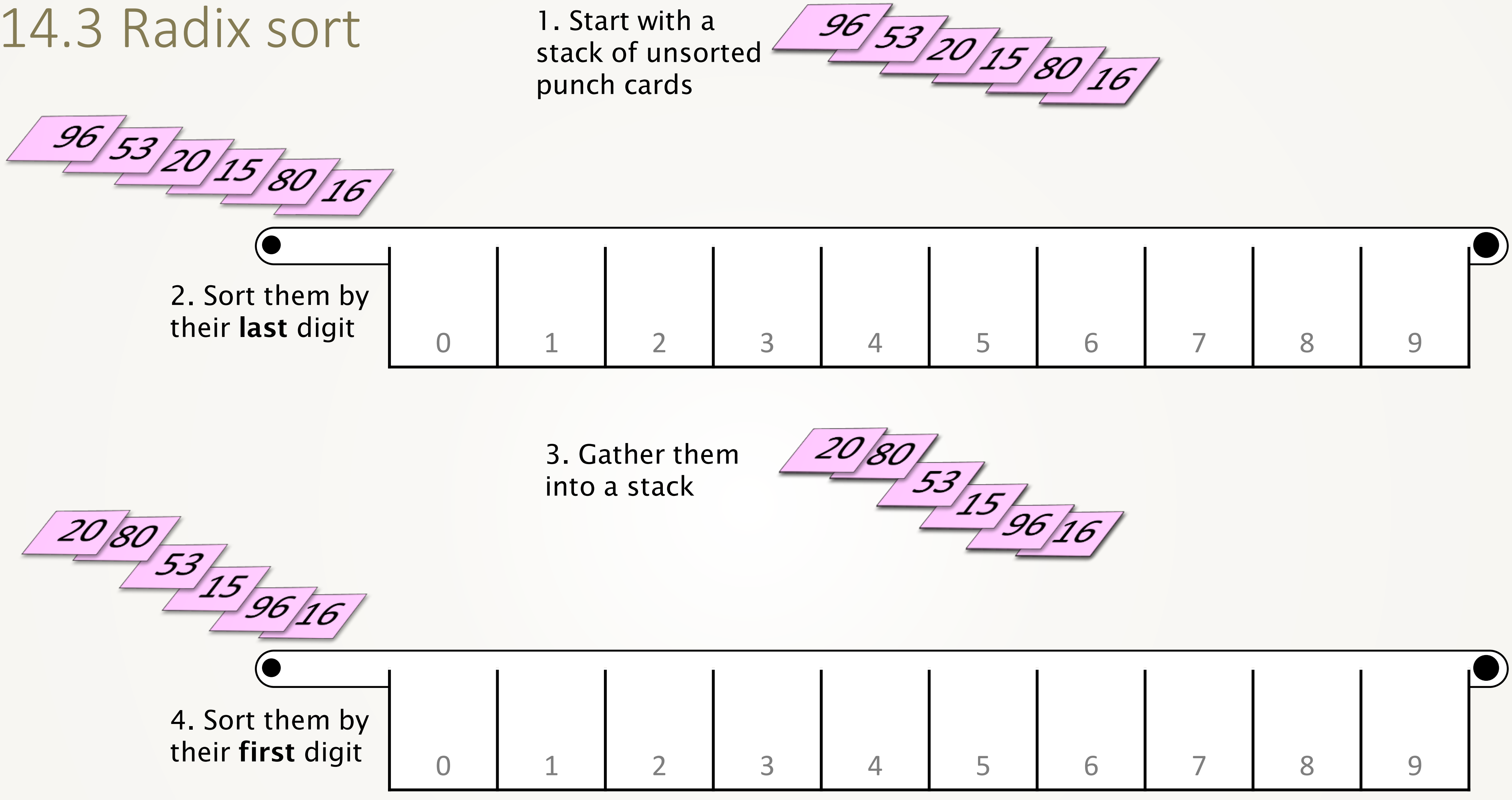
the "age" block
as punched for an 18 year old

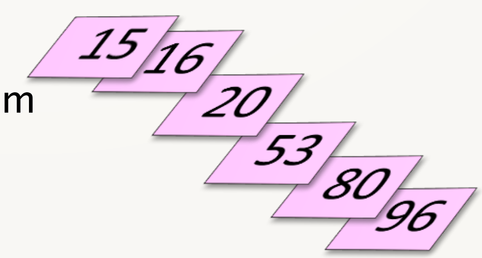


HOLLERITH 1890 CENSUS TABULATOR CARD
One of these cards was punched, for every census respondent

The tabulating machine could then be used to sort all the cards by age, using "radix sort" …

2.14.3 Radix sort



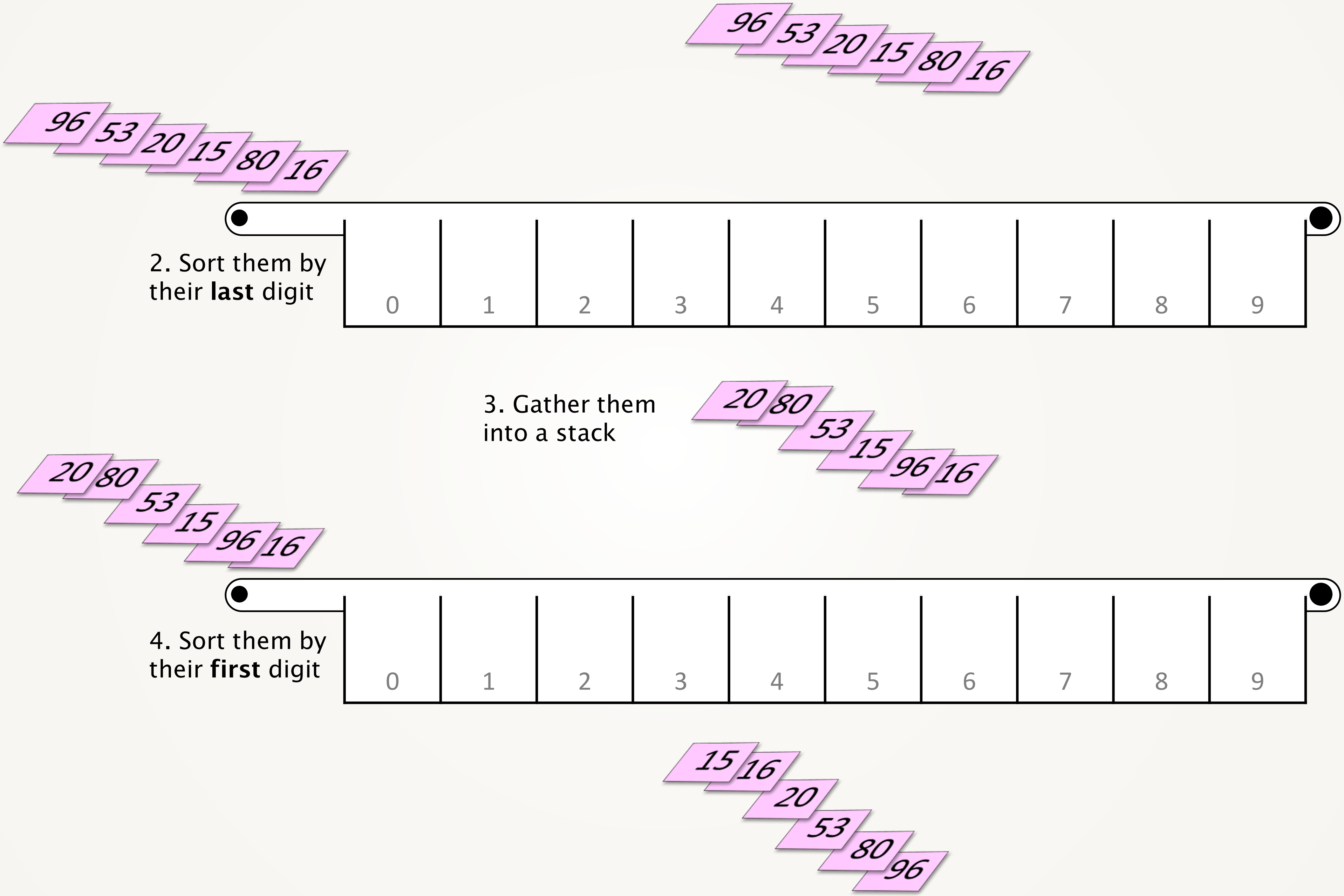


1000 cards / min.

Running time $= 2 \frac{n}{1000}$ min.

$\Theta(n)$

What sorcery is this!?


96 53 20 15 80 16
96 53 20 15 80 16
2. Sort them by their **last** digit
0 1 2 3 4 5 6 7 8 9
3. Gather them into a stack
20 80 53 15 96 16
20 80 53 15 96 16
4. Sort them by their **first** digit
0 1 2 3 4 5 6 7 8 9
15 16 20 53 80 96

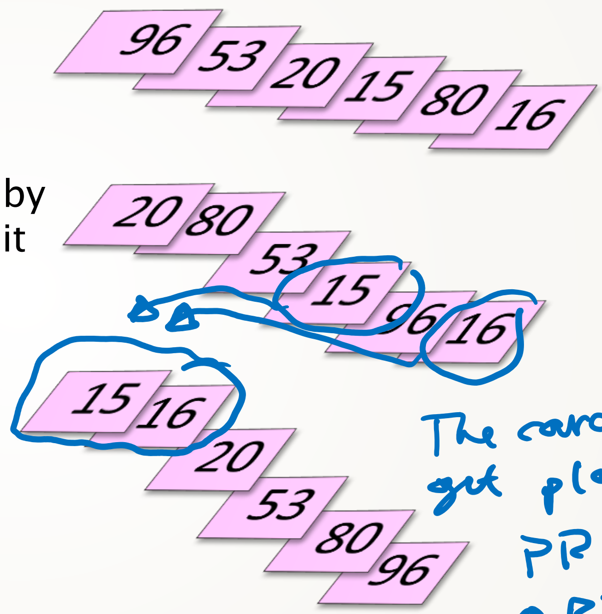
2.13 Stability

```
def radixsort(x):

  for each digit d, starting from
  the least significant:

    stably sort x by digit d

    # assert x is in order with
    # respect to digits d:end
```

A sorting algorithm is said to be *stable* if, for items with equal keys, their order in the input is preserved in the output.

2. Sort them by their **last** digit

4. Sort them by their **first** digit



The cards with first digit 1 get placed in bucket 1 PRESERVING THE ORDER THEY HAD PREVIOUSLY.

which, from the first pass, is order by 2nd digit.

Python’s built-in sort is stable.

If we want stability from a sorting algorithm that isn’t stable, simply extend the sort key to break ties by original position.

```
# WANT: stably sort these names by length
names = ['Charlie', 'Frances', 'Sian', 'Alex']

# 1. Extend the records to include their original position
names_ext = [(i,n) for i,n in enumerate(names)]

# 2. Sort by desired key, breaking ties by original position
sorted(names_ext, key = lambda v: (len(v[1]), v[0]))
```

```
[(2, 'Sian'), (3, 'Alex'), (0, 'Charlie'), (1, 'Frances')]
```

This takes space $\Theta(n)$ to store the extended records.

The cost of sorting

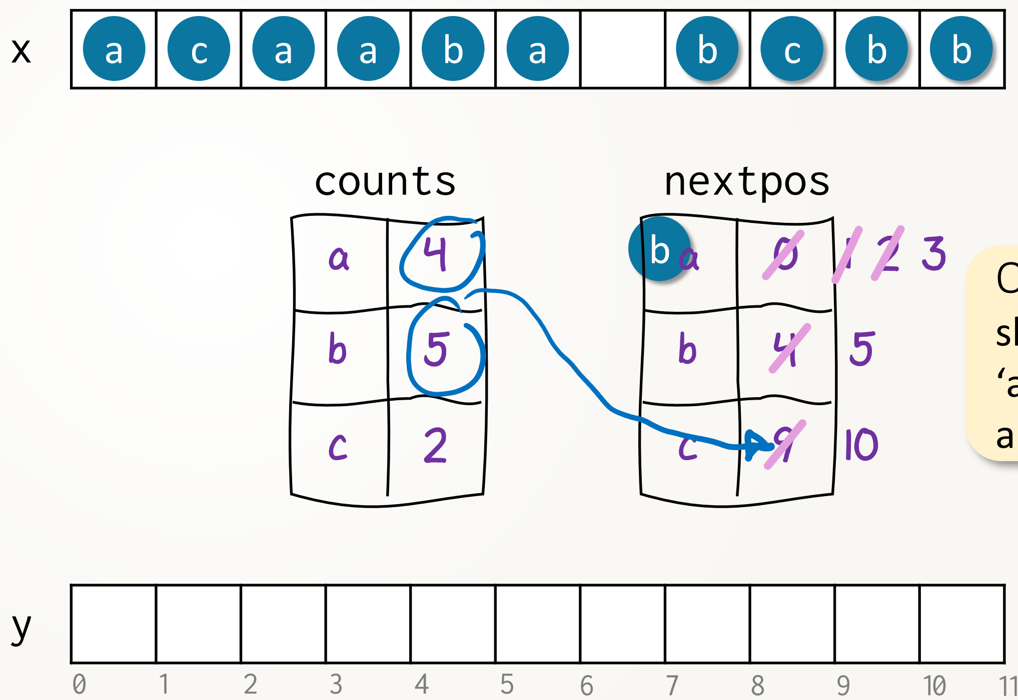
Algorithm	Worst-case running time
any algorithm	$\Omega(n \log n)$
InsertSort BinaryInsertSort SelectSort QuickSort	$\Theta(n^2)$
MergeSort HeapSort	$\Theta(n \log n)$
Tabulating machine	$\Theta(n)$

2.14.1 CountingSort

There's no "magic in the machine" that lets it bypass the running-time lower bound. Here's a code version of one pass of the tabulating machine.

Suppose we have n items to sort, and they are all integers in $\{1, \ldots, m\}$ where m is fixed.

```
def countingsort(x, m):
  # Count num.occurrences of each value
  counts = …

  # Figure out the first location
  # for each possible value
  nextpos = …  cumsum (counts)

  y = new array of same size as x

  # Go through x and place each item
  # into its correct location
  for each value v in x:
    y[nextpos[v]] = v
    nextpos[v] += 1

  return y
```



Q. What position should the first 'a' go in? And 'b', and 'c'?

Running time:

compute counts $\Theta(n)$

cumsum $O(1)$

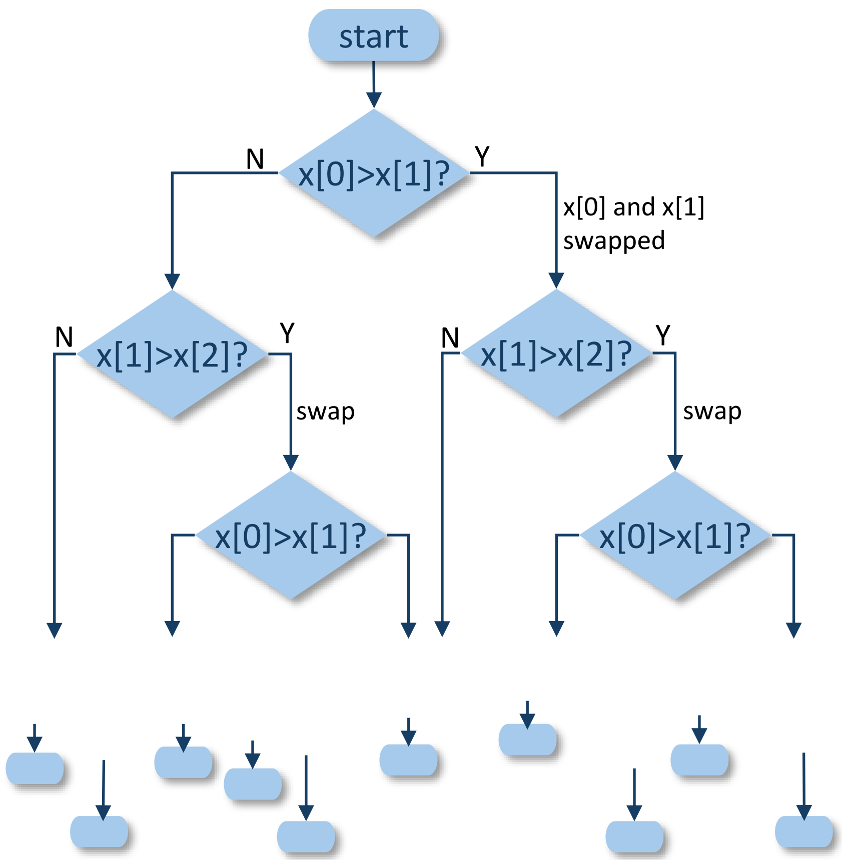
place items $\Theta(n)$

Theorem. Given any sorting algorithm, let $f(n)$ be its worst-case number of comparisons for inputs of size n. Then $f(n)$ is $\Omega(n \log n)$.

$f(n) = \max_{x:\ \text{size}(x)=n} g(x)$ where $g(x) =$ # comparisons used to sort input x.

Proof. Consider an arbitrary input x with $\text{size}(x) = n$. Consider the algorithm's decision tree

i.e. a tree where each node represents the state of the data structure just before it makes a comparison whose outcome depends on x.



Every path through this tree corresponds to a specific sequence of operations, and results in a specific permutation of the items of x.

There are $n!$ possible orderings of the items of x. Each ordering requires a different permutation to sort it. Therefore this tree has $\#\text{leaves} \geq n!$

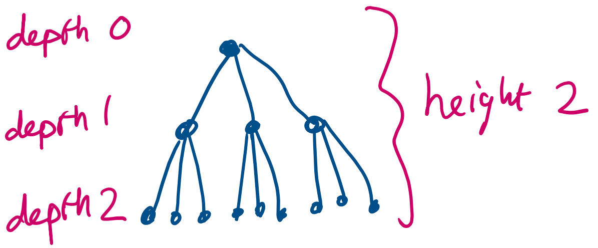
Note that any binary tree of height h has $\#\text{leaves} \leq 2^h$.

The height of this tree is $f(n)$, and so

$$n! \leq \#\text{leaves} \leq 2^{f(n)} \quad \text{hence} \quad f(n) \geq \log_2 n!$$

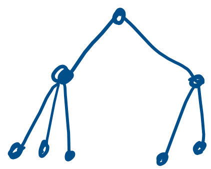
Since $\log_2 n! = \Theta(n \log n)$, we conclude $f(n)$ is $\Omega(n \log n)$.

Refresher about the geometry of trees:



A tree of height h whose nodes all have degree d has # leaves $= h^d$

____________________ $\leq d$ ____________ $\leq$ ____

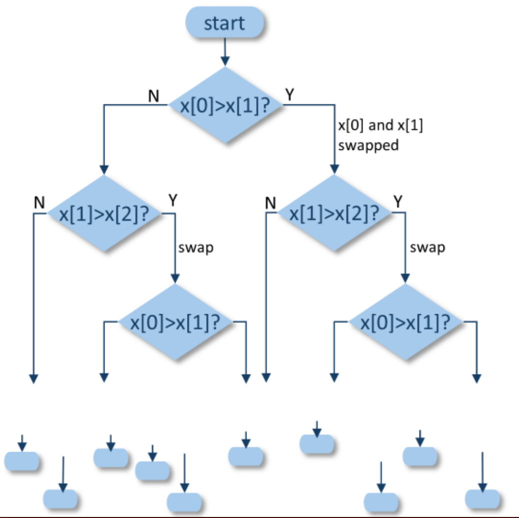


A tree with n leaves and maximum degree d has height $h \geq \log_d n$.

Theorem. Given any sorting algorithm, let $f(n)$ be its worst-case number of comparisons for inputs of size n. Then $f(n)$ is $\Omega(n \log n)$.

Proof. Consider an arbitrary input x with $\text{size}(x) = n$. Consider the algorithm's decision tree

i.e. a tree where each node represents the state of the data structure just before it makes a comparison whose outcome depends on x.



Every path through this tree corresponds to a specific sequence of operations, and results in a specific permutation of the items of x.

There are $n!$ possible orderings of the items of x. Each ordering requires a different permutation to sort it. Therefore this tree has $\#\text{leaves} \geq n!$

Note that any binary tree of height h has $\#\text{leaves} \leq 2^h$.

The height of this tree is $f(n)$, and so

$$n! \leq \#\text{leaves} \leq 2^{f(n)} \quad \text{hence} \quad f(n) \geq \log_2 n!$$

Since $\log_2 n! = \Theta(n \log n)$, we conclude $f(n)$ is $\Omega(n \log n)$.

There is a flaw in this reasoning; and if we know something about the keys, we can do better than $\Omega(n \log n)$.

What's the flaw? PLEASE ANSWER ON MOODLE

I am ∃loise. There **exists** a proof step which is wrong, and there **exists** a concrete example which demonstrates it's wrong.

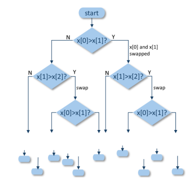
LET ME SHOW THEM TO YOU.

I am ∀belard. **All** the steps of my proof are true in **all** cases.

Theorem. Given any sorting algorithm, let $f(n)$ be its worst-case number of comparisons for inputs of size n. Then $f(n)$ is $\Omega(n \log n)$.

Proof. Consider an arbitrary input x with $\text{size}(x) = n$. Consider the algorithm's decision tree

i.e. a tree where each node represents the state of the data structure just before it makes a comparison whose outcome depends on x.



Every path through this tree corresponds to a specific sequence of operations, and results in a specific permutation of the items of x.

There are $n!$ possible orderings of the items of x. Each ordering requires a different permutation to sort it. Therefore this tree has $\#\text{leaves} \geq n!$

Note that any binary tree of height h has $\#\text{leaves} \leq 2^h$.

The height of this tree is $f(n)$, and so

$$n! \leq \#\text{leaves} \leq 2^{f(n)} \quad \text{hence} \quad f(n) \geq \log_2 n!$$

Since $\log_2 n! = \Theta(n \log n)$, we conclude $f(n)$ is $\Omega(n \log n)$.

There is a flaw i … asoning; and if we know something abou … e keys, we can do better than $\Omega(n \log n)$.

What's the flaw? PLEASE ANSWER ON MOODLE

2.14.2 BucketSort

Here's a different way to benefit from extra information about the sort keys.

Let's assume they are uniformly distributed in the range [0,1].

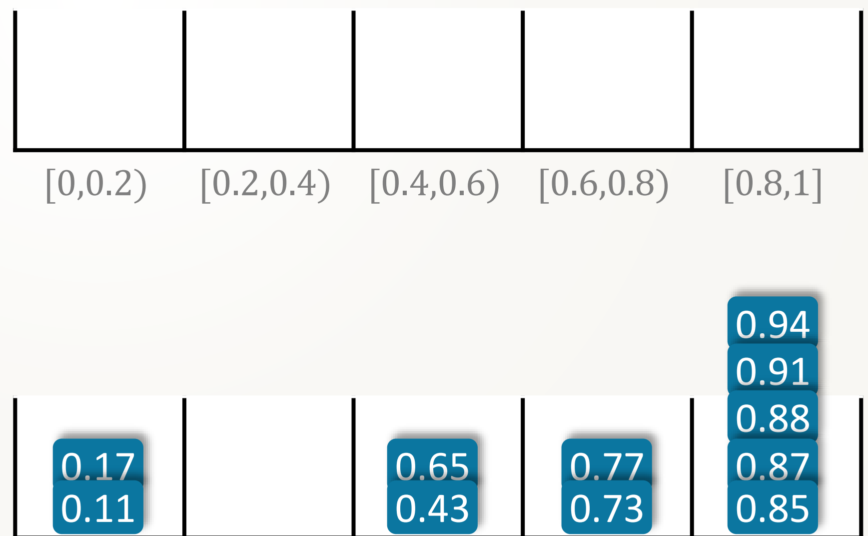
set to some small value, e.g. $a = 2.5$

```
def bucketsort(x, a):
  B = ⌈len(x)/a⌉
  buckets = array of B empty lists

  for each item v in x:
    append v to bucket ⌊key(v) × B⌋

  # assert: average number of items
  # in each bucket is ≈a

  for each bucket:
    sort it with a O(n²) algorithm
    output its values
```

$n = \text{len}(x) = 11$ items to be sorted

$a = 2.5$

$\rightarrow B = \lceil \frac{n}{a} \rceil = 5$



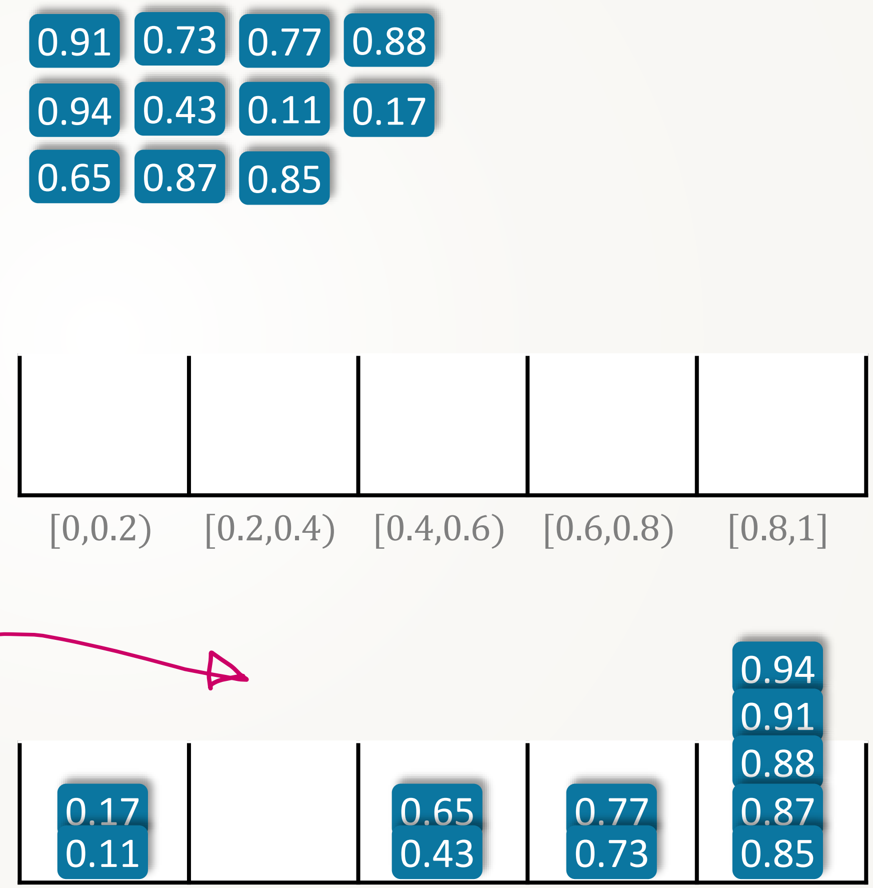
Average case running time:

- We choose a small constant a, e.g. $a = 2.5$
- We use $B = \lceil \frac{n}{a} \rceil \approx \frac{n}{a}$ buckets: takes time $\Theta(n)$ to allocate them.
- on average, each bucket has $\frac{\#\text{ items}}{\#\text{ buckets}} = \frac{n}{B} \approx \frac{n}{n/a} = a$ items.
- Sorting a items takes $O(1)$. Even though we're using an $O(n^2)$ algorithm, we're only sorting a small fixed number of items.
- Total time $=$ # buckets $\times$ time to sort a bucket $= B \times \text{const} \approx \frac{n}{a} \times \text{const} = \Theta(n)$.

2.14.2 BucketSort

Here's a different way to benefit from extra information about the sort keys.
Let's assume they are uniformly distributed in the range [0,1].

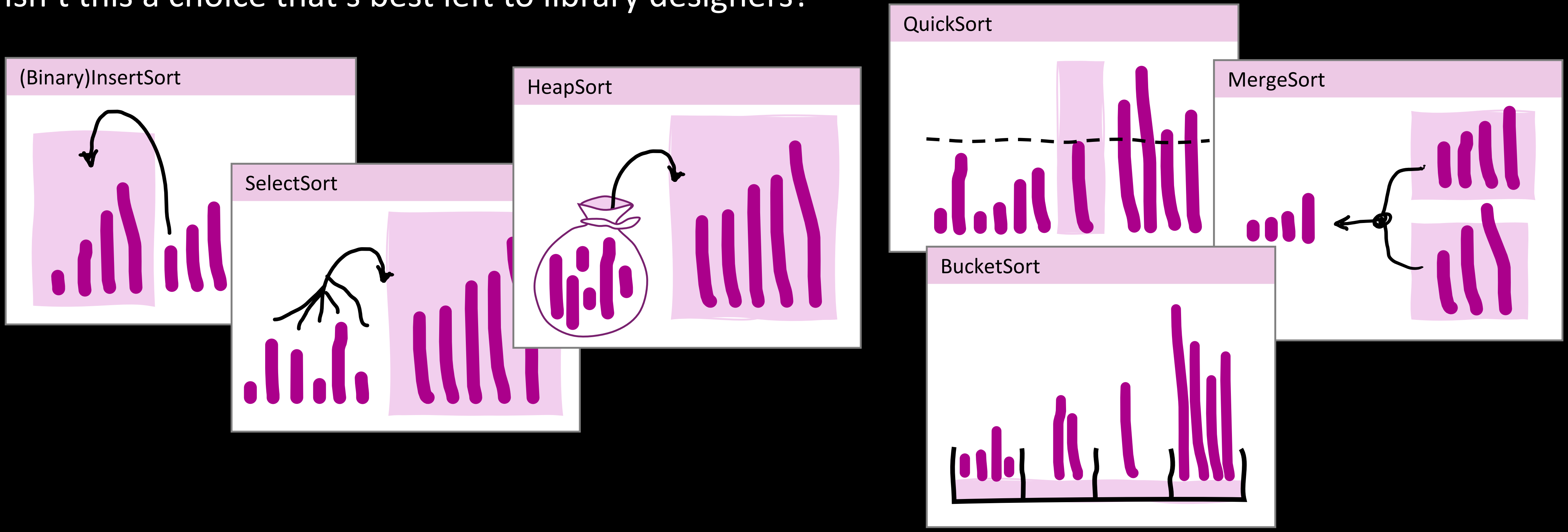
```
def bucketsort(x, a):
  B = ⌈len(x)/a⌉
  buckets = array of B empty lists

  for each item v in x:
    append v to bucket ⌊key(v) × B⌋

  # assert: average number of items
  # in each bucket is ≈a

  for each bucket:
    sort it with a O(n²) algorithm
    output its values
```



The average case analysis says "each bucket has ≈ a items". In fact, the bucket sizes are random: # items in a given bucket is a Bin$(n, \frac{1}{B})$ random variable, assuming the keys are uniformly distributed. If we're really unlucky we get all keys in one bucket. But the average case is still $\Theta(n)$. (There's a straightforward proof in the CLRS textbook.)

Who really needs to learn about sorting algorithms?

Isn’t this a choice that’s best left to library designers?



The point is to learn general lessons about algorithm design

- Always bear in mind the algorithm’s worst-case cost, in both running time and extra memory
- … but don’t take it too seriously: constants matter, and typical-case performance is important too
- … and there may be other criteria e.g. is it in-place? is it stable?

- If our algorithm’s O doesn’t match the problem’s Ω, we’re missing something
- … but maybe there’s special structure and a general-purpose lower bound isn’t right

- Identify the strong points of each algorithm. Mix-and-match, and re-use strategies.

Typical-case performance is important too

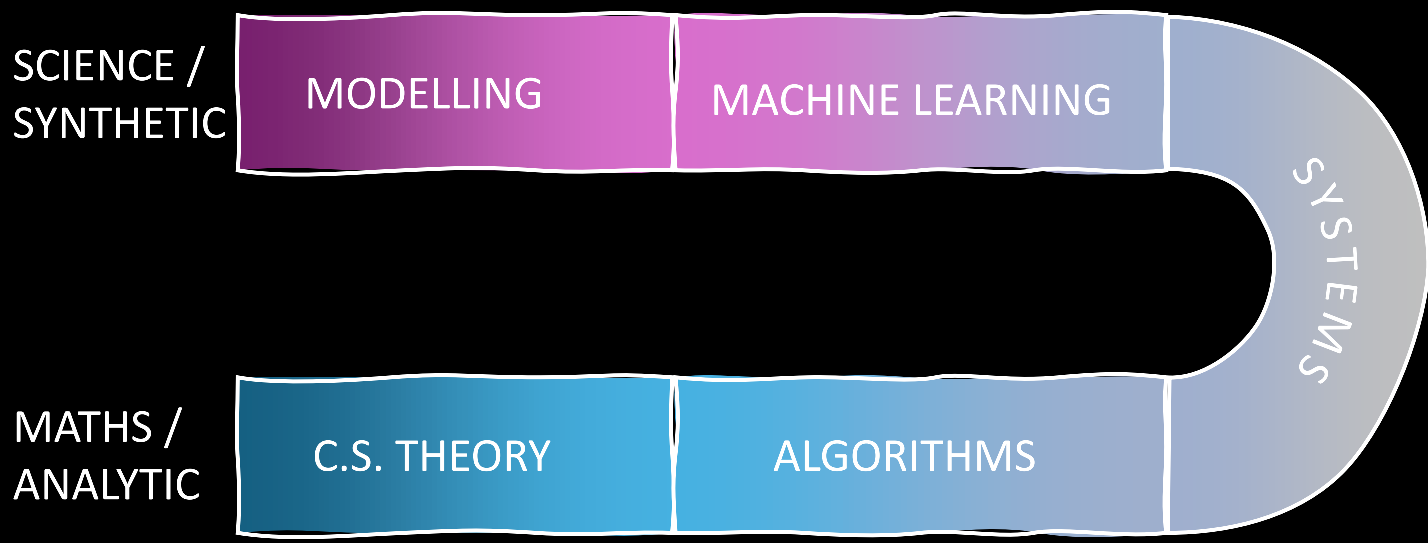
Python ≥2.3 uses TimSort, by Tim Peters.

“It has supernatural performance on many kinds of partially ordered arrays (less than $\log_2 n!$ comparisons needed, and as few as $n-1$).

“On arrays with many kinds of pre-existing order, this blows [the previous algorithm] out of the water. I believe that lists very often do have exploitable partial order in real life, and this is the strongest argument in favor of timsort.

“In a nutshell, the main routine marches over the array once, left to right, alternately identifying the next run, then merging it into the previous runs ‘intelligently’. Everything else is complication for speed, and some hard-won measure of memory efficiency.”

https://github.com/python/cpython/blob/main/Objects/listsort.txt



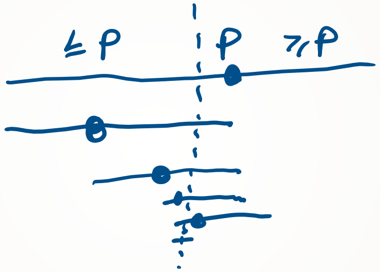
Let's now try to apply these lessons ...

2.12 Computing statistics

A *statistic* is a numerical summary of a dataset.
Examples: mean, median, maximum, variance, skewness, kurtosis.

QUESTION
We want to compute the **median** of an array of length n. What's a fast way to do this?

We could just sort the list and pick the middle item. But that's very wasteful. A cunning trick, based on quicksort, is:



(1) Do a "partition" pass. If the pivot ends up in the right half, we know that more than half the list is $\leq p$, so the median must be in the left half.

(2) Descend into the half that contains the median. Repeat.

(3) Stop when we've found the median

QUESTION
We want to compute the **maximum** of an array of length n. Give a lower bound on the number of comparisons needed.

By analogy with the $\Omega(n \log n)$ lower bound for sorting ....
There are n possible items that might be the maximum, ie the decision tree has n leaves
So height $\geq \log_2 n$. So $\boxed{\#\text{comparisons} \geq \log_2 n.}$
This rather surprising bound can actually be achieved (in some settings).

The final big lesson from this part of the course...

When we say $f(n)$ is $O(g(n))$, or Ω, or Θ, this is a "neutral" maths statement about two functions.

Here's how we typically use it in analysing algorithms:

"Let $f(n)$ be the worst-case cost over all inputs of size n. Then $f(n)$ is $O(g(n))$."

When we say "My algorithm is $O(n^2)$" we really mean "the worst-case is $O(n^2)$".

"Let $f(n)$ be the cost for a particular input I have designed of size n. Then $f(n)$ is $\Theta(g(n))$."

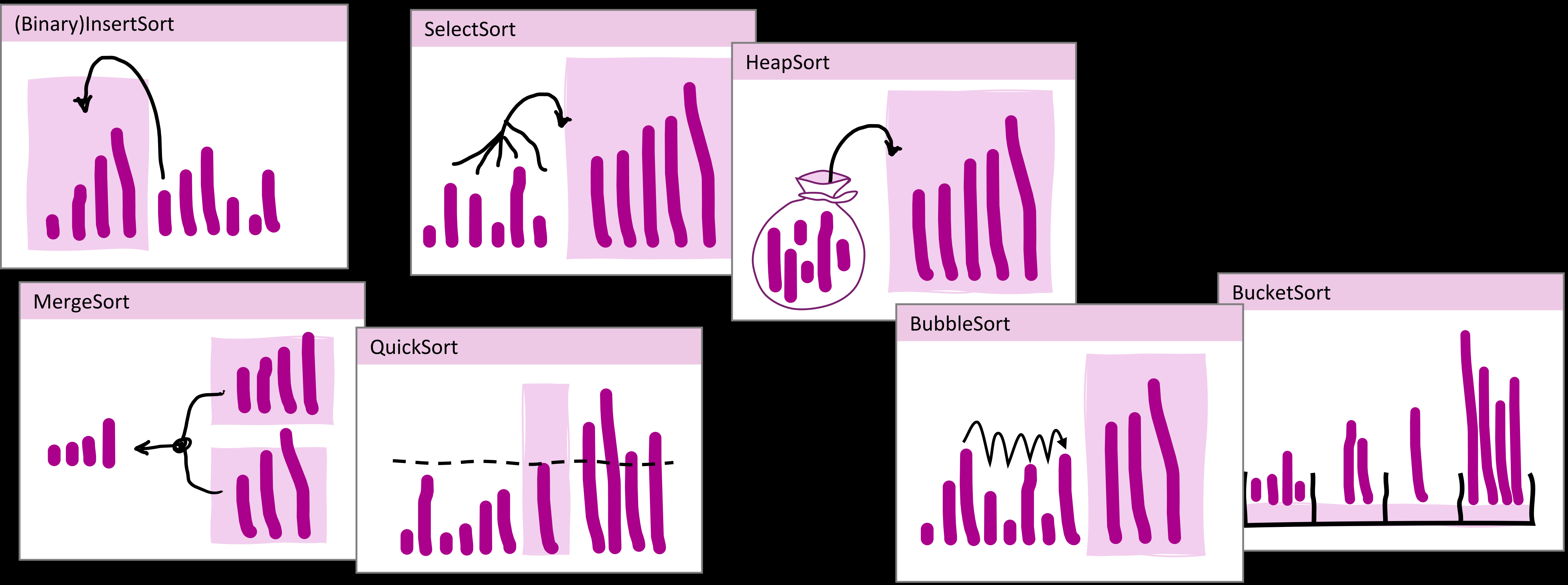
When we construct a particular troublesome input for which the cost is $\Theta(n^2)$, we can conclude "The worst case for my algorithm is $\Omega(n^2)$". (The actual worst case might be worse than our troublesome input, so we have to use Ω.)

"Let $f(n)$ be the best-case cost over all inputs of size n. Then $f(n)$ is $\Omega(g(n))$."

We might informally say "My algorithm is $\Omega(n^2)$", meaning that the best case is $\Omega(n^2)$.

"Let $f(n)$ be the expected (i.e. average) cost for a random input of size n. Then $f(n)$ is $\Theta(g(n))$."

The last two slides weren’t covered in the lecture.



Verifying correctness of algorithms

- Assertions are a stonking good idea
- … especially *invariants*, e.g. assertions of the form “at iteration i, the current state $\mathcal{S}$ satisfies property $P_i(\mathcal{S})$”

2.8 BubbleSort

This algorithm isn't useful for anything at all. Learn it, so you don't reinvent it!
(Plus there's a nice invariant for analysing its performance.)

```
def bubblesort(x):
  while True:
    any_swaps = False
    for i in 0..(len(x)-2):
      if x[i] > x[i+1]:
        swap x[i] with x[i+1]
        any_swaps = True
    if not any_swaps:
      break
```

QUESTION. What's the maximum number of passes needed?