IBM 83 Card Sorter (1955) sorts 1000 cards per minute
The "age" block
as punched for an 18 year old

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

1. Start with a stack of unsorted punch cards

2. Sort them by their last digit

3. Gather them into a stack

4. Sort them by their first digit

5. Gather them into a stack

1000 cards (1 min.
Running time = \( \frac{2n}{1000} \) min.
\( \Theta(n) \))
What sorcery is this!?
2. Sort them by their last digit

3. Gather them into a stack

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

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.

```python
# 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

<table>
<thead>
<tr>
<th>Algorithm</th>
<th>Worst-case running time</th>
</tr>
</thead>
<tbody>
<tr>
<td>any algorithm</td>
<td>$\Omega(n \log n)$</td>
</tr>
<tr>
<td>InsertSort</td>
<td>$\Theta(n^2)$</td>
</tr>
<tr>
<td>BinaryInsertSort</td>
<td></td>
</tr>
<tr>
<td>SelectSort</td>
<td></td>
</tr>
<tr>
<td>QuickSort</td>
<td></td>
</tr>
<tr>
<td>MergeSort</td>
<td>$\Theta(n \log n)$</td>
</tr>
<tr>
<td>HeapSort</td>
<td></td>
</tr>
<tr>
<td>Tabulating machine</td>
<td>$\Theta(n)$</td>
</tr>
</tbody>
</table>
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.

```python
def counting_sort(x, m):
    # Count num. occurrences of each value
    counts = ...
    # Figure out the first location
    # for each possible value
    nextpos = ...
    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
```

Running time:
- compute counts \( \mathcal{O}(n) \)
- compute \( \text{cumsum} \) \( \mathcal{O}(n) \)
- place items \( \mathcal{O}(n) \)

Q. What position should the first ‘a’ go in? And ‘b’, and ‘c’?
Theorem. Given any sorting algorithm, let $f(n)$ be its worst-case number of comparisons for inputs of size $n$. Then $f(n)$ is $\Omega(n \log n)$.

$$f(n) = \max_{\text{size}(x) = n} g(x)$$

where $g(x)$ is the number of comparisons used to sort input $x$.

Proof. Consider an arbitrary input $x$ with 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 $n! \leq \#\text{leaves} \leq 2^{f(n)}$ hence $f(n) \geq \log_2 n!$

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 $d$ have $\#\text{leaves} = h^d \leq d \leq 1$

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

```python
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 $O(n^2)$ algorithm
        output its values
```

Average case running time:
- We choose a small constant $a$, e.g. $a = 2.5$
- We use $B = \lceil \frac{n}{a} \rceil$ buckets:
  - Takes time $\Theta(n)$ to allocate them.
- On average, each bucket has $\frac{n}{B} \approx \frac{n}{a}$ 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 $= B \times \text{const} \approx \frac{n}{a} \times \text{const} = \Theta(n)$.

$n = \text{len}(x) = 11$ items to be sorted
$\Rightarrow B = \lceil \frac{n}{a} \rceil = 5$

<table>
<thead>
<tr>
<th>[0,0.2)</th>
<th>[0.2,0.4)</th>
<th>[0.4,0.6)</th>
<th>[0.6,0.8)</th>
<th>[0.8,1]</th>
</tr>
</thead>
<tbody>
<tr>
<td>0.85</td>
<td>0.91</td>
<td>0.73</td>
<td>0.77</td>
<td>0.88</td>
</tr>
<tr>
<td>0.87</td>
<td>0.94</td>
<td>0.43</td>
<td>0.11</td>
<td>0.17</td>
</tr>
<tr>
<td>0.65</td>
<td>0.87</td>
<td>0.85</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

0.17 0.11
0.65 0.43
0.77 0.73
0.94 0.91
0.88 0.87
0.85
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 $[0,1]$.

```python
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 $O(n^2)$ algorithm
        output its values
```

The average case analysis says "each bucket has $\approx a$ items".

In fact, the bucket sizes are random. The number of 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 $O(n)$. (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 $\Omega$, 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 log₂ 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
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?

**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, i.e. the decision tree has $n$ leaves

So height $\geq \log_2 n$. So

$\text{#comparisons} \geq \log_2 n$. 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 $\Omega$, or $\Theta$, 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 $O(n^2)$, 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 $\Omega$.)

“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 $S$ satisfies property $P_i(S)$”
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.)

```python
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?