# Lecture 19: Sorting

**Comp 102**

Fakhir Shaheen

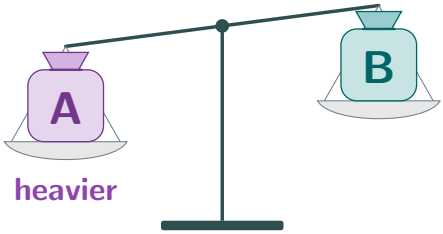
Forman Christian University

# Three Bags, Two Weighings

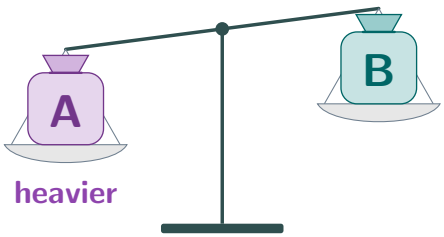
# The Shopkeeper's Puzzle



3 **different** weights



# The Shopkeeper's Puzzle



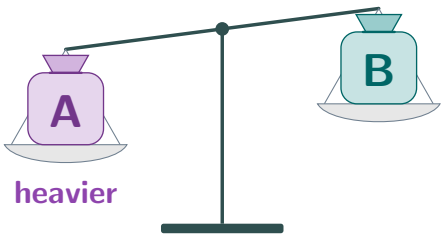



3 **different** weights




only **heavier**: no numbers

# The Shopkeeper's Puzzle






3 **different** weights

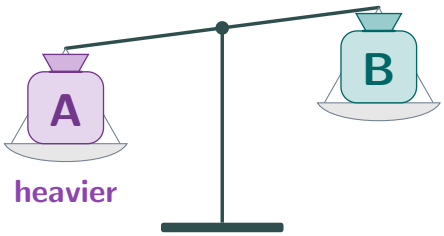



only **heavier**: no numbers




a line, **lightest first**

# The Shopkeeper's Puzzle










3 **different** weights

only **heavier**: no numbers

a line, **lightest first**

Can you **always** do it with only 2 weighings?

*write YES or NO (30 s)*

# You Try! *(fill the last column, 2 min)*

One plan: weigh **A** and **B**. Then weigh **B** and **C**. **A** < **B**: **A** is lighter

| 1st weighing | 2nd weighing | the line, lightest first |
|---|---|---|
| **A** < **B** | **B** < **C** | |
| **A** < **B** | **C** < **B** | |
| **B** < **A** | **B** < **C** | |
| **B** < **A** | **C** < **B** | |

# You Try! *(fill the last column, 2 min)*

One plan: weigh **A** and **B**. Then weigh **B** and **C**. **A** < **B**: **A** is lighter

| 1st weighing | 2nd weighing | the line, lightest first |
|---|---|---|
| **A** < **B** | **B** < **C** | **A B C** ✓ |
| **A** < **B** | **C** < **B** | |
| **B** < **A** | **B** < **C** | |
| **B** < **A** | **C** < **B** | **C B A** ✓ |

# You Try! *(fill the last column, 2 min)*

One plan: weigh **A** and **B**. Then weigh **B** and **C**. **A** < **B**: **A** is lighter

| 1st weighing | 2nd weighing | the line, lightest first |
|---|---|---|
| **A** < **B** | **B** < **C** | **A B C** ✓ |
| **A** < **B** | **C** < **B** | **A C B** or **C A B** ? |
| **B** < **A** | **B** < **C** | |
| **B** < **A** | **C** < **B** | **C B A** ✓ |

# You Try! *(fill the last column, 2 min)*

One plan: weigh **A** and **B**. Then weigh **B** and **C**. **A** < **B**: **A** is lighter

| 1st weighing | 2nd weighing | the line, lightest first |
|---|---|---|
| **A** < **B** | **B** < **C** | **A B C** ✓ |
| **A** < **B** | **C** < **B** | **A C B** or **C A B** ? |
| **B** < **A** | **B** < **C** | **B A C** or **B C A** ? |
| **B** < **A** | **C** < **B** | **C B A** ✓ |

# You Try! *(fill the last column, 2 min)*

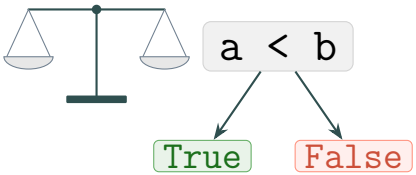
One plan: weigh **A** and **B**. Then weigh **B** and **C**. **A** < **B**: **A** is lighter

| 1st weighing | 2nd weighing | the line, lightest first |
|---|---|---|
| **A** < **B** | **B** < **C** | **A B C** ✓ |
| **A** < **B** | **C** < **B** | **A C B** or **C A B** ? |
| **B** < **A** | **B** < **C** | **B A C** or **B C A** ? |
| **B** < **A** | **C** < **B** | **C B A** ✓ |

this plan fails **2** times out of 4

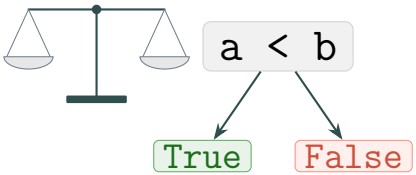
a cleverer plan? **Proof** today.

# Python Has the Same Balance



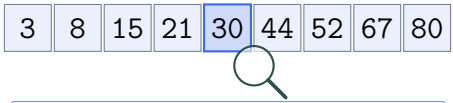

only **2** answers

# Python Has the Same Balance




only **2** answers

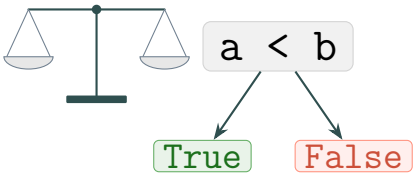
Lecture 18: bisection search




100 billion items: **37** looks

only if the list is **sorted**

# Python Has the Same Balance




only **2** answers

Lecture 18: bisection search

| 3 | 8 | 15 | 21 | 30 | 44 | 52 | 67 | 80 |
|---|---|---|---|---|---|---|---|---|

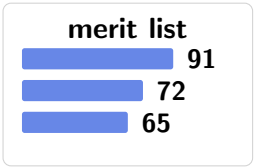
100 billion items: **37** looks

only if the list is **sorted**

**Today:** 1 a first sort → 2 a sieve for **primes** → 3 a **speed limit** → 4 a way **past** it

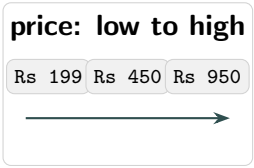
# Why Sort?

# Sorted Lists Are Everywhere



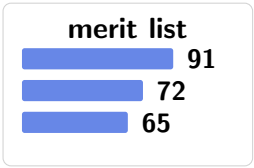






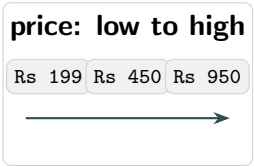



# Sorted Lists Are Everywhere









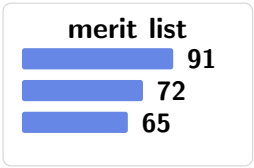




a sorted list makes jobs **easy**

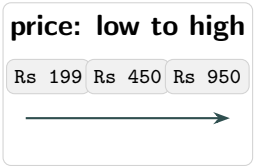
| 3 | 8 | 8 | 15 | 21 | 30 | 44 |
|---|---|---|---|---|---|---|

# Sorted Lists Are Everywhere





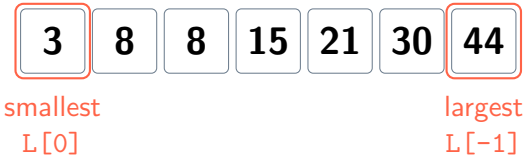




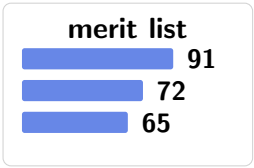




a sorted list makes jobs **easy**



# Sorted Lists Are Everywhere









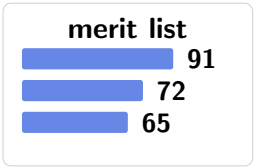




a sorted list makes jobs **easy**



# Sorted Lists Are Everywhere











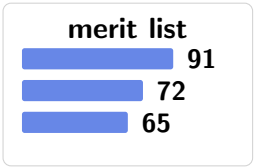


a sorted list makes jobs **easy**

repeats side by side



# Sorted Lists Are Everywhere













a sorted list makes jobs **easy**

repeats side by side






bisection: $O(\log n)$

# Finding a Repeat, Faster

The job: is any value in L **twice**? `[21, 8, 44, 3, 8]` → `True`

**slow** check **every pair**

```
# Lecture 18's mystery
def any_repeat(L):
    for i in range(len(L)):
        for j in range(i + 1, len(L)):
            if L[i] == L[j]:
                return True
    return False
```




10 pairs
$O(n^2)$

# Finding a Repeat, Faster

The job: is any value in L **twice**?

`[21, 8, 44, 3, 8]` → `True`

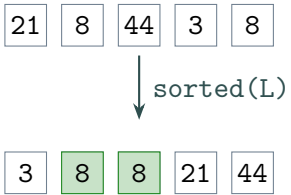
**slow** check **every pair**

```
# Lecture 18's mystery
def any_repeat(L):
    for i in range(len(L)):
        for j in range(i + 1, len(L)):
            if L[i] == L[j]:
                return True
    return False
```




10 pairs
$O(n^2)$

**better** **sort first**



# Finding a Repeat, Faster

The job: is any value in L **twice**?

`[21, 8, 44, 3, 8]` → `True`

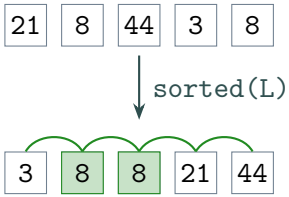
**slow** check **every pair**

```
# Lecture 18's mystery
def any_repeat(L):
    for i in range(len(L)):
        for j in range(i + 1, len(L)):
            if L[i] == L[j]:
                return True
    return False
```




10 pairs
$O(n^2)$

**better** **sort first**




4 checks
$O(n)$

a repeat lands **next to** itself,
so check only **neighbours**:

```
if L[i] == L[i + 1]:
```

# Finding a Repeat, Faster

The job: is any value in `L` **twice**?

`[21, 8, 44, 3, 8]` → `True`

**slow** check **every pair**

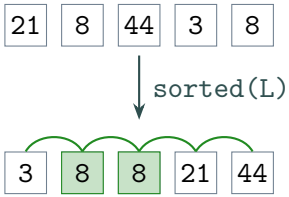
```
# Lecture 18's mystery
def any_repeat(L):
    for i in range(len(L)):
        for j in range(i + 1, len(L)):
            if L[i] == L[j]:
                return True
    return False
```




10 pairs
$O(n^2)$

total: **sort** once + $n - 1$ checks

**better** **sort first**




4 checks
$O(n)$

a repeat lands **next to** itself,
so check only **neighbours**:

```
if L[i] == L[i + 1]:
```

a win only if sorting beats $n^2$. Does it?

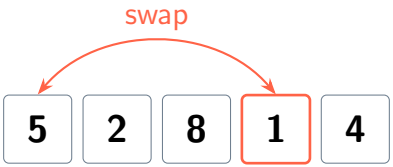
# Selection Sort

# Sorting Cards on a Table



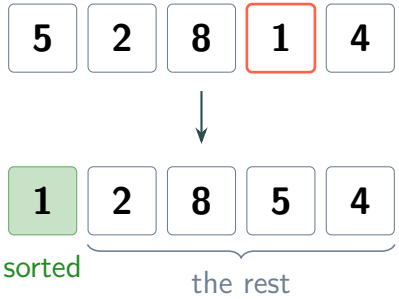

1 find the **smallest**

# Sorting Cards on a Table



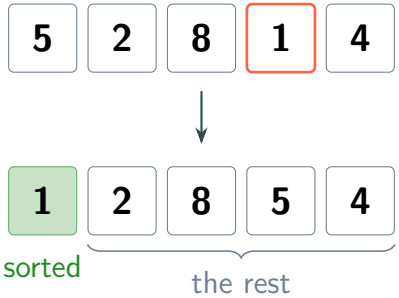

1. find the **smallest**
2. swap it to the **front**

# Sorting Cards on a Table




1. find the **smallest**
2. swap it to the **front**
3. do the same for **the rest**

# Sorting Cards on a Table




1. find the **smallest**
2. swap it to the **front**
3. do the same for **the rest**

each **pass**: the **green** part grows by **one** card

# Selection Sort, Step by Step

start | 5 | 2 | 8 | 1 | 4 |

comparisons

# Selection Sort, Step by Step

| | | | | | | | comparisons |
|---|---|---|---|---|---|---|---|
| start | **5** | **2** | **8** | **1** | **4** | | |
| pass 1 | **1** | **2** | **8** | **5** | **4** | $1 \leftrightarrow 5$ | 4 |

# Selection Sort, Step by Step

| | | | | | | | comparisons |
|---|---|---|---|---|---|---|---|
| start | **5** | **2** | **8** | **1** | **4** | | |
| pass 1 | **1** | **2** | **8** | **5** | **4** | **1** ↔ 5 | 4 |
| pass 2 | **1** | **2** | **8** | **5** | **4** | **2** stays ✓ | 3 |

# Selection Sort, Step by Step

| | | | | | | | comparisons |
|---|---|---|---|---|---|---|---|
| start | 5 | 2 | 8 | 1 | 4 | | |
| pass 1 | 1 | 2 | 8 | 5 | 4 | 1 ↔ 5 | 4 |
| pass 2 | 1 | 2 | 8 | 5 | 4 | 2 stays ✓ | 3 |
| pass 3 | 1 | 2 | 4 | 5 | 8 | 4 ↔ 8 | 2 |

# Selection Sort, Step by Step

| | | | | | | | comparisons |
|---|---|---|---|---|---|---|---|
| start | **5** | **2** | **8** | **1** | **4** | | |
| pass 1 | **1** | **2** | **8** | **5** | **4** | **1** ↔ 5 | 4 |
| pass 2 | **1** | **2** | **8** | **5** | **4** | **2** stays ✓ | 3 |
| pass 3 | **1** | **2** | **4** | **5** | **8** | **4** ↔ 8 | 2 |
| pass 4 | **1** | **2** | **4** | **5** | **8** | **5** stays ✓ | 1 |

# Selection Sort, Step by Step

| | | | | | | | comparisons |
|---|---|---|---|---|---|---|---|
| start | **5** | 2 | 8 | 1 | 4 | | |
| pass 1 | **1** | 2 | 8 | 5 | 4 | **1** ↔ 5 | 4 |
| pass 2 | **1** | **2** | 8 | 5 | 4 | **2** stays ✓ | 3 |
| pass 3 | **1** | **2** | **4** | 5 | 8 | **4** ↔ 8 | 2 |
| pass 4 | **1** | **2** | **4** | **5** | **8** | **5** stays ✓ | 1 |

5 cards: $4 + 3 + 2 + 1 = \mathbf{10}$ comparisons

# Swap Two Items in One Line

Lecture 2: three lines

```
1 temp = x
2 x = y
3 y = temp
```

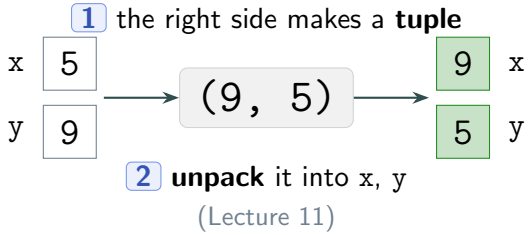
# Swap Two Items in One Line

Lecture 2: three lines

```
1 temp = x
2 x = y
3 y = temp
```

Python: one line

```
1 x, y = y, x
```



# Swap Two Items in One Line

Lecture 2: three lines

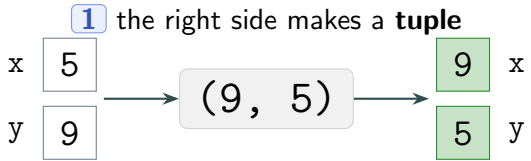
```
1 temp = x
2 x = y
3 y = temp
```

two places of a list

```
1 L = [5, 2, 8, 1, 4]
2 L[0], L[3] = L[3], L[0]
3 print(L)
```

Python: one line

```
1 x, y = y, x
```




(Lecture 11)

# Swap Two Items in One Line

Lecture 2: three lines

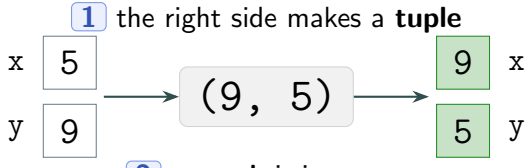
```
temp = x
x = y
y = temp
```

two places of a list

```
L = [5, 2, 8, 1, 4]
L[0], L[3] = L[3], L[0]
print(L)  # [1, 2, 8, 5, 4]
```

Python: one line

```
x, y = y, x
```




(Lecture 11)

# You Try! *(think first, 3 min)*

Sort `[5, 2, 8, 1, 4]` smallest first. **No** `sorted()`, **no** `.sort()`.

you may use: `min(L)` `L.remove(x)` `A.append(x)` `len(L)` `L[:]`

# You Try! *(think first, 3 min)*

Sort `[5, 2, 8, 1, 4]` smallest first. **No** `sorted()`, **no** `.sort()`.

you may use: `min(L)` `L.remove(x)` `A.append(x)` `len(L)` `L[:]`

**hint** the first item of the answer is `min(L)`. And the second?

# You Try! *(think first, 3 min)*

Sort `[5, 2, 8, 1, 4]` smallest first. **No** `sorted()`, **no** `.sort()`.

you may use: `min(L)` `L.remove(x)` `A.append(x)` `len(L)` `L[:]`

**hint** the first item of the answer is `min(L)`. And the second?

| rest | min | answer |
|---|---|---|
| `[5, 2, 8, 1, 4]` | 1 | `[1]` |
| `[5, 2, 8, 4]` | 2 | `[1, 2]` |
| `[5, 8, 4]` | 4 | `[1, 2, 4]` |
| `[5, 8]` | 5 | `[1, 2, 4, 5]` |
| `[8]` | 8 | `[1, 2, 4, 5, 8]` |

# You Try! *(think first, 3 min)*

Sort `[5, 2, 8, 1, 4]` smallest first. **No** `sorted()`, **no** `.sort()`.

you may use: `min(L)` `L.remove(x)` `A.append(x)` `len(L)` `L[:]`

**hint** the first item of the answer is `min(L)`. And the second?

| rest | min | answer |
|---|---|---|
| `[5, 2, 8, 1, 4]` | 1 | `[1]` |
| `[5, 2, 8, 4]` | 2 | `[1, 2]` |
| `[5, 8, 4]` | 4 | `[1, 2, 4]` |
| `[5, 8]` | 5 | `[1, 2, 4, 5]` |
| `[8]` | 8 | `[1, 2, 4, 5, 8]` |

```
def my_sort(L):
    rest = L[:]     # a copy
    answer = []
    while len(rest) > 0:
        smallest = min(rest)
        rest.remove(smallest)
        answer.append(smallest)
    return answer
```

# You Try! *(think first, 3 min)*

Sort `[5, 2, 8, 1, 4]` smallest first. **No** `sorted()`, **no** `.sort()`.

you may use: `min(L)` `L.remove(x)` `A.append(x)` `len(L)` `L[:]`

**hint** the first item of the answer is `min(L)`. And the second?

| rest | min | answer |
|---|---|---|
| `[5, 2, 8, 1, 4]` | 1 | `[1]` |
| `[5, 2, 8, 4]` | 2 | `[1, 2]` |
| `[5, 8, 4]` | 4 | `[1, 2, 4]` |
| `[5, 8]` | 5 | `[1, 2, 4, 5]` |
| `[8]` | 8 | `[1, 2, 4, 5, 8]` |

```
def my_sort(L):
    rest = L[:]      # a copy
    answer = []
    while len(rest) > 0:
        smallest = min(rest)
        rest.remove(smallest)
        answer.append(smallest)
    return answer
```

`min()` hides a loop: $4 + 3 + 2 + 1 = 10$ comparisons

# You Try! *(think first, 3 min)*

Sort `[5, 2, 8, 1, 4]` smallest first. **No** `sorted()`, **no** `.sort()`.

you may use: `min(L)` `L.remove(x)` `A.append(x)` `len(L)` `L[:]`

**hint** the first item of the answer is `min(L)`. And the second?

| rest | min | answer |
|---|---|---|
| `[5, 2, 8, 1, 4]` | 1 | `[1]` |
| `[5, 2, 8, 4]` | 2 | `[1, 2]` |
| `[5, 8, 4]` | 4 | `[1, 2, 4]` |
| `[5, 8]` | 5 | `[1, 2, 4, 5]` |
| `[8]` | 8 | `[1, 2, 4, 5, 8]` |

```
def my_sort(L):
    rest = L[:]     # a copy
    answer = []
    while len(rest) > 0:
        smallest = min(rest)
        rest.remove(smallest)
        answer.append(smallest)
    return answer
```

`min()` hides a loop: $4 + 3 + 2 + 1 = 10$ comparisons

this is **selection sort**!

# Selection Sort in Python

```
1  def selection_sort(L):
2      """Sorts L, smallest first."""
3      n = len(L)
4      for i in range(n - 1):
5          smallest = i
6          for j in range(i + 1, n):
7              if L[j] < L[smallest]:
8                  smallest = j
9          L[i], L[smallest] = L[smallest], L[i]
```

```
marks = [5, 2, 8, 1, 4]
selection_sort(marks)
print(marks)    # [1, 2, 4, 5, 8]
```

in the middle: **pass 3**



# Selection Sort in Python

```
1  def selection_sort(L):
2      """Sorts L, smallest first."""
3      n = len(L)
4      for i in range(n - 1):          1
5          smallest = i
6          for j in range(i + 1, n):
7              if L[j] < L[smallest]:
8                  smallest = j
9          L[i], L[smallest] = L[smallest], L[i]
```

```
marks = [5, 2, 8, 1, 4]
selection_sort(marks)
print(marks)    # [1, 2, 4, 5, 8]
```

in the middle: **pass 3**

1 i = 2: spot to fill

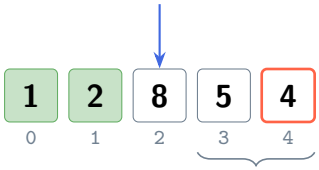


# Selection Sort in Python

```
1  def selection_sort(L):
2      """Sorts L, smallest first."""
3      n = len(L)
4      for i in range(n - 1):          1
5          smallest = i
6          for j in range(i + 1, n):   2
7              if L[j] < L[smallest]:
8                  smallest = j
9          L[i], L[smallest] = L[smallest], L[i]


marks = [5, 2, 8, 1, 4]
selection_sort(marks)
print(marks)    # [1, 2, 4, 5, 8]
```

in the middle: **pass 3**

1 `i = 2`: spot to fill

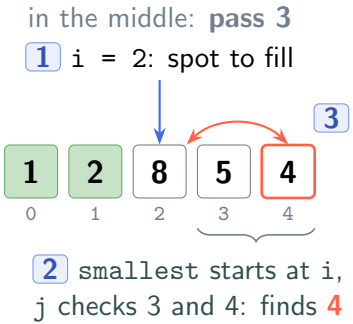



2 `smallest` starts at `i`, `j` checks 3 and 4: finds **4**

# Selection Sort in Python

```
1  def selection_sort(L):
2      """Sorts L, smallest first."""
3      n = len(L)
4      for i in range(n - 1):              1
5          smallest = i
6          for j in range(i + 1, n):       2
7              if L[j] < L[smallest]:
8                  smallest = j
9          L[i], L[smallest] = L[smallest], L[i]  3
```

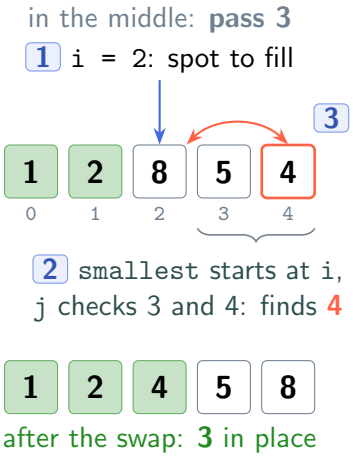
```
marks = [5, 2, 8, 1, 4]
selection_sort(marks)
print(marks)    # [1, 2, 4, 5, 8]
```



# Selection Sort in Python

```
1  def selection_sort(L):
2      """Sorts L, smallest first."""
3      n = len(L)
4      for i in range(n - 1):              1
5          smallest = i
6          for j in range(i + 1, n):       2
7              if L[j] < L[smallest]:
8                  smallest = j
9          L[i], L[smallest] = L[smallest], L[i]  3
```

```
marks = [5, 2, 8, 1, 4]
selection_sort(marks)
print(marks)    # [1, 2, 4, 5, 8]
```



# Selection Sort in Python

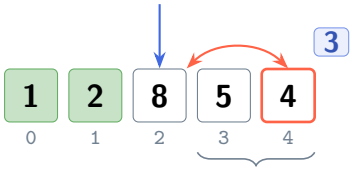
```
1  def selection_sort(L):
2      """Sorts L, smallest first."""
3      n = len(L)
4      for i in range(n - 1):                    1
5          smallest = i
6          for j in range(i + 1, n):             2
7              if L[j] < L[smallest]:
8                  smallest = j
9          L[i], L[smallest] = L[smallest], L[i] 3
```

```
marks = [5, 2, 8, 1, 4]
selection_sort(marks)
print(marks)    # [1, 2, 4, 5, 8]
```

changes L, returns None (like `L.sort()`)

in the middle: **pass 3**

1 `i = 2`: spot to fill




2 `smallest` starts at `i`, `j` checks 3 and 4: finds **4**




after the swap: **3** in place

# **You Try!** *(write each line, 2 min)*

`L = [6, 3, 5, 1, 2]`. Write `L` after each pass of `selection_sort`.

after pass 1:

after pass 2:

after pass 3:

after pass 4:

How many comparisons in all?

# **You Try!** *(write each line, 2 min)*

`L = [6, 3, 5, 1, 2]`. Write `L` after each pass of `selection_sort`.

after pass 1: `[1, 3, 5, 6, 2]`

after pass 2:

after pass 3:

after pass 4:

How many comparisons in all?

# **You Try!** *(write each line, 2 min)*

`L = [6, 3, 5, 1, 2]`. Write `L` after each pass of `selection_sort`.

after pass 1: `[1, 3, 5, 6, 2]`

after pass 2: `[1, 2, 5, 6, 3]`

after pass 3:

after pass 4:

How many comparisons in all?

# **You Try!** *(write each line, 2 min)*

`L = [6, 3, 5, 1, 2]`. Write `L` after each pass of `selection_sort`.

after pass 1: `[1, 3, 5, 6, 2]`

after pass 2: `[1, 2, 5, 6, 3]`

after pass 3: `[1, 2, 3, 6, 5]`

after pass 4:

How many comparisons in all?

# **You Try!** *(write each line, 2 min)*

`L = [6, 3, 5, 1, 2]`. Write `L` after each pass of `selection_sort`.

after pass 1: `[1, 3, 5, 6, 2]`

after pass 2: `[1, 2, 5, 6, 3]`

after pass 3: `[1, 2, 3, 6, 5]`

after pass 4: `[1, 2, 3, 5, 6]`

How many comparisons in all?

# **You Try!** *(write each line, 2 min)*

`L = [6, 3, 5, 1, 2]`. Write `L` after each pass of `selection_sort`.

after pass 1: `[1, 3, 5, 6, 2]`

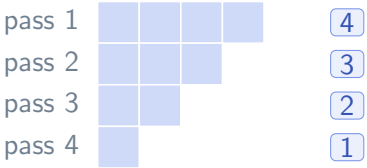
after pass 2: `[1, 2, 5, 6, 3]`

after pass 3: `[1, 2, 3, 6, 5]`

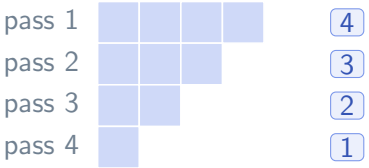
after pass 4: `[1, 2, 3, 5, 6]`

How many comparisons in all? $4 + 3 + 2 + 1 = \mathbf{10}$ same as before

# Counting the Comparisons

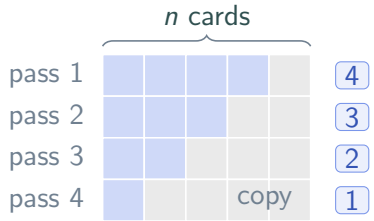


# Counting the Comparisons




$$(n-1)+(n-2)+\ldots+1$$

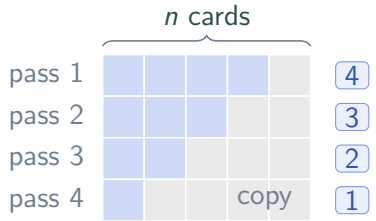
# Counting the Comparisons




$$(n-1)+(n-2)+\ldots+1$$

$$= \text{half of } n \times (n-1) = \frac{n(n-1)}{2}$$

# Counting the Comparisons



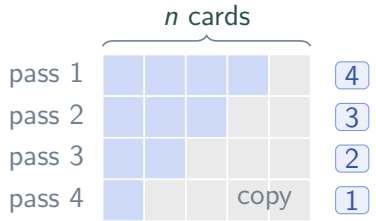

$$(n-1)+(n-2)+\ldots+1$$

$$= \text{half of } n \times (n-1) = \frac{n(n-1)}{2}$$

the **handshakes** of Lecture 18

$\rightarrow$ $O(n^2)$

# Counting the Comparisons




$$(n-1)+(n-2)+\ldots+1$$

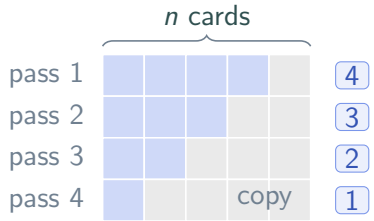
$$= \text{half of } n \times (n-1) = \frac{n(n-1)}{2}$$

the **handshakes** of Lecture 18

$\rightarrow$ $O(n^2)$

| $n$ | **comparisons** |
|---|---|
| 10 | 45 |
| 1,000 | 499,500 |
| 1,000,000 | **500 billion** |

# Counting the Comparisons




$$(n-1) + (n-2) + \ldots + 1$$

$$= \text{half of } n \times (n-1) = \frac{n(n-1)}{2}$$

the **handshakes** of Lecture 18

$\rightarrow$ $O(n^2)$

| $n$ | **comparisons** |
|---|---|
| 10 | 45 |
| 1,000 | 499,500 |
| 1,000,000 | **500 billion** |

**1** **2** **3** **4** **5** already sorted? still **10** comparisons: it never notices

# Bug Hunt! *(write the line number and the fix, 30 s)*

A friend typed selection sort from memory:

```
1  def selection_sort(L):
2      n = len(L)
3      for i in range(n - 1):
4          smallest = 0
5          for j in range(i + 1, n):
6              if L[j] < L[smallest]:
7                  smallest = j
8          L[i], L[smallest] = L[smallest], L[i]
9
10 marks = [5, 2, 8, 1, 4]
11 selection_sort(marks)
12 print(marks)
```

## Symptom:

`[8, 1, 2, 4, 5]`

The 8 jumped to the **front**!

# Bug Hunt! *(write the line number and the fix, 30 s)*

A friend typed selection sort from memory:

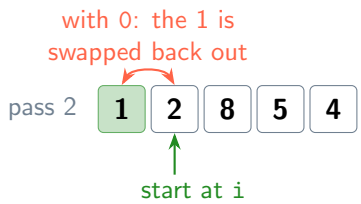
```
1  def selection_sort(L):
2      n = len(L)
3      for i in range(n - 1):
4          smallest = 0
5          for j in range(i + 1, n):
6              if L[j] < L[smallest]:
7                  smallest = j
8          L[i], L[smallest] = L[smallest], L[i]
9
10 marks = [5, 2, 8, 1, 4]
11 selection_sort(marks)
12 print(marks)
```

## Symptom:

`[8, 1, 2, 4, 5]`

The 8 jumped to the **front**!

**Line 4** `smallest = i`




Corner case: `[2, 1]` comes out right. Test with 3 or more cards.

# Crossing Out: The Sieve

# Which Numbers Are Prime?



$12 = 3 \times 4$

a rectangle: **not** prime

# Which Numbers Are Prime?



$12 = 3 \times 4$

a rectangle: **not** prime



13 only one row: **prime**

# Which Numbers Are Prime?



$12 = 3 \times 4$

a rectangle: **not** prime



13 only one row: **prime**

2, 3, 5, 7, 11, 13, 17, 19, 23, ...

# Which Numbers Are Prime?



$12 = 3 \times 4$

a rectangle: **not** prime



13 only one row: **prime**

2, 3, 5, 7, 11, 13, 17, 19, 23, ...

Lab 05's way: try every divisor.

```
1  def is_prime(n):
2      for d in range(2, n):
3          if n % d == 0:
4              return False
5      return True
```

# Which Numbers Are Prime?



$12 = 3 \times 4$

a rectangle: **not** prime



13 only one row: **prime**

2, 3, 5, 7, 11, 13, 17, 19, 23, ...

Lab 05's way: try every divisor.

```
1  def is_prime(n):
2      for d in range(2, n):
3          if n % d == 0:
4              return False
5      return True
```

`is_prime`: a prime $p$ costs $p - 2$ divisions

# Listing the Primes: The Slow Way

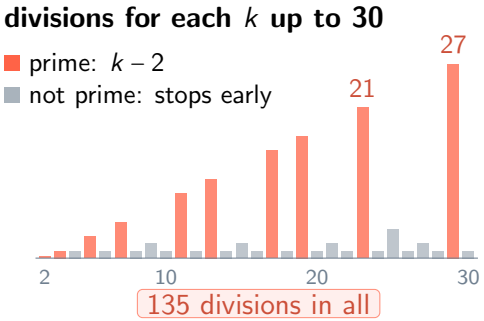
```
1 def primes_up_to(n):
2     primes = []
3     for k in range(2, n + 1):
4         if is_prime(k):
5             primes.append(k)
6     return primes
```

# Listing the Primes: The Slow Way

```
1  def primes_up_to(n):
2      primes = []
3      for k in range(2, n + 1):      1
4          if is_prime(k):            2
5              primes.append(k)
6      return primes
```

# Listing the Primes: The Slow Way

```
1  def primes_up_to(n):
2      primes = []
3      for k in range(2, n + 1):        1
4          if is_prime(k):              2
5              primes.append(k)
6      return primes
```

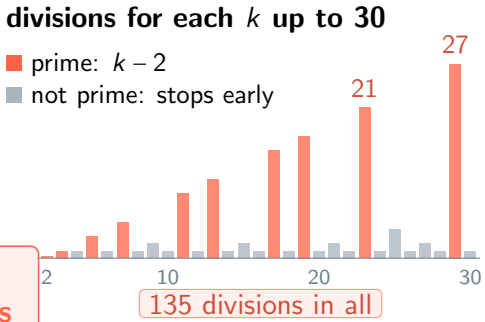


# Listing the Primes: The Slow Way

```
1  def primes_up_to(n):
2      primes = []
3      for k in range(2, n + 1):      1
4          if is_prime(k):            2
5              primes.append(k)
6      return primes
```

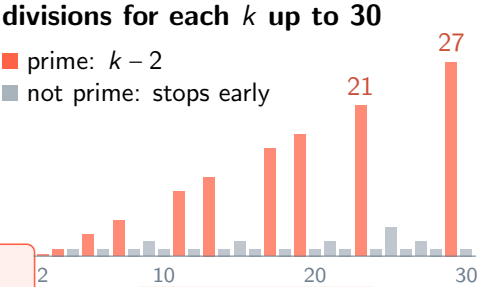


up to 100,000: **8.5 seconds**
up to 1,000,000: **about 12 minutes**



# Listing the Primes: The Slow Way

```
1  def primes_up_to(n):
2      primes = []
3      for k in range(2, n + 1):      1
4          if is_prime(k):            2
5              primes.append(k)
6      return primes
```






up to 100,000: **8.5 seconds**
up to 1,000,000: **about 12 minutes**

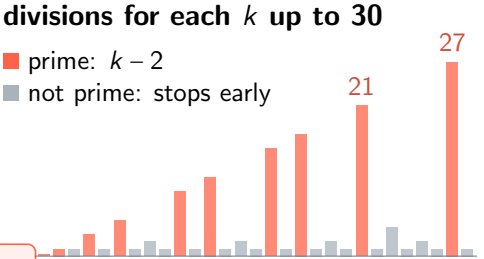
**1 primes cost most**:
29 takes 27 divisions, all "no"

**2 loop 1 in a loop 2**:
work grows almost like $n^2$

**3 no memory**: each number starts from scratch

# Listing the Primes: The Slow Way

```
1  def primes_up_to(n):
2      primes = []
3      for k in range(2, n + 1):      1
4          if is_prime(k):            2
5              primes.append(k)
6      return primes
```






up to 100,000: **8.5 seconds**
up to 1,000,000: **about 12 minutes**

**1 primes cost most**:
29 takes 27 divisions, all "no"

**2 loop 1 in a loop 2**:
work grows almost like $n^2$

**3 no memory**: each number starts from scratch

Can we find the primes **without dividing**?

# The Sieve of Eratosthenes (Greece, about 240 BC)

| | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|
| 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 |
| 11 | 12 | 13 | 14 | 15 | 16 | 17 | 18 | 19 | 20 |
| 21 | 22 | 23 | 24 | 25 | 26 | 27 | 28 | 29 | 30 |
| 31 | 32 | 33 | 34 | 35 | 36 | 37 | 38 | 39 | 40 |
| 41 | 42 | 43 | 44 | 45 | 46 | 47 | 48 | 49 | 50 |

Write the numbers.

1 is not prime.

# The Sieve of Eratosthenes (Greece, about 240 BC)




**2** is prime.

Cross out 4, 6, 8, . . .

# The Sieve of Eratosthenes (Greece, about 240 BC)




**3** is not crossed: prime.
Cross out 6, 9, 12, . . .

# The Sieve of Eratosthenes (Greece, about 240 BC)




**4** is crossed: skip it.

# The Sieve of Eratosthenes (Greece, about 240 BC)




**5** is not crossed: prime.
Cross out 10, 15, 20, ...

# The Sieve of Eratosthenes (Greece, about 240 BC)




**7** is not crossed: prime.
Cross out 14, 21, 28, ...

# The Sieve of Eratosthenes (Greece, about 240 BC)

| | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|
| 1 | 2 | 3 | ~~4~~ | 5 | ~~6~~ | 7 | ~~8~~ | ~~9~~ | ~~10~~ |
| 11 | ~~12~~ | 13 | ~~14~~ | ~~15~~ | ~~16~~ | 17 | ~~18~~ | 19 | ~~20~~ |
| ~~21~~ | ~~22~~ | 23 | ~~24~~ | ~~25~~ | ~~26~~ | ~~27~~ | ~~28~~ | 29 | ~~30~~ |
| 31 | ~~32~~ | ~~33~~ | ~~34~~ | ~~35~~ | ~~36~~ | 37 | ~~38~~ | ~~39~~ | ~~40~~ |
| 41 | ~~42~~ | 43 | ~~44~~ | ~~45~~ | ~~46~~ | 47 | ~~48~~ | ~~49~~ | ~~50~~ |

11, 13, . . . : nothing new.
Every number left is **prime**.

15 primes up to 50

# The Sieve of Eratosthenes (Greece, about 240 BC)

| | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|
| 1 | 2 | 3 | ~~4~~ | 5 | ~~6~~ | 7 | ~~8~~ | ~~9~~ | ~~10~~ |
| 11 | ~~12~~ | 13 | ~~14~~ | ~~15~~ | ~~16~~ | 17 | ~~18~~ | 19 | ~~20~~ |
| ~~21~~ | ~~22~~ | 23 | ~~24~~ | ~~25~~ | ~~26~~ | ~~27~~ | ~~28~~ | 29 | ~~30~~ |
| 31 | ~~32~~ | ~~33~~ | ~~34~~ | ~~35~~ | ~~36~~ | 37 | ~~38~~ | ~~39~~ | ~~40~~ |
| 41 | ~~42~~ | 43 | ~~44~~ | ~~45~~ | ~~46~~ | 47 | ~~48~~ | ~~49~~ | ~~50~~ |

11, 13, . . . : nothing new.
Every number left is **prime**.

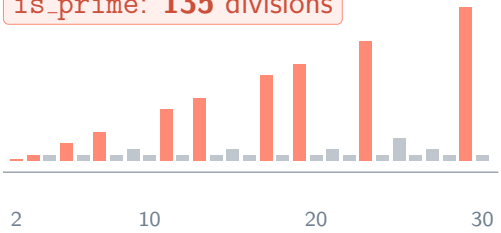
15 primes up to 50

no dividing: only count up in steps of $p$

# Why Is the Sieve So Fast?

**the same numbers, 2 to 30**

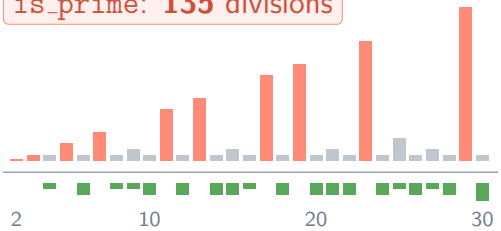
is_prime: **135** divisions



# Why Is the Sieve So Fast?

**the same numbers, 2 to 30**

`is_prime`: **135** divisions



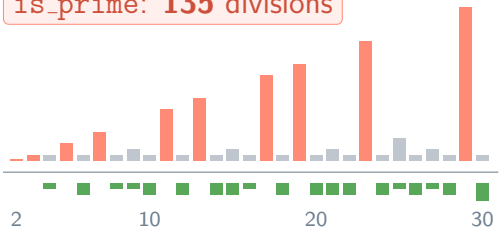

sieve: **33** crossings

primes: **0** work (never crossed)

# Why Is the Sieve So Fast?

**the same numbers, 2 to 30**

is_prime: **135** divisions




sieve: **33** crossings

primes: **0** work (never crossed)

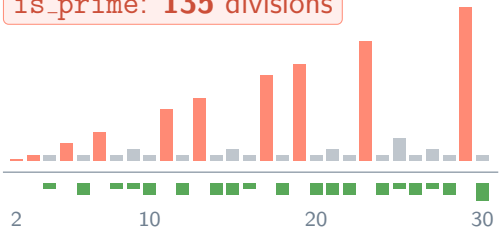
**slow** asks **every number**:
"who divides you?"
a prime says "no" $k-2$ times

**fast** tells **each multiple**:
"you are out" (jump $+p$)
no division, no wasted "no"

# Why Is the Sieve So Fast?

**the same numbers, 2 to 30**

is_prime: **135** divisions




sieve: **33** crossings

primes: **0** work (never crossed)

**slow** asks **every number**:
"who divides you?"
a prime says "no" $k-2$ times

**fast** tells **each multiple**:
"you are out" (jump $+p$)
no division, no wasted "no"

up to 1,000,000: **about 12 minutes** vs **0.05 seconds**

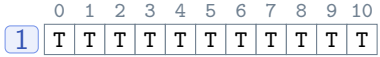
# The Sieve in Python

```
1  def sieve(n):
2      """List: True at k when k is prime."""
3      is_prime = [True] * (n + 1)
4      is_prime[0] = False
5      is_prime[1] = False
6      for p in range(2, n + 1):
7          if is_prime[p]:
8              for m in range(2 * p, n + 1, p):
9                  is_prime[m] = False
10     return is_prime
```

0 1 2 3 4 5 6 7 8 9 10

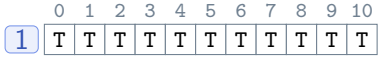
# The Sieve in Python

```
1  def sieve(n):
2      """List: True at k when k is prime."""
3      is_prime = [True] * (n + 1)   1
4      is_prime[0] = False
5      is_prime[1] = False
6      for p in range(2, n + 1):
7          if is_prime[p]:
8              for m in range(2 * p, n + 1, p):
9                  is_prime[m] = False
10     return is_prime
```



# The Sieve in Python

```
1  def sieve(n):
2      """List: True at k when k is prime."""
3      is_prime = [True] * (n + 1)   1
4      is_prime[0] = False
5      is_prime[1] = False
6      for p in range(2, n + 1):     2
7          if is_prime[p]:
8              for m in range(2 * p, n + 1, p):
9                  is_prime[m] = False
10     return is_prime
```

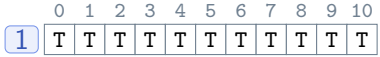



start: all True

2 p walks 2, 3, 4, . . .

# The Sieve in Python

```
1  def sieve(n):
2      """List: True at k when k is prime."""
3      is_prime = [True] * (n + 1)   1
4      is_prime[0] = False
5      is_prime[1] = False
6      for p in range(2, n + 1):     2
7          if is_prime[p]:           3
8              for m in range(2 * p, n + 1, p):
9                  is_prime[m] = False
10     return is_prime
```



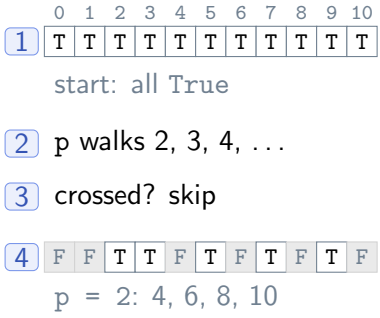

start: all True

2 p walks 2, 3, 4, . . .

3 crossed? skip

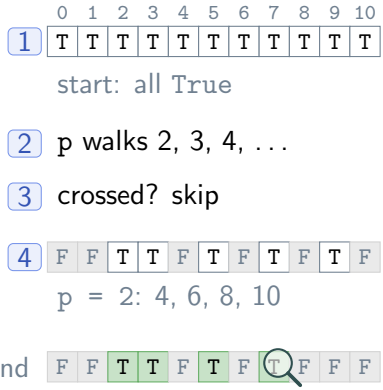
# The Sieve in Python

```
1  def sieve(n):
2      """List: True at k when k is prime."""
3      is_prime = [True] * (n + 1)   1
4      is_prime[0] = False
5      is_prime[1] = False
6      for p in range(2, n + 1):     2
7          if is_prime[p]:           3
8              for m in range(2 * p, n + 1, p):
9                  is_prime[m] = False   4
10     return is_prime
```



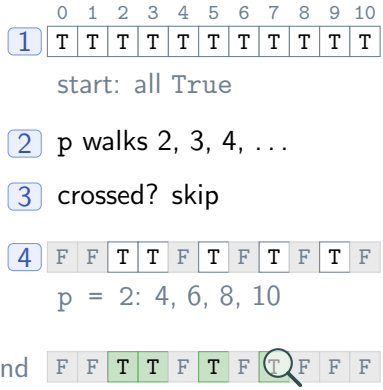
# The Sieve in Python

```
1  def sieve(n):
2      """List: True at k when k is prime."""
3      is_prime = [True] * (n + 1)   1
4      is_prime[0] = False
5      is_prime[1] = False
6      for p in range(2, n + 1):     2
7          if is_prime[p]:           3
8              for m in range(2 * p, n + 1, p):
9                  is_prime[m] = False   4
10     return is_prime

flags = sieve(10)
print(flags[7], flags[9])    # True False
```



# The Sieve in Python

```
1  def sieve(n):
2      """List: True at k when k is prime."""
3      is_prime = [True] * (n + 1)   1
4      is_prime[0] = False
5      is_prime[1] = False
6      for p in range(2, n + 1):     2
7          if is_prime[p]:           3
8              for m in range(2 * p, n + 1, p):
9                  is_prime[m] = False   4
10     return is_prime

flags = sieve(10)
print(flags[7], flags[9])    # True False
```




flags[7]: **one look**, no dividing

**You Try!** *(write your answers, 2 min)*

**(a)** In `sieve(20)`, which numbers does `p = 3` cross?

**(b)** What does the loop do when `p = 4`?

**(c)** In `sieve(50)`, `p = 7` crosses 14, 21, 28, 35, 42, 49. Which one was **not** crossed before?

**You Try!** *(write your answers, 2 min)*

**(a)** In `sieve(20)`, which numbers does `p = 3` cross?
6, 9, 12, 15, 18

**(b)** What does the loop do when `p = 4`?

**(c)** In `sieve(50)`, `p = 7` crosses 14, 21, 28, 35, 42, 49.
Which one was **not** crossed before?

**You Try!** *(write your answers, 2 min)*

**(a)** In `sieve(20)`, which numbers does `p = 3` cross?
6, 9, 12, 15, 18

**(b)** What does the loop do when `p = 4`?
Nothing. 4 is already crossed (by 2), so `3` skips it.

**(c)** In `sieve(50)`, `p = 7` crosses 14, 21, 28, 35, 42, 49.
Which one was **not** crossed before?

**You Try!** *(write your answers, 2 min)*

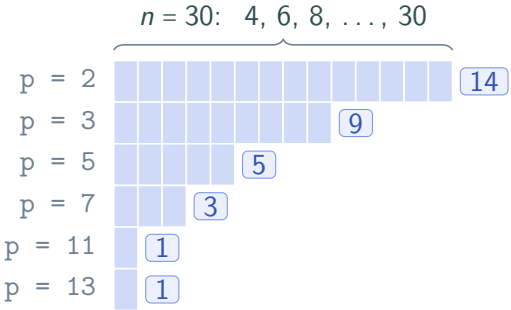
**(a)** In `sieve(20)`, which numbers does `p = 3` cross?
6, 9, 12, 15, 18

**(b)** What does the loop do when `p = 4`?
Nothing. 4 is already crossed (by 2), so **3** skips it.

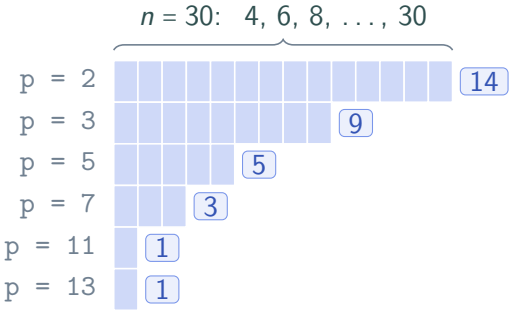
**(c)** In `sieve(50)`, `p = 7` crosses 14, 21, 28, 35, 42, 49.
Which one was **not** crossed before?

$49 = 7 \times 7$ the others have a smaller factor: 2, 3 or 5

# Counting the Crossings

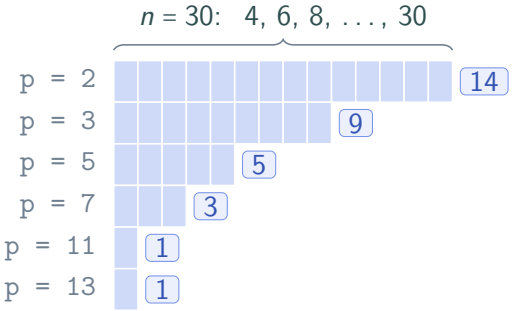


# Counting the Crossings




$$\frac{n}{2} + \frac{n}{3} + \frac{n}{5} + \frac{n}{7} + \ldots$$

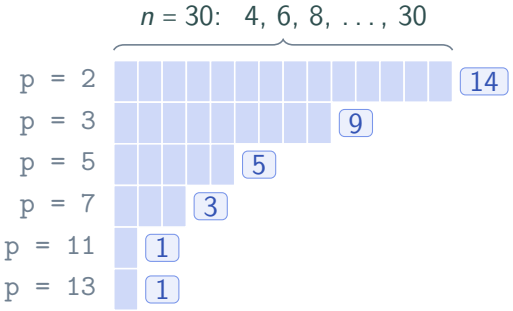
# Counting the Crossings




$$\frac{n}{2} + \frac{n}{3} + \frac{n}{5} + \frac{n}{7} + \ldots$$

| $n$ | is_prime | **sieve** |
|---|---|---|
| | divisions | crossings |
| 1,000 | 78,022 | 1,958 |
| 10,000 | 5,775,223 | 23,071 |
| 1,000,000 | 37.6 billion | **2.8 million** |

# Counting the Crossings

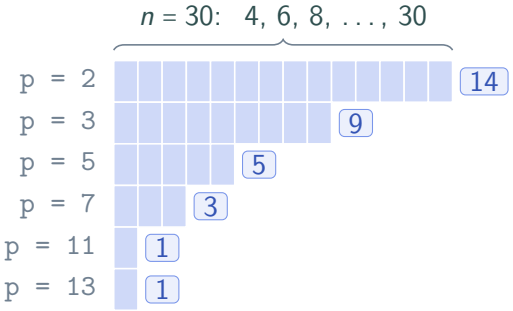



$$\frac{n}{2} + \frac{n}{3} + \frac{n}{5} + \frac{n}{7} + \ldots$$

| $n$ | is_prime | **sieve** |
|---|---|---|
| | divisions | crossings |
| 1,000 | 78,022 | 1,958 |
| 10,000 | 5,775,223 | 23,071 |
| 1,000,000 | 37.6 billion | **2.8 million** |

crossings $< 3n \rightarrow$ almost $O(n)$

# Counting the Crossings




$$\frac{n}{2} + \frac{n}{3} + \frac{n}{5} + \frac{n}{7} + \ldots$$

| $n$ | is_prime | **sieve** |
|---|---|---|
| | divisions | crossings |
| 1,000 | 78,022 | 1,958 |
| 10,000 | 5,775,223 | 23,071 |
| 1,000,000 | 37.6 billion | **2.8 million** |

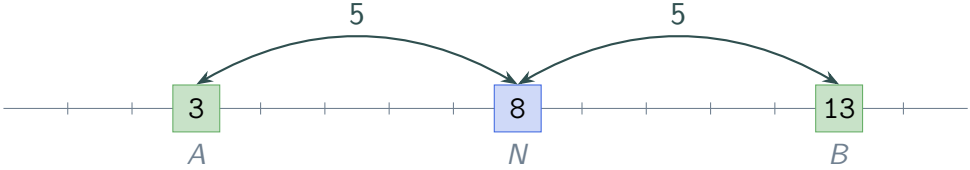
crossings $< 3n \rightarrow$ almost $O(n)$

selection sort's rows: shrink by **1** $O(n^2)$

the sieve's rows: $n/p$, shrink fast

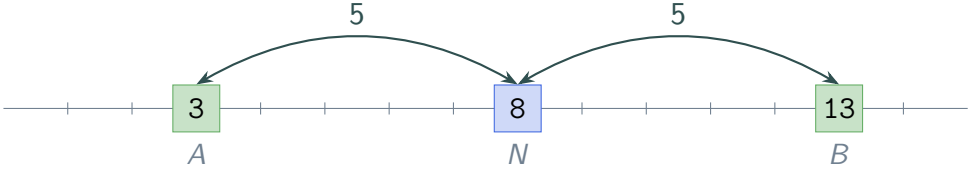
# Pretty Average Primes (a Canadian contest: CCC 2019, Senior #2)

Given $N$, find two primes $A$ and $B$ with $N$ exactly in the **middle**.



# Pretty Average Primes (a Canadian contest: CCC 2019, Senior #2)

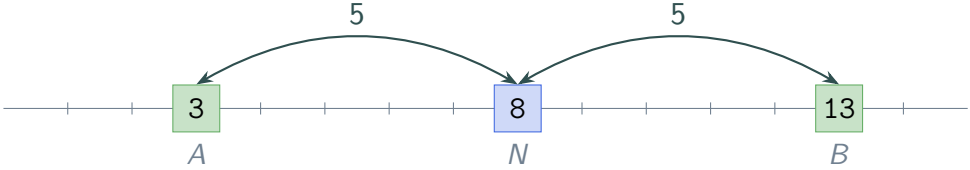
Given $N$, find two primes $A$ and $B$ with $N$ exactly in the **middle**.




$$N = \frac{A+B}{2} \quad \text{means} \quad A + B = 2N$$

# Pretty Average Primes (a Canadian contest: CCC 2019, Senior #2)

Given $N$, find two primes $A$ and $B$ with $N$ exactly in the **middle**.



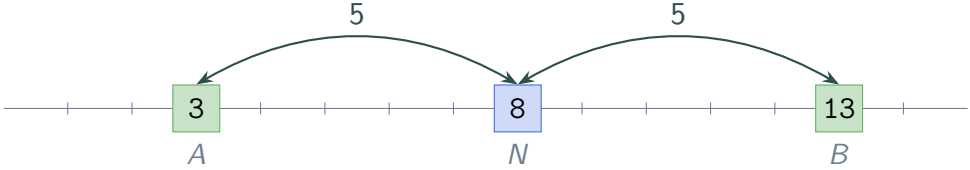

| input | one answer |
|---|---|
| 8 | 3 13 |
| 4 | 3 5 |
| 21 | 5 37 |

any right pair is accepted

$$N = \frac{A+B}{2} \quad \text{means} \quad A + B = 2N$$

# Pretty Average Primes (a Canadian contest: CCC 2019, Senior #2)

Given $N$, find two primes $A$ and $B$ with $N$ exactly in the **middle**.




| input | one answer |
|---|---|
| 8 | 3 13 |
| 4 | 3 5 |
| 21 | 5 37 |

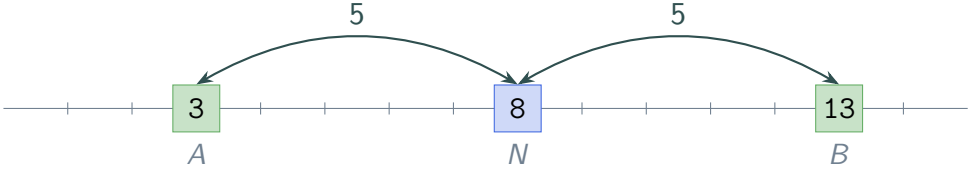
any right pair is accepted

$$N = \frac{A+B}{2} \quad \text{means} \quad A + B = 2N$$

many $N$, each up to **1,000,000**
$\rightarrow$ $B$ can be near **2,000,000**

# Pretty Average Primes (a Canadian contest: CCC 2019, Senior #2)

Given $N$, find two primes $A$ and $B$ with $N$ exactly in the **middle**.




| input | one answer |
|---|---|
| 8 | 3 13 |
| 4 | 3 5 |
| 21 | 5 37 |

any right pair is accepted

$$N = \frac{A+B}{2} \quad \text{means} \quad A + B = 2N$$

many $N$, each up to **1,000,000**
$\rightarrow$ $B$ can be near **2,000,000**

**Goldbach** (1742): every even number > 2 is a sum of two primes. **Nobody has proved it.**

# Sieve Once, Then Look

```
flags = sieve(2000000)

T = int(input())
for t in range(T):
    N = int(input())
    for A in range(2, N + 1):
        B = 2 * N - A
        if flags[A] and flags[B]:
            print(A, B)
            break
```

# Sieve Once, Then Look

```
1  flags = sieve(2000000)                    1
2
3  T = int(input())
4  for t in range(T):
5      N = int(input())
6      for A in range(2, N + 1):
7          B = 2 * N - A
8          if flags[A] and flags[B]:
9              print(A, B)
10             break
```

1 **once**, before any $N$

# Sieve Once, Then Look

```
1  flags = sieve(2000000)                    1
2
3  T = int(input())
4  for t in range(T):
5      N = int(input())
6      for A in range(2, N + 1):             2
7          B = 2 * N - A
8          if flags[A] and flags[B]:
9              print(A, B)
10             break
```

1. **once**, before any $N$
2. $N = 10$: try $A = 2, 3, \ldots$

`A=2, B=18` ✗ 18 crossed

# Sieve Once, Then Look

```
1  flags = sieve(2000000)                    1
2
3  T = int(input())
4  for t in range(T):
5      N = int(input())
6      for A in range(2, N + 1):             2
7          B = 2 * N - A
8          if flags[A] and flags[B]:         3
9              print(A, B)
10             break
```

1. **once**, before any $N$
2. $N = 10$: try $A = 2, 3, \ldots$

   `A=2, B=18` ✗ 18 crossed
3. `A=3, B=17` ✓

print 3 17, stop

# Sieve Once, Then Look

```
1  flags = sieve(2000000)                    1
2
3  T = int(input())
4  for t in range(T):
5      N = int(input())
6      for A in range(2, N + 1):             2
7          B = 2 * N - A
8          if flags[A] and flags[B]:         3
9              print(A, B)
10             break
```

1. **once**, before any $N$
2. $N = 10$: try $A = 2, 3, \ldots$
   - `A=2, B=18` ✗ 18 crossed
3. `A=3, B=17` ✓

print 3 17, stop

**1,000 questions, $N$ up to 1,000,000 (one laptop)**

| | |
|---|---|
| is_prime | about 18 seconds |
| sieve | 0.1 seconds |

**You Try!** *(trace the program, 2 min)*

**(a)** What does it print for `N = 12`?

**(b)** What does it print for `N = 15`?

**(c)** Why is `A = 2` never the answer when $N > 2$?

**You Try!** *(trace the program, 2 min)*

**(a)** What does it print for `N = 12`?

5 19 (`A = 2`: 22 ✗, `A = 3`: 21 ✗, `A = 4` ✗, `A = 5`: 19 ✓)

**(b)** What does it print for `N = 15`?

**(c)** Why is `A = 2` never the answer when $N > 2$?

**You Try!** *(trace the program, 2 min)*

**(a)** What does it print for `N = 12`?

5 19 (`A = 2`: 22 ✗, `A = 3`: 21 ✗, `A = 4` ✗, `A = 5`: 19 ✓)

**(b)** What does it print for `N = 15`?

7 23 (25, 27 and 28 are crossed)

**(c)** Why is `A = 2` never the answer when $N > 2$?

**You Try!** *(trace the program, 2 min)*

**(a)** What does it print for `N = 12`?

5 19 (`A = 2`: 22 ✗, `A = 3`: 21 ✗, `A = 4` ✗, `A = 5`: 19 ✓)

**(b)** What does it print for `N = 15`?

7 23 (25, 27 and 28 are crossed)

**(c)** Why is `A = 2` never the answer when $N > 2$?

$B = 2N - 2$ is **even** and bigger than 2: not prime.

**You Try!** *(trace the program, 2 min)*

**(a)** What does it print for `N = 12`?

5 19 (`A = 2`: 22 ✗, `A = 3`: 21 ✗, `A = 4` ✗, `A = 5`: 19 ✓)

**(b)** What does it print for `N = 15`?

7 23 (25, 27 and 28 are crossed)

**(c)** Why is `A = 2` never the answer when $N > 2$?

$B = 2N - 2$ is **even** and bigger than 2: not prime.

for every $N$ up to 1,000,000, $A$ is at most 601: the inner loop is short

# The Speed Limit

# How Many Orders?

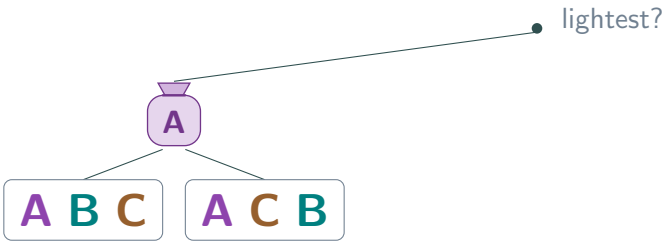
How fast can **any** sort be? Back to the three bags.

- lightest?

**A** **B** **C**:
**A** lightest
**C** heaviest

# How Many Orders?

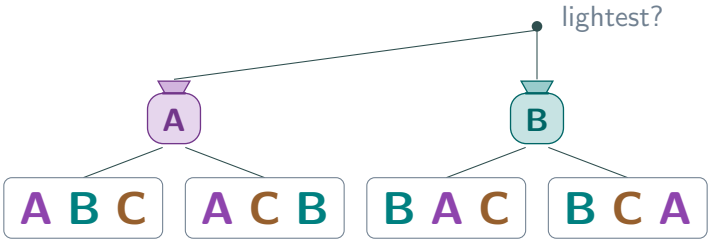
How fast can **any** sort be? Back to the three bags.




**A B C**:
**A** lightest
**C** heaviest

# How Many Orders?

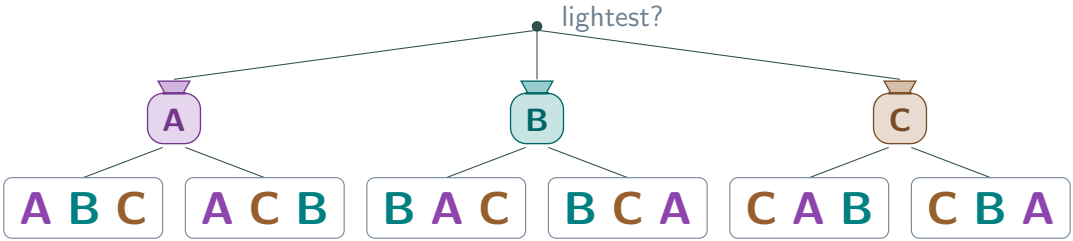
How fast can **any** sort be? Back to the three bags.




**A B C**:
**A** lightest
**C** heaviest

# How Many Orders?

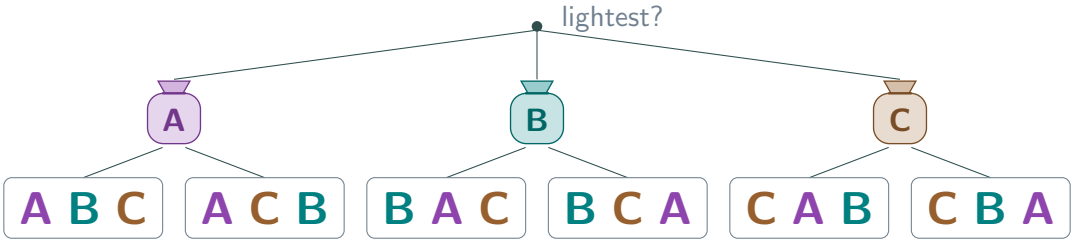
How fast can **any** sort be? Back to the three bags.




**A B C**:
**A** lightest
**C** heaviest

# How Many Orders?

How fast can **any** sort be? Back to the three bags.

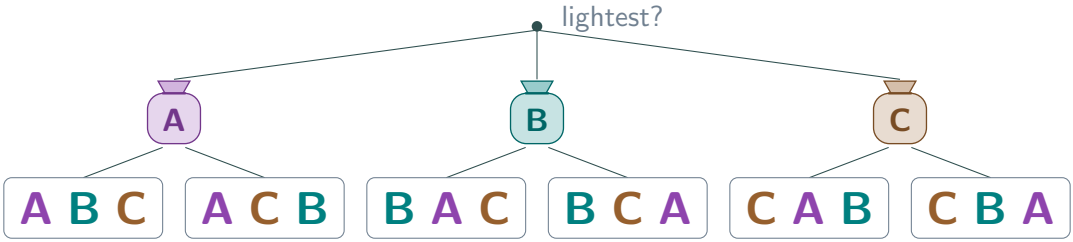



$$\underbrace{3}_{\text{lightest}} \times \underbrace{2}_{\text{middle}} \times \underbrace{1}_{\text{heaviest}} = 3! = 6 \text{ orders}$$

**A B C**:
**A** lightest
**C** heaviest

# How Many Orders?

How fast can **any** sort be? Back to the three bags.




$$\underbrace{3}_{\text{lightest}} \times \underbrace{2}_{\text{middle}} \times \underbrace{1}_{\text{heaviest}} = 3! = 6 \text{ orders}$$

**A B C**:
**A** lightest
**C** heaviest

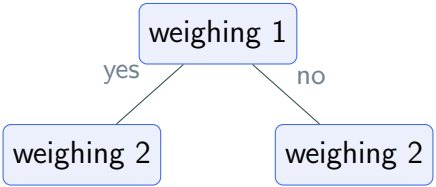
A correct plan must be able to end in **each** of the 6.

# How Many Answer Patterns?

weighing 1

each weighing: **yes** or **no**

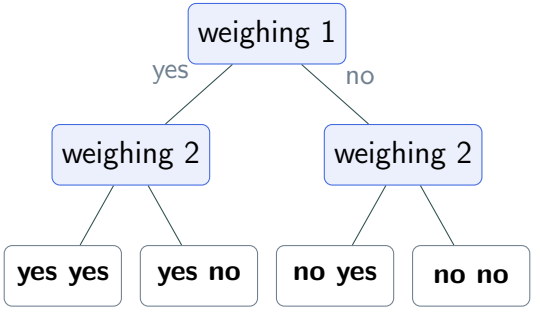
# How Many Answer Patterns?




each weighing: **yes** or **no**

1 weighing: **2** patterns

# How Many Answer Patterns?

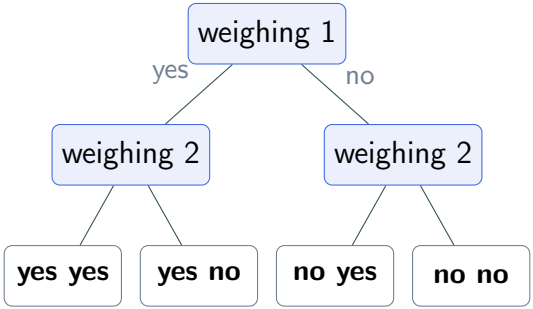



each weighing: **yes** or **no**

1 weighing: **2** patterns

2 weighings: $2 \times 2 =$ **4**

# How Many Answer Patterns?



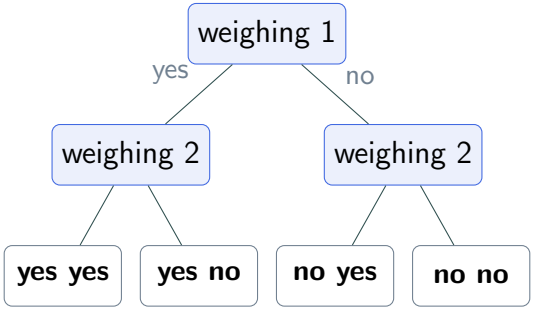

the 2nd weighing may depend on the 1st answer: still **4** patterns

each weighing: **yes** or **no**

1 weighing: **2** patterns

2 weighings: $2 \times 2 = \mathbf{4}$

# How Many Answer Patterns?




the 2nd weighing may depend on the 1st answer: still **4** patterns

each weighing: **yes** or **no**

1 weighing: **2** patterns

2 weighings: $2 \times 2 = \mathbf{4}$

*k* weighings: $\mathbf{2^k}$ patterns

# 6 Orders, 4 Patterns

The plan from the start: 1st: **A** < **B**? 2nd: **B** < **C**?

| yes yes | yes no | no yes | no no |
|---|---|---|---|
| | | | |

# 6 Orders, 4 Patterns

The plan from the start: 1st: A < B? 2nd: B < C?

| yes yes | yes no | no yes | no no |
|---|---|---|---|
| A B C | A C B<br>C A B | B A C<br>B C A | C B A |
| ✓ | | | ✓ |

# 6 Orders, 4 Patterns

The plan from the start: 1st: A < B? 2nd: B < C?

| yes yes | yes no | no yes | no no |
|---|---|---|---|
| A B C | A C B<br>C A B | B A C<br>B C A | C B A |

✓ same answers → same line → **wrong** for one of the two ✓

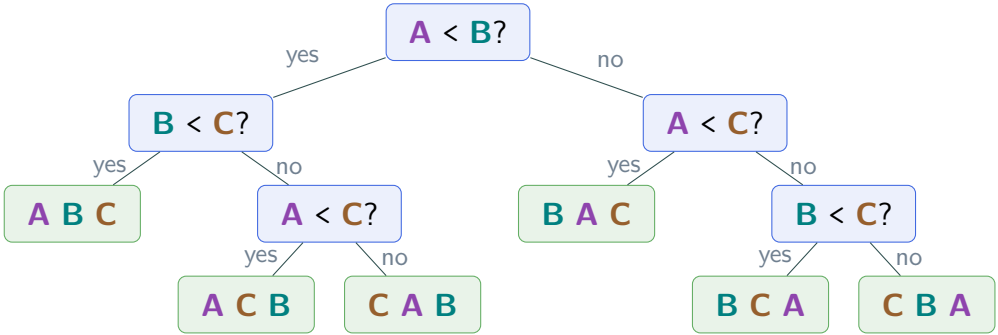
# 6 Orders, 4 Patterns

The plan from the start: 1st: A < B? 2nd: B < C?

| yes yes | yes no | no yes | no no |
|---|---|---|---|
| A B C | A C B<br>C A B | B A C<br>B C A | C B A |
| ✓ | same answers → same line → **wrong** for one of the two | | ✓ |

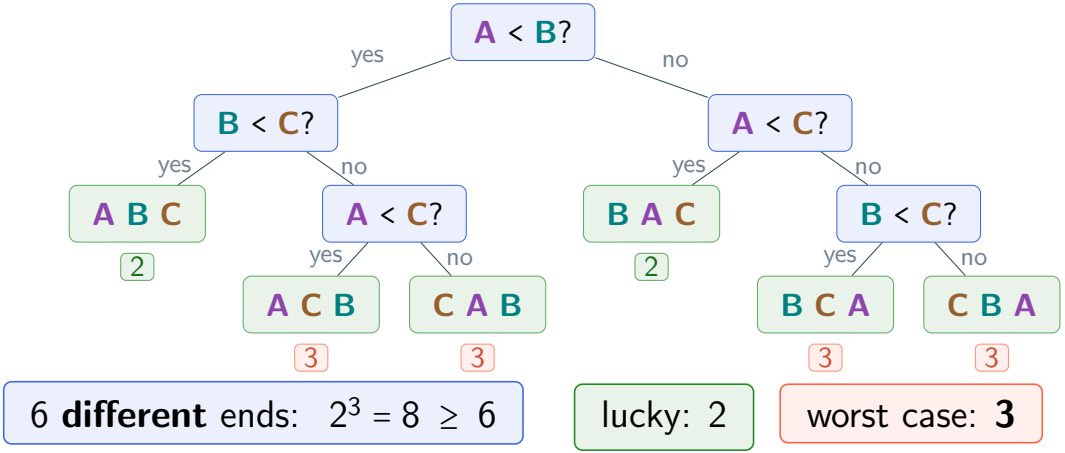
**any** plan: 6 orders, only 4 boxes, so some box gets 2
**NO!** 2 weighings are not always enough

# 3 Weighings Are Enough




6 **different** ends: $2^3 = 8 \geq 6$

# 3 Weighings Are Enough



# The Rule for $n$ Items

$$\underbrace{2^k}_{\text{patterns of } k \text{ comparisons}} \geq \underbrace{n!}_{\text{orders of } n \text{ items}}$$

# The Rule for $n$ Items

$$\underbrace{2^k}_{\text{patterns of } k \text{ comparisons}} \geq \underbrace{n!}_{\text{orders of } n \text{ items}}$$

each order needs its **own** pattern

# The Rule for $n$ Items

$$\underbrace{2^k}_{\text{patterns of } k \text{ comparisons}} \geq \underbrace{n!}_{\text{orders of } n \text{ items}}$$

each order needs its **own** pattern

| $n$ | orders $n!$ | fewest possible | selection sort |
|---|---|---|---|
| 3 | 6 | 3 | 3 |
| 4 | 24 | 5 | 6 |
| 10 | 3,628,800 | 22 | 45 |
| 1,000 | a 2,568-digit number | **8,530** | 499,500 |
| 1,000,000 | far bigger | **18,488,885** | 500 billion |

comparisons in the worst case

# The Rule for $n$ Items

$$\underbrace{2^k}_{\text{patterns of } k \text{ comparisons}} \geq \underbrace{n!}_{\text{orders of } n \text{ items}}$$

each order needs its **own** pattern

| $n$ | orders $n!$ | fewest possible | selection sort |
|---|---|---|---|
| 3 | 6 | 3 | 3 |
| 4 | 24 | 5 | 6 |
| 10 | 3,628,800 | 22 | 45 |
| 1,000 | a 2,568-digit number | **8,530** | 499,500 |
| 1,000,000 | far bigger | **18,488,885** | 500 billion |

comparisons in the worst case

fewest possible $\approx n \log_2 n$: Python's `sorted()` comes **close**

# **You Try!** *(write your answers, 2 min)*

Now the shopkeeper has **5** bags, all different.




**(a)** In how many orders can the 5 bags be?

**(b)** Are 6 weighings always enough? Hint: $2^6 = 64$.

**(c)** What is the fewest that is always enough?

# **You Try!** *(write your answers, 2 min)*

Now the shopkeeper has **5** bags, all different.




**(a)** In how many orders can the 5 bags be?

$5 \times 4 \times 3 \times 2 \times 1 = \mathbf{120}$

**(b)** Are 6 weighings always enough? Hint: $2^6 = 64$.

**(c)** What is the fewest that is always enough?

# You Try! *(write your answers, 2 min)*

Now the shopkeeper has **5** bags, all different.




**(a)** In how many orders can the 5 bags be?

$5 \times 4 \times 3 \times 2 \times 1 = \mathbf{120}$

**(b)** Are 6 weighings always enough? Hint: $2^6 = 64$.

**No.** 64 patterns, 120 orders: some pattern gets two orders.

**(c)** What is the fewest that is always enough?

## **You Try!** *(write your answers, 2 min)*

Now the shopkeeper has **5** bags, all different.




**(a)** In how many orders can the 5 bags be?

$5 \times 4 \times 3 \times 2 \times 1 = \mathbf{120}$

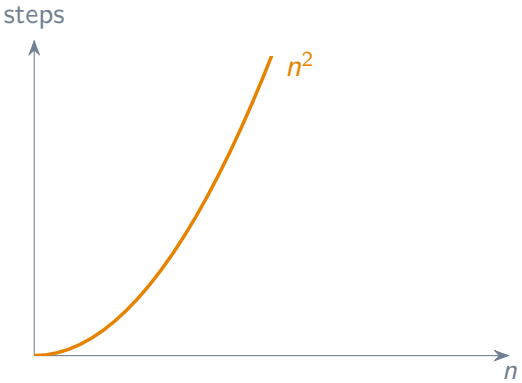
**(b)** Are 6 weighings always enough? Hint: $2^6 = 64$.

**No.** 64 patterns, 120 orders: some pattern gets two orders.

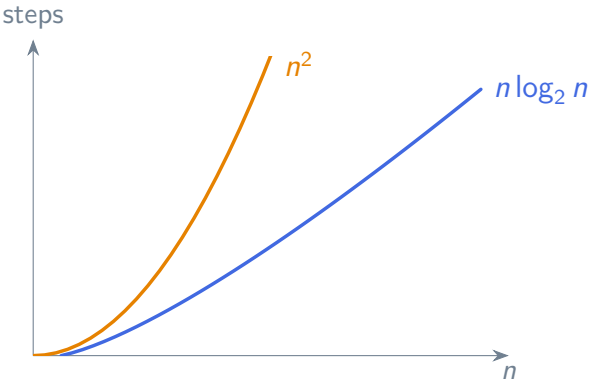
**(c)** What is the fewest that is always enough?

**7**, because $2^7 = 128 \geq 120$.

# A New Family: $O(n \log n)$

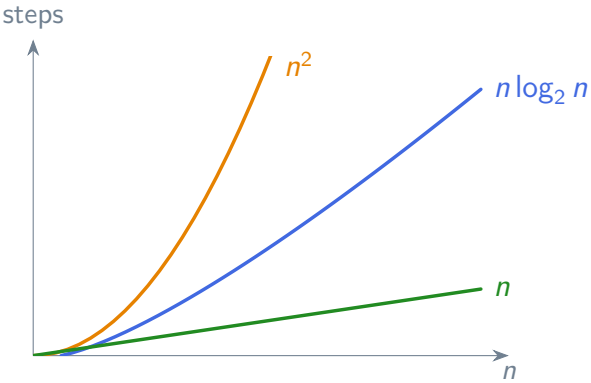


# A New Family: $O(n \log n)$



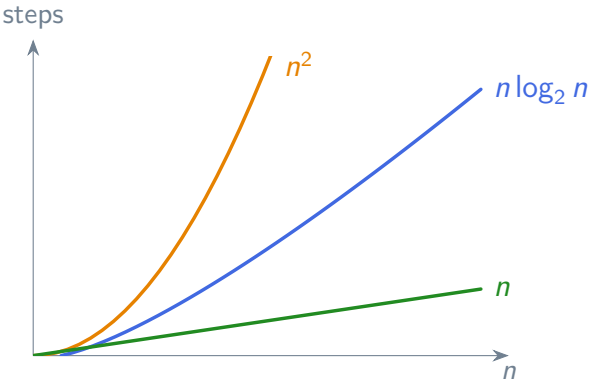

the speed limit: $\approx n \log_2 n$
$\rightarrow O(n \log n)$

# A New Family: $O(n \log n)$




the speed limit: $\approx n \log_2 n$
$\rightarrow O(n \log n)$

# A New Family: $O(n \log n)$

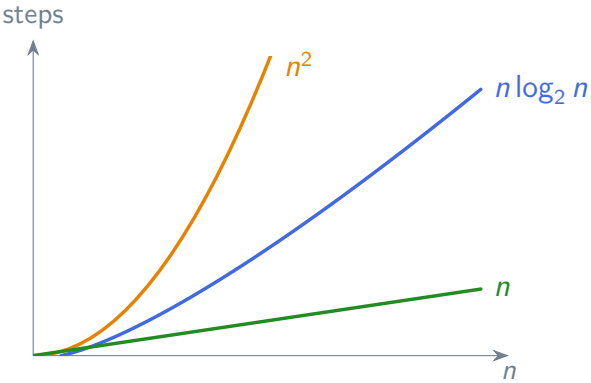



the speed limit: $\approx n \log_2 n$
$\rightarrow$ $O(n \log n)$

data $\times 2$

steps: a bit **more** than $\times 2$

# A New Family: $O(n \log n)$




the speed limit: $\approx n \log_2 n$
$\rightarrow$ $O(n \log n)$

data $\times 2$

steps: a bit **more** than $\times 2$

$$O(n) \; < \; O(n \log n) \; \ll \; O(n^2)$$

# You Try! *(write your answer, 1 min)*

A computer makes **10 million** comparisons each second. Sort **1,000,000** numbers.

**(a)** selection sort

`500,000,000,000` ÷  =

10 million / s

**(b)** Python's `sorted()`

`20,000,000` ÷  =

10 million / s

# You Try! *(write your answer, 1 min)*

A computer makes **10 million** comparisons each second. Sort **1,000,000** numbers.

**(a)** selection sort

`500,000,000,000` ÷  =  50,000 s ≈ **14 hours**

10 million / s

**(b)** Python's `sorted()`

`20,000,000` ÷  =

10 million / s

# You Try! *(write your answer, 1 min)*

A computer makes **10 million** comparisons each second. Sort **1,000,000** numbers.

**(a)** selection sort

`500,000,000,000` ÷  =  50,000 s ≈ **14 hours**

10 million / s

**(b)** Python's `sorted()`

`20,000,000` ÷  =  **2 seconds**

10 million / s

# You Try! *(write your answer, 1 min)*

A computer makes **10 million** comparisons each second. Sort **1,000,000** numbers.

**(a)** selection sort

`500,000,000,000` ÷  =  50,000 s ≈ **14 hours**

10 million / s

**(b)** Python's `sorted()`

`20,000,000` ÷  =  **2 seconds**

10 million / s

Same computer, same numbers. The **method** made the difference.

# You Try! *(predict, then I run it)*

`lec19.py` sorts **20,000** random numbers two ways. Guess each time.

| method | time |
|---|---|
| `selection_sort(L)` | |
| `sorted(L)` | |

# You Try! *(predict, then I run it)*

`lec19.py` sorts **20,000** random numbers two ways. Guess each time.

| method | time |
|---|---|
| `selection_sort(L)` | about **3** seconds |
| `sorted(L)` | |

# **You Try!** *(predict, then I run it)*

`lec19.py` sorts **20,000** random numbers two ways. Guess each time.

| **method** | **time** |
|---|---|
| `selection_sort(L)` | about **3** seconds |
| `sorted(L)` | about **0.0015** seconds! |

# You Try! *(predict, then I run it)*

`lec19.py` sorts **20,000** random numbers two ways. Guess each time.

| method | time |
|---|---|
| `selection_sort(L)` | about **3** seconds |
| `sorted(L)` | about **0.0015** seconds! |

`sorted`: **Timsort** (Tim Peters, 2002), written in C: $O(n \log n)$

Programming 2 (next semester) builds a sort like it: **merge sort**.

Times from one laptop. Yours will be a little different.

# Big Idea

Sorting by **comparing** needs about $n \log_2 n$ comparisons.

No program can do better: not yours, not Python's.

This is a **proof**, not a guess.

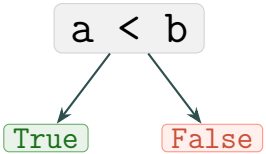
# The Loophole: Counting Sort

# Read the Proof Again

the proof used one fact: every question has only **2** answers

# Read the Proof Again

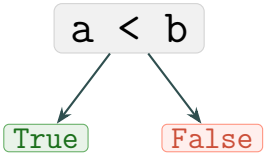
the proof used one fact: every question has only **2** answers




**2** answers

# Read the Proof Again

the proof used one fact: every question has only **2** answers

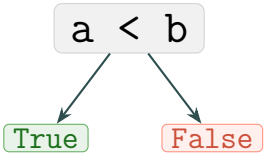



**2** answers

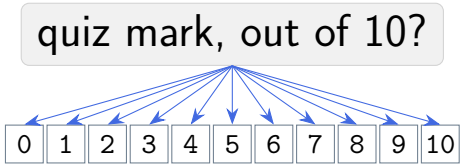
quiz mark, out of 10?

# Read the Proof Again

the proof used one fact: every question has only **2** answers



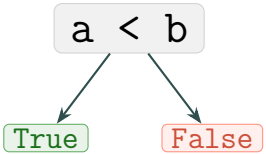

**2** answers



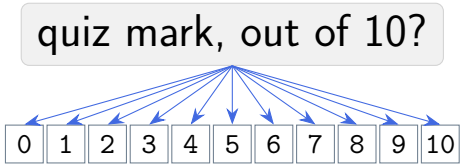

**11** answers

# Read the Proof Again

the proof used one fact: every question has only **2** answers




**2** answers

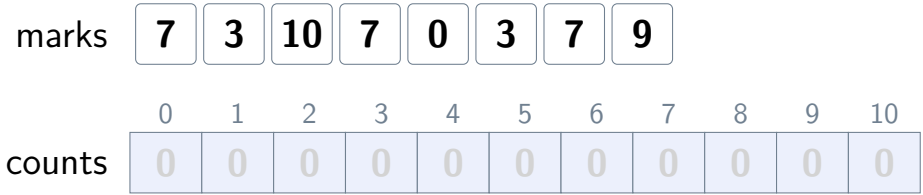



**11** answers

use the mark itself as a **place** in a list, like the sieve

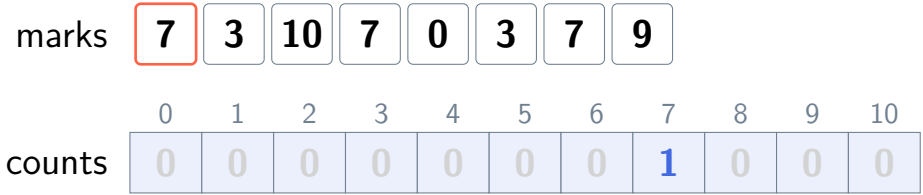
# Sorting Quiz Marks by Counting

Marks out of 10. One counter for each mark, 0 to 10.



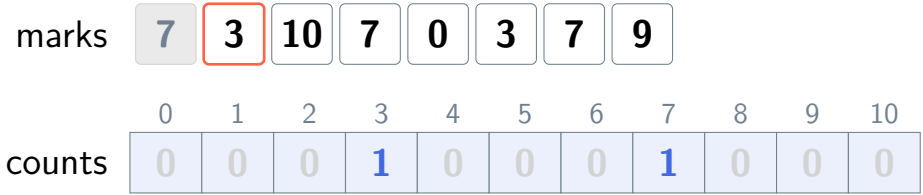
# Sorting Quiz Marks by Counting

Read a mark. Add 1 to its counter.



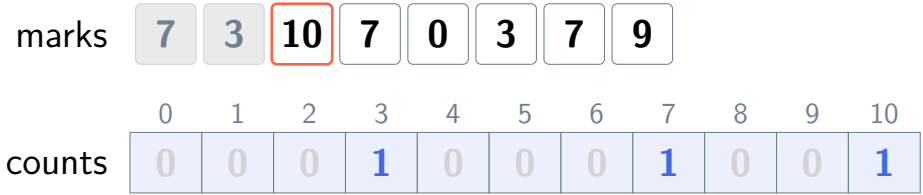
# Sorting Quiz Marks by Counting

Read a mark. Add 1 to its counter.



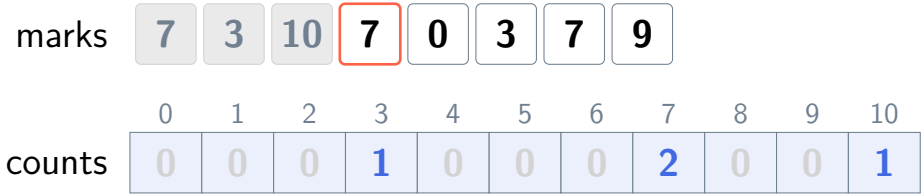
# Sorting Quiz Marks by Counting

Read a mark. Add 1 to its counter.



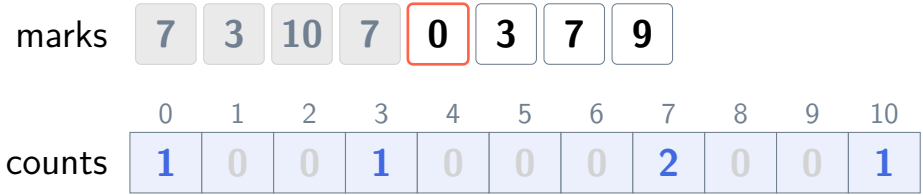
# Sorting Quiz Marks by Counting

Read a mark. Add 1 to its counter.



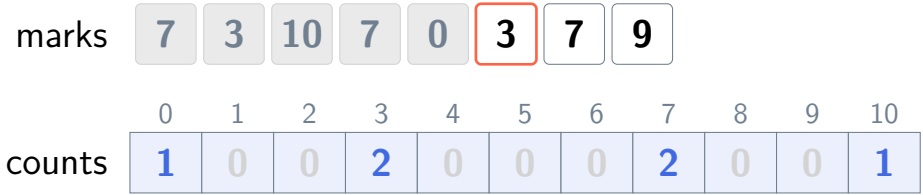
# Sorting Quiz Marks by Counting

Read a mark. Add 1 to its counter.



# Sorting Quiz Marks by Counting

Read a mark. Add 1 to its counter.



# Sorting Quiz Marks by Counting

Read a mark. Add 1 to its counter.



# Sorting Quiz Marks by Counting

Read a mark. Add 1 to its counter.

| marks | 7 | 3 | 10 | 7 | 0 | 3 | 7 | 9 |
|---|---|---|---|---|---|---|---|---|

| | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 |
|---|---|---|---|---|---|---|---|---|---|---|---|
| counts | 1 | 0 | 0 | 2 | 0 | 0 | 0 | 3 | 0 | 1 | 1 |

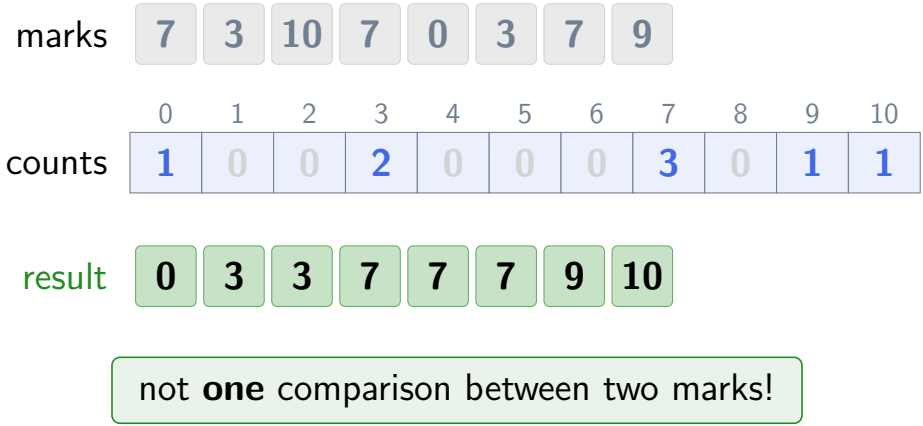
# Sorting Quiz Marks by Counting

Read the counters, left to right.



# Sorting Quiz Marks by Counting

Read the counters, left to right.




not **one** comparison between two marks!

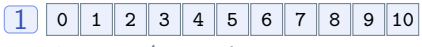
# Counting Sort in Python

```
1  def counting_sort(marks, top):
2      """marks: whole numbers, 0 to top."""
3      counts = [0] * (top + 1)
4      for m in marks:
5          counts[m] += 1
6      result = []
7      for value in range(top + 1):
8          for k in range(counts[value]):
9              result.append(value)
10     return result
```

```
print(counting_sort([7, 3, 10, 7, 0, 3, 7, 9], 10))
# [0, 3, 3, 7, 7, 7, 9, 10]
```

# Counting Sort in Python

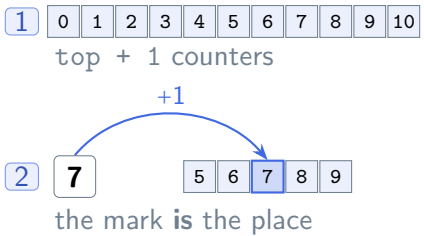
```
1  def counting_sort(marks, top):
2      """marks: whole numbers, 0 to top."""
3      counts = [0] * (top + 1)      1
4      for m in marks:
5          counts[m] += 1
6      result = []
7      for value in range(top + 1):
8          for k in range(counts[value]):
9              result.append(value)
10     return result
```




```
print(counting_sort([7, 3, 10, 7, 0, 3, 7, 9], 10))
# [0, 3, 3, 7, 7, 7, 9, 10]
```

# Counting Sort in Python

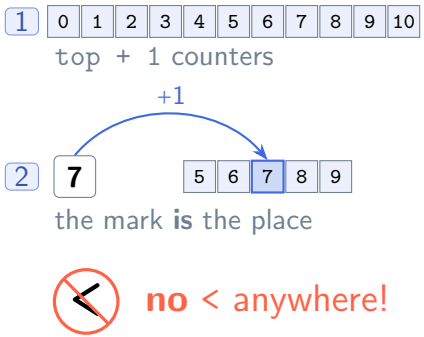
```
1  def counting_sort(marks, top):
2      """marks: whole numbers, 0 to top."""
3      counts = [0] * (top + 1)          1
4      for m in marks:
5          counts[m] += 1                2
6      result = []
7      for value in range(top + 1):
8          for k in range(counts[value]):
9              result.append(value)
10     return result
```




```
print(counting_sort([7, 3, 10, 7, 0, 3, 7, 9], 10))
# [0, 3, 3, 7, 7, 7, 9, 10]
```

# Counting Sort in Python

```
1  def counting_sort(marks, top):
2      """marks: whole numbers, 0 to top."""
3      counts = [0] * (top + 1)          1
4      for m in marks:
5          counts[m] += 1                2
6      result = []
7      for value in range(top + 1):
8          for k in range(counts[value]):
9              result.append(value)
10     return result
```




```
print(counting_sort([7, 3, 10, 7, 0, 3, 7, 9], 10))
# [0, 3, 3, 7, 7, 7, 9, 10]
```

# Faster Than the Speed Limit?

**slide 34** the **speed limit**: a sort that **compares** needs $\approx n \log_2 n$ comparisons

each comparison has only 2 answers, and there are $n!$ orders to tell apart

$n$ marks out of `top`:

# Faster Than the Speed Limit?

**slide 34** the **speed limit**: a sort that **compares** needs $\approx n \log_2 n$ comparisons

each comparison has only 2 answers, and there are $n!$ orders to tell apart

$n$ marks out of `top`:

$$\text{steps} \approx \underbrace{n}_{\text{count}} + \underbrace{\texttt{top} + 1}_{\text{counters}} + \underbrace{n}_{\text{append}}$$

# Faster Than the Speed Limit?

**slide 34** the **speed limit**: a sort that **compares** needs $\approx n \log_2 n$ comparisons

each comparison has only 2 answers, and there are $n!$ orders to tell apart

$n$ marks out of `top`:

$$\text{steps} \approx \underbrace{n}_{\text{count}} + \underbrace{\texttt{top} + 1}_{\text{counters}} + \underbrace{n}_{\text{append}}$$

$\rightarrow$ $O(n)$ (`top` fixed)

# Faster Than the Speed Limit?

**slide 34** the **speed limit**: a sort that **compares** needs $\approx n \log_2 n$ comparisons

each comparison has only 2 answers, and there are $n!$ orders to tell apart

$n$ marks out of `top`:

$$\text{steps} \approx \underbrace{n}_{\text{count}} + \underbrace{\texttt{top} + 1}_{\text{counters}} + \underbrace{n}_{\text{append}} \quad \rightarrow \quad O(n) \quad (\texttt{top fixed})$$

**sort 1,000,000 marks out of 100**

best sort that **compares** — **18.5 million comparisons**

counting sort — **2 million steps**

# Faster Than the Speed Limit?

**slide 34** the **speed limit**: a sort that **compares** needs $\approx n \log_2 n$ comparisons

each comparison has only 2 answers, and there are $n!$ orders to tell apart

$n$ marks out of `top`:

$$\text{steps} \approx \underbrace{n}_{\text{count}} + \underbrace{\texttt{top} + 1}_{\text{counters}} + \underbrace{n}_{\text{append}} \quad \rightarrow \quad O(n) \quad (\texttt{top fixed})$$

**sort 1,000,000 marks out of 100**

best sort that **compares** — **18.5 million comparisons**

counting sort — **2 million steps**

no rule is broken: counting sort **never compares**

`counts[m] += 1` asks "what is `m`?": **101** answers, not 2

# Bug Hunt! *(write the line number and the fix, 30 s)*

Sort some dice rolls (a die shows 1 to 6):

```
1 rolls = [3, 6, 1, 3, 5]
2 counts = [0] * 6
3 for r in rolls:
4     counts[r] += 1
5 result = []
6 for value in range(len(counts)):
7     for k in range(counts[value]):
8         result.append(value)
9 print(result)
```

## Symptom:

```
  line 4
IndexError: list index
out of range
```

Yesterday's rolls sorted fine!

# Bug Hunt! *(write the line number and the fix, 30 s)*

Sort some dice rolls (a die shows 1 to 6):

```
1  rolls = [3, 6, 1, 3, 5]
2  counts = [0] * 6
3  for r in rolls:
4      counts[r] += 1
5  result = []
6  for value in range(len(counts)):
7      for k in range(counts[value]):
8          result.append(value)
9  print(result)
```

## Symptom:

```
  line 4
IndexError: list index
out of range
```

Yesterday's rolls sorted fine!

**Line 2** `counts = [0] * 7`

| 0 | 1 | 2 | 3 | 4 | 5 | 6 | no place! |
|---|---|---|---|---|---|---|---|

`[0] * 6`: places 0 to 5

Place 0 is never used. Always test the **top** value.

# When Counting Sort Cannot Help

✓ **small whole numbers**

stars, 1 to 5




marks, 0 to 100



ages, 0 to 120



# When Counting Sort Cannot Help

✓ **small whole numbers**

stars, 1 to 5




marks, 0 to 100



ages, 0 to 120

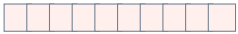


✗ **everything else**

`counts['Ali']` `TypeError`

`counts[1299.50]` `TypeError`

CNIC: 13 digits



10,000,000,000,000 counters!

# When Counting Sort Cannot Help

## ✓ small whole numbers

stars, 1 to 5




marks, 0 to 100



ages, 0 to 120

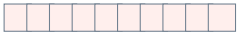


## ✗ everything else

`counts['Ali']` `TypeError`

`counts[1299.50]` `TypeError`

CNIC: 13 digits



10,000,000,000,000 counters!

most sorting must **compare** → `sorted()` compares

# Sorting in Python

# sorted() and .sort()

```
1 L = [5, 2, 8, 1]
2 M = sorted(L)
3 L.sort()
4 sorted(L, reverse=True)
5
```

# sorted() and .sort()

```
1 L = [5, 2, 8, 1]
2 M = sorted(L)
3 L.sort()
4 sorted(L, reverse=True)
5
```



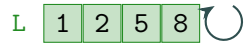
# sorted() and .sort()

```
1 L = [5, 2, 8, 1]
2 M = sorted(L)
3 L.sort()
4 sorted(L, reverse=True)
5
```

sorted(L)




L.sort()




same list

returns None

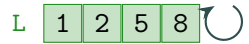
# sorted() and .sort()

```
1 L = [5, 2, 8, 1]
2 M = sorted(L)
3 L.sort()
4 sorted(L, reverse=True)
5 # [8, 5, 2, 1]
```

sorted(L)




L.sort()

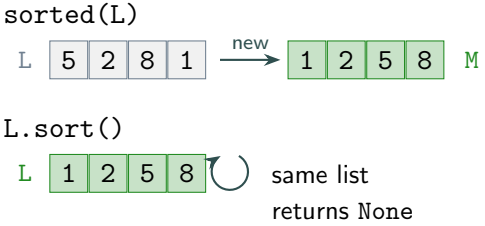



same list

returns None

# sorted() and .sort()

```
1 L = [5, 2, 8, 1]
2 M = sorted(L)
3 L.sort()
4 sorted(L, reverse=True)
5 # [8, 5, 2, 1]
```




**Watch out:** `A B ... Z` < `a b ... z` capitals come **first**

```
sorted(['bilal', 'Ali', 'zara', 'Hamza'])
```

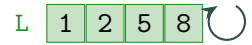
# sorted() and .sort()

```
1 L = [5, 2, 8, 1]
2 M = sorted(L)
3 L.sort()
4 sorted(L, reverse=True)
5 # [8, 5, 2, 1]
```

sorted(L)




L.sort()




same list

returns None

**Watch out:** `A B ... Z` < `a b ... z` capitals come **first**

```
sorted(['bilal', 'Ali', 'zara', 'Hamza'])
# ['Ali', 'Hamza', 'bilal', 'zara']
```

# Sort by Something Else: `key=`

```
1 cities = ['Karachi', 'Hub', 'Multan', 'Swat']
2 sorted(cities, key=len)
```

# Sort by Something Else: `key=`

```
1 cities = ['Karachi', 'Hub', 'Multan', 'Swat']
2 sorted(cities, key=len)    # ['Hub', 'Swat', 'Multan', 'Karachi']
```

Python compares `len`: `Hub` 3 | `Swat` 4 | `Multan` 6 | `Karachi` 7

# Sort by Something Else: `key=`

```
cities = ['Karachi', 'Hub', 'Multan', 'Swat']
sorted(cities, key=len)    # ['Hub', 'Swat', 'Multan', 'Karachi']
```

Python compares `len`: `Hub` 3 | `Swat` 4 | `Multan` 6 | `Karachi` 7

A **merit list**, highest mark first:

```
results = [('Ali', 72), ('Sana', 91), ('Omar', 65)]
merit = sorted(results, key=lambda r: r[1], reverse=True)
print(merit)
```

# Sort by Something Else: `key=`

```
cities = ['Karachi', 'Hub', 'Multan', 'Swat']
sorted(cities, key=len)    # ['Hub', 'Swat', 'Multan', 'Karachi']
```

Python compares `len`: `Hub` 3 | `Swat` 4 | `Multan` 6 | `Karachi` 7

A **merit list**, highest mark first:

```
results = [('Ali', 72), ('Sana', 91), ('Omar', 65)]
merit = sorted(results, key=lambda r: r[1], reverse=True)
print(merit)    # [('Sana', 91), ('Ali', 72), ('Omar', 65)]
```

# Sort by Something Else: `key=`

```
cities = ['Karachi', 'Hub', 'Multan', 'Swat']
sorted(cities, key=len)    # ['Hub', 'Swat', 'Multan', 'Karachi']
```

Python compares `len`: `Hub` 3 | `Swat` 4 | `Multan` 6 | `Karachi` 7

A **merit list**, highest mark first:

```
results = [('Ali', 72), ('Sana', 91), ('Omar', 65)]
merit = sorted(results, key=lambda r: r[1], reverse=True)
print(merit)    # [('Sana', 91), ('Ali', 72), ('Omar', 65)]
```

`('Sana', 91)` —`lambda r: r[1]`→ `91`

a function with no name (Lecture 10)

# You Try! *(predict, write each answer, 2 min)*

**(a)** `sorted([3, 1, 2], reverse=True)`

**(b)** `sorted(['pear', 'Fig', 'apple'])`

**(c)** `sorted(['ccc', 'a', 'bb'], key=len)`

**(d)**
```
L = [3, 1, 2]
x = L.sort()
print(x, L)
```

# You Try! *(predict, write each answer, 2 min)*

**(a)**
```
sorted([3, 1, 2], reverse=True)
```
```
[3, 2, 1]
```

**(b)**
```
sorted(['pear', 'Fig', 'apple'])
```

**(c)**
```
sorted(['ccc', 'a', 'bb'], key=len)
```

**(d)**
```
L = [3, 1, 2]
x = L.sort()
print(x, L)
```

# You Try! *(predict, write each answer, 2 min)*

**(a)**
```
sorted([3, 1, 2], reverse=True)
```
```
[3, 2, 1]
```

**(b)**
```
sorted(['pear', 'Fig', 'apple'])
```
```
['Fig', 'apple', 'pear']
```

**(c)**
```
sorted(['ccc', 'a', 'bb'], key=len)
```

**(d)**
```
L = [3, 1, 2]
x = L.sort()
print(x, L)
```

# You Try! *(predict, write each answer, 2 min)*

**(a)**
```
sorted([3, 1, 2], reverse=True)
```
```
[3, 2, 1]
```

**(b)**
```
sorted(['pear', 'Fig', 'apple'])
```
```
['Fig', 'apple', 'pear']
```

**(c)**
```
sorted(['ccc', 'a', 'bb'], key=len)
```
```
['a', 'bb', 'ccc']
```

**(d)**
```
L = [3, 1, 2]
x = L.sort()
print(x, L)
```

# You Try! *(predict, write each answer, 2 min)*

**(a)** `sorted([3, 1, 2], reverse=True)`

`[3, 2, 1]`

**(b)** `sorted(['pear', 'Fig', 'apple'])`

`['Fig', 'apple', 'pear']`

**(c)** `sorted(['ccc', 'a', 'bb'], key=len)`

`['a', 'bb', 'ccc']`

**(d)**
```
L = [3, 1, 2]
x = L.sort()
print(x, L)
```

`None [1, 2, 3]`

# Summary

# The Cheat Card

| | sort | comparisons | Big-O |
|---|---|---|---|
|  | **selection**: find the smallest, swap it in | $n(n-1)/2$ | $O(n^2)$ |
|  | **speed limit**: any sort that compares | $\geq \log_2 n! \approx n \log_2 n$ | |
|  | `sorted(L)`: Timsort, written in C | $\approx n \log_2 n$ | $O(n \log n)$ |
|  | **counting**: small whole numbers only | **0** | $O(n)$ |

sieve: cross out multiples, $\frac{n}{2} + \frac{n}{3} + \frac{n}{5} + \ldots < 3n$, then `flags[k]` is one look

`sorted(L)`: new list

`L.sort()`: changes L

`reverse=True`, `key=`

# Big Idea

Two weighings cannot always sort three bags:

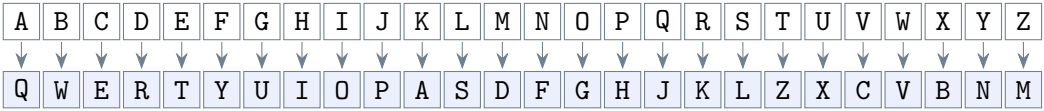
**6** orders, only **4** answer patterns.

That one idea is a **speed limit** for every sort that compares.

Counting sort gets past it: it **never compares**.

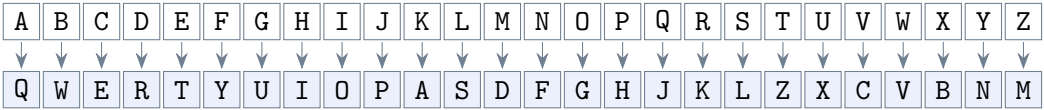
# Next Time: Codes & Codebreakers

mix up the 26 letters: a secret code



# Next Time: Codes & Codebreakers

mix up the 26 letters: a secret code




26! = 403,291,461,126,605,635,584,000,000 ways: too many to try

# Next Time: Codes & Codebreakers

mix up the 26 letters: a secret code

| A | B | C | D | E | F | G | H | I | J | K | L | M | N | O | P | Q | R | S | T | U | V | W | X | Y | Z |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| ↓ | ↓ | ↓ | ↓ | ↓ | ↓ | ↓ | ↓ | ↓ | ↓ | ↓ | ↓ | ↓ | ↓ | ↓ | ↓ | ↓ | ↓ | ↓ | ↓ | ↓ | ↓ | ↓ | ↓ | ↓ | ↓ |
| Q | W | E | R | T | Y | U | I | O | P | A | S | D | F | G | H | J | K | L | Z | X | C | V | B | N | M |

26! = 403,291,461,126,605,635,584,000,000 ways: too many to try




9th-century Baghdad: such codes were broken by **counting** letters

# Questions?

# Exit Ticket

**Today's question** *(2 minutes)*

**(a)** `L = [4, 3, 1, 2]`. Write `L` after the **first** pass of selection sort.

**(b)** Can **any** plan always sort **4** bags with **4** weighings? Say why.

**COMP 102 — Exit Ticket #19**

Name: _____

Roll #: _____ Date: _____

**Answer:**

*Hand it to me on your way out.*
*No name, no attendance.*