Lecture 18: Counting Steps, Big-O

Comp 102

Fakhir Shaheen

Forman Christian University

Less Than Half a Second

You Search. Google Answers.



About 3,140,000 results **(0.38 seconds)**

You Search. Google Answers.

python KeyError

About 3,140,000 results **(0.38 seconds)**



hundreds of billions of pages

You Search. Google Answers.





search them all in



hundreds of billions of pages

0.38 s

You Search. Google Answers.





search them all in

How?



hundreds of billions of pages

0.38 s

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

One idea: Google reads **every page** while you wait. How long?



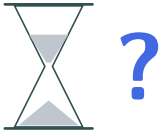
100,000,000,000
pages

÷



10,000,000
pages each second

=



you wait

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

One idea: Google reads **every page** while you wait. How long?



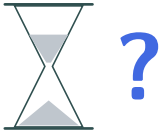
100,000,000,000
pages

÷



10,000,000
pages each second

=



you wait

100,000,000,000 ÷ 10,000,000 = **10,000 seconds**

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

One idea: Google reads **every page** while you wait. How long?



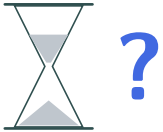
100,000,000,000
pages

÷



10,000,000
pages each second

=



you wait

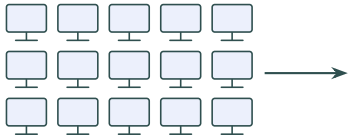
$100{,}000{,}000{,}000 \div 10{,}000{,}000 =$ **10,000 seconds**

$\approx$ **2 hours and 47 minutes** . . . for **one** search!

Use More Computers?

Use More Computers?

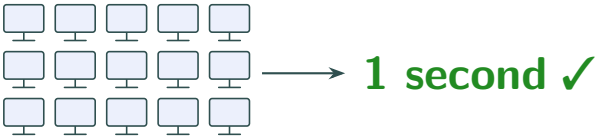
10,000 computers



→ **1 second ✓**

Use More Computers?

10,000 computers



100,000 searches **each second**



Use More Computers?

10,000 computers



→ **1 second ✓**

100,000 searches **each second**



$$\underbrace{10{,}000}_{\text{each search}} \times \underbrace{100{,}000}_{\text{searches each second}} = \mathbf{1{,}000{,}000{,}000}\ \text{computers!}$$

Use More Computers?

10,000 computers



→ **1 second ✓**

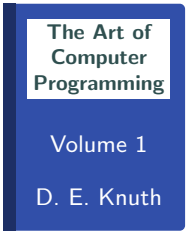
100,000 searches **each second**



$\underbrace{10{,}000}_{\text{each search}} \times \underbrace{100{,}000}_{\text{searches each second}} = \mathbf{1{,}000{,}000{,}000}$ computers!

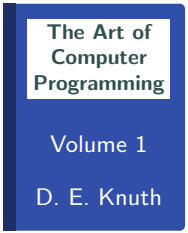
faster computer ✗ | bigger computer ✗ | **far fewer steps ✓**

Count the Steps



Donald Knuth, 1968

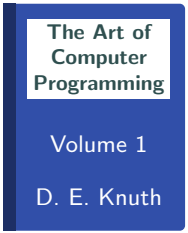
Count the Steps



Before you run a program, **count its steps**.

Donald Knuth, 1968

Count the Steps

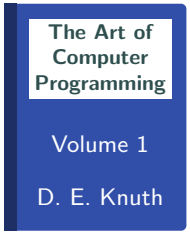


Donald Knuth, 1968

Before you run a program, **count its steps**.

his name for it: **analysis of algorithms**

Count the Steps



Before you run a program, **count its steps**.

his name for it: **analysis of algorithms**

Donald Knuth, 1968

Today: 1 count the steps → 2 watch them grow → 3 name it: **Big-O**

Counting Steps

Why Not Use a Stopwatch?

Same code. Same list of 1,000,000 numbers. Three runs, for example:



My laptop

0.08 s



Lab computer

0.21 s



My laptop again

0.11 s

Why Not Use a Stopwatch?

Same code. Same list of 1,000,000 numbers. Three runs, for example:



My laptop

0.08 s



Lab computer

0.21 s



My laptop again

0.11 s

time depends on code | data | the computer | what else it runs

Why Not Use a Stopwatch?

Same code. Same list of 1,000,000 numbers. Three runs, for example:



My laptop

0.08 s

2,000,002 steps



Lab computer

0.21 s

2,000,002 steps



My laptop again

0.11 s

2,000,002 steps

time depends on | code | data | the computer | what else it runs

steps depend on | code | data | **. . . only!**

What Is One Step?

One simple line = **one step**, **every time it runs**.

simple line

```
s = s + x
```
1

```
if x > 5:
```
1

```
return s
```
1

What Is One Step?

One simple line = **one step**, **every time it runs**.

simple line		loop line	
`s = s + x`	1	`for x in L:`	n
`if x > 5:`	1		
`return s`	1		



once **for each pass**
`in` hands out the next item

What Is One Step?

One simple line = **one step**, **every time it runs**.

simple line	loop line	hidden loop!
`s = s + x` 1	`for x in L:` n	`if x in L:` n
`if x > 5:` 1		
`return s` 1	once **for each pass** `in` hands out the next item	looks like 1 step, really *n* `in` **searches** the list

Counting the Steps of `total(L)`

n = `len(L)`, the number of items.

```
1  def total(L):                # runs
2      s = 0
3      for x in L:
4          s = s + x
5      return s
```

Counting the Steps of `total(L)`

n = `len(L)`, the number of items.

```
1  def total(L):                    # runs
2      s = 0                        1
3      for x in L:
4          s = s + x
5      return s
```

Counting the Steps of `total(L)`

n = `len(L)`, the number of items.

```
1  def total(L):                # runs
2      s = 0                    1
3      for x in L:              n
4          s = s + x
5      return s
```

Counting the Steps of `total(L)`

n = `len(L)`, the number of items.

```
1  def total(L):                # runs
2      s = 0                    1
3      for x in L:              n
4          s = s + x            n
5      return s
```

Counting the Steps of `total(L)`

n = `len(L)`, the number of items.

```
1  def total(L):                # runs
2      s = 0                    1
3      for x in L:              n
4          s = s + x            n
5      return s                 1
```

Counting the Steps of `total(L)`

n = `len(L)`, the number of items.

```
1  def total(L):                # runs
2      s = 0                    1
3      for x in L:              n
4          s = s + x            n
5      return s                 1
```

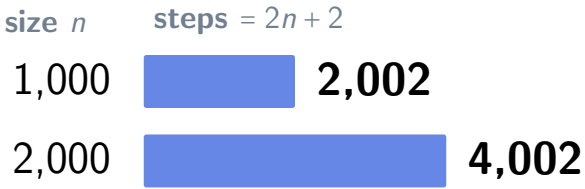
$$\text{Steps} = 1 + n + n + 1 = \mathbf{2n + 2}$$

The `def` line is not a step: it only gives the function its name.

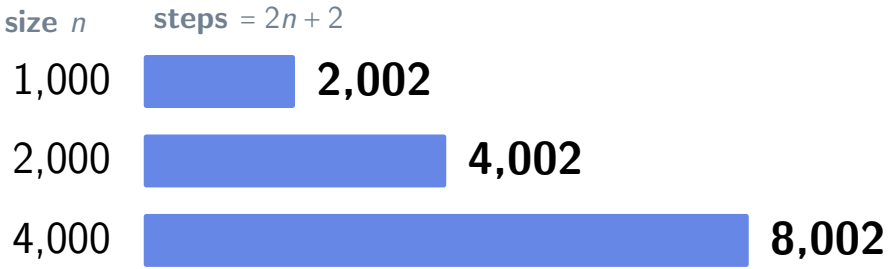
Make the List Bigger

size n	steps $= 2n + 2$
1,000	 **2,002**

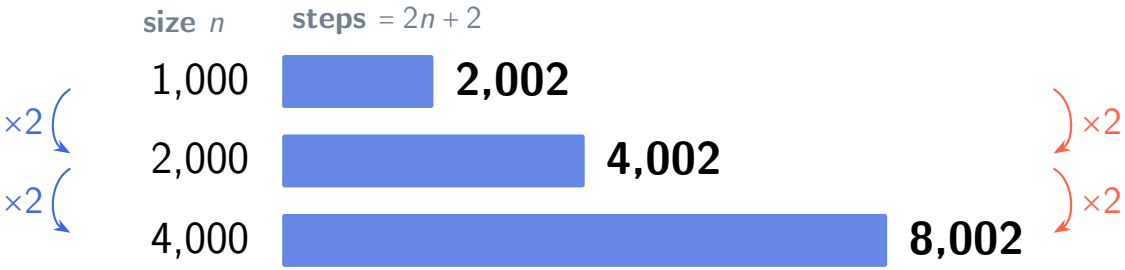
Make the List Bigger



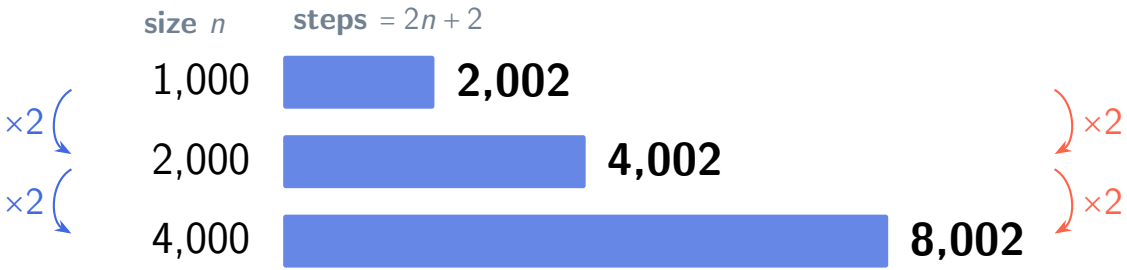
Make the List Bigger



Make the List Bigger



Make the List Bigger



$n = 1{,}000{,}000$: steps $= 2{,}000{,}000 + 2$

for big n, the +2 is too small to matter

You Try! *(write the numbers, 1 min)*

How many times does each line run? What is the total?

```
1 def times_table(n):                    # runs
2     for i in range(n):
3         for j in range(n):
4             print(i * j)
```

For $n = 3$:

You Try! *(write the numbers, 1 min)*

How many times does each line run? What is the total?

```
1 def times_table(n):                    # runs
2     for i in range(n):                 n
3         for j in range(n):
4             print(i * j)
```

For $n = 3$:

You Try! *(write the numbers, 1 min)*

How many times does each line run? What is the total?

```
1  def times_table(n):                  # runs
2      for i in range(n):               n
3          for j in range(n):           n * n
4              print(i * j)
```

For $n = 3$:

You Try! *(write the numbers, 1 min)*

How many times does each line run? What is the total?

```
1  def times_table(n):                # runs
2      for i in range(n):             n
3          for j in range(n):         n * n
4              print(i * j)           n * n
```

For $n = 3$:

You Try! *(write the numbers, 1 min)*

How many times does each line run? What is the total?

```
1  def times_table(n):                # runs
2      for i in range(n):             n
3          for j in range(n):         n * n
4              print(i * j)           n * n
```

$$\text{Steps} = n + n^2 + n^2 = \mathbf{2n^2 + n}$$

For $n = 3$:

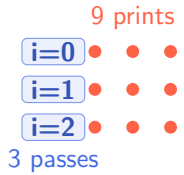
You Try! *(write the numbers, 1 min)*

How many times does each line run? What is the total?

```
1  def times_table(n):                  # runs
2      for i in range(n):               n
3          for j in range(n):           n * n
4              print(i * j)             n * n
```

Steps = $n + n^2 + n^2 = \mathbf{2n^2 + n}$

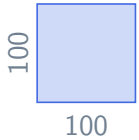
For $n = 3$: $3 + 9 + 9 =$ **21** steps



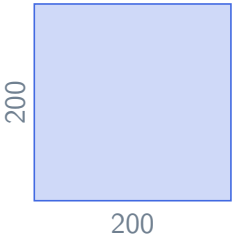
You Try! *(write your answer, 1 min)*

The data doubles. How many times more steps?

`times_table(100)`



`times_table(200)`

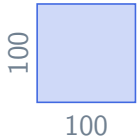


$$\text{steps} = 2n^2 + n$$

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

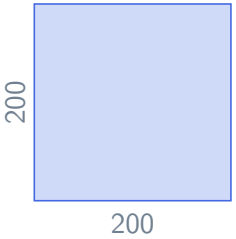
The data doubles. How many times more steps?

`times_table(100)`



20,100 steps

`times_table(200)`



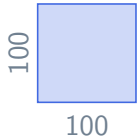
80,200 steps

$$\text{steps} = 2n^2 + n$$

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

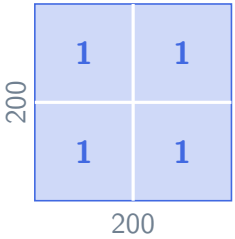
The data doubles. How many times more steps?

`times_table(100)`



20,100 steps

`times_table(200)`



80,200 steps

$$\text{steps} = 2n^2 + n$$

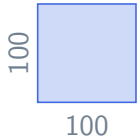
4 small squares

→ about 4 times!

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

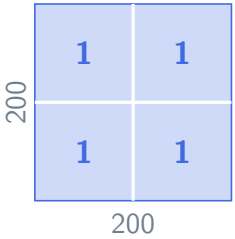
The data doubles. How many times more steps?

`times_table(100)`



20,100 steps

`times_table(200)`



80,200 steps

$$\text{steps} = 2n^2 + n$$

4 small squares

→ about 4 times!

data ×2 → steps ×4

The Doubling Question

Big Idea

Do not ask: *how many seconds?*

Ask: when the data **doubles**,
what happens to the **steps**?

`total`: $\times 2$ `times_table`: $\times 4$

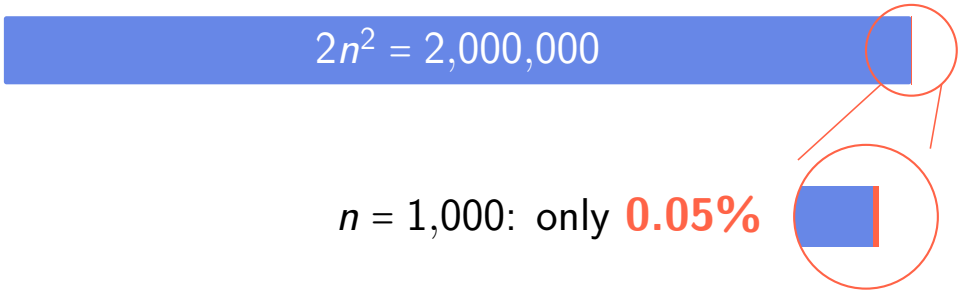
Keep the Biggest Part

`times_table`: $2n^2 + n$ at $n = 1{,}000$

$$2n^2 = 2{,}000{,}000$$

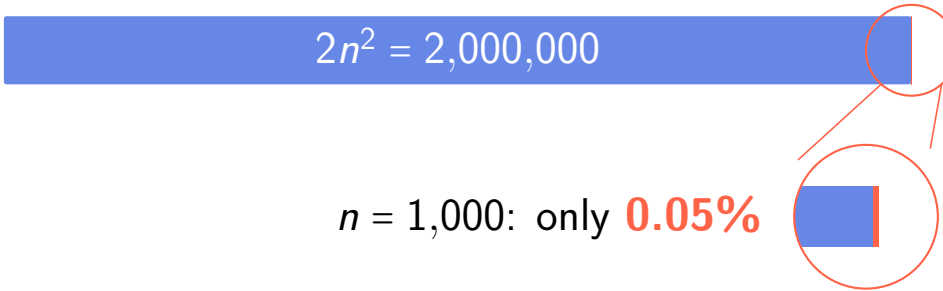
Keep the Biggest Part

`times_table`: $2n^2 + n$ at $n = 1{,}000$



Keep the Biggest Part

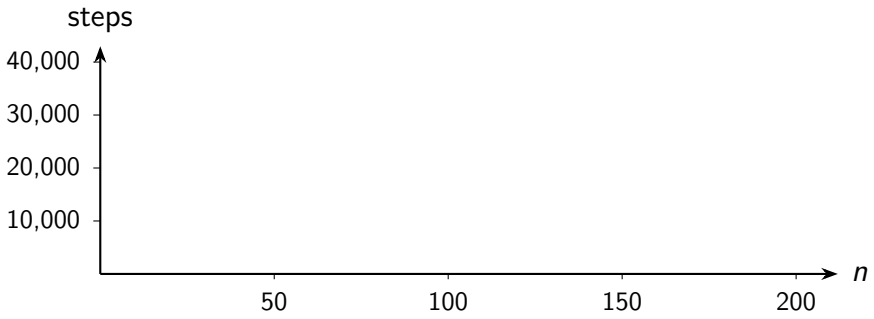
`times_table`: $2n^2 + n$ at $n = 1{,}000$



$2n^2 + \cancel{n} \rightarrow$ keep the **fastest-growing** part

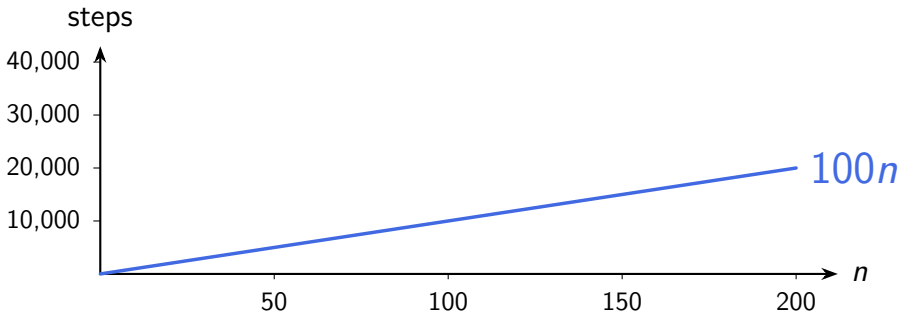
The Number in Front Loses

Which is bigger: $100n$ or n^2?



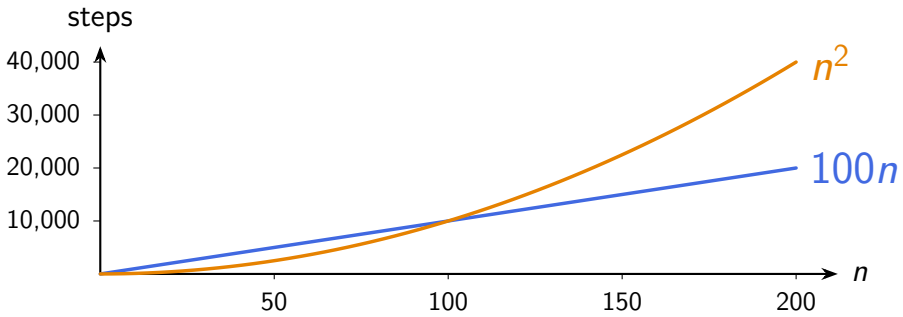
The Number in Front Loses

Which is bigger: $100n$ or n^2?



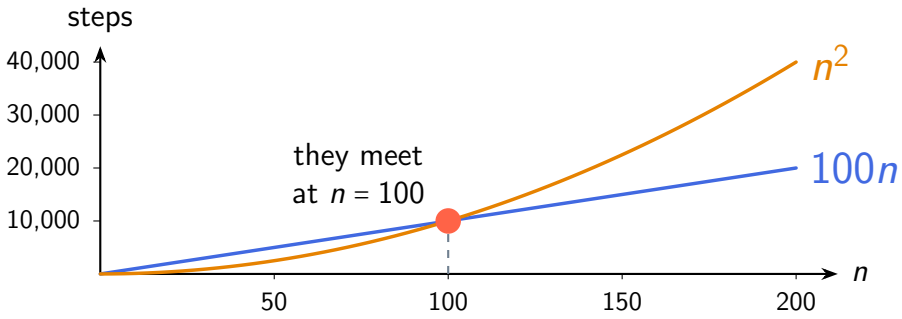
The Number in Front Loses

Which is bigger: $100n$ or n^2?



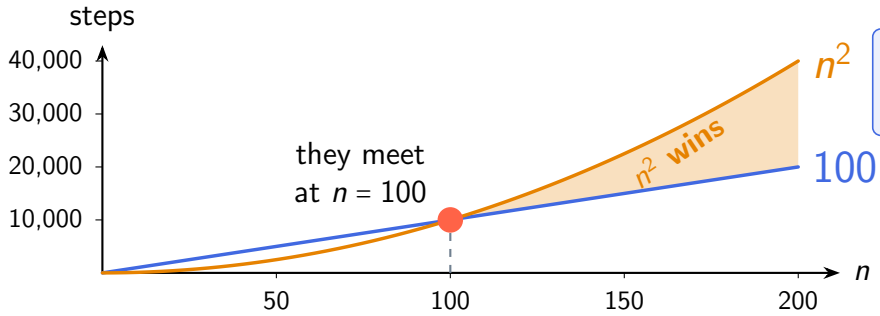
The Number in Front Loses

Which is bigger: $100n$ or n^2?



The Number in Front Loses

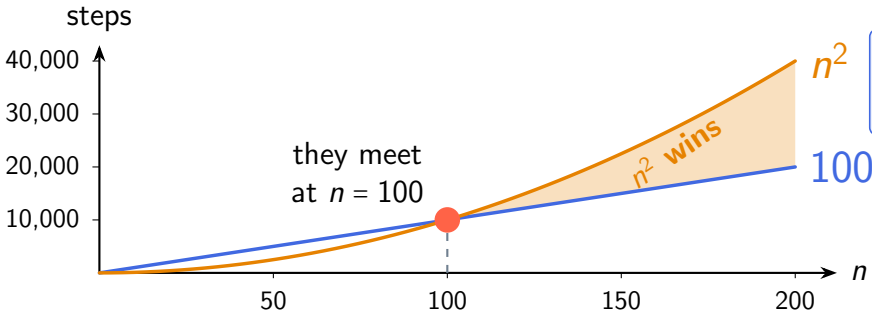
Which is bigger: $100n$ or n^2?



after that,
n^2 wins **forever**

The Number in Front Loses

Which is bigger: $100n$ or n^2?



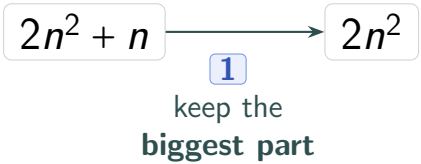
after that, n^2 wins **forever**

The **power** of n matters. The number in front does not.

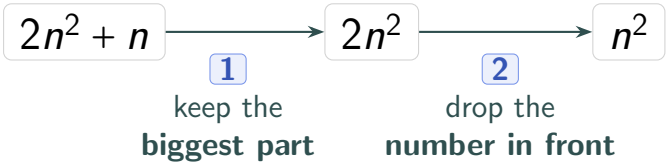
Big-O: The Name of the Family

$2n^2 + n$

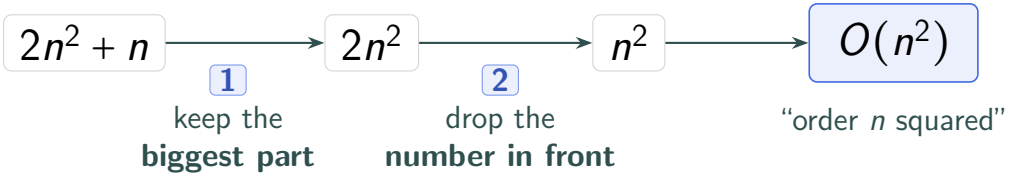
Big-O: The Name of the Family



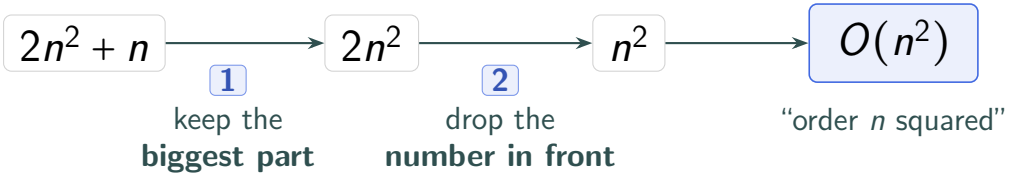
Big-O: The Name of the Family

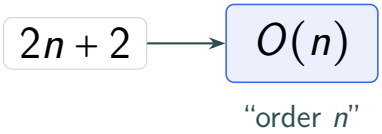


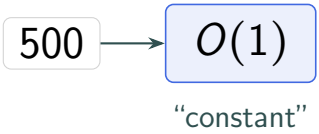
Big-O: The Name of the Family



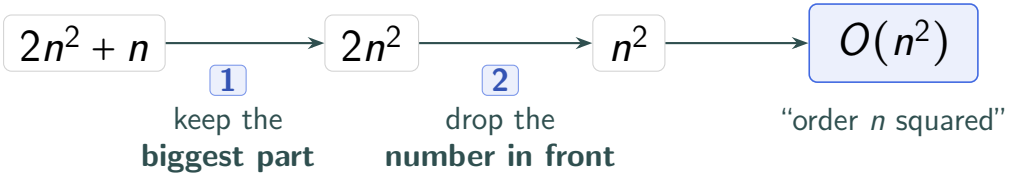
Big-O: The Name of the Family







Big-O: The Name of the Family





Big-O names the **family**, not the exact count.

You Try! *(write the Big-O, 1 min)*

(a) $7n + 100$

(b) $n^2 + 1000n$

(c) 42

(d) $n(n-1)/2$

(e) $3n + 5n$

You Try! *(write the Big-O, 1 min)*

(a) $7n + 100$ $O(n)$

(b) $n^2 + 1000n$

(c) 42

(d) $n(n-1)/2$

(e) $3n + 5n$

You Try! *(write the Big-O, 1 min)*

(a) $7n + 100$ $O(n)$

(b) $n^2 + 1000n$ $O(n^2)$ (for big n, n^2 wins)

(c) 42

(d) $n(n-1)/2$

(e) $3n + 5n$

You Try! *(write the Big-O, 1 min)*

(a) $7n + 100$ $O(n)$

(b) $n^2 + 1000n$ $O(n^2)$ (for big n, n^2 wins)

(c) 42 $O(1)$

(d) $n(n-1)/2$

(e) $3n + 5n$

You Try! *(write the Big-O, 1 min)*

(a) $7n + 100$ $O(n)$

(b) $n^2 + 1000n$ $O(n^2)$ (for big n, n^2 wins)

(c) 42 $O(1)$

(d) $n(n-1)/2$ $O(n^2)$ (it is $\frac{1}{2}n^2 - \frac{1}{2}n$)

(e) $3n + 5n$

You Try! *(write the Big-O, 1 min)*

(a) $7n + 100$ $O(n)$

(b) $n^2 + 1000n$ $O(n^2)$ (for big n, n^2 wins)

(c) 42 $O(1)$

(d) $n(n-1)/2$ $O(n^2)$ (it is $\frac{1}{2}n^2 - \frac{1}{2}n$)

(e) $3n + 5n$ $O(n)$ (it is $8n$)

Five Families

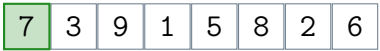
Linear: $O(n)$

```
1 def find(L, x):
2     for item in L:
3         if item == x:
4             return True
5     return False
```

Linear: $O(n)$

```
1 def find(L, x):
2     for item in L:
3         if item == x:
4             return True
5     return False
```

best case: x = 7

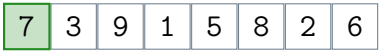


3 steps

Linear: $O(n)$

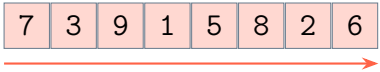
```
1 def find(L, x):
2     for item in L:
3         if item == x:
4             return True
5     return False
```

best case: x = 7



3 steps

worst case: x = 4, not there

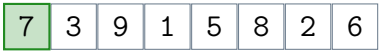


$2n + 1$ **steps**

Linear: $O(n)$

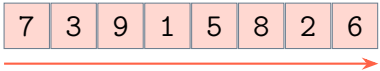
```
1  def find(L, x):
2      for item in L:
3          if item == x:
4              return True
5      return False
```

best case: `x = 7`



3 steps

worst case: `x = 4`, not there



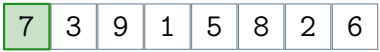
$2n + 1$ **steps**

plan for the **worst case**: $O(n)$

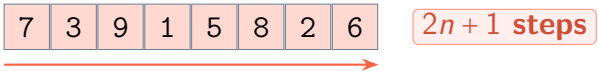
Linear: $O(n)$

```
1  def find(L, x):
2      for item in L:
3          if item == x:
4              return True
5      return False
```

best case: x = 7



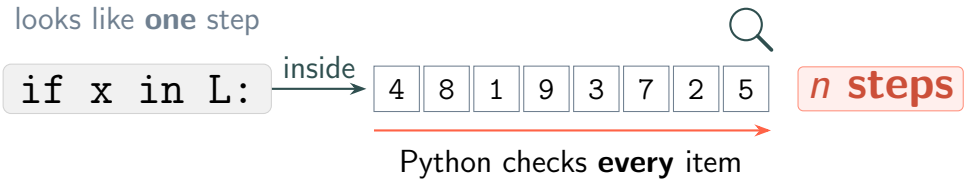
worst case: x = 4, not there



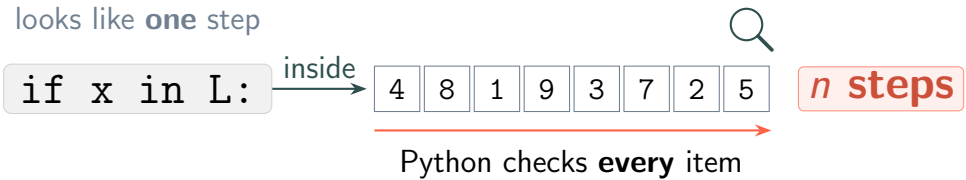
plan for the **worst case**: $O(n)$

data $\times 2$ → steps $\times 2$

Hidden Loops



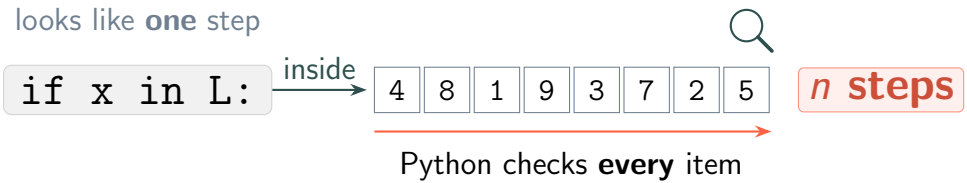
Hidden Loops



Same story:

L.index(x)	L.remove(x)	max(L)	sum(L)	L[:]	L.copy()
n	n	n	n	n	n

Hidden Loops



Same story:

`L.index(x)` n `L.remove(x)` n `max(L)` n `sum(L)` n `L[:]` n `L.copy()` n

Lecture 17: "`.index()` is slow." **Now you know why.**

But `for x in L:` is no search: its `in` just hands out the next item.

Constant: $O(1)$

100 billion pages, one by one: 3 hours. **Can we do better?**



Constant: $O(1)$

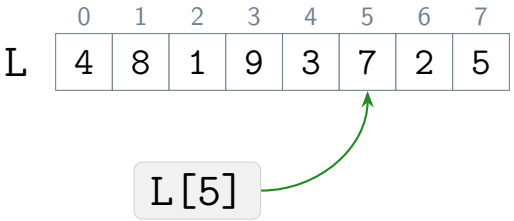
100 billion pages, one by one: 3 hours. **Can we do better?**



Python keeps it **written down**

Constant: $O(1)$

100 billion pages, one by one: 3 hours. **Can we do better?**



Constant: $O(1)$

100 billion pages, one by one: 3 hours. **Can we do better?**



`len = 8`

Python keeps it **written down**

`L[5]`

jumps straight there

`L[i]` 1 `len(L)` 1 `L.append(x)` 1 `d[key]` 1 `key in d` 1

Constant: $O(1)$

100 billion pages, one by one: 3 hours. **Can we do better?**



`len = 8`

Python keeps it **written down**

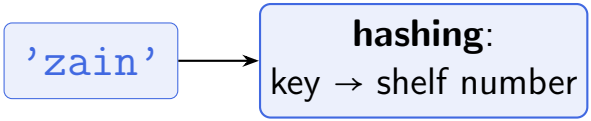
`L[5]`

jumps straight there

`L[i]` 1 `len(L)` 1 `L.append(x)` 1 `d[key]` 1 `key in d` 1

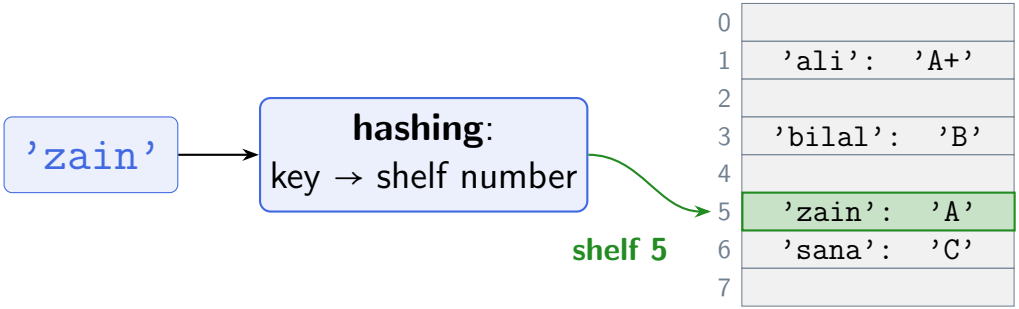
data ×2 → steps **stay the same**

How Can a Dictionary Do That?

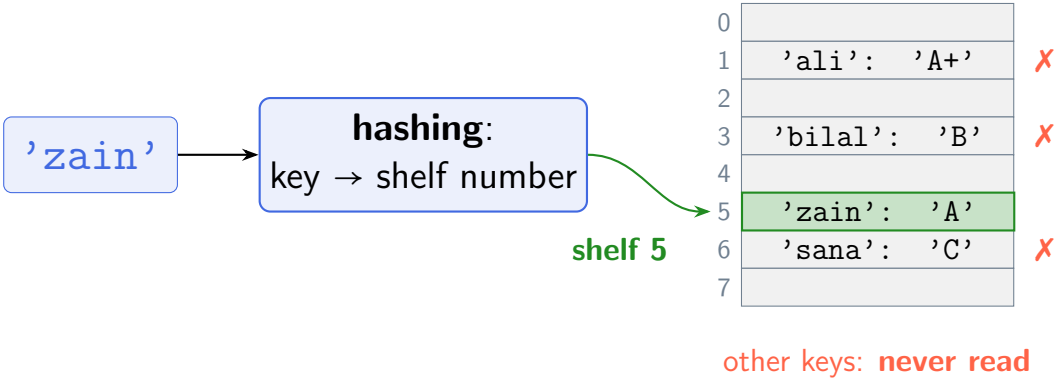


0	
1	'ali': 'A+'
2	
3	'bilal': 'B'
4	
5	
6	'sana': 'C'
7	

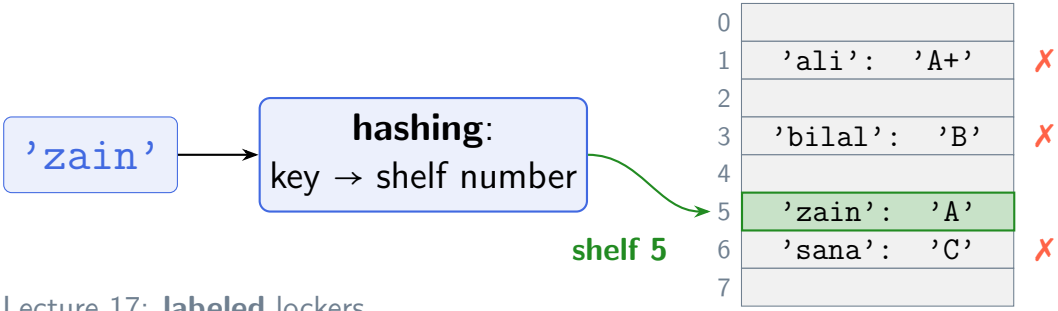
How Can a Dictionary Do That?



How Can a Dictionary Do That?



How Can a Dictionary Do That?



Lecture 17: **labeled** lockers.
Today: **fast** lockers.

You Try! *(write O(1) or O(n), 1 min)*

L is a list of *n* items. d is a dictionary with *n* keys.

```
1 len(L)
2 max(L)
3 L[0]
4 x in L
5 x in d
6 L.append(5)
```

You Try! *(write O(1) or O(n), 1 min)*

L is a list of *n* items. d is a dictionary with *n* keys.

```
1 len(L)                # O(1)
2 max(L)
3 L[0]
4 x in L
5 x in d
6 L.append(5)
```

You Try! *(write O(1) or O(n), 1 min)*

L is a list of *n* items. d is a dictionary with *n* keys.

```
1 len(L)                # O(1)
2 max(L)                # O(n)
3 L[0]
4 x in L
5 x in d
6 L.append(5)
```

You Try! *(write O(1) or O(n), 1 min)*

L is a list of *n* items. d is a dictionary with *n* keys.

```
1 len(L)                # O(1)
2 max(L)                # O(n)
3 L[0]                  # O(1)
4 x in L
5 x in d
6 L.append(5)
```

You Try! *(write O(1) or O(n), 1 min)*

L is a list of *n* items. d is a dictionary with *n* keys.

```
1 len(L)          # O(1)
2 max(L)          # O(n)
3 L[0]            # O(1)
4 x in L          # O(n)
5 x in d
6 L.append(5)
```

You Try! *(write O(1) or O(n), 1 min)*

L is a list of *n* items. d is a dictionary with *n* keys.

```
1 len(L)                # O(1)
2 max(L)                # O(n)
3 L[0]                  # O(1)
4 x in L                # O(n)
5 x in d                # O(1)
6 L.append(5)
```

You Try! *(write O(1) or O(n), 1 min)*

`L` is a list of n items. `d` is a dictionary with n keys.

```
1 len(L)          # O(1)
2 max(L)          # O(n)
3 L[0]            # O(1)
4 x in L          # O(n)
5 x in d          # O(1)
6 L.append(5)     # O(1)
```

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

1,000,000 numbers. Look for a number that is **not there**, 1,000 times. Which is faster? By how much?

```
x in L
```
list

```
x in d
```
dictionary

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

1,000,000 numbers. Look for a number that is **not there**, 1,000 times. Which is faster? By how much?

`x in L` (list)  **about 3 s**

`x in d` (dictionary) **0.0001 s or less**

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

1,000,000 numbers. Look for a number that is **not there**, 1,000 times. Which is faster? By how much?

`x in L` (list)  **about 3 s**

`x in d` (dictionary) **0.0001 s or less**

Same answer. More than **10,000 times** faster.

(My laptop. Run `demo_list_vs_dict()` in `lec18.py` to see yours.)

Logarithmic: $O(\log n)$

Lecture 15: on a **sorted** list, each look throws away **half**.



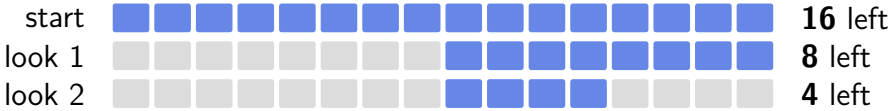
Logarithmic: $O(\log n)$

Lecture 15: on a **sorted** list, each look throws away **half**.



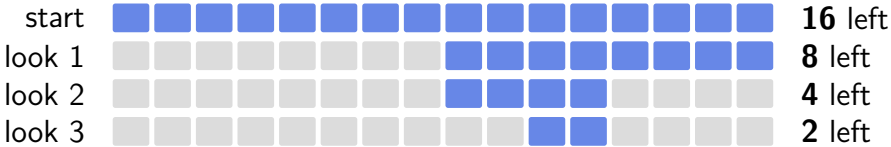
Logarithmic: $O(\log n)$

Lecture 15: on a **sorted** list, each look throws away **half**.



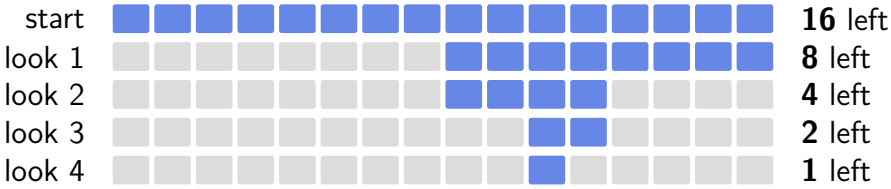
Logarithmic: $O(\log n)$

Lecture 15: on a **sorted** list, each look throws away **half**.



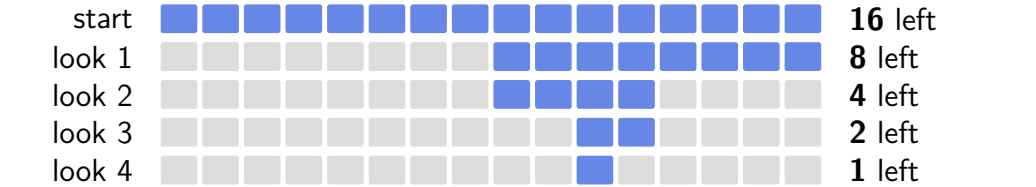
Logarithmic: $O(\log n)$

Lecture 15: on a **sorted** list, each look throws away **half**.



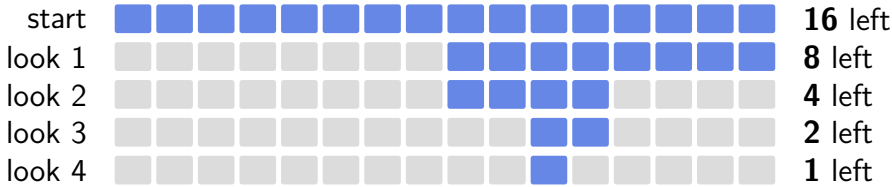
Logarithmic: $O(\log n)$

Lecture 15: on a **sorted** list, each look throws away **half**.



Logarithmic: $O(\log n)$

Lecture 15: on a **sorted** list, each look throws away **half**.



16 items: **4** looks —(×2, +1)→ 32 items: **5** looks —(×2, +1)→ 64 items: **6** looks

data ×2 → steps **just** +1

Bisection Search on a Sorted List

```
1  def find_sorted(L, x):
2      low = 0
3      high = len(L) - 1
4      while low <= high:
5          mid = (low + high) // 2
6          if L[mid] == x:
7              return True
8          elif L[mid] < x:
9              low = mid + 1
10         else:
11             high = mid - 1
12     return False
```

Bisection Search on a Sorted List

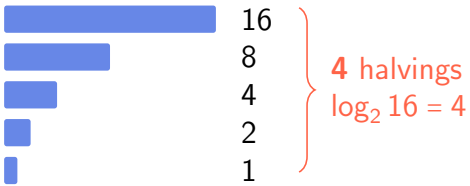
```
1  def find_sorted(L, x):
2      low = 0
3      high = len(L) - 1
4      while low <= high:
5          mid = (low + high) // 2
6          if L[mid] == x:
7              return True
8          elif L[mid] < x:
9              low = mid + 1
10         else:
11             high = mid - 1
12     return False
```

one pass ≈ **5** steps

Bisection Search on a Sorted List

```
1  def find_sorted(L, x):
2      low = 0
3      high = len(L) - 1
4      while low <= high:
5          mid = (low + high) // 2
6          if L[mid] == x:
7              return True
8          elif L[mid] < x:
9              low = mid + 1
10         else:
11             high = mid - 1
12     return False
```

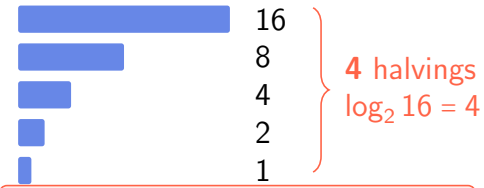
one pass ≈ **5** steps



Bisection Search on a Sorted List

```
1  def find_sorted(L, x):
2      low = 0
3      high = len(L) - 1
4      while low <= high:
5          mid = (low + high) // 2
6          if L[mid] == x:
7              return True
8          elif L[mid] < x:
9              low = mid + 1
10         else:
11             high = mid - 1
12     return False
```

one pass ≈ **5** steps

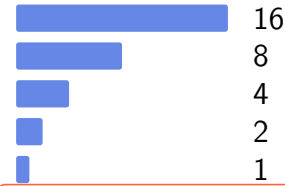


$\log_2 n$ = how many times you can **halve** n before you reach 1

Bisection Search on a Sorted List

```
1  def find_sorted(L, x):
2      low = 0
3      high = len(L) - 1
4      while low <= high:
5          mid = (low + high) // 2
6          if L[mid] == x:
7              return True
8          elif L[mid] < x:
9              low = mid + 1
10         else:
11             high = mid - 1
12     return False
```

one pass $\approx$ **5** steps



$\log_2 n$ = how many times you can **halve** n before you reach 1

steps $\approx 5 \log_2 n \rightarrow \mathbf{O(\log n)}$

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

Hint: $2^{10} = 1{,}024 \approx 1{,}000$ so 1,000 sorted items need about **10** looks.

sorted items	looks (worst case)
1,000	about 10
1,000,000	
1,000,000,000	
100,000,000,000 *(Google)*	

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

Hint: $2^{10} = 1{,}024 \approx 1{,}000$ so 1,000 sorted items need about **10** looks.

sorted items	looks (worst case)
1,000	about 10
1,000,000	about 20
1,000,000,000	
100,000,000,000 *(Google)*	

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

Hint: $2^{10} = 1{,}024 \approx 1{,}000$ so 1,000 sorted items need about **10** looks.

sorted items	looks (worst case)
1,000	about 10
1,000,000	about 20
1,000,000,000	about 30
100,000,000,000 *(Google)*	

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

Hint: $2^{10} = 1{,}024 \approx 1{,}000$ so 1,000 sorted items need about **10** looks.

sorted items	looks (worst case)
1,000	about 10
1,000,000	about 20
1,000,000,000	about 30
100,000,000,000 *(Google)*	about **37**

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

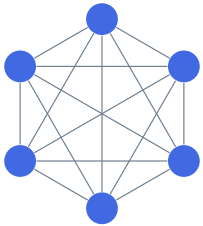
Hint: $2^{10} = 1{,}024 \approx 1{,}000$ so 1,000 sorted items need about **10** looks.

sorted items	**looks (worst case)**
1,000	about 10
1,000,000	about 20
1,000,000,000	about 30
100,000,000,000 *(Google)*	about **37**

37 looks, not 100 billion! But only if the list is **sorted**.

Quadratic: $O(n^2)$

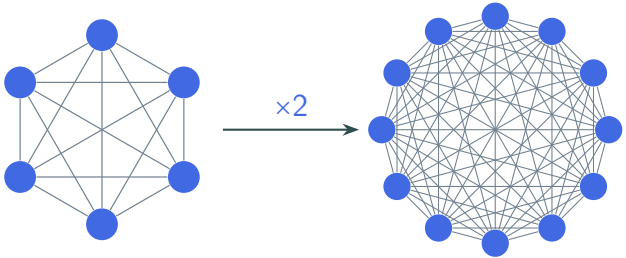
Everyone shakes hands with everyone, once.



6 people: **15**

Quadratic: $O(n^2)$

Everyone shakes hands with everyone, once.

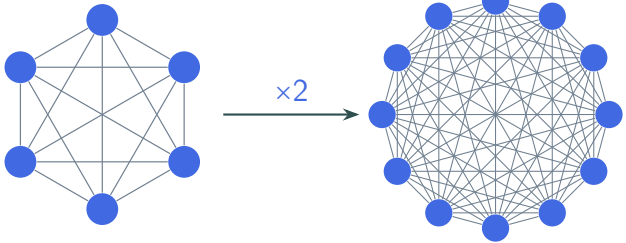


6 people: **15**

12 people: **66**

Quadratic: $O(n^2)$

Everyone shakes hands with everyone, once.



6 people: **15**

12 people: **66**

$$\text{handshakes} = \frac{n(n-1)}{2}$$

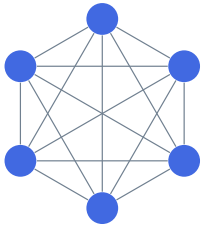
60 students: **1,770**

120 students: **7,140**

about ×4

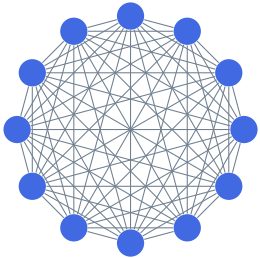
Quadratic: $O(n^2)$

Everyone shakes hands with everyone, once.



6 people: **15**





12 people: **66**

$$\text{handshakes} = \frac{n(n-1)}{2}$$

60 students: **1,770**

120 students: **7,140**

about ×4

In code: a **loop inside a loop** over the same data.

data ×2 → steps ×4

You Try! *(read and explain, 2 min)*

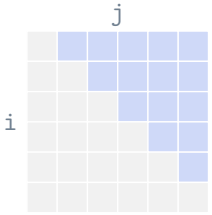
(a) In one sentence: what does `mystery` do? **(b)** What is its Big-O?

```
1 def mystery(L):
2     for i in range(len(L)):
3         for j in range(i + 1, len(L)):
4             if L[i] == L[j]:
5                 return True
6     return False
```

You Try! *(read and explain, 2 min)*

(a) In one sentence: what does `mystery` do? **(b)** What is its Big-O?

```
1  def mystery(L):
2      for i in range(len(L)):
3          for j in range(i + 1, len(L)):
4              if L[i] == L[j]:
5                  return True
6      return False
```



every **pair** once:
$n(n-1)/2$

(a) It returns `True` if some value is in `L` **twice**: it looks for a **duplicate**.

(b) Worst case (no duplicate): every pair, like the handshakes $\rightarrow$ $O(n^2)$

Exponential: $O(2^n)$, $O(10^n)$, ...

Lecture 15: guess a PIN when the only answer is **YES** or **NO**.

```
1 def crack(pin, d):   # d digits
2     for guess in range(10 ** d):
3         if guess == pin:
4             return guess
```

Exponential: $O(2^n)$, $O(10^n)$, ...

Lecture 15: guess a PIN when the only answer is **YES** or **NO**.

```
1 def crack(pin, d):    # d digits
2     for guess in range(10 ** d):
3         if guess == pin:
4             return guess
```

d digits $\rightarrow 10^d$ guesses

one more digit: $\times 10$

Exponential: $O(2^n)$, $O(10^n)$, ...

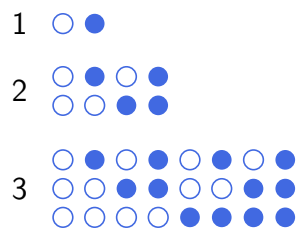
Lecture 15: guess a PIN when the only answer is **YES** or **NO**.

```
1  def crack(pin, d):    # d digits
2      for guess in range(10 ** d):
3          if guess == pin:
4              return guess
```

d digits $\rightarrow 10^d$ guesses

one more digit: $\times 10$

on/off switches



one more switch: $\times 2$

Exponential: $O(2^n)$, $O(10^n)$, ...

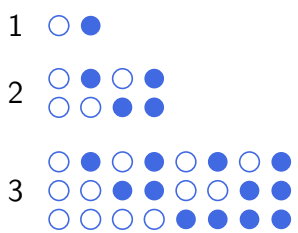
Lecture 15: guess a PIN when the only answer is **YES** or **NO**.

```
1 def crack(pin, d):   # d digits
2     for guess in range(10 ** d):
3         if guess == pin:
4             return guess
```

d digits $\rightarrow 10^d$ guesses

one more digit: $\times 10$

on/off switches



one more switch: $\times 2$

Add **one** item $\rightarrow$ the steps **multiply**

You Try! *(predict, 1 min)*

A computer tries **10 million** PINs each second. How long to try them all?

digits	PINs	time
4	10,000	0.001 s
6	1,000,000	0.1 s
8	100,000,000	
10	10,000,000,000	
12	10^{12}	
16	10^{16}	

You Try! *(predict, 1 min)*

A computer tries **10 million** PINs each second. How long to try them all?

digits	PINs	time
4	10,000	0.001 s
6	1,000,000	0.1 s
8	100,000,000	10 s
10	10,000,000,000	
12	10^{12}	
16	10^{16}	

You Try! *(predict, 1 min)*

A computer tries **10 million** PINs each second. How long to try them all?

digits	**PINs**	**time**
4	10,000	0.001 s
6	1,000,000	0.1 s
8	100,000,000	10 s
10	10,000,000,000	17 minutes
12	10^{12}	
16	10^{16}	

You Try! *(predict, 1 min)*

A computer tries **10 million** PINs each second. How long to try them all?

digits	PINs	time
4	10,000	0.001 s
6	1,000,000	0.1 s
8	100,000,000	10 s
10	10,000,000,000	17 minutes
12	10^{12}	28 hours
16	10^{16}	

You Try! *(predict, 1 min)*

A computer tries **10 million** PINs each second. How long to try them all?

digits	**PINs**	**time**
4	10,000	0.001 s
6	1,000,000	0.1 s
8	100,000,000	10 s
10	10,000,000,000	17 minutes
12	10^{12}	28 hours
16	10^{16}	**32 years**

You Try! *(predict, 1 min)*

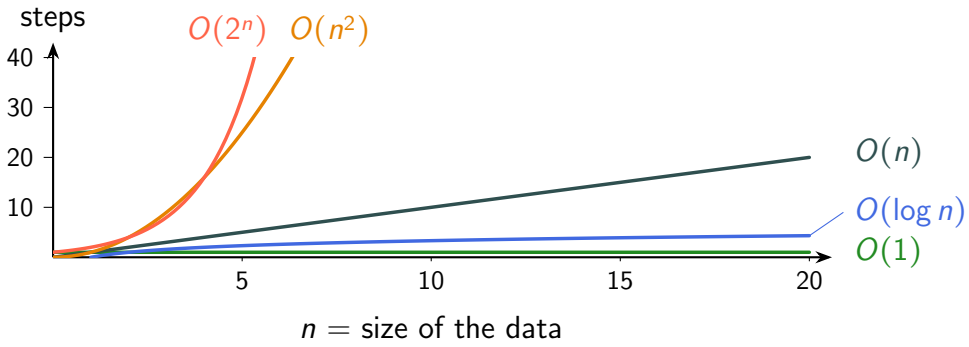
A computer tries **10 million** PINs each second. How long to try them all?

digits	**PINs**	**time**
4	10,000	0.001 s
6	1,000,000	0.1 s
8	100,000,000	10 s
10	10,000,000,000	17 minutes
12	10^{12}	28 hours
16	10^{16}	**32 years**

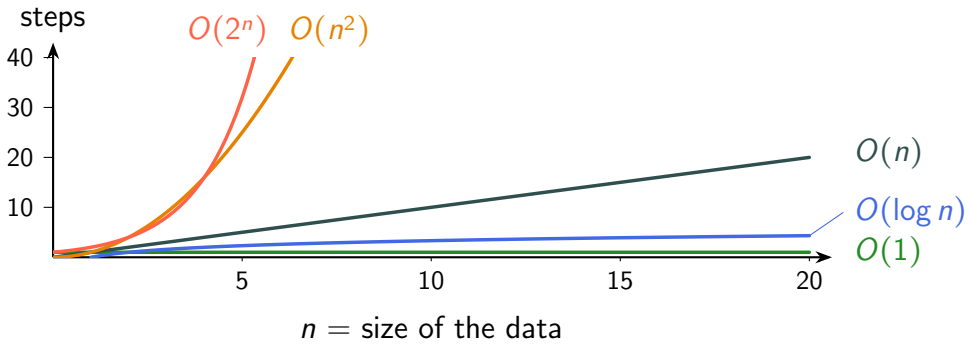
Here exponential growth is **your friend**: a long password is safe.

How Big Is Too Big?

Five Families, One Picture



Five Families, One Picture



For small n they all look close. Then n^2 and 2^n **shoot up**.

At 10 Million Steps per Second

n	$O(\log n)$	$O(n)$	$O(n^2)$	$O(2^n)$
10	instant	instant	instant	instant
1,000	instant	instant	0.1 s	**never ends**
1,000,000	instant	0.1 s	28 hours	**never ends**
100 billion	37 steps	3 hours	32 million years	**never ends**

At 10 Million Steps per Second

n	$O(\log n)$	$O(n)$	$O(n^2)$	$O(2^n)$
10	instant	instant	instant	instant
1,000	instant	instant	0.1 s	**never ends**
1,000,000	instant	0.1 s	28 hours	**never ends**
100 billion	37 steps	3 hours	32 million years	**never ends**

Small n: **everything** is fast. Big n: only the **family** matters.

At 10 Million Steps per Second

n	$O(\log n)$	$O(n)$	$O(n^2)$	$O(2^n)$
10	instant	instant	instant	instant
1,000	instant	instant	0.1 s	**never ends**
1,000,000	instant	0.1 s	28 hours	**never ends**
100 billion	37 steps	3 hours	32 million years	**never ends**

Small n: **everything** is fast. Big n: only the **family** matters.

"never ends": 2^{100} steps $\approx$ 4,000,000,000,000,000 years. The universe: 14 billion.

Buy a 1,000 Times Faster Computer?

The biggest n you can finish in **one hour**:

	today	**1,000× faster**
$O(n)$	36 billion	
$O(n^2)$	190,000	
$O(2^n)$	35	

Buy a 1,000 Times Faster Computer?

The biggest n you can finish in **one hour**:

	today		**1,000× faster**	
$O(n)$	36 billion	→	36 trillion	×1,000
$O(n^2)$	190,000			
$O(2^n)$	35			

Buy a 1,000 Times Faster Computer?

The biggest n you can finish in **one hour**:

	today		**1,000× faster**	
$O(n)$	36 billion	→	36 trillion	×1,000
$O(n^2)$	190,000	→	6,000,000	×32
$O(2^n)$	35			

Buy a 1,000 Times Faster Computer?

The biggest n you can finish in **one hour**:

	today		**1,000× faster**	
$O(n)$	36 billion	→	36 trillion	×1,000
$O(n^2)$	190,000	→	6,000,000	×32
$O(2^n)$	35	→	45	**just +10!**

Buy a 1,000 Times Faster Computer?

The biggest n you can finish in **one hour**:

	today		**1,000× faster**	
$O(n)$	36 billion	→	36 trillion	×1,000
$O(n^2)$	190,000	→	6,000,000	×32
$O(2^n)$	35	→	45	**just +10!**

A better **family** beats a faster **computer**.

Hunting the Slow Line

You Try! *(find the slow line, 1 min)*

Who is in **both** lists? The answer is right, but it is **slow**.

```
1 def common(A, B):
2     result = []
3     for x in A:
4         if x in B:
5             result.append(x)
6     return result
```

(a) Which line is slow? **(b)** What is the Big-O? (`A`, `B`: n names each)

You Try! *(find the slow line, 1 min)*

Who is in **both** lists? The answer is right, but it is **slow**.

```
1  def common(A, B):
2      result = []
3      for x in A:
4          if x in B:                    hidden loop:  n
5              result.append(x)
6      return result
```

(a) Which line is slow? **(b)** What is the Big-O? (`A`, `B`: n names each)

You Try! *(find the slow line, 1 min)*

Who is in **both** lists? The answer is right, but it is **slow**.

```
1  def common(A, B):
2      result = []
3      for x in A:                    runs n times
4          if x in B:                 hidden loop: n
5              result.append(x)
6      return result
```

(a) Which line is slow? **(b)** What is the Big-O? (`A`, `B`: n names each)

You Try! *(find the slow line, 1 min)*

Who is in **both** lists? The answer is right, but it is **slow**.

```
1  def common(A, B):
2      result = []
3      for x in A:                    runs n times
4          if x in B:                 hidden loop:  n
5              result.append(x)
6      return result
```

(a) Which line is slow? **(b)** What is the Big-O? (`A`, `B`: n names each)

line 4 runs n times, n steps each: $n \times n \rightarrow \mathbf{O(n^2)}$

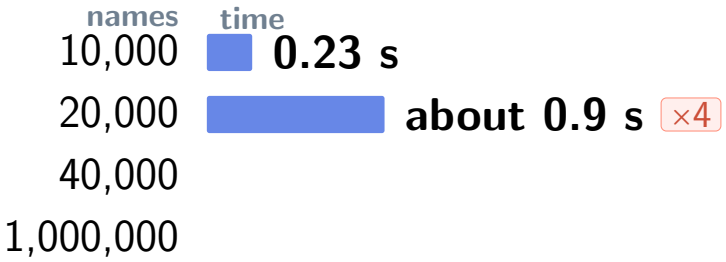
You Try! *(predict, then I run it)*

`common(A, B)` on my laptop. Predict the missing times.

names	time
10,000	**0.23 s**
20,000	
40,000	
1,000,000	

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

`common(A, B)` on my laptop. Predict the missing times.



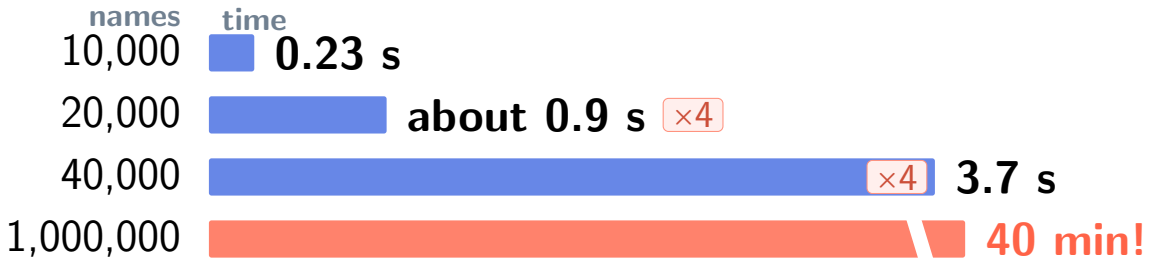
You Try! *(predict, then I run it)*

`common(A, B)` on my laptop. Predict the missing times.



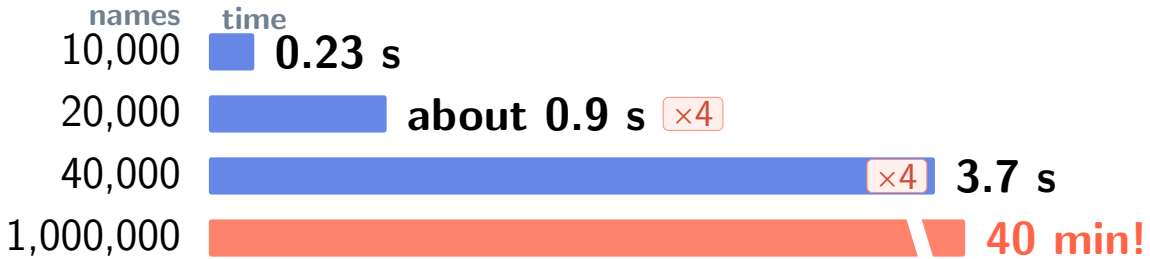
You Try! *(predict, then I run it)*

`common(A, B)` on my laptop. Predict the missing times.



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

`common(A, B)` on my laptop. Predict the missing times.



Data ×2 → time ×4, just as $O(n^2)$ says. *(Run `demo_common()`.)*

The Fix: Ask a Dictionary

```
1  def common_fast(A, B):
2      seen = {}
3      for x in B:
4          seen[x] = True
5      result = []
6      for x in A:
7          if x in seen:
8              result.append(x)
9      return result
```

The Fix: Ask a Dictionary

```
1  def common_fast(A, B):
2      seen = {}
3      for x in B:          n, once
4          seen[x] = True
5      result = []
6      for x in A:
7          if x in seen:
8              result.append(x)
9      return result
```

The Fix: Ask a Dictionary

```
1  def common_fast(A, B):
2      seen = {}
3      for x in B:          n, once
4          seen[x] = True
5      result = []
6      for x in A:          n
7          if x in seen:    1 step!
8              result.append(x)
9      return result
```

The Fix: Ask a Dictionary

```
1  def common_fast(A, B):
2      seen = {}
3      for x in B:          # n, once
4          seen[x] = True
5      result = []
6      for x in A:          # n
7          if x in seen:    # 1 step!
8              result.append(x)
9      return result
```

$$n + n \rightarrow \mathbf{O(n)}$$

The Fix: Ask a Dictionary

```
1  def common_fast(A, B):
2      seen = {}
3      for x in B:          # n, once
4          seen[x] = True
5      result = []
6      for x in A:          # n
7          if x in seen:    # 1 step!
8              result.append(x)
9      return result
```

$$n + n \rightarrow \mathbf{O(n)}$$

40,000 names



3.7 s

common

0.0014 s

common_fast

The Fix: Ask a Dictionary

```
1  def common_fast(A, B):
2      seen = {}
3      for x in B:              n, once
4          seen[x] = True
5      result = []
6      for x in A:              n
7          if x in seen:        1 step!
8              result.append(x)
9      return result
```

$n + n \rightarrow \mathbf{O(n)}$

40,000 names



common

0.0014 s

common_fast

Pay **once** to build a dictionary. Then every question is **one step**.

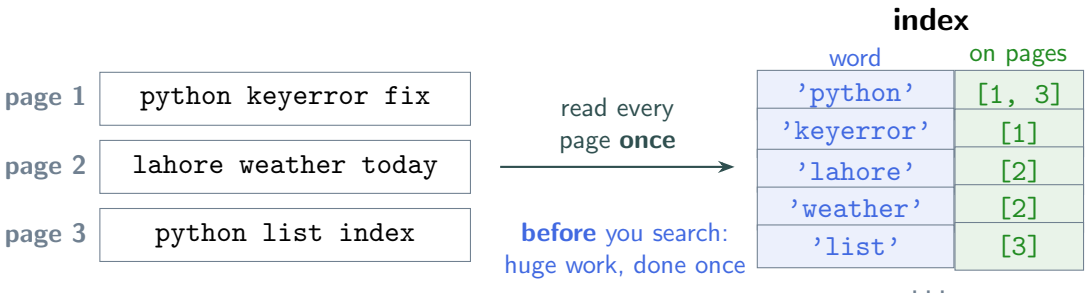
Google's Answer: The Index

Like the **index at the back of a book**: word → pages.



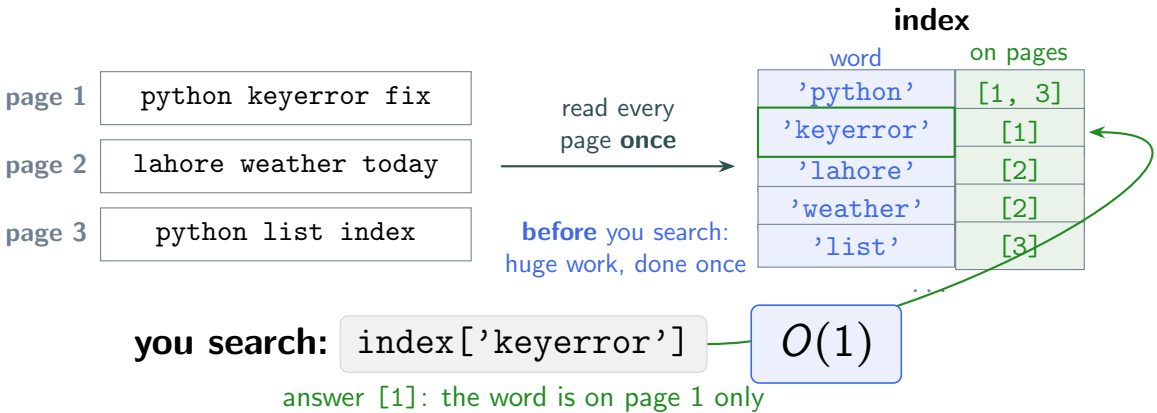
Google's Answer: The Index

Like the **index at the back of a book**: word → pages.



Google's Answer: The Index

Like the **index at the back of a book**: word → pages.



Build the Index

```
pages = {1: 'python keyerror fix',
         2: 'lahore weather today',
         3: 'python list index'}

def build_index(pages):
    index = {}
    for num, text in pages.items():
        for word in text.split():
            if word in index:
                index[word].append(num)
            else:
                index[word] = [num]
    return index

index = build_index(pages)



print(index['python'])      # [1, 3]
```

Build the Index

```
pages = {1: 'python keyerror fix',
         2: 'lahore weather today',
         3: 'python list index'}

def build_index(pages):
    index = {}
    for num, text in pages.items():
        for word in text.split():
            if word in index:
                index[word].append(num)
            else:
                index[word] = [num]
    return index

index = build_index(pages)
# index = {'python': [1, 3], 'keyerror': [1],
#   'fix': [1], 'lahore': [2], 'weather': [2],
#   'today': [2], 'list': [3], 'index': [3]}
print(index['python'])      # [1, 3]
```

Build the Index

```
1  pages = {1: 'python keyerror fix',
2           2: 'lahore weather today',
3           3: 'python list index'}
4
5  def build_index(pages):
6      index = {}
7      for num, text in pages.items():
8          for word in text.split():
9              if word in index:
10                 index[word].append(num)
11             else:
12                 index[word] = [num]
13     return index
14
15 index = build_index(pages)
16 # index = {'python': [1, 3], 'keyerror': [1],
17 #   'fix': [1], 'lahore': [2], 'weather': [2],
18 #   'today': [2], 'list': [3], 'index': [3]}
19 print(index['python'])      # [1, 3]
```

Lecture 17: count

```
count[w] = count[w] + 1
```

today: same pattern

```
index[w].append(num)
```

Build the Index

```
1  pages = {1: 'python keyerror fix',
2           2: 'lahore weather today',
3           3: 'python list index'}
4
5  def build_index(pages):
6      index = {}
7      for num, text in pages.items():
8          for word in text.split():
9              if word in index:
10                 index[word].append(num)
11             else:
12                 index[word] = [num]
13     return index
14
15 index = build_index(pages)
16 # index = {'python': [1, 3], 'keyerror': [1],
17 #   'fix': [1], 'lahore': [2], 'weather': [2],
18 #   'today': [2], 'list': [3], 'index': [3]}
19 print(index['python'])      # [1, 3]
```

Lecture 17: count

```
count[w] = count[w] + 1
```

today: same pattern

```
index[w].append(num)
```

real Google adds:
ranking and **many computers**

Build the Index

```
1  pages = {1: 'python keyerror fix',
2           2: 'lahore weather today',
3           3: 'python list index'}
4
5  def build_index(pages):
6      index = {}
7      for num, text in pages.items():
8          for word in text.split():
9              if word in index:
10                 index[word].append(num)
11             else:
12                 index[word] = [num]
13     return index
14
15 index = build_index(pages)
16 # index = {'python': [1, 3], 'keyerror': [1],
17 #   'fix': [1], 'lahore': [2], 'weather': [2],
18 #   'today': [2], 'list': [3], 'index': [3]}
19 print(index['python'])      # [1, 3]
```

Lecture 17: count

```
count[w] = count[w] + 1
```

today: same pattern

```
index[w].append(num)
```

real Google adds:
ranking and **many computers**

the heart: the **index**

The Cheat Card

	Big-O	name	example	data $\times 2 \rightarrow$ steps
	$O(1)$	constant	`d[key]`, `L[i]`, `len(L)`	the same
	$O(\log n)$	logarithmic	bisection search	$+1$
	$O(n)$	linear	one loop, `if x in L`	$\times 2$
	$O(n^2)$	quadratic	a loop inside a loop	$\times 4$
	$O(2^n)$	exponential	try every pattern	one more item: $\times 2$

keep the biggest part, drop the number in front

watch for **hidden loops**

Big Idea

Count **steps**, not seconds.

A better **family** beats a faster computer.

Google is fast not because it reads fast,
but because it reads **almost nothing**.

Next Time: Sorting



then bisection: $O(\log n)$

Next Time: Sorting



how many steps?

then bisection: $O(\log n)$

Next Time: Sorting



?

Is there a **speed limit**
that no sorting program can beat?

Questions?

Exit Ticket

Today's question *(2 minutes)*

```
for x in A:
    if x in B:
        print(x)
```

`A` has n items. Give the Big-O and say **why**:

(a) `B` is a **list** of n items.

(b) `B` is a **dictionary** with n keys.

COMP 102 — Exit Ticket #18

Name: _____

Roll #: _____ Date: _____

Answer:

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