

# SOFTWARE FUNDAMENTALS

Yogesh Kulkarni

November 27, 2021

# Data Structures

# Lists

# Lists: the most basic data structure?

## Afleveradres

Dhr. Stefan Hugtenburg



## Artikelen

Artikel	Prijs	Aantal	Totaal
	€ 1,35	1	€ 1,35
	€ 1,35	1	€ 1,35
	€ 1,35	1	€ 1,35
	€ 1,35	1	€ 1,35
	€ 1,35	1	€ 1,35
	€ 1,35	1	€ 1,35
	€ 1,35	1	€ 1,35
	€ 1,35	1	€ 1,35
	€ 1,35	1	€ 1,35
	€ 1,35	1	€ 1,35
	€ 1,35	1	€ 1,35

## Why study the list?

### USE CASES

What do we use lists for?

MANY THINGS!

- ▶ To store a collection of data.
- ▶ To build other more complex/refined data structures!

## Why study the list?

### USE CASES

What do we use lists for?

MANY THINGS!

- ▶ To store a collection of data.
- ▶ To build other more complex/refined data structures!

## Why study the list?

### USE CASES

What do we use lists for?

MANY THINGS!

- ▶ To store a collection of data.
- ▶ To build other more complex/refined data structures!

# Lists in Python

## LISTS IN PYTHON

How do lists in Python work?

## ARRAYS

They are array-based!

## ARRAYS?

So what's an array then?

# Lists in Python

## LISTS IN PYTHON

How do lists in Python work?

## ARRAYS

They are array-based!

## ARRAYS?

So what's an array then?

# Lists in Python

## LISTS IN PYTHON

How do lists in Python work?

## ARRAYS

They are array-based!

## ARRAYS?

So what's an array then?

# Arrays

## ARRAY

An array is a block of memory of fixed size that can hold multiple items of data.

# Arrays

## ARRAY

An array is a block of memory of fixed size that can hold multiple items of data.

# Arrays

## ARRAY

An array is a block of memory of fixed size that can hold multiple items of data.

```
1 print(a[0]) # Prints 1
```

src/array.py

# Arrays

## ARRAY

An array is a block of memory of fixed size that can hold multiple items of data.

```
1 print(a[0]) # Prints 1  
print(a[3]+a[1]) # Prints 209
```

src/array2.py

# Arrays

## ARRAY

An array is a block of memory of fixed size that can hold multiple items of data.

```
print(a[0]) # Prints 1  
2 print(a[3]+a[1]) # Prints 209  
a[4] = 4242
```

src/array3.py

# Arrays

## ARRAY

An array is a block of memory of fixed size that can hold multiple items of data.

```
print(a[0])  # Prints 1  
2 print(a[3]+a[1])  # Prints 209  
a[4] = 4242
```

src/array3.py

## HOW IS THIS DIFFERENT?

How does this differ from a list?

# Arrays

## ARRAY

An array is a block of memory of fixed size that can hold multiple items of data.

```
print(a[0])  # Prints 1  
2 print(a[3]+a[1])  # Prints 209  
a[4] = 4242
```

src/array3.py

## SEVERAL WAYS

- ▶ It's just memory, no things like `a.sort()`.
- ▶ It is *finite!*

## Array-based lists

### ARRAY-BASED LIST

An array-based list (like the `list` in Python) uses an array internally.

### GROWING A LIST?

How can we then grow the list (seemingly) infinitely?

- A. The list uses multiple arrays stitched together.
- B. The list also has a finite size from the start, we just never notice.
- C. The list creates a new array of size  $n + 1$  when the array if full.
- D. The list creates a new array of size  $n * 2$  when the array if full.
- E. I don't know.

# No! Bad Frankenstein!

## STITCHING ARRAYS TOGETHER

The list uses multiple arrays stitched together.

- ▶ Although technically possible, keeping track of all of it is hell not pleasant.
- ▶ There are also benefits to having one continuous block of memory.
  - ▶ For instance spacial caching benefits (Google it, if you are intrigued ;))
- ▶ But the idea isn't a bad one per se. Having only single items blocks, forms the basis of the list we study after the break!

# No! Bad Frankenstein!

## STITCHING ARRAYS TOGETHER

The list uses multiple arrays stitched together.

- ▶ Although technically possible, keeping track of all of it is hell not pleasant.
- ▶ There are also benefits to having one continuous block of memory.
  - ▶ For instance spacial caching benefits (Google it, if you are intrigued ;))
- ▶ But the idea isn't a bad one per se. Having only single items blocks, forms the basis of the list we study after the break!

## Too infinity and beyond

### A HIDDEN MAXIMUM SIZE?

The list also has a finite size from the start, we just never notice.

- ▶ Nope, we can grow it so long as there is memory available.

So... New arrays then?

A NEW ONE!

When the initial array is full, we create a new one with more capacity.  
We copy over all existing elements into the new array...  
And we now have new space to grow!

#### QUESTION

But by how much should we grow?

#### OBSERVATIONS

- ▶ Adding one item, can trigger a full copy of the array...
- ▶ Does that make append an  $O(n)$  operation?

So... New arrays then?

### A NEW ONE!

When the initial array is full, we create a new one with more capacity.  
We copy over all existing elements into the new array...  
And we now have new space to grow!

### QUESTION

But by how much should we grow?

### OBSERVATIONS

- ▶ Adding one item, can trigger a full copy of the array...
- ▶ Does that make append an  $O(n)$  operation?

So... New arrays then?

A NEW ONE!

When the initial array is full, we create a new one with more capacity.  
We copy over all existing elements into the new array...  
And we now have new space to grow!

QUESTION

But by how much should we grow?

OBSERVATIONS

- ▶ Adding one item, can trigger a full copy of the array...
- ▶ Does that make append an  $O(n)$  operation?

# It depends

## JUST ENOUGH ROOM

The list creates a new array of size  $n + 1$  when the array if full.

## DOING IT OFTEN

What happens when we add  $n$  elements to a list of size 1?

- ▶ Every time we need to copy the full list.
- ▶ So  $O(\sum_{i=1}^n i)$  time in total.
- ▶ So  $O(n^2)$  to add  $n$  elements...

## It depends

### MORE THAN ENOUGH ROOM

The list creates a new array of size  $n * 2$  when the array if full.

### DOING IT OFTEN

What happens when we add  $n$  elements?

## Amortised run time

### AMORTISED RUN TIME

Some operations have varying run times, but can be shown to be efficient when repeated multiple times. We call this an amortised run time.

### CONSIDER...

- ▶ Consider a list of size 1 and we add  $n$  elements to it.
- ▶ This means we double the size of the list  $\log_2(n)$  times.
- ▶ This means that in total we have:  $O(n)$  time to add all elements.
- ▶ And  $O\left(\sum_{i=1}^{\log_2(n)} 2^i\right)$  operations to copy when the list grows.
- ▶ This geometric sequence gets us to:  $O(2^{\log_2(n)}) = O(n)$  operations.
- ▶ So  $O(n)$  to add  $n$  items!

We call this an amortised run time of  $O(1)$  for the append operation!

## Amortised run time

### AMORTISED RUN TIME

Some operations have varying run times, but can be shown to be efficient when repeated multiple times. We call this an amortised run time.

### CONSIDER...

- ▶ Consider a list of size 1 and we add  $n$  elements to it.
- ▶ This means we double the size of the list  $\log_2(n)$  times.
- ▶ This means that in total we have:  $O(n)$  time to add all elements.
- ▶ And  $O\left(\sum_{i=1}^{\log_2(n)} 2^i\right)$  operations to copy when the list grows.
- ▶ This geometric sequence gets us to:  $O(2^{\log_2(n)}) = O(n)$  operations.
- ▶ So  $O(n)$  to add  $n$  items!

We call this an amortised run time of  $O(1)$  for the append operation!

## Amortised run time

### AMORTISED RUN TIME

Some operations have varying run times, but can be shown to be efficient when repeated multiple times. We call this an amortised run time.

### CONSIDER...

- ▶ Consider a list of size 1 and we add  $n$  elements to it.
- ▶ This means we double the size of the list  $\log_2(n)$  times.
- ▶ This means that in total we have:  $O(n)$  time to add all elements.
- ▶ And  $O\left(\sum_{i=1}^{\log_2(n)} 2^i\right)$  operations to copy when the list grows.
- ▶ This geometric sequence gets us to:  $O(2^{\log_2(n)}) = O(n)$  operations.
- ▶ So  $O(n)$  to add  $n$  items!

We call this an amortised run time of  $O(1)$  for the append operation!

## Amortised run time

### AMORTISED RUN TIME

Some operations have varying run times, but can be shown to be efficient when repeated multiple times. We call this an amortised run time.

### CONSIDER...

- ▶ Consider a list of size 1 and we add  $n$  elements to it.
- ▶ This means we double the size of the list  $\log_2(n)$  times.
- ▶ This means that in total we have:  $O(n)$  time to add all elements.
- ▶ And  $O\left(\sum_{i=1}^{\log_2(n)} 2^i\right)$  operations to copy when the list grows.
- ▶ This geometric sequence gets us to:  $O(2^{\log_2(n)}) = O(n)$  operations.
- ▶ So  $O(n)$  to add  $n$  items!

We call this an amortised run time of  $O(1)$  for the append operation!

## Amortised run time

### AMORTISED RUN TIME

Some operations have varying run times, but can be shown to be efficient when repeated multiple times. We call this an amortised run time.

### CONSIDER...

- ▶ Consider a list of size 1 and we add  $n$  elements to it.
- ▶ This means we double the size of the list  $\log_2(n)$  times.
- ▶ This means that in total we have:  $O(n)$  time to add all elements.
- ▶ And  $O\left(\sum_{i=1}^{\log_2(n)} 2^i\right)$  operations to copy when the list grows.
- ▶ This geometric sequence gets us to:  $O(2^{\log_2(n)}) = O(n)$  operations.
- ▶ So  $O(n)$  to add  $n$  items!

We call this an amortised run time of  $O(1)$  for the append operation!

## Amortised run time

### AMORTISED RUN TIME

Some operations have varying run times, but can be shown to be efficient when repeated multiple times. We call this an amortised run time.

### CONSIDER...

- ▶ Consider a list of size 1 and we add  $n$  elements to it.
- ▶ This means we double the size of the list  $\log_2(n)$  times.
- ▶ This means that in total we have:  $O(n)$  time to add all elements.
- ▶ And  $O\left(\sum_{i=1}^{\log_2(n)} 2^i\right)$  operations to copy when the list grows.
- ▶ This geometric sequence gets us to:  $O(2^{\log_2(n)}) = O(n)$  operations.
- ▶ So  $O(n)$  to add  $n$  items!

We call this an amortised run time of  $O(1)$  for the append operation!

## Amortised run time

### AMORTISED RUN TIME

Some operations have varying run times, but can be shown to be efficient when repeated multiple times. We call this an amortised run time.

### CONSIDER...

- ▶ Consider a list of size 1 and we add  $n$  elements to it.
- ▶ This means we double the size of the list  $\log_2(n)$  times.
- ▶ This means that in total we have:  $O(n)$  time to add all elements.
- ▶ And  $O\left(\sum_{i=1}^{\log_2(n)} 2^i\right)$  operations to copy when the list grows.
- ▶ This geometric sequence gets us to:  $O(2^{\log_2(n)}) = O(n)$  operations.
- ▶ So  $O(n)$  to add  $n$  items!

We call this an amortised run time of  $O(1)$  for the append operation!

## Inserter

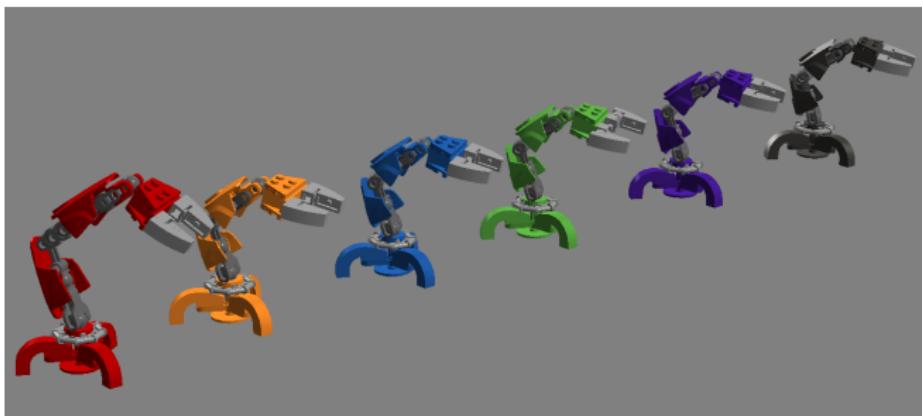


Image By: *TheMugbearer*

# Inserting

## LET'S INSERT

What is the time complexity of  
`l.insert(index,value)` when  
`len(l)=n`?

- A.  $O(1)$
- B.  $O(index)$
- C.  $O(n - index)$
- D.  $O(n)$
- E.  $O(n^2)$
- F. I don't know.

## LET'S INSERT

We need to shift all elements after  
index, so  $O(n - index)$

## INSERTING AT THE FRONT

This means prepending is  $O(n)$  for  
array-based lists!

# Inserting

## LET'S INSERT

What is the time complexity of  
`l.insert(index,value)` when  
`len(l)=n`?

- A.  $O(1)$
- B.  $O(index)$
- C.  $O(n - index)$
- D.  $O(n)$
- E.  $O(n^2)$
- F. I don't know.

## LET'S INSERT

We need to shift all elements after  
`index`, so  $O(n - index)$

## INSERTING AT THE FRONT

This means prepending is  $O(n)$  for  
array-based lists!

# Inserting

## LET'S INSERT

What is the time complexity of  
`l.insert(index,value)` when  
`len(l)=n`?

- A.  $O(1)$
- B.  $O(index)$
- C.  $O(n - index)$
- D.  $O(n)$
- E.  $O(n^2)$
- F. I don't know.

## LET'S INSERT

We need to shift all elements after  
index, so  $O(n - index)$

## INSERTING AT THE FRONT

This means prepending is  $O(n)$  for  
array-based lists!

## Getting rid of the trash



Image By: *Tgasser*

## Removing an item

### LET'S POP

What is the time complexity of  
`l.pop(index)` when `len(l)=n`?

- A.  $O(1)$
- B.  $O(index)$
- C.  $O(n - index)$
- D.  $O(n)$
- E.  $O(n^2)$
- F. I don't know.

### LET'S POP

We need to shift all elements after  
index, so  $O(n - index)$ .

### INSERTING AT THE FRONT

This means removing the first item  
is  $O(n)$  for array-based lists!

## Removing an item

### LET'S POP

What is the time complexity of  
`l.pop(index)` when `len(l)=n`?

- A.  $O(1)$
- B.  $O(index)$
- C.  $O(n - index)$
- D.  $O(n)$
- E.  $O(n^2)$
- F. I don't know.

### LET'S POP

We need to shift all elements after  
index, so  $O(n - index)$ .

### INSERTING AT THE FRONT

This means removing the first item  
is  $O(n)$  for array-based lists!

## Removing an item

### LET'S POP

What is the time complexity of  
l.pop(index) when len(l)=n?

- A.  $O(1)$
- B.  $O(index)$
- C.  $O(n - index)$
- D.  $O(n)$
- E.  $O(n^2)$
- F. I don't know.

### LET'S POP

We need to shift all elements after  
index, so  $O(n - index)$ .

### INSERTING AT THE FRONT

This means removing the first item  
is  $O(n)$  for array-based lists!

## Removing an item

### LET'S REMOVE

What is the time complexity of

`l.remove(value)` when

`len(l)=n`?

- A.  $O(1)$
- B.  $O(index)$ , where index is the index of the value.
- C.  $O(n - index)$ , where index is the index of the value.
- D.  $O(n)$
- E.  $O(n^2)$
- F. I don't know.

### LET'S REMOVE

We need to find the element so  $O(index)$ .

We need to shift all elements after index, so  $O(n - index)$ .

Together this is  $O(n)$ .

## Removing an item

### LET'S REMOVE

What is the time complexity of

`l.remove(value)` when

`len(l)=n`?

- A.  $O(1)$
- B.  $O(index)$ , where index is the index of the value.
- C.  $O(n - index)$ , where index is the index of the value.
- D.  $O(n)$
- E.  $O(n^2)$
- F. I don't know.

### LET'S REMOVE

We need to find the element so  $O(index)$ .

We need to shift all elements after index, so  $O(n - index)$ .

Together this is  $O(n)$ .

## Freeing up memory

### FREEING UP SPACE

When we remove sufficient items, we can free up space again.  
We do this when 25% of the capacity is used.

### WHY 25%?

Why not just when we drop below 50% again?

### THRASHING

Thrashing is repeatedly claiming and releasing memory (and in this case copying the array).

To avoid this, we use a different bound on when we release memory.

## Freeing up memory

### FREEING UP SPACE

When we remove sufficient items, we can free up space again.  
We do this when 25% of the capacity is used.

### WHY 25%?

Why not just when we drop below 50% again?

### THRASHING

Thrashing is repeatedly claiming and releasing memory (and in this case copying the array).

To avoid this, we use a different bound on when we release memory.

## Freeing up memory

### FREEING UP SPACE

When we remove sufficient items, we can free up space again.  
We do this when 25% of the capacity is used.

### WHY 25%?

Why not just when we drop below 50% again?

### THRASHING

Thrashing is repeatedly claiming and releasing memory (and in this case copying the array).

To avoid this, we use a different bound on when we release memory.

## Lists in Python

So to summarise:

- ▶ Insert first element:  $O(n)$ .
- ▶ Insert at index  $k$ :  $O(n - k)$ .
- ▶ Append: amortised  $O(1)$ .
- ▶ Remove first element:  $O(n)$ .
- ▶ Remove last element: amortised  $O(1)$ .
- ▶ Remove index  $k$ :  $O(n - k)$ .
- ▶ Search (discussed last week):  $O(n)$ .

## Linked Lists

## Linked list



Image By: *PiaCarrot*

## Another approach

### WHAT DO WE WANT TO SOLVE?

Two things in the array-based implementation that we hope to solve:

1. Only use space for items we actually use.
2. Allow for efficient ( $O(1)??$ ) adding and removing at the front of the list.

Why do we want this efficient adding/removing?

## Another approach

### WHAT DO WE WANT TO SOLVE?

Two things in the array-based implementation that we hope to solve:

1. Only use space for items we actually use.
2. Allow for efficient ( $O(1)??$ ) adding and removing at the front of the list.

Why do we want this efficient adding/removing?

## The notion of a linked list

- ▶ We build a list of blocks, starting with one item/block (we call this the **head**)
- ▶ We can then add another.
- ▶ These are connected, we can go from the first to the second item.
- ▶ So let's add one more item.
- ▶ We should also indicate when we have reached the end (we call this the **tail**).

## The notion of a linked list

- ▶ We build a list of blocks, starting with one item/block (we call this the **head**)
- ▶ We can then add another.
- ▶ These are connected, we can go from the first to the second item.
- ▶ So let's add one more item.
- ▶ We should also indicate when we have reached the end (we call this the **tail**).

## The notion of a linked list

- ▶ We build a list of blocks, starting with one item/block (we call this the **head**)
- ▶ We can then add another.
- ▶ These are connected, we can go from the first to the second item.
- ▶ So let's add one more item.
- ▶ We should also indicate when we have reached the end (we call this the **tail**).

## The notion of a linked list

- ▶ We build a list of blocks, starting with one item/block (we call this the **head**)
- ▶ We can then add another.
- ▶ These are connected, we can go from the first to the second item.
- ▶ So let's add one more item.
- ▶ We should also indicate when we have reached the end (we call this the **tail**).

## The notion of a linked list

- ▶ We build a list of blocks, starting with one item/block (we call this the **head**)
- ▶ We can then add another.
- ▶ These are connected, we can go from the first to the second item.
- ▶ So let's add one more item.
- ▶ We should also indicate when we have reached the end (we call this the **tail**).

## How does this work now?

Searching for an item

The approach:

1. Start at the head of the list.
2. If this is the item we need, return True.
3. Else if this is the tail, return False.
4. Else, move to the next item of the list and go to step 2.

## How does this work now?

Searching for an item

The approach:

1. Start at the head of the list.
2. If this is is the item we need, return True.
3. Else if this is the tail, return False.
4. Else, move to the next item of the list and go to step 2.

## How does this work now?

Searching for an item

The approach:

1. Start at the head of the list.
2. If this is the item we need, return True.
3. Else if this is the tail, return False.
4. Else, move to the next item of the list and go to step 2.

## How does this work now?

Searching for an item

The approach:

1. Start at the head of the list.
2. If this is the item we need, return True.
3. Else if this is the tail, return False.
4. Else, move to the next item of the list and go to step 2.

## How does this work now?

Searching for an item

The approach:

1. Start at the head of the list.
2. If this is the item we need, return True.
3. Else if this is the tail, return False.
4. Else, move to the next item of the list and go to step 2.

## How does this work now?

Searching for an item

The approach:

1. Start at the head of the list.
2. If this is is the item we need, return True.
3. Else if this is the tail, return False.
4. Else, move to the next item of the list and go to step 2.

```
1 class linkedList:  
2  
3     def contains(self, val: int) -> bool:  
4         cur = self.head  
5         while cur is not None:  
6             if cur.value == val:  
7                 return True  
8             cur = cur.next  
9         return False
```

src/ll\_search.py

## Getting an item!?

Getting item at index  $i$

- ▶ There is no easy way to access the  $i$ th item other than to 'walk' there.
- ▶ So  $O(index)$  to get the item at a certain index.

```
1 class LinkedList:  
2  
3     def get(self, index: int) -> int:  
4         if index >= self.size:  
5             return None # Invalid index  
6         cur = self.head  
7         cnt = 0  
8         while cnt < index:  
9             cur = cur.next  
10            cnt += 1  
11        return cur.value
```

src/ll.get.py

## Inserting at the head or tail

- ▶ Take the current head or tail.
- ▶ And put the new node before or after it :)
- ▶  $O(1)$  time!

## Inserting at the head or tail

- ▶ Take the current head or tail.
- ▶ And put the new node before or after it :)
- ▶  $O(1)$  time!

```
1 class LinkedList:  
2  
3     def add_first(self, val: int):  
4         if self.head is None:  
5             self.head = Node(val, None)  
6             self.tail = self.head  
7         else:  
8             self.head = Node(val,  
9             self.head)  
10            self.size += 1
```

src/ll\_prepend.py

## Inserting at the head or tail

- ▶ Take the current head or tail.
- ▶ And put the new node before or after it :)
- ▶  $O(1)$  time!

```
1 class LinkedList:  
  
3     def add_first(self, val: int):  
4         if self.head is None:  
5             self.head = Node(val, None)  
6             self.tail = self.head  
7         else:  
8             self.head = Node(val,  
9             self.head)  
          self.size += 1
```

src/ll\_prepend.py

```
1 class LinkedList:  
  
3     def add_last(self, val: int):  
4         if self.head is None:  
5             self.head = Node(val, None)  
6             self.tail = self.head  
7         else:  
8             self.tail.next = Node(val,  
9             None)  
             self.tail = self.tail.next  
          self.size += 1
```

src/ll\_append.py

## Inserting at the head or tail

- ▶ Take the current head or tail.
- ▶ And put the new node before or after it :)
- ▶  $O(1)$  time!

```
1 class LinkedList:  
2  
3     def add_first(self, val: int):  
4         if self.head is None:  
5             self.head = Node(val, None)  
6             self.tail = self.head  
7         else:  
8             self.head = Node(val,  
9             self.head)  
10            self.size += 1
```

src/ll\_prepend.py

```
1 class LinkedList:  
2  
3     def add_last(self, val: int):  
4         if self.head is None:  
5             self.head = Node(val, None)  
6             self.tail = self.head  
7         else:  
8             self.tail.next = Node(val,  
9             None)  
10            self.tail = self.tail.next  
11            self.size += 1
```

src/ll\_append.py

## Assumptions, Assumptions

### UNFORTUNATELY

To do `addLast` in constant time, we need a reference to the tail.  
Many implementations of Singly-Linked Lists do not have this.

### WAIT SINGLY?

What do I mean with 'Singly' Linked List?

### WELL, IT'S NOT DOUBLY LINKED

In other implementations, we can both go forwards and backwards!

## Assumptions, Assumptions

UNFORTUNATELY

To do `addLast` in constant time, we need a reference to the tail.  
Many implementations of Singly-Linked Lists do not have this.

WAIT SINGLY?

What do I mean with 'Singly' Linked List?

WELL, IT'S NOT DOUBLY LINKED

In other implementations, we can both go forwards and backwards!

## Assumptions, Assumptions

### UNFORTUNATELY

To do `addLast` in constant time, we need a reference to the tail.  
Many implementations of Singly-Linked Lists do not have this.

### WAIT SINGLY?

What do I mean with 'Singly' Linked List?

### WELL, IT'S NOT DOUBLY LINKED

In other implementations, we can both go forwards and backwards!

## Doubly-Linked Lists

### A SINGLY LINKED LIST

Only has connections in one direction.

## Doubly-Linked Lists

### A SINGLY LINKED LIST

Only has connections in one direction.

### A DOUBLY LINKED LIST

Has connections in both directions.

# Inserting an item

What to do?

- ▶ Navigate to the place where we want to insert the item ( $O(index)$ )
- ▶ Add the item:  $O(1)$
- ▶ So  $O(index)$  time!

```
1 class LinkedList:  
2  
3     def insert(self, index: int, value: int):  
4         if index >= self.size:  
5             self.append(value)  
6         elif index == 0:  
7             self.add_first(value)  
8         else:  
9             cur = self.head  
10            cnt = 0  
11            while cnt < index-1:  
12                cur = cur.next  
13                cnt += 1  
14            newNode = Node(value, cur.next)  
15            cur.next = newNode  
16            self.size += 1
```

src/ll\_insert.py

# Inserting an item

What to do?

- ▶ Navigate to the place where we want to insert the item ( $O(index)$ )
- ▶ Add the item:  $O(1)$
- ▶ So  $O(index)$  time!

```
1 class LinkedList:  
2  
3     def insert(self, index: int, value: int):  
4         if index >= self.size:  
5             self.append(value)  
6         elif index == 0:  
7             self.add_first(value)  
8         else:  
9             cur = self.head  
10            cnt = 0  
11            while cnt < index-1:  
12                cur = cur.next  
13                cnt += 1  
14            newNode = Node(value, cur.next)  
15            cur.next = newNode  
16            self.size += 1
```

src/ll\_insert.py

# Inserting an item

What to do?

- ▶ Navigate to the place where we want to insert the item ( $O(index)$ )
- ▶ Add the item:  $O(1)$
- ▶ So  $O(index)$  time!

```
1 class LinkedList:  
2  
3     def insert(self, index: int, value: int):  
4         if index >= self.size:  
5             self.append(value)  
6         elif index == 0:  
7             self.add_first(value)  
8         else:  
9             cur = self.head  
10            cnt = 0  
11            while cnt < index-1:  
12                cur = cur.next  
13                cnt += 1  
14            newNode = Node(value, cur.next)  
15            cur.next = newNode  
16            self.size += 1
```

src/ll\_insert.py

# Inserting an item

What to do?

- ▶ Navigate to the place where we want to insert the item ( $O(index)$ )
- ▶ Add the item:  $O(1)$
- ▶ So  $O(index)$  time!

```
1      class LinkedList:  
2  
3          def insert(self, index: int, value: int):  
4              if index >= self.size:  
5                  self.append(value)  
6              elif index == 0:  
7                  self.add_first(value)  
8              else:  
9                  cur = self.head  
10                 cnt = 0  
11                 while cnt < index-1:  
12                     cur = cur.next  
13                     cnt += 1  
14                 newNode = Node(value, cur.next)  
15                 cur.next = newNode  
16                 self.size += 1
```

src/ll\_insert.py

## Removing an item

### REMOVING AN ITEM

What is the time complexity of removing the first and last item in a singly linked list?

- A.  $O(1)$  for the first,  $O(1)$  for the last.
- B.  $O(1)$  for the first,  $O(n)$  for the last.
- C.  $O(n)$  for the first,  $O(1)$  for the last.
- D.  $O(n)$  for the first,  $O(n)$  for the last.
- E. I don't know.

## Removing an item

### REMOVING AN ITEM

What is the time complexity of removing the first and last item in a singly linked list?

```
1 class LinkedList:  
2  
3     def remove_first(self):  
4         if self.size == 1:  
5             self.head = None  
6             self.tail = self.head  
7         else:  
8             self.head = self.head.next  
9             self.size -= 1
```

$O(1)$  time.

src/ll\_remove\_first.py

## Removing an item

### REMOVING AN ITEM

What is the time complexity of removing the first and last item in a singly linked list?

```
1 class LinkedList:  
  
3     def remove_first(self):  
4         if self.size == 1:  
5             self.head = None  
6             self.tail = self.head  
7         else:  
8             self.head = self.head.next  
9             self.size -= 1
```

$O(1)$  time.

src/ll\_remove\_first.py

```
1 class LinkedList:  
  
3     def remove_last(self):  
4         if self.size == 1:  
5             self.head = None  
6             self.tail = self.head  
7         else:  
8             cur = self.head  
9             while cur.next != self.tail:  
10                 cur = cur.next  
11             cur.next = None  
12             self.tail = cur  
13             self.size -= 1
```

$O(n)$  time.

src/ll\_remove\_last.py

# Improvements!

## DOUBLY-LINKED LIST REMOVE LAST

Note that in a doubly-linked list we can remove the last item in  $O(1)$  time!  
We can just use `tail.prev` to find the last-but-one element in constant time!

## So to summarise

Operation	Array-based list	SLL	DLL
Get element $k$	$O(1)$	$O(k)$	$O(k)$
Insert first element	$O(n)$	$O(1)$	$O(1)$
Insert at index $k$	$O(n - k)$	$O(k)$	$O(\min(k, n - k))$
Append (amortised)	$O(1)$	$O(1)$	$O(1)$
Remove first element	$O(n)$	$O(1)$	$O(1)$
Remove last element	$O(1)$	$O(n)$	$O(1)$
Remove index $k$	$O(n - k)$	$O(k)$	$O(\min(k, n - k))$
Search	$O(n)$	$O(n)$	$O(n)$

## So to summarise

Operation	Array-based list	SLL	DLL
Get element $k$	$O(1)$	$O(k)$	$O(k)$
Insert first element	$O(n)$	$O(1)$	$O(1)$
Insert at index $k$	$O(n - k)$	$O(k)$	$O(\min(k, n - k))$
Append (amortised)	$O(1)$	$O(1)$	$O(1)$
Remove first element	$O(n)$	$O(1)$	$O(1)$
Remove last element	$O(1)$	$O(n)$	$O(1)$
Remove index $k$	$O(n - k)$	$O(k)$	$O(\min(k, n - k))$
Search	$O(n)$	$O(n)$	$O(n)$

## So to summarise

Operation	Array-based list	SLL	DLL
Get element $k$	$O(1)$	$O(k)$	$O(k)$
Insert first element	$O(n)$	$O(1)$	$O(1)$
Insert at index $k$	$O(n - k)$	$O(k)$	$O(\min(k, n - k))$
Append (amortised)	$O(1)$	$O(1)$	$O(1)$
Remove first element	$O(n)$	$O(1)$	$O(1)$
Remove last element	$O(1)$	$O(n)$	$O(1)$
Remove index $k$	$O(n - k)$	$O(k)$	$O(\min(k, n - k))$
Search	$O(n)$	$O(n)$	$O(n)$

## So to summarise

Operation	Array-based list	SLL	DLL
Get element $k$	$O(1)$	$O(k)$	$O(k)$
Insert first element	$O(n)$	$O(1)$	$O(1)$
Insert at index $k$	$O(n - k)$	$O(k)$	$O(\min(k, n - k))$
Append (amortised)	$O(1)$	$O(1)$	$O(1)$
Remove first element	$O(n)$	$O(1)$	$O(1)$
Remove last element	$O(1)$	$O(n)$	$O(1)$
Remove index $k$	$O(n - k)$	$O(k)$	$O(\min(k, n - k))$
Search	$O(n)$	$O(n)$	$O(n)$

# Queue

# The Queue ADT

## THE QUEUE

- ▶ `size()` (or `len(s)`) to get the number of items in the queue.
- ▶ `enqueue(item)` to add something to the queue.
- ▶ `dequeue()` to remove the first element from the queue.
- ▶ `peek()` to view the first element of the queue.

## DATA STRUCTURE?

What kind of data structure should we use to implement a Queue?

## ONE END ONLY

Only removing on one end and adding on the other? Seems like a DLL will do.

# The Queue ADT

## THE QUEUE

- ▶ `size()` (or `len(s)`) to get the number of items in the queue.
- ▶ `enqueue(item)` to add something to the queue.
- ▶ `dequeue()` to remove the first element from the queue.
- ▶ `peek()` to view the first element of the queue.

## DATA STRUCTURE?

What kind of data structure should we use to implement a Queue?

## ONE END ONLY

Only removing on one end and adding on the other? Seems like a DLL will do.

## The Queue ADT

### THE QUEUE

- ▶ `size()` (or `len(s)`) to get the number of items in the queue.
- ▶ `enqueue(item)` to add something to the queue.
- ▶ `dequeue()` to remove the first element from the queue.
- ▶ `peek()` to view the first element of the queue.

### DATA STRUCTURE?

What kind of data structure should we use to implement a Queue?

- A. An array
- B. A python list
- C. A linked list
- D. A dict

### ONE END ONLY

Only removing on one end and adding on the other? Seems like a DLL will do.

## The Queue ADT

### THE QUEUE

- ▶ `size()` (or `len(s)`) to get the number of items in the queue.
- ▶ `enqueue(item)` to add something to the queue.
- ▶ `dequeue()` to remove the first element from the queue.
- ▶ `peek()` to view the first element of the queue.

### DATA STRUCTURE?

What kind of data structure should we use to implement a Queue?

### ONE END ONLY

Only removing on one end and adding on the other? Seems like a DLL will do.

## Implementing a Linked-List based Queue

Queue operation	DLL operation	Time Complexity
size		
enqueue		
dequeue		
peek		

## Implementing a Linked-List based Queue

Queue operation	DLL operation	Time Complexity
size	size	$O(1)$
enqueue	add_last	$O(1)$
dequeue	remove_first	$O(1)$
peek	head	$O(1)$

## Implementing a Linked-List based Queue

Queue operation	DLL operation	Time Complexity
size	size	$O(1)$
enqueue	add_last	$O(1)$
dequeue	remove_first	$O(1)$
peek	head	$O(1)$

### ARRAYS

Could we also do this using an array-based list?

## Implementing a Linked-List based Queue

Queue operation	DLL operation	Time Complexity
size	size	$O(1)$
enqueue	add_last	$O(1)$
dequeue	remove_first	$O(1)$
peek	head	$O(1)$

### ARRAYS

Could we also do this using an array-based list?

### NO MORE GROWING

Sure, but only for fixed capacity queues.

## Implementing a Linked-List based Queue

Queue operation	DLL operation	Time Complexity
size	size	$O(1)$
enqueue	add_last	$O(1)$
dequeue	remove_first	$O(1)$
peek	head	$O(1)$

### ARRAYS

Could we also do this using an array-based list?

### NO MORE GROWING

Sure, but only for fixed capacity queues.

### SLL

What about an SLL?

## Implementing a Linked-List based Queue

Queue operation	DLL operation	Time Complexity
size	size	$O(1)$
enqueue	add_last	$O(1)$
dequeue	remove_first	$O(1)$
peek	head	$O(1)$

### ARRAYS

Could we also do this using an array-based list?

### NO MORE GROWING

Sure, but only for fixed capacity queues.

### SLL

What about an SLL?

### EEYORE

Only if we have the tail pointer.

# Snake games

## Nostalgia overload



Image By: Thomas Jensen

## Why a queue

### IN SNAKE...

- ▶ In snake, the body of the snake follows the head.
- ▶ If we first go left, then right, then every part of the body also needs to go first left then right.
- ▶ This is like a queue!

### MANY OTHERS?

But of course there are many other real-world examples.

- ▶ Ticket counters,
- ▶ Traffic jams,
- ▶ Coffee machines and printers,
- ▶ Take-out restaurants,
- ▶ etc.

## Why a queue

### IN SNAKE...

- ▶ In snake, the body of the snake follows the head.
- ▶ If we first go left, then right, then every part of the body also needs to go first left then right.
- ▶ This is like a queue!

### MANY OTHERS?

But of course there are many other real-world examples.

- ▶ Ticket counters,
- ▶ Traffic jams,
- ▶ Coffee machines and printers,
- ▶ Take-out restaurants,
- ▶ etc.

## Why a queue

### IN SNAKE...

- ▶ In snake, the body of the snake follows the head.
- ▶ If we first go left, then right, then every part of the body also needs to go first left then right.
- ▶ This is like a queue!

### MANY OTHERS?

But of course there are many other real-world examples.

- ▶ Ticket counters,
- ▶ Traffic jams,
- ▶ Coffee machines and printers,
- ▶ Take-out restaurants,
- ▶ etc.

## Why a queue

### IN SNAKE...

- ▶ In snake, the body of the snake follows the head.
- ▶ If we first go left, then right, then every part of the body also needs to go first left then right.
- ▶ This is like a queue!

### MANY OTHERS?

But of course there are many other real-world examples.

- ▶ Ticket counters,
- ▶ Traffic jams,
- ▶ Coffee machines and printers,
- ▶ Take-out restaurants,
- ▶ etc.

## Double-Ended Queues



Image By: *Chris Lott*

## Deques

### DEQUES

Deques, or Double-Ended Queues, allow both FIFO and LIFO operations in  $O(1)$  time.

### HANG ON...

Doesn't that sound familiar?

### BACK TO DLL

This is exactly what a DLL offers!

## Deques

### DEQUES

Deques, or Double-Ended Queues, allow both FIFO and LIFO operations in  $O(1)$  time.

### HANG ON...

Doesn't that sound familiar?

### BACK TO DLL

This is exactly what a DLL offers!

## Deques

### DEQUES

Deques, or Double-Ended Queues, allow both FIFO and LIFO operations in  $O(1)$  time.

### HANG ON...

Doesn't that sound familiar?

### BACK TO DLL

This is exactly what a DLL offers!

## deque

In python we can use a deque called d on which:

- ▶ We can *push* with d.append.
- ▶ We can *pop* with d.pop.
- ▶ We can *top* with d[-1]
- ▶ We can *enqueue* with d.append.
- ▶ We can *dequeue* with d.popleft.
- ▶ We can *peek* with d[0]
- ▶ Oh yeah and if you want, you can also add to the left with appendleft

## deque

In python we can use a deque called d on which:

- ▶ We can *push* with `d.append`.
- ▶ We can *pop* with `d.pop`.
- ▶ We can *top* with `d[-1]`
- ▶ We can *enqueue* with `d.append`.
- ▶ We can *dequeue* with `d.popleft`.
- ▶ We can *peek* with `d[0]`
- ▶ Oh yeah and if you want, you can also add to the left with `appendleft`

## deque

In python we can use a deque called d on which:

- ▶ We can *push* with `d.append`.
- ▶ We can *pop* with `d.pop`.
- ▶ We can *top* with `d[-1]`
- ▶ We can *enqueue* with `d.append`.
- ▶ We can *dequeue* with `d.popleft`.
- ▶ We can *peek* with `d[0]`
- ▶ Oh yeah and if you want, you can also add to the left with `appendleft`

# Stack

# Stacks



Image from *Pixabay*

# Me and my books

## A life-long story



Image By: Stefan Hugtenburg

Bookcovers and picture in the back by others

- ▶ This is how I used to store books I still wanted to read.
- ▶ A nice stack of books, with new ones going on the top.
- ▶ After finishing one, I would take the next one from the top.
- ▶ So after a few weeks...
- ▶ This uses the LIFO-principle.

# Me and my books

## A life-long story



Image By: Stefan Hugtenburg

Bookcovers and picture in the back by others

- ▶ This is how I used to store books I still wanted to read.
- ▶ A nice **stack** of books, with new ones going on the top.
- ▶ After finishing one, I would take the next one from the top.
- ▶ So after a few weeks...
- ▶ This uses the **LIFO-principle**.

# Me and my books

## A life-long story



Image By: Stefan Hugtenburg

Bookcovers and picture in the back by others

- ▶ This is how I used to store books I still wanted to read.
- ▶ A nice **stack** of books, with new ones going on the top.
- ▶ After finishing one, I would take the next one from the top.
- ▶ So after a few weeks...
- ▶ This uses the **LIFO-principle**.

# Me and my books

## A life-long story



Image By: Stefan Hugtenburg

Bookcovers and picture in the back by others

- ▶ This is how I used to store books I still wanted to read.
- ▶ A nice **stack** of books, with new ones going on the top.
- ▶ After finishing one, I would take the next one from the top.
- ▶ So after a few weeks...
- ▶ This uses the **LIFO-principle**.

# Me and my books

## A life-long story



Image By: Stefan Hugtenburg

Bookcovers and picture in the back by others

- ▶ This is how I used to store books I still wanted to read.
- ▶ A nice **stack** of books, with new ones going on the top.
- ▶ After finishing one, I would take the next one from the top.
- ▶ So after a few weeks...
- ▶ This uses the **LIFO**-principle.

## The what!?

LIFO

LIFO

The *Last-In-First-Out*, or LIFO, principle is the working of a stack.

The last thing we've added to the stack is the first thing we take out.

Similarly the first we have added to the stack, is the last to be taken out.

## The what!?

LIFO

LIFO

The *Last-In-First-Out*, or LIFO, principle is the working of a stack.

The last thing we've added to the stack is the first thing we take out.

Similarly the first we have added to the stack, is the last to be taken out.

## The what!?

LIFO

LIFO

The *Last-In-First-Out*, or LIFO, principle is the working of a stack.

The last thing we've added to the stack is the first thing we take out.

Similarly the first we have added to the stack, is the last to be taken out.

## The Stack ADT

### ADT

An ADT, or Abstract Data Type, is a description of the behaviour of a data structure.

### THE STACK

- ▶ `size()` (or `len(s)`) to get the number of items in the stack.
- ▶ `push(item)` to add something to the stack.
- ▶ `pop()` to remove the top element from the stack.
- ▶ `top()` to view the top element of the stack.

### DATA STRUCTURE?

What kind of data structure should we use to implement a Stack?

### ONE END ONLY

Only removing and adding on one end? Seems like a DLL will do.

## The Stack ADT

### THE STACK

- ▶ `size()` (or `len(s)`) to get the number of items in the stack.
- ▶ `push(item)` to add something to the stack.
- ▶ `pop()` to remove the top element from the stack.
- ▶ `top()` to view the top element of the stack.

### DATA STRUCTURE?

What kind of data structure should we use to implement a Stack?

### ONE END ONLY

Only removing and adding on one end? Seems like a DLL will do.

## The Stack ADT

### THE STACK

- ▶ `size()` (or `len(s)`) to get the number of items in the stack.
- ▶ `push(item)` to add something to the stack.
- ▶ `pop()` to remove the top element from the stack.
- ▶ `top()` to view the top element of the stack.

### DATA STRUCTURE?

What kind of data structure should we use to implement a Stack?

### ONE END ONLY

Only removing and adding on one end? Seems like a DLL will do.

## The Stack ADT

### THE STACK

- ▶ `size()` (or `len(s)`) to get the number of items in the stack.
- ▶ `push(item)` to add something to the stack.
- ▶ `pop()` to remove the top element from the stack.
- ▶ `top()` to view the top element of the stack.

### DATA STRUCTURE?

What kind of data structure should we use to implement a Stack?

- A. An array
- B. A python list
- C. A linked list
- D. A dict

### ONE END ONLY

Only removing and adding on one end? Seems like a DLL will do.

## The Stack ADT

### THE STACK

- ▶ `size()` (or `len(s)`) to get the number of items in the stack.
- ▶ `push(item)` to add something to the stack.
- ▶ `pop()` to remove the top element from the stack.
- ▶ `top()` to view the top element of the stack.

### DATA STRUCTURE?

What kind of data structure should we use to implement a Stack?

### ONE END ONLY

Only removing and adding on one end? Seems like a DLL will do.

## Implementing a Linked-List based Stack

Stack operation	DLL operation	Time Complexity
size		
push		
pop		
top		

## Implementing a Linked-List based Stack

Stack operation	DLL operation	Time Complexity
size	size	$O(1)$
push	add_last	$O(1)$
pop	remove_last	$O(1)$
top	tail	$O(1)$

## Implementing a Linked-List based Stack

Stack operation	DLL operation	Time Complexity
size	size	$O(1)$
push	add_last	$O(1)$
pop	remove_last	$O(1)$
top	tail	$O(1)$

### ARRAYS

Could we also do this using an array-based list?

## Implementing a Linked-List based Stack

Stack operation	DLL operation	Time Complexity
size	size	$O(1)$
push	add_last	$O(1)$
pop	remove_last	$O(1)$
top	tail	$O(1)$

### ARRAYS

Could we also do this using an array-based list?

### AMORTISED

Sure, but it's only amortised  $O(1)$ .  
So why would we?

## Implementing a Linked-List based Stack

Stack operation	DLL operation	Time Complexity
size	size	$O(1)$
push	add_last	$O(1)$
pop	remove_last	$O(1)$
top	tail	$O(1)$

### ARRAYS

Could we also do this using an array-based list?

### AMORTISED

Sure, but it's only amortised  $O(1)$ . So why would we?

### SLL

What about an SLL?

## Implementing a Linked-List based Stack

Stack operation	DLL operation	Time Complexity
size	size	$O(1)$
push	add_last	$O(1)$
pop	remove_last	$O(1)$
top	tail	$O(1)$

### ARRAYS

Could we also do this using an array-based list?

### AMORTISED

Sure, but it's only amortised  $O(1)$ . So why would we?

### SLL

What about an SLL?

### FRONT = BACK

Sure, but we add and remove from the front!

# Algorithms

# Sorting

# BogoSort

## BOGOSORT

The easiest form of sorting: Just try all permutations of the list.

TIME COMPLEXITY?

- A.  $\Theta(n)$
- B.  $\Theta(n^2)$
- C.  $\Theta(2^n)$
- D.  $\Theta(n^n)$
- E.  $\Theta(n!)$
- F. I don't know

# BogoSort

## BOGOSORT

The easiest form of sorting: Just try all permutations of the list.

### TIME COMPLEXITY?

- A.  $\Theta(n)$
- B.  $\Theta(n^2)$
- C.  $\Theta(2^n)$
- D.  $\Theta(n^n)$
- E.  $\Theta(n!)$
- F. I don't know

## Some questions and answers

### THE SCOPE OF OUR PROBLEM

Some observations and questions.

- ▶ The solution space is huge:  $n!$  for  $n$  items.
- ▶ The use case is not uncommon, so we hope/look for an efficient (polynomial time) algorithm.
- ▶ Humans have a lot of experience sorting items, can we use those strategies?
- ▶ Or are there methods that computers excel at, that would be terrible for humans?

YES!

- ▶ There are polynomial time algorithms,
- ▶ We can translate our own methods,
- ▶ But computers can definitely do better.

## Some questions and answers

### THE SCOPE OF OUR PROBLEM

Some observations and questions.

- ▶ The solution space is huge:  $n!$  for  $n$  items.
- ▶ The use case is not uncommon, so we hope/look for an efficient (polynomial time) algorithm.
- ▶ Humans have a lot of experience sorting items, can we use those strategies?
- ▶ Or are there methods that computers excel at, that would be terrible for humans?

YES!

- ▶ There are polynomial time algorithms,
- ▶ We can translate our own methods,
- ▶ But computers can definitely do better.

## Some questions and answers

### THE SCOPE OF OUR PROBLEM

Some observations and questions.

- ▶ The solution space is huge:  $n!$  for  $n$  items.
- ▶ The use case is not uncommon, so we hope/look for an efficient (polynomial time) algorithm.
- ▶ Humans have a lot of experience sorting items, can we use those strategies?
- ▶ Or are there methods that computers excel at, that would be terrible for humans?

YES!

- ▶ There are polynomial time algorithms,
- ▶ We can translate our own methods,
- ▶ But computers can definitely do better.

## Some questions and answers

### THE SCOPE OF OUR PROBLEM

Some observations and questions.

- ▶ The solution space is huge:  $n!$  for  $n$  items.
- ▶ The use case is not uncommon, so we hope/look for an efficient (polynomial time) algorithm.
- ▶ Humans have a lot of experience sorting items, can we use those strategies?
- ▶ Or are there methods that computers excel at, that would be terrible for humans?

YES!

- ▶ There are polynomial time algorithms,
- ▶ We can translate our own methods,
- ▶ But computers can definitely do better.

## Some questions and answers

### THE SCOPE OF OUR PROBLEM

Some observations and questions.

- ▶ The solution space is huge:  $n!$  for  $n$  items.
- ▶ The use case is not uncommon, so we hope/look for an efficient (polynomial time) algorithm.
- ▶ Humans have a lot of experience sorting items, can we use those strategies?
- ▶ Or are there methods that computers excel at, that would be terrible for humans?

YES!

- ▶ There are polynomial time algorithms,
- ▶ We can translate our own methods,
- ▶ But computers can definitely do better.

## Some questions and answers

### THE SCOPE OF OUR PROBLEM

Some observations and questions.

- ▶ The solution space is huge:  $n!$  for  $n$  items.
- ▶ The use case is not uncommon, so we hope/look for an efficient (polynomial time) algorithm.
- ▶ Humans have a lot of experience sorting items, can we use those strategies?
- ▶ Or are there methods that computers excel at, that would be terrible for humans?

YES!

- ▶ There are polynomial time algorithms,
- ▶ We can translate our own methods,
- ▶ But computers can definitely do better.

# Use case?

WHY?

What do we use sorting for?

MANY THINGS!

- ▶ Books in the library.
- ▶ Exams for an exam review.
- ▶ Closest pair of points.
- ▶ Convex hulls.
- ▶ Efficient look-ups in lists that do not change.
- ▶ And many many more (see also later lectures in graph algorithms).

## Use case?

WHY?

What do we use sorting for?

MANY THINGS!

- ▶ Books in the library.
- ▶ Exams for an exam review.
- ▶ Closest pair of points.
- ▶ Convex hulls.
- ▶ Efficient look-ups in lists that do not change.
- ▶ And many many more (see also later lectures in graph algorithms).

## Bubble Sort

## Bubble sort

Getting rid of inversions

LET'S FORMALISE

What did you do yesterday?

YOUR ALGORITHM

W

hile  $v_i > v_{i+1}$  for some  $i$  State Switch them around. EndWhile

# Bubble sort

Getting rid of inversions

## YOUR ALGORITHM

W

hile  $v_i > v_{i+1}$  for some  $i$  State Switch them around. EndWhile

# Bubble sort

Getting rid of inversions

YOUR ALGORITHM

W

hile  $v_i > v_{i+1}$  for some  $i$  State Switch them around. EndWhile

1	20	42	22	17
---	----	----	----	----

# Bubble sort

Getting rid of inversions

YOUR ALGORITHM

W

hile  $v_i > v_{i+1}$  for some  $i$  State Switch them around. EndWhile

1	20	42	22	17
---	----	----	----	----

# Bubble sort

Getting rid of inversions

YOUR ALGORITHM

W

hile  $v_i > v_{i+1}$  for some  $i$  State Switch them around. EndWhile

1	20	42	22	17
---	----	----	----	----

# Bubble sort

Getting rid of inversions

YOUR ALGORITHM

W

hile  $v_i > v_{i+1}$  for some  $i$  State Switch them around. EndWhile

1	20	42	22	17
---	----	----	----	----

# Bubble sort

Getting rid of inversions

YOUR ALGORITHM

W

hile  $v_i > v_{i+1}$  for some  $i$  State Switch them around. EndWhile

1	20	22	42	17
---	----	----	----	----

# Bubble sort

Getting rid of inversions

YOUR ALGORITHM

W

hile  $v_i > v_{i+1}$  for some  $i$  State Switch them around. EndWhile

1	20	22	42	17
---	----	----	----	----

# Bubble sort

Getting rid of inversions

YOUR ALGORITHM

W

hile  $v_i > v_{i+1}$  for some  $i$  State Switch them around. EndWhile

1	20	22	17	42
---	----	----	----	----

# Bubble sort

Getting rid of inversions

YOUR ALGORITHM

W

hile  $v_i > v_{i+1}$  for some  $i$  State Switch them around. EndWhile

1	20	22	17	42
---	----	----	----	----

# Bubble sort

Getting rid of inversions

YOUR ALGORITHM

W

hile  $v_i > v_{i+1}$  for some  $i$  State Switch them around. EndWhile

1	20	22	17	42
---	----	----	----	----

# Bubble sort

Getting rid of inversions

YOUR ALGORITHM

W

hile  $v_i > v_{i+1}$  for some  $i$  State Switch them around. EndWhile

1	20	22	17	42
---	----	----	----	----

# Bubble sort

Getting rid of inversions

YOUR ALGORITHM

W

hile  $v_i > v_{i+1}$  for some  $i$  State Switch them around. EndWhile

1	20	17	22	42
---	----	----	----	----

# Bubble sort

Getting rid of inversions

YOUR ALGORITHM

W

hile  $v_i > v_{i+1}$  for some  $i$  State Switch them around. EndWhile

1	20	17	22	42
---	----	----	----	----

# Bubble sort

Getting rid of inversions

YOUR ALGORITHM

W

hile  $v_i > v_{i+1}$  for some  $i$  State Switch them around. EndWhile

1	20	17	22	42
---	----	----	----	----

# Bubble sort

Getting rid of inversions

YOUR ALGORITHM

W

hile  $v_i > v_{i+1}$  for some  $i$  State Switch them around. EndWhile

1	20	17	22	42
---	----	----	----	----

# Bubble sort

Getting rid of inversions

YOUR ALGORITHM

W

hile  $v_i > v_{i+1}$  for some  $i$  State Switch them around. EndWhile

1	17	20	22	42
---	----	----	----	----

# Bubble sort

Getting rid of inversions

YOUR ALGORITHM

W

hile  $v_i > v_{i+1}$  for some  $i$  State Switch them around. EndWhile

1	17	20	22	42
---	----	----	----	----

# Bubble sort

Getting rid of inversions

YOUR ALGORITHM

W

hile  $v_i > v_{i+1}$  for some  $i$  State Switch them around. EndWhile

1	17	20	22	42
---	----	----	----	----

# Bubble sort

Getting rid of inversions

YOUR ALGORITHM

W

hile  $v_i > v_{i+1}$  for some  $i$  State Switch them around. EndWhile

1	17	20	22	42
---	----	----	----	----

# Bubble sort

Getting rid of inversions

YOUR ALGORITHM

W

hile  $v_i > v_{i+1}$  for some  $i$  State Switch them around. EndWhile

1	17	20	22	42
---	----	----	----	----

# Bubble sort

Getting rid of inversions

YOUR ALGORITHM

W

hile  $v_i > v_{i+1}$  for some  $i$  State Switch them around. EndWhile

1	17	20	22	42
---	----	----	----	----

# Bubble sort

Getting rid of inversions

YOUR ALGORITHM

W

hile  $v_i > v_{i+1}$  for some  $i$  State Switch them around. EndWhile

1	17	20	22	42
---	----	----	----	----

## Bubble sort

Getting rid of inversions

### YOUR ALGORITHM

W

while  $v_i > v_{i+1}$  for some  $i$  State Switch them around. EndWhile

1	17	20	22	42
---	----	----	----	----

### SOUNDS EASY...

So how many inversions can there be?

- A.  $\Theta(n)$
- B.  $\Theta(n^2)$
- C.  $\Theta(n^n)$
- D.  $\Theta(n!)$

## The number of inversions

What is the worst-case?

### A LIST IN REVERSE ORDER

Remember yesterday? The people that were at the wrong end took the longest!

Consider a list in reverse order:

- ▶ The first element is wrong compared to all others:  $n - 1$  inversions.
- ▶ The second is also wrong with all the ones that come after it:  $n - 2$  extra inversions.
- ▶ ...
- ▶ The one-but last one is also wrong with the last one: 1 inversion
- ▶ So  $\Theta(n^2)$  inversion!

## The number of inversions

What is the worst-case?

### A LIST IN REVERSE ORDER

Remember yesterday? The people that were at the wrong end took the longest!

Consider a list in reverse order:

- ▶ The first element is wrong compared to all others:  $n - 1$  inversions.
- ▶ The second is also wrong with all the ones that come after it:  $n - 2$  extra inversions.
- ▶ ...
- ▶ The one-but last one is also wrong with the last one: 1 inversion
- ▶ So  $\Theta(n^2)$  inversion!

## The number of inversions

What is the worst-case?

### A LIST IN REVERSE ORDER

Remember yesterday? The people that were at the wrong end took the longest!

Consider a list in reverse order:

- ▶ The first element is wrong compared to all others:  $n - 1$  inversions.
- ▶ The second is also wrong with all the ones that come after it:  $n - 2$  extra inversions.
- ▶ ...
- ▶ The one-but last one is also wrong with the last one: 1 inversion
- ▶ So  $\Theta(n^2)$  inversion!

## The number of inversions

What is the worst-case?

### A LIST IN REVERSE ORDER

Remember yesterday? The people that were at the wrong end took the longest!

Consider a list in reverse order:

- ▶ The first element is wrong compared to all others:  $n - 1$  inversions.
- ▶ The second is also wrong with all the ones that come after it:  $n - 2$  extra inversions.
- ▶ ...
- ▶ The one-but last one is also wrong with the last one: 1 inversion
- ▶ So  $\Theta(n^2)$  inversion!

## The number of inversions

What is the worst-case?

### A LIST IN REVERSE ORDER

Remember yesterday? The people that were at the wrong end took the longest!

Consider a list in reverse order:

- ▶ The first element is wrong compared to all others:  $n - 1$  inversions.
- ▶ The second is also wrong with all the ones that come after it:  $n - 2$  extra inversions.
- ▶ ...
- ▶ The one-but last one is also wrong with the last one: 1 inversion
- ▶ So  $\Theta(n^2)$  inversion!

## To summarise in code

```
1 from typing import List  
2  
3 def bubblesort(xs: List[int]):  
4     for i in range(len(xs)):  
5         for j in range(len(xs)-1):  
6             if xs[j] > xs[j+1]:  
7                 xs[j+1], xs[j] = xs[j],  
8                 xs[j+1]
```

src/bs.py

### PROS AND CONS?

What are the pros and cons of bubblesort? Hint: remember I told you it was slow!

## Bubblesort: Pros and Cons

### BUBBLESORT

While there are inversions: fix them.

### PROS

- ▶ Great in a distributed setting, with autonomous agents (like students in a lecture hall).
- ▶ When implemented as: “continue while there are inversion” can terminate after one iteration over a sorted list!
- ▶ Easiest to implement.

### CONS

- ▶ Terribly slow! (Often still slower than other  $\Theta(n^2)$  algorithms we will see later)

## Bucket Sort

## Bucket Sort

The Stephen Tatlock algorithm...

### BUCKET SORT

1. Sort the items into buckets (e.g. based on first letter of last name).
2. Sort every bucket.
3. Concatenate all buckets together.

### WHY IS THIS GOOD?

Why should we use this?

### WELL...

- ▶ Great for parallelisation!
- ▶ Great for humans (sorting a bucket can be done using a different algorithm, everyone can pick their own favourite).
- ▶ Average case analysis is interesting for this algorithm (and gets us to something close to linear time for average case).

## Bucket Sort

The Stephen Tatlock algorithm...

### BUCKET SORT

1. Sort the items into buckets (e.g. based on first letter of last name).
2. Sort every bucket.
3. Concatenate all buckets together.

### WHY IS THIS GOOD?

Why should we use this?

### WELL...

- ▶ Great for parallelisation!
- ▶ Great for humans (sorting a bucket can be done using a different algorithm, everyone can pick their own favourite).
- ▶ Average case analysis is interesting for this algorithm (and gets us to something close to linear time for average case).

## Bucket Sort

The Stephen Tatlock algorithm... Okay that was probably too obscure a reference

### BUCKET SORT

1. Sort the items into buckets (e.g. based on first letter of last name).
2. Sort every bucket.
3. Concatenate all buckets together.

### WHY IS THIS GOOD?

Why should we use this?

### WELL...

- ▶ Great for parallelisation!
- ▶ Great for humans (sorting a bucket can be done using a different algorithm, everyone can pick their own favourite).
- ▶ Average case analysis is interesting for this algorithm (and gets us to something close to linear time for average case).

## Bucket Sort

The Stephen Tatlock algorithm... Okay that was probably too obscure a reference

### BUCKET SORT

1. Sort the items into buckets (e.g. based on first letter of last name).
2. Sort every bucket.
3. Concatenate all buckets together.

### WHY IS THIS GOOD?

Why should we use this?

### WELL...

- ▶ Great for parallelisation!
- ▶ Great for humans (sorting a bucket can be done using a different algorithm, everyone can pick their own favourite).
- ▶ Average case analysis is interesting for this algorithm (and gets us to something close to linear time for average case).

## Bucket Sort

The Stephen Tatlock algorithm... Okay that was probably too obscure a reference

### BUCKET SORT

1. Sort the items into buckets (e.g. based on first letter of last name).
2. Sort every bucket.
3. Concatenate all buckets together.

### WHY IS THIS GOOD?

Why should we use this?

### WELL...

- ▶ Great for parallelisation!
- ▶ Great for humans (sorting a bucket can be done using a different algorithm, everyone can pick their own favourite).
- ▶ Average case analysis is interesting for this algorithm (and gets us to something close to linear time for average case).

## Selection Sort

## Selection Sort

THE SMALLEST OF VARIATIONS...

Rather than shifting every element to it's correct place...

We instead repeatedly search for the smallest element and put it at the end of our sorted list/sorted part.

## Selection Sort

THE SMALLEST OF VARIATIONS...

Rather than shifting every element to it's correct place...

We instead repeatedly search for the smallest element and put it at the end of our sorted list/sorted part.

## Selection Sort

THE SMALLEST OF VARIATIONS...

Rather than shifting every element to it's correct place...

We instead repeatedly search for the smallest element and put it at the end of our sorted list/sorted part.

1	20	42	22	17
---	----	----	----	----

## Selection Sort

THE SMALLEST OF VARIATIONS...

Rather than shifting every element to it's correct place...

We instead repeatedly search for the smallest element and put it at the end of our sorted list/sorted part.

1	20	42	22	17
---	----	----	----	----

## Selection Sort

THE SMALLEST OF VARIATIONS...

Rather than shifting every element to it's correct place...

We instead repeatedly search for the smallest element and put it at the end of our sorted list/sorted part.

1	20	42	22	17
---	----	----	----	----

## Selection Sort

THE SMALLEST OF VARIATIONS...

Rather than shifting every element to it's correct place...

We instead repeatedly search for the smallest element and put it at the end of our sorted list/sorted part.

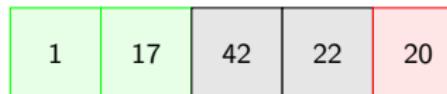
1	17	42	22	20
---	----	----	----	----

## Selection Sort

THE SMALLEST OF VARIATIONS...

Rather than shifting every element to it's correct place...

We instead repeatedly search for the smallest element and put it at the end of our sorted list/sorted part.



## Selection Sort

THE SMALLEST OF VARIATIONS...

Rather than shifting every element to it's correct place...

We instead repeatedly search for the smallest element and put it at the end of our sorted list/sorted part.

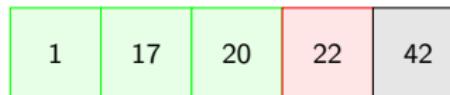
1	17	20	22	42
---	----	----	----	----

## Selection Sort

THE SMALLEST OF VARIATIONS...

Rather than shifting every element to it's correct place...

We instead repeatedly search for the smallest element and put it at the end of our sorted list/sorted part.



## Selection Sort

THE SMALLEST OF VARIATIONS...

Rather than shifting every element to it's correct place...

We instead repeatedly search for the smallest element and put it at the end of our sorted list/sorted part.

1	17	20	22	42
---	----	----	----	----

## Selection Sort

THE SMALLEST OF VARIATIONS...

Rather than shifting every element to it's correct place...

We instead repeatedly search for the smallest element and put it at the end of our sorted list/sorted part.

1	17	20	22	42
---	----	----	----	----

# Implementation

## TIME COMPLEXITY

Still  $\Theta(n^2)$  as we need to find the minimum  $O(n)$  times, which takes  $O(n)$  time.

## SEE NEXT WEEK'S HOMEWORK

You will tell me next week ;)

## In-place sorting algorithms

I'm not going anywhere!

- ▶ The sorting algorithms we have seen so-far are **in-place** sorting algorithms.
- ▶ They take a list and modify that list to become sorted.
- ▶ Of course we can also make them not in-place, by first making a copy of the list and sorting that.
- ▶ But after the break (when we discuss merge sort) we will see an example of an algorithm that is easier to make not in-place.

## In-place sorting algorithms

I'm not going anywhere!

- ▶ The sorting algorithms we have seen so-far are **in-place** sorting algorithms.
- ▶ They take a list and modify that list to become sorted.
- ▶ Of course we can also make them not in-place, by first making a copy of the list and sorting that.
- ▶ But after the break (when we discuss merge sort) we will see an example of an algorithm that is easier to make not in-place.

## In-place sorting algorithms

I'm not going anywhere!

- ▶ The sorting algorithms we have seen so-far are **in-place** sorting algorithms.
- ▶ They take a list and modify that list to become sorted.
- ▶ Of course we can also make them not in-place, by first making a copy of the list and sorting that.
- ▶ But after the break (when we discuss merge sort) we will see an example of an algorithm that is easier to make not in-place.

## In-place sorting algorithms

I'm not going anywhere!

- ▶ The sorting algorithms we have seen so-far are **in-place** sorting algorithms.
- ▶ They take a list and modify that list to become sorted.
- ▶ Of course we can also make them not in-place, by first making a copy of the list and sorting that.
- ▶ But after the break (when we discuss merge sort) we will see an example of an algorithm that is easier to make not in-place.

## Insertion Sort

# Insertion Sort

How does that work?

## INSERTION SORT

The main idea:

- ▶ We build the list, step by step.
- ▶ We keep our intermediate result sorted.
- ▶ At every step, we insert the next element into the right place.
- ▶ This means we need to shift over (worst-case)  $n$  elements for every element we insert...
- ▶ So in total  $\Theta(n^2)$  time.

# Insertion Sort

How does that work?

## INSERTION SORT

The main idea:

- ▶ We build the list, step by step.
- ▶ We keep our intermediate result sorted.
- ▶ At every step, we insert the next element into the right place.
- ▶ This means we need to shift over (worst-case)  $n$  elements for every element we insert...
- ▶ So in total  $\Theta(n^2)$  time.

# Insertion Sort

How does that work?

## INSERTION SORT

The main idea:

- ▶ We build the list, step by step.
- ▶ We keep our intermediate result sorted.
- ▶ At every step, we insert the next element into the right place.
- ▶ This means we need to shift over (worst-case)  $n$  elements for every element we insert...
- ▶ So in total  $\Theta(n^2)$  time.

# Insertion Sort

How does that work?

## INSERTION SORT

The main idea:

- ▶ We build the list, step by step.
- ▶ We keep our intermediate result sorted.
- ▶ At every step, we insert the next element into the right place.
- ▶ This means we need to shift over (worst-case)  $n$  elements for every element we insert...
- ▶ So in total  $\Theta(n^2)$  time.

# Insertion Sort

How does that work?

## INSERTION SORT

The main idea:

- ▶ We build the list, step by step.
- ▶ We keep our intermediate result sorted.
- ▶ At every step, we insert the next element into the right place.
- ▶ This means we need to shift over (worst-case)  $n$  elements for every element we insert...
- ▶ So in total  $\Theta(n^2)$  time.

## Insertion Sort

S

tate *i*gets1 Comment The first element forms a sorted list on its own While  
 $i < \text{len}(l)$  State *j*get*s<sub>i</sub>* While  $j > 0$  and  $v_{j-1} < v_j$  State swap  $v_j$  and  $v_{j-1}$  State  
*j*get*s<sub>j-1</sub>* EndWhile State *i*get*s<sub>i+1</sub>* EndWhile

## Insertion Sort

S

tate  $i \text{ gets } 1$  Comment The first element forms a sorted list on its own  
While  $i < \text{len}(l)$  State  $j \text{ gets } i$  While  $j > 0$  and  $v_{j-1} < v_j$  State swap  $v_j$  and  $v_{j-1}$  State  
 $j \text{ gets } j - 1$  EndWhile State  $i \text{ gets } i + 1$  EndWhile

## Insertion Sort

S

tate  $i \text{ gets } 1$  Comment The first element forms a sorted list on its own  
While  $i < \text{len}(l)$  State  $j \text{ gets } i$  While  $j > 0$  and  $v_{j-1} < v_j$  State swap  $v_j$  and  $v_{j-1}$  State  
 $j \text{ gets } j - 1$  EndWhile State  $i \text{ gets } i + 1$  EndWhile

## Insertion Sort

S

tate  $i \text{ gets } 1$  Comment The first element forms a sorted list on its own  
While  $i < \text{len}(l)$  State  $j \text{ gets } i$  While  $j > 0$  and  $v_{j-1} < v_j$  State swap  $v_j$  and  $v_{j-1}$  State  
 $j \text{ gets } j - 1$  EndWhile State  $i \text{ gets } i + 1$  EndWhile

## Insertion Sort

S

tate  $i \text{ gets } 1$  Comment The first element forms a sorted list on its own  
While  $i < \text{len}(l)$  State  $j \text{ gets } i$  While  $j > 0$  and  $v_{j-1} < v_j$  State swap  $v_j$  and  $v_{j-1}$  State  
 $j \text{ gets } j - 1$  EndWhile State  $i \text{ gets } i + 1$  EndWhile

## Insertion Sort

S

tate  $i \text{ gets } 1$  Comment The first element forms a sorted list on its own  
While  $i < \text{len}(l)$  State  $j \text{ gets } i$  While  $j > 0$  and  $v_{j-1} < v_j$  State swap  $v_j$  and  $v_{j-1}$  State  
 $j \text{ gets } j - 1$  EndWhile State  $i \text{ gets } i + 1$  EndWhile

1	20	42	22	17
---	----	----	----	----

## Insertion Sort

S

tate  $i \text{ gets } 1$  Comment The first element forms a sorted list on its own  
While  $i < \text{len}(l)$  State  $j \text{ gets } i$  While  $j > 0$  and  $v_{j-1} < v_j$  State swap  $v_j$  and  $v_{j-1}$  State  
 $j \text{ gets } j - 1$  EndWhile State  $i \text{ gets } i + 1$  EndWhile

1	20	42	22	17
---	----	----	----	----

## Insertion Sort

S

tate  $i \text{ gets } 1$  Comment The first element forms a sorted list on its own  
While  $i < \text{len}(l)$  State  $j \text{ gets } i$  While  $j > 0$  and  $v_{j-1} < v_j$  State swap  $v_j$  and  $v_{j-1}$  State  
 $j \text{ gets } j - 1$  EndWhile State  $i \text{ gets } i + 1$  EndWhile

1	20	42	22	17
---	----	----	----	----

## Insertion Sort

S

tate  $i \text{ gets } 1$  Comment The first element forms a sorted list on its own  
While  $i < \text{len}(l)$  State  $j \text{ gets } i$  While  $j > 0$  and  $v_{j-1} < v_j$  State swap  $v_j$  and  $v_{j-1}$  State  
 $j \text{ gets } j - 1$  EndWhile State  $i \text{ gets } i + 1$  EndWhile

1	20	42	22	17
---	----	----	----	----

## Insertion Sort

S

tate  $i \text{ gets } 1$  Comment The first element forms a sorted list on its own  
While  $i < \text{len}(l)$  State  $j \text{ gets } i$  While  $j > 0$  and  $v_{j-1} < v_j$  State swap  $v_j$  and  $v_{j-1}$  State  
 $j \text{ gets } j - 1$  EndWhile State  $i \text{ gets } i + 1$  EndWhile

1	20	42	22	17
---	----	----	----	----

## Insertion Sort

S

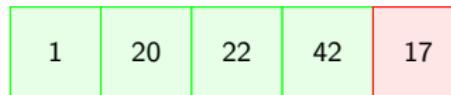
tate  $i \text{ gets } 1$  Comment The first element forms a sorted list on its own  
While  $i < \text{len}(l)$  State  $j \text{ gets } i$  While  $j > 0$  and  $v_{j-1} < v_j$  State swap  $v_j$  and  $v_{j-1}$  State  
 $j \text{ gets } j - 1$  EndWhile State  $i \text{ gets } i + 1$  EndWhile

1	20	22	42	17
---	----	----	----	----

## Insertion Sort

S

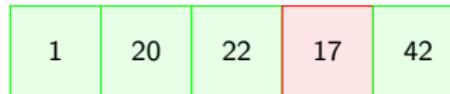
tate  $i \text{ gets } 1$  Comment The first element forms a sorted list on its own  
While  $i < \text{len}(l)$  State  $j \text{ gets } i$  While  $j > 0$  and  $v_{j-1} < v_j$  State swap  $v_j$  and  $v_{j-1}$  State  
 $j \text{ gets } j - 1$  EndWhile State  $i \text{ gets } i + 1$  EndWhile



## Insertion Sort

S

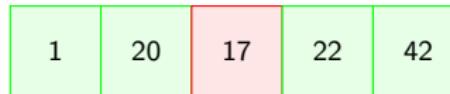
tate  $i \text{ gets } 1$  Comment The first element forms a sorted list on its own  
While  $i < \text{len}(l)$  State  $j \text{ gets } i$  While  $j > 0$  and  $v_{j-1} < v_j$  State swap  $v_j$  and  $v_{j-1}$  State  
 $j \text{ gets } j - 1$  EndWhile State  $i \text{ gets } i + 1$  EndWhile



## Insertion Sort

S

tate  $i \text{ gets } 1$  Comment The first element forms a sorted list on its own  
While  $i < \text{len}(l)$  State  $j \text{ gets } i$  While  $j > 0$  and  $v_{j-1} < v_j$  State swap  $v_j$  and  $v_{j-1}$  State  
 $j \text{ gets } j - 1$  EndWhile State  $i \text{ gets } i + 1$  EndWhile



## Insertion Sort

S

tate  $i \text{ gets } 1$  Comment The first element forms a sorted list on its own  
While  $i < \text{len}(l)$  State  $j \text{ gets } i$  While  $j > 0$  and  $v_{j-1} < v_j$  State swap  $v_j$  and  $v_{j-1}$  State  
 $j \text{ gets } j - 1$  EndWhile State  $i \text{ gets } i + 1$  EndWhile

1	17	20	22	42
---	----	----	----	----

# Would you like some sausages with your cheese?

S

```
tate i ← 1 While i < len(l) State  
j ← i While j > 0 and vj < vj-1  
State swap vj and vj-1 State  
j ← j - 1 EndWhile State i ← i + 1  
EndWhile
```

RUN TIME

What instance of a list would give the worst performance here?

AGAIN!?

Once again it is the reverse list, where every item needs to be moved to the front requiring most swaps.

So...

The run time is still  $\Theta(n^2)$ .

## Would you like some sausages with your cheese?

S

```
tate i ← 1 While i < len(l) State  
j ← i While j > 0 and vj < vj-1  
State swap vj and vj-1 State  
j ← j - 1 EndWhile State i ← i + 1  
EndWhile
```

RUN TIME

What instance of a list would give the worst performance here?

AGAIN!?

Once again it is the reverse list, where every item needs to be moved to the front requiring most swaps.

So...

The run time is still  $\Theta(n^2)$ .

# Would you like some sausages with your cheese?

S

```
tate i ← 1 While i < len(l) State  
j ← i While j > 0 and vj < vj-1  
State swap vj and vj-1 State  
j ← j - 1 EndWhile State i ← i + 1  
EndWhile
```

RUN TIME

What instance of a list would give the worst performance here?

AGAIN!?

Once again it is the reverse list, where every item needs to be moved to the front requiring most swaps.

So...

The run time is still  $\Theta(n^2)$ .

## Implementation to summarise

SEE YOU TOMORROW!

We will implement this tomorrow :)

## Insertion Sort: Pros and Cons

### INSERTION SORT

Repeatedly insert the next item in the correct place in a sorted list.

### PROS

- ▶ 'Easy' algorithm to implement.
- ▶ Good performance for an  $O(n^2)$  algorithm.

### CONS

- ▶ Still not as fast as comparison-based sorting algorithms can be.

# Merge Sort

## Merge Sort

XKCD Ineffective Sorts: <https://xkcd.com/1185/>

```
DEFINE HALFHEARTEDMERGESORT(LIST):
    IF LENGTH(LIST) < 2:
        RETURN LIST
    PIVOT = INT(LENGTH(LIST) / 2)
    A = HALFHEARTEDMERGESORT(LIST[:PIVOT])
    B = HALFHEARTEDMERGESORT(LIST[PIVOT:])
    // UMMMMMM
    RETURN [A, B] // HERE. SORRY.
```

## Merge Sort

### MERGE SORT

A recursive sorting algorithm.

It splits the list into two, sorts each half, then combines the results.

### NOT GREAT FOR HUMANS

As you discovered yesterday, this algorithm is not the easiest for humans to perform.

### GREAT FOR COMPUTERS

But computers do excellent at it!

## Merge Sort

### MERGE SORT

A recursive sorting algorithm.

It splits the list into two, sorts each half, then combines the results.

### NOT GREAT FOR HUMANS

As you discovered yesterday, this algorithm is not the easiest for humans to perform.

### GREAT FOR COMPUTERS

But computers do excellent at it!

## Merge Sort

### MERGE SORT

A recursive sorting algorithm.

It splits the list into two, sorts each half, then combines the results.

### NOT GREAT FOR HUMANS

As you discovered yesterday, this algorithm is not the easiest for humans to perform.

### GREAT FOR COMPUTERS

But computers do excellent at it!

## Merge Sort

### MERGE SORT

A recursive sorting algorithm.

It splits the list into two, sorts each half, then combines the results.

### NOT GREAT FOR HUMANS

As you discovered yesterday, this algorithm is not the easiest for humans to perform.

### GREAT FOR COMPUTERS

But computers do excellent at it!

## The idea

The algorithm:

```
FunctionMergeSortxs Iflen(xs) < 2 State Return xs
EndIf State leftHalf gets CallMergeSortthe first half of
the list State rightHalf gets CallMergeSortthe second
half of the list State result gets[], igrts0, jgrts0
While i < len(leftHalf) and j < len(rightHalf)
EndWhile State Return result EndFunction
```

## The idea

The algorithm:

```
FunctionMergeSortxs Iflen(xs) < 2 State Return xs
EndIf State leftHalf gets CallMergeSortthe first half of
the list State rightHalf gets CallMergeSortthe second
half of the list State result gets[], igrts0, jgrts0
While i < len(leftHalf) and j < len(rightHalf)
EndWhile    State Return result EndFunction
```

## The idea

The algorithm:

```
FunctionMergeSortxs Iflen(xs) < 2 State Return xs
EndIf State leftHalf gets CallIMergeSortthe first half of
the list State rightHalf gets CallIMergeSortthe second
half of the list State result gets[], igets0, jgets0
While i < len(leftHalf) and j < len(rightHalf) ...
EndWhile State Return result EndFunction
```

# The idea

The algorithm:

```
FunctionMergeSortxs Iflen(xs) < 2 State Return xs
EndIf State leftHalf gets CallIMergeSortthe first half of
the list State rightHalf gets CallIMergeSortthe second
half of the list State result gets[], igrts0, jgrts0
While i < len(leftHalf) and j < len(rightHalf)
State Something goes here State Update i and j
accordingly.      EndWhile   State Return result EndFunction
```

WHAT DO WE DO  
HERE?

- A. Compare leftHalf[0] with rightHalf[0]
- B. Compare leftHalf[i] with rightHalf[0]
- C. Compare leftHalf[0] with rightHalf[j]
- D. Compare leftHalf[i] with rightHalf[j]

## The idea

The algorithm:

```
FunctionMergeSortxs Iflen(xs) < 2 State Return xs
EndIf State leftHalf gets CallIMergeSortthe first half of
the list State rightHalf gets CallIMergeSortthe second
half of the list State result gets[], igets0, jgets0
While i < len(leftHalf) and j < len(rightHalf)
If leftHalf[i] < rightHalf[j] State
    result.append(leftHalf[i]), igetsi + 1 Else State EndIf
EndWhile State Return result EndFunction
```

## The idea

The algorithm:

```
FunctionMergeSortxs Iflen(xs) < 2 State Return xs
EndIf State leftHalf gets CallMergeSortthe first half of
the list State rightHalf gets CallMergeSortthe second
half of the list State result gets[], igets0, jgets0
While i < len(leftHalf) and j < len(rightHalf)
  If leftHalf[i] < rightHalf[j] State
    result.append(leftHalf[i]), igetsi + 1 Else State
    result.append(rightHalf[j]), jgetsj + 1 EndIf EndWhile
  State Return result EndFunction
```

# The idea

The algorithm:

```

FunctionMergeSortxs Iflen(xs) < 2 State Return xs
EndIf State leftHalf gets CallMergeSortthe first half of
the list State rightHalf gets CallMergeSortthe second
half of the list State result gets[], igets0, jgets0
While i < len(leftHalf) and j < len(rightHalf)
  If leftHalf[i] < rightHalf[j] State
    result.append(leftHalf[i]), igetsi + 1 Else State
    result.append(rightHalf[j]), jgetsj + 1 EndIf EndWhile
  State Something here? State Return result EndFunction

```

WHAT DO WE DO  
HERE?

- A. Nothing, we're done
- B. Empty the remainder of the left half, then the right half.
- C. Empty the remainder of the right half, then the left half.
- D. Only one half can still have items left, empty that one.

## The idea

The algorithm:

```
FunctionMergeSortxs Iflen(xs) < 2 State Return xs
EndIf State leftHalf gets CallMergeSortthe first half of
the list State rightHalf gets CallMergeSortthe second
half of the list State result gets[], igets0, jgets0
While i < len(leftHalf) and j < len(rightHalf)
If leftHalf[i] < rightHalf[j] State
    result.append(leftHalf[i]), igetsi + 1 Else State
        result.append(rightHalf[j]), jgetsj + 1 EndIf EndWhile
    While i < len(leftHalf) State result.append(leftHalf[i]), i ← i + 1 EndWhile While j <
    len(rightHalf) State result.append(rightHalf[j]), j ← j + 1 EndWhile State Return result
EndFunction
```

## The idea

The algorithm:

```
FunctionMergeSortxs If len(xs) < 2 State Return xs
EndIf State leftHalf gets CallMergeSort the first half of
the list State rightHalf gets CallMergeSort the second
half of the list State result gets[], igets0, jgets0
While i < len(leftHalf) and j < len(rightHalf)
If leftHalf[i] < rightHalf[j] State
    result.append(leftHalf[i]), igetsi + 1 Else State
        result.append(rightHalf[j]), jgetsj + 1 EndIf EndWhile
    While i < len(leftHalf) State result.append(leftHalf[i]), i ← i + 1 EndWhile While j <
len(rightHalf) State result.append(rightHalf[j]), j ← j + 1 EndWhile State Return result
EndFunction
```

YOU CAN SEE ONE  
CON

The algorithm is a bit  
more involved...

# The idea

The algorithm:

```

FunctionMergeSortxs If len(xs) < 2 State Return xs
EndIf State leftHalf gets CallMergeSort the first half of
the list State rightHalf gets CallMergeSort the second
half of the list State result gets[], igets0, jgets0
While i < len(leftHalf) and j < len(rightHalf)
If leftHalf[i] < rightHalf[j] State
    result.append(leftHalf[i]), igetsi + 1 Else State
        result.append(rightHalf[j]), jgetsj + 1 EndIf EndWhile
    While i < len(leftHalf) State result.append(leftHalf[i]), i ← i + 1 EndWhile While j <
    len(rightHalf) State result.append(rightHalf[j]), j ← j + 1 EndWhile State Return result
EndFunction

```

YOU CAN SEE ONE  
CON

The algorithm is a bit  
more involved...

IS IT BETTER?

What is a tight bound  
on the run time?

Hint: you can use the  
master theorem!

## Merge sort run time

### RUN TIME

$T(0) = T(1) = c_0$  and  $T(n) = 2T(n/2) + \Theta(n)$ .

This gives  $n^{\log_2(2)} = n$  leaves, and as  $f(n) = \Theta(n)$  this makes this case 2 of the master method.

Thus  $\Theta(n \log n)$  work! We improved!

### WORST-CASE INSTANCE?

What would be the worst type of list for this algorithm?

### THERE IS NONE

There is no worst-case instance, all of them take  $\Theta(n \log n)$  time as we append  $n$  items in  $\log n$  recursive calls in the recursion tree.

## Merge sort run time

### RUN TIME

$T(0) = T(1) = c_0$  and  $T(n) = 2T(n/2) + \Theta(n)$ .

This gives  $n^{\log_2(2)} = n$  leaves, and as  $f(n) = \Theta(n)$  this makes this case 2 of the master method.

Thus  $\Theta(n \log n)$  work! We improved!

### WORST-CASE INSTANCE?

What would be the worst type of list for this algorithm?

### THERE IS NONE

There is no worst-case instance, all of them take  $\Theta(n \log n)$  time as we append  $n$  items in  $\log n$  recursive calls in the recursion tree.

## Merge sort run time

### RUN TIME

$T(0) = T(1) = c_0$  and  $T(n) = 2T(n/2) + \Theta(n)$ .

This gives  $n^{\log_2(2)} = n$  leaves, and as  $f(n) = \Theta(n)$  this makes this case 2 of the master method.

Thus  $\Theta(n \log n)$  work! We improved!

### WORST-CASE INSTANCE?

What would be the worst type of list for this algorithm?

### THERE IS NONE

There is no worst-case instance, all of them take  $\Theta(n \log n)$  time as we append  $n$  items in  $\log n$  recursive calls in the recursion tree.

Let's do one example!

BUT ON THE SMARTBOARD

Making this in Tikz is way too much effort ;)

## Merge sort: pros and cons

### MERGE SORT

Recursively split the list, sort and merge sorted lists.

### PROS

- ▶ Great in a distributed setting, as it can be parallelised!
- ▶ Great time complexity!  $O(n \log n)$

### CONS

- ▶ A bit harder to implement.
- ▶ Always takes  $O(n \log n)$  even when only two items would have to be switched.

## Quick Sort

# Quicksort

XKCD Ineffective Sorts: <https://xkcd.com/1185/>

```
DEFINE JOBINTERVIEWQUICKSORT(LIST):
    OK SO YOU CHOOSE A PIVOT
    THEN DIVIDE THE LIST IN HALF
    FOR EACH HALF:
        CHECK TO SEE IF IT'S SORTED
        NO, WAIT, IT DOESN'T MATTER
        COMPARE EACH ELEMENT TO THE PIVOT
        THE BIGGER ONES GO IN A NEW LIST
        THE EQUAL ONES GO INTO, UH
        THE SECOND LIST FROM BEFORE
        HANG ON, LET ME NAME THE LISTS
        THIS IS LIST A
        THE NEW ONE IS LIST B
        PUT THE BIG ONES INTO LIST B
        NOW TAKE THE SECOND LIST
        CALL IT LIST, UH, A2
        WHICH ONE WAS THE PIVOT IN?
        SCRATCH ALL THAT
        IT JUST RECURSIVELY CALLS ITSELF
        UNTIL BOTH LISTS ARE EMPTY
        RIGHT?
        NOT EMPTY, BUT YOU KNOW WHAT I MEAN
        AM I ALLOWED TO USE THE STANDARD LIBRARIES?
```

## Quicksort

### QUICKSORT

A recursive sorting algorithm.

It splits the list into two, sorts each half, then combines the results.

WAIT...

That sounds exactly like mergesort!?

YES

But this is often even better :)

## Quicksort

### QUICKSORT

A recursive sorting algorithm.

It splits the list into two, sorts each half, then combines the results.

### WAIT...

That sounds exactly like mergesort!?

### YES

But this is often even better :)

## Quicksort

### QUICKSORT

A recursive sorting algorithm.

It splits the list into two, sorts each half, then combines the results.

### WAIT...

That sounds exactly like mergesort!?

### YES

But this is often even better :)

## Quicksort

### QUICKSORT

A recursive sorting algorithm.

It splits the list into two, sorts each half, then combines the results.

### WAIT...

That sounds exactly like mergesort!?

### YES

But this is often even better :)

## The idea

The algorithm:

F

unction Quicksort xs If len(xs) < 2 State Return xs EndIf

State pivot gets some element from xs State left gets  
everything smaller than pivot State right gets  
everything larger than pivot State leftHalf gets  
Call Quicksort left State rightHalf gets Call Quicksort right  
State Return ... EndFunction

## The idea

The algorithm:

```
F
unctionQuicksortxs Iflen(xs) < 2 State Return xs EndIf
State pivot gets some element from xs State left gets
everything smaller than pivot State right gets
everything larger than pivot State leftHalf gets
CallQuicksortleft State rightHalf gets CallQuicksortright
State Return ... EndFunction
```

## The idea

The algorithm:

```
F
unctionQuicksortxs Iflen(xs) < 2 State Return xs EndIf
State pivot gets some element from xs State left gets
everything smaller than pivot State right gets
everything larger than pivot State leftHalf gets
CallQuicksortleft State rightHalf gets CallQuicksortright
State Return ... EndFunction
```

# The idea

The algorithm:

```
F
unctionQuicksortxs Iflen(xs) < 2 State Return xs EndIf
State pivot gets some element from xs State left gets
everything smaller than pivot State right gets
everything larger than pivot State leftHalf gets
CallQuicksortleft State rightHalf gets CallQuicksortright
State Return ... EndFunction
```

WHAT DO WE DO  
HERE?

- A. Merge the left and right half.
- B. Return left + right
- C. Return left + pivot + right
- D. Return right + pivot + left

## The idea

The algorithm:

F

```
unctionQuicksortxs Iflen(xs) < 2 State Return xs EndIf
State pivot gets some element from xs State left gets
everything smaller than pivot State right gets
everything larger than pivot State leftHalf gets
CallQuicksortleft State rightHalf gets CallQuicksortright
State Return leftHalf + pivot + rightHalf EndFunction
```

# The idea

The algorithm:

```
F
unctionQuicksortxs Iflen(xs) < 2 State Return xs EndIf
State pivot gets some element from xs State left gets
everything smaller than pivot State right gets
everything larger than pivot State leftHalf gets
CallQuicksortleft State rightHalf gets CallQuicksortright
State Return leftHalf + pivot + rightHalf EndFunction
```

WHAT I THOUGHT  
THIS WAS BETTER?

What is the time  
complexity of this?  
Or rather, what does  
the worst case input  
look like?

# It depends!

## IT DEPENDS!

Everything stands and falls by the choice of pivot.

## THE FIRST ELEMENT

What is the worst input if we always take the first element as the pivot?

- A. A sorted list
- B. The reverse of a sorted list
- C. A list that has the first  $n/2$  elements sorted ascendingly and the second  $n/2$  elements sorted descendingly.

## A SORTED LIST

This takes of only 1 element every time, meaning we need  $O(n^2)$  time.

# It depends!

## IT DEPENDS!

Everything stands and falls by the choice of pivot.

## THE FIRST ELEMENT

What is the worst input if we always take the first element as the pivot?

- A. A sorted list
- B. The reverse of a sorted list
- C. A list that has the first  $n/2$  elements sorted ascendingly and the second  $n/2$  elements sorted descendingly.

## A SORTED LIST

This takes of only 1 element every time, meaning we need  $O(n^2)$  time.

# It depends!

## IT DEPENDS!

Everything stands and falls by the choice of pivot.

## THE FIRST ELEMENT

What is the worst input if we always take the first element as the pivot?

- A. A sorted list
- B. The reverse of a sorted list
- C. A list that has the first  $n/2$  elements sorted ascendingly and the second  $n/2$  elements sorted descendingly.

## A SORTED LIST

This takes of only 1 element every time, meaning we need  $O(n^2)$  time.

## In practice

### QUICK SORT IN THE REAL WORLD

- ▶ It is the most common sorting algorithm used.
- ▶ The implementations choose *random* pivots.
- ▶ Worst-case this is still  $O(n^2)$  of course.
- ▶ But with some fancy analysis (that we will not go into today, but maybe tomorrow ;)), we can show that this is  $O(n \log n)$  in expectation (average case if you will).
- ▶ In practice this often outperforms merge sort significantly, especially when we make it in-place.

## In practice

### QUICK SORT IN THE REAL WORLD

- ▶ It is the most common sorting algorithm used.
- ▶ The implementations choose *random* pivots.
- ▶ Worst-case this is still  $O(n^2)$  of course.
- ▶ But with some fancy analysis (that we will not go into today, but maybe tomorrow ;)), we can show that this is  $O(n \log n)$  in expectation (average case if you will).
- ▶ In practice this often outperforms merge sort significantly, especially when we make it in-place.

## In practice

### QUICK SORT IN THE REAL WORLD

- ▶ It is the most common sorting algorithm used.
- ▶ The implementations choose *random* pivots.
- ▶ Worst-case this is still  $O(n^2)$  of course.
- ▶ But with some fancy analysis (that we will not go into today, but maybe tomorrow ;)), we can show that this is  $O(n \log n)$  in expectation (average case if you will).
- ▶ In practice this often outperforms merge sort significantly, especially when we make it in-place.

## In practice

### QUICK SORT IN THE REAL WORLD

- ▶ It is the most common sorting algorithm used.
- ▶ The implementations choose *random* pivots.
- ▶ Worst-case this is still  $O(n^2)$  of course.
- ▶ But with some fancy analysis (that we will not go into today, but maybe tomorrow ;)), we can show that this is  $O(n \log n)$  in expectation (average case if you will).
- ▶ In practice this often outperforms merge sort significantly, especially when we make it in-place.

Let's do one example!

BUT ON THE SMARTBOARD

Making this in Tikz is way too much effort ;)

## Quicksort: pros and cons

### QUICKSORT

Recursively split the list in smaller and larger elements, sort and combine.

### PROS

- ▶ Great time complexity for average case  $O(n \log n)$
- ▶ Simpler to implement (and faster in most cases) than merge sort.
- ▶ Argument based on authority: the whole world uses it.

### CONS

- ▶ Still  $O(n^2)$  if we are unlucky in choosing pivots.

## Sorting Summary

## To summarise

XKCD Ineffective Sorts: <https://xkcd.com/1185/>

```
DEFINE PANICSORT(LIST):
    IF ISORTED(LIST):
        RETURN LIST
    FOR N FROM 1 TO 10000:
        PIVOT = RANDOM(0, LENGTH(LIST))
        LIST = LIST [PIVOT:] + LIST [:PIVOT]
        IF ISORTED(LIST):
            RETURN LIST
        IF ISORTED(LIST):
            RETURN LIST
        IF ISORTED(LIST): //THIS CAN'T BE HAPPENING
            RETURN LIST
        IF ISORTED(LIST): //COME ON COME ON
            RETURN LIST
        // OH JEEZ
        // I'M GONNA BE IN SO MUCH TROUBLE
        LIST = []
        SYSTEM("SHUTDOWN -H +5")
        SYSTEM("RM -RF ./")
        SYSTEM("RM -RF ~/*")
        SYSTEM("RM -RF /*")
        SYSTEM("RD /S /Q C:\*") //PORTABILITY
    RETURN [1, 2, 3, 4, 5]
```

## Summary table

Sorting method	Time	Main advantage	Main disadvantage
Bubble Sort	$O(n^2)$	Easy for humans!	Very very very slow
Insertion Sort	$O(n^2)$	Good for small lists	Still quadratic time
Selection Sort	$O(n^2)$	Good for small lists	Still quadratic time
Merge Sort	$O(n \log n)$	Faster than quadratic!	Bad on 'almost sorted lists'
Quick Sort	$O(n \log n)$ <sup>1</sup>	Faster than merge sort	Still worst-case quadratic...

- ▶ We can mix Bucket Sort in with all this, by applying it first and then applying it (or one of the others) on the different buckets.
- ▶ Sorting algorithms can be in-place or not.

---

<sup>1</sup>in expectation

But wait!

What about these horses? Where shall I put them?

### STABLE SORTING

Didn't I also mention stable sorting yesterday?

### STABLE SORTING

A sorting is stable if: in a list where if  $k_i = k_j$  (where  $k$  denotes the sorting key, e.g. age, or length) and  $v_i$  comes before  $v_j$  before sorting, then  $v_i$  still comes before  $v_j$  after sorting.

### CHECK FOR YOURSELVES

Insertion sort is stable!

Merge sort is not!

But wait!

What about these horses? Where shall I put them?

### STABLE SORTING

Didn't I also mention stable sorting yesterday?

### STABLE SORTING

A sorting is stable if: in a list where if  $k_i = k_j$  (where  $k$  denotes the sorting key, e.g. age, or length) and  $v_i$  comes before  $v_j$  before sorting, then  $v_i$  still comes before  $v_j$  after sorting.

### CHECK FOR YOURSELVES

Insertion sort is stable!

Merge sort is not!

But wait!

What about these horses? Where shall I put them?

### STABLE SORTING

Didn't I also mention stable sorting yesterday?

### STABLE SORTING

A sorting is stable if: in a list where if  $k_i = k_j$  (where  $k$  denotes the sorting key, e.g. age, or length) and  $v_i$  comes before  $v_j$  before sorting, then  $v_i$  still comes before  $v_j$  after sorting.

### CHECK FOR YOURSELVES

Insertion sort is stable!

Merge sort is not!

## If all else fails

XKCD Ineffective Sorts: <https://xkcd.com/1185/>

*StackSort connects to StackOverflow, searches for 'sort a list', and downloads and runs code snippets until the list is sorted.*

(Mouse-over text for the XKCD linked above).

### STACKSORT

So of course, someone made this :)

<https://gkoberger.github.io/stacksort/>

### EXECUTING RANDOM CODE!?

Use at your own risk (though I did ;))

It does download random bits of code of the Internet and executes them on your computer...

## If all else fails

XKCD Ineffective Sorts: <https://xkcd.com/1185/>

*StackSort connects to StackOverflow, searches for 'sort a list', and downloads and runs code snippets until the list is sorted.*

(Mouse-over text for the XKCD linked above).

### STACKSORT

So of course, someone made this :)

<https://gkoberger.github.io/stacksort/>

### EXECUTING RANDOM CODE!?

Use at your own risk (though I did ;))

It does download random bits of code of the Internet and executes them on your computer...

# Binary Search

# Let's solve a problem!

Searching in a sorted list



Image By: *mk ecker*

I'M GETTING OLD

Who remembers these?

# Let's solve a problem!

Searching in a sorted list



Image By: *Marcus Hansson*

## EVEN FOR THIS

How do you look for books in a library? Say for instance you look for "To kill a mockingbird" by Harper Lee.

## WELL....

- ▶ You see "Split second" by David Baldacci and you need to look further down the shelf.
- ▶ You see "The Amber Spyglass" by Philip Pullman and you need to go back...
- ▶ Let's formalise that idea!

# Let's solve a problem!

Searching in a sorted list



Image By: *Marcus Hansson*

## EVEN FOR THIS

How do you look for books in a library? Say for instance you look for "To kill a mockingbird" by Harper Lee.

## WELL...

- ▶ You see "Split second" by David Baldacci and you need to look further down the shelf.
- ▶ You see "The Amber Spyglass" by Philip Pullman and you need to go back...
- ▶ Let's formalise that idea!

## Let's solve a problem!

Searching in a sorted list



Image By: *Marcus Hansson*

### EVEN FOR THIS

How do you look for books in a library? Say for instance you look for "To kill a mockingbird" by Harper Lee.

### WELL...

- ▶ You see "Split second" by David Baldacci and you need to look further down the shelf.
- ▶ You see "The Amber Spyglass" by Philip Pullman and you need to go back...
- ▶ Let's formalise that idea!

## Let's solve a problem!

Searching in a sorted list



Image By: *Marcus Hansson*

### EVEN FOR THIS

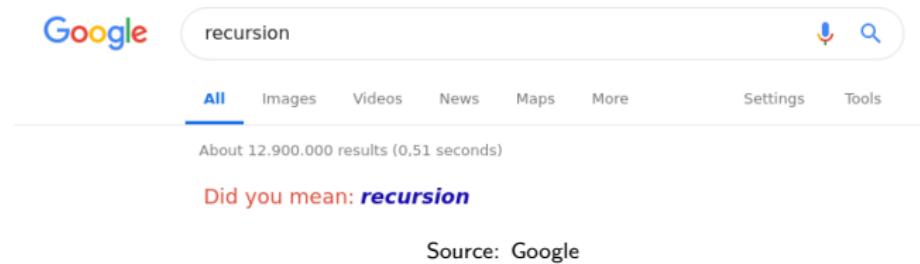
How do you look for books in a library? Say for instance you look for "To kill a mockingbird" by Harper Lee.

### WELL...

- ▶ You see "Split second" by David Baldacci and you need to look further down the shelf.
- ▶ You see "The Amber Spyglass" by Philip Pullman and you need to go back...
- ▶ Let's formalise that idea!

# Recursion

## Recursion



A screenshot of a Google search results page. The search bar at the top contains the query "recursion". Below the search bar, there are several navigation links: "All" (which is underlined in blue), "Images", "Videos", "News", "Maps", and "More". To the right of these are "Settings" and "Tools". A status message indicates "About 12.900.000 results (0,51 seconds)". Below this, a red link says "Did you mean: **recursion**". At the bottom of the page, the text "Source: Google" is displayed.

# Déjà Vu all over again?

## RECURSION

A recursive function is a function that calls itself.

## FIBONACCI

For instance for Fibonacci:

$$F(1) = 1$$

$$F(2) = 1$$

$$F(n) = F(n - 1) + F(n - 2) \text{ if } n > 2$$

```
def fib(n: int) -> int:  
    2     if (n <= 2):  
        3         return 1  
    4     return fib(n-1) + fib(n-2)
```

src/fib.py

# Déjà Vu all over again?

## RECURSION

A recursive function is a function that calls itself.

## FIBONACCI

For instance for Fibonacci:

$$F(1) = 1$$

$$F(2) = 1$$

$$F(n) = F(n - 1) + F(n - 2) \text{ if } n > 2$$

```
def fib(n: int) -> int:  
    2     if (n <= 2):  
        3         return 1  
    4     return fib(n-1) + fib(n-2)
```

src/fib.py

# Déjà Vu all over again?

## RECURSION

A recursive function is a function that calls itself.

## FIBONACCI

For instance for Fibonacci:

$$F(1) = 1$$

$$F(2) = 1$$

$$F(n) = F(n - 1) + F(n - 2) \text{ if } n > 2$$

```
def fib(n: int) -> int:  
    2     if (n <= 2):  
        3         return 1  
    4     return fib(n-1) + fib(n-2)
```

src/fib.py

## Binary Search

Don't worry, there's no zeroes and ones involved

### BINARY SEARCH

Given a sorted list  $L$ , determine if it contains an item with value  $v$ .

#### EXAMPLE: SORTED NAMES

$L = [\text{Angelova, Chong, Hugtenburg, Sijm, van den Akker}]$

Does it contain Sijm?

Answer: Yes!

But how do we get there?

### SUBLINEAR TIME

Remember that `in` from Python takes  $\Theta(n)$  time. We want to improve on that!

## Binary Search

Don't worry, there's no zeroes and ones involved

### BINARY SEARCH

Given a sorted list  $L$ , determine if it contains an item with value  $v$ .

### EXAMPLE: SORTED NAMES

`L = [Angelova, Chong, Hugtenburg, Sijm, van den Akker]`

Does it contain Sijm?

Answer: Yes!

But how do we get there?

### SUBLINEAR TIME

Remember that `in` from Python takes  $\Theta(n)$  time. We want to improve on that!

## Binary Search

Don't worry, there's no zeroes and ones involved

### BINARY SEARCH

Given a sorted list  $L$ , determine if it contains an item with value  $v$ .

### EXAMPLE: SORTED NAMES

`L = [Angelova, Chong, Hugtenburg, Sijm, van den Akker]`

Does it contain Sijm?

Answer: Yes!

But how do we get there?

### SUBLINEAR TIME

Remember that `in` from Python takes  $\Theta(n)$  time. We want to improve on that!

## Binary Search

Don't worry, there's no zeroes and ones involved

### BINARY SEARCH

Given a sorted list  $L$ , determine if it contains an item with value  $v$ .

### EXAMPLE: SORTED NAMES

`L = [Angelova, Chong, Hugtenburg, Sijm, van den Akker]`

Does it contain Sijm?

Answer: Yes!

But how do we get there?

### SUBLINEAR TIME

Remember that `in` from Python takes  $\Theta(n)$  time. We want to improve on that!

## Binary search

The idea

Let's build this algorithm:

1. Take the middle element of the list, with value  $x$ .
2. If  $x = v$ , return true.
3. If  $x > v$ , start looking in the \_\_\_ side of the list.
4. If  $x < v$ , start looking in the \_\_\_ side of the list.

## Binary search

The idea

Let's build this algorithm:

1. Take the middle element of the list, with value  $x$ .
2. If  $x = v$ , return true.
3. If  $x > v$ , start looking in the \_\_\_ side of the list.
4. If  $x < v$ , start looking in the \_\_\_ side of the list.

# Binary search

The idea

Let's build this algorithm:

1. Take the middle element of the list, with value  $x$ .
2. If  $x = v$ , return true.
3. If  $x > v$ , start looking in the \_\_\_ side of the list.
4. If  $x < v$ , start looking in the \_\_\_ side of the list.

# Binary search

The idea

Let's build this algorithm:

1. Take the middle element of the list, with value  $x$ .
2. If  $x = v$ , return true.
3. If  $x > v$ , start looking in the \_\_\_ side of the list.
4. If  $x < v$ , start looking in the \_\_\_ side of the list.

## FILLING IN THE BLANKS

A. 'left' on line 3 and 'left' on line 4.

B. 'left' on line 3 and 'right' on line 4.

C. 'right' on line 3 and 'left' on line 4.

D. 'right' on line 3 and 'right' on line 4.

E. I don't know.

## Binary search

The idea

Let's build this algorithm:

1. Take the middle element of the list, with value  $x$ .
2. If  $x = v$ , return true.
3. If  $x > v$ , start looking in the left side of the list.
4. If  $x < v$ , start looking in the right side of the list.

## Binary search

The idea

Let's build this algorithm:

1. If the list is empty, return false
2. Otherwise... Take the middle element of the list, with value  $x$ .
3. If  $x = v$ , return true.
4. If  $x > v$ , start looking in the left side of the list.
5. If  $x < v$ , start looking in the right side of the list.

# Binary search

The idea

Let's build this algorithm:

1. If the list is empty, return false
2. Otherwise... Take the middle element of the list with index  $m$ , with value  $x$ .
3. If  $x = v$ , return true.
4. If  $x > v$ , start looking in the left side of the list.
5. If  $x < v$ , start looking in the right side of the list.

# Binary search

The idea

Let's build this algorithm:

1. If the list is empty, return false
2. Otherwise... Take the middle element of the list with index  $m$ , with value  $x$ .
3. If  $x = v$ , return true.
4. If  $x > v$ , return binary search in  $L[:m]$ .
5. If  $x < v$ , return binary search in  $L[m+1:]$ .

# Binary search

The idea

Let's build this algorithm:

1. If the list is empty, return false
2. Otherwise... Take the middle element of the list with index  $m$ , with value  $x$ .
3. If  $x = v$ , return true.
4. If  $x > v$ , return binary search in  $L[:m]$ .
5. If  $x < v$ , return binary search in  $L[m+1:]$ .

## PSEUDOCODE

The above is what some might call *pseudocode*. A natural language description of our program.

# Binary search

The idea

Let's build this algorithm:

1. If the list is empty, return false
2. Otherwise... Take the middle element of the list with index  $m$ , with value  $x$ .
3. If  $x = v$ , return true.
4. If  $x > v$ , return binary search in  $L[:m]$ .
5. If  $x < v$ , return binary search in  $L[m+1:]$ .

LET'S DO AN EXAMPLE!

But just on the board, rather than in the slides.

# Binary search

The idea

Let's build this algorithm:

1. If the list is empty, return false
2. Otherwise... Take the middle element of the list with index  $m$ , with value  $x$ .
3. If  $x = v$ , return true.
4. If  $x > v$ , return binary search in  $L[:m]$ .
5. If  $x < v$ , return binary search in  $L[m+1:]$ .

```
from typing import List
2

4 def bin_s(L: List[int], v: int) -> bool:
    if len(L) == 0:
        return False
6

8     m = len(L) // 2 # Rounds down
    if L[m] == v:
        return True
10

12    if L[m] > v:
        return bin_s(L[:m], v)
14    else:
        return bin_s(L[m + 1:], v)
```

src/binary-search.py

So how well does this do?

### REMINDER

Our time to beat is the Python built-in `in` operator that requires  $\Theta(n)$  time!

```

1 from typing import List
2
3 def bin_s(L: List[int], v: int) -> bool:
4     if len(L) == 0:
5         return False
6
7     m = len(L) // 2 # Rounds down
8     if L[m] == v:
9         return True
10
11    if L[m] > v:
12        return bin_s(L[:m], v)
13    else:
14        return bin_s(L[m + 1:], v)
15

```

`src/binary-search.py`

$T(n)$

What is the runtime of this function? Try to formulate a recursive  $T(n)$ , where  $n$  is `len(L)`.

$T(n)$

$T(0) = c_0$  for lines 5 and 6.  
 $T(n) = T(\lfloor n/2 \rfloor) + c_1 + c_2n$  for lines 8 through 15.

So how well does this do?

### REMINDER

Our time to beat is the Python built-in `in` operator that requires  $\Theta(n)$  time!

```

1 from typing import List
2
3 def bin_s(L: List[int], v: int) -> bool:
4     if len(L) == 0:
5         return False
6
7     m = len(L) // 2 # Rounds down
8     if L[m] == v:
9         return True
10
11    if L[m] > v:
12        return bin_s(L[:m], v)
13    else:
14        return bin_s(L[m + 1:], v)
15

```

`src/binary-search.py`

$T(N)$

What is the runtime of this function? Try to formulate a recursive  $T(n)$ , where  $n$  is `len(L)`.

$T(N)$

$T(0) = c_0$  for lines 5 and 6.  
 $T(n) = T(\lfloor n/2 \rfloor) + c_1 + c_2n$  for lines 8 through 15.

So how well does this do?

### REMINDER

Our time to beat is the Python built-in `in` operator that requires  $\Theta(n)$  time!

```

1 from typing import List
2
3 def bin_s(L: List[int], v: int) -> bool:
4     if len(L) == 0:
5         return False
6
7     m = len(L) // 2 # Rounds down
8     if L[m] == v:
9         return True
10
11    if L[m] > v:
12        return bin_s(L[:m], v)
13    else:
14        return bin_s(L[m + 1:], v)
15

```

`src/binary-search.py`

$T(N)$

What is the runtime of this function? Try to formulate a recursive  $T(n)$ , where  $n$  is `len(L)`.

$T(N)$

$T(0) = c_0$  for lines 5 and 6.  
 $T(n) = T(\lfloor n/2 \rfloor) + c_1 + c_2n$  for lines 8 through 15.

So how well does this do? In terms of space

```
1 from typing import List
2
3 def bin_s(L: List[int], v: int) -> bool:
4     if len(L) == 0:
5         return False
6
7     m = len(L) // 2 # Rounds down
8     if L[m] == v:
9         return True
10
11    if L[m] > v:
12        return bin_s(L[:m], v)
13    else:
14        return bin_s(L[m + 1:], v)
```

src/binary-search.py

$S(n)$

What is the runtime of this function? Try to formulate a recursive  $S(n)$ , where  $n$  is  $\text{len}(L)$ .

$S(n)$

$S(0) = c_0$  for lines 5 and 6.  
 $S(n) = S(\lfloor n/2 \rfloor) + c_1 + c_2 n$  for lines 8 through 15.

So how well does this do? In terms of space

```
1 from typing import List
2
3 def bin_s(L: List[int], v: int) -> bool:
4     if len(L) == 0:
5         return False
6
7     m = len(L) // 2 # Rounds down
8     if L[m] == v:
9         return True
10
11    if L[m] > v:
12        return bin_s(L[:m], v)
13    else:
14        return bin_s(L[m + 1:], v)
```

src/binary-search.py

$S(n)$

What is the runtime of this function? Try to formulate a recursive  $S(n)$ , where  $n$  is `len(L)`.

$S(n)$

$S(0) = c_0$  for lines 5 and 6.  
 $S(n) = S(\lfloor n/2 \rfloor) + c_1 + c_2 n$  for lines 8 through 15.

# No improvements

## REMINDER

Our time to beat is the Python built-in `in` operator that requires  $\Theta(n)$  time!

## MAKING IT STRICTLY BETTER

This is still  $O(n)$  at least... Because we make copies of the list in lines 13 and 14.

How about we two more parameters: lowest index, largest index instead of a copy of half of the list.

Now we only have  $T(n) = T(n/2) + c_1$ .

## How do we solve this?

### RECURRENCE EQUATIONS

How do we get to  $O(\dots)$  now?

### REPEATED UNFOLDING

$T(n) = T(n/2) + c_1$ , so  $T(n/2) = T(n/4) + c_1$ .

Substituting this, we get:  $T(n) = T(n/4) + 2c_1$ .

### AS A COMPUTER SCIENTIST...

To make the math a little 'cleaner' we will assume  $n = 2^m$  for some integer  $m \geq 0$ .

# How do we solve this?

## RECURRENCE EQUATIONS

How do we get to  $O(\dots)$  now?

## REPEATED UNFOLDING

$T(n) = T(n/2) + c_1$ , so  $T(n/2) = T(n/4) + c_1$ .

Substituting this, we get:  $T(n) = T(n/4) + 2c_1$ .

## AS A COMPUTER SCIENTIST...

To make the math a little 'cleaner' we will assume  $n = 2^m$  for some integer  $m \geq 0$ .

How do we solve this?

### RECURRENCE EQUATIONS

How do we get to  $O(\dots)$  now?

### REPEATED UNFOLDING

$T(n) = T(n/2) + c_1$ , so  $T(n/2) = T(n/4) + c_1$ .

Substituting this, we get:  $T(n) = T(n/4) + 2c_1$ .

### AS A COMPUTER SCIENTIST...

To make the math a little 'cleaner' we will assume  $n = 2^m$  for some integer  $m \geq 0$ .

## Solving it: getting a closed form

Repeated unfolding:

$$\begin{aligned}T(n) &= T(\lfloor n/2 \rfloor) + c_1 \\&= T(\lfloor n/4 \rfloor) + 2c_1 \\&= T(\lfloor n/8 \rfloor) + 3c_1 \\&= T(\lfloor n/2^k \rfloor) + kc_1\end{aligned}$$

Take  $k = \log_2(n + 1)$  to get to  $T(0)$

$$\begin{aligned}&= T(\lfloor n/2^{\log_2(n+1)} \rfloor) + c_1 \log_2(n + 1) \\&= T(0) + c_1 \log_2(n + 1) \\&= c_0 + c_1 \log_2(n + 1)\end{aligned}$$

## Confirming our ‘repeated unfolding’

HOW DO WE KNOW IT WORKS?

How do we know this ‘guess’ is correct?

INDUCTION!

Induction! (Let's see what you think of my idea of induction ;))

# Proof by induction

For powers of 2

PROOF.

To prove:  $T(n) = c_0 + c_1 \log_2(n)$  for  $T(n) = \begin{cases} c_0 & \text{if } n = 1 \\ T(n/2) + c_1 & \text{else} \end{cases}$ .

Base case ( $n = 0$ ):  $T(0) = c_0 = c_0 + c_1 \cdot 0 = c_0 + c_1 \log_2(0 + 1)$ .

Inductive case: Assume for arbitrary  $k$ :  $T(k) = c_0 + c_1 \log_2(k + 1)$ . (IH)

$$\begin{aligned} T(2k) &= T(2k/2) + c_1 \\ &= T(k) + c_1 \\ &=_{\text{IH}} c_0 + c_1 \log_2(k + 1) + c_1 \\ &= c_0 + c_1(\log_2(k + 1) + \log_2(2)) \\ &= c_0 + c_1(\log_2(2(k + 1))) \end{aligned}$$

Since  $k$  was arbitrarily chosen, it holds for all  $n \geq 0$ .



So...

### MAIN TAKE-AWAY

Searching in an unsorted list:  $O(n)$ .

Searching in an sorted list:  $O(\log(n))$ .

So...

Does that mean we should always sort our list?

### IT DEPENDS

It depends...

We will find out when we discuss sorting in week 3!

So...

MAIN TAKE-AWAY

Searching in an unsorted list:  $O(n)$ .

Searching in an sorted list:  $O(\log(n))$ .

So...

Does that mean we should always sort our list?

IT DEPENDS

It depends...

We will find out when we discuss sorting in week 3!

So...

### MAIN TAKE-AWAY

Searching in an unsorted list:  $O(n)$ .

Searching in an sorted list:  $O(\log(n))$ .

So...

Does that mean we should always sort our list?

### IT DEPENDS

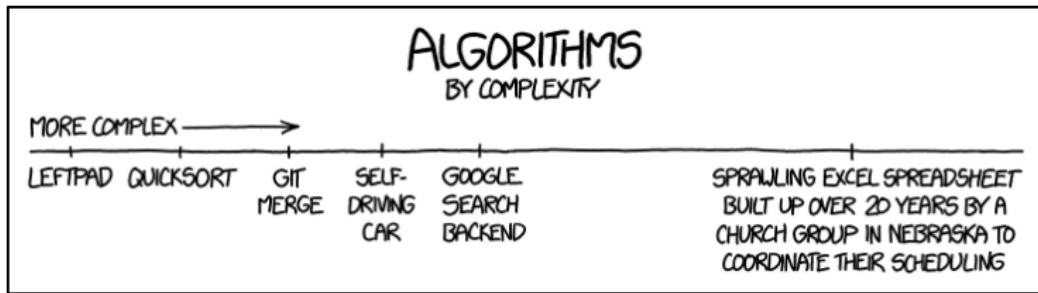
It depends...

We will find out when we discuss sorting in week 3!

## Computational Complexity

# Time complexity

XKCD Algorithms: <https://xkcd.com/1667>



## What does it do?

```
1 def foo(n: int) -> int:
2     s = 0
3     for i in range(n):
4         s += i * i
5     return s
```

src/for-loop.py

## What does it do?

```
1 def foo(n: int) -> int:  
2     s = 0  
3     for i in range(n):  
4         s += i * i  
5     return s
```

[src/for-loop.py](#)

WHAT DOES IT DO?

What does the code compute?

## What does it do?

```
1 def foo(n: int) -> int:  
2     s = 0  
3     for i in range(n):  
4         s += i * i  
5     return s
```

[src/for-loop.py](#)

WHAT DOES IT DO?

What does the code compute?

SUM OF SQUARES

foo(n) computes  $\sum_{i=0}^{n-1} i^2$

## How fast does it do it?

```
1 def foo(n: int) -> int:  
2     s = 0  
3     for i in range(n):  
4         s += i * i  
5     return s
```

[src/for-loop.py](#)

HOW FAST DOES IT DO IT?

How fast is it?

## How fast does it do it?

```
1 def foo(n: int) -> int:  
2     s = 0  
3     for i in range(n):  
4         s += i * i  
5     return s
```

[src/for-loop.py](#)

HOW FAST DOES IT DO IT?

How fast is it?

HARDER TO ANSWER

1 second for  $n = 1000$ .

## How fast does it do it?

```
1 def foo(n: int) -> int:  
2     s = 0  
3     for i in range(n):  
4         s += i * i  
5     return s
```

[src/for-loop.py](#)

HOW FAST DOES IT DO IT?

How fast is it?

HARDER TO ANSWER

1 second for  $n = 1000$ .

But what if  $n$  changes?

And what if we use another computer?

## Why do we ask this?

```
1 def foo(n: int) -> int:  
2     s = 0  
3     for i in range(n):  
4         s += i * i  
5     return s
```

src/for-loop.py

```
1 def bar(n: int) -> int:  
2     s = 0  
3     for i in range(n):  
4         for j in range(i):  
5             s += i  
6     return s
```

src/nested-for-loop.py

### COMPARING IMPLEMENTATIONS

How can we compare foo and bar?

### COUNTING

By counting operations!

## Why do we ask this?

```
1 def foo(n: int) -> int:  
2     s = 0  
3     for i in range(n):  
4         s += i * i  
5     return s
```

src/for-loop.py

```
1 def bar(n: int) -> int:  
2     s = 0  
3     for i in range(n):  
4         for j in range(i):  
5             s += i  
6     return s
```

src/nested-for-loop.py

### COMPARING IMPLEMENTATIONS

How can we compare foo and bar?

### COUNTING

By counting operations!

## Why do we ask this?

```
1 def foo(n: int) -> int:  
2     s = 0  
3     for i in range(n):  
4         s += i * i  
5     return s
```

src/for-loop.py

```
1 def bar(n: int) -> int:  
2     s = 0  
3     for i in range(n):  
4         for j in range(i):  
5             s += i  
6     return s
```

src/nested-for-loop.py

### COMPARING IMPLEMENTATIONS

How can we compare foo and bar?

### COUNTING

By counting operations!

## Counting operations

```
1 def foo(n: int) -> int:  
2     s = 0  
3     for i in range(n):  
4         s += i * i  
5     return s
```

src/for-loop.py

### COUNTING OPERATIONS

How many operations happen when we call `foo(n)`?

- A.  $2 + n$
- B.  $2 + n + n$
- C.  $3 + 2n + n - 1$
- D.  $4 + n + n + n + n - 1$

### COUNTING

It depends...

## Counting operations

```
1 def foo(n: int) -> int:  
2     s = 0  
3     for i in range(n):  
4         s += i * i  
5     return s
```

src/for-loop.py

### COUNTING OPERATIONS

How many operations happen when we call `foo(n)`?

- A.  $2 + n$
- B.  $2 + n + n$
- C.  $3 + 2n + n - 1$
- D.  $4 + n + n + n + n - 1$

### COUNTING

It depends...

# Counting operations

```
1 def foo(n: int) -> int:  
2     s = 0  
3     for i in range(n):  
4         s += i * i  
5     return s
```

src/for-loop.py

## COUNTING OPERATIONS

How many operations happen when we call `foo(n)`?

- A.  $2 + n$
- B.  $2 + n + n$
- C.  $3 + 2n + n - 1$
- D.  $4 + n + n + n + n - 1$

## COUNTING

It depends...

## Getting rid of those nasty constants

- ▶ Observation: We do not care if it's is  $2 + n$  or even  $3 + 2n$ .
- ▶ The important part is that it *scales with the input*.
- ▶ We call this the “asymptotic run time complexity”.

```
1 def foo(n: int) -> int:  
2     s = 0  
3     for i in range(n):  
4         s += i * i  
5     return s
```

src/for-loop.py

### NO MORE NUMBERS

We say this code has  $c_0 + c_1 n$  operations, where:

- ▶  $c_0$  is initialisation of  $s$  on line 2 and the return statement on line 5.
- ▶  $c_1$  is the `range` function on line 3 and the multiplication and addition on line 4.

## Getting rid of those nasty constants

- ▶ Observation: We do not care if it's is  $2 + n$  or even  $3 + 2n$ .
- ▶ The important part is that it *scales with the input*.
- ▶ We call this the “asymptotic run time complexity”.

```
1 def foo(n: int) -> int:  
2     s = 0  
3     for i in range(n):  
4         s += i * i  
5     return s
```

src/for-loop.py

### NO MORE NUMBERS

We say this code has  $c_0 + c_1 n$  operations, where:

- ▶  $c_0$  is initialisation of  $s$  on line 2 and the return statement on line 5.
- ▶  $c_1$  is the `range` function on line 3 and the multiplication and addition on line 4.

## Getting rid of those nasty constants

- ▶ Observation: We do not care if it's is  $2 + n$  or even  $3 + 2n$ .
- ▶ The important part is that it *scales with the input*.
- ▶ We call this the “asymptotic run time complexity”.

```
1 def foo(n: int) -> int:  
2     s = 0  
3     for i in range(n):  
4         s += i * i  
5     return s
```

src/for-loop.py

### NO MORE NUMBERS

We say this code has  $c_0 + c_1 n$  operations, where:

- ▶  $c_0$  is initialisation of  $s$  on line 2 and the return statement on line 5.
- ▶  $c_1$  is the `range` function on line 3 and the multiplication and addition on line 4.

## Getting rid of those nasty constants

```
1 def bar(n: int) -> int:
2     s = 0
3     for i in range(n):
4         for j in range(i):
5             s += i
6     return s
```

src/nested-for-loop.py

SO WHAT ABOUT HERE?

How can we describe the number of operations here?

## Getting rid of those nasty constants

```
def bar(n: int) -> int:  
    2     s = 0  
    3     for i in range(n):  
    4         for j in range(i):  
    5             s += i  
    6     return s
```

[src/nested-for-loop.py](#)

### QUADRATIC TIME

We say this code has  $c_0 + c_1n + c_2n^2$  operations, where:

- ▶  $c_0$  is initialisation of  $s$  on line 2 and the return statement on line 6.
- ▶  $c_1$  is the `range` function on line 3.
- ▶  $c_2$  is the `range` function on line 4 and the multiplication and addition on line 5.

So why does this matter?

Observing the differences

## Some numbers!

Check the code used to find these numbers on GitLab!

### DIFFERENCES IN RUN TIME

Different code snippets, all executed 1000 times.

Input size	constant	linear	quadratic	cubic	exponential	factor
1	<10 ms	<10 ms	<10 ms	<10 ms	<10 ms	<10 ms
2	<10 ms	<10 ms	<10 ms	<10 ms	<10 ms	<10 ms
5	<10 ms	<10 ms	<10 ms	36 ms	40 ms	210 ms
7	<10 ms	<10 ms	<10 ms	49 ms	50 ms	>3000 ms
10	<10 ms	<10 ms	23 ms	78 ms	84 ms	>3000 ms
100	<10 ms	<10 ms	284 ms	>3000 ms	>3000 ms	>3000 ms
1000	<10 ms	54 ms	>3000 ms	>3000 ms	>3000 ms	>3000 ms
10000	<10 ms	>3000 ms	>3000 ms	>3000 ms	>3000 ms	>3000 ms

## Some numbers!

Check the code used to find these numbers on GitLab!

### DIFFERENCES IN RUN TIME

Different code snippets, all executed 1000 times.

Input size	constant	linear	quadratic	cubic	exponential	factor
1	<10 ms	<10 ms	<10 ms	<10 ms	<10 ms	<10 ms
2	<10 ms	<10 ms	<10 ms	<10 ms	<10 ms	<10 ms
5	<10 ms	<10 ms	<10 ms	36 ms	40 ms	210 ms
7	<10 ms	<10 ms	<10 ms	49 ms	50 ms	>3000 ms
10	<10 ms	<10 ms	23 ms	78 ms	84 ms	>3000 ms
100	<10 ms	<10 ms	284 ms	>3000 ms	>3000 ms	>3000 ms
1000	<10 ms	54 ms	>3000 ms	>3000 ms	>3000 ms	>3000 ms
10000	<10 ms	>3000 ms	>3000 ms	>3000 ms	>3000 ms	>3000 ms

## Some numbers!

Check the code used to find these numbers on GitLab!

### DIFFERENCES IN RUN TIME

Different code snippets, all executed 1000 times.

Input size	constant	linear	quadratic	cubic	exponential	factor
1	<10 ms	<10 ms	<10 ms	<10 ms	<10 ms	<10 ms
2	<10 ms	<10 ms	<10 ms	<10 ms	<10 ms	<10 ms
5	<10 ms	<10 ms	<10 ms	36 ms	40 ms	210 ms
7	<10 ms	<10 ms	<10 ms	49 ms	50 ms	>3000 ms
10	<10 ms	<10 ms	23 ms	78 ms	84 ms	>3000 ms
100	<10 ms	<10 ms	284 ms	>3000 ms	>3000 ms	>3000 ms
1000	<10 ms	54 ms	>3000 ms	>3000 ms	>3000 ms	>3000 ms
10000	<10 ms	>3000 ms	>3000 ms	>3000 ms	>3000 ms	>3000 ms

## Some numbers!

Check the code used to find these numbers on GitLab!

### DIFFERENCES IN RUN TIME

Different code snippets, all executed 1000 times.

Input size	constant	linear	quadratic	cubic	exponential	factor
1	<10 ms	<10 ms	<10 ms	<10 ms	<10 ms	<10 ms
2	<10 ms	<10 ms	<10 ms	<10 ms	<10 ms	<10 ms
5	<10 ms	<10 ms	<10 ms	36 ms	40 ms	210 ms
7	<10 ms	<10 ms	<10 ms	49 ms	50 ms	>3000 ms
10	<10 ms	<10 ms	23 ms	78 ms	84 ms	>3000 ms
100	<10 ms	<10 ms	284 ms	>3000 ms	>3000 ms	>3000 ms
1000	<10 ms	54 ms	>3000 ms	>3000 ms	>3000 ms	>3000 ms
10000	<10 ms	>3000 ms	>3000 ms	>3000 ms	>3000 ms	>3000 ms

## Some numbers!

Check the code used to find these numbers on GitLab!

### DIFFERENCES IN RUN TIME

Different code snippets, all executed 1000 times.

Input size	constant	linear	quadratic	cubic	exponential	factor
1	<10 ms	<10 ms	<10 ms	<10 ms	<10 ms	<10 ms
2	<10 ms	<10 ms	<10 ms	<10 ms	<10 ms	<10 ms
5	<10 ms	<10 ms	<10 ms	36 ms	40 ms	210 ms
7	<10 ms	<10 ms	<10 ms	49 ms	50 ms	>3000 ms
10	<10 ms	<10 ms	23 ms	78 ms	84 ms	>3000 ms
100	<10 ms	<10 ms	284 ms	>3000 ms	>3000 ms	>3000 ms
1000	<10 ms	54 ms	>3000 ms	>3000 ms	>3000 ms	>3000 ms
10000	<10 ms	>3000 ms	>3000 ms	>3000 ms	>3000 ms	>3000 ms

## Some numbers!

Check the code used to find these numbers on GitLab!

### DIFFERENCES IN RUN TIME

Different code snippets, all executed 1000 times.

Input size	constant	linear	quadratic	cubic	exponential	factor
1	<10 ms	<10 ms	<10 ms	<10 ms	<10 ms	<10 ms
2	<10 ms	<10 ms	<10 ms	<10 ms	<10 ms	<10 ms
5	<10 ms	<10 ms	<10 ms	36 ms	40 ms	210 ms
7	<10 ms	<10 ms	<10 ms	49 ms	50 ms	>3000 ms
10	<10 ms	<10 ms	23 ms	78 ms	84 ms	>3000 ms
100	<10 ms	<10 ms	284 ms	>3000 ms	>3000 ms	>3000 ms
1000	<10 ms	54 ms	>3000 ms	>3000 ms	>3000 ms	>3000 ms
10000	<10 ms	>3000 ms	>3000 ms	>3000 ms	>3000 ms	>3000 ms

# Some numbers!

Check the code used to find these numbers on GitLab!

## DIFFERENCES IN RUN TIME

Different code snippets, all executed 1000 times.

Input size	constant	linear	quadratic	cubic	exponential	factor
1	<10 ms	<10 ms	<10 ms	<10 ms	<10 ms	<10 ms
2	<10 ms	<10 ms	<10 ms	<10 ms	<10 ms	<10 ms
5	<10 ms	<10 ms	<10 ms	36 ms	40 ms	210 ms
7	<10 ms	<10 ms	<10 ms	49 ms	50 ms	>3000 ms
10	<10 ms	<10 ms	23 ms	78 ms	84 ms	>3000 ms
100	<10 ms	<10 ms	284 ms	>3000 ms	>3000 ms	>3000 ms
1000	<10 ms	54 ms	>3000 ms	>3000 ms	>3000 ms	>3000 ms
10000	<10 ms	>3000 ms	>3000 ms	>3000 ms	>3000 ms	>3000 ms

## Some numbers!

Check the code used to find these numbers on GitLab!

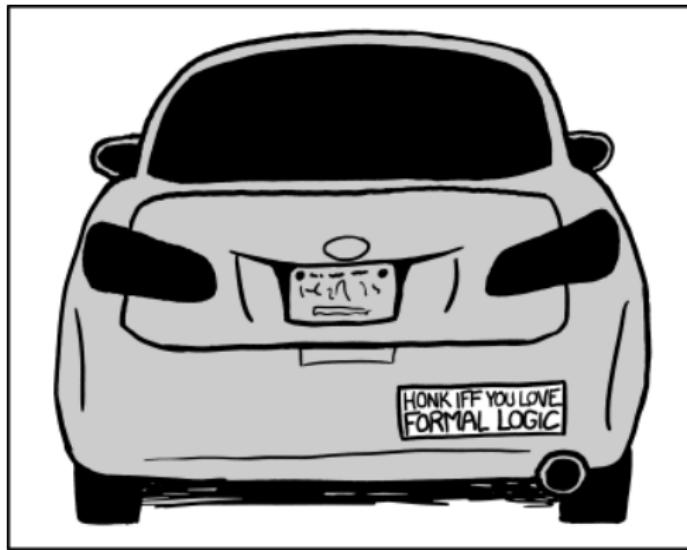
### DIFFERENCES IN RUN TIME

Different code snippets, all executed 1000 times.

Input size	constant	linear	quadratic	cubic	exponential	factor
1	<10 ms	<10 ms	<10 ms	<10 ms	<10 ms	<10 ms
2	<10 ms	<10 ms	<10 ms	<10 ms	<10 ms	<10 ms
5	<10 ms	<10 ms	<10 ms	36 ms	40 ms	210 ms
7	<10 ms	<10 ms	<10 ms	49 ms	50 ms	>3000 ms
10	<10 ms	<10 ms	23 ms	78 ms	84 ms	>3000 ms
100	<10 ms	<10 ms	284 ms	>3000 ms	>3000 ms	>3000 ms
1000	<10 ms	54 ms	>3000 ms	>3000 ms	>3000 ms	>3000 ms
10000	<10 ms	>3000 ms	>3000 ms	>3000 ms	>3000 ms	>3000 ms

## Classifying these run times

XKCD: Formal Logic <https://xkcd.com/1033>



## Big-Oh notation

- ▶ We care about what we call: “asymptotic run time complexity”.
- ▶ We denote this using big-Oh, e.g.  $f(n) = 3n + 2$  is  $O(n)$ .

A FRIEND OF MINE TOLD ME...

Have you seen this notation before?

SOMETHING SOMETHING, MATHEMATICS?

- ▶ You (may?) have used big-Oh to a certain point before. E.g. as  $n$  approaches 5.
- ▶ In computer science we only think about when  $n$  approaches  $\infty$ .

## Big-Oh notation

- ▶ We care about what we call: “asymptotic run time complexity”.
- ▶ We denote this using big-Oh, e.g.  $f(n) = 3n + 2$  is  $O(n)$ .

A FRIEND OF MINE TOLD ME...

Have you seen this notation before?

SOMETHING SOMETHING, MATHEMATICS?

- ▶ You (may?) have used big-Oh to a certain point before. E.g. as  $n$  approaches 5.
- ▶ In computer science we only think about when  $n$  approaches  $\infty$ .

## Big-Oh notation

- ▶ We care about what we call: “asymptotic run time complexity”.
- ▶ We denote this using big-Oh, e.g.  $f(n) = 3n + 2$  is  $O(n)$ .

A FRIEND OF MINE TOLD ME...

Have you seen this notation before?

SOMETHING SOMETHING, MATHEMATICS?

- ▶ You (may?) have used big-Oh to a certain point before. E.g. as  $n$  approaches 5.
- ▶ In computer science we only think about when  $n$  approaches  $\infty$ .

## Big-Oh notation

- ▶ We care about what we call: “asymptotic run time complexity”.
- ▶ We denote this using big-Oh, e.g.  $f(n) = 3n + 2$  is  $O(n)$ .

A FRIEND OF MINE TOLD ME...

Have you seen this notation before?

SOMETHING SOMETHING, MATHEMATICS?

- ▶ You (may?) have used big-Oh to a certain point before. E.g. as  $n$  approaches 5.
- ▶ In computer science we only think about when  $n$  approaches  $\infty$ .

## Formally

### DEFINITION (BIG-OH)

A function  $f(n)$  is  $O(g(n))$  iff there is a positive real constant  $c$  and a positive integer  $n_0$  such that for all  $n \geq n_0$  it holds that  $f(n) \leq cg(n)$ . In other words:  
 $\exists c \in \mathbb{R}, \exists n_0 \in \mathbb{N} (c > 0 \wedge n_0 \geq 1 \wedge (\forall n \in \mathbb{N} (n \geq n_0 \rightarrow f(n) \leq cg(n))))$ .

So...

Which of the following is/are true?

- A.  $n^2$  is  $O(n^3)$
- B.  $8n^2$  is  $O(n^2)$
- C.  $16n^2 + 5n + 2$  is  $O(n^2)$
- D.  $16n^2 + 5n \log n$  is  $O(n^2)$
- E.  $16n^2 \log n$  is  $O(n^2)$

Formally

### DEFINITION (BIG-OH)

A function  $f(n)$  is  $O(g(n))$  iff there is a positive real constant  $c$  and a positive integer  $n_0$  such that for all  $n \geq n_0$  it holds that  $f(n) \leq cg(n)$ . In other words:  
 $\exists c \in \mathbb{R}, \exists n_0 \in \mathbb{N} (c > 0 \wedge n_0 \geq 1 \wedge (\forall n \in \mathbb{N} (n \geq n_0 \rightarrow f(n) \leq cg(n))))$ .

So...

Which of the following is/are true?

- A.  $n^2$  is  $O(n^3)$
- B.  $8n^2$  is  $O(n^2)$
- C.  $16n^2 + 5n + 2$  is  $O(n^2)$
- D.  $16n^2 + 5n \log n$  is  $O(n^2)$
- E.  $16n^2 \log n$  is  $O(n^2)$

Let's prove that shall we?

Well, I say "we"...

Prove that  $f(n) = 16n^2 + 5n + 2$  is  $O(n^2)$ .

PROOF.

To prove:  $\exists c > 0, n_0 \geq 1$  such that  $\forall n \geq n_0 16n^2 + 5n + 2 \leq cn^2$ .

Take  $n_0 = 1$ , now for all  $n \in \mathbb{N}$  with  $n \geq n_0$ :

$$\begin{aligned} 16n^2 + 5n + 2 &\leq 16n^2 + 5n^2 + 2n^2 \\ &= 23n^2 \end{aligned}$$

So take  $c = 23$ .



Let's prove that shall we?

Well, I say "we"...

Prove that  $f(n) = 16n^2 + 5n + 2$  is  $O(n^2)$ .

PROOF.

To prove:  $\exists c > 0, n_0 \geq 1$  such that  $\forall n \geq n_0$   $16n^2 + 5n + 2 \leq cn^2$ .

Take  $n_0 = 1$ , now for all  $n \in \mathbb{N}$  with  $n \geq n_0$ :

$$\begin{aligned} 16n^2 + 5n + 2 &\leq 16n^2 + 5n^2 + 2n^2 \\ &= 23n^2 \end{aligned}$$

So take  $c = 23$ .



Let's prove that shall we?

Well, I say "we"...

Prove that  $f(n) = 16n^2 + 5n + 2$  is  $O(n^2)$ .

PROOF.

To prove:  $\exists c > 0, n_0 \geq 1$  such that  $\forall n \geq n_0$   $16n^2 + 5n + 2 \leq cn^2$ .

Take  $n_0 = 1$ , now for all  $n \in \mathbb{N}$  with  $n \geq n_0$ :

$$\begin{aligned} 16n^2 + 5n + 2 &\leq 16n^2 + 5n^2 + 2n^2 \\ &= 23n^2 \end{aligned}$$

So take  $c = 23$ .



# Polynomial run time

## DEFINITION (POLYNOMIAL RUN TIME)

A function has a polynomial run time  $T(n)$  if  $T(n)$  is  $O(n^c)$  for some constant  $c$ .

## THIS COURSE

Most of the algorithms treated in this course have a polynomial run time.

## REMEMBER THIS

We will revisit the notion of polynomial run times in the very last lecture, where we study some problems that we believe to have no polynomial time solution!

# Polynomial run time

## DEFINITION (POLYNOMIAL RUN TIME)

A function has a polynomial run time  $T(n)$  if  $T(n)$  is  $O(n^c)$  for some constant  $c$ .

## THIS COURSE

Most of the algorithms treated in this course have a polynomial run time.

## REMEMBER THIS

We will revisit the notion of polynomial run times in the very last lecture, where we study some problems that we believe to have no polynomial time solution!

# Polynomial run time

## DEFINITION (POLYNOMIAL RUN TIME)

A function has a polynomial run time  $T(n)$  if  $T(n)$  is  $O(n^c)$  for some constant  $c$ .

## THIS COURSE

Most of the algorithms treated in this course have a polynomial run time.

## REMEMBER THIS

We will revisit the notion of polynomial run times in the very last lecture, where we study some problems that we believe to have no polynomial time solution!

# Some numbers! Revisited

Check the code used to find these numbers on GitLab!

## DIFFERENCES IN RUN TIME

We can now formalise our previous table a little bit:

Input size	$O(1)$	$O(n)$	$O(n^2)$	$O(n^3)$	$O(2^n)$	$O(n!)$
1	<10 ms	<10 ms	<10 ms	<10 ms	<10 ms	<10 ms
2	<10 ms	<10 ms	<10 ms	<10 ms	<10 ms	<10 ms
5	<10 ms	<10 ms	<10 ms	36 ms	40 ms	210 ms
7	<10 ms	<10 ms	<10 ms	49 ms	50 ms	>3000 ms
10	<10 ms	<10 ms	23 ms	78 ms	84 ms	>3000 ms
100	<10 ms	<10 ms	284 ms	>3000 ms	>3000 ms	>3000 ms
1000	<10 ms	54 ms	>3000 ms	>3000 ms	>3000 ms	>3000 ms
10000	<10 ms	>3000 ms				

Next lecture we will finish formalising this table.

# Some numbers! Revisited

Check the code used to find these numbers on GitLab!

## DIFFERENCES IN RUN TIME

We can now formalise our previous table a little bit:

Input size	$O(1)$	$O(n)$	$O(n^2)$	$O(n^3)$	$O(2^n)$	$O(n!)$
1	<10 ms	<10 ms	<10 ms	<10 ms	<10 ms	<10 ms
2	<10 ms	<10 ms	<10 ms	<10 ms	<10 ms	<10 ms
5	<10 ms	<10 ms	<10 ms	36 ms	40 ms	210 ms
7	<10 ms	<10 ms	<10 ms	49 ms	50 ms	>3000 ms
10	<10 ms	<10 ms	23 ms	78 ms	84 ms	>3000 ms
100	<10 ms	<10 ms	284 ms	>3000 ms	>3000 ms	>3000 ms
1000	<10 ms	54 ms	>3000 ms	>3000 ms	>3000 ms	>3000 ms
10000	<10 ms	>3000 ms				

Next lecture we will finish formalising this table.

# Some numbers! Revisited

Check the code used to find these numbers on GitLab!

## DIFFERENCES IN RUN TIME

We can now formalise our previous table a little bit:

Input size	$O(1)$	$O(n)$	$O(n^2)$	$O(n^3)$	$O(2^n)$	$O(n!)$
1	<10 ms	<10 ms	<10 ms	<10 ms	<10 ms	<10 ms
2	<10 ms	<10 ms	<10 ms	<10 ms	<10 ms	<10 ms
5	<10 ms	<10 ms	<10 ms	36 ms	40 ms	210 ms
7	<10 ms	<10 ms	<10 ms	49 ms	50 ms	>3000 ms
10	<10 ms	<10 ms	23 ms	78 ms	84 ms	>3000 ms
100	<10 ms	<10 ms	284 ms	>3000 ms	>3000 ms	>3000 ms
1000	<10 ms	54 ms	>3000 ms	>3000 ms	>3000 ms	>3000 ms
10000	<10 ms	>3000 ms				

Next lecture we will finish formalising this table.

## Revisiting our code snippets

```
def foo(n: int) -> int:  
    2      s = 0  
        for i in range(n):  
            4          s += i * i  
        return s
```

src/for-loop.py

### SO WHICH IS IT?

Which of these describes the run time  $T(n)$ ?

- A.  $T(n)$  is  $O(1)$ .
- B.  $T(n)$  is  $O(n)$ .
- C.  $T(n)$  is  $O(n^2)$ .
- D.  $T(n)$  is  $O(n^3)$ .
- E. I don't know.

### MULTIPLE CORRECT ANSWERS

B through D are correct.

### TIGHTEST BOUND

We often request the tightest bound. Which in this case is  $O(n)$ .

## Revisiting our code snippets

```
1 def foo(n: int) -> int:  
2     s = 0  
3     for i in range(n):  
4         s += i * i  
5     return s
```

src/for-loop.py

### SO WHICH IS IT?

Which of these describes the run time  $T(n)$ ?

- A.  $T(n)$  is  $O(1)$ .
- B.  $T(n)$  is  $O(n)$ .
- C.  $T(n)$  is  $O(n^2)$ .
- D.  $T(n)$  is  $O(n^3)$ .
- E. I don't know.

### MULTIPLE CORRECT ANSWERS

B through D are correct.

### TIGHTEST BOUND

We often request the tightest bound. Which in this case is  $O(n)$ .

## Revisiting our code snippets

```
1 def foo(n: int) -> int:  
2     s = 0  
3     for i in range(n):  
4         s += i * i  
5     return s
```

src/for-loop.py

### SO WHICH IS IT?

Which of these describes the run time  $T(n)$ ?

- A.  $T(n)$  is  $O(1)$ .
- B.  $T(n)$  is  $O(n)$ .
- C.  $T(n)$  is  $O(n^2)$ .
- D.  $T(n)$  is  $O(n^3)$ .
- E. I don't know.

### MULTIPLE CORRECT ANSWERS

B through D are correct.

### TIGHTEST BOUND

We often request the tightest bound. Which in this case is  $O(n)$ .

## Which case?

```
1 from typing import List  
  
3  
4 def foo(x: List[int]) -> int:  
5     s = 0  
6     for i in x:  
7         if i > 5:  
8             s += i * i  
9     return s
```

src/for-loop-wc.py

So WHICH IS IT?

Which of these forms a tight bound on the run time  $T(n)$ ?

- A.  $O(1)$ .
- B.  $O(n)$ .
- C.  $O(n^2)$ .
- D. I don't know.

Only B

WORST KAAS

We talk about the  
*worst-case*.

## Which case?

```
1 from typing import List  
  
3  
4 def foo(x: List[int]) -> int:  
5     s = 0  
6     for i in x:  
7         if i > 5:  
8             s += i * i  
9     return s
```

src/for-loop-wc.py

So WHICH IS IT?

Which of these forms a tight bound on the run time  $T(n)$ ?

- A.  $O(1)$ .
- B.  $O(n)$ .
- C.  $O(n^2)$ .
- D. I don't know.

Only B

WORST KAAS

We talk about the  
*worst-case.*

## Which case?

```
1 from typing import List  
  
3  
4 def foo(x: List[int]) -> int:  
5     s = 0  
6     for i in x:  
7         if i > 5:  
8             s += i * i  
9     return s
```

src/for-loop-wc.py

So WHICH IS IT?

Which of these forms a tight bound on the run time  $T(n)$ ?

- A.  $O(1)$ .
- B.  $O(n)$ .
- C.  $O(n^2)$ .
- D. I don't know.

Only B

WORST KAAS

We talk about the  
*worst-case*.

## Space complexity

Anywhere in space and time



Image By: *anna\_thetical*

## Space complexity

- ▶ So far we have only looked at the required **time** of a function.
- ▶ What about the required **space**?
- ▶ Just like time, space is a finite resource.
- ▶ So it is important to be able to set bounds on the usage.

### DIFFERENCE?

Is there any difference in how time and space are used by functions?

### RECYCLING

Yes! Space can be reused, whereas time cannot.

## Space complexity

- ▶ So far we have only looked at the required **time** of a function.
- ▶ What about the required **space**?
- ▶ Just like time, space is a **finite** resource.
- ▶ So it is important to be able to set bounds on the usage.

### DIFFERENCE?

Is there any difference in how time and space are used by functions?

### RECYCLING

Yes! Space can be reused, whereas time cannot.

## Space complexity

- ▶ So far we have only looked at the required **time** of a function.
- ▶ What about the required **space**?
- ▶ Just like time, space is a **finite** resource.
- ▶ So it is important to be able to set bounds on the usage.

### DIFFERENCE?

Is there any difference in how time and space are used by functions?

### RECYCLING

Yes! Space can be reused, whereas time cannot.

## Space complexity

- ▶ So far we have only looked at the required **time** of a function.
- ▶ What about the required **space**?
- ▶ Just like time, space is a **finite** resource.
- ▶ So it is important to be able to set bounds on the usage.

### DIFFERENCE?

Is there any difference in how time and space are used by functions?

### RECYCLING

Yes! Space can be reused, whereas time cannot.

# A first example

## Creating a list

```
1 a = [x for x in range(n)]
```

src/comprehension-complexity.py

### SPACE

What is the space complexity of this list comprehension?

- A.  $\Theta(1)$
- B.  $\Theta(n)$
- C.  $\Theta(n^2)$
- D. I don't know.

### LINEAR TIME

We have  $n$  integers, each requiring some constant amount of space  $c$ . Thus  $S(n) = cn$ , so  $\Theta(n)$  space is required.

## A first example

Creating a list

```
1 a = [x for x in range(n)]
```

src/comprehension-complexity.py

### SPACE

What is the space complexity of this list comprehension?

- A.  $\Theta(1)$
- B.  $\Theta(n)$
- C.  $\Theta(n^2)$
- D. I don't know.

### LINEAR TIME

We have  $n$  integers, each requiring some constant amount of space  $c$ . Thus  $S(n) = cn$ , so  $\Theta(n)$  space is required.

## A first example

Creating a list

```
1 a = [x for x in range(n)]
```

src/comprehension-complexity.py

### SPACE

What is the **space** complexity of this list comprehension?

- A.  $\Theta(1)$
- B.  $\Theta(n)$
- C.  $\Theta(n^2)$
- D. I don't know.

### LINEAR TIME

We have  $n$  integers, each requiring some constant amount of space  $c$ . Thus  $S(n) = cn$ , so  $\Theta(n)$  space is required.

## Memory in a computer

The following slides are based on slides originally created by Robbert Krebbers



Image By: Petr Kratochvil

# The call stack in action

Based on a slide by Robbert Krebbers

**Let us call `foo(3)`:**

```
% [linebackgroundcolor=t%
% \btLstHL<1>{}%
% \btLstHL<2>{5-8}%
% \btLstHL<3>{6-8}%
% \btLstHL<5>{7-8}%
% \btLstHL<7>{8-8}%
% \btLstHL<4,6>{2}%
%
% }]
def bar(n: int) -> int:
    return n;

def foo(n: int) -> int:
    res = 8
    res += bar(n-1)
    res += bar(n-2)
    return res
```

**Stack:**

**When a function returns:**

- ▶ The *stack frame* is *removed*
- ▶ Control returns to the *return address*

**When a function is called:**

- ▶ A *stack frame* is *added* to the stack

# The call stack in action

Based on a slide by Robbert Krebbers

**Let us call `foo(3)`:**

```
% [linebackgroundcolor=t%
% \btLstHL<1>{}%
% \btLstHL<2>{5-8}%
% \btLstHL<3>{6-8}%
% \btLstHL<5>{7-8}%
% \btLstHL<7>{8-8}%
% \btLstHL<4,6>{2}%
% }]
def bar(n: int) -> int:
    return n;

def foo(n: int) -> int:
    res = 8
    res += bar(n-1)
    res += bar(n-2)
    return res
```

**Stack:**

**When a function returns:**

- ▶ The *stack frame* is *removed*
- ▶ Control returns to the *return address*

**When a function is called:**

- ▶ A *stack frame* is *added* to the stack

# The call stack in action

Based on a slide by Robbert Krebbers

**Let us call `foo(3)`:**

```
% [linebackgroundcolor=t%
% \btLstHL<1>{}%
% \btLstHL<2>{5-8}%
% \btLstHL<3>{6-8}%
% \btLstHL<5>{7-8}%
% \btLstHL<7>{8-8}%
% \btLstHL<4,6>{2}%
% }]
def bar(n: int) -> int:
    return n;

def foo(n: int) -> int:
    res = 8
    res += bar(n-1)
    res += bar(n-2)
    return res
```

**Stack:**

**When a function returns:**

- ▶ The *stack frame* is *removed*
- ▶ Control returns to the *return address*

**When a function is called:**

- ▶ A *stack frame* is *added* to the stack

# The call stack in action

Based on a slide by Robbert Krebbers

Let us call `foo(3)`:

```
% [linebackgroundcolor=%
% \btLstHL<1>{}%
% \btLstHL<2>{5-8}%
% \btLstHL<3>{6-8}%
% \btLstHL<5>{7-8}%
% \btLstHL<7>{8-8}%
% \btLstHL<4,6>{2}%
% }]
def bar(n: int) -> int:
    return n;

def foo(n: int) -> int:
    res = 8
    res += bar(n-1)
    res += bar(n-2)
    return res
```

Stack:

When a function returns:

- ▶ The *stack frame* is removed
- ▶ Control returns to the *return address*

When a function is called:

- ▶ A *stack frame* is added to the stack

# A long story short

Observations:

- ▶ Calling a function takes space!
- ▶ This is important when dealing with recursive functions (which we will discuss after the break).
- ▶ All of the parameters are stored in this bit of space as well.

WHAT ABOUT LISTS?

Does this mean that the “stack frame” for `baz(my_list)` requires  $O(n)$  space?

NOPE

No! We pass a *reference* instead of a copy. We tell `baz` where the list is so that it can access (or change!) it. Thus this call still requires only  $O(1)$  space.

See this excellent StackOverflow post explaining this in more detail if you are interested:

<https://stackoverflow.com/a/986145>.

## A long story short

Observations:

- ▶ Calling a function takes space!
- ▶ This is important when dealing with recursive functions (which we will discuss after the break).
- ▶ All of the parameters are stored in this bit of space as well.

WHAT ABOUT LISTS?

Does this mean that the “stack frame” for `baz(my_list)` requires  $O(n)$  space?

NOPE

No! We pass a *reference* instead of a copy. We tell `baz` where the list is so that it can access (or change!) it. Thus this call still requires only  $O(1)$  space.

See this excellent StackOverflow post explaining this in more detail if you are interested:

<https://stackoverflow.com/a/986145>.

## A long story short

Observations:

- ▶ Calling a function takes space!
- ▶ This is important when dealing with recursive functions (which we will discuss after the break).
- ▶ All of the parameters are stored in this bit of space as well.

WHAT ABOUT LISTS?

Does this mean that the “stack frame” for `baz(my_list)` requires  $O(n)$  space?

NOPE

No! We pass a *reference* instead of a copy. We tell `baz` where the list is so that it can access (or change!) it. Thus this call still requires only  $O(1)$  space.

See this excellent StackOverflow post explaining this in more detail if you are interested:

<https://stackoverflow.com/a/986145>.

## A second example

Using a list

```
def maya(n: int) -> int:  
    2     x = list()  
    4     for i in range(n):  
        5         for j in range(n):  
            6             x += [i * j]  
    8     s = 0  
     for i in x:  
        9         s += i ** 1.5  
    10    return s
```

src/big-oh-example.py

### SPACE

What is the space complexity of the function `maya`?

- A.  $\Theta(1)$
- B.  $\Theta(n)$
- C.  $\Theta(n^2)$
- D. I don't know.

### QUADRATIC SPACE

We create a list of  $n^2$  items, so we need  $\Theta(n^2)$  space. We could say  $S(n) = c_0 + c_1 n^2$ , where  $c_0$  is for the stack frame,  $s$ ,  $i$  and  $j$ .  $c_1$  is for  $x$ .

## A second example

Using a list

```
1 def maya(n: int) -> int:
2     x = list()
3     for i in range(n):
4         for j in range(n):
5             x += [i * j]
6     s = 0
7     for i in x:
8         s += i ** 1.5
9     return s
```

src/big-oh-example.py

### SPACE

What is the space complexity of the function `maya`?

- A.  $\Theta(1)$
- B.  $\Theta(n)$
- C.  $\Theta(n^2)$
- D. I don't know.

### QUADRATIC SPACE

We create a list of  $n^2$  items, so we need  $\Theta(n^2)$  space. We could say  $S(n) = c_0 + c_1 n^2$ , where  $c_0$  is for the stack frame,  $s$ ,  $i$  and  $j$ .  $c_1$  is for  $x$ .

## A second example

Using a list

```
1 def maya(n: int) -> int:
2     x = list()
3     for i in range(n):
4         for j in range(n):
5             x += [i * j]
6     s = 0
7     for i in x:
8         s += i ** 1.5
9     return s
```

src/big-oh-example.py

### SPACE

What is the space complexity of the function `maya`?

- A.  $\Theta(1)$
- B.  $\Theta(n)$
- C.  $\Theta(n^2)$
- D. I don't know.

### QUADRATIC SPACE

We create a list of  $n^2$  items, so we need  $\Theta(n^2)$  space. We could say  $S(n) = c_0 + c_1 n^2$ , where  $c_0$  is for the stack frame,  $s$ ,  $i$  and  $j$ .  $c_1$  is for  $x$ .

## A second example - modified

Doing without the list

```
1 def mia(n: int) -> int:
2     s = 0
3     for i in range(n):
4         for j in range(n):
5             s += (i * j) ** 1.5
6
7 return s
```

src/big-oh-example-v2.py

### SPACE

What is the space complexity of the function mia?

- A.  $\Theta(1)$
- B.  $\Theta(n)$
- C.  $\Theta(n^2)$
- D. I don't know.

### CONSTANT SPACE

We now only require to store the variable  $s$  and call the function `range`. Both of these require constant space, so  $S(n) = c_0$  is  $\Theta(1)$ .

## A second example - modified

Doing without the list

```
def mia(n: int) -> int:  
    2     s = 0  
    4     for i in range(n):  
        4         for j in range(n):  
            6             s += (i * j) ** 1.5  
    6     return s
```

src/big-oh-example-v2.py

### SPACE

What is the space complexity of the function mia?

- A.  $\Theta(1)$
- B.  $\Theta(n)$
- C.  $\Theta(n^2)$
- D. I don't know.

### CONSTANT SPACE

We now only require to store the variable s and call the function range. Both of these require constant space, so  $S(n) = c_0$  is  $\Theta(1)$ .

## A second example - modified

Doing without the list

```
def mia(n: int) -> int:  
    2      s = 0  
    3      for i in range(n):  
    4          for j in range(n):  
    5              s += (i * j) ** 1.5  
    6      return s
```

src/big-oh-example-v2.py

### SPACE

What is the space complexity of the function mia?

- A.  $\Theta(1)$
- B.  $\Theta(n)$
- C.  $\Theta(n^2)$
- D. I don't know.

### CONSTANT SPACE

We now only require to store the variable `s` and call the function `range`. Both of these require constant space, so  $S(n) = c_0$  is  $\Theta(1)$ .

## A final example - modified

Using a list from a parameter

```
from typing import List  
2  
4 def sum(a: List[int]) -> int:  
    s = 0  
    6    for i in a:  
        s += i  
    8    return s
```

src/big-oh-example-v3.py

### SPACE

What is the space complexity of the function `sum`?

- A.  $\Theta(1)$
- B.  $\Theta(n)$
- C.  $\Theta(n^2)$
- D. I don't know.

### CONSTANT SPACE

Remember that a list that is passed as input, is a *reference* and does not take space!

### OBSERVATION

Had input contributed to the space complexity, there would be no sub-linear space complexities!

## A final example - modified

Using a list from a parameter

```
from typing import List  
2  
4 def sum(a: List[int]) -> int:  
    s = 0  
    6    for i in a:  
        s += i  
    8    return s
```

src/big-oh-example-v3.py

### SPACE

What is the space complexity of the function `sum`?

- A.  $\Theta(1)$
- B.  $\Theta(n)$
- C.  $\Theta(n^2)$
- D. I don't know.

### CONSTANT SPACE

Remember that a list that is passed as input, is a *reference* and does not take space!

### OBSERVATION

Had input contributed to the space complexity, there would be no sub-linear space complexities!

## A final example - modified

Using a list from a parameter

```
from typing import List  
2  
4 def sum(a: List[int]) -> int:  
    s = 0  
    6    for i in a:  
        s += i  
    8    return s
```

src/big-oh-example-v3.py

### SPACE

What is the space complexity of the function `sum`?

- A.  $\Theta(1)$
- B.  $\Theta(n)$
- C.  $\Theta(n^2)$
- D. I don't know.

### CONSTANT SPACE

Remember that a list that is passed as input, is a *reference* and does not take space!

### OBSERVATION

Had input contributed to the space complexity, there would be no sub-linear space complexities!

## A final example - modified

Using a list from a parameter

```
from typing import List  
2  
4 def sum(a: List[int]) -> int:  
    s = 0  
    6    for i in a:  
        s += i  
8    return s
```

src/big-oh-example-v3.py

### SPACE

What is the space complexity of the function `sum`?

- A.  $\Theta(1)$
- B.  $\Theta(n)$
- C.  $\Theta(n^2)$
- D. I don't know.

### CONSTANT SPACE

Remember that a list that is passed as input, is a *reference* and does not take space!

### OBSERVATION

Had input contributed to the space complexity, there would be no sub-linear space complexities!

## Relations between time and space?

### IT'S ALL (A) RELATIVE (DIMENSION)

Given that a function `foo` uses  $\Theta(n)$  space, what, if anything, can we conclude about the amount of time  $T(n)$  `foo` requires?

- A.  $T(n)$  is  $\Omega(n)$
- B.  $T(n)$  is  $\Theta(n)$
- C.  $T(n)$  is  $O(n)$
- D. We cannot conclude anything.
- E. I don't know

### A NICE LOWER BOUND

Claiming all of this memory (space) requires time! So we need  $\Omega(n)$  time to execute `foo`!

## Relations between time and space?

### IT'S ALL (A) RELATIVE (DIMENSION)

Given that a function `foo` uses  $\Theta(n)$  space, what, if anything, can we conclude about the amount of time  $T(n)$  `foo` requires?

- A.  $T(n)$  is  $\Omega(n)$
- B.  $T(n)$  is  $\Theta(n)$
- C.  $T(n)$  is  $O(n)$
- D. We cannot conclude anything.
- E. I don't know

### A NICE LOWER BOUND

Claiming all of this memory (space) requires time! So we need  $\Omega(n)$  time to execute `foo`!

## Do you remember?

### DEFINITION (BIG-OH)

A function  $f(n)$  is  $O(g(n))$  iff there is a positive real constant  $c$  and a positive integer  $n_0$  such that for all  $n \geq n_0$  it holds that  $f(n) \leq cg(n)$ . In other words:  
 $\exists c \in \mathbb{R}, \exists n_0 \in \mathbb{N} (c > 0 \wedge n_0 \geq 1 \wedge (\forall n \in \mathbb{N} (n \geq n_0 \rightarrow f(n) \leq cg(n))))$ .

```
def maya(n: int) -> int:
    2     x = list()
    3     for i in range(n):
    4         for j in range(n):
    5             x += [i * j]
    6     s = 0
    7     for i in x:
    8         s += i ** 1.5
    9     return s
```

src/big-oh-example.py

So...

What is a tight upper bound on the run time of `maya`?

- A.  $O(1)$
- B.  $O(n)$
- C.  $O(n^2)$
- D.  $O(n^3)$
- E. I don't know.

## Do you remember?

### DEFINITION (BIG-OH)

A function  $f(n)$  is  $O(g(n))$  iff there is a positive real constant  $c$  and a positive integer  $n_0$  such that for all  $n \geq n_0$  it holds that  $f(n) \leq cg(n)$ . In other words:  
 $\exists c \in \mathbb{R}, \exists n_0 \in \mathbb{N} (c > 0 \wedge n_0 \geq 1 \wedge (\forall n \in \mathbb{N} (n \geq n_0 \rightarrow f(n) \leq cg(n))))$ .

```
1 def maya(n: int) -> int:
2     x = list()
3     for i in range(n):
4         for j in range(n):
5             x += [i * j]
6     s = 0
7     for i in x:
8         s += i ** 1.5
9     return s
```

src/big-oh-example.py

So...

What is a tight upper bound on the run time of `maya`?

- A.  $O(1)$
- B.  $O(n)$
- C.  $O(n^2)$
- D.  $O(n^3)$
- E. I don't know.

## Do you remember?

### DEFINITION (BIG-OH)

A function  $f(n)$  is  $O(g(n))$  iff there is a positive real constant  $c$  and a positive integer  $n_0$  such that for all  $n \geq n_0$  it holds that  $f(n) \leq cg(n)$ . In other words:  
 $\exists c \in \mathbb{R}, \exists n_0 \in \mathbb{N} (c > 0 \wedge n_0 \geq 1 \wedge (\forall n \in \mathbb{N} (n \geq n_0 \rightarrow f(n) \leq cg(n))))$ .

```
1 def maya(n: int) -> int:
2     x = list()
3     for i in range(n):
4         for j in range(n):
5             x += [i * j]
6     s = 0
7     for i in x:
8         s += i ** 1.5
9     return s
```

src/big-oh-example.py

So...

What is a tight upper bound on the run time of `maya`?

- A.  $O(1)$
- B.  $O(n)$
- C.  $O(n^2)$
- D.  $O(n^3)$
- E. I don't know.

# Do you remember?

## DEFINITION (BIG-OH)

A function  $f(n)$  is  $O(g(n))$  iff there is a positive real constant  $c$  and a positive integer  $n_0$  such that for all  $n \geq n_0$  it holds that  $f(n) \leq cg(n)$ . In other words:  
 $\exists c \in \mathbb{R}, \exists n_0 \in \mathbb{N} (c > 0 \wedge n_0 \geq 1 \wedge (\forall n \in \mathbb{N} (n \geq n_0 \rightarrow f(n) \leq cg(n))))$ .

```

1 def maya(n: int) -> int:
2     x = list()
3     for i in range(n):
4         for j in range(n):
5             x += [i * j]
6     s = 0
7     for i in x:
8         s += i ** 1.5
9     return s

```

src/big-oh-example.py

## RUN TIME

The run time is described as  $T(n) = c_0 + c_1n + c_2n^2$ , where

- ▶  $c_0$  is for lines 2, 5, and 9.
- ▶  $c_1n$  is for the range in line 3.
- ▶  $c_2n^2$  is for lines 4, 5, and 8.

Thus  $T(n)$  is  $O(n^2)$ .

## Which case?

```
1 from typing import List  
  
3  
4 def foo(x: List[int]) -> int:  
5     s = 0  
6     for i in x:  
7         if i > 5:  
8             s += i * i  
9     return s
```

src/for-loop-wc.py

So WHICH IS IT?

Which of these forms a tight bound on the run time  $T(n)$ ?

- A.  $O(1)$ .
- B.  $O(n)$ .
- C.  $O(n^2)$ .
- D. I don't know.

Only B

WORST KAAS

We talk about the  
*worst-case*.

## Which case?

```
1 from typing import List  
  
3  
4 def foo(x: List[int]) -> int:  
5     s = 0  
6     for i in x:  
7         if i > 5:  
8             s += i * i  
9     return s
```

src/for-loop-wc.py

So WHICH IS IT?

Which of these forms a tight bound on the run time  $T(n)$ ?

- A.  $O(1)$ .
- B.  $O(n)$ .
- C.  $O(n^2)$ .
- D. I don't know.

Only B

WORST KAAS

We talk about the  
*worst-case.*

## Which case?

```
1 from typing import List  
  
3  
4 def foo(x: List[int]) -> int:  
5     s = 0  
6     for i in x:  
7         if i > 5:  
8             s += i * i  
9     return s
```

src/for-loop-wc.py

So WHICH IS IT?

Which of these forms a tight bound on the run time  $T(n)$ ?

- A.  $O(1)$ .
- B.  $O(n)$ .
- C.  $O(n^2)$ .
- D. I don't know.

Only B

WORST KAAS

We talk about the  
*worst-case*.

## More practice?

- ▶ We will practice this more in tomorrow's tutorial!
- ▶ As well as big-Oh proofs (i.e. finding  $c$  and  $n_0$ , such that...).

## Some python built-in functions

- ▶ You already know about a number of built-in python functions.
  - ▶ `range`
  - ▶ `in` (like: `if 8 in x:`)
  - ▶ list-comprehensions
- ▶ What is their time complexity?

## Some python built-in functions

- ▶ You already know about a number of built-in python functions.
  - ▶ `range`
  - ▶ `in` (like: `if 8 in x:)`
  - ▶ `list-comprehensions`
- ▶ What is their time complexity?

## Some python built-in functions

- ▶ You already know about a number of built-in python functions.
  - ▶ `range`
  - ▶ `in` (like: `if 8 in x:)`
  - ▶ `list-comprehensions`
- ▶ What is their time complexity?

## Some python built-in functions

- ▶ You already know about a number of built-in python functions.
  - ▶ `range`
  - ▶ `in` (like: `if 8 in x:)`
  - ▶ list-comprehensions
- ▶ What is their time complexity?

# On the topic of ranges

Get your cowboy boots ready!

```
1 range(n)
  range(0, 100)
3 range(n / 2, n)
  range(n - 100, n)
```

`src/range-complexity.py`

## GROUP BY

Group the different lines by their run time complexity (are they  $O(1)$ ,  $O(n)$ ,  $O(n^2)$ , etc?)

## SO WHAT ARE THEY?

1.  $O(n)$ , we go through  $n$  items.
2.  $O(1)$ , there are a constant number of items (100).
3.  $O(n)$ , although we go through only  $n/2$  items, this still grows linearly as  $n$  grows.
4.  $O(1)$ , this is again a constant number of items (100).

# On the topic of ranges

Get your cowboy boots ready!

```
range(n)
2 range(0, 100)
range(n / 2, n)
4 range(n - 100, n)
```

src/range-complexity.py

## GROUP BY

Group the different lines by their run time complexity (are they  $O(1)$ ,  $O(n)$ ,  $O(n^2)$ , etc?)

## SO WHAT ARE THEY?

1.  $O(n)$ , we go through  $n$  items.
2.  $O(1)$ , there are a constant number of items (100).
3.  $O(n)$ , although we go through only  $n/2$  items, this still grows linearly as  $n$  grows.
4.  $O(1)$ , this is again a constant number of items (100).

# On the topic of ranges

Get your cowboy boots ready!

```
2 range(n)
  range(0, 100)
  range(n / 2, n)
4 range(n - 100, n)
```

src/range-complexity.py

## GROUP BY

Group the different lines by their run time complexity (are they  $O(1)$ ,  $O(n)$ ,  $O(n^2)$ , etc?)

## SO WHAT ARE THEY?

1.  $O(n)$ , we go through  $n$  items.
2.  $O(1)$ , there are a constant number of items (100).
3.  $O(n)$ , although we go through only  $n/2$  items, this still grows linearly as  $n$  grows.
4.  $O(1)$ , this is again a constant number of items (100).

## What about in?

Everyone get in here!

42 `in` my\_list

`src/in-complexity.py`

### SEARCHING

What is the time complexity of this operation?

- A.  $O(1)$
- B.  $O(n)$  where  $n = \text{len}(\text{my\_list})$ .
- C.  $O(n^2)$  where  $n = \text{len}(\text{my\_list})$ .
- D. I don't know.

### LINEAR TIME

Worst case we need to check every element, so  $O(n)$  time.

# What about in?

Everyone get in here!

```
1 42 in my_list
```

src/in-complexity.py

## SEARCHING

What is the time complexity of this operation?

- A.  $O(1)$
- B.  $O(n)$  where  $n = \text{len}(\text{my\_list})$ .
- C.  $O(n^2)$  where  $n = \text{len}(\text{my\_list})$ .
- D. I don't know.

## LINEAR TIME

Worst case we need to check every element, so  $O(n)$  time.

## What about in?

Everyone get in here!

```
1 42 in my_list
```

src/in-complexity.py

### SEARCHING

What is the time complexity of this operation?

- A.  $O(1)$
- B.  $O(n)$  where  $n = \text{len}(\text{my\_list})$ .
- C.  $O(n^2)$  where  $n = \text{len}(\text{my\_list})$ .
- D. I don't know.

### LINEAR TIME

Worst case we need to check every element, so  $O(n)$  time.

# List comprehensions

Comprehende?

```
1 a = [x for x in range(n)]
```

src/comprehension-complexity.py

## COMPREHENSION

What is the time complexity of this list comprehension?

- A.  $O(1)$
- B.  $O(n)$
- C.  $O(n^2)$
- D. I don't know.

## LINEAR TIME

The answer is in the for loop. This is  $O(n)$  and so the creation of the list is also  $O(n)$ .

## WHY THOUGH?

We will see why exactly when we discuss lists next week.

## List comprehensions

Comprehende?

```
1 a = [x for x in range(n)]
```

src/comprehension-complexity.py

### COMPREHENSION

What is the time complexity of this list comprehension?

- A.  $O(1)$
- B.  $O(n)$
- C.  $O(n^2)$
- D. I don't know.

### LINEAR TIME

The answer is in the for loop. This is  $O(n)$  and so the creation of the list is also  $O(n)$ .

### WHY THOUGH?

We will see why exactly when we discuss lists next week.

## List comprehensions

Comprehende?

```
1 a = [x for x in range(n)]
```

src/comprehension-complexity.py

### COMPREHENSION

What is the time complexity of this list comprehension?

- A.  $O(1)$
- B.  $O(n)$
- C.  $O(n^2)$
- D. I don't know.

### LINEAR TIME

The answer is in the for loop. This is  $O(n)$  and so the creation of the list is also  $O(n)$ .

### WHY THOUGH?

We will see why exactly when we discuss lists next week.

## List comprehensions

Comprehende?

```
1 a = [x for x in range(n)]
```

src/comprehension-complexity.py

### COMPREHENSION

What is the time complexity of this list comprehension?

- A.  $O(1)$
- B.  $O(n)$
- C.  $O(n^2)$
- D. I don't know.

### LINEAR TIME

The answer is in the for loop. This is  $O(n)$  and so the creation of the list is also  $O(n)$ .

### WHY THOUGH?

We will see why exactly when we discuss lists next week.

## More bounds



Image By: *David Mulder*

## A lower bound

Omega

### DEFINITION (BIG- $\Omega$ (OMEGA))

A function  $f(n)$  is  $\Omega(g(n))$  iff there is a positive real constant  $c$  and a positive integer  $n_0$  such that for all  $n \geq n_0$  it holds that  $f(n) \geq cg(n)$ . In other words:  
 $\exists c \in \mathbb{R}, \exists n_0 \in \mathbb{N}(c > 0 \wedge n_0 \geq 1 \wedge (\forall n \in \mathbb{N}(n \geq n_0 \rightarrow f(n) \geq cg(n))))$ .

### WHAT OF IT?

Assume that  $f(n)$  is  $O(g(n))$  what, if anything, can we now conclude?

- A.  $f(n)$  is  $\Omega(g(n))$
- B.  $g(n)$  is  $O(f(n))$
- C.  $g(n)$  is  $\Omega(f(n))$
- D. None of the above.
- E. I don't know.

## A lower bound

Omega

### DEFINITION (BIG- $\Omega$ (OMEGA))

A function  $f(n)$  is  $\Omega(g(n))$  iff there is a positive real constant  $c$  and a positive integer  $n_0$  such that for all  $n \geq n_0$  it holds that  $f(n) \geq cg(n)$ . In other words:  
 $\exists c \in \mathbb{R}, \exists n_0 \in \mathbb{N}(c > 0 \wedge n_0 \geq 1 \wedge (\forall n \in \mathbb{N}(n \geq n_0 \rightarrow f(n) \geq cg(n))))$ .

### WHAT OF IT?

Assume that  $f(n)$  is  $O(g(n))$  what, if anything, can we now conclude?

- A.  $f(n)$  is  $\Omega(g(n))$
- B.  $g(n)$  is  $O(f(n))$
- C.  $g(n)$  is  $\Omega(f(n))$
- D. None of the above.
- E. I don't know.

# So what?

WHY DO WE CARE?

What can we use big- $\Omega$  for?

NOT MUCH!

Very very little :)

Though occasionally we can prove things require e.g.  $\Omega(n)$  steps, even if we do not know how to solve it exactly.

But if something is both  $O(f(n))$  and  $\Omega(f(n))$ ...

DEFINITION (BIG- $\Theta$  (THETA))

A function  $f(n)$  is  $\Theta(g(n))$  iff there are positive real constants  $c_0, c_1$  and a positive integer  $n_0$  such that for all  $n \geq n_0$  it holds that

$c_0g(n) \leq f(n) \leq c_1g(n)$ . In other words:

$\exists c_0, c_1 \in \mathbb{R}, \exists n_0 \in \mathbb{N} (c_0 > 0 \wedge c_1 > 0 \wedge n_0 \geq 1 \wedge (\forall n \in \mathbb{N} (n \geq n_0 \rightarrow c_1g(n) \leq f(n) \leq c_2g(n))))$ .

## So what?

WHY DO WE CARE?

What can we use big- $\Omega$  for?

NOT MUCH!

Very very little :)

Though occasionally we can prove things require e.g.  $\Omega(n)$  steps, even if we do not know how to solve it exactly.

But if something is both  $O(f(n))$  and  $\Omega(f(n))$ ...

DEFINITION (BIG- $\Theta$  (THETA))

A function  $f(n)$  is  $\Theta(g(n))$  iff there are positive real constants  $c_0, c_1$  and a positive integer  $n_0$  such that for all  $n \geq n_0$  it holds that

$c_0g(n) \leq f(n) \leq c_1g(n)$ . In other words:

$\exists c_0, c_1 \in \mathbb{R}, \exists n_0 \in \mathbb{N} (c_0 > 0 \wedge c_1 > 0 \wedge n_0 \geq 1 \wedge (\forall n \in \mathbb{N} (n \geq n_0 \rightarrow c_1g(n) \leq f(n) \leq c_2g(n))))$ .

## So what?

WHY DO WE CARE?

What can we use big- $\Omega$  for?

NOT MUCH!

Very very little :)

Though occasionally we can prove things require e.g.  $\Omega(n)$  steps, even if we do not know how to solve it exactly.

But if something is both  $O(f(n))$  and  $\Omega(f(n))$ ...

DEFINITION (BIG- $\Theta$  (THETA))

A function  $f(n)$  is  $\Theta(g(n))$  iff there are positive real constants  $c_0, c_1$  and a positive integer  $n_0$  such that for all  $n \geq n_0$  it holds that

$c_0g(n) \leq f(n) \leq c_1g(n)$ . In other words:

$\exists c_0, c_1 \in \mathbb{R}, \exists n_0 \in \mathbb{N} (c_0 > 0 \wedge c_1 > 0 \wedge n_0 \geq 1 \wedge (\forall n \in \mathbb{N} (n \geq n_0 \rightarrow c_1g(n) \leq f(n) \leq c_2g(n))))$ .

## So what?

WHY DO WE CARE?

What can we use big- $\Omega$  for?

NOT MUCH!

Very very little :)

Though occasionally we can prove things require e.g.  $\Omega(n)$  steps, even if we do not know how to solve it exactly.

But if something is both  $O(f(n))$  and  $\Omega(f(n))\dots$

DEFINITION (BIG- $\Theta$  (THETA))

A function  $f(n)$  is  $\Theta(g(n))$  iff there are positive real constants  $c_0, c_1$  and a positive integer  $n_0$  such that for all  $n \geq n_0$  it holds that

$c_0g(n) \leq f(n) \leq c_1g(n)$ . In other words:

$\exists c_0, c_1 \in \mathbb{R}, \exists n_0 \in \mathbb{N} (c_0 > 0 \wedge c_1 > 0 \wedge n_0 \geq 1 \wedge (\forall n \in \mathbb{N} (n \geq n_0 \rightarrow c_1g(n) \leq f(n) \leq c_2g(n))))$ .

## So what?

WHY DO WE CARE?

What can we use big- $\Omega$  for?

NOT MUCH!

Very very little :)

Though occasionally we can prove things require e.g.  $\Omega(n)$  steps, even if we do not know how to solve it exactly.

But if something is both  $O(f(n))$  and  $\Omega(f(n))\dots$

DEFINITION (BIG- $\Theta$  (THETA))

A function  $f(n)$  is  $\Theta(g(n))$  iff there are positive real constants  $c_0, c_1$  and a positive integer  $n_0$  such that for all  $n \geq n_0$  it holds that

$c_0g(n) \leq f(n) \leq c_1g(n)$ . In other words:

$\exists c_0, c_1 \in \mathbb{R}, \exists n_0 \in \mathbb{N} (c_0 > 0 \wedge c_1 > 0 \wedge n_0 \geq 1 \wedge (\forall n \in \mathbb{N} (n \geq n_0 \rightarrow c_1g(n) \leq f(n) \leq c_2g(n))))$ .

## WHY DO WE CARE ABOUT THIS?

So is big- $\Theta$  any use?

YES!

It is basically the “tight upper bound” we discussed yesterday.

```
1 def maya(n: int) -> int:
2     x = list()
3     for i in range(n):
4         for j in range(n):
5             x += [i * j]
6     s = 0
7     for i in x:
8         s += i ** 1.5
9
10    return s
```

src/big-oh-example.py

So . . .

What is a tight bound on the run time of `maya`?

- A.  $\Theta(1)$
- B.  $\Theta(n)$
- C.  $\Theta(n^2)$
- D.  $\Theta(n^3)$
- E. I don't know.

## WHY DO WE CARE ABOUT THIS?

So is big- $\Theta$  any use?

YES!

It is basically the “tight upper bound” we discussed yesterday.

```
1 def maya(n: int) -> int:
2     x = list()
3     for i in range(n):
4         for j in range(n):
5             x += [i * j]
6     s = 0
7     for i in x:
8         s += i ** 1.5
9
10    return s
```

src/big-oh-example.py

So . . .

What is a tight bound on the run time of `maya`?

- A.  $\Theta(1)$
- B.  $\Theta(n)$
- C.  $\Theta(n^2)$
- D.  $\Theta(n^3)$
- E. I don't know.

## WHY DO WE CARE ABOUT THIS?

So is big- $\Theta$  any use?

YES!

It is basically the “tight upper bound” we discussed yesterday.

```
1 def maya(n: int) -> int:
2     x = list()
3     for i in range(n):
4         for j in range(n):
5             x += [i * j]
6     s = 0
7     for i in x:
8         s += i ** 1.5
9
10    return s
```

src/big-oh-example.py

So . . .

What is a tight bound on the run time of `maya`?

- A.  $\Theta(1)$
- B.  $\Theta(n)$
- C.  $\Theta(n^2)$
- D.  $\Theta(n^3)$
- E. I don't know.

## WHY DO WE CARE ABOUT THIS?

So is big- $\Theta$  any use?

YES!

It is basically the “tight upper bound” we discussed yesterday.

```
1 def maya(n: int) -> int:
2     x = list()
3     for i in range(n):
4         for j in range(n):
5             x += [i * j]
6     s = 0
7     for i in x:
8         s += i ** 1.5
9
10    return s
```

src/big-oh-example.py

So...

What is a tight bound on the run time of `maya`?

- A.  $\Theta(1)$
- B.  $\Theta(n)$
- C.  $\Theta(n^2)$
- D.  $\Theta(n^3)$
- E. I don't know.

## WHY DO WE CARE ABOUT THIS?

So is big- $\Theta$  any use?

YES!

It is basically the “tight upper bound” we discussed yesterday.

```
1 def maya(n: int) -> int:
2     x = list()
3     for i in range(n):
4         for j in range(n):
5             x += [i * j]
6     s = 0
7     for i in x:
8         s += i ** 1.5
9
10    return s
```

src/big-oh-example.py

## RUN TIME

The run time is described as  $T(n) = c_0 + c_1 n + c_2 n^2$ , where

- ▶  $c_0$  is for lines 2, 5, and 9.
- ▶  $c_1 n$  is for the range in line 3.
- ▶  $c_2 n^2$  is for lines 4, 5, and 8.

Thus  $T(n)$  is  $\Theta(n^2)$ .

WHY DO WE CARE ABOUT THIS?

So is big- $\Theta$  any use?

YES!

It is basically the “tight upper bound” we discussed yesterday.

DESPITE ALL THAT...

We still often *just* ask for “a tight upper bound” and will accept a big-Oh.

Thanks ... yogeshkulkarni@yahoo.com