Searching

Searching

- The algorithmic process of finding a particular item in a collection of items.

- Use-cases:
  - Membership test (is this item present?)
  - Return item/position if found

- We will focus on membership test
  - returning the item if found, is a small modification

The Python way

- A lot algorithms we will (tediously) manually build from the ground up, are already available in the basic Python or in a Python library.

- E.g. (code visualization)

```
In [2]: 10 in [1, 2, 3, 4]
Out[2]: False

In [3]: 4 in [1,2,3,4]
Out[3]: True
```

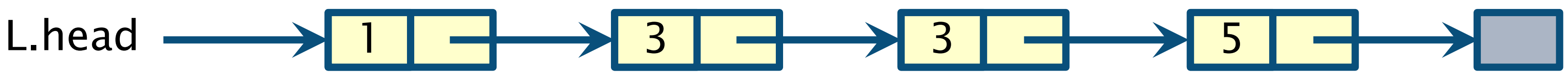
- But using them defeats the purpose of the course.
  - So, we will skip the Tardis and take the long route...
  - We will build (and analyze) search functions from the ground up

Search in singly linked lists

- Find the first element with key k in list L by a simple linear search
  - If found, return a pointer to this element
  - If no object with key k appears in the list, then return NIL
- Iterator *pattern*

```
SEARCH(L,k)
  i := L.head
  while i != NIL and i.key != k
    i := i.next
  return i
```

- Complexity O(n)
- Example
  - Find k=3 in the list below



Search in binary search trees

- **Search for a node with a given key in a BST**
  - Given a pointer to the root of the tree node and a key k, return a pointer to a node with key k if one exists; return NIL otherwise

```
SEARCH(node,key)
  if node==NIL or k==node.key
    return node
  if k < node.key
    return SEARCH(node.left,key)
  else
    return SEARCH(node.right,key)
```

- **Recursive definition**
  - Start with searching at the root node
  - If k is smaller than node.key, continue the search in the left subtree of node
  - The search continues in the right subtree otherwise

- **A recursive algorithm works well here, because the *data structure itself is recursive***
  - Complexity: O(h); can be O(log n) if balanced

- **The correctness of the procedure follows from the binary-search-tree property**

The Sequential Search

An excuse for understanding the
“**incremental approach**” to algorithm design

The Sequential Search

An excuse for understanding “incremental approach” to algorithm design

This is the search you would do when looking for “name against number” in a telephone book.

Items in a sequence, no order to them.

You start at one end, go all the way to the other (or until you find the item)

Let’s code!

[Code visualization for self-study]

Sequential Search

- "Naïve" approach:
  - Keep searching "myopically"
  - Start from one end and go all the way to the other, until you find the item
  - *No use of additional structural properties* of the underlying data structure, like ordering

Complexity

If item is present:	Best case ? Worst case ?
If item is not present:	Best case ? Worst case ?

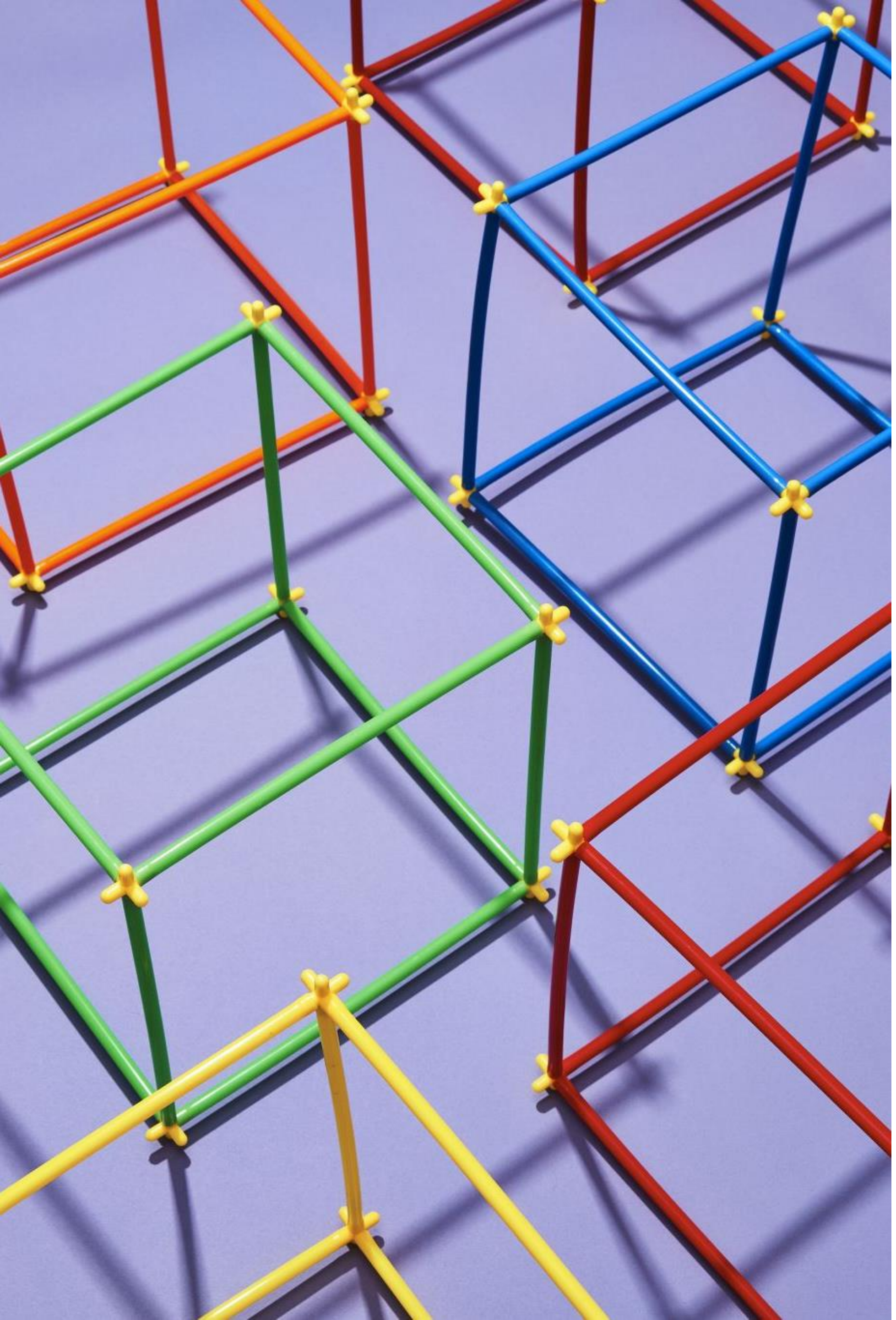
Sequential Search

- “Naïve” approach:
  - Keep searching “myopically”
  - Start from one end and go all the way to the other, until you find the item
  - *No use of additional structural properties* of the underlying data structure, like ordering

* Where n is the size of the array/data structure we are searching in

Complexity*

If item is present:	Best case $O(1)$ Worst case $O(n)$
If item is not present:	Best case $O(n)$ Worst case $O(n)$



Algorithmic design patterns: **Incremental Approach**

- We are going to use searching and sorting as a means to understanding some common algorithmic design patterns

- The one you just saw in the sequential search, is called the **incremental approach** to algorithm design.

- Basically: solution works: **one element at a time**
  - or, equivalently, one *element* each *step*
  - O(n) complexity algorithm

What if?

- The list was ordered?
- We can stop searching when go past the value of interest?
- Let’s code!
- Link to code visualization for self-review

Have we really used the sorted list to our advantage?

- We are still at O(n) complexity
- You reduce the time/steps only for the case where the item is *not* in the list
  - In the worst case (item not present or is at the last location), we still have to iterate through the entire list
- How would you go about searching for a name in a sorted directory/dictionary?
  - Or a number in a sorted deck of numbered cards?


I have a cunning plan.

The Binary Search

- We can take greater advantage of the ordered list if we be a bit clever
- E.g., looking for 100 in [1,2,3,4,15,17,19,20,21,25,28,100,120]

mid

1	2	3	4	15	17	19	20	21	25	28	100	120

0 mid+1

The Binary Search Algorithm

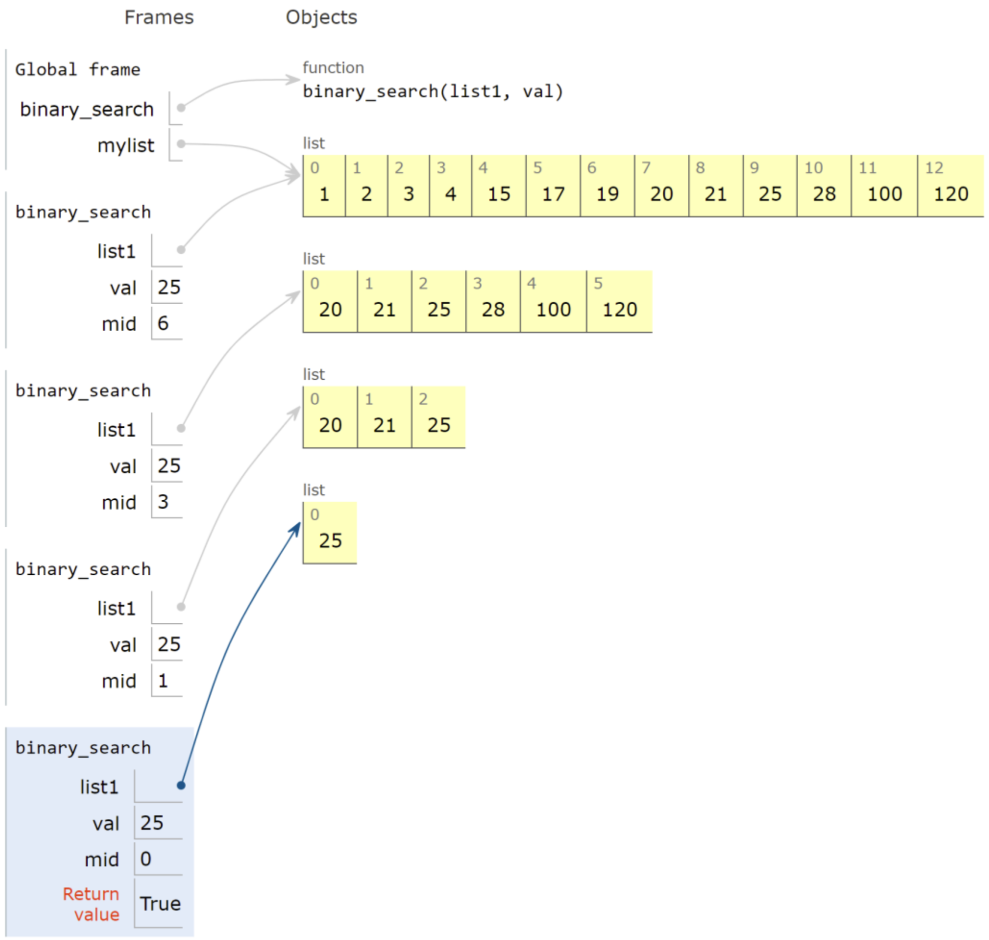
1. Look at the middle value; then
2. if search item is **greater than** the value at the middle, it must be on its right. Discard middle item and left half. Repeat 1
3. if search item is less than the value at the middle, it must be on its left. Discard the middle item and right half. Repeat 1
4. If the middle item is the search item, or no more elements left, exit.

The **Divide and Conquer** Algorithmic Pattern

- Binary search is a great example of this pattern
- In general: divide and conquer means we:
  - divide the problem into smaller pieces,
  - solve the smaller pieces, and then
  - (if applicable) reassemble the problem back up to get the result
- Very often you can program it using a recursive function

Let's code this gremlin

1. Look at the middle value; then
2. if search item is **greater than** the value at the middle, it must be on its right. Discard middle item and left half. Repeat 1
3. if search item is less than the value at the middle, it must be on its left. Discard the middle item and right half. Repeat 1
4. If the middle item is the search item, or no more elements left, exit.



[Link to code and visualization]

Analysing Binary search

- When problem size is halved in every step, this is O(log n) complexity

- Or:

Analysing Binary search

- When problem size is halved in every step, this is O(log n) complexity
- Or:

iter	size
1	$n/2$
2	$n/4 = \frac{n}{2\times 2}$
3	$n/8 = \frac{n}{2\times 2\times 2}$
$\vdots$	
i	$\frac{n}{2^i} = \frac{n}{2\times 2\times \cdots \times 2}$

$$\boxed{\frac{n}{2^i} = 1}$$

$$n = 2^i$$

$$i = \log n$$