ETH zürich

Departement Informatik

Programming and Problem-Solving

Binary Search an Recursion

Dennis Komm

Spring 2021 – April 29, 2021

Searching

Linear Search

Linear Search

Run once through list from left to right and compare each element to he sought one

- Most straightforward strategy to search
- Works for unsorted data

Linear Search

Run once through list from left to right and compare each element to he sought one

- Most straightforward strategy to search
- Works for unsorted data
- Needs up to n comparisons on list of length n if sought element is at last position (or does not appear)

Linear Search

Run once through list from left to right and compare each element to he sought one

- Most straightforward strategy to search
- Works for unsorted data
- Needs up to n comparisons on list of length n if sought element is at last position (or does not appear)
- Time complexity in $\mathcal{O}(n)$

Linear Search

```python
def linsearch(data, searched):
    index = 0
    while index < len(data):
        if data[index] == searched:
            return index
        index += 1
    return -1
```

Linear Search

```python
def linsearch(data, searched):
    index = 0
    while index < len(data):
        if data[index] == searched:
            return index
        index += 1
    return -1
```

```python
def linsearch(data, searched):
    if searched in data:
        return True
    else:
        return False
```

Linear Search

```python
def linsearch(data, searched):
    index = 0
    while index < len(data):
        if data[index] == searched:
            return index
        index += 1
    return -1
```

```python
def linsearch(data, searched):
    if searched in data:
        return True
    else:
        return False
```

```python
def linsearch(data, searched):
    return searched in data
```

Searching

Binary Search

Binary Search

Example

Given a list with the first 8 prime numbers, find the position of 17

Binary Search

Example

Given a list with the first 8 prime numbers, find the position of 17

2	3	5	7	11	13	17	19
0	1	2	3	4	5	6	7

$\boxed{17}$

Binary Search

Example

Given a list with the first 8 prime numbers, find the position of 17

2	3	5	7	11	13	17	19
0	1	2	3	4	5	6	7

$\boxed{17}$

Binary Search

Example

Given a list with the first 8 prime numbers, find the position of 17

2	3	5	7	11	13	17	19
0	1	2	3	4	5	6	7

$\boxed{17}$

Binary Search

Example

Given a list with the first 8 prime numbers, find the position of 17

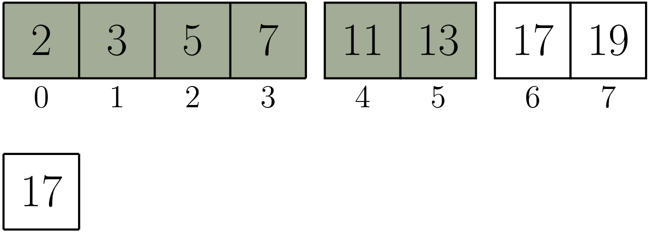
2	3	5	7	11	13	17	19
0	1	2	3	4	5	6	7

17

Binary Search

Example

Given a list with the first 8 prime numbers, find the position of 17



Binary Search

Example

Given a list with the first 8 prime numbers, find the position of 17

2	3	5	7	11	13	17	19
0	1	2	3	4	5	6	7

17

Binary Search

Example

Given a list with the first 8 prime numbers, find the position of 17

2	3	5	7	11	13	17	19
0	1	2	3	4	5	6	7

$\boxed{17}$

Binary Search

Use that data is sorted

Binary Search

Use that data is sorted

- Two variables `left` and `right`
- These specify **search space**
- Look at value in the middle (index `current`)

Binary Search

Use that data is sorted

- Two variables `left` and `right`
- These specify **search space**
- Look at value in the middle (index `current`)
- If this value is what we searched for, we are done

Binary Search

Use that data is sorted

- Two variables `left` and `right`
- These specify **search space**
- Look at value in the middle (index `current`)
- If this value is what we searched for, we are done
- If this value is too small, then also everything left of it is too small

⇨ `left = current + 1`

Binary Search

Use that data is sorted

- Two variables `left` and `right`
- These specify **search space**
- Look at value in the middle (index `current`)
- If this value is what we searched for, we are done
- If this value is too small, then also everything left of it is too small

⇨ `left = current + 1`

- If this value is too small, then also everything right of it is too large

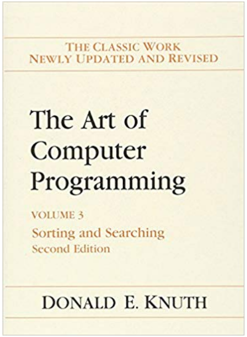
⇨ `right = current - 1`

Binary Search

Sorting and searching data are two of the fundamental tasks of computer scientists

Binary Search

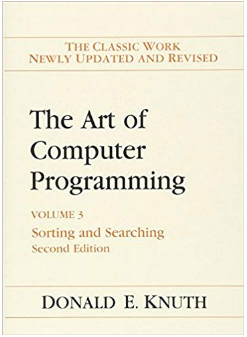
Sorting and searching data are two of the fundamental tasks of computer scientists



Binary Search

Sorting and searching data are two of the fundamental tasks of computer scientists

The first binary search was published in 1946 (and the principle was known long before), but the first version that works correctly for all n only appeared 14 years later



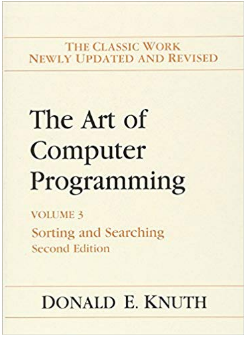
Binary Search

Sorting and searching data are two of the fundamental tasks of computer scientists

The first binary search was published in 1946 (and the principle was known long before), but the first version that works correctly for all n only appeared 14 years later

"Although the basic idea of binary search is comparatively straightforward, the details can be surprisingly tricky..."

–Donald Knuth



Exercise – Binary Search

Implement binary search

- as Python function
- using three “pointers” `left`, `right`, and `current`
- Initially, set `left = 0` and `right = len(data) - 1`
- In every step, shrink search space as described
- If element is found, its position is returned
- Otherwise, -1 is returned

Binary Search

```python
def binsearch(data, searched):
    left = 0
    right = len(data) - 1
    while left <= right:
        current = (left + right) // 2
        if data[current] == searched:
            return current
        elif data[current] > searched:
            right = current - 1
        else:
            left = current + 1
    return -1
```

Searching

Complexity of Binary Search

Complexity of Binary Search

- At first, there are n elements

Complexity of Binary Search

- At first, there are n elements
- With every iteration, the **search space** is halved

Complexity of Binary Search

- At first, there are n elements
- With every iteration, the **search space** is halved
- After the first iteration, there remain $n/2$ elements

Complexity of Binary Search

- At first, there are n elements
- With every iteration, the **search space** is halved
- After the first iteration, there remain $n/2$ elements
- After the second iteration, there remain $n/4$ elements

Complexity of Binary Search

- At first, there are n elements
- With every iteration, the **search space** is halved
- After the first iteration, there remain $n/2$ elements
- After the second iteration, there remain $n/4$ elements
- …

Complexity of Binary Search

- At first, there are n elements
- With every iteration, the **search space** is halved
- After the first iteration, there remain $n/2$ elements
- After the second iteration, there remain $n/4$ elements
- $\ldots$
- After how many iterations x does there remain only one element?

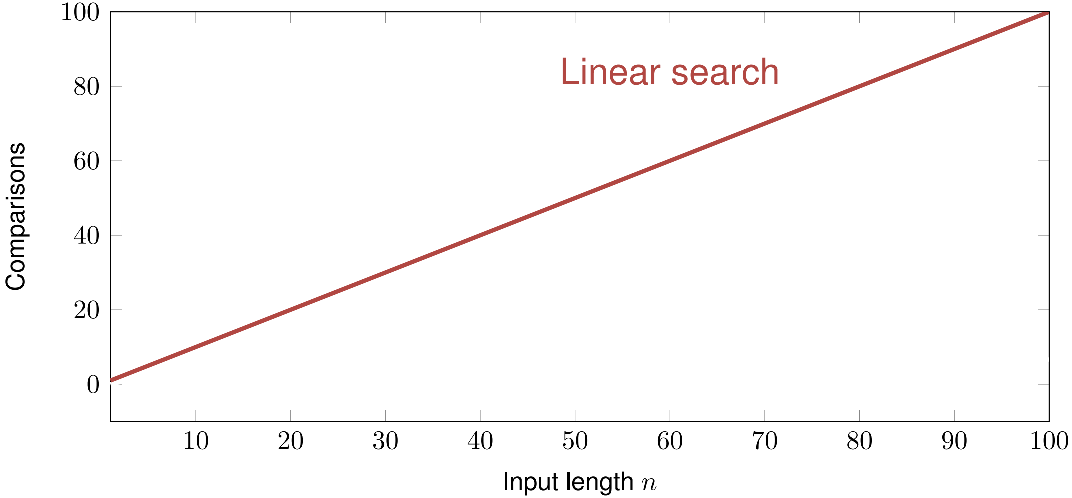
Complexity of Binary Search

- At first, there are n elements
- With every iteration, the **search space** is halved
- After the first iteration, there remain $n/2$ elements
- After the second iteration, there remain $n/4$ elements
- $\ldots$
- After how many iterations x does there remain only one element?
- $n/2^x = 1 \iff n = 2^x \iff x = \log_2 n$

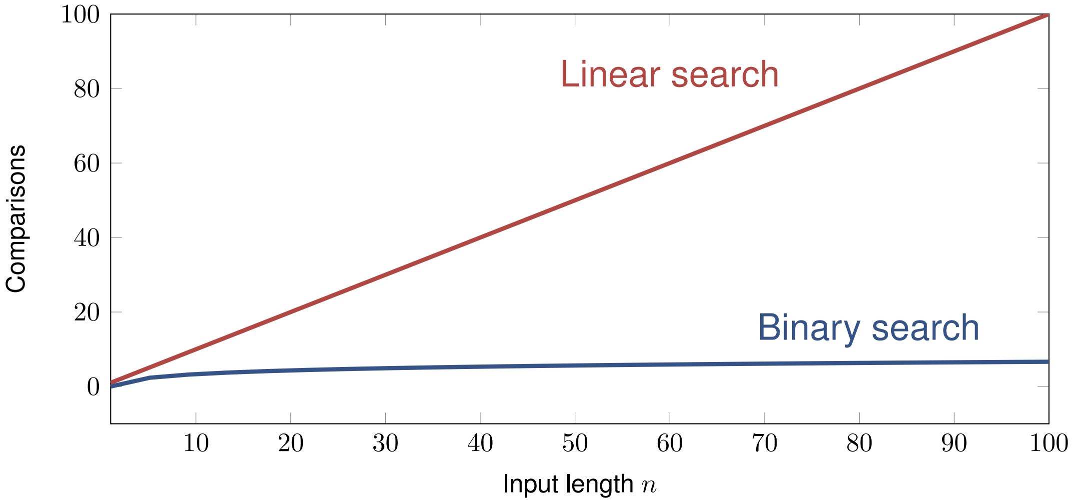
Complexity of Binary Search

- At first, there are n elements
- With every iteration, the **search space** is halved
- After the first iteration, there remain $n/2$ elements
- After the second iteration, there remain $n/4$ elements
- $\dots$
- After how many iterations x does there remain only one element?
- $n/2^x = 1 \iff n = 2^x \iff x = \log_2 n$
- Time complexity in $\mathcal{O}(\log_2 n)$

Complexity of Binary Search



Complexity of Binary Search



Complexity of Binary Search

- We again use a variable `counter` to count the comparisons
- Algorithm is executed on sorted lists with values 1 to n
- The value of n grows by 1 with every iteration
- Initially, n is 1, at the end $1\,000\,000$
- The first element 1 is always sought
- Results are stored in a list and plotted using `matplotlib`

Complexity of Binary Search

Worst Case

```
values = []
data = [1]


for i in range(1, 1000001):
    data.append(data[-1] + 1)
    values.append(binsearch(data, 1))


plt.plot(values, color="red")
plt.show()
```

Complexity of Binary Search

Worst Case

```
values = []
data = [1]


for i in range(1, 1000001):
    data.append(data[-1] + 1)
    values.append(binsearch(data, 1))


plt.plot(values, color="red")
plt.show()
```

Add element that is larger by 1 than the currently last

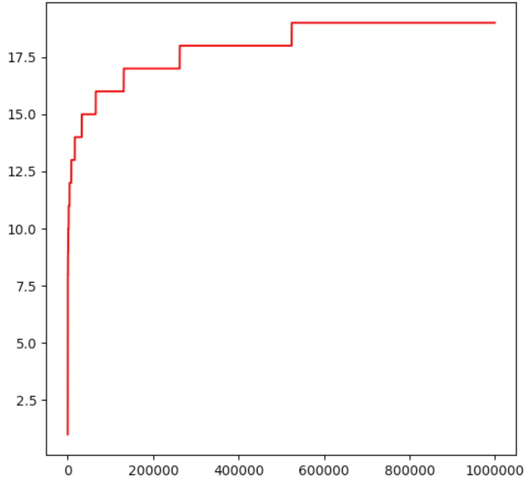
Complexity of Binary Search

Worst Case

```python
values = []
data = [1]


for i in range(1, 1000001):
    data.append(data[-1] + 1)
    values.append(binsearch(data, 1))


plt.plot(values, color="red")
plt.show()
```



Complexity of Binary Search

What happens if data is unsorted?

Complexity of Binary Search

What happens if data is unsorted?

- Linear search always works for unsorted lists and is in $\mathcal{O}(n)$

Complexity of Binary Search

What happens if data is unsorted?

- Linear search always works for unsorted lists and is in $\mathcal{O}(n)$
- Sorting can pay off for multiple searches
- Sorting is in $\mathcal{O}(n \log_2 n)$ and is therefore slower than linear search
- Binary search is in $\mathcal{O}(\log_2 n)$ and is consequently much faster than linear search

Complexity of Binary Search

What happens if data is unsorted?

- Linear search always works for unsorted lists and is in $\mathcal{O}(n)$
- Sorting can pay off for multiple searches
- Sorting is in $\mathcal{O}(n \log_2 n)$ and is therefore slower than linear search
- Binary search is in $\mathcal{O}(\log_2 n)$ and is consequently much faster than linear search

When does sorting pay off?

Complexity of Binary Search

What happens if data is unsorted?

- Linear search always works for unsorted lists and is in $\mathcal{O}(n)$
- Sorting can pay off for multiple searches
- Sorting is in $\mathcal{O}(n \log_2 n)$ and is therefore slower than linear search
- Binary search is in $\mathcal{O}(\log_2 n)$ and is consequently much faster than linear search

When does sorting pay off?

If more than $\log_2 n$ searches are made

Recursive Functions

Recursive Functions

`def f():` $\iff$ Python "learns" new word `f`

Recursive Functions

`def f():` $\Longleftrightarrow$ Python “learns” new word `f`

From Merriam-Webster dictionary

re·frig·er·a·tor

A room or appliance for keeping food or other items cool

Recursive Functions

`def f():` $\Longleftrightarrow$ Python "learns" new word `f`

From Merriam-Webster dictionary

re·frig·er·a·tor

A room or appliance for keeping food or other items cool

- This analogy is not entirely correct
- Such functions are called **recursive functions**

Recursive Functions

`def f():` $\Longleftrightarrow$ Python “learns” new word `f`

From Merriam-Webster dictionary

re·frig·er·a·tor

A room or appliance for keeping food or other items cool

- This analogy is not entirely correct
- Such functions are called **recursive functions**

Not from Merriam-Webster dictionary

re·frig·er·a·tor

A refrigerator

Recursive Functions

This results in an **endless loop**

Recursive Functions

This results in an **endless loop**

```
def f():
    print("Hello world!")
    f()
```

Recursive Functions

This results in an **endless loop**

```
def f():
    print("Hello world!")
    f()
```

```
f()
```

Recursive Functions

This results in an endless loop

```python
def f():
    print("Hello world!")
    f()
```

```python
f()
```

```python
print("Hello world!")
f()
```

Recursive Functions

This results in an **endless loop**

```python
def f():
    print("Hello world!")
    f()
```

```
f()
```

```
print("Hello world!")
f()
```

```
print("Hello world!")
f()
```

Recursive Functions

This results in an **endless loop**

```
def f():
    print("Hello world!")
    f()
```

```
f()
```

```
print("Hello world!")
f()
```

```
print("Hello world!")
f()
```

```
print("Hello world!")
f()
```

Recursive Functions

This results in an **endless loop**

```python
def f():
    print("Hello world!")
    f()
```

```python
f()
```

```python
print("Hello world!")
f()
```

```python
print("Hello world!")
f()
```

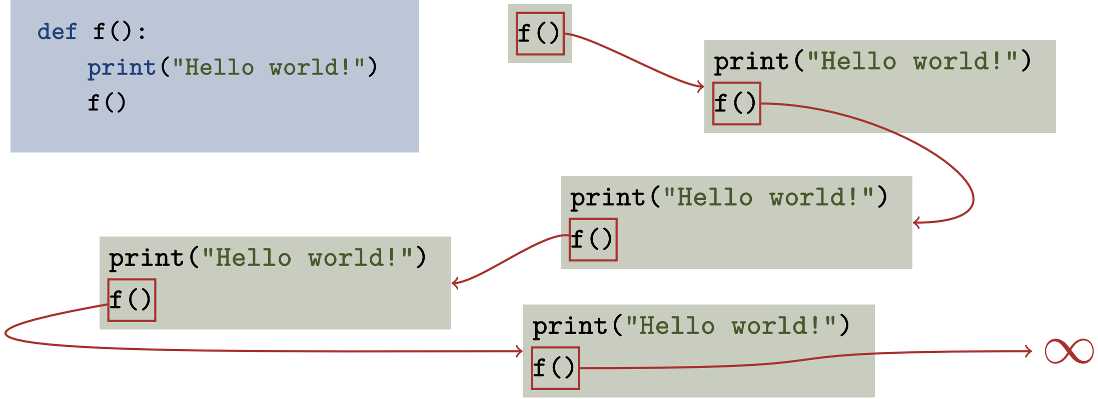
```python
print("Hello world!")
f()
```

```python
print("Hello world!")
f()
```

Recursive Functions

This results in an **endless loop**

```python
def f():
    print("Hello world!")
    f()
```



Recursive Functions

We use parameters to end after a finite number of calls

Recursive Functions

We use parameters to end after a finite number of calls

```
def f(k):
    print(k)
    if k == 1:
        return
    else:
        f(k-1)
```

Recursive Functions

We use parameters to end after a finite number of calls

```
def f(k):
    print(k)
    if k == 1:
        return
    else:
        f(k-1)
```

Parameter (or any local variable) is newly created for every function call

Recursive Functions

We use parameters to end after a finite number of calls

```
def f(k):
    print(k)
    if k == 1:
        return
    else:
        f(k-1)
```

```
f(4)
```

Recursive Functions

We use parameters to end after a finite number of calls

```
def f(k):
    print(k)
    if k == 1:
        return
    else:
        f(k-1)
```

```
f(4)
```

```
print(4)
if 4 == 1:
    return
else:
    f(3)
```

Recursive Functions

We use parameters to end after a finite number of calls

```
def f(k):
    print(k)
    if k == 1:
        return
    else:
        f(k-1)
```

```
f(4)
```

```
print(4)
if 4 == 1:
    return
else:
    f(3)
```

```
print(3)
if 3 == 1:
    return
else:
    f(2)
```

Recursive Functions

We use parameters to end after a finite number of calls

```python
def f(k):
    print(k)
    if k == 1:
        return
    else:
        f(k-1)
```

```
f(4)
```

```
print(4)
if 4 == 1:
    return
else:
    f(3)
```

```
print(3)
if 3 == 1:
    return
else:
    f(2)
```

```
print(2)
if 2 == 1:
    return
else:
    f(1)
```

Recursive Functions

We use parameters to end after a finite number of calls

```
def f(k):
    print(k)
    if k == 1:
        return
    else:
        f(k-1)
```

```
f(4)
```

```
print(4)
if 4 == 1:
    return
else:
    f(3)
```

```
print(3)
if 3 == 1:
    return
else:
    f(2)
```

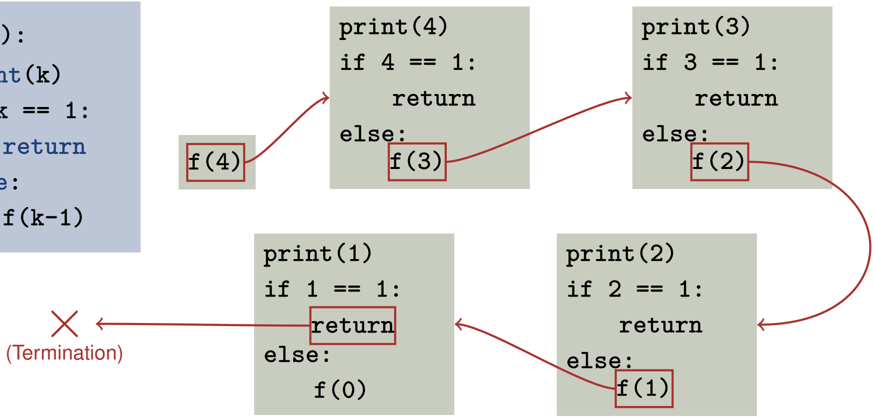
```
print(2)
if 2 == 1:
    return
else:
    f(1)
```

```
print(1)
if 1 == 1:
    return
else:
    f(0)
```

Recursive Functions

We use parameters to end after a finite number of calls

```
def f(k):
    print(k)
    if k == 1:
        return
    else:
        f(k-1)
```



Factorial and Sum

Computing the Factorial Recursively

- Factorial of a natural number n is defined by

$$\mathrm{fact}(n) = n! = n \cdot (n-1) \cdot (n-2) \cdot \dots \cdot 2 \cdot 1$$

Computing the Factorial Recursively

- Factorial of a natural number n is defined by

$$\mathrm{fact}(n) = n! = n \cdot (n-1) \cdot (n-2) \cdot \dots \cdot 2 \cdot 1$$

- For instance, $7! = 7 \cdot 6 \cdot 5 \cdot 4 \cdot 3 \cdot 2 \cdot 1 = 5040$

Computing the Factorial Recursively

- Factorial of a natural number n is defined by

$$\mathrm{fact}(n) = n! = n \cdot (n-1) \cdot (n-2) \cdot \cdots \cdot 2 \cdot 1$$

- For instance, $7! = 7 \cdot 6 \cdot 5 \cdot 4 \cdot 3 \cdot 2 \cdot 1 = 5040$
- We observe

$$n! = n \cdot (n-1)! = n \cdot (n-1) \cdot (n-2)! = \ldots$$

Computing the Factorial Recursively

- Factorial of a natural number n is defined by

$$\mathrm{fact}(n) = n! = n \cdot (n-1) \cdot (n-2) \cdot \dots \cdot 2 \cdot 1$$

- For instance, $7! = 7 \cdot 6 \cdot 5 \cdot 4 \cdot 3 \cdot 2 \cdot 1 = 5040$
- We observe

$$n! = n \cdot (n-1)! = n \cdot (n-1) \cdot (n-2)! = \dots$$

- Function can be computed recursively by

$$\mathrm{fact}(1) = 1 \quad \text{and} \quad \mathrm{fact}(n) = n \cdot \mathrm{fact}(n-1)$$

Computing the Factorial Recursively – in Python

```python
def fact(n):
    if n == 1:
        return 1
    else:
        return n * fact(n-1)
```

Computing the Factorial Recursively – in Python

```
def fact(n):
    if n == 1:
        return 1
    else:
        return n * fact(n-1)
```

As before, parameter is newly created for every function call

Computing the Factorial Recursively – in Python

```
def fact(n):
    if n == 1:
        return 1
    else:
        return n * fact(n-1)
```

Last call returns fixed value 1

Computing the Factorial Recursively – in Python

```
def fact(n):
    if n == 1:
        return 1
    else:
        return n * fact(n-1)
```

Other calls return
n times “factorial of `n-1`”

Computing the Factorial Recursively – in Python

```
def fact(n):
    if n == 1:
        return 1
    else:
        return n * fact(n-1)
```

Call Stack

`fact(7)`

Computing the Factorial Recursively – in Python

```python
def fact(n):
    if n == 1:
        return 1
    else:
        return n * fact(n-1)
```

Call Stack

`fact(6)`
`fact(7)`

Computing the Factorial Recursively – in Python

```python
def fact(n):
    if n == 1:
        return 1
    else:
        return n * fact(n-1)
```

Call Stack

fact(5)

fact(6)

fact(7)

Computing the Factorial Recursively – in Python

```python
def fact(n):
    if n == 1:
        return 1
    else:
        return n * fact(n-1)
```

Call Stack

```
fact(4)
fact(5)
fact(6)
fact(7)
```

Computing the Factorial Recursively – in Python

```
def fact(n):
    if n == 1:
        return 1
    else:
        return n * fact(n-1)
```

Call Stack

```
fact(3)
fact(4)
fact(5)
fact(6)
fact(7)
```

Computing the Factorial Recursively – in Python

```
def fact(n):
    if n == 1:
        return 1
    else:
        return n * fact(n-1)
```

Call Stack

fact(2)

fact(3)

fact(4)

fact(5)

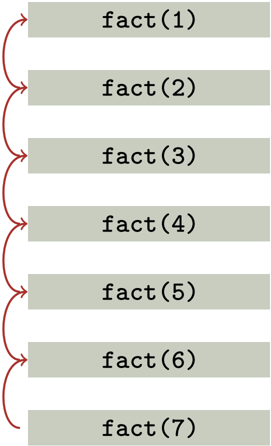
fact(6)

fact(7)

Computing the Factorial Recursively – in Python

```python
def fact(n):
    if n == 1:
        return 1
    else:
        return n * fact(n-1)
```

Call Stack



Computing the Factorial Recursively – in Python

```
def fact(n):
    if n == 1:
        return 1
    else:
        return n * fact(n-1)
```

Call Stack

fact(1) return 1

fact(2)

fact(3)

fact(4)

fact(5)

fact(6)

fact(7)

Computing the Factorial Recursively – in Python

```python
def fact(n):
    if n == 1:
        return 1
    else:
        return n * fact(n-1)
```

Call Stack

fact(2) return 2 * 1

fact(3)

fact(4)

fact(5)

fact(6)

fact(7)

Computing the Factorial Recursively – in Python

```python
def fact(n):
    if n == 1:
        return 1
    else:
        return n * fact(n-1)
```

Call Stack

`fact(2)` `return 2`

`fact(3)`

`fact(4)`

`fact(5)`

`fact(6)`

`fact(7)`

Computing the Factorial Recursively – in Python

```python
def fact(n):
    if n == 1:
        return 1
    else:
        return n * fact(n-1)
```

Call Stack

```
fact(3)        return 3 * 2
fact(4)
fact(5)
fact(6)
fact(7)
```

Computing the Factorial Recursively – in Python

```python
def fact(n):
    if n == 1:
        return 1
    else:
        return n * fact(n-1)
```

Call Stack

`fact(3)` `return 6`

`fact(4)`

`fact(5)`

`fact(6)`

`fact(7)`

Computing the Factorial Recursively – in Python

```
def fact(n):
    if n == 1:
        return 1
    else:
        return n * fact(n-1)
```

Call Stack

`fact(4)`   `return 4 * 6`

`fact(5)`

`fact(6)`

`fact(7)`

Computing the Factorial Recursively – in Python

```python
def fact(n):
    if n == 1:
        return 1
    else:
        return n * fact(n-1)
```

Call Stack

```
fact(4)        return 24

fact(5)

fact(6)

fact(7)
```

Computing the Factorial Recursively – in Python

```
def fact(n):
    if n == 1:
        return 1
    else:
        return n * fact(n-1)
```

Call Stack

fact(5)	return 5 * 24
fact(6)	
fact(7)	

Computing the Factorial Recursively – in Python

```
def fact(n):
    if n == 1:
        return 1
    else:
        return n * fact(n-1)
```

Call Stack

`fact(5)`	`return 120`
`fact(6)`	
`fact(7)`	

Computing the Factorial Recursively – in Python

```
def fact(n):
    if n == 1:
        return 1
    else:
        return n * fact(n-1)
```

Call Stack

```
fact(6)        return 6 * 120
fact(7)
```

Computing the Factorial Recursively – in Python

```
def fact(n):
    if n == 1:
        return 1
    else:
        return n * fact(n-1)
```

Call Stack

`fact(6)`	`return 720`
`fact(7)`	

Computing the Factorial Recursively – in Python

```
def fact(n):
    if n == 1:
        return 1
    else:
        return n * fact(n-1)
```

Call Stack

`fact(7)`	`return 7 * 720`

Computing the Factorial Recursively – in Python

```
def fact(n):
    if n == 1:
        return 1
    else:
        return n * fact(n-1)
```

Call Stack

`fact(7)` `return 5040`

Exercise – Computing a Sum Recursively

Implement a recursive function that

- takes a parameter `n`
- and returns the sum of the first `n` natural numbers

Exercise – Computing a Sum Recursively

Both recursive functions can be implemented with the same idea

```
def fact(n):
    if n == 1:
        return 1
    else:
        return n * fact(n-1)
```

```
def thesum(n):
    if n == 1:
        return 1
    else:
        return n + thesum(n-1)
```

Recursion vs. Iteration

- There are alternatives using loops

```
def fact(n):
    i = 1
    result = 1
    while i < n:
        i += 1
        result *= i
    return result
```

```
def thesum(n):
    i = 1
    result = 1
    while i < n:
        i += 1
        result += i
    return result
```

Recursion vs. Iteration

- There are alternatives using loops

```
def fact(n):
    i = 1
    result = 1
    while i < n:
        i += 1
        result *= i
    return result
```

```
def thesum(n):
    i = 1
    result = 1
    while i < n:
        i += 1
        result += i
    return result
```

- For the sum, there is also a closed form (from the Bubblesort analysis)

```
def thesum(n):
    return n * (n+1) / 2
```

Recursion vs. Iteration

If repeated statements are implemented using loops, we speak of
iterative programming

Recursion vs. Iteration

If repeated statements are implemented using loops, we speak of **iterative programming**

- For all problems, there exist both iterative and recursive solutions

Recursion vs. Iteration

If repeated statements are implemented using loops, we speak of
iterative programming

- For all problems, there exist both iterative and recursive solutions
- The recursive solution can often be viewed as more “elegant”

Recursion vs. Iteration

If repeated statements are implemented using loops, we speak of
iterative programming

- For all problems, there exist both iterative and recursive solutions
- The recursive solution can often be viewed as more “elegant”
- The implementation using recursion is often shorter (more concise) to write
- . . . but almost never faster to execute

Recursion vs. Iteration

If repeated statements are implemented using loops, we speak of **iterative programming**

- For all problems, there exist both iterative and recursive solutions
- The recursive solution can often be viewed as more “elegant”
- The implementation using recursion is often shorter (more concise) to write
- . . . but almost never faster to execute
- What should be used, depends on multiple factors

Euclid’s Algorithm Recursively

Euclid's Algorithm Recursively

Euclid's Algorithm

known from the first lecture

- Input: integers $a > 0$, $b > 0$
- Output: gcd of a and b

```
def euclid(a, b):
    while b != 0:
        if a > b:
            a = a - b
        else:
            b = b - a
    return a
```

Euclid's Algorithm Recursively

Euclid's Algorithm

known from the first lecture

- Input: integers $a > 0$, $b > 0$
- Output: gcd of a and b

```python
def euclid(a, b):
    while b != 0:
        if a > b:
            a = a - b
        else:
            b = b - a
    return a
```

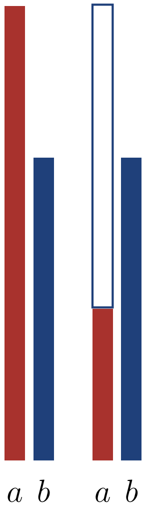
Euclid's Algorithm Recursively

Euclid's Algorithm

known from the first lecture

- Input: integers $a > 0$, $b > 0$
- Output: gcd of a and b

```
def euclid(a, b):
    while b != 0:
        if a > b:
            a = a - b
        else:
            b = b - a
    return a
```



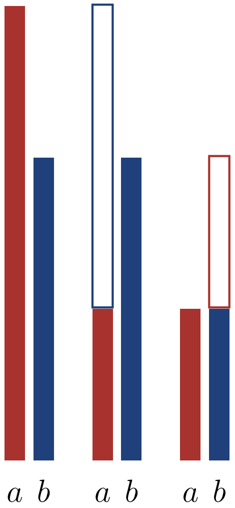
Euclid's Algorithm Recursively

Euclid's Algorithm

known from the first lecture

- Input: integers $a > 0$, $b > 0$
- Output: gcd of a and b

```python
def euclid(a, b):
    while b != 0:
        if a > b:
            a = a - b
        else:
            b = b - a
    return a
```



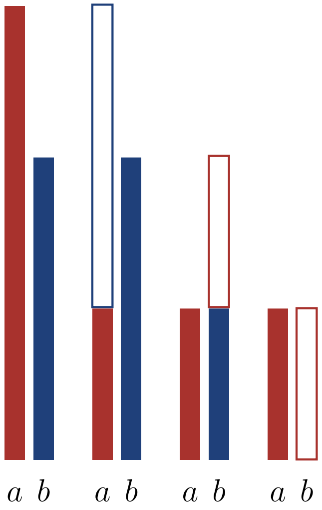
Euclid's Algorithm Recursively

Euclid's Algorithm

known from the first lecture

- Input: integers $a > 0$, $b > 0$
- Output: gcd of a and b

```
def euclid(a, b):
    while b != 0:
        if a > b:
            a = a - b
        else:
            b = b - a
    return a
```



Exercise – Computing the GCD Recursively

Implement Euclid's Algorithm

- as a recursive Python function
- that takes two parameters `a` and `b`

```python
def euclid(a, b):
    while b != 0:
        if a > b:
            a = a - b
        else:
            b = b - a
    return a
```

Computing the GCD Recursively

```python
def euclid(a, b):
    if b == 0:
        return a
    else:
        if a > b:
            return euclid(a - b, b)
        else:
            return euclid(a, b - a)
```

Computing the GCD Recursively

```python
def euclid(a, b):
    if b == 0:
        return a
    else:
        if a > b:
            return euclid(a - b, b)
        else:
            return euclid(a, b - a)
```

```python
def euclid(a, b):
    while b != 0:


        if a > b:
            a = a - b
        else:
            b = b - a
    return a
```

Computing the GCD Recursively

```python
def euclid(a, b):
    if b == 0:
        return a
    else:
        if a > b:
            return euclid(a - b, b)
        else:
            return euclid(a, b - a)
```

Computing the GCD Recursively

```
def euclid(a, b):
    if b == 0:
        return a
    else:
        if a > b:
            return euclid(a - b, b)
        else:
            return euclid(a, b - a)
```

Call Stack

`return` value is passed through

`euclid(119, 68)`

Computing the GCD Recursively

```
def euclid(a, b):
    if b == 0:
        return a
    else:
        if a > b:
            return euclid(a - b, b)
        else:
            return euclid(a, b - a)
```

Call Stack

`return` value is passed through

```
euclid(51, 68)
euclid(119, 68)
```

Computing the GCD Recursively

```python
def euclid(a, b):
    if b == 0:
        return a
    else:
        if a > b:
            return euclid(a - b, b)
        else:
            return euclid(a, b - a)
```

Call Stack

`return` value is passed through

`euclid(51, 17)`

`euclid(51, 68)`

`euclid(119, 68)`

Computing the GCD Recursively

```python
def euclid(a, b):
    if b == 0:
        return a
    else:
        if a > b:
            return euclid(a - b, b)
        else:
            return euclid(a, b - a)
```

Call Stack

`return` value is passed through

```
euclid(34, 17)
euclid(51, 17)
euclid(51, 68)
euclid(119, 68)
```

Computing the GCD Recursively

```python
def euclid(a, b):
    if b == 0:
        return a
    else:
        if a > b:
            return euclid(a - b, b)
        else:
            return euclid(a, b - a)
```

Call Stack

`return` value is passed through

`euclid(17, 17)`

`euclid(34, 17)`

`euclid(51, 17)`

`euclid(51, 68)`

`euclid(119, 68)`

Computing the GCD Recursively

```python
def euclid(a, b):
    if b == 0:
        return a
    else:
        if a > b:
            return euclid(a - b, b)
        else:
            return euclid(a, b - a)
```

Call Stack

`return` value is passed through

```
euclid(17, 0)
euclid(17, 17)
euclid(34, 17)
euclid(51, 17)
euclid(51, 68)
euclid(119, 68)
```

Computing the GCD Recursively

```python
def euclid(a, b):
    if b == 0:
        return a
    else:
        if a > b:
            return euclid(a - b, b)
        else:
            return euclid(a, b - a)
```

Call Stack

`return` value is passed through

```
euclid(17, 0)        return 17
euclid(17, 17)
euclid(34, 17)
euclid(51, 17)
euclid(51, 68)
euclid(119, 68)
```

Computing the GCD Recursively

```python
def euclid(a, b):
    if b == 0:
        return a
    else:
        if a > b:
            return euclid(a - b, b)
        else:
            return euclid(a, b - a)
```

Call Stack

`return` value is passed through

```
euclid(17, 17)      return 17
euclid(34, 17)
euclid(51, 17)
euclid(51, 68)
euclid(119, 68)
```

Computing the GCD Recursively

```python
def euclid(a, b):
    if b == 0:
        return a
    else:
        if a > b:
            return euclid(a - b, b)
        else:
            return euclid(a, b - a)
```

Call Stack

`return` value is passed through

```
euclid(34, 17)      return 17
euclid(51, 17)
euclid(51, 68)
euclid(119, 68)
```

Computing the GCD Recursively

```python
def euclid(a, b):
    if b == 0:
        return a
    else:
        if a > b:
            return euclid(a - b, b)
        else:
            return euclid(a, b - a)
```

Call Stack

`return` value is passed through

`euclid(51, 17)`   `return 17`

`euclid(51, 68)`

`euclid(119, 68)`

Computing the GCD Recursively

```python
def euclid(a, b):
    if b == 0:
        return a
    else:
        if a > b:
            return euclid(a - b, b)
        else:
            return euclid(a, b - a)
```

Call Stack

`return` value is passed through

```
euclid(51, 68)      return 17
euclid(119, 68)
```

Computing the GCD Recursively

```python
def euclid(a, b):
    if b == 0:
        return a
    else:
        if a > b:
            return euclid(a - b, b)
        else:
            return euclid(a, b - a)
```

Call Stack

`return` value is passed through

`euclid(119, 68)` `return 17`

Recursive Sorting and Searching

Binary Search

Iterative Binary Search

```
def binsearch(data, searched):
    left = 0
    right = len(data) - 1
    while left <= right:
        current = (left + right) // 2
        if data[current] == searched:
            return current
        elif data[current] > searched:
            right = current - 1
        else:
            left = current + 1
    return -1
```

Recursive Binary Search

Recursive Implementation

- Function again takes parameters `data` and for the given list and the searched element

Recursive Binary Search

Recursive Implementation

- Function again takes parameters `data` and for the given list and the searched element
- Two parameters `left` and `right` define the current **search space**
- In a single call, `left` and `right` are not changed

⇨ **No** loop

Recursive Binary Search

Recursive Implementation

- Function again takes parameters `data` and for the given list and the searched element
- Two parameters `left` and `right` define the current **search space**
- In a single call, `left` and `right` are not changed

⇨ **No** loop

- `current` is again computed as `(left + right) // 2`
- Again consider position `data[current]`

Recursive Binary Search

Recursive Implementation

- Function again takes parameters `data` and for the given list and the searched element
- Two parameters `left` and `right` define the current **search space**
- In a single call, `left` and `right` are not changed

⇨ **No** loop

- `current` is again computed as `(left + right) // 2`
- Again consider position `data[current]`
- If `searched` is not found, call the function recursively and either adjust `left` or `right` accordingly

Exercise – Recursive Binary Search

Implement binary search

- as a recursive Python function
- with four parameters
  `data`, `left`, `right`, and `searched`
- Follow the ideas of the iterative variant

```
def binsearch(data, searched):
    left = 0
    right = len(data) - 1
    while left <= right:
        current = (left + right) // 2
        if data[current] == searched:
            return current
        elif data[current] > searched:
            right = current - 1
        else:
            left = current + 1
    return -1
```

Recursive Binary Search

```python
def binsearch(data, left, right, searched):


    if left <= right:
        current = (left + right) // 2
        if data[current] == searched:
            return current
        elif data[current] > searched:
            return binsearch(data, left, current-1, searched)
        else:
            return binsearch(data, current+1, right, searched)
    else:
        return -1
```

Recursive Binary Search

```
def binsearch(data, left, right, searched):


    if left <= right:
        current = (left + right) // 2
        if data[current] == searched:
            return current
        elif data[current] > searched:
            return binsearch(data, left, current-1, searched)
        else:
            return binsearch(data, current+1, right, searched)
    else:
        return -1
```

```
def binsearch(data, searched):
    left = 0
    right = len(data) - 1
    while left <= right:
        current = (left + right) // 2
        if data[current] == searched:
            return current
        elif data[current] > searched:
            right = current - 1
        else:
            left = current + 1

    return -1
```

Recursive Binary Search

Call Stack

Recursive Binary Search

Call Stack

```
binsearch([2, 3, 5, 8, 10, 19, 21, 25, 28, 32, 36, 37, 42], 0, 12, 45)
```

Recursive Binary Search

Call Stack

`binsearch([2, 3, 5, 8, 10, 19, 21, 25, 28, 32, 36, 37, 42], 7, 12, 45)`

`current` $= 6$,
recursive call

`binsearch([2, 3, 5, 8, 10, 19, 21, 25, 28, 32, 36, 37, 42], 0, 12, 45)`

Recursive Binary Search

Call Stack

`binsearch([2, 3, 5, 8, 10, 19, 21, 25, 28, 32, 36, 37, 42], 10, 12, 45)`

current $= 9$,
recursive call

`binsearch([2, 3, 5, 8, 10, 19, 21, 25, 28, 32, 36, 37, 42], 7, 12, 45)`

current $= 6$,
recursive call

`binsearch([2, 3, 5, 8, 10, 19, 21, 25, 28, 32, 36, 37, 42], 0, 12, 45)`

Recursive Binary Search

Call Stack

`binsearch([2, 3, 5, 8, 10, 19, 21, 25, 28, 32, 36, 37, 42], 12, 12, 45)`

`current` $= 11$, recursive call

`binsearch([2, 3, 5, 8, 10, 19, 21, 25, 28, 32, 36, 37, 42], 10, 12, 45)`

`current` $= 9$, recursive call

`binsearch([2, 3, 5, 8, 10, 19, 21, 25, 28, 32, 36, 37, 42], 7, 12, 45)`

`current` $= 6$, recursive call

`binsearch([2, 3, 5, 8, 10, 19, 21, 25, 28, 32, 36, 37, 42], 0, 12, 45)`

Call Stack

```
binsearch([2, 3, 5, 8, 10, 19, 21, 25, 28, 32, 36, 37, 42], 13, 12, 45)
```

current $= 12$, recursive call

```
binsearch([2, 3, 5, 8, 10, 19, 21, 25, 28, 32, 36, 37, 42], 12, 12, 45)
```

current $= 11$, recursive call

```
binsearch([2, 3, 5, 8, 10, 19, 21, 25, 28, 32, 36, 37, 42], 10, 12, 45)
```

current $= 9$, recursive call

```
binsearch([2, 3, 5, 8, 10, 19, 21, 25, 28, 32, 36, 37, 42], 7, 12, 45)
```

current $= 6$, recursive call

```
binsearch([2, 3, 5, 8, 10, 19, 21, 25, 28, 32, 36, 37, 42], 0, 12, 45)
```

Recursive Binary Search

Call Stack

`binsearch([2, 3, 5, 8, 10, 19, 21, 25, 28, 32, 36, 37, 42], 13, 12, 45)` `return -1`

current $= 12$,
recursive call

`binsearch([2, 3, 5, 8, 10, 19, 21, 25, 28, 32, 36, 37, 42], 12, 12, 45)`

current $= 11$,
recursive call

`binsearch([2, 3, 5, 8, 10, 19, 21, 25, 28, 32, 36, 37, 42], 10, 12, 45)`

current $= 9$,
recursive call

`binsearch([2, 3, 5, 8, 10, 19, 21, 25, 28, 32, 36, 37, 42], 7, 12, 45)`

current $= 6$,
recursive call

`binsearch([2, 3, 5, 8, 10, 19, 21, 25, 28, 32, 36, 37, 42], 0, 12, 45)`

Recursive Binary Search

Call Stack

`binsearch([2, 3, 5, 8, 10, 19, 21, 25, 28, 32, 36, 37, 42], 12, 12, 45)` `return -1`

`current` $= 11$,
recursive call

`binsearch([2, 3, 5, 8, 10, 19, 21, 25, 28, 32, 36, 37, 42], 10, 12, 45)`

`current` $= 9$,
recursive call

`binsearch([2, 3, 5, 8, 10, 19, 21, 25, 28, 32, 36, 37, 42], 7, 12, 45)`

`current` $= 6$,
recursive call

`binsearch([2, 3, 5, 8, 10, 19, 21, 25, 28, 32, 36, 37, 42], 0, 12, 45)`

Recursive Binary Search

Call Stack

`binsearch([2, 3, 5, 8, 10, 19, 21, 25, 28, 32, 36, 37, 42], 10, 12, 45)` `return -1`

current $= 9$,
recursive call

`binsearch([2, 3, 5, 8, 10, 19, 21, 25, 28, 32, 36, 37, 42], 7, 12, 45)`

current $= 6$,
recursive call

`binsearch([2, 3, 5, 8, 10, 19, 21, 25, 28, 32, 36, 37, 42], 0, 12, 45)`

Recursive Binary Search

Call Stack

`binsearch([2, 3, 5, 8, 10, 19, 21, 25, 28, 32, 36, 37, 42], 7, 12, 45)` **return -1**

$\texttt{current} = 6,$
recursive call

`binsearch([2, 3, 5, 8, 10, 19, 21, 25, 28, 32, 36, 37, 42], 0, 12, 45)`

Recursive Binary Search

Call Stack

`binsearch([2, 3, 5, 8, 10, 19, 21, 25, 28, 32, 36, 37, 42], 0, 12, 45)` `return -1`

Thanks for your attention