



BABEȘ-BOLYAI UNIVERSITY

Faculty of Mathematics and Computer Science



Algorithms and Programming

Lecture 8 – Recursion. Computational complexity

Camelia Chira

Course content

Programming in the large

- Introduction in the software development process
- Procedural programming
- Modular programming
- Abstract data types
- Software development principles
- Testing and debugging

Programming in the small

- **Recursion**
- **Complexity of algorithms**
- Search and sorting algorithms
- Backtracking and other problem solving methods
- Recap

Last time

- Testing
 - Black box testing
 - White box testing
 - Examples
- Design patterns
 - GRASP
 - Information Expert
 - Pure Fabrication (Repository)
 - Controller

Today

- Recursion
 - Basic concept
 - Mechanism
 - Examples
- Computational complexity
 - Why?
 - Examples
 - Analyzing the efficiency of a program

Recursion

- What is recursion?
 - A way to solve a problem by reducing it to simpler versions of itself
 - A programming technique where a function calls itself
- Basic concepts
 - Recursive element – an element that is defined by itself
 - Recursive algorithm – an algorithm that calls itself
 - Note: condition to stop recursion
- Recursion can be:
 - **Direct** – a function calls itself (**f** calls **f**)
 - **Indirect** – a function **f** calls a function **g**, function **g** calls **f**

Example: factorial

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

$$n! = \begin{cases} 1 & \text{if } n = 0 \\ n * (n-1)! & \text{otherwise} \end{cases}$$

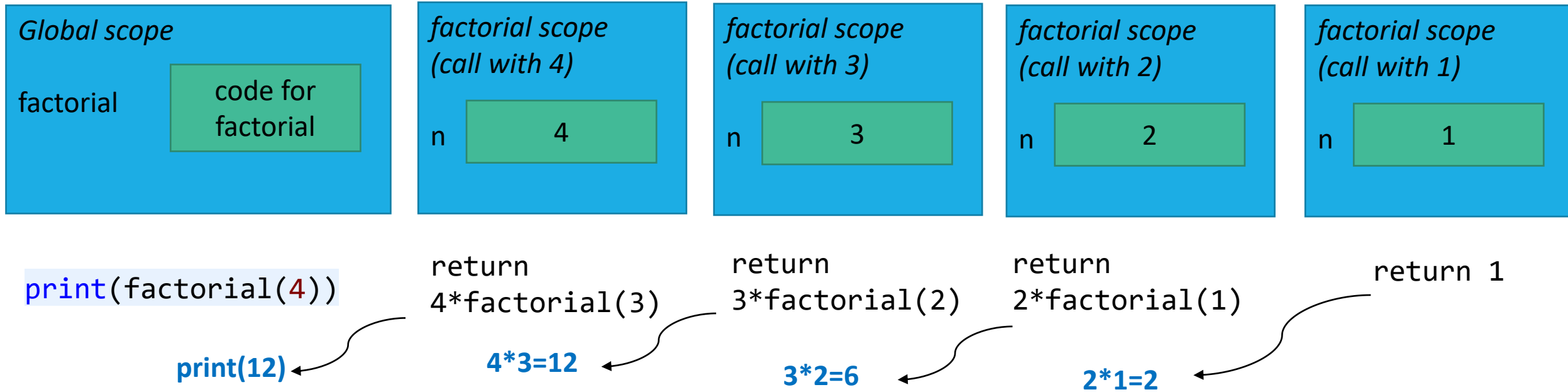
```
def factorial(n):  
    '''  
    Computes n! - the factorial for the given positive integer  
    input: n - positive integer  
    output: an integer 1 * 2 * 3 * ... * n  
    '''  
  
    # base case  
    if n == 1:  
        return 1  
    else:  
        # Recursive step: progresses toward the base case  
        return n * factorial(n - 1)
```

```
def testFactorial():  
    assert factorial(5) == 120  
    assert factorial(4) == 24  
    assert factorial(1) == 1  
    assert factorial(0) == 1  
  
testFactorial()
```

Example: factorial

```
def factorial(n):  
    if n == 1:  
        return 1  
    else:  
        return n * factorial(n - 1)  
  
print(factorial(4))
```

- Function scope



Recursion: mechanism

- Main idea of developing a recursive algorithm for a problem of size n
 - **Base case**
 - How to stop recursion
 - Identify the base case solution (for $n=1$)
 - **Inductive step**
 - Break the problem into a simpler version of the same problem plus some other steps
 - e.g. A smaller problem (of size $n-1$) and some simple computations
 - e.g. Two smaller problems (of size n_1, n_2 such that $n_1+n_2=n-1$) and simple computations
- How recursion works
 - On each method invocation a new symbol table is created: it contains all the parameters and the local variables defined in the function
 - The symbol tables are stored in a stack: when a function is returning, the current symbol table is removed from the stack

Inductive reasoning

- How do you know that your recursive algorithm works?
- Mathematical induction
 - To prove a statement is true for all values of n :
 - Prove that the statement is true for smallest value of n ($n=0$ or $n=1$)
 - Assume that the statement is true for n , then prove that it is also true for $n+1$
- Same logic applies here:
 - **Base case:** $n=1$, correctly returns 1
 - **Recursive case:** assume that the answer is correct for problem size smaller than k , then for $k+1$ the answer will be $(k+1)*k! = (k+1)!$
 - By induction, the result returned is correct

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

Example: sum and product

Note: Inductive step

```
def sumList(lst):  
    '''  
    Compute the sum of the elements in the list  
    input: lst - the list  
    output: The sum of the elements  
    '''  
  
    # base case  
    if len(lst) == 0:  
        return 0  
    else: # lst has at least one element  
        return lst[0] + sumList(lst[1:])  
  
def testSumList():  
    assert sumList([]) == 0  
    assert sumList([0]) == 0  
    assert sumList([1, 2, 6]) == 9  
    assert sumList([-1, 4, -100, 50]) == -47  
    assert sumList([1, 2, 3, 4, 5, 6]) == 21  
  
testSumList()
```

```
def product(lst):  
    '''  
    Computes the product of elements from a list  
    input: a list of integers  
    output: product of elements  
    '''  
  
    if (len(lst) == 0):  
        return 1  
    else:  
        middle = len(lst) // 2  
        return product(lst[0:middle]) * lst[middle] *  
            product(lst[middle+1:])  
  
def testProduct():  
    assert product([1,2,3,4]) == 24  
    assert product([]) == 1  
    assert product([1,2,3,4,5]) == 120  
    assert product([2]) == 2  
  
testProduct()
```

Example: palindrome

Note: symbol table

```
>>> isPalindrom("noon")
49018080    {'s': 'noon'}
49171216    {'s': 'oo'}
48909600    {'s': ''}
True

>>> isPalindrom("redivider")
49020384    {'s': 'redivider'}
49018080    {'s': 'edivide'}
49171216    {'s': 'divid'}
48909600    {'s': 'ivi'}
48973552    {'s': 'v'}
True
```

```
def isPalindrom(s):
    """
    Checks if a string is palindrom
    Input: a string
    Res: true, if str is palindrom and false, otherwise
    """
    dico = locals()
    print(id(dico), " ", dico)

    if len(s) <= 1:
        return True
    else:
        return s[0]==s[-1] and isPalindrom(s[1:-1])

def testIsPalindrom():
    assert isPalindrom("abcba") == True
    assert isPalindrom("abccba") == True
    assert isPalindrom("abcdba") == False

testIsPalindrom()
```

“Able was I, ere I saw Elba” (Napoleon)
“Are we not drawn onward, we few, drawn onward to new era?” (Anne Michaels)

Example: belongs

Note: symbol table

```
>>> belongs(2, [4, 2, 3, 5])
49171168 {'l': [4, 2, 3, 5], 'el': 2}
48909648 {'l': [2, 3, 5], 'el': 2}
True
```

```
>>> belongs(7, [4, 2, 3, 5])
48909600 {'l': [4, 2, 3, 5], 'el': 7}
49171168 {'l': [2, 3, 5], 'el': 7}
48909648 {'l': [3, 5], 'el': 7}
49018080 {'l': [5], 'el': 7}
49171600 {'l': [], 'el': 7}
False
```

```
def belongs(el, l):
    """
    Checks if an element belongs to a list
    Input: an integer and a list of integers
    Output: true, if el is in list
           and false, otherwise
    """
    dico = locals()
    print(id(dico), " ", dico)

    if (l == []):
        return False
    else:
        if (el == l[0]):
            return True
        else:
            return belongs(el, l[1:])

def testBelongs():
    assert belongs(5, []) == False
    assert belongs(5, [5,2,6,3]) == True
    assert belongs(5, [1,2,5,4,3]) == True
    assert belongs(5, [6,2,5]) == True
    assert belongs(5, [1,2,3]) == False

testBelongs()
```

Iteration vs. Recursion

```
def factorial_iter(n):  
    res = 1  
    for i in range(1, n+1):  
        res *= i  
    return res
```

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

✓ Recursive code is simple and intuitive

- However:
 - May be efficient from programming perspective, but
 - May not be efficient from the computer perspective
 - Why?
 - Large memory needed for in-depth recursion
 - Each function self call creates a new symbol table

Computational complexity

- What is it?
 - Study the efficiency of an algorithm from a mathematical perspective
- Why?
 - Problem: search for a number in a list
 - Solution:
 - Iterative search
 - Recursive search – one sub-problem
 - Recursive search – two sub-problems
 - **Which solution is better?**
 - **Algorithm efficiency**
 - Compare algorithms based on:
 - **Time needed for computations**
 - **Extra memory needed for temporary data**
 - Run time depends on:
 - Entry data (structure and size)
 - Changes from one run to another (due to hardware and software environment)
 - Hardware

Example: Fibonacci numbers

$$F_n = F_{n-1} + F_{n-2} \text{ and } F_0 = F_1 = 1$$

```
def fibonacci_recurziv(n):  
    '''  
    Computes the Fibonacci number  
    data: a positive integer  
    res: fibonacci number of n  
    '''  
    if n == 0 or n == 1:  
        return 1  
    return fibonacci_recurziv(n-1) +  
           fibonacci_recurziv(n-2)
```

```
def fibonacci_iterativ(n):  
    ''' ...  
    '''  
    s1 = 1  
    s2 = 1  
    fibo = 0  
    for i in range(2, n + 1):  
        fibo = s1 + s2  
        s1 = s2  
        s2 = fibo  
    return fibo
```

```
def timeFibonacci(n):  
    import time  
    start_time = time.time()  
    print("computing Fibonacci_iterativ(", n, ") = ",  
          fibonacci_iterativ(n))  
  
    end_time = time.time()  
    print(" takes ", end_time - start_time, " seconds")  
    start_time = time.time()  
    print("computing Fibonacci_recurziv(", n, ") = ",  
          fibonacci_recurziv(n))  
  
    end_time = time.time()  
    print(" takes ", end_time - start_time, " seconds")
```

```
>>> timeFibonacci(23)  
computing Fibonacci_iterativ( 23 ) = 46368  
takes 0.07200431823730469 seconds  
computing Fibonacci_recurziv( 23 ) = 46368  
takes 0.13400769233703613 seconds
```

```
>>> timeFibonacci(33)  
computing Fibonacci_iterativ( 33 ) = 5702887  
takes 0.07800436019897461 seconds  
computing Fibonacci_recurziv( 33 ) = 5702887  
takes 11.261644124984741 seconds
```

Algorithm efficiency

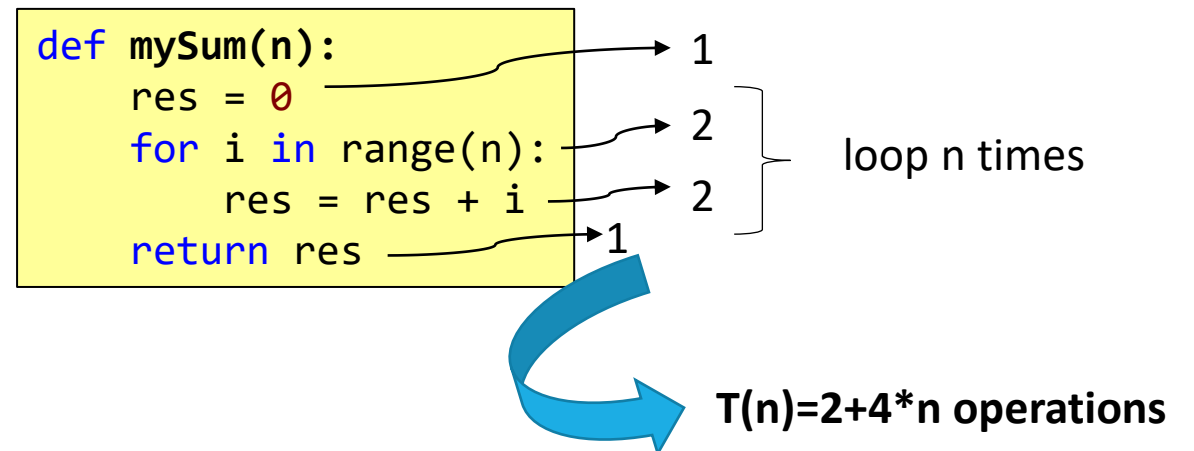
- To analyse the efficiency of an algorithm (function):
 - The amount of resources used
 - **Time** and space efficiency of a program
 - Measure efficiency
 - **Asymptotic analysis**
 - Can provide the efficiency for all possible input data
 - Can not provide exact execution times
 - **Empirical analysis**
 - Can not predict the performance of the algorithm for all possible input data
 - Can determine the execution time for specific set of inputs
 - Run time is studied in connection with the size of the input data

Run time complexity

- Complexity in time
- Running time of an algorithm:
 - It is not a fixed number
 - It is a function $T(n)$ that depends on the size n of the input data
 - Measures the basic steps the algorithm makes

- Example

- Steps that take constant time:
 - Mathematical operations
 - Assignments
 - Comparisons
 - Accessing objects in memory
 - Return statement
- Count the number of operations executed as a function of the input size



An example

```
def searchElement(l, e):  
    for i in l:  
        if e == i:  
            return True  
    return False
```

- e is the first element in the list – **best case**
 - Minimum running time over all possible inputs
- e is not in the list – **worst case**
 - Maximum running time over all possible inputs
- e is found after about half of the list is searched – **average case**
 - Average running time over all possible inputs

Run time complexity

- **Best case (BC)**

- For the entry data leading to minimum running time of the algorithm
- Algorithm complexity: $BC(A) = \min_{I \in D_A} E_A(I)$
- Gives a lower bound to the running time

- **Worst case (WC)**

- For the entry data leading to maximum running time of the algorithm
- Algorithm complexity: $WC(A) = \max_{I \in D_A} E_A(I)$
- Gives an upper bound to the running time

- **Average case (AC)**

- Average running time of the algorithm
- Algorithm complexity: $AC(A) = \sum_{I \in D_A} P_A(I) E_A(I)$
- Offers a prediction of the running time

where

A – algorithm

D_A – domain of the input data

I – an instance of input data

$E_A(I)$ – number of operations performed by algorithm A having input data I

$P_A(I)$ – probability that algorithm A receives input data I

Run time complexity: Examples

```
def sumOfFirstNumbers(n):  
    '''  
    computes the sum of first n natural numbers  
    data: a natural number  
    res: the sum of first n numbers  
    '''  
    s = 0  
    for i in range(1, n + 1):  
        s = s + i  
    return s
```

$$T(n) = \sum_{i=1}^n 1 = n$$

Case	T(n)
Best case	$\sum_{i=1}^n 1$
Worst case	$\sum_{i=1}^n 1$
Average case	$\sum_{i=1}^n 1$

Run time complexity: Examples

```
def search(el, list):
    '''
    checks if an element belongs to a list
    data: an integer and a list of integers
    res: true, if elements belongs to list
    false, otherwise
    '''
    for i in range(0, len(list)):
        if (list[i] == el):
            return True
    return False
```

Case	len(list)=n	T(n)
Best case	el=list[0]	1
Worst case	el=list[n-1] el is not in list	$\sum_{i=0}^{n-1} 1 = n$ $\sum_{i=0}^{n-1} 1 + 1 = n + 1$
Average case	el=list[0] el=list[1] el=list[2] ... el=list[n-1] el is not in list	1 2 3 ... n n+1 <hr/> $(1+2+3+\dots+n+(n+1))/(n+1)=(n+2)/2$

Big Oh notation

- $O(n)$ measure
 - How the running time grows depending on the input data size
 - Expression for the number of operations \rightarrow asymptotic behavior as the problem gets bigger

Exact steps vs
Big Oh or
 $O()$ notation

```
def factorial_iter(n):  
    i = 1  
    res = 1  
    while i <= n:  
        res = res * i  
        i = i + 1  
    return res
```

- Exact steps $T(n) = 1 + 1 + n * 5 + 1$
- Worst case asymptotic complexity $O(n)$
 - Ignore additive constants
 - Ignore multiplicative constants

$O()$

- Drop constants and multiplicative factors
- Focus on dominant terms (consider the leading term)

$O(n)$ $T(n) = 2n + 2$

$O(n^2)$ $T(n) = n^2 + 2n + 2$

$O(n^3)$ $T(n) = n^3 + 1000n + 3^{1000}$

$O(n)$ $T(n) = n + \log n$

$O(n \log n)$ $T(n) = n \log n + 100n$

$O(2^n)$ $T(n) = n^2 + 2^n$



$T(n) = n^3 + 100n^2 + 10n \log_2 n + 2\sqrt{n} + 27^{100}$

```
for i in range(n):  
    print(i)  
for i in range(n):  
    for j in range(n):  
        print(i, " ", j)
```

$O(n)$

$O(n) * O(n) = O(n * n) = O(n^2)$



$O(n) + O(n^2) = O(n + n^2) = O(n^2)$

Run time complexity: theoretical aspects

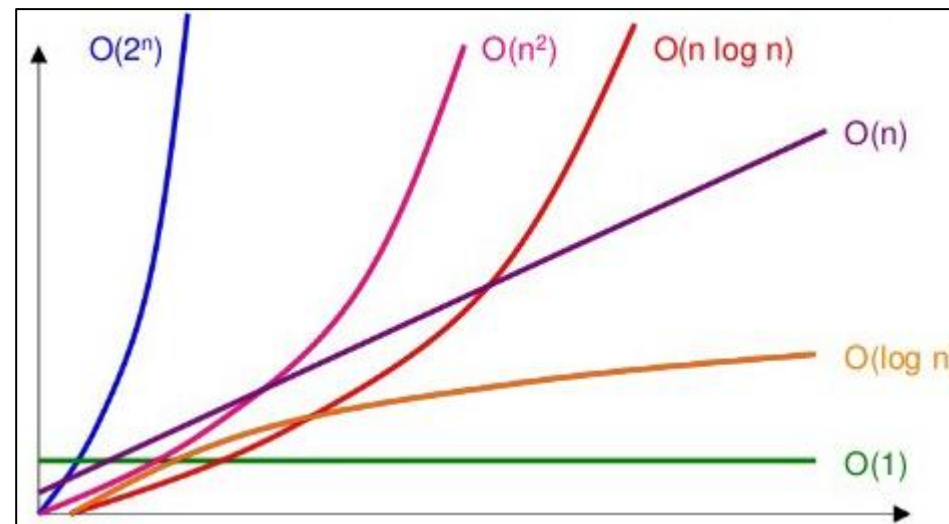
- For a function $f: N \rightarrow R$ and $T: N \rightarrow N$, how to decide the complexity class of T ?
- $T(n) \in O(f(n))$ if there exist 2 positive and independent constants c and n_0 such that $0 \leq T(n) \leq c * f(n), \forall n \geq n_0$
- Examples
 - $O(1) = 1, 5, 500$
 - $O(n) = n, 2n + 1, 5n - 200$
 - $O(n^2) = n^2, n^2 + 5, 2n^2 + 3n - 1$
- If $T(n) \in O(f(n))$ then $\lim_{n \rightarrow \infty} \frac{T(n)}{f(n)} < \infty$ (the limit is constant)

Complexity classes

$O(1)$	Constant running time	e.g. 1, 47, 100	Add an element to a list
$O(\log n)$	Logarithmic running time	e.g. $10 + \log n$	Find an element in a sorted list
$O(n)$	Linear running time	e.g. n , $3n$, $10n+100$	Find an entry in an unsorted list
$O(n \log n)$	Log-linear running time	e.g. $n + n \log n$	Sort a list (MergeSort, QuickSort)
$O(n^c)$, c is constant	Polynomial running time	e.g. n^2+1 , n^3+n^2+5n	Shortest path between two nodes
$O(c^n)$, c is constant	Exponential running time	e.g. 2^n+1 , 3^n	Traveling Salesman Problem (TSP)

$O(n^2)$ - quadratic time

$O(n^3)$ - cubic time



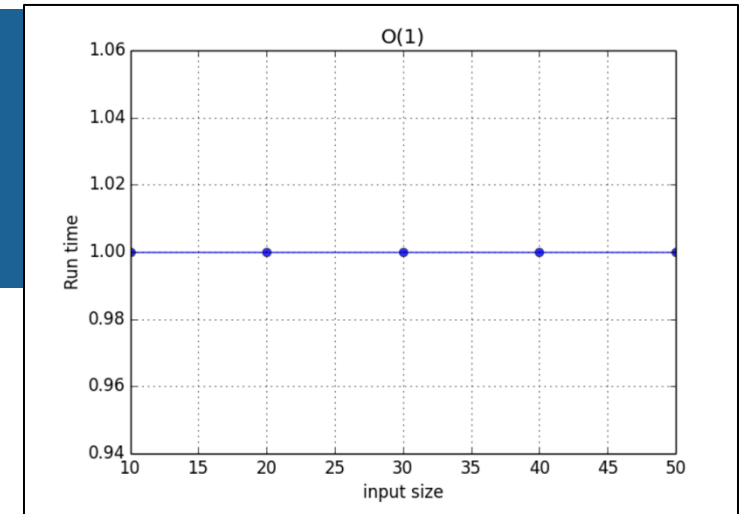
Complexity classes: Constant

- $T(n) \in O(1)$
 - **Constant running time**
- Very good complexity (the algorithm executes a constant number of steps regardless the size of input data)
- Example: add an element to a list, access an element from a list, modify information in an object

```
def isFirstElementNone(l):  
    return l[0] == None
```

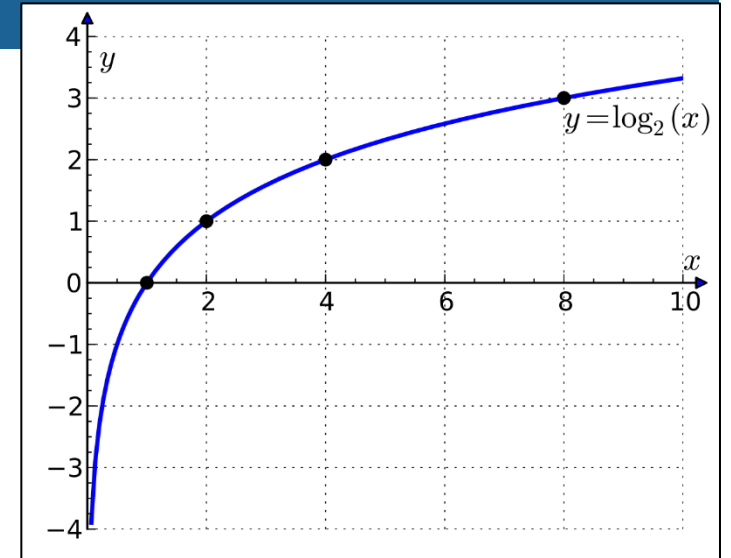
	n=1	n=10	n=100	n=1000
O(1)	1	1	1	1

If n doubles,
O(1) remains
unchanged



Complexity classes: Logarithmic

- $T(n) \in O(\log n)$
 - **Logarithmic running time**
 - Very good complexity
- Example: binary search, the height of a binary tree



- *Q: How many times to divide a problem of size n until arriving to a problem of size 1?*
- *$n = 2^x$, $x = ?$*
- *See the binary search algorithm in the next lecture*

	n=1	n=10	n=100	n=1000
$O(\log n)$	0	1	2	3

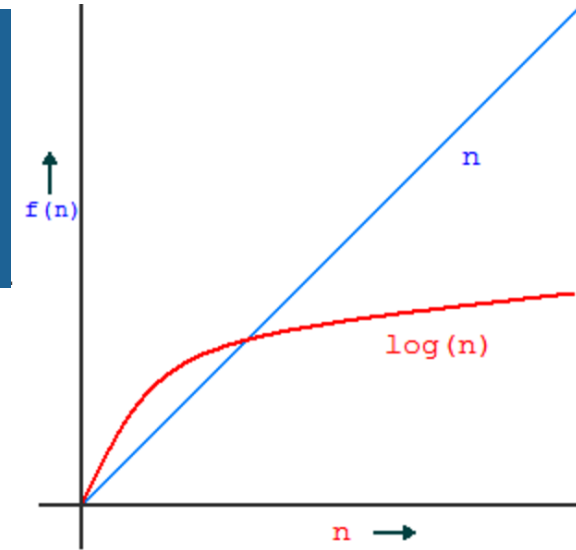
If n doubles,
 $O(\log n)$ increases
slightly

Complexity classes: Linear

- $T(n) \in O(n)$

- Linear running time
- Good complexity
- Performance grows linearly and in direct proportion to the size of the input data set

- Example: find the minimum / maximum in an unsorted list, find an element in a list



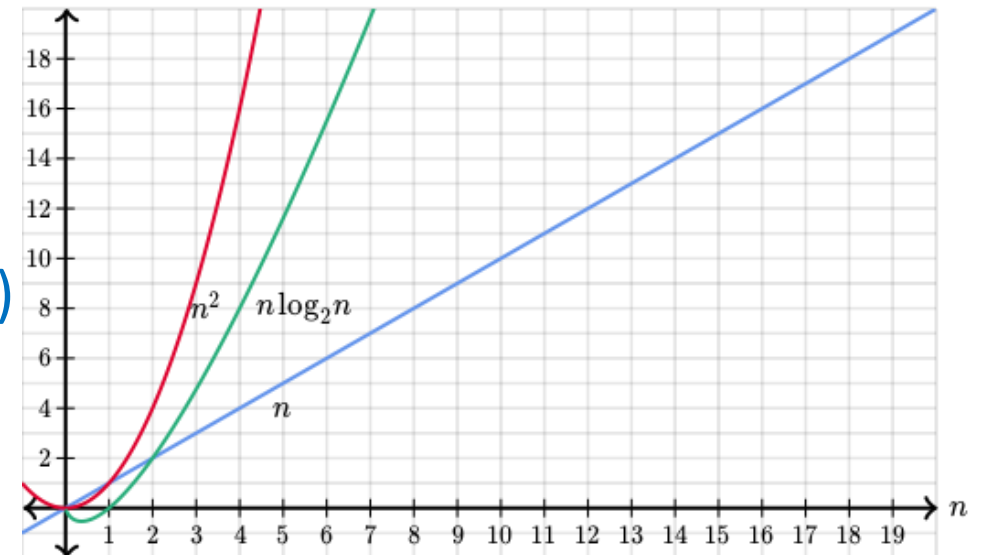
```
def containsValue(l, val):  
    for i in range(0, len(l)):  
        if (l[i] == val):  
            return True  
    return False
```

	n=1	n=10	n=100	n=1000
O(n)	1	10	100	1000

If n doubles,
O(n) doubles

Complexity classes: Log-linear

- $T(n) \in O(n \log n)$
 - Log-linear running time
 - Good complexity
 - Example: sort a list (MergeSort / QuickSort)
 - See the algorithms in the next lecture



	n=1	n=10	n=100	n=1000
$O(n \log n)$	0	10	200	3000

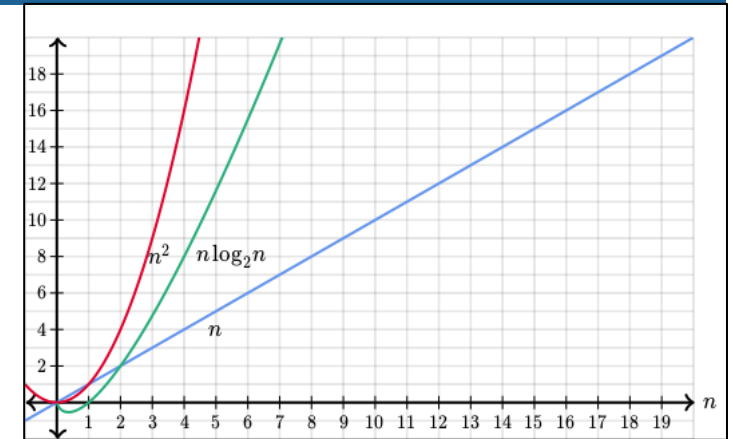
If n doubles,
 $O(n \log n)$ slightly
more than doubles

Complexity classes: Quadratic

- $T(n) \in O(n^2)$
 - Quadratic running time
 - Good complexity if n is multiple of 1000
 - Bad complexity if n is multiple of 1 mil.
 - **Example: sort a list (BubbleSort)**
 - *See the algorithms in the next lecture*
 - Common when the algorithm involves nested iterations

Example

```
def containsDuplicates(l):  
    for i in range(len(l)):  
        for j in range(len(l)):  
            if i != j and l[i]==l[j]:  
                return True  
    return False
```



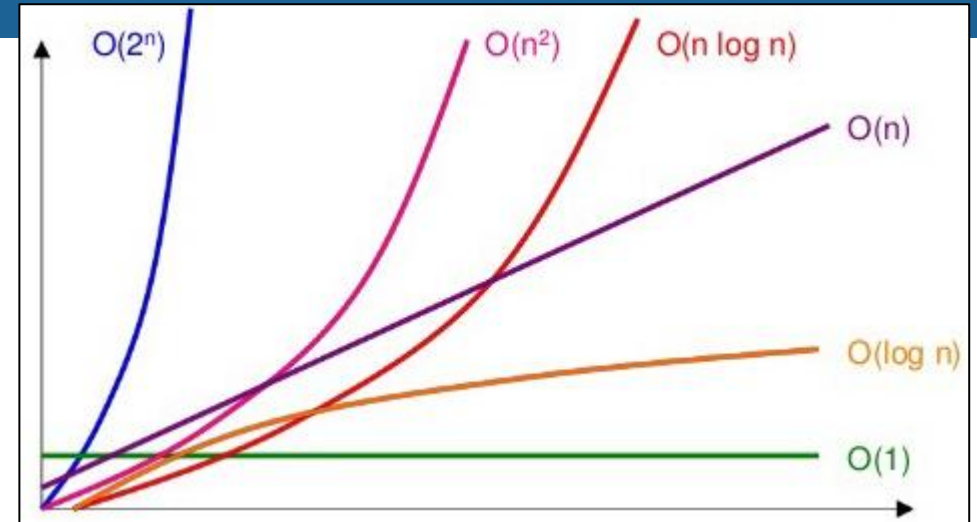
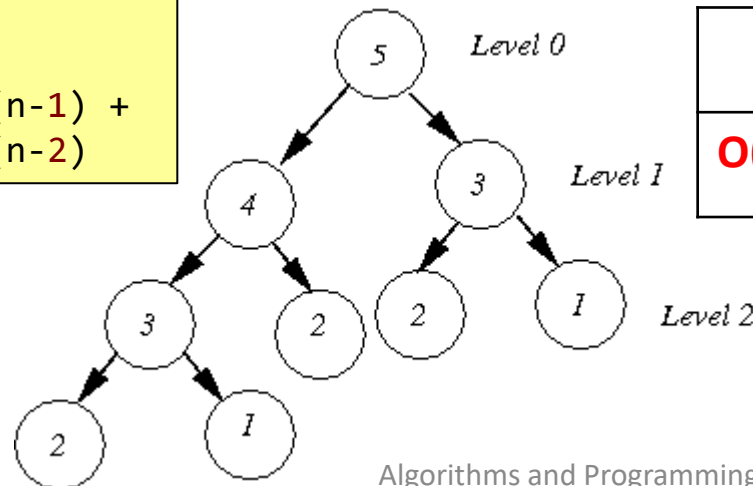
	n=1	n=10	n=100	n=1000
$O(n^2)$	1	100	10000	1 mil.

If n doubles,
 $O(n^2)$ quadruples

Complexity classes: Exponential

- $T(n) \in O(2^n)$
 - **Exponential running time**
 - Bad complexity
 - Example: TSP, Fibonacci recursively

```
def fibonacci_recursiv(n):  
    if n == 0 or n == 1:  
        return 1  
    return fibonacci_recursiv(n-1) +  
           fibonacci_recursiv(n-2)
```



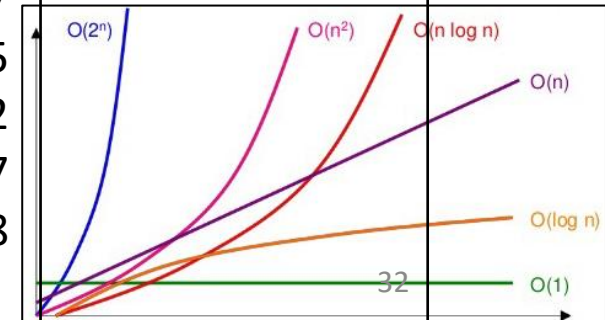
	n=1	n=10	n=100	n=1000
$O(2^n)$	2	2^{10}	2^{100}	2^{1000}

If n doubles,
 $O(2^n)$ multiplies drastically

$2^{100} \approx 10^{30}$
HUGE!!

Complexity growth

Order of growth	n=10	n=100	n=1000	n=1000000
$O(1)$	1	1	1	1
$O(\log n)$	1	2	3	6
$O(n)$	10	100	1000	1000000
$O(n \log n)$	10	200	3000	6000000
$O(n^2)$	100	10000	1000000	1000000000000
$O(2^n)$	1024	1.267.650.600.228. 229.401.496.703.2 05.376	1071508607186267320948425049060001810 5614048117055336074437503883703510511 2493612249319837881569585812759467291 7553146825187145285692314043598457757 4698574803934567774824230985421074605 0623711418779541821530464749835819412 6739876755916554394607706291457119647 7686542167660429831652624386837205668 069376	??



Run time complexity analysis: example

```
def sumOfFirstNumbers(n):  
    '''  
    computes the sum of first n natural numbers  
    data: a natural number  
    res: the sum of first n numbers  
    '''  
    sum = 0  
    for i in range(1, n + 1):  
        sum = sum + i  
    return sum  
  
def test_sum():  
    assert sumOfFirstNumbers(5) == 15  
    assert sumOfFirstNumbers(1) == 1
```

Case	T(n)
Best case	$\sum_{i=1}^n 1 = n$
Worst case	$\sum_{i=1}^n 1 = n$
Average case	$\sum_{i=1}^n 1 = n$

 **$O(n)$**

Run time complexity analysis: example

```
def sumOfFirstNumbers(n):  
    '''  
    computes the sum of first n natural numbers  
    data: a natural number  
    res: the sum of first n numbers  
    '''  
    sum = 0  
    i = 1  
    while (i<=n):  
        sum = sum + i  
        i = i + 1  
    return sum  
  
def test_sum():  
    assert sumOfFirstNumbers(5) == 15  
    assert sumOfFirstNumbers(1) == 1
```

Case	T(n)
Best case	$\sum_{i=1}^n 1 = n$
Worst case	$\sum_{i=1}^n 1 = n$
Average case	$\sum_{i=1}^n 1 = n$

 **$O(n)$**

Run time complexity analysis: example

```
def searchEven(list):  
    '''  
    checks if a list contains at least an even number  
    data: a list of integers  
    res: true, if list contains at least an even  
    number false, otherwise  
    '''  
    i = 0  
    while ((i < len(list)) and (list[i] % 2 != 0)):  
        i = i + 1  
    return (i < len(list))  
  
def test_searchEven():  
    assert searchEven([2,4,6]) == True  
    assert searchEven([1,3,5]) == False  
    assert searchEven([1,2,3]) == True
```

Case	T(n)
Best case	$1 + 1 = 2$
Worst case	$1 + \sum_{i=1}^n 1 + 1 = n + 2$
Average case	$\sum_{i=1}^n i / n = (n + 1) / 2$

 $O(n)$

Run time complexity analysis: example

```
def sumOfElemFromMatrix(m):  
    s = 0  
    for i in range(0, len(m)):  
        for j in range(0, len(m[i])):  
            s = s + m[i][j]  
    return s  
  
def test_sumOfElemFromMatrix():  
    assert sumOfElemFromMatrix([[1,2],[4,5],[7,9]]) == 28  
    assert sumOfElemFromMatrix([[1,2,3],[4,5,6],[7,8,9]]) == 45
```

Case	T(n)
Best case	$\sum_{i=1}^n \sum_{j=1}^m 1 = n * m$
Worst case	$\sum_{i=1}^n \sum_{j=1}^m 1 = n * m$
Average case	$\sum_{i=1}^n \sum_{j=1}^m 1 = n * m$

 $O(n^2)$

Run time complexity analysis: example

```
class Book:
    def __init__(self, t, na, a):
        self.title = t
        self.noAuthors = na
        self.authors = a

    def getAuthors(self):
        return self.authors

def searchBooksOfAnAuthor(books, author):
    res = []
    for b in books:
        authors = b.getAuthors()
        i = 0
        while (i < len(authors)):
            if (authors[i] == author):
                res.append(b)
                i = len(authors)
            else:
                i = i + 1
    return res
```

```
def test_searchBooksOfAnAuthor():
    b1 = Book("title1", 2, ["author1", "author2"])
    b2 = Book("title2", 3, ["author2", "author3", "author4"])
    b3 = Book("title3", 1, ["author4"])
    books = [b1, b2, b3]
    assert searchBooksOfAnAuthor(books, "a") == []
    assert searchBooksOfAnAuthor(books, "author5") == []
    assert searchBooksOfAnAuthor(books, "author1") == [b1]
    assert searchBooksOfAnAuthor(books, "author2") == [b1, b2]
    assert searchBooksOfAnAuthor(books, "author4") == [b2, b3]
```

Case	T(n)
Best case	$\sum_{i=1}^n 1 = n$
Worst case	$\sum_{i=1}^n \sum_{j=1}^m 1 = n * m$
Average case	$\sum_{i=1}^n m = m * n$

$O(n^2)$

n - number of books
m - average number of authors for a book

Run time complexity analysis: example

```
def sum(l):  
    '''  
    computes the sum of elements from a list  
    data: a list of integers  
    res: sum of elements  
    '''  
    if (len(l) == 0):  
        return -1  
    else: #len(l) > 0  
        return l[0] + sum(l[1:])  
  
def testSum():  
    assert sum([1,2,3,4]) == 10  
    assert sum([]) == 0  
    assert sum([3]) == 3
```

$$T(n) = \begin{cases} 1, & \text{if } n = 0 \\ T(n-1) + 1, & \text{otherwise} \end{cases}$$

$$T(n) = T(n-1) + 1$$

$$T(n-1) = T(n-2) + 1$$

$$T(n-2) = T(n-3) + 1$$

...

$$T(1) = T(0) + 1$$

$$T(n) = n + 1$$

Complexity in space

- Estimates the **space** (memory) that an algorithm needs to store input data, output data and any temporary data
- Uses the same notations of run time complexity

```
def minArray_v1():  
    a = []  
    n = int(input("n = "))  
    for i in range(0, n):  
        el = int(input("el = "))  
        a.append(el)  
  
    minim = a[0]  
    for i in range(1, n):  
        if (a[i] < minim):  
            minim = a[i]  
    print("minim is " + str(minim))
```

```
def minArray_v2():  
    n = int(input("n = "))  
    minim = int(input("el = "))  
    for i in range(1, n):  
        el = int(input("el = "))  
        if (el < minim):  
            minim = el  
    print("minim is " + str(minim))
```

$$S(n)=1+1+1=3$$

$O(1)$

$$S(n)=1+n+1=n+2$$

$O(n)$

Recap today

- Recursion
 - Basic concept
 - Mechanism
 - Examples
- Computational complexity
 - Examples
 - The efficiency of a program
 - Time and space complexity

Next time

- Algorithms
 - Sort
 - Search

Reading materials and useful links

1. The Python Programming Language - <https://www.python.org/>
2. The Python Standard Library - <https://docs.python.org/3/library/index.html>
3. The Python Tutorial - <https://docs.python.org/3/tutorial/>
4. M. Frentiu, H.F. Pop, Fundamentals of Programming, Cluj University Press, 2006.
5. MIT OpenCourseWare, Introduction to Computer Science and Programming in Python, <https://ocw.mit.edu>, 2016.
6. K. Beck, Test Driven Development: By Example. Addison-Wesley Longman, 2002. http://en.wikipedia.org/wiki/Test-driven_development
7. M. Fowler, Refactoring. Improving the Design of Existing Code, Addison-Wesley, 1999. <http://refactoring.com/catalog/index.html>