BABEŞ-BOLYAI UNIVERSITY
Faculty of Mathematics and Computer Science

Algorithms and Programming

Lecture 9 – Computational complexity, Search and sorting algorithms

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

- Recursion
    - Basic concept
    - Mechanism
    - Recursive functions

- Computational complexity
    - Analyzing the efficiency of a program
    - Run time complexity
    - Classes of complexity

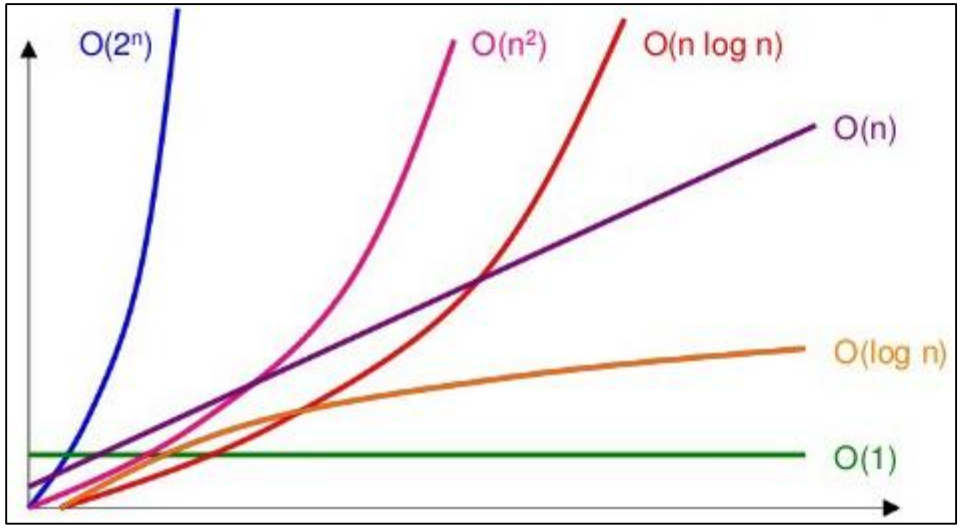
Today

- Search
  - Objective and problem specification
  - Types
    - Sequential seach
    - Binary search
- Sort
  - Objective and problem specification
  - Types
    - Selection sort
    - Insert sort
    - Bubble sort
    - Quick sort

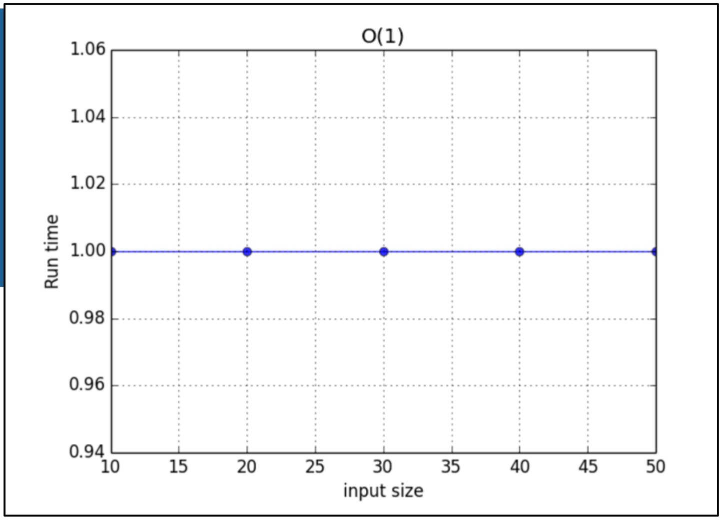
Complexity classes

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

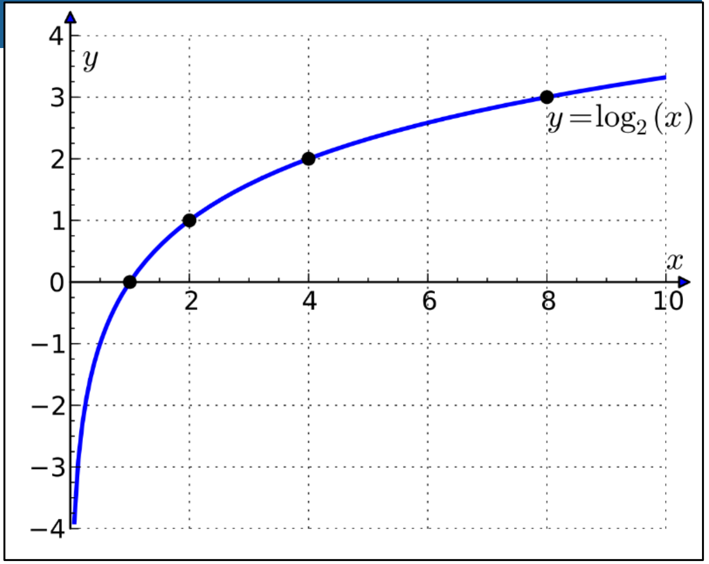
```python
def isFirstElementNone(l):
    return l[0] == None
```

	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

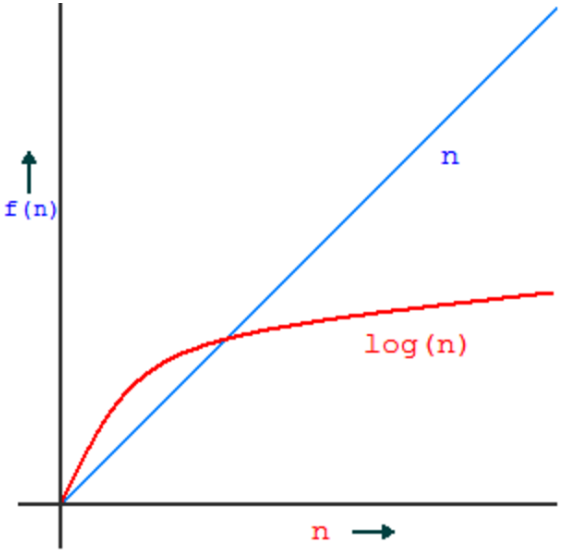


- *Q: How many times to divide a problem of size n until arriving to a problem of size 1?*
- *$n = 2^x$, $x = ?$*
- *See the binary search algorithm in the next lecture*

	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

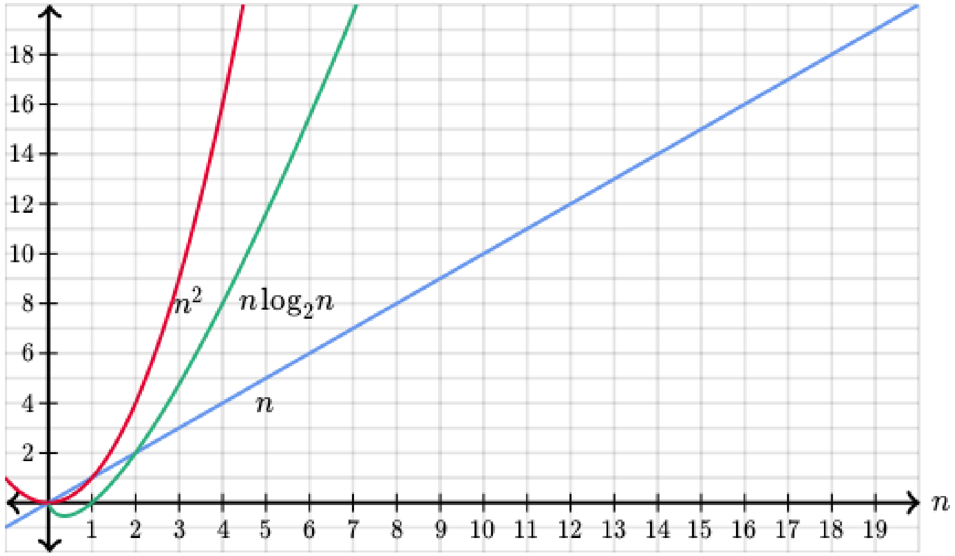
```
def containsValue(l, val):
    for i in range(0, len(l)):
        if (l[i] == val):
            return True
    return False
```

	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*

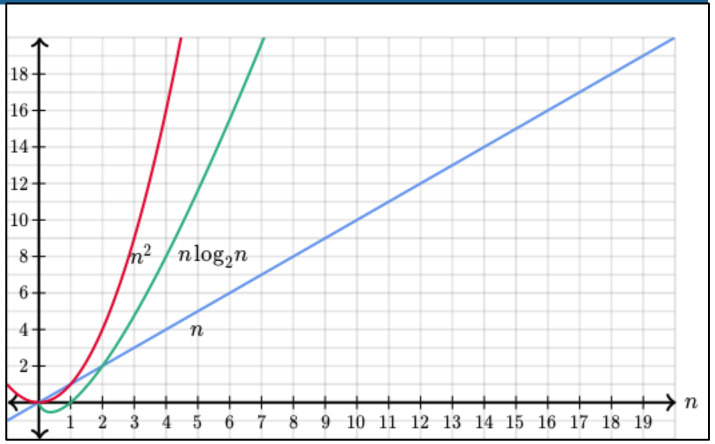


	n=1	n=10	n=100	n=1000
O(n log n)	0	10	200	3000

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

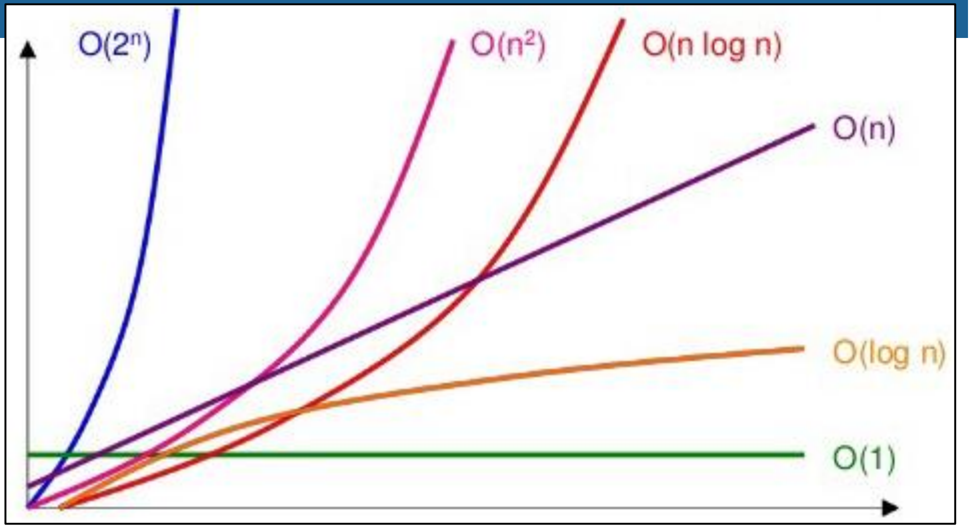
```
def containsDuplicates(l):
    for i in range(len(l)):
        for j in range(len(l)):
            if i != j and l[i]==l[j]:
                    return True
    return False
```

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

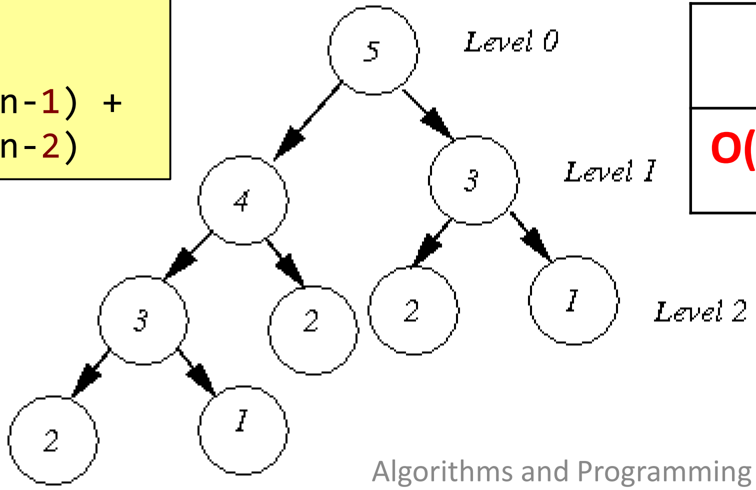
If n doubles,
$O(n^2)$ quadruples

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)
```



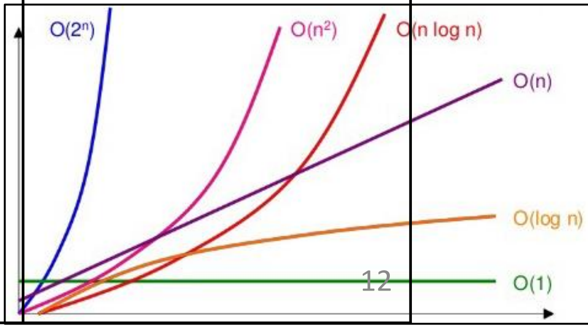
	n=1	n=10	n=100	n=1000
$O(2^n)$	2	2^{10}	2^{100}	2^{1000}

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	1.267.650.600.228.229.401.496.703.205.376	10715086071862673209484250490600018105614048117055336074437503883703510511249361224931983788156958581275946729175531468251871452856923140435984577574698574803934567774824230985421074605062371141877954182153046474983581941267398767559165543946077062914571196477686542167660429831652624386837205668069376	??



Run time complexity analysis: example

```python
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	$\sum_{i=1}^{n} 1 = \mathrm{n}$
Worst case	$\sum_{i=1}^{n} 1 = \mathrm{n}$
Average case	$\sum_{i=1}^{n} 1 = \mathrm{n}$

→ ***O(n)***

Run time complexity analysis: example

```python
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	$\sum_{i=1}^{n} 1 = \mathrm{n}$
Worst case	$\sum_{i=1}^{n} 1 = \mathrm{n}$
Average case	$\sum_{i=1}^{n} 1 = \mathrm{n}$

$\Rightarrow$ ***O(n)***

Run time complexity analysis: example

```python
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	$\sum_{i=1}^{n}\sum_{j=1}^{m} 1 = n * m$

$\Rightarrow$ ***O(n²)***

Run time complexity analysis: example

```python
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
```

```python
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

```python
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
```

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

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

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

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

...

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

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

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_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)

```
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))
```

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

O(n)

Algorithms:
Search and Sort

Search methods: Objective

- For a set of data stored in memory as a list of elements (el1, el2, ..., eln)
  - The list may contain elements in any order
  - The list contains elements ordered by some criteria
- Look for
  - A certain element
  - Elements that satisfy different criteria
- Return
  - True or False – if the element(s) exist in the list
  - The index of the element found

Search methods: problem specification

- Unorderd list of elements
  - Input data:
    - elem, n, list = ($list_i$) i=0,1, 2,...,n-1 (n – natural number)
  - Results:
    - p, where $0 \leq p \leq n-1$, if elem = list[p] or -1, if elem is not in the list
- Ordered list of elements
  - Input data:
    - elem, n, list = ($list_i$), list[0]<list[1]<...<list[n-1], i=0,1, 2,...,n-1 (n – natural number)
  - Results:
    - p, where $0 \leq p \leq n-1$, if elem = list[p] or -1, if elem is not in the list

Search methods: implementation

- **Sequential search**
  - Basic idea: the elements of the list are examined one by one (the list can be ordered or not)
  - Versions: simple and improved
- **Binary search**
  - Basic idea: the problem is divided in two similar but smaller subproblems (the list has to be ordered)

- Python
  - Functions index and find

Sequential search: implementation
Unordered list

```python
def searchSeq(el, l):
    '''
    Descr: search for an element in a list
    Data: an element and a list
    Res: the position of element in list or -1 if the elemnt is not in the list
    '''
    pos = -1
    for i in range(0, len(l)):
        if (el == l[i]):
            pos = i
    return pos

def test_searchSeq():
    assert searchSeq(2, [3,2,4]) == 1
    assert searchSeq(2, [3,5,7,2]) == 3
    assert searchSeq(2, [2,5,4]) == 0
    assert searchSeq(2, [3,7,4]) == -1
    assert searchSeq(2, [3,2,4,2,7]) == 3

test_searchSeq()
```

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

O(n)

Sequential search: implementation
Unordered list – Improved version

```
def searchSeq_v2(el, l):
    '''
    Descr: search for an element in a list
    Data: an element and a list
    Res: the position of element in list or
    -1 if the elemnt is not in the list
    '''
    i = 0
    while ((i < len(l)) and (l[i] != el)):
        i = i + 1
    if (i < len(l)):
        return i
    else:
        return -1

def test_searchSeq_v2():
    assert searchSeq_v2(2, [3,2,4]) == 1
    assert searchSeq_v2(2, [3,5,7,2]) == 3
    assert searchSeq_v2(2, [2,5,4]) == 0
    assert searchSeq_v2(2, [3,7,4]) == -1
    assert searchSeq_v2(2, [3,2,4,2,7]) == 1
test_searchSeq_v2()
```

Case	**T(n)**
Best case	1
Worst case	$\sum_{i=1}^{n} 1 = \mathrm{n}$
Average case	(0+1+2+...+n-1)/n

Sequential search: implementation *Ordered list*

```
def searchSeqOrder(el, l):
    '''
    Descr: search for an element in a list
    Data: an element and a list of ordered elements
    Res: the position of element in list or the position where the element can be inserted
    '''
    if (len(l) == 0): #l==[]
        return 0
    pos = -1
    for i in range(len(l) - 1, -1, -1):
        if (el <= l[i]):
            pos = i
    if (pos == -1):
        return len(l)
    return pos

def test_searchSeqOrder():
    assert searchSeqOrder(2, [2,3,4]) == 0
    assert searchSeqOrder(4, [2,3,4,5]) == 2
    assert searchSeqOrder(2, [1,3,5,7]) == 1
    assert searchSeqOrder(9, [1,2,3]) == 3

test_searchSeqOrder()
```

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

Sequential search: implementation
Ordered list – Improved version

```
def searchSeqOrder_v2(el, l):
    '''
    Descr: search for an element in a list
    Data: an element and a list of ordered elements
    Res: the position of element in list or
    the position where the element can be inserted
    '''
    if (len(l) == 0): #l==[]
        return 0
    if (el <= l[0]):
        return 0
    if (el > l[len(l)-1]):
        return len(l)
    i = 0
    while ((i < len(l)) and (l[i] < el)):
        i = i + 1
    return i
def test_searchSeqOrder_v2():
    assert searchSeqOrder_v2(2, [2,3,4]) == 0
    assert searchSeqOrder_v2(4, [2,3,4,5]) == 2
    assert searchSeqOrder_v2(2, [1,3,5,7]) == 1
    assert searchSeqOrder_v2(9, [1,2,3]) == 3
test_searchSeqOrder_v2()
```

Case	**T(n)**
Best case	1
Worst case	$\sum_{i=1}^{n} 1 = n$
Average case	(0+1+2+…+n-1)/n

Binary search: implementation
Ordered list – recursive version

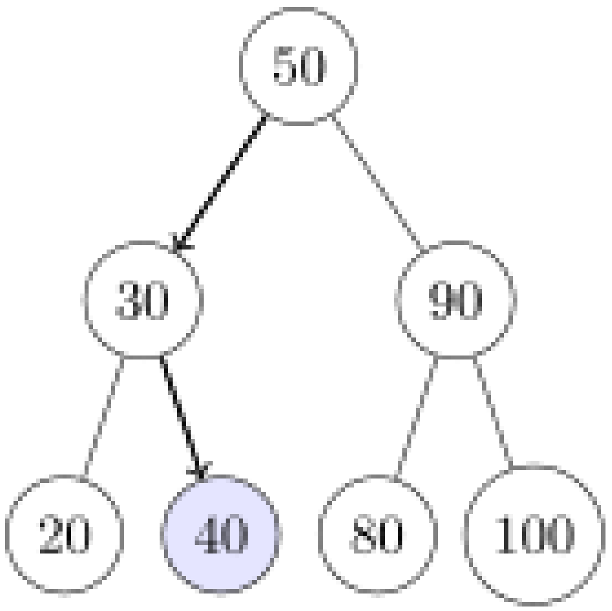
```
def binarySearch(el, l, start, end):
    if (start > end):
        return -1
    middle = (start + end) // 2
    if (el < l[middle]):
        return binarySearch(el, l, start, middle)
    elif (el > l[middle]):
        return binarySearch(el, l, middle + 1, end)
    else: #el == l[middle]
        return middle

def binarySearchRec(el, l):
    #Descr: search for an element in a list
    #Data: an element and a list
    #Res: the position of element in list or
    # -1 if the element is not in the list
    if (len(l) == 0):
        return -1
    elif (el < l[0]) or (el > l[len(l) -1]):
        return -1
    else:
        return binarySearch(el, l, 0, len(l)-1)
```

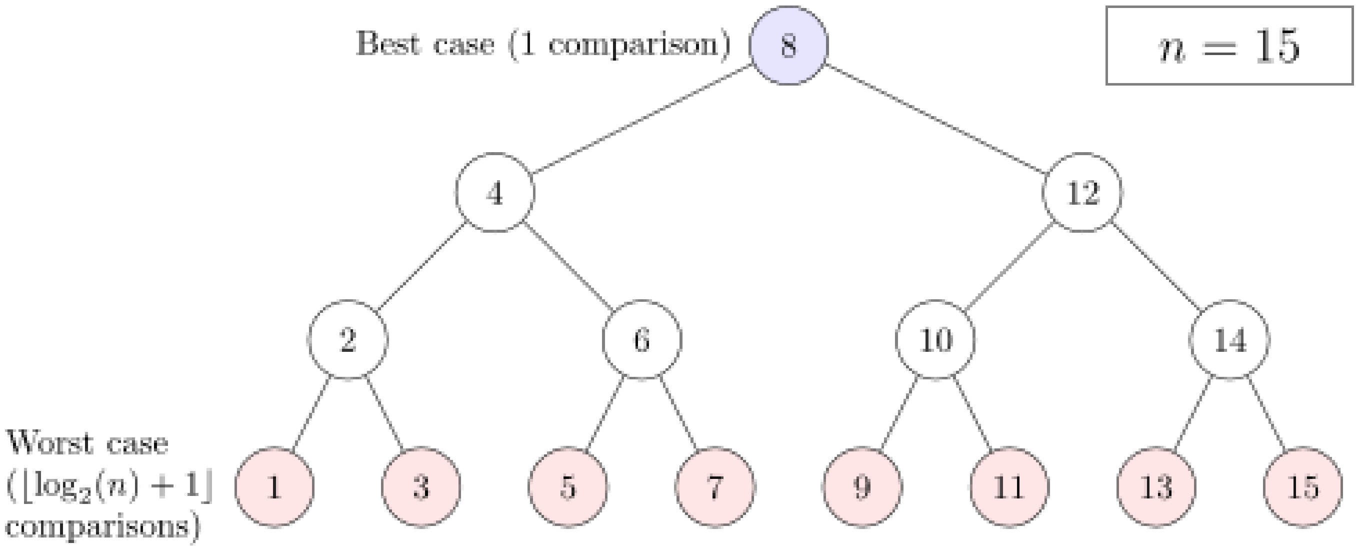
```
def test_binarySearchRec():
    assert binarySearchRec(4, [3,4,5]) == 1
    assert binarySearchRec(2, [-3,0,1,2]) == 3
    assert binarySearchRec(2, [2,5,6]) == 0
    assert binarySearchRec(2, [3,7,9]) == -1
    assert binarySearchRec(2, [1,2,2,5,6]) == 2

test_binarySearchRec()
```

Case	T(n)
Best case	1
Worst case	$\log_2 n$
Average case	$\log_2 n$

Binary search: example

- List is [20, 30, 40, 50, 80, 90, 100]
- Search for element 40





https://en.wikipedia.org/wiki/Binary_search_algorithm

Binary search: implementation
Ordered list – iterative version

```python
def binarySearchIter(el, l):
    '''
    Descr: search for an element in a list
    Data: an element and a list
    Res: the position of element in list or
    -1 if the elemnt is not in the list
    '''
    if (len(l) == 0):
        return -1
    elif (el < l[0]) or (el > l[len(l) -1]):
        return -1
    else:
        start = 0
        end = len(l) - 1
        while (start <= end):
            middle = (start + end) // 2
            if (el < l[middle]):
                end = middle
            elif (el > l[middle]):
                start = middle + 1
            else:
                return middle
```

```python
def test_binarySearchIter():
    assert binarySearchIter(4, [3,4,5]) == 1
    assert binarySearchIter(2, [-3,0,1,2]) == 3
    assert binarySearchIter(2, [2,5,6]) == 0
    assert binarySearchIter(2, [3,7,9]) == -1
    assert binarySearchIter(2, [1,2,2,5,6]) == 2
test_binarySearchIter()
```

$$T(n) = \begin{cases} 1, if\ n = 1 \\ T(n/2) + 1, otherwise \end{cases}$$

$$if\ n = 2^k \rightarrow T(2^k) = T(2^{k-1}) + 1$$

$$T(2^k) = T(2^{k-1}) + 1$$

$$T(2^{k-1}) = T(2^{k-2}) + 1$$

$$\ldots$$

$$T(2^1) = T(2^0) + 1$$

$$T(2^k) = k + 1 \qquad k = \log_2 n \rightarrow T(n) = \log_2 n + 1$$

Search methods: Python functions

- list.index(element)
  - Returns the index of the element in the list
  - If the element does not exist in the list, throws an exception
- list.count(element)
  - Returns the number of times the element appears in the list (if it exists)
  - Returns 0 if the element is not in the list

```python
def test_index():
    l = [7,2,13,4,1]
    assert l.index(2) == 1
    assert l.index(1) == 4
    try:
        l.index(3)
        assert False
    except ValueError as ex:
        print("elem not found")
        assert True

test_index()
```

```python
def test_count():
    l = [7,2,13,4,1]
    assert l.count(2) == 1
    assert l.count(1) == 1
    assert l.count(3) == 0

test_count()
```

Sorting methods

- Objective
  - Rearrange the elements of a container such that they are in a certain relation of order
- Problem specification
  - Input data:
    - n, list = ($list_i$) i=0,1, 2,...,n-1 (n – natural number)
  - Results:
    - n, list’ = ($list_i$’) i=0,1, 2,...,n-1, orderRelation($list_i$’, $list_{i+1}$’)=True for any i=0,1,...,n-2

Sorting methods: taxonomy

- Place where the elements are stored
  - **Internal sort** – data to be sorted is available in the internal memory
  - **External sort** – data available from files (external memory)
- The order relation
  - **Ascending sort**
  - **Descending sort**
- Keeping the initial order of the elements
  - **Stable sort –** keep the initial order of equal elements
  - **Instable sort** – the initial order of equal elements is not kept
- Space complexity
  - **In-place sort** – the additional space (to that needed for the container) is small
  - **Not-in-place / Out-of-place sort** – large additional space
- Mechanism
  - **Selection sort**
  - **Insert sort**
  - **Bubble sort**
  - **Quick sort**

Selection Sort

- Basic idea
  - Determine the smallest element from the collection and place it in first position (swap the smallest element with the first one)
  - Repeat the first step for all elements different from the smallest element
- Algorithm
- Complexity
  - Time

    $T(n) = \sum_{i=0}^{n-2}\left(\sum_{j=i}^{n-1} 1 + 3\right) \approx n(n-1)/2$ -> ***O(n²)***
  - Space

    S(n)=n+1+1

```python
def selectionSort(l):
    '''
    descr: sorts the leemnts of a list
    data: a list of elements
    res: the ordered list
    '''
    for i in range(0, len(l) - 1):
        min_pos = i
        for j in range(i + 1, len(l)):
            if (l[j] < l[min_pos]):
                min_pos = j
        if (i < min_pos):
            aux = l[i]
            l[i] = l[min_pos]
            l[min_pos] = aux
    return l
```

```python
l[i], l[min_pos] = l[min_pos], l[i]
```

```python
def test_selectionSort():
    assert selectionSort([1,2,3]) == [1,2,3]
    assert selectionSort([3,2,1]) == [1,2,3]
    assert selectionSort([1,2,1]) == [1,1,2]

test_selectionSort()
```

Insertion Sort

- Basic idea
  - Traverse the elements of the container and insert each element at the correct position in the sub-container with the elements already sorted
  - At the end of the algorithm, the sub-container will have all the initial elements sorted
- Algorithm
- Complexity
  - Time

    $T(n) = \sum_{i=1}^{n-1}\left(1 + \sum_{j=0}^{i-1} 2 + 1\right) \approx n^2 + n - 2$ -> ***O(n²)***
  - Space

    S(n)=n+1+1+1

```python
def insertSort(l):
    '''
    descr: sorts the elemnts of a list
    data: a list of elements
    res: the ordered list
    '''
    for i in range(1, len(l)):
        noOfAlreadySort = i - 1
        crtElem = l[i]
        # insert crtElem in the right position
        # (<= noOfAlreadySort)
        j = noOfAlreadySort
        while ((j >= 0) and (crtElem < l[j])):
            l[j + 1] = l[j]
            j = j - 1
        l[j + 1] = crtElem
    return l

def test_insertSort():
    assert insertSort([1,2,3]) == [1,2,3]
    assert insertSort([3,2,1]) == [1,2,3]
    assert insertSort([1,2,1]) == [1,1,2]

test_insertSort()
```

Bubble Sort

- Basic idea
  - Compare any 2 consecutive elements
    - If they are not in correct order, swap them
  - Until any 2 consecutive elements are in the correct order
- Algorithm
- Complexity
  - Time

    $T(n)$ -> ***O(n²)***
  - Space

    S(n)=n+1+1+1

```python
def bubbleSort(l):
    '''
    descr: sorts the elemnts of a list
    data: a list of elements
    res: the ordered list
    '''
    isSort = False
    while (not isSort):
        isSort = True
        for i in range(0, len(l) - 1):
            if (l[i] > l[i + 1]):
                aux = l[i]
                l[i] = l[i + 1]
                l[i + 1] = aux
                isSort = False
    return l

def test_bubbleSort():
    assert bubbleSort([1,2,3]) == [1,2,3]
    assert bubbleSort([3,2,1]) == [1,2,3]
    assert bubbleSort([1,2,1]) == [1,1,2]

test_bubbleSort()
```

Quick Sort

- Basic idea
  - Divide and conquer technique
  1. **Divide:** divide the container in 2 parts such that any element in the first sub-container ≤ any element in the second sub-container
  2. **Conquer:** sort the two sub-containers (recursively)
- Algorithm

```python
def partition(l, start, end):
    pivot = l[start]
    i = start
    j = end
    while (i != j):
        while ((pivot <= l[j]) and (i < j)):
            j = j - 1
        l[i] = l[j]
        while ((l[i] <= pivot) and (i < j)):
            i = i + 1
        l[j] = l[i]
        l[i] = pivot
    return i

def quickSortRec(l, start, end):
    pivotPos = partition(l, start, end)
    if (start < pivotPos - 1):
        quickSortRec(l, start, pivotPos - 1)
    if (pivotPos + 1 < end):
        quickSortRec(l, pivotPos + 1, end)

def quickSort(l):
    '''
    descr: sorts the elemnts of a list
    data: a list of elements
    res: the ordered list
    '''
    quickSortRec(l, 0, len(l) - 1)
    return l
```

```python
def test_quickSort():
    assert quickSort([1,2,3]) == [1,2,3]
    assert quickSort([3,2,1]) == [1,2,3]
    assert quickSort([1,2,1]) == [1,1,2]

test_quickSort()
```

Quick Sort: complexity

- The run time of quick-sort depends on the distribution of splits
  - The partitioning function requires linear time
  - Best case is when the partitioning function splits the array evenly

Best case	$T(n)=2*T(n/2)+n \rightarrow O(n \log_2 n)$
Worst case	T(n)=T(1)+T(n-1)+n=T(n-1)+n+1 T(n-1)=T(1)+T(n-2)+(n-1)=T(n-2)+n ... T(2)=T(1)+T(1)+2=4 _______________ T(n)=(n+1)(n+2)/2-(1+2+3)=>$O(n^2)$
Average case	L(n)=2*U(n/2)+n U(n)=L(n-1)+n L(n)=2*(L(n/2-1)+n/2)+n=2L(n/2-1_+2n=>$O(n \log_2 n)$

- Space complexity
  - Average: S(n) = $\log_2 n$
  - Worst: S(n) = n

Sorting in Python

- `list.sort()`
- `sorted(lista)`

```
>>> t = (5, 1, 17, 12)
>>> sorted(t)
[1, 5, 12, 17]
>>> t
(5, 1, 17, 12)
```

```
>>> a = [2, 1, 5, 7, 9]
>>> a
[2, 1, 5, 7, 9]
>>> sorted(a)
[1, 2, 5, 7, 9]
>>> a
[2, 1, 5, 7, 9]
>>> sorted(a, reverse=True)
[9, 7, 5, 2, 1]
>>> a
[2, 1, 5, 7, 9]
>>> a.sort()
>>> a
[1, 2, 5, 7, 9]
>>> a.sort(reverse=True)
>>> a
[9, 7, 5, 2, 1]
```

Sorting in Python: list of lists / tuples

- Use the key argument in sorted/sort

```
>>> my_list = [[2,1,3], [1,5,7], [7,2,1]]
>>> def getKey(item):
        return item[1]

>>> sorted(my_list, key=getKey)
[[2, 1, 3], [7, 2, 1], [1, 5, 7]]
>>> my_list
[[2, 1, 3], [1, 5, 7], [7, 2, 1]]
>>> sorted([(1,1,1), (15,0,16), (25,5,0)], key=getKey)
[(15, 0, 16), (1, 1, 1), (25, 5, 0)]
```

key argument &
lambda expressions

```
>>> sorted(my_list, key=lambda x: x[1])
[[2, 1, 3], [7, 2, 1], [1, 5, 7]]
>>> sorted(my_list, key=lambda x: x[2])
[[7, 2, 1], [2, 1, 3], [1, 5, 7]]
>>> sorted(my_list, key=lambda x: x[0])
[[1, 5, 7], [2, 1, 3], [7, 2, 1]]
```

Sorting in Python: list of custom objects

```python
class Student:
    def __init__(self, name, grade):
        self.__name = name
        self.__grade = grade

    def getName(self):
        return self.__name

    def getGrade(self):
        return self.__grade
```

```
>>> st_list = [Student("Sara", 8), Student("Erin", 10), Student("Emma", 9)]
>>> def getKey(s):
        return s.getGrade()

>>> sorted(st_list, key=getKey)
[<__main__.Student object at 0x02E5BFD0>, <__main__.Student object at 0x02E73090
>, <__main__.Student object at 0x02E5B3B0>]
>>> st_list
[<__main__.Student object at 0x02E5BFD0>, <__main__.Student object at 0x02E5B3B0
>, <__main__.Student object at 0x02E73090>]
```

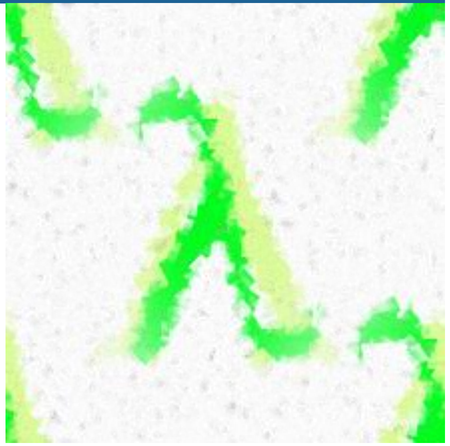
Sorting in Python: list of custom objects

```
class Student:
    def __init__(self, name, grade):
        self.__name = name
        self.__grade = grade

    def getName(self):
        return self.__name

    def getGrade(self):
        return self.__grade

    def __repr__(self):
        return self.__name + " - " + str(self.__grade)
```

```
>>> sorted(st_list, key=lambda x: x.getGrade())
[Sara - 8, Emma - 9, Erin - 10]
>>> sorted(st_list, key=lambda x: x.getName())
[Emma - 9, Erin - 10, Sara - 8]
>>>
>>> sorted(st_list, key=lambda Student: Student.getName())
[Emma - 9, Erin - 10, Sara - 8]
```

```
>>> st_list = [Student("Sara", 8), Student("Erin", 10), Student("Emma", 9)]
>>> def getKey(s):
        return s.getGrade()

>>> sorted(st_list, key=getKey)
[Sara - 8, Emma - 9, Erin - 10]
>>> st_list
[Sara - 8, Erin - 10, Emma - 9]
```

Lambda expressions



- Small anonymous functions
- Defined and used in the same place

✓ *Syntactically restricted to a single expression*

✓ *Can reference variables from the containing scope (just like nested functions)*

✓ *They are syntactic sugar for a function definition*

Lambda expressions

- Syntax

```
lambda arg1, arg2, ...argN : expression using arguments
```

- Lambda is an expression

(def - a function with name, statement)

- Body – is simply an expression

(not a block of stataments,

no return statement)

```
>>> def f(x):
        return x**3

>>> f(2)
8
>>> g = lambda x: x**3
>>> g(2)
8
>>> (lambda x: x**3) (2)
8
```

Map and Lambda expressions

- Function map
  - r=map(function, sequence)

```
>>> m = map(f, [1,2,3])
>>> list(m)
[1, 8, 27]
>>> list(map(lambda x: x**3, [1,2,3]))
[1, 8, 27]
>>> list(map(lambda x,y: x**y, [1,2,3], [2, 3, 4]))
[1, 8, 81]
```

Filter and lambda expressions

- Function filter
  - `filter(function, sequence)`

```
>>> my_list = [1, 4, 5, 8, 9, 10]
>>> filter(lambda x: x%2 == 0, my_list)
<filter object at 0x02E4CFB0>
>>> list(filter(lambda x: x%2 == 0, my_list))
[4, 8, 10]
>>>
>>> fibonacci = [0,1,1,2,3,5,8,13,21,34,55]
>>> odd_numbers = list(filter(lambda x: x % 2, fibonacci))
>>> odd_numbers
[1, 1, 3, 5, 13, 21, 55]
>>>
>>> names = ["Zara", "Erin", "Carla", "Ana", "Nico"]
>>> filtered_names = list(filter(lambda x: x[-1] == "a", names))
>>> filtered_names
['Zara', 'Carla', 'Ana']
```

Sort and Lambda expressions

```
class Person:
    def __init__(self, n, a):
        self.name = n
        self.age = a

    def getName(self):
        return self.name

    def getAge(self):
        return self.age

    def __repr__(self):
        return self.name + "-" + str(self.age)
```

```
def sort_python():
    l1 = [4,2,3,1]
    l1.sort()
    print(l1)

    l1s = sorted(l1)
    print(l1s)

    p1 = Person("nnnn", 20)
    p2 = Person("eeee", 21)
    p3 = Person("ttt", 10)
    l2 = [p1, p2, p3]
    l2s = sorted(l2, key=lambda Person:Person.getName())
    print(l2s)
    l2s = sorted(l2, key=lambda Person:Person.getAge())
    print(l2s)

sort_python()
```

Recap today

- Search
  - Sequential seach
  - Binary search

- Sort
  - Selection sort
  - Insert sort
  - Bubble sort
  - Quick sort

Next time

- Algorithms
  - Backtracking
  - Divide and conquer

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

Bibliography

The content of this course has been prepared using the reading materials from previous slide, different sources from the Internet as well as lectures on Fundamentals of Programming held in previous years by:

- Prof. Dr. Laura Dioşan - www.cs.ubbcluj.ro/~lauras
- Conf. Dr. Istvan Czibula - www.cs.ubbcluj.ro/~istvanc
- Lect. Dr. Andreea Vescan - www.cs.ubbcluj.ro/~avescan
- Lect. Dr. Arthur Molnar - www.cs.ubbcluj.ro/~arthur