Introduction to computer programming with python

Complexity Part 1

Spring Semester, 2023

Contents

Complexity:

- Motivation
- O notation
- Types of order of growth
- Law of addition, law of multiplication
- Complexity growth (via search algorithms)

Complexity - Motivation

Example – constant functions

```python
def c_to_f(c):
    return (c*9/5) + 32 # 3 operations
```

```python
t0 = t.time()
c_to_f(10)
dt = t.time()-t0
print(dt)
```

C=10 or c=1000000 run @ constant time & same # of operations

Complexity - Motivation

Example – iterative function

```python
def ave(l):
    res = 0 # 1 op
    for i in l: # n ops
        res+=i # 2*n ops
    return res/len(l) # 1 op
```

```python
t0 = t.time()
ave(testList)
dt = t.time()-t0
```

Input size len(l)	Measured time dt	Operations 2+3*n
7	0 s	23
7,000,000	0.35 s	21,000,002

n = len(l)

Timing programs isn't reliable indication of program's efficiency

- Runtime varies between algorithms ✓
- Runtime varies between implementations, e.g. ✗

```
if k in dict: print(d[k])
```

```
d.get('k')
```

- Runtime varies between computers ✗
- program's runtime is not predictable based on small inputs' sampling ✗

Input size len(l)	Measured time dt
7	0 s
7,000,000	0.35 s

Exact # of operations versus 'O'

Example: factorial calculation, n>=0

```
def fact(n):
    res = 1 # 1 Op
    for i in range(n,1,-1): # n-1 Ops
        res*=i # 2*(n-1) ops
    return res
```

Exact # of operations: 1+3*(n-1)

'O' represents the ***worst case asymptotic complexity***

- Ignore additive constants
- Ignore multiplicative constants

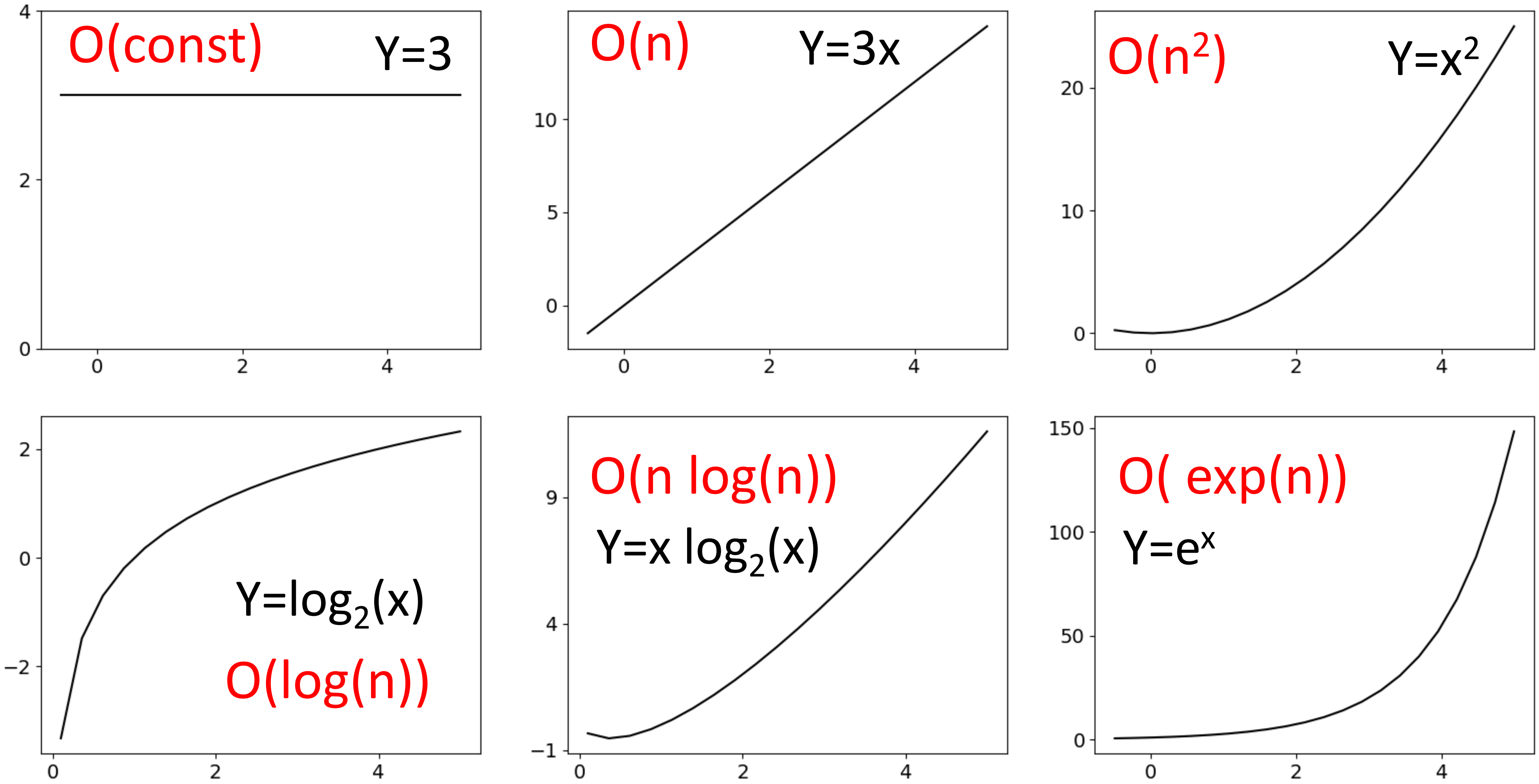
O(1+3*(n-1)) ➔ O(n)

Exact # of operations versus 'O'

Numeric examples:

- $n^2 + 2n + 2 \longrightarrow O(n^2)$
- $n^2 + 100000n + 3^{1000} \longrightarrow O(n^2)$
- $\log(n) + n + 4 \longrightarrow O(n)$
- $0.0001*n*\log(n) + 300n \longrightarrow O(n\cdot\log(n))$
- $2n^{30} + 3^n \longrightarrow O(3^n)$

Types of orders of growth



Orders of growth law of addition

$$O(f(n)) + O(g(n)) = O(f(n)+g(n))$$

Example:

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

O's addition

O's approximation by dominant term

Orders of growth
law of addition

$$O(f(n)) + O(g(n)) = O(f(n)+g(n))$$

Example:

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

O's addition

Ignore multiplicative constants

Orders of growth
law of multiplication

$O(f(n)) * O(g(n)) = O(f(n)*g(n))$

Example:

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

See nested loops

law of multiplication
$O(n^x)$ - nested loops

```
def isSubset(L1, L2):
    for e1 in L1:
        matched = False
        for e2 in L2:
            if e1 == e2:
                matched = True
                break
        if not matched:
            return False
    return True
```

len(L1) → O(n)

len(L2) → O(n)

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

Complexity growth

notation	complexity	n=10	n=100	n=1000	n=1000000
constant	O(1)	1	1	1	1
logarithmic	O(log(n))	1	2	3	6
linear	O(n)	10	100	1000	1000000
log-linear	O(n log n)	10	200	3000	6000000
n^c - Polynomial n^2 - quadratic	$O(n^2)$	100	10000	1000000	10^{12}
exponential	$O(2^n)$	1024	12676506 00228229 40149670 3205376	1071508607186267320948425049060001810561404811705533607443750388370351051124936122493198378815695858127594672917553146825187145285692314043598457757469857480393456777482423098542107460506237114187795418215304647498358194126739876755916554394607706291457119647768654216766042983165262438683720566806937 6	TLDR (אמל"ק)

Searching for an element in a list
e.g., linear search

```python
def linear_search(L, e):
    found = False # 1 op
    for i in range(len(L)): # n ops
        if e == L[i]: # n ops
            found = True # 1 ops
    return found
```

Orders of growth estimation

O(2+n+n) = O(2+2*n) => O(n)

By dominant term

Searching for an element in a sorted list
Still a linear search

```python
def search(L, e):
    for i in range(len(L)):
        if L[i] == e:
            return True
        if L[i] > e:
            return False
    return False
```

len(L) — O(n)

> len(L) — O(n)

Orders of growth estimation

O(n+(<n)) => O(n)

By dominant term

Bisection (binary) search
recursive version

```
def binary_search(arr, low, high, x):
    # Check base case
    if high >= low:

        mid = (high + low) // 2

        # If element is present at the middle itself
        if arr[mid] == x:
            return mid

        # If element is smaller than mid, then it can only
        # be present in left sub-array
        elif arr[mid] > x:
            return binary_search(arr, low, mid - 1, x)

        # Else the element can only be present in right subarray
        else:
            return binary_search(arr, mid + 1, high, x)

    else:
        # Element is not present in the array
        return -1
```

Returns the index of x in sorted array if exists otherwise returns -1

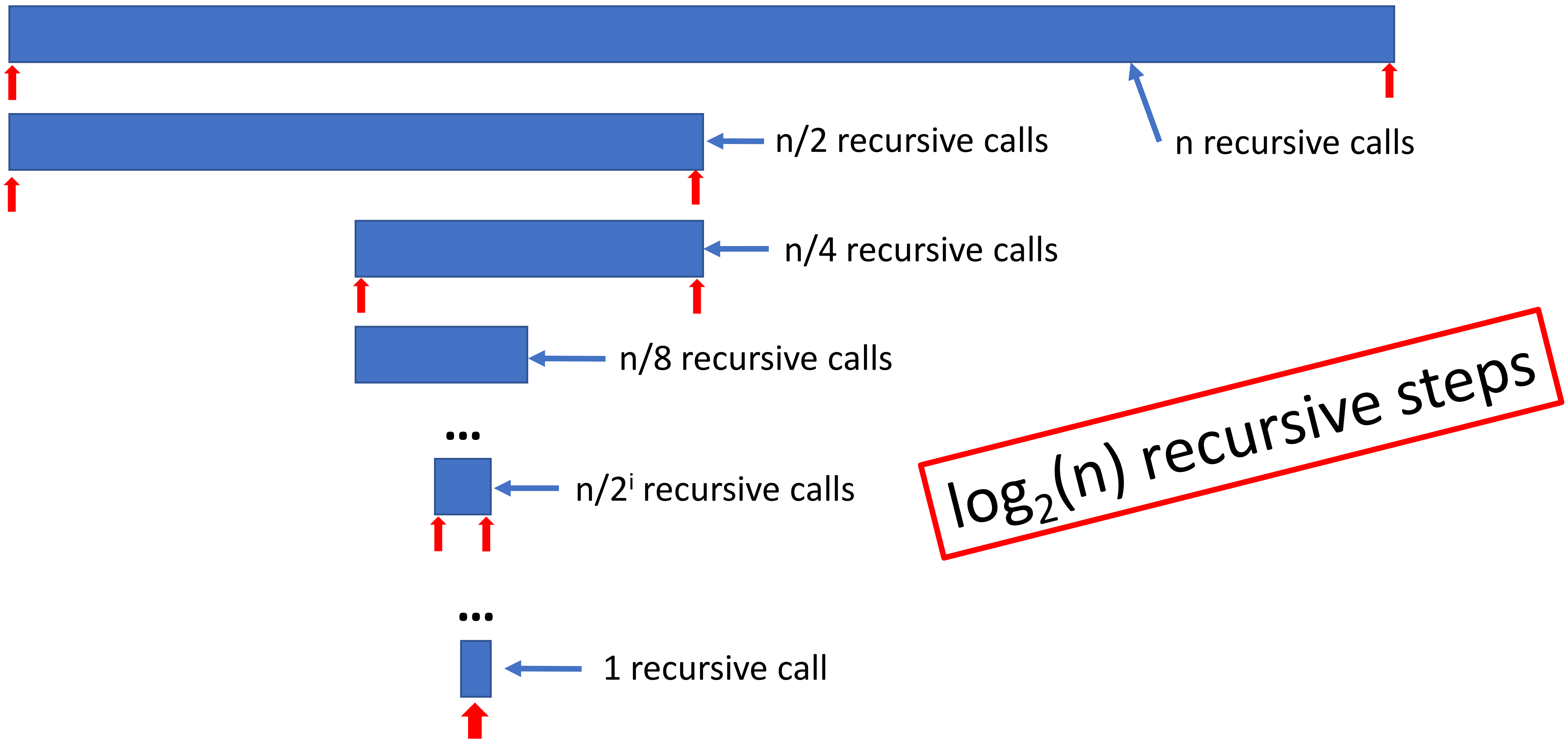
Bisection (binary) search

Iterative version

```
def binary_search(arr, x):
    low = 0
    high = len(arr) - 1
    mid = 0

    while low <= high:

        mid = (high + low) // 2

        # If x is greater, ignore left half
        if arr[mid] < x:
            low = mid + 1

        # If x is smaller, ignore right half
        elif arr[mid] > x:
            high = mid - 1

        # means x is present at mid
        else:
            return mid

    # If we reach here, then the element was not present
    return -1
```

Returns the index of x in sorted array if exists otherwise returns -1

Bisection (binary) search
(log) Complexity analysis



Complexity of iterative Fibonacci

```
def itFib(n):
    if n==0:
        return 0
    elif n==1:
        return 1
    else:
        fi = 0
        fii = 1
        for i in range(n-1):
            fi, fii = fii, fi+fii
        return fii
```

constant
O(1)

constant
O(1)

linear
O(n)

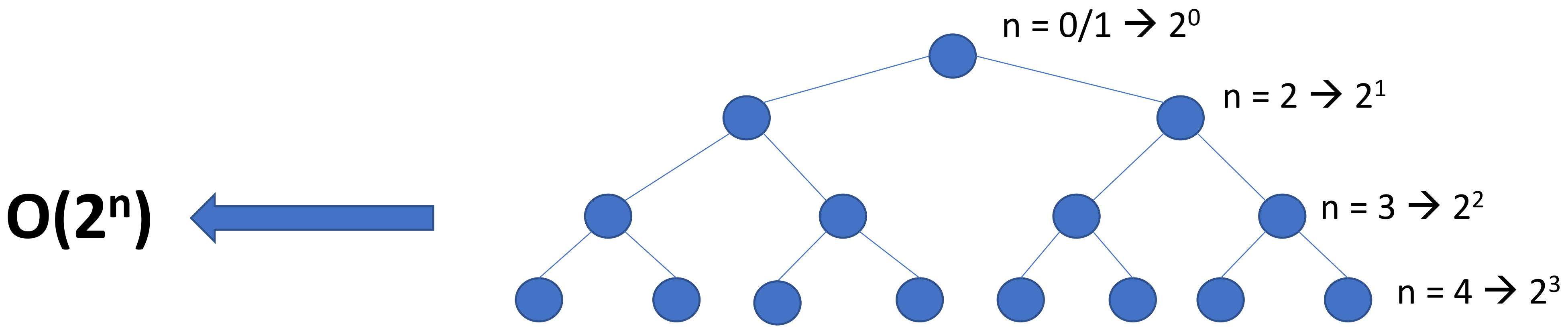
Best case:
O(1)

Worst case:
O(1) + O(1) + O(n) → O(n)

Complexity of recursive Fibonacci

```python
def recFib(n):
    if n==0:
        return 0
    elif n==1:
        return 1
    else:
        return recFib(n-1) + recFib(n-2)
```

# of recursive calls



Fibonacci with dictionary

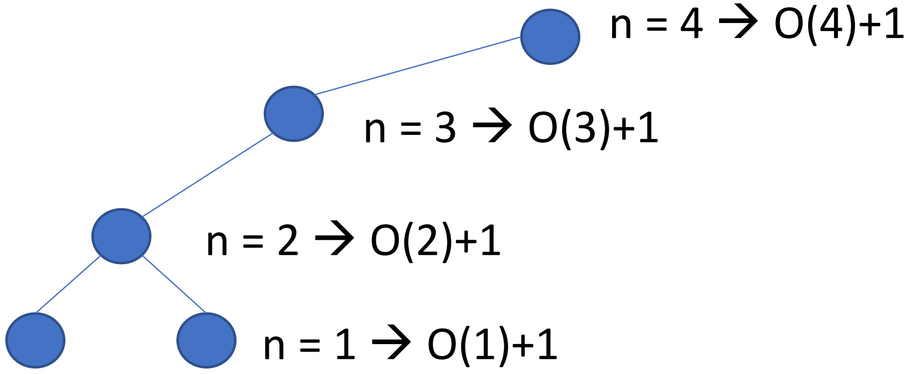
```python
def fibbonaci_dict(n,d):
    if n in d:
        return d[n]
    else:
        res = fibbonaci_dict(n-1,d)+fibbonaci_dict(n-2,d) # memoization
        d[n] = res
        return res
d = {1:1,2:2}
print(fibbonaci_dict(6,d))
```

Initialize dictionary with base cases

Complexity of recursive Fibonacci with memoization

```python
def fibbonaci_dict(n,d):
    if n in d:
        return d[n]
    else:
        res = fibbonaci_dict(n-1,d)+fibbonaci_dict(n-2,d) # memoization
        d[n] = res
        return res
d = {1:1,2:2}
print(fibbonaci_dict(6,d))
```

O(n)+1→ O(n) ←



Summary of Complexity growth

notation	complexity	Example
constant	$O(1)$	Input independent functions (c to f)
logarithmic	$O(\log(n))$	Binary search in a sorted list
linear	$O(n)$	Single loops
log-linear	$O(n \log n)$	Merge sort (to be cont.)
n^c - Polynomial n^2 - quadratic	$O(n^2)$	Nested loops
exponential	$O(2^n)$	Recursions with two recursive calls (Fibonacci)