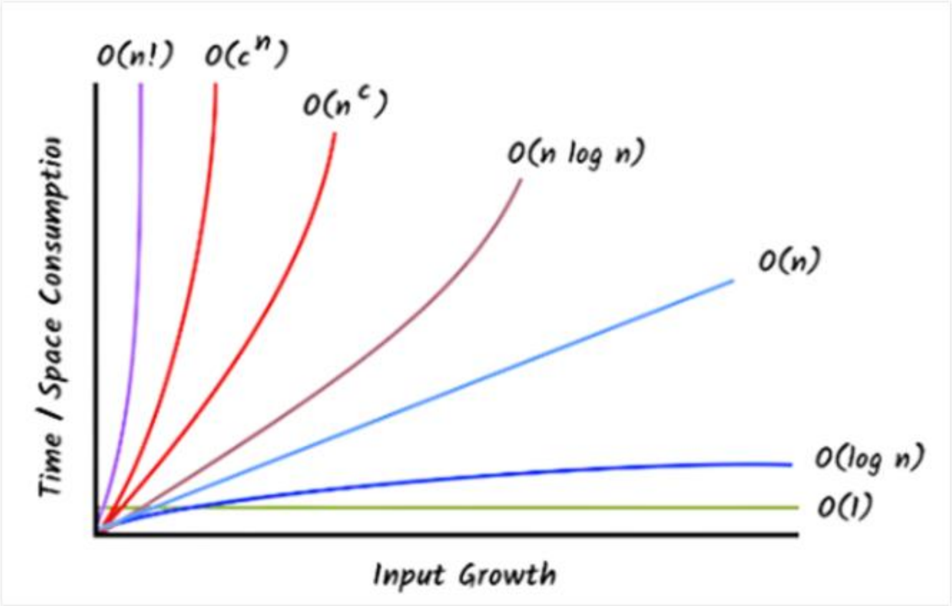
CDS2003: Data Structures and Object-Oriented Programming

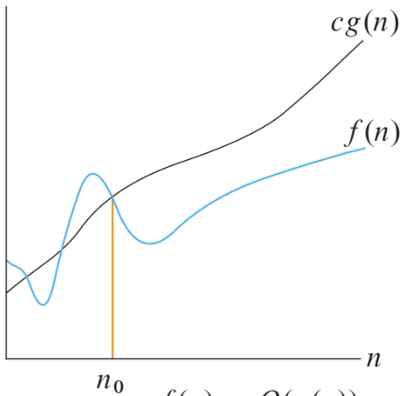
Lecture: Algorithm Analysis

Review

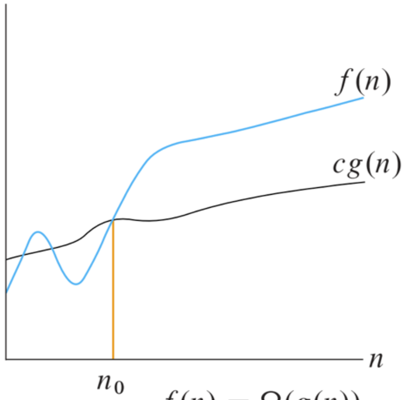
- Algorithm analysis after implementation
  - Time complexity
  - Space time complexity

- Algorithm analysis:
  - Big-Oh notation $f(n) = O(g(n))$
  - Big-Omega notation $f(n) = \Omega(g(n))$
  - Big-Theta notation $f(n) = \Theta(g(n))$

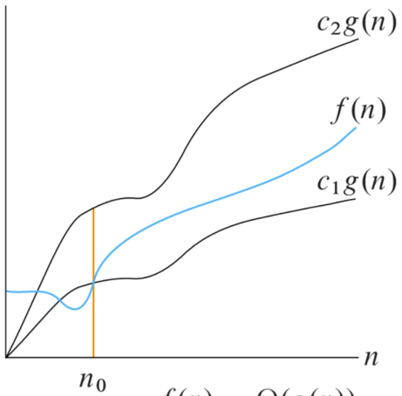




$f(n) = O(g(n))$



$f(n) = \Omega(g(n))$

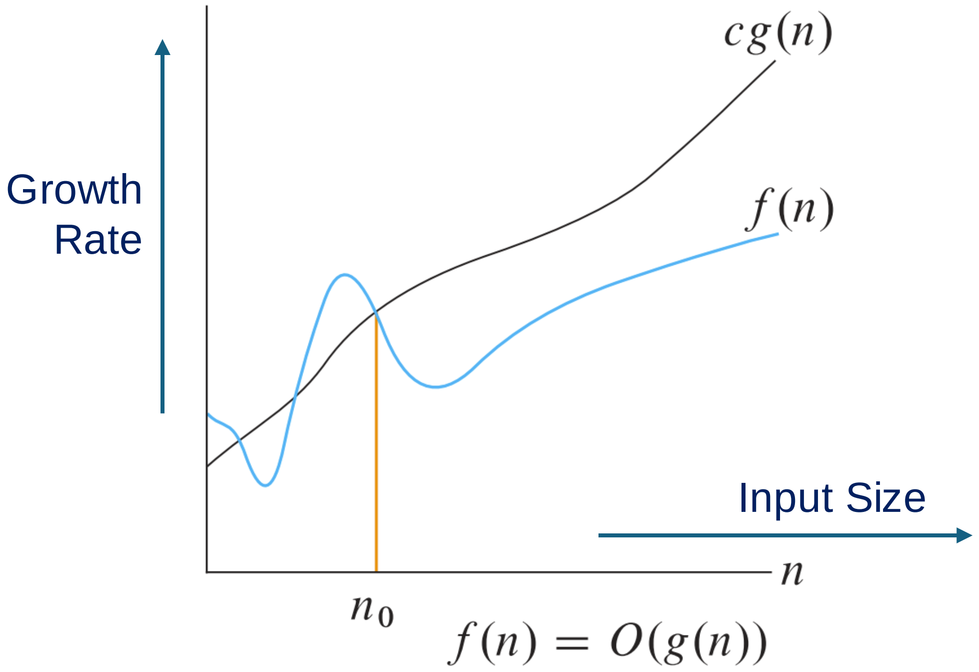


$f(n) = \Theta(g(n))$

[1] https://www.scholarhat.com/tutorial/datastructures
[2] Jain, H. (2016). Problem Solving in Data Structures & Algorithms Using Java: The Ultimate Guide to Programming. CreateSpace Independent Publishing Platform.

Algorithm analysis – Big-Oh notation

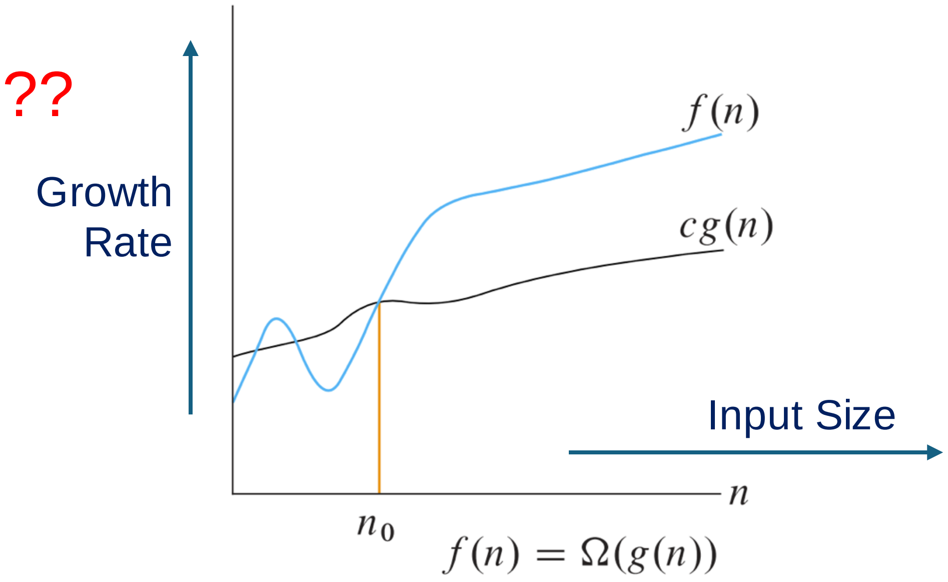
- The "Big-Oh notation" is commonly used for algorithm complexity.
  - $f(n) = O(g(n))$ if there exist positive constants c and n_0. such that $f(n) \le cg(n)$ when $n \ge n_0$.
  - In other words, $cg(n)$ gives an upper bound for $f(n)$.
  - The function $f(n)$ growth is slower than $cg(n)$.
  - Example: $n^2 + n = O(n^2)$.
  - Example: $n^2 + n = O(n^3)$???
  - There are many upper bounds.
  - Which one is better?



[1] Jain, H. (2016). Problem Solving in Data Structures & Algorithms Using Java: The Ultimate Guide to Programming. CreateSpace Independent Publishing Platform.

Algorithm analysis – Big-Omega notation

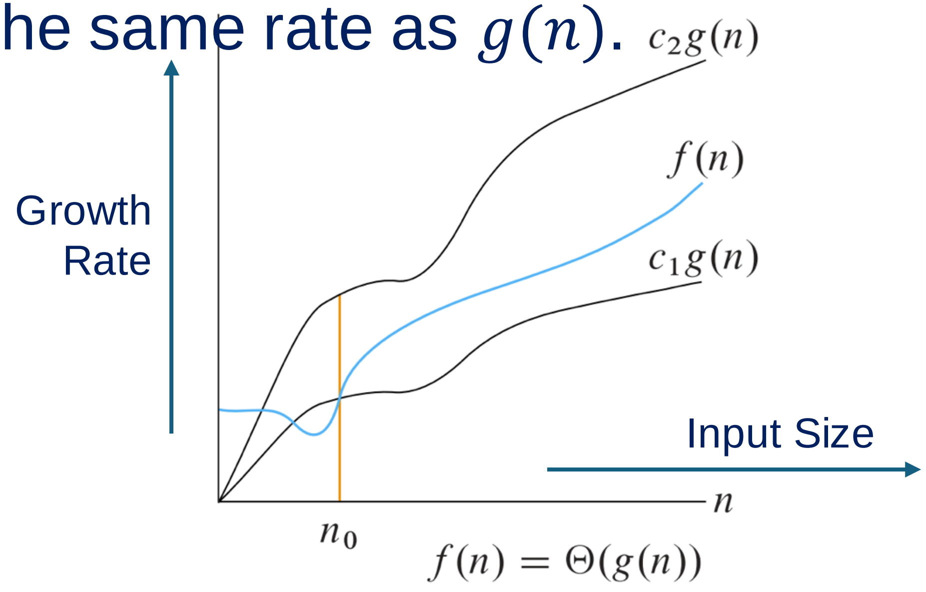
- The "Big-Omega notation"
  - $f(n) = \Omega(g(n))$ if there exist positive constants c and n_0. such that $f(n) \geq cg(n)$ when $n \geq n_0$.
  - In other words, $cg(n)$ gives a lower bound for $f(n)$.
  - The function $f(n)$ growth is faster than $cg(n)$.
  - Example: $f(n) = c^n$ and $g(n) = n^c$ give $f(n) = \Omega(g(n))$.
  - Example: $f(n) = n^3 + 2n^2 = \Omega(n^3)$.
  - Example: $f(n) = n^3 + 2n^2 = \Omega(n^{2.5})$???
  - There are many lower bounds.
  - Which one is better?



[1] Jain, H. (2016). Problem Solving in Data Structures & Algorithms Using Java: The Ultimate Guide to Programming. CreateSpace Independent Publishing Platform.

Algorithm analysis – Big-Theta notation

- The “Big-Theta notation”
  - $f(n) = \Theta(g(n))$ if there exists positive constants c_1, c_2, and n_1. such that $c_1 g(n) \leq f(n) \leq c_2 g(n)$ when $n \geq n_1$.
  - In other words, $c_1 g(n)$ gives a lower bound for $f(n)$,
  - And $c_2 g(n)$ gives an upper bound for $f(n)$,
  - The function $g(n)$ is an asymptotically tight bound on $f(n)$.
  - In other words, the function $f(n)$ grows at the same rate as $g(n)$.
  - Example: $f(n) = n^3 + n^2 + n = \Theta(n^3)$.

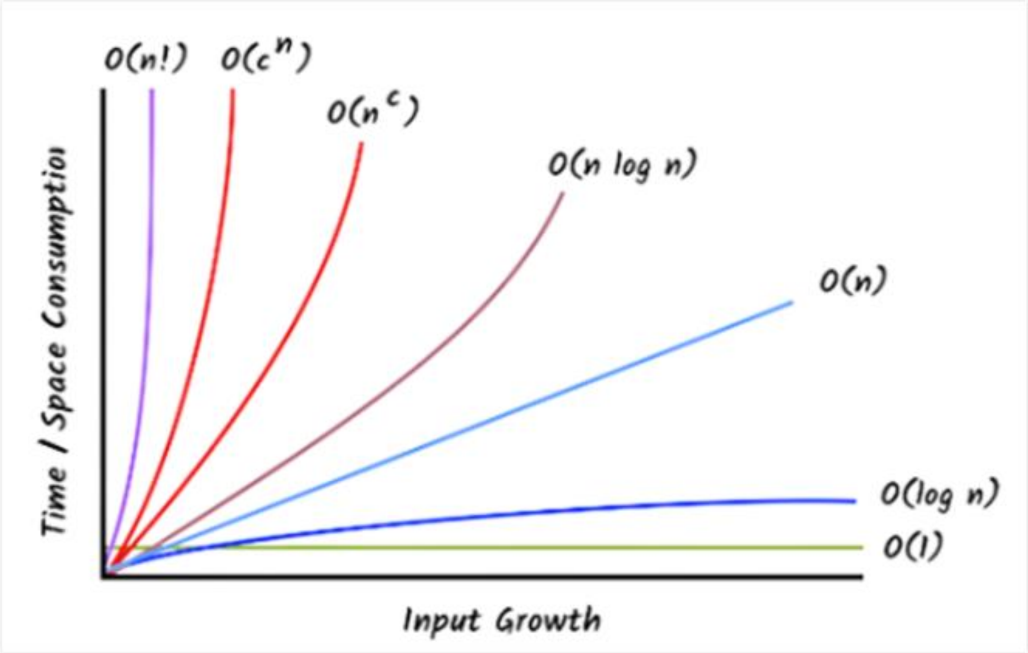


[1] Jain, H. (2016). Problem Solving in Data Structures & Algorithms Using Java: The Ultimate Guide to Programming. CreateSpace Independent Publishing Platform.

Main types of complexities

- Constant complexity $O(1)$
  - Independent of the input size n
- Logarithmic complexity $O(\log n)$
- Square root complexity $O(\sqrt{n})$
- Linear complexity $O(n)$
- N-LogN complexity $O(n \log n)$
- Quadratic complexity $O(n^2)$
- Polynomial complexity $O(n^c)$
- Exponential complexity $O(c^n)$
- Factorial complexity $O(n!)$ or $O(n^n)$

Note that $c > 1$ is a constant.



Constant time complexity $O(1)$

- The running time is independent of the input size n.
  - Each statement is assumed to take a constant amount of time to run.

- Examples
  - Assigning a value to a variable
  - Determining a number is odd or even
  - Printing out a phase like “Hello World”
  - Accessing n^{th} element of an array
  - A push or pop operation of a stack
  - …

```
a = 5
print(a % 2 == 1)
print("Hello World!")
b = [0, 2, 1]
x = b[1]
b.append(a)
print(a)
```

[1] https://www.simplilearn.com/tutorials/data-structure-tutorial

Linear time complexity $O(n)$

- The running time is proportional to the input size
- When a function checks all values in an input data set or traverses all the nodes of a data structure, the complexity is no less than $O(n)$.
- Examples
  - Array operations like searching element, finding min, finding max, and so on
  - Linked list operations like traversal, finding min, finding max, and so on

```python
def main(n):
    for i in range(n):
        print(i)
```

[1] https://www.simplilearn.com/tutorials/data-structure-tutorial

Logarithmic time complexity $O(\log n)$

- The running time is proportional to the logarithm of the input size.

- An example
  - 1, 2, 4, 8, 16, …, 2^k,…
  - $2^k \leq n \Rightarrow k \leq \log_2 n$

```
def log_print(n):
   i = 1
   while i <= n:
      print("Hello World !!!")
      i = 2 * i
```

[1] https://www.geeksforgeeks.org/understanding-time-complexity-simple-examples/

N-LogN time complexity $O(n \log n)$

- An example
  - The inner loop: $\log_2 n$ iterations
  - The outer loop : n iterations

```
def nlog_print(n):
    for j in range(n):
        i = 1
        while i <= n:
            print("Hello World !!!")
            i = 2 * i
```

[1] https://www.geeksforgeeks.org/understanding-time-complexity-simple-examples/

Double logarithmic time complexity $O(\log \log n)$

- An example
  - $j = 1, i = 3 \rightarrow 9 = 3^2$
  - $j = 2, i = 9 \rightarrow 81 = 3^4 = 3^{2^2}$
  - $j = 3, i = 81 \rightarrow 3^8 = 3^{2^3}$
  - $\ldots\ j = k, i = 3 \rightarrow 3^{2^k} \ldots$
  - $3^{2^k} \leq n \Rightarrow \log 2^k \leq \log n$
  - $\Rightarrow k \leq \log \log n$

```
def loglog_print(n):
   i = 3
   for j in range(2,n+1):
      if(i >= n):
         break
      print("Hello World !!!")
      i *= i
```

[1] https://www.geeksforgeeks.org/understanding-time-complexity-simple-examples/

Quadratic time complexity $O(n^2)$

- The running time grows quadratically with the input size.

```
def main2(n):
    for i in range(n):
        for j in range(n):
            print(i+j)
```

- An example: nested loops
  - Inner loop: n iterations
  - Outer loop: n iterations

```
def main3(n):
    a = []
    for i in range(n):
        a.append(i)
        for j in range(n):
            a[i] = a[i] + j
            print(a[i])
    return a
```

- Other examples
  - Bubble-sort
  - Selection-sort
  - Insertion-sort

[1] https://www.simplilearn.com/tutorials/data-structure-tutorial

Exponential time complexity $O(c^n)$

- The running time grows exponentially with the input size.

- Examples
  - A brute-force search of all possible subsets of the input data set
  - The recursive calculation of Fibonacci numbers (space complexity $O(n)$)

```python
# Algorithm 1
def get_fn_1(n):
    if n < 2:
        fn = n
    else:
        fn = get_fn_1(n-1) + get_fn_1(n-2)
    return fn
```

[1] https://www.simplilearn.com/tutorials/data-structure-tutorial

Factorial time complexity $O(n!)$

- The running time grows factorially with the input size.

- Examples
  - A brute-force search of all possible permutations of the input elements
  - A naïve solution to the Traveling Salesman problem

```
def factorial(n):
    for _ in range(n):
        print(n)
        factorial(n-1)
```

General rules for deriving the time complexity

- Constants
  - Each statement takes a constant time to run.

- Consecutive statements
  - Just add the time complexity of all the consecutive statements

- If-Else Statement
  - Consider the time complexity of the larger of “if” block or “else” block.

[1] Jain, H. (2016). Problem Solving in Data Structures & Algorithms Using Java: The Ultimate Guide to Programming. CreateSpace Independent Publishing Platform.

General rules for deriving the time complexity

- Loops
  - The time complexity of a loop is a product of the number of iterations in the loop and the time complexity of the statements inside the loop.

- Nested loop
  - The time complexity of a nested loop is a product of the time complexity of the statements inside loop multiplied by a product of the size of all the loops.

- Logarithmic statement
  - If each iteration the input size is decreased by a constant factor.

[1] Jain, H. (2016). Problem Solving in Data Structures & Algorithms Using Java: The Ultimate Guide to Programming. CreateSpace Independent Publishing Platform.

Exercise

- Please give the time complexity of the following algorithms using the big-O notation

```python
def algorithm_1(n):
    a = 0
    b = 0
    if n < 1:
        a += n
    else:
        b -= n
    c = a * b
    return c
```

```python
def algorithm_2(n):
    a = 0
    i = 0
    while i < n:
        a += 1
        i += 1
    return a
```

```python
def algorithm_3(n):
    a = 0
    for i in range(n):
        for j in range(100):
            a += 1
    return a
```

Exercise

- Please give the time complexity of the following algorithms using the big-O notation

```
def fun1(n):
    m = 0
    i = 0
    while i < n:
        m += 1
        i += 1
    return m

def fun2(n):
    m = 0
    i = 0
    while i < n:
        j = 0
        while j < n:
            m += 1
            j += 1
        i += 1
    return m
```

```
def fun3(n):
    m = 0
    i = 0
    while i < n:
        j = 0
        while j < i:
            m += 1
            j += 1
        i += 1
    return m

def fun4(n):
    m = 0
    i = 1
    while i < n:
        m += 1
        i = i * 2
    return m
```

```
def fun5(n):
    m = 0
    i = n
    while i > 1:
        m += 1
        i = i / 2
    return m

def fun6(n):
    m = 0
    i = 0
    while i < n:
        j = 0
        while j < n:
            k = 0
            while k < n:
                m += 1
                k += 1
            j += 1
        i += 1
    return m
```

```
def fun7(n):
    m = 0
    i = 0
    while i < n:
        j = 0
        while j < n:
            m += 1
            j += 1
        i += 1
    i = 0
    while i < n:
        k = 0
        while k < n:
            m += 1
            k += 1
        i += 1
    return m
```

Exercise

- Please give the time complexity of the following algorithms using the big-O notation

```python
def fun8(n):
    import math
    m, i = 0, 0
    while i < n:
        j = 0
        while j < math.sqrt(n):
            m += 1
            j += 1
        i += 1
    return m

def fun9(n):
    m = 0
    i = n
    while i > 1:
        j = 0
        while j < i:
            m += 1
            j += 1
        i /= 2
    return m
```

```python
def fun10(n):
    m, i = 0, 0
    while i < n:
        j = i
        while j > 0:
            m += 1
            j -= 1
        i += 1
    return m

def fun11(n):
    m, i = 0, 0
    while i < n:
        j = i
        while j < n:
            k = j + 1
            while k < n:
                m += 1
                k += 1
            j += 1
        i += 1
    return m
```

```python
def fun12(n):
    m, i = 0, 0
    while i < n:
        j = 0
        while j < n:
            m += 1
            j += 1
        i += 1
    return m

def fun13(n):
    m, i = 0, 0
    while i <= n:
        j = 0
        while j <= i:
            m += 1
            j += 1
        i *= 2
    return m
```

Constant space complexity $O(1)$

```python
def algorithm_1(n):
    a = 0
    a += 1
    b = a + n
    return b
```

```python
def algorithm_2(n):
    a = 0
    i = 0
    while i < n:
        a += 1
        i += 1
    return a
```

Linear space complexity $O(n)$

```python
def seq_gen(n):
    a = []
    i = 0
    while i < n:
        a.append(i)
        i += 1
    return a
```

```python
def sum_n(inputs):
    result = 0
    for i in inputs:
        result += i
    return result
```

```python
# factorial with Recursion
def factorial_Recur(n):
    if n == 0:
        return 1
    return n * factorial_Recur(n-1)
```

Quadratic space complexity $O(n^2)$

```
def algorithm_sq(n):
    a = []
    i = 0
    while i < n:
        b = []
        j = 0
        while j < n:
            b.append(j)
            j += 1
        a.append(b)
        i += 1
    return a
```

Three cases in algorithm analysis

- Worst-case complexity
  - The complexity of solving the problem for the worst input of size n. It provides the upper bound for the algorithm. This is the most common analysis used.
  - Maximum amount of resource
- Average-case complexity
  - This complexity is defined with respect to the distribution of the values in the input data. Usually, if the distribution of the input values are not specified, we calculate the complexity for all the possible inputs and then take an average of it.
- Best-case complexity
  - The complexity of solving the problem for the best input of size n.
  - Minimum amount of resource

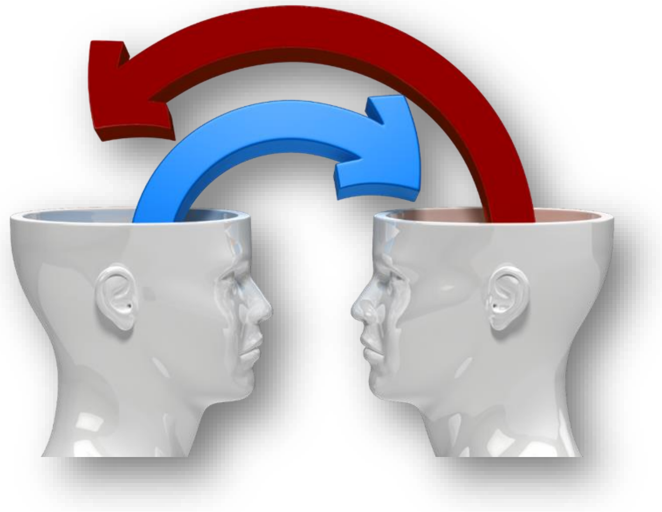
Three cases in algorithm analysis

- Time complexity
  - Worst case and average case $O(\log n)$
  - Best case $O(1)$

```
# Return index of x in arr if present, else -1
def binary_search_recursive(arr, low, high, x):
    # Check condition
    if high >= low:
        mid = (high + low) // 2
        # If the element is at the middle
        if arr[mid] == x:
            return mid
        # If the element is smaller than mid, go to the left subarray
        elif arr[mid] > x:
            return binary_search(arr, low, mid - 1, x)
        # Else go to the right subarray
        else:
            return binary_search(arr, mid + 1, high, x)
    else:
        # The element is not in the array
        return -1
```

[1] https://www.geeksforgeeks.org/binary-search/

Discussion



Q & A!