ENG 346
Data Structures and Algorithms for Artificial Intelligence
Runtime Complexity of the Algorithms

Dr. Mehmet PEKMEZCİ

mpekmezci@gtu.edu.tr

https://github.com/mehmetpekmezci/GTU-ENG-346

ENG-346-FALL-2025 Teams code is **0uv7jlm**

Complexity

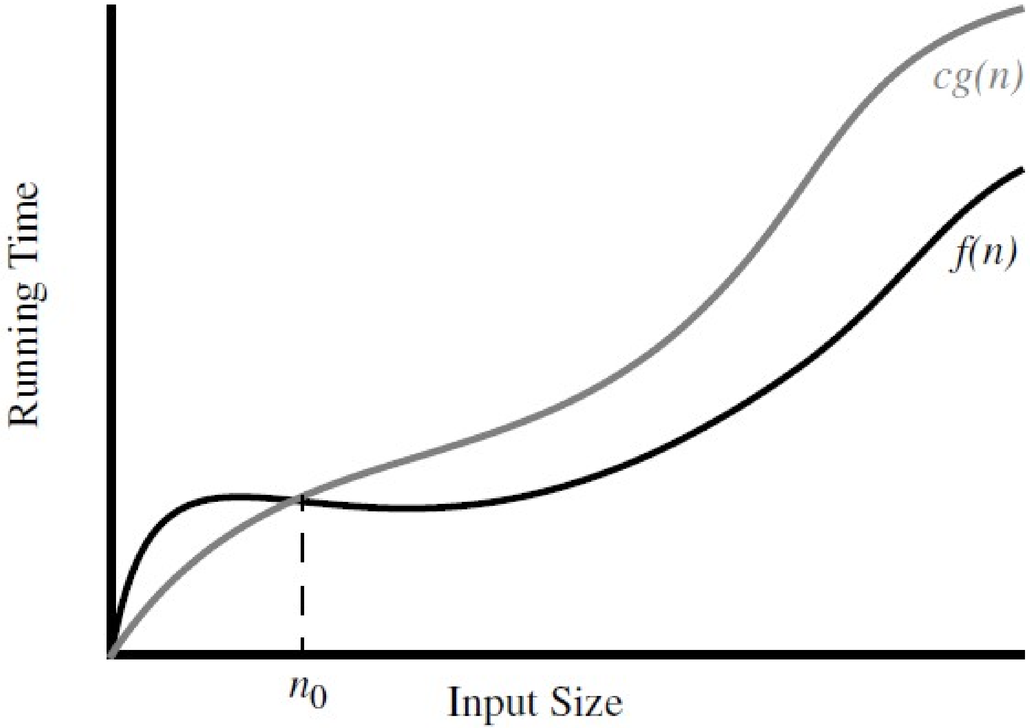
- **Time complexity** measures the amount of time (CPU Cycles) an algorithm takes to complete as a function of the input size. It's a way to estimate the running time of an algorithm.
    - Big O Notation (O-notation): This is used to describe the upper bound of an algorithm's running time. It tells you how the runtime scales with the size of the input.
- **Space complexity** measures the amount of memory (RAM) an algorithm uses as a function of the input size.
    - Big O Notation (O-notation): Just like time complexity, space complexity can be expressed in Big O notation.

Definitions: Big O

- Worst Case Scenario
- Upper-bound of a function f(n)
- Let f(n) and g(n) be functions mapping positive integers to positive real numbers. We say that f(n) is O(g(n)) if
  - there is a real constant $c > 0$ and
  - an integer constant $n0 \geq 1$ such that

    $f(n) \leq c\, g(n)$, for $n \geq n0$.

- f(n) is O(g(n))



Big O Rules

- Simplifications:

- If is f(n) a polynomial of degree d, then f(n) is O(n^d), i.e.,
  - Drop lower-order terms
  - Drop constant factors
- Use the smallest possible class of functions
  - Say “2n is O(n)” instead of “2n is O(n^2)”
- Use the simplest expression of the class
  - Say “3n + 5 is O(n)” instead of “3n + 5 is O(3n)”

Time Complexity Calculation n^2

```
N   = 100
sum = 0
for outer_loop_index in range(N):
    for inner_loop_index in range(N):
        sum += 1
print(sum)
```

$O(N^2) = (100)^2 = 10000$

Time Complexity Calculation n

```python
sorted_array=[1,3,5,8,12,14,18,20,22,25,26,27,30,35,36,38,39,40]
N=len(sorted_array)
print(f"N={N}")
searched_value=25
index_of_value=-1
for loop_index in range(N):
    if sorted_array[loop_index]==searched_value:
        index_of_value=loop_index
        break
print(index_of_value)
```

$$O(N) = (100) = 100$$

Time Complexity Calculation log(n)

```python
sorted_array=[1,3,5,8,12,14,18,20,22,25,26,27,30,35,36,38,39,40]
N=len(sorted_array)
searched_value=25

def binarySearch(arr, targetVal):
  left = 0
  right = len(arr) - 1

  while left <= right:
    mid = (left + right) // 2

    if arr[mid] == targetVal:
      return mid

    if arr[mid] < targetVal:
      left = mid + 1
    else:
      right = mid - 1

  return -1

print(binarySearch(sorted_array,searched_value))
```

Binary Search Algorithm

$O(\log_2(N)) = \log_2(100)$

MASTER THEOREM :

(Complexity for Recursive Algos.)

$T(N) = aT(N/b) + f(N)$

a = recursive call

b = divided sub problems

f(N)= complexity of one loop/call step

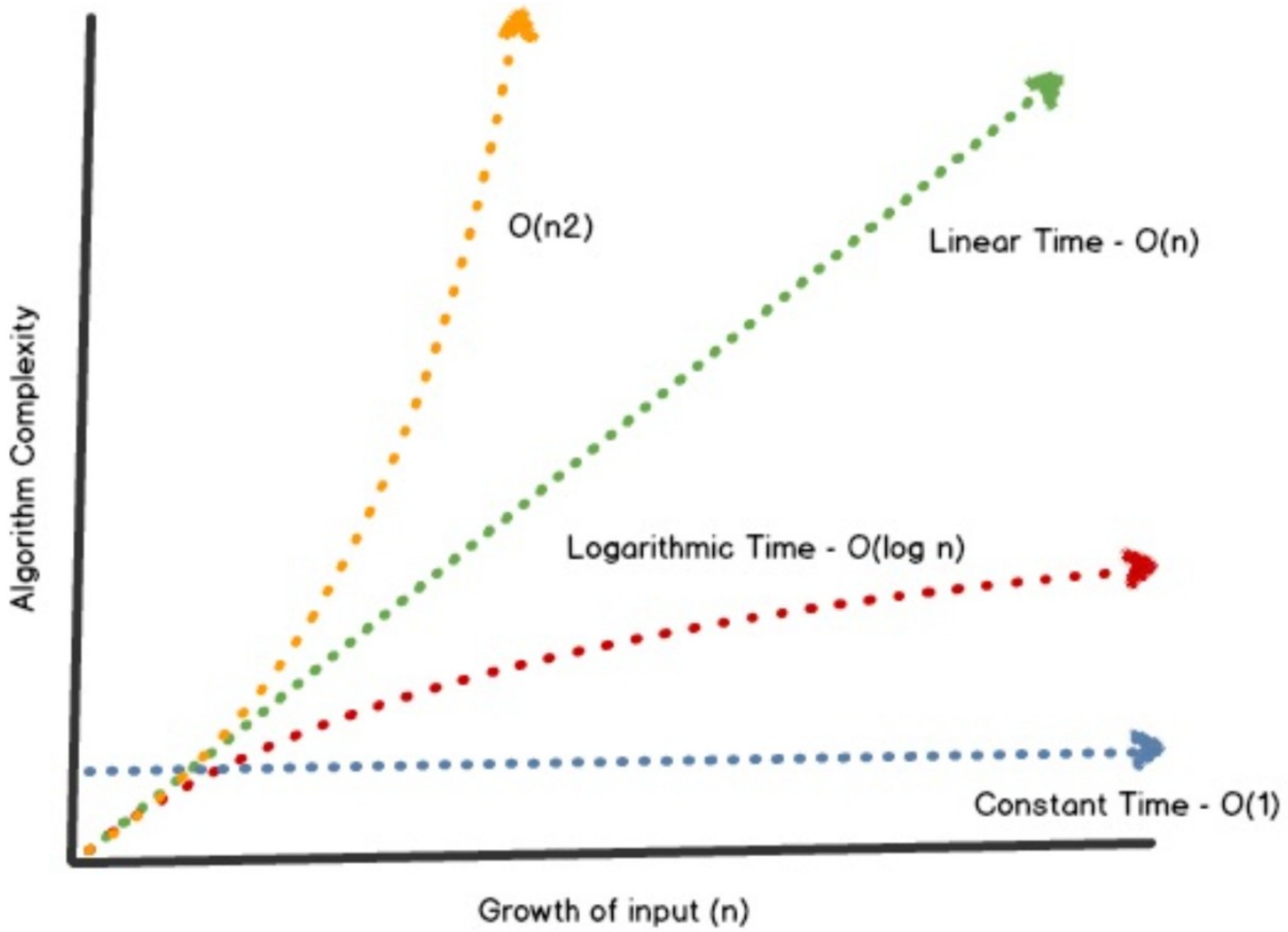
$T(N) = T(N/2) + O(1)$

→ Apply the rule in theorem.

Basics

Name	Function	Relation	Example
Constant Time	f(n) = c	Does not depend on input size.	Accessing array elements.
Logarithmic Time	f(n) = log n	Running time increases logarithmically with the input size.	Binary search.
Linear Time	f(n) = n	Running time increases linearly with the input size.	Iterating through an array or list.
Linearithmic Time	f(n) = n log n	The running time grows slower than O(n^2) but faster than O(n).	Efficient sorting algorithms like quicksort and mergesort.
Quadratic Time	f(n) = n^2	Running time grows proportionally to the square of the input size.	Algorithms with nested loops, such as selection sor tor bubble sort.
Polynomial Time	f(n) = n^k	Running time is a polynomial function of the input size.	Algorithms with “k” nested loops.
Exponential Time	f(n) = 2^n	Running times that grow very rapidly with the input size.	N-P complete problems, such as traveling salesman.

Growth Rates



Examples:

- $7n-2$ is $O(n)$
- $3n^3 + 20n^2 + 5$ is $O(n^3)$
- $3 \log n + 5$ is $O(\log n)$

Time-Space Complexity Trade-off

<u>Time Complexity :</u> Recalculate the values in each step of computation.

<u>Space Complexity:</u> Cache the calculated values and use pre-calculated values if possible.

- Compressed or Uncompressed data
- Re Rendering or Stored images
- Smaller code or loop unrolling
- Lookup tables or Recalculation

Access Times

- CPU Speed = 1 cycle (1 cycle = 0.3 ns for a 3GHz CPU)
- CPU Register = 1 cycle
- L1 Cache = 3 cycles
- L2 Cache = 10 cycles
- L3 Cache = 40 cycles
- RAM = 100 cycles
- SSD = 10K cycles
- HDD = 10M cycles

Exercises

- Book: R-3.1