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

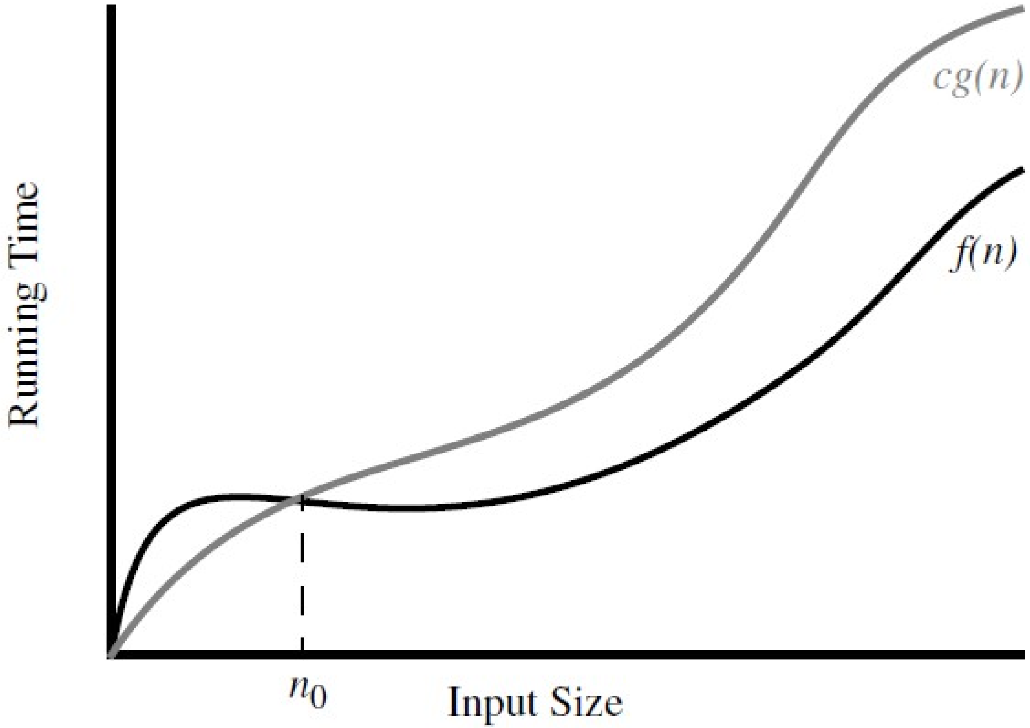
- **Time complexity** measures the amount of time (CPU Cycles) an algorithm takes to complete as a function of the input size. It's a way to estimate the running time of an algorithm.
  - Big O Notation (O-notation): This is used to describe the upper bound of an algorithm's running time. It tells you 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

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

- 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

```python
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

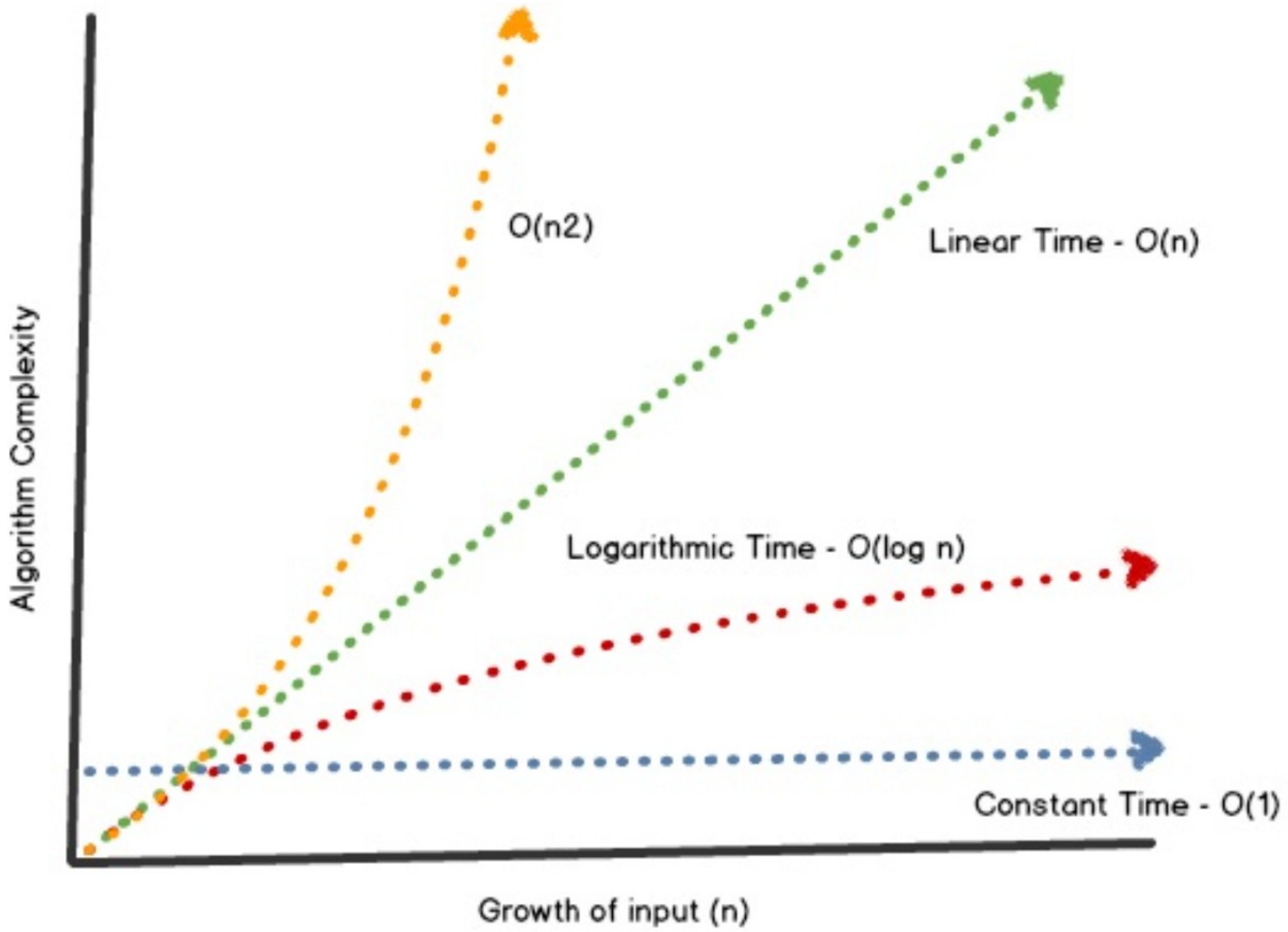
$T(N) = T(N/2) + O(1)$

→ 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 $O(\log n)$

Example : Complexity of Search Algorithms

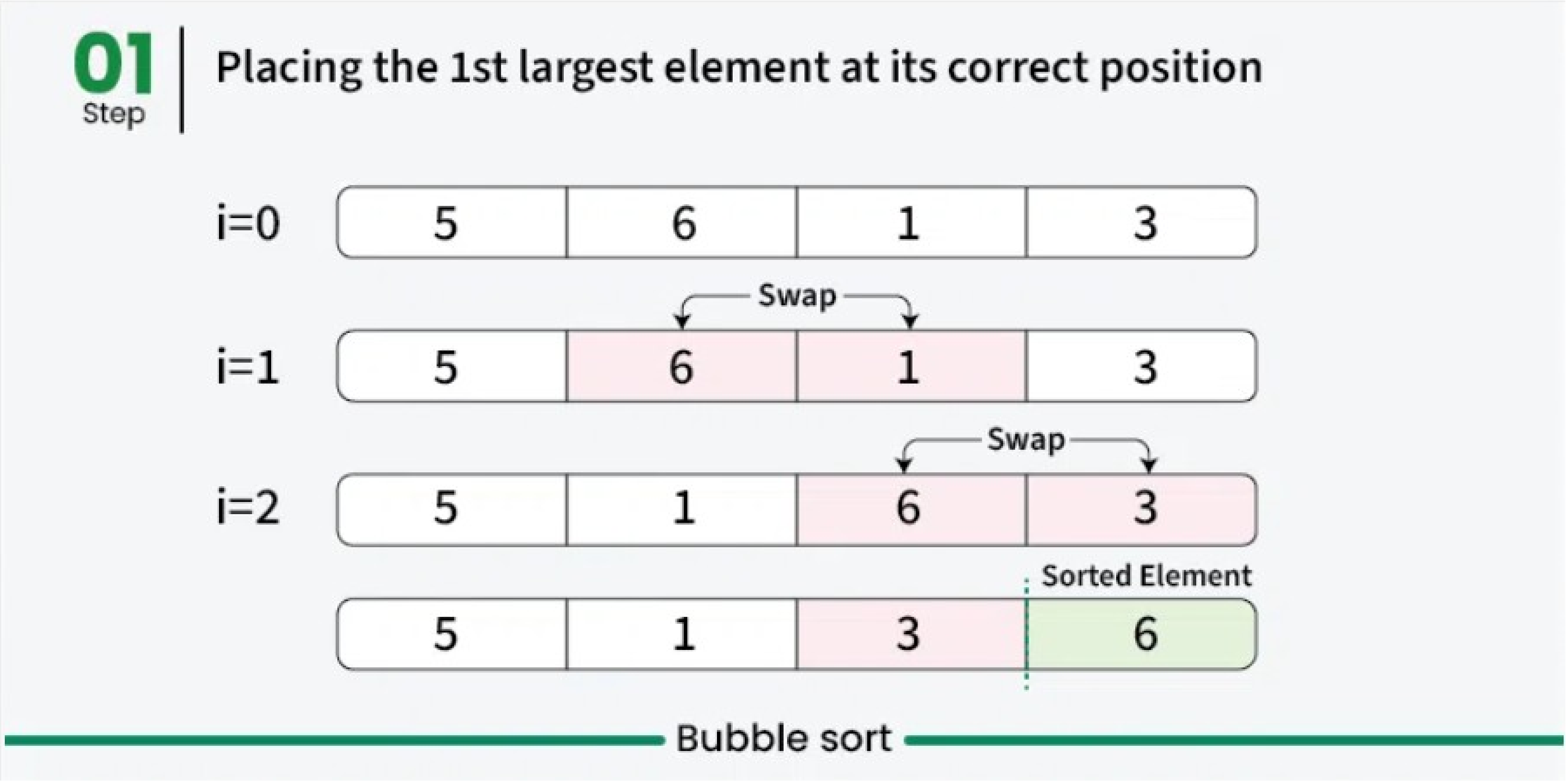
Sorting Algorithms	Time Complexity			Space Complexity
	Best Case	Average Case	Worst Case	Worst Case
Bubble Sort	Ω(N)	Θ(N^2)	O(N^2)	O(1)
Selection Sort	Ω(N^2)	Θ(N^2)	O(N^2)	O(1)
Insertion Sort	Ω(N)	Θ(N^2)	O(N^2)	O(1)
Quick Sort	Ω(N log N)	Θ(N log N)	O(N^2)	O(N)
Merge Sort	Ω(N log N)	Θ(N log N)	O(N log N)	O(N)
Heap Sort	Ω(N log N)	Θ(N log N)	O(N log N)	O(1)

Bubble Sort – algorithm

```
def bubbleSort(arr):
    n = len(arr)
    # Traverse through all array elements
    for i in range(n):
        swapped = False
        # Last i elements are already in place
        for j in range(0, n-i-1):
            # Traverse the array from 0 to n-i-1
            # Swap if the element is greater than the next
element
            if arr[j] > arr[j+1]:
                arr[j], arr[j+1] = arr[j+1], arr[j]
                swapped = True
        if (swapped == False):
            break
```

Time Complexity: $O(n^2)$
Auxiliary Space: $O(1)$

Bubble Sort


01
Step
Placing the 1st largest element at its correct position
i=0
5
6
1
3
Swap
i=1
5
6
1
3
Swap
i=2
5
1
6
3
Sorted Element
5
1
3
6
Bubble sort

Merge Sort

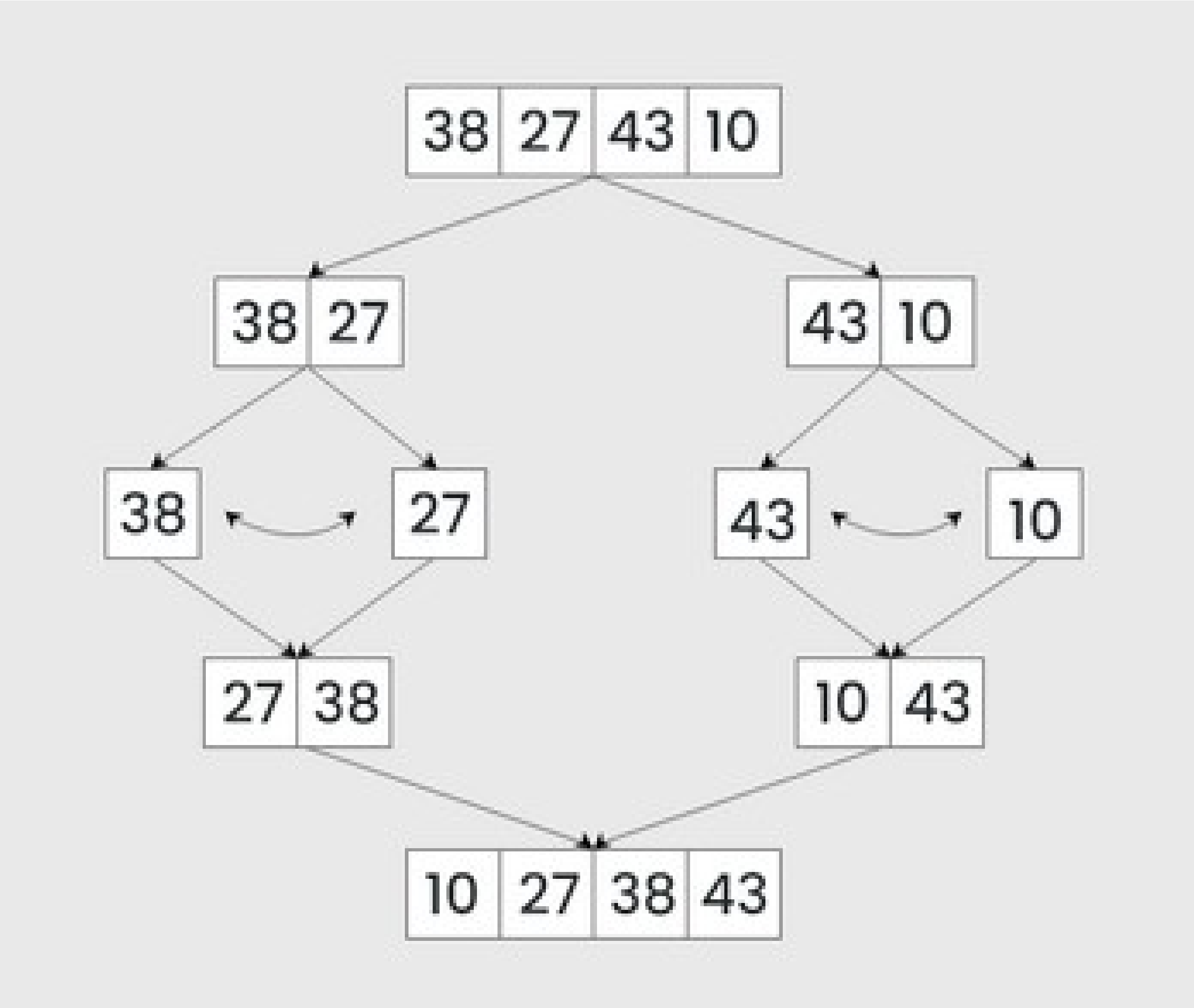
```python
def merge(left, right):
    result = []
    i = j = 0

    while i < len(left) and j < len(right):
        if left[i] < right[j]: result.append(left[i]); i += 1
        else: result.append(right[j]); j += 1

    result.extend(left[i:])
    result.extend(right[j:])

    return result
```

```python
def mergeSort(arr):
    step = 1  # Starting with sub-arrays of length 1
    length = len(arr)

    while step < length:
        for i in range(0, length, 2 * step):
            left = arr[i:i + step]
            right = arr[i + step:i + 2 * step]
            merged = merge(left, right)
            # Place the merged array back into the original array
            for j, val in enumerate(merged): arr[i + j] = val

        step *= 2  # Double the sub-array length for the next iteration

    return arr
```

```python
unsortedArr = [3, 7, 6, -10, 15, 23.5, 55, -13]
sortedArr = mergeSort(unsortedArr)
print("Sorted array:", sortedArr)
```

Time Complexity: O(n logn)
Auxiliary Space: O(n)

Merge Sort



Real Life Complexity Reduction Strategies

- Use a mathematical formula, if there exists :)
- Choose Efficient Algorithms :
  - Depends on problem type : Try to find the name of your problem in literature, then search for the efficient algorithms for that specific problem.
  - Breath First Search vs Depth First Search
  - Don’t trust what ChatGPT says, these are just suggestions.
- Use appropriate data structures (E.g. Lists, Double Linked Lists, Trees, Hash Maps, …)
- Implement Caching Mechanisms

Real Life Complexity Reduction Strategies

- Try to apply **Divide And Conquer** method to your problem (E.g. Merge Sort)
- Try to apply **Dynamic Programming** algorithms to your problem (E.g. Traveling Slaesman problem / Dijkstra Algorithm)
- Try to use **Memoization** (store intermediate results)
  - One of the reasons for GPU RAM PROBLEM :)
- Try to use **Big Data Algorithms** like MapReduce

Example : Fibonacci Numbers

- Example : Fibonacci Numbers : $a_n = a_{n-1} + a_{n-2}$
- $a_{100000} = ?$
- **Math** : Binet's Formula (Generating Functions) **O(log(n))**

$$F(n) = \frac{\phi^n - (1-\phi)^n}{\sqrt{5}} \qquad \phi = \frac{1+\sqrt{5}}{2}$$

- **Algorithm** : Find an algorithm that calculates faster with less resource :

```
def nth_fibonacci(n):
    if n <= 1: return n
    return nth_fibonacci(n - 1) + nth_fibonacci(n - 2)

print(nth_fibonacci(5))
```

O(2^n)

```
F_n=0 ; F_n_1=2; F_n_2=1
n=5
for  in range(n):
  F_n =  F_n_1 + F_n_2
  F_n_2=F_n_1
  F_n_1=F_n
print(F_n)
```

O(n)

Time-Space Complexity Trade-off

Time Complexity : Recalculate the values in each step of computation.

Space Complexity: 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