Data Structures and Algorithms

Dr. Farshid Mehrdoust
Kiarash Dadpour

University of Guilan

01

Algorithm Analysis

The Complexity of an Algorithm

The complexity of an algorithm refers to the measure of the computational resources it requires, primarily time and space, as a function of input size n.

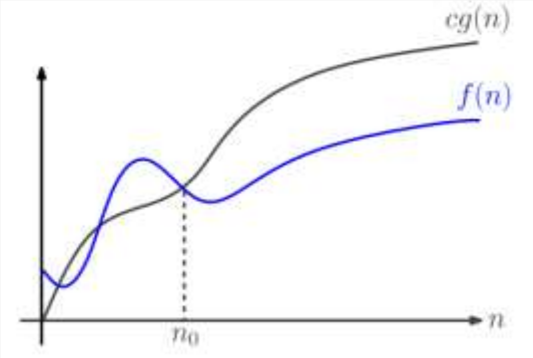
Time complexity quantifies the number of basic operations performed, while space complexity measures the amount of memory utilized.

Understanding complexity helps in comparing different algorithms and choosing the most efficient one for a given problem.

The Big-O Notation

O-notation characterizes an upper bound on the asymptotic behavior of a function.

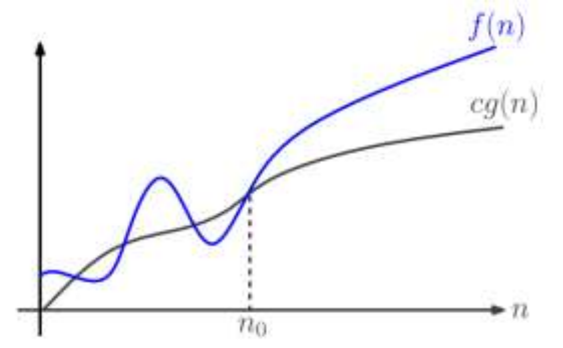
In other words, it says that a function grows no faster than a certain rate, based on the highest-order term.



The Ω Notation

Ω-notation characterizes a lower bound on the asymptotic behavior of a function.

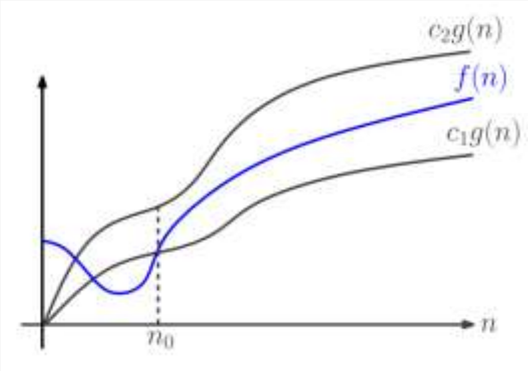
In other words, it says that a function grows at least as fast as a certain rate, based on the highest-order term.



The θ Notation

θ-notation characterizes a tight bound on the asymptotic behavior of a function.

It says that a function grows precisely at a certain rate, based on the highest-order term.



Asymptotic Notation: Formal Definitions

O-notation:

$$O(g(n)) = \{f(n) : \text{there exist positive constants } c \text{ and } n_0 \text{ such that } 0 \le f(n) \le cg(n) \text{ for all } n \ge n_0\}\ .$$[1]

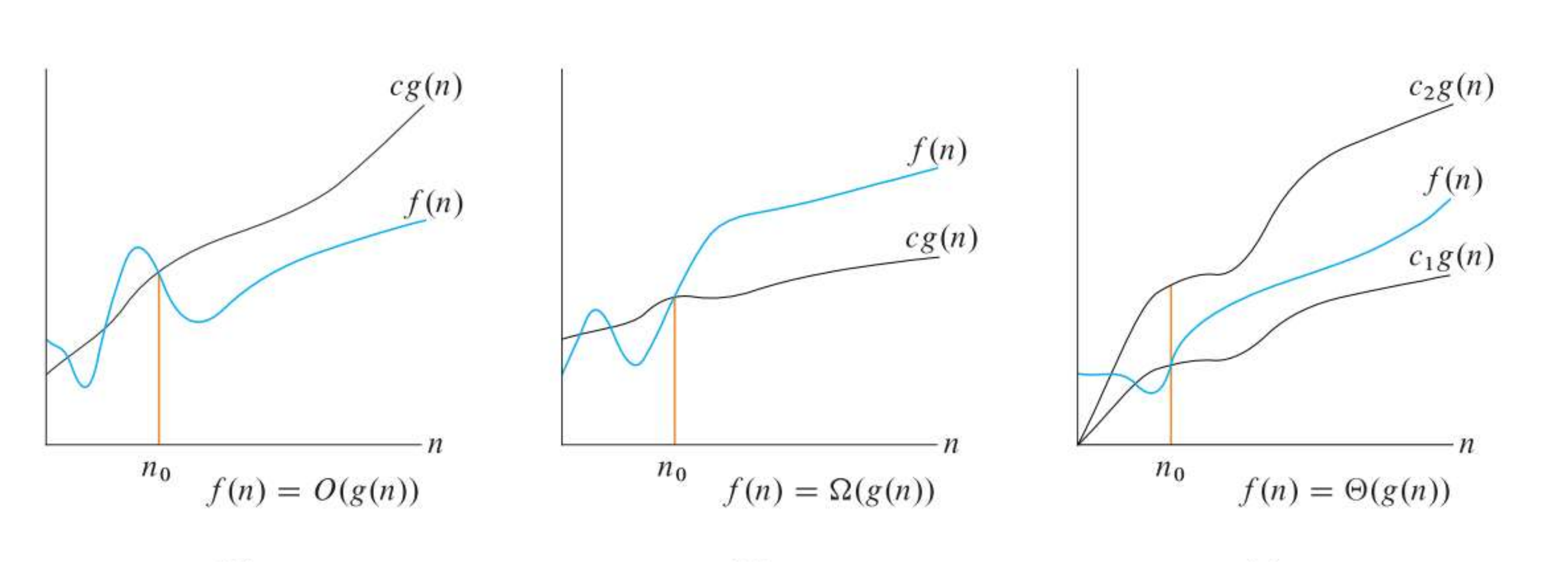
Ω-notation:

$$\Omega(g(n)) = \{f(n) : \text{there exist positive constants } c \text{ and } n_0 \text{ such that } 0 \le cg(n) \le f(n) \text{ for all } n \ge n_0\}\ .$$

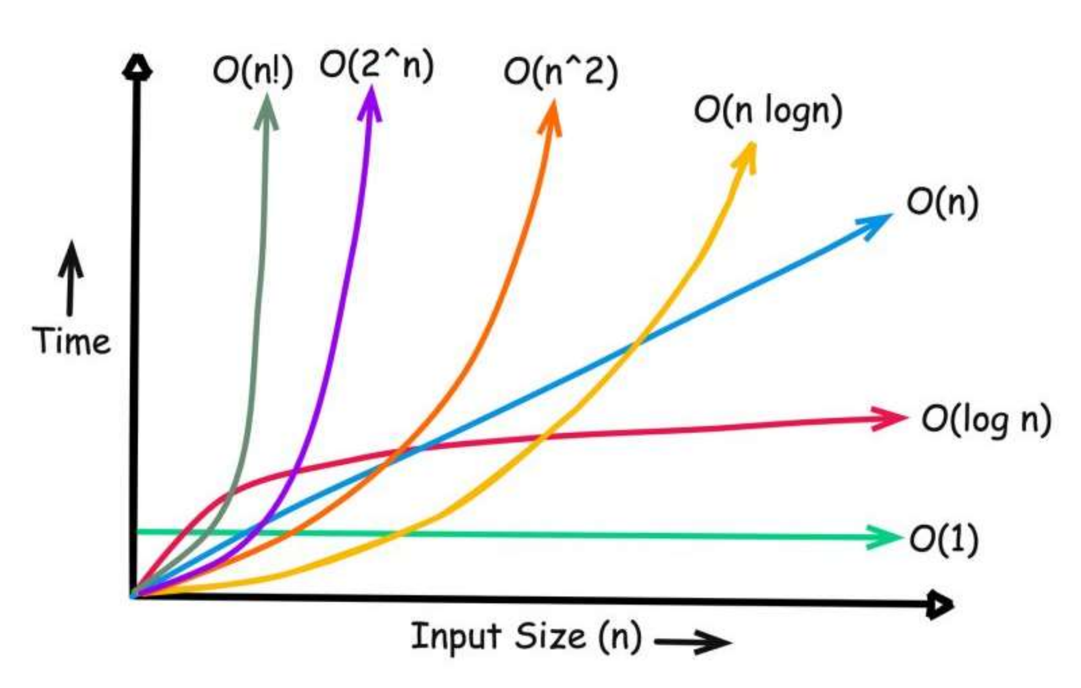
θ-notation

$$\Theta(g(n)) = \{f(n) : \text{there exist positive constants } c_1, c_2, \text{ and } n_0 \text{ such that } 0 \le c_1 g(n) \le f(n) \le c_2 g(n) \text{ for all } n \ge n_0\}\ .$$

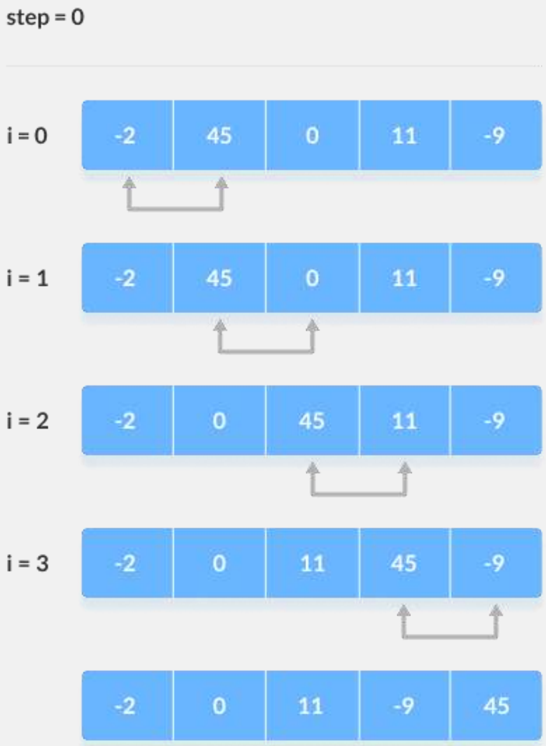
Asymptotic Notation: Formal Definitions



Different Time Complexities



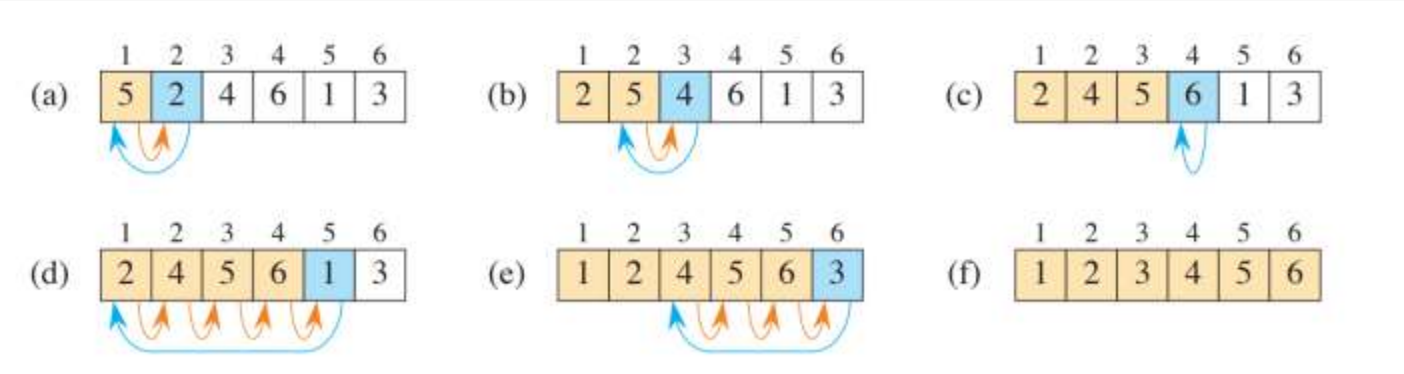
Bubble Sort



Bubble Sort

```python
def bubble_sort(A):
    n = len(A)
    for i in range(n):
        for j in range(0, n - i - 1):
            if A[j] > A[j + 1]:
                A[j], A[j + 1] = A[j + 1], A[j]
    return A
```

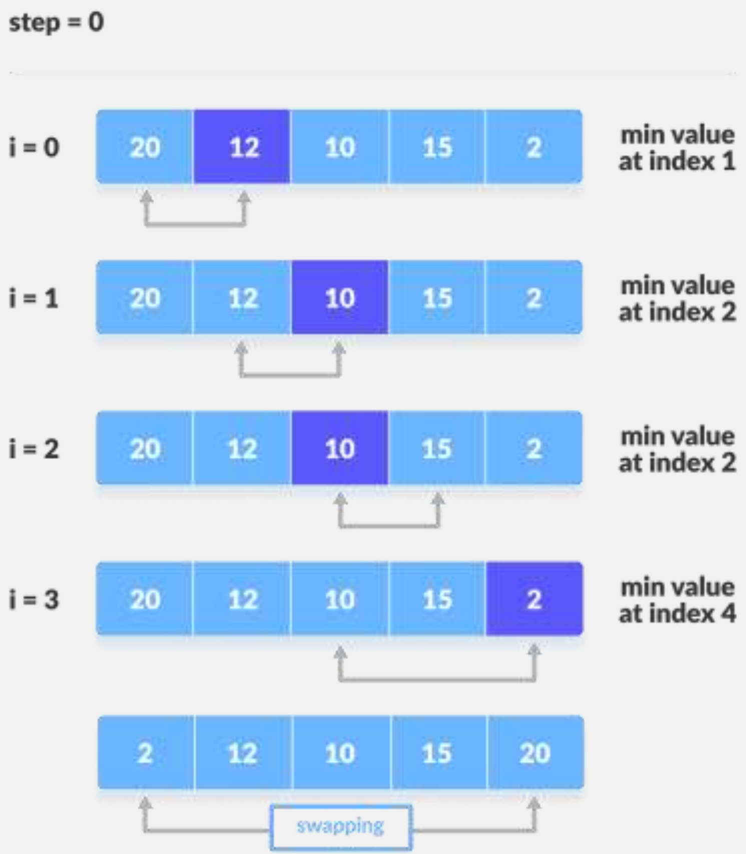
Insertion Sort



Insertion Sort

```python
def insertion_sort(A):
    n = len(A)
    for i in range(1, n):
        key = A[i]
        j = i - 1
        while j >= 0 and key < A[j]:
            A[j + 1] = A[j]
            j -= 1
        A[j + 1] = key
    return A
```

Selection Sort



Selection Sort

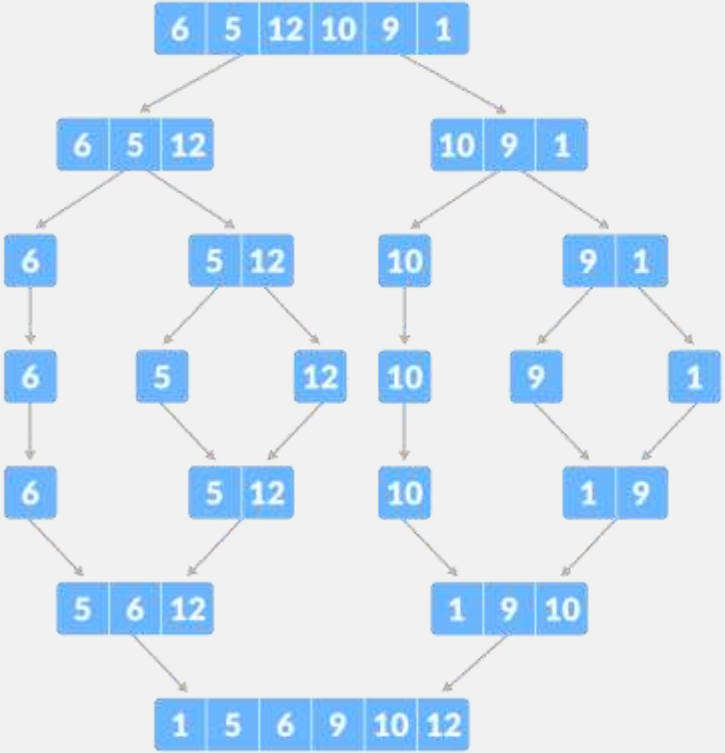
```
def selection_sort(A):
    n = len(A)
    for i in range(n):
        m_index = i
        for j in range(i + 1, n):
            if A[m_index] > A[j]:
                m_index = j
        A[m_index], A[i] = A[i], A[m_index]
    return A
```

Solving the Recurrence Equations

$$T(n) = \begin{cases} \theta(1) & n \leq 1 \\ 4T(\frac{n}{2}) + n & o.w \end{cases}$$

$$T(n) = \begin{cases} \theta(1) & n \leq 1 \\ 2T(\frac{n}{2}) + O(1) + O(n) & o.w \end{cases}$$

Merge Sort



6 5 12 10 9 1
6 5 12
10 9 1
6
5 12
10
9 1
6
5
12
10
9
1
6
5 12
10
1 9
5 6 12
1 9 10
1 5 6 9 10 12

Merge Sort

```python
def merge_sort(A):
    if len(A) <= 1:
        return A

    mid = len(A) // 2
    left_half = merge_sort(A[:mid])
    right_half = merge_sort(A[mid:])

    return merge(left_half, right_half)
```

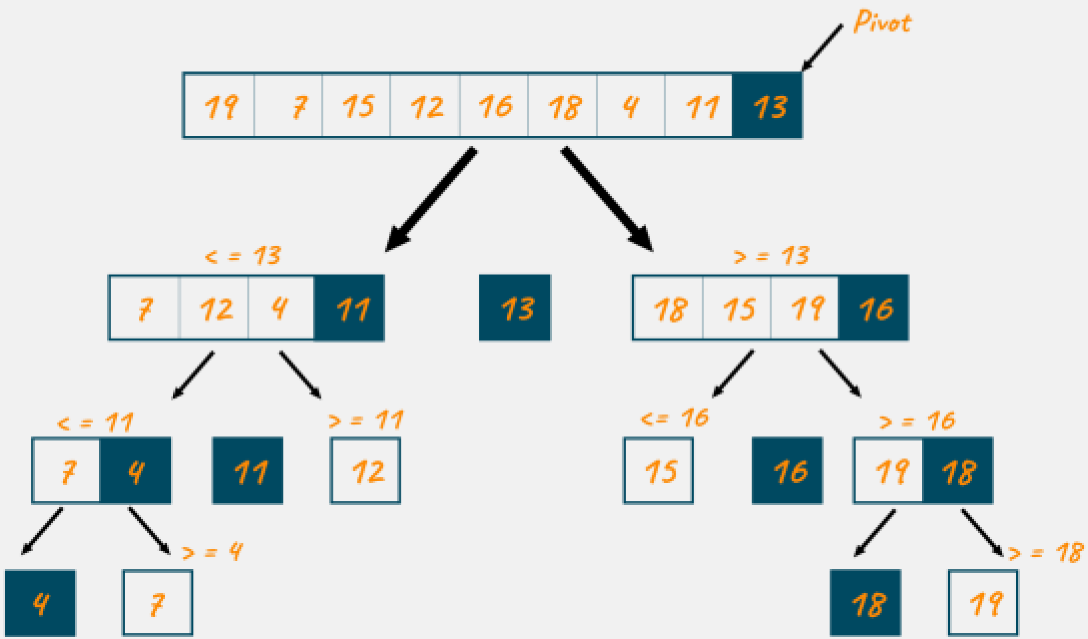
```python
def merge(left, right):
    sorted_array = []
    i = j = 0

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

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

    return sorted_array
```

Quick Sort



Pivot
19 7 15 12 16 18 4 11 13
< = 13
7 12 4 11
13
> = 13
18 15 19 16
< = 11
7 4
11
> = 11
12
<= 16
15
16
> = 16
19 18
> = 4
4
7
> = 18
18
19

Quick Sort

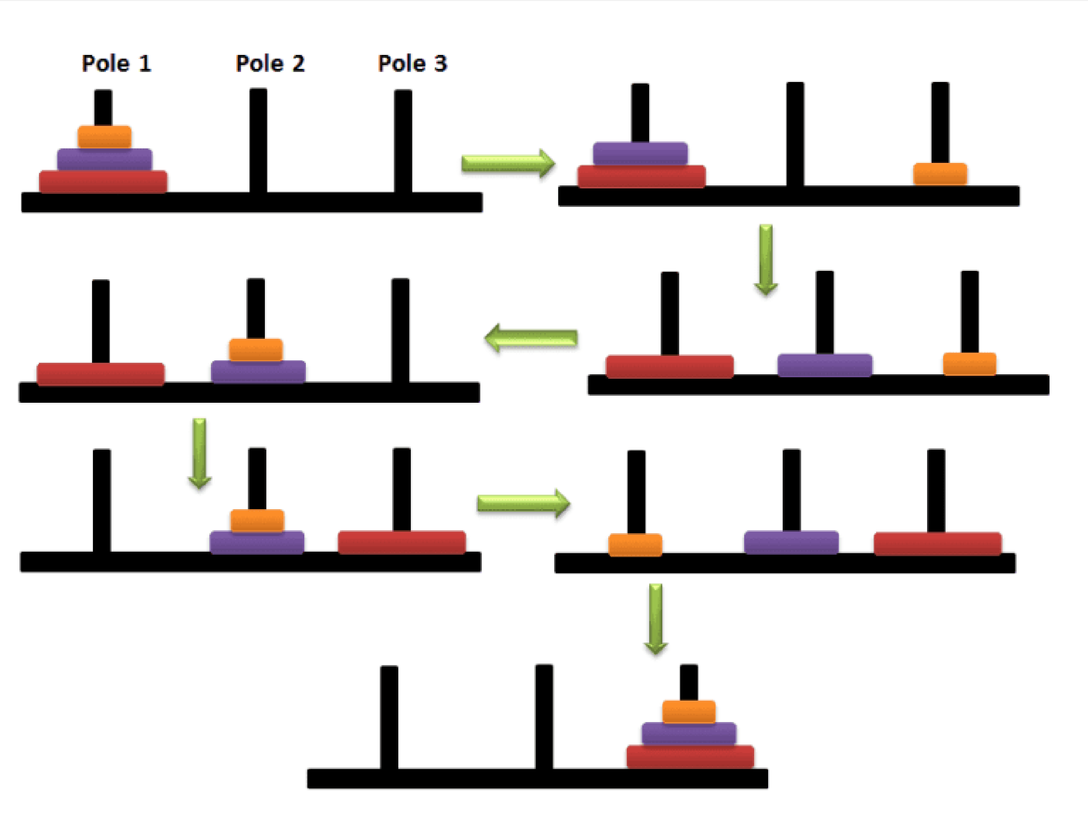
```python
def quick_sort(A):
    if len(A) <= 1:
        return A

    pivot = A[-1]
    left = []
    middle = []
    right = []

    for num in A:
        if num < pivot:
            left.append(num)
        elif num > pivot:
            right.append(num)
        else:
            middle.append(num)

    return quick_sort(left) + middle + quick_sort(right)
```

Hanoi Tower



Hanoi Tower

```
def hanoi(n, source, auxiliary, target):
    if n == 1:
        print(f"Move disk 1 from {source} to {target}")
        return
    hanoi(n - 1, source, target, auxiliary)
    print(f"Move disk {n} from {source} to {target}")
    hanoi(n - 1, auxiliary, source, target)
```