Stanford Divide & Conquer

Analysis of Algorithms

Question

What should be the CRITERIA for analysis of Algorithms?

Time Complexity

- How much time does the algorithm take to perform the procedure?
- The procedure or algorithm must be time efficient.
- The algorithm must be fast.
- After analyzing the time we get the **Time Function**.
- We **Do Not** get time in hours, minutes, seconds, milliseconds…..

Analysis of Algorithms

Space Complexity

How much memory or space the algorithm or procedure will consume?

Question

Is there any other CRITERIA for analysis of Algorithms?

Analysis of Algorithms

Time and **Space complexities** are the major criterias on which the algorithms are analyzed.

There can be **several** other criterias as well depending on the requirements.

Network Transfer

How much data would be transferred? Is algorithm transferring larger size of data or can it be reduced?

Power Consumption

In the era of increasingly handheld devices, how much power the algorithm is consuming can also an important criteria.

Analysis of Algorithms

The criterias you need to analyze depends on the project and its requirements. Some criterias for analysis of algorithms are as follows.

- Time Complexity
- Space Complexity
- Security
- Network Transfer
- Bandwidth Consumption
- Power Consumption
- Latency
- Reliability
- Scalability
- Cache Efficiency
- Concurrency
- I/O Operations

Time Complexity

We calculate Time complexity in mathematical notations rather than in absolute time due to some of the following reasons.

- Independence from Input size
- Predictive Power
- Machine and Environment Variability
- Consistency Across Implementations

Time Complexity

Some common mathematical notations to calculate Time complexity are as follows.

- Big O Notation (O)
- Big Omega Notation (Ω)
- Big Theta Notation (Θ)
- Little O Notation (o)
- Little Omega Notation (□)

Time Complexity

Big O Notation (O)

- Big O Notation **(O)** focuses on worst case complexity of an algorithm.
- Big O gives the upper bound on the growth rate of an algorithm.
- E.g. For a sorting algorithm, Big O shows the longest possible time it might take.
- We prioritize the term that grows the fastest as the input size increases.

Time Complexity

Big O Notation (O)

- Big O Notation ignores constant and less significant terms.
  - **O**(3n * 1000000000) = ?
  - **O**($5000n^2 + 3n^3$ + 10^9) = ?
  - **O**(n ÷ 2) = ?
  - **O**(n ÷ 1000000000) = ?

Time Complexity

Big O Notation (O)

- Focuses on the worst case.
- It represents upper bound of an algorithm's time or space complexity.
- The algorithm performs **at most** linear time in the worst case.

Big Omega Notation (Ω)

- Focuses on the best case.
- It represents lower bound of an algorithm's time or space complexity.
- The algorithm performs **at least** a constant time, no matter the input size.

Time Complexity

Big Theta Notation (Θ)

- Focuses on the average case.
- It represents tight bound of an algorithm's time or space complexity.
- It gives both the upper bound and lower bound of the algorithm performance.
- We represent theta notation when lower and upper bound are same or when worst or best cases are same.
- We do not express theta when upper and lower bounds are different.
- Example: ?

Time Complexity

```python
def print_elements(arr):
    # Iterate over the list and print each element
    for element in arr:
        print(element)  # Constant time operation
```

Worst case: **O(n)**

Best case: **Ω(n)**

Overall (tight bound): **Θ(n)**

Time Complexity

```python
def linear_search(arr, target):
    # Best case: Target is the first element
    if arr[0] == target:
        return 0

    # Worst case: Target is not found, or it's the last element
    for i in range(1, len(arr)):
        if arr[i] == target:
            return i

    return -1  # Target not found
```

Worst case: **O(n)**

Best case: **Ω(1)**

Time Complexity

Time Complexity as a Function

After Time Complexity analysis we get a function of time.

- Time function tells us how the algorithm behaves as the input grows.
- It describes the rate of growth based on the input size (n).
- **f(n) = O(n)**: The algorithm runs in linear time.
- **$T(n) = O(n^2)$**: The algorithm runs in quadratic time.
- **T(n) = O(1)**: The algorithm runs in constant time.

Time Complexity

Some common Time Complexities are as follows.

- Constant **$O(1)$**
- Logarithmic **$O(\log n)$**
- Linear **$O(n)$**
- Linearithmic **$O(n \log n)$**
- Quadratic **$O(n^2)$**
- Cubic **$O(n^3)$**
- Exponential **$O(2^n)$**
- Factorial **$O(n!)$**

Time Complexity

Constant O(1)

```python
def add_two_numbers(a, b):
    return a + b

print(add_two_numbers(5, 7))
```

Time Complexity

Linear O(n)

```python
def sum_elements(arr):
    total = 0
    for num in arr:
        total += num
    return total

result = sum_elements([100, 50, 90, 80])
print(result)
```

Time Complexity

Logarithmic O(log n) - Exponents & Logs

$q^4 = q \cdot q \cdot q \cdot q = 16$

$2^4 = 2 \cdot 2 \cdot 2 \cdot 2 = 16$

$2^x = 16$. What is **x** here ?

$2^x = 1024$. What is **x** here ?

$2^x = 1000000000$. What is **x** here ?

- **Base**
- **Exponents**
- **Logs**

Time Complexity

Logarithmic O(log n) - Exponents & Logs

$\text{Log}_3\ 81 = x$ What is **x** here ?

$3^x = 81$

Logarithmic form		Exponential form
$\log_2(8) = 3$	$\Longleftrightarrow$	$2^3 = 8$
$\log_3(81) = 4$	$\Longleftrightarrow$	$3^4 = 81$
$\log_5(25) = 2$	$\Longleftrightarrow$	$5^2 = 25$

Time Complexity

Logarithmic O(log n) - Exponents & Logs

PROBLEM 1

Which of the following is equivalent to $2^5 = 32$?

Choose 1 answer:

- (A) $\log_2(32) = 5$
- (B) $\log_5(2) = 32$
- (C) $\log_{32}(5) = 2$

Time Complexity

Logarithmic O(log n) - Exponents & Logs

PROBLEM 2

Which of the following is equivalent to $5^3 = 125$?

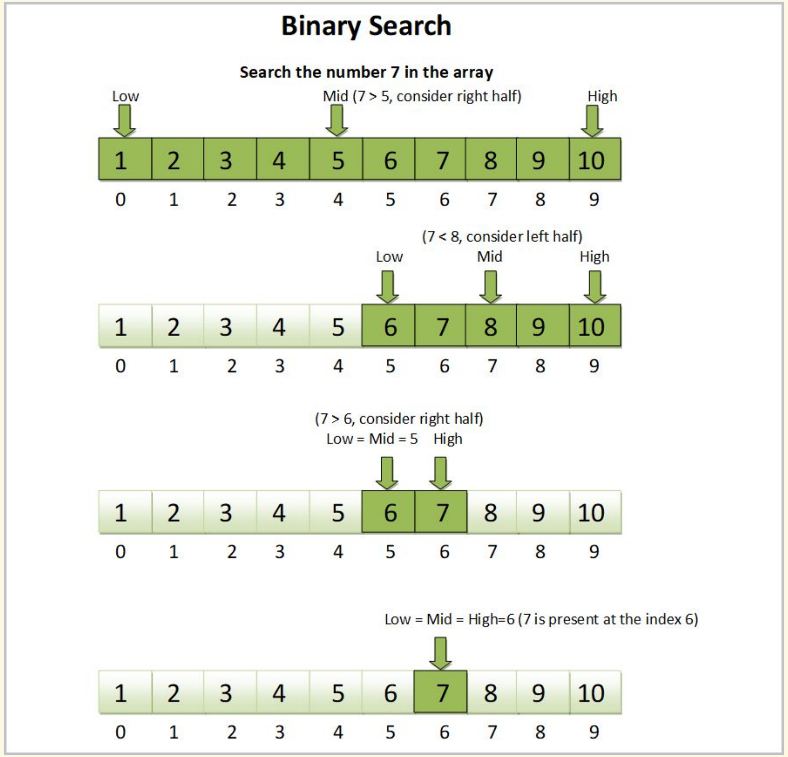
Choose 1 answer:

- (A) $\log_3(125) = 5$
- (B) $\log_5(125) = 3$
- (C) $\log_{125}(5) = 3$

Time Complexity

Logarithmic O(log n)

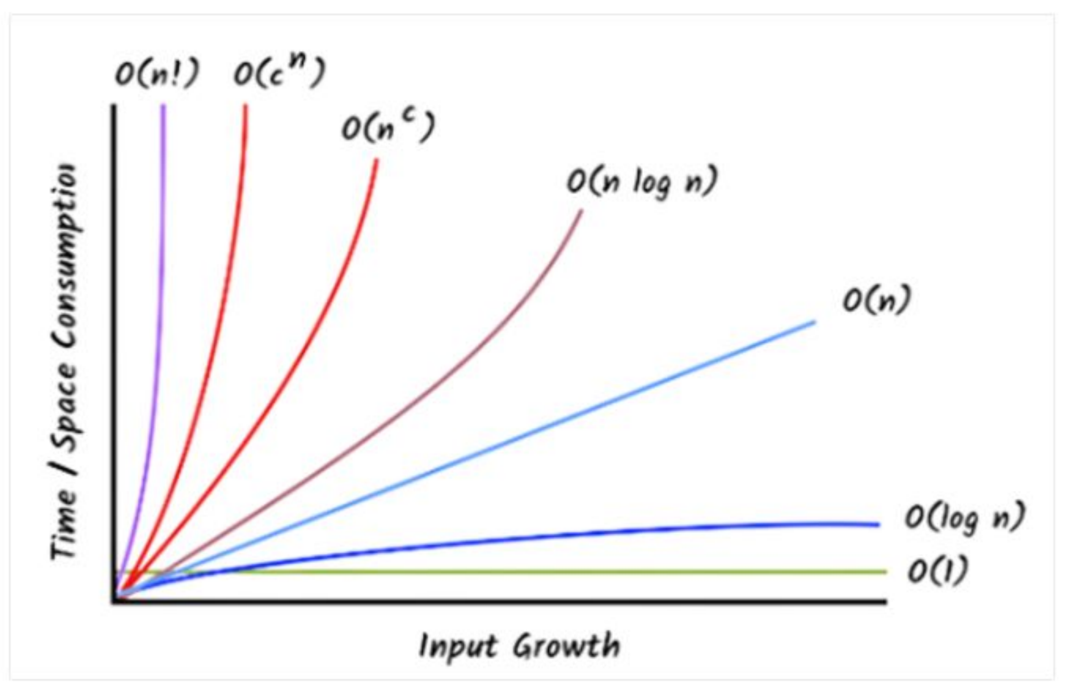
- **Binary Search**



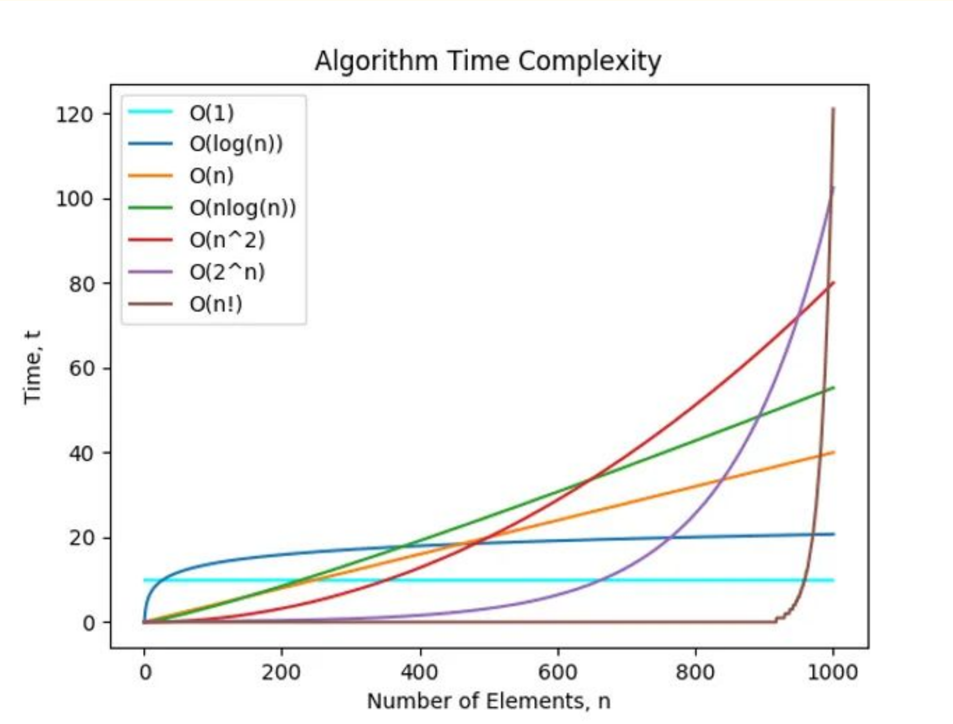
Time Complexity

Orders of Growth (Good to Bad)	
O(1)	Constant
O(log n)	Logarithmic
O(n)	Linear
O(n log n)	Linearithmic
$O(n^2)$	Quadratic
$O(n^3)$	Cubic
$O(2^n)$	Exponential
O(n!)	Factorial

Time Complexity



Time Complexity



Time Complexity

EXAMPLE ALGORITHM:	BINARY SEARCH	SIMPLE SEARCH	QUICKSORT	SELECTION SORT	THE TRAVELING SALESMAN

ARRAY SIZE	$O(\log n)$	$O(n)$	$O(n \log n)$	$O(n^2)$	$O(n!)$
10	0.3 sec	1 sec	3.3 sec	10 sec	4.2 days
100	0.6 sec	10 sec	66.4 sec	16.6 min	2.9×10^{149} years
1000	1 sec	100 sec	996 sec	27.7 hours	1.27×10^{2559} years

Estimates based on a slow computer that performs 10 operations per second

Analysis of Algorithms

Time Complexities in increasing order of growth.

$$O(1) < O(\log n) < O(\sqrt{n}) < O(n) < O(n \log n) < O(n^2) < O(n^3) < O(n^{100}) < \ldots..$$

$$< O(2^n) < O(3^n) < O(1000^n) < \ldots.. < O(n!) < O(n^n)$$

- Each next item will grow faster than the previous one
- Slowest growing to fastest growing.
- $O(1)$ grows the slowest, $O(n^n)$ grows the fastest.

Analysis of Algorithms

Time Complexities in decreasing order of growth.

$$O(n^n) > O(n!) > O(1000^n) > O(3^n) > O(2^n) > O(n^{100}) > O(n^3) > O(n^2) > O(n \log n) >$$

$$O(n) > O(\sqrt{n}) > O(\log n) > O(1)$$

- Each next item will grow slower than the previous one.
- Fastest growing to slowest growing.
- $O(n^n)$ grows the fastest, $O(1)$ grows the slowest.

Resources

[Datacamp - Understanding Big O Notation]

[Khan Academy - Understanding Logarithms]

[Khan Academy - Logarithms & Exponentials in Practice]

Thank You !