Time Complexity Part 2

Contents

[I. Introduction 3](#_Toc199603395)

[1. Why do we study time and space complexity? 3](#_Toc199603396)

[2. Time Complexity 5](#_Toc199603397)

[3. Asymptotic Analysis 5](#_Toc199603398)

[4. Examples and Comparisons 6](#_Toc199603399)

[Compare vs. 6](#_Toc199603400)

[5. Big O Notation 6](#_Toc199603401)

[6. Examples 7](#_Toc199603402)

[Find the time complexity of 7](#_Toc199603403)

[Find the time complexity of 7](#_Toc199603404)

[Find the time complexity of 8](#_Toc199603405)

[Find the time complexity of 8](#_Toc199603406)

[Find the time complexity of 9](#_Toc199603407)

[Some examples: 9](#_Toc199603408)

[7. Why Do We Neglect Lower-Order Terms in Big O? 10](#_Toc199603409)

[II. Issues with Big O Notation 11](#_Toc199603410)

[1. Big O Hides Constant Factors 11](#_Toc199603411)

[2. Big O Doesn’t Help with Small Inputs 11](#_Toc199603412)

[3. Big O Doesn't Show Actual Behaviour 11](#_Toc199603413)

[4. Big O Assumes Large Inputs 12](#_Toc199603414)

[5. When Should You Trust Big O? 12](#_Toc199603415)

[III. Space Complexity 13](#_Toc199603416)

[1. Analysing Space Step-by-Step 13](#_Toc199603417)

[Example 1: Basic Variables 13](#_Toc199603418)

[Example 2: Array of Size n 13](#_Toc199603419)

[Example 3: 2D Array of Size n×n 14](#_Toc199603420)

[2. Space Can Change Over Time 14](#_Toc199603421)

[IV. Time Limit Exceeded (TLE) 15](#_Toc199603422)

[What is it? 15](#_Toc199603423)

[How to Avoid? 15](#_Toc199603424)

[Example: 15](#_Toc199603425)

[Big O is deceiving 16](#_Toc199603426)

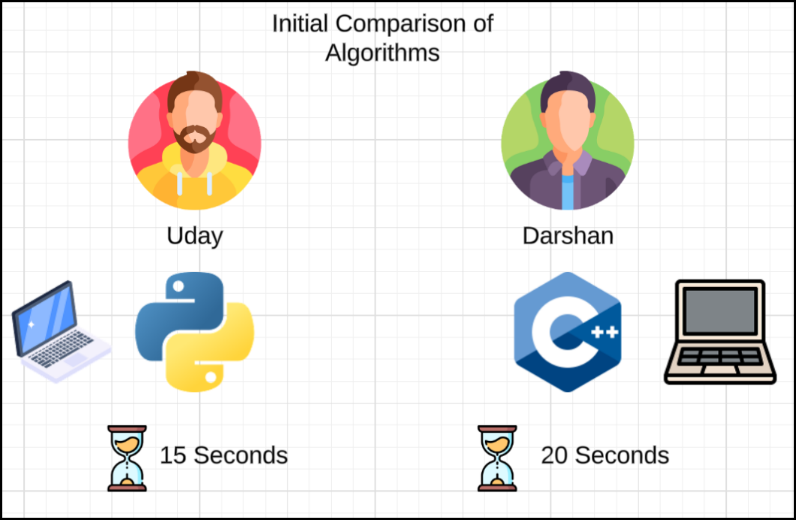
[When Do You Get TLE? 16](#_Toc199603427)

[What Happens Under the Hood 16](#_Toc199603428)

# Introduction

## Why do we study time and space complexity?

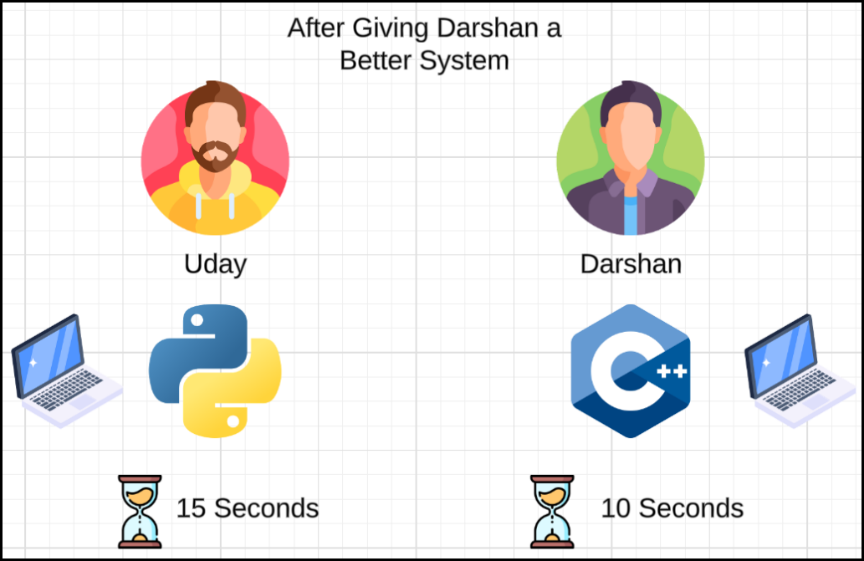
* Imagine **Uday** and **Darshan each have written an algorithm to sort number.**



|  |  |  |
| --- | --- | --- |
| Initial Comparison of Algorithms | | |
| Person | **Uday** | **Darshan** |
| Language | Python | C++ |
| Time Taken | 15 Seconds | 20 Seconds |

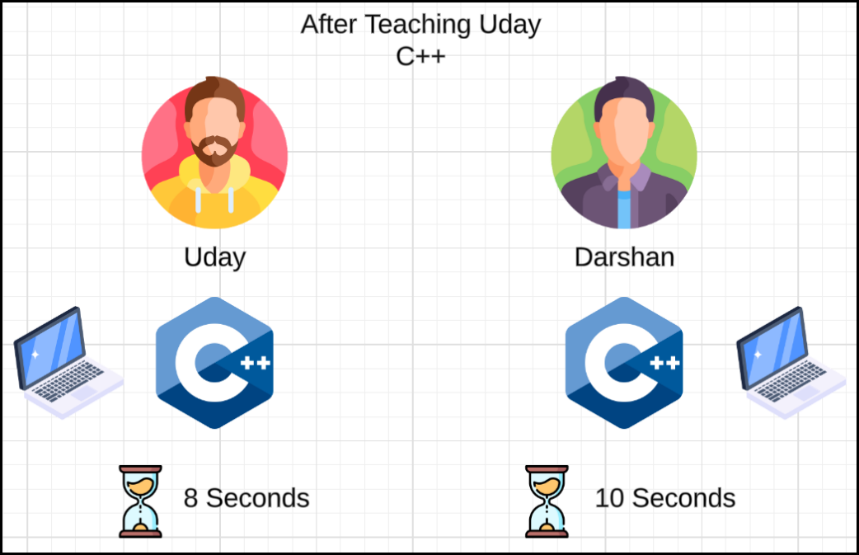
Looks like **Uday** is better, but **their systems differ**...

* After Giving **Darshan** a Better System (Mac)



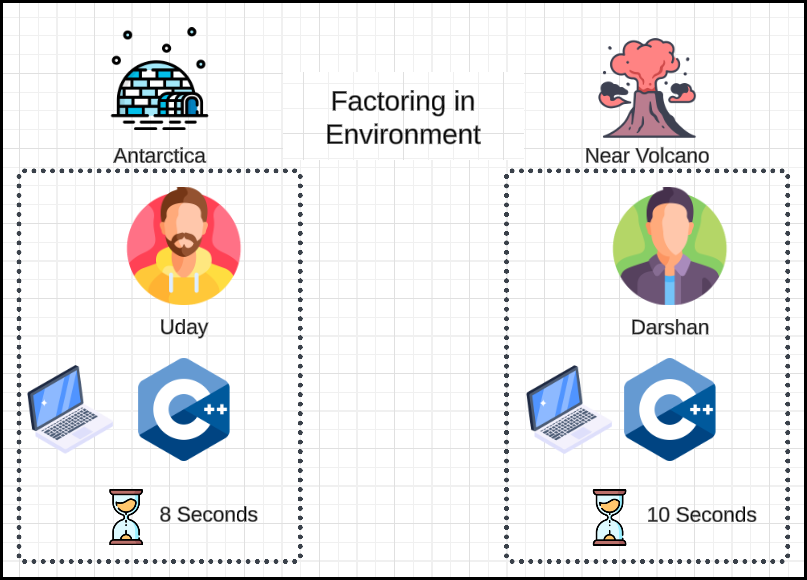
|  |  |  |
| --- | --- | --- |
| After Giving Darshan a MAC | | |
| Person | **Uday** | **Darshan** |
| Language | Python | C++ |
| Time Taken | 15 Seconds | 10 Seconds |

Now Darshan looks better. But **Python vs C++ is unfair**...

* After Teaching Uday C++ (Equal Language)

|  |  |  |
| --- | --- | --- |
| After Teaching Uday C++ | | |
| Person | **Uday** | **Darshan** |
| Language | C++ | C++ |
| Time Taken | 8 Seconds | 10 Seconds |

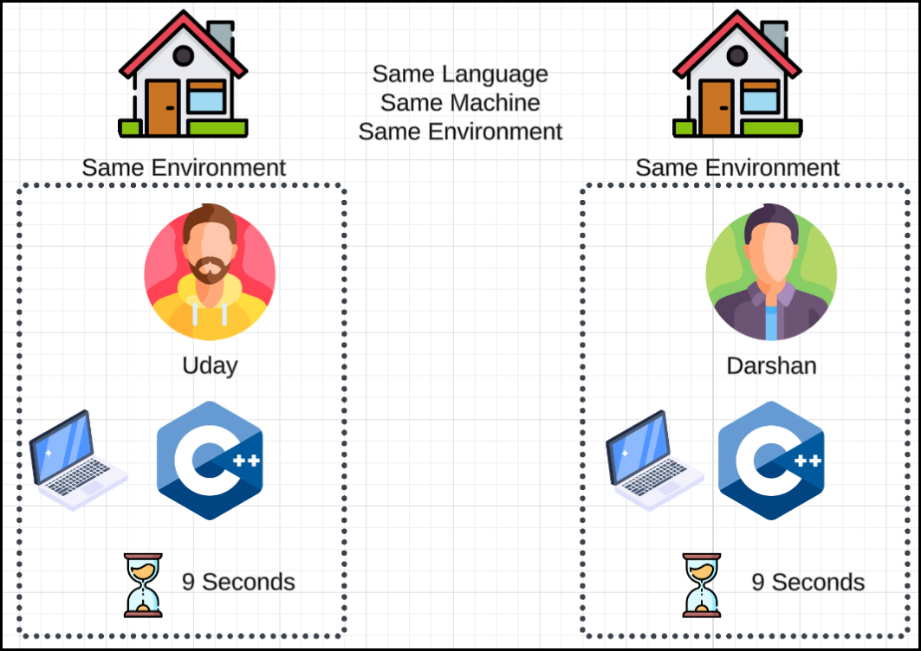
Uday wins... but what about **environment differences**?

* Factoring in Environment (Antarctica vs Volcano)
  + Uday is sitting in Antarctica.
  + Darshan is sitting near a Volcano.

|  |  |  |
| --- | --- | --- |
| Factoring in Environment | | |
| Person | **Uday** | **Darshan** |
| Environment | Antarctica | Volcano |
| Language | C++ | C++ |
| Time Taken | 8 Seconds | 10 Seconds |

Still not fair. **Volcano overheats system**.

* Final: Same Language, Same Machine, Same Env



|  |  |  |
| --- | --- | --- |
| Same Language, Same Machine, Same Environment | | |
| Person | **Uday** | **Darshan** |
| Environment | Home | Home |
| Language | C++ | C++ |
| Time Taken | 9 Seconds | 9 Seconds |

* To understand how efficiently our code runs irrespective of hardware, programming language, or environmental conditions.
* Execution time is not reliable because:
  + Different machines (slow vs. fast CPU).
  + Different languages (e.g., Python is slower than C++).
  + External conditions (e.g., temperature, memory availability).

## Time Complexity

* To compare algorithms fairly:
  + Count number of **operations** or **iterations**.
  + This gives a machine-independent measure of performance.

## Asymptotic Analysis

* It is the study of algorithm performance for large input sizes.
* Helps us ignore unimportant details and focus on growth rate.
* This includes notations like **Big O**, **Theta**, and **Omega** (though focus is mostly on Big O for now).

## Examples and Comparisons

### Compare vs.

* For small n,
  + is faster.
* For large n,
  + becomes faster.
* A graph would show the point where the better algorithm switches.

## Big O Notation

* Definition
  + Describes the upper bound on time (i.e., worst-case performance).
  + Used to classify algorithms by how their run time grows with input size .
* How to Calculate
  1. Count the operations based on input size.
  2. Ignore lower-order terms (e.g., → focus on ).
  3. Ignore constant coefficients (e.g., → just ).
* Example:

for (int i = 0; i < n; i++) {

// do constant work

}

* + - Number of iterations:
    - Work per iteration: constant
    - Time Complexity:

## Examples

### Find the time complexity of

* Step 1: Count the operations based on input size
  + This is already provided:
    - Each term corresponds to work done as input size increases.
* Step 2: Ignore lower-order terms
  + We identify the order of each term:
    - → order = 2
    - → order = 1
    - 6 → order = 0
  + Keep the highest-order term and discard the rest:
* Step 3: Ignore constant coefficients
  + We remove the coefficient 10 (constants do not affect growth rate):

### Find the time complexity of

* Step 1: Count the operations based on input size
  + The number of operations is already represented as:
* Step 2: Ignore lower-order terms
  + We identify the order (growth rate) of each term:
    - → order = 3
    - → order = 2
    - → → order = 1.5
    - → order < 1
  + The highest order term is: . So, we discard the rest and keep only:
* Step 3: Ignore constant coefficients
  + Remove the constant 600 as it does not affect asymptotic growth:

### Find the time complexity of

* Step 1: Count the operations based on input size
  + The number of operations depends on two terms:
* Step 2: Ignore lower-order terms
  + Compare the order of growth:
    - grows faster than for large n, because the logarithmic factor increases.
    - Therefore, is a lower-order term and can be ignored.
* Step 3: Ignore constant coefficients
* Remove the constant 60 to simplify the expression:

### Find the time complexity of

* This can be written as
* Higher order term is .
* Ignore the constant coefficient 500.
* Time Complexity is:

### Find the time complexity of

* Step 1: Count the operations based on input size
  + We are given a function with two variables,
  + and , and the number of operations is:
* Step 2: Ignore lower-order terms
  + Group by variable:
    - Terms with n:
    - Terms with m:
    - Constant term: 6
  + Keep the highest-order term for each variable group:
    - From n: Keep , drop
    - From m: Keep , drop
    - Drop constant term 6
* Step 3: Ignore constant coefficients
  + Drop constants 40 and 59:

### Some examples:

* Step 1: Count the operations based on input size
  + The function consists of two terms:
* Step 2: Ignore lower-order terms
  + In this case, you cannot ignore either term, because:
    - If then dominates.
    - If then dominates.
  + Without knowing the relationship between n and m, both are significant.
  + So, we keep both terms.
* Step 3: Ignore constant coefficients
  + There are no constants here to remove, so we retain both terms as-is.

## Why Do We Neglect Lower-Order Terms in Big O?

* When analysing an algorithm's **time complexity**, we often **remove lower-order terms**. But **why is this justified**?
* Let’s consider:
  + Case 1: Small Input,
    - Now, compute the **contribution of the lower-order term** n:
  + Case 2: Realistic Input Size,
    - ,
  + Now compute the contribution:
* Lower-order terms are negligible when the input size is large.
* Big O focuses on large inputs—where scalability matters most.
* Dropping smaller terms helps simplify and focus on what really dominates performance.

# Issues with Big O Notation

* Big O notation gives a high-level idea of how the number of operations (or time/space required) grows as input size increases.
* It focuses on the order of growth and **ignores constants** and **less significant terms**.

## Big O Hides Constant Factors

* Suppose you have two algorithms:
  + Algorithm A → takes operations
  + Algorithm B → takes operations
* Big O for both is , but clearly, B is better than A.
* Big O **hides** these constant factors (like 100 vs 4), even though they significantly affect performance in practice.

## Big O Doesn’t Help with Small Inputs

* Consider:
  + Algorithm C → **10⁴ × n** operations → O(n)
  + Algorithm D → **5n²** operations → O(n²)
* Example:
  + If n = 10
    - C → = 100,000 operations
    - D → = 500 operations
* Despite being **O(n²)**, **D is far better** for small n.

## Big O Doesn't Show Actual Behaviour

* Big O gives a qualitative understanding (growth rate) but not the quantitative cost (actual number of steps).
* So, two **O(n²)** algorithms may perform very differently due to hidden constants or lower-level optimizations.

## Big O Assumes Large Inputs

* It’s designed for scalability analysis.
* But real-world systems often work with small or mid-sized data, making raw operation count more relevant than asymptotic behaviour.

## When Should You Trust Big O?

* **When input is very large** → Big O is useful for comparing algorithms.
* When input n is known and small → **count actual operations** instead of relying on Big O.

# Space Complexity

* Space Complexity refers to the total amount of memory used by an algorithm as a function of input size, including:
  + Input variables
  + Temporary variables
  + Data structures (arrays, lists, etc.)
  + Function call stack

## Analysing Space Step-by-Step

### Example 1: Basic Variables

|  |
| --- |
| int x = n;  int y = 10;  double d = 10.0;  long l = 123456789; |

* Memory usage:
  + int → 4 bytes each → 2 integers = 8 bytes
  + double → 8 bytes
  + long → 8 bytes
  + Total space used = 4 + 4 + 8 + 8 = 24 bytes
* This is independent of input size 𝑛

### Example 2: Array of Size n

|  |
| --- |
| int x = 4;  int y = 10;  double d = 10.0;  int arr[n]; |

* Each integer takes 4 bytes. So, for 𝑛 integers:
  + bytes
* Other variables (e.g., int x = 10) might take a few more bytes, say 16 total, so:
* Now apply Big O rules:
  + Remove lower-order terms: keep
  + Remove constant coefficient:

### Example 3: 2D Array of Size n×n

int matrix[n][n];

* Each integer takes 4 bytes:
* Apply Big O simplification:

## Space Can Change Over Time

* Realistically, space usage **can increase and decrease** as your program runs:

|  |  |
| --- | --- |
| Time | Memory Used |
| t = 0 | 0 MB |
| t = 1s | 1 GB |
| t = 2s | 2 GB |
| t = 3s | 3 GB |
| t = 4s | 1.5 GB |

* Although the memory usage fluctuates, **space complexity is defined as the peak usage**, i.e.:

# Time Limit Exceeded (TLE)

### What is it?

* When your code takes longer than allowed (e.g., >1 second on operations).

### How to Avoid?

* Estimate time:
* Assume 1 second ≈ operations.
* If your algorithm does more than that for the given input constraints, you'll get TLE.

### Example:

Given Problem:

Where n in the number of elements and A[i] is array element.

* If you have written an algorithm for the above given problem and the algorithm executes operations.
* The problem has an input where
  + Then the number of operations =
  + Too much! Likely TLE.
* Reduce the operation to or better.
* If wrote another algorithm and we reduced the number of operations to
* Now calculate **number of operations** for the input size

### Big O is deceiving

* Imagine that we have written an algorithm which takes **operations**.
* In terms of Big O, we write
* Since the Big O neglects the lower order terms and constant coefficient, for the above problem input where , if we calculate number of operations, then
  + Number of operations: , where

### When Do You Get TLE?

* When Do You Get TLE on Platforms Like LeetCode, HackerRank, and HackerEarth?
* When you code takes more than 1 sec for a test case to pass, then you will get TLE.
* General Rule of Thumb
  + Most online judges assume:
* So, if your algorithm is doing more than that for a given input size, **you'll likely get TLE**.

### What Happens Under the Hood

* A loop with N iterations may perform:
  + Assignment operations
  + Addition, subtraction
  + Multiplication, division
* Each of these operations translates into low-level machine instructions, and each instruction consumes a certain number of CPU clock cycles.
* A 1 GHz processor performs approximately:
* A 4 GHz processor performs approximately:
* Let’s say on average:
  + A simple addition takes
  + A multiplication/division may take more say
* Now, assume:
  + Your processor runs at
  + Each operation takes cycles (worst case)
  + Then:
* So, we assume: