Data Structures and Algorithms
(ES221)

Computational Complexity of Algorithms

Dr. Zubair Ahmad

- **Attendance?**

  - Active Attendance
  - **Dead Bodies.**
  - **Active Minds**
  - Mobiles in hands -> Mark as absent
  - 80% mandatory

Few Important Announcements at the end of Class

Scenario!!

Approach 1 (Slow Way)

Which approach do you think is faster?

How can we measure "fast" in a systematic way?

"Imagine you have a huge list of names (e.g., student records). If I ask you to find a specific name, how would you do it?

Approach 2 (Faster Way): If the list is sorted, start from the middle and eliminate half the list each time.

Scenario!!

"Computers can solve problems in different ways, but not all methods are efficient. As input size grows, some algorithms become painfully slow. Our goal is to understand how an algorithm's execution time grows with input size."

"Would you rather use a method that takes 1 second or 1 hour for the same problem?"

Computational Complexity

Computational complexity indicates how much effort is needed to apply an algorithm or how costly it is.

Computational complexity refers to the study of the efficiency of algorithms in terms of the resources they consume.

Effort/efficiency criteria: **time and space**

Computational Complexity

Asymptotic Notation

Asymptotic notation is used in computer science to describe the efficiency of algorithms, specifically their time and space complexity

It helps to understand how an algorithm performs as the input size grows.

expressing the *main component* of the cost of an algorithm, using idealized units of computational work

Computational Complexity

Asymptotic Notation

Simplify analysis

Compare algorithms

Why is Asymptotic Notation Important?

Focus on growth rates

Asymptotic Notation

Big O Notation (O)

Big O tells us the worst-case scenario (how fast the algorithm will run in the worst case)

the *upper bound* of an algorithm's time or space complexity

It provides a worst-case scenario for the growth rate of the algorithm

O(f(n)) means that the running time or space requirement grows at most as fast as f(n) when n becomes large

Asymptotic Notation

Big O Notation (O)

The statement $T(N) = O(f(N))$ means that the growth rate of $T(N)$ is no more than the growth rate of $f(N)$.

if there exist positive constantscandn0such that
$T(N) \geq c \cdot g(N)$ when **$N \geq n0$**

What it means:

- **T(N)**: This is the running time (or space usage) of the algorithm when the input size is N.
- **O(f(N))**: The algorithm’ s running time is bounded by the function f(N) for large inputs.
- **c**: A constant multiplier that scales the function f(N).
- **n0**: A threshold for N. For values of N greater than or equal to n0, the formula holds true

Asymptotic Notation

Omega Notation (Ω)

how slow the algorithm can be

the *lower bound* of an algorithm's time or space complexity

It provides the best-case scenario

Ω(f(n)) means that the algorithm will take at least as long as f(n) in the best case for large n.

Asymptotic Notation

Omega Notation (Ω)

The formula is:

$T(N)=\Omega(g(N))$

if there exist positive constants **c** and **n0** such that

$T(N) \geq c \cdot g(N)$ when $N \geq n0$

What it means:

- **T(N):** Running time (or space usage) of the algorithm.
- **Ω(g(N))**: The algorithm running time will be **at least** **c * g(N)** for large values of **N**.
- **c:** A constant multiplier that scales the function **g(N).**
- **n0**: The threshold input size where this lower bound applies.

Asymptotic Notation

Theta Notation (Θ)

both the upper and lower bounds of an algorithm's time or space complexity

the algorithm grows at a rate that is bounded both above and below by f(n).

Θ(n): The algorithm has linear growth both in the best and worst case.

Asymptotic Notation

Theta Notation (Θ)

The formula is:
T(N)=Θ(h(N))
if and only if
T(N)=O(h(N)) and **T(N)=Ω(h(N))**

What it means:

- **T(N) = Θ(h(N))**: This means that the algorithm's time complexity is **bounded both above and below** by the function h(N). The running time grows exactly as h(N) for large N.
- **O(h(N))**: The algorithm's running time is **bounded above** by **h(N).**
- **Ω(h(N))**: The algorithm's running time is **bounded below** by **h(N)**

Asymptotic Notation

Little o Notation (o)

an upper bound that is not tight.

an upper bound that the algorithm's growth rate will be strictly smaller than the function.

o(f(n)) means that the time or space complexity grows strictly slower than f(n) for large n.

Asymptotic Notation

Little o Notation (o)

The formula is:

$T(N)=o(p(N))$
if for all constants **c** there exists an **n0** such that
$T(N)<c \cdot p(N)$ when **$N \geq n0$**

What it means:

- **T(N)**: Running time of the algorithm.
- **o(p(N))**: The algorithm's running time grows **strictly slower** than p(N) for large N.
- **c**: A constant multiplier.
- **n0**: The threshold input size beyond which the inequality holds.

Asymptotic Notation

Little ω Notation (ω)

the inverse of little o notation

a lower bound that is strictly greater than the function.

ω(f(n)) means that the algorithm’ s time or space complexity grows strictly faster than f(n) for large n

Asymptotic Notation

Little ω Notation (ω)

A function **$T(N) = \omega(f(N))$**
if for **every positive constant c**, there exists an
n_0 such that:
$T(N) > c \cdot f(N)$ for all **$N \geq n0$**

Key Points:
•This means **T(N**) grows **faster than f(N)** for sufficiently large **N**.
•No constant **c** can make **T(N)** stay **below or equal to c ∗ f(N)** indefinitely.
•It is a **strict** lower bound, unlike **Big Omega (Ω)**, which allows equality.

Time Complexity

Constant time complexity

An algorithm has **O(1) (constant time complexity)** if its execution time does **not** depend on the input size nnn. It always takes a **fixed** amount of time to complete, regardless of how large the input is.

Looking up a student's name in a class register by roll number

"What is the name of the student with roll number 23?

If students' names are stored in a **list (array) by roll number**, finding a student’ s name is an **O(1) operation**

Since the roll number directly maps to the index in an array, we retrieve the name **instantly**, no matter how many students are in the register

Time Complexity

Constant time complexity

The function **getFirstElement()** accesses the first element of the array in **O(1) time**.

No matter how large the array is, accessing arr[0] always takes **constant time**

```
#include <iostream>
using namespace std;

int getFirstElement(int arr[], int size) {
    return arr[0];
}

int main() {
    int numbers[] = {10, 20, 30, 40, 50};
    int size = sizeof(numbers) /
sizeof(numbers[0]);
    cout << "First element: " <<
getFirstElement(numbers, size) << endl;
    return 0;
}
```

Time Complexity

Constant time complexity

```
#include <iostream>
using namespace std;

bool isEven(int n) {
    return (n % 2 == 0); // Single operation, O(1)
}

int main() {
    int num = 42;
    if (isEven(num))
        cout << num << " is even" << endl;
    else
        cout << num << " is odd" << endl;
    return 0;
}
```

Time Complexity

Linear Time Complexity – O(n)

An algorithm has **O(n) (linear time complexity)** if the execution time grows **proportionally** with the input size n

double the input size, the execution time will also **double**.

Time Complexity

Linear Time Complexity – O(n)

Checking Student Attendance in a Class

Suppose you have a class list with **100 students**.

You need to **call each student’ s name one by one** to check attendance.

if the class size grows to **200 students**, the time taken will also **double**.

Time Complexity

Linear Time Complexity – O(n)

The function **printArray()** loops through **each element** in the array

.If size = 5, the loop runs **5 times**.

If size = 1000, the loop runs **1000 times**

```
#include <iostream>
using namespace std;

void printArray(int arr[], int size) {
    for (int i = 0; i < size; i++) { //
Loop runs 'size' times
        cout << arr[i] << " ";
    }
    cout << endl;
}

int main() {
    int numbers[] = {10, 20, 30, 40, 50};
    int size = sizeof(numbers) /
sizeof(numbers[0]); // Calculate array
size

    printArray(numbers, size);
    return 0;
}
```

Time Complexity

Logarithmic Time Complexity – Binary Search

if the number of operations **reduces exponentially** as the input size increases

Time Complexity

Logarithmic Time Complexity – Binary Search

Finding a Video on YouTube's Search Algorithm 🎥

When you search for a video on **YouTube**, the platform **doesn’ t** check every video (**O(n)** search). Instead

It first **narrows down** videos based on your query

It ranks them based on **relevance, popularity, and personalization**

The search space is **continuously reduced**, making it **logarithmic**

Time Complexity

Logarithmic Time Complexity – Binary Search

```
#include <iostream>
using namespace std;

int binarySearch(int arr[], int size, int
target) {
    int left = 0, right = size - 1;

    while (left <= right) {
        int mid = left + (right - left) / 2;
        if (arr[mid] == target)  // Found the
target
            return mid;
        else if (arr[mid] < target)
            left = mid + 1;  // Search the right
half
        else
            right = mid - 1; // Search the left
half
    }

    return -1; // Not found
}
```

```
int main() {
    int numbers[] = {2, 5, 8, 12, 16, 23,
38, 45, 56, 72, 91};
    int size = sizeof(numbers) /
sizeof(numbers[0]);

    int target = 23;
    int result = binarySearch(numbers,
size, target);

    if (result != -1)
        cout << "Element found at index: "
<< result << endl;
    else
        cout << "Element not found!" <<
endl;

    return 0;
}
```

Time Complexity

Quadratic Time Complexity – $O(n^2)$

if the number of operations **grows quadratically** with the input size

If the input size **doubles**, the number of operations **quadruples**

If the input size **triples**, the number of operations **becomes nine times larger**

Time Complexity

Quadratic Time Complexity – O(n²)

Example: Imagine a classroom where every student has to **shake hands** with every other student.

If there are **5 students**, there are **5 × 5 = 25** possible interactions

If there are **10 students**, there are **10 × 10 = 100** interactions.

Thats quadratic growth!

Time Complexity

Quadratic Time Complexity – O(n²)

```
#include <iostream>
using namespace std;

void bubbleSort(int arr[], int n) {
    for (int i = 0; i < n - 1; i++) {   //
Outer loop (n)
        for (int j = 0; j < n - i - 1;
j++) {  // Inner loop (n)
            if (arr[j] > arr[j + 1]) {  //
Swap if out of order
                swap(arr[j], arr[j + 1]);
            }
        }
    }
}
```

```
int main() {
    int arr[] = {5, 2, 9, 1, 5, 6};
    int n = sizeof(arr) / sizeof(arr[0]);

    bubbleSort(arr, n);

    cout << "Sorted Array: ";
    for (int i = 0; i < n; i++)
        cout << arr[i] << " ";

    return 0;
}
```

Time Complexity

Exponential Time Complexity

if the number of operations **doubles** with each additional input element

If the input size **increases by 1**, the time taken **doubles**

If the input **doubles**, the time taken **grows exponentially** (much faster than quadratic $O(n^2)$)

Example:
For **$n = 10$**, operations = **$2^{10} = 1,024$**.
For **$n = 20$**, operations = **$2^{20} = 1,048,576$** 😱.

Exponential algorithms quickly become impractical for large inputs!

Time Complexity

Exponential Time Complexity

Password Cracking (Brute Force Attack) 🔓

A hacker trying **every possible password** needs to check $\mathbf{2^n}$ combinations.

That’ s why long passwords (e.g., 12+ characters) are much harder to crack

Time Complexity

Factorial Time Complexity – O(n!)

if the number of operations **grows factorially** as the input size increases

Factorial Growth Example:
If **n = 5**, operations = **5! = 5 × 4 × 3 × 2 × 1 = 120**

If **n = 10**, operations = **10! = 3,628,800** 😱

If **n = 20**, operations = **20! = 2,432,902,008,176,640,000** (Massive explosion!)

Factorial time complexity is the slowest-growing complexity and quickly becomes infeasible for large inputs.

Time Complexity

Factorial Time Complexity – O(n!)

Assigning Jobs to Workers (Job Scheduling)

If you have **n workers** and **n tasks**, and each worker can do any task,

The number of ways to assign jobs = **n!** (factorial complexity)

Organizing People in a Photo Shoot

If you have **n people** and must arrange them in a line,

The number of different ways to arrange them is **O(n!)**.

Time Complexity

Factorial Time Complexity – O(n!)

```
#include <iostream>
#include <chrono>

using namespace std;
using namespace chrono;

long long factorial(int n) {
    long long result = 1;
    for (int i = 1; i <= n; ++i) {
        result *= i;
    }
    return result;
}
```

```
int main() {
    int num;

    cout << "Enter a number: ";
    cin >> num;

    auto start = high_resolution_clock::now();
// Start time

    long long result = factorial(num);

    auto end = high_resolution_clock::now();  //
End time

    auto duration =
duration_cast<microseconds>(end - start);  //
Calculate duration

    cout << "Factorial of " << num << " is " <<
result << endl;
    cout << "Time taken: " << duration.count()
<< " microseconds" << endl;

    return 0;
}
```

Important!!!!!

- First Quiz and Assignment – May be in this week/next week

- Course meeting – each group = 5 students – Will share the time later

Questions?

zahmaad.github.io