
# SCS1308 - Foundations of Algorithm

## Tutorial - 01
## Time Complexity

# What is an Algorithm?

An **algorithm** is a **step-by-step procedure** to solve a problem or perform a task.

- Has clear, finite steps
- Takes input and produces output
- Each step is unambiguous
- Must end after a finite number of steps

**Example:**

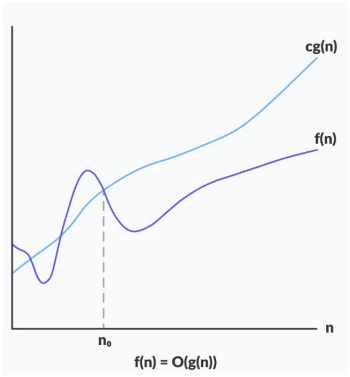
Steps to add two numbers

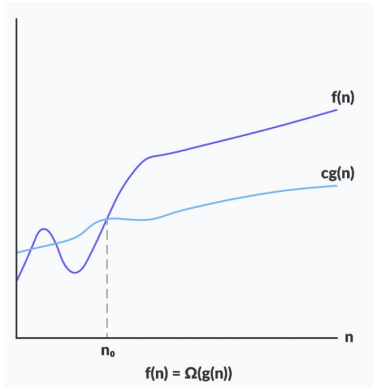
1. Start
2. Read A and B
3. Compute Sum = A + B
4. Display Sum
5. Stop

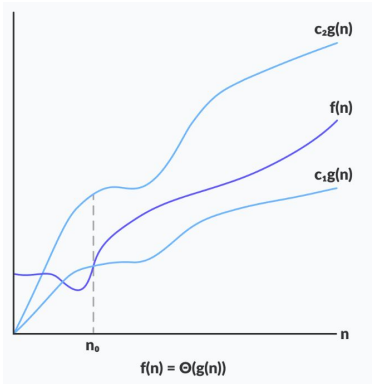
# What is an Algorithm?

Algorithm Efficiency

- ➔ Efficiency depends on time and space complexity.
- ➔ Types of analysis
  - Worst case
  - Best case
  - Average case
- ➔ Comparisons often focus on growth rates (Big-O, Omega, Theta).



# Analyzing Iterative Algorithms

➔ Loops - Depends on the number of iterations

```
for (int i = 0; i < n; i++) {
   // Executes n times
}
```

➔ Nested loops - Multiply the number of iterations of each loop

```
for (int i = 0; i < n; i++) {
   for (int j = 0; j < n; j++) {
      // Executes n * (n) times
   }
}
```

# Analyzing Iterative Algorithms

➔ Consecutive statements - If multiple independent statements execute sequentially, their complexities are added.

```
for (int i = 0; i < n; i++) {
   // O(n)
}
for (int j = 0; j < m; j++) {
   // O(m)
}
```

➔ If-then-else-statement - Complexity depends on the branch that is executed most frequently or the most expensive branch.

```
if (n > 1000) {
   for (int i = 0; i < n; i++) {  // O(n)
      ...
   }
} else {
   for (int j = 0; j < log(n); j++) {  // O(log n)
      ...
   }
}
```

# Analyzing Iterative Algorithms

➔ Logarithmic Complexity - Algorithms exhibit logarithmic complexity when the input size is divided in each iteration.

```
while (n > 1) {

   n = n / 2;  // Input size divided each time

}
```

1. Initial value: n
2. Each iteration: n is divided by 2 → n = n / 2
3. After k iterations:
   $n/2^k=1$
4. Solve for k:
   $2^k=n$
5. Take log base 2:
   $k=\log_2(n)$

Therefore, the loop runs $\log_2(n)$ times → O(log n)

# Growth Rate Classes

We have seen that when we analyze functions asymptotically

- ➔ Constant: Θ(k), for example Θ(1)
- ➔ Linear: Θ(n)
- ➔ Logarithmic: Θ( $\log_k$ n)
- ➔ nlogn: Θ(n$\log_k$ n)
- ➔ Quadratic: Θ($n^2$)
- ➔ Polynomial: Θ($n^k$)
- ➔ Exponential: Θ($k^n$)

Only the leading term is important.

Constants don't make a significant difference.

The following inequalities hold asymptotically

$$c < \log n < \log 2^n < \sqrt{n} < n < n \log n < n^{(1.1)} < n^2 < n^3 < n^4 < 2^n$$

In other words, an algorithm that is Θ(nlog(n)) is more efficient than an algorithm that is Θ($n^3$).

| $\log n$ | $n$ | $n \log n$ | $n^2$ | $n^3$ | $2^n$ |
|---|---|---|---|---|---|
| 0 | 1 | 0 | 1 | 1 | 2 |
| 0.6931 | 2 | 1.39 | 4 | 8 | 4 |
| 1.099 | 3 | 3.30 | 9 | 27 | 8 |
| 1.386 | 4 | 5.55 | 16 | 64 | 16 |
| 1.609 | 5 | 8.05 | 25 | 125 | 32 |
| 1.792 | 6 | 10.75 | 36 | 216 | 64 |
| 1.946 | 7 | 13.62 | 49 | 343 | 128 |
| 2.079 | 8 | 16.64 | 64 | 512 | 256 |
| 2.197 | 9 | 19.78 | 81 | 729 | 512 |
| 2.303 | 10 | 23.03 | 100 | 1000 | 1024 |
| 2.398 | 11 | 26.38 | 121 | 1331 | 2048 |
| 2.485 | 12 | 29.82 | 144 | 1728 | 4096 |
| 2.565 | 13 | 33.34 | 169 | 2197 | 8192 |
| 2.639 | 14 | 36.95 | 196 | 2744 | 16384 |
| 2.708 | 15 | 40.62 | 225 | 3375 | 32768 |
| 2.773 | 16 | 44.36 | 256 | 4096 | 65536 |
| 2.833 | 17 | 48.16 | 289 | 4913 | 131072 |
| 2.890 | 18 | 52.03 | 324 | 5832 | 262144 |
| $\log \log m$ | $\log m$ | | | | $m$ |

| $n$ | $100n$ | $n^2$ | $11n^2$ | $n^3$ | $2^n$ |
|---|---|---|---|---|---|
| 1 | 100 | 1 | 11 | 1 | 2 |
| 2 | 200 | 4 | 44 | 8 | 4 |
| 3 | 300 | 9 | 99 | 27 | 8 |
| 4 | 400 | 16 | 176 | 64 | 16 |
| 5 | 500 | 25 | 275 | 125 | 32 |
| 6 | 600 | 36 | 396 | 216 | 64 |
| 7 | 700 | 49 | 539 | 343 | 128 |
| 8 | 800 | 64 | 704 | 512 | 256 |
| 9 | 900 | 81 | 891 | 729 | 512 |
| 10 | 1000 | 100 | 1100 | 1000 | 1024 |
| 11 | 1100 | 121 | 1331 | 1331 | 2048 |
| 12 | 1200 | 144 | 1584 | 1728 | 4096 |
| 13 | 1300 | 169 | 1859 | 2197 | 8192 |
| 14 | 1400 | 196 | 2156 | 2744 | 16384 |
| 15 | 1500 | 225 | 2475 | 3375 | 32768 |
| 16 | 1600 | 256 | 2816 | 4096 | 65536 |
| 17 | 1700 | 289 | 3179 | 4913 | 131072 |
| 18 | 1800 | 324 | 3564 | 5832 | 262144 |
| 19 | 1900 | 361 | 3971 | 6859 | 524288 |

| $n$ | $n^2$ | $n^2 - n$ | $n^2 + 99$ | $n^3$ | $n^3 + 234$ |
|---|---|---|---|---|---|
| 2 | 4 | 2 | 103 | 8 | 242 |
| 6 | 36 | 30 | 135 | 216 | 450 |
| 10 | 100 | 90 | 199 | 1000 | 1234 |
| 14 | 196 | 182 | 295 | 2744 | 2978 |
| 18 | 324 | 306 | 423 | 5832 | 6066 |
| 22 | 484 | 462 | 583 | 10648 | 10882 |
| 26 | 676 | 650 | 775 | 17576 | 17810 |
| 30 | 900 | 870 | 999 | 27000 | 27234 |
| 34 | 1156 | 1122 | 1255 | 39304 | 39538 |
| 38 | 1444 | 1406 | 1543 | 54872 | 55106 |
| 42 | 1764 | 1722 | 1863 | 74088 | 74322 |
| 46 | 2116 | 2070 | 2215 | 97336 | 97570 |
| 50 | 2500 | 2450 | 2599 | 125000 | 125234 |
| 54 | 2916 | 2862 | 3015 | 157464 | 157698 |
| 58 | 3364 | 3306 | 3463 | 195112 | 195346 |
| 62 | 3844 | 3782 | 3943 | 238328 | 238562 |
| 66 | 4356 | 4290 | 4455 | 287496 | 287730 |
| 70 | 4900 | 4830 | 4999 | 343000 | 343234 |
| 74 | 5476 | 5402 | 5575 | 405224 | 405458 |

## Barometer Method

- A technique used to analyze the time complexity of an algorithm by counting the most significant operations that dominate the total running time.
- Choose a “barometer” operation — the one that repeats most and contributes most to the total cost.
  *(e.g., comparisons, assignments, additions)*
- Count how many times that operation executes as the input size (n) grows.

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

    sum = sum + i;   // barometer: addition

}
```

Barometer operation: `sum = sum + i`

Executes: n times

T(n) = n → O(n)

# Thank you