Made By:
Kenan Gazwan

Complexity Analysis

ICS202-Summary

King Fahd University of Petroleum and Minerals

Telegram: @KenanGazwan

(Complexity Analysis)

✓ Efficiency:

The efficiency of any algorithm can be measured according to the:

- Time Efficiency, and
- Space Efficiency

To evaluate the efficiency of an algorithm, we use logical units that express a relationship such as:

- (n) → the size of an array or file
- T → the amount of time required to process the data

Hence, the relationship between $[n]$ and $[T]$ is $T(n)$

✓ Basic Operations (Counted as One)

- Arithmetic operations: (*), (/), (%), (+), (-)
- Boolean operations: &&, ||, !
- Assignment statements
- Reading of primitive types
- Writing of primitive types
- Simple conditionals: if (x<12)
- Method calls (Note: Execution time of a method itself may not be constant)
- Return statement
- Memory access (such array indexing)
- ++, +=, or *= consist of two basic operations

```
Example(){
sum = 0;     //1
for(k = 1; k <= 5; k++){//1 + 6 + 5*2
      sum = sum + 5;      //5*2
}
return k;   //1
}
Total = 1 + 1 + 6 + 10 + 10 + 1 = 29
T(n) = 29, independent of n, T(n) = c
```

✓ Asymptotic Complexity

Since counting the exact number of basic operations takes a lot of time and sometimes impossible, we can always focus on **Asymptotic Complexity.**

Asymptotic Complexity is used by disregarding lower order terms of a function to express the efficiency of an algorithm.

- ☒ **Example: For the following function →** $\boldsymbol{T(n) = n^2 + 100n + 100}$
  We need just the dominant term $[n^2]$, and the rest of terms are disregarded.
  Hence, $\boldsymbol{T(n) \approx n^2}$

Why are the lower terms disregarded?
This is because for very large values of n, the lower terms are insignificant:

	Dominant term	Total	Lower terms	Contribution of lower terms
N	n^2	$n^2 + 4n + 4$	$4n + 4$	Lower Terms Contribution in %
10	100	144	44	30.55 %
100	10,000	10,404	404	3.88 %
1,000	1,000,000	1,004,004	4004	0.39 %
10,000	100,000,000	100,040,004	40,004	0.039 %

Large Value! →

← Insignificant Contribution!

$$\boldsymbol{1 < log n < \sqrt{n} < n < n \, log n < n^2 < n^3 < \cdots < 2^n < 3^n < \cdots < n^n}$$

✓ **Big-O Notation** (Upper Bounds of Functions)

It specifies asymptotic complexity that estimates **the rate of function growth.**
Definition: $f(n) is\ O(g(n))$ if there exists positive numbers c and N such that:

$$f\ (n) \leq cg(n)\ for\ all\ \ n \geq N$$

Types Of Time Functions: -

$O(1)$	Constant
$O(logn)$	Logarithmic
$O(n)$	Linear
$O(n^2)$	Quadratic
$O(n^3)$	Cubic
$O(2^n)$	Exponential

Properties of Big-*O* Notation: -

If $f(n)$ is $O(g(n))$ and $g(n)$ is $O(h(n))$, then $f(n)$ is $O(h(n))$
If $f(n)$ is $O(h(n))$ and $g(n)$ is $O(h(n))$, then $(f(n) + g(n))$ is $O(h(n))$
The function an^k is $O(n^k)$,where a is a constant
The function n^k is $O(n^{k+\epsilon})$ for any $\epsilon \geq 0$
If $f(n) = cg(n)$, then $f(n)$ is $O(g(n))$, where c is a constant
If $f(n) = g(n) + h(n)$, then $f(n)$ is $O(\max\{g(n), h(n)\})$
If $f(n) = g(n)*h(n)$, then $f(n)$ is $O(g(n)*h(n))$
The function $\log_a n$ is $O(\log_b n)$ for any positive numbers $a > 1$ and $b > 1$
$\log_a n$ is $O(\lg n)$ for any positive number $a > 1$, where $\lg n$ is $\log_2 n$

Warnings about *O*-Notation: -

- Big-O notation cannot compare algorithms in the same complexity class.
- Big-O notation only gives sensible comparisons of algorithms in different complexity classes when n is large.

✓ Big-Ω [Big-Omega] Notation (Lower Bounds of Functions)

Definition: $f(n) is\ O(g(n))$ if there exists positive numbers c and N such that:

$$f(n) \geq cg(n)\ for\ all\ \ n \geq N$$

✓ The difference between this definition and the definition of big-O notation is **the direction of inequality**.
One definition can be turned into the other by replacing "≥" with "≤"

✓ There is an **interconnection** between O and Ω notations expressed by the equivalence:

$$f(n)\ is\ \Omega(g(n))\ if\ and\ only\ if\ \ g(n)\ is\ O(f(n))$$

✓ Big-Θ [Big-Theta] Notation (Average)

Definition: f(n) is Θ(g(n)) if there exist positive numbers c_1, c_2, and N such that
$c1\ g(n) \leq f(n) \leq c2\ g(n)\ for\ all\ n \geq N$

✓ Finding c and n given a function: -

Ex: $\boldsymbol{f(n) \leq cg(n)}$

1. Divide both sides of the inequality by $g(n)$ → $\boldsymbol{\frac{f(n)}{g(n)} \leq c}$
2. Choose any value you want for $\boldsymbol{n}$, and substitute it in the inequality.
3. Evaluating the inequality, the value of **c** must be found → $\boldsymbol{resultNumber \leq c}$
4. Finally,
   - $n\ \ =$ The value chosen in step2.
   - $c = resultNumber$ <u>or</u> $c\ \ >\ \ resultNumber$

✓ Best, Worst, and Average Case Complexities: -

Best-Case: The smallest number of operations carried out by the algorithm for given input

Worst-Case: The largest number of operations carried out by the algorithm for a given input

Average-Case: The number of operations carried out by the algorithm on average for all inputs.

$$\sum\nolimits_{\boldsymbol{for\ each\ input\ i}} (\boldsymbol{Probability\ of\ input\ i\ *\ Cost\ of\ input\ i})$$

✓ We are usually interested in the **worst-case** complexity.

✓ **Finding Asymptotic Complexity** (Some Common Loops): -

- `for (int i = 1; i <= n; i++)` ➔ `O(n)`

$$\sum_{i=1}^{n} 1 = n \approx O(n)$$

- `for (int i = 0; i <= n; i=i+2)` ➔ `O(n)`

Notice: we cannot use i *as an iterator because it does not increase by 1*, so we must find another iterator that increases by 1.

i: 0, 2, 4, …., 2R (R = number of iterations) ➔ n = 2R → R = n/2

$$\sum_{S=0}^{R} 1 = (R - S) + 1 = \left(\frac{n}{2} - 0\right) + 1 \approx O(n)$$

The General Formula for This Case:

- `for (int i = k; i <= n; i = i + m)`

Finding an Iterator that increases by 1:

i: k, k + m, k + 2m, …, k + mR (R = number of iterations, k = starting value) ➔ n = k + mR → R = (n – k) / m

$$\sum_{S=0}^{R} 1 = (R - S) + 1 = \left(\frac{n-k}{m} - 0\right) + 1 \approx O(n)$$

- `for (int i = 1; i <= n; i=i*2)` ➔ `O(Log(n))`

Notice: we cannot use i as an iterator because it does not increase by 1, so we must find another iterator that increases by 1.

i: 2^0, 2^1, 2^2, 2^3, 2^4, …, 2^R (R = number of iterations) ➔ n = 2^R → *R = logn*

$$\sum_{S=0}^{R} 1 = (R - S) + 1 = (logn - 0) + 1 \approx O(\log(n))$$

- ```
  for (int i = 1; i <= n; i++) {
    for (int j = 1; j <= n; j++) {
        STATEMENT;
    }
  }
  ```

$$\sum_{i=1}^{n}\sum_{j=1}^{n} 1 = \sum_{i=1}^{n} n = n\sum_{i=1}^{n} 1 = n^2 = \approx O(\mathrm{n}^2)$$

- ```
  for (int i = 1; i <= n; i++) {
    for (int j = 1; j <= i; j++) {
        STATEMENT;
    }
  }
  ```

$$\sum_{i=1}^{n}\sum_{j=1}^{i} 1 = \sum_{i=1}^{n} i = \frac{n(n+1)}{2} = \approx O(\mathrm{n}^2)$$

Summary: -

- *for (int i = 1; i <= n; i++)* ➔ *O(n)*
- *for (int i = 0; i <= n; i=i+2)* ➔ *O(n)*
- *for (int i = k; i < n; i=i+2)* ➔ *O(n)*
- *for (int i = n; i > 1; i--)* ➔ *O(n)*
- *for (int i = 1; i < n; i=i*2)* ➔ *O(Log(n))*
- *for (int i = n; i > 1; i=i/2)* ➔ *O(Log(n))*