02 - Computational Complexity Analysis

[KOMS119602] & [KOMS120403]

Design and Analysis of Algorithm (2021/2022)

Dewi Sintiari

Prodi S1 Ilmu Komputer
Universitas Pendidikan Ganesha

Week 14-18 February 2022

Euclidean algorithm to compute gcd

From last week...

Computing gcd:

- Input: two integers a and b
- Output: the greatest common divisor of m and n

Algorithm 1 Naive gcd algorithm of two integers

```
procedure GCD(a, b)
    r = 1
    x = min(a, b)
    for i = 1 to x do
        if a mod i == 0 and b mod i == 0 then r = i
        end if
    end for
end procedure
```

Complexity? homework!

Euclidean algorithm to compute gcd

From last week...

Example

Using the Euclidean algorithm, find the gcd of 210 and 45.

Solution:

$$
\begin{aligned}
210 &= 4 \cdot 45 + 30 \\
45 &= 1 \cdot 30 + 15 \\
30 &= 2 \cdot 15 + 0
\end{aligned}
$$

So $gcd(210, 45) = 15$

Euclidean algorithm to compute gcd

From last week...

Algorithm 2 Euclidean algorithm

```
procedure EuclidGcd(a, b)
    while b ≠ 0 do
        r = a mod b
        a = b
        b = r
    end while
    return a
end procedure
```

Why does it terminate?
Complexity? homework!

Computational complexity model

Can you recall what is complexity of an algorithm, and why should we study it?

Computational complexity model

A part of *algorithm analysis* is computing the *computational complexity* of an algorithm.

The computational complexity or simply complexity of an algorithm is the amount of resources (*time* and *memory*) required to run it.

- Time efficiency: how fast an algorithm is executed
- Space efficiency: how much memory needed to execute an algorithm

How do we compute the complexity of an algorithm?

Computational complexity model

Example

Let a supercomputer executes an algorithm A, and a PC executes an algorithm B. Both computers have to sort an array of 1 million elements. The supercomputer can execute 100 million instructions in one second, while the PC is only able to execute 1 million instructions in one second.

Algorithm A needs $2n^2$ instructions to sort n elements, and algorithm B needs $50n \log n$ instructions. Compute the amount of time to sort 1 million elements in each computer!

Example

Let a supercomputer executes an algorithm A, and a PC executes an algorithm B. Both computers have to sort an array of 1 million elements. The supercomputer can execute 100 million instructions in one second, while the PC is only able to execute 1 million instructions in one second.

Algorithm A needs $2n^2$ instructions to sort n elements, and algorithm B needs $50n \log n$ instructions. Compute the amount of time to sort 1 million elements in each computer!

Solution:

- Supercomputer: $\frac{2 \cdot (10^6)^2 \text{ instructions}}{10^8 \text{ instructions / sec}} = 20000 \text{ sec} \approx 5.56 \text{ hours}$
- PC: $\frac{50 \cdot 10^6 \log 10^6 \text{ instructions}}{10^6 \text{ instructions / sec}} \approx 1000 sec \approx 16.67 \text{ minutes}$

Remark. So, the number of executions matters!

What affects computational complexity?

Time (and space) complexity depends on lots of things like *hardware*, *OS*, *processors*, *programming language* and *compiler*, etc. But we don't consider any of these factors when analyzing the algorithm.

Remarks:

- Our focus on this subject will be on time complexity.
- We assume that our machine uses only one processor (i.e. *generic one-processor*).
- Time complexity is computed based on the number of operations/instructions
- The running time of an algorithm increases (or remains constant in case of constant running time) as the input size (n) increases.

Computational complexity model

Algorithm 3 Average of an array of integers

```
1: procedure AVERAGE(A[1..n])
2:     sum ← 0
3:     for i = 1 to n do
4:         sum ← sum + A[i]
5:     end for
6:     avg ← sum/n
7: end procedure
```

The number of operations:

- Assignment: lines 2, 4, 6; with $1 + n + 1 = n + 2$ operations
- Summation: line 4, with n operations
- Division: line 6, with 1 operation

Complexity: $T(n) = (n + 2) + n = 2n + 2$ operations.

Computational complexity model

Three measurements of resource usage:

- **Worst-case** ($T_{\max}(n)$): it measures the resources (e.g. running time, memory) that an algorithm requires in the worst case i.e. most difficult case, given an input of arbitrary size n (usually denoted in asymptotic notation).
- **Best-case** ($T_{\min}(n)$): describe an algorithm's behavior under optimal conditions.
- **Average-case** ($T_{\text{avg}}(n)$): the amount of computational time used by the algorithm, averaged over all possible inputs.

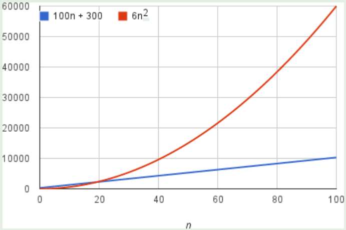
Asymptotic notation and order of magnitude

- The running time of an algorithm is measured as a *function of the size of its input*.
- Rate of growth of the running time measures how fast a function grows with the input size. Asymptotically means the function matters *only for large values of n*.
- The order of magnitude function describes the part of the function that increases the fastest as the value of n increases.

Asymptotic notation and order of magnitude

Example

Suppose that an algorithm, running on an input of size n, takes $6n^2 + 100n + 300$.



We only keep the most significant term. We say that the function $6n^2$ has a higher order of magnitude than $100n + 300$.

Big-$\mathcal{O}$ notation (asymptotic upper-bound)

Worst-case complexity measures the resources an algorithm needs in the *worst-case*. It gives an upper bound on the resources required by the algorithm.

Why learn worst-case complexity?

- provides information about the maximum resource requirements
- naturally, it often happens in a system

Big-$\mathcal{O}$ notation (asymptotic upper-bound)

Big-O ($\mathcal{O}(\cdot)$) notation: a mathematical notation that describes the limiting behavior of a function when the argument tends towards a particular value or infinity.

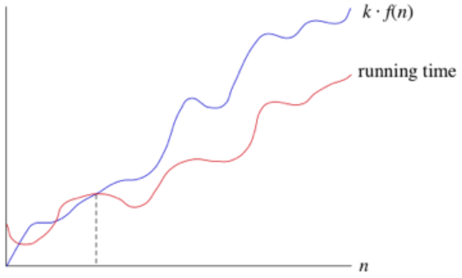
Definition

$g(n) \in \mathcal{O}(f(n))$ if $\exists\ k > 0$ and n_0 s.t. $g(n) \leq k \cdot f(n),\ \ \forall n \geq n_0$.

Big-$\mathcal{O}$ notation (asymptotic upper-bound)

Definition

$g(n) \in \mathcal{O}(f(n))$ if $\exists\ k > 0$ and n_0 s.t. $g(n) \leq k \cdot f(n), \ \ \forall n \geq n_0$.



Big-$\mathcal{O}$ notation (linear and polynomial functions)

Example

Show that $g(n) = 5n + 3$ is in $\mathcal{O}(n)$.

Big-$\mathcal{O}$ notation (linear and polynomial functions)

Example

Show that $g(n) = 5n + 3$ is in $\mathcal{O}(n)$.

Solution:

Note that $5n + 3 \leq 5n + 3n = 8n$ for all $n \geq 1$. In this case, $k = 8$ and $n_0 = 1$. So, $g(n) \in \mathcal{O}(n)$.

Example

Show that $g(n) = 3n^2 - 5n + 6$ is in $\mathcal{O}(n^2)$.

Big-$\mathcal{O}$ notation (linear and polynomial functions)

Example

Show that $g(n) = 3n^2 - 5n + 6$ is in $\mathcal{O}(n^2)$.

Solution:

Note that $3n^2 - 5n + 6 \leq 3n^2 + 0 + 6n^2 = 9n^2$ for all $n \geq 1$. In this case, $k = 9$ and $n_0 = 1$. So, $g(n) \in \mathcal{O}(n^2)$.

Big-$\mathcal{O}$ notation (linear and polynomial functions)

We denote by $T(n)$ a function of time complexity.

Theorem (Big-O of a polynomial complexity)

If $T(n) = a_m n^m + a_{m-1} n^{m-1} + \cdots + a_1 n + a_0$ is a polynomial of order m, then $T(n) \in \mathcal{O}(n^m)$.

Theorem (Arithmetic operations on Big-O)

Let $T_1(n) \in \mathcal{O}(f(n))$ and $T_2(n) \in \mathcal{O}(g(n))$, then:

1. $T_1(n) + T_2(n) \in \mathcal{O}(f(n)) + \mathcal{O}(g(n)) \in \mathcal{O}(\max(f(n), g(n)))$
2. $T_1(n) T_2(n) \in \mathcal{O}(f(n))\mathcal{O}(g(n)) \in \mathcal{O}(f(n)g(n))$
3. $\mathcal{O}(cf(n)) \in \mathcal{O}(f(n))$, *where c is a constant*
4. $f(n) \in \mathcal{O}(f(n))$

Proof: homework!

Big-$\mathcal{O}$ notation (linear and polynomial functions)

Example (Arithmetic operations on Big-O)

1. Let $T_1(n) \in \mathcal{O}(n)$ and $T_2(n) \in \mathcal{O}(n^2)$, then:

$$T_1(n) + T_2(n) \in \mathcal{O}(\max(n, n^2)) \in \mathcal{O}(n^2)$$

2. Let $T_1(n) \in \mathcal{O}(n)$ and $T_2(n) \in \mathcal{O}(n^2)$, then:

$$T_1(n)T_2(n) \in \mathcal{O}(n \cdot n^2) = \mathcal{O}(n^3)$$

3. $\mathcal{O}(5n^2) \in \mathcal{O}(n^2)$
4. $n^2 \in \mathcal{O}(n^2)$

Big-$\mathcal{O}$ notation (logarithmic function)

Review logarithms and exponents

$$\log_b a = c \Leftrightarrow b^c = a$$

- $a > 0$ is the power
- $b > 0$ is the base
- c is the exponent

Remark. If the base $b = 2$, then it is called binary logarithm. The base is often omitted.

Big-$\mathcal{O}$ notation (logarithmic function)

In Computer Science, we usually use base-two logarithm complexity by default. Why?

Big-$\mathcal{O}$ notation (logarithmic function)

In Computer Science, we usually use base-two logarithm complexity by default. Why?

- It is common to work with binary numbers or divide input data in half
- In Big-O notation (upper bound growth), all logarithms are *asymptotically equivalent* (the only difference is there multiplicative constant factor)
- So, we do not specify the base, and only write it as $\mathcal{O}(\log n)$

Big-$\mathcal{O}$ notation (logarithmic function)

Some properties of logarithmic function

- $\log_b 1 = 0$ for any $b \geq 0$
- **Change of bases:** $\log_b a = \frac{\log_p a}{\log_p b}$
- **Addition:** $\log_p m + \log_p n = \log_p mn$
- **Subtraction:** $\log_p m - \log_p n = \log_p \frac{m}{n}$
- **Power:** $\log_p a^x = x \cdot \log_p a$
- **Inverse:** $\log_p \frac{1}{a} = -log_p a$
- Many others...

Example

Show that $g(n) = (n+3)\log(n^2+1) + 2n^2$ is in $\mathcal{O}(n^2)$

Big-$\mathcal{O}$ notation (logarithmic function)

Example

Show that $g(n) = (n+3)\log(n^2+1) + 2n^2$ is in $\mathcal{O}(n^2)$

Solution:

Note that:
$\log(n^2+1) \leq \log(2n^2) = \log 2 + \log n^2 \leq 2\log n^2 = 4\log n$.
So, $\log(n^2+1) \in \mathcal{O}(\log n)$.

Since $n+3 \in \mathcal{O}(n)$, then
$(n+3)\log(n^2+1) \in \mathcal{O}(n) \cdot \mathcal{O}(\log n) \in \mathcal{O}(n\log n)$.

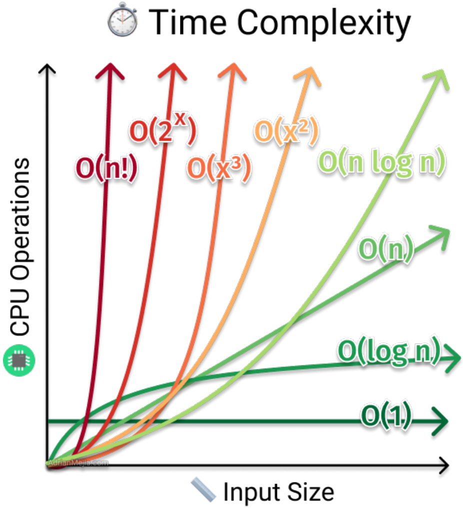
Since $2n^2 \in \mathcal{O}(n^2)$, and $\max(n\log n, n^2) = n^2$, then $g(n) \in \mathcal{O}(n^2)$.

The classification of algorithms based on the worst-time complexity

Complexity	Class
$\mathcal{O}(1)$	constant
$\mathcal{O}(\log n)$	logarithmic
$\mathcal{O}(n)$	linear
$\mathcal{O}(n \log n)$	quasi-logarithmic
$\mathcal{O}(n^2)$	square
$\mathcal{O}(n^3)$	cubic
$\mathcal{O}(n^k),\ k \geq 2$	polynomial
$\mathcal{O}(2^n)$	exponential
$\mathcal{O}(n!)$	factorial

$$\underbrace{\mathcal{O}(1) < \mathcal{O}(\log n) < \mathcal{O}(n) < \mathcal{O}(n \log n) < \mathcal{O}(n^2) < \mathcal{O}(n^3) < \cdots <}_{\text{polynomial algorithms}} \underbrace{\mathcal{O}(2^n) < \mathcal{O}(n!)}_{\text{exponential algorithms}}$$

Big-$\mathcal{O}$ notation (classification of algorithms)



Big-$\mathcal{O}$ notation (determining asymptotic complexity)

1. **Assignment of values** (*comparison, arithmetic operations, read, write*) needs $\mathcal{O}(1)$
2. **Accessing** an element of an array, or selecting a field from a record needs $\mathcal{O}(1)$

Example

- $\text{read}(x) \to \mathcal{O}(1)$
- $x : x + a[k] \to \mathcal{O}(1)$
- $\text{print}(x) \to \mathcal{O}(1)$

Big-$\mathcal{O}$ notation (determining asymptotic complexity)

3. **If-Else condition:** IF C THEN A1 ELSE A2 needs time: $T_C + \max(T_{O1}, T_{O2})$

Example

```
read(x)
if x mod 2 = 0 then
    x := x + 1
    print("Even")
else
    print("Odd")
end if
```

Asymptotic TC: $\mathcal{O}(1) + \mathcal{O}(1) + \max(\mathcal{O}(1) + \mathcal{O}(1), \mathcal{O}(1)) \in \mathcal{O}(1)$

4. **For loop:** the time complexity is the number of iterations multiplied with the time complexity of the *body loop* (i.e. *loop statements*)

Example (Single for loop)

```
for i = 1 to n do
    sum:= sum + a[1]
end for
```

Asymptotic TC: $n \cdot \mathcal{O}(1) = \mathcal{O}(n)$

Big-$\mathcal{O}$ notation (determining asymptotic complexity)

Example (Two nested for loops with one instruction)

---

for $i = 1$ to n **do**
    for $j = 1$ to n **do**
        $a[i,j] := i + j$
    end for
end for

Asymptotic TC: $n \cdot \mathcal{O}(n) = \mathcal{O}(n^2)$

Big-$\mathcal{O}$ notation (determining asymptotic complexity)

Example (Two nested for loops with two instructions)

```
for i = 1 to n do
    for j = 1 to i do
        a := a + 1
        b := b − 1
    end for
end for
```

The outer loop is executed n times, and the inner loop is executed i times for each j. The number of iterations: $1 + 2 + \cdots + n = \frac{n(n+1)}{2} \in \mathcal{O}(n^2)$.

The body loop needs $\mathcal{O}(1)$-time.

Asymptotic time complexity: $\mathcal{O}(n^2)$

Big-$\mathcal{O}$ notation (determining asymptotic complexity)

5. **While loop:** WHILE C DO A; and REPEAT A UNTIL C.
   Time complexity $=$ # iterations $\times$ T_{body}

Example (Single loop with $n-1$ iterations)

```
i := 2
while i ≤ n do
    sum:= sum + a[i]
    i := i + 1
end while
```

Asymptotic TC:

$$\mathcal{O}(1) + (n-1)(\mathcal{O}(1) + \mathcal{O}(1) + \mathcal{O}(1)) = \mathcal{O}(1) + \mathcal{O}(n-1) \in \mathcal{O}(n)$$

Big-$\mathcal{O}$ notation (determining asymptotic complexity)

Example (Infinite loop)

$x := 0$
while $x < 5$ **do**
    $x := 1$
    $x := x + 1$
end while

In this situation, x will never be greater than 5, since at the start of the while loop, x is given the value of 1, thus, the loop will always end in 2 and the loop will never break.

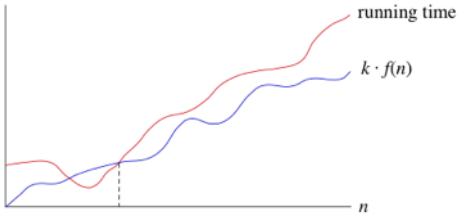
Big-Ω notation (asymptotic lower-bound)

We can also say that an algorithm takes *at least a certain amount of time*, without providing an upper bound.

Big-Omega ($\Omega(\cdot)$) notation

Definition

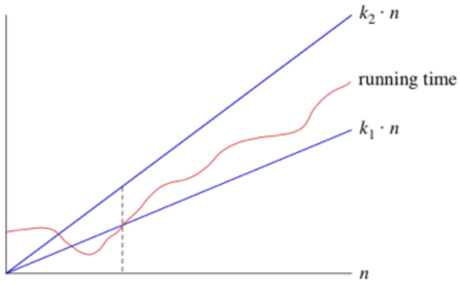
$g(n) \in \Omega(f(n))$ if $\exists\ k > 0$ and n_0 s.t. $g(n) \geq k \cdot f(n),\ \ \forall n \geq n_0$.



Big-Θ notation (asymptotically tight-bound)

Definition

$g(n) \in \Theta(f(n))$ if $\exists$ $k_1, k_2 > 0$ and n_0 s.t.
$k_1 \cdot f_n \leq g(n) \leq k_2 \cdot f(n), \ \forall n \geq n_0$.



QUIZ

Which kind of growth best characterizes each of these functions?

	Constant	**Linear**	**Polynomial**	**Exponential**
$(3/2)^n$				
1				
$(3/2)\ n$				
$2n^3$				
2^n				
$3n^2$				
1000				
$3n$				

Exc 1: Growth of function in Big-$\mathcal{O}$

Which kind of growth best characterizes each of these functions?

	Constant	**Linear**	**Polynomial**	**Exponential**
$(3/2)^n$				$\checkmark$
1	$\checkmark$			
$(3/2)\,n$		$\checkmark$		
$2n^3$			$\checkmark$	
2^n				$\checkmark$
$3n^2$			$\checkmark$	
1000	$\checkmark$			
$3n$		$\checkmark$		

Match each function with an equivalent function that satisfies $g(n) = \Theta(f(n))$.

$g(n)$	$f(n)$
$n + 30$	$n^2 + 3n$
$n^2 + 2n - 10$	n^4
$n^3 * 3n$	$\log_2 2x$
$\log_2 x$	$3n - 1$

Exc 2: Comparing function growth

Recall that $g(n) \in \Theta(f(n))$ if $\exists k_1, k_2 > 0$ s.t. for all sufficiently large n, we have

$$k_1 \cdot f(n) \leq g(n) \leq k_2 \cdot f(n)$$

We drop the constants and lower order terms (i.e. only keep the most significant term).

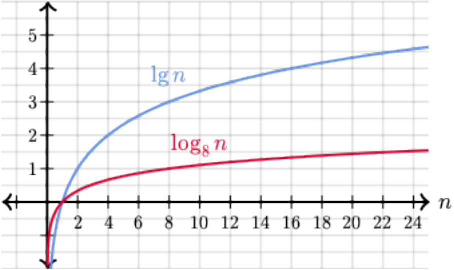
$g(n)$	**simplified**	$f(n)$	**simplified**
$n + 30$	n	$n^2 + 3n$	n^2
$n^2 + 2n - 10$	n^2	n^4	n^4
$n^3 * 3n$	n^4	$\log_2 2x$	$\log x$
$\log_2 x$	$\log x$	$3n - 1$	n

Two functions match if the corresponding simplified functions are equal.

For the functions $\log_2 n$ and $\log_8 n$, what is the asymptotic relationship between these functions?

- $\log_2 n$ is in $\mathcal{O}(\log_8 n)$
- $\log_2 n$ is in $\Omega(\log_8 n)$
- $\log_2 n$ is in $\Theta(\log_8 n)$

Both $\log_2 n$ and $\log_8 n$ are functions with logarithmic growth, with their base as the only difference.

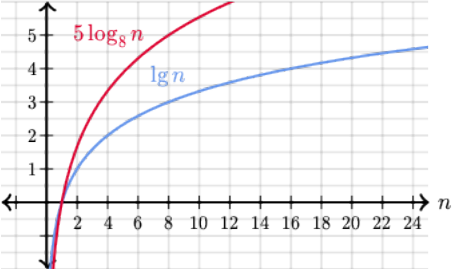


- Is $\log_2 n$ in $\mathcal{O}(\log_8 n)$?

Recall that $\log_a n = \frac{\log_b n}{\log_b a}$.

So, $\log_8 n = \frac{\log_2 n}{\log_2 8} = \frac{\log_2 n}{3} = \frac{1}{3} \cdot \log_2 n$.

We can take $k = 5$, so that: $\log_2 n \leq 5 \log_8 n$.



- Is $\log_2 n$ in $\Omega(\log_8 n)$?

Since $\log_8 n = \frac{1}{3} \cdot \log_2 n$, then $\log_2 n \geq \log_8 n$ for all $n \geq 1$.

So, $\log_2 n \in \Omega(\log_8 n)$

- Is $\log_2 n$ in $\Theta(\log_8 n)$?

Clearly, $\log_8 n \leq \log_2 n \leq 5 \cdot \log_8 n$ for all $n > 1$.

So, $\log_2 n \in \Theta(\log_8 n)$.