UNIVERSITAS INDONESIA
Veritas, Probitas, Iustitia

FAKULTAS ILMU KOMPUTER

Basic Algorithm Analysis (1)

Running Time, Growth of Function, and Asymptotic Notation

DAA Term 2 2023/2024

Computational Tractability

- A computational problem is said to be **tractable** or **easy** if there exists an **efficient** algorithm for it.
  - Mathematically, the problem can be solved in **polynomial time**.
  - It is said to be **intractable** or **hard** if it is solvable by super polynomial time algorithm (i.e. cannot be bounded above by any polynomial)

Computational Tractability

- An algorithm is said to be **polynomial time** if it satisfy the following scaling property:
  - Desirable scaling property: When the input size **doubles**, the algorithm should **slow down by at most some multiplicative constant factor c**.

  There exist **constants c > 0 and d > 0** such that, for every input of size n, the algorithm performs **≤ cN^d** **primitive computational steps**.

- Illustration:
  - Suppose an algorithm runs in **cN^d** for **N** input size
  - Increase the input size to **2N**, then it runs in **$c(2N)^d = c2^dN^d$**
  - It **slows down by a factor of 2^d** (it is constant since **d** is a constant)

Analyzing Algorithm

- Analyzing algorithms involves thinking about how their resource requirements (**the amount of time and space they use**) will scale with **increasing input size**.
    - Efficiency in running time: we want algorithms that run quickly, but it is important that algorithms be efficient in their use of memory as well.
- Also includes predicting the resources (computing time, memory, communication bandwidth, etc) that the algorithm requires.
    - Most often it is **the computing time** that we want to measure.

Random-Access Machine (RAM) and Running Time

- We use a generic on processor **RAM model computation** and implement our algorithms as computer programs on that machine.
- Suppose the RAM model contains such instructions: arithmetic, data movement (copy, load, store), control (conditional and unconditional branch, subroutine, return).
- Now we are going to analyze the **running time** of an algorithm: the number of **primitive computational steps** executed.
  - It depends on the input size and the characteristics of the input (e.g. sorted or not)

Running Time Computation

- Example 1

```
public static int sum( int n )
{
  1 int partialSum;

  2 partialSum = 0;
  3 for( int i = 1; i <= n; i++ )
  4       partialSum += i * i * i;
  5 return partialSum;
}
```

Line 1	0
Line 2	1
Line 3	2n+2
Line 4	4n
Line 5	1
Total time	6n+4

Running Time Computation

- Example 2

```
public static int sum( int n )
{
  1 int partialSum;

  2 partialSum = 0;
  3 for( int i = 1; i <= n; i++ )
  4     partialSum += i * i * i;
  5 return partialSum;
}
```

Line 1	c_1	0
Line 2	c_2	1
Line 3	c_3	n+1
Line 4	c_4	n
Line 5	c_5	1
Total time	$c_2 + c_3(n+1) + c_4n + c_5$	

$$\boldsymbol{T(n) = (c_3 + c_4)n + (c_2 + c_3 + c_5)}$$

Running Time

What to consider in analyzing the running time of an algorithm?

- **Best Case Condition (never)**
- **Average Case Condition (sometimes)** → Obtain bound on running time of algorithm on **random inputs** as a function of input size **n.**
    - It is hard (or impossible) to accurately model the real instance by a random distribution.
    - When an algorithm tuned for a certain distribution may perform poorly on other inputs.
- **Worst Case Condition (usually)** → Obtain bound on **the largest possible running time** of algorithm on input of a given size **n**.
    - It occurs often
    - Worst case running time describes the upper bound running time
    - The average case sometimes as bad as the worst case

Running Time

Table 2.1 The running times (rounded up) of different algorithms on inputs of increasing size, for a processor performing a million high-level instructions per second. In cases where the running time exceeds 10^{25} years, we simply record the algorithm as taking a very long time.

	n	$n \log_2 n$	n^2	n^3	1.5^n	2^n	$n!$
$n = 10$	< 1 sec	< 1 sec	< 1 sec	< 1 sec	< 1 sec	< 1 sec	4 sec
$n = 30$	< 1 sec	< 1 sec	< 1 sec	< 1 sec	< 1 sec	18 min	10^{25} years
$n = 50$	< 1 sec	< 1 sec	< 1 sec	< 1 sec	11 min	36 years	very long
$n = 100$	< 1 sec	< 1 sec	< 1 sec	1 sec	12,892 years	10^{17} years	very long
$n = 1{,}000$	< 1 sec	< 1 sec	1 sec	18 min	very long	very long	very long
$n = 10{,}000$	< 1 sec	< 1 sec	2 min	12 days	very long	very long	very long
$n = 100{,}000$	< 1 sec	2 sec	3 hours	32 years	very long	very long	very long
$n = 1{,}000{,}000$	1 sec	20 sec	12 days	31,710 years	very long	very long	very long

Running Time

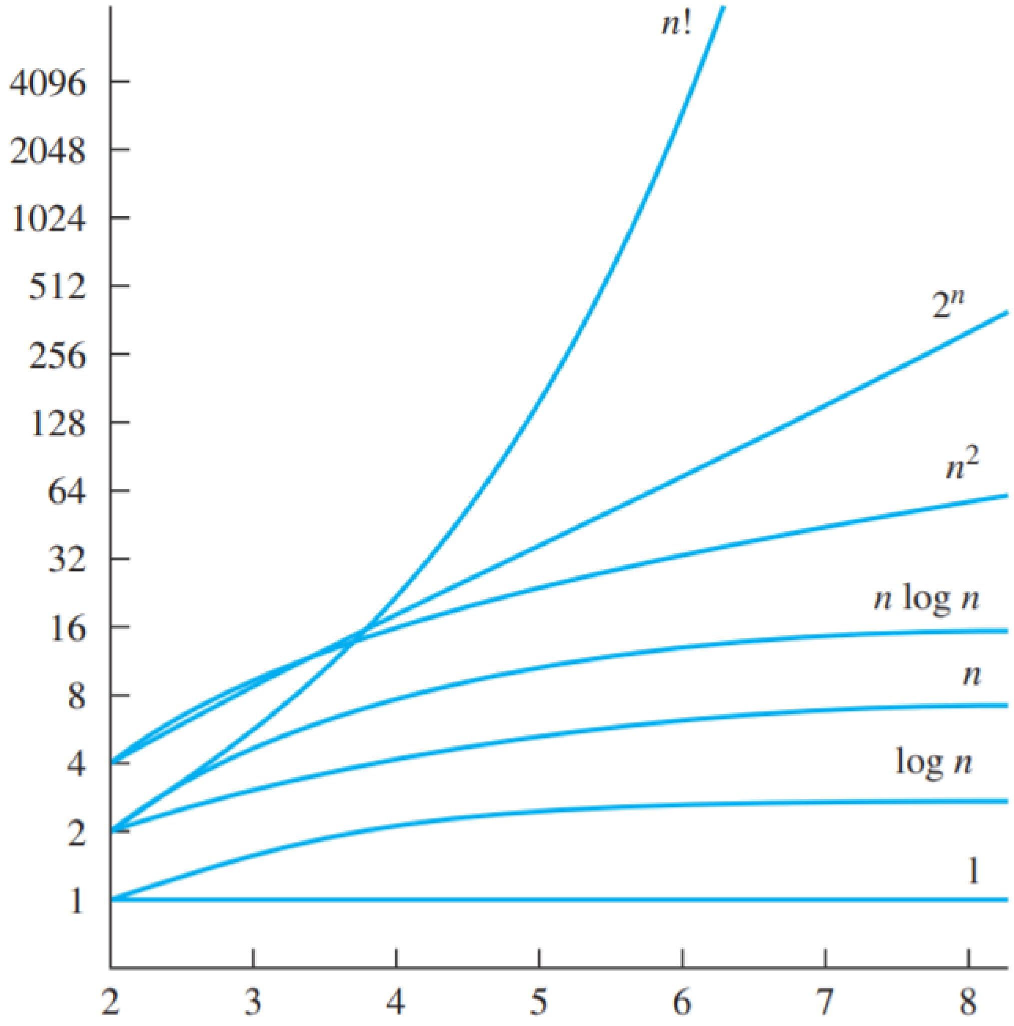
- Worst Case Polynomial Time
  - Recall that: an **easy** or **efficient** algorithm runs in **polynomial time**
    - It usually **works in practice** since a polynomial time algorithms almost always have low constants and low exponents.
  - It would be useless in practice if a polynomial time algorithm has high constants or exponents, e.g: $\mathbf{6.02 \times 10^{23} \times n^{20}}$
    - However, such algorithm does exist and widely used because the worst-case instance seem to be rare. Example: simplex method, unix grep

The order of growth

- A more simplifying abstraction to ease analysis and focus on important features.
- As the value of n becomes larger, we only need to focus on the highest order.
- Only consider the leading term of the formula for running time.
  - Drop lower order terms
  - Ignore the constant coefficient in the leading term
- Example: $\boldsymbol{T(n) = (c_3 + c_4)n + (c_2 + c_3 + c_5)}$
  - $T(n) = an + b$ (by abstracting away the actual statement costs) → $T(n) = an$ (by dropping the lower order terms) → $T(n) = n$ (by ignoring the constant coefficient in the leading term). Then the corresponding algorithm grows like n

Growth of Function

- Faster growth means less efficiency

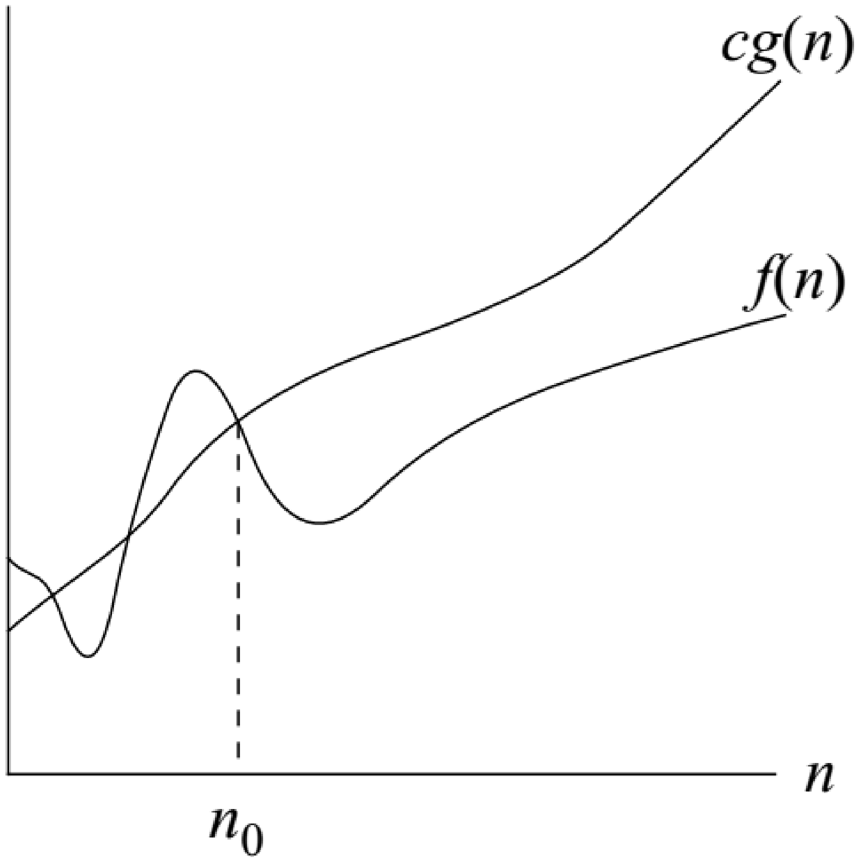


Asymptotic Notation

- To **describe the running times or asymptotic efficiency** of algorithms
- It applies to functions (that characterize the running times of algorithms, in this case)
- Which running time we want to express?
  - The worst-case running time?
  - Running time that covers all input?

Objective: To express running time in a right asymptotic notation!

Big Oh (O) notation



$f(n) = O(g(n))$ means:
$g(n)$ is an **asymptotic upper bound** for $f(n)$

$$O(g(n)) = \{f(n) : \text{there exist positive constants } c \text{ and } n_0 \text{ such that } 0 \le f(n) \le c\,g(n) \text{ for all } n \ge n_0\}.$$

as n grows larger, $f(n)$ grows slower than $cg(n)$

It is also acceptable to write
$\boldsymbol{f(n)}$ ***is*** $\boldsymbol{O(g(n))}$ or $\boldsymbol{f(n) \in O(g(n))}$

Big Oh ($\boldsymbol{O}$) notation

Example (1)

- Show that $f(n) = n^2 + 2n + 1$ is $O(n^2)$
  - Find c and n_0 such that $0 \leq f(n) \leq cn^2$ for all $n \geq n_0$
  - How do we find c and n_0?
    - We know that $n^2 + 2n + 1 \leq n^2 + n^2 + n^2 = 3n^2$ because for $n \geq 2$, $2n \leq n^2$ and $1 \leq n^2$. Therefore, we choose $c = 3$ and $n_0 = 2$.
    - Sometimes it is better to choose a large enough value of n_0 or c. For example, we can set $c = 100$ for every $n \geq 1$
  - For the complete proof, we can use mathematical induction to show that $\boldsymbol{n^2 + 2n + 1 \leq cn^2}$ **for all** $\boldsymbol{n \geq n_0}$ based on the value of c and n_0 that have been chosen.

Big Oh ($\boldsymbol{O}$) notation

Example (1) cont'd

- Show that $f(n) = n^2 + 2n + 1$ is $O(n^2)$
  - For the complete proof, we can use mathematical induction to show that $\boldsymbol{n^2 + 2n + 1 \leq cn^2}$ **for all** $\boldsymbol{n \geq n_0}$ based on the value of c and n_0 that have been chosen.

Big Oh ($\boldsymbol{O}$) notation

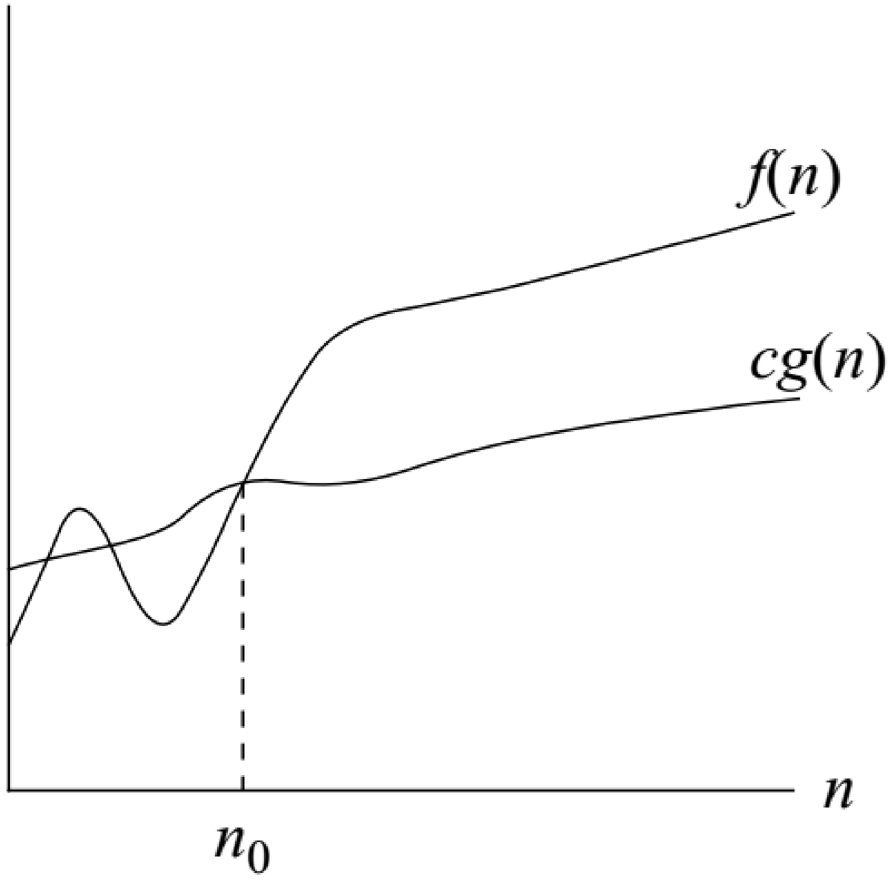
Example (2)

- Show that $6n^3 \neq O(n^2)$
  - Proof by contradiction?

- (True or False?) $2^{n+1} = O(2^n)$

- (True or False?) $2^{2n} = O(2^n)$

Big Omega (Ω) notation



$f(n) = \Omega(g(n))$ means:
$g(n)$ is **an asymptotic lower bound** for $f(n)$

$$\boxed{\Omega(g(n)) = \{f(n) : \text{there exist positive constants } c \text{ and } n_0 \text{ such that } 0 \le c\,g(n) \le f(n) \text{ for all } n \ge n_0\}.}$$

as n grows larger, $f(n)$ grows faster than $cg(n)$

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

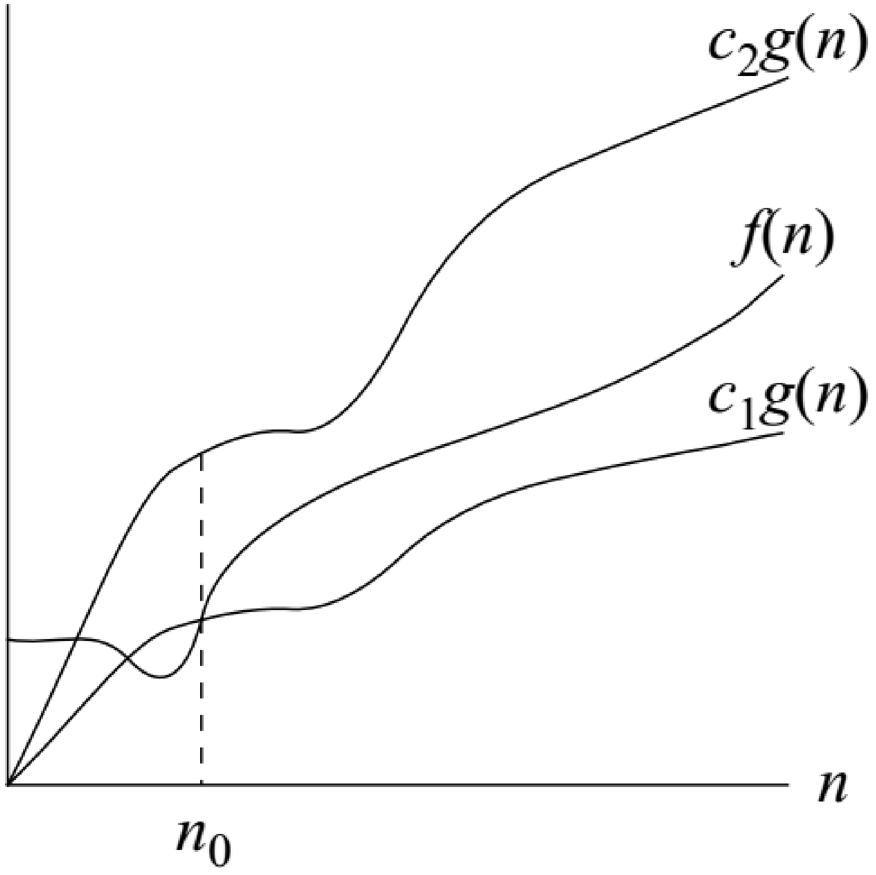
Big Omega (Ω) notation

Example

- Show that $8n^3 + 5n^2 + 7 = \Omega(n^3)$

- Show that $\sqrt{n} = \Omega(\lg n)$

Big Theta (Θ) notation



$f(n) = \Theta(g(n))$ means:
$g(n)$ is an **asymptotic tight bound** for $f(n)$

$$\boxed{\begin{aligned}\Theta(g(n)) = \{ f(n) : &\text{ there exist positive constants } c_1, c_2, \text{ and } n_0 \\ &\text{such that } 0 \le c_1 g(n) \le f(n) \le c_2 g(n) \text{ for all } n \ge n_0 \}\end{aligned}}$$

If $\lim_{n\to\infty} \frac{f(n)}{g(n)} = c$ for some constant $0 < c < \infty$, then $f(n)$ is $\Theta(g(n))$

Big Theta ($\boldsymbol{\Theta}$) notation

Example

- Show that $\frac{1}{2}n^2 - 3n = \Theta(n^2)$

Properties of the **"Big Notations"**

- $f(n) = \Theta(g(n)) \rightarrow f(n) = O(g(n))$
- $\Theta(g(n)) \subseteq O(g(n))$
- $f(n) = \Theta(g(n))$ if and only if $f(n) = O(g(n))\ and\ f(n) = \Omega(g(n))$

- Transitivity Property
  - If $f(n) = O(g(n))$ and $g(n) = O(h(n))$, then $f(n) = O(h(n))$
- Additivity Property
  - If $f(n) = O(h(n))$ and $g(n) = O(h(n))$, then $(f+g)(n) = O(h(n))$

Transitivity and Additivity properties apply on Ω and Θ as well.

Exercise

- Show that $n^2 + 4n + 17 = O(n^3)$ but that n^3 is not $O(n^2 + 4n + 17)$
- Show that $3n^2 + 8n \log n = O(n^2)$
- Express the following running time in different asymptotic notations.

$$T(n) = 32n^2 + 17n + 32$$

Little Oh ($\boldsymbol{o}$) notation

- Example:
  - $2n^2 = O(n^2)$ is asymptotically tight
  - $2n = O(n^2)$ is not asymptotically tight

- Little oh (o) notation is used to denote a not asymptotically tight upper bound.
  - A function grows at **a significantly slower rate** than another
  - $\lim_{n \to \infty} \frac{f(n)}{g(n)} = 0$

    (This proposition also works for Big Oh notation)

- Which one is true? $2n^2 = o(n^2)$ or $2n = o(n^2)$?

Little Omega (ω) notation

- Little omega notation is used to denote a <u>not asymptotically tight lower bound.</u>
- $f(n) = \omega\big(g(n)\big)$ if and only if $g(n) = o\big(f(n)\big)$
- $\lim_{n\to\infty} \frac{f(n)}{g(n)} = \infty$ or $\lim_{n\to\infty} \frac{g(n)}{f(n)} = 0$
  (This proposition also works for Big Omega notation)
- Example:
  - $\frac{n^2}{2} = \omega(n)$
  - $\frac{n^2}{2} \neq \omega(n^2)$

Discussion

- Explain why the statement "The running time of an algorithm is at least $\mathrm{O}(n^2)$" is meaningless!

- What is the difference of following statements?
  - The worst-case running time of an algorithm is $\mathrm{O}(n^2)$
  - The worst-case running time of an algorithm is $\Theta(n^2)$

- Explain why the following statements are equivalent.
  - The running time of an algorithm is $\mathrm{O}(n^2)$
  - The worst-case running time of an algorithm is $\mathrm{O}(n^2)$

Summary

For **a function** :

- Big Oh (O) denotes the **upper bound**
- Big Theta (Θ) denotes the **tight bound**
- Big Omega (Ω) denotes the **lower bound**
- Little Oh (o) denotes the **not tight upper bound**
- Little Omega (ω) denotes the **not tight lower bound**

Summary 2

- We use **Big Oh to represent the worst-case running time** of an algorithm. It means for other cases it is possible for this algorithm to run faster (this condition is covered by Big Oh notation).
  - If we use Big Theta to represent the worst-case running time of an algorithm, note that it does not cover the running time for other cases.
- We use **Big Omega to represent the best-case running time** of an algorithm, it covers the running time in other cases.

Some Functions and Notations

- Logarithmic

$$\lg n = \log_2 n \qquad \text{(binary logarithm)}$$
$$\ln n = \log_e n \qquad \text{(natural logarithm)}$$
$$\lg^k n = (\lg n)^k \qquad \text{(exponentiation)}$$
$$\lg \lg n = \lg(\lg n) \quad \text{(composition)}$$
$$\lg n + k = (\lg n) + k$$

For all real $a > 0, b > 0, c > 0$, and n,

$$a = b^{\log_b a}$$
$$\log_c(ab) = \log_c a + \log_c b$$
$$\log_b a^n = n \log_b a$$
$$\log_b a = \frac{\log_c a}{\log_c b}$$
$$\log_b(1/a) = -\log_b a$$
$$\log_b a = \frac{1}{\log_a b}$$
$$a^{\log_b c} = c^{\log_b a}$$

Some Functions and Notations

- Functional Iteration

$$f^{(i)}(n) = \begin{cases} n & \text{if } i = 0 , \\ f(f^{(i-1)}(n)) & \text{if } i > 0 . \end{cases}$$

For example, if $f(n) = 2n$, then $f^{(i)}(n) = 2^i n$.

- The Iterated Logarithm Function

$$\lg^* n = \min\{i \geq 0 : \lg^{(i)} n \leq 1\}$$

The iterated logarithm is a *very* slowly growing function:

$$\begin{aligned} \lg^* 2 &= 1 , \\ \lg^* 4 &= 2 , \\ \lg^* 16 &= 3 , \\ \lg^* 65536 &= 4 , \\ \lg^*(2^{65536}) &= 5 . \end{aligned}$$

Some Functions and Notations

- **Factorial**

$$n! = \begin{cases} 1 & \text{if } n = 0 , \\ n \cdot (n-1)! & \text{if } n > 0 . \end{cases}$$

  - The weak upper bound for factorial function is $n! \leq n^n$
  - Stirling's Approximation:

$$n! = \sqrt{2\pi n}\left(\frac{n}{e}\right)^n \left(1 + \Theta\left(\frac{1}{n}\right)\right)$$

    - From Stirling's Approximation, we obtain $\lg(n!) = O(n \lg n)$. (This conclusion will be used in the analysis of the lower bound for comparison-based sorting)

References

- Lecturer Slides by Bapak L. Yohanes Stefanus
- Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, and Clifford Stein. 2009. Introduction to Algorithms, Third Edition (3rd. ed.). The MIT Press.
- https://www.cs.princeton.edu/~wayne/kleinberg-tardos/pdf/02AlgorithmAnalysis.pdf