香港中文大學(深圳)
The Chinese University of Hong Kong

# CSC3100 Data Structures
# Lecture 5: Asymptotic notations

Li Jiang
School of Data Science (SDS)
The Chinese University of Hong Kong, Shenzhen

# How to evaluate the "efficiency" of an algorithm?

- A naive approach is to do experiment
  - It requires to implement the algorithm and run it
  - Results may not be indicative of the running time on other inputs which are not included in the experiment
  - In order to compare two algorithms, we need to make a fair comparison such that
    - The algorithms are implemented by the same person with the same programming language
    - They are evaluated on same hardware and software environments

# Basic operations in pseudocodes

- **Algorithms** require every operation to be basic enough
  - Arithmetic operations
    - a+b*c+d/e, a mod 7
  - Assigning a value to a variable
    - x ← 3
  - Indexing into an array / accessing value of objects/structs
    - arr[i], arr.length, student.name
  - Calling a method
    - x ← factorial(n)
  - Returning from a method
    - return y
  - Comparison
    - x==y, x>y, x<y, x>=y, x<=y, x!=y

> A reasonable assumption:
> each basic operation takes constant time

# Running time: theoretical analysis

- Worst case analysis:
  - Why worst case?
    - Imagine that you use google and it takes 100 seconds to finish…
  - Why not analyze the average case?
    - Because it is often as bad as the worst case

  - How to do it?
    - Use pseudocode to describe the algorithm
    - Characterize the running time as a function of the input size, *n*
    - Take into account all possible inputs
    - Evaluate the speed of an algorithm independent of the hardware/software environment

# Analyzing the running time (i)

- How do we analyze an algorithm?
  - Count the number of basic operations

***sum(A,n)***

```
tempsum = 0
for i = 0 to n-1
    tempsum += A[i]
return tempsum
```

Number of basic operations

1

2*n +2

2*n

1

Total: 4*n +4

Why 2*n +2? The actual effect of the loop is:

for (i=0; i<n ; i++)

We assign 0 to i once, increment i for n times, compare i<n n+1 times

Why 2*n? Accessing A[i] n times, addition to tempsum n times

It is annoying to count the detailed number of basic operations, we will show how to simplify the counting process later

# Analyzing the running time (ii)

- The *Sum* algorithm executes 4*n+4 basic operations
  - The running time depends on its input size, i.e., the number of elements that can be accessed by the algorithm
    - a pointer of an array is passed as the input, but we can access all the n elements in the array, so the input size is n

- Other inputs might also impact the running time of an algorithm besides the input size

*LinearSearch(A,n,searchnum)*

```
for i = 0 to n -1
    if A[i] == searchnum
        return i
return -1
```

| | | | | | | | | |
|---|---|---|---|---|---|---|---|---|
| 3 | 7 | 9 | 12 | 13 | 18 | 20 | 23 | 27 |

searchnum = 3 vs searchnum = 20

The number of basic operations depends on where the **searchnum** value is in the given array

# Practice

- What is the number of basic operations executed by the maxSubArraySum algorithm in the worst case?

***maxSubArraySum(A, n)***

| | |
|---|---|
| `maxSum = A[0]` | 2 |
| `for i = 0 to n-1` | $2n+2$ |
| `  subSumFromI = 0` | $n$ |
| `  for j = i to n-1` | $(2\cdot n+2)+(2\cdot(n-1)+2)+\cdots+(2\cdot 1+2)$ |
| `    subSumFromI += A[j]` | $2(n+(n-1)+(n-2)+\cdots+1)$ |
| `    if subSumFromI > maxSum` | $(n+(n-1)+(n-2)+\cdots+1)$ |
| `      maxSum = subSumFromI` | $(n+(n-1)+(n-2)+\cdots+1)$ |
| `return maxSum` | 1 |

Total: $3n^2+8n+5$

The expression above is too complicated – we need asymptotic analysis

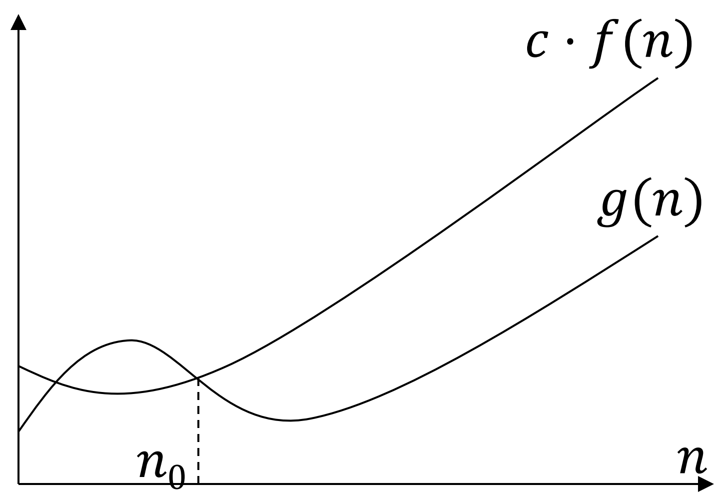
## Quantifying the growth rate of functions: **Big-Oh**

"Asymptotic" refers to a function approaching a given value or condition, as an expression containing a variable approaches a limit

# Big-Oh notation

- Big-Oh **definition**:
  - $g(n) = O(f(n))$ if and only if there exist positive constants $c$ and $n_0$ such that $g(n) \le c \cdot f(n)$ for all $n \ge n_0$
    - $g(n)$ grows asymptotically no faster than $f(n)$
    - $f(n)$ is usually much simpler than $g(n)$

- Two implications
  - $n \ge n_0$ means that it cares a large size of the input data, larger than some given number $n_0$
  - $g(n) \le c \cdot f(n)$ means that $g(n)$ grows no faster than a constant $c$ time of the simplified function $f(n)$



# Examples of Big-Oh (i)

- For any linear function $g(n) = a \cdot n + b$, with $a \geq 0, b \geq 0$, we have $g(n) = a \cdot n + b = O(n)$

> Proof: We can find $c = (a+b)$, $n_0 = 1$ such that
> $$g(n) = an + b \leq an + bn \text{ for } n \geq 1.$$
> Therefore $g(n) = O(n)$.

- For two log functions with different base $a > 1$ and $b > 1$, we have $g(n) = \log_a n = O(\log_b n)$

$\log_a n = \log_b n \,/\, \log_b a$

> Proof: We can find $c = 1/\log_b a$, $n_0 = 1$ such that
> $$g(n) = \log_a n \leq c \cdot \log_b n \text{ for } n \geq 1.$$
> Thus $g(n) = \log_a n = O(\log_b n)$.

> The base does not affect the complexity:
> $$O(\log_b n) = O(\log n)$$
> Where you may assume $\log n = \log_{10} n$ or $\log n = \log_2 n$.

# The polynomial rule (i)

- Given $a > b \geq 0, n^a$ grows faster than $n^b$
  - $\lim_{n \to \infty} \frac{n^b}{n^a} \to 0$

The biggest eats all

If $g(n)$ is a non-negative polynomial of degree **d**, i.e., $g(n) = a_d \cdot n^d + a_{d-1} \cdot n^{d-1} + \cdots + a_1 \cdot n + a_0$, then $g(n)$ is $O(n^d)$.

Proof:

$$g(n) = a_d \cdot n^d + a_{d-1} \cdot n^{d-1} + \cdots + a_1 \cdot n + a_0$$
$$\leq |a_d| \cdot n^d + |a_{d-1}| \cdot n^{d-1} + \cdots + |a_1| \cdot n + |a_0|$$
$$(x \leq |x|)$$
$$= n^d(|a_d| + |a_{d-1}| \cdot \frac{1}{n} + \cdots + |a_1| \cdot \frac{1}{n^{d-1}} + |a_0| \cdot \frac{1}{n^d})$$
$\leq n^d(|a_d| + |a_{d-1}| + \cdots |a_1| + |a_0|)$ for any $n \geq 1$

($\frac{1}{n^i} \leq 1$ for any ($\mathrm{n} \geq 1, i \geq 0$) )

# Examples

- $g(n) = n^4 - n^3 + n^2$
  - $O(n^4)$
- $g(n) = 1 + 1000000n + n^2$
  - $O(n^2)$

- Extensions to polynomial of non-integer degree
  - $g(n) = 1 + 1000000n + n^{1.5}$
    - $O(n^{1.5})$
  - $g(n) = 1 + 1000000n^{0.3} + n^{0.5}$
    - $O(n^{0.5})$

# Product rule (ii)

- Product property
  - $g_1(n) = O(f_1(n)), g_2(n) = O(f_2(n))$
    - Then, $g_1(n) \cdot g_2(n) = O(f_1(n) \cdot f_2(n))$

- Example

  The big multiplies the big
  - $g(n) = (n^3 + n^2 + 1)(n^7 + n^{12})$
  - What is the time complexity of $g(n)$?
    - Let $g_1(n) = (n^3 + n^2 + 1)$, $g_2(n) = (n^7 + n^{12})$
      - $g_1(n) = O(n^3), g_2(n) = O(n^{12})$
      - $g(n) = g_1(n) \cdot g_2(n) = O(n^3 \cdot n^{12}) = O(n^{15})$

# Big-Oh rules (iii)

- Sum property

  The bigger of the two big

  - $g_1(n) = O\big(f_1(n)\big), g_2(n) = O(f_2(n))$
    - Then, $g_1(n) + g_2(n) = O(\ \max(f_1(n), f_2(n)\ )\ )$

- Example
  - $g(n) = (n^3 + n^2 + 1) + (n^4 + n^7 + n^{20} + n^{30})$
  - $g_1 = (n^3 + n^2 + 1), g_2 = (n^4 + n^7 + n^{20} + n^{30})$
    - $g_1 = O(n^3), g_2 = O(n^{30})$
  - $g(n) = O(\max(n^3, n^{30}))$
    - $O(n^{30})$
  - We can easily extend it to the case when $g(n) = g_1(n) + g_2(n) \cdots + g_c(n)$, which is the sum of a constant number $c$ of functions

# Big-Oh rules (iv)

- Log functions grow slower than power functions
  - $\lim_{n\to\infty} \frac{(\log n)^a}{n^b} \to 0$, for $a, b > 0$

- If $g(n) = (\log n)^a + n^b$ where $a, b > 0$,
  - Then $g(n) = O(n^b)$.
- If $g(n) = (\log n)^a + b \cdot (\log n)^x$, where $b > 0, a > x \geq 0$
  - Then $g(n) = O((\log n)^a)$.

- Example
  - $g(n) = (\log n)^2 + n^{0.0001}$
    - Let $a = 2, b = 0.0001$
    - $O(n^{0.0001})$

log only beats constant

# Big-Oh rules (v)

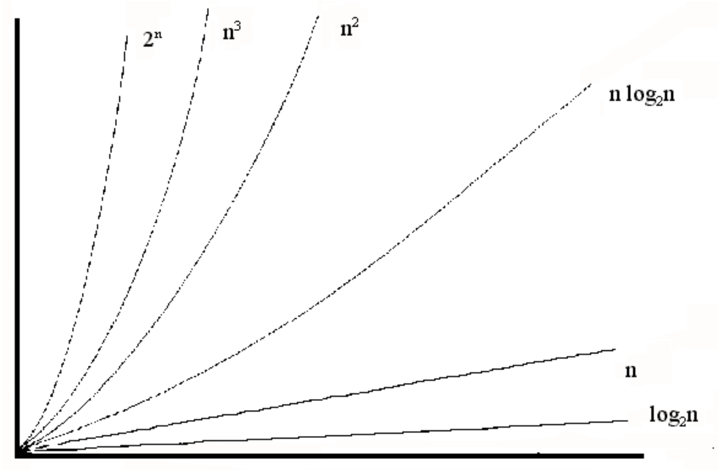
- Exponential functions grow faster than power functions
  - $\lim_{n\to\infty} \frac{n^k}{a^n} \to 0$ if $a > 1$ for any $k$

- If $g(n) = a^n + n^b$, where $a > 1$
  - Then $g(n) = O(a^n)$
- If $g(n) = a^n + b^n$, where $a > b > 1$
  - Then $g(n) = O(a^n)$

Exponential beats powers

- Examples
  - $g(n) = 2^n + n^{10000}$
    - $O(2^n)$
  - $g(n) = 2^n + 3^n$
    - $O(3^n)$, why? (for $c = 2, n_0 = 1$ $g(n) = 2^n + 3^n \leq 3^n + 3^n$ for $n_0 \geq 1$)

# Common Big-Oh functions/classes




| | n=1 | n=2 | n=4 | n=8 | n=16 | n=32 |
|---|---|---|---|---|---|---|
| $O(1)$ | 1 | 1 | 1 | 1 | 1 | 1 |
| $O(\log_2 n)$ | 0 | 1 | 2 | 3 | 4 | 5 |
| $O(n)$ | 1 | 2 | 4 | 8 | 16 | 32 |
| $O(n \log_2 n)$ | 0 | 2 | 8 | 24 | 64 | 160 |
| $O(n^2)$ | 1 | 4 | 16 | 64 | 256 | 1024 |
| $O(n^3)$ | 1 | 8 | 64 | 512 | 4096 | 32768 |
| $O(2^n)$ | 2 | 4 | 16 | 235 | 65536 | 4294967296 |

**BETTER**

- $O(1)$ constant time
- $O(\log n)$ log time
- $O(n)$ linear time
- $O(n \cdot \log n)$ log linear time
- $O(n^2)$ quadratic time
- $O(n^3)$ cubic time
- $O(2^n)$ exponential time

**WORSE**

# Practice

- Prove or disprove: $g(n) = 2n = O(n^2)$

Fina a constant $c, n_0$ such that
$2n \leq cn^2$ for all $n \geq n_0 \Leftrightarrow 2 \leq cn$ for all $n \geq n_0$.
Let $c = 2, n_0 = 1$, the above inequality holds. $g(n) = O(n^2)$

- Wait a second? Isn't $g(n) = 2n = O(n)$?
  - Recall Big-Oh denotes the upper bound of the grow rate of $g(n)$
  - There might exist many upper bound
    - $g(n) = 2n = O(n \cdot \log n) = O(n^2) = O(n^3) = O(2^n)$
  - But we want to find the tightest
    - We also need the lower bound of the growth rate

# Typical mistakes

- $f(n) = \sum_{i=1}^{n} 1$
  - Set $g_1 = g_2 = \cdots g_n = 1$, then $O(f(n) = O(\max(1,1,1 \cdots, 1)) = O(1)$
  - However, $f(n) = n \neq O(1)$
  - What is wrong here?
    - The number of functions depends on $n$
    - The sum property only applies when the number of functions are constants

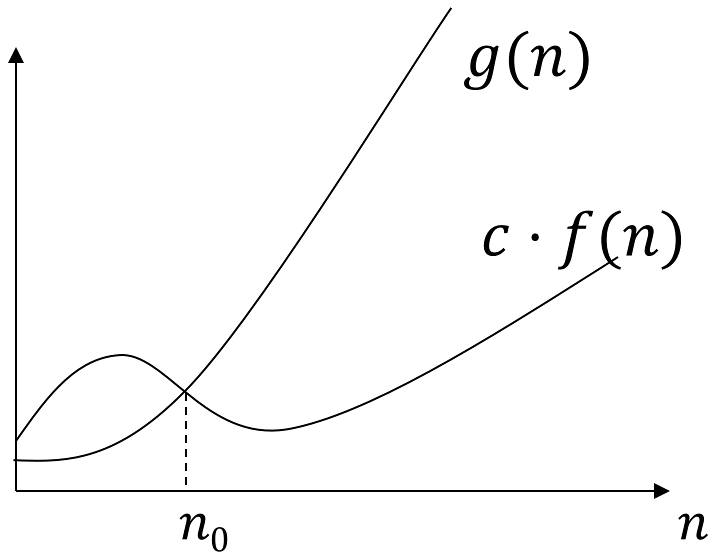
- $2n = O(n^2)$ and $3n^2 = O(n^2)$, so $2n = 3n^2$ ?
  - What is wrong here?
    - Big-Oh is just a notation!

Quantifying the growth rate of functions: **Big-Omega**

# Big-Omega notation

- Big-Omega **definition**:
  - $g(n) = \Omega\big(f(n)\big)$ if and only if there exist positive constants $c$ and $n_0$ such that $g(n) \geq c \cdot f(n)$ for all $n \geq n_0$
    - $g(n)$ grows asymptotically no slower than $f(n)$
    - $f(n)$ is usually much simpler than $g(n)$

- Implication
  - $c \cdot f(n)$ grows slower than $g(n)$
  - $g(n)$ will surpass $c \cdot f(n)$ at some point, which we denote as $n_0$



# Example of Big-Omega (i)

- For linear search: $g(n) = 4n + 4$, prove $g(n) = \Omega(n)$

Proof: We can find $c = 4,\ n_0 = 1$ such that

$g(n) = 4n + 4 \geq 4n$ for $n \geq 1$.

Therefore $g(n) = \Omega(n)$.

- $g(n) = 2n$, prove that $g(n) \neq \Omega(n^2)$

Proof: If $g(n) = \Omega(n^2)$, then we need to find a positive constant c and $n_0$ such that

$g(n) = 2n \geq cn^2$ for all $n \geq n_0$.

That is to say we need to guarantee that:

$2 \geq c \cdot n$ for $n \geq n_0$.

However, $c$ is a constant, and the above inequality does not always hold, contradiction. Therefore, $g(n) \neq \Omega(n^2)$.

# Example of Big-Omega (ii)

- If $g(n) = \Omega\big(f(n)\big)$, then $f(n) = \mathrm{O}\big(g(n)\big)$
- If $g(n) = O\big(f(n)\big)$, then $f(n) = \Omega\big(g(n)\big)$

Proof: We prove the first part. The second part can be proved in a similar way.

According to the definition, we know that there exist constants positive constants $c, n_0$, such that

$$g(n) \geq c \cdot f(n) \text{ for } n \geq n_0.$$

That is to say:

$$f(n) \leq 1/c \cdot g(n) \text{ for } n \geq n_0.$$

We find a constant $c' = 1/c, n_0' = n_0$ such that $f(n) \leq c' \cdot g(n)$ for $n \geq n_0'$.

Therefore, $f(n) = O(g(n))$ according to the definition.

# Big-Omega rules

- Big-Oh rules (i) –(v) all apply to Big-Omega
  - Rule (i): The polynomial rule (the biggest eats all)
    - $g(n) = n^3 + n^2 + n$
    - $n^3$ has the biggest power, it eats all other terms, so $g(n) = \Omega(n^3)$
  - Rule (ii): Product property (the big multiplies the big)
    - $g(n) = (n+1) \cdot (n + n^5)$
    - The first big $\Omega(n)$; the second big $\Omega(n^5)$, so $g(n) = \Omega(n \cdot n^5) = \Omega(n^6)$
  - Rule (iii): Sum property (the bigger of the two big)
    - $g(n) = (n+1) + (n + n^5)$
    - The first big $\Omega(n)$; the second big $\Omega(n^5)$, so $g(n) = \Omega(\max(n, n^5)) = \Omega(n^5)$

# Big-Omega rules (cont.)

- Big-Oh rules (i) –(v) all apply to Big-Omega
  - Rule (iv): log only beats constant
    - $g(n) = n^{0.000001} + (\log n)^{1000}$
      - $\Omega(n^{0.000001})$
    - $g(n) = 0.00000001 \cdot \log n + 100000$
      - $\Omega(\log n)$
  - Rule (v): exponential beats powers
    - $g(n) = 1.00001^n + n^{999999}$
      - $\Omega(1.00001^n)$
    - $g(n) = 3^n + 4^n$
      - $\Omega(4^n)$

# The limitation of Big-Omega

- Like Big-Oh, Big-Omega is not tight
  - Big-Omega denotes the asymptotic lower bound of the grow rate of $g(n)$
  - There may exist more than one lower bound
    - $g(n) = n^3 = \Omega(n^2) = \Omega(n \cdot \log n) = \Omega(n) = \Omega(1)$, etc.
  - But we want to find the tightest
    - We need more precise notations

Quantifying the growth rate of functions: **Big-Theta**

# Big-Theta

- We use Big-Theta ($\Theta$) to make the growth-rate tight
  - If $g(n) = O\big(f(n)\big)$ and $g(n) = \Omega(f(n))$
  - Then, $g(n) = \Theta(f(n))$
    - $g(n) = O\big(f(n)\big)$: $g(n)$ grows asymptotically no faster than $f(n)$
    - $g(n) = \Omega\big(f(n)\big)$: $g(n)$ grows asymptotically no slower than $f(n)$

    - If $g(n) = \Theta(f(n))$, it means the growth rate of $g(n)$ is asymptotically the same as $f(n)$

# Practice

- Verify the following
  - $g(n) = 3n + 4 = \Theta(n)$
  - $g(n) = n \cdot \log n + 2n^2 = \Theta(n^2)$
  - $g(n) = (3n^2 + 2\sqrt{n}) \cdot (n \log n + n) = \Theta(n^3 \cdot \log n)$

# Steps for worst-case analysis

- Step 1: find the worst-case number of basic operations in the algorithm as a function of the input size
  - Example: Linear search
    - We counted that the number of basic operations by Linear Search is $4n + 4$

- Step 2: Use Big-Oh and Big-Omega to analyze the algorithm, and derive the Big-Theta if possible
  - We may not be lucky enough to derive that Big-Oh and Big-Omega to be the same
  - How to handle this case? (Next page)

# Asymptotic algorithm analysis

- If two algorithms have the same Big-Theta function, they can be considered as equally good
  - $g_{1(n)} = 10000n = \Theta(n)$ and $g_{2(n)} = 9n = \Theta(n)$
- An algorithm with the slower growth rate $\Theta(f(n))$ than others is asymptotically faster than other algorithms (solving the same problem)
  - An algorithm with $\Theta(\log n)$ time complexity is asymptotically better than the one with $\Theta(n)$ since the former grows slower than the latter
    - E.g., binary search is asymptotically better than linear search
  - Less rigorously, we may also say that an algorithm with $O(\log n)$ time complexity is better than the one with $O(n)$, when we cannot derive that Big-Oh and Big-Omega to be the same

# Recommended reading

- Reading this week
  - Chapter 3, textbook
- Next lecture
  - Chapter 2, textbook