

# Lecture 2

# Algorithm Analysis

---

Bo Tang @ SUSTech, Fall 2022

Several pages are based on the notes by Dr. Ken Yiu (PolyU) and Dr. Yufei Tao (CUHK)

# Our Roadmap

- ◆ RAM Computation Model

  - ◆ Memory, CPU, Algorithm

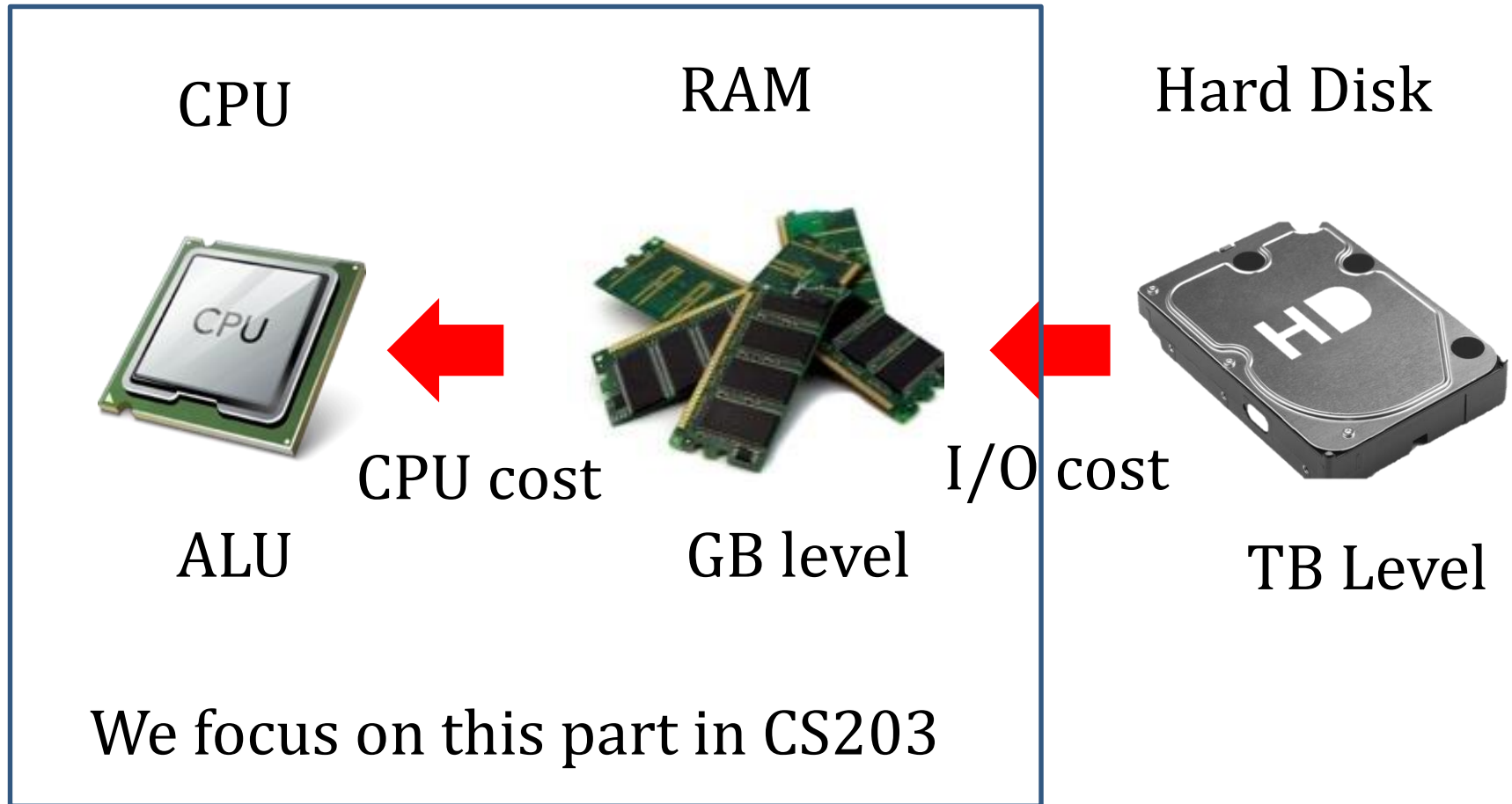
  - ◆ Algorithm, Pseudocode

- ◆ Worst Case Analysis

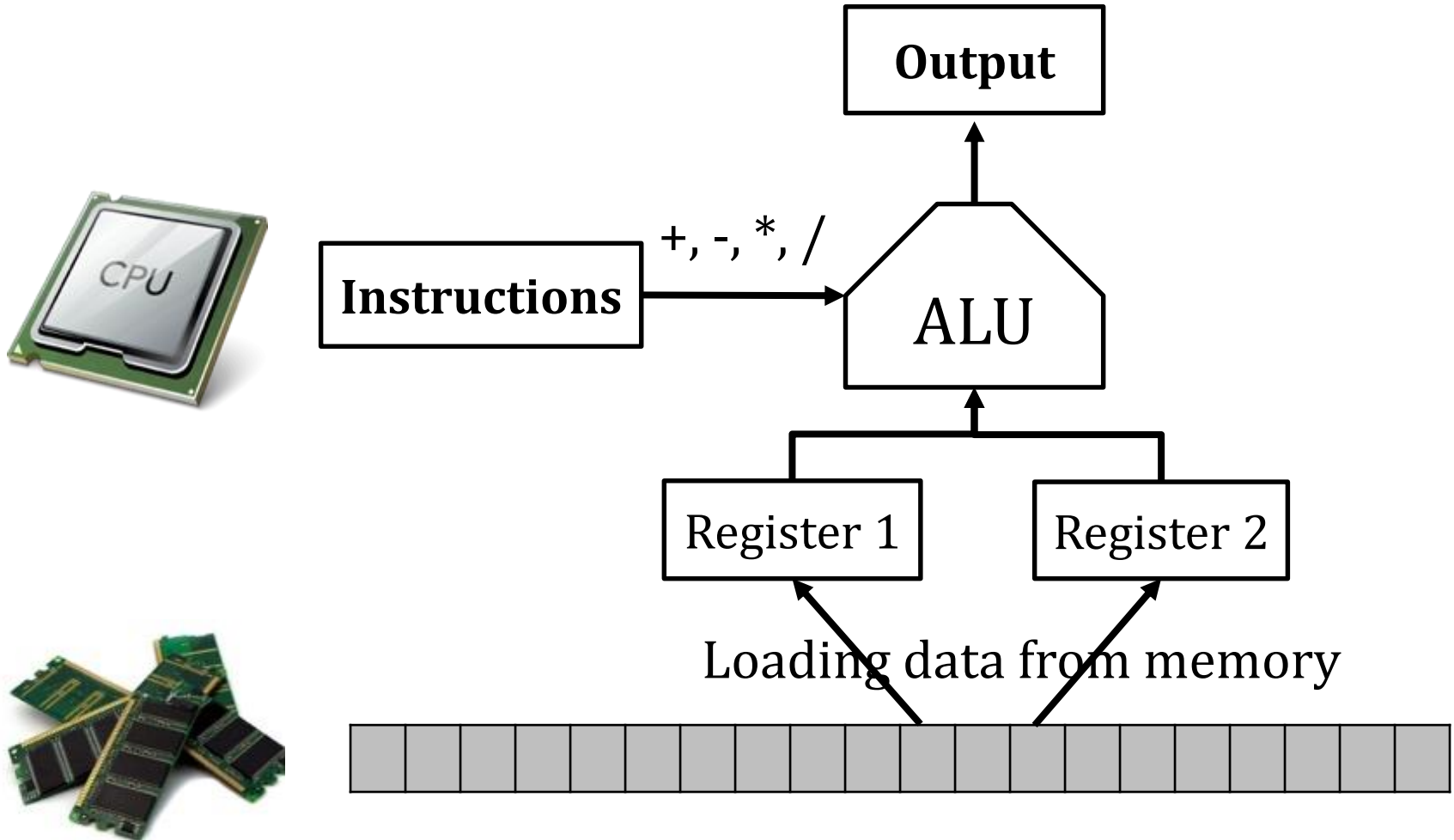
  - ◆ Binary Search Problem

  - ◆ Big O notation

# RAM Computation Model

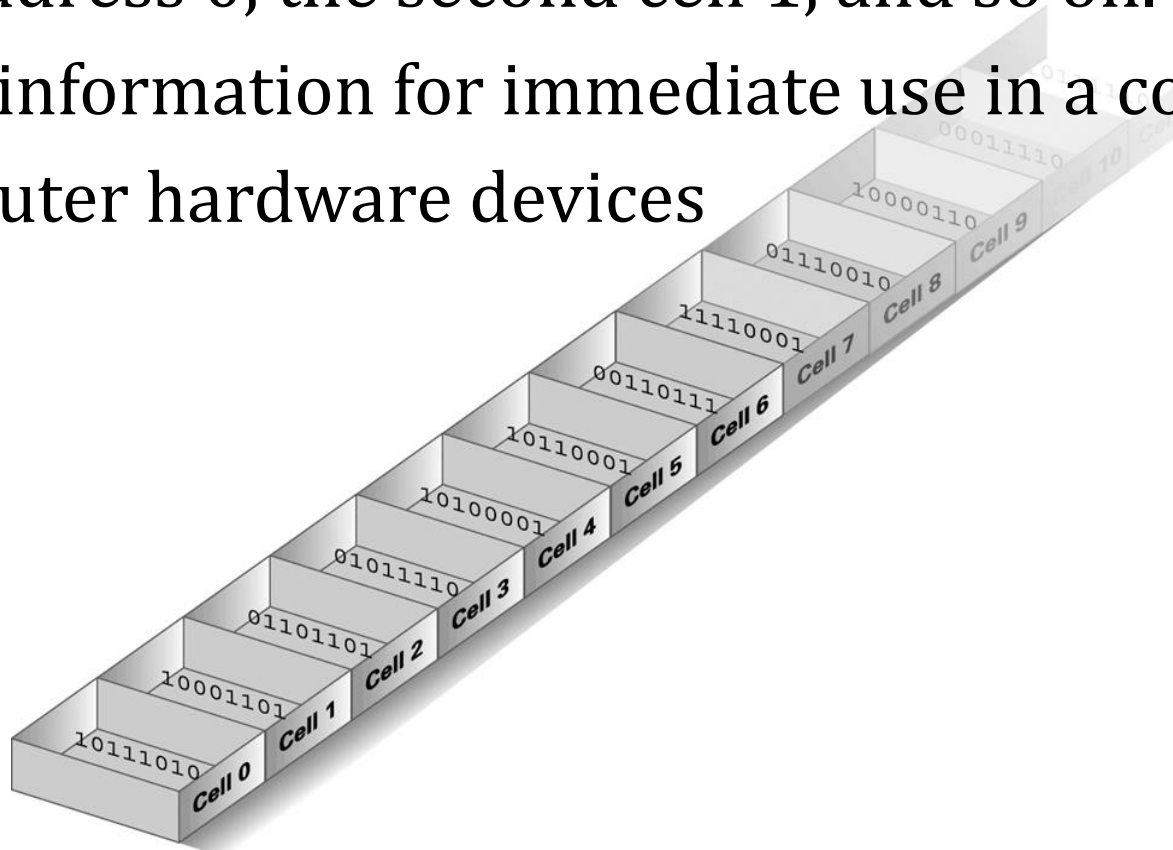


# RAM Computation Model



# Memory

- ❖ A finite sequence of cells, each cell has the same number of bits.
- ❖ Every cell has an address: the first cell of memory has address 0, the second cell 1, and so on.
- ❖ Store information for immediate use in a computer
- ❖ Computer hardware devices



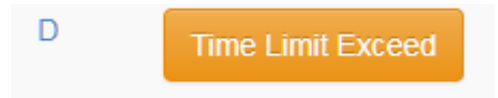
# Center Process Unit (CPU)

- ◆ Contains a fixed number of registers
- ◆ Basic (atomic) operations
  - ◆ **Initialization**
    - ◆ Set a register to a fixed values (e.g., 100, 1000, etc.)
  - ◆ **Arithmetic (ALU)**
    - ◆ Take integers  $a$ ,  $b$  stored in two registers, calculate one of  $\{+, -, *, /\}$  and store the result in a register
  - ◆ **Comparison / Branching**
    - ◆ Take integers  $a$ ,  $b$  stored in two registers, compare them, and learn which of  $\{a < b, a = b, a > b\}$  is true.
  - ◆ **Memory Access**
    - ◆ Take a memory address  $A$  currently stored in a register, Do the READ (i.e., load data from memory) or WRITE (i.e., flush data to memory) operator

# Algorithm Analysis

## ◆ Algorithm

- ◆ A sequence of basic operations



## ◆ Algorithm Analysis



### ◆ Cost analysis

- ◆ Algorithm cost (running time) is the length of the sequences, i.e., the number of basic operations
- ◆ My algorithm is correct, why my submission is **TLE**?
- ◆ Is your algorithm fast?
  - ◆ Focus on the order of growth (how the running time grows for large  $n$ )
- ◆ Unless otherwise stated, we refer algorithm analysis as cost analysis in CS203

# Algorithm Correctness Analysis

D	Wrong Answer
A	Wrong Answer

- ◆ **Correctness analysis**

- ◆ I have passed all test cases, why is still **WA**?
- ◆ It is not enough even if you have tested your algorithm on many instances
  - ◆ Will your algorithm fail on some other instances?
- ◆ Proof your algorithm is correct
- ◆ Guarantee your implementation is correct

- ◆ Software testing is an individual course in other many Universities

- ◆ We will not introduce software testing techniques in this course.



# Example I: Summation

◆ **Problem:** given integer  $n$ , calculate  $1+2+3+\dots+n$

◆ **Algorithm:**

◆ Initialize variable  $a$  to 1,  $b$  to  $n$ ,  $c$  to 0

◆ Repeat the following until  $a > b$ :

◆ Calculate  $c$  plus  $a$ , and store the result to  $c$ .

◆ Calculate  $a$  plus 1, and store the result to  $a$ .

◆ **Cost of the algorithm:**

◆  $3 + n + n + n = 3n + 3$

◆ Which atomic operations are performed?

◆ Algorithm is described by English words

# Example 1: Summation

## ♦ Algorithm:

1. load  $n$  from memory to register  $b$
2. register  $a \leftarrow 1$ ,  $c \leftarrow 0$
3. **repeat**
4.        $c \leftarrow c + a$
5.        $a \leftarrow a + 1$
6. **until**  $a > b$
7. **return**  $c$

- ♦ The above is **pseudocode**, it serves the purpose of express (without **ambiguity**) how our algorithm runs.
- ♦ **Pseudocode** does not rely on any particular programming language

# Example II: Summation

- ◆ **Problem:** given integer  $n$ , calculate  $1+2+3+\dots+n$
- ◆ **Cost** of the above algorithm:  $3n + 3$
- ◆ Can we make it faster?
- ◆ In our middle school math course:
$$1+2+3+\dots+n = (1+n)*n / 2$$

# Example II: Summation

## ♦ Algorithm:

1. load  $n$  from memory to register  $b$
2. register  $a \leftarrow 1$
3.  $a \leftarrow a + b$
4.  $a \leftarrow a * b$
5.  $a \leftarrow a / 2$
6. return  $a$

## ♦ Cost of the algorithm = 5

- ♦ This is significantly faster than the previous algorithm
- ♦ The time of the previous algorithm increases linearly with  $n$
- ♦ The time of this algorithm remains constant with  $n$

# Our Roadmap

- ◆ RAM Computation Model

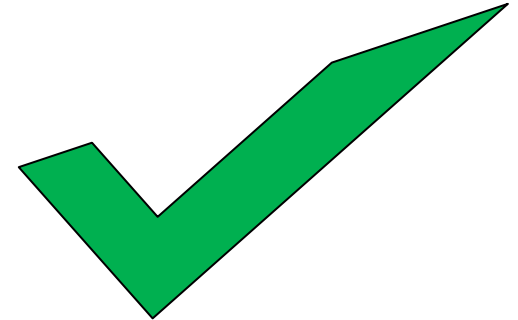
  - ◆ Memory, CPU, Algorithm

  - ◆ Algorithm, Pseudocode

- ◆ Worst Case Analysis

  - ◆ Binary Search Problem

  - ◆ Big O notation



# Search Problem

- ◆ An array  $A$  of  $n$  integers have been sorted in ascending order. Design an algorithm to determine whether given value  $t$  exists in  $A$ .

- ◆ Example

$A$	5	8	10	13	16	19	27	46	51	86
-----	---	---	----	----	----	----	----	----	----	----

- ◆  $t = 16$ , the result is “TRUE”
- ◆  $t = 17$ , the result is “FALSE”

# Search Problem

- ◆ The First Algorithm
  - ◆ Simply read the value of  $A[i]$  for each  $i \in [1, n]$
  - ◆ If any of those cell equals to  $t$ , return “TRUE”, otherwise return “FALSE”
- ◆ Pseudocode:
  1. variable  $i \leftarrow 1$
  2. **Repeat**
  3.     if  $A[i] = t$  then
  4.         return “TRUE”
  5.      $i \leftarrow i + 1$
  6. **until**  $i > n$
  7. **return** “FALSE”

# Running Time of the First Algorithm

<b>A</b>	5	8	10	13	16	19	27	46	51	86
----------	---	---	----	----	----	----	----	----	----	----

- ◆ How much time does the algorithm require?
  - ◆ If  $t$  is 5, the algorithm has running time = 3
  - ◆ If  $t$  is 6, the algorithm has running time =  $4n + 1 = 41$
- ◆ In computer science, it is an art to design algorithms with performance guarantees.
- ◆ What is the largest running time on the worst input with  $n$  integers?



# Worst-Case Running Time

The worst-case running time (or worst case cost) of an algorithm under a problem size  $n$ , is defined to be the largest running time of the algorithm on all the inputs of the same size  $n$ .

# Worst-Case Time of Search Problem

- ◆ Our algorithm has worst-case time

$$f(n) = 4n + 1$$

- ◆ In other words, the algorithm will terminate with a cost at most  **$4n+1$** .
- ◆ This is a performance guarantee on every  **$n$**
- ◆ Can we make it faster?
  - ◆ **Binary search algorithm**

# Binary Search Algorithm

- ◆ We utilize the fact that array  $A$  has been sorted in ascending order.
- ◆ Let us compare  $t$  to the element  $x$  in the middle of  $A$  (i.e.,  $A[n/2]$ )
  - ◆ If  $t = A[n/2]$ , we have found  $t$ , return “TRUE”, terminate
  - ◆ If  $t < A[n/2]$ , we can ignore  $A[n/2+1]$  to  $A[n]$
  - ◆ If  $t > A[n/2]$ , we can ignore  $A[0]$  to  $A[n/2]$
- ◆ In the 2<sup>nd</sup> and 3<sup>rd</sup> cases, we have at most  $n/2$  elements. Then repeat the above on these left elements.

# Binary Search Algorithm

A	5	8	10	13	16	19	27	46	51	86	$t=27$
---	---	---	----	----	----	----	----	----	----	----	--------

A	5	8	10	13	16	19	27	46	51	86	$< t$
---	---	---	----	----	----	----	----	----	----	----	-------

A						19	27	46	51	86	$> t$
---	--	--	--	--	--	----	----	----	----	----	-------

A						19	27	46			$= t$
---	--	--	--	--	--	----	----	----	--	--	-------

# Binary Search Algorithm

## ◆ Binary Search in Pseudocode

```
1.  left  $\leftarrow$  1, right  $\leftarrow$  n
2.  repeat
3.    mid  $\leftarrow$  (left+right)/2
4.    if (t = A[mid]) then
5.      return TRUE
6.    else if (t < A[mid]) then
7.      right  $\leftarrow$  mid -1
8.    else
9.      left  $\leftarrow$  mid + 1
10.  until left > right
11.  return FLASE
```

# Worst-Case Time of Binary Search

- ◆ We call the elements from left to right as surviving elements
- ◆ Line 1: initialization: 2 basic operations
- ◆ Line 2 – 10: iteration, each iteration performs at most 9 basic operations
- ◆ Line 11: termination
- ◆ How many iterations in the algorithm?

# Worst-Case Time of Binary Search

- ◆ How many iterations in the algorithm?
  - ◆ After the **1<sup>st</sup>** iteration, the number of surviving elements is at most  **$n/2$**
  - ◆ After the **2<sup>nd</sup>** iteration, the number of surviving elements is at most  **$n/4$**
  - ◆ In general, after  **$i$ -th** iteration, the number of surviving elements is at most  **$n / 2^i$**
  - ◆ Suppose that there are  **$h$**  iterations in total, it holds that  **$h$**  is the smallest integer satisfying (why?):

$$n / 2^h < 1$$

- ◆ Then,  **$h > \log_2 n \Rightarrow h = 1 + \log_2 n$**
- ◆ Thus, the worst case time of binary search is at most:

$$g(n) = 2 + 9h = 2 + 9(1 + \log_2 n)$$

- ◆ This is a performance guarantee that holds on all values of  **$n$** .

# Search Problem

- ◆ Running time of two algorithms, with input size  $n$ 
  - ◆ Algorithm 1:  $f(n) = 4n + 1$  (operations)
  - ◆ Algorithm 2:  $g(n) = 9\log_2 n + 11$  (operations)
- ◆ Which algorithm is better?
  - ◆ Algorithm 2. Why?
  - ◆ We care about the running time at *large input size*
  - ◆ Constant factors do not affect *the order of growth*



# Asymptotic Analysis

- ◆ Running time of two algorithms, with input size  $n$ 
  - ◆ Algorithm 1:  $f(n) = 4n + 1$  (operations)
  - ◆ Algorithm 2:  $g(n) = 9\log_2 n + 11$  (operations)
- ◆ In computer science, we rarely calculate the time to such a level.
- ◆ We ignore all the constants, but only worry about the dominating term.
  - ◆ Why not constant?  $10n$  VS.  $5n$ ? Which one is faster?
  - ◆ “it depends”,  $10n$  comparison,  $5n$  multiplication
  - ◆ Why dominating term:  $3n$  VS.  $\log_2 n$  ? Which one is faster
  - ◆ “ $\log_2 n$  ” is better than  $3n$  in theoretical computer science

# Big-O notation

- ◆ Let  $f(n)$  and  $g(n)$  be two functions of  $n$ .
- ◆ We say that  $f(n)$  **grows asymptotically no faster than**  $g(n)$  if there is a constant  $c_1 > 0$  such that:
$$f(n) \leq c_1 \cdot g(n)$$
holds for **all**  $n \geq c_2$ .
- ◆ We denote this by  $f(n) = O(g(n))$
- ◆ We say that  $5n$  is considered equally fast as on with  $10n$ , why?
- ◆ Big-O capture this by having both of following true (can you prove that?):

$$10n = O(5n)$$

$$5n = O(10n)$$

# Big-O example

- ◆  $10000\log_2 n$  is considered better than  $n$ . Big-O capture this by having both of following true:

$$10000\log_2 n = O(n)$$

$$n \neq O(10000\log_2 n)$$

- ◆ Proof of  **$10000\log_2 n = O(n)$**
- ◆ There are constants  $c_1 = 1$ ,  $c_2 = 2^{20}$  such that

$$10000\log_2 n \leq c_1 n$$

holds for all  $n \geq c_2$

# Big-O example

- ◆ Proof of  $n \neq O(10000 \log_2 n)$
- ◆ We can prove it by contradiction. Suppose that there are constant  $c_1, c_2$  such that

$$n \leq c_1 \cdot 10000 \log_2 n$$

holds for **all**  $n \geq c_2$ . The above can be rewritten as:

$$\frac{n}{\log_2 n} \leq c_1 \cdot 10000$$

however,  $\frac{n}{\log_2 n}$  tends to be  $\infty$  as  $n$  increases.

Therefore, the inequality cannot hold for **all**  $n \geq c_2$

# Exercise

- ◆ Is  $(5n^2 + 3n) = O(n^2)$  ?
  - ◆ Fix  $c=6$  and  $n_0=3$ , then prove  $f(n) \leq c g(n)$   
[note: other choices also possible]
- ◆ Is  $(5n^2 + 3n) = O(n^3)$  ?
- ◆ Is  $(5n^2 + 3n) = O(n)$  ?

- ◆ Proof the following statements:

$$10000 = O(1)$$

$$100\sqrt{n} + 10n = O(n)$$

$$1000n^{1.5} = O(n^2)$$

$$(\log_2 n)^3 = O(\sqrt{n})$$

$$\log_a n = O(\log_b n) \text{ for } a>1, b>1$$

# Asymptotic Analysis

- ◆ Henceforth, we will describe the running time of an algorithm only in the asymptotical (i.e., big-O) form, which is also called the algorithm's time complexity.
- ◆ Instead of saying the running time of binary search is  $g(n) = 8\log_2 n + 10$ , we will say  $g(n) = O(\log n)$ , which captures the fastest-growing term in the running time. This is also the binary search's time complexity.

# Worst-Case of Algorithms

<i>Complexity</i>		<i>Algorithm</i>
$O(1)$	Constant time	E.g., Compare two numbers
$O(\log n)$	Logarithmic	E.g., Binary search (on a sorted array)
$O(n)$	Linear time	E.g., Search (on a unsorted array)
$O(n \log n)$		E.g., Merge sort
$O(n^2)$	Quadratic	E.g., Selection sort
$O(n^3)$	Cubic	E.g., Matrix multiplication
$O(2^n)$	Exponential	E.g., Brute-force search on boolean satisfiability
$O(n!)$	Factorial	E.g., Brute-force search on traveling salesman

# Big- $\Omega$ notation

- ◆ Let  $f(n)$  and  $g(n)$  be two functions of  $n$ .
- ◆ We say that  $f(n)$  **grows asymptotically no slower than**  $g(n)$  if there is a constant  $c_1 > 0$  such that:

$$f(n) \geq c_1 \cdot g(n)$$

holds for **all**  $n \geq c_2$ .

- ◆ We denote this by  $f(n) = \Omega(g(n))$
- ◆ Examples:
  - ◆  $\log_2 n = \Omega(1)$
  - ◆  $0.001n = \Omega(\sqrt{n})$



# Big- $\Theta$ notation

- ◆ Let  $f(n)$  and  $g(n)$  be two functions of  $n$ .
- ◆ If  $f(n) = O(g(n))$  and  $f(n) = \Omega(g(n))$ , then we define:  $f(n) = \Theta(g(n))$  to indicate  $f(n)$  grows asymptotically as fast as  $g(n)$
- ◆ Examples:
  - ◆  $1000 + 30 \log n + 1.5\sqrt{n} = \Theta(\sqrt{n})$