TRƯỜNG ĐẠI HỌC BÁCH KHOA HÀ NỘI
VIỆN CÔNG NGHỆ THÔNG TIN VÀ TRUYỀN THÔNG

Data structures and Algorithms

Nguyễn Khánh Phương

Computer Science department
School of Information and Communication technology
E-mail: phuongnk@soict.hust.edu.vn

Course outline

TRƯỜNG ĐẠI HỌC BÁCH KHOA HÀ NỘI
VIỆN CÔNG NGHỆ THÔNG TIN VÀ TRUYỀN THÔNG

Chapter 1. Fundamentals

Nguyễn Khánh Phương

Computer Science department
School of Information and Communication technology
E-mail: phuongnk@soict.hust.edu.vn

Contents

Contents

NGUYỄN KHÁNH PHƯƠNG
SOICT– HUST

Example: The maximum subarray problem

- Given an array of n numbers:

  $a_1, a_2, \ldots, a_n$

  The contiguous subarray $a_i, a_{i+1}, \ldots, a_j$ with $1 \le i \le j \le n$ is a subarray of the given array and $\sum_{k=i}^{j} a_k$ *is called as the value of this subarray*

  The task is to find the maximum value of all possible subarrays, *in other words, find the maximum* $\sum_{k=i}^{j} a_k$. The subarray with the maximum value is called as the maximum subarray.

Example: Given the array -2, **11, -4, 13**, -5, 2 then the maximum subarray is 11, -4, 13 with the value = 11+ (-4)+13 =20

➔ This problem can be solved using several different algorithmic techniques, including brute force, divide and conquer, dynamic programming, etc.

1. Introductory example: the max subarray problem

1.1.1. Brute force

1.1.2. Brute force with better implement

1.1.3. Dynamic programming

NGUYỄN KHÁNH PHƯƠNG
SOICT–HUST

1. Introductory example: the max subarray problem

1.1.1. Brute force

1.1.2. Brute force with better implement

1.1.3. Dynamic programming

1.1.1. Brute force algorithm to solve max subarray problem

- The first simple algorithm that one could think about is: browse all possible sub-arrays:

  $a_i, a_{i+1}, \ldots, a_j$ với $1 \le i \le j \le n$,

  then calculate the value of each sub-array in order to find the maximum value.
- The number of all possible sub-arrays:

  $$C(n, 1) + C(n, 2) = n^2/2 + n/2$$

NGUYỄN KHÁNH PHƯƠNG
SOICT–HUST

Brute force algorithm: browse all possible sub-array

Index i	0	1	2	3	4	5
a[i]	-2	11	-4	13	-5	2

```
i = 0: (-2), (-2, 11), (-2,11, -4), (-2,11,-4,13), (-2,11,-4,13,-5), (-2,11,-4,13,-5,2)
i = 1: (11), (11, -4), (11, -4, 13), (11, -4, 13, -5), (11, -4, 13, -5, 2)
i = 2: (-4), (-4, 13), (-4, 13, -5), (-4,13,-5,2)
i = 3: (13), (13,-5), (13, -5,2)
i = 4: (-5), (-5, 2)
i = 5: (2)
```

```
int maxSum = a[0];
for (int i=0; i<n; i++) {
   for (int j=i; j<n; j++) {
      int sum = 0;
      for (int k=i; k<=j; k++)
          sum += a[k];
      if (sum > maxSum)
          maxSum = sum;
   }
}
```

Brute force algorithm: browse all possible sub-array

- **Analyzing time complexity**: we count the number of additions that the algorithm need to perform, it means we count the statement

```
sum += a[k]
```

must perform how many times.

The number of additions:

$$\sum_{i=0}^{n-1}\sum_{j=i}^{n-1}(j-i+1)=\sum_{i=0}^{n-1}(1+2+...+(n-i))=\sum_{i=0}^{n-1}\frac{(n-i)(n-i+1)}{2}$$

$$=\frac{1}{2}\sum_{k=1}^{n}k(k+1)=\frac{1}{2}\left[\sum_{k=1}^{n}k^2+\sum_{k=1}^{n}k\right]=\frac{1}{2}\left[\frac{n(n+1)(2n+1)}{6}+\frac{n(n+1)}{2}\right]$$

$$=\frac{n^3}{6}+\frac{n^2}{2}+\frac{n}{3}$$

```
int maxSum = a[0];
for (int i=0; i<n; i++) {
   for (int j=i; j<n; j++) {
      int sum = 0;
      for (int k=i; k<=j; k++)
          sum += a[k];
      if (sum > maxSum)
          maxSum = sum;
   }
}
```

1. Introductory example: the max subarray problem

1.1.1. Brute force

1.1.2. Brute force with better implement

1.1.3. Dynamic programming

NGUYỄN KHÁNH PHƯƠNG
SOICT– HUST

1.1.2. A better implementation
Brute force algorithm: browse all possible sub-array

Index i	0	1	2	3	4	5
a[i]	-2	11	-4	13	-5	2

i = 0:

(-2), (-2, 11), (-2,11, -4), (-2,11,-4,13), (-2,11,-4,13,-5), (-2,11,-4,13,-5,2)

-2+11 = 9 9

9 + (-4)=5 5

5+13=18 18

18 + (-5)=13 13

We could get the sum of elements from *i* to *j* by just using one addition:

$$\underbrace{\sum_{k=i}^{j} a[k]}_{\text{The sum of elements from } i \text{ to } j} = a[j] + \underbrace{\sum_{k=i}^{j-1} a[k]}_{\text{The sum of elements from } i \text{ to } j-1}$$

1.1.2. A better implementation
Brute force algorithm: browse all possible sub-array

Index i	0	1	2	3	4	5
a[i]	-2	11	-4	13	-5	2

i = 0: (-2), (-2, 11), (-2,11, -4), (-2,11,-4,13), (-2,11,-4,13,-5), (-2,11,-4,13,-5,2)

i = 1: (11), (11, -4), (11, -4, 13), (11, -4, 13, -5), (11, -4, 13, -5, 2)

i = 2: (-4), (-4, 13), (-4, 13, -5), (-4,13,-5,2)

i = 3: (13), (13,-5), (13, -5,2)

i = 4: (-5), (-5, 2)

i = 5: (2)

We could get the sum of elements from *i* to *j* by just using one addition:

$$\underbrace{\sum_{k=i}^{j} a[k]}_{\text{The sum of elements from } i \text{ to } j} = a[j] + \underbrace{\sum_{k=i}^{j-1} a[k]}_{\text{The sum of elements from } i \text{ to } j-1}$$

```
int maxSum = a[0];//or maxSum=-∞;
for (int i=0; i<n; i++) {
   for (int j=i; j<n; j++) {
      int sum = 0;
      for (int k=i; k<=j; k++)
          sum += a[k];
      if (sum > maxSum)
          maxSum = sum;
   }
}
```

```
int maxSum = a[0];
for (int i=0; i<n; i++) {
   int sum = 0;
   for (int j=i; j<n; j++) {
      sum += a[j];
      if (sum > maxSum)
          maxSum = sum;
   }
}
```

1.1.2. A better implementation
Brute force algorithm: browse all possible sub-array

- **A better implementation:**

We could get the sum of elements from *i* to *j* by just using one addition:

$$\underbrace{\sum_{k=i}^{j} a[k]}_{\text{The sum of elements from } i \text{ to } j} = a[j] + \underbrace{\sum_{k=i}^{j-1} a[k]}_{\text{The sum of elements from } i \text{ to } j-1}$$

```
int maxSum = a[0];
for (int i=0; i<n; i++) {
   for (int j=i; j<n; j++) {
      int sum = 0;
      for (int k=i; k<=j; k++)
          sum += a[k];
      if (sum > maxSum)
          maxSum = sum;
   }
}
```

```
int maxSum = a[0];
for (int i=0; i<n; i++) {
   int sum = 0;
   for (int j=i; j<n; j++) {
      sum += a[j];
      if (sum > maxSum)
          maxSum = sum;
   }
}
```

1.1.2. A better implementation
Brute force algorithm: browse all possible sub-array

- **Analyzing time complexity**: we again count the number of additions that the algorithm need to perform, it means we count the statement

```
Sum += a[j]
```

must perform how many times.

The number of additions:

$$\sum_{i=0}^{n-1}(n-i) = n + (n-1) + ... + 1 = \frac{n^2}{2} + \frac{n}{2}$$

This number is exactly the number of all possible sub-arrays ➔ it seems this implementation is good as we examine each subarray exactly once.

```
int maxSum = a[0];
for (int i=0; i<n; i++) {
   int sum = 0;
   for (int j=i; j<n; j++) {
      sum += a[j];
      if (sum > maxSum)
          maxSum = sum;
   }
}
```

Max subarray problem: compare the time complexity between algorithms

The number of additions that the algorithm need to perform:

1.1.1. Brute force $\frac{n^3}{6}+\frac{n^2}{2}+\frac{n}{3}$

1.1.2. Brute force with better implement $\frac{n^2}{2}+\frac{n}{2}$

➔ For the same problem (max subarray), we propose 2 algorithms that requires different number of addition operations, and therefore, they will require different computation time.

The following tables show the computation time of these 2 algorithms with the assumption: the computer could do 10^8 addition operation per second

Complexity	n=10	Time (sec)	n=100	Time (sec)	n=10^4	Time	n=10^6	Time
n^3	10^3	10^{-5}	10^6	10^{-2} sec	10^{12}	2.7 hours	10^{18}	115 days
n^2	100	10^{-6}	10000	10^{-4} sec	10^8	1 sec	10^{12}	2.7 hours

NGUYỄN KHÁNH PHƯƠNG
SOICT– HUST

Max subarray problem: compare the time complexity between algorithms

Complexity	n=10	Time (sec)	n=100	Time (sec)	n=10^4	Time	n=10^6	Time
n^3	10^3	10^{-5}	10^6	10^{-2} sec	10^{12}	2.7 hours	10^{18}	115 days
n^2	100	10^{-6}	10000	10^{-4} sec	10^8	1 sec	10^{12}	2.7 hours

- With small n, the calculation time is negligible.
- The problem becomes more serious when $n > 10^6$. At that time, only the third algorithm is applicable in real time.
- Can we do better?

Yes! It is possible to propose an algorithm that requires only n additions!

NGUYỄN KHÁNH PHƯƠNG
SOICT– HUST

1. Introductory example: the max subarray problem

1.1.1. Brute force

$$\frac{n^3}{6}+\frac{n^2}{2}+\frac{n}{3}$$

1.1.2. Brute force with better implement

$$\frac{n^2}{2}+\frac{n}{2}$$

1.1.3. Dynamic programming

n

1.1.3. Dynamic programming to solve max subarray problem

The primary steps of dynamic programming:

1. Divide: Partition the given problem into sub problems

(Sub problem: have the same structure as the given problem but with smaller size)

2. Note the solution: store the solutions of sub problems in a table
3. Construct the final solution: from the solutions of smaller size problems, try to find the way to construct the solutions of the larger size problems until get the solution of **the given problem (the sub problem with largest size)**

NGUYỄN KHÁNH PHƯƠNG
SOICT– HUST

1.1.3. Dynamic programming to solve max subarray problem

The primary steps of dynamic programming:

1. Divide:

- Define s_i the value of max subarray of the array $a_0, a_1, ..., a_i$, $i = 0, 1, 2, ..., n$-1.
- Clearly, s_{n-1} is the solution.

3. Construct the final solution:

- $s_0 = a_0$ $\quad$ $s_1 = \max\{a_0, a_1, a_0 + a_1\}$
- Assume we already know the value of $s_0, s_1, s_2, ..., s_{i-1}$, i >=1. Now we need to calculate the value of s_i which is the value of max subarray of the array:

  $a_0, a_1, ..., a_{i-1}, a_i$.
- We see that: the max subarray of this array $a_0, a_1, ..., a_{i-1}, a_i$ could either include the element a_i or not include the element a_i ➔ therefore, the max subarray of the array $a_0, a_1, ..., a_{i-1}, a_i$ could only be one of these 2 arrays:
  - The max subarray of the array $a_0, a_1, ..., a_{i-1}$ $= s_{i-1}$
  - The max subarray of the array $a_0, a_1, ..., a_i$ *ending at* a_i. $= e_i$

➔ Thus, we have $s_i = \max\{s_{i-1}, e_i\}$, i = 1,2, ..., n-1.

where e_i is the value of the max subarray $a_0, a_1, ..., a_i$ ending at a_i.

To calculate e_i, *we could use the recursive relation*:

- $e_0 = a_0$;
- $e_i = \max\{a_i, e_{i-1} + a_i\}$, i = 1,2, ..., n-1.

```
MaxSub(a)
{
    smax = a[0];        // smax: the value of max subarray
    ei   = a[0];        //ei    : the value of max subarray ending at a[i]
    for i = 1 to n-1 {
          ei = max{a[i],ei + a[i]};
          smax = max{smax, ei};
    }
}
```

1.1.3. Dynamic programming to solve max subarray problem

```
MaxSub(a)
{
    smax = a[0];            // smax : the value of max subarray
    ei   = a[0];            // ei    : the value of max subarray ending at a[i]
    imax = 0;               // imax  : the index of the last element of the max sub array
    for i = 1 to n-1 {
            u = ei + a[i];
            v = a[i];
            if (u > v) ei = u;
            else ei = v;
            if (ei > smax) {
                smax = ei;
                imax = i;
            }
    }
}
```

```
MaxSub(a)
{
    smax = a[0];        // smax: the value of max subarray
    ei   = a[0];        //ei    : the value of max subarray ending at a[i]
    for i = 1 to n-1 {
          ei = max{a[i],ei + a[i]};
          smax = max{smax, ei};
    }
}
```

Analyzing time complexity:

the number of addition operations need to be performed in the algorithm

= the number of times the statement u = ei + a[i]; need to be executed

= n

NGUYỄN KHÁNH PHƯƠNG
SOICT– HUST

Comparison of 3 algorithms

- The following table shows the estimated running time of the four proposed algorithms above (assuming the computer could perform 10^8 addition operations per second).

Algorithm	Complexity	$n=10^4$	time	$n=10^6$	time
Brute force	n^3	10^{12}	2.7 hours	10^{18}	115 days
Brute force with better implementation	n^2	10^8	1 sec	10^{12}	2.7 hours
Dynamic programming	n	10^4	10^{-4} sec	10^6	$2*10^{-2}$ sec

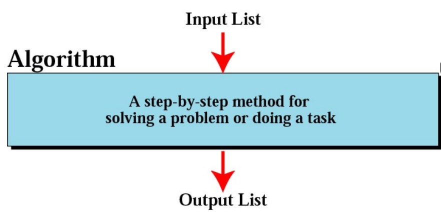
This example shows how the development of effective algorithms could significantly reduce the cost of running time.

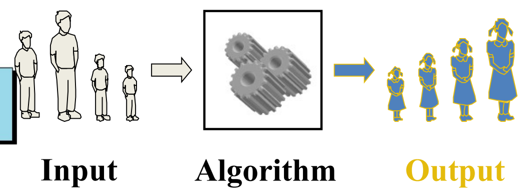
NGUYỄN KHÁNH PHƯƠNG
SOICT–HUST

Contents

Algorithm

- The word *algorithm* comes from the name of a Persian mathematician Abu Ja'far Mohammed ibn-i Musa al Khowarizmi.
- In computer science, this word refers to a special method consisting of a sequence of unambiguous instructions useable by a computer for solution of a problem.
- Informal definition of an algorithm in a computer:





- Example: The problem of finding the largest integer among a number of positive integers
  - Input: the array of n positive integers $a_1, a_2, \ldots, a_n$
  - Output: the largest
  - Example: Input 12 8 13 9 11 ➔ Output: 12
  - Question: Design the algorithm to solve this problem

Algorithm

- All algorithms must satisfy the following criteria:
  (1) **Input**. The algorithm receives data from a certain set.
  (2) **Output**. For each set of input data, the algorithm gives the solution to the problem.
  (3) **Precision**. Each instruction is clear and unambiguous.
  (4) **Finiteness**. If we trace out the instructions of an algorithm, then for all cases, the algorithm terminates after a finite (possibly very large) number of steps.
  (5) **Uniqueness**. The intermediate results of each step of the algorithm are uniquely determined and depend only on the input and the result of the previous steps.
  (6) **Generality.** The algorithm could be applied to solve any problem with a given form

NGUYỄN KHÁNH PHƯƠNG
SOICT– HUST

Comparing Algorithms

- Given 2 or more algorithms to solve the same problem, how do we select the best one?
- Some criteria for selecting an algorithm:
  1) Is it easy to implement, understand, modify?
  2) How long does it take to run it to completion? **TIME**
  3) How much of computer memory does it use? **SPACE**

In this lecture we are interested in the second and third criteria:

- **Time complexity**: The amount of time that an algorithm needs to run to completion
- **Space complexity**: The amount of memory an algorithm needs to run

We will occasionally look at space complexity, but we are mostly interested in time complexity in this course. Thus in this course the better algorithm is the one which runs faster (has smaller time complexity)

How to Calculate Running time

- Most algorithms transform input objects into output objects

- The running time of an algorithm typically grows with the input size
  - Idea: analyze running time as a function of input size
  - Even on inputs of the same size, running time can be very different
    - Example: In order to find the first prime number in an array: the algorithm scans the array from left to right
      - Array 1: **3** 9 8 12 15 20 (algorithm stops when considering the first element)
      - Array 2: 9 8 **3** 12 15 20 (algorithm stops when considering the 3rd element)
      - Array 3: 9 8 12 15 20 **3** (algorithm stops when considering the last element)

  ➔ Idea: analyze running time in the
    - best case
    - worst case
    - average case

Kind of analyses

Best-case:

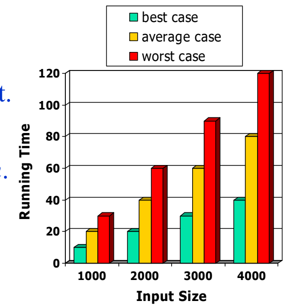
- T(n) = minimum time of algorithm on any input of size n.
- Cheat with a slow algorithm that works fast on some input.

Average-case:

- T(n) = expected time of algorithm over all inputs of size n.
- Need assumption of statistical distribution of inputs
- Very useful but often difficult to determine

Worst-case:

- T(n) = maximum time of algorithm on any input of size n.
- Easier to analyze



To evaluate the running time: 2 ways:

- Experimental evaluation of running time
- Theoretical analysis of running time

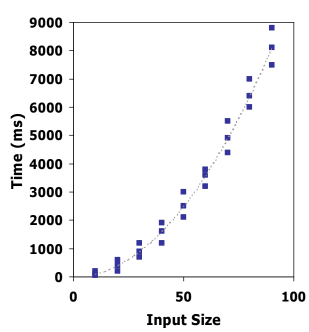
NGUYỄN KHÁNH PHƯƠNG
SOICT– HUST

Experimental Evaluation of Running Time

- Write a program implementing the algorithm
- Run the program with inputs of varying size and composition
- Use a method like clock( ) to get an accurate measure of the actual running time

```
clock_t startTime = clock();
doSomeOperation();
clock_t endTime = clock();
clock_t clockTicksTaken = endTime - startTime;
double timeInSeconds = clockTicksTaken / (double) CLOCKS_PER_SEC;
```

- Plot the results



Limitations of Experiments when evaluating the running time of an algorithm

- Experimental evaluation of running time is very useful but
  - It is necessary to implement the algorithm, which may be difficult
  - Results may not be indicative of the running time on other inputs not included in the experiment
  - In order to compare two algorithms, the same hardware and software environments must be used

➔ We need: **Theoretical Analysis of Running Time**

NGUYỄN KHÁNH PHƯƠNG
SOICT– HUST

Theoretical Analysis of Running Time

- Uses a pseudo-code description of the algorithm instead of an implementation
- Characterizes running time as a function of the input size, ***n***
- Takes into account all possible inputs
- Allows us to evaluate the speed of an algorithm independent of the hardware/software environment (Changing the hardware/software environment affects the running time by a constant factor, but does not alter the growth rate of the running time)

NGUYỄN KHÁNH PHƯƠNG
SOICT– HUST

Contents

1.3. Asymptotic notation

Θ, Ω, O, o, ω

» What these symbols do are:
- give us a notation for talking about how fast a function goes to infinity, which is just what we want to know when we study the running times of algorithms.
- defined for functions over the natural numbers
- used to compare the order of growth of 2 functions

  Example: $f(n) = \Theta(n^2)$: Describes how $f(n)$ grows in comparison to n^2.

» Instead of working out a complicated formula for the exact running time, we can just say that the running time is for example $\Theta(n^2)$ [read as theta of n^2]: that is, the running time is proportional to n^2 plus lower order terms. For most purposes, that's just what we want to know.

Θ - Theta notation

- For a given function $g(n)$, we denote by $\mathbf{\Theta}(g(n))$ the set of functions

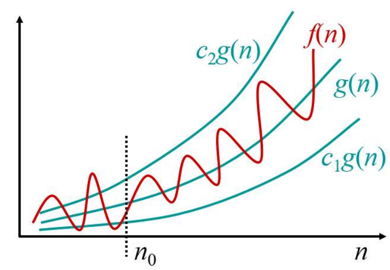
$$\Theta(g(n)) = \left\{ \begin{array}{l} f(n) : \text{there exist positive constants } c_1, c_2, \text{and } n_0 \text{ s.t.} \\ \quad 0 \le c_1 g(n) \le f(n) \le c_2 g(n) \text{ for all } n \ge n_0 \end{array} \right\}$$

Intuitively: Set of all functions that have the same *rate of growth* as $g(n)$.

- A function $f(n)$ belongs to the set $\mathbf{\Theta}(g(n))$ if there exist positive constants c_1 and c_2 such that it can be "sand- wiched" between $c_1g(n)$ and $c_2g(n)$ for sufficienly large n
  - $f(n) = \mathbf{\Theta}(g(n))$ means that there exists some constant c_1 and c_2 s.t.

$$c_1 g(n) \le f(n) \le c_2 g(n) \text{ for large enough } n.$$

- When we say that one function is theta of another, we mean that neither function goes to infinity faster than the other.



$f(n) = \Theta(g(n))$ ⟺ $\exists c_1, c_2, n_0 > 0 \; : \; \forall n \ge n_0, \quad c_1 g(n) \le f(n) \le c_2 g(n)$

Example 1: Show that $10n^2 - 3n = \mathbf{\Theta}(n^2)$

- With which values of the constants n_0, c_1, c_2 then the inequality in the definition of the theta notation is correct:

$$c_1 n^2 \le f(n) = 10n^2 - 3n \le c_2 n^2 \; \forall n \ge n_0$$

- Suggestion: Make c_1 a little smaller than the leading (the highest) coefficient, and c_2 a little bigger.

➔ Select: $c_1 = 1$, $c_2 = 11$, $n_0 = 1$ then we have

$$n^2 \le 10n^2 - 3n \le 11n^2, \textit{ with } n \ge 1.$$

➔ $\forall n \ge 1$: $10n^2 - 3n = \mathbf{\Theta}(n^2)$

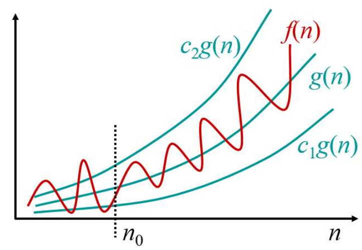
- Note: For polynomial functions: To compare the growth rate, it is necessary to look at the term with the highest coefficient

$f(n) = \Theta(g(n))$ $\Longleftrightarrow$ $\exists c_1, c_2, n_0 > 0 \ : \ \forall n \geq n_0, \ \ c_1 g(n) \leq f(n) \leq c_2 g(n)$

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

We must find n_0, c_1 and c_2 such that

$$c_1 n^2 \leq f(n) = \frac{1}{2}n^2 - 3n \leq c_2 n^2 \ \forall n \geq n_0$$

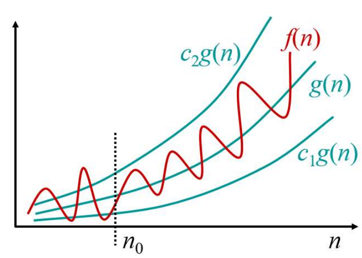


$f(n) = \Theta(g(n))$ $\Longleftrightarrow$ $\exists c_1, c_2, n_0 > 0 \ : \ \forall n \geq n_0, \ \ c_1 g(n) \leq f(n) \leq c_2 g(n)$

Example 3: Show that $f(n) = 23n^3 - 10\, n^2 \log_2 n + 7n + 6 = \boldsymbol{\Theta}(n^3)$

We must find n_0, c_1 and c_2 such that

$$c_1 n^3 \leq f(n) = 23n^3 - 10\, n^2 \log_2 n + 7n + 6 \leq c_2 n^3 \forall n \geq n_0$$



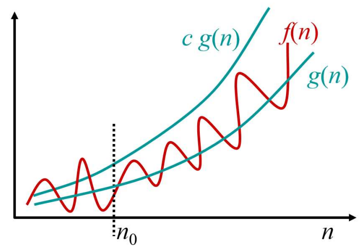
O - big Oh notation

For a given function $g(n)$, we denote by $O(g(n))$ the set of functions

$$O(g(n)) = \left\{ \begin{array}{c} f(n): \text{there exist positive constants } c \text{ and } n_0 \text{ s.t.} \\ 0 \le f(n) \le cg(n) \text{ for all } n \ge n_0 \end{array} \right\}$$

Intuitively: Set of all functions whose *rate of growth* is the same as or lower than that of $g(n)$.

- We say: $g(n)$ is asymptotic upper bound of the function $f(n)$, to within a constant factor, and write $f(n) = O(g(n))$.
- $f(n) = O(g(n))$ means that there exists some constant c such that $f(n)$ is always $\le cg(n)$ for large enough n.
- $O(g(n))$ is the set of functions that go to infinity no faster than $g(n)$.



Graphic Illustration

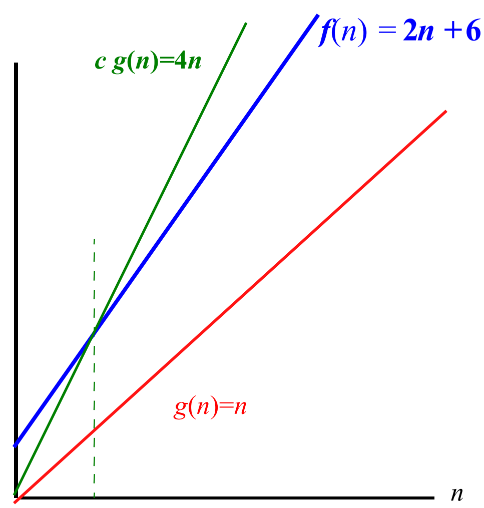
$O(g(n))$ = {$f(n)$: $\exists$ positive constants c and n_0, such that $\forall n \ge n_0$, we have $0 \le f(n) \le cg(n)$ }

- $f(n) = 2n+6$
- Conf. def:
  - Need to find a function $g(n)$ and constants c and n_0 such as $f(n) < cg(n)$ when $n>n_0$

→ $g(n) = n$, $c = 4$ and $n_0=3$

→ $f(n)$ is $O(n)$

The order of $f(n)$ is n



Big-Oh Examples

$O(g(n))$ = {$f(n)$: ∃ positive constants c and n_0, such that ∀$n \geq n_0$, we have $0 \leq f(n) \leq cg(n)$ }

- Example 1: Show that $2n + 10 = O(n)$

➔ $f(n) = 2n+10$, $g(n) = n$
  - Need constants c and n_0 such that $2n + 10 \leq cn$ for $n \geq n_0$
  - $(c - 2)\, n \geq 10$
  - $n \geq 10/(c - 2)$
  - Pick $c = 3$ and $n_0 = 10$
- Example 2: Show that $7n$-2 is $O(n)$

➔ $f(n) = 7n$-2, $g(n) = n$
  - Need constants c and n_0 such that $7n - 2 \leq cn$ for $n \geq n_0$
  - $(7 - c)\, n \leq 2$
  - $n \leq 2/(7 - c)$
  - Pick $c = 7$ and $n_0 = 1$

NGUYỄN KHÁNH PHƯƠNG
SOICT–HUST

Note

- The values of positive constants n_0 and c **are not unique** when proof the asymptotic formulas
- Example: show that $100n + 5 = O(n^2)$
  - $100n + 5 \leq 100n + n = 101n \leq 101n^2 \quad \forall\, n \geq 5$

    $n_0 = 5$ and c = 101 are constants need to determine
  - $100n + 5 \leq 100n + 5n = 105n \leq 105n^2\ \forall\, n \geq 1$

    $n_0 = 1$ and $c = 105$ are also constants need to determine
- Only need to find **some** positive constants c and n_0 satisfying the equality in the definition of asymptotic notation

Big-Oh Examples

$O(g(n)) = \{f(n) : \exists$ positive constants c and n_0, such that $\forall n \geq n_0$, we have $0 \leq f(n) \leq cg(n)$ $\}$

- Example 3: Show that $3n^3 + 20n^2 + 5$ is $O(n^3)$

  Need constants c and n_0 such that $3n^3 + 20n^2 + 5 \leq \boldsymbol{cn^3}$ for $n \geq n_0$

  ……
- Example 4: Show that $3 \log n + 5$ is $O(\log n)$

  Need constants c and n_0 such that $3 \log n + 5 \leq c \log n$ for $n \geq n_0$

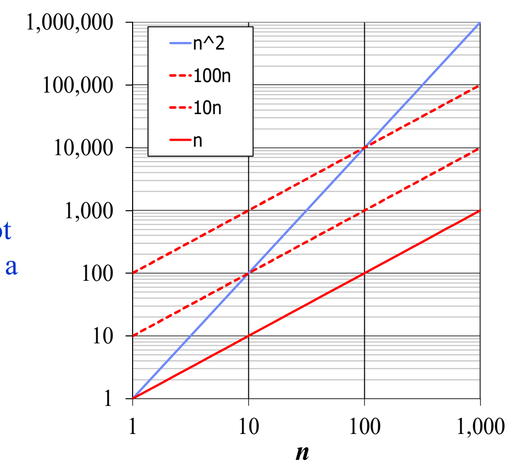
  ……

NGUYỄN KHÁNH PHƯƠNG
SOICT–HUST

Big-Oh Examples

$O(g(n)) = \{f(n) : \exists$ positive constants c and n_0, such that $\forall n \geq n_0$, we have $0 \leq f(n) \leq cg(n)$ $\}$

- Example 5: the function $\boldsymbol{n^2}$ is not $\boldsymbol{O(n)}$
  - $\boldsymbol{n^2 \leq cn}$
  - $\boldsymbol{n \leq c}$
  - The above inequality cannot be satisfied since $\boldsymbol{c}$ must be a constant



Big-Oh and Growth Rate

- The big-Oh notation gives an upper bound on the growth rate of a function
- The statement "$f(n)$ is $O(g(n))$" means that the growth rate of $f(n)$ is no more than the growth rate of $g(n)$
- We can use the big-Oh notation to rank functions according to their growth rate

	$f(n)$ is $O(g(n))$	$g(n)$ is $O(f(n))$
$g(n)$ grows more	Yes	No
$f(n)$ grows more	No	Yes
Same growth	Yes	Yes

NGUYỄN KHÁNH PHƯƠNG
SOICT– HUST

Inappropriate Expressions

$$f(n) \cancel{\leq} O(g(n))$$

$$f(n) \cancel{\geq} O(g(n))$$

Big-Oh Examples

- $50n^3 + 20n + 4$ is $O(n^3)$
  - Would be correct to say is $O(n^3+n)$
    - Not useful, as n^3 exceeds by far n, for large values
  - Would be correct to say is $O(n^5)$
    - OK, but $g(n)$ should be as close as possible to $f(n)$
- $3\log(n) + \log(\log(n)) = O(\ ?\ )$

•Simple Rule: Drop lower order terms and constant factors

NGUYỄN KHÁNH PHƯƠNG
SOICT– HUST

Useful Big-Oh Rules

- If $f(n)$ is a polynomial of degree d: $f(n) = a_0 + a_1 n + a_2 n^2 + \ldots + a_d n^d$

then $f(n)$ is $O(n^d)$, i.e.,

1. Drop lower-order terms
2. Drop constant factors

Example: $3n^3 + 20n^2 + 5$ is $O(n^3)$

- If $f(n) = O(n^k)$ then $f(n) = O(n^p)$ with $\forall\ p > k$

Example: $2n^2 = O(n^2)$ then $2n^2 = O(n^3)$

When evaluate asymptotic $f(n) = O(g(n))$, we want to find function $g(n)$ with a slower growth rate as possible

- Use the smallest possible class of functions

Example: Say "$2n$ is $O(n)$" instead of "$2n$ is $O(n^2)$"

- Use the simplest expression of the class

Example: Say "$3n + 5$ is $O(n)$" instead of "$3n + 5$ is $O(3n)$"

NGUYỄN KHÁNH PHƯƠNG
SOICT– HUST

O Notation Examples

- All these expressions are O(n):
  - n, $3n$, $61n + 5$, $22n – 5$, …
- All these expressions are O(n^2):
  - n^2, $9 n^2$, $18 n^2+ 4n – 53$, …
- All these expressions are O($n \log n$):
  - $n(\log n)$, $5n(\log 99n)$, $18 + (4n – 2)(\log (5n + 3))$, …

NGUYỄN KHÁNH PHƯƠNG
SOICT– HUST

Properties

- If $f(n)$ is $O(g(n))$ then $af(n)$ is $O(g(n))$ for any a
- If $f(n)$ is $O(g_1(n))$ and $h(n)$ is $O(g_2(n))$ then
  - $f(n)+h(n)$ is $O(g_1(n)+g_2(n))$
  - $f(n)h(n)$ is $O(g_1(n)\, g_2(n))$
- If $f(n)$ is $O(g(n))$ and $g(n)$ is $O(h(n))$ then $f(n)$ is $O(h(n))$
- If $p(n)$ is a polynomial in n then $\log p(n)$ is $O(\log(n))$
- If $p(n)$ is a polynomial of degree d, then $p(n)$ is $O(n^d)$
- $n^x = O(a^n)$, for any fixed $x > 0$ and $a > 1$
  - An algorithm of order n to a certain power is better than an algorithm of order a (> 1) to the power of n
- $\log n^x$ is $O(\log n)$, for $x > 0$ – how?
- $\log^x n$ is $O(n^y)$ for $x > 0$ and $y > 0$
  - An algorithm of order $\log n$ (to a certain power) is better than an algorithm of n raised to a power y.

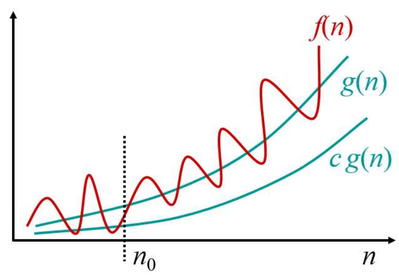
Ω-Omega notation

- For a given function $g(n)$, we denote by $\Omega(g(n))$ the set of functions

$$\Omega(g(n)) = \left\{ \begin{array}{l} f(n) : \text{there exist positive constants } c \text{ and } n_0 \text{ s.t.} \\ \quad 0 \le cg(n) \le f(n) \text{ for all } n \ge n_0 \end{array} \right\}$$

Intuitively: Set of all functions whose *rate of growth* is the same as or higher than that of $g(n)$.

- We say: $g(n)$ is asymptotic lower bound of the function $f(n)$, to within a constant factor, and write $f(n) = \Omega(g(n))$.
- $f(n) = \Omega(g(n))$ means that there exists some constant c such that $f(n)$ is always $\ge cg(n)$ for large enough n.
- $\Omega(g(n))$ is the set of functions that go to infinity no slower than $g(n)$



Omega Examples

$\Omega(g(n))$ = {$f(n)$: ∃ positive constants c and n_0, such that $\forall n \ge n_0$, we have $0 \le cg(n) \le f(n)$}

- Example 1: Show that $5n^2$ is $\Omega(n)$

  Need constants c and n_0 such that $\boldsymbol{cn} \le 5n^2$ for $n \ge n_0$

  this is true for $c = 1$ and $n_0 = 1$

  Comment:

  - If $f(n) = \Omega(n^k)$ then $f(n) = \Omega(n^p)$ with $\forall\ p < k$.
    - Example: If $f(n) = \Omega(n^5)$ then $f(n) = \Omega(n)$
  - When evaluate asymptotic $f(n) = \Omega(g(n))$, we want to find function $g(n)$ with a faster growth rate as possible

Asymptotic notation in equations

Another way we use asymptotic notation is to simplify calculations:

- Use asymptotic notation in equations to replace expressions containing lower-order terms

Example:

$$4n^3 + 3n^2 + 2n + 1 = 4n^3 + 3n^2 + \Theta(n)$$
$$= 4n^3 + \Theta(n^2) = \Theta(n^3)$$

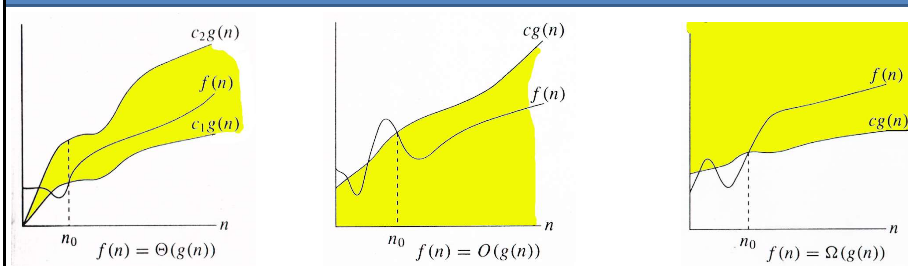
How to interpret?

In equations, $\Theta(f(n))$ always stands for an ***anonymous function*** $g(n) \in \Theta(f(n))$

– In this example, we use $\Theta(n^2)$ stands for $3n^2 + 2n + 1$

NGUYỄN KHÁNH PHƯƠNG
SOICT– HUST

Asymptotic notation



Graphic examples of $\mathbf{\Theta}$, O, and $\mathbf{\Omega}$

Theorem: For any two functions $f(n)$ and $g(n)$, we have $f(n) = \mathbf{\Theta}(g(n))$ if and only if

$$f(n) = O(g(n)) \text{ and } f(n) = \Omega(g(n))$$

Theorem: For any two functions $f(n)$ and $g(n)$, we have $f(n) = \Theta(g(n))$ if and only if $f(n) = O(g(n))$ and $f(n) = \Omega(g(n))$

Example 1: Show that $f(n) = 5n^2 = \Theta(n^2)$

Because:

- $5n^2 = O(n^2)$

  $f(n)$ is $O(g(n))$ if there is a constant $c > 0$ and an integer constant $n_0 \geq 1$ such that $f(n) \leq cg(n)$ for $n \geq n_0$

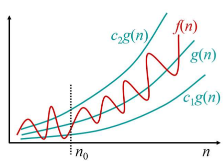
  let $c = 5$ and $n_0 = 1$

- $5n^2 = \Omega(n^2)$

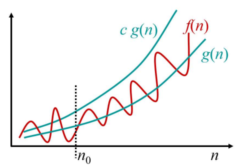
  $f(n)$ is $\Omega(g(n))$ if there is a constant $c > 0$ and an integer constant $n_0 \geq 1$ such that $f(n) \geq cg(n)$ for $n \geq n_0$

  let $c = 5$ and $n_0 = 1$

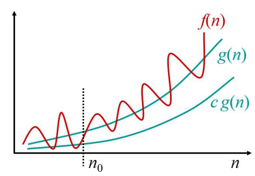
Therefore: $f(n) = \Theta(n^2)$



$f(n) = \Theta(g(n))$



$f(n) = O(g(n))$



$f(n) = \Omega(g(n))$

Theorem: For any two functions $f(n)$ and $g(n)$, we have $f(n) = \Theta(g(n))$ if and only if $f(n) = O(g(n))$ and $f(n) = \Omega(g(n))$

Example 2: Show that $f(n) = 3n^2 - 2n + 5 = \Theta(n^2)$

Because:

$3n^2 - 2n + 5 = O(n^2)$

$f(n)$ is $O(g(n))$ if there is a constant $c > 0$ and an integer constant $n_0 \geq 1$ such that $f(n) \leq cg(n)$ for $n \geq n_0$

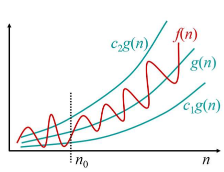
➔ pick $c = ?$ and $n_0 = ?$

$3n^2 - 2n + 5 = \Omega(n^2)$

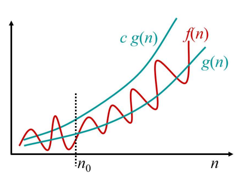
$f(n)$ is $\Omega(g(n))$ if there is a constant $c > 0$ and an integer constant $n_0 \geq 1$ such that $f(n) \geq cg(n)$ for $n \geq n_0$

➔ pick $c = ?$ and $n_0 = ?$

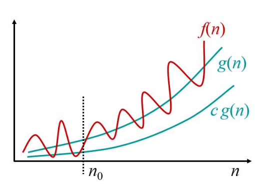
Therefore: $f(n) = \Theta(n^2)$



$f(n) = \Theta(g(n))$



$f(n) = O(g(n))$



$f(n) = \Omega(g(n))$

Exercise 1

Show that: $100n + 5 \neq \Omega(n^2)$

Ans: Contradiction

- Assume: $100n + 5 = \Omega(n^2)$

➔ $\exists\ c, n_0$ such that: $0 \leq cn^2 \leq 100n + 5$

- We have: $100n + 5 \leq 100n + 5n = 105n \quad \forall\ n \geq 1$
- Therefore: $cn^2 \leq 105n \Rightarrow n(cn - 105) \leq 0$
- As $n > 0 \Rightarrow cn - 105 \leq 0 \Rightarrow n \leq 105/c$

The above inequality cannot be satisfied since ***c*** must be a constant

Exercise 2

Show that: $n \neq \Theta(n^2)$

Ans: Contradiction

- Assume: $n = \Theta(n^2)$

Exercise 3:Show that

a) $6n^3 \neq \Theta(n^2)$

Ans: Contradiction

– Assume: $6n^3 = \Theta(n^2)$

b) $n \neq \Theta(\log_2 n)$

Ans: Contradiction

– Assume: $n = \Theta(\log_2 n)$

NGUYỄN KHÁNH PHƯƠNG
SOICT–HUST

The way to talk about the running time

- When people say "The running time for this algorithm is $O(f(n))$", it means that **the worst case running time is $O(f(n))$** (that is, no worse than $c*f(n)$ for large n, since big Oh notation gives an upper bound).
  - It means the worst case running time could be determined by some function $g(n) \in O(f(n))$

$$O(f(n)) = \left\{ \begin{array}{l} g(n): \text{there exist positive constants } c \text{ and } n_0 \text{ s.t.} \\ \quad 0 \le g(n) \le cf(n) \text{ for all } n \ge n_0 \end{array} \right\}$$

- When people say "The running time for this algorithm is $\Omega(f(n))$", it means that **the best case running time is $\Omega(f(n))$** (that is, no better than $c*f(n)$ for large n, since big Omega notation gives a lower bound).
  - It means the best case running time could be determined by some function $g(n) \in \Omega(f(n))$

$$\Omega(f(n)) = \left\{ \begin{array}{l} g(n): \text{there exist positive constants } c \text{ and } n_0 \text{ s.t.} \\ \quad 0 \le cf(n) \le g(n) \text{ for all } n \ge n_0 \end{array} \right\}$$

o- Little oh notation

- For a given function $g(n)$, we denote by $o(g(n))$ the set of functions

$$o(g(n)) = \left\{ \begin{array}{l} f(n): \text{there exist positive constants } c \text{ and } n_0 \text{ s.t.} \\ \quad 0 \le f(n) < cg(n) \text{ for all } n \ge n_0 \end{array} \right\}$$

$f(n)$ becomes insignificant relative to $g(n)$ as n approaches infinity:

$$\lim_{n \to \infty} [f(n) / g(n)] = 0$$

$g(n)$ is an ***upper bound*** for $f(n)$ that is not asymptotically tight.

NGUYỄN KHÁNH PHƯƠNG
SOICT–HUST

ω - Little omega notation

- For a given function $g(n)$, we denote by $o(g(n))$ the set of functions

$$\omega(g(n)) = \left\{ \begin{array}{l} f(n): \text{there exist positive constants } c \text{ and } n_0 \text{ s.t.} \\ \quad 0 \le cg(n) < f(n) \text{ for all } n \ge n_0 \end{array} \right\}$$

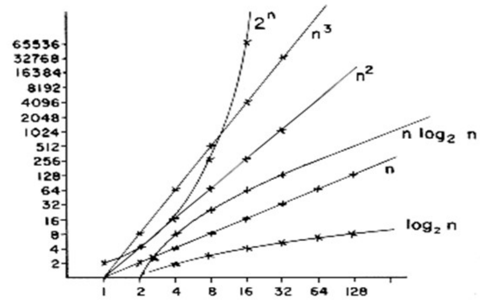
$f(n)$ becomes arbitrarily large relative to $g(n)$ as n approaches infinity:

$$\lim_{n \to \infty} [f(n) / g(n)] = \infty$$

$g(n)$ is a ***lower bound*** for $f(n)$ that is not asymptotically tight.

Basic functions

- Often appear in algorithm analysis:
  - Constant ≈ *1*
  - Logarithmic ≈ $\log_2 n$
  - Linear ≈ n
  - N-Log-N ≈ $n \log_2 n$
  - Quadratic ≈ n^2
  - Cubic ≈ n^3
  - Exponential ≈ 2^n



Let's practice classifying functions

NGUYỄN KHÁNH PHƯƠNG
SOICT– HUST

Basic Functions

Which are more alike ?

$$\underbrace{n^{1000} \qquad n^2}_{\text{polynomial}} \qquad\qquad 2^n$$

Basic Functions

Which are more alike ?

$$\underbrace{1000n^2 \qquad\qquad 3n^2}_{\text{quadratic}} \qquad\qquad\qquad 2n^3$$

Basic functions growth rates

n	$\log n$	n	$n\log n$	n^2	n^3	2^n
4	2	4	8	16	64	16
8	3	8	24	64	512	256
16	4	16	64	256	4,096	65,536
32	5	32	160	1,024	32,768	4,294,967,296
64	6	64	384	4,094	262,144	$1.84 * 10^{19}$
128	7	128	896	16,384	2,097,152	$3.40 * 10^{38}$
256	8	256	2,048	65,536	16,777,216	$1.15 * 10^{77}$
512	9	512	4,608	262,144	134,217,728	$1.34 * 10^{154}$
1024	10	1,024	10,240	1,048,576	1,073,741,824	$1.79 * 10^{308}$

Algorithm Types

- Time takes to solve an instance of a
  - Linear Algorithm is
    - Never greater than ***c*n***
  - Quadratic Algorithm is
    - Never greater than $c*n^2$
  - Cubic Algorithm is
    - Never greater than $c*n^3$
  - Polynomial Algorithm is
    - Never greater than n^k
  - Exponential Algorithm is
    - Never greater than c^n

  where c & k are appropriate constants

NGUYỄN KHÁNH PHƯƠNG
SOICT– HUST

The way to remember these notations

Theta	$f(n) = \Theta(g(n))$	$f(n) \approx c\, g(n)$
Big Oh	$f(n) = O(g(n))$	$f(n) \leq c\, g(n)$
Big Omega	$f(n) = \Omega(g(n))$	$f(n) \geq c\, g(n)$
Little Oh	$f(n) = o(g(n))$	$f(n) \ll c\, g(n)$
Little Omega	$f(n) = \omega(g(n))$	$f(n) \gg c\, g(n)$

NGUYỄN KHÁNH PHƯƠNG
SOICT– HUST

The analogy between comparing functions and comparing numbers

One thing you may have noticed by now is that these relations are kind of like the “<, >” relations for the numbers

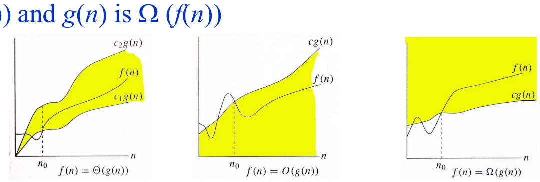
$$f \leftrightarrow g \;\approx\; a \leftrightarrow b$$

$$\begin{aligned} f(n) &= \Theta(g(n)) \;\approx\; a = b \\ f(n) &= O(g(n)) \;\approx\; a \leq b \\ f(n) &= \Omega(g(n)) \;\approx\; a \geq b \\ f(n) &= o(g(n)) \;\approx\; a < b \\ f(n) &= \omega(g(n)) \;\approx\; a > b \end{aligned}$$

Theta	$f(n) = \Theta(g(n))$	$f(n) \approx c\, g(n)$
Big Oh	$f(n) = O(g(n))$	$f(n) \leq c\, g(n)$
Big Omega	$f(n) = \Omega(g(n))$	$f(n) \geq c\, g(n)$
Little Oh	$f(n) = o(g(n))$	$f(n) \ll c\, g(n)$
Little Omega	$f(n) = \omega(g(n))$	$f(n) \gg c\, g(n)$

“Relatives” of notations

- “Relatives” of the Big-Oh
  - $\Omega\,(g(n))$: Big Omega – asymptotic *lower* bound
  - $\Theta\,(g(n))$: Big Theta – asymptotic *tight* bound
- Big-Omega – think of it as the inverse of $O(n)$
  - $f(n)$ is $\Omega\,(g(n))$ if $g(n)$ is $O(f(n))$
- Big-Theta – combine both Big-Oh and Big-Omega
  - $f(n)$ is $\Theta\,(g(n))$ if $f(n)$ is $O(g(n))$ and $g(n)$ is $\Omega\,(f(n))$
- Make the difference:
  - $3n+3$ is $O(n)$ and is $\Theta\,(n)$
  - $3n+3$ is $O(n^2)$ but is not $\Theta\,(n^2)$



- Little-oh – $f(n)$ is $o(g(n))$ if $f(n)$ is $O(g(n))$ and $f(n)$ is not $\Theta\,(g(n))$
  - $2n+3$ is $o(n^2)$
  - $2n + 3$ is $o(n)$?

NGUYỄN KHÁNH PHƯƠNG
SOICT– HUST

Math you need to Review

- Exponents:
  $a^{(b+c)} = a^b a^c$
  $a^{bc} = (a^b)^c$
  $a^b / a^c = a^{(b-c)}$
  $b = a^{\log_a b}$
  $b^c = a^{c*\log_a b}$
- Logarithms:

  $x = \log_b a$ is the exponent for $a = b^x$.

  Natural log: $\ln a = \log_e a$
  Binary log: $\lg a = \log_2 a$

  $\lg^2 a = (\lg a)^2$
  $\lg \lg a = \lg (\lg a)$

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

Logarithms and exponentials – Bases

- If the base of a logarithm is changed from one constant to another, the value is altered by a constant factor.
  - **Ex:** $\log_{10} n * \mathbf{\log_2 10} = \log_2 n$.
  - Base of logarithm is not an issue in asymptotic notation.
- Exponentials with different bases differ by a exponential factor (not a constant factor).
  - **Ex:** $2^n = \mathbf{(2/3)^n} * 3^n$.

NGUYỄN KHÁNH PHƯƠNG
SOICT– HUST

Exercise

- Order the following functions by their asymptotic growth rates
  1. $n\log_2 n$
  2. $\log_2 n^3$
  3. n^2
  4. $n^{2/5}$
  5. $2^{\log_2 n}$
  6. $\log_2(\log_2 n)$
  7. $\mathrm{Sqr}(\log_2 n)$

NGUYỄN KHÁNH PHƯƠNG
SOICT–HUST

Properties

- Transitivity (truyền ứng)

$$f(n) = \Theta(g(n)) \;\&\; g(n) = \Theta(h(n)) \Rightarrow f(n) = \Theta(h(n))$$
$$f(n) = O(g(n)) \;\&\; g(n) = O(h(n)) \Rightarrow f(n) = O(h(n))$$
$$f(n) = \Omega(g(n)) \;\&\; g(n) = \Omega(h(n)) \Rightarrow f(n) = \Omega(h(n))$$

- Reflexivity

$$f(n) = \Theta(f(n)) \qquad f(n) = O(g(n)) \qquad f(n) = \Omega(g(n))$$

- Symmetry (đối xứng)

$f(n) = \Theta(g(n))$ if and only if $g(n) = \Theta(f(n))$

- Transpose Symmetry (Đối xứng chuyển vị)

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

Example: $A = 5n^2 + 100n$, $B = 3n^2 + 2$. Show that $A \in \Theta(B)$

Ans: $A \in \Theta(n^2)$, $n^2 \in \Theta(B) \Rightarrow A \in \Theta(B)$

Limits

- $\lim_{n\to\infty} [f(n) / g(n)] = 0 \Rightarrow f(n) \in o(g(n))$
- $\lim_{n\to\infty} [f(n) / g(n)] < \infty \Rightarrow f(n) \in O(g(n))$
- $0 < \lim_{n\to\infty} [f(n) / g(n)] < \infty \Rightarrow f(n) \in \Theta(g(n))$
- $0 < \lim_{n\to\infty} [f(n) / g(n)] \Rightarrow f(n) \in \Omega(g(n))$
- $\lim_{n\to\infty} [f(n) / g(n)] = \infty \Rightarrow f(n) \in \omega(g(n))$
- $\lim_{n\to\infty} [f(n) / g(n)]$ undefined $\Rightarrow$ can't say

Exercise: Express functions in A in asymptotic notation using functions in B.

A	B	
$\log_3(n^2)$	$\log_2(n^3)$	$A \in \Theta(B)$
$\log_b a = \log_c a / \log_c b$; $A = 2\lg n / \lg 3$, $B = 3\lg n$, $A/B = 2/(3\lg 3) \Rightarrow A \in \Theta(B)$		
$n^{\lg 4}$	$3^{\lg n}$	$A \in \omega(B)$
$a^{\log b} = b^{\log a}$; $B = 3^{\lg n} = n^{\lg 3}$; $A/B = n^{\lg(4/3)} \to \infty$ as $n\to\infty \Rightarrow A \in \omega(B)$		

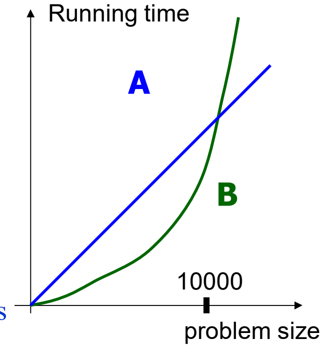
Exercise

Show that

1) $3n^2 - 100n + 6 = O(n^2)$
2) $3n^2 - 100n + 6 = O(n^3)$
3) $3n^2 - 100n + 6 \neq O(n)$
4) $3n^2 - 100n + 6 = \Omega(n^2)$
5) $3n^2 - 100n + 6 \neq \Omega(n^3)$
6) $3n^2 - 100n + 6 = \Omega(n)$
7) $3n^2 - 100n + 6 = \mathbf{\Theta}(n^2)$
8) $3n^2 - 100n + 6 \neq \mathbf{\Theta}(n^3)$
9) $3n^2 - 100n + 6 \neq \mathbf{\Theta}(n)$

Final notes

- Even though in this course we focus on the asymptotic growth using big-Oh notation, practitioners do care about constant factors occasionally
- Suppose we have 2 algorithms
  - Algorithm A has running time $30000n$
  - Algorithm B has running time $3n^2$
- Asymptotically, algorithm A is better than algorithm B
- However, if the problem size you deal with is always less than 10000, then the quadratic one is faster



NGUYỄN KHÁNH PHƯƠNG
SOICT–HUST

Contents

Running time calculation

- Experimental evaluation of running time:
  - Write a program implementing the algorithm
  - Run the program and measure the running time
  - Cons of experimental evaluation:
    - It is necessary to implement the algorithm, which may be difficult
    - Results may not be indicative of the running time on other inputs not included in the experiment
    - In order to compare two algorithms, the same hardware and software environments must be used

  ➔ We need: **Theoretical Analysis of Running Time**
- Theoretical Analysis of Running Time:
  - Uses a pseudo-code description of the algorithm instead of an implementation
  - Characterizes running time as a function of the input size, ***n***
  - Takes into account all possible inputs
  - Allows us to evaluate the speed of an algorithm independent of the hardware/software environment (Changing the hardware/software environment affects the running time by a constant factor, but does not alter the growth rate of the running time)

Primitive Operations

- For theoretical analysis, we will count **primitive** or **basic** operations, which are $\underbrace{\text{simple computations performed by an algorithm}}_{}$

  could be implemented within the running time that is bounded above by a constant independent of the input data size.

- Examples of primitive operations:
  - Evaluating an expression x^2+e^y
  - Assigning a value to a variable cnt ← cnt+1
  - Indexing into an array A[5]
  - Calling a method mySort(A,n)
  - Returning from a method return(cnt)

NGUYỄN KHÁNH PHƯƠNG
SOICT– HUST

Running Time Calculations: General rules

1. **Consecutive Statements**: The sum of running time of each segment.
- *Running time of* "P; Q", where P is implemented first, then Q, is

  $Time(P; Q) = Time(P) + Time(Q)$,

  or if using asymptotic Theta:

  $Time(P; Q) = \Theta(\max(Time(P), Time(Q)).$

2. **FOR loop**: The number of iterations times the time of the inside statements.

```
for i =1 to m do P(i);
```

Assume running time of $P(i)$ is $t(i)$, then the running time of for loop is $\sum_{i=1}^{m} t(i)$

3. **Nested loops**: The product of the number of iterations times the time of the inside statements.

```
for i =1 to n do
  for j =1 to m do P(j);
```

Assume the running time of $P(j)$ is $t(j)$, then the running time of this nested loops is:

Some Examples

Case1:
```
for (i=0; i<n; i++)
    for (j=0; j<n; j++)
        k++;
```

$O(n^2)$

$O(n)$ work followed by $O(n^2)$ work, is also $O(n^2)$

$O(n^2)$

Running Time Calculations: General rules

4. If/Else

```
if (condition )
    S1;
else
    S2;
```

The testing time plus the larger running time of the S1 and S2.

Characteristic statement

- Definition. The characteristic statement is the statement being executed with frequency at least as well as any statement in the algorithm.
- If the execution time of each statement is bounded above by a constant, then the running time of the algorithm will be the same size as the number of times the execution of the characteristic statement

=> To evaluate the running time, one can count the number of times the characteristic statement being executed

Example: Calculating Fibonacci Sequences

```
function Fibrec(n)
    if n <2 then return n;
    else return Fibrec(n-1)+Fibrec(n-2);
```

```
function Fibiter(n)
    i=0;
    j=1;
    for k=1 to n do
        j=i + j;
        i=j – i;
    return j;
```

Characteristic statement

- Fibonacci Sequence:
  - $f_0=0$;
  - $f_1=1$;
  - $f_n=f_{n-1}+f_{n-2}$

• The number of times this characteristic statement being executed is n ➔ The running time of Fibiter is $O(n)$

n	10	20	30	50	100
Fibrec	8ms	1sec	2min	21days	10^9years
Fibiter	0.17ms	0.33ms	0.5ms	0.75ms	1.5ms

Exercise 1: Maximum Subarray Problem

Given an array of integers A_1, A_2, …, A_N, find the maximum value of $\sum_{k=i}^{j} A_k$

For convenience, the maximum subsequence sum is zero if all the integers are negative.

NGUYỄN KHÁNH PHƯƠNG
SOICT– HUST

Algorithm 1. Brute force

```
int maxSum = 0;
for (int i=0; i<n; i++) {
    for (int j=i; j<n; j++) {
        int sum = 0;
        for (int k=i; k<=j; k++)
            sum += a[k];
        if (sum > maxSum)
            maxSum = sum;
    }
}
```

Select the statement **sum+=a[k]** as the characteristic statement
➔ Running time of the algorithm: $O(n^3)$

NGUYỄN KHÁNH PHƯƠNG
SOICT– HUST

Algorithm 2. Brute force with better implement

```
int maxSum = a[0];
for (int i=0; i<n; i++) {
    int sum = 0;
    for (int j=i; j<n; j++) {
        sum += a[j];
        if (sum > maxSum)
            maxSum = sum;
    }
}
```

$O(n^2)$

NGUYỄN KHÁNH PHƯƠNG
SOICT– HUST

Algorithm 3. Dynamic programming

The primary steps of dynamic programming:

1. Divide:

- Define s_i the value of max subarray of the array $a_0, a_1, ..., a_i$, $i = 0, 1, ..., n$-1.
- Clearly, s_{n-1} is the solution.

3. Construct the final solution:

- $s_0 = a_0$
- Assume $i > 0$ and we already know the value of s_k with $k = 0, 1, ..., i$-1. Now we need to calculate the value of s_i which is the value of max subarray of the array:

  $a_0, a_1, ..., a_{i-1}, a_i$.
- We see that: the max subarray of this array $a_0, a_1, ..., a_{i-1}, a_i$ could either include the element a_i or not include the element a_i ➔ therefore, the max subarray of the array $a_0, a_1, ..., a_{i-1}, a_i$ could only be one of these 2 arrays:
  – The max subarray of the array $a_0, a_1, ..., a_{i-1}$
  – The max subarray of the array $a_0, a_1, ..., a_i$ *ending at* a_i.

➔ Thus, we have $s_i = max \{s_{i-1}, e_i\}$, $i = 1,2, ..., n$-1.

where e_i is the value of the max subarray $a_0, a_1, ..., a_i$ ending at a_i.

To calculate e_i, *we could use the recursive relation*:

– $e_0 = a_0$;
– $e_i = \max \{a_i, e_{i-1} + a_i\}$, $i = 1,2, ..., n$-1.

```
MaxSub(a)
{
    smax = a[0];          // smax: the value of max subarray
    ei   = a[0];          //ei   : the value of max subarray ending at a[i]
    for i = 1 to n-1 {
        ei = max{a[i],ei + a[i]};
        smax = max{smax, ei};
    }
}
```

Exercise 2: Selection sort

- Sort a sequence of numbers in ascending order
- Algorithm:
  – Find the smallest and move it to the first place
  – Find the next smallest and move it to the second place
  – Find the next smallest and move it to the 3rd place
  – …

```
void selectionSort(int a[], int n){
  int i, j, index_min;
  for (i = 0; i < n-1; i++) {
    index_min = i;
    //Find the smallest element from a[i+1] till the last element
    for (j = i+1; j < n; j++)
       if (a[j] < a[index_min]) index_min = j;
    //move the element a[index_min] to the ith place:
    swap(a[i], a[index_min]);
  }
}
```

```
void swap(int &a,int &b)
{
   int temp = a;
   a = b;
   b = temp;
}
```

	i=0	1	2	3	4	5	6
42	13	13	13	13	13	13	13
20	20	14	14	14	14	14	14
17	17	17	15	15	15	15	15
13	42	42	42	17	17	17	17
28	28	28	28	28	20	20	20
14	14	20	20	20	28	23	23
23	23	23	23	23	23	28	28
15	15	15	17	42	42	42	42

Exercise 3

- Give asymptotic big-Oh notation for the running time $T(n)$ of the following statement segment:

```
for (int i = 1; i<=n; i++)
  for (int j = 1; j<= i*i*i; j++)
       for (int k = 1; k<=n; k++)
                  x = x + 1;
```

- Ans:

NGUYỄN KHÁNH PHƯƠNG
SOICT– HUST

Exercise 4

- Give asymptotic big-Oh notation for the running time $T(n)$ of the following statement segment:

a)
```
int x = 0;
 for (int i = 1; i <=n; i *= 2)
        x=x+1;
```

- Ans:

```
int x = 0;
     for (int i = n; i > 0; i /= 2)
            x=x+1;
```

- Ans:

NGUYỄN KHÁNH PHƯƠNG
SOICT– HUST

Exercise 5

Give asymptotic big-Oh notation for the running time $T(n)$ of the following statement segment:

```
int n;
if (n<1000)
   for (int i=0; i<n; i++)
      for (int j=0; j<n; j++)
         for (int k=0; k<n; k++)
            cout << "Hello\n";
else
   for (int j=0; j<n; j++)
      for (int k=0; k<n; k++)
         cout << "world!\n";
```

Ans:

- $T(n)$ is the constant when n<1000. $T(n) = O(n^2)$.

NGUYỄN KHÁNH PHƯƠNG
SOICT–HUST