การอบรมคอมพิวเตอร์โอลิมปิก สอวน. ค่ายที่ 2 ปีการศึกษา 2563

ศูนย์สอวน. โรงเรียนสามเสนวิทยาลัย โรงเรียนสตรีศรีสุริโยทัย
และ โรงเรียนสตรีวัดมหาพฤฒาราม
โดย ภาควิชาวิทยาการคอมพิวเตอร์
คณะวิทยาศาสตร์และเทคโนโลยี ม.ธรรมศาสตร์

COMPUTER SCIENCE
THAMMASAT UNIVERSITY

ช่วงที่ 3: อัลกอริทึม (Algorithms)

อ. ดร.วนิดา พฤทธิวิทยา

อ. พิสิษฐ์ มรรคไพสิษฐ์

อ. ดร.ฐาปนา บุญชู

เค้าโครงการอบรม

COMPUTER SCIENCE
THAMMASAT UNIVERSITY

- บทนำ (Introduction)
- การวิเคราะห์ความซับซ้อนของอัลกอริทึม (Algorithmic Complexity)
- กลวิธีทางอัลกอริทึม
  - Brute Force
  - Divide-and-Conquer
  - Decrease-and-Conquer
  - Transform-and-Conquer
  - String Matching Algorithms
  - Backtracking & Branch-and-Bound
  - Greedy Algorithms
  - Graph Algorithms
  - Dynamic Programming

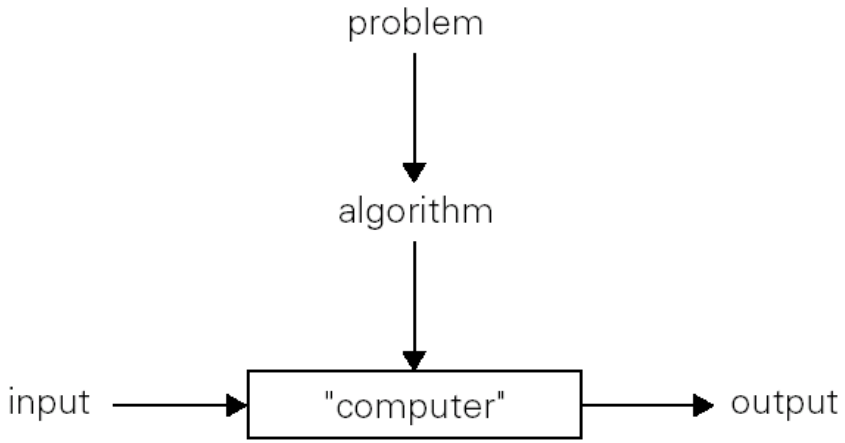
บทนำ (Introduction)

COMPUTER SCIENCE
THAMMASAT UNIVERSITY

Introduction: What Is an Algorithm?

COMPUTER SCIENCE
THAMMASAT UNIVERSITY

- an **algorithm** is a sequence of unambiguous instructions for solving a *well-specified computational problem*
- an **algorithm** is a sequence of computational steps that transforms the *legitimate input* into the *desired output* in a *finite amount of time*



Introduction: ตัวอย่าง Computational Problems: The Sorting Problem

COMPUTER SCIENCE
THAMMASAT UNIVERSITY

Sorting (การเรียงลำดับข้อมูล)

- **Problem Statement:**
  - *Input:* a sequence of n numbers $\langle a_1, a_2, \ldots, a_n \rangle$
  - *Output:* a reordering of the input sequence $\langle a_1, a_2, \ldots, a_n \rangle$ such that $a_i \leq a_j$ whenever $i < j$
- a problem Instance: the sequence <31, 41, 59, 26, 41, 58>
- *Algorithms:* Insertion Sort, Selection Sort, Bubble Sort, Merge Sort, Quick Sort

Introduction: ตัวอย่าง Computational Problems: The Sorting Problem (2)

COMPUTER SCIENCE
THAMMASAT UNIVERSITY

- given a problem instance: the *input* sequence <31, 41, 59, 26, 41, 58>
- any sorting algorithm that works correctly would yield the desired *output*, that is, the sequence <26, 31, 41, 41, 58, 59>

Introduction: ตัวอย่าง Computational Problems:

The Sorting Problem: -- in action

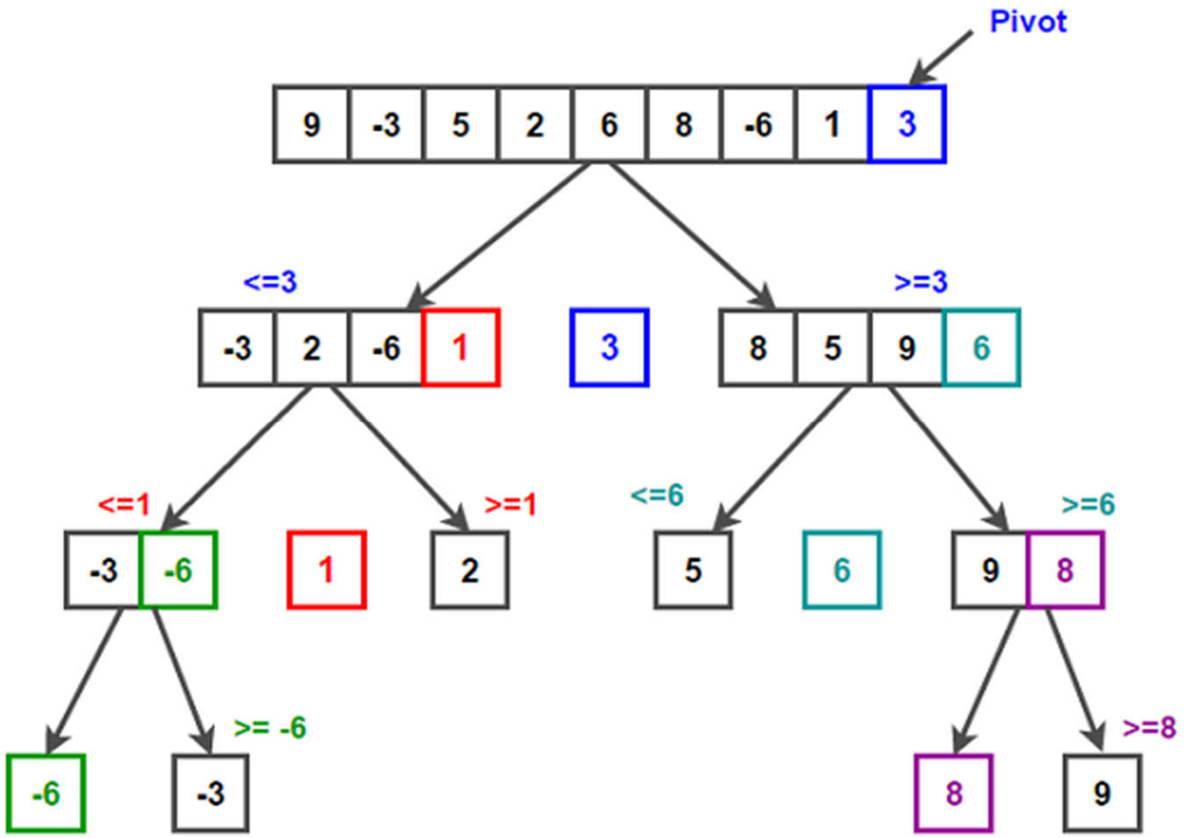
COMPUTER SCIENCE
THAMMASAT UNIVERSITY

- Bubble Sort
- Insertion Sort
- Selection Sort

See https://visualgo.net/en/sorting

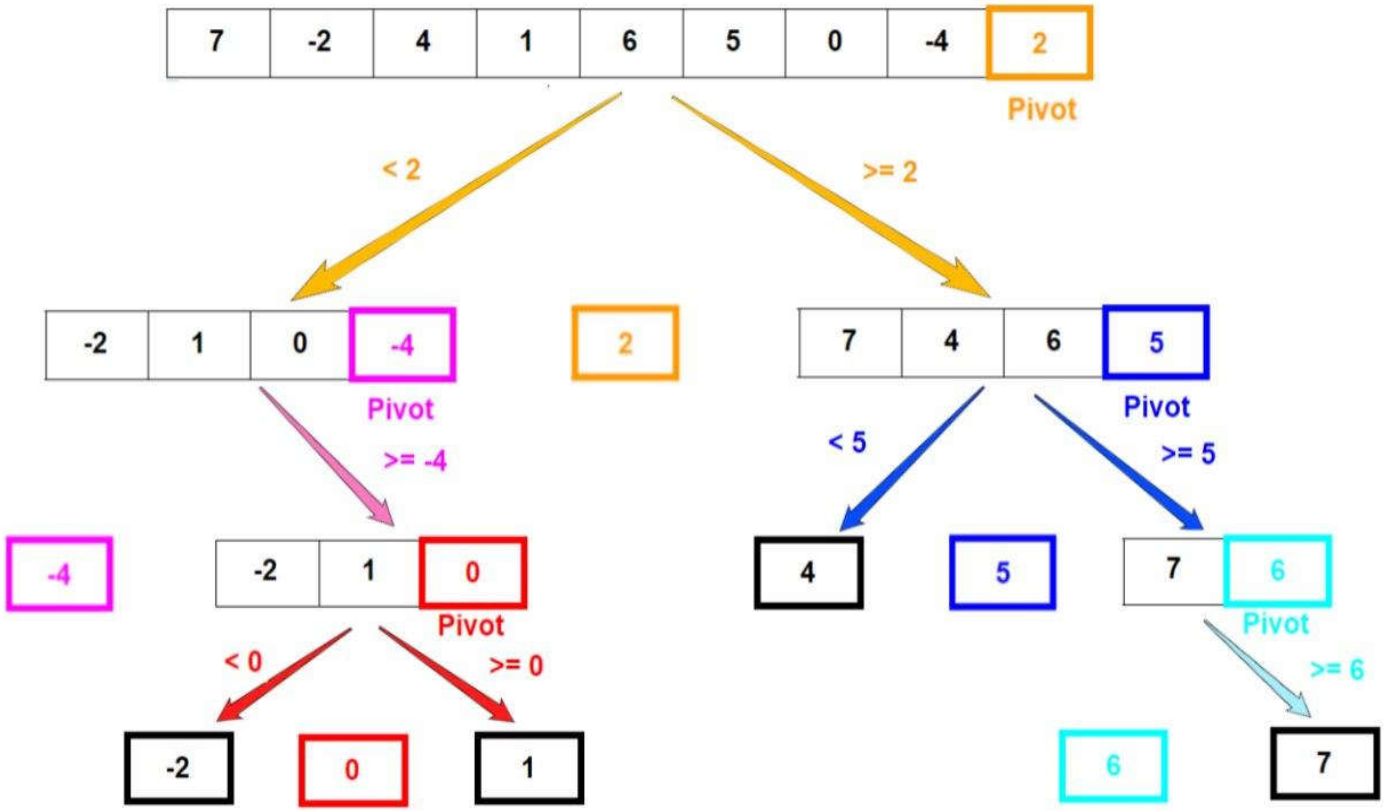
Introduction: ตัวอย่าง Computational Problems:

The Sorting Problem: Quick Sort -- in action



https://www.techiedelight.com/quicksort/

Introduction: ตัวอย่าง Computational Problems:
The Sorting Problem: Quick Sort -- in action (2)



https://morioh.com/p/b0deaa623ac4

Introduction: ตัวอย่าง Computational Problems:

The Sorting Problem: Quick Sort -- in action (3)

- **Worst-Case Scenario:** จากตัวอย่าง ลิสต์มีข้อมูลจำนวน 7 ตัว จะพบว่า

  รอบที่ 1: ทำการเปรียบเทียบ 6 ครั้ง

  รอบที่ 2: ทำการเปรียบเทียบ 5 ครั้ง

  รอบที่ 3: ทำการเปรียบเทียบ 4 ครั้ง

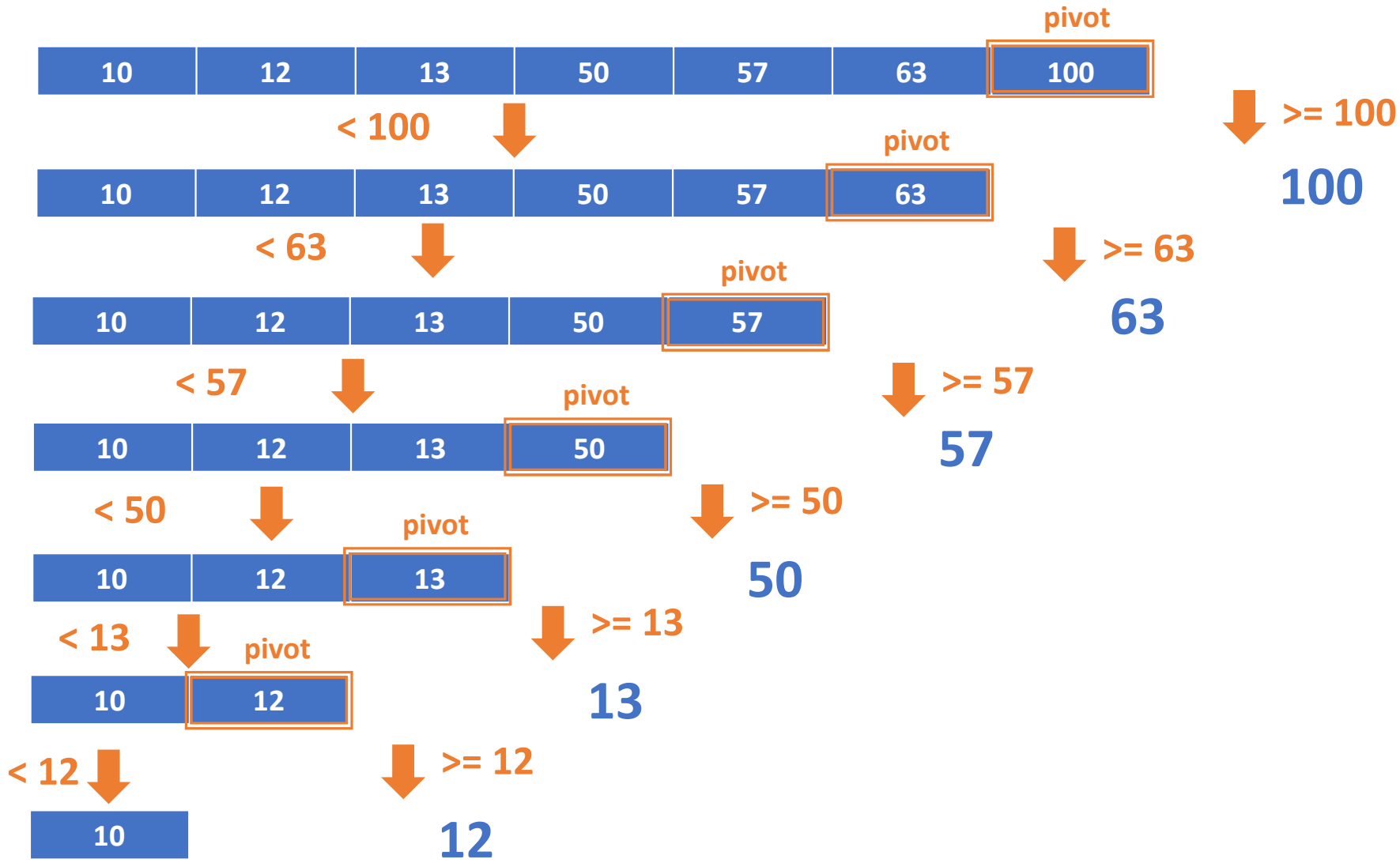
  . . .

  รอบที่ 7: ทำการเปรียบเทียบ 1 ครั้ง

  เมื่อ n = 7, $\sum_{i=1}^{7}(7-i) = \sum_{i=1}^{n}(n-i) = \sum_{i=1}^{n} n - \sum_{i=1}^{n} i$

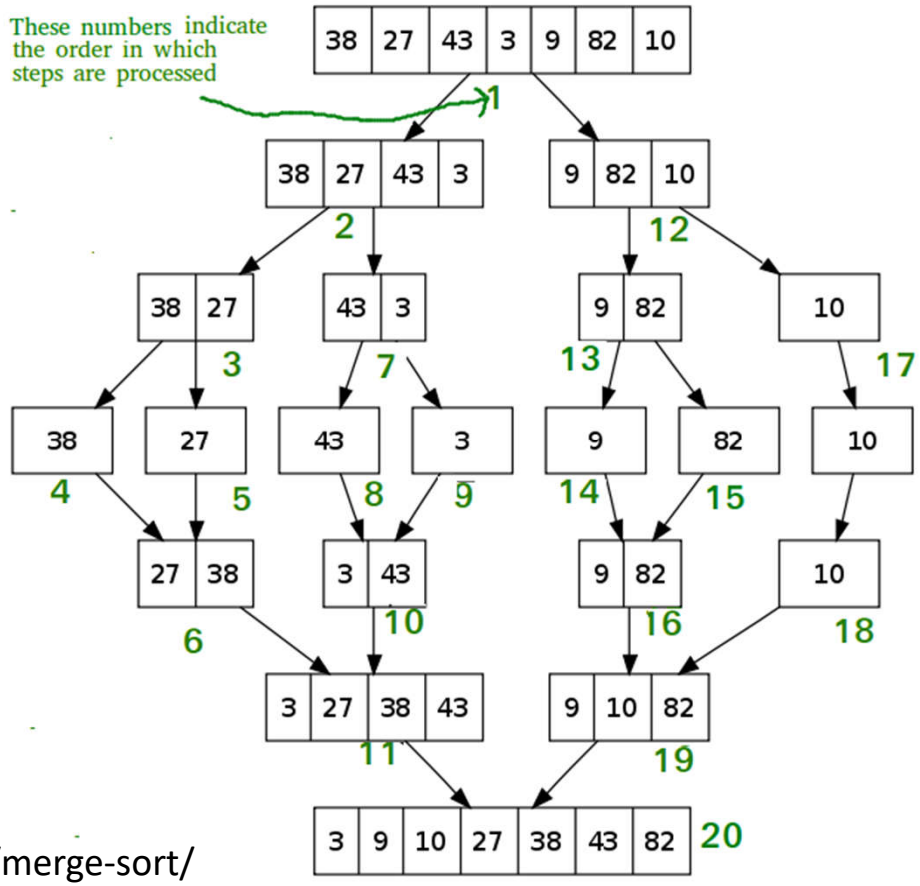
$$= \underbrace{(n+n+\cdots+n)}_{\text{n ครั้ง}} - \frac{n\times(n+1)}{2}$$

$$= (n \times n) - \frac{n^2+n}{2} = \frac{n^2-n}{2}$$


pivot
10
12
13
50
57
63
100
< 100
>= 100
100
pivot
10
12
13
50
57
63
< 63
>= 63
63
pivot
10
12
13
50
57
< 57
>= 57
57
pivot
10
12
13
50
< 50
>= 50
50
pivot
10
12
13
< 13
>= 13
13
pivot
10
12
< 12
>= 12
12
10

Introduction: ตัวอย่าง Computational Problems:

The Sorting Problem: Merge Sort -- in action



https://www.geeksforgeeks.org/merge-sort/

Introduction: ตัวอย่าง Computational Problems : gcd(m, n)

COMPUTER SCIENCE
THAMMASAT UNIVERSITY

Finding the greatest common divisor (gcd) of two non-negative, not both zero integers

- **Problem Statement:**
  - *Input:* two non-negative, not both zero integers called m and n , $m \geq n$
  - *Output:* the greatest common divisor of m and n
- a problem Instance: (18, 12), (60, 24), (12, 0)
- *Algorithms:* the intuitive algorithm, Euclid's Algorithm

Introduction: ตัวอย่าง Computational Problems: gcd(m, n) : the Consecutive Integers Checking algorithm

COMPUTER SCIENCE
THAMMASAT UNIVERSITY

- Intuitively based on the definition of gcd

Algorithm gcd(***m***, ***n***)

If ***n*** = 0 Return ***m*** and stop.

While ***n*** > 1 do

Begin

    If ***m*** is divisible by ***n*** Return ***n*** and stop.

    Otherwise, decrease ***n*** by one.

End

Return ***n***.

Introduction: ตัวอย่าง Computational Problems: gcd(m, n): Euclid's algorithm

- gcd (***m***, ***n***) = gcd (***n***, ***m*** mod ***n***)

  gcd (***m***, ***0***) = ***m***

- Ex. gcd (***18***, ***12***)

  = gcd (***12***, ***6***)

  = gcd (***6***, ***0***)

  = ***6***

Algorithm gcd(***m***, ***n***)

While ***n*** ≠ 0 do

Begin

r = ***m*** mod ***n***

m = ***n***

n = ***r***

End.

Return ***m***.

Introduction (13)

COMPUTER SCIENCE
THAMMASAT UNIVERSITY

- several algorithms for solving the same problem may exist
  - different people have different ideas when solving a problem
  - different ***algorithm design techniques***
- one algorithm can be implemented in several different ways using different ***data structures***
- as a result, *different algorithms* that are based on *different ideas* and employed different data structures could produce the desired result with largely ***different speeds*** (different ***execution time*** or ***run time***)

Introduction (14)

COMPUTER SCIENCE
THAMMASAT UNIVERSITY

- given an algorithm implemented with one algorithm design technique using one type of data structure
- there are many factors that still play a role in the *running time* of the algorithm
- Let's look at the Sorting problem as our example
  - **the number of items to be sorted (input size)**
  - **the extent to which the items are already somewhat sorted**
  - the architecture of the computer
  - the kind of storage devices to be used: M/M, disks, or tapes

Introduction:
Fundamentals of Algorithmic Problem Solving

COMPUTER SCIENCE
THAMMASAT UNIVERSITY

1. Understand the problem
2. Choose between exact and approximate problem solving
3. Decide on algorithm design techniques and appropriate data structures
4. Specify the algorithm: natural language, pseudocode, flowcharts
5. Prove the algorithm's correctness
    1. Exact algorithms: the algorithm yields a required result for every legitimate input
    2. Approximate algorithms: the error produced by the algorithm does not exceed a pre-defined limit
6. Analyze the algorithm
7. Coding the algorithm

Introduction: วัตถุประสงค์การเรียนรู้ (Our Goals)

COMPUTER SCIENCE
THAMMASAT UNIVERSITY

- learn several **algorithm design techniques** by studying a standard set of important algorithms from different areas of computing (different problem types)
- discuss the general framework for efficiency analysis
- so that, students
  - can develop algorithms on their own when given a computational problem
  - can formally argue the correctness of their algorithms
  - can analyze the efficiency of their algorithms

Introduction: Important Problem Types

COMPUTER SCIENCE
THAMMASAT UNIVERSITY

- Sorting
- Searching
- String Processing e.g. string matching - searching for a given word in a text
- Graph problems
- Combinatorial problems
- Geometric problems
- Numerical problems

พื้นฐานการวิเคราะห์ความซับซ้อนของอัลกอริทึม

COMPUTER SCIENCE
THAMMASAT UNIVERSITY

What is an Analysis of Algorithms?

COMPUTER SCIENCE
THAMMASAT UNIVERSITY

- "Analysis of algorithms" means an investigation of an algorithm's efficiency with respect to two resources: running time and memory space
  - การประเมินประสิทธิภาพของอัลกอริทึม
- time efficiency ประสิทธิภาพเชิงเวลา (time complexity) indicates how fast an algorithm in question runs
- space efficiency ประสิทธิภาพเชิงพื้นที่ (space complexity) refers to the amount of memory units the algorithm requires

Analysis of Algorithms: the Analysis Framework

COMPUTER SCIENCE
THAMMASAT UNIVERSITY

- it is logical to investigate an algorithm's efficiency as "a function of some parameter *n* indicating **the algorithm's input size**"

Analysis of Algorithms: the Analysis Framework (2)

COMPUTER SCIENCE
THAMMASAT UNIVERSITY

- Example: the problem of sorting elements in a list
  - the **algorithm's input size** is indicated by **the list's size**
- Example: the problem of searching for the maximum element in a list
  - the **algorithm's input size** is indicated by **the list's size**
- Example: problems of matrix addition, matrix subtraction
  - the **algorithm's input size** is indicated by **the matrix's size**
- Example: problem of evaluating a polynomial degree k

$$p(x) = a_k x^k + a_{k-1} x^{k-1} + \ldots + a_1 x + a_0$$

  - the **algorithm's input size** is indicated by **the degree of polynomial**

Analysis of Algorithms: the Analysis Framework: Time Efficiency

COMPUTER SCIENCE
THAMMASAT UNIVERSITY

- **Time efficiency** is analyzed by counting the number of times an algorithm's basic operation is executed on input of size *n*
  - basic operation: the operation contributing the most to the total running time

Rule for finding algorithm's basic operation: it is the most time-consuming operation in the algorithm's innermost loop

Analysis of Algorithms: the Analysis Framework

Time Efficiency (2)

COMPUTER SCIENCE
THAMMASAT UNIVERSITY

Example: the problem of searching for the maximum element in a list

```
Algorithm FindMax(A[0..n-1])
max = A[0];
For (i = 1; i < n; i++) do
        if (max < A[i]) max = A[i];
```

- basic operation: key comparison inside loop for

Analysis of Algorithms: the Analysis Framework

Time Efficiency (3)

COMPUTER SCIENCE
THAMMASAT UNIVERSITY

Example: the problem of sorting all elements in a list

```
Algorithm BubbleSort(A[0..n-1])
For (i = 0; i < n-1; i++)
        For (j = 0; j < n – i – 1; j++)
                if (A[j] > A[j+1])
                        Swap(A[j], A[j+1]);
```

- basic operations: key comparison inside the inner loop for
  swap operation inside the inner loop for

Analysis of Algorithms: the Analysis Framework: Time Efficiency (4)

COMPUTER SCIENCE
THAMMASAT UNIVERSITY

- Let **n** be the algorithm's input size
- let c_{op} be the execution time of an algorithm's basic operation on a particular computer
- let **C(n)** be the number of times this basic operation needs to be executed for this algorithm Ex. **C(n) = 0.5n(n-1)**
- then, the **running time T(n)** of a program implementing this algorithm on this particular computer can be estimated by the formula

$$T(n) \approx c_{op} \times C(n)$$

Analysis of Algorithms: the Analysis Framework: Time Efficiency (5)

COMPUTER SCIENCE
THAMMASAT UNIVERSITY

- We usually focus on how much the running time changes as the input size increases, not the running time for a particular input size

How much longer will the algorithm run if we double its input size?

$$\frac{T(2n)}{T(n)} = \frac{c_{op} \times C(2n)}{c_{op} \times C(n)}$$

note that we were able to answer the question w/o actually knowing the value of c_{op}: the value was neatly **cancelled out** in the ratio

Analysis of Algorithms: the Analysis Framework: Time Efficiency (6)

COMPUTER SCIENCE
THAMMASAT UNIVERSITY

- it is for these reasons that **the time efficiency analysis framework ignores multiplicative constants** and **concentrates on the count**'s **order of growth** to within a constant multiple for **large-size inputs**

$$T(n) \approx C(n)$$

Analysis of Algorithms: the Analysis Framework: Time Efficiency (7)

COMPUTER SCIENCE
THAMMASAT UNIVERSITY

- When discussing **the count**'s **order of growth** to within a constant multiple for **large-size inputs**, we do not need to be exact
- We usually use **asymptotic notations** to express the time efficiency of an algorithm (**the count**'s **order of growth**)

Analysis of Algorithms: the Analysis Framework:
Time Efficiency: Asymptotic Notations

COMPUTER SCIENCE
THAMMASAT UNIVERSITY

- to **compare** and **rank** the **orders of growth** of an algorithm's **basic operation counts**, computer scientists use three notations:

1. ***O*** (big Oh)
2. **Ω** (big Omega)
3. **Θ** (big Theta)

Analysis of Algorithms: the Analysis Framework: Time Efficiency: Asymptotic Notations: big-Oh

COMPUTER SCIENCE
THAMMASAT UNIVERSITY

- **O(g(n))** is the set of all functions with a **smaller or same order of growth** as **g(n)** (to within a constant multiple, as **n** goes to infinity).
- **g(n)** is an "upper-bound".
- for example,

$$n \in O(n^2), \qquad 100n + 5 \in O(n^2), \qquad \frac{1}{2}n(n-1) \in O(n^2)$$

$$n^3 \notin O(n^2), \qquad 0.00001n^3 \notin O(n^2), \qquad n^4 + n + 1 \notin O(n^2)$$

Analysis of Algorithms: the Analysis Framework: Time Efficiency: Asymptotic Notations: big-Omega

COMPUTER SCIENCE THAMMASAT UNIVERSITY

- **Ω(g(n))**, stands for the set of all functions with a **larger or same order of growth** as **g(n)** (to within a constant multiple, as **n** goes to infinity).
- **g(n)** is a "lower-bound".
- for example,

$$n^3 \in \Omega(n^2), \qquad \frac{1}{2}n(n-1) \in \Omega(n^2), \qquad \text{but } 100n + 5 \notin \Omega(n^2).$$

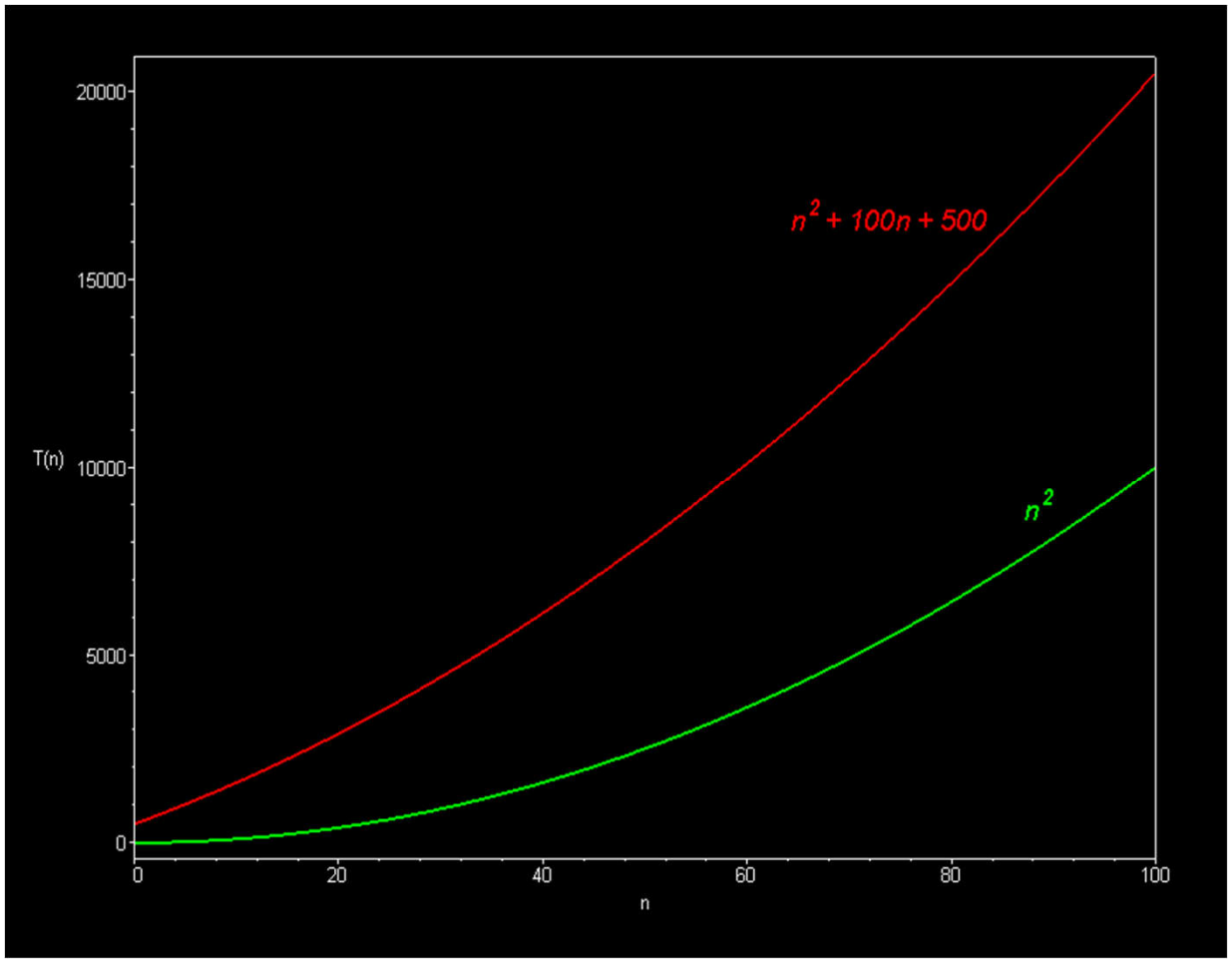
Analysis of Algorithms: the Analysis Framework: Time Efficiency: Asymptotic Notations: big-Theta

COMPUTER SCIENCE
THAMMASAT UNIVERSITY

- $\Theta(g(n))$ is the set of all functions that have the **same order of growth** as $g(n)$ (to within a constant multiple, as n goes to infinity).
- thus, every quadratic function $an^2 + bn + c$ with $a > 0$ is in $\Theta(n^2)$

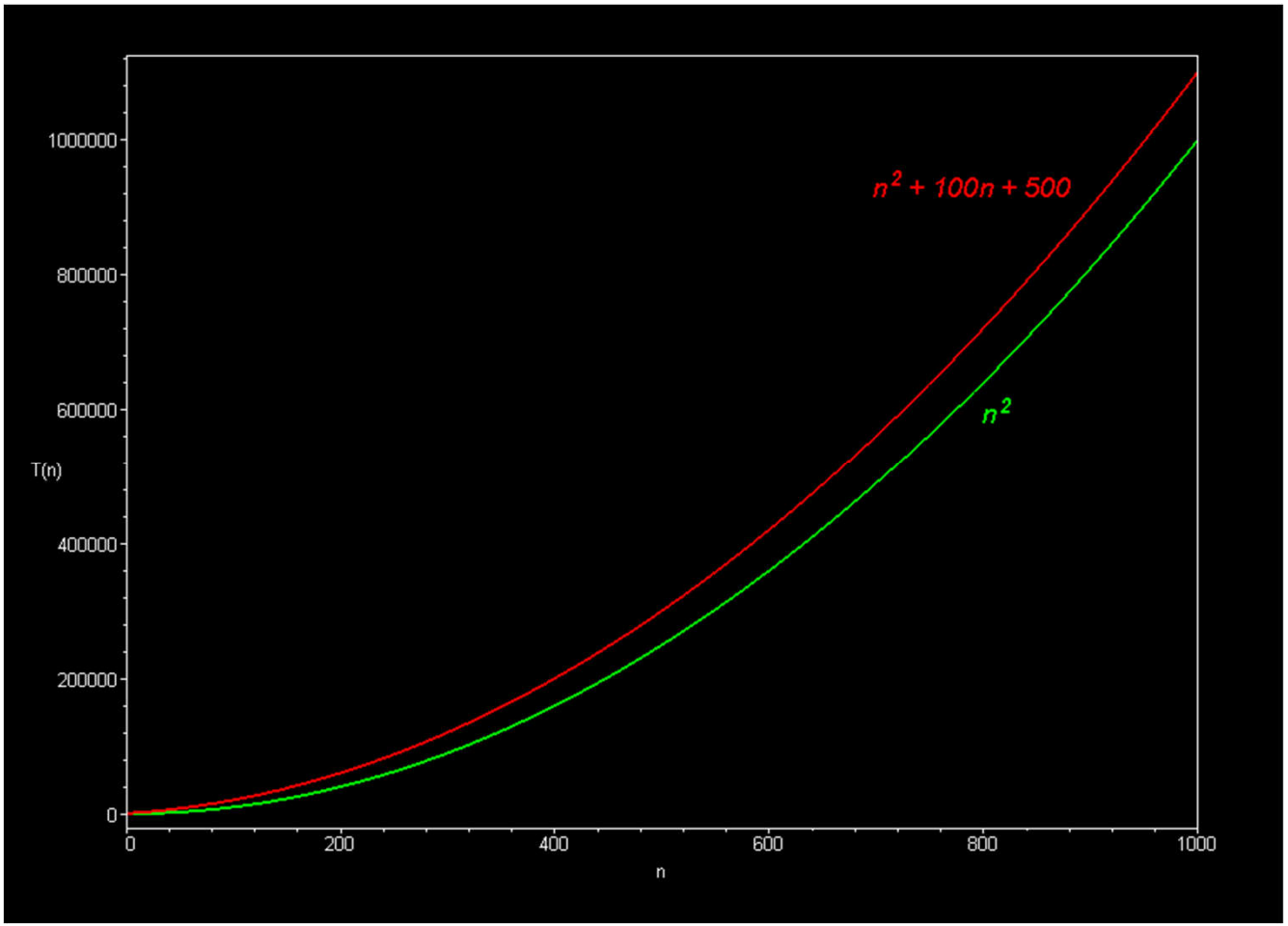
source: http://science.slc.edu/~jmarshall/courses/2002/spring/cs50/BigO/

Asymptotic Notations: Informal Introduction: big-Theta (2)



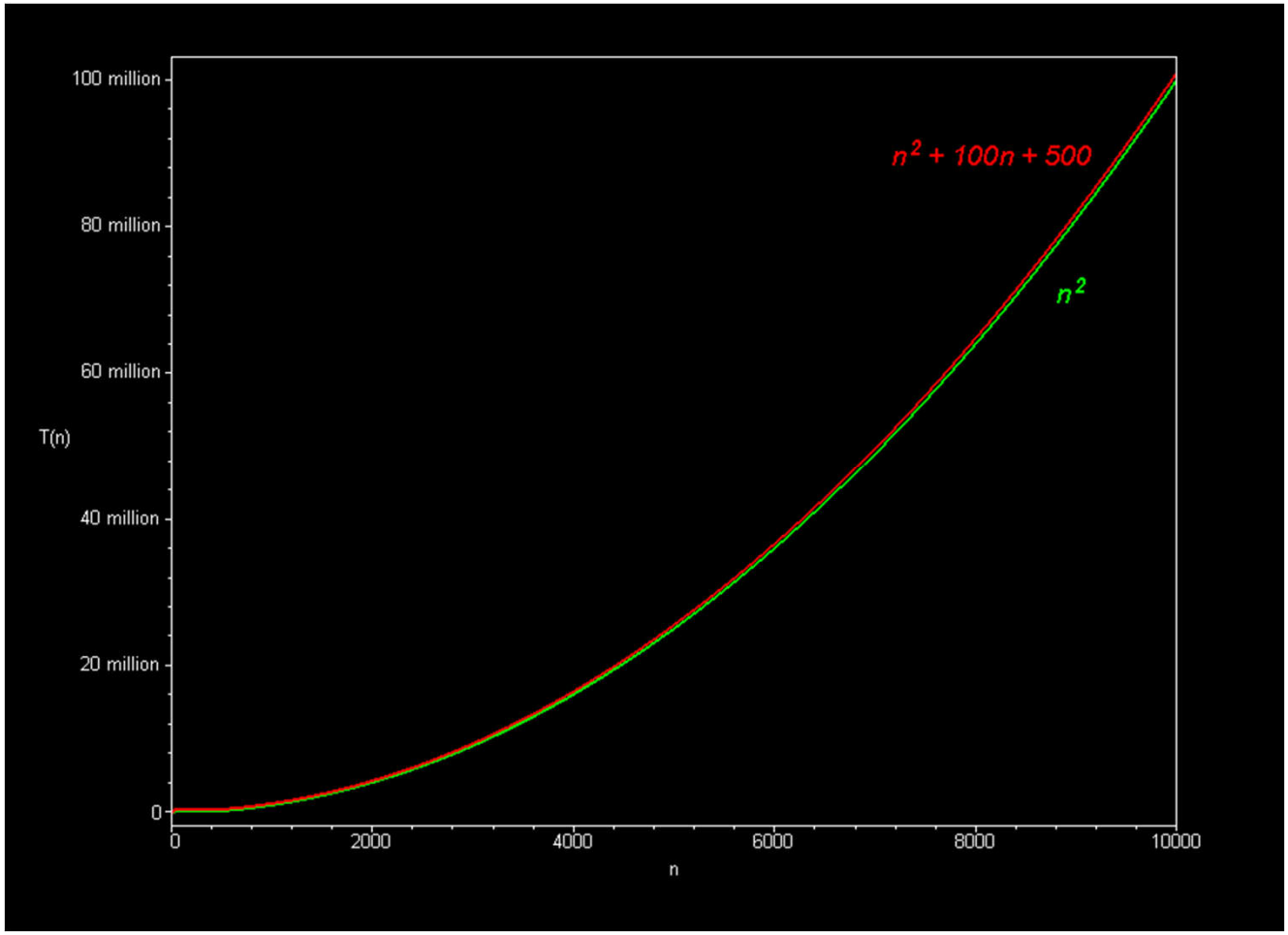
source: http://science.slc.edu/~jmarshall/courses/2002/spring/cs50/BigO/

Asymptotic Notations: Informal Introduction: big-Theta (3)



source: http://science.slc.edu/~jmarshall/courses/2002/spring/cs50/BigO/

Asymptotic Notations: Informal Introduction: big-Theta (4)



Analysis of Algorithms: the Analysis Framework:

Time Efficiency: Basic Asymptotic Efficiency Classes

COMPUTER SCIENCE
THAMMASAT UNIVERSITY

Class	Name
1	Constant
log n	Logarithmic
n	linear
n log n	“n-log-n”
n^2	Quadratic
n^3	Cubic
2^n	exponential
n!	factorial

Analysis of Algorithms: the Analysis Framework: Time Efficiency: Order of Growth

COMPUTER SCIENCE
THAMMASAT UNIVERSITY

TABLE 2.1 Values (some approximate) of several functions important for analysis of algorithms

n	$\log_2 n$	n	$n \log_2 n$	n^2	n^3	2^n	$n!$
10	3.3	10^1	$3.3 \cdot 10^1$	10^2	10^3	10^3	$3.6 \cdot 10^6$
10^2	6.6	10^2	$6.6 \cdot 10^2$	10^4	10^6	$1.3 \cdot 10^{30}$	$9.3 \cdot 10^{157}$
10^3	10	10^3	$1.0 \cdot 10^4$	10^6	10^9		
10^4	13	10^4	$1.3 \cdot 10^5$	10^8	10^{12}		
10^5	17	10^5	$1.7 \cdot 10^6$	10^{10}	10^{15}		
10^6	20	10^6	$2.0 \cdot 10^7$	10^{12}	10^{18}		

Logarithmic
Linear
Linearithmetic
Quadratic
Cubic
Exponential
Factorial

Analysis of Algorithms: the Analysis Framework: Time Efficiency: Order of Growth (2)

COMPUTER SCIENCE
THAMMASAT UNIVERSITY

- the function ***growing the slowest*** among these is the **logarithmic function**
- the function ***growing the fastest*** among these is the **factorial function**
- a program implementing an algorithm with a **logarithmic basic-operation count** runs practically on inputs of all realistic sizes
- a program implementing an algorithm that requires an **exponential number of basic-operations** are practical for only problems of very small sizes

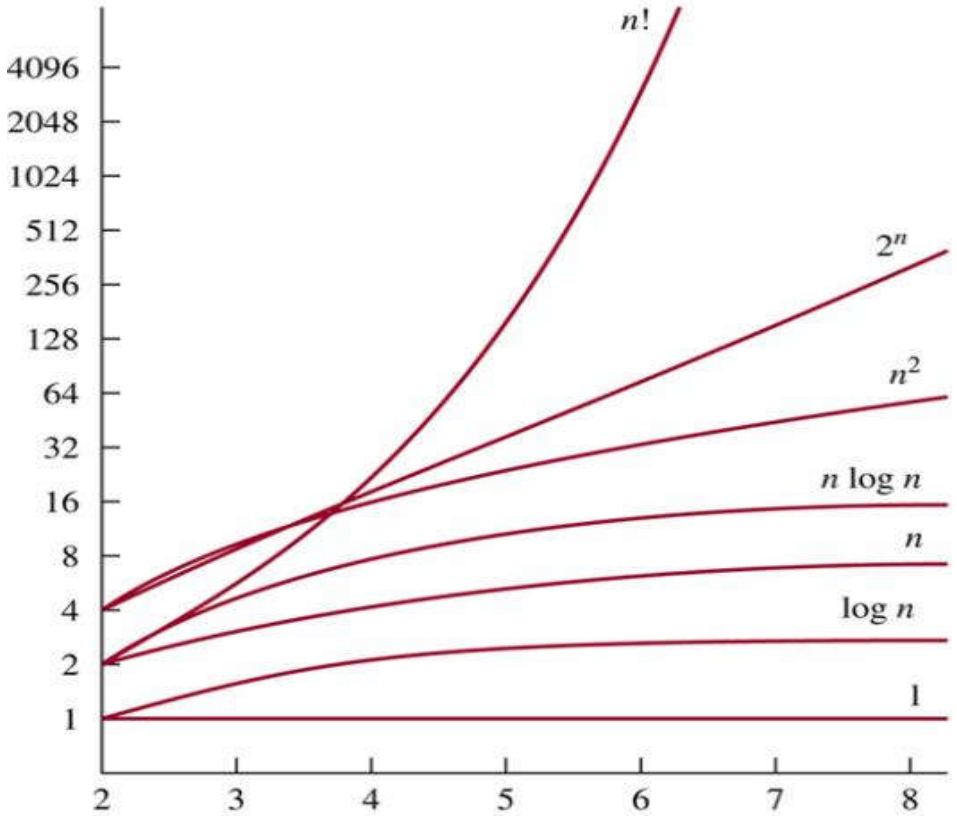
Analysis of Algorithms: the Analysis Framework: Time Efficiency: Order of Growth (3)

COMPUTER SCIENCE THAMMASAT UNIVERSITY

- Example. See how functions **C(n)** with different growths react to a twofold increase in the value of their arguments (**n**)

	C(n)	C(2n)
1. Logarithmic	$\log_2 n$	$\log_2 2n = \log_2 2 + \log_2 n = 1 + \log_2 n$ [i.e. **increase by just 1**]
2. Linear	n	$2n$ [i.e. **twofold**]
3. Linearithmic	$n \log_2 n$	$2n(1 + \log_2 n) = 2n + 2n \log_2 n$ [i.e. **slightly more than twofold**]
4. Quadratic	n^2	$(2n)^2 = 4n^2$ [i.e. **fourfold**]
5. Cubic	n^3	$(2n)^3 = 8n^3$ [i.e. **eightfold**]
6. Exponential	2^n	$2^{2n} = (2^n)^2$ [i.e. **2^n-fold**]
7. Factorial	$n!$	$(2n)!$

Analysis of Algorithms: the Analysis Framework: Time Efficiency: Order of Growth (4)



Analysis of Algorithms: the Analysis Framework

Time Efficiency: Example (1)

COMPUTER SCIENCE
THAMMASAT UNIVERSITY

For an algorithm with **$C(n) = 0.5n(n-1)$**

- $C(n) = 0.5n^2 - 0.5n$
- big-Oh for the time efficiency of the algorithm is **$O(n^2)$**

Analysis of Algorithms: the Analysis Framework

Time Efficiency: Example (2)

COMPUTER SCIENCE
THAMMASAT UNIVERSITY

Example: the problem of searching for the maximum element in a list

```
Algorithm FindMax(A[0..n-1])
max = A[0];
For (i = 1; i < n; i++) do
        if (max < A[i]) max = A[i];
```

- basic operation: key comparison inside loop for
- $C(n) = n$
- big-Oh for the time efficiency of the algorithm is $O(n)$

Analysis of Algorithms: the Analysis Framework

Time Efficiency: Example (3)

COMPUTER SCIENCE
THAMMASAT UNIVERSITY

Example: the problem of sorting all elements in a list

```
Algorithm BubbleSort(A[0..n-1])
For (i = 0; i < n-1; i++)
    For (j=0; j < n – i -1; j++)
        if (A[j] > A[j+1])
            Swap(A[j], A[j+1]);
```

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

$$= 2\left[\sum_{i=0}^{n-2} n - \sum_{i=0}^{n-2} i - \sum_{i=0}^{n-2} 1\right]$$

$$= n^2 - n$$

big-Oh for the time efficiency of the algorithm = $O(n^2)$

- basic operations: key comparison inside the inner loop for

  swap operation inside the inner loop for

Analysis of Algorithms: the Analysis Framework

Time Efficiency: Worst-Case, Best-Case, Average-Case

COMPUTER SCIENCE
THAMMASAT UNIVERSITY

- There are many algorithms for which running time depends *not only* on an ***input size*** *but also* on ***the specifics of a particular input***:
  - worst-case, best-case, and average-case running time

Analysis of Algorithms: the Analysis Framework

Time Efficiency: Worst-Case

COMPUTER SCIENCE
THAMMASAT UNIVERSITY

- the **worst-case efficiency** of an algorithm is its efficiency for **the worst-case input** of size **n**, which is an input of size **n** for which the algorithm **runs the longest** among all possible inputs of that size

Analysis of Algorithms: the Analysis Framework

Time Efficiency: Best-Case

COMPUTER SCIENCE
THAMMASAT UNIVERSITY

- the **best-case efficiency** of an algorithm is its efficiency for **the best-case input** of size **n**, which is an input of size **n** for which the algorithm runs **the fastest** among all possible inputs of that size.
- **best case does not mean the smallest input**

Analysis of Algorithms: the Analysis Framework

Time Efficiency: Average-Case

COMPUTER SCIENCE
THAMMASAT UNIVERSITY

- **the average-case efficiency seeks to provide the algorithm's behavior on a "typical" or "random" input.**
- to analyze the algorithm's **average-case efficiency**, we must make some **assumptions** about possible inputs **of size** `n`.

Analysis of Algorithms: the Analysis Framework

Time Efficiency: Worst-Case, Best-Case, Average-Case: Example

COMPUTER SCIENCE

- Example: Sequential Search Algorithm

ALGORITHM SequentialSearch (A[0..(n-1)], K)

// Search for a given value in a given array by sequential search

// **Input:** an array `A[0..n-1]` and a search key `K`

// **Output:** return the index of the first element of `A` that matches `K`

// or -1 if there are no matching elements

```
i ← 0
while i < n and A[i] ≠ K do
    i ← i + 1
if i < n return i
else return -1
```

Analysis of Algorithms: the Analysis Framework

Time Efficiency: Worst-Case, Best-Case, Average-Case:

Example (2)

COMPUTER SCIENCE

- Sequential Search Algorithm: Worst Case:
  - when there are no matching elements or
  - the first matching element happens to be the last one on the list

$$C_{worst}(n) = n$$

- the **worst-case analysis** works as an **upper-bound for the running time of an algorithm**: it guarantees that for any instance of size n, the running time **will not exceed** $C_{worst}(n)$

Analysis of Algorithms: the Analysis Framework

Time Efficiency: Worst-Case, Best-Case, Average-Case:

Example (2)

COMPUTER SCIENCE

- Sequential Search Algorithm: Best Case:
  - when the first element in the list equals to the search key

$$C_{best}(n) = 1$$

Analysis of Algorithms: the Analysis Framework

Time Efficiency: Worst-Case, Best-Case, Average-Case:

Example (3)

COMPUTER SCIENCE

- Sequential Search Algorithm: Average Case:
- **Standard assumptions**

(1) the probability of a successful search is equal to `p`

(0 <= `p` <= 1)

and

(2) the probability of the first match occurring

in the i^{th} position of the list is the same for every `i`

Analysis of Algorithms: the Analysis Framework

Time Efficiency: Worst-Case, Best-Case, Average-Case:

Example (4)

COMPUTER SCIENCE

- in the case of a **successful search**, **the probability of the first match occurring in the *i*th position** of the list is **`p/n` for every `i`**, and **the number of comparisons** made by the algorithm in such a situation is obviously **`i`**
- in the case of an **unsuccessful search**, **the number of comparisons** is **`n`** with the **probability of such a search being** `(1-p)`.
- Therefore,

$$C_{avg}(n) = [1 \cdot \frac{p}{n} + 2 \cdot \frac{p}{n} + \cdots + i \cdot \frac{p}{n} + \cdots + n \cdot \frac{p}{n}] + n \cdot (1 - p)$$

$$= \frac{p}{n}[1 + 2 + \cdots + i + \cdots + n] + n(1 - p)$$

$$= \frac{p}{n}\frac{n(n+1)}{2} + n(1 - p) = \frac{p(n+1)}{2} + n(1 - p).$$

Brute Force

COMPUTER SCIENCE
THAMMASAT UNIVERSITY

Introduction

COMPUTER SCIENCE
THAMMASAT UNIVERSITY

- ***Brute force*** is a straightforward approach to solving a problem, usually directly based on the problem statement and definitions of the concepts involved.
- ***Brute force*** = "just do it"
- Often, the brute-force strategy is indeed the one that is easiest to apply.

Brute-Force Algorithm: Exponentiation

ALGORITHM Exponentiation (b, n)

// Compute b^n = b x b x .... x b (total n times)

// Input: a base b which is an integer and an exponent n which is a positive integer

// Output: return the result of b to the power n, b^n

```
result ← 1
for (i = 1; i <= n; i++) do
    result ← result * b
return result
```

consider the multiplication to be the algorithm's basic operation

ลูปนี้ทำงาน n รอบ

แต่ละรอบทำงาน 'คูณ' 1 ครั้ง

$C(n) = \sum_{i=1}^{n} 1 = n$

$\therefore$ big-Oh = O(n)

Brute-Force Algorithm: gcd(m, n) : the Consecutive Integers Checking algorithm

COMPUTER SCIENCE
THAMMASAT UNIVERSITY

ALGORITHM gcd (m, n)
// Compute the Greatest Common Divisor (GCD) for two non-negative, not both zero
// integers ***m*** and ***n***
// Input: two non-negative, not both zero integers called m and n AND $m \geq n$
// Output: return gcd(m, n)
If ***n*** = 0 Return ***m*** and stop.
While ***n*** > 1 do
Begin
    If ***m*** is divisible by ***n*** Return ***n*** and stop.
    Otherwise, decrease ***n*** by one.
End
Return ***n***.

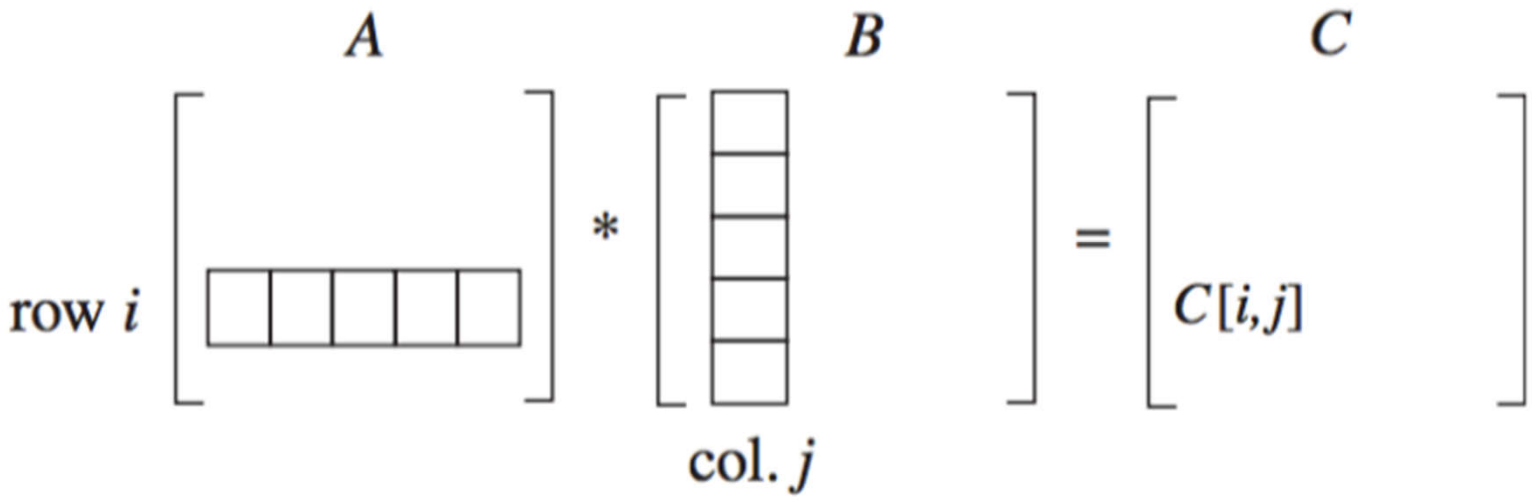
consider the division to be the algorithm's basic operation

ลูปนี้ทำงาน n รอบ (worst-case)
แต่ละรอบทำงาน 'หาร' 1 ครั้ง

$C(n) = \sum_{i=1}^{n} 1 = n$

$\therefore$ big-Oh = O(n)

Brute-Force Algorithm: Matrix Multiplication



$$C[i,j] = A[i,0]B[0,j] + \ldots + A[i,k]B[k,j] + \ldots + A[i,n-1]B[n-1,j]$$

where $0 \leq i, j \leq n-1$.

Brute-Force Algorithm: Matrix Multiplication (2)

COMPUTER SCIENCE
THAMMASAT UNIVERSITY

ALGORITHM *MatrixMultiplication*$(A[0..n-1, 0..n-1], B[0..n-1, 0..n-1])$

```
//Multiplies two square matrices of order n by the definition-based algorithm
//Input: Two n × n matrices A and B
//Output: Matrix C = AB
for i ← 0 to n − 1 do
    for j ← 0 to n − 1 do
        C[i, j] ← 0.0
        for k ← 0 to n − 1 do
            C[i, j] ← C[i, j] + A[i, k] * B[k, j]
return C
```

consider the multiplication to be the algorithm's basic operation

The measurement of the input size is 'n' (the matrix order)

Brute-Force Algorithm: Matrix Multiplication (3)

- just **one multiplication** is made **for each repetition of the innermost loop**, i.e., for each value of the loop's variable **`k`** between its limits **`0`** and **`n-1`**
- that is, the number of **multiplications** made for every specific pair of variables **`i`** and **`j`** is

$$\sum_{k=0}^{n-1} 1$$

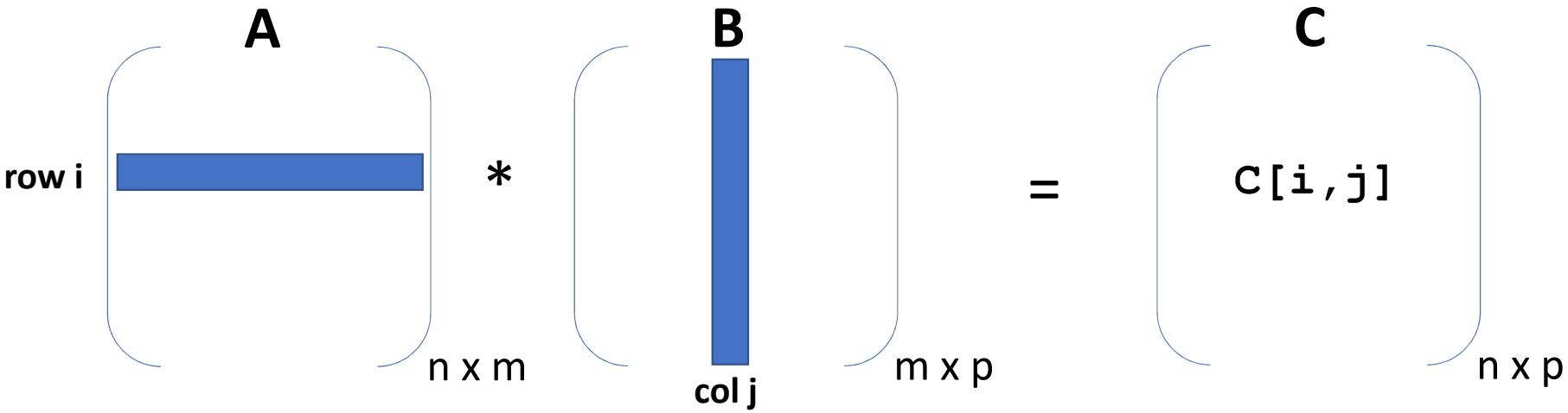
- the total number of **multiplications** **`M(n)`** is expressed by the following:

$$M(n) = \sum_{i=0}^{n-1}\sum_{j=0}^{n-1}\sum_{k=0}^{n-1} 1 = \sum_{i=0}^{n-1}\sum_{j=0}^{n-1} n = \sum_{i=0}^{n-1} n^2 = n^3 \in \Theta(n^3)$$

Brute-Force Algorithm: Exercise 1

Matrix Multiplication: given one `n` x `m` matrix (called `A`) and one `m` x `p` matrix (called `B`),

- write a brute-force version of MatrixMultiplication algorithm for computing their product `C` = `A` x `B` and
- find the time efficiency of such definition-based algorithm



Brute-Force and The Sorting Problem

COMPUTER SCIENCE
THAMMASAT UNIVERSITY

the application of the brute-force approach to the problem of sorting

The Sorting Problem: given a list of ***n*** orderable items, re-arrange them in non-decreasing order

- What would be the most straight-forward method for solving the sorting problem?

Brute-Force and The Sorting Problem:

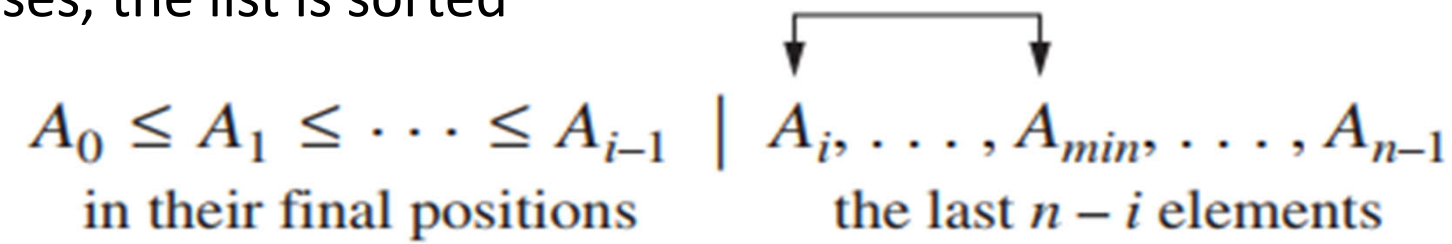
Selection Sort

COMPUTER SCIENCE
THAMMASAT UNIVERSITY

- The selection sort algorithm sorts an array by repeatedly finding the minimum element from unsorted part and putting it at the beginning.
- The algorithm maintains two subarrays in a given array.

1) The subarray which is already sorted.
2) Remaining subarray which is unsorted.

- In every iteration of selection sort, the minimum element from the unsorted subarray is picked and moved to the end of sorted subarray.
- After n-1 passes, the list is sorted



Brute-Force and The Sorting Problem:

Selection Sort : Example

COMPUTER SCIENCE
THAMMASAT UNIVERSITY

| 89	45	68	90	29	34	**17**
17 |	45	68	90	**29**	34	89
17	29 |	68	90	45	**34**	89
17	29	34 |	90	**45**	68	89
17	29	34	45 |	90	**68**	89
17	29	34	45	68 |	90	**89**
17	29	34	45	68	89 |	90

Brute-Force and The Sorting Problem: Selection Sort (3)

COMPUTER SCIENCE
THAMMASAT UNIVERSITY

ALGORITHM *SelectionSort*$(A[0..n-1])$

```
//Sorts a given array by selection sort
//Input: An array A[0..n − 1] of orderable elements
//Output: Array A[0..n − 1] sorted in nondecreasing order
for i ← 0 to n − 2 do
    min ← i
    for j ← i + 1 to n − 1 do
        if A[j] < A[min]  min ← j
    swap A[i] and A[min]
```

Brute-Force and The Sorting Problem:

Selection Sort (4)

COMPUTER SCIENCE
THAMMASAT UNIVERSITY

- the input size is given by the number of elements in the list ***n***
- the basic operation is the key comparison $A[j] < A[min]$

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

= (n-1) + (n-2) + (n-3) + … + 3 + 2 + 1 = $\sum_{i=1}^{n-1} i = \frac{(n-1)n}{2}$

$\in O(n^2)$

Brute-Force and The Sorting Problem: Bubble Sort

- Bubble Sort is the simplest sorting algorithm that works by repeatedly swapping the adjacent elements if they are in wrong order
- Pass *i (*$0 \le i \le n - 2$*)* of bubble sort can be represented by the following diagram:

$$A_0, \ldots, A_j \overset{?}{\leftrightarrow} A_{j+1}, \ldots, A_{n-i-1} \mid \underbrace{A_{n-i} \le \cdots \le A_{n-1}}_{\text{in their final positions}}$$

Brute-Force and The Sorting Problem:
Bubble Sort (2)

COMPUTER SCIENCE
THAMMASAT UNIVERSITY

- Example: sort the list (5, 1, 4, 2, 8) in non-decreasing order
  (เรียงจากน้อยไปมาก)

First Pass:

(**5 1** 4 2 8) –> (**1 5** 4 2 8),
(1 **5 4** 2 8) –> (1 **4 5** 2 8),
(1 4 **5 2** 8) –> (1 4 **2 5** 8),
(1 4 2 **5 8**) –> (1 4 2 **5 8**), I

Second Pass:

(**1 4** 2 5 8) –> (**1 4** 2 5 8)
(1 **4 2** 5 8) –> (1 **2 4** 5 8),
(1 2 **4 5** 8) –> (1 2 **4 5** 8)
(1 2 4 **5 8**) –> (1 2 4 **5 8**)

Third Pass:

(**1 2** 4 5 8) –> (**1 2** 4 5 8)
(1 **2 4** 5 8) –> (1 **2 4** 5 8)
(1 2 **4 5** 8) –> (1 2 **4 5** 8)
(1 2 4 **5 8**) –> (1 2 4 **5 8**)

Brute-Force and The Sorting Problem:

Bubble Sort (3)

COMPUTER SCIENCE
THAMMASAT UNIVERSITY

ALGORITHM *BubbleSort*$(A[0..n-1])$

//Sorts a given array by bubble sort
//Input: An array $A[0..n-1]$ of orderable elements
//Output: Array $A[0..n-1]$ sorted in nondecreasing order
for $i \leftarrow 0$ **to** $n-2$ **do**
    for $j \leftarrow 0$ **to** $n-2-i$ **do**
        if $A[j+1] < A[j]$ swap $A[j]$ and $A[j+1]$

- consider key comparison and key swap as the basic operations of the algorithm
- consider the worst-case where key swaps are done at every iteration

Brute-Force and The Sorting Problem:
Bubble Sort (4)

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

= (n-1) + (n-2) + (n-3) + … + 3 + 2 + 1 = $\sum_{i=1}^{n-1} i = \frac{(n-1)n}{2} \in O(n^2)$

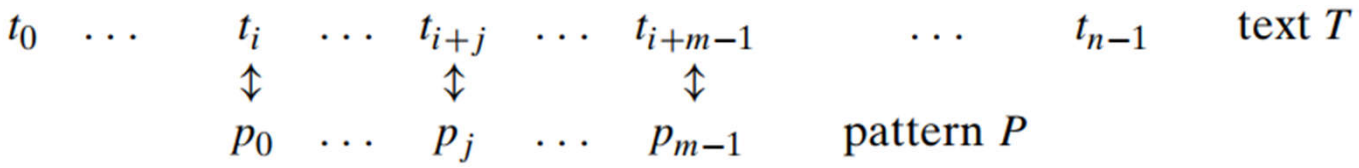
$S_{worst}(n) = C(n) \in O(n^2)$

The overall time complexity of the algorithm is $O(n^2)$

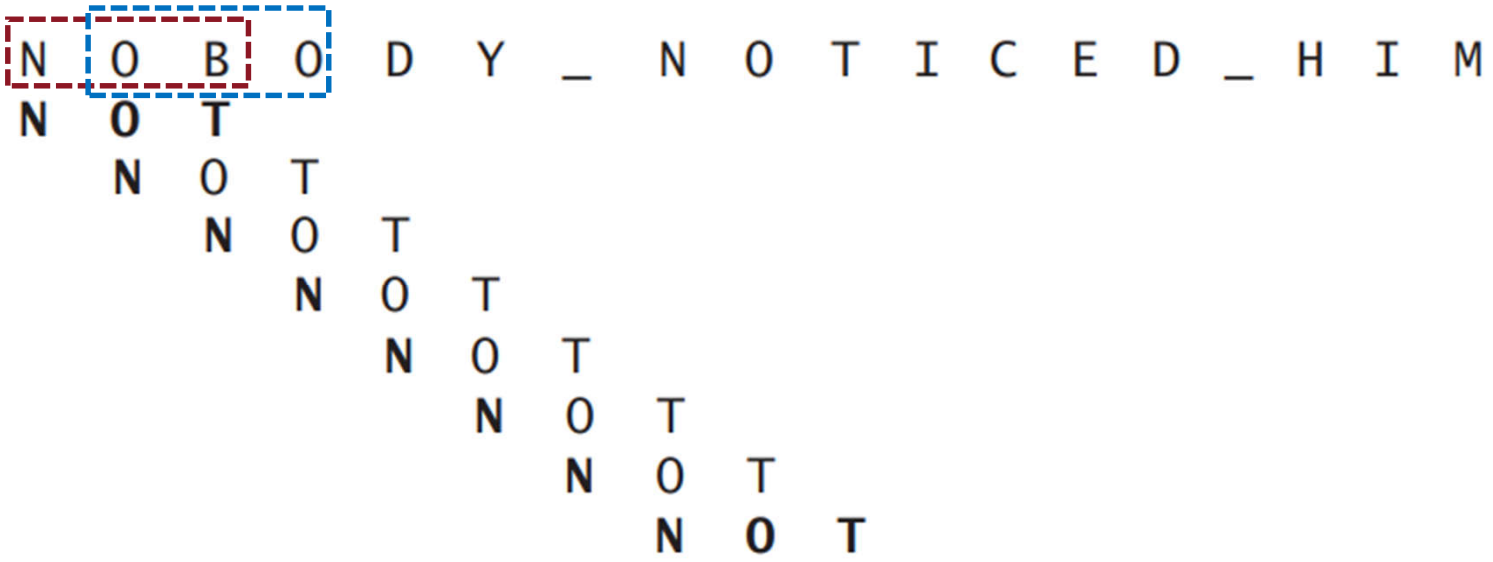
Brute-Force String Matching

COMPUTER SCIENCE
THAMMASAT UNIVERSITY

- given a string of *n* characters called the ***text***
  and a string of *m* characters *(m ≤ n)* called the ***pattern***,
  find a substring of the text that matches the pattern.



Brute-Force String Matching (2)



- Pattern = “NOT”

Brute-Force String Matching (3)

COMPUTER SCIENCE
THAMMASAT UNIVERSITY

ALGORITHM *BruteForceStringMatch*$(T[0..n-1],\ P[0..m-1])$

```
//Implements brute-force string matching
//Input: An array T[0..n − 1] of n characters representing a text and
//       an array P[0..m − 1] of m characters representing a pattern
//Output: The index of the first character in the text that starts a
//        matching substring or −1 if the search is unsuccessful
for i ← 0 to n − m do
    j ← 0
    while j < m and P[j] = T[i + j] do
        j ← j + 1
    if j = m return i
return −1
```

comparisons between corresponding characters in Pattern and current Text window

- The basic operation is the comparison between each character in Pattern and those in the current text window.
- worst-case scenario: we need to make all m comparisons prior to realizing that this is not a match for every text window

Brute-Force String Matching (4)

COMPUTER SCIENCE
THAMMASAT UNIVERSITY

- C(n) = $\sum_{i=0}^{n-m}(m) = m\ (n - m\ - 0 + 1)$

  = nm – m^2 + m
- Since $m \leq n$, C(n) $\in$ O($n\ m$)

Exercise 1: Prime Numbers

COMPUTER SCIENCE
THAMMASAT UNIVERSITY

- Given a positive integer `N` > 1, check whether `N` is a prime number.

Definitions:

I. A prime number is a whole number greater than 1 that **cannot** be made by multiplying other whole numbers.

II. A prime number is a whole number greater than 1, that has only two factors – 1 and the number itself.

III. A prime number is divisible only by the number 1 or itself.

Exercise 2: Nugget Numbers

COMPUTER SCIENCE
THAMMASAT UNIVERSITY

- ร้านฟาสต์ฟู้ดแห่งหนึ่งขายนักเก็ตเป็นกล่อง โดยมีกล่องนักเก็ตอยู่ 3 ขนาด ได้แก่ ขนาดเล็ก ขนาดกลาง และขนาดใหญ่ บรรจุนักเก็ต จำนวน 6 ชิ้น, 9 ชิ้น, และ 20 ชิ้น ตามลำดับ
- เลขนักเก็ต คือ เลขจำนวนเต็มบวกที่เกิดจากผลรวมของจำนวนชิ้นนักเก็ตในกล่องขนาดต่าง ๆ ยกตัวอย่างเช่น
  - เลข 6 เป็นเลขนักเก็ต เพราะเป็นจำนวนนักเก็ตในกล่องขนาดเล็กหนึ่งกล่อง
  - เลข 9 เป็นเลขนักเก็ต เพราะเป็นจำนวนนักเก็ตในกล่องขนาดกลางหนึ่งกล่อง
  - เลข 12 เป็นเลขนักเก็ต เพราะเกิดจากการรวมกันของจำนวนนักเก็ตในกล่องขนาดเล็ก 2 กล่อง
  - เลข 15 เป็นเลขนักเก็ต เพราะเกิดจากการรวมกันของจำนวนนักเก็ตในกล่องขนาดเล็ก และ ขนาดกลาง อย่างละหนึ่งกล่อง
  - เลข 20 เป็นเลขนักเก็ต เพราะเป็นจำนวนนักเก็ตในกล่องขนาดใหญ่หนึ่งกล่อง
  - เลข 4 และ 10 ไม่เป็นเลขนักเก็ต
- ให้นักศึกษาเขียนโปรแกรมเพื่อรับเลขจำนวนเต็มบวก N >= 6 และคำนวณพร้อมแสดงผลลัพธ์เป็นเลขนักเก็ตทั้งหมดที่มีค่าน้อยกว่าหรือเท่ากับ N

Brute-Force: Exhaustive Search

COMPUTER SCIENCE
THAMMASAT UNIVERSITY

Brute-Force: Exhaustive Search

COMPUTER SCIENCE
THAMMASAT UNIVERSITY

- ***Exhaustive search*** is simply a brute-force approach to combinatorial problems.
- It suggests generating each and every element of the problem domain, selecting those of them that satisfy all the constraints, and then finding a desired element (e.g., the one that optimizes some objective function).

- Traveling Salesman Problem, Knapsack Problem, Assignment Problem

Exhaustive Search: Traveling Salesman Problem

COMPUTER SCIENCE
THAMMASAT UNIVERSITY

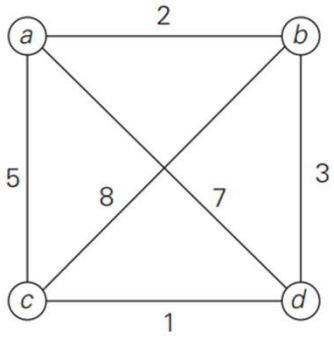
- Given a set of ***n*** cities, find the shortest tour through the whole city set such that the salesman visits each city only once before returning to the city where the tour started

Exhaustive Search: Traveling Salesman Problem

COMPUTER SCIENCE
THAMMASAT UNIVERSITY

- The problem can be modeled a weighted undirected graph G=(V, E):
  - V represents the set of cities
  - E maintains the road information between cities
  - Each edge in E has an associated weight specifying distance between two cities
- Then, the problem can be stated as the problem of finding the shortest ***Hamiltonian circuit*** of the graph.
- a Hamiltonian circuit is defined as a cycle that passes through all the vertices of the graph exactly once.

Exhaustive Search: Traveling Salesman Problem (2)



Tour	Length	
$a \rightarrow b \rightarrow c \rightarrow d \rightarrow a$	$l = 2 + 8 + 1 + 7 = 18$	
$a \rightarrow b \rightarrow d \rightarrow c \rightarrow a$	$l = 2 + 3 + 1 + 5 = 11$	optimal
$a \rightarrow c \rightarrow b \rightarrow d \rightarrow a$	$l = 5 + 8 + 3 + 7 = 23$	
$a \rightarrow c \rightarrow d \rightarrow b \rightarrow a$	$l = 5 + 1 + 3 + 2 = 11$	optimal
$a \rightarrow d \rightarrow b \rightarrow c \rightarrow a$	$l = 7 + 3 + 8 + 5 = 23$	
$a \rightarrow d \rightarrow c \rightarrow b \rightarrow a$	$l = 7 + 1 + 8 + 2 = 18$	

Exhaustive Search: Knapsack Problem

COMPUTER SCIENCE
THAMMASAT UNIVERSITY

- Given *n* items of known weights $w_1, w_2, \ldots, w_n$ and values $v_1, v_2, \ldots, v_n$ and a knapsack of capacity *W*,
  find the most valuable subset of the items that fit into the knapsack.
- Example: W = 10

	weight	value
1	7	$42
2	3	$12
3	4	$40
4	5	$25

Exhaustive Search: Knapsack Problem (2)

	weight	value
1	7	$42
2	3	$12
3	4	$40
4	5	$25

- list all combinations:
  C(n,0) + C(n,1) + C(n,2) + … + C(n,n)
- Ex n = 4
  all combinations = 1 + 4 + 6 + 4 + 1
  = 16 possible patterns

Subset	Total weight	Total value
∅	0	$ 0
{1}	7	$42
{2}	3	$12
{3}	4	$40
{4}	5	$25
{1, 2}	10	$54
{1, 3}	11	not feasible
{1, 4}	12	not feasible
{2, 3}	7	$52
{2, 4}	8	$37
{3, 4}	**9**	**$65**
{1, 2, 3}	14	not feasible
{1, 2, 4}	15	not feasible
{1, 3, 4}	16	not feasible
{2, 3, 4}	12	not feasible
{1, 2, 3, 4}	19	not feasible

Exhaustive Search: Assignment Problem

COMPUTER SCIENCE
THAMMASAT UNIVERSITY

- There are *n* people who need to be assigned to execute *n* jobs, one person per job. (That is, each person is assigned to exactly one job and each job is assigned to exactly one person.)
- The cost that would accrue if the i^{th} person is assigned to the j^{th} job is a known quantity $C[i, j]$ for each pair $i, j = 1, 2, \ldots, n$.
- The problem is to find an assignment with the minimum total cost.

Exhaustive Search: Assignment Problem (2)

COMPUTER SCIENCE
THAMMASAT UNIVERSITY

Example n = 4

- the exhaustive-search approach to the assignment problem would require ***generating all the permutations*** of integers 1, 2, . . . , *n*,
- computing the total cost of each assignment by summing up the corresponding elements of the cost matrix,
- and finally selecting the one with the smallest sum.

	Job 1	**Job 2**	**Job 3**	**Job 4**
Person 1	9	2	7	8
Person 2	6	4	3	7
Person 3	5	8	1	8
Person 4	7	6	9	4