Final: พุธ 25 ต.ค. 13.00-16.00

Data Structures and Algorithms

การเรียงลำดับ

Lecture 24: Sorting in Data Structures

Nopadon Juneam
Department of Computer Science
Kasetsart university

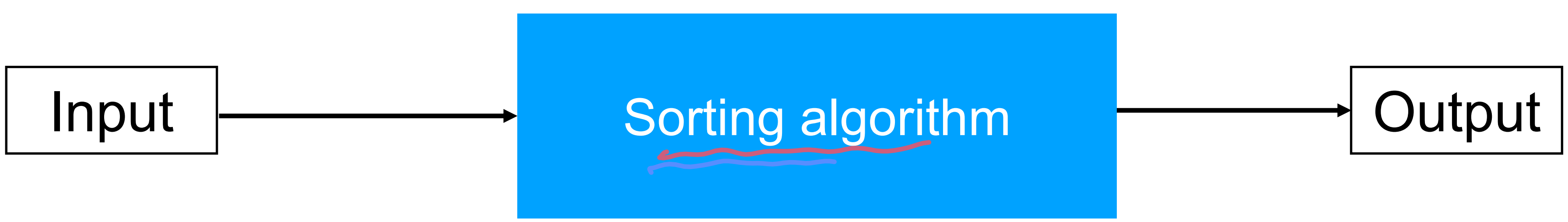
Outlines

- Sorting problem
- Basics of analysis of algorithms (reviews)
  - Time Complexity
  - Case Analysis
  - Asymptotic Notations
- Some basic comparison-based sorting algorithms and their analyses

analysis แยกเคส

Sorting Problem

- *Input:* A sequence of n numbers $a_1, a_2, \ldots, a_n$
- *Output:* A permutation $a'_1, a'_2, \ldots, a'_n$ such that
  $a'_1 \leq a'_2 \leq \ldots \leq a'_n$



Analysis of Algorithms

ประสิทธิภาพ

- It is the study of the *performance* of an algorithm.

ความซับซ้อนทางการคำนวณ

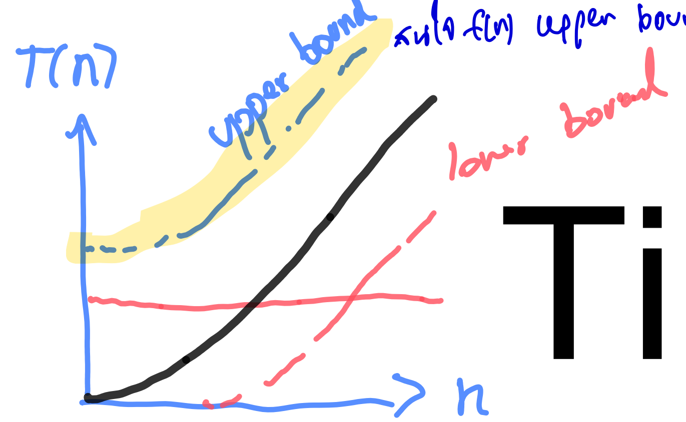
- The analysis consists in finding the **computational complexity** of the algorithm

การใช้งานทรัพยากรทางการคำนวณ

สูง performance ต่ำ

สูง

  - i.e., to find out about the computational resources such as running time (*time complexity*), memory storage (*space complexity*) needed to execute the algorithm



สนใจ f(n) upper bound

Time แปรตาม input size

Time Complexity

วัดเวลาที่ algorithm แต่ละๆ ใช้ในการทำงาน

- **Time complexity** measures the amount of time it takes to run an algorithm, and it varies with
  - The size of the input (e.g., sorting 10 numbers vs 10^6 numbers)
  - The input itself (e.g., the execution paths on different inputs of the same size are usually different)
- Generally, the time complexity is expressed as "*a function of the size of the input*." Usually, the size of the input is denoted by *n*

  T(n)
- Moreover, unless mentioned otherwise, we are mostly interested in the function that provides "*an upper bound on the time it takes to run the algorithm for any input*"

Case Analysis

$T_{max}(n) = n^2$

$T(n) = 100n^2 + 10n$

- **Worst case analysis** (usual analysis): The analysis consists in finding

  $T_{max}(n)$:= maximum time on any input of size *n*

- **Average case analysis**: The analysis consists in finding

  $T_{avg}(n)$:= average (expected) time over all inputs of size *n*

  This kind of analysis typically needs some assumption of statistical distribution of inputs.

$T(n) = 100n^2 + 9n$ ต่างกัน n เท่า

- **Best case analysis**: The analysis consists in finding

  $T_{min}(n)$:= minimum time on any input of size *n*

  This kind of analysis is bogus as slow algorithms may work fast for particular inputs

$T_{min}(n) = n^2$

$$\boxed{T(n) = \underbrace{300}_{c_1}n^2 + \underbrace{40}_{c_2}n + \underbrace{150}_{c_3}}$$

exact function

สนใจ behavior ของ f(n) เมื่อ n เข้าใกล้ infinity

Asymptotic Analysis

$$O(T(n)) = O(n^2)$$

order of growth

- In the analysis of algorithms, one commonly focuses on the **asymptotic behavior** of the time complexity, that is, the behavior of the function that describes the time complexity when the input size increases (*the growth of the function*). The reasons are
  - It is often difficult to compute the function exactly
  - The time it takes to run an algorithm with small inputs is usually not consequential
- Therefore, the time complexity is commonly expressed using *asymptotic notations* (e.g., Big-O, Big-Theta), and hence commonly estimated by *the number of elementary operations* performed by an algorithm (suppose each elementary operation takes a constant amount of time to execute).

Asymptotic Notations

- **Asymptotic notations** are used to describe the behavior of functions in the limit, i.e., to describe the growth of the functions

- Informally, when expressing a function with asymptotic notation, we are allowed to

  - Abstract low-order terms and constants, and focus more on the highest terms which have the most effects on the growth of the function

  - Compare "*sizes*" of the functions.
    Big-O (use "$O(\cdot)$" like "≤")
    Big-Omega (use "$\Omega(\cdot)$" like "≥")
    Big-Theta (use "$\Theta(\cdot)$" like "=")

$n+2 = O(\cancel{n^2+2n+2})$

$n+2 = O(n^2)$

$n+2 = \Theta(n)$

f(n) ที่ ≤ ที่เป็นขอบบน

$10n+5 \le 15n$

$10n+5 = O(n)$

Big-O Notation (Informal Notion) (1)

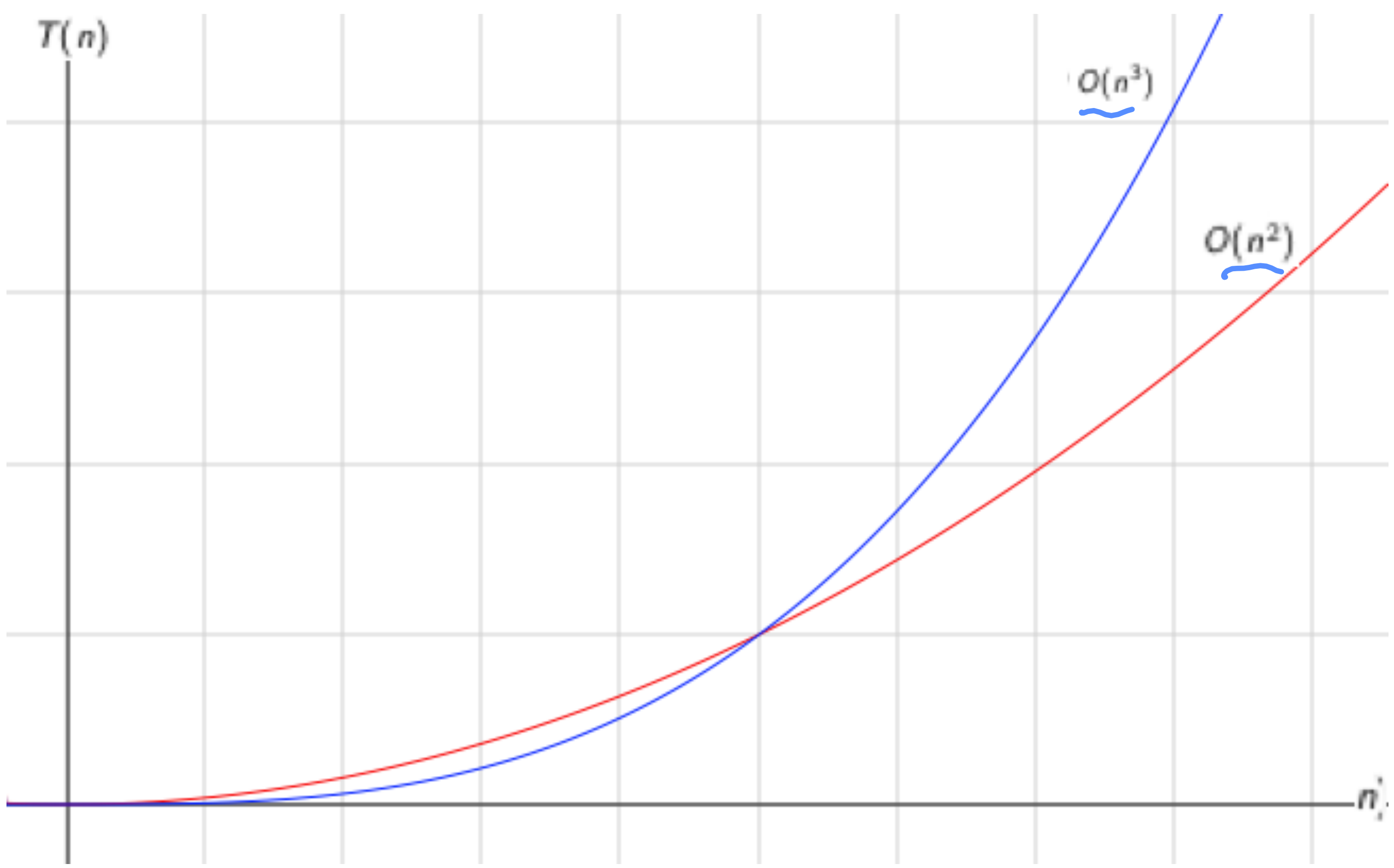
- ***O*-notation:** Informally, we obtain $O(T(n))$ by dropping the low-order terms and ignoring leading constants in the original function

- For example,
  $4n^3 + 9n^2 + 11n + 102 = O(n^3)$
  $100n^2 + 1000n + 40560 = O(n^2)$

- Informally, we can also say that if an algorithm has complexity $O(T(n))$, then the amount of time it takes to run the algorithm behaves like $T(n)$

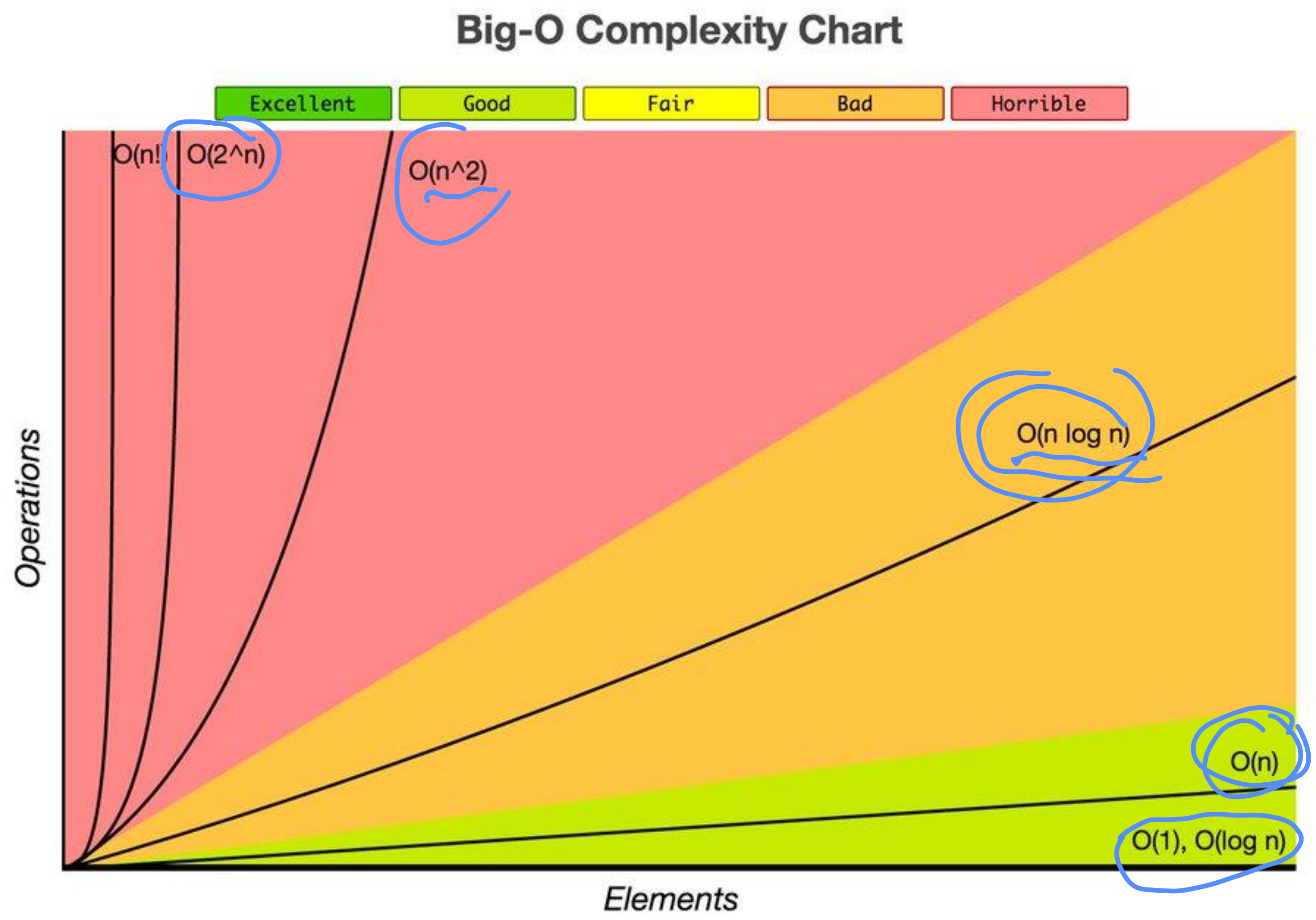
Big-O Notation (Informal Notion) (2)

- As $n \to \infty$, any algorithm with complexity $O(n^2)$ always outperforms any algorithm with complexity $O(n^3)$



Big-O and Order of Growth

- The letter *O* is used as the growth rate of a function is also refereed to as "**Order of the Function**"



Big-O Notation (Definition)

- ***O*-notation:** Formally, $O(g(n)) = \{f(n) \mid$ there exist positive constants c and n_0 such that $0 \le f(n) \le c\, g(n)$, for all $n > n_0\}$

ถูกตามนิยาม

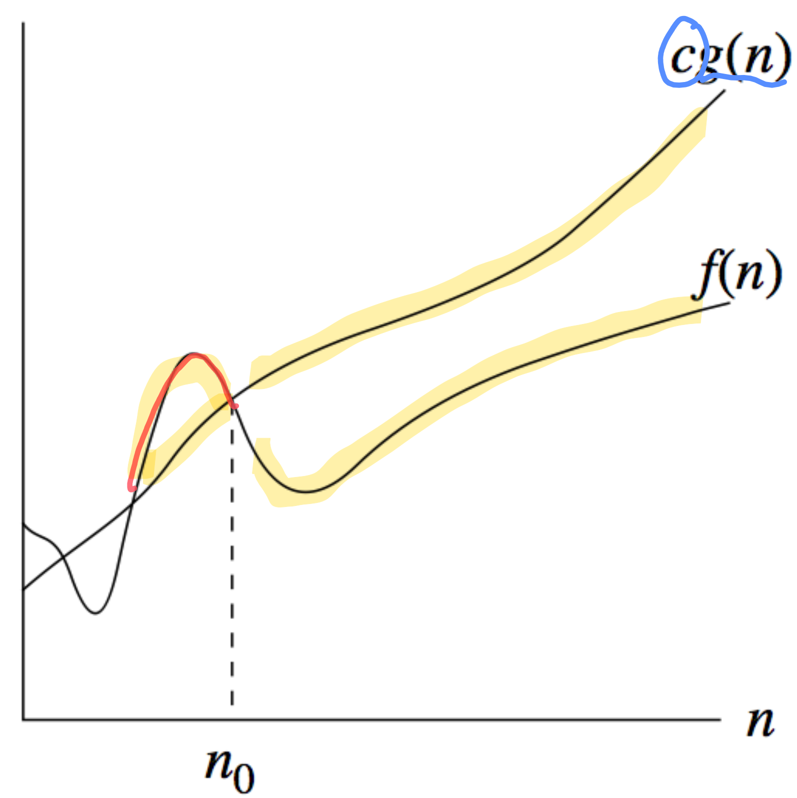
f(n)

เป็น fn ของ O(n)

"$400n^2+10n \in O(n^2)$"

"$400n^2+10n = O(n^3)$"

ไม่ถูกตามนิยาม



$400n^2+10 \in O(n^2)$

$400n^2+10 \le c \cdot n^2$

410

$400n^2+10 \le 400n^2+10n^2$

$400n^2+10 \in O(n^3)$

$400n^2+10 \le 410n^3$

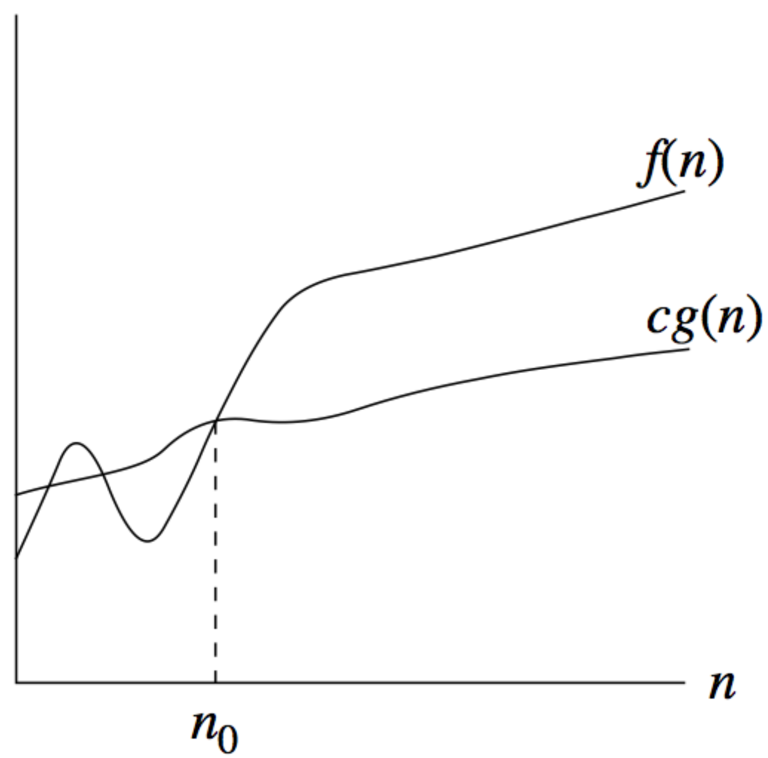
That is, $g(n)$ is an "*asymptotic upper bound*" for $f(n)$.
When $f(n) \in O(g(n))$, we also write $f(n) = O(g(n))$

$3n^2+2n+1 \le 6 \cdot n^2$

$3n^2+2n^2+1$

Big-Omega Notation (Definition)

- ***Ω*-notation:** Formally, $\Omega(g(n)) = \{f(n) \mid$ there exist positive constants c and n_0 such that $0 \le c\, g(n) \le f(n)$ for all $n > n_0\}$



$\Theta(n^2)$

$c_1 n^2 + c_2 n + c_3 = O(n^2)$

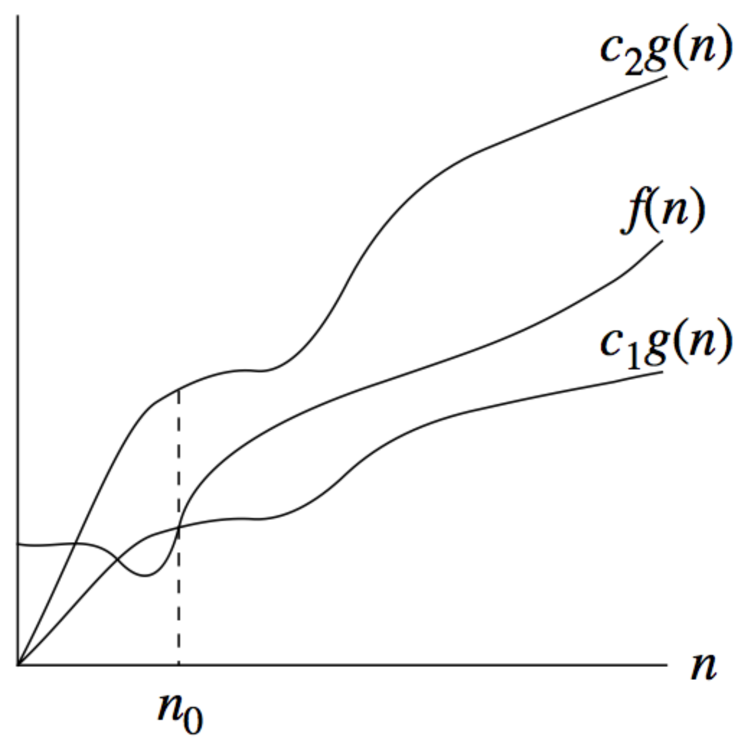
$= \Omega(n^2)$

$\Theta(n^2)$

That is, $g(n)$ is an "*asymptotic lower bound*" for $f(n)$.
When $f(n) \in \Omega(g(n))$, we also write $f(n) = \Omega(g(n))$

Big-Theta Notation (Definition)

- ***Θ*-notation:** Formally, $\Theta(g(n)) = \{f(n) \mid$ there exist positive constants c_1, c_2 and n_0 such that $0 \le c_1 g(n) \le f(n) \le c_2 g(n)$ for all $n > n_0\}$



$\boxed{\Theta(n^2)}$

$O(n^2)$

$\Omega(n^2)$

That is, $g(n)$ is an "*asymptotic tight bound*" for $f(n)$.
When $f(n) \in \Theta(g(n))$, we also write $f(n) = \Theta(g(n))$

sorting algorithms

Comparison-Based Sort

- A type of *sorting algorithms* that sort a given sequence of elements based only on *comparison* of the elements in the desired ordering
  - i.e., a comparison sort is based on comparison operations such as $a_i < a_j$ or $a_i \leq a_j$ (to determine whether a_i comes before a_j)
- Well-known comparison sorts include *Selection Sort*, *Insertion Sort*, *Merge~~r~~ Sort*, *Quick Sort*, and *Heap Sort*
- ****Fact (Sorting Lower Bound):** Any comparison-based sorting algorithm requires at least Ω ($n \log n$) operations

Sorting in Array

- Assume the input is presented as an array A[1..n] where each A[i] = a_i

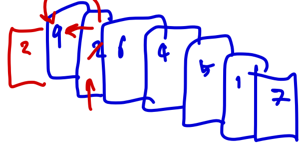
- Famous sorting algorithms that operates on array includes

  ไม่ต้องการใช้ space มากกว่าการเก็บ array เอง หรือไม่ต้องการ Extra space

  - *In-place sorts:* Insert Sort, Selection Sort, Heap Sort, etc.

    ion

  - *Out-of-place sorts* (extra space required)*:* Merge Sort, BST Sort (Tree Sort), etc.

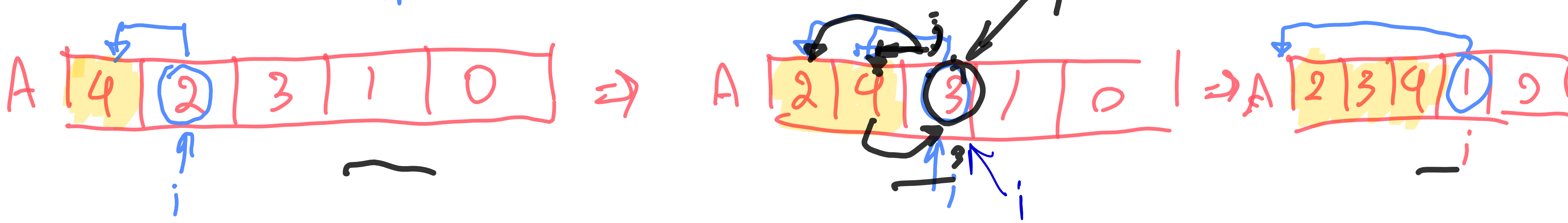


T < A[j]

Insertion Sort (1)

การเรียงลำดับแบบแทรก

ต้องเทียบใน sorted region ทุกตัว



- *Insertion Sort* is a simple sorting algorithm that tries to grow the sorted output sequence in each iteration.

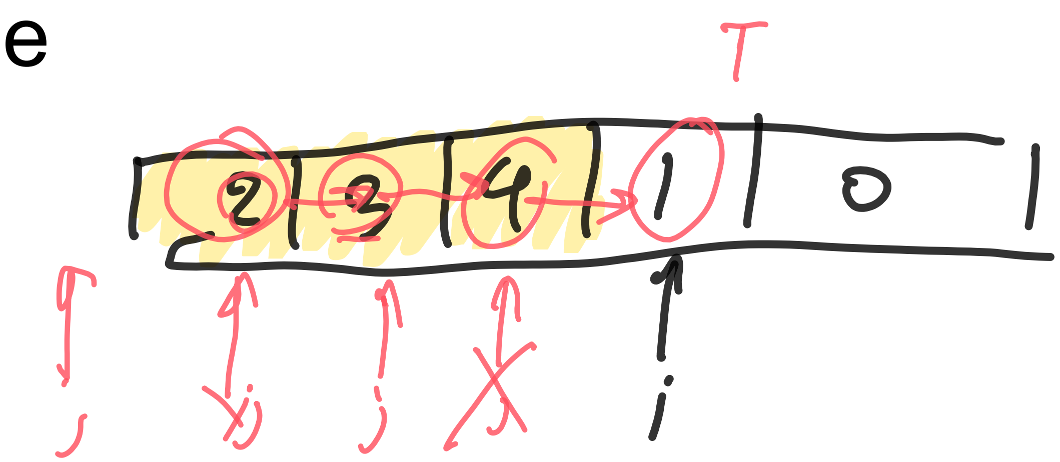
- The algorithm performs n-1 iterations. At the i-th iteration, $2 \leq i \leq n$, the algorithm finds the appropriate position of element a_i in the sorted subarray A[1..i-1], and insert it at that position. After the insertion, the subarray A[1..i] becomes sorted.

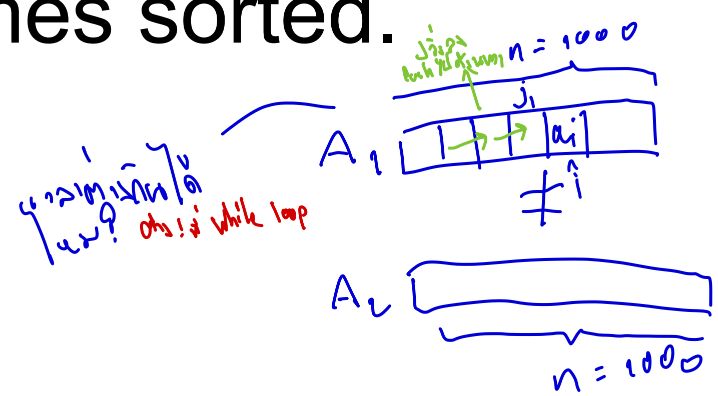
Insertion Sort (2)

- The algorithm performs *n*-1 iterations.

- At the *i*-th iteration, 2 ≤ *i* ≤ *n*, the algorithm finds the appropriate position of element a_i in the sorted subarray A[1..*i*-1], and insert it at that position. After the insertion, the subarray A[1..*i*] becomes sorted.

```
InsertionSort(A, i, n):
    for i=2 to n:
        T = A[i]
        j = i-1
        while j > 0 and A[j] > T:
            A[j+1] = A[j]
            j = j-1
        j = j+1
        A[j] = T
```





Analysis of Insertion Sort (1)

- The complexity of Insertion Sort varies, depending on the execution of the while loop

  B.C.

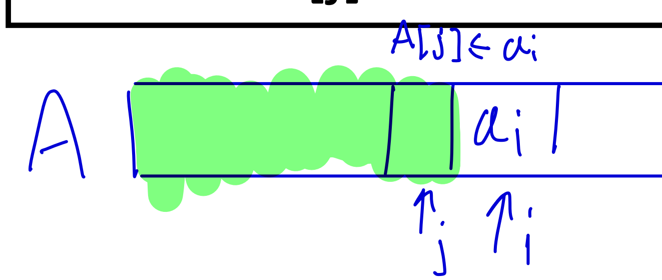
  - When the input is already sorted, the while loop test is executed one time

  W.C.

  - When the input is already reverse-sorted, the while loop test is executed *i*-1 times during iteration *i* of the for loop

```
InsertionSort(A, i, n):
    for i=2 to n:
        T = A[i]
        j = i-1
        while j > 0 and A[j] > T:
            A[j+1] = A[j]
            j = j-1
        j = j+1
        A[j] = T
```

เป็นจริง



B.C. A[i] ≥ A[i-1] เสมอ สำหรับทุกค่า i

Analysis of Insertion Sort (2)

- Let's assume that
  - คำสั่งที่ปรากฏในบรรทัดที่ l
  - The instruction at the *l*-th line takes time constant c_l ← cost
  - จำนวนการloop
  - For *i* = 2,3, …, *n*, let t_i be the number of times that the while loop test is executed for that value of *i*

```
InsertionSort(A, i, n):
1    for i=2 to n:          ← c1 n
2         T = A[i]          c2 (n-1)
3         j = i-1           c3 (n-1)
4         while j > 0 and A[j] > T:
5              A[j+1] = A[j]   ← c5
6              j = j-1         ← c6
7         j = j+1           ← c7 (n-1)
8         A[j] = T          ← c8 (n-1)
```

$\sum_{i=2} (c_4 t_i + c_5(t_i-1)$ ~ ต่อไปก็

- The complexity of Insertion Sort is expressed as

$$T(n) = c_1 n + c_2(n-1) + c_3(n-1) + c_4\left(\sum_{i=2}^{n} t_i\right) + c_5\left(\sum_{i=2}^{n} (t_i-1)\right) + c_6\left(\sum_{i=2}^{n} (t_i-1)\right) + c_7(n-1) + c_8(n-1)$$

Best Case Analysis of Insertion Sort

$\sum_{i=2}^{n} t_i = \sum_{i=2}^{n} 1 = n-1$

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

- **Best case analysis:** When the input is already sorted, the while loop test is executed one time.

  while loop ทำงานรอบเดียว

  - The while loop test always finds that A[j] ≤ T.
  - So, all $t_i = 1$
- The best-case complexity of Insertion Sort is expressed as
  $T_{min}(n) = c_1 n + c_2(n\text{-}1) + c_3(n\text{-}1) + c_4(n\text{-}1) + c_7(n\text{-}1) + c_8(n\text{-}1)$
  $= \Theta(n)$

  linear function

Worst Case Analysis of Insertion Sort

- **Worst case analysis:** When the input is already reverse-sorted, the while loop test is executed *i*-1 times during iteration *i* of the for loop

  - So, all $t_i = i$-1, and thus
    $\sum_{i=2}^{n} t_i = \sum_{i=2}^{n} 1 = n(n+1)/2$
    $\sum_{i=2}^{n} (t_i\text{-}1) = \sum_{i=1}^{n-1} 1 = n(n\text{-}1)/2$

$\sum_{i=2}^{n} t_i = \sum_{i=2}^{n} (i-1) = 1+2+\dots+(n-1)$

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

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

$\frac{(n-2)(n-1)}{2}$

- The worst-case complexity of Insertion Sort is expressed as

$$
\begin{aligned}
T_{max}(n) = {} & c_1 n + c_2(n\text{-}1) + c_3(n\text{-}1) + c_4(n(n+1)/2) \\
& + c_5(n(n-1)/2) + c_6(n(n-1)/2) + c_7(n\text{-}1) \\
& + c_8(n\text{-}1) \\
= {} & \Theta(n^2)
\end{aligned}
$$

$c_1 n^2 + c_2 n + c_3$

Complexity of In-Place Sorts

Algorithm	B.C. Complexity	W.C. complexity
Insertion Sort	$\Theta(n)$ => $O(n)$	$\Theta(n^2)$ => $O(n^2)$

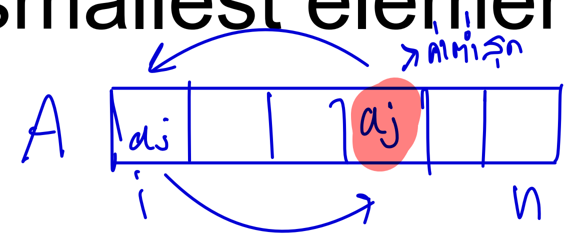
Selection Sort (1)

comparison, in-plance sort

- *Selection Sort* is one of the simplest sorting algorithms. It is noted for its *simplicity*

แสดงหา Index ของสมาชิกที่มีค่าต่ำที่สุด

- Selection Sort employs the operation Find-Min-Index(A, i, n) to finding the index of a smallest element as its main subroutine.



- The algorithm performs in *n* iterations. At the *i*-th iteration, it swaps an element at index *i* with the element at index found by Find-Min-Index(A, i, n)

Selection Sort (2)

SelectionSort(A, i, n):
    for i=1 to n:
        j = Find-Min-Index(A, i, n)
        swap(A[i], A[j])

Find-Min-Index(A, i, n):
    j = ~~1~~ i
    T = A[i] } By default
    i = i+1
    while i < n: → n-i
        if A[i] < T: → n-i
            T = A[i]
            j = ~~1~~ i
        i = i+1
    return j $\Theta(1)$

$\Theta(n-i) =$

$\Theta(n-1)$

$T(n) = \Theta(n) + \sum_{i=1}^{n} \Theta(n-i)$

- The complexity of Find-Min-Index is $\Theta(n\text{-}i)$

- The complexity of Selection Sort is expressed as $T(n) = \sum_{i=1}^{n} \Theta(n-i) = \Theta(n^2)$

W.C & B.C

Analysis of Selection Sort

- The complexity of Selection Sort is always the same for any input
    - The complexity of Find-Min-Index is always $\Theta(n\text{-}i)$
- Therefore, the complexity of Selection Sort is expressed as $T(n) = \sum_{i=1}^{n} \Theta(n-i) = \Theta(n^2)$

Complexity of In-Place Sorts

Algorithm	B.C. Complexity	W.C. complexity
Insertion Sort	$\Theta(n)$	$\Theta(n^2)$
Selection Sort	$\Theta(n^2)$	$\Theta(n^2)$

Heap Sort (1)

$\Theta(n \log n)$

comparison

low time complexity

- *Heap Sort* is one of the most efficient in-place sorting algorithm. The idea of the algorithm is like Selection Sort, but it operates on heap as the underlying data structure

- Heap Sort works as follows:

  - At the first step, the algorithm applies Build-Max-Heap(A, *n*) so that the array A[1..*n*] becomes a heap

    ตัว root เป็น Max ทันที

  - Then, the algorithm iterates *n*-1 times (from *i*=*n* down to *i*=2). At the iteration of *i*, it swaps the values between A[i] and A[1], and calls Max-Heapify(A, 1, i-1)

    restore A[1 .... i-1]

    swap (A[n], A[i])

Heap Sort (2)

- Heap Sort works as follows:
  - At the first step, the algorithm applies Build-Max-Heap(A, *n*) so that the array A[1..*n*] becomes a heap
  - Then, the algorithm iterates *n*-1 times (from *i*=*n* down to *i*=2). At the iteration of *i,* it swaps the values between A[i] and A[1], and calls Max-Heapify(A, 1, i-1)

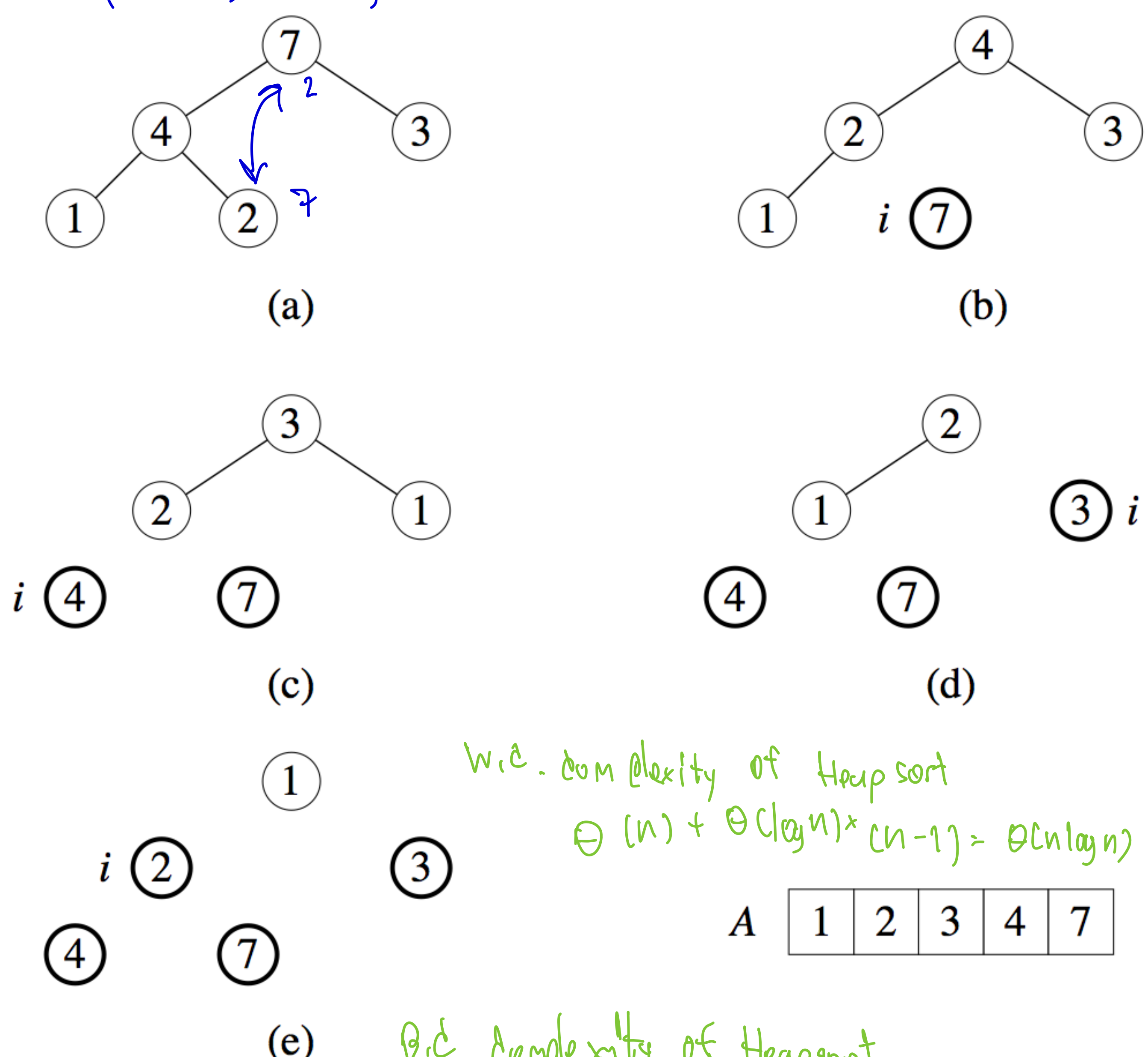
```
HeapSort(A, n):
    Build-Max-Heap(A, n)
    for i=n down to 2:
        swap(A[1], A[i])
        Max-Heapify(A, 1, i-1)
```

W.C. complexity of Build Max Heap is O(n log n)

Heap Sort in Actions

A

7	4	3	1	2
1	2	3	4	5



claim: B.C & W.C complexity of Build max Heap is $\Theta(n)$

W.C. complexity of Heap sort
$\Theta(n) + \Theta(\log n) * (n-1) = \Theta(n \log n)$

B.C complexity of Heapsort
$\Theta(n) + \Theta(1) * (n-1)$
$= \Theta(n)$

complexity varies

```
HeapSort(A, n):
    Build-Max-Heap(A, n)
    for i=n down to 2:    Θ(n)
        swap(A[1], A[i])
        Max-Heapify(A, 1, i-1)
```

W.C = $O(h)$ = $\Theta(\log n)$ B.C = $O(1)$

Analysis of Heap Sort

- Assume the input array contains *n* distinct numbers. Then, the complexity of Heap Sort is almost the same for any input
  - When the input is reverse sorted, the array A[1..*n*] is already a heap. The call to Build-Max-Heap takes time $\Theta(n)$. The call to Max-Heapify at the iteration of *i* takes time $\Theta(\log i)$.
  - For any input, the call to Build-Max-Heap still takes time $\Theta(n)$ time. The call to Max-Heapify at the iteration of *i* takes time $\Theta(\log i)$
- Therefore, the complexity of Heap Sort is expressed as $T_{max}(n) = \Theta(n) + \sum_{i=2}^{n} \Theta(\log i) = \Theta(n \log n)$
- Note that if the input array contains *n* equal numbers, then the complexity of the Heap Sort is just $\Theta(n)$. Why?

Complexity of In-Place Sorts

Algorithm	B.C. Complexity	W.C. complexity
Insertion Sort	$\Theta(n)$	$\Theta(n^2)$
Selection Sort	$\Theta(n^2)$	$\Theta(n^2)$
Heap Sort - n distinct numbers - n equal numbers	 $\Theta(n \log n)$ $\Theta(n)$	 $\Theta(n \log n)$ $\Theta(n)$

Heapsort | $\Theta(n)$ | $\Theta(n\log n)$

4 2 1 0 3
2 1 0 3 4
1 0 2 3 4
0 1 2 3 4

Bubble Sort (1)

- *Bubble Sort* is another simplest sorting algorithm, but it performs poorly in practice.

- The algorithm perform in passes. In each pass, it repeatedly steps through the input element by element, comparing the current element with the one after it, and making swaps if necessary.

- These passes through the input are repeated until no swaps is founded during a pass, meaning that all elements are already sorted

Bubble Sort (2)

- The algorithm perform in passes. In each pass, it repeatedly steps through the input element by element, comparing the current element with the one after it, and making swaps if necessary.

- These passes through the input are repeated until no swaps is founded during a pass, meaning that all elements are already sorted

```
BubbleSort(A, n):
    notFinish = True
    while(notFinish):
        notFinish = False
        for i=1 to n-1:
            if A[i] > A[i+1]:
                swap(A[i], A[i+1])
                notFinish = True
            n=n-1
```

Analysis of Bubble Sort (1)

- The analysis of Bubble Sort is like that of Insertion Sort
- The complexity of Bubble Sort varies, depending on whether a swap is occurred in a pass

B.c

  - When the input is already sorted, the algorithm performs just one pass

W.c

  - When the input is already reverse-sorted, the algorithm performs just n-1 passes

$\Theta(n+1-i)$

  - The complexity of the i-th pass is just $\Theta(n\text{-i})$

Analysis of Bubble Sort (2)

- **Best case analysis:** When the input is already sorted, the algorithm performs just one pass.

$$T_{min}(n) = \cancel{\Theta(n\text{-}1)} = \Theta(n)$$

- **Worst case analysis:** When the input is already reverse-sorted, the algorithm performs just n-1 passes.

$\Theta(n+1-i) = \Theta(n-i)$

$$T_{max}(n) = \sum_{i=1}^{n-1} \Theta(n-i) = \Theta(n^2)$$

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

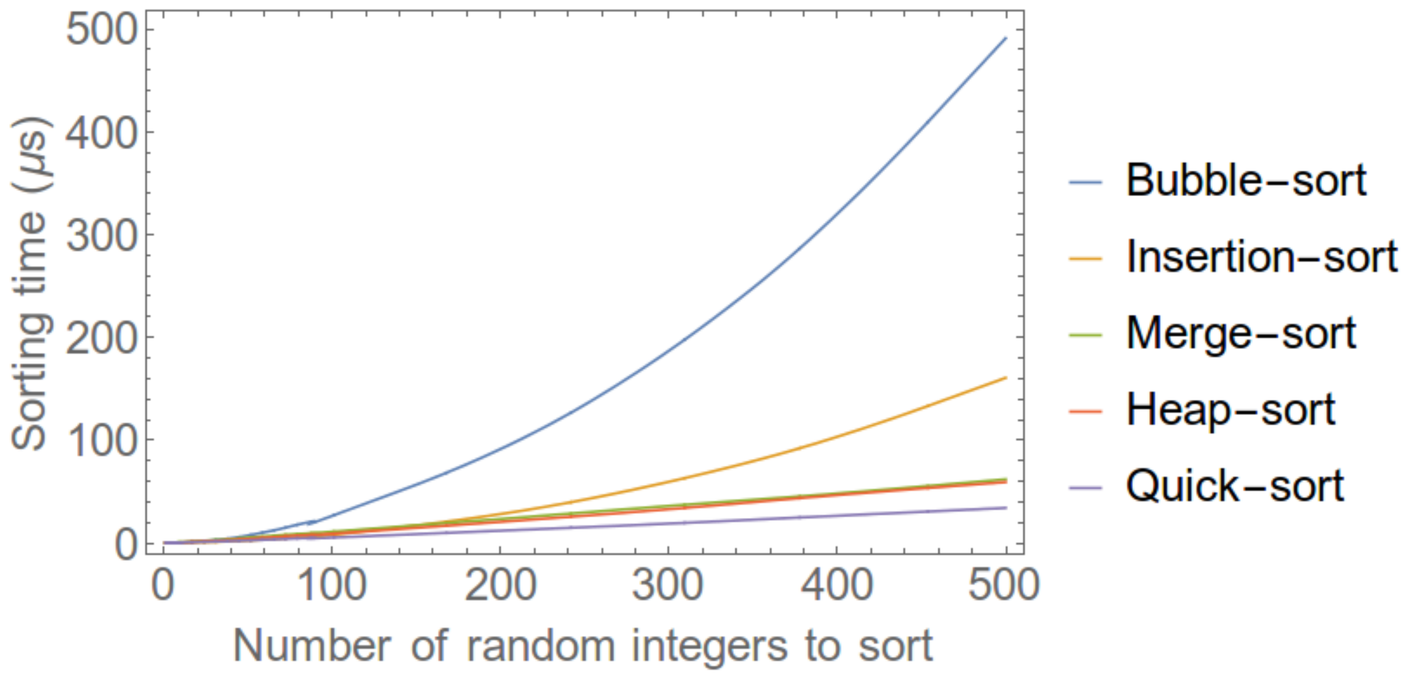
$\Theta(\sum_{i=1}^{n-1}(n-i))$

$= \Theta(n^2)$

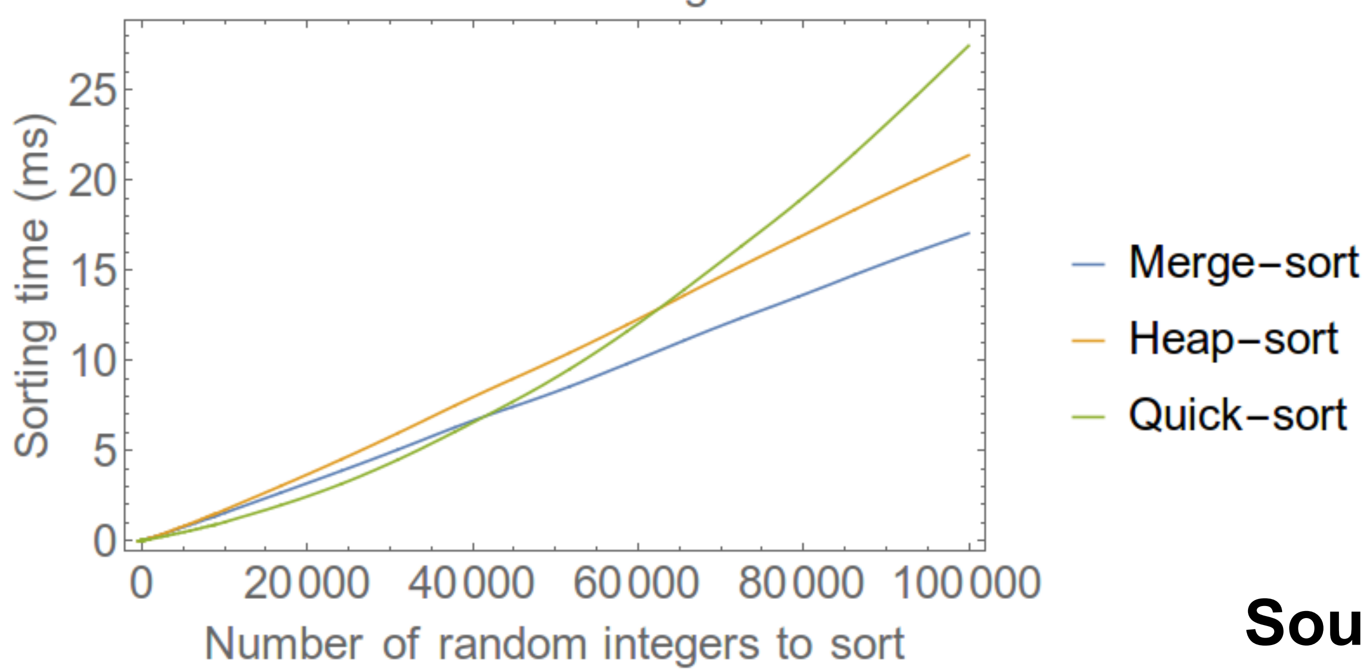
Complexity of In-Place Sorts

Algorithm	B.C. Complexity	W.C. complexity
Insertion Sort	$\Theta(n)$	$\Theta(n^2)$
Selection Sort	$\Theta(n^2)$	$\Theta(n^2)$
Heap Sort - n distinct numbers - n equal numbers	 $\Theta(n \log n)$ $\Theta(n)$	 $\Theta(n \log n)$ $\Theta(n)$
Bubble Sort	$\Theta(n)$	$\Theta(n^2)$

Experimental Performance of Sorting Algorithms



– Input ไม่เยอะ ต้องไม่ใช้กัน



Source: [Virtual Labs (vlabs.ac.in)]