Algorithm Analysis

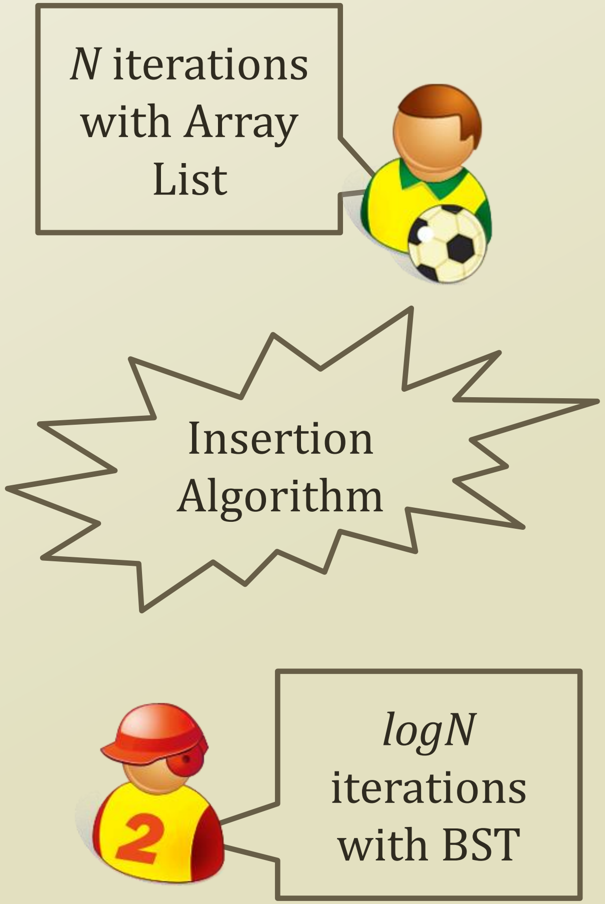
Il-Chul Moon
Dept. of Industrial and Systems Engineering
KAIST
icmoon@kaist.ac.kr

Weekly Objectives

- This week, we learn how to analyze the efficiency of our program
  - Algorithm analysis
- Objectives are
  - Memorizing the definition and the rules of the big-Oh notation
  - Understanding what determines the efficiency of programs
  - Understanding simple algorithms
    - Memorizing the insert and the delete of lists, stacks, and queues
    - Memorizing the bubble sort
  - Able to apply the big-Oh notation analysis to programs

Factors of program's efficiency

- Algorithm
  - A clearly specified set of simple instructions to be followed to solve a problem
    - Takes a set of values as inputs
    - Produces a set of values as outputs
  - Specified in
    - English
    - A computer program
    - Pseudo-code
- Data structures
  - Methods of organizing data
- Program
  - = algorithms + data structures



Bubble sort algorithm

- Examples of algorithms
  - Insertion, deletion, search of linked lists, stacks, queues…
  - Sorting of linked lists…
    - Various sorting methods
      - Bubble sort, Quick sort, Merge sort…
- Bubble Sort(list)
  - For itr1=0 to length(list)
    - For itr2=itr+1 to length(list)
      - If list[itr1] < list[itr2]
        - Swap list[itr1], list[itr2]
  - Return list
- This program uses
  - Data structure: List
  - Algorithm: Bubble sort

```python
import random

def performSelectionSort(lst):
    for itr1 in range(0, len(lst)):
        for itr2 in range(itr1+1, len(lst)):
            if lst[itr1] < lst[itr2]:
                lst[itr1], lst[itr2] =\
                lst[itr2], lst[itr1]
    return lst

N = 10
lstNumbers = list(range(N))
random.shuffle(lstNumbers)

print(lstNumbers)
print(performSelectionSort(lstNumbers))

lstNumbers2 = [2, 5, 0, 3, 3, 3, 1, 5, 4, 2]

print(lstNumbers2)
print(performSelectionSort(lstNumbers2))
```

```
[9, 3, 6, 7, 1, 5, 0, 2, 4, 8]
[9, 8, 7, 6, 5, 4, 3, 2, 1, 0]
[2, 5, 0, 3, 3, 3, 1, 5, 4, 2]
[5, 5, 4, 3, 3, 3, 2, 2, 1, 0]
```

Example of bubble sort execution

- Let's observe the execution of the bubble sort

```python
import random

def performSelectionSort(lst):
    for itr1 in range(0, len(lst)):
        for itr2 in range(itr1+1, len(lst)):
            if lst[itr1] < lst[itr2]:
                lst[itr1], lst[itr2] =\
                lst[itr2], lst[itr1]
    return lst
```

[2, 5, 0, 3, 3, 3, 1, 5, 4, 2]

→ **(itr1 = 0, itr2=1..9) = 9 iterations**
  - → (itr1 = 0, itr2 = 1)
    - → 2 < 5, Hit and swap!!!
    - → list[0] = 5, list[1] = 2 from now
  - → (itr1 = 0, itr2 = 2)
    - → 5<0, No hit
  - → (itr1 = 0, itr2 = 3)
    - → 5<3, No hit
  - → .....

→ **(itr1 = 1, itr2=2..9) = 8 iterations**
  - → ....

→ ....

→ **(itr1 = 8, itr2=9..9) = 1 iterations**
  - → .....

- Total iterations
  - = 9+8+....+1
  - =45 iterations
  - $=\frac{n(n-1)}{2}=\frac{1}{2}n^2-\frac{1}{2}n$

Why do we care about efficiency?

- Writing a working program is not good enough
  - The program could be inefficient
  - If the program runs on a large data, the running time becomes a big issue
    - Sometimes, a program may not be usable because of the efficiency
    - Imagine a transaction system of a financial company
      - 1 transaction = 0.001 sec
      - 10 transactions by 10,000 account holders = 100 sec
      - Side effect
        → If there is no reaction from the system, the users click the request again!
        → Increased requests when there is a delay
    - Imagine a bubble sorting function for bank accounts
      - 10,000 accounts → roughly 50,000,000 iterations for sorting
- Therefore, we need a guarantee of the worst-case scenario
  - The worst-case running time of a single transaction
  - The worst-case transaction request numbers of a single day

Definition of Algorithm Analysis

- Analyzing an algorithm
  - Estimating the resources that the algorithm requires
    - Memory
    - Communication bandwidth
    - Computational time (the most important resource in the most of cases)
- Factors affecting the running time
  - Computer used for executions
  - Algorithms
  - Data structures
  - Input data size
- After analyzing the algorithms
  - We estimate the worst-case of the costs by the factors
    - i.e. Computational time by input data size
    - i.e. Iterations by input data size

Simple algorithm analysis

```
def calculateIntegerRangeSum(intFrom, intTo):
    intSum = 0                                1

    for itr in range(intFrom, intTo):         2
        intSum = intSum + itr                 3

    return intSum                             4

print(calculateIntegerRangeSum(0, 10))
```

- Line 1 to 4
  - Line 1 : 1 iteration
  - Line 2, 3:
    (intTo-intFrom) iterations X 2 lines= N iterations X 2lines
    = 2N iterations
  - Line 4: 1 iteration
- Total # of iterations = 2N+2 iterations=**O(N)**

Bubble sort algorithm analysis

```
def performBubbleSort(lst):
    for itr1 in range(0, len(lst)):                 1
        for itr2 in range(itr1+1, len(lst)):        2
            if lst[itr1] < lst[itr2]:               3
                lst[itr1], lst[itr2] =\             4
                lst[itr2], lst[itr1]
    return lst                                      5
```

- Line 1 to 5
  - Line 1 : N iterations
  - Line 2, 3, 4 : *N-i* iterations (*i* is from *0* to *N-1*) X 3 lines
    - 1 to N, 2 to N, ….., N-1 to N
    - In other words, $(\sum_{i=0}^{N-1}\sum_{j=i+1}^{N-1}1)$ iterations X 3 lines
    - Assuming that "*if*" always results in true
  - Line 5: 1 iteration
- Total # of iterations = $\frac{3}{2}n^2 - \frac{3}{2}n + n + 1$ iterations=**O(N^2)**

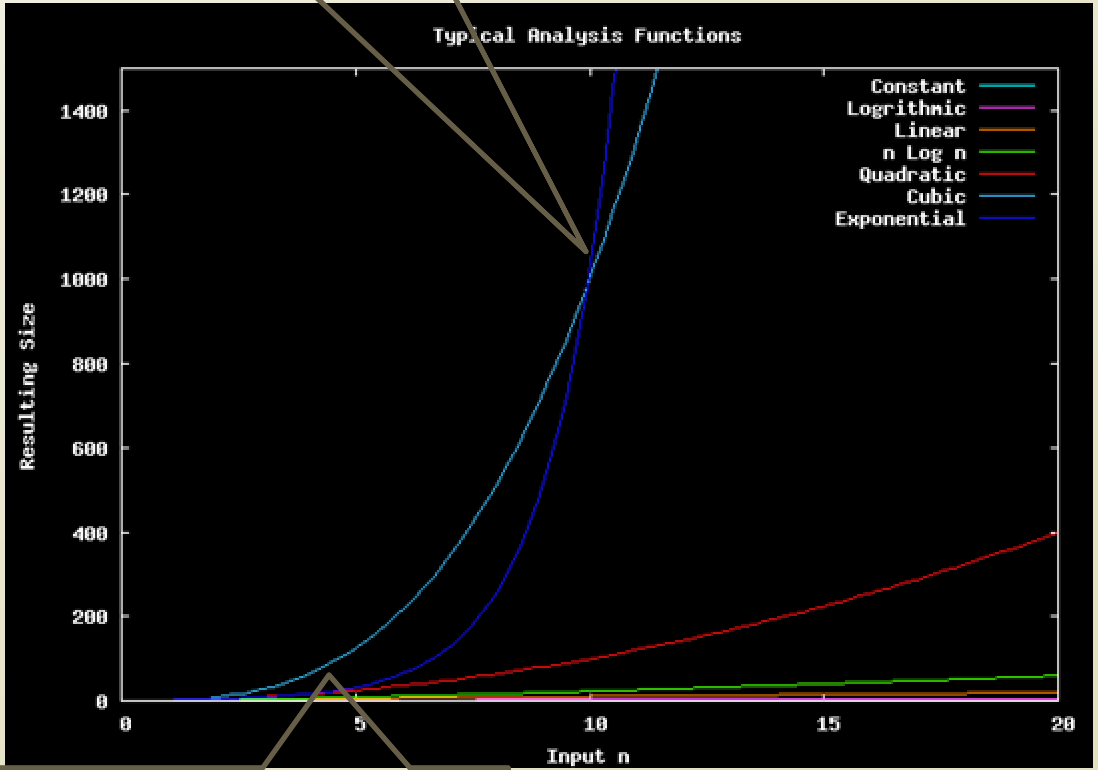
Asymptotic notation: Big-Oh

- What do O(N) and O(N^2) mean?
- That's the Big-Oh notation
  - Notation to show the **worst-case** running time
    - Do you remember?
      - Assuming that "*if*" always results in true
      - So, this is a worst scenario for the run-time
        - Because the program should run the statements in the "if" block
- Definition of the Big-Oh notations
  - $f(N) = O(g(N))$
  - There are positive constants c and n_0 such that
    - $f(N) \leqq c\, g(N)$ when $N \geqq n_0$
  - The growth rate of $f(N)$ is less than or equal to the growth rate of $g(N)$
  - $g(N)$ is an upper bound on $f(N)$

Growth rate

- Definition of the Big-Oh notations
  - $f(N) = O(g(N))$
  - There are positive constants c and n_0 such that
    - $f(N) \leqq c\, g(N)$ when $N \geqq n_0$
  - **The growth rate of *f(N)* is less than or equal to the growth rate of *g(N)***
  - $g(N)$ is an upper bound on $f(N)$

Exponential function (c^n) grows more than cubic function (n^3)



Quadratic function (n^2) grows more than linear function (n)

Examples of Big-Oh notation

- Assume $f(N) = 7N^2$. Then
  - $f(N) = O(N^4)$
  - $f(N) = O(N^3)$
  - $f(N) = O(N^2)$ (best answer, asymptotically tight)
- $N^2 / 2 - 3N$
  - $O(N^2)$
- $1 + 4N$
  - $O(N)$
- $7N^2 + 10N + 3$
  - $O(N^2)$
- $\log_{10} N = \log_2 N / \log_2 10$
  - $O(\log_2 N) = O(\log N)$
- $\sin N$
  - $O(1)$
- 10
  - $O(1)$
- 10^{10}
  - $O(1)$
- $\log N + N$
  - $O(N)$

Rules of Big-Oh notation

- When considering the growth rate of a function using Big-Oh
  - Ignore the lower order terms and the coefficients of the highest-order ter m
    - When we have N^3, then N^2 and N means nothing in terms of Big-Oh
    - From the growth rate order
      - $c^N > N^k > N^2 > N\log N > N > \log N > C$
      - $C >= 2$ and $k > 2$
  - No need to specify the base of logarithm
    - $O(\log N) = O(\log_C N)$
- If $T_1(N) = O(f(N))$ and $T_2(N) = O(g(N))$, then
  - $T_1(N) + T_2(N) = \max(O(f(N)),\ O(g(N)))$
    - $\max(O(N), O(N^2)) = O(N^2)$
  - $T_1(N) * T_2(N) = O(f(N) * g(N))$
    - $O(N) * O(\log N) = O(N\log N)$

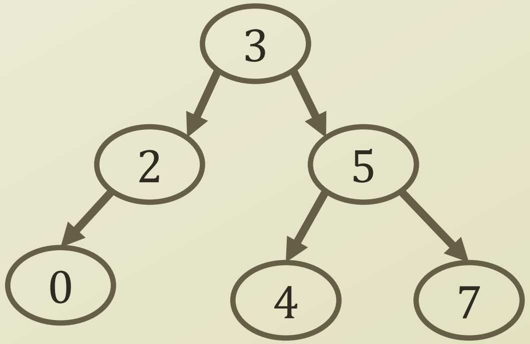
Big-Oh notation of list, stack and queue

	List	**Stack**	**Queue**
Pop	X	1 retrieval O(1)	X
Push	X	1 retrieval O(1)	X
Enqueue	X	X	1 retrieval O(1)
Dequeue	X	X	1 retrieval O(1)
Search	*i* retrieval (if the target instance at i^{th} in the list) O(N)	X (Does not allow search in the stack)	X (Does not allow search in the queue)

Detour: Performance of binary search tree

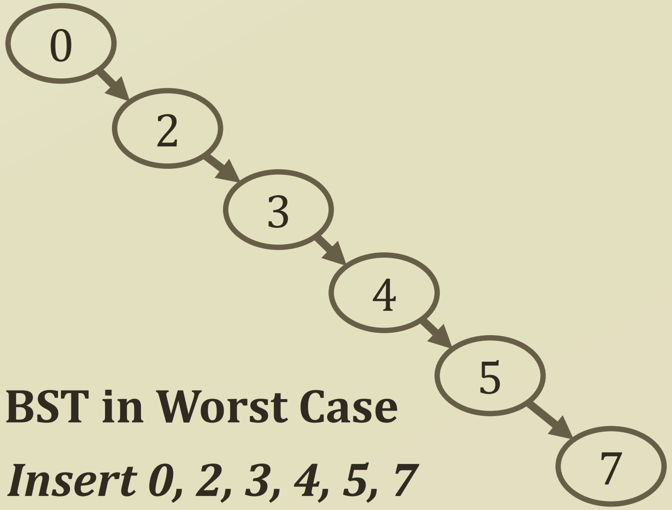
Coming from divide and conquer

	Linked List	**BST in Average**	**BST in Worst Case**
Search	O(n)	O(log n)	O(n)
Insert after search	O(1)	O(1)	O(1)
Delete after search	O(1)	O(1)	O(1)
Traverse	O(n)	O(n)	O(n)



BST in Average

Insert 3, 2, 0, 5, 4, 7



BST in Worst Case

Insert 0, 2, 3, 4, 5, 7

Further Readings

- Introductions to Algorithms by Cormen et al.
  - pp. 5-61