Chapter 1. (Part 2)

Algorithms:
Efficiency, Analysis, and Order

경북대학교 배준현 교수

(joonion@knu.ac.kr)

Contents

1.3 Analysis of Algorithms

- Two Types of **Algorithm Analysis**:
  - The ***correctness*** of an algorithm
    - develops a *proof* that the algorithm actually does
      - what it is supposed to do.
  - The ***efficiency*** of an algorithm
    - determines *how efficiently* the algorithm *solves* a problem
      - in terms of either *time* or *space*.

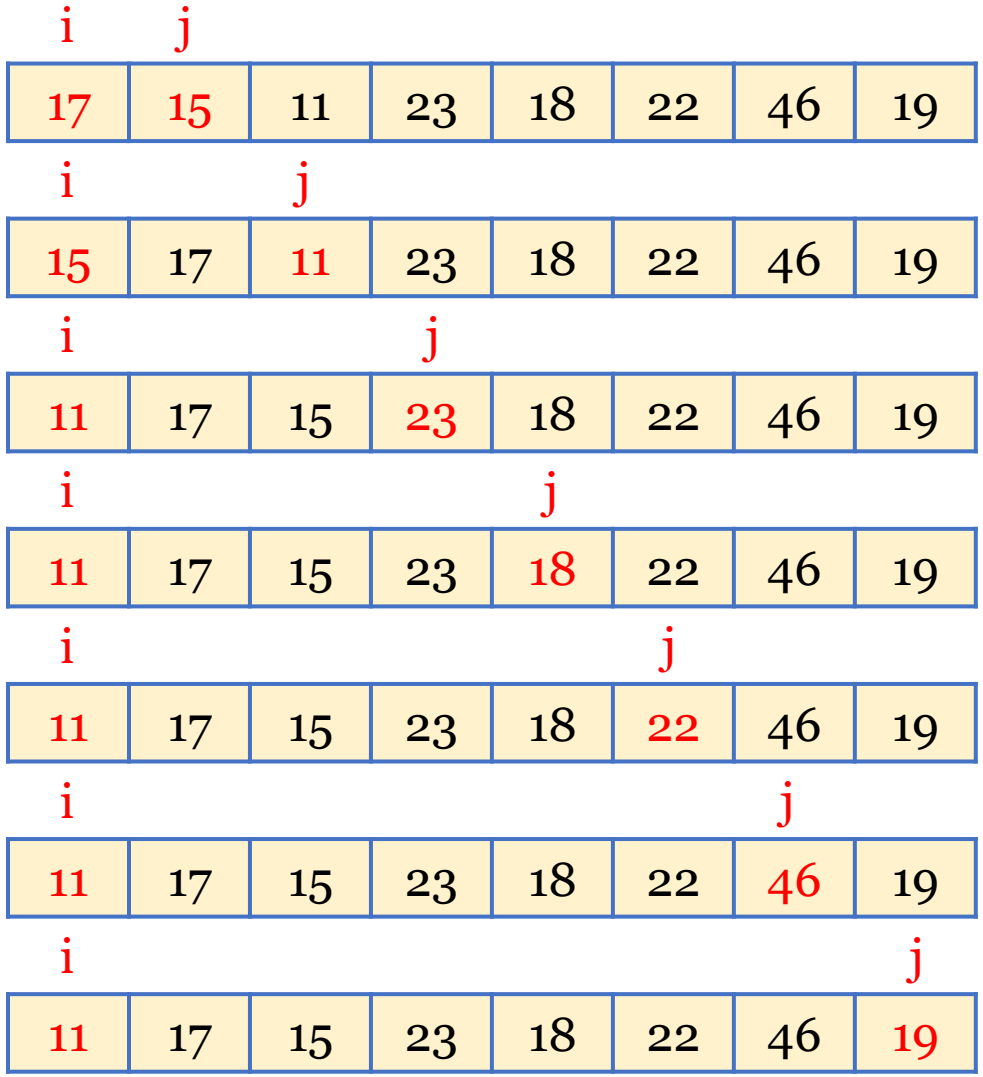
1.3 Analysis of Algorithms

- *Prove* the *correctness* of Exchange Sort
  - Note that an algorithm should *solve all instances* of a problem.
  - Hence, we should show that Algorithm 1.3
    - *always finds* a *correct solution*
    - for *all instances* of the problem.

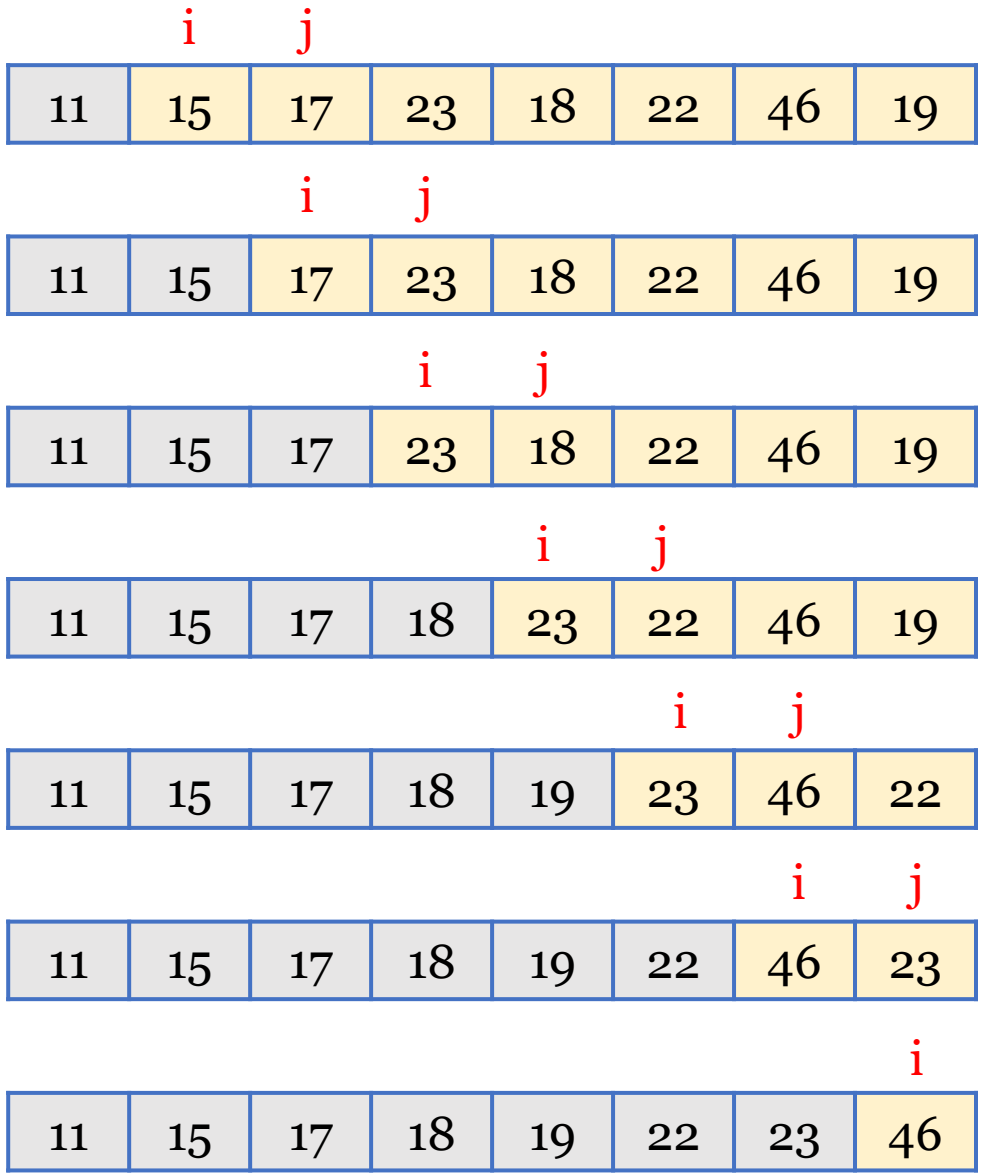
ALGORITHM 1.3: Exchange Sort

```
void exchangesort(int n, vector<int>& S)
{
    for (int i = 1; i <= n; i++)
        for (int j = i + 1; j <= n; j++)
            if (S[i] > S[j])
                swap(S[i], S[j]);
}
```

The steps in the *inner* for-loop



1.3 Analysis of Algorithms



The steps in the *outer* for-loop

- *Proof* for the *Correctness* of the Algorithm 1.3:
  - Let n be the length of A.
  - After the outer for-loop completes its iteration for $i = t$,
    - we have a list of integers A, such as $A[t] \leq A[k],\ t < k < n$.
  - On subsequent iterations, for $i > t$,
    - there is *no change* in the values from $A[0]$ through $A[t]$.
  - Hence, following the last iteration of the outer for-loop,
    - we have a list of integers A, such as $A[0] \leq A[1] \leq \cdots \leq A[n-1]$.

1.3 Analysis of Algorithms

- *Analyzing* the ***Efficiency*** of an Algorithm
  - Two important considerations:
    - *How fast* does an algorithm run *according to* ***the input size***?
    - *How much memories* does it require to run?
  - We need to determine the efficiency
    - *independent to* a *particular computer* on which the algorithm runs.
    - *independent to* a particular *programming language*.
    - *independent to* all the *complex details* of the algorithm.

1.3 Analysis of Algorithms

loop invariunt

- ***Complexity Analysis***:
  - a standard technique for analyzing the efficiency of an algorithm.
  - In general, analyze the efficiency of an algorithm
    - by determining the number of times some *basic operation* is done
      - as a function of the *input size*.

1.3 Analysis of Algorithms

- *Complexity Analysis* of the Algorithm 1.3 (Exchange Sort):
  - Finding a reasonable measure of the *input size*:
    - n: the number of elements in A.
  - Choosing a reasonable set of the *basic operation*:
    - *comparison*: a good candidate for the basic operation.
  - The total work done by the algorithm is
    - roughly *proportional to* the number of times
      - that the basic operation is done.
  - The number of comparisons with the input size n:
    - $(n-1)+(n-2)+\cdots+1=\frac{(n-1)n}{2}=\frac{1}{2}(n^2-n)$

1.3 Analysis of Algorithms

- ***Time Complexity** Analysis* is
  - the determination of *how many times the basic operation is done*
    - for each value of the *input size*.
  - Let $T(n)$ be the number of times for an input size n.
    - $T(n)$ is called the ***every-case** time complexity* of the algorithm.
  - However, in some cases,
    - $T(n)$ depends *not only* on the *input size*, *but also* on the *input's value*.

1.3 Analysis of Algorithms

- Three kinds of time complexities:
  - The ***best-case*** time complexity
    - the *minimum* number of times for an input size n.
  - The ***worst-case*** time complexity
    - the *maximum* number of times for an input size n.
  - The ***average-case*** time complexity
    - the *average* (*expected*) number of times for an input size n.

1.3 Analysis of Algorithms

- *Time Complexity* of the Algorithm 1.1 (Sequential Search):
  - Best-case: $B(n) = 1$.
  - Worst-case: $W(n) = n$.
  - Average-case time complexity:
    - for the case in which it is known that x is in A,

$$A(n) = \sum_{k=1}^{n} \left(k \times \frac{1}{n} \right) = \frac{1}{n} \times \sum_{k=1}^{n} k = \frac{1}{2}(n+1)$$

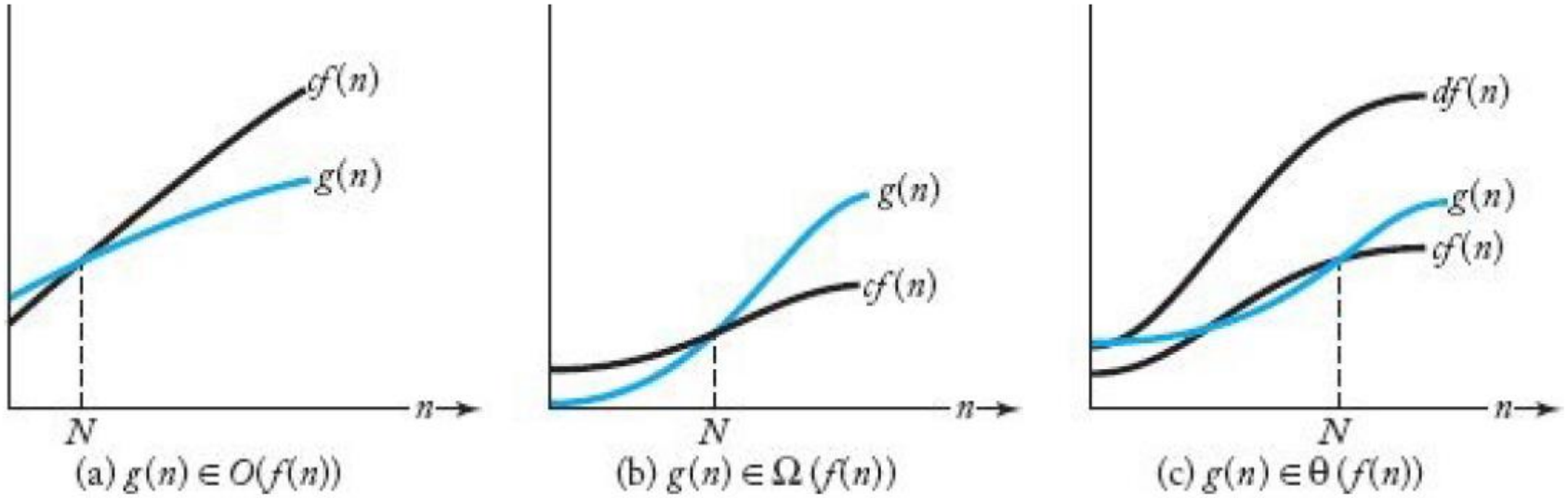
    - for the case in which x may *not* be in A.
      - refer to the textbook.

1.4 Order

- *Comparing* the Efficiency of Different Algorithms
  - Suppose that we have two algorithms for the same problem:
    - Algorithm A.1 with the time complexity $T(n) = n$.
    - Algorithm A.2 with the time complexity $T(n) = n^2$.
    - Which one is *faster* than the other?
  - Now, two algorithms are:
    - Algorithm A.3 with $T(n) = 100n$.
    - Algorithm A.4 with $T(n) = 0.01n^2$.
    - Then, which one is *faster* than the other, *eventually*?

1.4 Order

- ***Asymptotic Notations***: O, Ω, Θ
  - O(*Big O*) puts an *asymptotic upper bound* on a function.
  - Ω(*Omega*) puts an *asymptotic lower bound* on a function.
  - Θ(*Theta*) puts the ***order*** on a function.



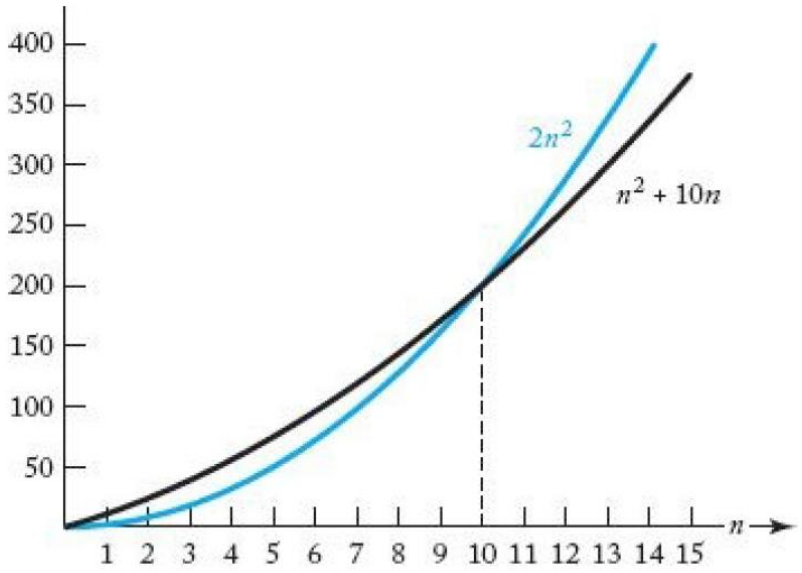
1.4 Order

- Definition of O(Big O)

For a given complexity function $f(n)$, $O(f(n))$ is the set of complexity function $g(n)$ for which there exists some positive real constant c and some nonnegative integer N such that for all $n \geq N$,

$$g(n) \leq c \times f(n).$$

- Although $g(n)$ starts out above $cf(n)$,
  - *eventually* it falls *beneath* $cf(n)$.
- For example,
  - $n^2 + 10n$ is initially above $2n^2$,
  - however, it goes below, for $n \geq 10$.



1.4 Order

- Definition of Ω(Omega)

For a given complexity function $f(n)$, $\Omega(f(n))$ is the set of complexity function $g(n)$ for which there exists some positive real constant c and some nonnegative integer N such that for all $n \geq N$,

$$g(n) \geq c \times f(n).$$

- Do we need to prove the *lower bound* of any algorithm?
- Consider *various* sorting algorithms based on comparison.
  - Is it *possible* to design a better algorithm than *quicksort*?

1.4 Order

- Definition of Θ(Theta)

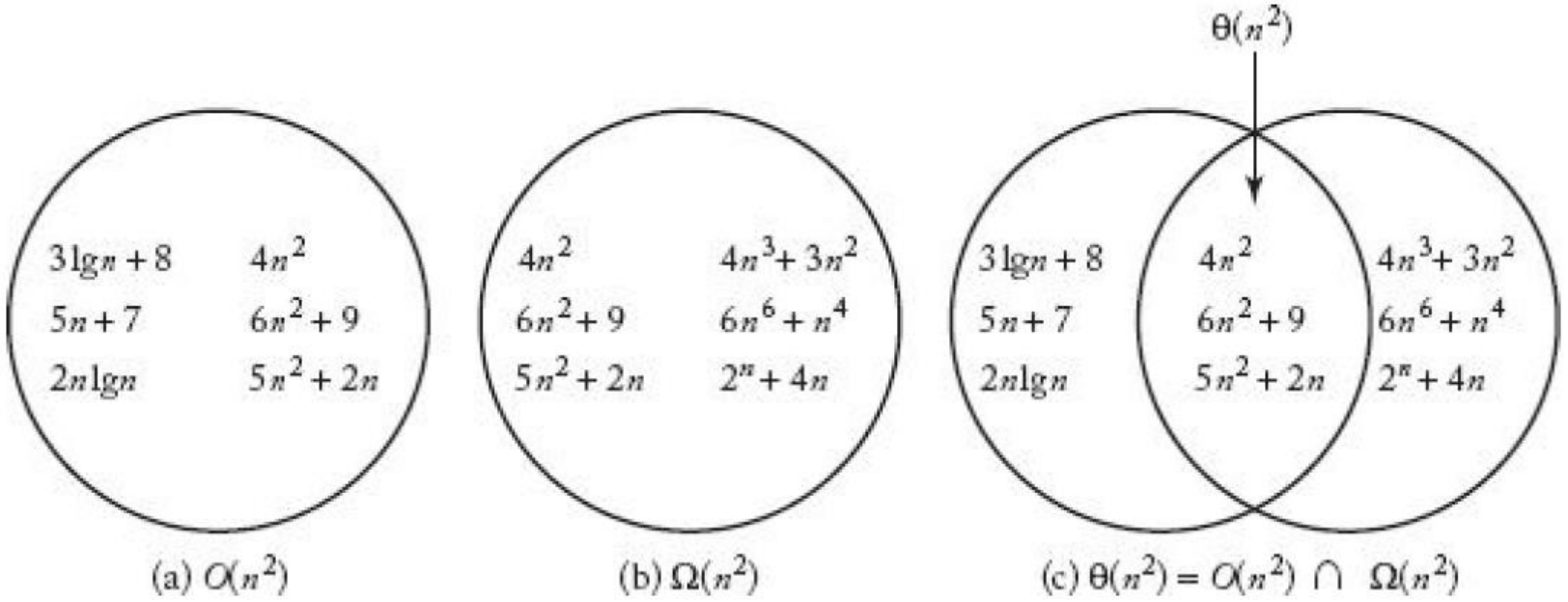
For a given complexity function $f(n)$,

$$\Theta(f(n)) = O(f(n)) \cap \Omega(f(n)).$$

- If $g(n) \in \Theta(f(n))$,
  - we say that $g(n)$ is the ***order*** of $f(n)$.

1.4 Order

- Some *exemplary* members of $O(n^2)$, $\Omega(n^2)$, $\Theta(n^2)$.
  - Note that any logarithmic or linear complexity function is in $O(n^2)$.



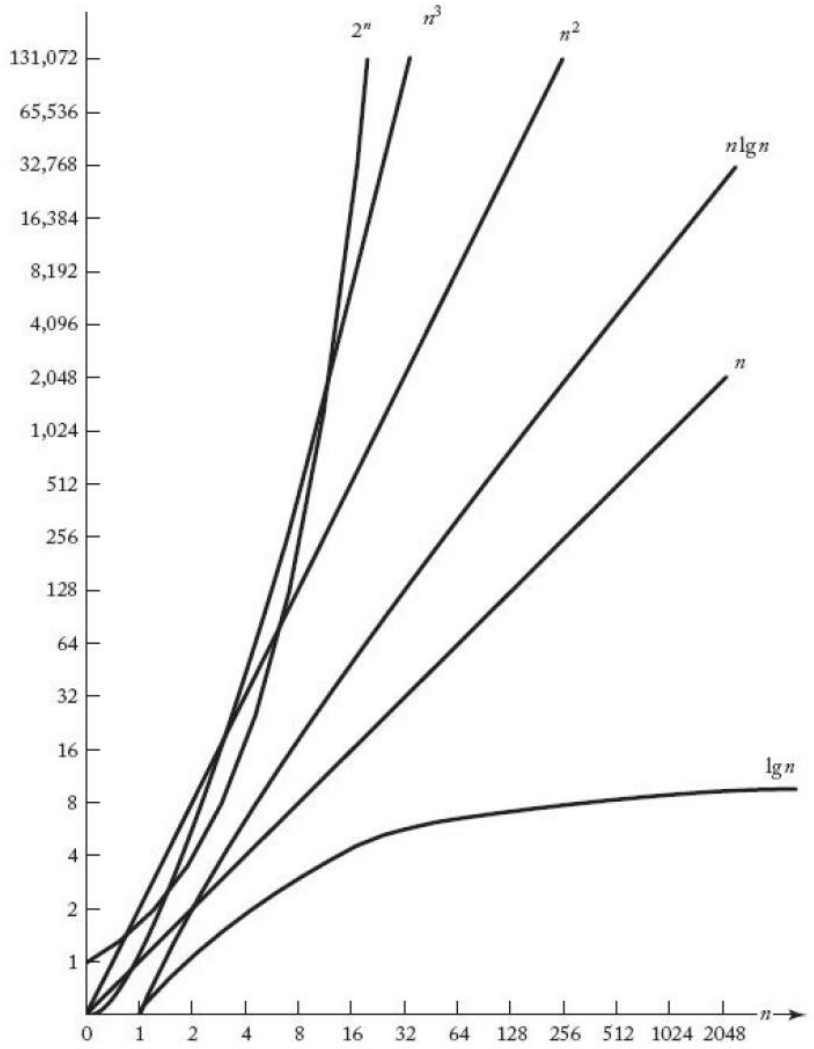
(a) $O(n^2)$ (b) $\Omega(n^2)$ (c) $\theta(n^2) = O(n^2) \cap \Omega(n^2)$

1.4 Order

- *Categorical Classification* of Complexities:
  - A set of algorithms can be *classified*
    - by the *order* of *complexity functions*, such as $\Theta(n)$, $\Theta(n^2)$, and $\Theta(2^n)$.
  - The most common complexity categories are:
    - $\Theta(1)$: ***constant*** time complexity
    - $\Theta(\lg n)$: ***logarithmic*** time complexity
    - $\Theta(n)$: ***linear*** time complexity
    - $\Theta(n \lg n)$: *linear logarithmic* time complexity
    - $\Theta(n^2)$: *quadratic* time complexity
    - $\Theta(n^3)$: *cubic* time complexity
    - $\Theta(2^n)$: ***exponential*** time complexity
    - $\Theta(n!)$: *factorial* time complexity

1.4 Order

- Growth rates of some common complexity functions



- ***Polynomial**-time* complexity:
  - regarded as an *efficient* algorithm.
- ***Exponential**-time* complexity:
  - regarded as an *inefficient* algorithm.

1.4 Order

- Examples of time complexity analysis:
  - Algorithm 1.1 (Sequential Search)
    - In average-case: $T(n) = \frac{1}{2}(n+1) \in \Theta(n) \in O(n)$.
  - Algorithm 1.3 (Exchange Sort)
    - $T(n) = \frac{1}{2}(n^2 - n) \in \Theta(n^2) \in O(n^2)$.
  - Algorithm 1.5 (Binary Search)
    - In worst case: $T(n) = \lg n + 1 \in \Theta(\lg n) \in O(\lg n)$.
  - Algorithm 1.6 (*Recursive* nth Fibonacci Term)
    - $T(n) = T(n-1) + T(n-2) + 1$

      $> 2^{(n-1)/2} + 2^{(n-1)/2} + 1 = 2^{n/2} \in O(2^n)$.

Exercises

1. Write an algorithm that finds the largest number in a list (an array) of n numbers.

Exercises

6. Write an algorithm that finds both the smallest and largest numbers in a list of n numbers.

Exercises

- Analyze the time complexity $T(n)$ of the following algorithm.

```
void algorithm1(int n) {
    for (int i = 1; i <= 2*n; i++)
        cout << "Basic Operation";
    for (int i = n/2; i >= 1; i--)
        cout << "Basic Operation";
}
```

Exercises

- Analyze the time complexity $T(n)$ of the following algorithm.

```
void algorithm2(int n) {
    for (int i = 1; i <= 2*n; i++)
        for (int j = n/2; j >= 1; j--)
            cout << "Basic Operation";
}
```

Exercises

- Analyze the time complexity $T(n)$ of the following algorithm.

```
void algorithm3(int n) {
    for (int i = 1; i <= n; i *= 2)
        for (int j = n; j >= 1; j /= 2)
            cout << "Basic Operation";
}
```

Exercises

- Analyze the time complexity $T(n)$ of the following algorithm.

```
void algorithm4(int n, int m) {
    for (int i = 1; i <= 2*n; i++)
        cout << "Basic Operation";
    for (int i = 1; i <= m/2; i++)
        cout << "Basic Operation";
}
```

Exercises

- Analyze the time complexity $T(n)$ of the following algorithm.

```
void algorithm5(int n) {
    algorithm1(n);
    for (int i = 1; i <= n; i++) {
        algorithm2(n);
        for (int j = 1; j <= n; j++)
            algorithm3(n);
    }
}
```

X.1: Recursion

- *Recursion*: Basis for Algorithmic Thinking
  - 종료조건: 재귀 호출을 종료할 수 있는 시점에서 해답을 리턴
  - 재귀호출: 문제해결을 위해 부분문제에 대해 자기 자신을 재귀적으로 호출

```
int function(int n) {
    if (n == 0)
        return 1
    else
        return n * function(n - 1)
}
```

exit condition

recursive call

X.1: Recursion

- **문제: 주어진 자연수 n의 팩토리얼 $n!$을 구하라.**
  - **종료조건:** $0! = 1$
  - **재귀호출:** $n! = n \times (n-1)!$

```
def factorial(n):
    if n == 0:
        return 1
    else:
        return n * factorial(n - 1)
```

X.1: Recursion

- **문제: 두 자연수 n, m $(n \geq m)$의 최대공약수 $\gcd(n, m)$을 구하라.**
  - **배경지식: 유클리드의 정리**
    - $n > 0, m > 0$**이면** $\gcd(n, m) = \gcd(m, n\%m)$**이 성립**
  - **종료조건:** $\gcd(n, m) = n$, if $m = 0$
  - **재귀호출:** $\gcd(n, m) = \gcd(n, n\%m)$, if $m > 0$

```
def gcd(n, m):
    if m == 0:
        return n
    else:
        return gcd(m, n % m)
```

X.1: Recursion

- 문제: 자연수 n에 대해 콜라츠 수열을 출력하시오.
  - 배경지식: 콜라츠 추측(Collatz Conjecture)
    - 다음 함수 $T(n)$을 반복해서 적용하면 결국 1에 도달할 것이다.
    - $T(n) = n/2, if\ n\ is$ even.
    - $T(n) = 3n + 1, if\ n\ is$ odd.
  - 종료조건: $exit, if\ n = 1$
  - 재귀호출:
    - $collatz(n) = \text{collatz}(n/2)\text{, if } n\%2 = 0$
    - $collatz(n) = \text{collatz}(3n + 1)\text{, if } n\%2 = 1$

```
def collatz(n):
    print(n, end=' ')
    if n == 1:
        return
    else:
        if n % 2 == 0:
            collatz(n // 2)
        else:
            collatz(3 * n + 1)
```

X.1: Recursion

- 문제: 하노이의 탑에 n개의 원반이 있을 때 원반의 이동 순서를 출력하라.
  - 입력조건: 원반의 개수 n,
  - 현재 기둥 src, 목적지 기둥 dst, 경유할 기둥 via
  - 종료조건: $n = 1$일 때 src에서 dst 로 이동하면 끝.
  - 재귀호출:
    - $n - 1$개의 기둥을 src에서 via로 이동
    - 남은 하나의 기둥을 src에서 dst로 이동
    - $n - 1$개의 기둥을 via에서 dst로 이동

```
def hanoi(n, src, via, dst):
    if n == 1:
        print(f"{src} -> {dst}")
    else:
        hanoi(n - 1, src, dst, via)
        hanoi(1, src, via, dst)
        hanoi(n - 1, via, src, dst)
```

Exercises

- Analyze the time complexity $T(n)$ of the following algorithm.

```
void algorithm6(int n) {
    if (n <= 1) {
        cout << "Basic Operation";
    }
    else {
        algorithm6(n/2);
        algorithm6(n/2);
        cout << "Basic Operation";
    }
}
```

Exercises

- Analyze the time complexity $T(n)$ of the following algorithm.

```
void algorithm7(int n) {
    if (n <= 1) {
        cout << "Basic Operation";
    }
    else {
        algorithm7(n/4);
        algorithm7(n/4);
        algorithm7(n/4);
        cout << "Basic Operation";
    }
}
```

Any Questions?