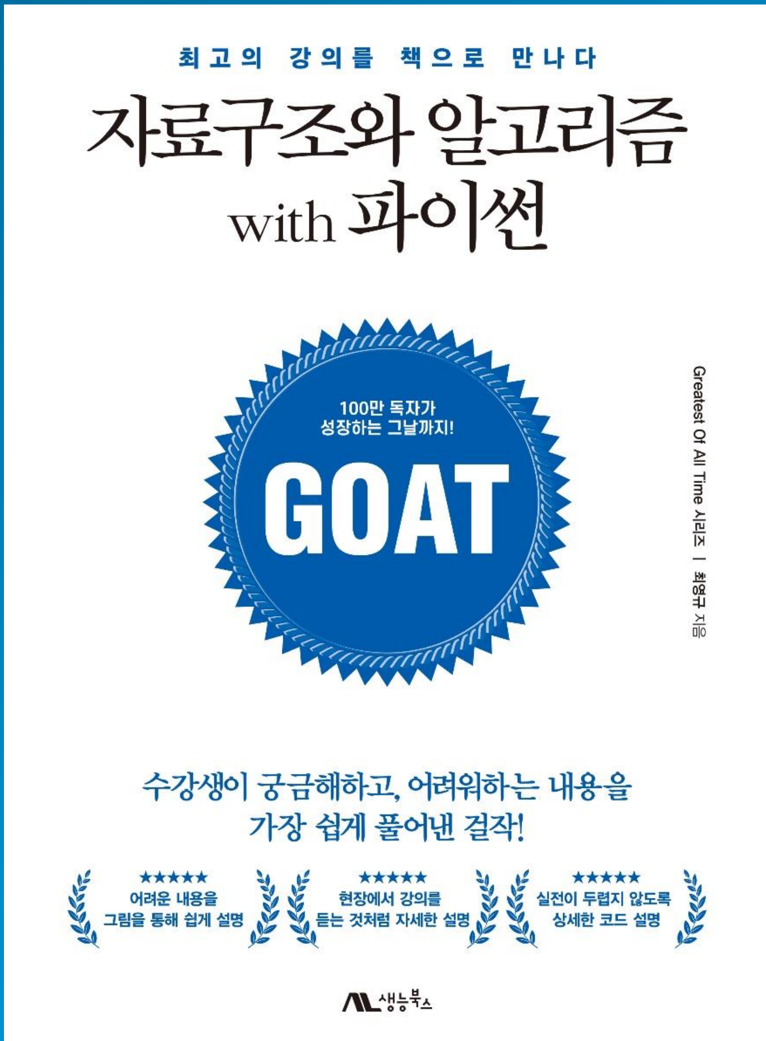
SW알고리즘개발
7주차



4장. 트리

04-1 트리란?

04-2 이진 트리

04-3 이진 트리의 연산

04-4 모스 코드 결정 트리

04-5 수식 트리

4.4 모스 코드 결정 트리

- 새뮤얼 모스(Samuel Morse)
  - SOS → ••• —— •••

문자	부호	문자	부호	문자	부호
A	• –	J	• – – –	S	• • •
B	– • • •	K	– • –	T	–
C	– • – •	L	• – • •	U	• • –
D	– • •	M	– –	V	• • • –
E	•	N	– •	W	• – –
F	• • – •	O	– – –	X	– • • –
G	– – •	P	• – – •	Y	– • – –
H	• • • •	Q	– – • –	Z	– – • •
I	• •	R	• – •		

ASCII 코드 표 (알파벳 ASCII Code Table)

10진수	부호	10진수	부호
065	A	097	a
066	B	098	b
067	C	099	c
068	D	100	d
069	E	101	e
070	F	102	f
071	G	103	g
072	H	104	h
073	I	105	i
074	J	106	j
075	K	107	k
076	L	108	l
077	M	109	m
078	N	110	n
079	O	111	o
080	P	112	p
081	Q	113	q
082	R	114	r
083	S	115	s
084	T	116	t
085	U	117	u
086	V	118	v
087	W	119	w
088	X	120	x
089	Y	121	y
090	Z	122	z

문자를 모스 코드로 변환하는 과정: 인코딩

- 문자에 대응되는 코드를 표에서 찾아 순서대로 출력
  - 코드 표:

```
table =[('A', '.-'),    ('B', '-...'), ('C', '-.-.'), ('D', '-..'),
        ('E', '.'),     ('F', '..-.'), ('G', '--.'),  ('H', '....'),
        ('I', '..'),    ('J', '.---'), ('K', '-.-'),  ('L', '.-..'),
        ('M', '--'),    ('N', '-.'),   ('O', '---'),  ('P', '.--.'),
        ('Q', '--.-'),  ('R', '.-.'),  ('S', '...'),  ('T', '-'),
        ('U', '..-'),   ('V', '...-'), ('W', '.--'),  ('X', '-..-'),
        ('Y', '-.--'),  ('Z', '--..') ]
```

- 인코딩 함수

```
def encode(ch):
    idx = ord(ch)-ord('A')   # 리스트에서 해당 문자의 인덱스
    return table[idx][1]     # 해당 문자의 모스 부호 반환
```

모스 코드를 문자로 변환하는 과정: 디코딩

- 모스 코드가 주어졌을 때 해당하는 알파벳을 추출
  - 표의 모든 항목을 하나씩 검사해야 함 → 비효율적

- 비효율적인 디코딩 함수

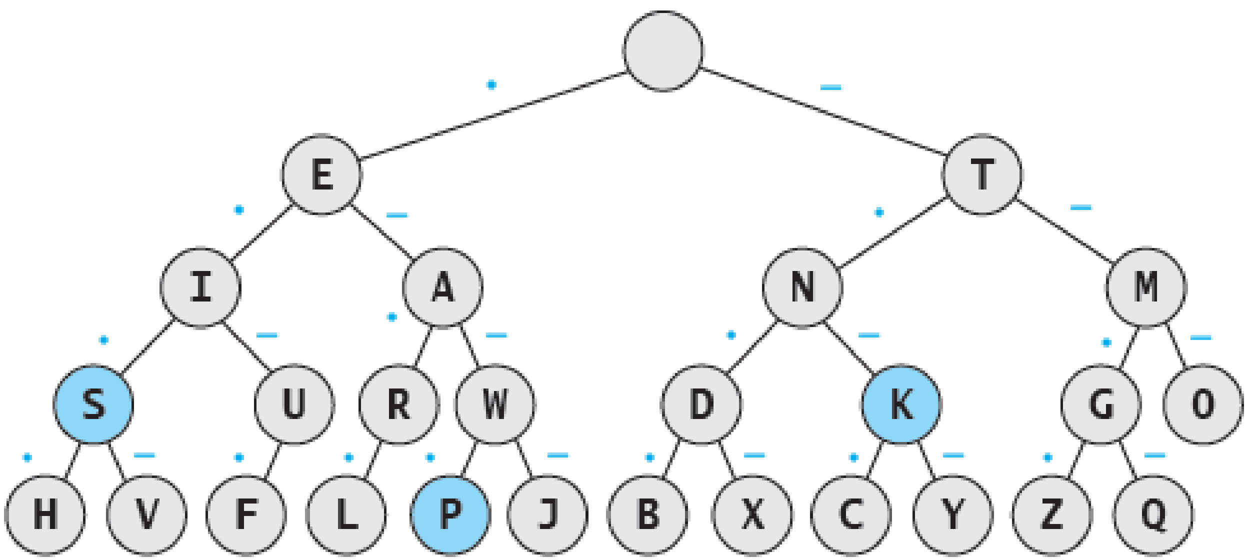
이 코드에서 효율성이나 시간 관련 문제의 답은 n

```
def decode_simple(morse):
    for tp in table :                  # 모스 코드 표의 모든 문자에 대해
        if morse == tp[1] :            # 찾는 코드와 같으면
            return tp[0]               # 그 코드의 문자를 반환
```

- 개선 방안? 결정 트리를 이용한 디코딩

결정 트리를 이용한 모스 코드의 디코딩

- 결정 트리(decision tree)
  - 여러 단계의 복잡한 조건을 갖는 문제에 대해 조건과 그에 따른 해결방법을 트리 형태로 나타낸 것
- 모스 코드를 위한 결정 트리
  - 최대 트리의 높이 만큼만 비교 → 디코딩이 효율적



예:

• • • : S
– • – : K
• – – • : P
– – • • • : 코드 없음

결정 트리를 이용한 모스 코드의 디코딩

- 모스 코드를 위한 결정 트리 만들기
  ① 빈 루트 노드를 만들고 모스 코드표의 각 문자를 하나씩 트리에 추가
  ② 문자를 추가할 때 루트부터 시작하여 트리를 타고 내려감. 만약 타고 내려갈 자식 노드가 None이면 새로운 노드를 추가하는데, 노드만 추가할 뿐이지 그 노드의 문자는 아직 결정할 수 없음
  - 각 모스 코드에 대해, ·이면 왼쪽으로, -이면 오른쪽으로 트리의 노드를 따라 감
  ③ 마지막 코드의 노드에 도달하면 그 노드에 문자를 할당
- 결정 트리를 이용한 디코딩

```
def decode(root, code):
    node = root                             # 루트 노드에서 시작
    for c in code :                         # 각 부호에 대해
        if c == '·' : node = node.left      # 점(·): 왼쪽으로 이동
        elif c=='-' : node = node.right     # 선(-): 오른쪽으로 이동
    return node.data                        # 문자 반환
```

모스 코드 결정 트리

```
def make_morse_tree():
    root = TNode( None, None, None )
    for tp in table :
        code = tp[1]                       # 모스 코드
        node = root
        for c in code :                    # 맨 마지막 문자 이전까지 --> 이동
            if c == '.' :                  # 왼쪽으로 이동
                if node.left == None :     # 비었으면 빈 노드 만들기
                    node.left = TNode (None, None, None)
                node = node.left           # 그쪽으로 이동
            elif c == '-' :                # 오른쪽으로 이동
                if node.right == None :    # 비었으면 빈 노드 만들기
                    node.right = TNode (None, None, None)
                node = node.right        # 그쪽으로 이동

        node.data = tp[0]                  # 코드의 알파벳
    return root
```

테스트 프로그램

```
morseCodeTree = make_morse_tree()
str = input("입력 문장 : ")
mlist = []
for ch in str:
    code = encode(ch)
    mlist.append(code)
print("Morse Code: ", mlist)
print("Decoding : ", end='')
for code in mlist:
    ch = decode(morseCodeTree, code)
    print(ch, end='')
print()
```

모스코드 결정트리를 만듦.
morseCodeTree가 루트 노드

입력 문자열의 각 문자를 순서 대로
모스코드로 변환하여 리스트에 추가

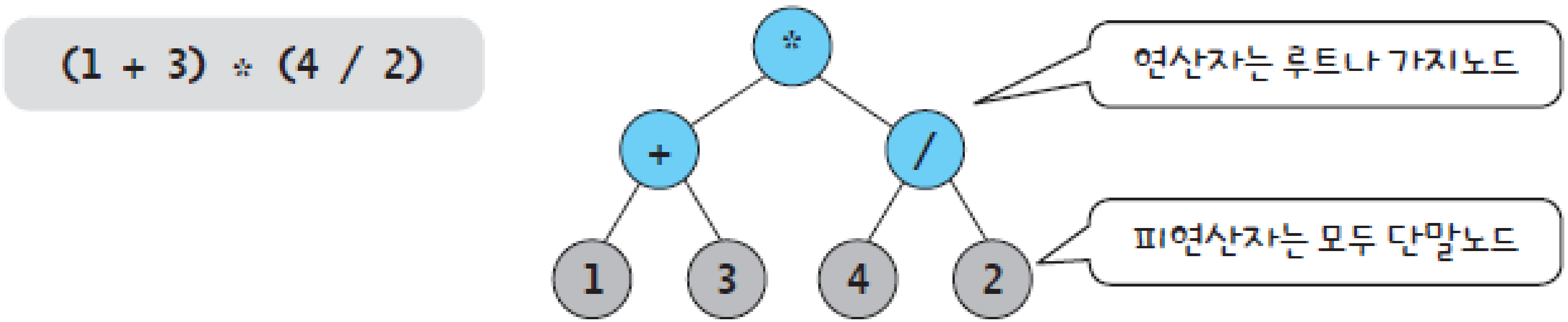
리스트의 모스 코드를 순서 대로 디코딩한
문자를 화면에 출력

실행 결과

```
입력 문장 : GAMEOVER
                   A      M      E     O       V       E     R
Morse Code: G ['--.', '.-', '--', '.', '---', '...-', '.', '.-.']
Decoding  : GAMEOVER
```

4.5 수식 트리

- 수식 트리(Expression Tree)
  - 산술식을 트리 형태로 표현한 이진 트리

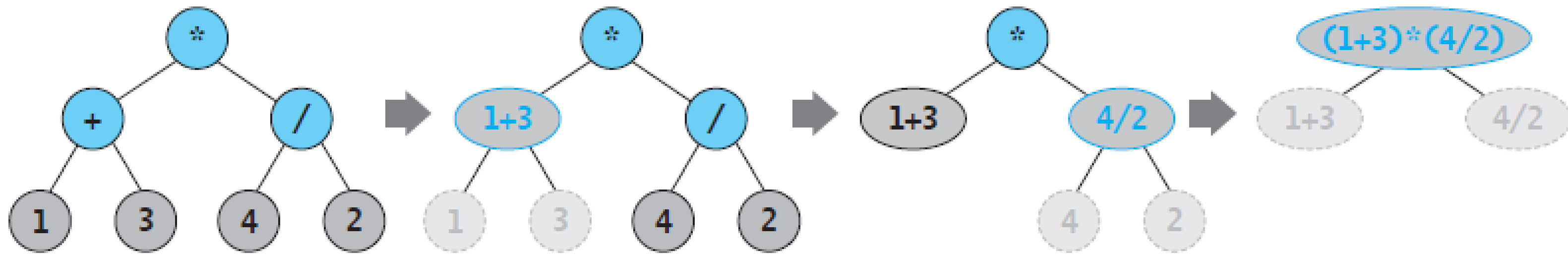


- 수식 트리 만들기
- 수식 트리의 계산

수식 트리의 계산

(1 + 3) * (4 / 2)

- 후위 순회 사용



수식 트리 (a)왼쪽 서브 트리 계산 (b)오른쪽 서브 트리 계산 (c)루트 노드 계산

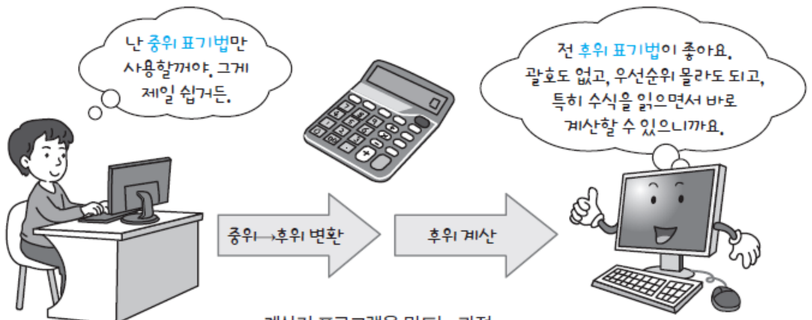
```
def evaluate(node) :
    if node is None :                    # 공백 트리이면 0 반환
        return 0
    elif node.isLeaf() :                 # 단말 노드이면 -> 피연산자
        return node.data                 # 그 노드의 값(데이터) 반환
    else :                               # 루트나 가지노드라면 -> 연산자
        op1 = evaluate(node.left)
        op2 = evaluate(node.right)
        if node.data == '+' : return op1 + op2
        elif node.data == '-' : return op1 - op2
        elif node.data == '*' : return op1 * op2
        elif node.data == '/' : return op1 / op2
```

왼쪽과 오른쪽 서브트리를 먼저 계산해야 루트를 계산할 수 있음.

루트(현재 노드)를 처리, 후위순회

수식의 표현 방법

전위(prefix)	중위(infix)	후위(postfix)
연산자 피연산자1 피연산자2	피연산자1 **연산자** 피연산자2	피연산자1 피연산자2 **연산자**
+ A B	A + B	A B +
+ 5 * A B	5 + A * B	5 A B * +

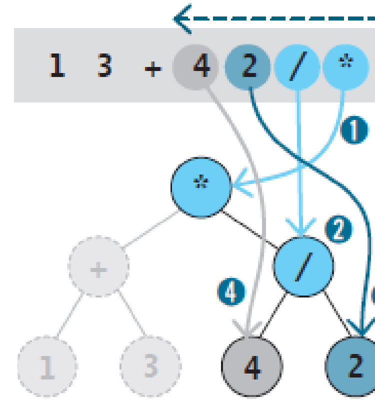


계산기 프로그램을 만드는 과정

- 후위 표기의 장점
  - 괄호를 사용하지 않아도 계산 순서를 알 수 있다.
  - 수식을 읽으면서 바로 계산 – 중위표기는 괄호와 연산자의 우선순위때문에 식을 끝까지 읽은 다음에야 계산할 수 있음
  - 연산자의 우선순위를 생각할 필요가 없음. – 식 자체에 우선 순위가 이미 포함

후위 표기 식으로 수식 트리 만들기

- 맨 뒤에서 앞으로 읽으면서 노드을 생성하고, 재귀적으로 자식 노드를 채워 감
- 오른쪽 자식이 먼저 왼쪽이 그 다음에 처리
- 입력 수식 : 후위 표기 수식 (예:**1 3 + 4 2 / ***)
- 루트 노드 반환



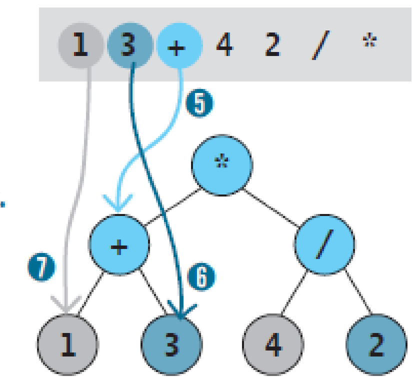

```
def buildETree(expr):
    if len(expr) == 0 :
        return None

    token = expr.pop()
    if token in "+-*/" :
        node = BTNode(token)
        node.right= buildETree(expr)
        node.left = buildETree(expr)
        return node
    else :
        return BTNode(float(token))
```

후위표기 수식을 expr로 전달. 예를 들어, 그림 4.25의 수식 트리는 ['1','3','+','4','2','/','*']와 같이 전달됨.

후위순회는 수식을 뒤에서 앞으로 처리. 따라서 pop()으로 맨 뒤의 요소를 꺼냄.

연산자이면 노드를 만들고, 오른쪽과 왼쪽순으로 서브트리를 순환호출을 이용해 만듦. 마지막으로 노드 반환.

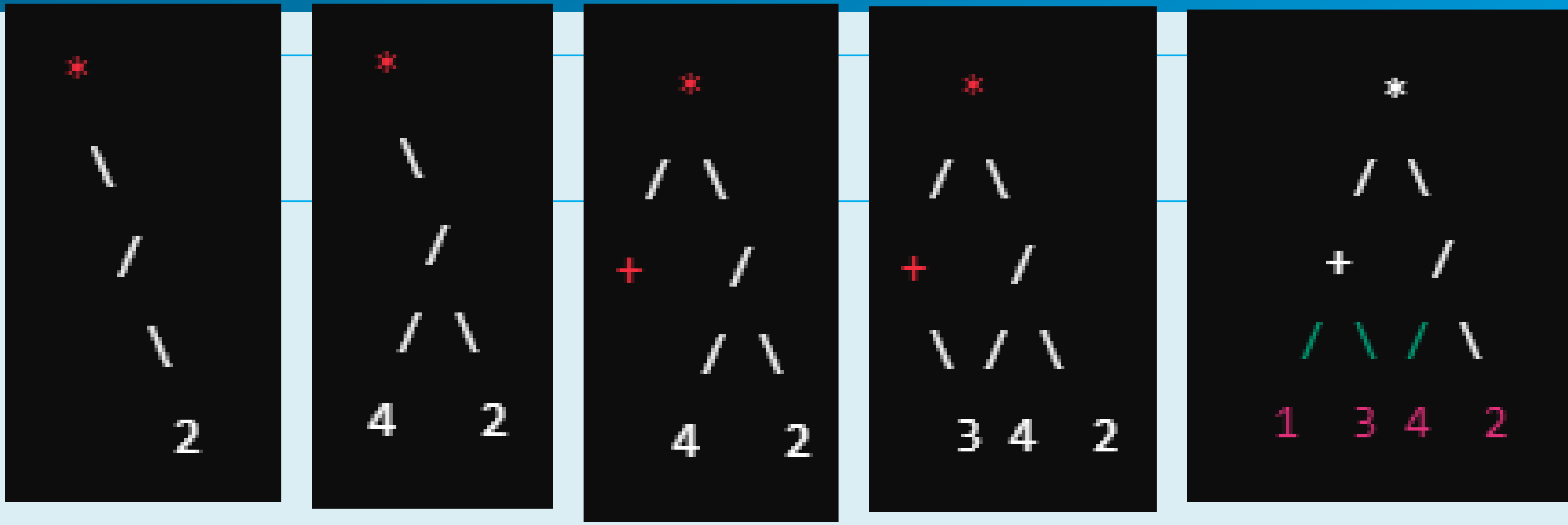
피연산자이면 단말노드이므로 노드를 만들어 바로 반환



예: 후위 표기 식으로 수식 트리 만들기

- 후위표기 수식: 1 3 + 4 2 / *

1. 첫 번째 연산: *. *는 두 개의 피연산자를 필요. 이 연산자는 트리의 **루트 노드**가 되며, 오른쪽과 왼쪽 자식을 차례로 채워가게 됨
2. 두 번째 연산 : /. *의 오른쪽 자식 노드를 찾기 위해 왼쪽으로 이동하여 / 처리. / 는 두 개의 피연산자가 필요
3. 세 번째 연산: 2 . /의 오른쪽 피연산자는 2. 이 값은 리프노드로 삽입
4. 네 번째 연산: 4 . /의 왼쪽 피연산자는 4. 이 값은 리프노드로 삽입. 즉 /의 왼쪽 자식 트리 완성
5. 다섯 번째 연산: + . * 연산자의 왼쪽 피연산자를 찾기 위해 + 연산를 처리. + 연산자는 두 개의 피연산자을 필요로하며, 트리의 왼쪽 서브트리의 루트가 됨
6. 여섯 번째 연산: 3. + 연산자의 오른쪽 피연산자는 3. 리프노드로 삽입
7. 일곱 번째 연산: 1. + 연산자의 왼쪽 피연산자는 1. 리프노드로 삽입



수식 트리 순회:

- 전위 순회 (Preorder): `* + 1 3 / 4 2`
- 중위 순회 (Inorder): `1 + 3 * 4 / 2`
- 후위 순회 (Postorder): `1 3 + 4 2 / *`

중위표기 수식을 후위표기 수직으로 변환

- 중위표기식을 **왼쪽에서 오른쪽으로 읽으면서** 연산자와 피연산자를 적절하게 스택과 출력 리스트로 옮겨 후위표기식을 생성

1. **연산자 우선순위 정의**
   - +, - : 우선 순위 1
   - *, / : 우선 순위 2
   - 괄호 (와) : 우선 순위 3
2. **알고리즘**
   - **출력 리스트** (후위표기식을 저장)
   - **스택** (연산자와 괄호를 임시로 저장)
   - 단계 1: 수식을 왼쪽에서 오른쪽으로 차례대로 한 토큰씩 읽기
   - 단계 2: 각 토큰에 대해 처리
     - **숫자**는 후위표기식에서 바로 사용되므로 **출력 리스트에 추가**
     - **연산자**가 스택에 있는 연산자보다 우선순위가 낮거나 같으면, 스택에 있는 연산자를 출력 리스트로 옮김. 그러고 나서 **현재 연산자를 스택에 추가**
     - **여는 괄호 (인 경우 스택에 추가**
     - **닫는 괄호)인 경우** 스택에서 여는 괄호가 나올 때까지 연산자를 출력 리스트로 옮김
   - 단계 3: 수식이 끝나면, **스택에 남아있는 모든 연산자를 출력 리스트로 옮김**

예시

예시: `3 + 5 * ( 2 - 8 )`

단계	읽은 토큰	출력 리스트	스택	설명
1	`3`	`3`		숫자이므로 출력 리스트에 추가
2	`+`	`3`	`+`	연산자이므로 스택에 추가
3	`5`	`3 5`	`+`	숫자이므로 출력 리스트에 추가
4	`*`	`3 5`	`+ *`	연산자이므로 스택에 추가 (우선순위가 높음)
5	`(`	`3 5`	`+ * (`	여는 괄호이므로 스택에 추가
6	`2`	`3 5 2`	`+ * (`	숫자이므로 출력 리스트에 추가
7	`-`	`3 5 2`	`+ * ( -`	연산자이므로 스택에 추가
8	`8`	`3 5 2 8`	`+ * ( -`	숫자이므로 출력 리스트에 추가
9	`)`	`3 5 2 8 -`	`+ *`	닫는 괄호이므로 괄호 안의 연산자를 출력
10	끝	`3 5 2 8 - * +`		남은 연산자를 모두 출력 리스트로 옮김

결과 (후위표기식): `3 5 2 8 - * +`

테스트 프로그램

```
str = input("입력(후위표기): ")      # 후위표기식 입력
expr = str.split()                   # 토큰 리스트로 변환
print("토큰분리(expr): ", expr)
root = buildETree(expr)
print('\n 전위순회: ', end=''); preorder(root)
print('\n 중위순회: ', end=''); inorder(root)
print('\n후위순회: ', end=''); postorder(root)
print('\n계산 결과 : ', evaluate(root))     # 수식 트리 계산
```

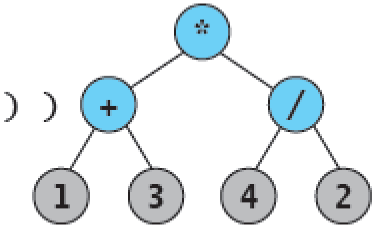
← 후위 표기식을 수식트리로 만들고 루트를 반환

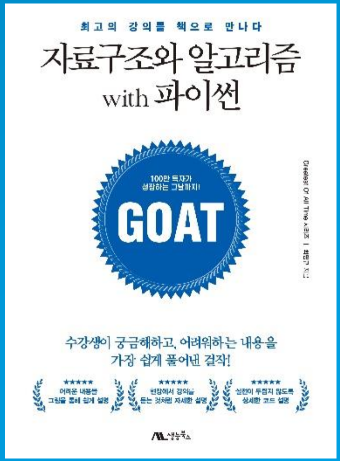
실행 결과

토큰 분리 결과(공백으로 분리)

```
입력(후위표기): 1 3 + 4 2 / *
토큰분리(expr):  ['1', '3', '+', '4', '2', '/', '*']

전위 순회: ( * ( + ( 1.0 ) ( 3.0 ) ) ( / ( 4.0 ) ( 2.0 ) ) )
중위 순회: 1.0 + 3.0 * 4.0 / 2.0
후위 순회: 1.0 3.0 + 4.0 2.0 / *
계산 결과 :  8.0
```

계산 결과





05 CHAPTER

알고리즘 개요

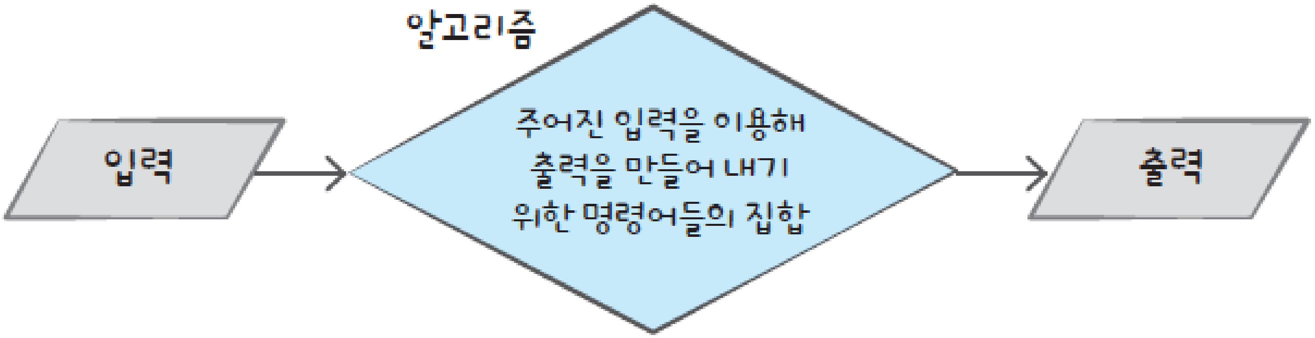
5장. 알고리즘 개요

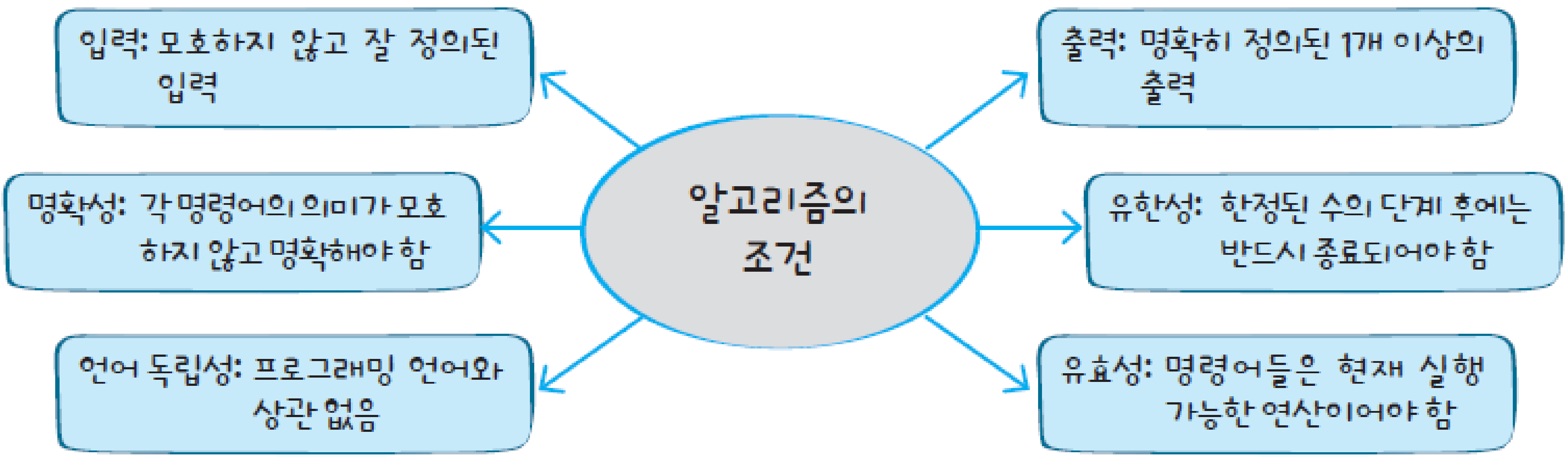
05-1 알고리즘이란?

05-2 알고리즘의 성능 분석

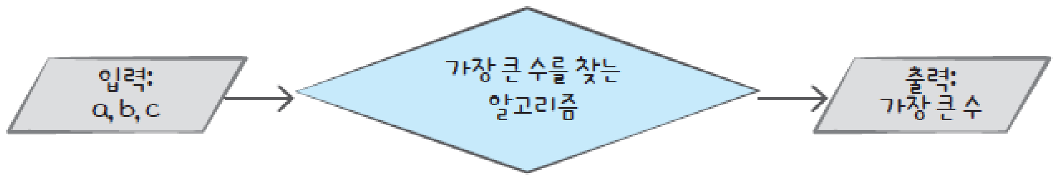
5.1 알고리즘이란?

- 알고리즘 (algorithm)
  - 주어진 문제를 해결하기 위한 단계적인 절차





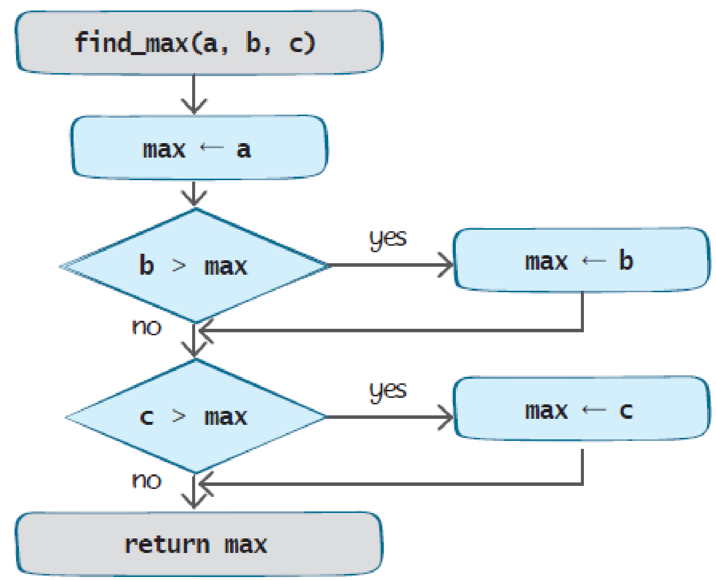
알고리즘의 기술 방법



(1) 자연어 표현

```
find_max( a, b, c )
    a를 최댓값을 저장하는 변수 max에 복사합니다.
    만약 b가 max보다 크면 b를 max에 복사합니다.
    만약 c가 max보다 크면 c를 max에 복사합니다.
    max를 반환합니다.
```

(2) 흐름도 표현



알고리즘의 기술 방법

(3) 유사 코드 표현

```
find_max( a, b, c )
        max ← a
        if b > max then
            max ← b
        if c > max then
            max ← c
        return max
```

(4) 파이썬 표현

```
def find_max( a, b, c ) :
        max = a
        if b > max :
            max = b
        if c > max :
            max = c
        return max
```

바로 실행할 수 있음!

5.2 알고리즘의 성능 분석

- 성능의 기준
  - 연산량 : 알고리즘이 얼마나 적은 연산을 수행하는가?
  - 메모리 사용량 : 얼마나 적은 메모리 공간을 사용하는가?

- 시간 효율성 분석 방법
  - 실행 시간 측정 방법
  - 복잡도 분석

실행 시간 측정 방법

- time 모듈을 이용한 실행시간 측정 예

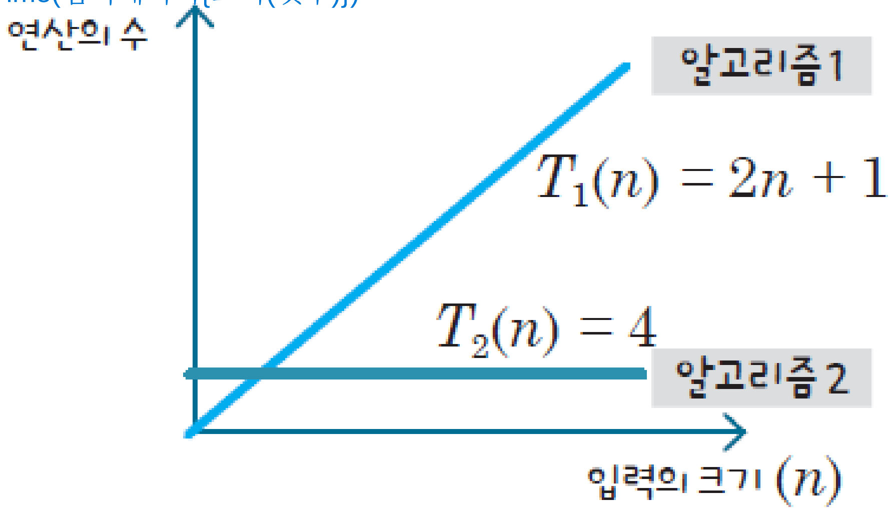
```
import time                           # time 모듈 불러오기
start = time.time()                   # 현재 시각을 start에 저장(시작 시각)
testAlgorithm(input)                  # 실행시간을 측정하려는 알고리즘 호출
…
end = time.time()                     # 현재 시각을 end에 저장(종료 시각)
print("실행시간 = ", end-start)       # 실제 실행시간(종료-시작)을 출력
```

- 문제점
  - 구현해야 함
  - 같은 조건의 HW, SW 환경과 언어로 비교해야 함
  - 실험된 입력에 대해서만 실행 시간을 주장 가능

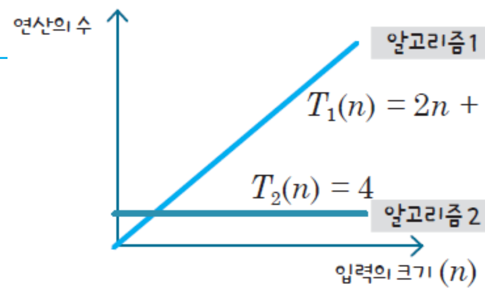
복잡도 분석(Complexity Analysis) 방법

- 알고리즘의 성능 평가 → 알고리즘의 복잡도 분석
  - 알고리즘을 구현하지 않고
  - 알고리즘의 시간복잡도(Time Complexity)
  - 알고리즘의 공간 복잡도(Space Complexity)
- 알고리즘의 복잡도 표기 ; **복잡도 함수** $\boldsymbol{T(n)}$
  - **알고리즘에서 얼마나 많은 연산이 실행되는지를 계산**
  - $T(n)$: **연산의 실행 횟수**를 **입력크기** $\boldsymbol{n}$**에 대한 함수로** 표기

T(n) = Time(입력데이터{크기(갯수)})



예: 알고리즘의 복잡도 함수 계산



- 1부터 n까지 합을 구하는 두 가지 알고리즘
  - 알고리즘 1: 반복문 이용

```
calc_sum1( n )
      sum ← 0                    # 1회 수행
      for i ← 1 to n then        # n회 수행(반복 제어부)
          sum ← sum + i          # n회 수행(반복문 내부)
      return sum                 # 1회 수행
```

$T_1(n) = 2n$

T1(n) = 2n+ 1

  - 알고리즘 2: 합 공식 이용

```
calc_sum2( n )
      sum ← n * (n+1) / 2        # 1회 수행
      return sum                 # 1회 수행
```

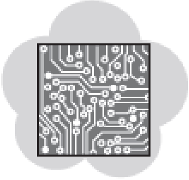
T2(n) = 4

알고리즘의 복잡도의 점근적 표기

- 어느 알고리즘이 효율적일까?

문제: n개의 숫자를
오름차순으로 정렬하라.

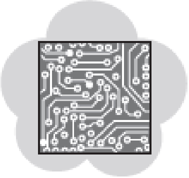
알고리즘 A



65536n + 2000000

$T(n) = 65536n + 200000$

알고리즘 B



$n^2 + 2n$

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

n(입력의 크기)	알고리즘 A $65536n + 2000000$	비교	알고리즘 B $n^2 + 2n$
1	2,065,536	>	3
10	2,655,360	>	120
100	8,553,600	>	10,200
1,000	67,536,000	>	1,002,000
10,000	657,360,000	>	100,020,000
100,000	6,555,600,000	<	10,000,200,000
1,000,000	65,538,000,000	<	1,000,002,000,000
10,000,000	655,362,000,000	<	100,000,020,000,000
100,000,000	6,553,602,000,000	<	10,000,000,200,000,000
1,000,000,000	65,536,002,000,000	<	1,000,000,002,000,000,000

n이 커질수록 엄청난 연산이 필요함

알고리즘 복잡도의 표현 방법 : 점근적 표기

- 점근적 표기(asymptotic notation) :
  - 여러 항을 갖는 복잡도 함수를 최고차항만을 계수 없이 취해 단순하게 표현하는 방법
  - 입력의 크기 n의 증가에 따라 연산량이 얼마나 빨리 증가하는가?
  - 증가속도를 표현
  - 상한/하한/ 동일 등급의 개념 적용
    - 빅오(O)
      - 예: $O(n)$
    - 빅오메가(Ω)
      - 예: $\Omega(n)$
    - 빅세타(Θ)
      - 예시: $\Theta(n)$

n(입력의 크기)	알고리즘 A $65536n + 2000000$	비교	알고리즘 B $n^2 + 2n$

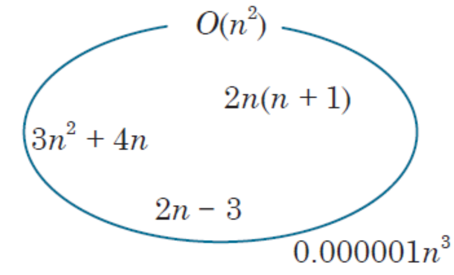
$_1T(n) = 65536n + 200000$ $_2T(n) = n^2 + 2n$

1T(n)과 2T(n)의 경우 n의 최고 차항을 찾아서 크기 비교
1T(n) = 65536n 2T(n) = n의 제곱이므로 2T(n)이 상한

암기

빅오(big-O) 표기법

$$3n^2 + 4n \in O(n^2)$$
$$2n - 3 \in O(n^2)$$
$$2n(n+1) \in O(n^2)$$
$$3n^2 + 4n \notin O(n)$$
$$0.000001n^3 \notin O(n^2)$$
$$1000^n \in O(n!)$$



무조건 시험 나옴

- $O(g(n))$
  - 증가속도가 $g(n)$과 **갈거나 낮은** 모든 복잡도 함수($T(n)$)를 포함하는 집합 T(n) = O(g(n)) -> {T(n) <= G(n)}일 경우 G(n) = upper bound
  - **처리 시간의 상한(upper bound)**
  - 이 알고리즘은 어떤 경우에도 $g(n)$에 **비례하는 시간 안에는 반드시 완료**된다는 의미
  - $\boldsymbol{g(n)}$**보다 더 빨리 처리**될 수는 있지만 절대로 $g(n)$보다 **더 걸릴 수는 없음**
- 시간복잡도를 정확히 분석하기 어려운 경우는 상한을 구해 빅오 표기법 사용
  - **최악의 상황을 고려한 해결책을 찾기 때문에 빅오 표기법을 주로 사용**

\- 예시: $T(n) = 2n + 1 \Rightarrow \boldsymbol{T(n) \in O(n^2)}$ => $\boldsymbol{T(n)}$**은** $\boldsymbol{O(n^2)}$**에 속한다.**

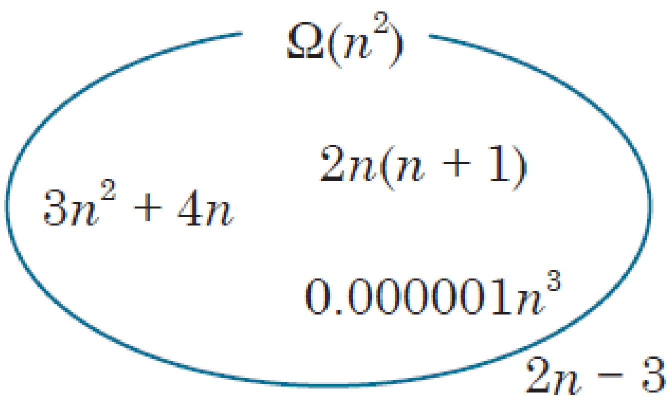
O(n의 자승)에서 n의 자승보다 T(n)의 n이 더 작기에 O(n의 자승)에 속한다.

빅오메가(big-omega) 표기법

알고리즘은 상한값(최악의 경우)을 구하지 하한값을 고려하는 경우는 거의 없기에 시험 x

- $\Omega(g(n))$
  - 증가속도가 $\boldsymbol{g(n)}$과 **갈거나 높은** 모든 복잡도 함수를 포함하는 집합
  - **처리 시간의 하한(Lower Bound)**
  - $\Omega(g(n))$은 아무리 빨리 처리하더라도 $\boldsymbol{g(n)}$**에 비례하는 시간 이상은 반드시 걸린다**
- **예시**: $T(n) = 2n^3 + 3n \Rightarrow T(n) \in \Omega(n^2)$ => $\boldsymbol{T(n)}$**은** $\Omega(\boldsymbol{n^2})$**에 속한다.**

$$2n^3 + 3n \in \Omega(n^2)$$
$$2n(n+1) \in \Omega(n^2)$$
$$100000n + 8 \notin \Omega(n^2)$$

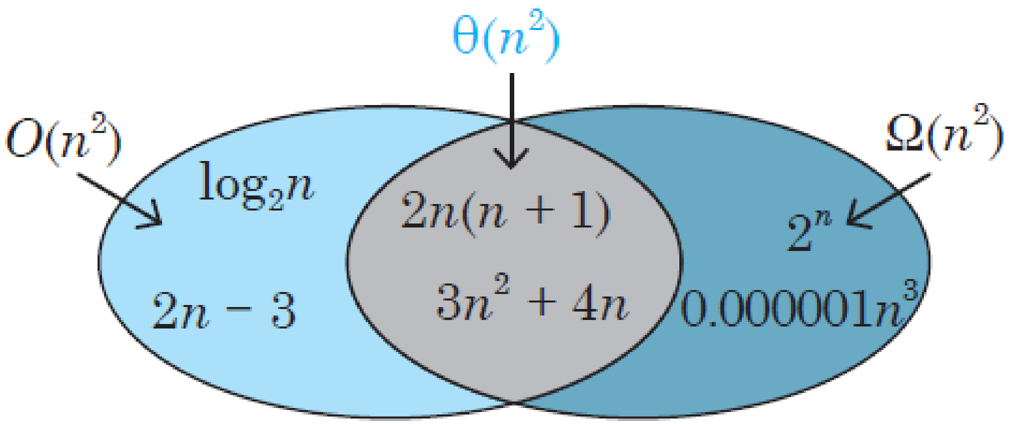


빅세타(big-theta) 표기법

시험 x

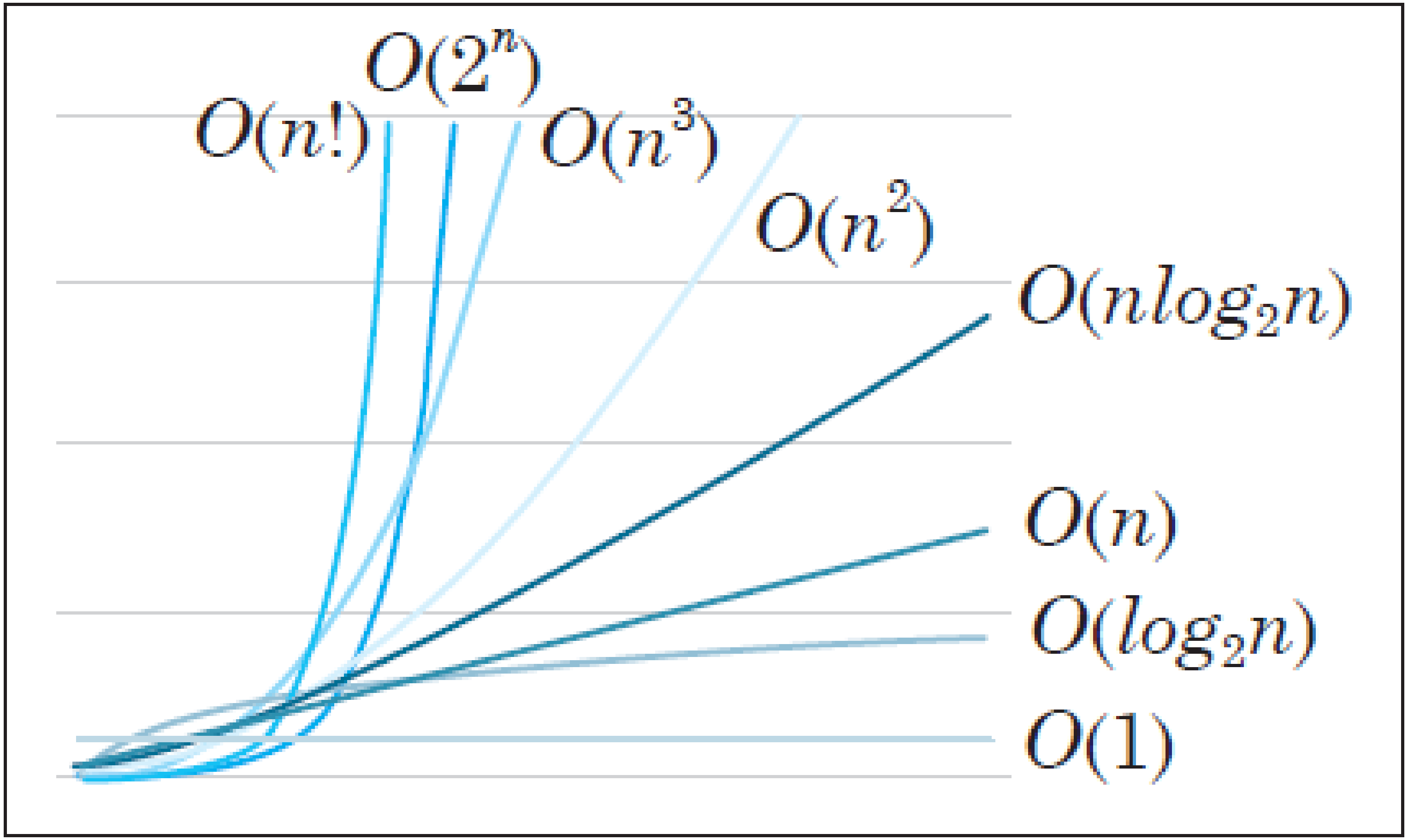
- $\Theta(g(n))$
  - 증가속도가 $g(n)$과 **같은** 모든 복잡도 함수를 포함하는 집합
  - 처리 시간이 상한인 동시에 하한
- 시간 복잡을 정확히 계산할 수 있으면 빅세타 표기법을 사용

$$3n^2 + 4n \in \Theta(n^2)$$
$$2n - 3 \notin \Theta(n^2)$$
$$0.000001n^3 \notin \Theta(n^2)$$



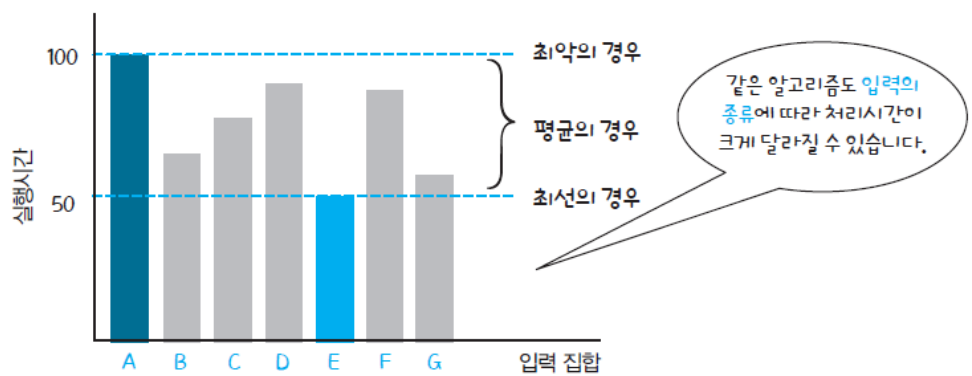
빅오 표기: n이 증가함에 따라 시간복잡도 함수의 증가속도

$$O(1) < O(\log n) < O(n) < O(n\log n) < O(n^2) < O(n^3) < O(2^n) < O(3^n) < O(n!)$$



최선, 최악, 평균적인 효율성

- 최선의 경우(best case)
  - 실행 시간이 가장 적은 경우. 알고리즘 분석에는 큰 의미가 없음
- 평균적인 경우(average case)
  - '평균' : 모든 데이터가 **균일하게 탐색 되는** 상황을 이미
    - 만일 숫자가 key로 사용될 가능성이 $1/n$로 동일
  - 알고리즘의 **모든 입력을 고려**하고 각 **입력이 발생할 확률을** 고려한 평균 실행 시간. 정확히 계산이 어려움
- 최악의 경우(worst case)
  - **입력의 구성이** 알고리즘의 실행시간을 많이 요구하는 경우. **알고리즘 분석하는데 가장 중요**



예: 리스트에서 최댓값을 찾는 알고리즘

```
01:  def find_max( A ):
02:      n = len(A)             # 입력의 크기
03:      max = A[0]             # max 초기화
04:      for i in range(n) :    # 반복 제어부
05:          if A[i] > max :    # 반복문 내부 -> n번 반복(가장 많이 처리)
06:              max = A[i]
07:      return max             # 결과 반환
```

- 입력의 크기 n : 리스트의 크기
- $T(n)$: n이 증가함에 따라 비례하는 수의 연산은?
  - 가장 많이 처리되는 부분: 5행
  - $T(n) = n$ 알고리즘의 시간 복잡도를 계산하고 그 계산법을 적어준다.(시험)
- **입력의 구성에 따라 알고리즘의 실행 시간에 따른 효율성 차이**
  - **순차탐색(linear search)**
  - 리스트의 크기가 같다면 내용에 상관없이 효율성에 차이가 없음
  - **최선의 경우, 평균적인 경우, 최악의 경우 동일**
  - $\boldsymbol{O(n), \Omega(n), \Theta(n)}$ T(n) = O(n)

예: 리스트에서 어떤 값을 찾는 알고리즘

```
01:  def find_key( A, key ):
02:      n = len(A)                # 입력의 크기
03:      for i in range(n) :       # 반복 제어부
04:          if A[i] == key :      # 탐색 성공 --> 인덱스 반환
05:              return i
06:      return -1                 # 탐색 실패 --> -1 반환
```

- 입력의 크기 n : 리스트의 크기
- 순차탐색(linear search)

32	14	5	17	23	9	11	4	26	29

  – 가장 많이 처리되는 부분: 4행
  – **입력의 구성에 따라 탐색 횟수가 달라짐**
  – **복잡도 함수의** 효율성 차이 → 있음
    - **최선의 경우 : $T(n) = 1 \Rightarrow O(1)$** 찾는 것이 32인경우
      – key <- A[0]
    - **최악의 경우 : $T(n) = n \Rightarrow O(n)$** 찾는 것이 29이거나 존재하지 않을 경우
      – Key >= A[$n-1$] 또는 리스트에 존재하지 않는 경우
    - **평균의 경우: $T(n) = \frac{1+2+\ldots+n}{n} = \frac{\frac{n(n+1)}{2}}{n} = \frac{n+1}{2} \Rightarrow O(n)$** (10+1)/2 = 5.5가 O(n)