01 알고리즘은 작업 과정의 묘사

02 알고리즘은 생각하는 방법의 훈련

03 알고리즘은 자료구조의 확장

Chapter 01 알고리즘이란

컴퓨터인공지능학부
장 재 우 교수

알고리즘은 작업 과정의 묘사

- 어떤 작업을 수행하기 위한 과정을 애매하지 않게 기술한 것

- 어떤 작업을 수행하기 위해 입력을 받아 출력을 만들어내는 과정을 애매하지 않게 기술한 것

바람직한 알고리즘

■ 명확해야 한다

- 이해하기 쉽고 가능하면 간명하도록
- 지나친 기호적 표현은 명확성을 떨어뜨림
- 명확성을 해치지 않으면 일반언어의 사용도 무방

■ 효율적이어야 한다

- 같은 문제를 해결하는 알고리즘들의 수행 시간이 수백만 배 이상 차이날 수 있다

그림 1-1 같은 문제를 해결하는 데 100년이 걸리는 알고리즘 vs. 1분이 걸리는 알고리즘

입출력과 알고리즘 예

- **문제**
  - 100명의 학생의 시험점수의 최대값을 찾으라
- **입력**
  - 100개의 점수
- **출력**
  - 입력된 100개의 점수들 중 최댓값

- **알고리즘**

maxScore(x[], n):
    $x[1...n]$의 값을 차례대로 보면서 최댓값을 계산한다
    return 위에서 찾은 최댓값

알고리즘은 생각하는 방법의 훈련

- 문제 자체를 해결하는 알고리즘을 배운다
- 그 과정에 깃든 '생각하는 방법'을 배우는 것이 더 중요하다
- 미래에 다른 문제를 해결하는 생각의 빌딩블록을 제공한다

그림 1-2 알고리즘은 생각하는 방법을 훈련하는 도구

알고리즘은 자료구조의 확장

■ 선행 과목

- 프로그래밍, 자료구조

■ 자료구조

- 건축의 건축 자재나 모듈 같은 것
- 자동차 제작의 부품이나 모듈 같은 것

그림 1–3 부품(자료구조) 선택의 중요성

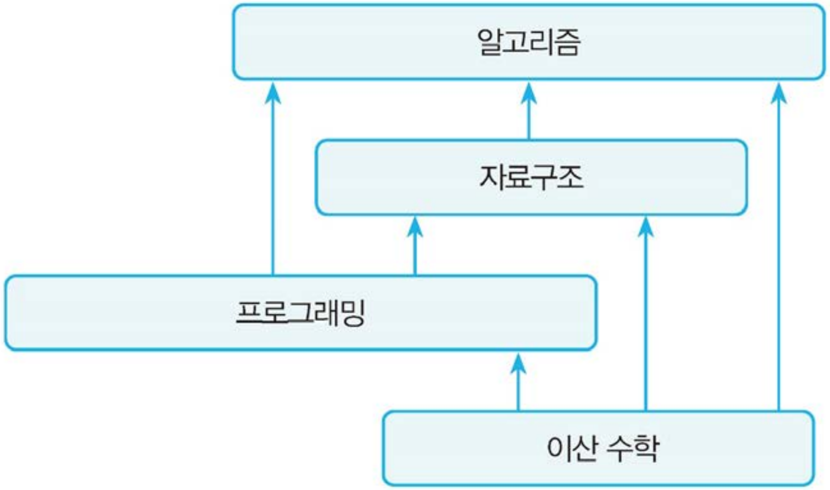


그림 1–4 알고리즘, 자료구조, 프로그래밍, 이산 수학의 관계

알고리즘 표기법

- 자연어를 이용한 서술적 표현
  - 서술적일 뿐만 아니라 쓰는 사람에 따라 일관성이나 명확성을 유지하기 어려움
  - 누구라도 쉽게 이해하고 쓸 수 있어야 하는 알고리즘을 표현하는 데는 한계가 있음
- 순서도(flow chart)를 이용한 도식화
  - 명령의 흐름을 쉽게 파악할 수 있지만 복잡한 알고리즘을 표현하는 데는 한계가 있음
- 프로그래밍 언어를 이용한 구체화
  - 해당 언어를 모르면 이해하기 어려움
  - 다른 프로그래밍 언어로 프로그램을 개발하는 경우에는 다른 프로그래밍 언어로 변환해야 하므로 범용성이 떨어짐
- 가상코드를 이용한 추상화
  - 가상코드(Pseudo-Code)는 직접 실행할 수는 없지만 일반적인 프로그래밍 언어와 형태가 유사해 프로그래밍 언어로 구체화하기가 쉬움

이 책에서 사용하는 알고리즘 표기법

알고리즘 4-4	버블 정렬 2

```
bubbleSort(A[], n):  ❶ ▷ A[0...n-1]을 정렬한다.
        sorted ← FALSE  ❷
        for last ← n-1 downto 1
    ❹       sorted ← TRUE
            for i ← 0 to last-1
                if (A[i] > A[i+1])
          ❸         A[i] ↔ A[i+1]  ❺ ▷ 원소 교환
                    sorted ← FALSE
            if (sorted = TRUE) return
```

❶ *함수 시작*

❷ *문장 뒤 세미콜론 생략*

❸ *If 뒤의 then 생략*

❹ *for 루프에 속하는 문장 들여쓰기*

❺ *주석*

01 알고리즘 분석을 위한 기초 개념

02 점근적 표기

03 점근적 표기의 엄밀한 정의

Chapter 02 알고리즘 설계와 분석의 기초

컴퓨터인공지능학부
장 재 우 교수

1. 알고리즘을 왜 분석하는가?

- 무결성 확인
- 자원 사용의 효율성 파악
  - 자원
    - 시간
    - 공간 – 메모리 크기

2. 알고리즘의 분석

- 크기가 작은 문제
  - 알고리즘의 효율성이 중요하지 않음
  - 비효율적인 알고리즘도 무방
- 크기가 충분히 큰 문제
  - 알고리즘의 효율성이 중요
  - 비효율적인 알고리즘은 치명적
- 입력의 크기가 충분히 큰 경우에 대한 분석을 **점근적 분석**

3. 알고리즘의 수행시간

- 알고리즘의 수행시간을 좌우하는 기준
  - for 루프의 반복횟수
  - 특정한 행이 수행되는 횟수
  - 함수의 호출횟수
- 예를 통해 알고리즘 수행시간을 살펴봄

알고리즘의 수행시간

```
sample1(A[ ], n)
{
        k = n/2 ;
        return A[k] ;
}
```

✓ n에 관계없이 상수 시간이 소요된다.

알고리즘의 수행시간

```
sample2(A[ ], n)
{
    sum ← 0 ;
    for i ← 1 to n
        sum← sum+ A[i] ;
    return sum ;
}
```

✓ n에 비례하는 시간이 소요된다.

알고리즘의 수행시간

```
sample3(A[ ], n)
{
    sum ← 0 ;
    for i ← 1 to n
        for j ← 1 to n
            sum← sum+ A[i]*A[j] ;
    return sum ;
}
```

✓ n^2에 비례하는 시간이 소요된다.

알고리즘의 수행시간

```
sample4(A[ ], n)
{
    sum ← 0 ;
    for i ← 1 to n
        for j ← 1 to n {
            k ← A[1 ... n] 에서 임의로 n/2 개를 뽑을 때
            이 가운데 최대값 ;
            sum ← sum + k ;
        }
    return sum ;
}
```

✓ n^3에 비례하는 시간이 소요된다.

알고리즘의 수행시간

```
sample5(A[ ], n)
{
        sum ← 0 ;
        for  i ← 1 to n
                for  j ←  i+1 to n
                        sum← sum+ A[i]*A[j] ;
        return sum ;
}
```

✓ sample 3(A[ ], n) 와 마찬가지로 n^2에 비례하는 시간이 소요된다.

알고리즘의 수행시간

```
factorial(n)
{
        if (n=1) return 1 ;
        return n*factorial(n-1) ;
}
```

✓ 재귀적으로 호출하기 때문에 **n!** 에 비례

4. 재귀와 귀납적 사고

- 재귀=자기호출(recurrence)
- 재귀적 구조
  - 어떤 문제 안에 크기만 다를 뿐 성격이 똑같은 작은 문제가 포함되어 있는 것
  - 예1: factorial
    - N! = N×(N-1)!
  - 예2: 병합정렬
    - 분할 정복(Divide and Conquer) 개념

분할 정복(Divide and Conquer) 개념

- 문제의 해답을 구하기 위해 아래로 내려가면서 하위의 해답을 찾는 하향식 알고리즘(Top-Down Algorithm)

1. 분할: 문제를 소문제로 분할
2. 정복: 각각의 소문제를 해결
3. 통합: 소문제의 해결 결과를 이용하여 전체 문제를 해결

- **소문제**란 원래 문제에서 크기만 줄인 것을 말하며, 분할 정복법에서 재귀를 활용하는 부분

```
mergeSort(A[ ], p, r)  {
    if (p < r) {
        q = (p+ r)/2;   --------------- ①   ▷ 분할 지점 계산
        mergeSort(A, p, q);  --------②   ▷ 전반부 정렬(정복)
        mergeSort(A, q+1, r); -----③   ▷ 후반부 정렬(정복)
        merge(A, p, q, r);   ---------④   ▷ 통합
    } }
```

재귀 및 분할 정복 예제: Mergesort (병합정렬)

```
mergeSort(A[ ], p, r)
▷ A[p ... r]을 정렬한다
{
    if (p < r) then {
        q ← (p+r)/2;   ----------------------- ①  ▷ p, r의 중간 지점 계산
        mergeSort(A, p, q);  ---------------- ②  ▷ 전반부 정렬
        mergeSort(A, q+1, r); -------------- ③  ▷ 후반부 정렬
        merge(A, p, q, r);  ------------------ ④  ▷ 전반부와 후반부 병합
    }
}
```

```
merge(A[ ], p, q, r)
▷ 이미 정렬된 배열의 전반부와 후반부를 병합
{
    정렬되어 있는 두 배열 A[p ... q]와 A[q+1 ... r]을 병합하여
    정렬된 하나의 배열 A[p ... r]을 만든다.
}
```

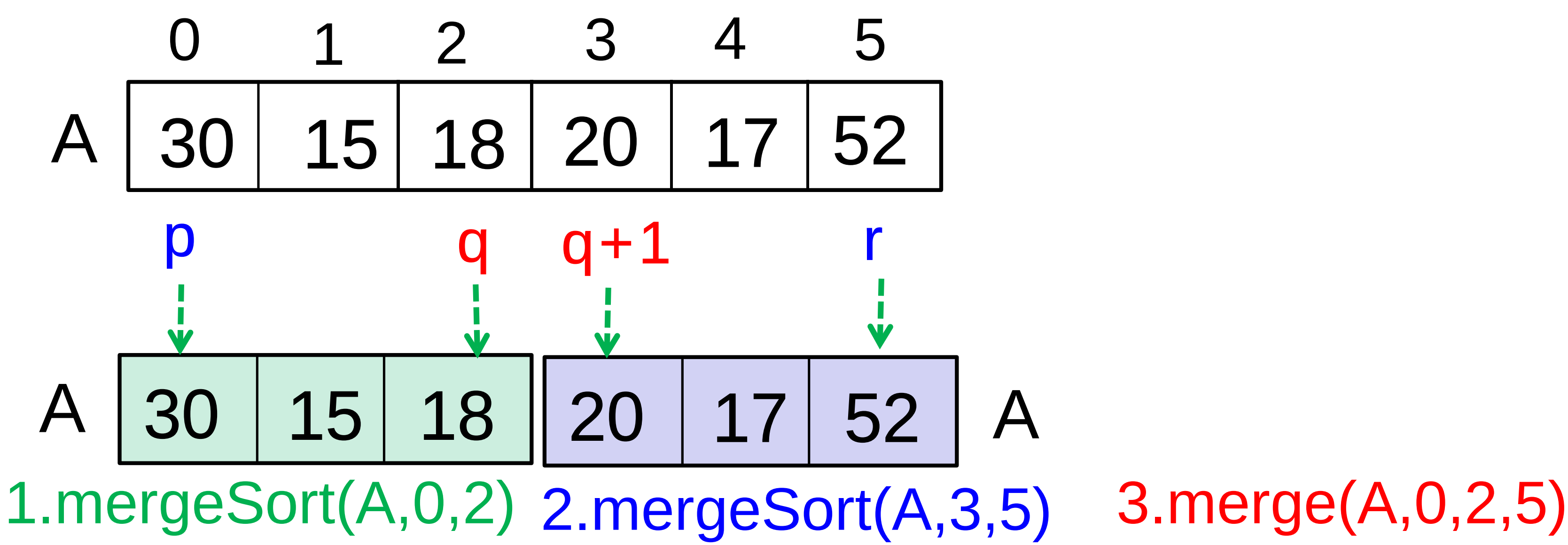
재귀 및 분할 정복 예제: Mergesort

```
merge(A[ ], p, q, r)
{
    i ← p; j ← q+1; t ← 0;
    while (i <= q and j <= r) {
        if (A[i] <= A[j])
        then tmp[t++] ← A[i++] ;
        else tmp[t++] ← A[j++] ;
    }
    while (i <= q)                ▷ 전반부 정렬이 남은 경우
        tmp[t++] ← A[i++] ;
    while (j <= r)                ▷ 후반부 정렬이 남은 경우
        tmp[t++] ← A[j++] ;
    i ← p; t ← 1;
    while (i <= r)                ▷ 결과를 A[p .... r] 에 저장
        A[i++] ← tmp[t++] ;

}
```

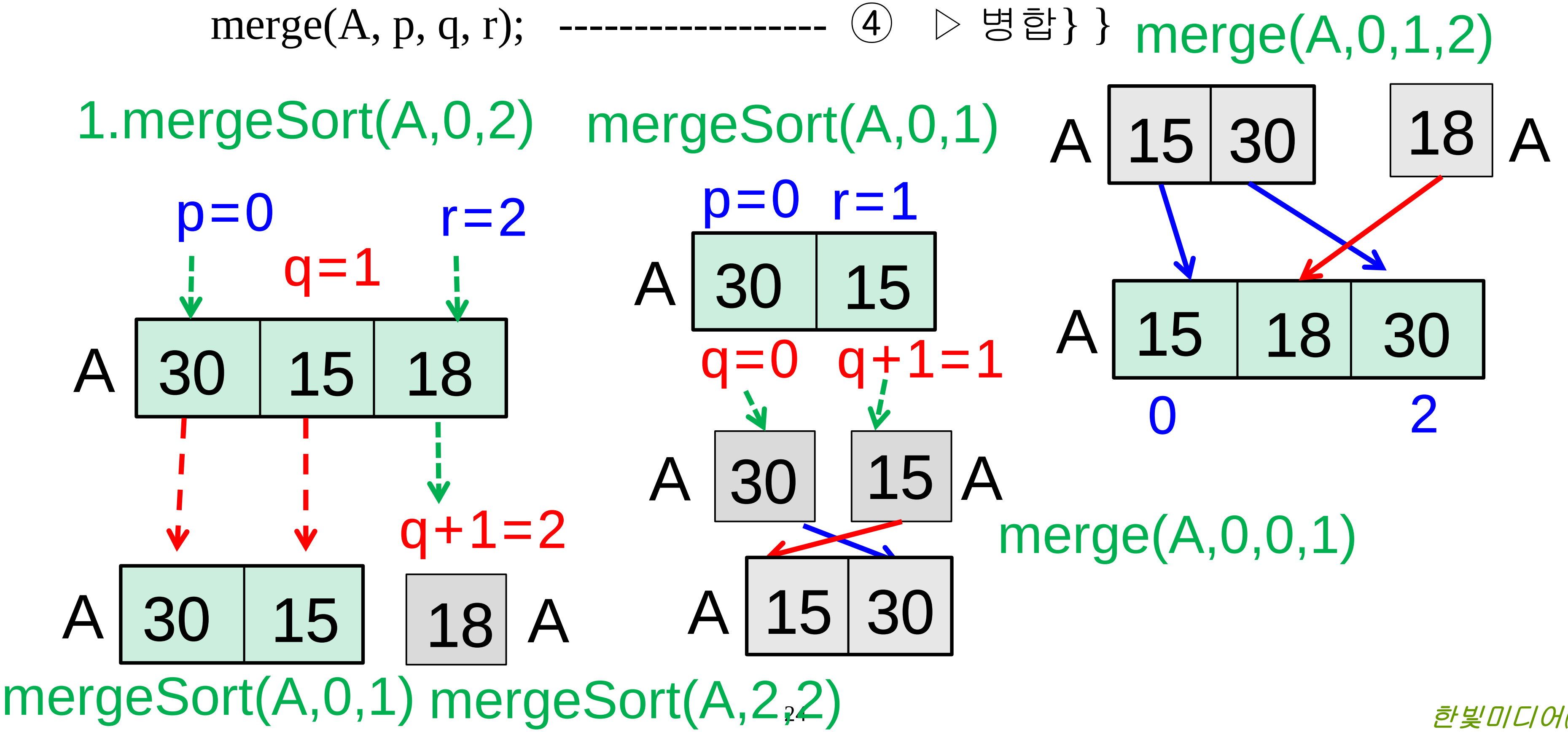
재귀 및 분할 정복 예제: Mergesort (병합정렬)

```
mergeSort(A[ ], p, r)  {
      if (p < r) {
            q = (p+r)/2;   ------------------------  ①    ▷ p, r의 중간 지점 계산
            mergeSort(A, p, q);  ----------------  ②  ▷ 전반부 정렬
            mergeSort(A, q+1, r); --------------  ③  ▷ 후반부 정렬
            merge(A, p, q, r);   -----------------  ④  ▷ 병합
} }
```



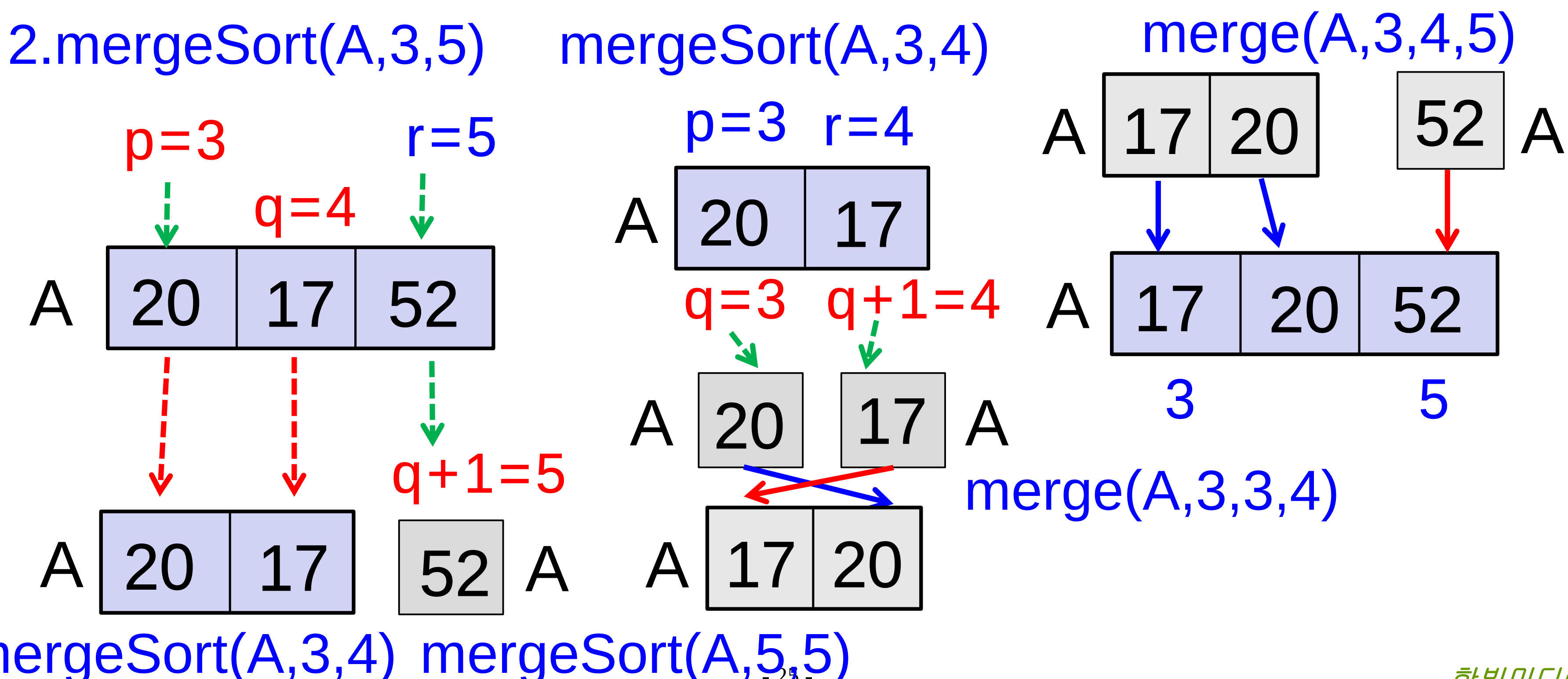
재귀 및 분할 정복 예제: Mergesort (병합정렬)

```
mergeSort(A[ ], p, r)  {
     if (p < r) {
          q = (p+r)/2;   -------------------------  ①  ▷ 중간 지점 계산
          mergeSort(A, p, q);  ----------------  ②  ▷ 전반부 정렬
          mergeSort(A, q+1, r); --------------  ③  ▷ 후반부 정렬
          merge(A, p, q, r);   -----------------  ④  ▷ 병합} }
```



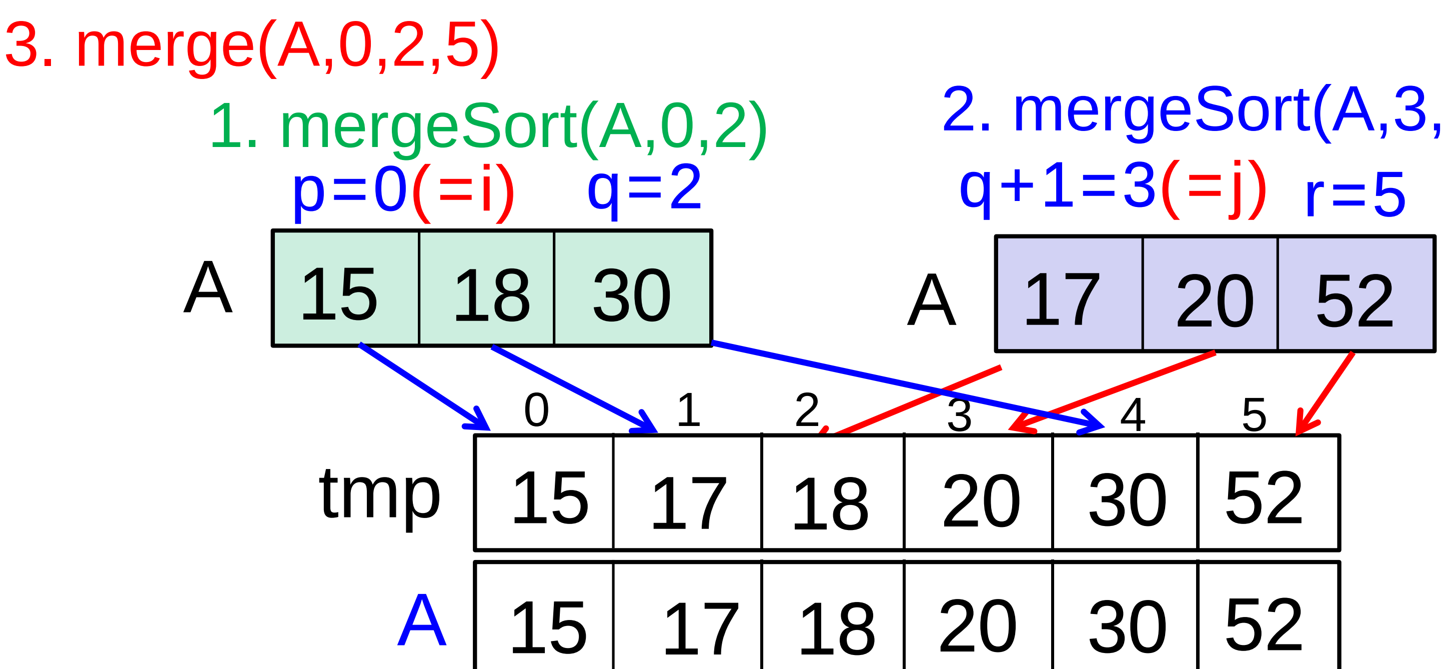
재귀 및 분할 정복 예제: Mergesort (병합정렬)

```
mergeSort(A, p, r)  {
    if (p < r) {
        q = (p+r)/2;   ----------------------  ①    ▷ 중간 지점 계산
        mergeSort(A, p, q);  ---------------  ②    ▷ 전반부 정렬
        mergeSort(A, q+1, r); -------------  ③    ▷ 후반부 정렬
        merge(A, p, q, r);   -----------------  ④    ▷ 병합 } }
```



재귀 및 분할 정복 예제: Mergesort (병합정렬)

```
merge(A[ ], p, q, r)
{
    i ← p; j ← q+1; t ← 0;
    while (i <= q and j <= r) then {
        if (A[i] <= A[j])
        then tmp[t++] ← A[i++] ;
        else tmp[t++] ← A[j++] ;
    }
    while (i <= q)  ▷ 전반부 정렬이 남은 경우
            tmp[t++] ← A[i++] ;
    while (j <= r)  ▷ 후반부 정렬이 남은 경우
            tmp[t++] ← A[j++] ;
        i ← p; t ← 1;
    while (i <= r) ▷ 결과를 A[p ..r] 저장
            A[i++] ← tmp[t++] ;
}
```



요약: 분할 정복(Divide and Conquer) 개념

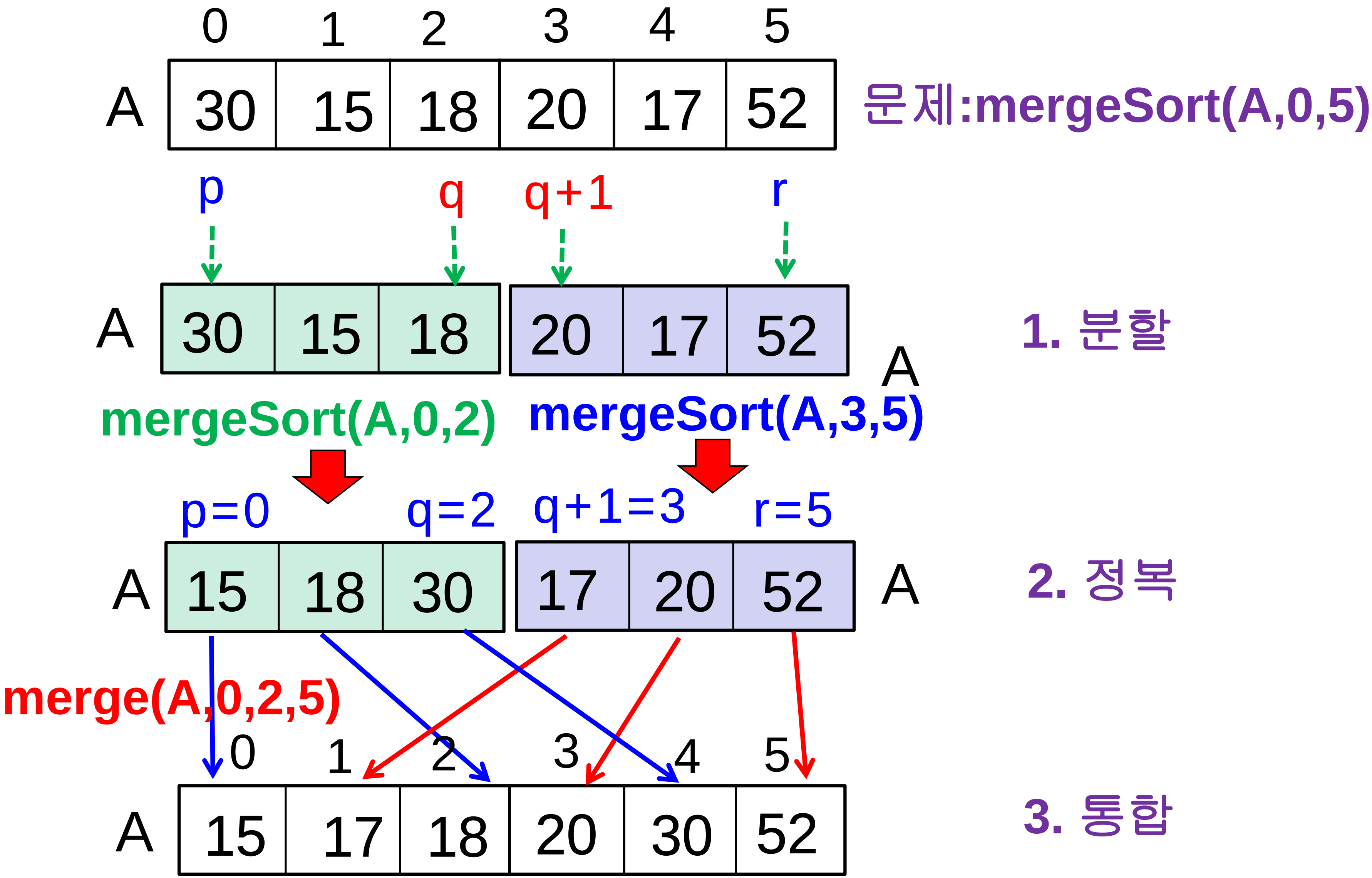
- 문제의 해답을 구하기 위해 아래로 내려가면서 하위의 해답을 찾는 하향식 알고리즘(Top-Down Algorithm)

1. 분할: 문제를 소문제로 분할
2. 정복: 각각의 소문제를 해결
3. 통합: 소문제의 해결 결과를 이용하여 전체 문제를 해결

- **소문제**란 원래 문제에서 크기만 줄인 것을 말하며, 분할 정복법에서 재귀를 활용하는 부분

```
mergeSort(A[ ], p, r)  {
      if (p < r) {
            q = (p+ r)/2;   --------------- ①   ▷ p, q의 중간 지점 계산
            mergeSort(A, p, q);  --------②   ▷ 전반부 정렬
            mergeSort(A, q+1, r); -----③   ▷ 후반부 정렬
            merge(A, p, q, r);   ---------④   ▷ 병합
      } }
```

요약: 분할 정복 예제-병합정렬



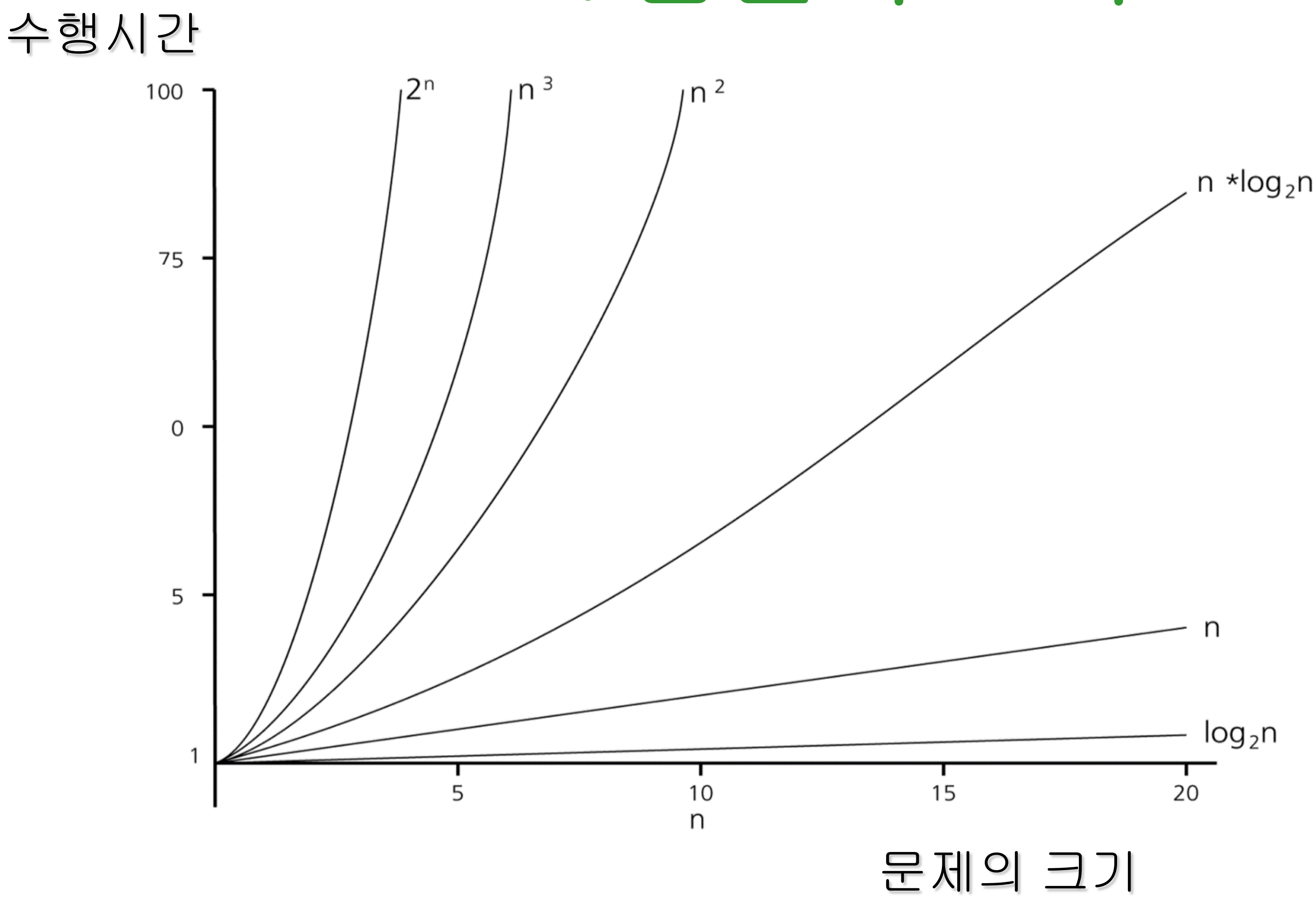
5. 알고리즘으로 어떤 문제를 푸는가?

- 카 네비게이션 : 그래프 알고리즘
- 스케쥴링
  - TSP, 차량 라우팅, 작업공정,
- Human Genome Project
  - 매칭, 계통도, functional analyses,
- 검색
  - 데이터베이스, 웹페이지,
- 자원의 배치
- 반도체 설계
  - Partitioning, placement, routing

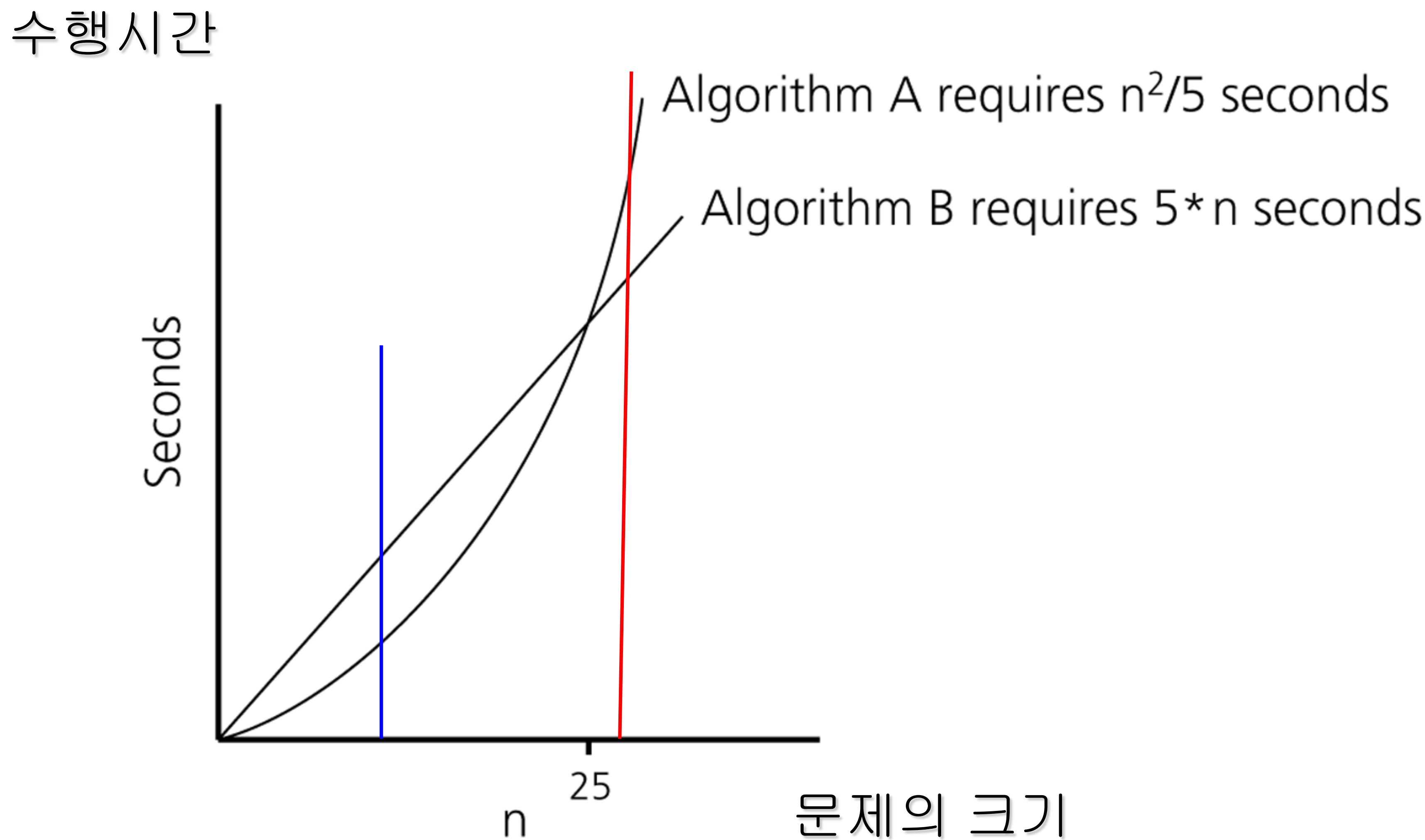
알고리즘 분석

Function	10	100	1,000	10,000	100,000	1,000,000
1	1	1	1	1	1	1
$\log_2 n$	3	6	9	13	16	19
n	10	10^2	10^3	10^4	10^5	10^6
$n * \log_2 n$	30	664	9,965	10^5	10^6	10^7
n^2	10^2	10^4	10^6	10^8	10^{10}	10^{12}
n^3	10^3	10^6	10^9	10^{12}	10^{15}	10^{18}
2^n	10^3	10^{30}	10^{301}	$10^{3,010}$	$10^{30,103}$	$10^{301,030}$

II. 점근적 표기



알고리즘 분석


수행시간
Algorithm A requires $n^2/5$ seconds
Algorithm B requires 5*n seconds
Seconds
25
n
문제의 크기

점근적 분석 Asymptotic Analysis

- 입력의 크기가 충분히 큰 경우에 대한 분석
- 이미 알고있는 점근적 개념의 예

$$\lim_{n \to \infty} f(n)$$

- O, Ω, Θ, ω, o 표기법

II-1. 점근법 표기법 Asymptotic Notations

- $O(\ f(n)\ )$
  - 기껏해야 $f(n)$의 비율로 증가하는 함수
  - e.g., $O(n)$, $O(n \log n)$, $O(n^2)$, $O(2^n)$, …
- Formal definition
  - $O(\ \boldsymbol{f}(n)\ ) = \{\ \boldsymbol{g}(n) \mid \exists\, c > 0, n_0 \geq 0 \text{ s.t. } \forall\, n \geq n_0, c\boldsymbol{f}(n) \geq \boldsymbol{g}(n)\ \}$
  - $\boldsymbol{g}(n) = O(\boldsymbol{f}(n))$
- 직관적 의미(상한값: upper bound)
  - $\boldsymbol{g}(n) = O(\boldsymbol{f}(n)) \Rightarrow \boldsymbol{g}$ 는 $\boldsymbol{f}$ 보다 빠르게 증가하지 않는다
  - 상수 비율의 차이는 무시

점근적 표기법

- $O(f(n))$:정의 [Big "oh"]
  - 모든 n, n≥no에 대해 g(n)≤cf(n)인 조건을 만족하는 두 양의 상수 c와 no가 존재하기만 하면 g(n)=O(f(n))이다.
- 예제
  - $n \geq no=2$, $g(n)= 3n+2 \leq 4n = cf(n)$
    ⇨ $g(n)= 3n+2=O(n)$
  - $n \geq 3$, $g(n)= 3n+3 \leq 4n$ ⇨ $g(n)= 3n+3=O(n)$
  - $n \geq no=6$, $g(n)= 10n+6 \leq 11n = cf(n)$ ⇨ $g(n)= 10n+6=O(n)$
  - $n \geq no=5$, $g(n)=10n^2+4n+2 \leq 11n^2 = cf(n)$
    ⇨ $g(n)=10n^2+4n+2 = O(n^2)$
  - $n \geq 4$, $6*2^n+n^2 \leq 7*2^n$ ⇨ $6*2^n+n^2 =O(2^n)$

점근적 표기법

- $\Omega(f(n))$
  - 적어도 $f(n)$의 비율로 증가하는 함수
  - $O(f(n))$과 대칭적
- Formal definition
  - $\Omega(\boldsymbol{f}(n)) = \{ \boldsymbol{g}(n) \mid \exists c > 0, n_0 \geq 0 \text{ s.t.} \forall n \geq n_0, c\boldsymbol{f}(n) \leq \boldsymbol{g}(n) \}$
- 직관적 의미(하한값: lower bound)
  - $\boldsymbol{g}(n) = \Omega(\boldsymbol{f}(n)) \Rightarrow \boldsymbol{g}$ 는 $\boldsymbol{f}$ 보다 느리게 증가하지 않는다

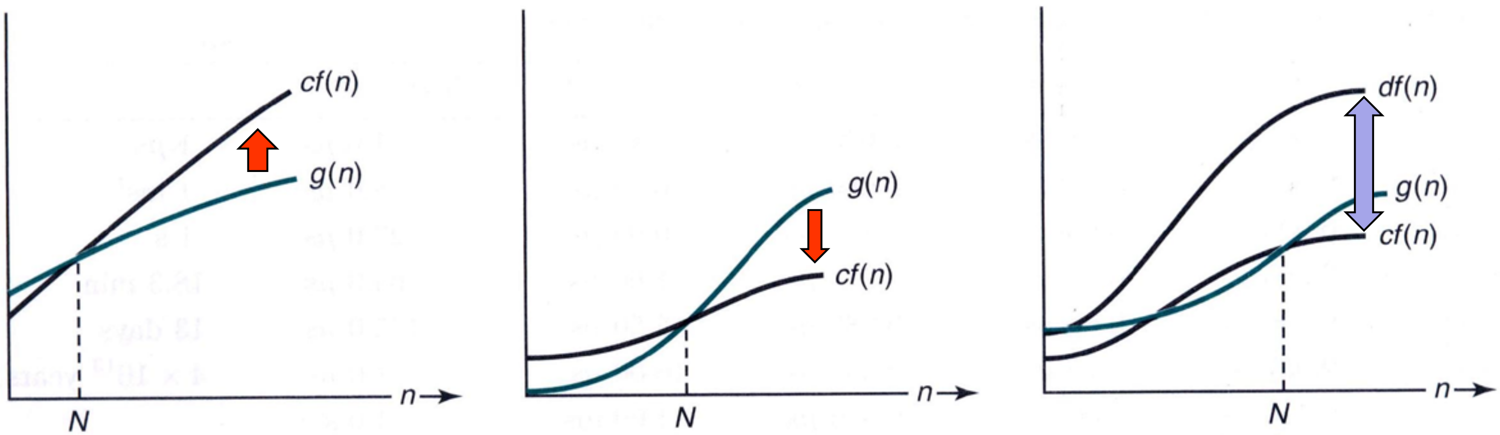
점근적 표기법

- $\Theta(f(n))$
  - $f(n)$의 비율로 증가하는 함수
- Formal definition
  - $\Theta(\boldsymbol{f}(n)) = O(f(n)) \cap \Omega(f(n))$
- 직관적 의미 (동일한 값)
  - $\boldsymbol{g}(n) = \Theta(\boldsymbol{f}(n)) \Rightarrow$ $\boldsymbol{g}$ 는 $\boldsymbol{f}$ 와 같은 정도로 증가한다

각 점근적 표기법의 직관적 의미

- $O(\ f(n)\ )$
  - Upper bound
- $\Omega(\ f(n)\ )$
  - Lower bound
- $\Theta(\ f(n)\ )$
  - Tight bound

각 점근적 표기법의 직관적 의미



(a) $g(n) \in O(f(n))$ (b) $g(n) \in \Omega(f(n))$ (c) $g(n) \in \Theta(f(n))$

- $O(\ f(n)\ )$
  - Upper bound
- $\Omega(\ f(n)\ )$
  - Lower bound
- $\Theta(\ f(n)\ )$
  - Tight bound

점근적 복잡도의 예

- 정렬 알고리즘들의 복잡도 표현
  - 병합정렬
    - $O(n\log n)$
  - 선택정렬(4장)
    - $\Theta(n^2)$

시간 복잡도 분석의 종류

- Worst-case
  - Analysis for the worst-case input(s)
- Average-case
  - Analysis for all inputs
  - More difficult to analyze

크기 *n*인 배열에서 원소 찾기

- Sequential search
  - 배열이 아무렇게나 저장되어 있을 때
  - Worst case: $\Theta(n)$
  - Average case: $\Theta(n)$

n=6

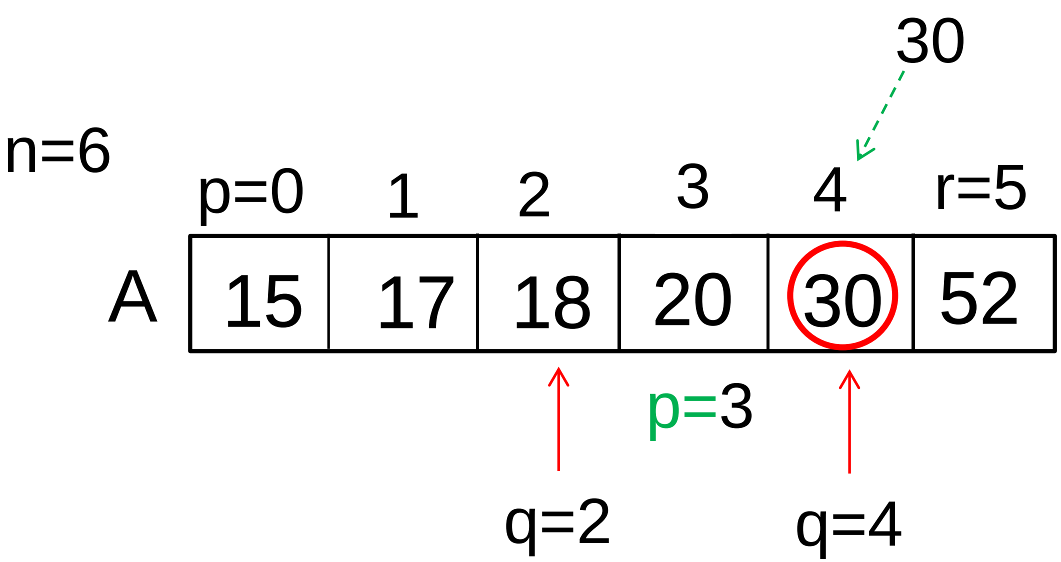
	0	1	2	3	4	5
A	30	15	18	20	17	52

15

17

크기 *n*인 배열에서 원소 찾기

- Binary search
  - 배열이 정렬되어 있을 때
  - Worst case: $\Theta(\log n)$
  - Average case: $\Theta(\log n)$



Thank You !