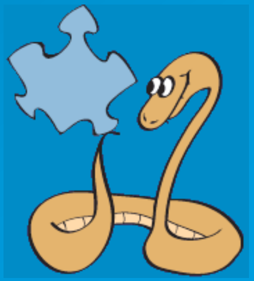
파이썬 알고리즘

ALGORITHMS IN PYTHON 최영규 지음



CHAPTER 04

축소 정복 기법

학습 내용

4.1 삽입 정렬(insertion sort)

4.2 위상 정렬

4.3 이진 탐색

4.4 거듭제곱 문제

4.5 선택 문제: k번째 작은 수 찾기

4.6 축소 정복 기법의 추가적인 예

알고리즘의 설계 기법들

- 억지(brute-force) 기법과 완전 탐색: 3장
  - 문제 정의를 가장 직접 사용, 원하는 답 구할때까지 모든 경우 테스트
- **축소 정복(decrease-and-conquer): 4장**
  - **주어진 문제를 하나의 좀 더 작은 문제로 축소하여 해결**
- 분할 정복(divide-and-conquer): 5장
  - 주어진 문제를 여러 개의 더 작은 문제로 반복적으로 분할하여 해결 가능한 충분히 작은 문제로 분할 후 해결
- 공간을 이용해 시간을 버는 전략: 6장
  - 추가적인 공간을 사용하여 처리시간 줄이는 전략
- 동적 계획법(divide-and-conquer): 7장
  - 더 작은 문제로 나누는 분할정복과 유사하지만, 작은분제 먼저해결 저장하고 다음에 더 큰 문제 해결
- 탐욕적(greedy) 기법: 8장
  - 단순하고 직관적인 방법으로 모든 경우 고려하여 가장 좋은 답을 찾는 것이 아니라 "그 순간에 최적"이라고 생각되는 것을 선택
- 백트래킹과 분기 한정 기법: 9장
  - 상태공간에서 단계적 해 찾기, 현재의 최종 해가 않된다면 더 이상 탐색하지 않고 백트래킹(되돌아가서)해서 다른 후보 해 탐색

축소 정복 기법(decrease-and-conquer)

- 주어진 문제와 더 작은 문제간의 관계를 이용하는 전략
  - 예: n! = n * (n-1)!
  - 분할 정복 기법의 일종
- 적용 방법
  - 하향식(top-down): 순환구조
  - 상향식(bottom-up): 반복구조

알고리즘 4.1 팩토리얼(하향식 축소 정복 기법)

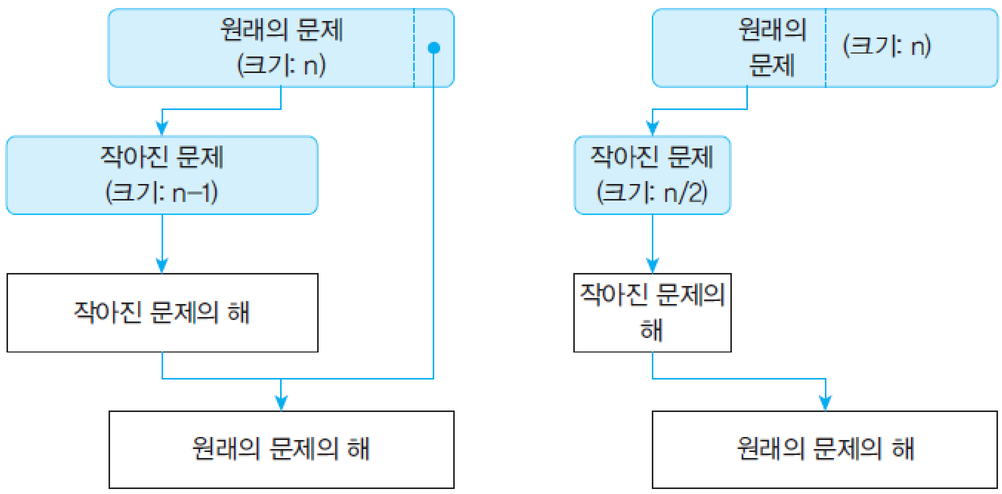
```
def factorial_recur(n) :
    if n == 1 :
        return 1
    else :
        return (n * factorial_recur(n - 1))
```

알고리즘 4.2 팩토리얼(상향식 축소 정복 기법)

```
def factorial_iter(n) :
    result = 1
    for k in range(1,n+1) :
        result = result * k
    return result
```

문제 축소 형태

- 고정 크기 축소(decrease by a constant):
  - 예: 팩토리얼
- 고정 비율 축소(decrease by a constant factor):
  - 예: 이진 탐색
- 가변 크기 축소(variable size decrease):
  - 예: 유클리드 알고리즘

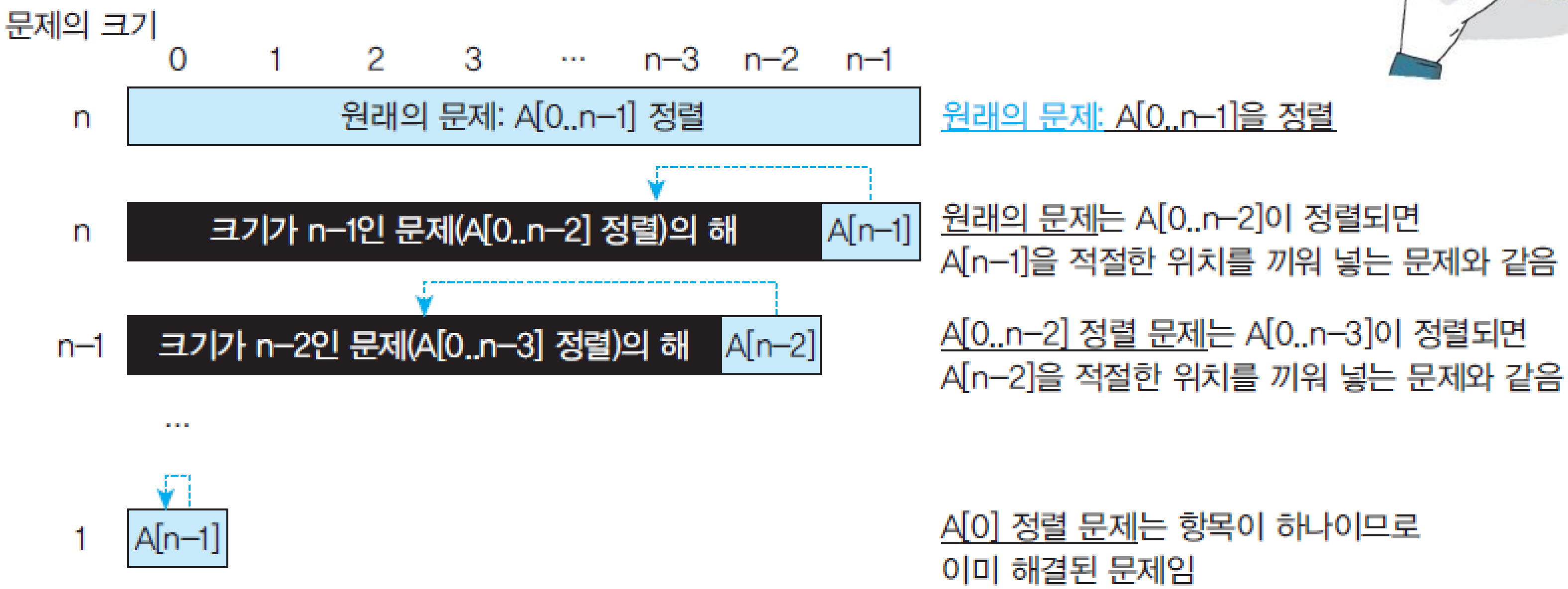


고정 크기(1) 축소 정복 기법
(decrease by a constant)

고정 비율(1/2) 축소 정복 기법
(decrease by a constant factor)

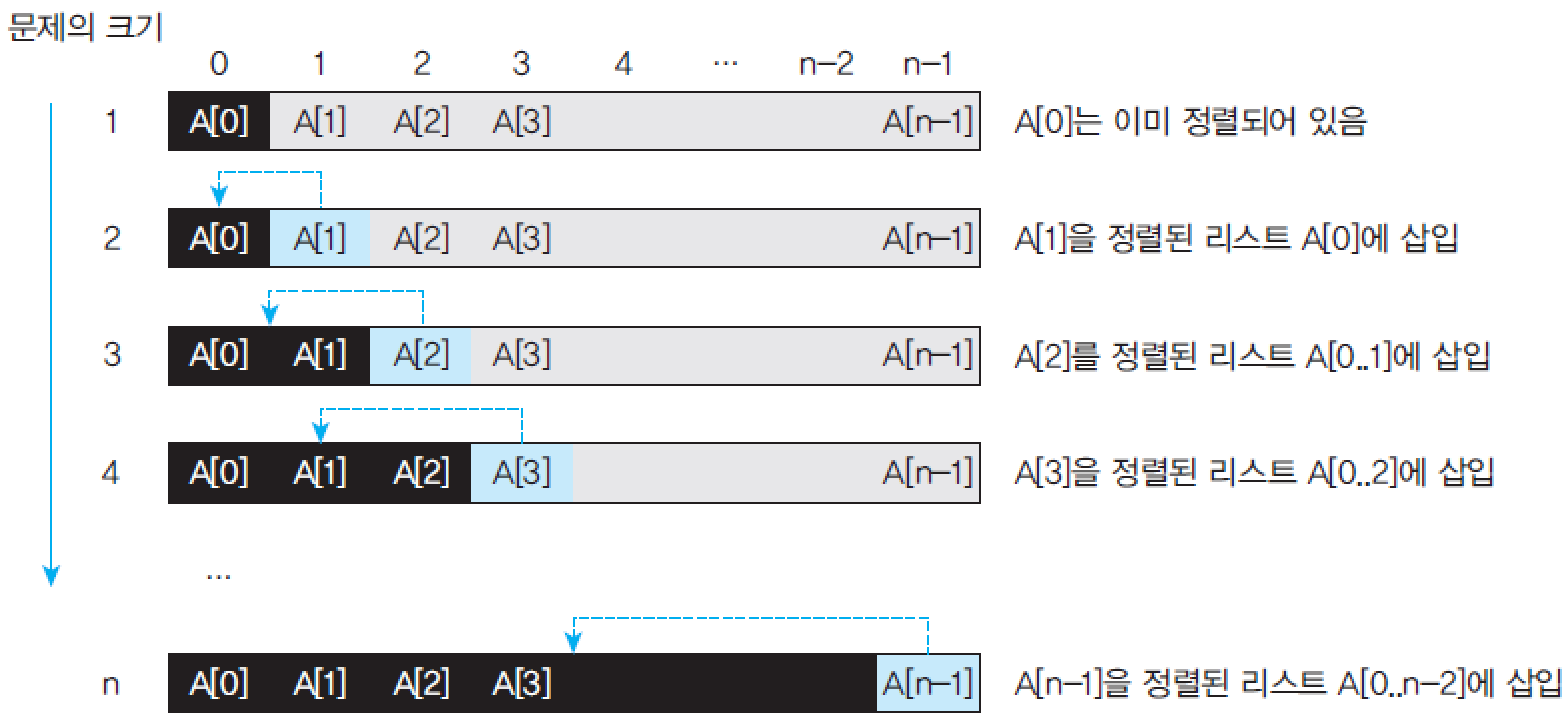
4.1 삽입 정렬(insertion sort)

- 정렬을 위한 축소 정복 전략
  - A[0..n-1]의 정렬 → A[0..n-2]의 정렬 + A[n-1]을 끼워 넣기



[그림 4.3] 정렬을 위한 축소 정복 전략

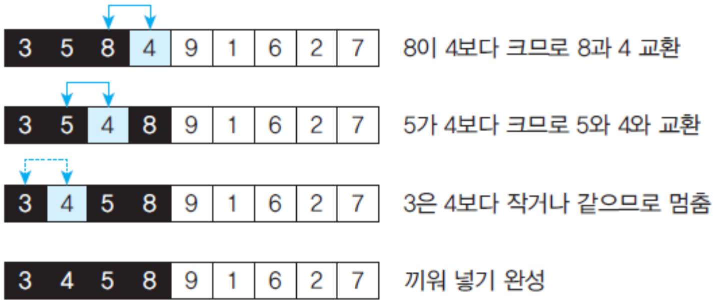
상향식 축소 정복 전략



[그림 4.4] 정렬을 위한 축소 정복 전략(상향식)

효율적인 끼워 넣기를 위한 삽입 위치 찾기

- 뒤에서 앞으로 넣을 위치를 찾아나감
  - 항목을 미리 뒤로 옮길 수 있음



[그림 4.7] 정렬된 부분 리스트에 항목을 끼워 넣는 과정

알고리즘

알고리즘 4.3 삽입 정렬

```
def insertion_sort(A) :
    n = len(A)
    for i in range(1, n) :
        key = A[i]
        j = i-1
        while j>=0 and A[j] > key :
            A[j + 1] = A[j]
            j = j - 1
        A[j + 1] = key
        printStep(A, i)
```

알고리즘 테스트 삽입 정렬

```
data = [ 5, 3, 8, 4, 9, 1, 6, 2, 7 ]
print("Original  : ", data)
insertion_sort(data)
print("Insertion : ", data)
```

C:\WINDOWS\system32\cmd.exe

```
Original  : [5, 3, 8, 4, 9, 1, 6, 2, 7]
  Step  1 = [3, 5, 8, 4, 9, 1, 6, 2, 7]
  Step  2 = [3, 5, 8, 4, 9, 1, 6, 2, 7]
  Step  3 = [3, 4, 5, 8, 9, 1, 6, 2, 7]
  Step  4 = [3, 4, 5, 8, 9, 1, 6, 2, 7]
  Step  5 = [1, 3, 4, 5, 8, 9, 6, 2, 7]
  Step  6 = [1, 3, 4, 5, 6, 8, 9, 2, 7]
  Step  7 = [1, 2, 3, 4, 5, 6, 8, 9, 7]
  Step  8 = [1, 2, 3, 4, 5, 6, 7, 8, 9]
Insertion : [1, 2, 3, 4, 5, 6, 7, 8, 9]
```

- 입력의 크기?
- 기본 연산?

복잡도 분석

- 입력 구성에 따른 차이가 있음
- 최선의 경우: $O(n)$
  - 정렬된 리스트?

$$T_{best}(n) = (n-1) \cdot 1 = n-1$$

- 최악의 경우: $O(n^2)$
  - 역으로 정렬된 리스트?

$$T_{worst}(n) = \sum_{i=1}^{n-1} \sum_{j=0}^{i-1} 1$$

$$= \sum_{i=1}^{n-1} i = \frac{n(n-1)}{2} \; O(n^2)$$

- 평균적인 경우 : $O(n^2)$
  - 최악의 경우의 절반 정도

$$T_{avg}(n) \approx \frac{n^2}{4} \approx \frac{1}{2} T_{worst}(n) \in O(n^2)$$

- 특징
  - 특히 많은 항목(레코드)을 이동이 필요
  - 레코드가 이미 정렬되어 있는 경우 효율적 → 셸(Shell) 정렬

Shell 정렬

- 정렬해야 할 리스트의 각 k번째 요소를 추출해서 부분 리스트를 만든다. 이때, k를 ‘간격(gap)'
- **간격의 초깃값: (정렬할 값의 수)/2**
- 생성된 부분 리스트의 개수는 gap과 같다.
  각 회전마다 간격 k를 절반으로 줄인다.
  간격은 홀수로 하는 것이 좋다. 간격을 절반으로 줄일 때 짝수가 나오면 +1을 해서 홀수로 만든다.
  간격 k가 1이 될 때까지 반복한다.

배열에 10, 8, 6, 20, 4, 3, 22, 1, 0, 15, 16이 저장되어 있다고 가정하고 자료를 오름차순으로 정렬해 보자.

초기상태

10	8	6	20	4	3	22	1	0	15	16

정렬할 값의 수: 10
간격(gap) k의 초깃값: 10/2 = 5

1

간격 k=5 일 때의 부분 리스트들

10					3					16
	8					22				
		6					1			
			20					0		
				4					15	

하나의 부분 리스트

간격 k=5 일 때의 부분 리스트를
각각 삽입 정렬로 정렬

3					10					16
	8					22				
		1					6			
			0					20		
				4					15	

1회전 결과

3	8	1	0	4	10	22	6	20	15	16

다음 k의 값: (5/2)+1 = 3

2

간격 k=3 일 때의 부분 리스트들

3			0			22			15	
	8			4			6			16
		1			10			20		

간격 k=3 일 때의 부분 리스트를
각각 삽입 정렬로 정렬

0			3			15			22	
	4			6			8			16
		1			10			20		

2회전 결과

0	4	1	3	6	10	15	8	20	22	16

다음 k의 값: 3/2 = 1

3

간격 k=1 일 때의 부분 리스트들

0	4	1	3	6	10	15	8	20	22	16

간격 k=1 일 때의 부분 리스트를
각각 삽입 정렬로 정렬

0	1	3	4	6	8	10	15	16	20	22

3회전 결과

0	1	3	4	6	8	10	15	16	20	22

오름차순
완성상태

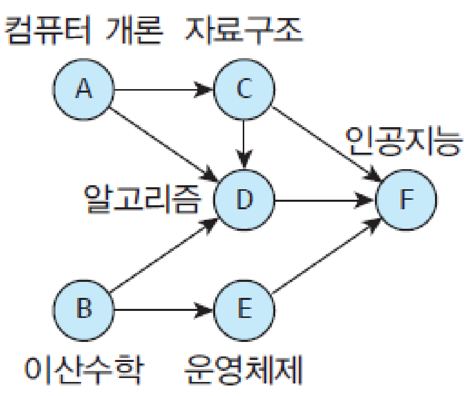
0	1	3	4	6	8	10	15	16	20	22

4.2 위상 정렬(방향그래프와 관련)

방향 그래프 G = (V, E)가 주어졌다. G에 존재하는 각 정점들의 선행 순서를 위배하지 않으면서 모든 정점들을 순서대로 나열하라.

과목 번호	과목명	선수과목
A	컴퓨터 개론	없음
B	이산수학	없음
C	자료구조	A
D	알고리즘	A, B, C
E	운영체제	B
F	인공지능	C, D, E

교과목의 선후수 관계표



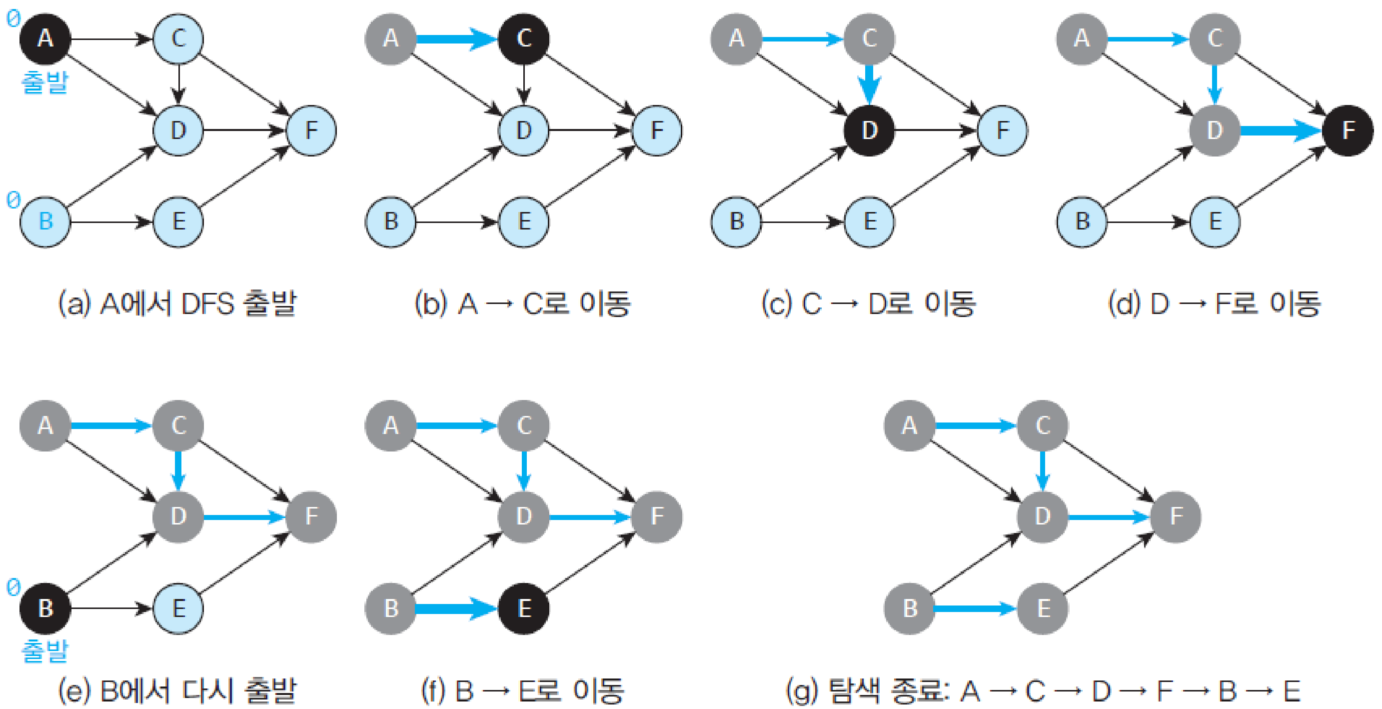
방향 그래프로 표시한 선후수 관계

- <A,B,C,D,E,F>, <B,A,C,D,E,F>, <A,C,B,E,D,F>: 가능한 수강 순서
- <C,A,B,D,E,F>: 가능하지 않은 수강 순서

모든 방향그래프가 위상정렬 가능X, 가능하려면 사이클이 존재X
사이클이 있으면 모든 정점(과목)이 선행정점을 갖게되고, 어떤 교과목도 수강X

DFS 기반 알고리즘(억지 기법)

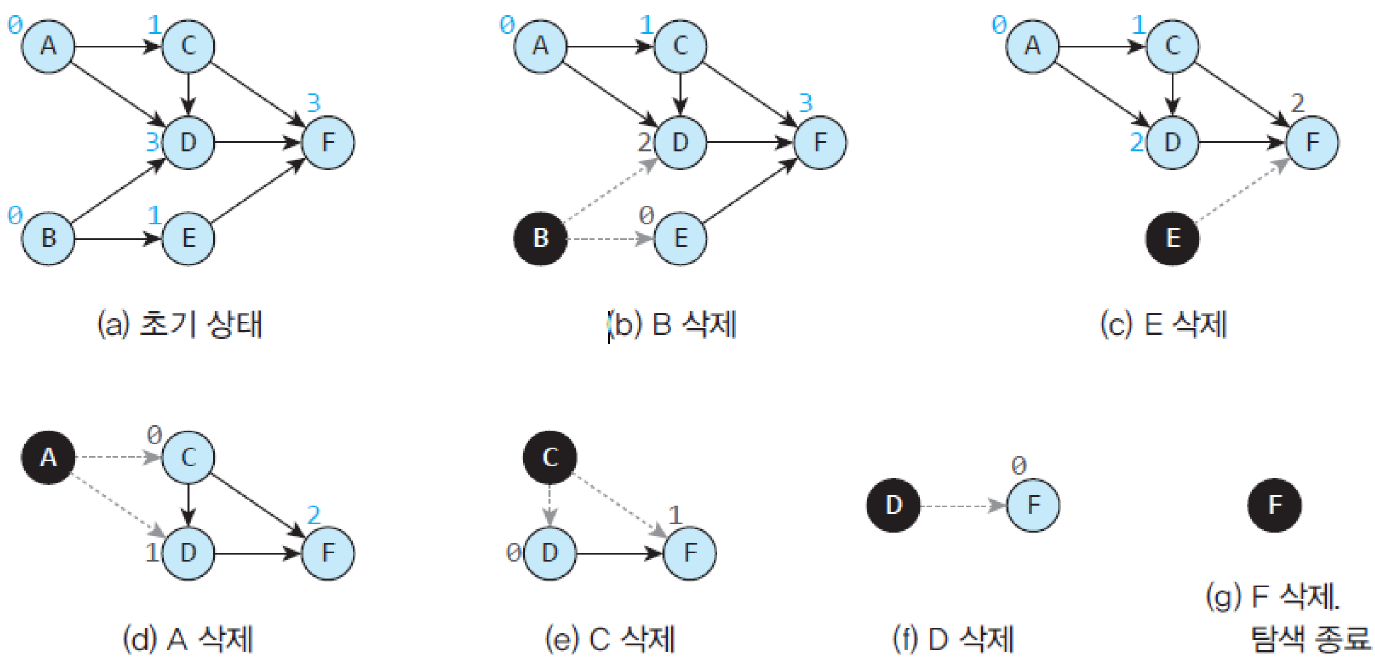
- 알고리즘
  - 진입 차수가 0인 임의의 정점을 시작 정점으로 선택
  - 깊이 우선 탐색으로 그래프의 정점들을 방문
  - 더 이상 갈 수 있는 정점이 없으면 다른 진입 차수가 0인 정점



[그림 4.9] DFS 기반 위상 정렬 과정의 예: A-C-D-F-B-E

축소 정복 기법의 알고리즘

- 알고리즘
  - 진입 차수가 0인 임의의 정점 선택(0인 정점 없다면 위상정렬X)
  - 선택된 정점을 삭제, 이 정점에서 진출하는 모든 간선들도 삭제, 차수 갱신, 이 과정을 반복 → 문제의 크기가 1 줄어듦



[그림 4.10] 위상 정렬 과정의 예: B-E-A-C-D-F

알고리즘

알고리즘 4.4 위상 정렬(축소 정복 기법)

```python
def topological_sort(graph) :
    inDeg = {}
    for v in graph :
        inDeg[v] = 0
    for v in graph :
        for u in graph[v]:
            inDeg[u] += 1

    vlist = []
    for v in graph :
        if inDeg[v]==0 :
            vlist.append(v)

    while vlist :
        v = vlist.pop()
        print(v, end=' ')

        for u in graph[v] :
            inDeg[u] -= 1
            if inDeg[u]==0 :
                vlist.append(u)
```

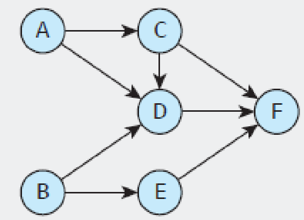
$O(n)$

$O(e)$

$O(n)$

$O(e)$

```python
mygraph = { "A" : {"C", "D"},
            "B" : {"D", "E"},
            "C" : {"D", "F"},
            "D" : {"F"},
            "E" : {"F"},
            "F" : {}
          }
```



알고리즘 테스트 위상 정렬

```python
mygraph = { "A" : {"C", "D"}, ... }
print('topological_sort: ')
topological_sort(mygraph)
print()
```

C:\WINDOWS\system32\cmd.exe

```
topological_sort:
B E A C D F
```

$O(n+e)$

4.3 이진 탐색

정렬되지 않은 리스트는 순차탐색이 답
정렬되어 있다면?

문제 4.2 정렬된 리스트에서의 탐색 문제

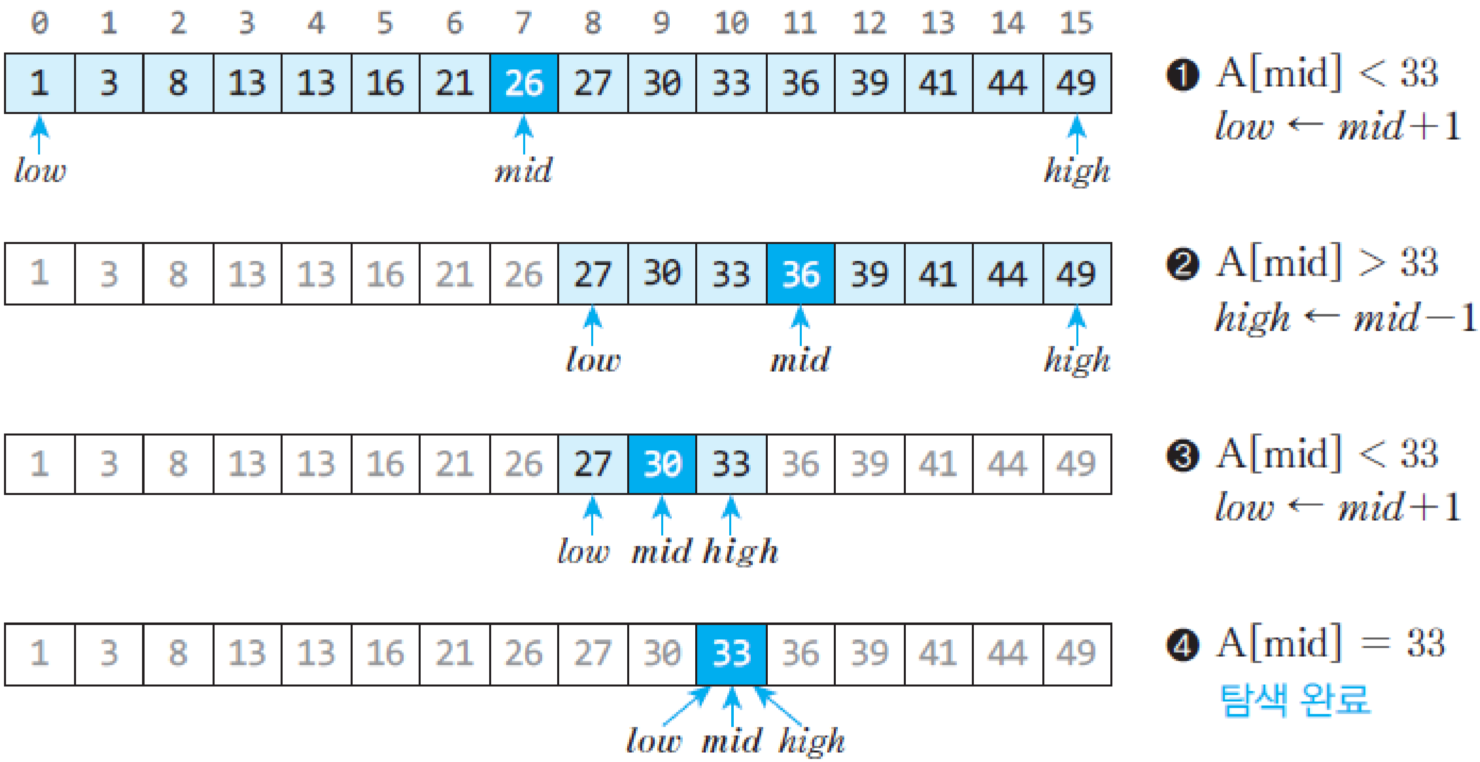
> 리스트에 n개의 항목이 들어있다. 이 리스트에서 "탐색키"를 가진 항목을 찾아라. 단, 리스트의 항목들은 정렬되어 있다.

- 축소 정복 탐색 전략
  - 중앙에 있는 항목을 먼저 조사

$$\underbrace{A[0]\ A[1]\ \cdots\ A[m-1]}_{\substack{A[m] < \text{key이면} \\ \text{답은 이 곳에 있음}}}\quad \underbrace{\overset{\substack{\text{Key} \\ \updownarrow}}{A[m]}}_{\substack{A[m] == \text{key이면} \\ \text{탐색 완료}}}\quad \underbrace{A[m+1]\ \cdots \le A[n-1]}_{\substack{A[m] > \text{key이면} \\ \text{답은 이 곳에 있음}}}$$

[그림 4.11] 이진 탐색의 문제 축소 과정

축소 정복 탐색 전략: 이진 탐색



[그림 4.12] 정렬된 리스트 A에서 33을 이진 탐색으로 찾는 과정

알고리즘

알고리즘 4.5 이진 탐색(순환구조)

```python
def binary_search(A, key, low, high) :
    if (low <= high) :
        mid = (low + high) // 2
        if key == A[mid] :
            return mid
        elif key < A[mid] :
            return binary_search(A, key, low, mid-1)
        else :
            return binary_search(A, key, mid+1, high)
    return -1
```

알고리즘 4.6 이진 탐색(반복구조)

```python
def binary_search_iter(A, key, low, high) :
    while (low <= high) :
        mid = (low + high) // 2
        if key == A[mid]:
            return mid
        elif key > A[mid]):
            low = mid + 1
        else:
            high = mid - 1
    return -1
```

알고리즘 테스트 순환과 반복 구조의 이진탐색 알고리즘

```python
listA = [1, 3, 8, 13, 13, 16, 21, 26, 27, 30, 33, 36, 39, 41, 44, 49]
print("입력 리스트 =", listA)   # 리스트는 반드시 정렬되어 있어야 한다.
print("33 탐색(순환) -->", binary_search(listA, 33, 0, len(listA)-1) )
print("33 탐색(반복) -->", binary_search_iter(listA, 33, 0, len(listA)-1) )
print("32 탐색(순환) -->", binary_search(listA, 32, 0, len(listA)-1) )
print("32 탐색(반복) -->", binary_search_iter(listA, 32, 0, len(listA)-1) )
```

C:\WINDOWS\system32\cmd.exe

```
입력 리스트 = [1, 3, 8, 13, 13, 16, 21, 26, 27, 30, 33, 36, 39, 41, 44, 49]
33 탐색(순환) --> 10
33 탐색(반복) --> 10
32 탐색(순환) --> -1
32 탐색(반복) --> -1
```

탐색 성공(항목의 인덱스)

탐색 실패(–1)

복잡도 분석

- 입력 구성에 따른 차이가 있음
  - $T_{best}(n) \in O(1)$
  - 최악: $T_{worst}(n) \in O(log_2 n)$
    - 리스트에 찾는 값이 없는 경우
    - $k = log_2 n$

$$
\begin{aligned}
T_{worst}(n) &= T_{worst}(\lfloor n/2 \rfloor) + 1 \\
&= T_{worst}(1) + k \\
&= 1 + k
\end{aligned}
$$

- 특징
  - 반드시 리스트가 정렬되어 있어야 함
  - 데이터의 삽입이나 삭제가 빈번한 응용에는 적합하지 않음

4.4 거듭제곱 문제

x의 n−거듭제곱인 x^n을 계산하라.

알고리즘 4.7 거듭제곱(억지 기법)

```
def slow_power(x, n) :
    result = 1.0
    for i in range(n):
        result = result * x
    return result
```

- 억지 기법

- 축소 정복 아이디어

$$
\begin{aligned}
x^{10} &= x \cdot x \cdot x \cdot x \cdot x \cdot x \cdot x \cdot x \cdot x \cdot x \\
&= (x \cdot x) \cdot (x \cdot x) \cdot (x \cdot x) \cdot (x \cdot x) \cdot (x \cdot x) \\
&= (x \cdot x)^5 = (x^2)^5 = y^5
\end{aligned}
$$

$$x^{10} = (x^2)^5 = x^2 \cdot (x^2)^4 = x^2 \cdot ((x^2)^2)^2$$

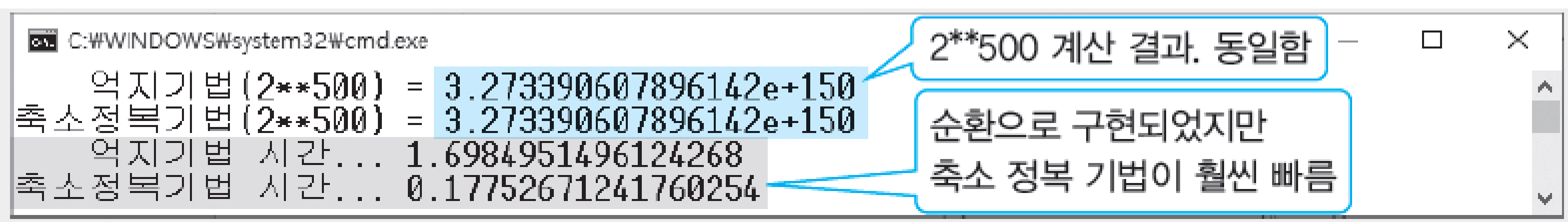
$$
x^n = \begin{cases} 1 & n = 0 \\ (x^2)^{n/2} & n\text{이 짝수} \\ x \cdot (x^2)^{(n-1)/2} & n\text{이 홀수} \end{cases}
$$

  – 곱셈의 수?

축소 정복 알고리즘

알고리즘 4.8 거듭제곱(축소 정복 기법)

```python
def power(x, n) :
    if n == 0 :
        return 1
    elif (n % 2) == 0 :
        return power(x*x, n//2)
    else :
        return x * power(x*x, (n-1)//2)
```



복잡도 분석

- 순환 호출을 한번 할 때마다 n이 크기가 절반씩 줄어듦
  - $n = 2^k$

$$2^k \rightarrow 2^{k-1} \rightarrow 2^{k-2} \rightarrow \cdots \rightarrow 2^1 \rightarrow 2^0$$

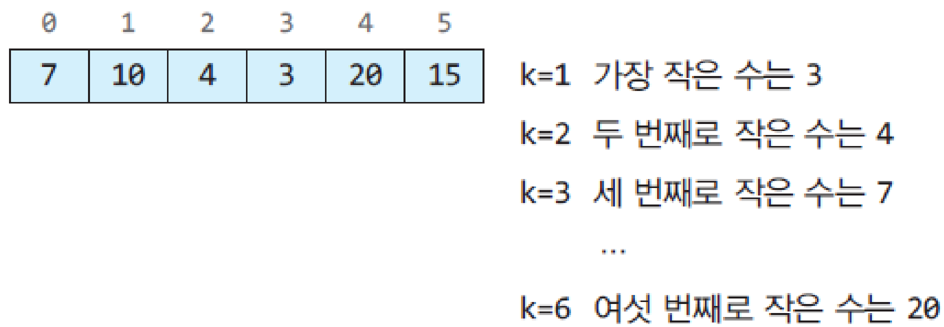
  - 복잡도: $O(log_2 n)$

- 행렬의 거듭제곱 문제
  - 동일한 적용이 가능

```python
def powerMat(x, n) :
    if n == 1 :
        return x
    elif (n % 2) == 0 :
        return powerMat(multMat(x,x), n // 2)
    else :
        return multMat(x, powerMat(multMat(x,x), (n - 1) // 2))
```

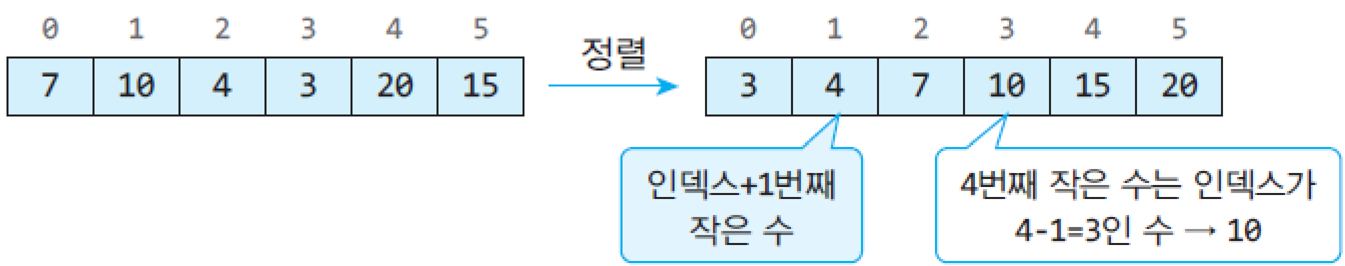
4.5 선택 문제: *k*번째 작은 수 찾기

리스트에서 *k*번째로 작은 항목을 찾아라. 단, 리스트는 정렬되어 있지 않다.



[그림 4.13] 리스트에서 *k*번째로 작은 수를 찾는 선택 문제의 예

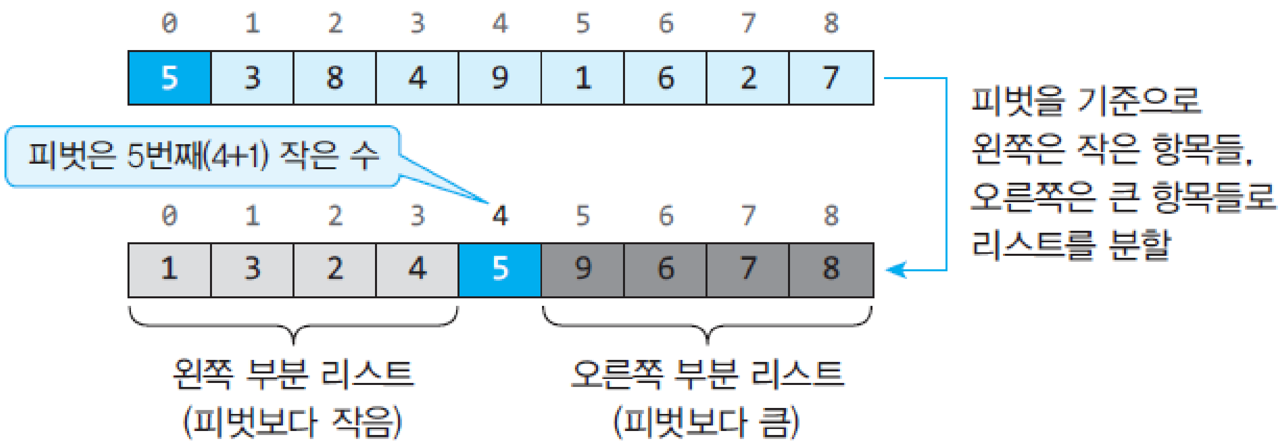
- 방법1: 정렬을 이용



- 방법2: 축소 정복 기법 (quick_select)

축소 정복을 이용한 k번째 작은 수 찾기

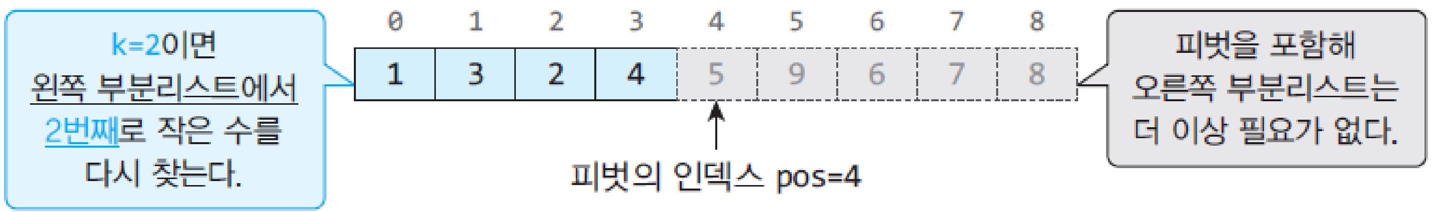
- 리스트의 한 항목을 **피벗(pivot)**으로 선택한다. 보통 리스트의 맨 왼쪽 항목을 선택한다.
- 리스트의 나머지 항목들을 모두 검사하여 피벗보다 작으면 피벗의 왼쪽으로 옮기고 피벗보다 크면 피벗의 오른쪽으로 옮긴다. 결과적으로 피벗을 중심으로 왼쪽은 피벗보다 작은 요소들로 구성되고, 오른쪽은 피벗보다 큰 요소들로 구성된다.



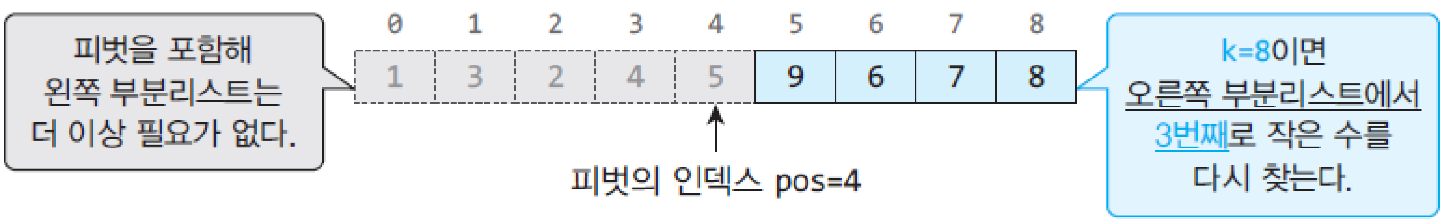
[그림 4.15] 첫 번째 원소로 피벗을 선택하고 피벗을 중심으로 리스트를 분할하는 예

세 가지 경우

- Case 1) $k == pos+1$: 답은 구해짐
- Case 2) $k < pos+1$: 답은 왼쪽 부분 리스트에



- Case3: $k > pos+1$: 답은 오른쪽 부분 리스트에



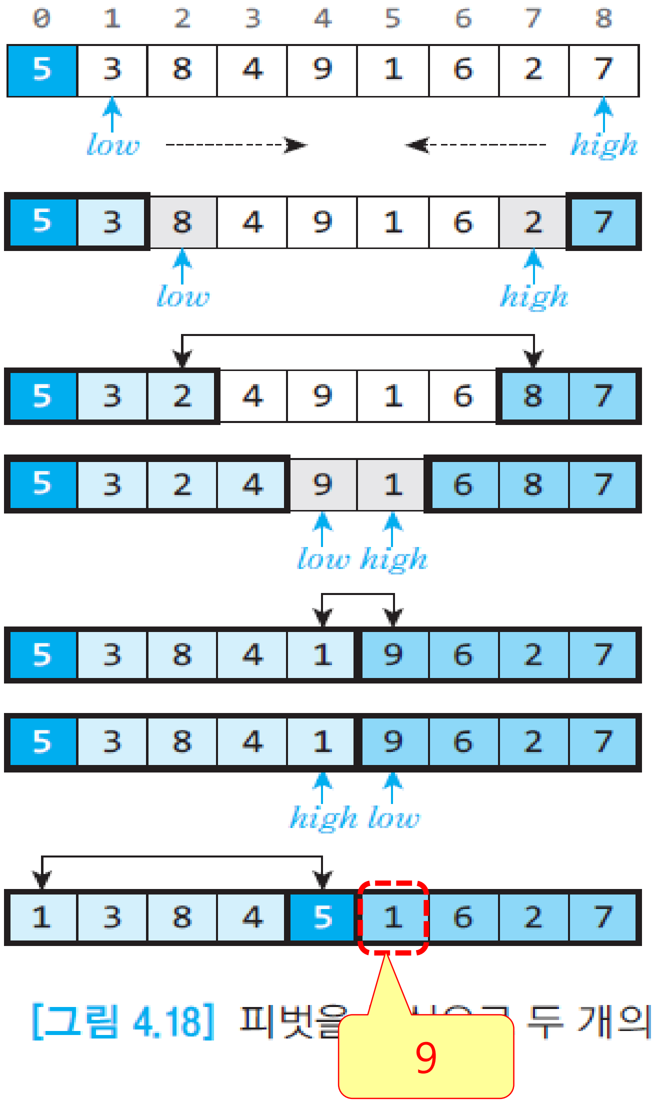
quick_select()알고리즘

알고리즘 4.11 축소 정복 기법을 이용한 *k*번째 작은 수 찾기

```
def quick_select(A, left, right, k):
    pos = partition(A, left, right)

    if (pos+1 == left+k):
        return A[pos]
    elif (pos+1 > left+k):
        return quick_select(A, left, pos-1, k)
    else :
        return quick_select(A, pos+1, right, k-(pos+1-left))
```

호어(Hoare) 분할



[그림 4.18] 피벗을 … 두 개의 리스트로 나누는 호어 분할 과정

partition(): 알고리즘

알고리즘 4.12 리스트 분할(Hoare Partition)

```
def partition(A, left, right) :
    low = left + 1                          # 왼쪽 부분 리스트의 인덱스 (증가방향)
    high = right                            # 오른쪽 부분 리스트의 인덱스 (감소방향)
    pivot = A[left]                         # 피벗 설정
    while (low <= high) :                   # low와 high가 역전되지 않는 한 반복
        while low <= right and A[low] < pivot : low += 1
        while high >= left and A[high]> pivot : high-= 1

        if low < high :                     # 선택된 두 레코드 교환
            A[low], A[high] = A[high], A[low]

    A[left], A[high] = A[high], A[left]     # 마지막으로 high와 피벗 항목 교환
    return high                             # 피벗의 위치 반환
```

<=

복잡도 분석

- 입력 구성에 따른 차이가 있음
  - 최선
    - 한번의 분할로 답을 찾는 경우
    - $T_{best}(n) \in O(n)$
  - 최악
    - 불균등 분할이 계속 되는 경우
    - $T_{worst}(n) \in O(n^2)$
  - 평균
    - 가정이 필요
    - $T_{avg}(n) \in O(n)$
      - 증명: 172~173쪽 심화학습

- quick_select()
  - 가변적인 크기(variable size decrease)로 문제의 크기가 줄어드는 전형적인 축소 정복 알고리즘

4.6 축소 정복 기법의 추가적인 예

- 위조 동전 찾기: 고정 비율 축소

동일하게 생긴 n개의 동전과 양팔 저울이 주어졌다. 동전들 중에서 하나는 가짜 동전이고 진짜 동전보다 약간 가볍다. 당연히 양팔 저울은 동전의 무게를 직접 측정할 수 없다. 양쪽의 무게가 같은지 또는 어느 쪽이 더 무거운지 만을 알 수 있을 뿐이다. 이 저울을 최소한의 횟수만 사용하여 어느 동전이 가짜 동전인지를 찾아라.

  – 고정 비율 축소(1/2)

$$T(n) = T(\lfloor n/2 \rfloor) + 1 \qquad \text{for } n > 1$$

$$T(n) = \lfloor \log_2 n \rfloor \in O(\log_2 n)$$

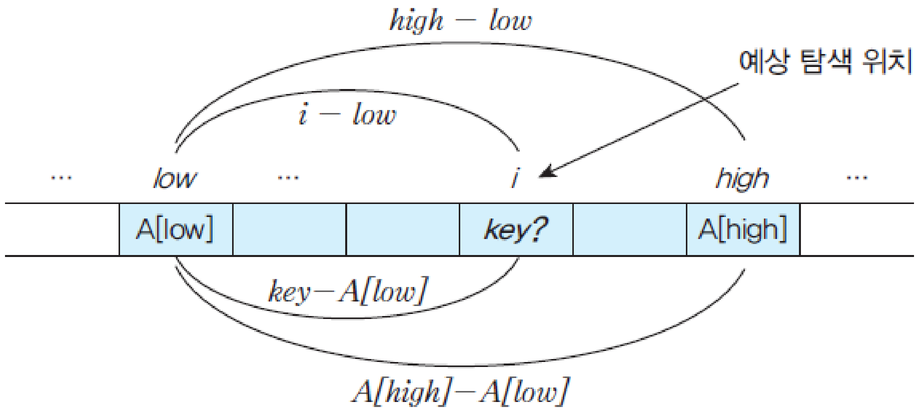
  – 고정 비율 축소(1/3)

$$T(n) = T(\lfloor n/3 \rfloor) + 1 \qquad \text{for } n > 1$$

$$O(k) = O(\log_3 n)$$

보간 탐색: 가변 크기 축소

- 사전에서 단어를 찾을 때와 같이 탐색키가 존재할 위치를 예측하여 탐색하는 방법



[그림 4.19] 보간 탐색은 찾는 값과 위치가 비례한다고 가정한다.

$$탐색위치 = low + (high - low) \cdot \frac{key - A[low]}{A[high] - A[low]}$$

```
mid = int(low + (high-low) * (key-A[low].key) / (A[high].key-A[low].key))
```

실습 과제

감사합니다!