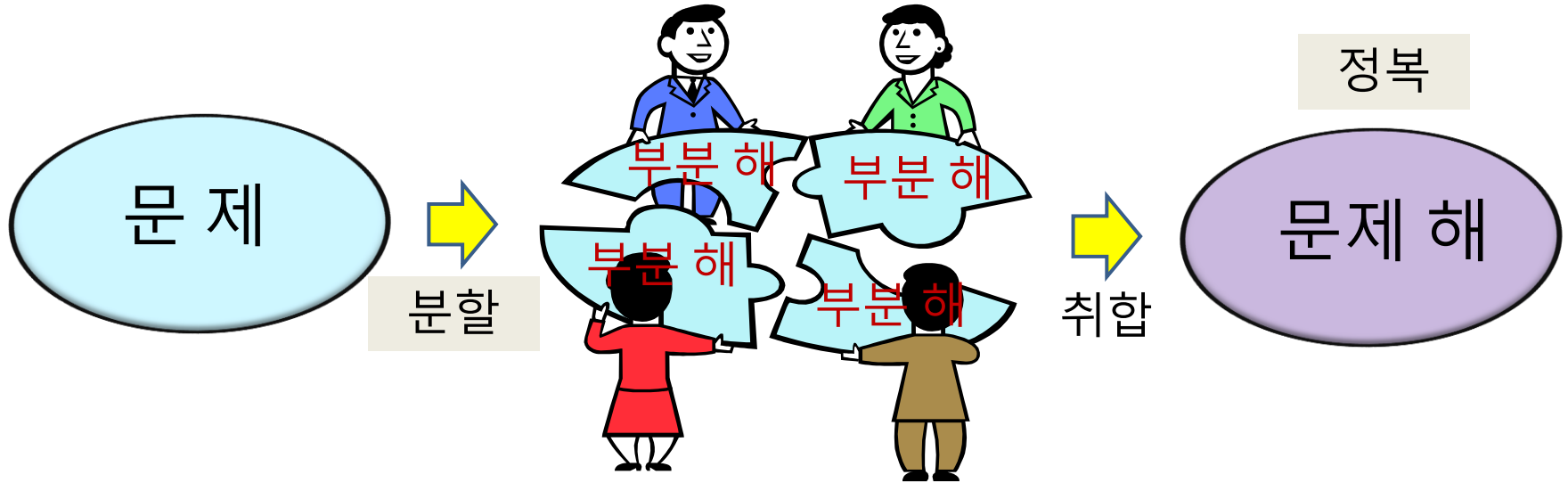


제3장 분할 정복 알고리즘

분할 정복 (Divide-and-Conquer) 알고리즘

- 주어진 문제의 입력을 분할하여 문제를 해결 (정복)하는 방식의 알고리즘
- 분할한 입력에 대하여 동일한 알고리즘을 적용하여 해를 계산하며, 이들의 해를 취합하여 원래 문제의 해를 얻는다.
- 분할된 입력에 대한 문제를 부분문제 (subproblem)라고 하고, 부분 문제의 해를 부분해라고 한다.
- 부분문제는 더 이상 분할할 수 없을 때까지 계속 분할한다.
 - 또는 충분히 간단한 문제가 될 때까지

부분 문제



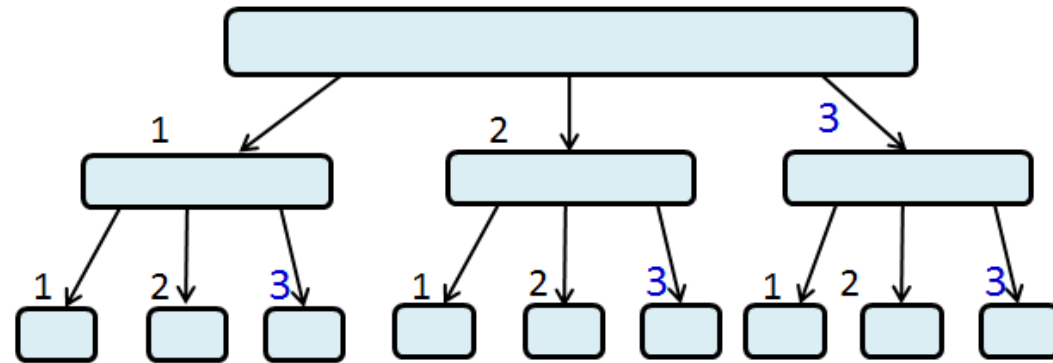
- 크기가 n 인 입력을 **3개로 분할**하고, 각각 분할된 부분 문제의 크기가 **$n/2$** 라고 하면, 아래의 그림처럼 문제가 분할된다.

입력 크기

n

$\frac{n}{2}$

$\frac{n}{2^2}$



⋮

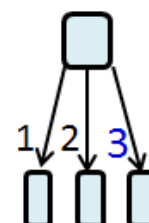
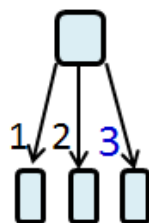
⋮

⋮

$\frac{n}{2^{k-1}}$

...

$\frac{n}{2^k}$

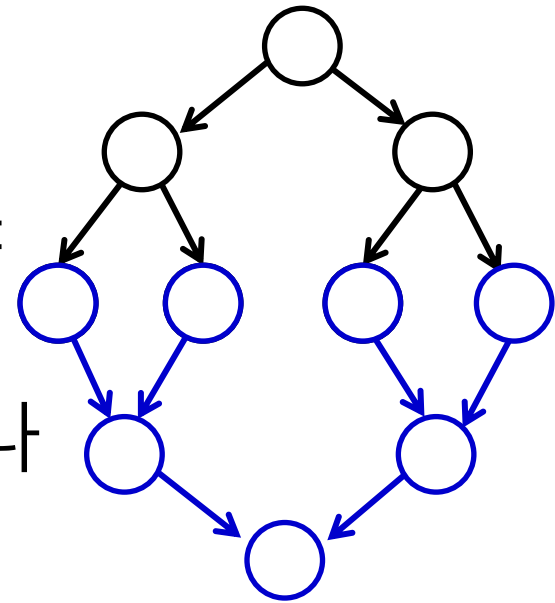


- 입력 크기가 n 일 때 총 몇 번 분할하여야 더 이상 분할할 수 없는 크기인 1이 될까?
- 답을 계산하기 위해서 총 분할한 횟수 = k 라고 하자.
- 1번 분할 후 각각의 입력 크기가 $n/2$
- 2번 분할 후 각각의 입력 크기가 $n/2^2$
- ...
- k 번 분할 후 각각의 입력 크기가 $n/2^k$
- 따라서 $n/2^k = 1$ 일 때 더 이상 분할할 수 없다.
- $k = \log_2 n$ 이다.

정복 과정

- 대부분의 분할 정복 알고리즘은 문제의 입력을 단순히 분할만 해서는 해를 구할 수 없다.
- 따라서 분할된 부분 문제들을 정복해야 한다. 즉, 부분해를 찾아야 한다.
- 정복하는 방법은 문제에 따라 다르나 일반적으로 부분 문제들의 해를 취합하여 보다 큰 부분 문제의 해를 구한다.

분할 과정



정복 (취합) 과정

분할 정복 알고리즘의 분류

- 문제가 a 개로 분할되고, 부분 문제의 크기가 $1/b$ 로 감소하는 알고리즘:
 - $a=b=2$ 인 경우 합병 정렬 (3.1절), 최근접 점의 쌍 찾기 (3.4절), 공제선 문제 (연습문제 25)
 - $a=3, b=2$ 큰 정수의 곱셈 (연습문제 21)
 - $a=4, b=2$ 큰 정수의 곱셈 (연습문제 20)
 - $a=7, b=2$ 인 경우, 스트라센(Strassen)의 행렬 곱셈 알고리즘 (연습문제 22)

- 문제가 2개로 분할, 부분문제의 크기가 일정하지 않은 크기로 감소하는 알고리즘: **퀵 정렬** (3.2절)
- 문제가 2개로 분할, 그 중에 1개의 부분 문제는 고려할 필요 없으며, 부분 문제의 크기가 $1/2$ 로 감소하는 알고리즘: **이진탐색** (1.2절)
- 문제가 2개로 분할, 그 중에 1개의 부분문제는 고려할 필요 없으며, 부분문제의 크기가 일정하지 않은 크기로 감소하는 알고리즘: **선택 문제 알고리즘** (3.3절)
- 부분문제의 크기가 1, 2개씩 감소하는 알고리즘: **삽입 정렬** (6.3절), **피보나치 수** (3.5절) 등

3.1 합병 정렬

- 합병 정렬 (Merge Sort)은 입력이 2개의 부분 문제로 분할, 부분문제의 크기가 $1/2$ 로 감소하는 분할 정복 알고리즘
- n 개의 숫자들을 $n/2$ 개씩 2개의 부분문제로 분할하고, 각각의 부분문제를 재귀적으로 합병 정렬한 후, 2개의 정렬된 부분을 합병하여 정렬 (정복)한다.
- 합병 과정이 (문제를) 정복하는 것이다.

합병 (merge)

- 2개의 각각 정렬된 숫자들을 1개의 정렬된 숫자들로 합치는 것

배열 A: 6 14 18 20 29

⇒ 배열 C: 1 2 6 14 15 18 20 25 29 30 45

배열 B: 1 2 15 25 30 45

합병 정렬 알고리즘

MergeSort(A, p, q)

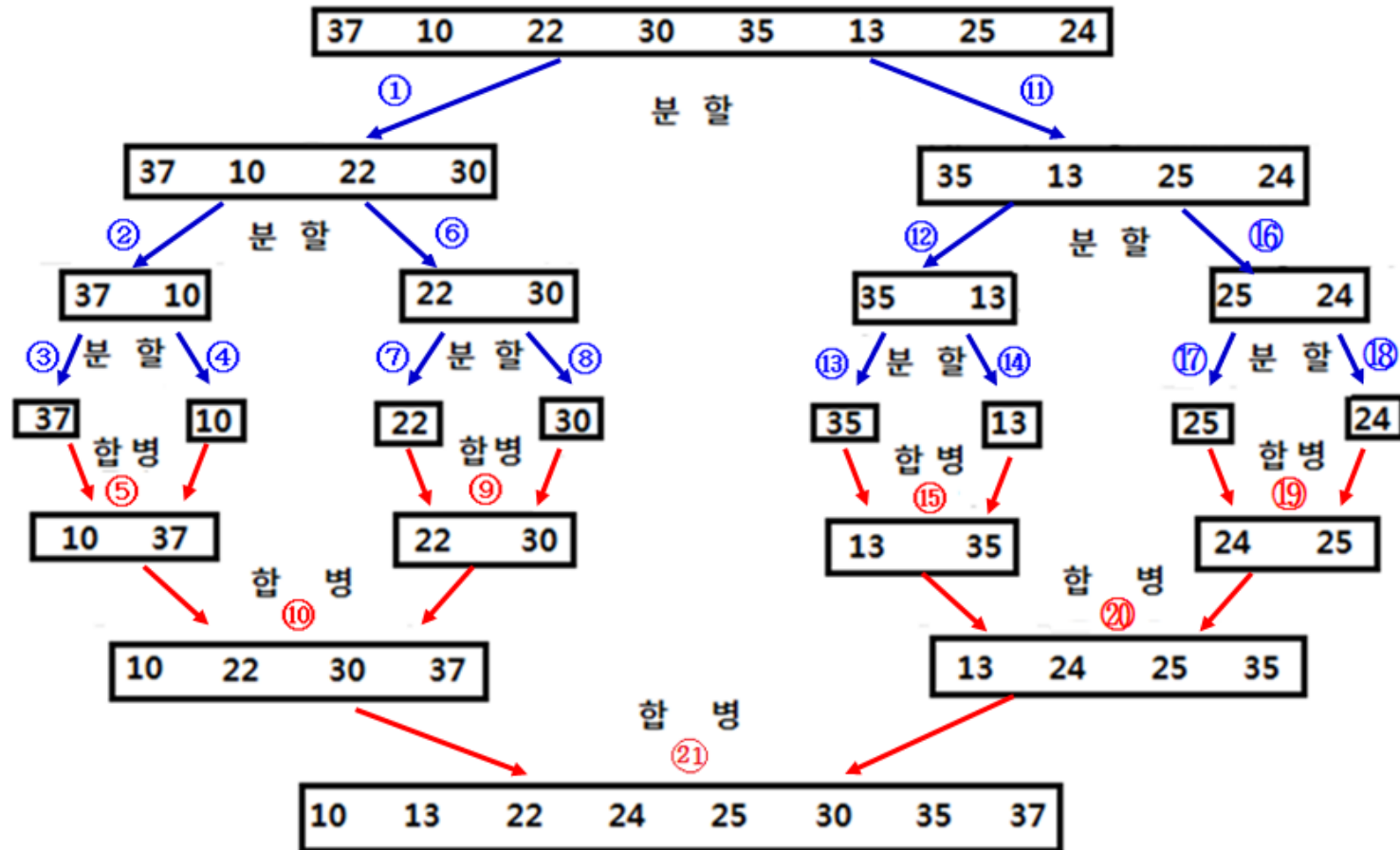
입력: $A[p] \sim A[q]$

출력: 정렬된 $A[p] \sim A[q]$

1. if ($p < q$) { // 배열의 원소의 수가 2개 이상이면
2. $k = \lfloor (p + q) / 2 \rfloor$ // k: 반으로 나누기 위한 중간 원소의 인덱스
3. **MergeSort**(A, p, k) // 앞부분 재귀 호출
4. **MergeSort**(A, k + 1, q) // 뒷부분 재귀 호출
5. $A[p] \sim A[k]$ 와 $A[k + 1] \sim A[q]$ 를 합병한다.
- }

- Line 1에서는 정렬할 부분의 원소의 수 (n)가 2개 이상 일 때에만 다음 단계 수행.
 - 만일 $n=1$ 이면, 그 자체로 정렬된 것이므로 어떤 수행할 필요 없이 이전 호출했던 곳으로 리턴
- Line 2에서는 정렬할 부분의 원소들을 $1/2$ 로 나누기 위해, $k = \lfloor (p + q) / 2 \rfloor$ 를 계산. 즉, 원소의 수가 홀수인 경우에는 k 는 소수점 이하는 버림
- Line 3 ~ 4에서는 MergeSort(A, p, k)와 MergeSort($A, k + 1, q$)를 재귀 호출하여 각각 정렬
- Line 5에서는 line 3 ~ 4에서 각각 정렬된 부분을 임시 배열에 합병한다.
 - 합병 과정의 마지막에는 임시 배열에 있는 합병된 원소들을 배열 A 로 복사.
 - 즉, 임시 배열 $B[p] \sim B[q]$ 를 $A[p] \sim A[q]$ 로 복사.

- 입력 크기가 $n = 8$ 인 배열 $A = [37, 10, 22, 30, 35, 13, 25, 24]$ 에 대하여



함수 merge() code

```
def merge(arr, p, k, q):  
    c = []  
    i = p  
    j = k+1  
    while i <= k and j <= q:  
        if arr[i] < arr[j]:  
            c.append(arr[i])  
            i += 1  
        elif arr[i] > arr[j]:  
            c.append(arr[j])  
            j += 1  
        else:  
            c.append(arr[i])  
            c.append(arr[j])  
            i += 1  
            j += 1
```

```
    while i <= k:  
        c.append(arr[i])  
        i += 1  
    while j <= q:  
        c.append(arr[j])  
        j += 1  
    j = 0  
    for i in range(p, q+1):  
        arr[i] = c[j]  
        j += 1
```

test code

```
def merge(arr, p, k, q):
```

```
    ...
```

```
alist = [2, 4, 6, 9, 1, 3, 6, 7]
```

```
merge(alist, 0, 3, 7)
```

```
print(alist)
```

함수 mergeSort()와 test code

```
def mergeSort(arr, p, q):
```

```
    if p < q:
```

```
        k = (p + q)//2
```

```
        mergeSort(arr, p, k)
```

```
        mergeSort(arr, k+1, q)
```

```
        merge(arr, p, k, q)
```

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

```
mergeSort(alist, 0, 8)
```

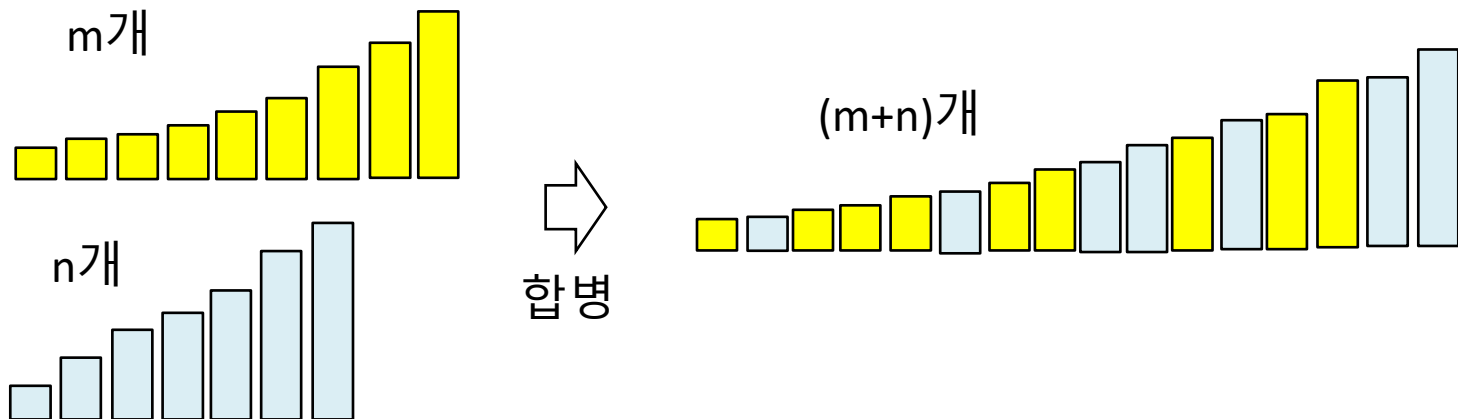
```
print(alist)
```


연습 문제 (작은 과제 2-3)

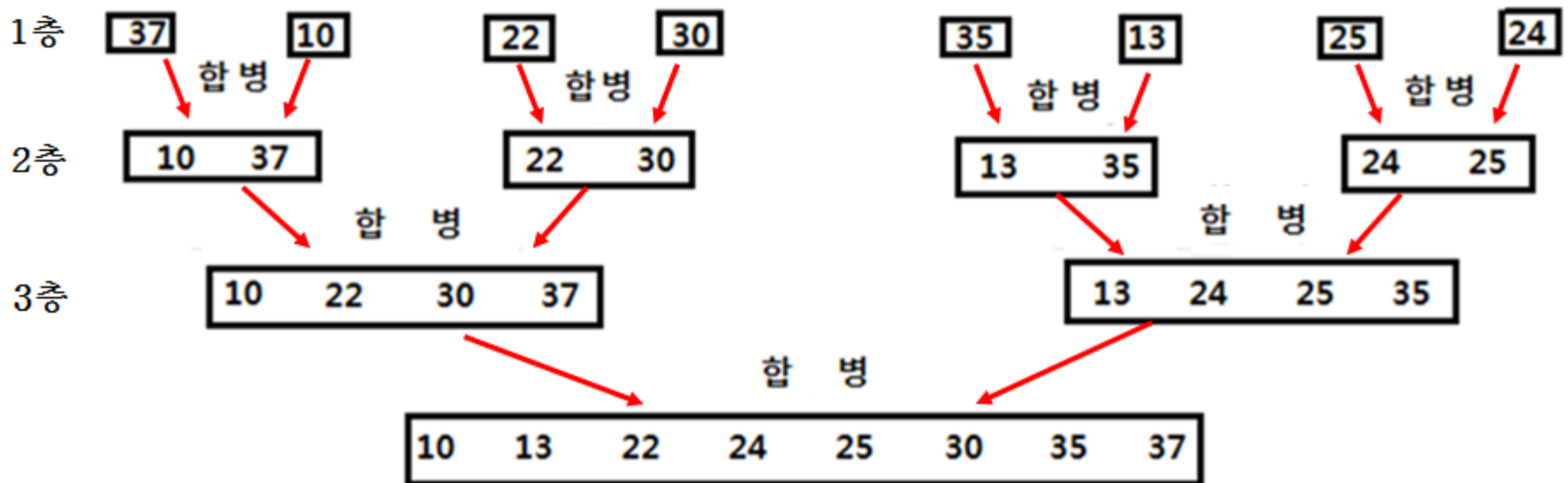
- 입력 배열 [6, 1, 3, 9, 4, 5, 8, 2, 7]에 대해 앞의 13쪽 그림과 같이 합병 정렬을 수행하시오.
 - 그림을 통해 합병 정렬 과정을 보이시오.
 - 종이에 작성했으면 스마트폰으로 찍어, 스냅샷을 제출

시간복잡도

- 분할하는 부분은 배열의 중간 인덱스 계산과 2번의 재귀 호출이므로 $O(1)$ 시간 소요
- 합병의 수행 시간은 입력의 크기에 비례. 즉, 2개의 정렬된 배열 A와 B의 크기가 각각 n 과 m 이라면, 최대 비교 횟수 = $(n + m - 1)$.



- 각 층을 살펴보면 모든 숫자(즉, $n=8$ 개의 숫자)가 합병에 참여
- 합병은 입력 크기에 비례하므로 각 층에서 수행된 비교 횟수는 $O(n)$



- 층수를 세어보면, 8개의 숫자를 반으로, 반의 반으로, 반의 반의 반으로 나눈다.
- 이 과정을 통하여 3층이 만들어진다.

입력 크기	예	층
n	8	
$n/2$	4	1층
$n/4 = n/2^2$	2	2층
$n/8 = n/2^3$	1	3층

- 입력의 크기가 n 일 때 몇 개의 층이 만들어질까?
- n 을 계속하여 $1/2$ 로 나누다가, 더 이상 나눌 수 없는 크기인 1 이 될 때 분할을 중단한다.
- 따라서 k 번 $1/2$ 로 분할했으면 k 개의 층이 생기는 것이고, k 는 $n=2^k$ 으로 부터 $\log_2 n$ 임을 알 수 있다.
- 합병 정렬의 시간복잡도:
(층수) $\times O(n) = \log_2 n \times O(n) = O(n \log n)$

합병 정렬의 단점

- 합병 정렬의 **공간 복잡도: $O(n)$**
- 입력을 위한 메모리 공간 (입력 배열)외에 추가로 입력과 같은 크기의 공간 (임시 배열)이 별도로 필요.
- 2개의 정렬된 부분을 하나로 합병하기 위해, 합병된 결과를 저장할 곳이 필요하기 때문

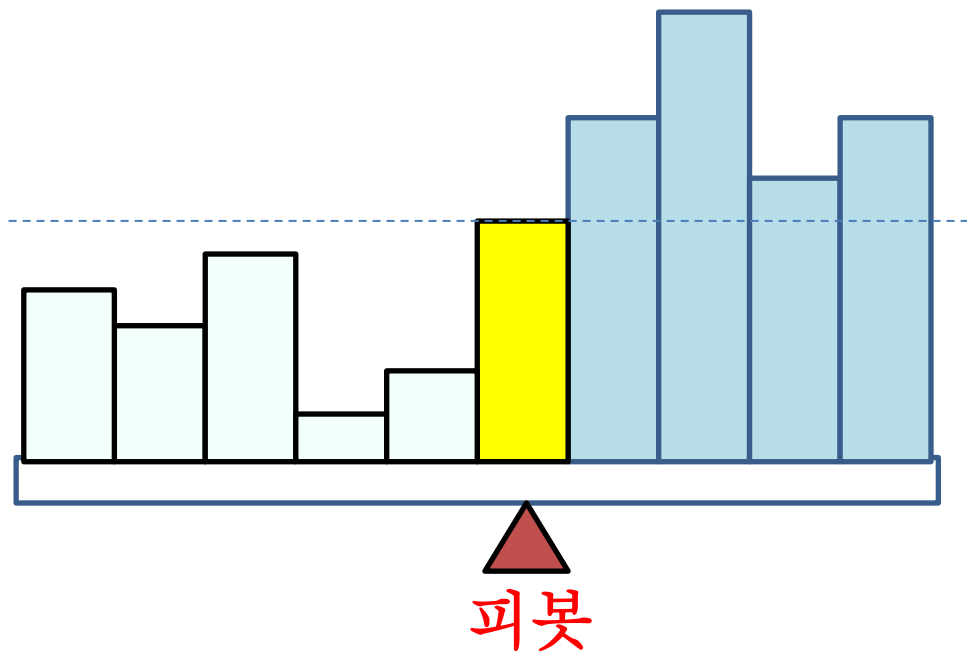
응용

- 합병 정렬은 외부정렬의 기본이 되는 정렬 알고리즘이다.
- 연결 리스트에 있는 데이터를 정렬할 때에도 퀵 정렬이나 힙 정렬 보다 훨씬 효율적이다.
- 멀티코어 (Multi-Core) CPU와 다수의 프로세서로 구성된 그래픽 처리 장치 (Graphic Processing Unit)의 등장으로 정렬 알고리즘을 병렬화하는 데에 합병 정렬 알고리즘이 활용

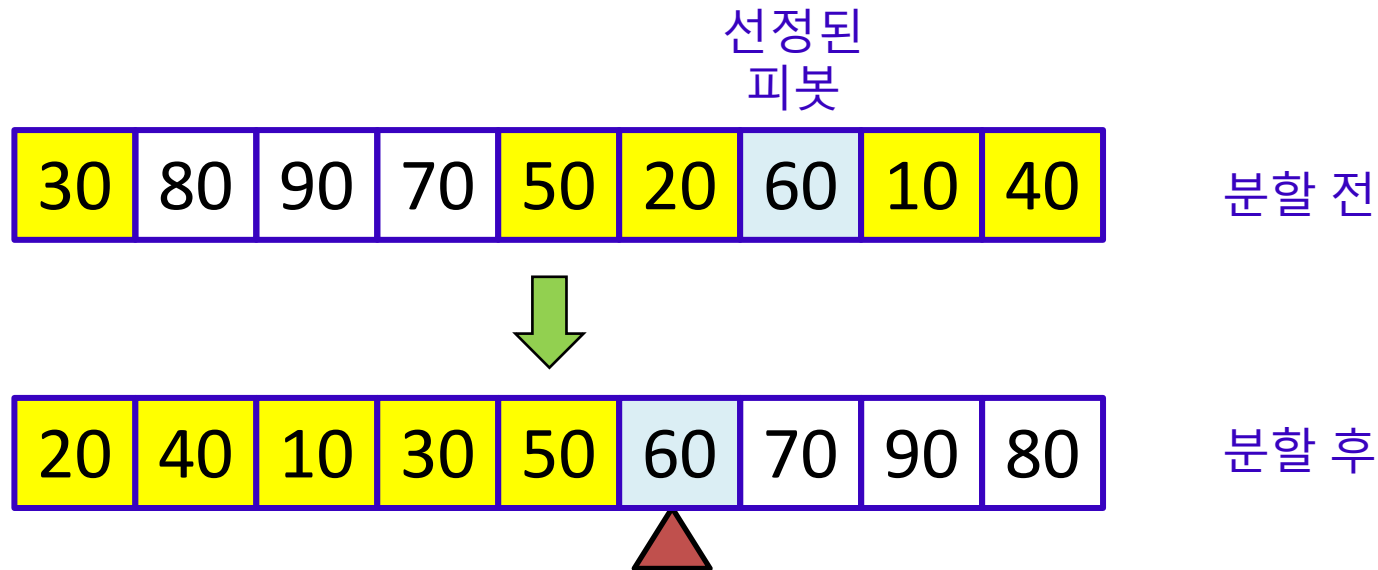
3.2 퀵 정렬

- 퀵 정렬 (Quick Sort)은 분할 정복 알고리즘으로 분류되나, 사실 알고리즘이 수행되는 과정을 살펴보면 **정복 후 분할**하는 알고리즘이다.
- 퀵 정렬 알고리즘은 문제를 2개의 부분문제로 분할하는데, 각 부분문제의 크기가 일정하지 않은 형태의 분할 정복 알고리즘

- 퀵 정렬은 **피벗 (pivot)**이라 일컫는 배열의 원소(숫자)를 기준으로 피벗보다 작은 숫자들은 왼편으로, 피벗보다 큰 숫자들은 오른편에 위치하도록 분할하고, 피벗을 그 사이에 놓는다.
- 퀵 정렬은 분할된 부분문제들에 대하여서도 위와 동일한 과정을 재귀적으로 수행하여 정렬



- 피벗은 분할된 왼편이나 오른편 부분에 포함되지 않는다. 피벗이 60이라면, 60은 [20 40 10 30 50]과 [70 90 80] 사이에 위치한다.



퀵 정렬 알고리즘

QuickSort(A, left, right)

입력: 배열 $A[\text{left}] \sim A[\text{right}]$

출력: 정렬된 배열 $A[\text{left}] \sim A[\text{right}]$

1. if (left < right) {
2. 피벗을 $A[\text{left}] \sim A[\text{right}]$ 중에서 선택하고, 피벗을 $A[\text{left}]$ 와 자리를 바꾼 후, 피벗과 배열의 각 원소를 비교하여 피벗보다 작은 숫자들은 $A[\text{left}] \sim A[p-1]$ 로 옮기고, 피벗보다 큰 숫자들은 $A[p+1] \sim A[\text{right}]$ 로 옮기며, 피벗은 $A[p]$ 에 놓는다.
3. **QuickSort**(A, left, p-1) // 피벗보다 작은 그룹
4. **QuickSort**(A, p+1, right) // 피벗보다 큰 그룹
- }

- Line 1에서는 배열 A 의 가장 왼쪽 원소의 인덱스 ($left$)가 가장 오른쪽 원소의 인덱스($right$)보다 작으면, line 2~4에서 정렬을 수행한다. 만일 그렇지 않으면 1개의 원소를 정렬하는 경우. 1개의 원소는 그 자체가 이미 정렬되어 있으므로, line 2~4의 정렬 과정을 수행할 필요 없이 그대로 호출을 마친다.
- Line 2에서는 $A[left] \sim A[right]$ 에서 피벗을 선택하고, 배열 $A[left+1] \sim A[right]$ 의 원소들을 피벗과 비교하여, 피벗보다 작은 그룹인 $A[left] \sim A[p-1]$ 과 피벗보다 큰 그룹인 $A[p+1] \sim A[right]$ 로 분할하고 $A[p]$ 에 피벗을 위치시킨다. 즉, p 는 피벗이 위치한 배열 A 의 인덱스이다.

- Line 3에서는 피벗보다 작은 그룹인 $A[\text{left}] \sim A[p-1]$ 을 재귀적으로 호출한다.
- Line 4에서는 피벗보다 큰 숫자들은 $A[p+1] \sim A[\text{right}]$ 를 재귀적으로 호출한다.

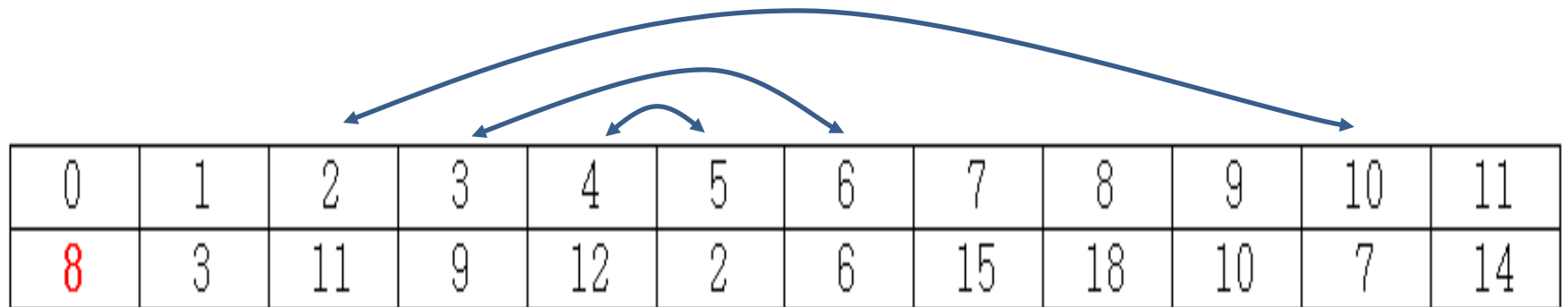
QuickSort(A,0,11) 호출

0	1	2	3	4	5	6	7	8	9	10	11
6	3	11	9	12	2	8	15	18	10	7	14

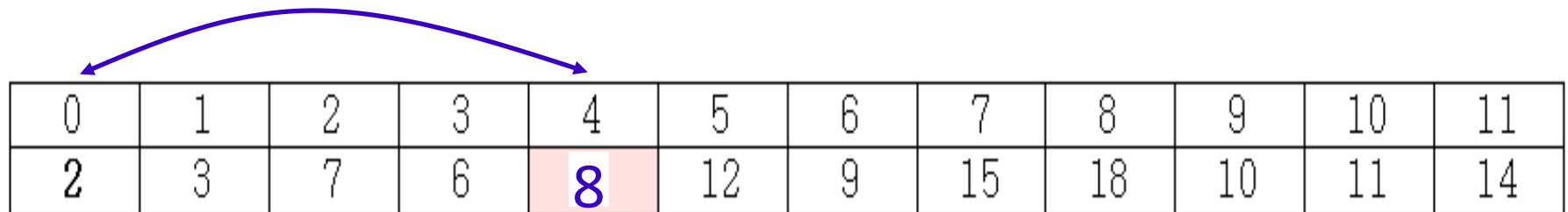
피벗 $A[6]=8$ 이라면, line 2에서 아래와 같이 차례로 원소들의 자리를 바꾼다. 먼저 피벗을 가장 왼쪽으로 이동시킨다.

0	1	2	3	4	5	6	7	8	9	10	11
8	3	11	9	12	2	6	15	18	10	7	14

- 그 다음엔 피벗보다 큰 수와 피벗보다 작은 수를 다음과 같이 각각 교환



0	1	2	3	4	5	6	7	8	9	10	11
8	3	11	9	12	2	6	15	18	10	7	14



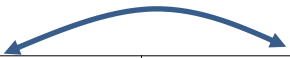
0	1	2	3	4	5	6	7	8	9	10	11
2	3	7	6	8	12	9	15	18	10	11	14

- line 3에서 $\text{QuickSort}(A, 0, 4-1) = \text{QuickSort}(A, 0, 3)$ 이 호출되고, 그 다음 line 4에서 $\text{QuickSort}(A, 4+1, 11) = \text{QuickSort}(A, 5, 11)$ 이 호출


QuickSort(A,0,3) 호출

0	1	2	3
2	3	7	6

- 피벗 $A[3]=6$ 이라면, line 2에서 아래와 같이 원소들의 자리를 바꾼다.



0	1	2	3
6	3	7	2



0	1	2	3
2	3	6	7

- line 3에서 $\text{QuickSort}(A,0,2-1) = \text{QuickSort}(A,0,1)$ 이 호출되고, 그 다음 line 4에서 $\text{QuickSort}(A,2+1,3) = \text{QuickSort}(A,3,3)$ 이 호출


QuickSort(A,0,1) 호출

0	1
2	3

- 피벗 $A[1]=3$ 이라면, line 2에서 아래와 같이 원소들의 자리를 바꾼다.

0	1
3	2

0	1
2	3



- line 3에서 QuickSort(A,0,1-1) = QuickSort(A,0,0)이 호출되고, line 4에서 QuickSort(A,1+1,1) = QuickSort(A,2,1)이 호출
- QuickSort(A,0,0) 호출: Line 1의 if-조건이 '거짓'이 되어서 알고리즘을 더 이상 수행하지 않는다.
- QuickSort(A,2,1) 호출: Line 1의 if-조건이 '거짓'이므로 알고리즘을 수행하지 않는다.
- 위의 과정과 유사하게 A[2]~A[3]도 정렬되며, QuickSort(A,0,3)이 아래와 같이 완성된다.

0	1	2	3
2	3	7	6

Quicksort Python code

```
import random

def partition(A, left, right):
    p = random.randint(left, right)
    if p != left:
        A[left], A[p] = A[p], A[left]
    i = left+1
    j = right
    done = False
    while not done:
        while i <= j and A[i] <= A[left]: i += 1
        while j >= i and A[j] >= A[left] : j -= 1
        if i > j:
            done = True
        else:
            A[i], A[j] = A[j], A[i]
    A[left], A[j] = A[j], A[left]
    return j
```

```
def qsort(A, left, right):
    if left < right:
        p = partition(A, left, right)
        qsort(A, left, p-1)
        qsort(A, p+1, right)

A = [6, 3, 11, 9, 12, 2, 8, 15, 18, 10, 7, 14]
qsort(A, 0, len(A)-1)
print(A)
```

Better Python Quicksort code

```
import random

def qsort(A):
    less = []
    equal = []
    greater = []
    if len(A) > 1:
        pivot = A[random.randrange(0, len(A))]
        for x in A:
            if x < pivot:
                less.append(x)
            elif x == pivot:
                equal.append(x)
            else:
                greater.append(x)
        return qsort(less) + equal + qsort(greater)
    else:
        return A
```

```
A = [6, 3, 11, 9, 12, 2, 8, 15, 18, 10, 7, 14]
B = qsort(A)
print(B)
```

시간복잡도

- 퀵 정렬의 성능은 피벗 선택이 좌우한다. 피벗으로 가장 작은 숫자 또는 가장 큰 숫자가 선택되면, 한 부분으로 치우치는 분할을 야기한다.

피벗

1	17	42	9	18	23	31	11	26
---	----	----	---	----	----	----	----	----



1	9	42	17	18	23	31	11	26
---	---	----	----	----	----	----	----	----



...

1	9	11	17	18	23	26	31	42
---	---	----	----	----	----	----	----	----

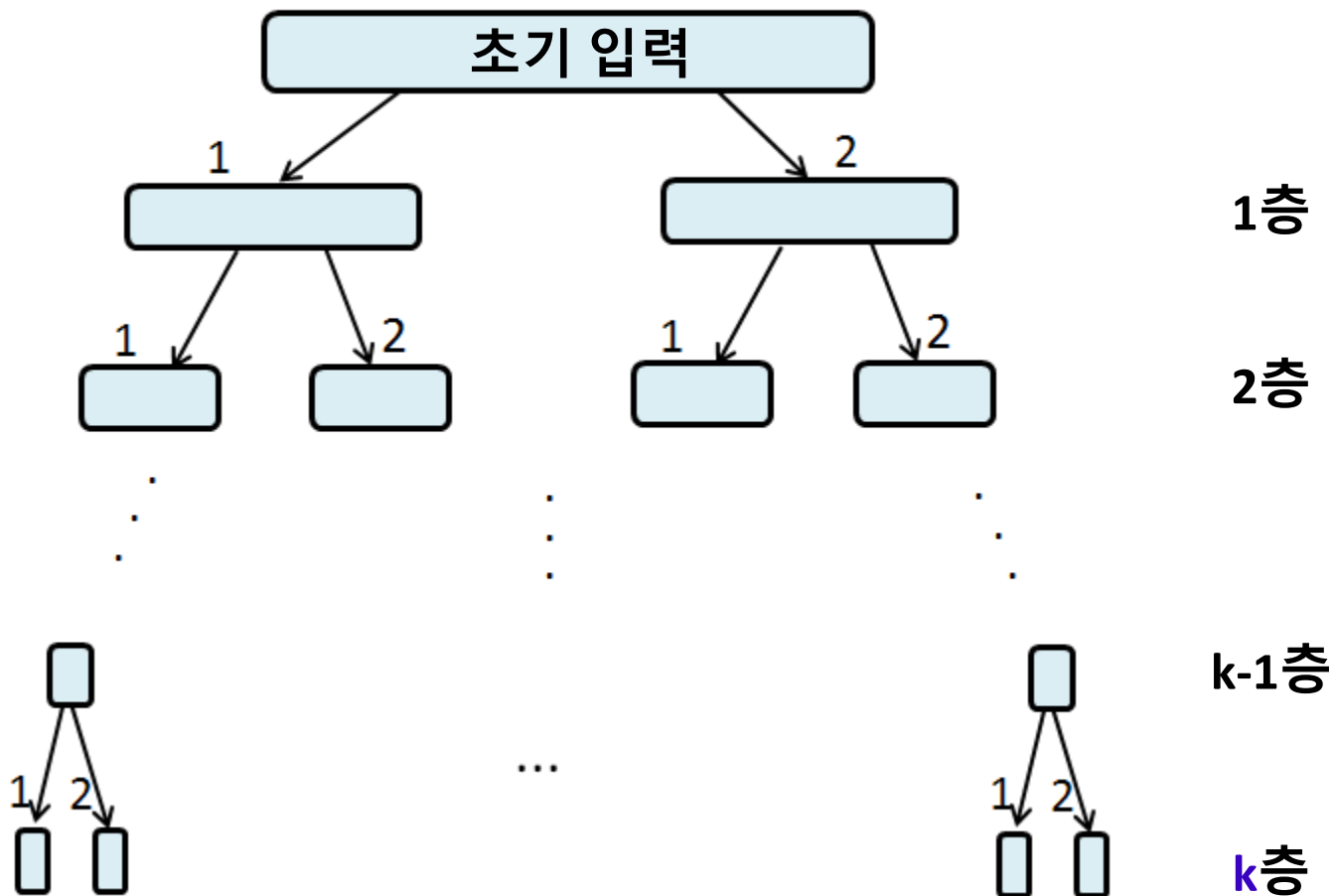


- 피벗=1일 때: 8회 - [17 42 9 18 23 31 11 26]과 각각 1회씩 비교
- 피벗=9일 때: 7회 - [42 17 18 23 31 11 26]과 각각 1회씩 비교
- 피벗=11일 때: 6회 - [17 18 23 31 42 26]과 각각 1회씩 비교
- ...
- 피벗=31일 때: 1회 - [42]와 1회 비교
- 총 비교 횟수는 $8+7+6+\dots+1 = 36$ 이다.
- 입력의 크기가 n 이라면, 퀵 정렬의 최악 경우 시간 복잡도 = $(n-1)+(n-2)+(n-3)+\dots+2+1 = n(n-1)/2 = O(n^2)$

최선 경우의 분할

각 부분의 크기

n

$$\frac{n}{2}$$
$$\frac{n}{2^2}$$
$$\frac{n}{2^{k-1}}$$
$$\frac{n}{2^k} = 1$$


- 각 층에서는 각각의 원소가 각 부분의 피봇과 1회씩 비교된다. 따라서 비교 횟수 = $O(n)$
- 총 비교 횟수 = $O(n) \times (\text{층수}) = O(n) \times (\log_2 n)$
- 층수가 $\log_2 n$ 인 이유는 $n/2^k = 1$ 일 때 $k = \log_2 n$ 이기 때문
- 퀵 정렬의 **최선 경우 시간복잡도: $O(n \log_2 n)$**

평균 경우 시간복잡도

- 피벗을 항상 랜덤하게 선택한다고 가정하면, 퀵정렬의 평균 경우 시간복잡도를 계산할 수 있다. 이때의 시간복잡도도 역시 최선 경우와 동일하게 $O(n\log_2 n)$ 이다.

피벗 선정 방법

- 랜덤하게 선정하는 방법
 - 3 숫자의 중앙값으로 선정하는 방법: 가장 왼쪽 숫자, 중간 숫자, 가장 오른쪽 숫자 중에서 중앙값으로 피벗을 정한다. 아래의 예제를 보면, 31, 1, 26 중에서 중앙값인 26을 피벗으로 사용한다.

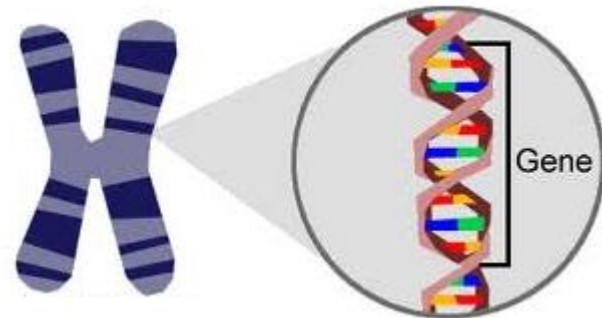
31	17	42	9	1	23	18	11	26
----	----	----	---	---	----	----	----	----

성능 향상 방법

- 입력의 크기가 매우 클 때, 쿼 정렬의 성능을 더 향상시키기 위해서, **삽입 정렬**이 동시에 사용
- 입력의 크기가 작을 때에는 쿼 정렬이 삽입 정렬보다 빠르지만은 않다. 왜냐하면 쿼 정렬은 재귀 호출로 수행되기 때문
- 부분문제의 크기가 작아지면 (예를 들어, 25 ~ 50이 되면), 더 이상의 분할(재귀 호출)을 중단하고 삽입 정렬을 사용하는 것이다.

응 용

- 켜 정렬은 커다란 크기의 입력에 대해서 가장 좋은 성능을 보이는 정렬 알고리즘이다.
- 켜 정렬은 실질적으로 어느 정렬 알고리즘보다 좋은 성능을 보인다.
- 생물 정보 공학(Bioinformatics)에서 특정 유전자를 효율적으로 찾는데 접미 배열(suffix array)과 함께 켜 정렬이 활용된다.

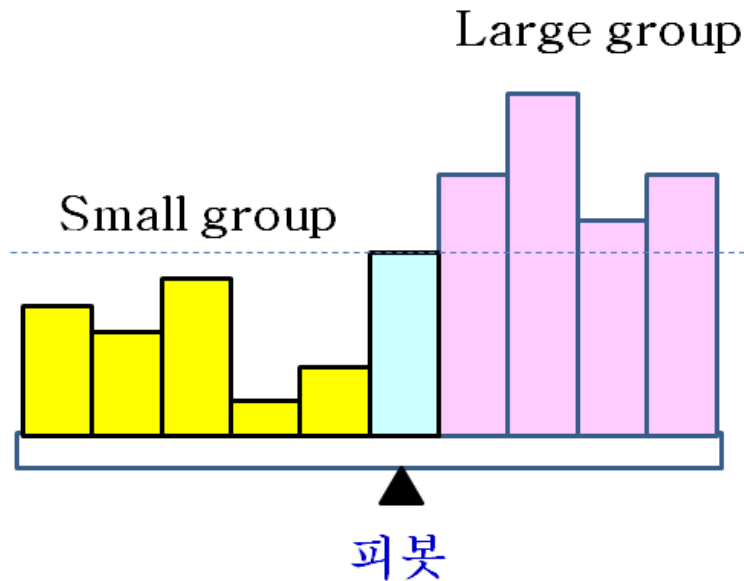


3.3 선택 문제

- 선택 (Selection) 문제는 n 개의 숫자들 중에서 k 번째로 작은 숫자를 찾는 문제
- 선택 문제를 해결하기 위한 간단한 방법은
 - 최소 숫자를 k 번 찾는다. 단, 최소 숫자를 찾은 뒤에는 입력에서 최소 숫자를 제거한다.
 - 숫자들을 정렬한 후, k 번째 숫자를 찾는다.
- 위의 알고리즘들은 각각 최악의 경우 $O(kn)$ 과 $O(n\log n)$ 의 수행 시간이 걸린다.

아이디어

- **이진탐색**은 정렬된 입력의 중간에 있는 숫자와 찾고자 하는 숫자를 비교함으로써, 입력을 $1/2$ 로 나눈 두 부분 중에서 한 부분만을 검색
- 선택 문제는 입력이 정렬되어 있지 않으므로, 입력 숫자들 중에서 (퀵 정렬에서와 같이) **피벗**을 선택하여 아래와 같이 분할



- Small group은 피봇보다 작은 숫자의 그룹이고, Large group은 피봇보다 큰 숫자의 그룹이다.
- 분할 후, 각 그룹의 크기를 알아야, k 번째 작은 숫자가 어느 그룹에 있는지를 알 수 있고, 그 다음엔 그 그룹에서 몇 번째로 작은 숫자를 찾아야 하는지를 알 수 있다.

- Small group에 k 번째 작은 숫자가 속한 경우: k 번째 작은 숫자를 Small group에서 찾는다.
- Large group에 k 번째 작은 숫자가 있는 경우: $(k - |\text{Small group}| - 1)$ 번째로 작은 숫자를 Large group에서 찾아야 한다. 여기서 $|\text{Small group}|$ 은 Small group에 있는 숫자의 개수이고, 1은 피봇에 해당된다.

선택 문제 알고리즘

Selection(A, left, right, k)

입력: $A[\text{left}] \sim A[\text{right}]$ 와 k, 단, $1 \leq k \leq |A|$, $|A| = \text{right} - \text{left} + 1$

출력: $A[\text{left}] \sim A[\text{right}]$ 에서 k 번째 작은 원소

1. 피벗을 $A[\text{left}] \sim A[\text{right}]$ 에서 랜덤하게 선택하고, 피벗과 $A[\text{left}]$ 의 자리를 바꾼 후, 피벗과 배열의 각 원소를 비교하여 피벗보다 작은 숫자는 $A[\text{left}] \sim A[p-1]$ 로 옮기고, 피벗보다 큰 숫자는 $A[p+1] \sim A[\text{right}]$ 로 옮기며, 피벗은 $A[p]$ 에 놓는다.
2. $S = (p-1) - \text{left} + 1$ // S = Small group의 크기
3. if ($k \leq S$) **Selection**(A, left, p-1, k) // Small group에서 찾기
4. else if ($k = S + 1$) return $A[p]$ // 피벗 = k번째 작은 숫자
5. else **Selection**(A, p+1, right, k-S-1) // large group에서 찾기

- Line 1은 피벗을 랜덤하게 선택하는 것을 제외하고는 퀵 정렬 알고리즘의 line 2와 동일
- Line 2에서는 입력을 두 그룹으로 분할된 후, $A[p]$ 가 피벗이 있는 곳이기 때문에 Small group의 크기를 알 수 있다. 즉, Small group의 가장 오른쪽 원소의 인덱스가 $(p-1)$ 이므로, Small group의 크기 $S = (p-1) - \text{left} + 1$ 이다.
- Line 3은 k 번째 작은 수가 Small group에 속한 경우이므로 $\text{Selection}(A, \text{left}, p-1, k)$ 호출
- Line 4는 k 번째 작은 수가 피벗인 $A[p]$ 와 같은 경우이므로 해 발견

- Line 5에서는 k 번째 작은 수가 Large group에 속한 경우이므로 $\text{Selection}(A, p+1, \text{right}, k-S-1)$ 호출
- 이때에는 $(k-S-1)$ 번째 작은 수를 Large group에서 찾아야; 왜냐하면 피벗이 k 번째 작은 수보다 작고, S 는 Small group의 크기이므로

Selection 알고리즘의 수행 과정

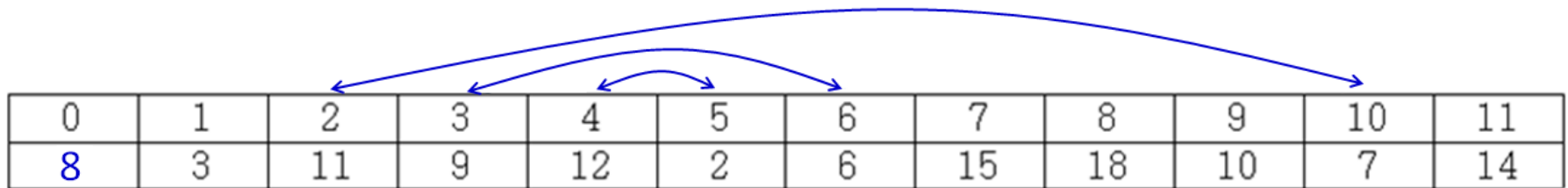
- $k=7$

0	1	2	3	4	5	6	7	8	9	10	11
6	3	11	9	12	2	8	15	18	10	7	14

- 최초로 Selection(A,0,11,7) 호출
- $k=7$, left=0, right=11
- Line 1에서 피벗 $A[6]=8$ 이라면, 피벗이 $A[0]$ 에 오도록 $A[0]$ 과 $A[6]$ 을 서로 바꾼다.

0	1	2	3	4	5	6	7	8	9	10	11
8	3	11	9	12	2	6	15	18	10	7	14

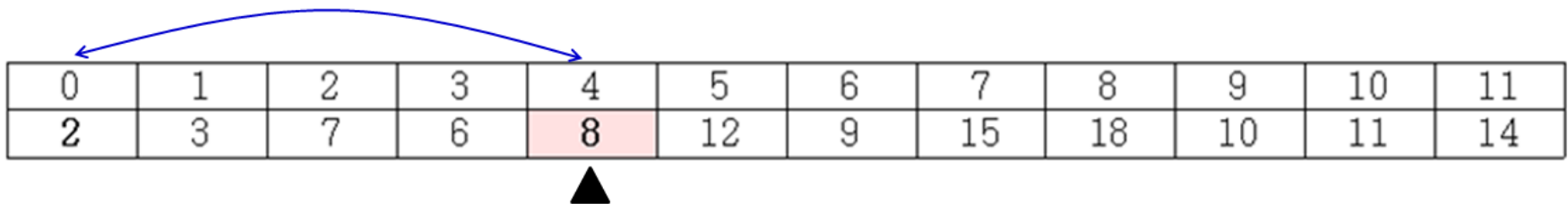
- 아래와 같이 차례로 원소들이 자리를 서로 바꾼다.



0	1	2	3	4	5	6	7	8	9	10	11
8	3	11	9	12	2	6	15	18	10	7	14



0	1	2	3	4	5	6	7	8	9	10	11
8	3	7	6	2	12	9	15	18	10	11	14



0	1	2	3	4	5	6	7	8	9	10	11
2	3	7	6	8	12	9	15	18	10	11	14

- Line 2: Small group의 크기 계산 (즉, $S = (p-1)-left+1 = (4-1)-0+1 = 4$)
- 따라서 Small group에는 7 번째 작은 수가 없고, line 4의 if-조건 $(7=S+1) = (7=4+1) = (7=5)$ 가 '거짓'이 되어, line 5에서 $Selection(A,p+1,right,k-S-1) = Selection(A,4+1,11,7-4-1) = Selection(A,5,11,2)$ 호출
- 즉, Large group에서 2 번째로 작은 수를 찾는다.


- Selection(A,5,11,2) 호출
- k=2, left=5, right=11

5	6	7	8	9	10	11
12	9	15	18	10	11	14

- Line 1에서 피벗 A[11]=14라면, 피벗이 A[5]에 오도록 A[5]와 A[11]을 서로 바꾼다.


5	6	7	8	9	10	11
14	9	15	18	10	11	12

- 그 다음에는 아래와 같이 차례로 원소들이 자리를 서로 바꾼다.




5	6	7	8	9	10	11
14	9	15	18	10	11	12

5	6	7	8	9	10	11
14	9	12	11	10	18	15



5	6	7	8	9	10	11
10	9	12	11	14	18	15

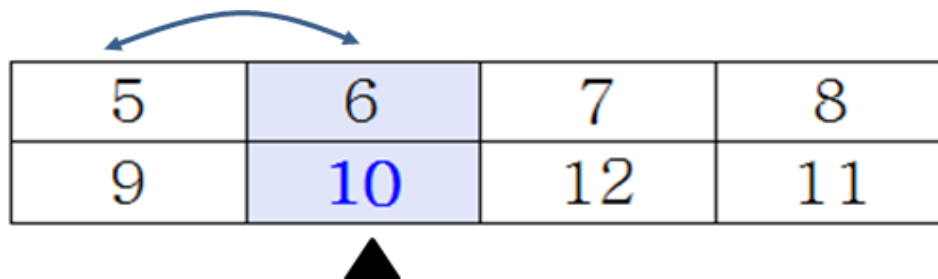


- Line 2: Small group의 크기 계산 (즉, $S = (p-1) - \text{left} + 1 = (9-1) - 5 + 1 = 4$)
- 따라서 $k=2$ 번째 작은 수를 찾아야 하므로 line 3의 if-조건인 $(k \leq S) = (2 \leq 4)$ 가 '참'이 되어
 $\text{Selection}(A, \text{left}, p-1, k) = \text{Selection}(A, 5, 9-1, 2) =$
 $\text{Selection}(A, 5, 8, 2)$ 호출
- 즉, Small group에서 2 번째로 작은 수를 찾는다.

- Selection(A, 5, 8, 2) 호출
- k=2, left=5, right=8

5	6	7	8
10	9	12	11

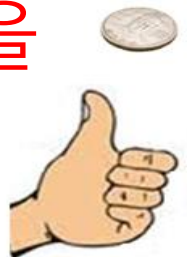
- Line 1에서 피벗 $A[5]=10$ 이라면, 원소 간 자리바꿈 없이 아래와 같이 된다.



5	6	7	8
9	10	12	11

- Line 2: Small group의 크기 계산 (즉, $S = (p-1)-left+1 = (6-1)-5+1 = 1$)
- 따라서 $k=2$ 번째 작은 수를 찾아야 하고, line 3의 if-조건 $(k \leq S) = (2 \leq 1)$ 이 '거짓'이 되지만, line 4의 if-조건 $(2 = S+1) = (2 = 1+1) = (2 = 2)$ 가 '참'이 되므로, 최종적으로 $A[6]=10$ 을 $k=7$ 번째 작은 수로서 해를 리턴한다.

- Selection 알고리즘은 분할 정복 알고리즘이기도 하지만 **랜덤 (random) 알고리즘**이기도 하다.
- 왜냐하면 선택 알고리즘의 line 1에서 피벗을 랜덤하게 정하기 때문이다.
- 만일 피벗이 입력 리스트를 너무 한 쪽으로 치우치게 분할하면, 즉, $|Small\ group| \ll |Large\ group|$ 또는 $|Small\ group| \gg |Large\ group|$ 일 때에는 알고리즘의 수행 시간이 길어진다.
- 선택 알고리즘이 호출될 때마다 line 1에서 입력을 한쪽으로 치우치게 분할될 확률은 마치 **동전을 던질 때 한쪽 면이 나오는 확률과 같다.**



Selection Python code

```
import random

def partition(A, left, right):
    p = random.randint(left, right)
    if p != left:
        A[left], A[p] = A[p], A[left]
    i = left+1
    j = right
    pvalue = A[left]
    done = False
    while not done:
        while i <= j and A[i] <= pvalue: i += 1
        while j >= i and A[j] >= pvalue: j -= 1
        if i > j:
            done = True
        else:
            A[i], A[j] = A[j], A[i]
    A[left], A[j] = A[j], A[left]
    return j
```

```
def selection(A, left, right, k):
    p = partition(A, left, right)
    S = (p - 1) - left + 1
    if k <= S:
        return selection(A, left, p-1, k)
    elif k == S + 1:
        return A[p]
    else:
        return selection(A, p+1, right, k-S-1)

A = [6, 3, 11, 9, 12, 2, 8, 15, 18, 10, 7, 14]
data = selection(A, 0, len(A)-1, 7)
print(data)
```

Another selection Python code

```
import random

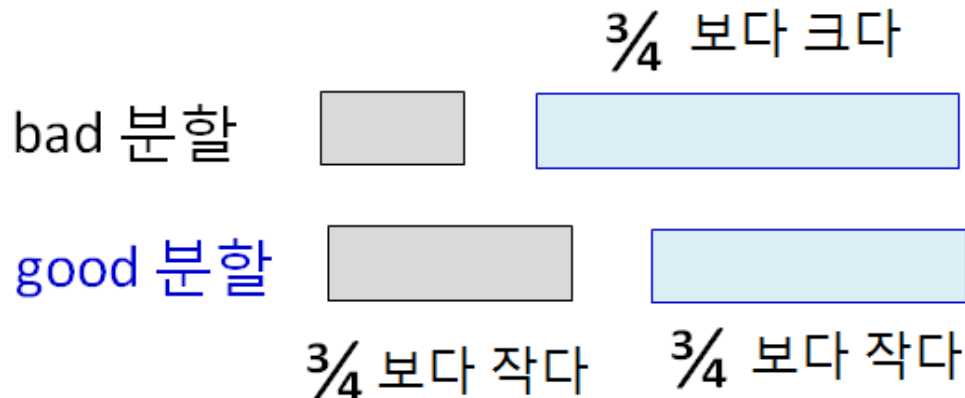
def partition(A):
    less = []
    equal = []
    greater = []
    pivot = A[random.randrange(0, len(A))]
    for x in A:
        if x < pivot:
            less.append(x)
        elif x == pivot:
            equal.append(x)
        else:
            greater.append(x)
    return less, equal, greater
```

```
def selection(A, k):
    less, equal, greater = partition(A)
    if k <= len(less) :
        return selection(less, k)
    elif k <= len(less) + len(equal):
        return equal[0]
    else:
        return selection(greater, k-len(less)-
len(equal))

A = [6, 3, 11, 9, 12, 2, 8, 15, 18, 10, 7, 14]
data = selection(A, 7)
print(data)
```

good/bad 분할

- 분할된 두 그룹 중의 하나의 크기가 입력 크기의 $3/4$ 과 같거나 그 보다 크게 분할하면 **나쁜 (bad) 분할**이라고 정의하자.
- **좋은 (good) 분할**은 그 반대의 경우이다.

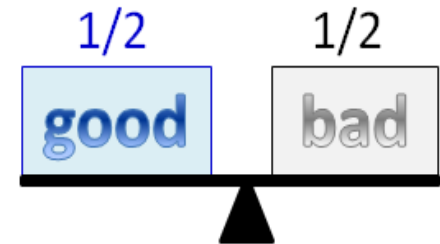


- 아래의 예를 살펴보면 good 분할이 되는 피봇을 선택할 확률과 bad 분할이 되는 피봇을 선택할 확률이 각각 **1/2로 동일**함을 확인할 수 있다. 다음과 같이 16개의 숫자가 있다고 가정하자.

1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16

- 16개 숫자들 중에서 **5~12** 중의 하나가 피봇이 되면 good 분할.

- **1~4** 또는 **13~16** 중 하나가 피봇으로 정해지면 bad 분할.



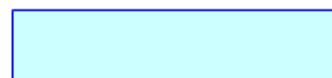
시간복잡도

- 피봇을 랜덤하게 정했을 때 **good 분할이 될 확률이 $1/2$** 이므로 평균 2회 연속해서 랜덤하게 피봇을 정하면 good 분할을 할 수 있다.
- 즉, 매 2회 호출마다 good 분할이 되므로, good 분할만 연속하여 이루어졌을 때만의 시간복잡도를 구하여, 그 값에 2를 곱하면 평균 경우 시간복잡도를 얻을 수 있다.

1 번째 good 분할



$\frac{3}{4}$ 보다 작다



$\frac{3}{4}$ 보다 작다

2 번째 good 분할



$(\frac{3}{4})^2$ 보다 작다

3 번째 good 분할



$(\frac{3}{4})^3$ 보다 작다

...

...

i 번째 good 분할

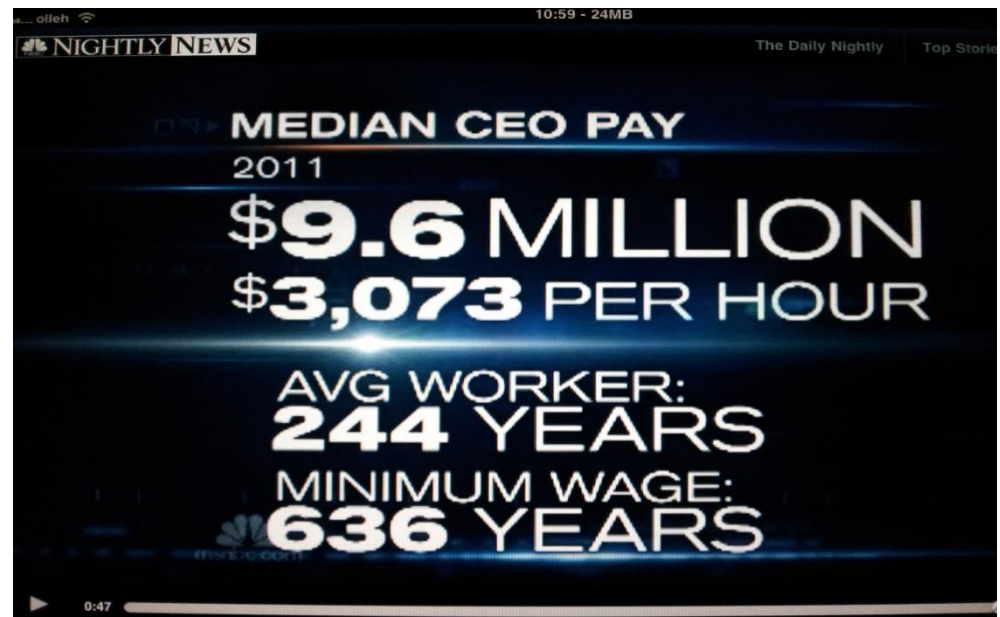


$(\frac{3}{4})^i$ 보다 작다

- 리스트 크기가 n 에서부터 $3/4$ 배로 연속적으로 감소되고, 리스트 크기가 1일 때에는 더 이상 분할할 수 없게 된다. 그러므로 Selection 알고리즘의 평균 경우 시간복잡도는 다음과 같다.
- $O[n + 3/4n + (3/4)^2n + (3/4)^3n + \dots + (3/4)^{i-1}n + (3/4)^in]$
 $= O(n)$
- Selection 알고리즘의 평균 경우 시간복잡도는
 $- 2 \times O(n) = O(n)$

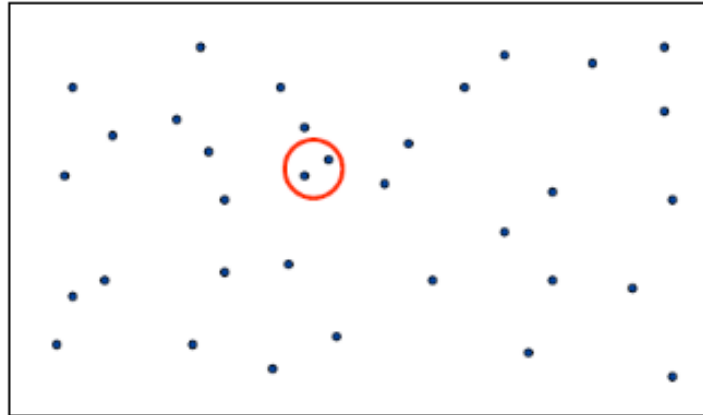
응 용

- 선택 알고리즘은 데이터 분석을 위한 **중앙값 (median)**을 찾는데 활용된다. 데이터 분석에서 평균값도 유용하지만, 중앙값이 더 설득력 있는 데이터 분석을 제공하기도 한다. 예를 들어, 대부분의 데이터가 1이고, 오직 1개의 숫자가 매우 큰 숫자 (노이즈 (noise), 잘못 측정된 데이터)이면, 평균값은 매우 왜곡된 분석이 된다.
- 실제로 대학 졸업 후 바로 취업한 직장인의 연간 소득을 분석할 때에 평균값보다 중앙값이 더 의미 있는 분석 자료가 된다.



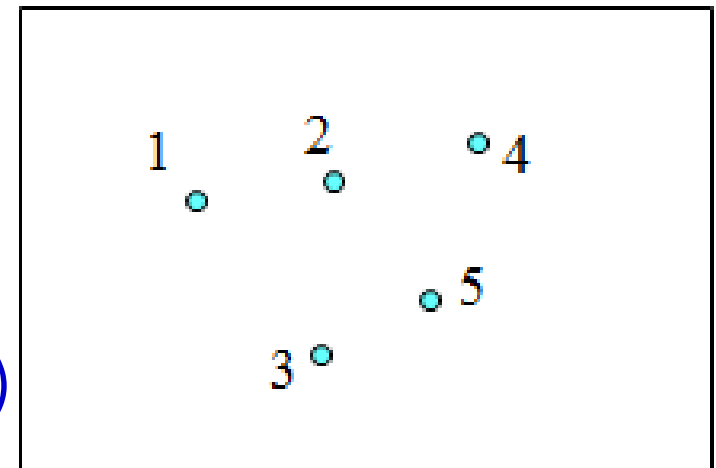
3.4 최근접 점의 쌍 찾기

- 최근접 점의 쌍 (Closest Pair) 문제는 2차원 평면상의 n 개의 점이 입력으로 주어질 때, 거리가 가장 가까운 한 쌍의 점을 찾는 문제이다.

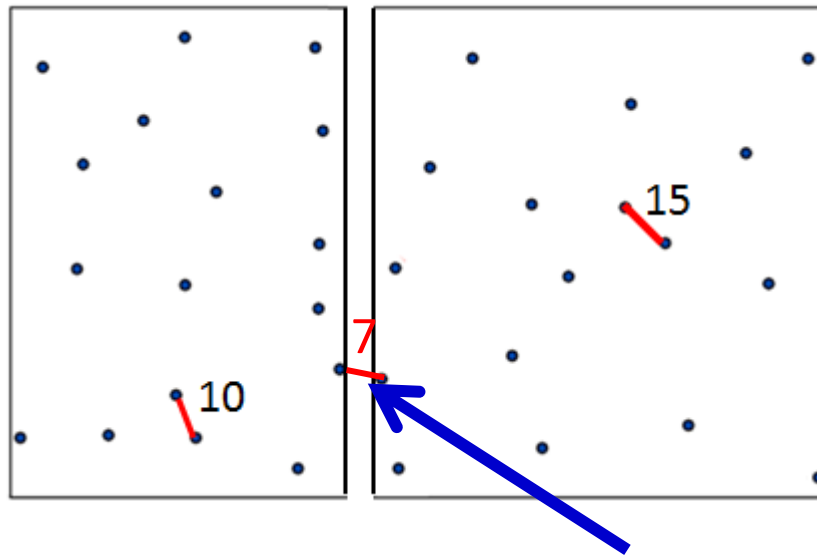


- 간단한 방법:

- 모든 점에 대하여 각각의 두 점 사이의 거리를 계산하여 가장 가까운 점의 쌍을 찾다.
- 예를 들어, 5개의 점이 아래의 [그림]처럼 주어지면, 1-2, 1-3, 1-4, 1-5, 2-3, 2-4, 2-5, 3-4, 3-5, 4-5 사이의 거리를 각각 계산하여 그 중에 최소 거리를 가진 쌍이 최근접 점의 쌍이 되는 것이다. 그러면 비교해야 할 쌍은 몇 개인가?
- ${}_nC_2 = n(n-1)/2$
- $n=5$ 이면, $5(5-1)/2 = 10$
- $n(n-1)/2 = O(n^2)$
- 한 쌍의 거리 계산은 $O(1)$ 시간
- 시간복잡도는 $O(n^2) \times O(1) = O(n^2)$

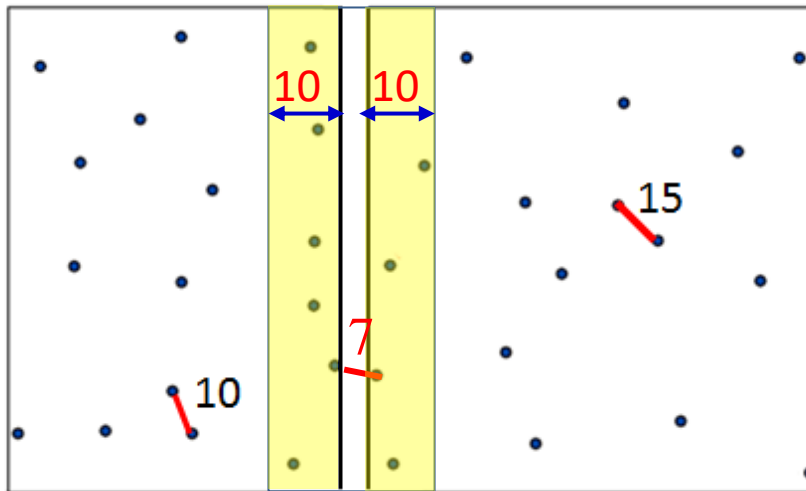


- $O(n^2)$ 보다 효율적인 분할 정복 이용:
 - n 개의 점을 $1/2$ 로 분할하여 각각의 부분 문제에서 최근접 점의 쌍을 찾고, 2개의 부분해 중에서 짧은 거리를 가진 점의 쌍을 일단 찾는다.
 - 그리고 2개의 부분해를 취합할 때에는 반드시 다음과 같은 경우를 고려해야 한다.



- 왼쪽 부분 문제의 최근접 쌍의 거리가 10이고, 오른쪽 부분 문제의 최근접 쌍의 거리가 15이다,
- 왼쪽 부분 문제의 가장 오른쪽 점과 오른쪽 부분 문제의 가장 왼쪽 점 사이의 거리가 7이다.
- 따라서 2개의 부분 문제의 해를 취합할 때 단순히 10과 15 중에서 짧은 거리인 10을 해라고 할 수 없는 것이다.


- 그러므로 아래의 그림에서와 같이 각각 거리가 10이내의 중간 영역 안에 있는 점들 중에 최근접 점의 쌍이 있는지도 확인해보아야 한다.



- 배열에 점의 좌표가 저장되어 있을 때, 중간 영역에 있는 점들을 찾는 방법
- 단, $d = \min\{\text{왼쪽 부분의 최근접 점의 쌍 사이의 거리}, \text{오른쪽 부분의 최근접 점의 쌍 사이의 거리}\}$ 이다.
- 아래의 배열에는 점들이 x-좌표의 오름차순으로 정렬되어 있고, 각 점의 y-좌표는 생략되었다.

왼쪽 부분 문제의
가장오른쪽 점

오른쪽 부분 문제의
가장왼쪽 점



(1,-)	(13,-)	(17,-)	(25,-)	(26,-)	(28,-)	(30,-)	(37,-)	(45,-)	(56,-)

- 중간 영역에 속한 점들은 왼쪽 부분 문제의 가장 오른쪽 점 (왼쪽 중간점)의 x-좌표에서 d 를 뺀 값과 오른쪽 부분 문제의 가장 왼쪽 점 (오른쪽 중간점)의 x-좌표에 d 를 더한 값 사이의 x-좌표 값을 가진 점들이다.
- $d=10$ 이라면, 점 $(17,-)$, $(25,-)$, $(26,-)$, $(28,-)$, $(30,-)$, $(37,-)$ 이 중간 영역에 속한다.

$d = 10$

0	1	2	3	4	5	6	7	8	9
$(1,-)$	$(13,-)$	$(17,-)$	$(25,-)$	$(26,-)$	$(28,-)$	$(30,-)$	$(37,-)$	$(45,-)$	$(56,-)$

$$26-d = 16$$

$$28+d = 38$$

최근접 점의 쌍 분할 정복 알고리즘

ClosestPair(S)

입력: x-좌표의 오름차순으로 정렬된 배열 s 에는 i 개의 점 (단, 각 점은 (x,y) 로 표현된다.)

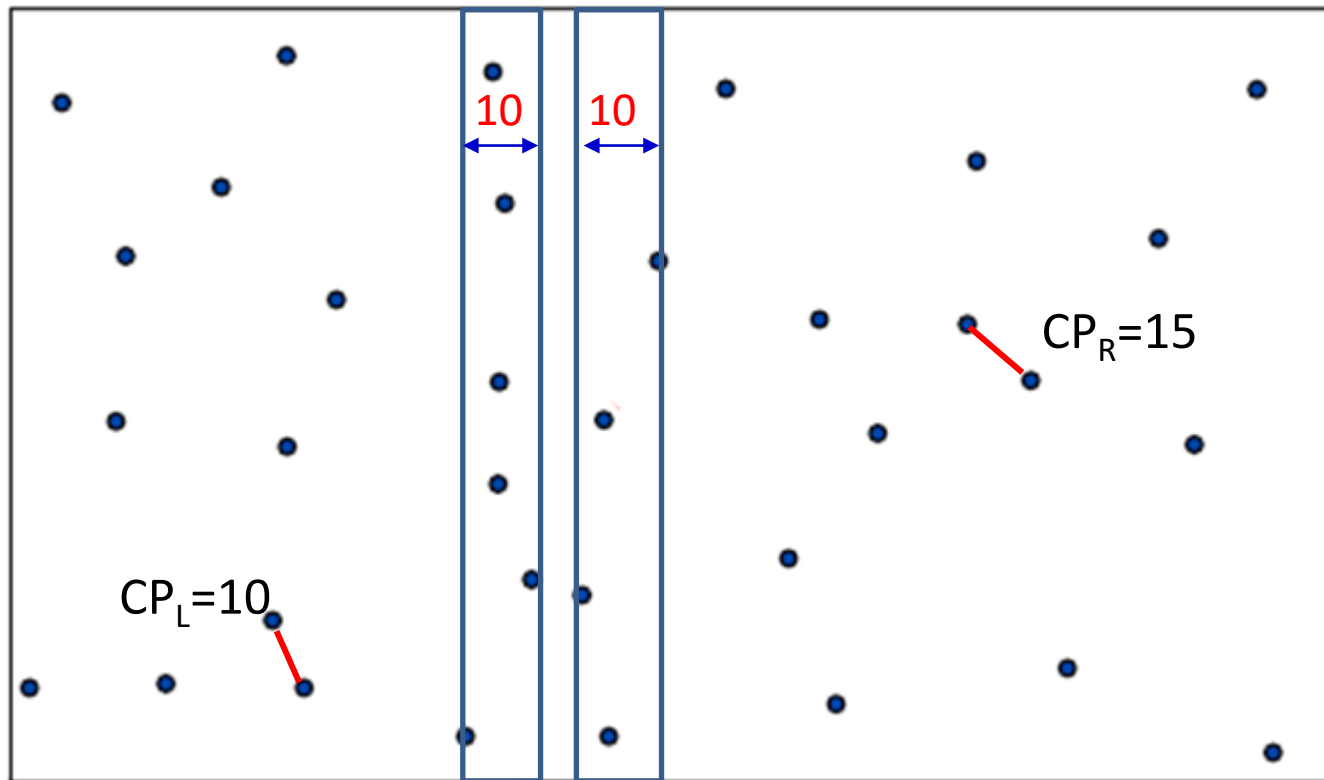
출력: s 에 있는 점들 중 최근접 점의 쌍의 거리

1. if ($i \leq 3$) return (2 또는 3개의 점들 사이의 최근접 쌍)
2. 정렬된 s 를 같은 크기의 s_L 과 s_R 로 분할한다. $|s|$ 가 홀수이면, $|s_L| = |s_R| + 1$ 이 되도록 분할한다.
3. $CP_L = \text{ClosestPair}(s_L)$ // CP_L 은 s_L 에서의 최근접 점의 쌍
4. $CP_R = \text{ClosestPair}(s_R)$ // CP_R 은 s_R 에서의 최근접 점의 쌍
5. $d = \min\{\text{dist}(CP_L), \text{dist}(CP_R)\}$ // $\text{dist}()$ 는 두 점 사이의 거리
6. $CP_C = d$ 로 얻은 중간영역에서 최근접 점의 쌍
7. return (CP_L, CP_C, CP_R 중에서 거리가 가장 짧은 쌍)

- Line 1에서는 s 에 있는 점의 수가 3개 이하이면 더 이상 분할하지 않는다. s 에 2개의 점이 있으면 s 를 그대로 리턴하고, 3개의 점이 있으면 3개의 쌍에 대하여 최근접 점의 쌍을 리턴한다.
- Line 2에서는 x -좌표로 정렬된 s 를 왼쪽과 오른쪽에 같은 개수의 점을 가지는 s_L 과 s_R 로 분할한다. 만일 s 의 점의 수가 홀수이면 s_L 쪽에 1개 많게 분할한다.
- Line 3~4에서는 분할된 s_L 과 s_R 에 대해서 재귀적으로 최근접 점의 쌍을 찾아서 각각을 CP_L 과 CP_R 이라고 놓는다.

- Line 6은 d 를 이용하여 중간 영역에 속하는 점들을 찾고, 이 점들 중에서 최근접 점의 쌍을 찾아서 이를 CP_C 라고 놓는다.

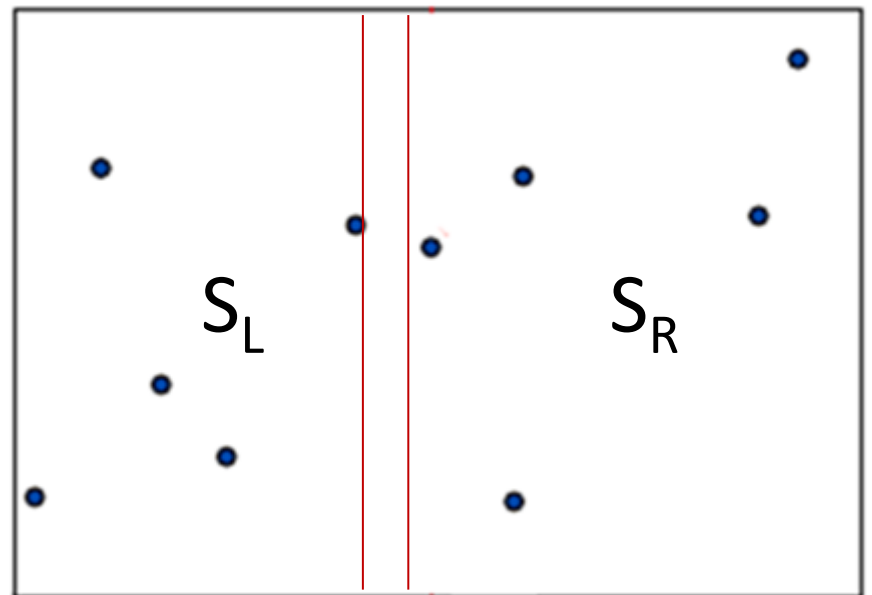
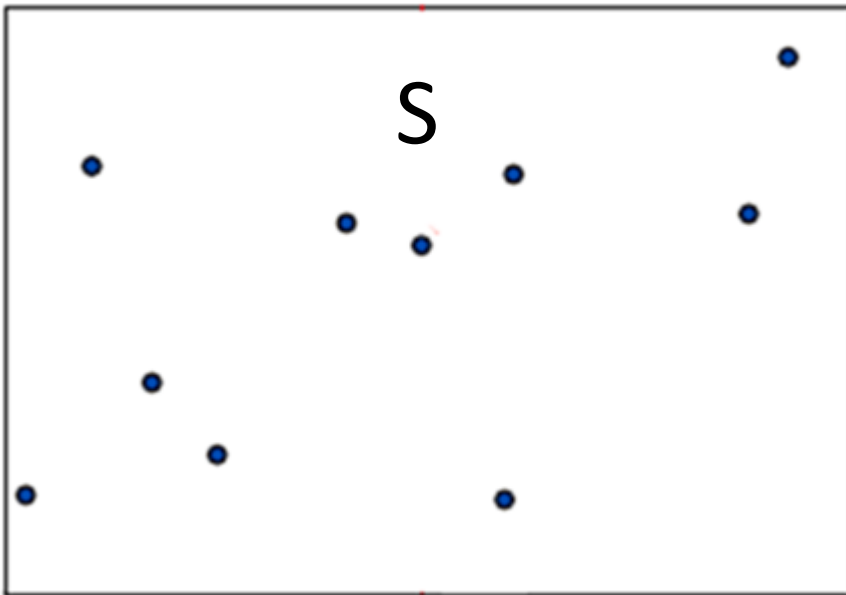
$$d = \min \{CP_L, CP_R\} = \min \{10, 15\} = 10$$



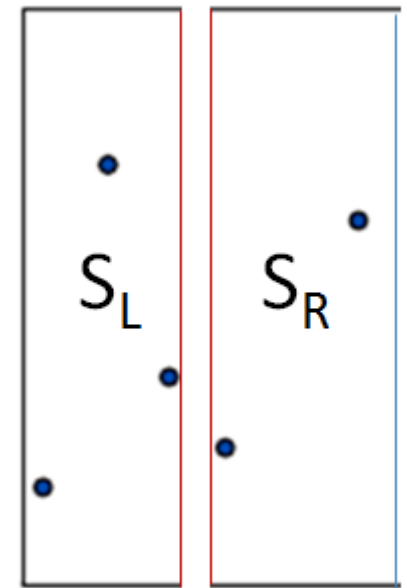
- Line 7에서는 line 3~4에서 각각 찾은 최근접 점의 쌍 CP_L 과 CP_R 과 line 6에서 찾은 CP_C 중에서 가장 짧은 거리를 가진 쌍을 해로서 리턴한다.

ClosestPair의 수행 과정

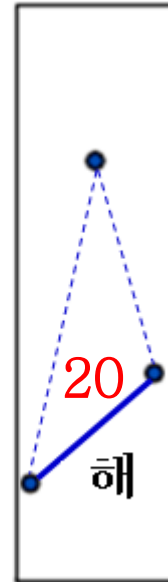
- ClosestPair(S)로 호출 **[1]**: 단, s 는 아래의 그림
- Line 1: S 의 점의 수 > 3 이므로 다음 line을 수행
- Line 2: S 를 S_L 과 S_R 로 분할



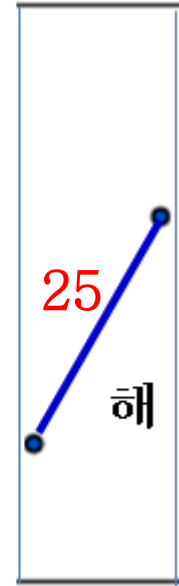
- Line 3: $\text{ClosestPair}(S_L)$ 호출: $\text{ClosestPair}(S_L)$ 을 수행한 후 리턴된 점의 쌍을 CP_L 이라고 놓은 후에 line 4~6을 차례로 수행
- $\text{ClosestPair}(S_L)$ 호출 **[2]**:
- Line 1의 if-조건이 '거짓'이므로 line 2에서 다시 분할
- Line 3: $\text{ClosestPair}(S_L)$ 호출 (여기서의 S_L 은 처음 S_L 의 왼쪽 반)
- $\text{ClosestPair}(S_L)$ 을 수행한 후, 리턴된 점의 쌍을 CP_L 이라고 놓은 후에 line 4~6을 차례로 수행



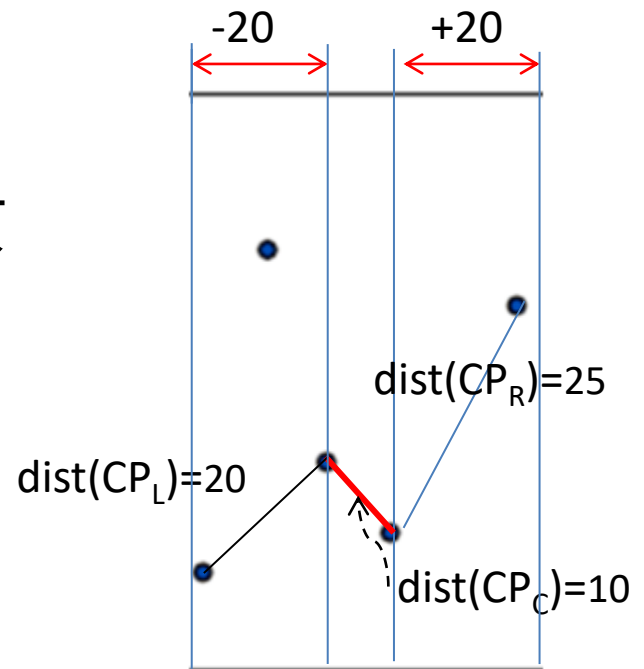
- ClosestPair(S_L) 호출:
- Line 1의 if-조건이 '참'; S_L 의 3개의 점들에 대해서 최근접 점의 쌍을 찾는다. 옆의 그림과 같이 3개의 쌍에 대해 거리를 각각 계산하여 최근접 쌍을 해로서 리턴
- 최근접 점의 쌍의 거리를 **20**이라고 가정



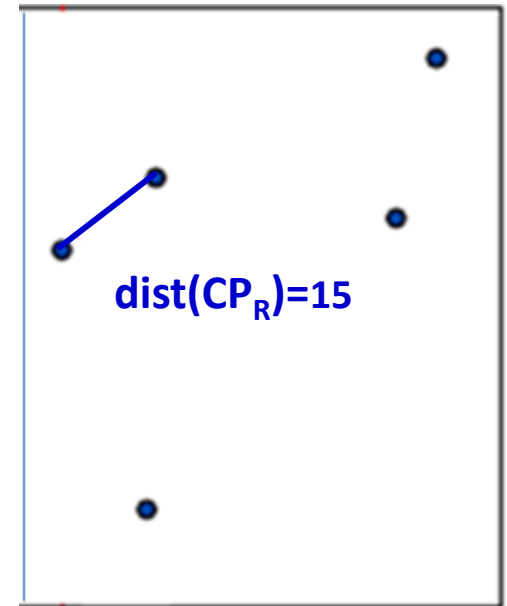
- ClosestPair(S_R) 호출
- Line 1에서 점의 수가 2이므로,
이 두 점을 최근접 점의 쌍으로
리턴
- 최근접 점의 쌍의 거리를 **25**라고
가정



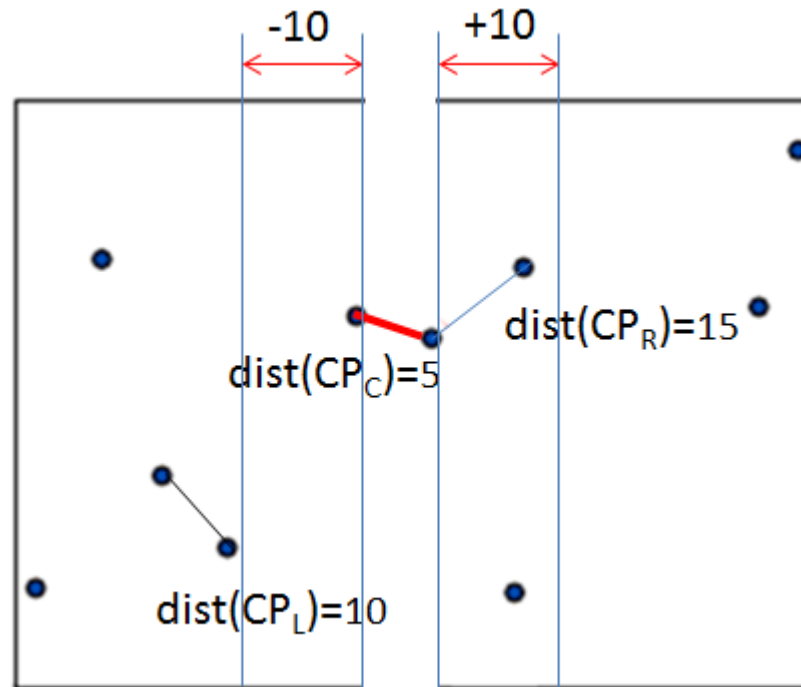
- **[2]**의 ClosestPair(S_L) 호출 당시 line 3~4가 수행되었고, 이제 line 5가 수행된다.
- Line 3~4에서 찾은 최근접 점의 쌍 사이의 거리인 $\text{dist}(CP_L)=20$ 과 $\text{dist}(CP_R)=25$ 중에 작은 값을 **$d=20$** 라고 놓는다.
- Line 6에서 왼쪽 중간점의 x-좌표에서 20을 뺀 값과 오른쪽 중간점의 x-좌표에 20을 더한 값 사이의 x-좌표 값을 가진 점들 중에서 CP_C 를 찾는다.
- 옆의 그림과 같이 거리가 10인 CP_C 를 찾는다.



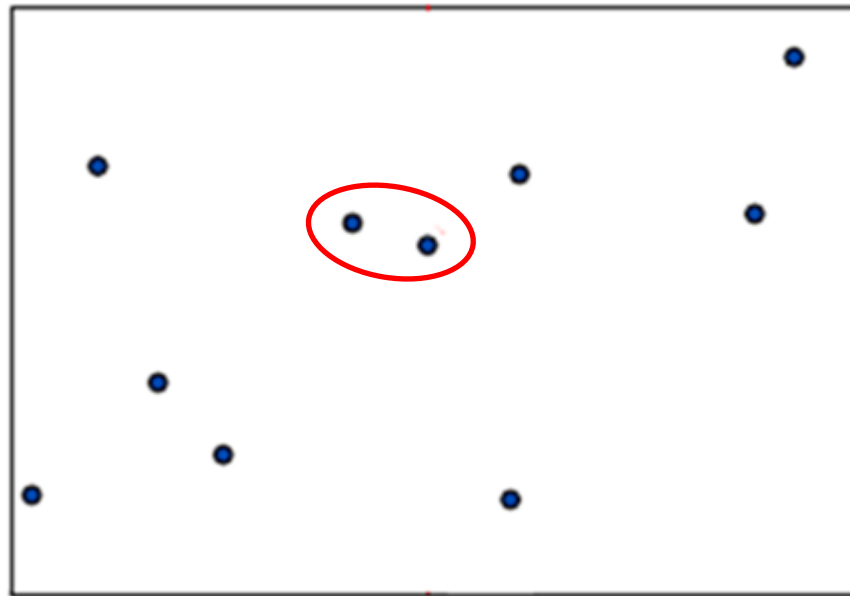
- Line 7: $\text{dist}(CP_L)=20$, $\text{dist}(CP_C)=10$, $\text{dist}(CP_R)=25$ 중에서 가장 거리가 짧은 쌍인 $CP_C=10$ 을 리턴
- **[1]**의 ClosestPair(S) 호출 당시 line 3이 수행되었고, 이제 line 4에서는 ClosestPair(S_R)을 호출한다. 여기서 S_R 은 초기 입력의 오른쪽 반인 영역이다. ClosestPair(S_R) 호출 결과로 옆 그림의 최근접 점의 쌍을 리턴하고 이를 CP_R 로 놓는다.
- 이때 $\text{dist}(CP_R)=15$ 라고 하자.



- Line 5~6: line 3~4에서 찾은 최근접 점의 쌍 사이의 거리인 $\text{dist}(\text{CP}_L)=10$ 과 $\text{dist}(\text{CP}_R)=15$ 중에 작은 값을 $d=10$ 이라고 놓는다. 그리고 중간 영역에 있는 점들 중에서 CP_C 를 찾는다. 여기서는 아래의 그림과 같이 거리가 5인 CP_C 를 최종적으로 찾는다.



- Line 6: $\text{dist}(\text{CP}_L)=10$, $\text{dist}(\text{CP}_C)=5$, $\text{dist}(\text{CP}_R)=15$ 중에서 가장 거리가 짧은 쌍인 **CP_C** 를 최근접 쌍의 점으로 리턴(최종해)



ClosestPair algorithm Python code 1

```
import math
def dist(p1, p2):
    return math.sqrt((p1[0] - p2[0]) ** 2 + (p1[1] - p2[1]) ** 2)

def brute(ax):
    mi = dist(ax[0], ax[1])
    p1 = ax[0]
    p2 = ax[1]
    ln_ax = len(ax)
    if ln_ax == 2:
        return p1, p2, mi
    for i in range(ln_ax-1):
        for j in range(i + 1, ln_ax):
            if i != 0 and j != 1:
                d = dist(ax[i], ax[j])
                if d < mi: # Update min_dist and points
                    mi = d
                    p1, p2 = ax[i], ax[j]
    return p1, p2, mi
```

<https://medium.com/@andriylazorenko/closest-pair-of-points-in-python-79e2409fc0b2>

ClosestPair algorithm Python code 2

```
def closest_split_pair(p_x, p_y, delta, best_pair):
    ln_x = len(p_x)  # store length - quicker
    mx_x = p_x[ln_x // 2][0]  # select midpoint on x-sorted array
    # Create a subarray of points not further than delta from
    # midpoint on x-sorted array
    s_y = [x for x in p_y if mx_x - delta <= x[0] <= mx_x + delta]
    best = delta  # assign best value to delta
    ln_y = len(s_y)  # store length of subarray for quickness
    for i in range(ln_y - 1):
        for j in range(i+1, min(i + 7, ln_y)):
            p, q = s_y[i], s_y[j]
            dst = dist(p, q)
            if dst < best:
                best_pair = p, q
                best = dst
    return best_pair[0], best_pair[1], best
```

ClosestPair algorithm Python code 3

```
def closest_pair(ax, ay):
    ln_ax = len(ax)  # It's quicker to assign variable
    if ln_ax <= 3:
        return brute(ax)  # A call to brute force comparison
    mid = ln_ax // 2  # Division without remainder, need int
    Qx = ax[:mid]  # Two-part split
    Rx = ax[mid:]
    # Determine midpoint on x-axis
    midpoint = ax[mid][0]
    Qy = list()
    Ry = list()
    for x in ay:  # split ay into 2 arrays using midpoint
        if x[0] <= midpoint:
            Qy.append(x)
        else:
            Ry.append(x)
```

ClosestPair algorithm Python code 4

```
# Call recursively both arrays after split
(p1, q1, mi1) = closest_pair(Qx, Qy)
(p2, q2, mi2) = closest_pair(Rx, Ry)
# Determine smaller distance between points of 2 arrays
if mi1 <= mi2:
    d = mi1
    mn = (p1, q1)
else:
    d = mi2
    mn = (p2, q2)
# Call function to account for points on the boundary
(p3, q3, mi3) = closest_split_pair(ax, ay, d, mn)
# Determine smallest distance for the array
if d <= mi3:
    return mn[0], mn[1], d
else:
    return p3, q3, mi3
```

ClosestPair algorithm Python code 5

```
def solution(x, y):  
    a = list(zip(x, y))    # This produces list of tuples  
    ax = sorted(a, key=lambda x: x[0])    # Presorting x-wise  
    ay = sorted(a, key=lambda x: x[1])    # Presorting y-wise  
    p1, p2, mi = closest_pair(ax, ay)    # Recursive D&C function  
    return mi  
  
xarr = [3, 8, 4, 11, 6, 6, 5, 1, 11, 10]  
yarr = [3, 3, 6, 7, 6, 8, 1, 7, 1, 9]  
dist = solution(xarr, yarr)  
print("The shortest distance is", dist)
```

sorted() 함수

- sorted(a): list 'a'를 정렬
- 각 원소의 key로 특정한 필터링을 하고 싶다면?

```
def ft(x):  
    return x[0]  
  
sorted(a, key=ft) # a[i]의 0번째 원소를 정렬에 사용
```

- 예

```
def ft1(x):  
    return x[0]  
  
a = {(4, 2), (1, 5), (3, 3), (0, 1)}  
b = sorted(a, key=ft1)  
print(b)  
[(0, 1), (1, 5), (3, 3), (4, 2)]
```

lambda

- 함수가 너무 짧아 inline으로 바꾸고 싶은데?

```
a = {(4, 2), (1, 5), (3, 3), (0, 1)}  
b = sorted(a, key = def ft1(x): return x[0])  
print(b)
```

invalid syntax
error

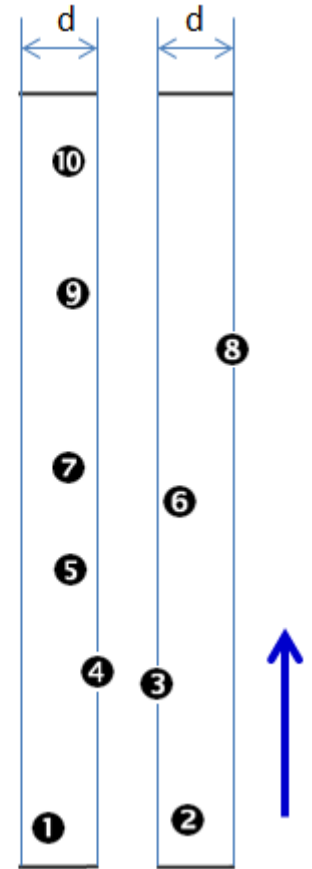
- lambda – anonymous function

```
a = {(4, 2), (1, 5), (3, 3), (0, 1)}  
b = sorted(a, key = lambda x: x[0])  
print(b)  
[(0, 1), (1, 5), (3, 3), (4, 2)]
```

시간복잡도

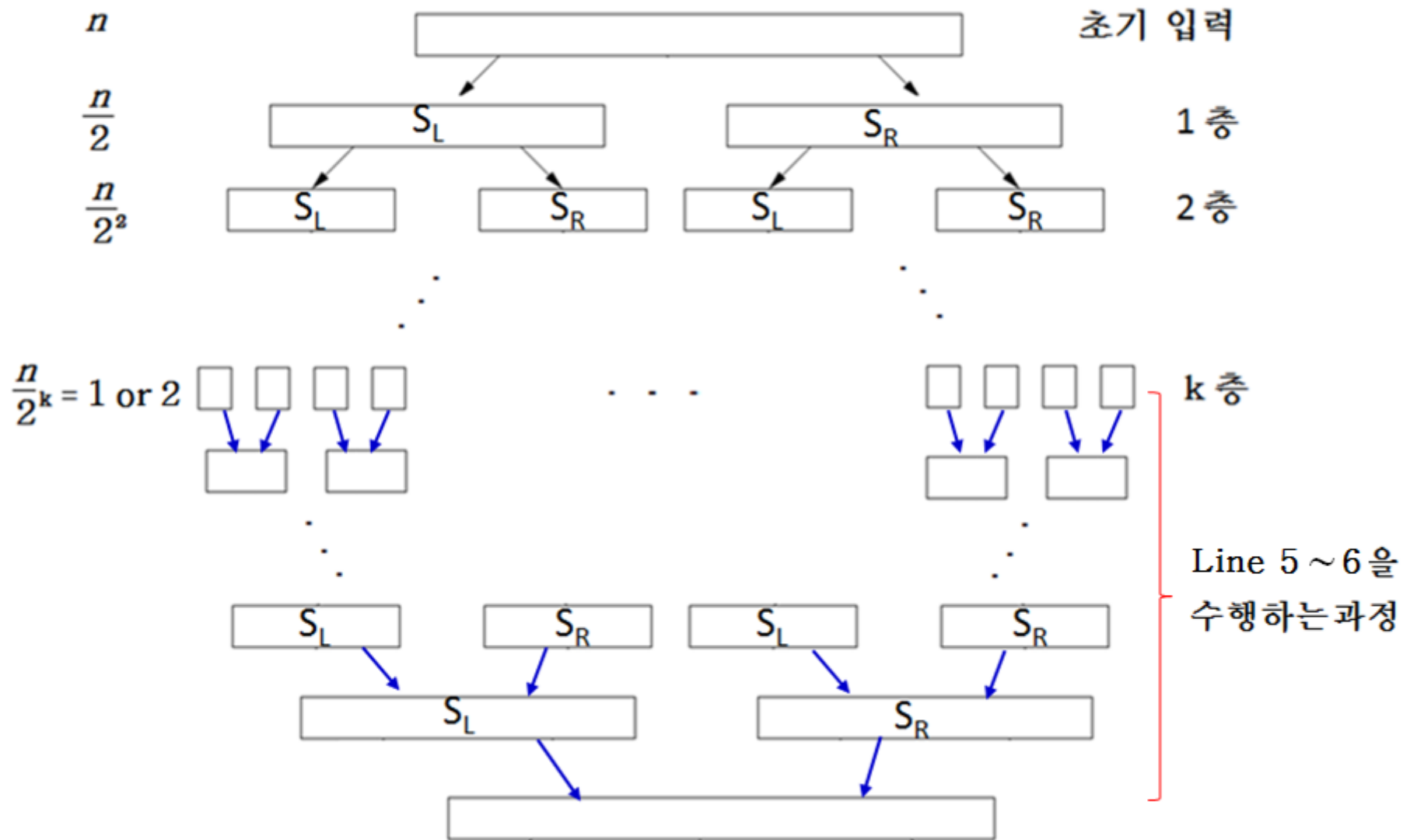
- 입력 s 에 n 개의 점이 있다면
- 전처리 (preprocessing) 과정으로서 s 의 점을 x -좌표로 정렬: $O(n \log n)$
- Line 1: s 에 3개의 점이 있는 경우에 3번의 거리 계산이 필요하고, s 의 점의 수가 2이면 1번의 거리 계산이 필요하므로 $O(1)$ 시간이 걸린다.
- Line 2: 정렬된 s 를 s_L 과 s_R 로 분할하는데, 이미 배열에 정렬되어 있으므로, 배열의 중간 인덱스로 분할하면 된다. 이는 $O(1)$ 시간 걸린다.
- Line 3~4: s_L 과 s_R 에 대하여 각각 ClosestPair를 호출하는데, 분할하며 호출되는 과정은 합병 정렬과 동일

- Line 6: $d = \min\{\text{dist}(CP_L), \text{dist}(CP_R)\}$ 일 때 중간 영역에 속하는 점들 중에서 최근접 점의 쌍을 찾는다.
- 이를 위해 먼저 **중간 영역에 있는 점들을 y-좌표 기준으로 정렬**한 후에, 아래에서 위로 (또는 위에서 아래로) 각 점을 기준으로 거리가 d 이내인 주변의 점들 사이의 거리를 각각 계산하며, 이 영역에 속한 점들 중에서 최근접 점의 쌍을 찾는다.
- 따라서 y-좌표로 정렬하는데 $O(n \log n)$ 시간이 걸리고, 그 다음에는 아래에서 위로 올라가며 각 점에서 주변의 점들 사이의 거리를 계산하는데 $O(1)$ 시간이 걸린다. 왜냐하면 각 점과 거리 계산해야 하는 주변 점들의 수는 $O(1)$ 개이기 때문이다.



- Line 7: 3개의 점의 쌍 중에 가장 짧은 거리를 가진 점의 쌍을 리턴하므로 $O(1)$ 시간이 걸린다.
- ClosestPair 알고리즘의 분할과정은 합병 정렬의 분할과정과 동일
- 그러나 ClosestPair 알고리즘에서는 해를 취합하여 올라가는 과정인 line 5~6에서 $O(n\log n)$ 시간이 필요
- 다음의 그림에서 k층까지 분할된 후, 층별로 line 5~6이 수행되는 (취합) 과정을 보여준다. 이때 각 층의 수행 시간은 $O(n\log n)$ 이다. 여기에 층 수인 $\log n$ 을 곱하면 **$O(n\log^2 n)$** 이 된다. (**ClosestPair 알고리즘의 시간복잡도**)

각 부분의
크기

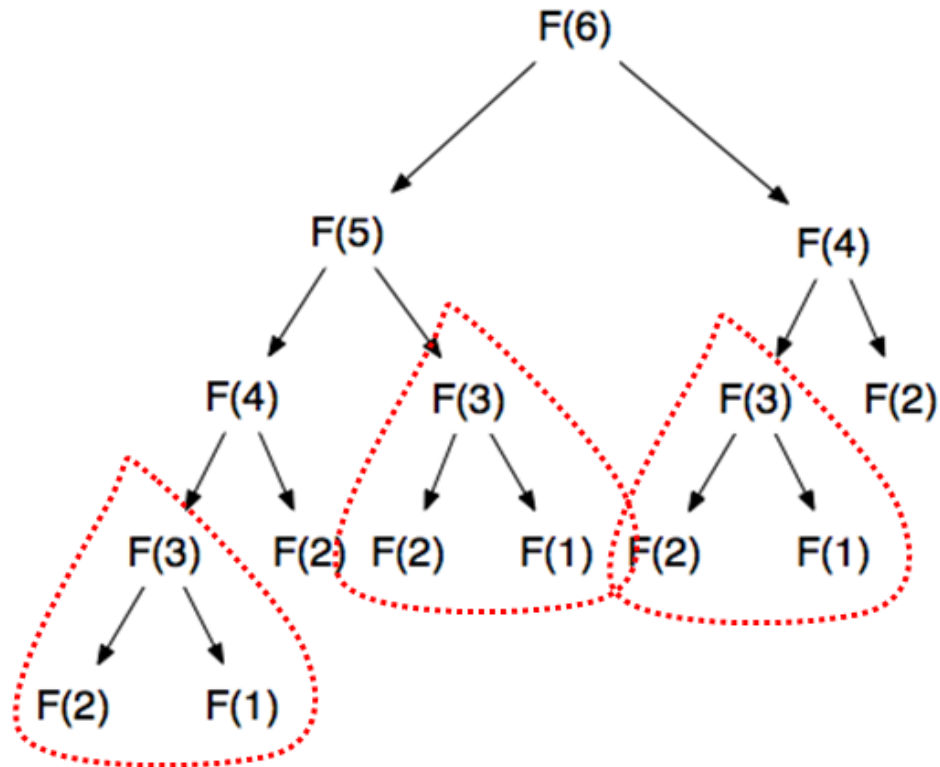


응 용

- 컴퓨터 그래픽스
- 컴퓨터 비전 (Vision)
- 지리 정보 시스템 (Geographic Information System, GIS)
- 분자 모델링 (Molecular Modeling)
- 항공 트래픽 조정 (Air Traffic Control)
- 마케팅 (주유소, 프랜차이즈 신규 가맹점 등의 위치 선정) 등

3.5 분할 정복을 적용하는데 있어서 주의할 점

- 분할 정복이 부적절한 경우는 입력이 분할될 때마다 분할된 부분문제의 입력 크기의 합이 분할되기 전의 입력 크기보다 매우 커지는 경우이다.



- 예를 들어, n 번째의 피보나치 수를 구하는데 $F(n) = F(n-1) + F(n-2)$ 로 정의되므로 재귀 호출을 사용하는 것이 자연스러워 보이나, 이 경우의 입력은 1개이지만, 사실상 **n 의 값 자체가 입력 크기**인 것이다.
- 따라서 n 이라는 숫자로 인해 2개의 부분 문제인 $F(n-1)$ 과 $F(n-2)$ 가 만들어지고, 2개의 입력 크기의 합이 $(n-1) + (n-2) = (2n-3)$ 이 되어서, **분할 후 입력 크기가 거의 2배로 늘어난다**. 이전 슬라이드의 그림은 피보나치 수 $F(5)$ 를 구하기 위해 분할된 부분 문제들을 보여준다. $F(2)$ 를 5번이나 중복하여 계산해야 하고, $F(3)$ 은 3번 계산된다.

피보나치 수 계산을 위한 $O(n)$ 시간 알고리즘

FibNumber(n)

1. $F[0]=0$

2. $F[1]=1$

3. for $i=2$ to n

4. $F[i] = F[i-1] + F[i-2]$

- 주어진 문제를 분할 정복 알고리즘으로 해결하려고 할 때에 주의해야 하는 또 하나의 요소는 취합(정복) 과정이다.
- 입력을 분할만 한다고 해서 효율적인 알고리즘이 만들어지는 것은 아니다.
- 3장에서 살펴본 문제들은 취합 과정이 간단하거나 필요 없었고, 최근접 점의 쌍을 위한 알고리즘만이 조금 복잡한 편이었다.
- 또한 기하 (geometry)에 관련된 다수의 문제들이 효율적인 분할 정복 알고리즘으로 해결되는데, 이는 기하 문제들의 특성상 취합 과정이 문제 해결에 잘 부합되기 때문이다.

요약

- 분할 정복 (Divide-and-Conquer) 알고리즘: 주어진 문제의 입력을 분할하여 문제를 해결 (정복)하는 방식의 알고리즘이다.
- 합병 정렬 (Merge sort): n 개의 숫자들을 $n/2$ 개씩 2개의 부분 문제로 분할하고, 각각의 부분 문제를 재귀적으로 합병 정렬한 후, 2개의 정렬된 부분을 합병하여 정렬 (정복)한다. 시간복잡도는 $O(n \log n)$ 이다.
- 합병 정렬의 공간 복잡도는 $O(n)$ 이다.

요약

- 퀵 정렬 (Quick sort): 피벗 (pivot)이라 일컫는 배열의 원소를 기준으로 피벗보다 작은 숫자들은 왼편으로, 피벗보다 큰 숫자들은 오른편에 위치하도록 분할하고, 피벗을 그 사이에 놓는다. 퀵 정렬은 분할된 부분 문제들에 대하여서도 위와 동일한 과정을 재귀적으로 수행하여 정렬한다.
- 퀵 정렬의 평균 경우 시간복잡도는 $O(n \log n)$, 최악 경우 시간복잡도는 $O(n^2)$, 최선 경우 시간복잡도는 $O(n \log n)$ 이다.
- 선택 (Selection) 문제: k 번째 작은 수를 찾는 문제로서, 입력에서 퀵 정렬에서와 같이 피벗을 선택하여 피벗보다 작은 부분과 큰 부분으로 분할한 후에 k 번째 작은 수가 들어있는 부분을 재귀적으로 탐색한다. 평균 경우 시간복잡도는 $O(n)$ 이다.

요약

- 최근접 점의 쌍 (Closest Pair) 문제: n 개의 점들을 $1/2$ 로 분할하여 각각의 부분 문제에서 최근접 점의 쌍을 찾고, 2개의 부분해 중에서 짧은 거리를 가진 점의 쌍을 일단 찾는다. 그리고 2개의 부분해를 취합할 때, 반드시 중간 영역 안에 있는 점들 중에 최근접 점의 쌍이 있는지도 확인해보아야 한다. 시간복잡도는 $O(n\log^2 n)$ 이다.
- 분할 정복이 부적절한 경우는 입력이 분할될 때마다 분할된 부분문제들의 입력 크기의 합이 분할되기 전의 입력 크기보다 커지는 경우이다. 또 하나 주의해야 할 요소는 취합 (정복) 과정이다.