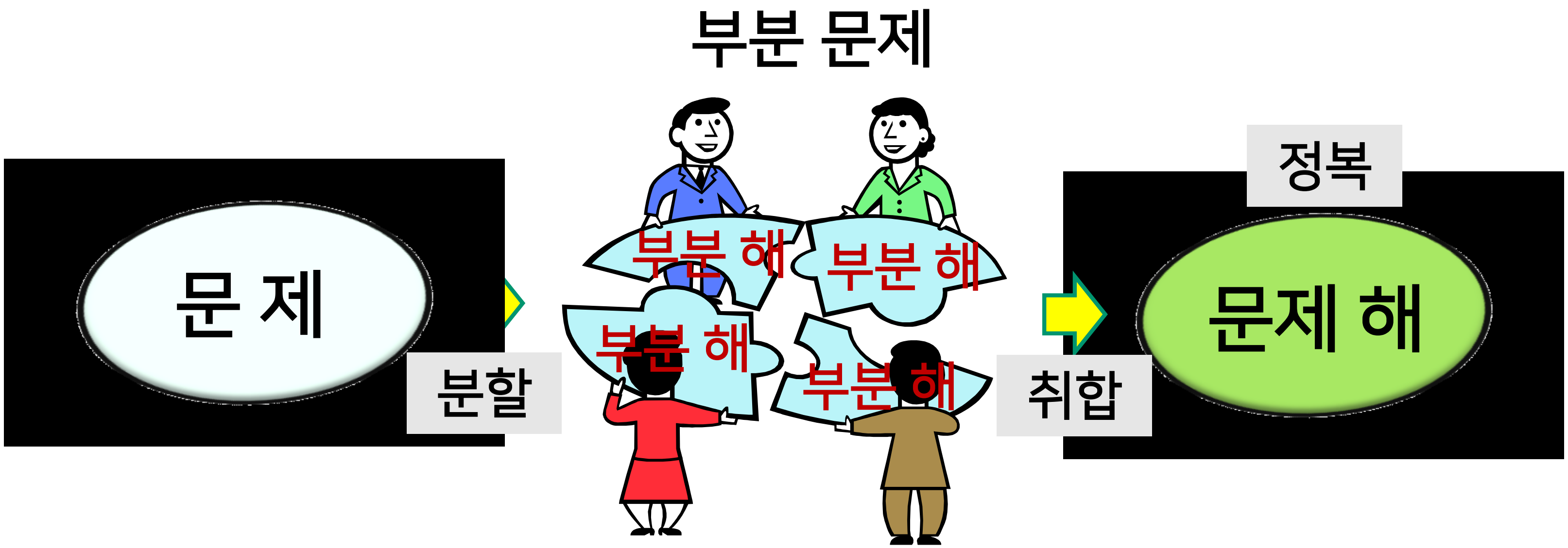
Chapter 3
분할 정복 알고리즘

차례

분할 정복 알고리즘

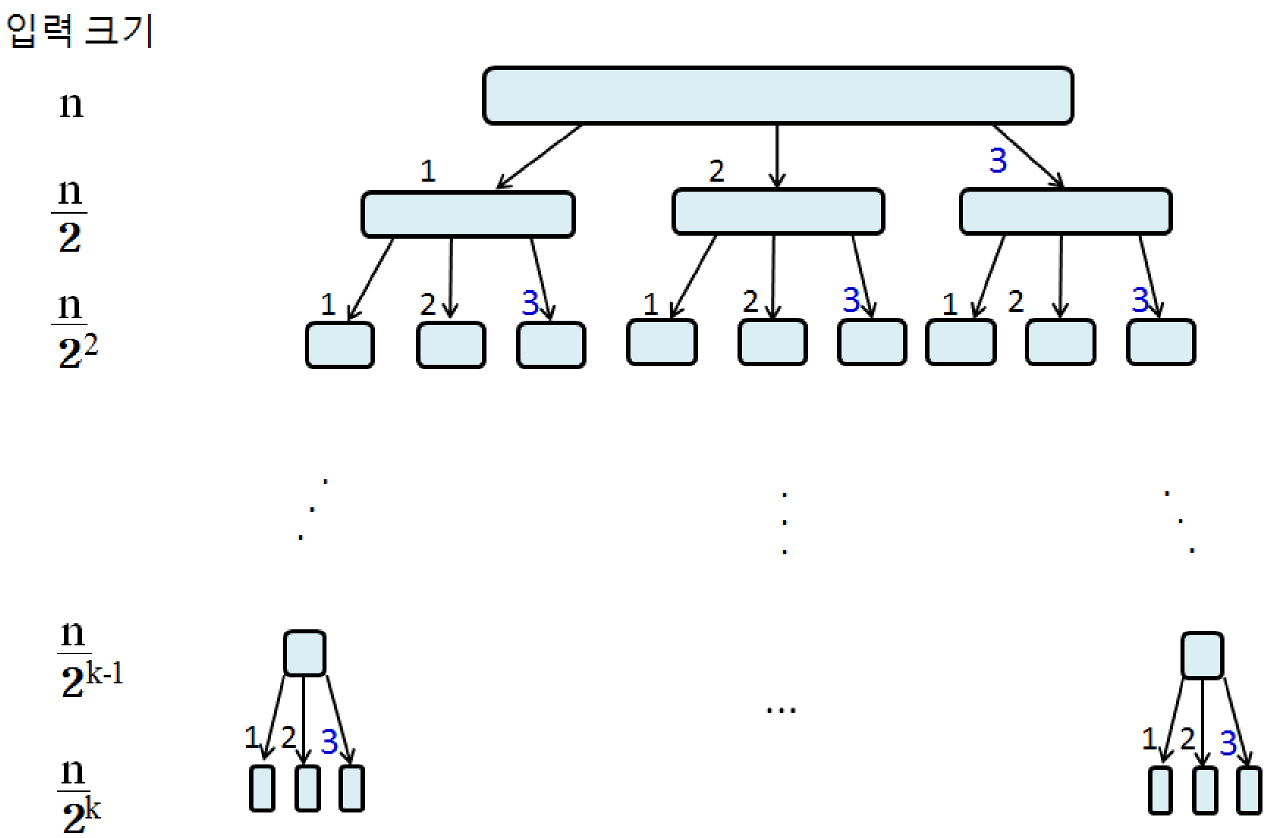
- 주어진 문제의 입력을 분할하여 문제를 해결(정복)하는 방식의 알고리즘
  - 분할한 입력에 대하여 동일한 알고리즘을 적용하여 해를 계산
  - 이들의 해를 취합하여 원래 문제의 해를 얻음
- 부분 문제와 부분 해
  - 분할된 입력에 대한 문제를 부분 문제 (subproblem)
  - 부분 문제의 해를 부분 해
  - 부분 문제는 더 이상 분할할 수 없을 때까지 분할

분할 정복 알고리즘



분할 과정

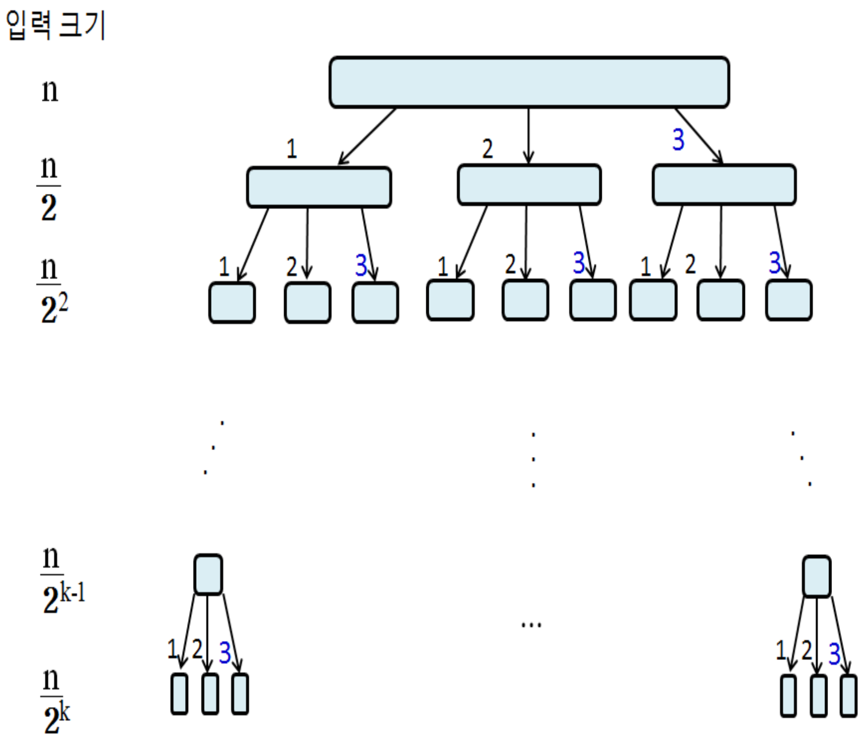
- 크기가 n인 입력을 3개로 분할하고, 각각 분할된 부분 문제의 크기가 n/2일 경우의 분할 예



입력 크기가 n일 때 총 분할 횟수

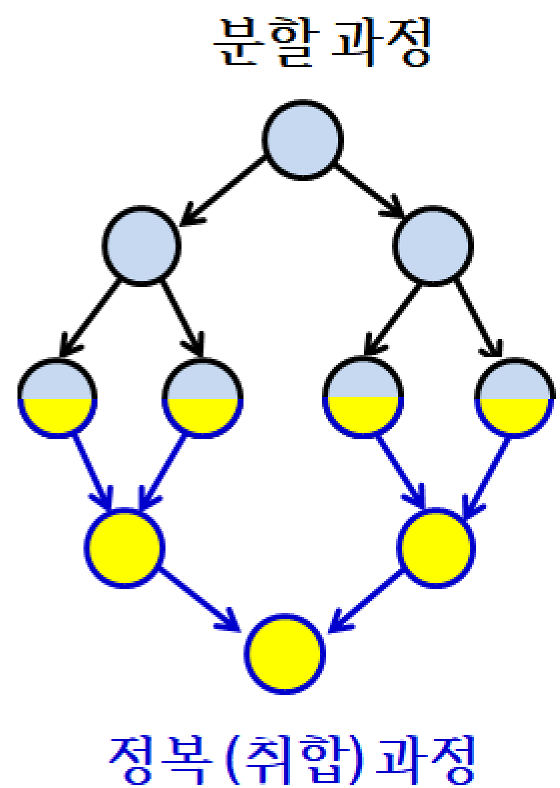
- 총 분할한 횟수 = k라면
- 1번 분할 후 각 입력 크기 n/2
- 2번 분할 후 각 입력 크기 $n/2^2$

  ⋮
- k번 분할 후 각 입력 크기 $n/2^k$
- $n/2^k = 1$일 때 분할 못함
- $k = \log_2 n$



정복 과정

- 대부분의 분할 정복 알고리즘
  - 문제의 입력을 단순히 분할만 해서는 해를 구할 수 없다.
- 따라서 분할된 부분 문제들을 정복해야 함
  - 부분 해를 찾아야 한다.
  - 정복하는 방법은 문제에 따라 다르다.
  - 일반적으로 부분 문제들의 해를 취합하여 보다 큰 부분 문제의 해를 구한다.



분할 정복 알고리즘의 분류

- 분할 정복 알고리즘은 분할되는 부분 문제의 수와 부분 문제의 크기에 따라서 다음과 같이 분류

- 문제가 a개로 분할되고, 부분 문제의 크기가 1/b로 감소하는 알고리즘
  - a=b=2인 경우: 합병 정렬, 최근접 점의 쌍 찾기, 공제선 문제
  - a=3, b=2인 경우: 큰 정수의 곱셈
  - a=4, b=2인 경우: 큰 정수의 곱셈
  - a=7, b=2인 경우: 스트라센(Strassen)의 행렬 곱셈 알고리즘

분할 정복 알고리즘의 분류

- 문제가 2개로 분할되고, 부분 문제의 크기가 일정하지 않은 크기로 감소하는 알고리즘
  - 퀵 정렬
- 문제가 2개로 분할되나, 그 중에 1개의 부분 문제는 고려할 필요가 없으며, 부분 문제의 크기가 1/2로 감소하는 알고리즘
  - 이진 탐색
- 문제가 2개로 분할되나, 그 중에 1개의 부분 문제는 고려할 필요가 없으며, 부분 문제의 크기가 일정하지 않은 크기로 감소하는 알고리즘
  - 선택 문제 알고리즘
- 부분 문제의 크기가 1, 2개씩 감소하는 알고리즘
  - 삽입 정렬, 피보나치 수의 계산

3.1 합병 정렬 (Merge Sort)

- 합병 정렬은 입력이 2개의 부분 문제로 분할되고, 부분 문제의 크기가 1/2로 감소하는 분할 정복 알고리즘
  - n개의 숫자들을 n/2개씩 2개의 부분 문제로 분할
  - 각각의 부분 문제를 순환으로 합병 정렬
  - 2개의 정렬된 부분을 합병하여 정렬(정복)

- 합병 과정이 문제를 정복하는 것

합병 (merge)

- 2개의 각각 정렬된 숫자들을 1개의 정렬된 숫자로 합치는 것

배열 A: 6 14 18 20 29

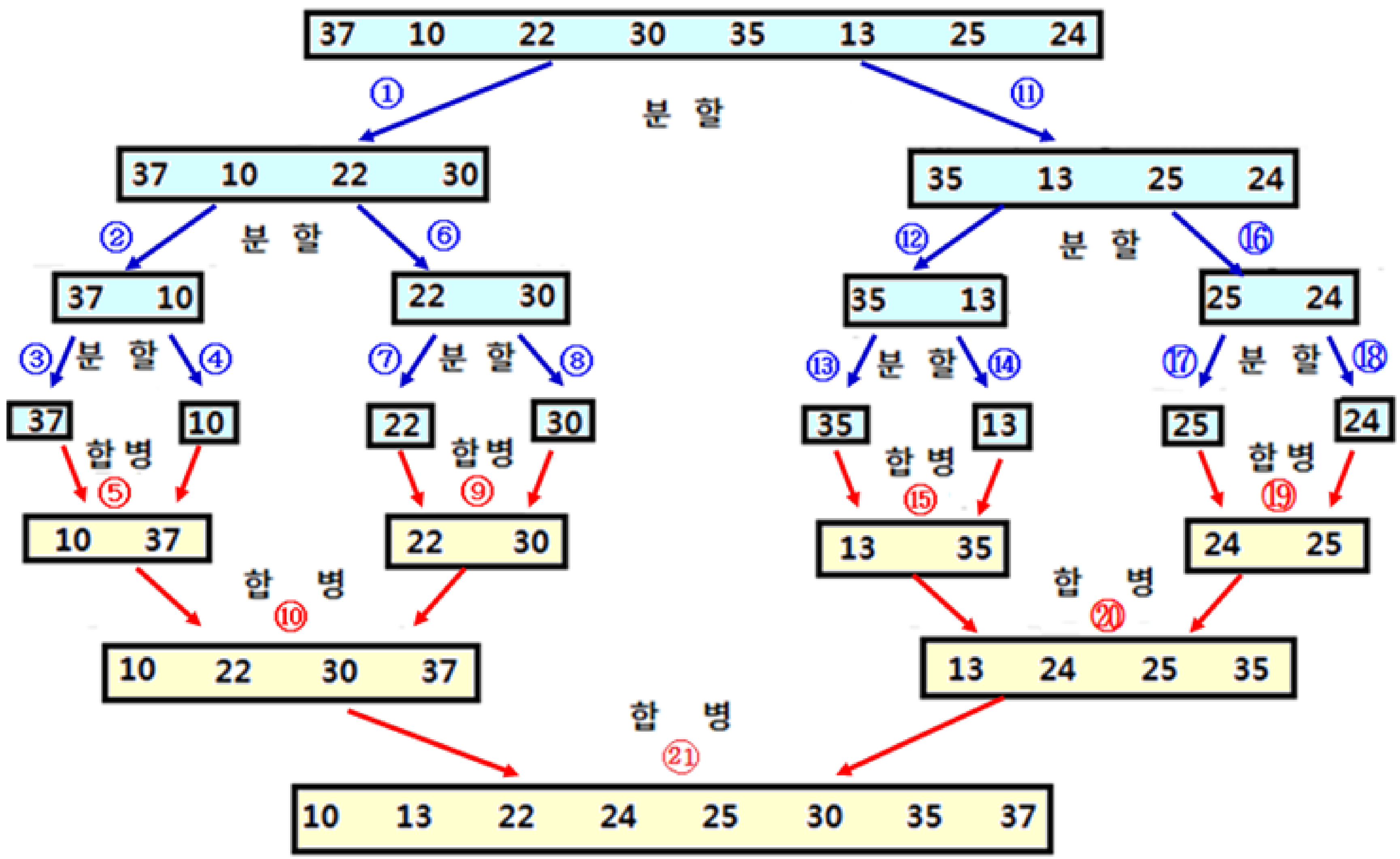
배열 B: 1 2 15 25 30 45

배열 C: 1 2 6 14 15 18 20 25 29 30 45

알고리즘

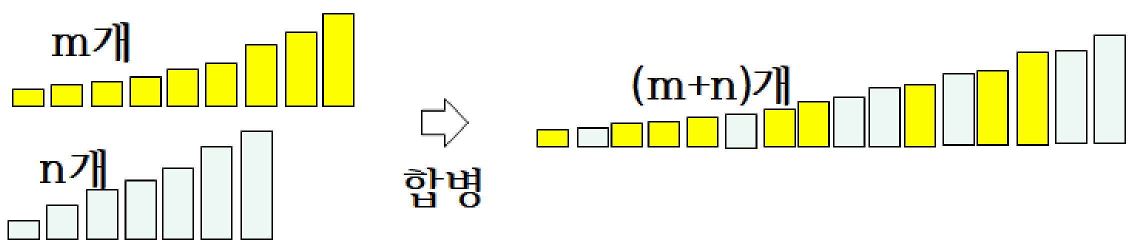
```
MergeSort(A,p,q)
입력: A[p]~A[q]
출력: 정렬된 A[p]~A[q]

if ( p < q ) {             // 배열의 원소의 수가 2개 이상이면
   k = ⌊(p+q)/2⌋           // k는 중간 원소의 인덱스
   MergeSort(A,p,k)        // 앞부분 순환 호출
   MergeSort(A,k+1,q)      // 뒷부분 순환 호출
   A[p]~A[k]와 A[k+1]~A[q]를 합병
}
```

n=8, A=[37, 10, 22, 30, 35, 13, 25, 24]



시간 복잡도

- 분할하는 부분은 배열의 중간 인덱스 계산과 2회의 순환 호출이므로 O(1) 시간 소요
- 합병의 수행 시간은 입력의 크기에 비례.
  - 2개의 정렬된 배열 A와 B의 크기가 각각 m과 n이라면, 최대 비교 횟수 = (m+n-1)
  - 합병의 시간 복잡도 = O(m+n)

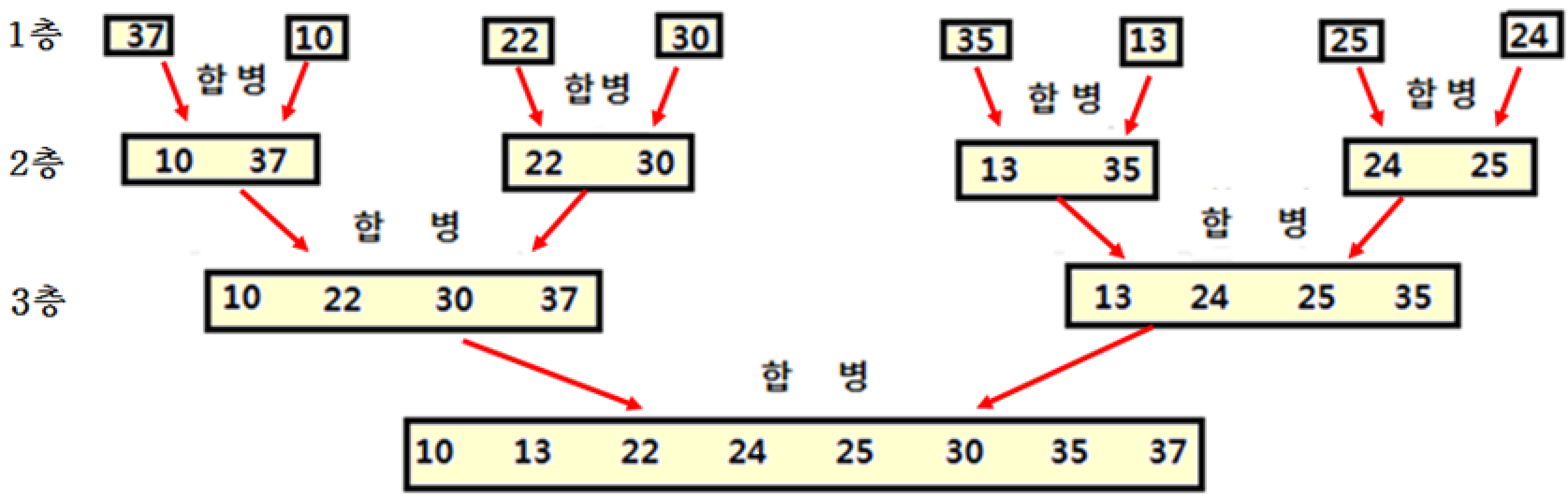


- 합병 정렬에서 수행되는 총 비교 횟수
  - 각각의 합병에 대해서 몇 번의 비교가 수행되었는지를 계산하여 이들을 모두 합한 수

시간 복잡도

➢ 층별 비교 횟수

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



시간 복잡도

층수의 계산

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

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

시간 복잡도

- 입력의 크기가 n일 때 몇 개의 층이 만들어질까?
  - n을 계속하여 1/2로 나누다가, 더 이상 나눌 수 없는 크기인 1이 될 때 분할을 중단한다.
  - 따라서 k번 1/2로 분할했으면 k개의 층이 생기는 것이고, k는 $2^k=n$으로부터 $\log_2 n$임을 알 수 있다.

- 합병 정렬의 시간 복잡도:
  - (층수) x O(n) = $\log_2 n$ x O(n) = O(nlogn)

합병 정렬의 단점

- 대부분의 정렬 알고리즘들은 입력을 위한 메모리 공간과 O(1) 크기의 메모리 공간만을 사용하면서 정렬 수행
  - O(1) 크기의 메모리 공간이란 입력 크기 n과 상관없는 크기의 공간(예를 들어, 변수, 인덱스 등)을 의미
- 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

Applications

- 합병 정렬은 외부 정렬의 기본이 되는 정렬 알고리즘
- 연결 리스트에 있는 데이터를 정렬할 때에도 퀵 정렬이나 힙 정렬 보다 훨씬 효율적
- 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 (Quick Sort)

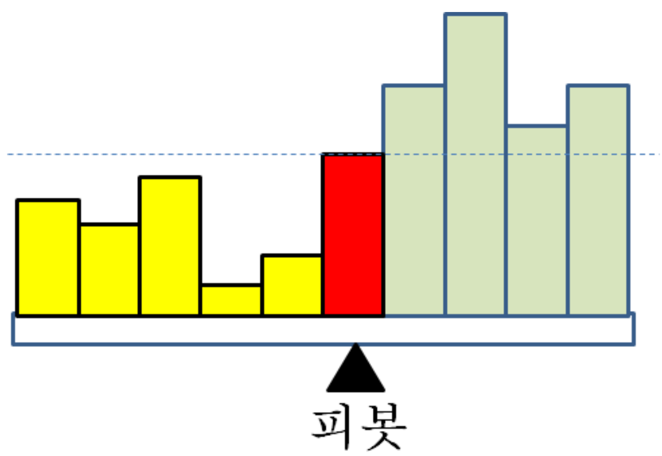
- 퀵 정렬은 분할 정복 알고리즘으로 분류
  - 사실 알고리즘이 수행되는 과정을 살펴보면 정복 후 분할하는 알고리즘

- 퀵 정렬 알고리즘은 문제를 2개의 부분 문제로 분할
  - 각 부분 문제의 크기가 일정하지 않은 형태의 분할 정복 알고리즘

아이디어

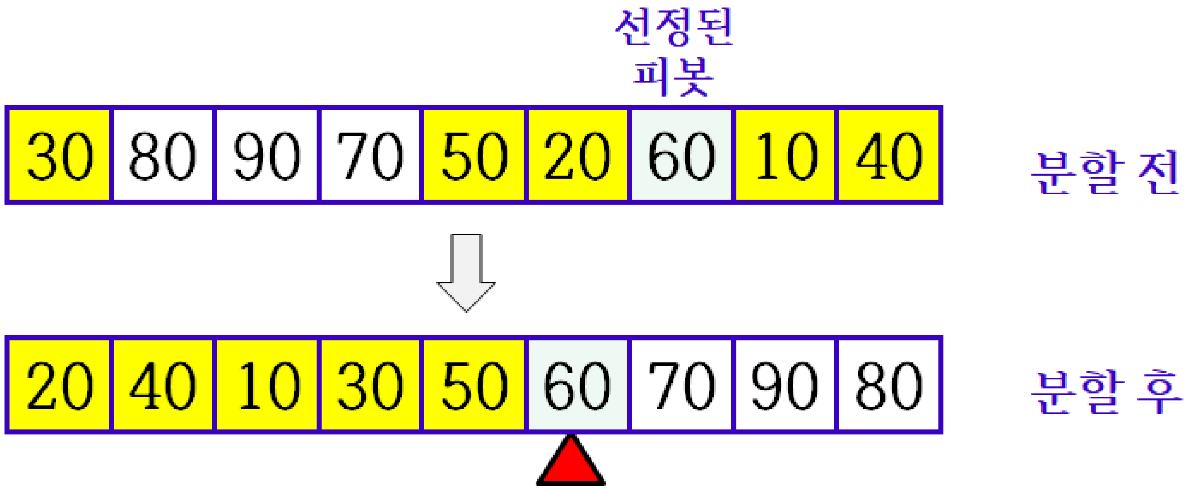
퀵 정렬의 아이디어

- 퀵 정렬은 피봇(pivot)이라 일컫는 배열의 원소(숫자)를 기준으로 피봇보다 작은 숫자들은 왼편으로, 피봇보다 큰 숫자들은 오른편에 위치하도록 분할하고, 피봇을 그 사이에 놓는다.
- 퀵 정렬은 분할된 부분문제들에 대해서도 위와 동일한 과정을 순환으로 수행하여 정렬



피봇

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



알고리즘

```
QuickSort(A, left, right)
입력: 배열 A[left]~A[right]
출력: 정렬된 배열 A[left]~A[right]
1. if (left < right) {
2.      피봇을 A[left]~A[right]에서 선택하고,
        피봇을 A[left]와 자리를 바꾼 후, 피봇과 배열의 각
        원소를 비교하여 피봇보다 작은 숫자들은
         A[left]~A[p-1]로 옮기고, 피봇보다 큰 숫자들은
         A[p+1]~A[right]로 옮기며,  피봇은 A[p]에 놓는다.
3.       QuickSort(A, left, p-1)   // 피봇보다 작은 그룹
4.       QuickSort(A, p+1 right)   // 피봇보다 큰 그룹
   }
```

수행 과정

`QuickSort(A, 0, 11)`

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

Line 2:

피봇 이동

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

비교 후 자리바꿈

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

수행 과정

Line 2: **피봇을 제자리로 이동**

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

순환 호출 `QuickSort(A, 0, 3)`

0	1	2	3
2	3	7	6

피봇 이동 및 자리바꿈

0	1	2	3
6	3	7	2

피봇을 제자리로 이동

0	1	2	3
2	3	6	7

0	1	2	3
2	3	6	7

QuickSort(A, 0, 1) 순환 호출

0	1
2	3

피봇 이동 자리바꿈

0	1
3	2

자리바꿈 없음

피봇을 제자리로 이동

0	1
2	3

순환 호출 QuickSort(A, 3, 3)

3
7

수행 과정

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

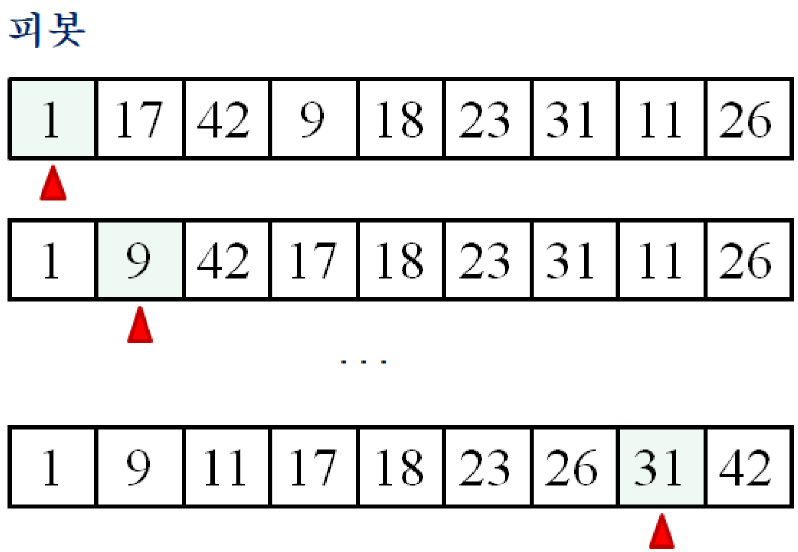
순환 호출

`QuickSort(A, 5, 11)`

⋮

시간 복잡도

- 퀵 정렬의 성능은 피봇 선택이 좌우한다. 피봇으로 가장 작은 숫자 또는 가장 큰 숫자가 선택되면, 한 부분으로 치우치는 분할을 야기
- 피봇으로 항상 가장 작은 숫자가 선택되는 경우



최악 경우 시간 복잡도

- 피봇=1일 때: 8회 - [17 42 9 18 23 31 11 26]과 각각 1회 비교
- 피봇=9일 때: 7회 - [42 17 18 23 31 11 26]과 각각 1회 비교
- 피봇=11일 때: 6회 - [17 18 23 31 42 26]과 각각 1회 비교

  …

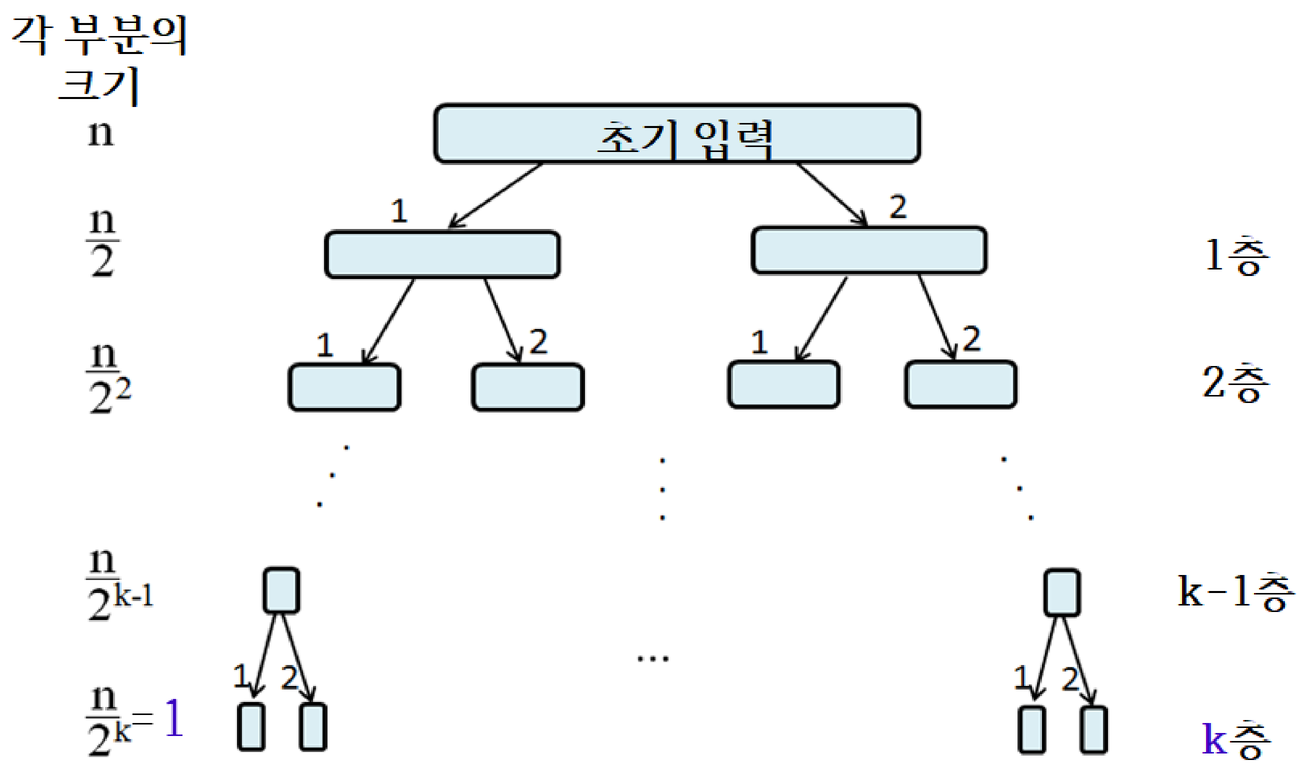
- 피봇=31일 때: 1회 - [42]와 1회 비교
- 총 비교 횟수는 8 + 7 + 6 + ⋯ + 1 = 36

➢ 퀵 정렬의 최악 경우 시간 복잡도

$$(n-1)+(n-2)+(n-3)+\cdots+2+1 = n(n-1)/2 = O(n^2)$$

최선 경우 시간 복잡도

- 최선 경우의 분할



- 각 층에서는 각각의 원소가 각 부분의 피봇과 1회씩 비교된다. 따라서 비교 횟수 = O(n)
- 총 비교 횟수 = O(n)x(층수) = O(n)x(logn)

  $n/2^k=1$일 때 $k=\log n$이므로

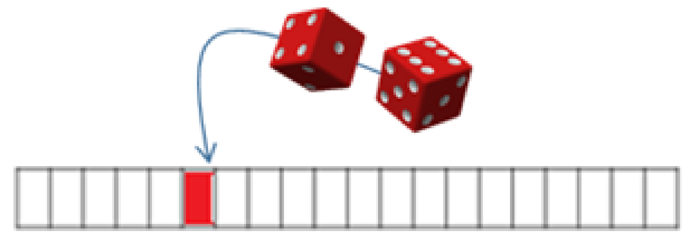
➢ 퀵 정렬의 최선 경우 시간 복잡도: O(nlogn)

평균 경우 시간 복잡도

- 평균 경우 시간 복잡도
  - 피봇을 항상 랜덤하게 선택한다고 가정하면, 퀵 정렬의 평균 경우 시간 복잡도를 계산할 수 있다.
  - 최선 경우와 동일한 O(nlogn)이다.

피봇 선정 방법

- 랜덤하게 선정하는 방법



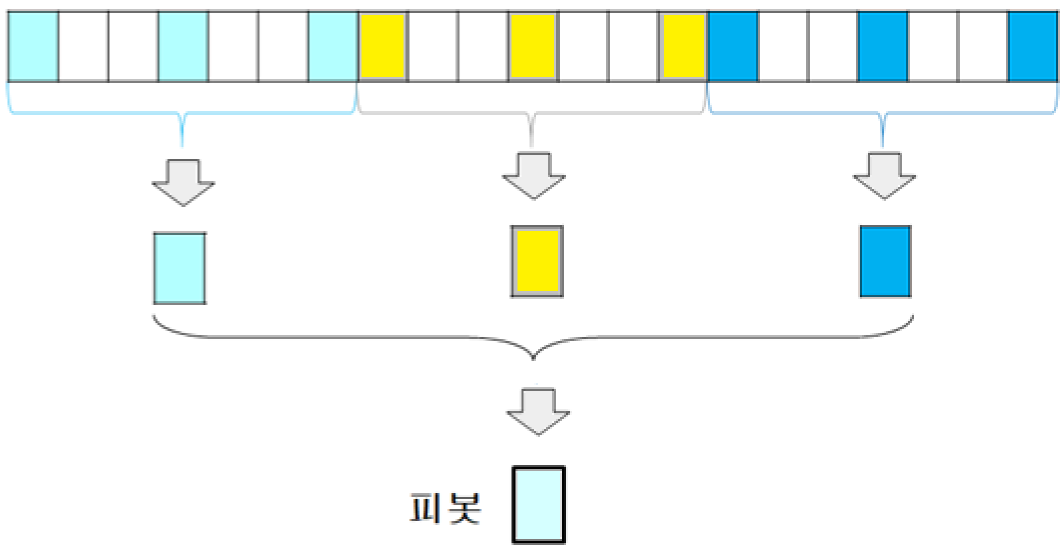
- 3 숫자의 중앙값으로 선정하는 방법(Median-of-Three)
  - 가장 왼쪽 숫자, 중간 숫자, 가장 오른쪽 숫자 중에서 중앙값으로 피봇을 정한다.
  - 아래의 예제를 보면, [31, 1, 26] 중에서 중앙값인 26을 피봇으로 사용

31		⋯		1		⋯		26

피봇 선정 방법

- Median-of-Medians(Tukey's Ninther)

  3 등분한 후 각 부분에서 가장 왼쪽 숫자, 중간 숫자, 가장 오른쪽 숫자 중에서 중앙값을 찾은 후, 세 중앙값들 중에서 중앙값을 피봇을 정한다.



성능 향상 방법

- 입력의 크기가 매우 클 때, 퀵 정렬의 성능을 더 향상시키기 위해서, 삽입 정렬을 동시에 사용
  - 입력의 크기가 작을 때에는 퀵 정렬이 삽입 정렬보다 빠르지만은 않다.

    퀵 정렬은 순환 호출로 수행되기 때문

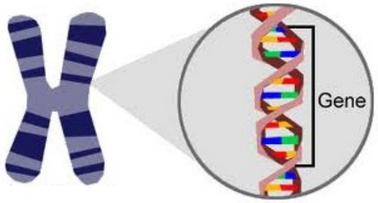
  - 부분 문제의 크기가 작아지면 (예를 들어, 25에서 50이 되면), 더 이상의 분할(순환 호출)을 중단하고 삽입 정렬을 사용

Applications

- 퀵 정렬은 커다란 크기의 입력에 대해서 가장 좋은 성능을 보이는 정렬 알고리즘이다.

- 퀵 정렬은 실질적으로 어느 정렬 알고리즘보다 좋은 성능을 보인다.

- 생물 정보 공학(Bioinformatics)에서 특정 유전자를 효율적으로 찾는데 접미 배열(suffix array)과 함께 퀵 정렬이 활용된다.



3.3 선택(Selection) 문제

- 선택 문제는 n개의 숫자들 중에서 k 번째로 작은 숫자를 찾는 문제
- 단순한 알고리즘
  - 최소 숫자를 k 번 찾는다.
    - 단, 최소 숫자를 찾은 뒤에는 입력에서 최소 숫자를 제거한다.
  - 숫자들을 정렬한 후, k번째 숫자를 찾는다.
  - 위의 알고리즘들은 각각 최악의 경우 O(kn)과 O(nlogn)의 수행 시간이 걸린다.

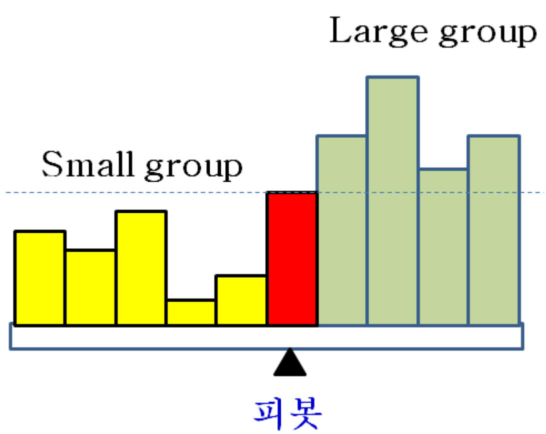
아이디어

이진 탐색

- 정렬된 입력의 중간에 있는 숫자와 찾고자 하는 숫자를 비교함으로써, 입력을 1/2로 나눈 두 부분 중에서 한 부분만을 검색

선택 문제

- 입력이 정렬되어 있지 않으므로, 입력 숫자들 중에서 (퀵 정렬과 같이) 피봇을 선택하여 분할



아이디어

- Small group은 피봇보다 작은 숫자의 그룹이고, Large group은 피봇보다 큰 숫자의 그룹
- 이렇게 분할했을 때 알아야 할 것은 각 그룹의 크기, 즉, 숫자의 개수
  - 각 그룹의 크기를 알면,
  - k번째 작은 숫자가 어느 그룹에 있는지를 알 수 있고,
  - 그 다음에는 그 그룹에서 몇 번째로 작은 숫자를 찾아야 하는지를 알 수 있다.

- Small group에 k번째 작은 숫자가 속한 경우
  - k번째 작은 숫자를 Small group에서 찾는다.
- Large group에 k번째 작은 숫자가 있는 경우
  - `(k-|Small group|-1)`번째로 작은 숫자를 Large group에서 찾아야 한다.
  - |Small group|은 Small group에 있는 숫자의 개수이고, 1은 피봇에 해당된다.

알고리즘

`Selection(A, left, right, k)`

입력: A[left]~A[right]와 k, 단, 1≤k≤|A|, |A|=right-left+1

출력: A[left]~A[right]에서 k 번째 작은 원소

1. 피봇을 A[left]~A[right]에서 랜덤하게 선택하고, 피봇과 A[left]의 자리를 바꾼 후, 피봇과 배열의 각 원소를 비교하여 피봇보다 작은 숫자는 A[left]~A[p-1]로 옮기고, 피봇보다 큰 숫자는 A[p+1]~ A[right]로 옮기며, 피봇은 A[p]에 놓는다.
2. S = (p-1)-left+1 // S = Small group의 크기
3. if (k ≤ S) Selection(A, left, p-1, k) // Small group에서 찾기
4. else if (k = S + 1) return A[p] // 피봇 = k번째 작은 숫자
5. else Selection(A, p+1, right, k-S-1) // Large group에서 찾기

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

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

피봇 이동

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

수행 과정

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

비교 후 자리바꿈

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

피봇을 제자리로 이동

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

수행 과정

Line 2: Small group의 크기 계산

S = (p-1)-left+1 = (4-1)-0+1 = 4

Selection(A,5,11,2) 호출

k=2, left=5, right=11

k-S-1 = 2

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

피봇 선택

피봇 이동

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

수행 과정

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

비교 후 자리바꿈

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

피봇을 제자리로 이동

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

수행 과정

S = (p-1)-left+1 = (9-1)-5+1 = 4

Selection(A, 5, 8, 2) 호출

k=2, left=5, right=8

5	6	7	8
10	9	12	11

피봇 선택

- 원소 간 자리바꿈 없이 아래와 같이 된다.

피봇을 제자리로 이동

5	6	7	8
9	10	12	11

수행 과정

- Line 2: Small group의 크기 계산

```
S = (p-1)-left+1 = (6-1)-5+1 = 1
```

5	6	7	8
9	10	12	11

- Line 3의 if-조건 (k ≤ S) = (2 ≤ 1)은 false
- Line 4의 if-조건 (2 = S+1) = (2 = 1+1) = (2 = 2)이 true 이므로

  최종적으로 A[6]=10을 7번째 작은 수로 리턴

Selection 알고리즘 고려 사항

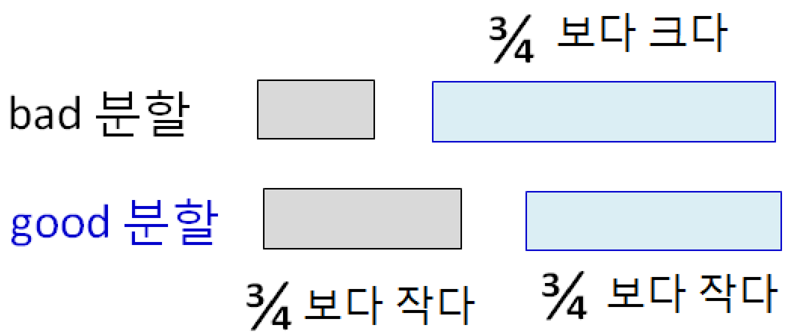
- Selection 알고리즘은 분할 정복 알고리즘이기도 하지만 랜덤(random) 알고리즘이기도 하다.
  - 선택 알고리즘의 line 1에서 피봇을 랜덤하게 정하기 때문

- 피봇이 입력을 너무 한쪽으로 치우치게 분할하면
  - 즉, |Small group| << |Large group| 또는 |Small group| >> |Large group|일 때에는 <u>알고리즘의 수행 시간이 길어진다.</u>

- 선택 알고리즘이 호출될 때마다 line 1에서 입력을 한쪽으로 치우치게 분할될 확률은?
  - 마치 동전을 던질 때 한쪽 면이 나오는 확률과 동일

good/bad 분할 정의

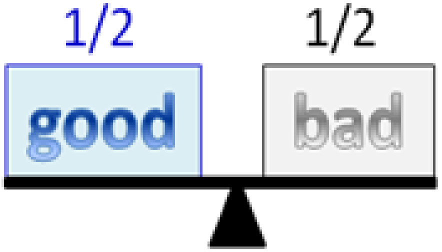
- 분할된 두 그룹 중의 하나의 크기가 입력 크기의 3/4과 같거나 그 보다 크게 분할하면 나쁜 (bad) 분할이라고 정의하자.
- 좋은 (good) 분할은 그 반대의 경우이다.



good/bad 분할

- 다음과 같이 16개의 숫자가 있다면

- good 분할이 되는 피봇을 선택할 확률과 bad 분할이 되는 피봇을 선택할 확률이 각각 1/2로 동일



시간 복잡도

- 피봇을 랜덤하게 정했을 때 good 분할이 될 확률이 1/2이므로 평균 2회 연속해서 랜덤하게 피봇을 정하면 good 분할을 할 수 있다.

- 매 2회 호출마다 good 분할이 되므로, good 분할만 연속하여 이루어졌을 때만의 시간 복잡도를 구하여, 그 값에 2를 곱하면 평균 경우 시간 복잡도를 얻을 수 있다.

연속된 good 분할

1번째 good 분할 — $\frac{3}{4}$보다 작다 $\frac{3}{4}$보다 작다

2번째 good 분할 — $\left(\frac{3}{4}\right)^2$보다 작다

3번째 good 분할 — $\left(\frac{3}{4}\right)^3$보다 작다

$\vdots$

i 번째 good 분할 — $\left(\frac{3}{4}\right)^i$ 보다 작다

평균 경우 시간 복잡도

- 입력 크기가 n에서부터 3/4배로 연속적으로 감소되고, 크기가 1일 때에는 더 이상 분할할 수 없다.
- $n + 3/4n + (3/4)^2 n + (3/4)^3 n + \cdots + (3/4)^{i-1} n + (3/4)^i n$

  $= n[1 + 3/4 + (3/4)^2 + (3/4)^3 + \cdots + (3/4)^{i-1} + (3/4)^i]$

  $\leq 2n = O(n)$
- 평균 2회에 good 분할이 되므로 $2 \times O(n) = O(n)$

선택 알고리즘과 이진 탐색

- [유사성]
  - 이진 탐색은 분할과정을 진행하면서 범위를 1/2씩 좁혀가며 찾고자 하는 숫자를 탐색
  - 선택 알고리즘은 피봇으로 분할하여 범위를 좁혀감

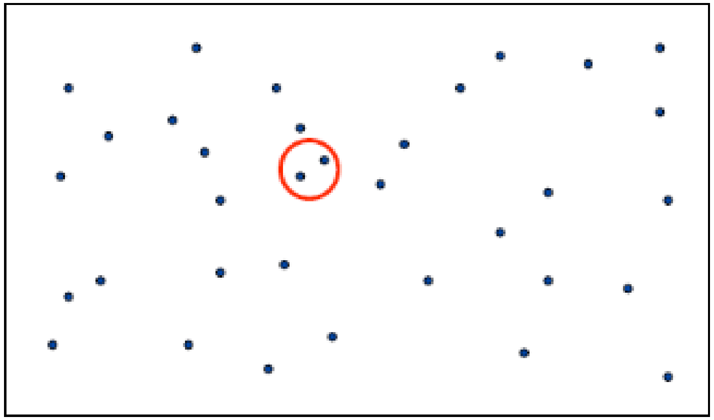
- [공통점]
  - 부분 문제들을 취합하는 과정이 별도로 필요 없다.

Applications

- 선택 알고리즘은 데이터 분석을 위한 중앙값 (median)을 찾는데 활용
  - 데이터 분석에서 평균값도 유용하지만, 중앙값이 더 설득력 있는 데이터 분석을 제공
  - 예를 들어, 대부분의 데이터가 1이고, 오직 1개의 숫자가 매우 큰 숫자 (노이즈 (noise), 잘못 측정된 데이터)이면, 평균값은 매우 왜곡된 분석이 된다.

3.4 최근접 점의 쌍 찾기

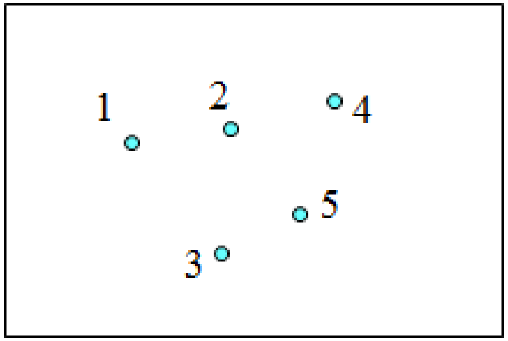
- 최근접 점의 쌍 (Closest Pair) 문제는 2차원 평면상의 n개의 점이 입력으로 주어질 때, 거리가 가장 가까운 한 쌍의 점을 찾는 문제



최근접 점의 쌍 찾기

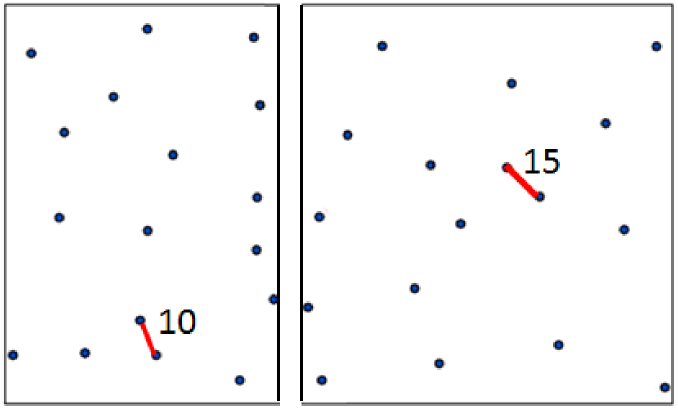
간단한 방법

- 모든 점에 대하여 각각의 두 점 사이의 거리를 계산하여 가장 가까운 점의 쌍을 찾는다.
- 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 된다.
- 비교해야 할 쌍은 몇 개인가?
  - ${}_nC_2$ = n(n-1)/2
  - n=5이면, 5(5-1)/2 = 10
  - n(n-1)/2 = O(n^2)
  - 한 쌍의 거리 계산은 O(1)
  - 시간 복잡도는 O(n^2)xO(1) = O(n^2)

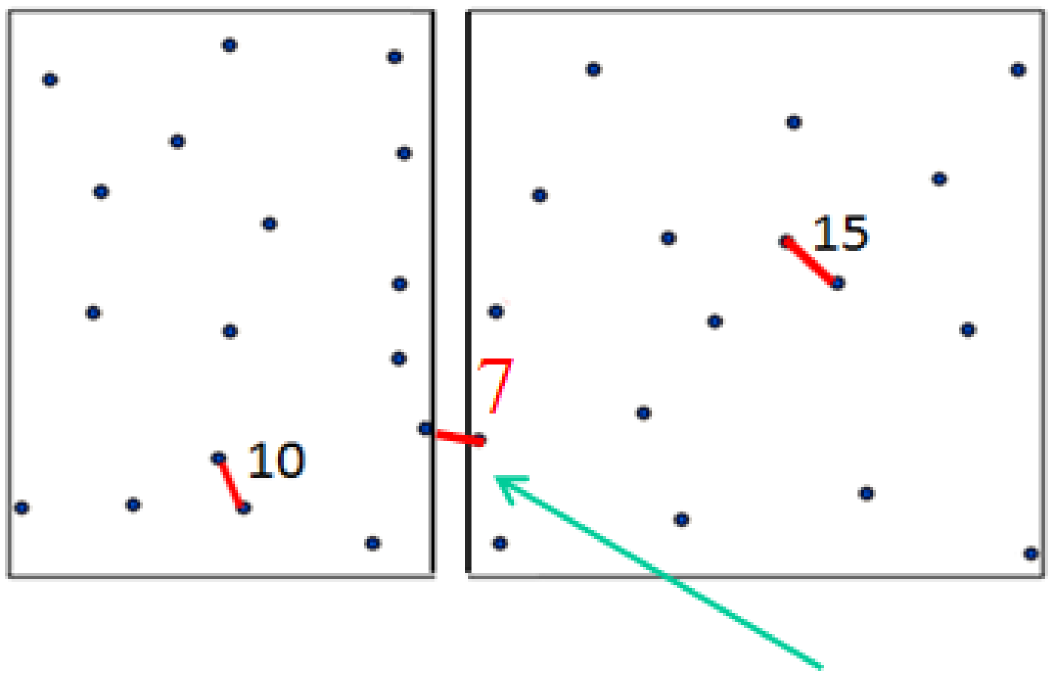


최근접 점의 쌍 찾기

- O(n^2)보다 효율적인 분할 정복 이용
  - n개의 점을 1/2로 분할하여 각각의 부분 문제에서 최근접 점의 쌍을 찾고, 2개의 부분 해 중에서 짧은 거리를 가진 점의 쌍을 일단 찾는다.

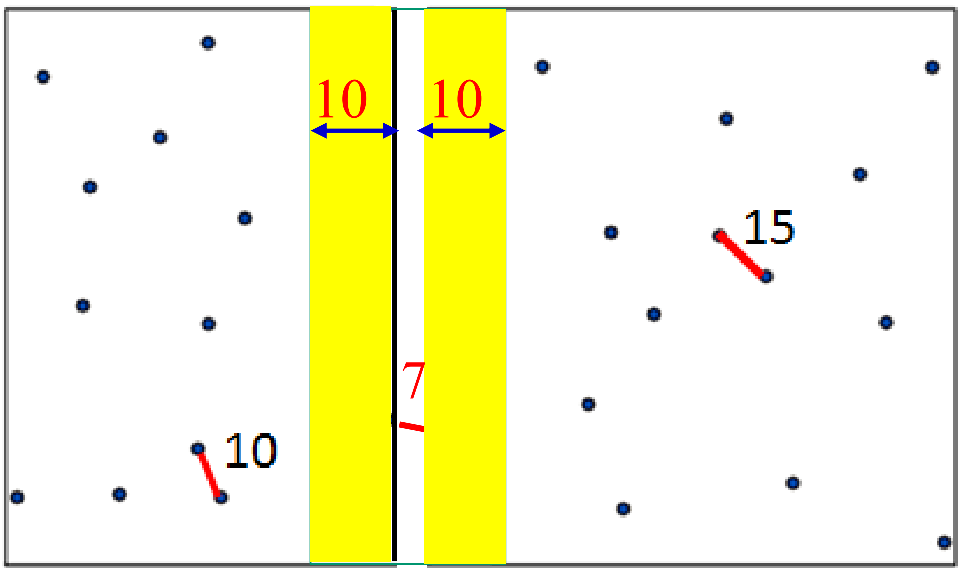


취합할 때 중간 영역을 고려해야



중간 영역 안의 점들

- 10과 15 중에서 짧은 거리인 10 이내의 중간 영역 안에 있는 점들 중에 더 근접한 점의 쌍이 있는지 확인해야



중간 영역에 있는 점들을 찾는 방법

- d = min{왼쪽 부분의 최근접 점의 쌍 사이의 거리, 오른쪽 부분의 최근접 점의 쌍 사이의 거리}
- 배열에는 점들이 x-좌표의 오름차순으로 정렬되어 있고, 각 점의 y-좌표는 생략

왼쪽 부분문제의
가장 오른쪽 점

오른쪽 부분문제의
가장 왼쪽 점

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

중간 영역에 있는 점들을 찾는 방법

- 중간 영역에 속한 점 = {왼쪽 부분 문제의 가장 오른쪽 점 (왼쪽 중간점)의 x-좌표에서 d를 뺀 값과 오른쪽 부분 문제의 가장 왼쪽 점 (오른쪽 중간점)의 x-좌표에 d를 더한 값 사이의 x-좌표 값을 가진 점들}
- d=10이면, 점 (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

알고리즘

ClosestPair(S)

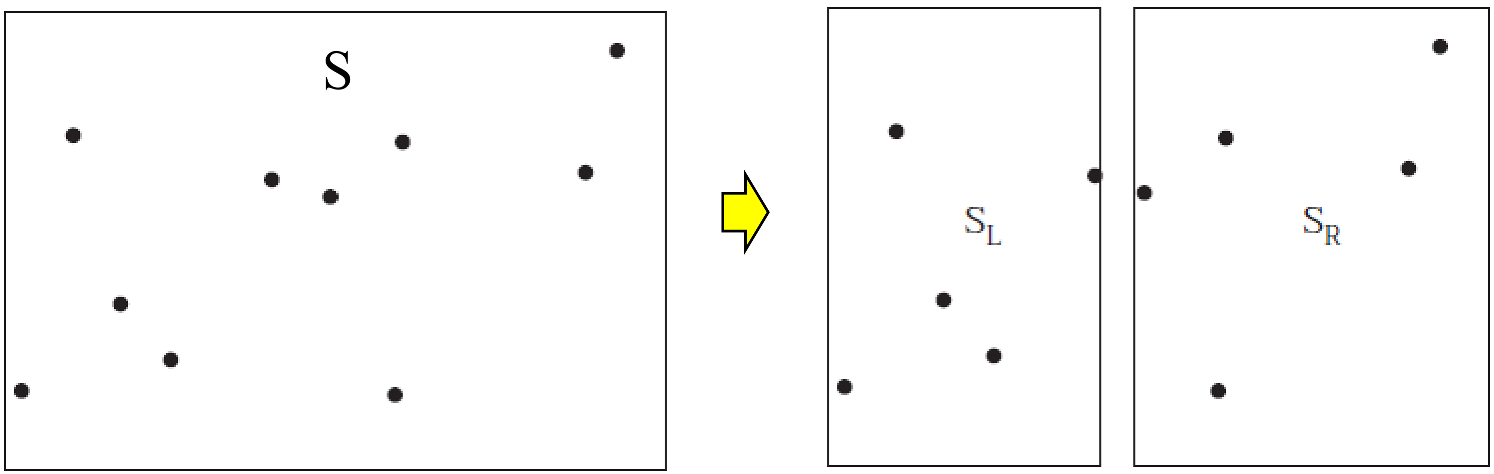
입력: x-좌표의 오름차순으로 정렬된 배열 S에 있는 i개의 점. 단, 각 점은 (x,y)로 표현

출력: S에 있는 점들 중 최근접 점의 쌍의 거리

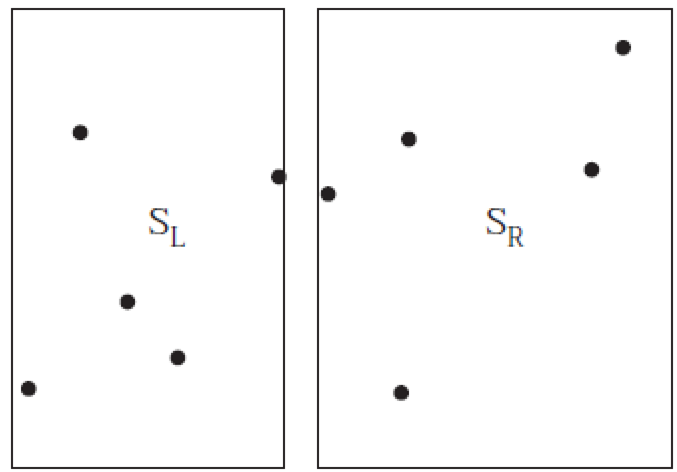
1. if (i ≤ 3) return (2 또는 3개의 점들 사이의 최근접 쌍)
2. 정렬된 S를 같은 크기의 S_L과 S_R로 분할한다. |S|가 홀수이면, $|S_L| = |S_R|+1$이 되도록 분할
3. CP_L = ClosestPair(S_L) // CP_L은 S_L에서의 최근접 점의 쌍
4. CP_R = ClosestPair(S_R) // CP_R은 S_R에서의 최근접 점의 쌍
5. d = min{dist(CP_L), dist(CP_R)}일 때, 중간 영역에 속하는 점들 중에서 최근접 점의 쌍을 찾아서 이를 CP_C라고 하자. 단, dist()는 두 점 사이의 거리
6. return (CP_L, CP_C, CP_R 중에서 거리가 가장 짧은 쌍)

수행 과정

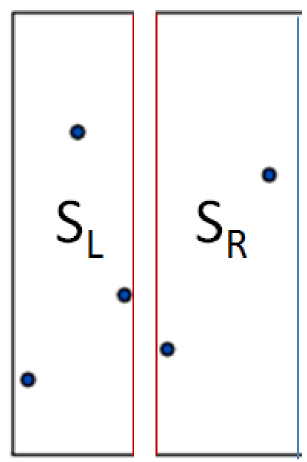
- ClosestPair(S)로 호출 [1]
  - Line 1: S의 점의 수 > 3이므로 다음 line을 수행
  - Line 2: S를 S_L과 S_R로 분할



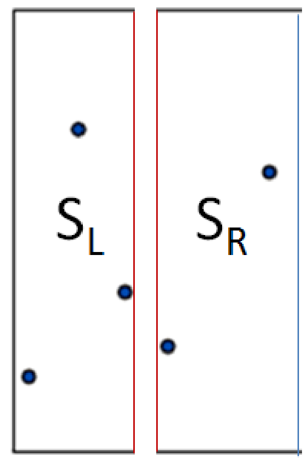
수행 과정



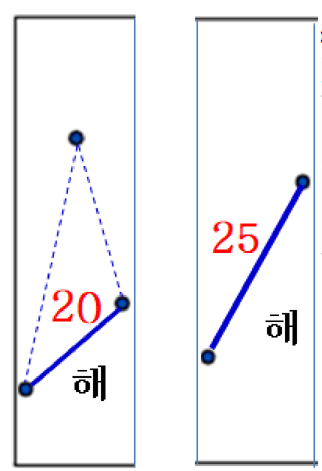
ClosestPair(S_L) 호출 [2]



수행 과정

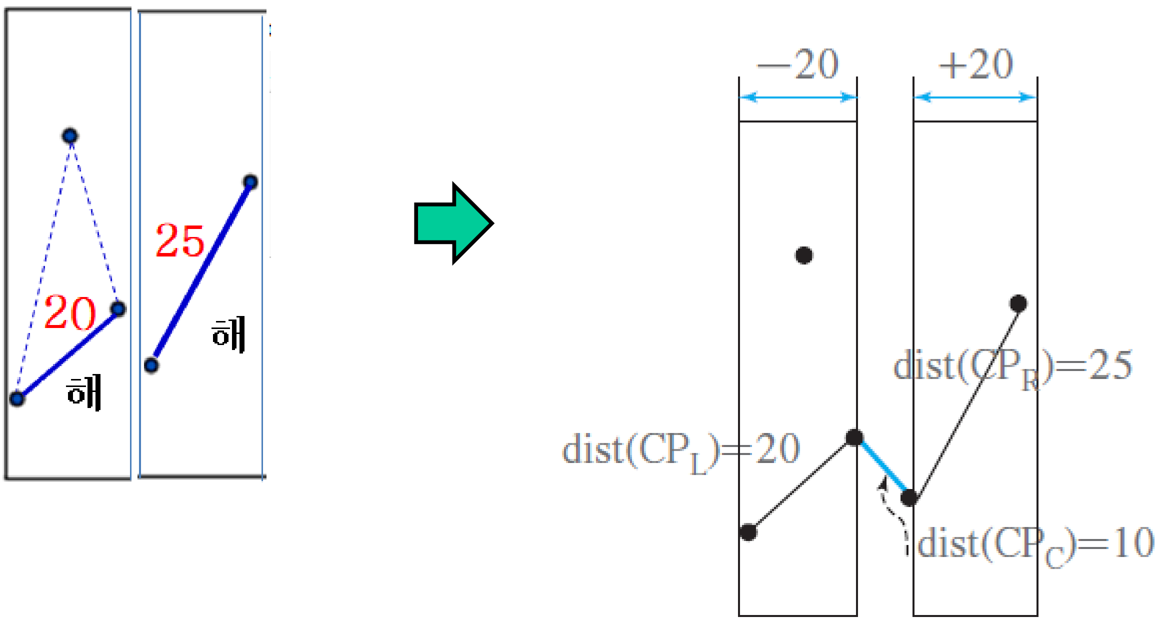


ClosestPair(S_L) 호출 ClosestPair(S_R) 호출



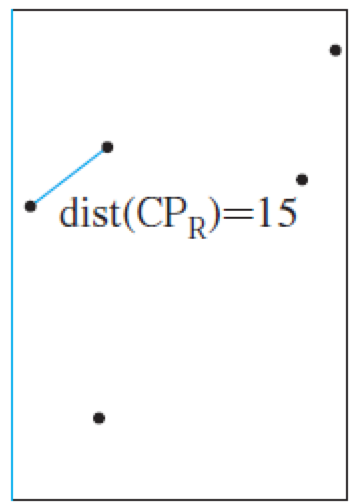
수행 과정

[2]의 ClosestPair(S_L) 호출 당시 line 3~4가 수행되었고, 이제 line 5를 수행, 즉, 중간 영역에서 CP_C찾음



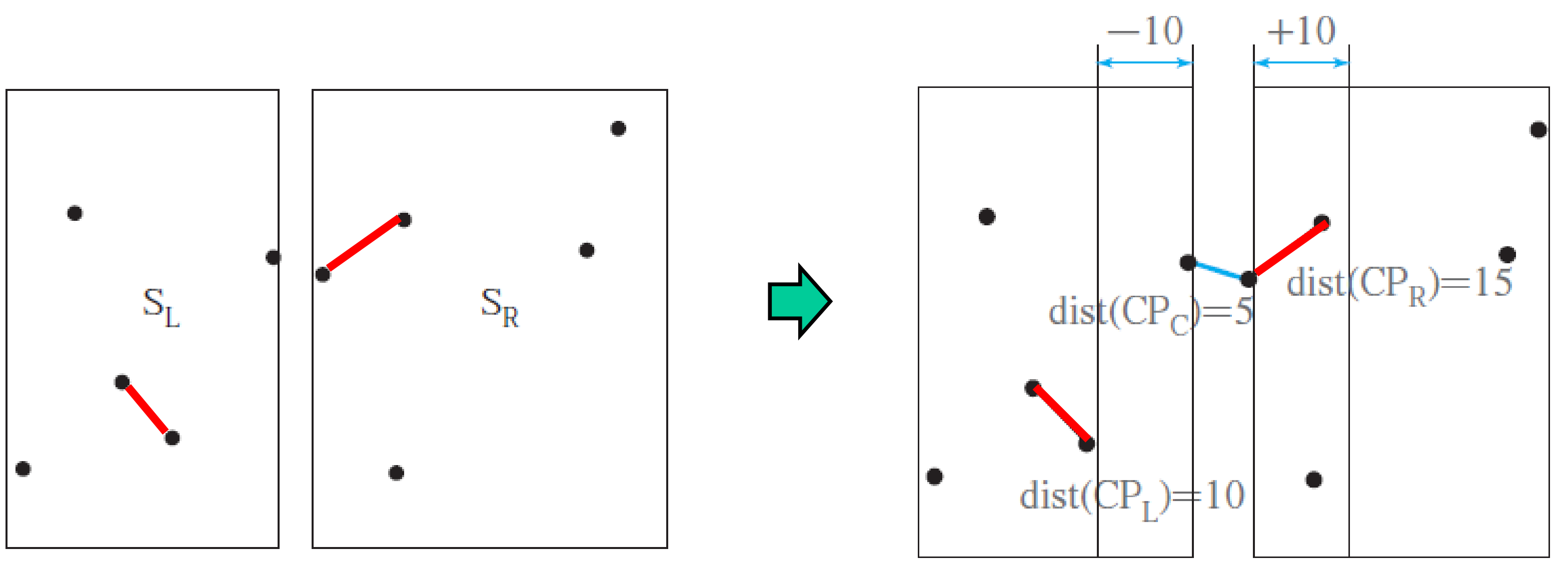
수행 과정

- [1]의 ClosestPair(S) 호출 당시 line 3이 수행되었고, 이제 line 4에서는 ClosestPair(S_R) 호출 결과



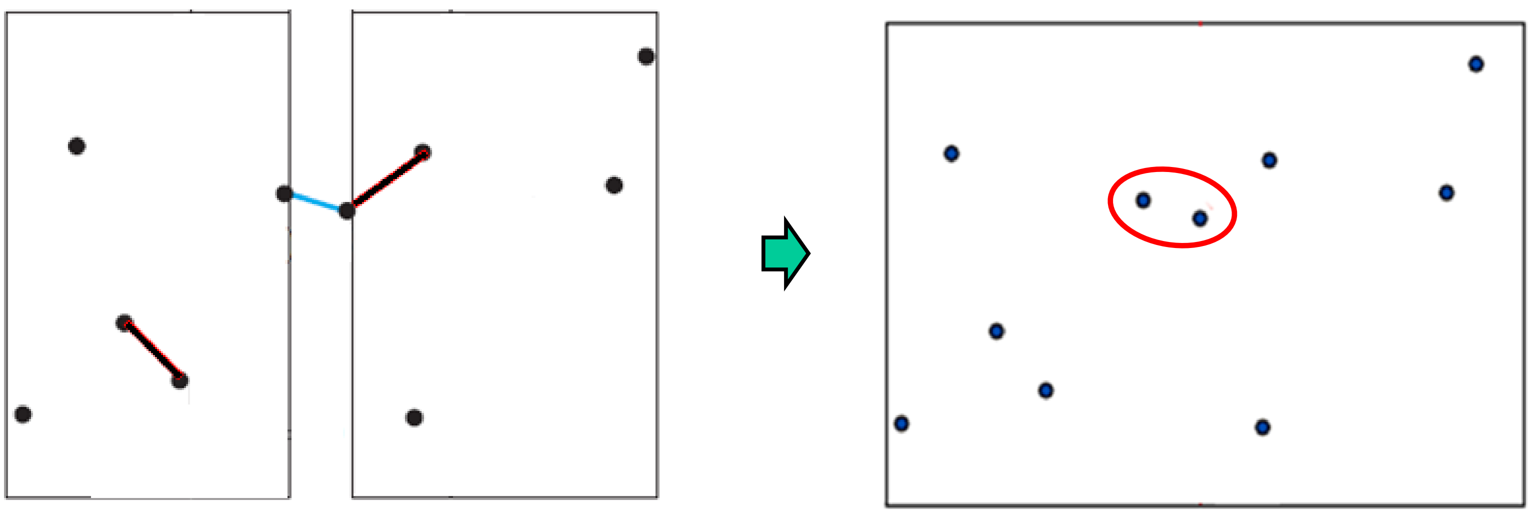
수행 과정

- Line 5: line 3~4에서 찾은 최근접 점의 쌍 사이의 거리인 dist(CP_L)=10과 dist(CP_R)=15 중에 작은 값을 d이라고 놓는다.
- 중간 영역에 있는 점들 중에서 CP_C를 찾는다.

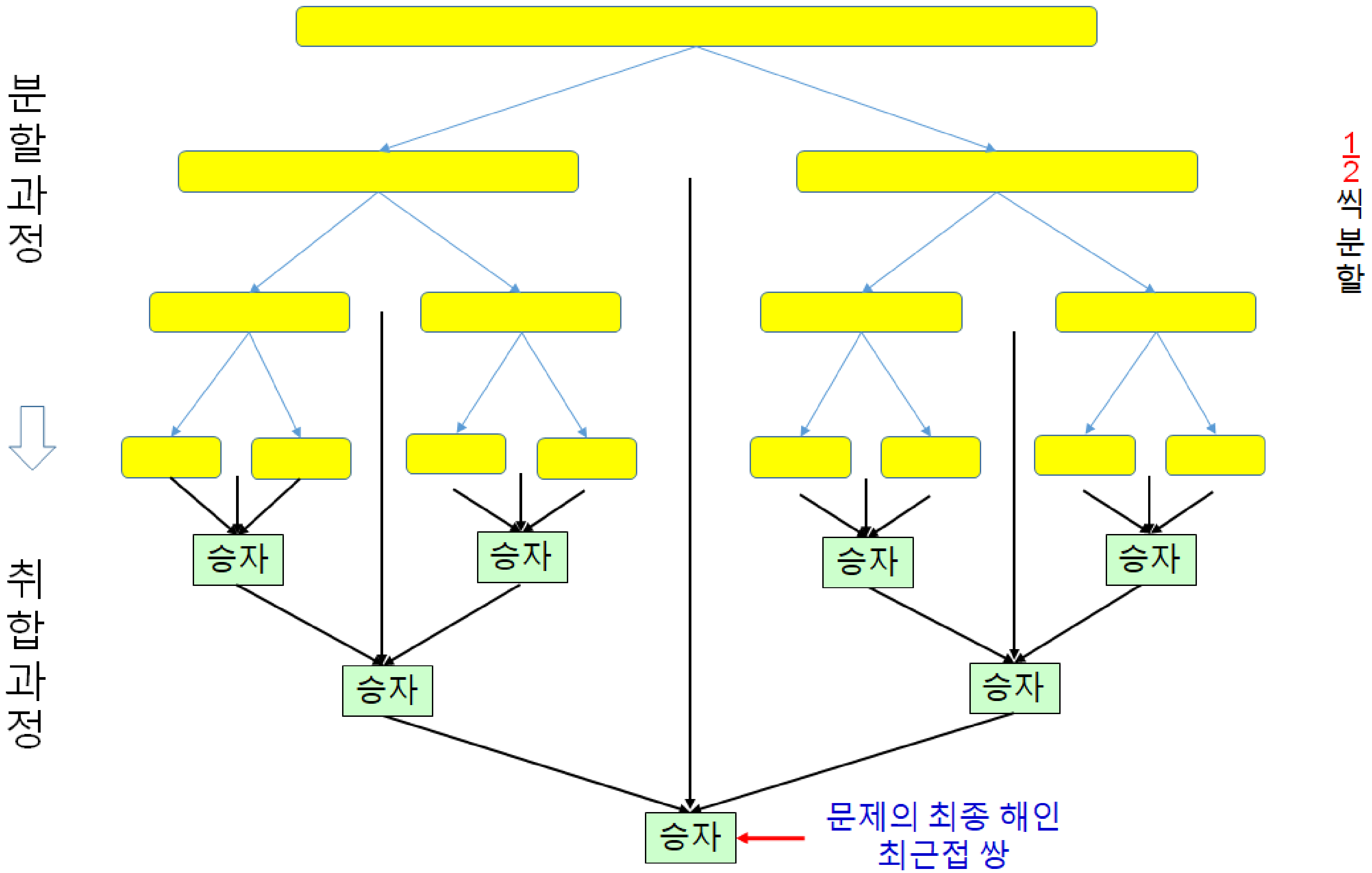


수행 과정

- Line 6: dist(CP_L)=10, dist(CP_C)=5, dist(CP_R)=15 중에서 가장 거리가 짧은 쌍인 CP_C를 최근접 쌍의 점으로 최종적으로 리턴



분할 정복 개념도

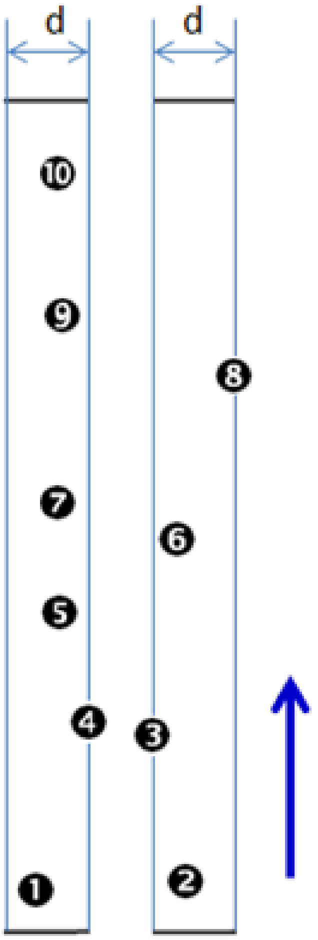


시간 복잡도

- S에 n개의 점이 있으면 전처리 (preprocessing) 과정으로서 S의 점을 x-좌표로 정렬: O(nlogn)
- Line 1: S에 3개의 점이 있는 경우에 3번의 거리 계산이 필요하고, S의 점의 수가 2이면 1번의 거리 계산이 필요하므로 O(1) 시간이 걸린다.
- Line 2: 정렬된 S를 S_L과 S_R로 분할하는데, 이미 배열에 정렬되어 있으므로, 배열의 중간 인덱스로 분할하면 된다. 이는 O(1) 시간 걸린다.
- Line 3~4: S_L과 S_R에 대하여 각각 ClosestPair를 호출하는데, 분할하며 호출되는 과정은 합병 정렬과 동일

시간 복잡도

- Line 5
  - d = min{dist(CP_L), dist(CP_R)}일 때 중간 영역에 속하는 점들 중에서 최근접 점의 쌍을 찾는다.
  - 이를 위해 먼저 중간 영역에 있는 점들을 y-좌표 기준으로 정렬한 후에, 아래에서 위로 각 점을 기준으로 거리가 d이내인 주변의 점들 사이의 거리를 각각 계산하며, 이 영역에 속한 점들 중에서 최근접 점의 쌍을 찾는다.
  - y-좌표로 정렬하는데 O(nlogn) 시간이 걸리고, 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 때문



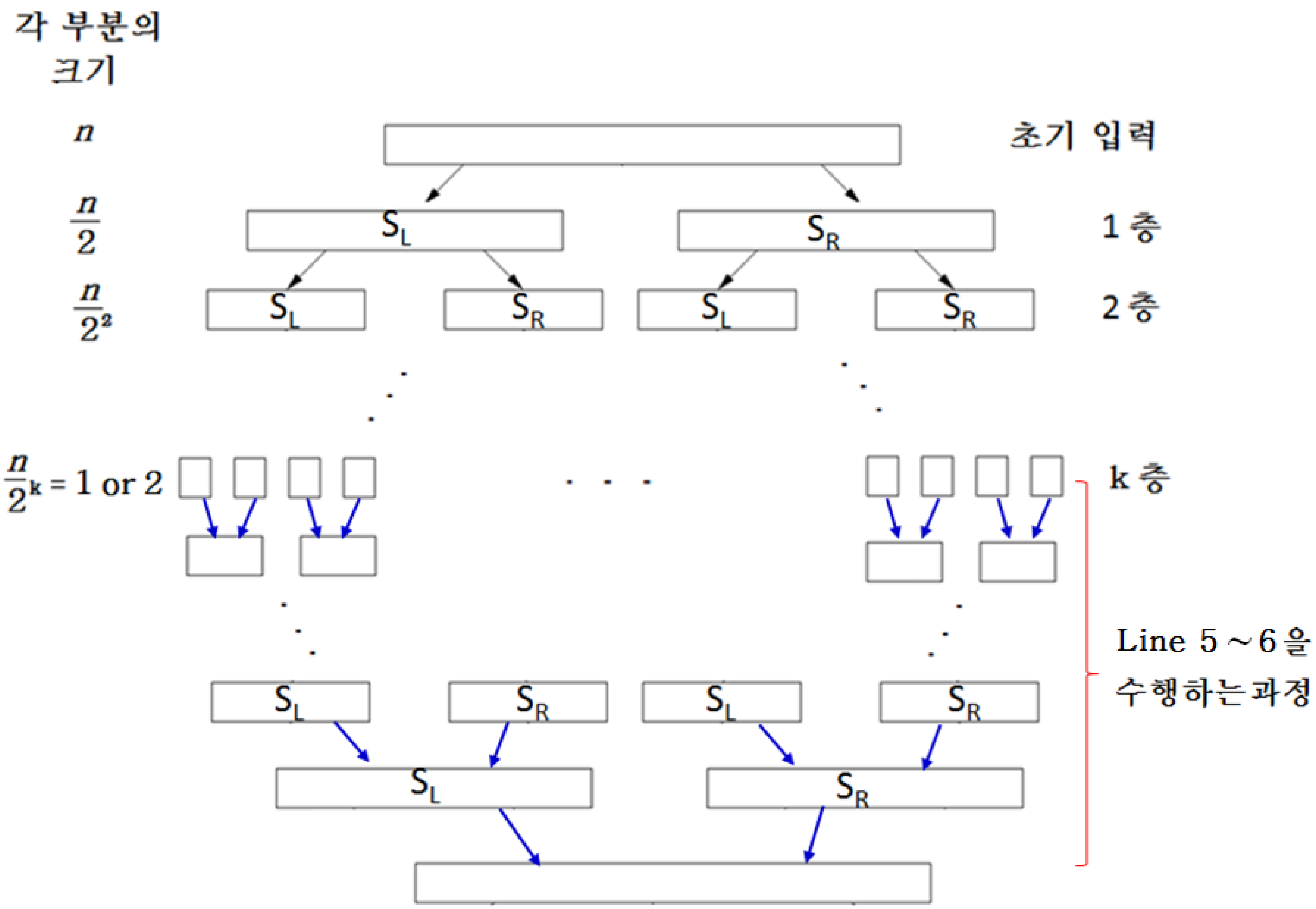
시간 복잡도

- Line 6: 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(nlogn) 시간이 필요

- k층까지 분할된 후, 층별로 line 5~6이 수행되는 (취합) 과정을 보여준다. (다음 슬라이드)
  - 각 층의 수행 시간은 O(nlogn)
  - 여기에 층 수인 logn을 곱하면 $O(n\log^2 n)$

시간 복잡도 개요



Applications

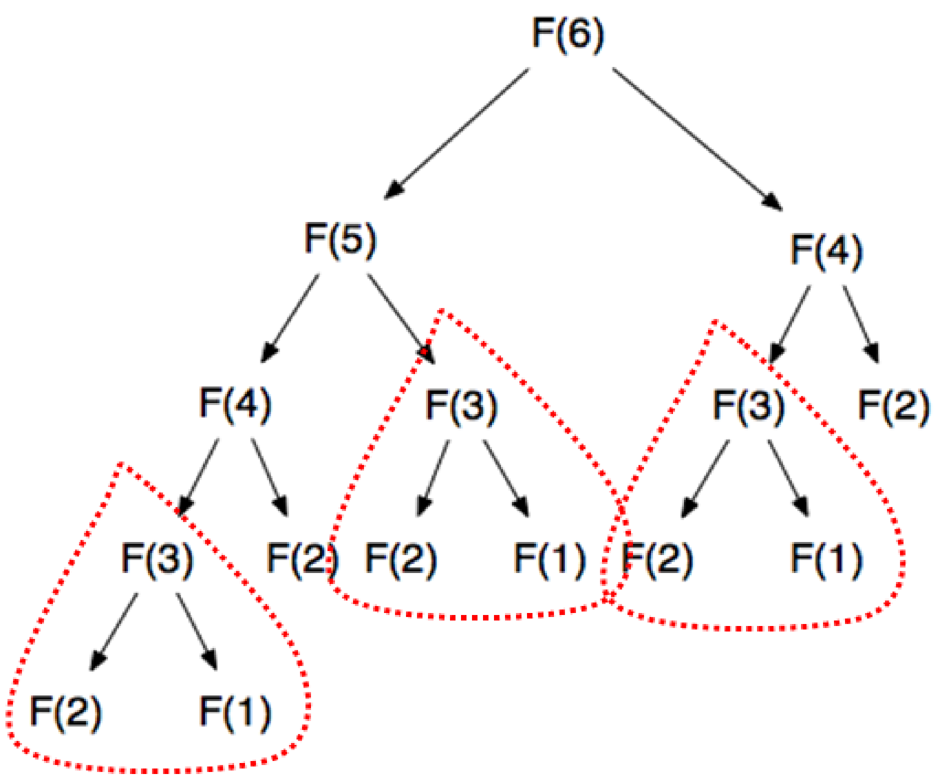
- 컴퓨터 그래픽스
- 컴퓨터 비전 (Vision)
- 지리 정보 시스템 (Geographic Information System, GIS)
- 분자 모델링 (Molecular Modeling)
- 항공 트래픽 조정 (Air Traffic Control)
- 마케팅 (주유소, 프랜차이즈 신규 가맹점 등의 위치 선정) 등

3.5 분할 정복 적용에 있어 주의할 점

- 분할 정복이 부적절한 경우
  - 입력이 분할될 때마다 분할된 부분 문제의 입력 크기의 합이 분할되기 전의 입력 크기보다 커지는 경우
- n 번째의 피보나치 수를 구하기
  - F(n) = F(n-1) + F(n-2)로 정의되므로 순환 호출을 사용하는 것이 자연스러워 보이나, 이 경우의 입력은 1개이지만, 사실상 n의 값 자체가 입력 크기인 것이다.
  - 2개의 부분 문제인 F(n-1)과 F(n-2)의 입력 크기는 (n-1) + (n-2) = (2n-3)이 되어서, 분할 후 입력 크기가 거의 2배로 증가함

피보나치 수

- 피보나치 수 F(6)을 구하기 위해 분할된 부분 문제들
  - F(2)를 5번이나 중복하여 계산해야 하고, F(3)은 3번 계산된다.



피보나치 수

➢ 피보나치 수 계산을 위한 O(n) 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]
```

분할 정복 적용에 있어 주의할 점

- 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.
- 기하(geometry)에 관련된 다수의 문제들이 효율적인 분할 정복 알고리즘으로 해결되는데, 이는 기하 문제들의 특성상 취합 과정이 문제 해결에 잘 부합되기 때문

요약

- 분할 정복 (Divide-and-Conquer) 알고리즘: 주어진 문제의 입력을 분할하여 문제를 해결 (정복)하는 방식의 알고리즘이다.

- 합병 정렬 (Merge sort): n개의 숫자들을 n/2개씩 2개의 부분 문제로 분할하고, 각각의 부분 문제를 재귀적으로 합병 정렬한 후, 2개의 정렬된 부분을 합병하여 정렬 (정복)한다. 시간 복잡도는 O(nlogn)이다.

- 합병 정렬의 공간 복잡도는 O(n)이다.

- 퀵 정렬 (Quick sort): 피봇 (pivot)이라 일컫는 배열의 원소를 기준으로 피봇보다 작은 숫자들은 왼편으로, 피봇보다 큰 숫자들은 오른편에 위치하도록 분할하고, 피봇을 그 사이에 놓는다. 퀵 정렬은 분할된 부분 문제들에 대하여서도 위와 동일한 과정을 순환으로 수행하여 정렬

요약

- 퀵 정렬의 평균 경우 시간 복잡도는 $O(n\log n)$, 최악 경우 시간 복잡도는 $O(n^2)$, 최선 경우 시간 복잡도는 $O(n\log n)$

- 선택 (Selection) 문제: k 번째 작은 수를 찾는 문제로서, 입력에서 퀵 정렬에서와 같이 피봇을 선택하여 피봇보다 작은 부분과 큰 부분으로 분할한 후에 k 번째 작은 수가 들어있는 부분을 순환으로 탐색한다. 평균 경우 시간 복잡도는 $O(n)$

- 최근접 점의 쌍 (Closest Pair) 문제: n개의 점들을 1/2로 분할하여 각각의 부분 문제에서 최근접 점의 쌍을 찾고, 2개의 부분 해 중에서 짧은 거리를 가진 점의 쌍을 일단 찾는다. 그리고 2개의 부분해를 취합할 때, 중간 영역 안에 있는 점들 중에 최근접 점의 쌍이 있는지도 확인해야 한다. 시간 복잡도는 $O(n\log^2 n)$

요약

- 분할 정복이 부적절한 경우는 입력이 분할될 때마다 분할된 부분 문제들의 입력 크기의 합이 분할되기 전의 입력 크기보다 커지는 경우이다. 또 하나 주의해야 할 요소는 취합 (정복) 과정이다.