Chapter 3. 분할 정복 알고리즘

분할 정복(Divide-and-Conquer) 알고리즘

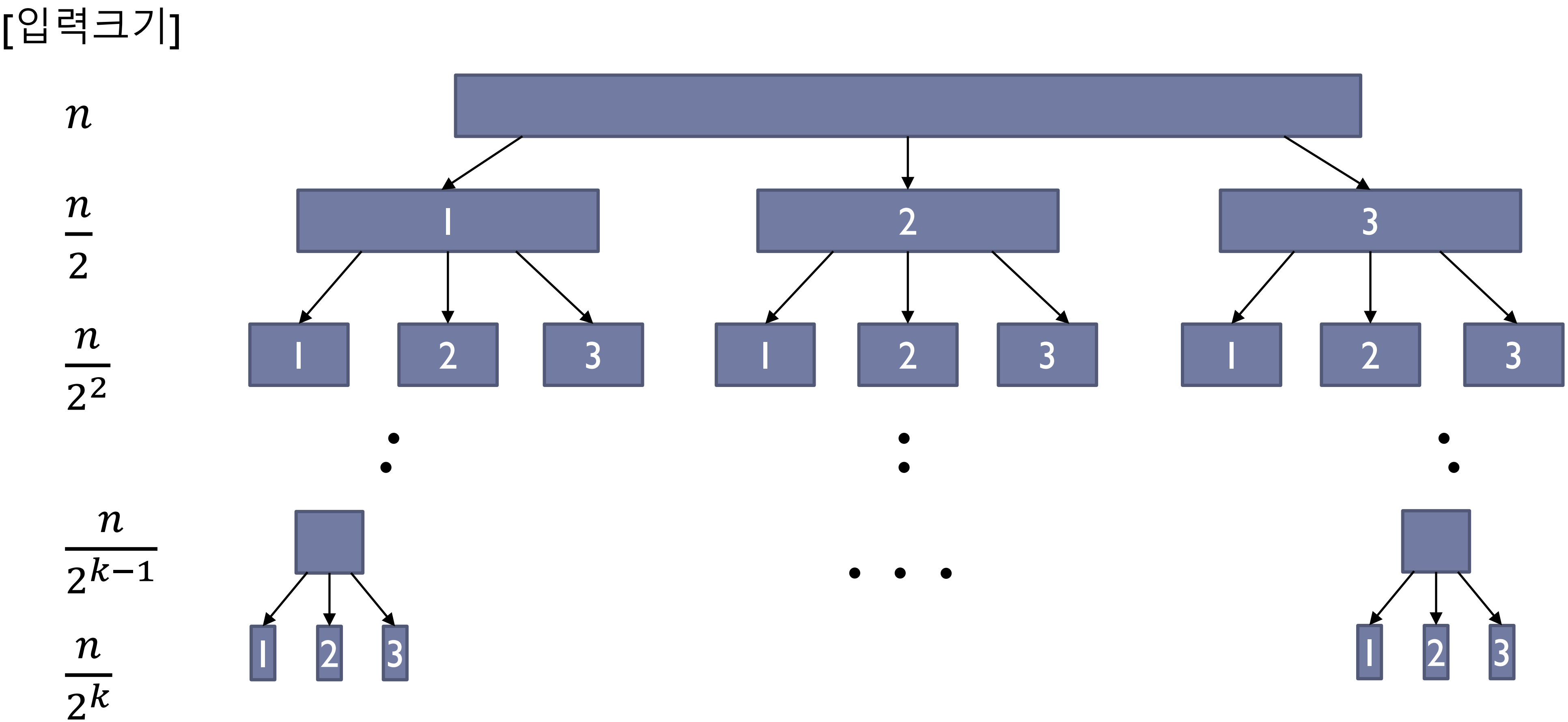
- 주어진 문제의 입력을 분할하여 문제를 해결(정복)하는 방식의 알고리즘이다.
- 분할한 입력에 대하여 동일한 알고리즘을 적용하여 해를 계산하며, 이들의 해를 취합하여 원래 문제의 해를 얻는다.
- 분할된 입력에 대한 문제를 부분문제(subproblem)라고 하고, 부분문제의 해를 부분해라고 한다.
- 부분문제는 더 이상 분할할 수 없을 때까지 계속 분할한다.

2 5 | 7 6 | 3 2 | 4 1

5 7 | 3 4

7 4

분할 정복(Divide-and-Conquer) 알고리즘



분할 정복(Divide-and-Conquer) 알고리즘

- 입력 크기가 n일 때 총 몇 번 분할하여야 더 이상 분할할 수 없는 크기인 1이 될까?
  - 답을 계산하기 위해서 총 분할한 횟수= k라고 하자.
  - 1번 분할 후 각각의 입력 크기가 $n/2$
  - 2번 분할 후 각각의 입력 크기가 $n/2^2$
  - …
  - k번 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

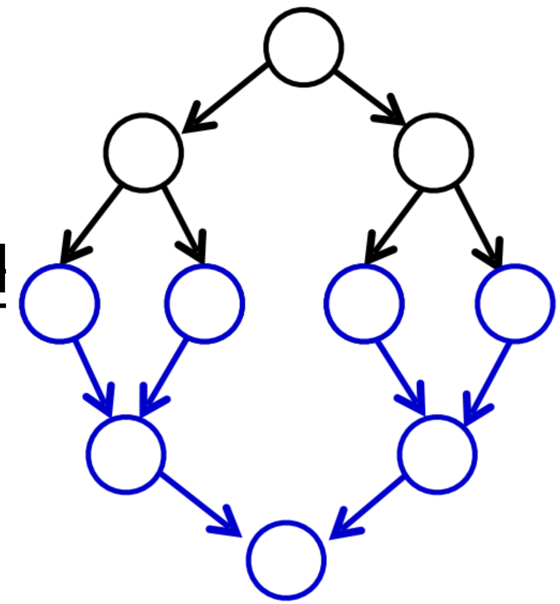
- 대부분의 분할 정복 알고리즘은 문제의 입력을 단순히 분할만 해서는 해를 구할 수 없다.

- 따라서 분할된 부분문제들을 정복해야 한다. 즉, 부분해를 찾아야 한다.

- 정복하는 방법은 문제에 따라 다르나 일반적으로 부분문제들의 해를 취합하여 보다 큰 부분문제의 해를 구한다.

효율적인DAH

분할 과정



정복 (취합) 과정

분할 정복 알고리즘의 분류

- 문제가 a개로 분할되고, 부분문제의 크기가 1/b로 감소하는 알고리즘
  - a=b=2 합병 정렬(3.1절), 최근접 점의 쌍 찾기(3.4절), 공제선 문제
  - a=3, b=2 큰 정수의 곱셈
  - a=4, b=2 큰 정수의 곱셈
  - a=7, b=2 스트라센(Strassen)의 행렬 곱셈 알고리즘
- 문제가 2개로 분할, 부분문제의 크기가 일정하지 않은 크기로 감소하는 알고리즘
  - 퀵 정렬(3.2절)

분할 정복 알고리즘의 분류

- 문제가 2개로 분할, 그 중에 1개의 부분문제는 고려할 필요 없으며, 부분문제의 크기가 1/2로 감소하는 알고리즘
  - 이진탐색(1.2절) 하나는 안찾아도 된다.
- 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)

3.1 합병 정렬

- 합병 정렬(Merge Sort)은 입력이 2개의 부분문제로 분할, 부분문제의 크기가 1/2로 감소하는 분할 정복 알고리즘이다.

- n개의 숫자들을 n/2개씩 2개의 부분문제로 분할하고, 각각의 부분문제를 재귀적으로 합병 정렬한 후, 2개의 정렬된 부분을 합병하여 정렬(정복)한다.

- 합병 과정이 (문제를) 정복하는 것이다.

앞 50개 . 뒤 50개 정렬은 X . 2개를 합쳐서

이미 정렬되어있는 순차그룹

합병(merge)

- 합병이란 2개의 각각 정렬된 숫자들을 1개의 정렬된 숫자들로 합치는 것이다.

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

맨 앞에 있는 1개를 서로 비교

합병 정렬 알고리즘

n-1번

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⌋            // k=반으로 나누기 위한 중간 원소의 인덱스
  MergeSort(A, p, k)       // 앞부분 재귀 호출
  MergeSort(A, k+1, q)     // 뒷부분 재귀 호출
  A[p]~A[k]와 A[k+1]~A[q]를 합병한다.
}
```

합병 정렬 알고리즘

Line 1: 정렬할 부분의 원소의 수가 2개 이상일 때에만 다음 단계가 수행되도록 한다. 만일 p=q (즉, 원소의 수가 1) 이면, 그 자체로 정렬된 것이므로 어떤 수행할 필요없이 이전 호출했던 곳으로 리턴한다.

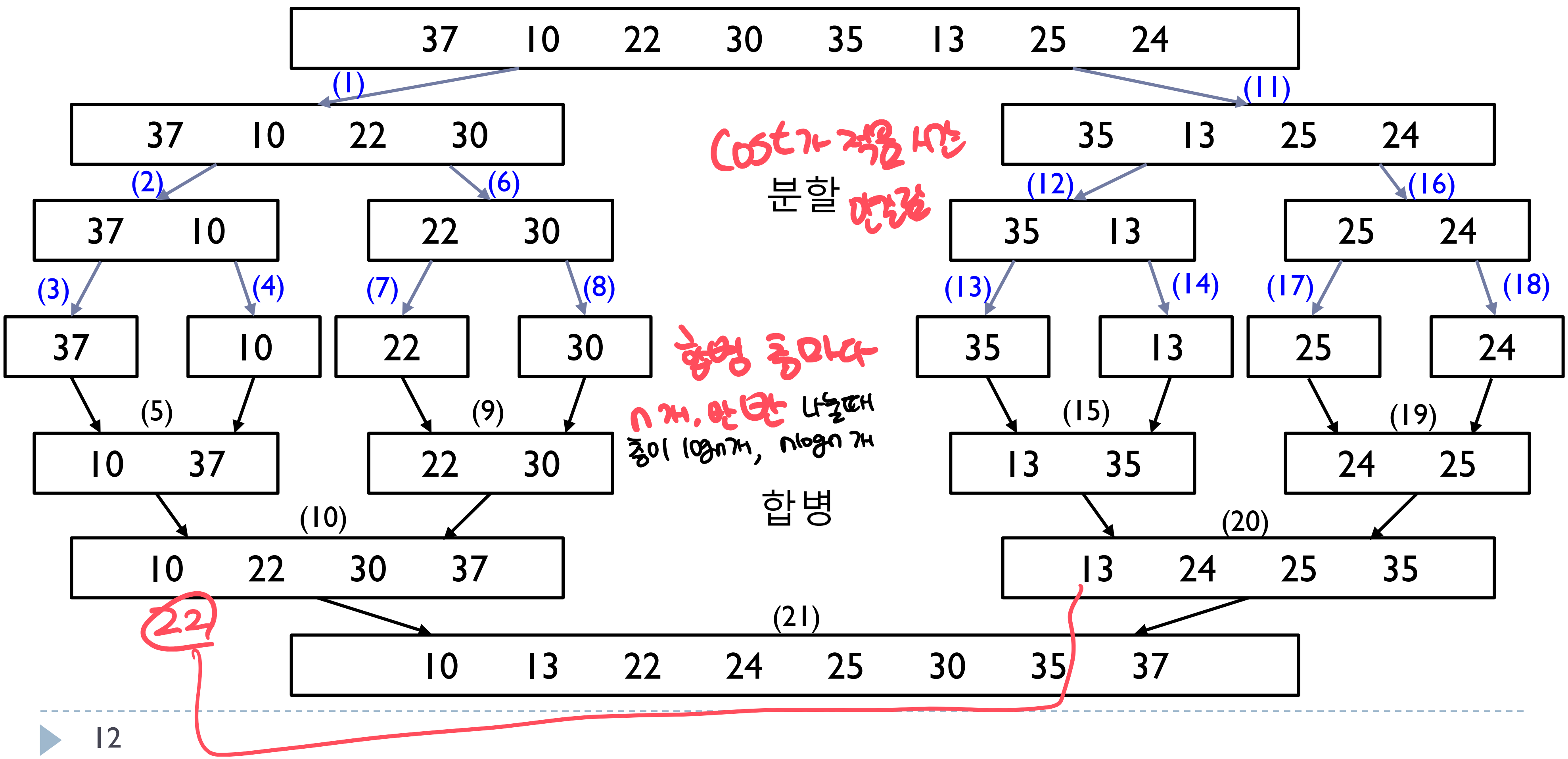
Line 2: 정렬할 부분의 원소들을 1/2로 나누기 위해, $k = \lfloor (p+q)/2 \rfloor$를 계산한다. 즉, 원소의 수가 홀수인 경우, k는 소수점 이하는 버린 정수이다.

Line 3~4: MergeSort(A, p, k)와 MergeSort(A, k+1, q)를 재귀 호출하여 각각 정렬한다.

Line 5: line 3~4에서 각각 정렬된 부분을 합병한다. 합병 과정의 마지막에는 임시 배열에 있는 합병된 원소들을 배열 A로 복사한다.

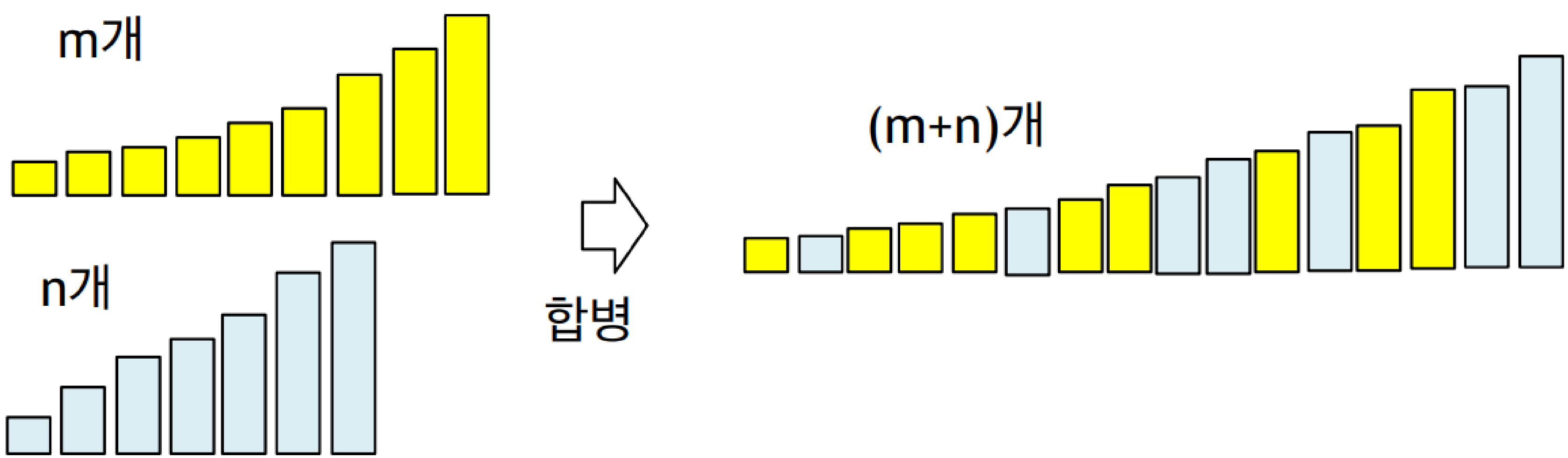
합병 정렬 알고리즘

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



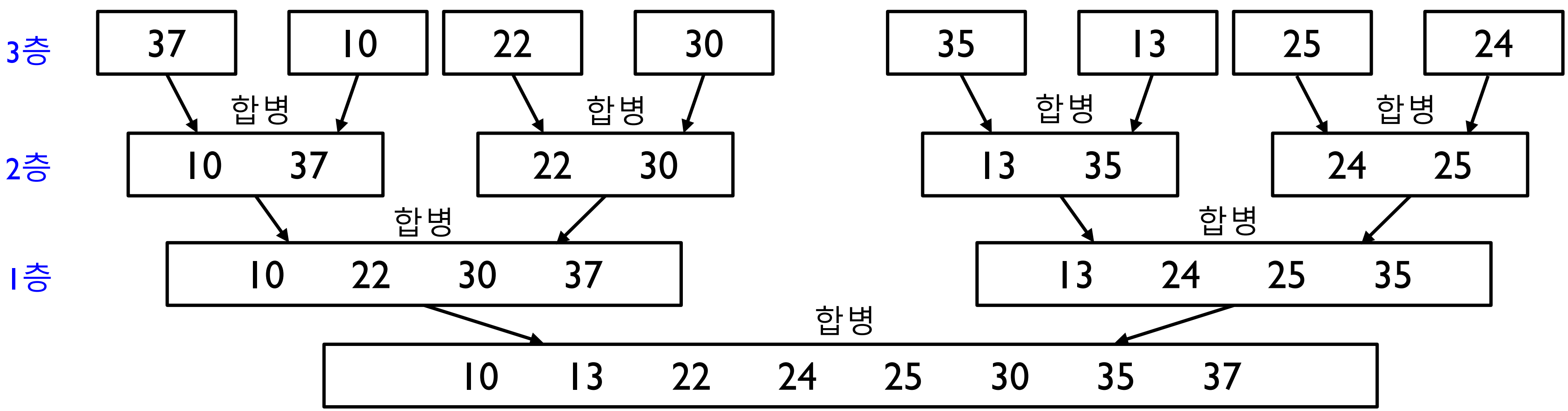
시간복잡도

- 분할하는 부분은 배열의 중간 인덱스 계산과 2번의 재귀 호출이므로 O(1) 시간이 걸린다.
- 합병의 수행 시간은 입력의 크기에 비례한다. 즉, 2개의 정렬된 배열 A와 B의 크기가 각각 m과 n이라면, 최대 비교 횟수는 (m+n−1)이다. 따라서 합병은 O(m+n)이다.



시간복잡도

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



시간복잡도

- 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

시간복잡도

- 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 $n=2^k$으로부터 $\log_2 n$임을 알 수 있다.

- 합병 정렬의 시간복잡도
  - (층수) × O(n) = $\log_2 n$ × O(n) = O(nlogn)

입력개수 : n

합병 정렬의 단점

- 합병 정렬의 공간 복잡도는 O(n)이다.
- 입력을 위한 메모리 공간(입력 배열)외에 추가로 입력과 같은 크기의 공간(임시 배열)이 별도로 필요하다.
- 2개의 정렬된 부분을 하나로 합병하기 위해, 합병된 결과를 저장할 곳이 필요하기 때문이다.

응용

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

3.2 퀵 정렬

나누면서 해결

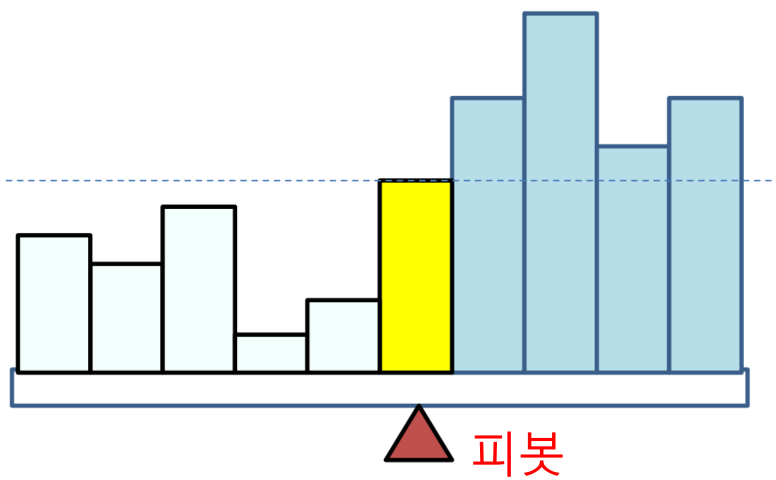
- 퀵 정렬(Quick Sort)은 분할 정복 알고리즘으로 분류되나, 사실 알고리즘이 수행되는 과정을 살펴보면 정복 후 분할하는 알고리즘이다.
- 퀵 정렬 알고리즘은 문제를 2개의 부분문제로 분할하는데, 각 부분문제의 크기가 일정하지 않은 형태의 분할 정복 알고리즘이다.

퀵 정렬

값의 기준

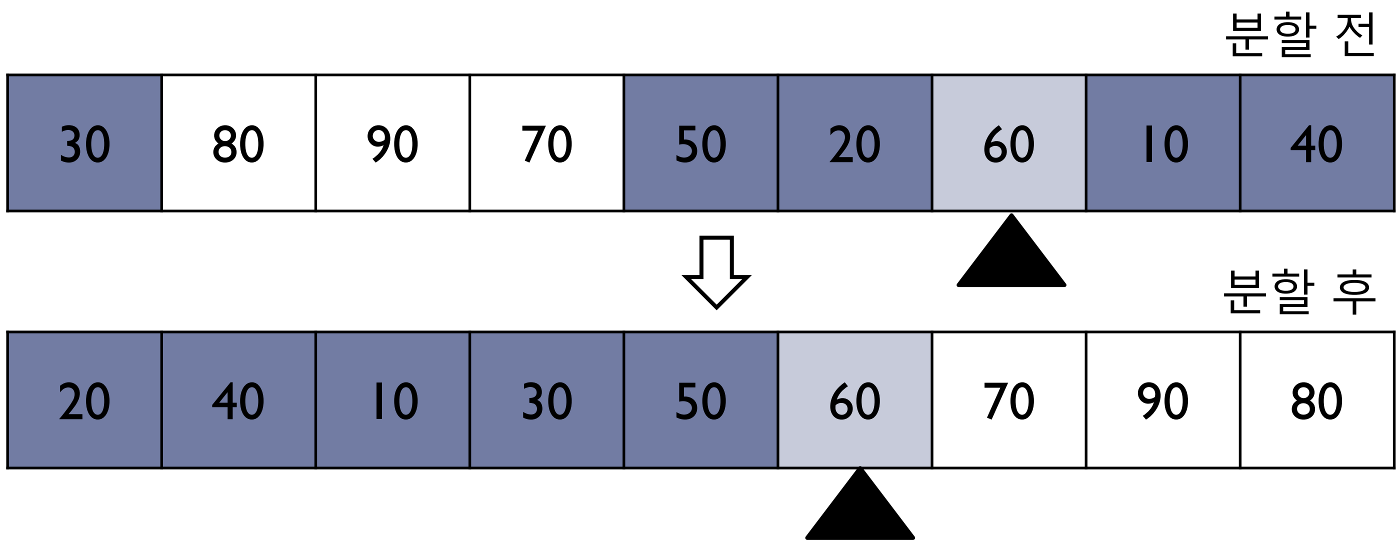
- 퀵 정렬은 피봇(pivot)이라 일컫는 배열의 원소(숫자)를 기준으로 피봇보다 작은 숫자들은 왼편으로, 피봇보다 큰 숫자들은 오른편에 위치하도록 분할하고, 피봇을 그 사이에 놓는다.
- 퀵 정렬은 분할된 부분문제들에 대하여서도 위와 동일한 과정을 재귀적으로 수행하여 정렬한다.

Cost가 있음 피봇기준
다 비교하니까 n 만큼 듦.
총이 최악의경우 n개. n^2
반을 잘 나누면 $\log_2 n$
so. $n \log_2 n$



퀵 정렬

- 단, 주의할 점은 피봇은 분할된 왼편이나 오른편 부분에 포함되지 않는다.



퀵 정렬 알고리즘

QuickSort(A, left, right)

입력: 배열 A[left]~A[right]
출력: 정렬된 배열 A[left]~A[right]

```
if ( left < right ) {
  피봇을 A[left]~A[right] 중에서 선택하고, 피봇을 A[left]와 자리를
  바꾼 후, 피봇과 배열의 각 원소를 비교하여 피봇보다 작은 숫자들
  은 A[left]~A[p–1]로 옮기고, 피봇보다 큰 숫자들은 A[p+1]~A[right]
  로 옮기며, 피봇은 A[p]에 놓는다.
  QuickSort(A, left, p–1)   // 피봇보다 작은 그룹
  QuickSort(A, p+1, right) // 피봇보다 큰 그룹
}
```

퀵 정렬 알고리즘

2개 이상이면

Line 1: 배열 A의 가장 왼쪽 원소의 인덱스(left)가 가장 오른쪽 원소의 인덱스 (right)보다 작으면 line 2~4에서 정렬을 수행한다. 만일 그렇지 않으면 1개의 원소를 정렬하는 경우이다. 1개의 원소는 그 자체가 이미 정렬된 것이므로, line 2~4의 정렬 과정을 수행할 필요없이 그대로 호출을 마친다.

Line 2: A[left]~A[right]에서 피봇을 선택하고, 배열A[left+1]~A[right]의 원소들을 피봇과 비교하여, 피봇보다 작은 그룹인 A[left]~A[p−1]과 피봇보다 큰 그룹인 A[p+1]~A[right]로 분할하고 A[p]에 피봇을 위치시킨다. 즉, p는 피봇이 위치한 배열 A의 인덱스이다.

Line 3: 피봇보다 작은 그룹인 A[left]~A[p−1]을 재귀적으로 호출한다.

Line 4: 피봇보다 큰 숫자들은 A[p+1]~A[right]를 재귀적으로 호출한다.

퀵 정렬 알고리즘 예제

- 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

  - 피봇 A[6]=8이라면, line 2에서 아래와 같이 차례로 원소들의 자리를 바꾼다. 먼저 피봇을 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

퀵 정렬 알고리즘 예제

- 그 다음에는 피봇보다 큰 수와 피봇보다 작은 수를 다음과 같이 각각 교환한다.

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

퀵 정렬 알고리즘 예제

- 마지막으로 피봇을 A[4]로 옮기기 위해 A[0]과 교환한다. 피봇을 A[4]로 이동하는 이유는 피봇(즉, 8)보다 작으면서 가장 오른쪽에 있는 숫자(즉, 2)가 A[4]에 있기 때문이다.

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 QuickSort(A, 0, 4–1) = QuickSort(A, 0, 3)이 호출되고, 그 다음 line 4에서 QuickSort(A, 4+1, 11) = QuickSort(A, 5, 11)이 호출된다.

퀵 정렬 알고리즘 예제

- QuickSort(A, 0, 3) 호출

0	1	2	3
2	3	7	6

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

0	1	2	3
6	3	7	2

⇨

0	1	2	3
6	3	2	7

0	1	2	3
2	3	6	7

퀵 정렬 알고리즘 예제

- 그리고 line 3에서 QuickSort(A, 0, 2–1) = QuickSort(A, 0, 1)이 호출되고, 그 다음 line 4에서 QuickSort(A, 2+1, 3) = QuickSort(A, 3, 3)이 호출된다.

- QuickSort(A, 0, 1) 호출

0	1
2	3

Basecase 기준

1개 남으면 단순하죠?.

노하우

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

0	1
3	2

0	1
2	3

퀵 정렬 알고리즘 예제

- 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 호출된다.
- 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.

- QuickSort(A, 3, 3) 호출
  - A[3] 자체로 이미 정렬되어 있다.
- 따라서, QuickSort(A, 0, 3)은 아래와 같이 완성된다.

0	1	2	3
2	3	6	7

- QuickSort(A, 5, 11)도 유사한 과정을 거쳐 정렬된다.

최악 경우의 분할: 가장 작거나 큰 피봇

- 퀵 정렬의 성능은 피봇 선택이 좌우한다. 피봇으로 가장 작은 숫자(또는 가장 큰 숫자)가 선택되면, 한 부분으로 치우치는 분할을 야기한다.

1	17	42	9	18	23	31	11	26

1	9	42	17	18	23	31	11	26

1	9	11	17	18	23	31	42	26

⋮

1	9	11	17	18	23	26	31	42

최악 경우의 분할: 시간복잡도

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

- 총 비교 횟수는 8+7+6+…+1 = 36이다.
- 입력의 크기가 n이라면, 퀵 정렬의 최악 경우 시간복잡도는 $(n-1)+(n-2)+(n-3)+\ldots+2+1 = n(n-1)/2 = O(n^2)$이다.

최선 경우의 분할: 1/2씩 균등 분할하는 피봇

각 부분의
크기

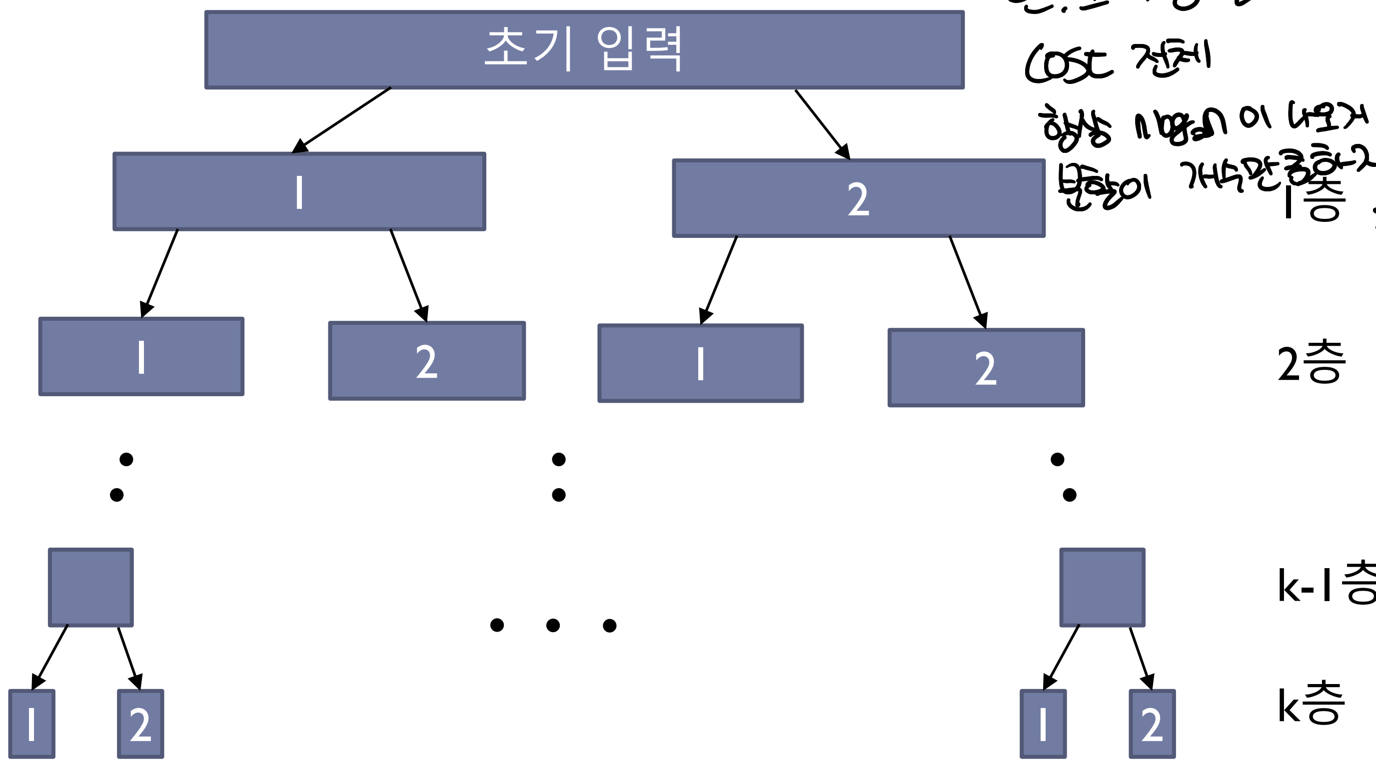
n

$\frac{n}{2}$

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

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

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



왼, 오 정렬이다 피봇 계산. 나눈데
cost 전체
항상 nlogn 이 나오지 않는 이유는
분할이 개수만큼 균등하지 않기 때문에
피봇 선택 3개

최선 경우의 분할: 시간복잡도

- 각 층에서는 각각의 원소가 각 부분의 피봇과 1회씩 비교된다. 따라서 비교 횟수는 O(n)이다.
- 총 비교 횟수는 O(n)×(층수) = O(n)×($\log_2 n$)이다.
- 층수가 $\log_2 n$인 이유는 $n/2^k=1$일 때 $k=\log_2 n$이기 때문이다.
- 퀵 정렬의 최선 경우 시간복잡도는 $O(n\log_2 n)$이다.

옮기는건 머지보다 적어서 속도가 빠름

평균 경우의 분할: 시간복잡도

- 피봇을 항상 랜덤하게 선택한다고 가정하면, 퀵 정렬의 평균 경우 시간복잡도를 계산할 수 있다.
- 평균 경우 시간복잡도는 최선 경우와 동일하게 $O(n\log_2 n)$ 이다.

피봇 선정 방법

1) 랜덤하게 선정하는 방법
2) **3개 숫자의 중앙값으로 선정하는 방법**
   - 가장 왼쪽 숫자, 중간 숫자, 가장 오른쪽 숫자 중에서 중앙값으로 피봇을 정한다.
   - 아래의 예제를 보면, **31, 1, 26** 중에서 중앙값인 **26**을 피봇으로 사용한다.

31	17	42	9	1	23	18	11	26

3개 or 5개 노하우

성능 향상 방법

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

1까지 기다리는 게 아니라 중간에 방법을 바꿈

응용

- 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 문제

- 선택(Selection) 문제는 n개의 숫자들 중에서 k번째로 작은 숫자를 찾는 문제이다.
- 선택 문제를 해결하기 위한 간단한 방법들
  - 알고리즘 1: 최소 숫자를 k번 찾는다. 단, 최소 숫자를 찾은 뒤에는 입력에서 최소 숫자를 제거한다.
  - 알고리즘 2: 숫자들을 정렬한 후, k번째 숫자를 찾는다.
- 위의 알고리즘들은 각각 최악의 경우 O(kn)과 O(nlogn)의 수행 시간이 걸린다.

+n

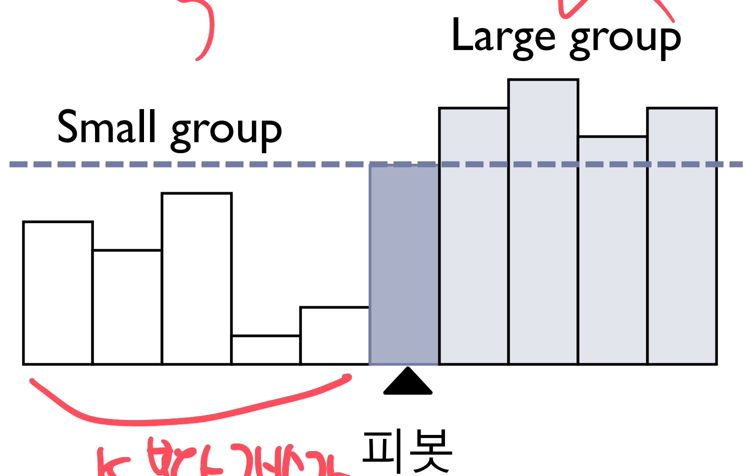
n= 입력크기

선형

핵심아이디어

- 이진탐색(Binary Search)은 정렬된 입력의 중간에 있는 숫자와 찾고자 하는 숫자를 비교함으로써, 입력을 1/2로 나눈 두 부분 중에서 한 부분만을 검색한다.
- 선택 문제는 입력이 정렬되어 있지 않으므로, 입력 숫자들 중에서 (퀵 정렬에서와 같이) 피봇을 선택하여 아래와 같이 분할한다.

5 k+(?)



c(x) 2

k번째? k보다 개수가

핵심아이디어

- Small group은 피봇보다 작은 숫자의 그룹이고, Large group은 피봇보다 큰 숫자의 그룹이다.
- 분할 후, 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. 여기서 |Small group|은 Small group에 있는 숫자의 개수이고, 1은 피봇에 해당된다.

선택 문제 분할 정복 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 k번째 작은 원소	
1	피봇을 A[left]~A[right]에서 랜덤하게 선택하고, 피봇과 A[left]의 자리를 바꾼 후, 피봇과 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

피봇이면

S.L 중 하나만 고름

선택 문제 분할 정복 알고리즘

Line 1: 피봇을 랜덤하게 선택하는 것을 제외하고는 퀵 정렬 알고리즘의 line 2와 동일하다.
Line 2: 입력을 두 그룹으로 분할 후, A[p]가 피봇이 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)–left+1이다.
Line 3: k번째 작은 수가 Small group에 속한 경우이므로 Selection(A, left, p–1, k)를 호출한다.
Line 4: k번째 작은 수가 피봇인 A[p]와 같은 경우이므로 해를 찾은 것이다.
Line 5: k번째 작은 수가 Large group에 속한 경우이므로 Selection(A, p+1, right, k–S–1)를 호출한다. 이때에는 (k–S–1)번째 작은 수를 Large group에서 찾아야 한다. 왜냐하면 피봇이 k번째 작은 수보다 작고, S는 Small group의 크기이기 때문이다.

Selection 알고리즘 예제

- k = 7 번째 작은 숫자를 찾기 위해 Selection 알고리즘 수행

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

번째 작은?.

  - k=7, left=0, right=11
  - Line 1에서 피봇 A[6]=8 이라고 가정하면, 피봇이 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

Selection 알고리즘 예제

- 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

5

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

4 5 7-4-1 7

Selection 알고리즘 예제

- Line 2에서 Small group의 크기를 계산한다.
  즉, S = (p–1)–left+1 = (4–1)–0+1 = 4 이다.
- 따라서 Small group에는 7번째 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) 호출한다.
- 즉, Large group에서 2번째로 작은 수를 찾는다.

Selection 알고리즘 예제

- 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 피봇 A[11]=14라면, 피봇이 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

Selection 알고리즘 예제

- 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

Selection 알고리즘 예제

- Line 2에서 Small group의 크기를 계산한다.
  즉, S = (p–1)–left+1 = (9–1)–5+1 = 4 이다.
- 따라서 k=2번째 작은 수를 찾아야 하므로 line 3의 if-조건인 (k≤S) = (2≤4)가 '참'이 되어 Selection(A, left, p–1, k) = Selection(A, 5, 9–1, 2) = Selection(A, 5, 8, 2)를 호출한다.
- 즉, Small group에서 2번째로 작은 수를 찾는다.

Selection 알고리즘 예제

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

5	6	7	8
10	9	12	11

- Line 1에서 피봇 A[5]=10 이라면, 원소간 자리바꿈 없이 아래와 같이 된다.

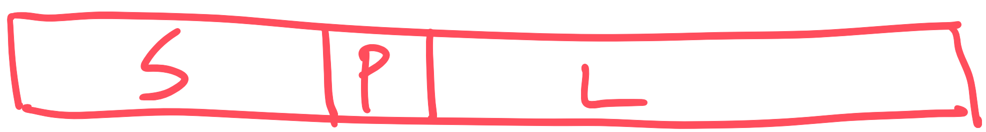
5	6	7	8
10	9	12	11

5	6	7	8
9	10	12	11

Selection 알고리즘 예제

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

퀵정렬을 하고 개수를 따지고 좌우중에 k번째를 찾는다



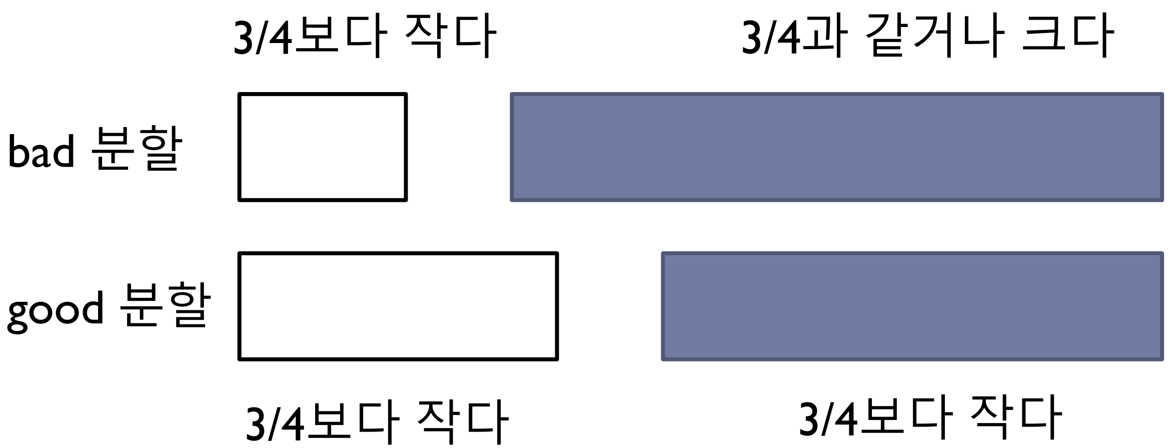
S 크기계산 (P-1) - left +1

Selection 알고리즘

- Selection 알고리즘은 분할 정복 알고리즘이기도 하지만 랜덤(random) 알고리즘이기도 하다.
  - 왜냐하면 선택 알고리즘의 line 1에서 피봇을 랜덤하게 정하기 때문이다.
- 만일 피봇이 입력 리스트를 너무 한 쪽으로 치우치게 분할하면, 즉, |Small group| ≪ |Large group| 또는 |Small group| ≫ |Large group|일 때에는 알고리즘의 수행 시간이 길어진다.

good/bad 분할

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



good/bad 분할

- 아래의 예를 살펴보면 good 분할이 되는 피봇을 선택할 확률과 bad 분할이 되는 피봇을 선택할 확률이 각각 1/2로 동일함을 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 분할이 된다.
- 1~4 또는 13~16 중 하나가 피봇으로 정해지면 bad 분할이 된다.

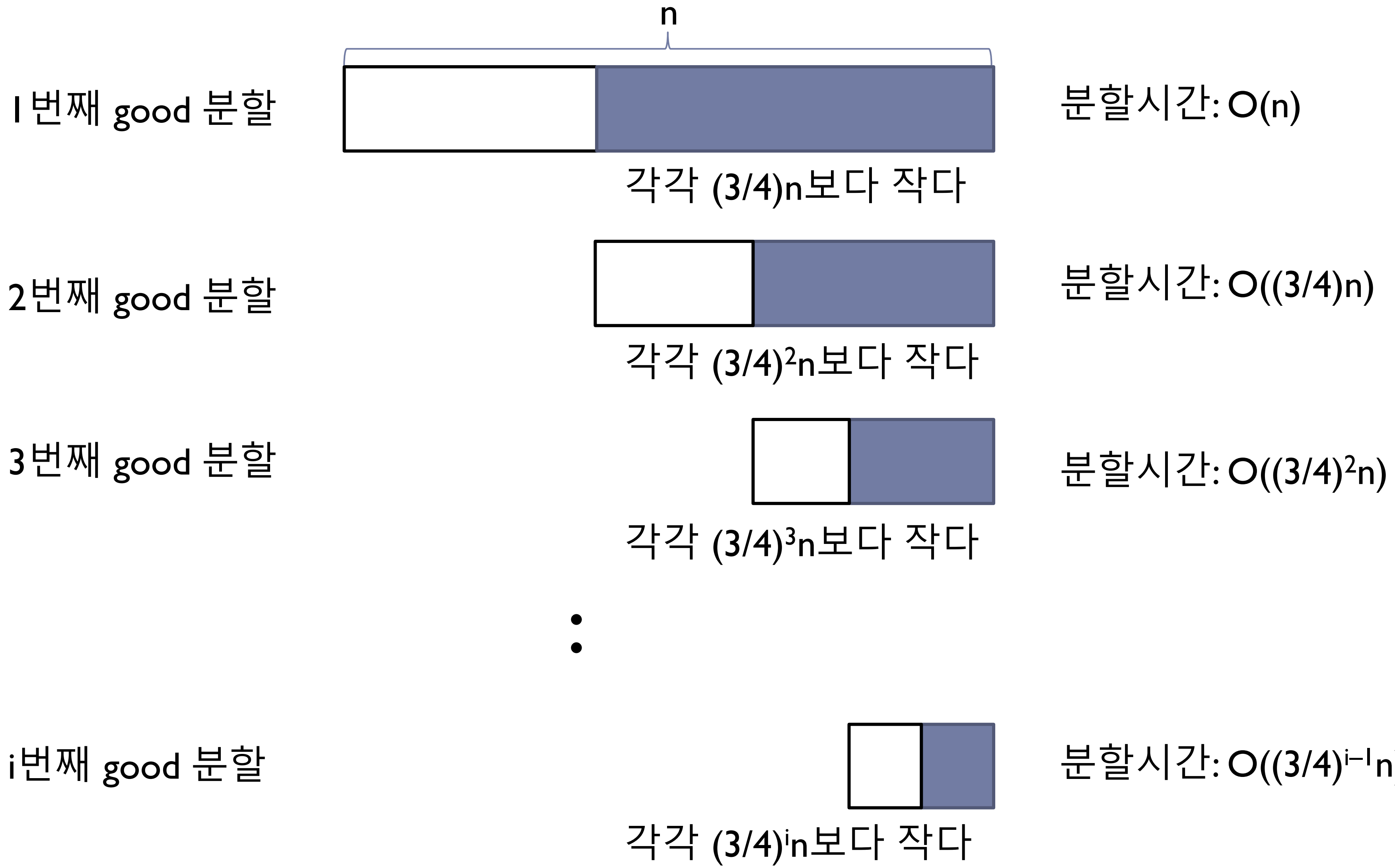
각각 $\frac{3}{4}$보다 작아야 좋음

시간복잡도

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

매번 good 분할로 생각. 결과 나오고 x2

시간복잡도



n
1번째 good 분할
분할시간: O(n)
각각 (3/4)n보다 작다
2번째 good 분할
분할시간: O((3/4)n)
각각 $(3/4)^2$n보다 작다
3번째 good 분할
분할시간: $O((3/4)^2 n)$
각각 $(3/4)^3$n보다 작다
:
i번째 good 분할
분할시간: $O((3/4)^{i-1} n)$
각각 $(3/4)^i$n보다 작다

시간복잡도

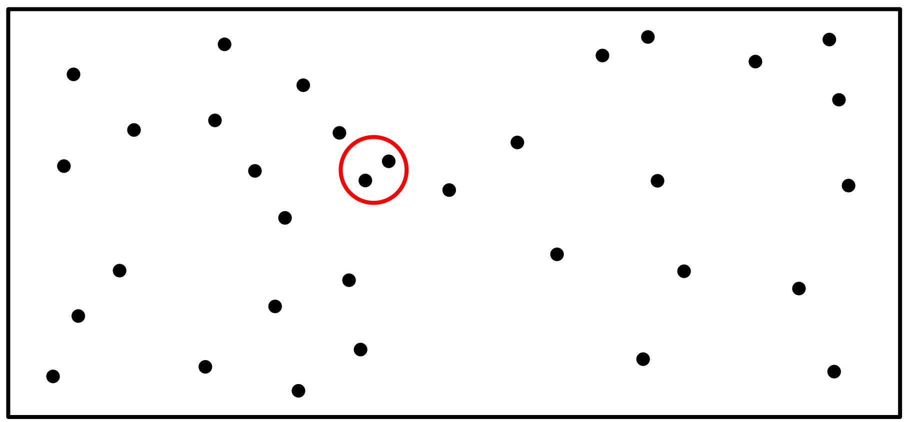
- 입력 크기가 n에서부터 3/4배로 연속적으로 감소되고, 입력 크기가 1일 때에는 더 이상 분할할 수 없게 된다. 그러므로 good 분할만 연속적으로 이루어졌을 때의 Selection 알고리즘 평균 경우 시간복잡도는 다음과 같다.
  - $O[n + (3/4)n + (3/4)^2 n + (3/4)^3 n + \ldots + (3/4)^{i-1} n + (3/4)^i n]$
    $= O[n\times(1-(3/4)^{i+1})/(1-3/4)]$
    $= O[4n\times(1-(3/4)^{\log_{4/3} n+1})]$
    $= O[4n\times(1-3/(4n))]$
    $= O[4n-3]$
    $= O(n)$
- 평균 2번 만에 good 분할이 되기 때문에 최종적으로 Selection 알고리즘의 평균 경우 시간복잡도는 다음과 같다.
  - $2\times O(n) = O(n)$

응용

- 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 1개 의 숫자가 매우 큰 숫자(노이즈(noise), 잘못 측정된 데이터)이면, 평균값은 매우 왜곡된 분석이 된다.
- 직장인의 연간 소득을 분석할 때 평균값보다 중앙값이 더 의미 있는 분석 자료가 된다.

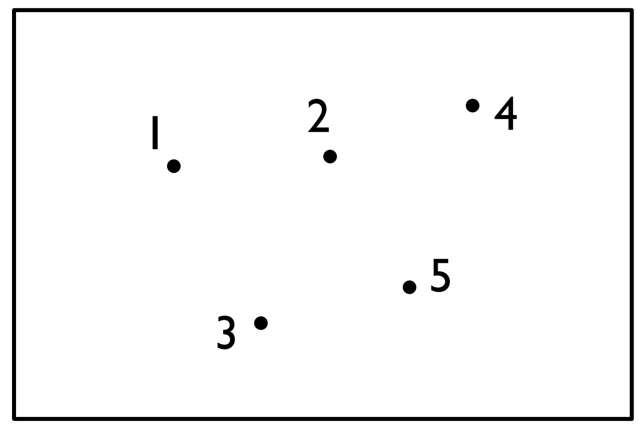
3.4 최근접 점의 쌍 찾기

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



최근접 점의 쌍 찾기

- 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?
  - ${}_{n}C_{2} = n(n-1)/2$
  - n=5이면, $5(5-1)/2 = 10$
  - $n(n-1)/2 = O(n^2)$
  - 한 쌍의 거리 계산은 O(1) 시간
  - 시간복잡도는 $O(n^2) \times O(1) = O(n^2)$



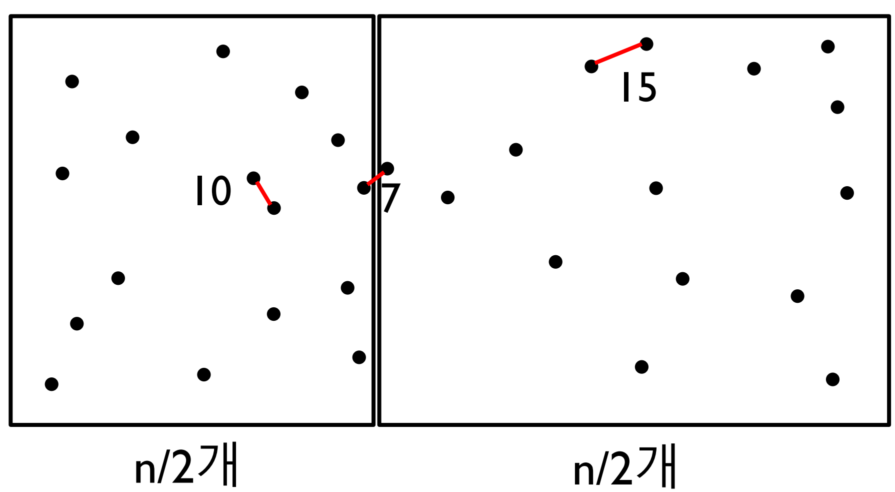
분할 정복을 이용한 최근접 점의 쌍 찾기

- O(n^2)보다 효율적인 방법은 분할 정복을 이용하는 것이다.
  - n개의 점을 1/2로 분할하여 각각의 부분문제에서 최근접 점의 쌍을 찾고, 2개의 부분해 중 짧은 거리를 가진 점의 쌍을 찾는다.
  - 2개의 부분해를 취합할 때에는 다음과 같은 경우를 고려해야 한다. 왼쪽 부분문제의 최근접 쌍의 거리가 10이고, 오른쪽 부분문제의 최근접 쌍의 거리가 15이다. 왼쪽 부분문제의 가장 오른쪽 점과 오른쪽 부분문제의 가장 왼쪽 점 사이의 거리가 7이다.

중간영역계산 $n\log n$

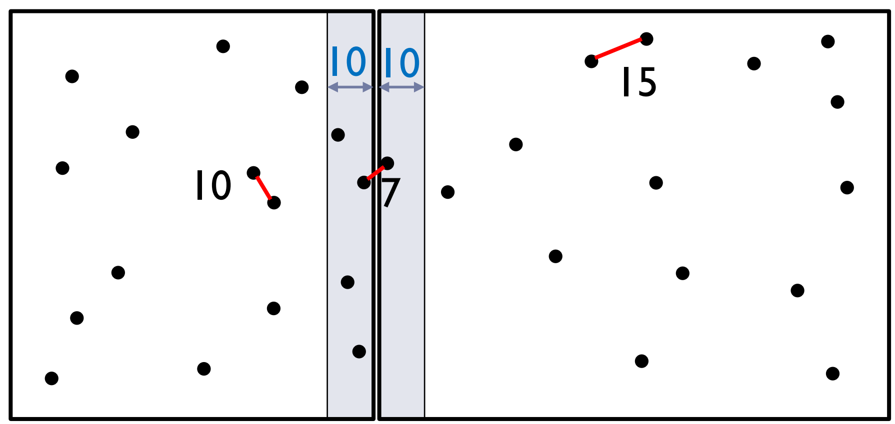
sort $\log n$

$50 \cdot n\log_2 n^2$



분할 정복을 이용한 최근접 점의 쌍 찾기

- 따라서 2개의 부분문제의 해를 취합할 때 단순히 10과 15 중에서 짧은 거리인 10을 해라고 할 수 없다.
- 그러므로 아래의 그림에서와 같이 각각 거리가 10이내의 중간 영역 안에 있는 점들 중에 최근접 점의 쌍이 있는지도 확인해보아야 한다.



9시 5
10시 6
11시 7
12시 8
1시 9
2시 10
3시 11
4시

분할 정복을 이용한 최근접 점의 쌍 찾기

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

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, -)

10 10

분할 정복을 이용한 최근접 점의 쌍 찾기

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

d=10

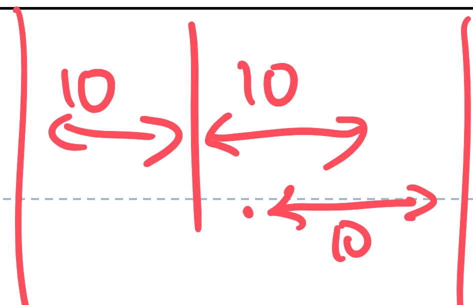
0	1	2	3	4	5	6	7	8	9
(1,-)	(13, -)	(17, -)	(25, -)	(26, -)	(28,-)	(30, -)	(37, -)	(45, -)	(56, -)

26 – d = 16

28 + d = 38

최근접 점의 쌍을 찾는 분할 정복 알고리즘

ClosestPair(S)	
입력: x-좌표의 오름차순으로 정렬된 배열 S에는 i개의 점 (단, 각 점은 (x,y)로 표현된다.) 출력: S에 있는 점들 중 최근접 점의 쌍의 거리	
1	if (i ≤ 3) return (2 또는 3개의 점들 사이의 최근접 쌍) 점에 분포에따라 4.5도 가능
2	정렬된 S를 같은 크기의 S_L과 S_R로 분할한다. 단, |S|가 홀수이면, $\|S_L\| = \|S_R\|+1$이 되도록 분할한다.
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 거리이다.
6	return(CP_L, CP_C, CP_R 중에서 거리가 가장 짧은 쌍)



중간에서 d

최근접 점의 쌍을 찾는 분할 정복 알고리즘

Line 1: S에 있는 점의 수가 3개 이하이면 더 이상 분할하지 않는다. S에 2개의 점이 있으면 S를 그대로 리턴하고, 3개의 점이 있으면 3개의 쌍에 대하여 최근접 점의 쌍을 리턴한다.

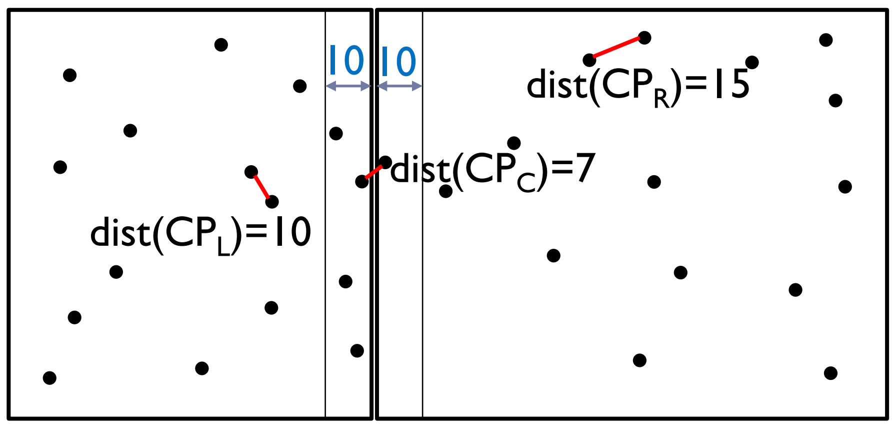
Line 2: x-좌표로 정렬된 S를 왼쪽과 오른쪽에 같은 개수의 점을 가지는 S_L과 S_R로 분할한다. 만일 S의 점의 수가 홀수이면 S_L쪽에 1개 많게 분할한다.

Line 3~4: 분할된 S_L과 S_R에 대해서 재귀적으로 최근접 점의 쌍을 찾아서 각각을 CP_L과 CP_R이라고 놓는다.

최근접 점의 쌍을 찾는 분할 정복 알고리즘

Line 5: d를 이용하여 중간 영역에 속하는 점들을 찾고, 이 점들 중에서 최근접 점의 쌍을 찾아서 이를 CP_C라고 한다.

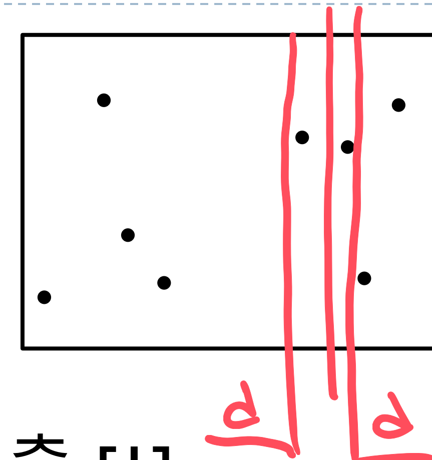
$$d = \min\{dist(CP_L), dist(CP_R)\} = \min\{10 , 15\} = 10$$



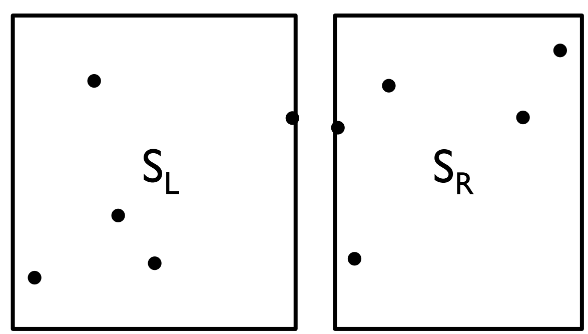
Line 6: line 3~4에서 각각 찾은 최근접 점의 쌍 CP_L과 CP_R과 line 5에서 찾은 CP_C 중에서 가장 짧은 거리를 가진 쌍을 해로서 리턴한다.

ClosestPair 예제

- 입력 S



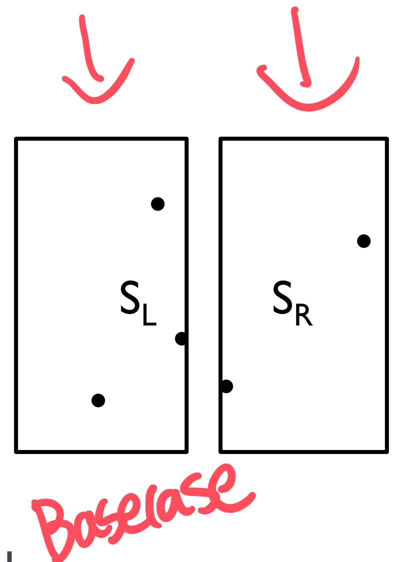
- ClosestPair(S)로 호출 [1]
  - Line 1: S의 점의 수 > 3 이므로 다음 line을 수행한다.
  - Line 2: S를 S_L과 S_R로 분할한다.



ClosestPair 예제

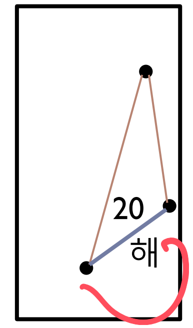
- Line 3: ClosestPair(S_L)를 호출한다. ClosestPair(S_L)을 수행한 후 리턴된 점의 쌍을 CP_L이라고 놓은 후에 line 4~6 을 차례로 수행한다.

- **ClosestPair(S_L) 호출 [2]:**
  - Line 1의 if-조건이 '거짓'이므로 line 2에서 다시 분할한다.
  - Line 3: ClosestPair(S_L)를 호출한다. (여기서의 S_L은 처음 S_L의 왼쪽 반이다.)
  - ClosestPair(S_L)을 수행한 후, 리턴된 점의 쌍을 CP_L이라고 놓은 후에 line 4~6을 차례로 수행한다.



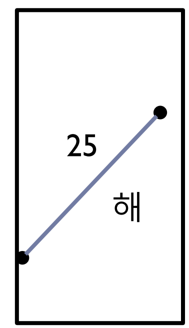
ClosestPair 예제

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



return CPL

- ClosestPair(S_R) 호출:
  - Line 1에서 점의 수가 2이므로, 이 두 점을 최근접 점의 쌍으로 리턴한다.
  - 최근접 점의 쌍의 거리를 25라고 가정하자.



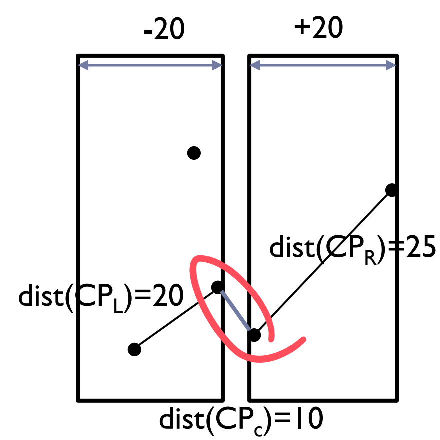
CPR
25

return

25이므로 d를 20으로 둔다

ClosestPair 예제

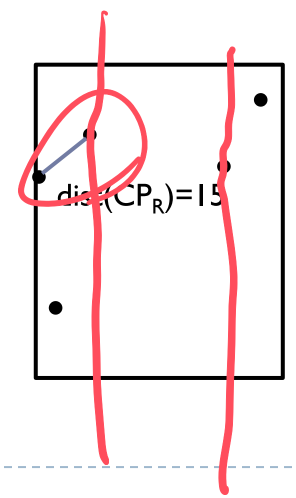
- [2]의 ClosestPair(S_L) 호출 당시 line 3~4가 수행되었고, 이제 line 5가 수행된다.
  - Line 3~4에서 찾은 최근접 점의 쌍 사이의 거리인 dist(CP_L)=20과 dist(CP_R)=25 중에 작은 값을 d=20 으로 놓는다.
  - 왼쪽 중간점의 x-좌표에서 20을 뺀 값과 오른쪽 중간점의 x-좌표에 20을 더한 값 사이의 x-좌표 값을 가 진 점들 중에서 CP_C를 찾는다.
  - 옆의 그림과 같이 거리가 10인 CP_C를 찾는다.



CP_c를 return

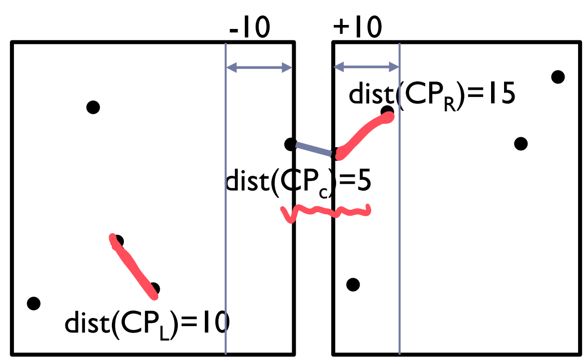
ClosestPair 예제

- Line 6: dist(CP_L)=20, dist(CP_C)=10, dist(CP_R)=25 중에서 가장 거리가 짧은 쌍인 CP_C를 리턴한다.
- [1]의 ClosestPair(S) 호출 당시 line 3이 수행되었고, 이제 line 4에서는 ClosestPair(S_R)을 호출한다. 여기서 S_R은 초기 입력의 오른쪽 반인 영역이다. ClosestPair(S_R) 호출 결과로 아래 그림의 최근접 점의 쌍을 리턴하고 이를 CP_R로 놓는다. 이때 dist(CP_R)=15라고 하자.



ClosestPair 예제

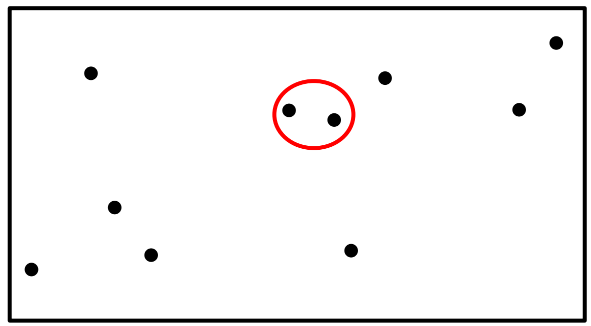
- Line 5: line 3~4에서 찾은 최근접 점의 쌍 사이의 거리인 dist(CP_L)=10과 dist(CP_R)=15 중에 작은 값을 d=10 이라고 놓는다. 그리고 중간 영역에 있는 점들 중에서 CP_C를 찾는다. 여기서는 다음 그림과 같이 거리가 5인 CP_C를 최종적으로 찾는다.



d= min(10.15)
= 10 중간영역

ClosestPair 예제

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



시간복잡도

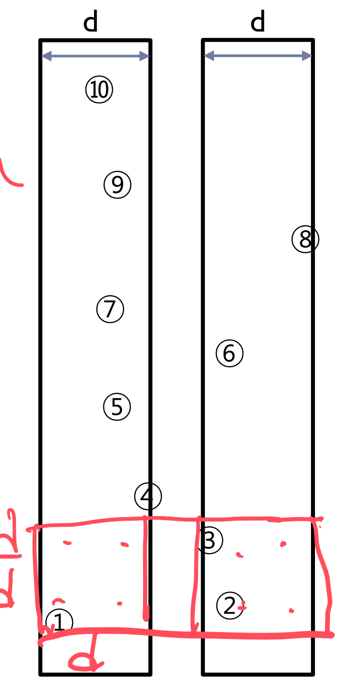
정렬 O(nlogn)이므로 최소
O(nlogn)

- 입력 S에 n개의 점이 있다고 가정한다.
- 알고리즘의 전처리(preprocessing) 과정으로서 S의 점을 x-좌표로 정렬하는데, O(nlogn)의 시간이 소요된다.
  - 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 배열이 x-좌표로 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 동일하다.

시간복잡도

- 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(또는 위에서 아래로) 각 점을 기준으로 거리가 d이내인 주변의 점들 사이의 거리를 각각 계산하며, 이 영역에 속한 점들 중에서 최근접 점의 쌍을 찾는다.
- 따라서 y-좌표로 정렬하는데 O(nlogn) 시간이 걸리고, 그 다음에는 아래에서 위로 올라가며 각 점 에서 주변의 점들 사이의 거리를 계산하는데 O(1) 시간이 걸린다. 왜냐하면 각 점과 거리 계산해야 하는 주변 점들의 수는 O(1)개이기 때문이다.

상하에서만 고려



중간영역 중요

7개 이하

시간복잡도

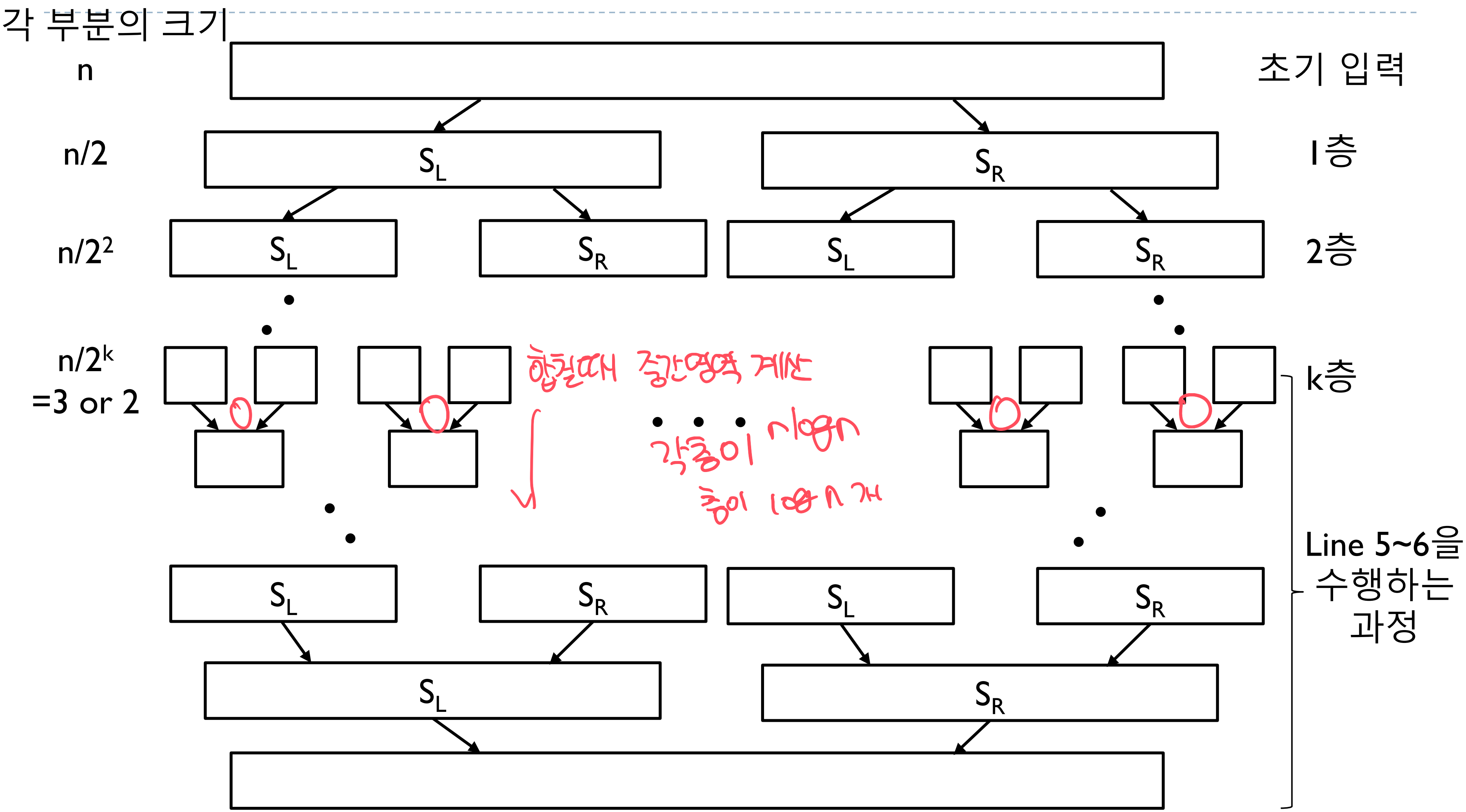
- Line 6: 3개의 점의 쌍 중에 가장 짧은 거리를 가진 점의 쌍을 리턴하므로 O(1) 시간이 걸린다.
- ClosestPair 알고리즘의 분할과정은 합병 정렬의 분할과정과 동일하다.
- ClosestPair 알고리즘에서는 해를 취합하여 올라가는 과정인 line 5~6에서 O(nlogn)시간이 걸린다.
- 다음의 그림에서 k층까지 분할된 후, 층별로 line 5~6이 수행되는 (취합) 과정을 보여준다. 이때 각 층의 수행 시간은 O(nlogn)이다. 여기에 층 수인 logn을 곱하면 O(nlog²n)이 된다.
- 이것이 ClosestPair 알고리즘의 시간복잡도 O(nlog²n)이다.

$O(n^2) = n \times n$

기본

분할할때 Base case 가 가면 / 중간영역이 시간 걸림

시간복잡도

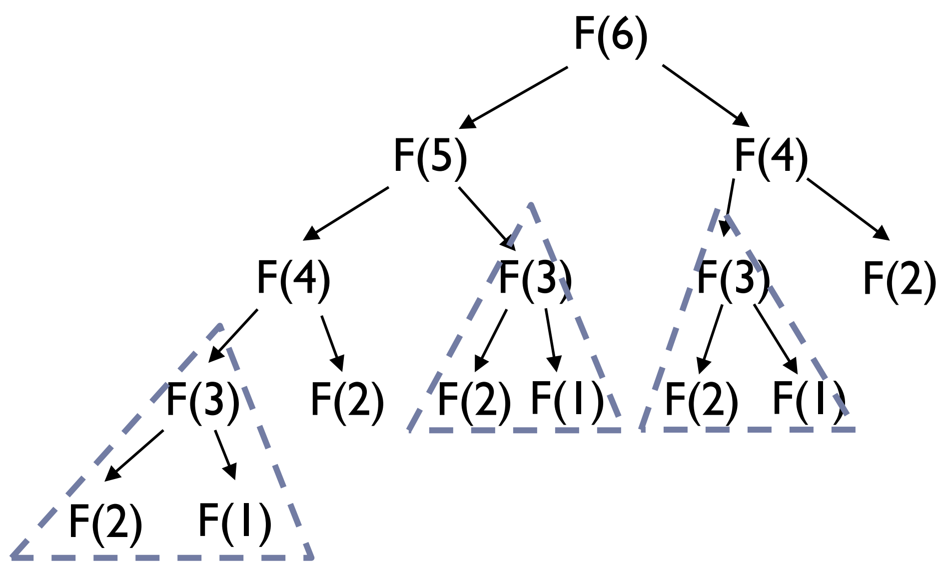


응용

- 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.



똑같은걸 3번이나 함

분할 정복을 적용하는데 있어서 주의할 점

- 예를 들어, n번째의 피보나치 수를 구하는데 F(n) = F(n–1) + F(n–2)로 정의되므로 재귀 호출을 사용하는 것이 자연스러워 보이나, 이 경우의 입력은 1개이지만, 사실상 n의 값 자체가 입력 크기인 것이다.

  입력 n이 2n-3이 됨. 분할정복 별로

- 따라서 n이라는 숫자로 인해 2개의 부분문제인 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(6)을 구하기 위해 분할된 부분문제들을 보여준다. F(2)를 5번이나 중복하여 계산해야 하고, F(3)은 3번 계산된다.

피보나치 수 계산을 위한 O(n) 시간 알고리즘

```
FibNumber(n)
F[0] = 0
F[1] = 1
for i=2 to n
  F[i] = F[i–1]+ F[i–2]
```

- 피보나치 수 계산의 경우 분할 정복 알고리즘을 사용하는 것은 매우 부적절하며 위와 같이 for-루프를 사용하여 중복된 계산 없이 간단하게 구할 수 있다. 이 알고리즘의 시간복잡도는 루프의 수행 횟수로서 O(n)이다.

분할 정복을 적용하는데 있어서 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.
- 3장에서 살펴본 문제들은 취합 과정이 간단하거나 필요가 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.