ALGORITHM
알고리즘

Lecture 04

정렬 (2)

컴퓨터과학과 | 이관용 교수

한국방송통신대학교

학습목차

01. 퀵 정렬

퀵 정렬?

- **특정 데이터를 기준으로 주어진 배열을 2개의 부분배열로 분할하고, 각 부분배열에 대해서 퀵 정렬을 순환적으로 적용하는 방식**

- **피벗** pivot, **분할 원소**
  - 주어진 배열을 두 부분배열로 분할하는 기준이 되는 특정 데이터
    - ✓ 보통 주어진 배열의 첫 번째 데이터로 지정

퀵 정렬의 원리

피벗이 제자리를 잡도록 하여 정렬하는 방식

입력 배열 A[]

0	1	2	3	4	5	6	7
30	45	20	15	40	25	35	10

피벗

분할 후 상태

0	1	2	3	4	5	6	7
25	10	20	15	30	40	35	45

왼쪽 부분배열 / 피벗 / 오른쪽 부분배열

왼쪽 부분배열의 모든 데이터 < 오른쪽 부분배열에서 가장 작은 데이터

왼쪽 부분배열에서 가장 큰 데이터 < 오른쪽 부분배열의 모든 데이터

왼쪽 부분배열의 모든 값 < 피벗 < 오른쪽 부분배열의 모든 값

퀵 정렬의 전체적인 처리 과정

퀵 정렬 알고리즘

퀵 정렬(A[ ], n)

피벗

A[8] = { 30 45 20 15 40 25 35 10 }

① 피벗위치 j = 분할함수(A[0..n-1], n)

0	1	2	3	4	5	6	7
25	10	20	15	30	40	35	45

j

왼쪽 부분배열 오른쪽 부분배열

② 퀵 정렬(왼쪽부분배열 A[0 .. j-1], j)

③ 퀵 정렬(오른쪽부분배열 A[j+1..n-1], n-(j-1))

퀵 정렬 알고리즘


```
QuickSort (A[ ], n)
{
  if (n > 1) {
    // ① 피벗을 기준으로 두 부분배열로 분할
    // pivot은 제자리를 잡은 피벗의 위치(인덱스)를 표시
    pivot = Partition (A[0..n-1], n);

    // ② 왼쪽 부분배열에 대한 퀵 정렬의 순환 호출
    QuickSort (A[0..pivot-1], pivot);

    // ③ 오른쪽 부분배열에 대한 퀵 정렬의 순환 호출
    QuickSort (A[pivot+1..n-1], n-pivot-1);
  }
}
```

분할 과정


피벗보다 큰(같은) 값을 찾음
피벗보다 작은 값을 찾음
피벗=A[0]
Left=1
Right=7
30 45 20 15 40 25 35 10
피벗
Left=1
Right=7
30 45 20 15 40 25 35 10
교환
Left < Right
피벗
Left=4
Right=5
30 10 20 15 40 25 35 45
교환
Left < Right
피벗
Right=4
Left=5
30 10 20 15 25 40 35 45
교환
Left > Right
25 10 20 15 30 40 35 45
왼쪽 부분배열
피벗
오른쪽 부분배열

분할 함수_Partition( )


```
int Partition (A[ ], n)
{
  Left = 1; Right = n-1;
  while (Left < Right) {
     while (Left < n && A[Left] < A[0]) Left++;
     while (Right > 0 && A[Right] >= A[0]) Right--;
     if (Left < Right)
        A[Left]와 A[Right]의 위치 교환
     else
        피벗 A[0]와 A[Right]의 위치 교환
  }
  return (Right);
}
```

오른쪽으로 진행하면서 피벗 A[0]보다 큰 값의 위치를 찾음

왼쪽으로 진행하면서 피벗 A[0]보다 작은 값의 위치를 찾음

피벗과의 위치 교환 후 첫 번째 while문 종료

퀵 정렬의 예

피벗 Left Right

30 50 17 40 88 15 44 55 22 11 66 13 85

30 50 17 40 88 15 44 55 22 11 66 13 85

Left Right

30 13 17 40 88 15 44 55 22 11 66 50 85

Left Right

30 13 17 11 88 15 44 55 22 40 66 50 85

Left Right

30 13 17 11 22 15 44 55 88 40 66 50 85

Right Left

15 13 17 11 22 30 44 55 88 40 66 50 85

왼쪽 부분배열 피벗 오른쪽 부분배열

퀵 정렬의 예_계속

15 13 17 11 22 30 44 55 88 40 66 50 85

15 13 17 11 22
Left Right

15 13 11 17 22
Right Left

11 13 15 17 22 30 44 55 88 40 66 50 85

11 13
Right Left

11 13 15 17 22 30 44 55 88 40 66 50 85

11 13 15 17 22 30 44 55 88 40 66 50 85

퀵 정렬의 예_계속

11 13 15 17 22 30 44 55 88 40 66 50 85

11 13 15 17 22 30 44 55 88 40 66 50 85

11 13 15 17 22 30 44 55 88 40 66 50 85

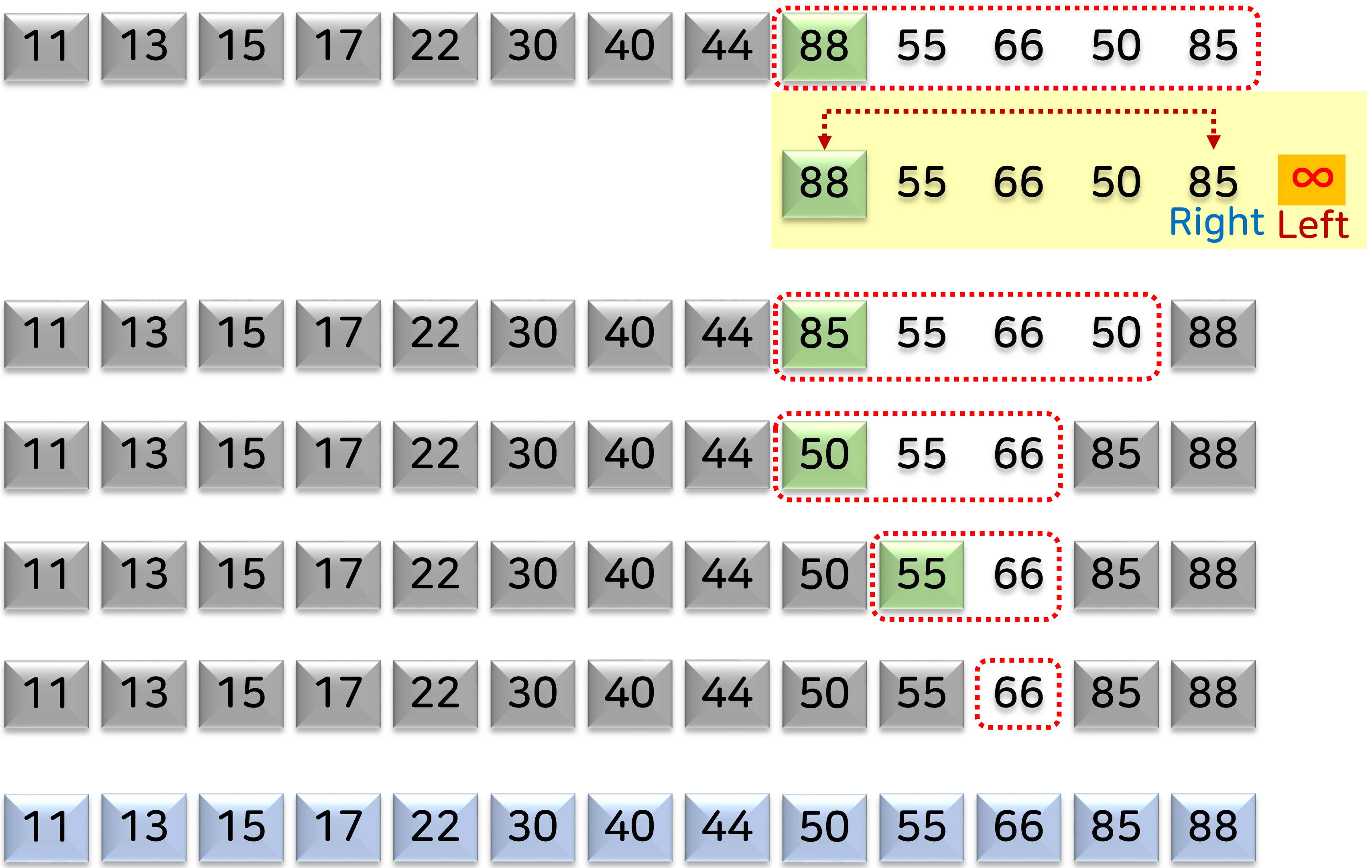
44 55 88 40 66 50 85
Left Right

44 40 88 55 66 50 85
Right Left

11 13 15 17 22 30 40 44 88 55 66 50 85

11 13 15 17 22 30 40 44 88 55 66 50 85

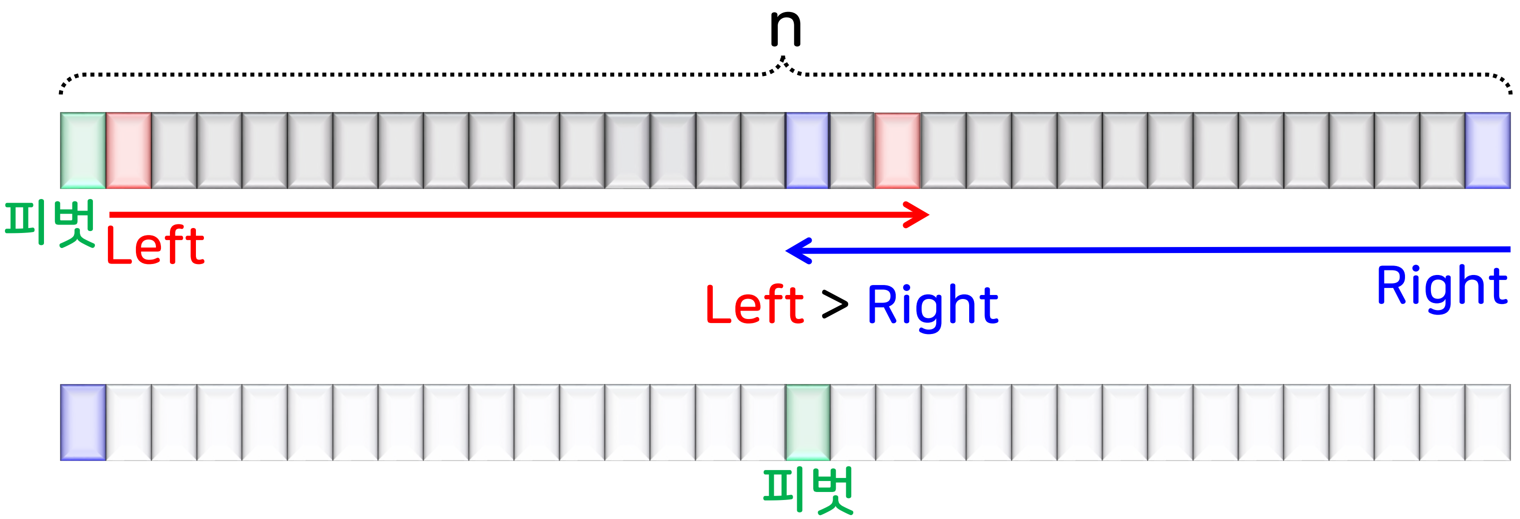
퀵 정렬의 예_계속


11 13 15 17 22 30 40 44 88 55 66 50 85
88 55 66 50 85 ∞
Right Left
11 13 15 17 22 30 40 44 85 55 66 50 88
11 13 15 17 22 30 40 44 50 55 66 85 88
11 13 15 17 22 30 40 44 50 55 66 85 88
11 13 15 17 22 30 40 44 50 55 66 85 88
11 13 15 17 22 30 40 44 50 55 66 85 88

성능과 특징

분할 함수 Partition()의 성능

- 피벗과의 비교 횟수?



각 데이터는 피벗과 1회 또는 많아야 2회씩 비교

$\Theta(n)$

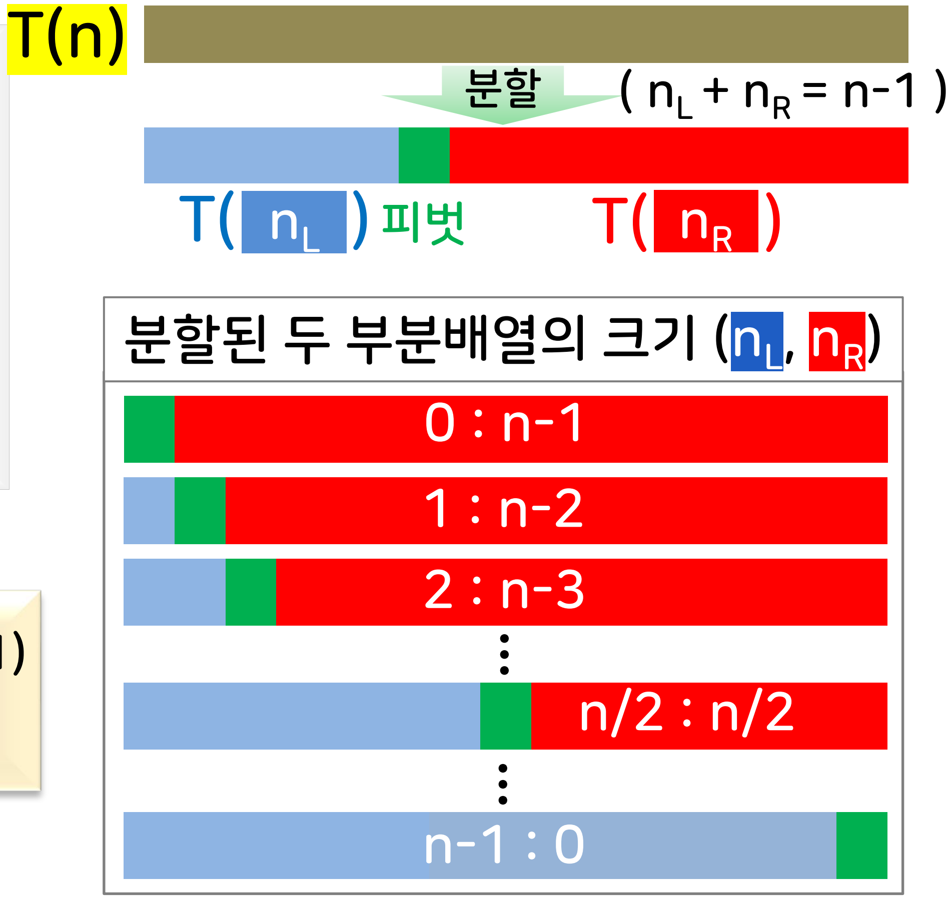
성능과 특징

퀵 정렬 QuickSort()의 수행시간은 분할되는 두 부분배열의 크기에 따라 달라짐

```
QuickSort (A[ ], n) {
  if (n > 1) {
    pivot = Partition (A[0..n-1], n);
    QuickSort (A[0..pivot-1], pivot);
    QuickSort (A[pivot+1..n-1], n-pivot-1);
  }
}
```

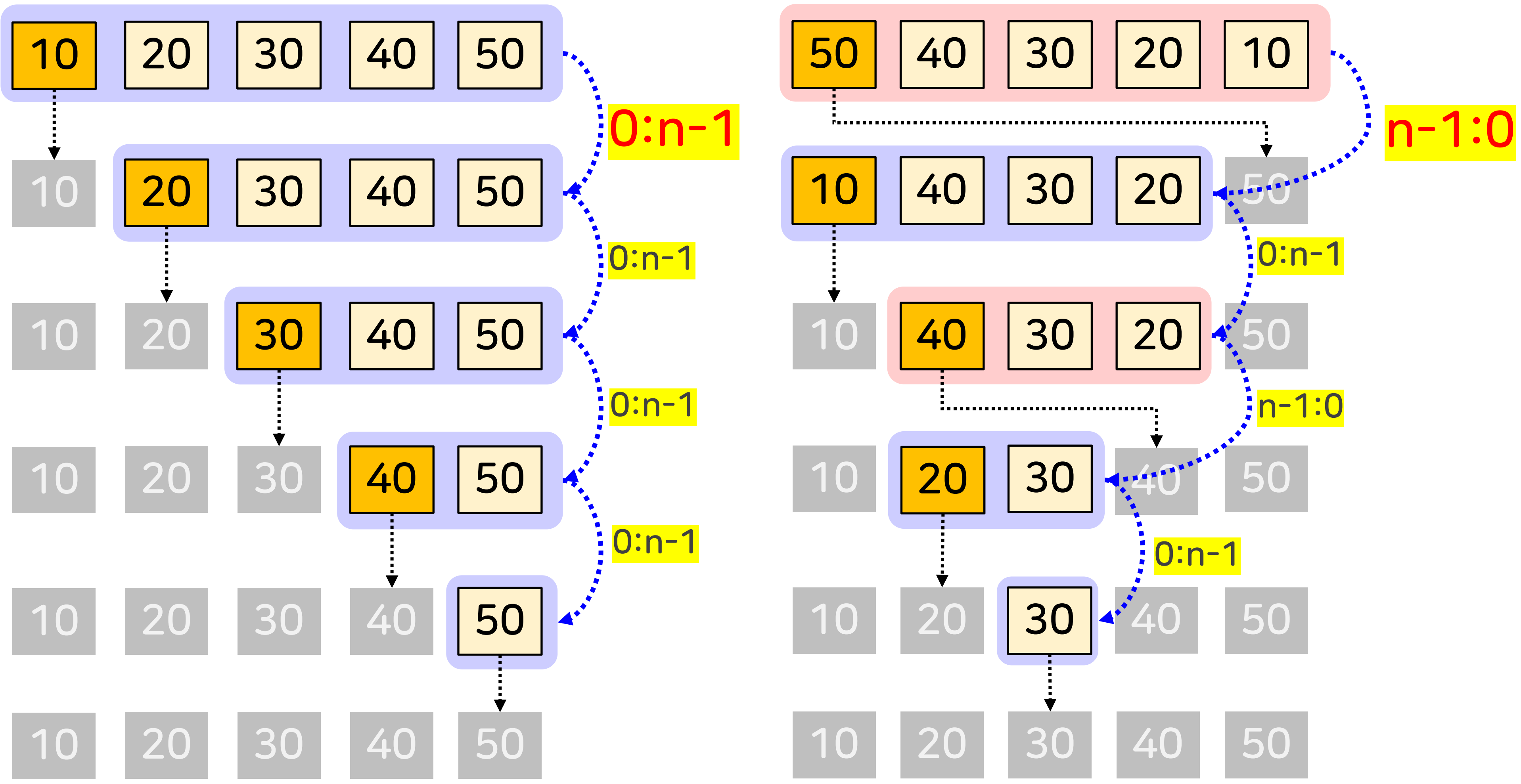
$T(n) = T(n_L) + T(n_R) + \Theta(n) \quad (n>1)$

$T(1) = \Theta(1)$



성능과 특징

▶ 배열이 항상 0:n-1 또는 n-1:0으로 분할되는 경우



성능과 특징

배열이 항상 0:n-1 또는 n-1:0으로 분할되는 경우

- 극심한 불균형적 분할 → 최악의 경우
  - ✓ 0:n-1, n-1:0 → 피벗만 제자리를 잡고 나머지 모든 원소가 하나의 부분배열이 되는 경우
  - ✓ 피벗이 항상 부분배열의 최솟값 또는 최댓값이 되는 경우
  - ✓ 입력 데이터가 정렬된 경우 AND 피벗을 배열의 첫 번째 원소로 정한 경우

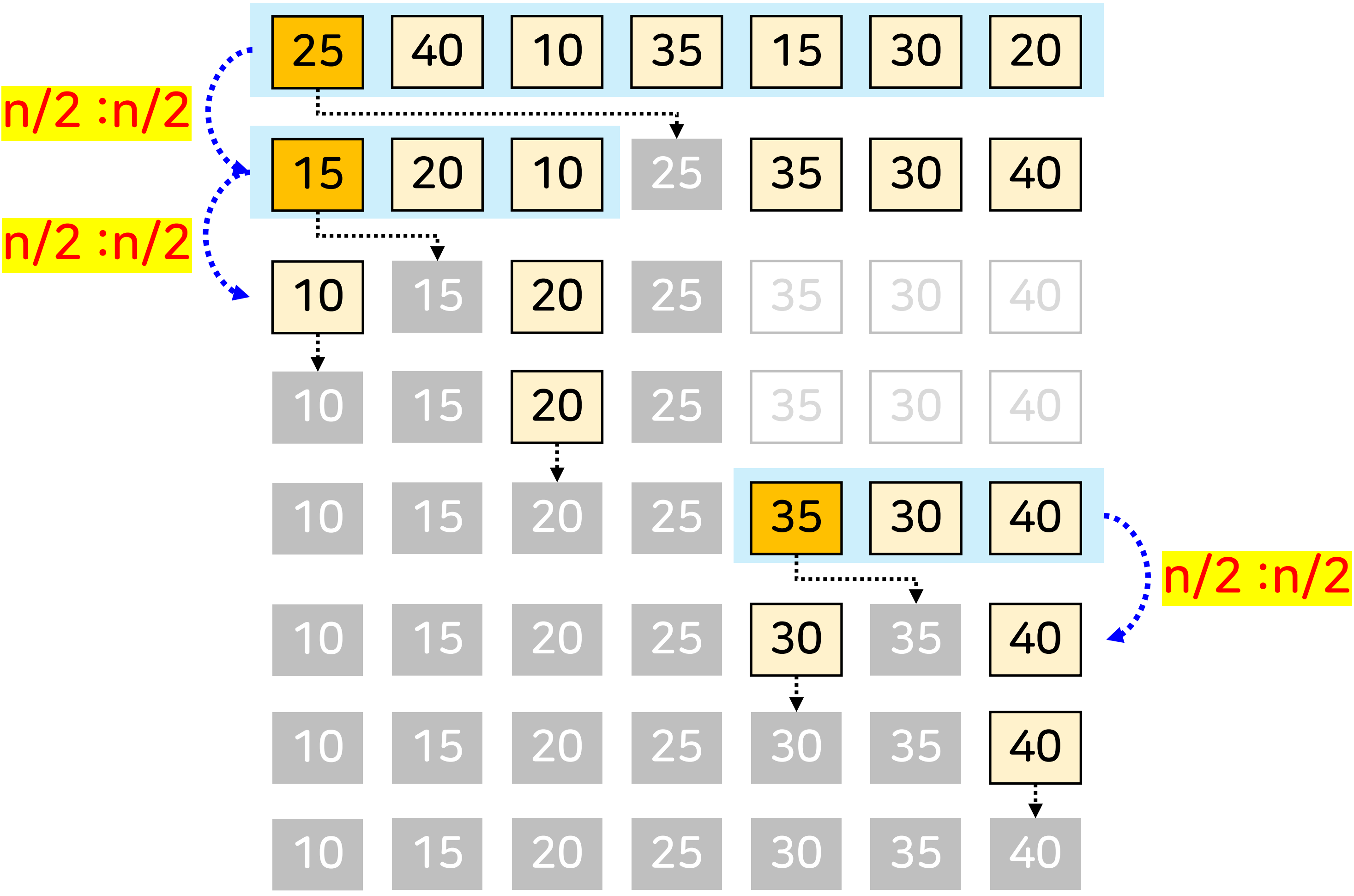
$$T(n) = T(n-1) + T(0) + \Theta(n) \quad (n>1),\ T(1)=1$$

$$T(n) = T(n-1) + \Theta(n)$$

$$T(n) = O(n^2)$$

성능과 특징

배열이 항상 $\frac{n}{2}:\frac{n}{2}$으로 분할되는 경우



성능과 특징

배열이 항상 $\frac{n}{2}:\frac{n}{2}$로 분할되는 경우

- 가장 균형적인 분할 → 최선의 경우
  - ✓ 피벗을 중심으로 항상 동일한 크기의 두 부분배열로 분할되는 경우
  - ✓ 피벗이 항상 배열의 중간값이 되는 경우

$$T(n) = T(\lfloor n/2 \rfloor) + T(\lfloor n/2 \rfloor) + \Theta(n) \ (n>1),\ T(1)=1$$

$$T(n) = 2T(n/2) + \Theta(n)$$

$$T(n) = O(n\log n)$$

성능과 특징

퀵 정렬의 평균 수행시간

- 부분배열의 모든 분할 비율에 따른 수행시간의 평균
  - ✓ 피벗은 동일한 확률을 가지고 분할 후 배열의 어느 곳에나 위치 가능
  - ✓ 0:n-1, 1:n-2, 2:n-3, ···, n-2:1, n-1:0

$$T(1) = T(0) = 0$$

$$T(n) = \frac{1}{n}\sum_{i=1}^{n}\left(T(i-1) + T(n-i)\right) + \Theta(n), \quad n \geq 2$$

T(n) = O(nlogn)

성능과 특징

피벗 선택의 임의성만 보장되면 평균 수행시간을 보장

- 최선/평균 수행시간 → O(nlogn)
- 최악의 수행시간 → $O(n^2)$
  - ✓ 피벗을 배열의 첫 번째 원소로 지정하는 경우 AND 배열이 정렬된 경우

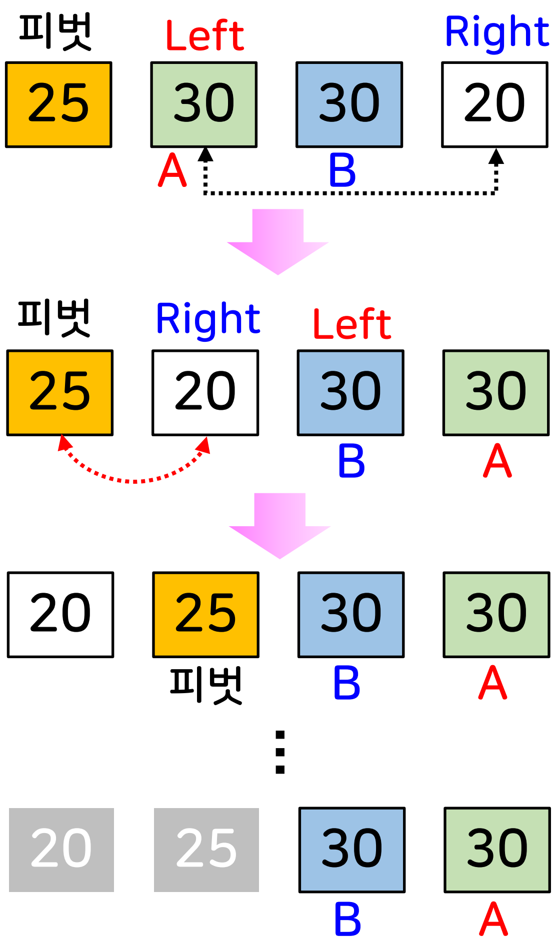
    → 배열에서 임의의 값을 선택한 후, 배열의 첫 번째 원소와 서로 교환한 후 정렬 수행

제자리 정렬 알고리즘

- 입력 배열 이외에 추가적인 저장 공간을 상수 개(Left, Right, tmp, n, pivot)만 사용

성능과 특징

안정적이지 않은 정렬 알고리즘



성능과 특징

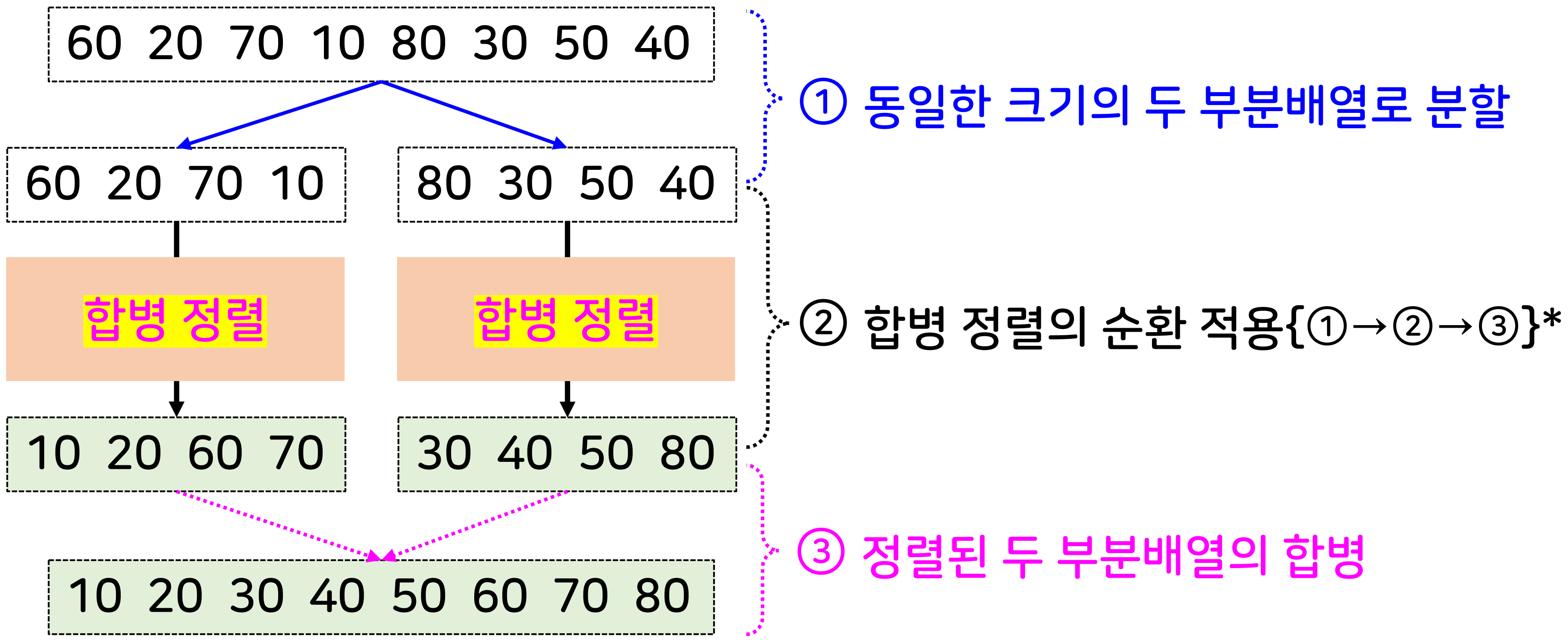
분할정복 방법이 적용된 알고리즘

- 분할
  - ✓ 피벗을 기준으로 주어진 배열을 두 부분배열로 분할 → 두 부분배열의 크기는 일정하지 않음
- 정복
  - ✓ 두 부분배열에 대해서 퀵 정렬을 순환적으로 적용하여 각 부분배열을 정렬함
- 결합
  - ✓ 필요 없음

02. 합병 정렬

합병 정렬?

주어진 배열을 동일한 크기의 두 부분배열로 분할하고, 각 부분배열에 순환적으로 합병 정렬을 적용하여 정렬시킨 후, 정렬된 두 부분배열을 합병하여 하나의 정렬된 배열을 만듦



합병 정렬 알고리즘

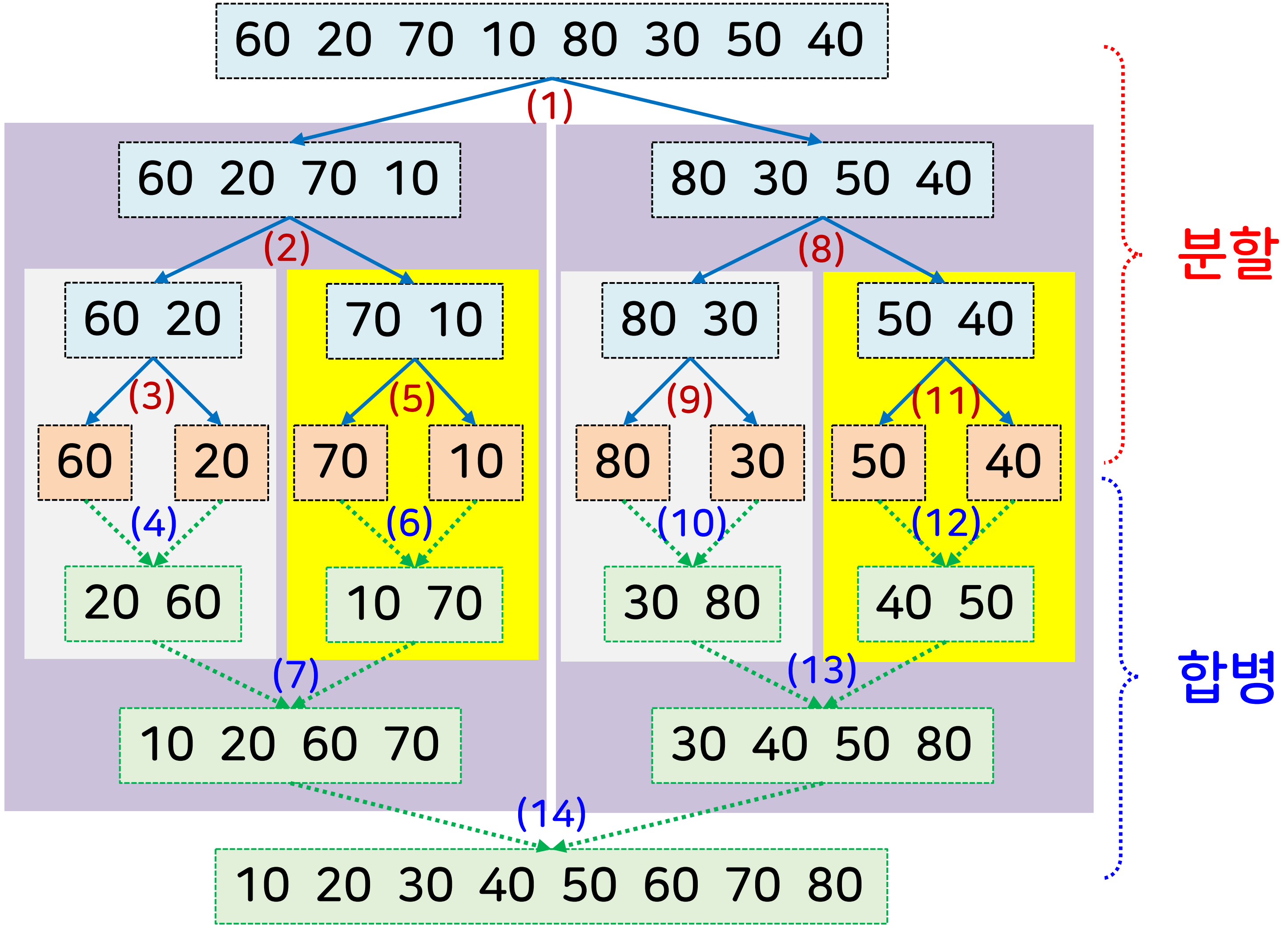

```
MergeSort (A[ ], n)
{
  if (n > 1) {
    Mid = ⌊ n / 2 ⌋;
    B[0..Mid-1] = MergeSort(A[0..Mid-1], Mid);
    C[0..n-Mid-1] = MergeSort(A[Mid..n-1], n-Mid);
    A[0..n-1] = Merge(B[0..Mid-1], C[0..n-Mid-1], Mid, n-Mid);
  }
  return (A);
}
```

왼쪽 부분배열의 순환 호출
→ 크기 n/2인 정렬된 배열 반환

오른쪽 부분배열의 순환 호출
→ 크기 n/2인 정렬된 배열 반환

정렬된 두 부분배열 B[ ]와 C[ ]의 합병: A[ ] = B[ ] + C[ ]

합병 정렬의 전체적인 수행 과정



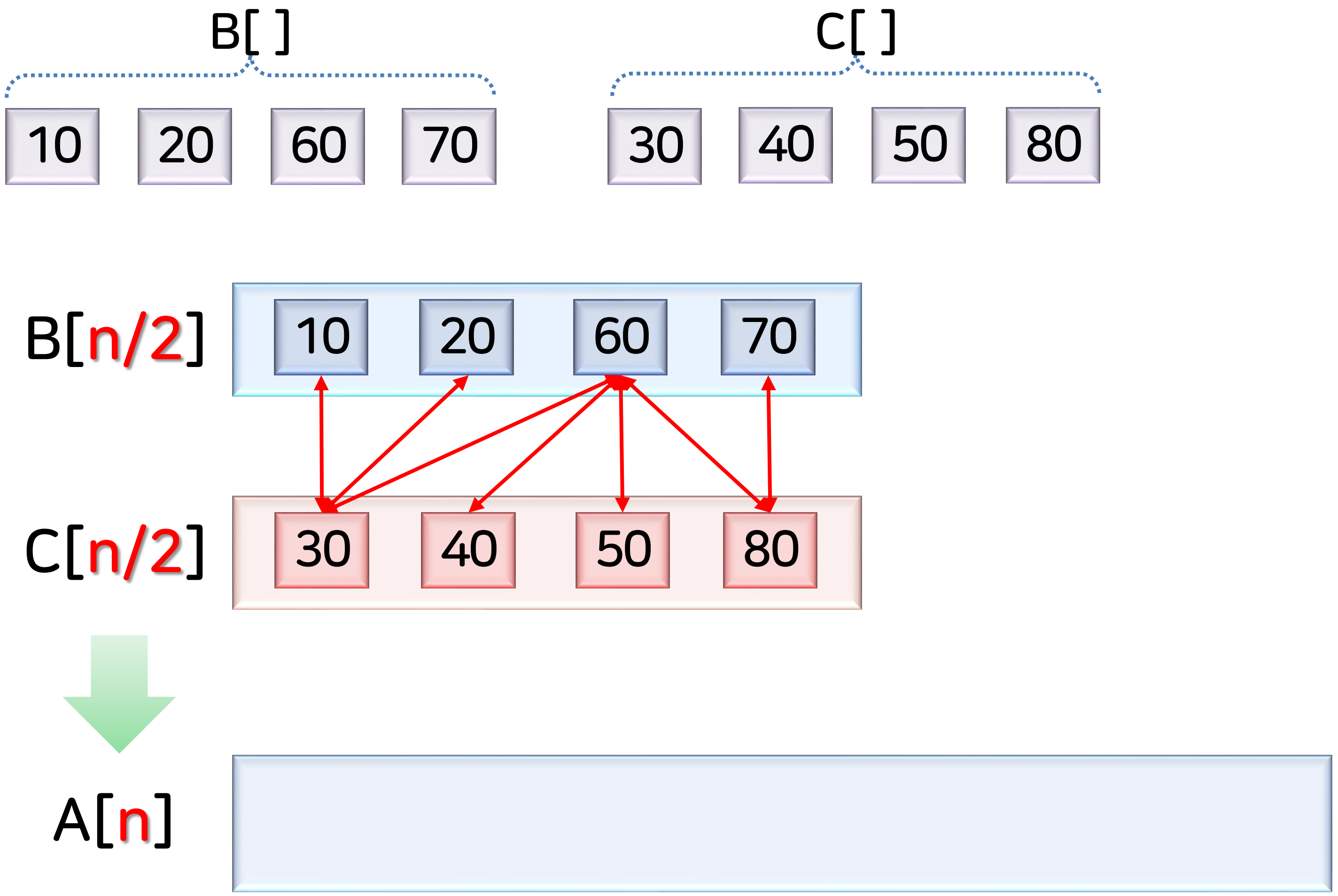
합병 함수_Merge( )


```
Merge (B[ ], C[ ], n, m)
{
  i = j = k = 0;
  while (i < n && j < m)
     if (B[i] <= C[j])
        A[k++] = B[i++];
     else A[k++] = C[j++];
  for ( ; i < n; i++) A[k++] = B[i];
  for ( ; j < m; j++) A[k++] = C[j];
  return (A[0..n+m-1]);
}
```

정렬된 부분배열 B[i]와 C[j]를 비교해서 작은 데이터를 A[k]에 복사

정렬된 부분배열 B[ ] 또는 C[ ]에 남아 있는 데이터를 A[ ]로 복사

합병 함수 Merge()의 동작

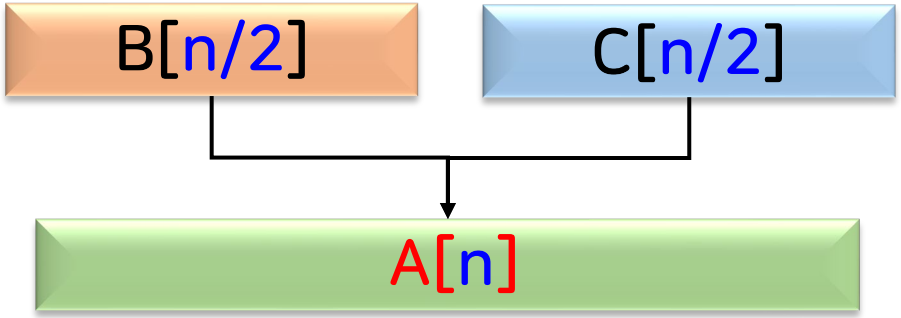


합병 함수 Merge()의 동작

	입력 배열 B[i]				입력 배열 C[j]				결과 배열 A[k] ← B[i]+C[j]							
k	0	1	2	3	0	1	2	3	0	1	2	3	4	5	6	7
0	10	20	60	70	30	40	50	80	10							
1	10	20	60	70	30	40	50	80	10	20						
2	10	20	60	70	30	40	50	80	10	20	30					
3	10	20	60	70	30	40	50	80	10	20	30	40				
4	10	20	60	70	30	40	50	80	10	20	30	40	50			
5	10	20	60	70	30	40	50	80	10	20	30	40	50	60		
6	10	20	60	70	30	40	50	80	10	20	30	40	50	60	70	
7	10	20	60	70	30	40	50	80	10	20	30	40	50	60	70	80

성능과 특징

합병 함수 Merge()의 수행시간



두 부분배열 B[ ]와 C[ ]간의 비교 횟수

$$\frac{n}{2} \sim \left(\frac{n}{2}+\frac{n}{2}-1=n-1\right)$$

최악의 경우

Θ(n)

성능과 특징

합병 정렬 MergeSort()의 수행시간

```
MergeSort (A[ ], n)
{
  if (n > 1) {
    Mid = ⌊ n / 2 ⌋ ;
    B[0..Mid-1] = MergeSort(A[0..Mid-1], Mid);
    C[0..n-Mid-1] = MergeSort(A[Mid..n-1], n-Mid);
    A[0..n-1] = Merge(B[0..Mid-1], C[0..n-Mid-1], Mid, n-Mid);
  }
  return (A);
}
```

$T(n)$

$T(\lfloor n/2 \rfloor)$

$T(\lceil n/2 \rceil)$

$\theta(n)$

$$T(n) = T(\lfloor n/2 \rfloor) + T(\lceil n/2 \rceil) + \Theta(n) \quad (n>1),\ T(1)=0$$

$$T(n) = 2T(n/2) + \Theta(n)$$

(최선, 최악, 평균) $T(n) = O(n\log n)$

성능과 특징

안정적인 정렬 알고리즘

- 합병 과정에서 동일한 두 데이터에 대해서 항상 왼쪽 데이터를 먼저 선택함

제자리 정렬 알고리즘이 아님

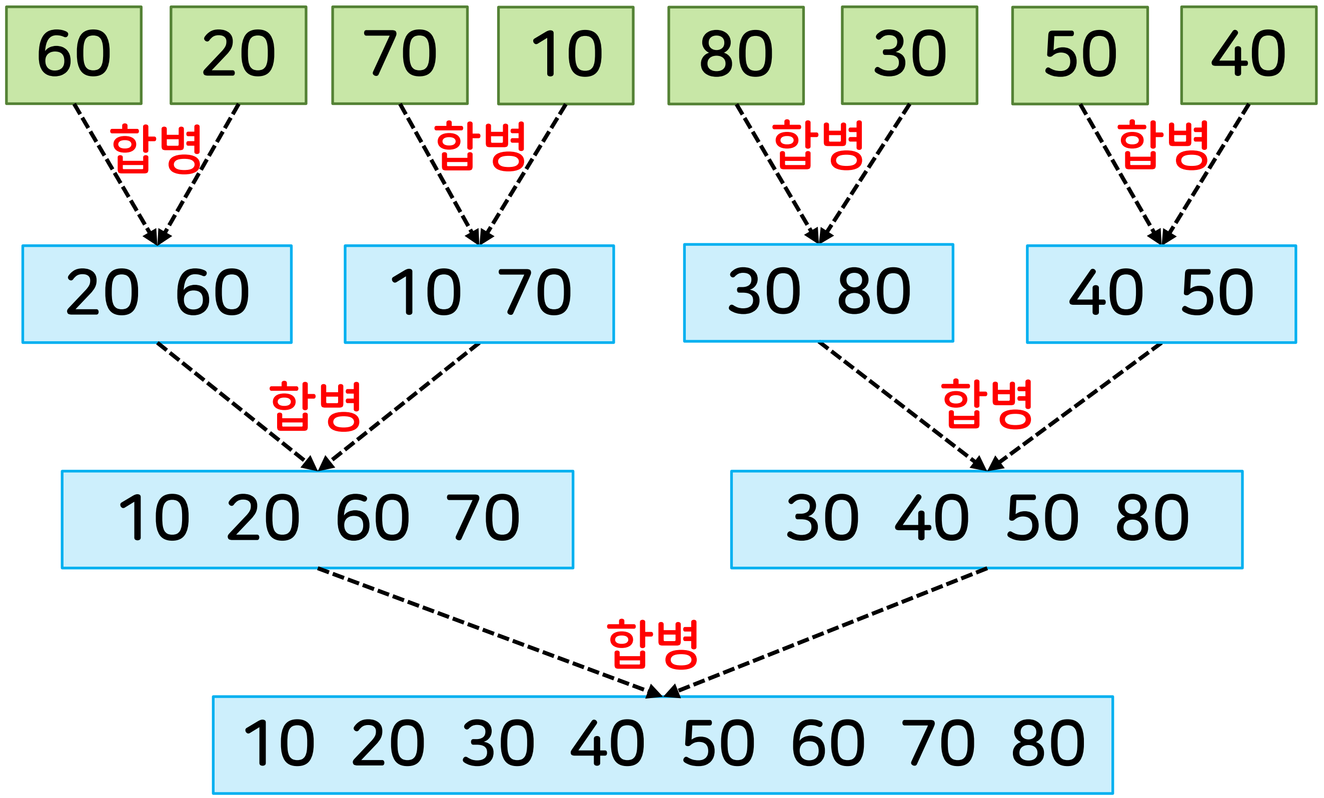
- A[n] = B[n/2] + C[n/2] → 입력 크기 n만큼의 추가적인 공간을 요구

전형적인 분할정복 방법이 적용됨

- 분할 → 주어진 배열을 동일한 크기의 2개의 부분배열로 분할
- 정복 → 각 부분배열에 대해서 합병 정렬을 순환적으로 적용하여 정렬함
- 결합 → 정렬된 두 부분배열을 합병하여 하나의 정렬된 배열을 만듦

성능과 특징

비순환적 방식의 합병 정렬



정리하기

1. 퀵 정렬

- "피벗", 분할 함수 O(n)
- 분할되는 두 부분배열의 크기에 따라 성능이 달라짐
  → 최악 $O(n^2)$, 최선/평균 O(nlogn)
- 불안정적, 제자리
- 피벗 선택의 임의성만 보장되면 평균적인 성능 O(nlogn)을 보임
- 분할정복 방법이 적용됨

2. 합병 정렬

- 합병 함수 O(n), 최악/최선/평균 O(nlogn)
- 안정적 정렬, 제자리 정렬이 아님
- 전형적인 분할정복 방법이 적용됨

ALGORITHM
알고리즘

다음시간에는

Lecture **05**

정렬 (3)

컴퓨터과학과 | **이관용 교수**