정렬 (Sorting)

강 지 훈
jhkang@cnu.ac.kr

충남대학교 컴퓨터공학과

정렬 (Sorting)

정렬 (Sorting)

- 컴퓨터가 하는 일 중에서 상당한 부분이 정렬이다.
  - 25~50 %
- 효율적인 방법이 매우 중요
- 모든 경우에 적용할 수 있는 최선의 정렬 방법은 없다 !!!

정렬은 순열 (permutation) 을 찾는 것

- 예: (32, 10, 44, 21,57)
  - 레코드 R_2 의 키 값 = 44
  - σ = (1, 3, 0, 2, 4) 는 주어진 정렬로 찾은 순열 (permutation):
    - $R_{\sigma(0)} = R_1 = 10$
    - $R_{\sigma(1)} = R_3 = 21$
    - $R_{\sigma(2)} = R_0 = 32$
    - $R_{\sigma(3)} = R_2 = 44$
    - $R_{\sigma(4)} = R_4 = 57$
  - 당연히, 다음 부등식이 성립:

    $R_{\sigma(0)} \le R_{\sigma(1)} \le R_{\sigma(2)} \le R_{\sigma(3)} \le R_{\sigma(4)}$.
  - 그러므로 정렬된 순서의 데이터는,

    $(R_{\sigma(0)}, R_{\sigma(1)}, R_{\sigma(2)}, R_{\sigma(3)}, R_{\sigma(4)}) = (10, 21, 32, 44, 57)$

안정적인 (stable) 정렬

- 정렬 방법 S가 안정적 (stable):
  - S를 사용하여 정렬했을 때, 동일한 키 값을 갖는 두 레코드의 정렬 전의 상대적 순서가 정렬 후에 바뀌지 않는다.

- 예: (14, 19, 12, 13, 14)
  - 다른 두 정렬 방법 S 와 T:
    - σ_S = (2, 3, 0, 4, 1) : 정렬 방법 S 가 찾아준 순열
    - σ_T = (2, 3, 4, 0, 1) : 정렬 방법 T 가 찾아준 순열
  - 정렬 S 와 T 가 각각 찾아준 정렬된 키의 순서는 동일:
    - (12, 13, 14, 14, 19)
  - 그렇지만 엄밀하게는:
    - S: (12, 13, 14, 14, 19)
    - T: (12, 13, 14, 14, 19)

안정적인 (stable) 정렬

- 정렬방법 S는 안정적이다
  - R $\sigma_S(2)$ = R $\sigma_S(3)$ = 14
  - $\sigma_S(2) = 0$, $\sigma_S(3) = 4$. 그러므로, $\sigma_S(2) < \sigma_S(3)$ 이 성립한다.
  - 즉, 동일한 키 값을 갖는 두 레코드의 상대적 순서가 정렬 후에 바뀌지 않았다.
- 정렬방법 T는 안정적이지 않다
  - R$\sigma_T(2)$ = R$\sigma_T(3)$ = 14
  - $\sigma_T(2) = 4$, $\sigma_T(3) = 0$. 그러므로, $\sigma_T(2) < \sigma_T(3)$ 이 성립하지 않는다.
  - 즉, 동일한 키 값을 갖는 두 레코드의 상대적 순서가 정렬 후에 바뀌었다.

내부 정렬과 외부 정렬

내부 정렬과 외부 정렬

- 내부 정렬 (Internal Sort)
  - 정렬할 원소의 리스트가 메모리에 있다.
  - 정렬하는 동안 메모리만 사용한다.
- 외부 정렬 (External Sort)
  - 리스트가 너무 커서 한번에 모든 원소를 메모리에 저장할 수 없다.
  - 하드 디스크와 같은 보조 기억장치를 사용한다.

외부 정렬의 기본적인 방법

- 단계 1: (Sort)
  - 원래 리스트를 메모리에 가져올 수 있을 만큼의 여러 개의 작은 부분 리스트로 분할한다.
  - 각각의 분할된 부분 리스트를 내부 정렬 방법을 사용하여 정렬(Sort)한다.
  - 정렬된 것은 보조 기억장치에 보관한다.
- 단계 2: (Merge)
  - 보조 기억장치에 저장된 부분 리스트들을 몇 개씩 묶어 병합(merge)하여 하나의 정렬된 리스트로 만든다.

삽입 정렬 (Insertion Sort)

삽입 정렬 (Insertion Sort)

- 초기에, (R_0 , R_1)은 이미 정렬된 상태
  - R_0 : 키 값으로 -∞를 갖는 보초(sentinel) 레코드
    - 보초(sentinel) 레코드는 단지 프로그램의 효율을 위한 것
    - 리스트의 왼쪽 끝에 -∞ 의 보초를 세운 것
- 매번 다음 레코드를 제 위치에 삽입한다.

i	[0]	[1]	[2]	[3]	[4]	[5]
-	-∞	4	2	5	1	3
1	-∞	2	4	5	1	3
2	-∞	2	4	5	1	3
3	-∞	1	2	4	5	3
4	-∞	1	2	3	4	5

(R_0, R_1)
(R_0, R_2, R_1)
(R_0, R_2, R_1, R_3)
$(R_0, R_4, R_2, R_1, R_3)$
$(R_0, R_4, R_2, R_5, R_1, R_3)$

예제: 최악의 경우와 최선의 경우

최악의 경우

i	[0]	[1]	[2]	[3]	[4]	[5]
-	-∞	5	4	3	2	1
1	-∞	4	5	3	2	1
2	-∞	3	4	5	2	1
3	-∞	2	3	4	5	1
4	-∞	1	2	3	4	5

비교 회수
2
3
4
5
14

최선의 경우

i	[0]	[1]	[2]	[3]	[4]	[5]
-	-∞	1	2	3	4	5
1	-∞	1	2	3	4	5
2	-∞	1	2	3	4	5
3	-∞	1	2	3	4	5
4	-∞	1	2	3	4	5

비교 회수
1
1
1
1
4

삽입 정렬의 구현

```
public void  insertionSort (Element elements[], int size)
{
    int  i, j ;
    element  next ;
    for ( i = 2 ; i <= size ; i++ ) {
        next = elements[i] ;
        for (j = i -1; next.key < elements[j].key ; j--)
            elements[j+1] = elements[j] ;
        elements[j+1] = next ;
    }
}
```

```
// 보초(sentinel) 레코드가 사용되지 않을 경우
(j >= 0) &&  (next.key < elements[j].key)
```

삽입 정렬의 분석

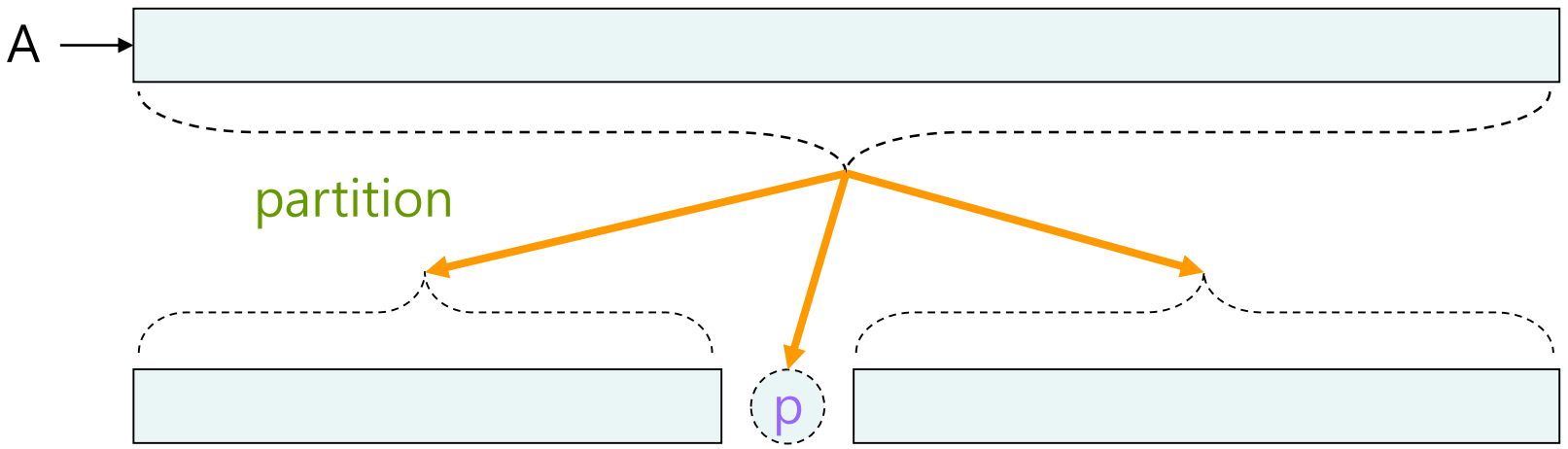
- 최악의 경우: 역순으로 정렬되어 있는 경우
  - $O(n^2)$
- 최선의 경우: 이미 정렬되어 있는 경우
  - $O(n)$
- 관찰
  - 헝클어진 정도가 심하지 않다면, 좋은 성능을 보여준다.
  - 자료가 20 개 이하일 때 좋다.

퀵 정렬 (Quick Sort)

퀵 정렬

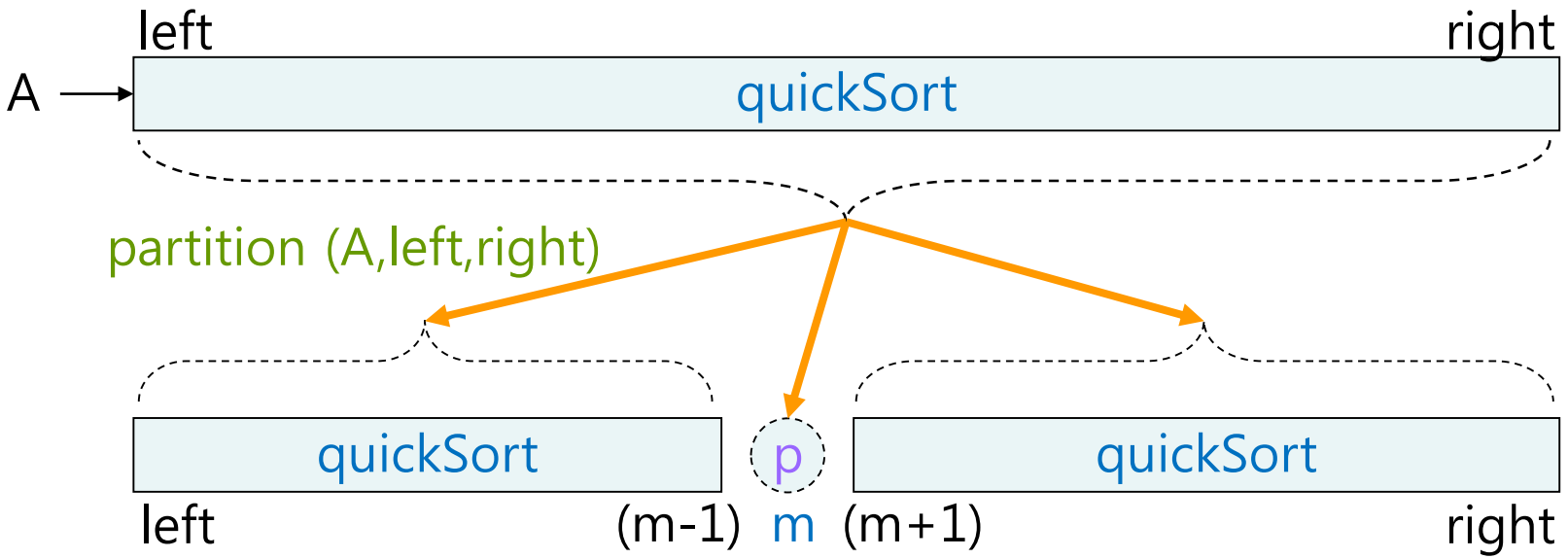
- C.A.R Hoare 가 발명
- 재귀 알고리즘
- 평균적 성능이 아주 좋다

퀵 정렬의 아이디어: 파티션



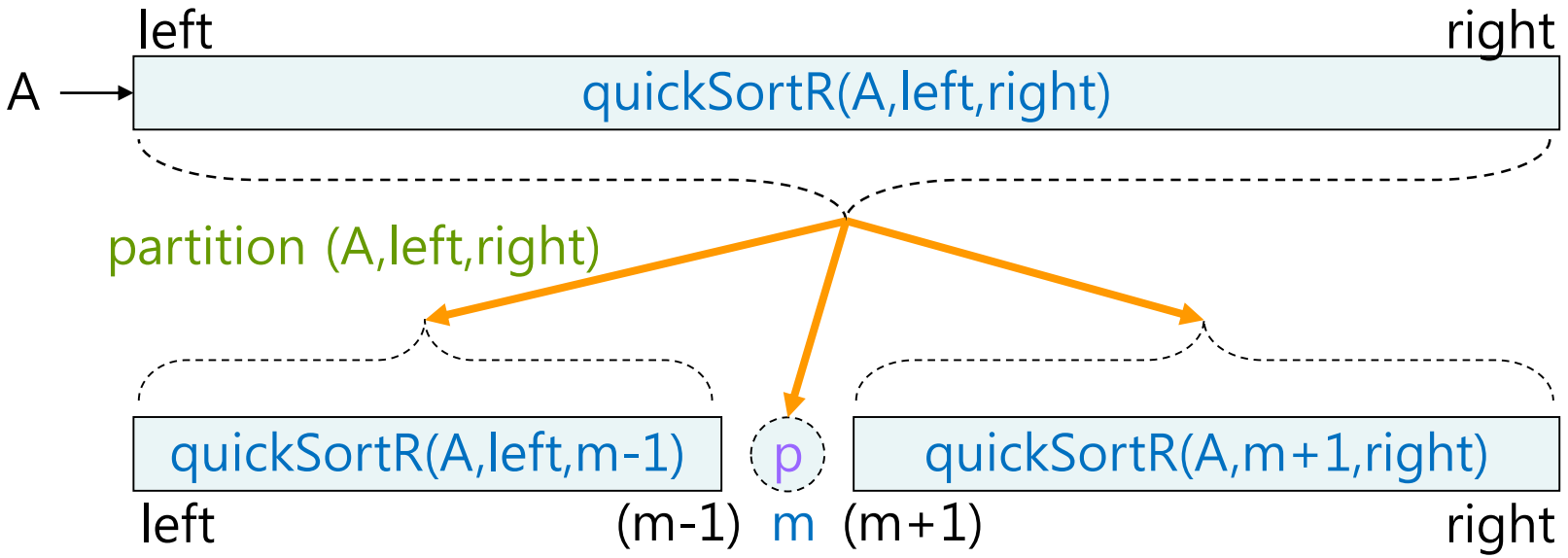
- 파티션(partition) 하기
  - 원소들을 특정 값을 기준으로 두 부분으로 나눈다.
    - 특정 값보다 작은 원소들은 왼쪽에, 큰 원소들은 오른쪽에 오게 하고, 그 사이에 특정 값이 놓이게 한다. (오름차순으로 정렬한다고 가정)
    - 특정 값을 피봇 (pivot) 값이라고 한다.

퀵 정렬의 아이디어: 재귀적으로



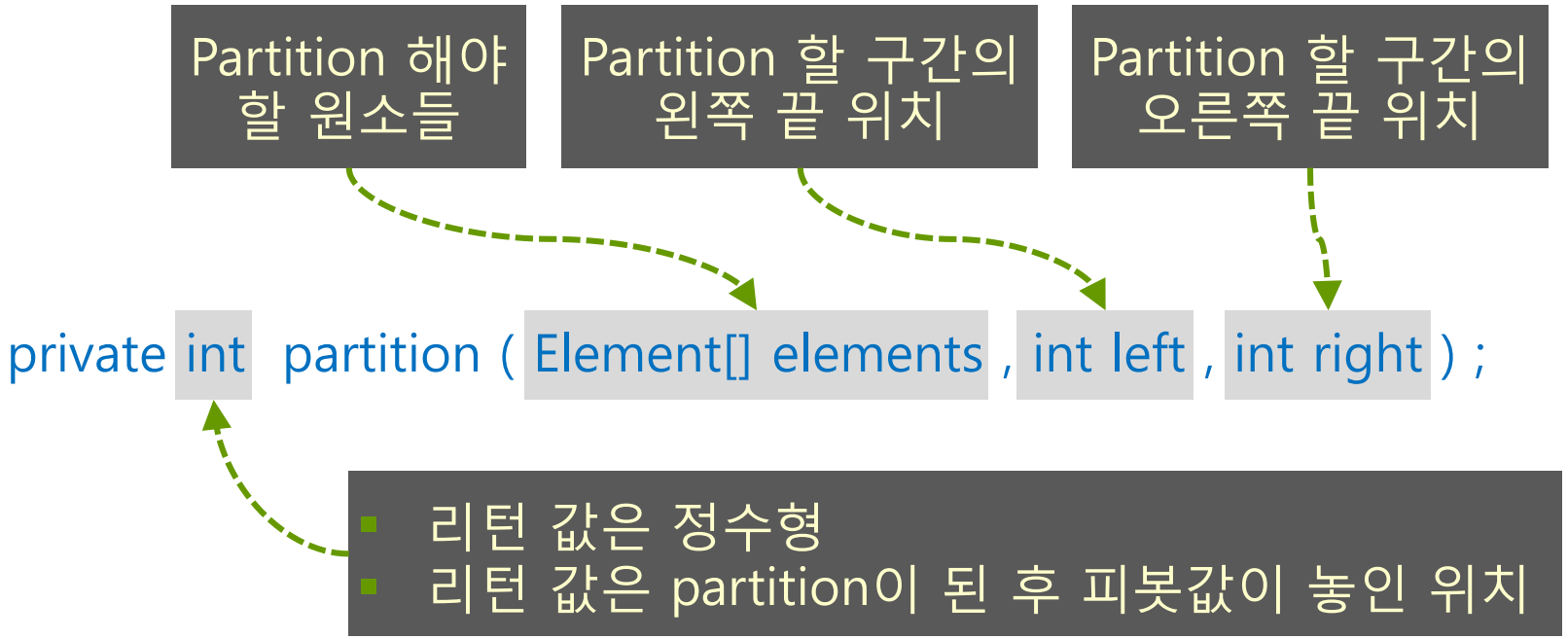
- 전체를 quickSort 하기 위해서는, partition을 수행하여 만들어진 크기가 작아진 각각의 부분을 quickSort 한다.
  - 그러므로, 주어진 크기의 배열을 quickSort 하는 문제가, 크기가 작아진 두 개의 quickSort 문제로 바뀌었다.
- 탈출(exit) 경우
  - quickSort 할 구간의 크기가 0 또는 1 이면, 그 자체가 이미 정렬이 되어 있는 상태이다.

재귀 알고리즘 개요




```
public void quickSortRecursively (Element[] A, int left, int right)
{
    if (left < right) /* 구간의 크기가 2 이상이면 */ {
        int mid = partition (A, left, right) ;          // DIVIDE
        quickSortRecursively (A, left, mid-1) ;         // CONQUER
        quickSortRecursively (A, mid+1, right) ;        // CONQUER
    }
}
```

함수 partition()



피봇값 정하기

- 구간 안의 아무 원소를 피봇으로 정해도 된다
- 간편한 방법: 가장 왼쪽 값을 택하여 피봇 값으로 정한다.
  - A[left]의 값이 피봇이 된다.

파티션의 예

$[R_0$	R_1	R_2	R_3	R_4	R_5	R_6	R_7	R_8	$R_9]$
26	5	37	1	61	11	59	15	48	19
		u=2							d=9

[0	1	2	3	4	5	6	7	8	9]
26	5	19	1	61	11	59	15	48	37
				u=4			d=7		

[0	1	2	3	4	5	6	7	8	9]
26	5	19	1	15	11	59	61	48	37
					d=5	u=6			

At this time, u>d

[0	1	2	3	4	5	6	7	8	9]
11	5	19	1	15	26	59	61	48	37

함수 partition()

```
private int  partition (Element[] elements, int left, int right)
{
    int  pivot = elements[left] ;
    int  up = left ;
    int  down = right+1 ;
    do {
        do { up++ } while (elements[up] < pivot) ;
        do { down-- } while (elements[down] > pivot) ;
        if (up < down) {
                swap (elements, up, down) ;
        }
    } while (up < down) ;
    swap (elements, left, down) ;
    return down ; // 피봇 위치가 down 이다
}
```

완성된 퀵 정렬 알고리즘 [1]

```
public void quickSortRecursively (Element[] elements, int left, int right)
{
    if (left < right) {
        // 파티션
            // pivot의 왼쪽에는 pivot 의 key 값보다 작은 원소들이 오게 하며,
            // pivot의 오른쪽에는 pivot 의 key 값보다 큰 원소들이 오게 한다 ;
        int mid = partition(elements, left, right) ; //파티션 후의 피봇 위치
        quickSortRecursively (elements, left, mid-1) ;
        quickSortRecursively (elements, mid+1, right) ;
    }
}
```

완성된 퀵 정렬 알고리즘 [2]

```
public void quickSortRecursively (Element[] elements, int left, int right)
{
    if (left < right) {
        // 파티션
            // pivot의 왼쪽에는 pivot 의 key 값보다 작은 원소들이 오게 하며,
            // pivot의 오른쪽에는 pivot 의 key 값보다 큰 원소들이 오게 한다
        int  pivot = elements[left] ; // pivot 원소를 정한다
        int  up= left ;
        int  down = right+1 ;
        do {
            do { up++; } while (element[up] < pivot) ;
            do { down--; } while (element[down] > pivot) ;
            if ( up < down ) {
                swap(elements, up, down) ;
            }
        } while ( up < down) ;
        swap (elements, left, down) ; // pivot과 down 위치의 원소를 맞바꾼다
        int  mid = down ; //파티션 후의 피봇 위치가 mid 이다
        quickSortRecursively (elements, left, mid-1) ;
        quickSortRecursively (elements, mid+1, right) ;
    }
}
```

예: 퀵 정렬

R_0	R_1	R_2	R_3	R_4	R_5	R_6	R_7	R_8	R_9	R_{10}	l	r
(26	5	37	1	61	11	59	15	48	19)	+∞	0	9
(11	5	19	1	15)	26	(59	61	48	37)		0	4
(1	5)	11	(19	15)	26	(59	61	48	37)		0	1
()1	(5)	11	(19	15)	26	(59	61	48	37)		0	-1
1	(5)	11	(19	15)	26	(59	61	48	37)		1	1
1	5	11	(19	15)	26	(59	61	48	37)		3	4
1	5	11	(15)	19()	26	(59	61	48	37)		3	3
1	5	11	15	19()	26	(59	61	48	37)		5	4
1	5	11	15	19	26	(59	61	48	37)		6	9
1	5	11	15	19	26	(48	37)	59	(61)		6	7
1	5	11	15	19	26	(37)	48()	59	(61)		6	6
1	5	11	15	19	26	37	48()	59	(61)		8	7
1	5	11	15	19	26	37	48	59	(61)		9	9
1	5	11	15	19	26	37	48	59	61			

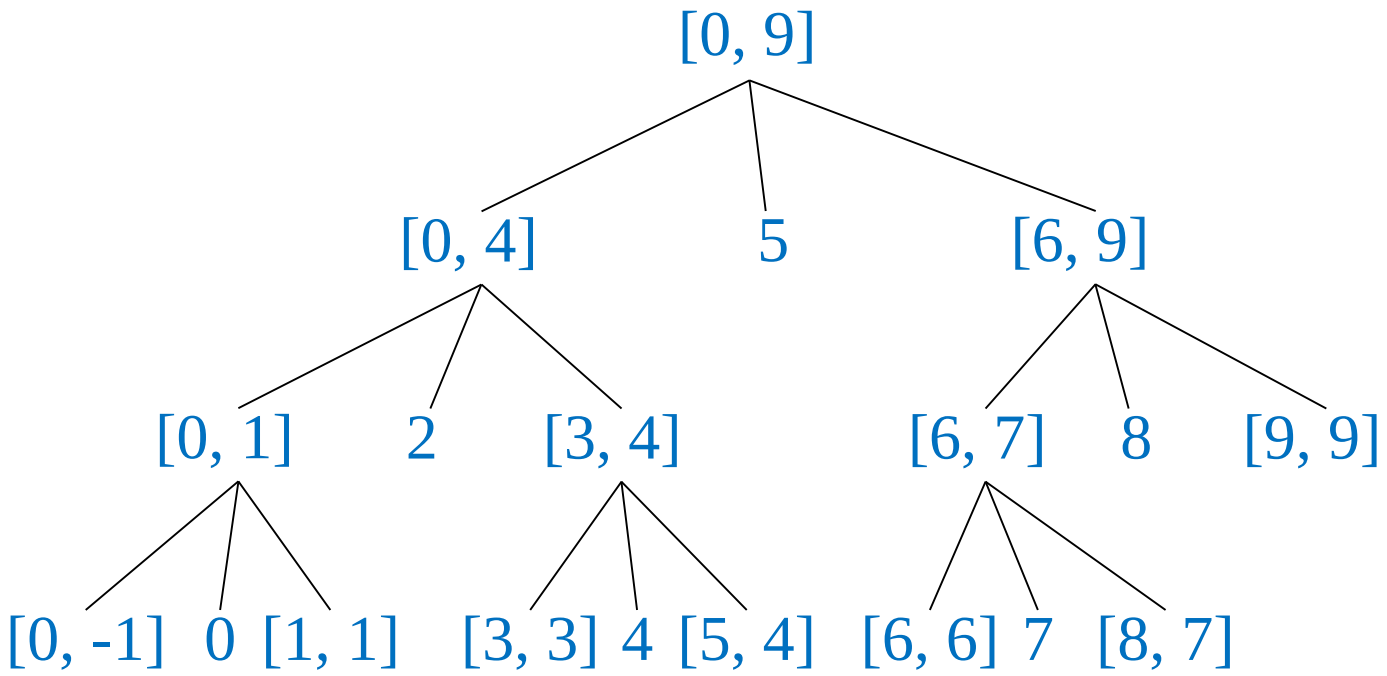
최종 퀵 정렬 알고리즘

```
public void quickSort (Element[] elements, int size)
{
	if (size > 1) {
		maxLoc = 0 ;
		for (int i = 1 ; i < size ; i++) {
			if (elements[i] > elements[maxLoc] ) {
				maxLoc = i ;
			}
		}
		swap (elements, maxLoc, size-1) ;
		quickSortRecursively (elements, 0, size-2) ;
	}
}

void  quickSortRecursively (Element[] elements, int left, int right)
{
	int  pivot, up , down ;
	if (left < right) {
		pivot = left ; // pivot 원소를 정한다 ;
		// pivot의 왼쪽에는 pivot 의 key 값보다 작은 원소들이 오게 하며,
		// pivot의 오른쪽에는 pivot 의 key 값보다 큰 원소들이 오게 한다 ;
		up = left ;
		down = right+1 ;
		do {
			do { up++; } while (elements[pivot] > elements[up]) ;
			do { down--; } while (elements[pivot] < elements[down]) ;
			if ( up < down ) {
				swap(elements, up, down) ;
			}
		} while ( up < down )
		swap (elements, pivot, down) ; // pivot과 down 위치의 원소를 맞바꾼다
		mid = down ; //파티션 후의 피봇 위치를 mid 라고 하자 ;
		quickSortRecursively (elements, left, mid-1) ;
		quickSortRecursively (elements, mid+1, right) ;
	}
}
```

```
private void swap(Element[] elements, int left, int right)
{
	Element temp = elements[left] ;
	elements[left] = elements[right] ;
	elements[right] =temp ;
}
```

예: 퀵 정렬에서의 재귀 호출



- 재귀 호출의 총 회수: 13
- 재귀 호출의 최대 깊이: 4

퀵 정렬의 시간복잡도

- 크기가 n인 배열이 크기가 각각 j 와 (n-j-1) 인 두 개의 작은 배열로 파티션 된다고 하자

- 시간복잡도를 나타내는 점화식

$$T(n) = \begin{cases} c_0 & \text{if } n = 0 \\ c_1 n + T(j) + T(n-j-1) + c_2 & \text{if } n \geq 1 \end{cases}$$

최악의 경우의 예: 이미 오름차순으로

R_0	R_1	R_2	R_3	R_4	R_5	R_6	R_7	R_8	R_9	R_{10}	l	r
(10	11	22	33	44	55	66	77	88	99)	+∞	0	9
()10	(11	22	33	44	55	66	77	88	99)		0	-1
10	(11	22	33	44	55	66	77	88	99)		1	9
10	()11	(22	33	44	55	66	77	88	99)		1	0
10	11	(22	33	44	55	66	77	88	99)		2	9
10	11	()22	(33	44	55	66	77	88	99)		2	1
10	11	22	(33	44	55	66	77	88	99)		3	9
10	11	22	()33	(44	55	66	77	88	99)		3	2
10	11	22	33	(44	55	66	77	88	99)		4	9
10	11	22	33	()44	(55	66	77	88	99)		4	3
10	11	22	33	44	(55	66	77	88	99)		5	9
10	11	22	33	44	()55	(66	77	88	99)		5	4
10	11	22	33	44	55	(66	77	88	99)		6	9
10	11	22	33	44	55	()66	(77	88	99)		6	5
10	11	22	33	44	55	66	(77	88	99)		7	9
10	11	22	33	44	55	66	()77	(88	99)		7	6
10	11	22	33	44	55	66	77	(88	99)		8	9
10	11	22	33	44	55	66	77	()88	(99)		8	7
10	11	22	33	44	55	66	77	88	(99)		9	9
10	11	22	33	44	55	66	77	88	99			

최악의 시간복잡도: 이미 내림차순으로

R_0	R_1	R_2	R_3	R_4	R_5	R_6	R_7	R_8	R_9	R_{10}	l	r
(99	88	77	66	55	44	33	22	11	10)	+∞	0	9
(10	88	77	66	55	44	33	22	11)	99()		0	8
()10	(88	77	66	55	44	33	22	11)	99()		0	-1
10	(88	77	66	55	44	33	22	11)	99()		1	8
10	(11	77	66	55	44	33	22)	88()	99()		1	7
10	()11	(77	66	55	44	33	22)	88()	99()		1	0
10	11	(77	66	55	44	33	22)	88()	99()		2	7
10	11	(22	66	55	44	33)	77()	88()	99()		2	6
10	11	()22	(66	55	44	33)	77()	88()	99()		2	1
10	11	22	(66	55	44	33)	77()	88()	99()		3	6
10	11	22	(33	55	44)	66()	77()	88()	99()		3	5
10	11	22	()33	(55	44)	66()	77()	88()	99()		3	2
10	11	22	33	(55	44)	66()	77()	88()	99()		4	5
10	11	22	33	(44)	55()	66()	77()	88()	99()		4	4
10	11	22	33	44	55()	66()	77()	88()	99()		6	5
10	11	22	33	44	55	66()	77()	88()	99()		7	6
10	11	22	33	44	55	66	77()	88()	99()		8	7
10	11	22	33	44	55	66	77	88()	99()		9	8
10	11	22	33	44	55	66	77	88	99()		10	9
10	11	22	33	44	55	66	77	88	99			

퀵 정렬의 최악 시간복잡도

- 최악의 경우: 이미 정렬되어 있는 경우
  - T(n)에서 j 가 항상 0

$$T_{wc}(n) = \begin{cases} c_0 & \text{if } n = 0 \\ c_1 n + T_{wc}(0) + T_{wc}(n-1) + c_2 & \text{if } n \geq 1 \end{cases}$$

- $T_{wc}(n) = O(n^2)$

퀵 정렬의 최악 시간복잡도

- 퀵 정렬에서 대부분의 시간은 파티션

$$
\begin{aligned}
T_{wc}(n) &= c_1 n + T_{wc}(0) + T_{wc}(n-1) + c_2 \\
&= c_1 n + c_0 + (c_1(n-1) + T_{wc}(0) + T_{wc}(n-2) + c_2) + c_2 \\
&= c_1(n + (n-1)) + 2(c_0 + c_2) + T_{wc}(n-2) \\
&\cdots\cdots \\
&= c_1 \sum_{i=1}^{n} i + n(c_0 + c_2) + T_{wc}(0) \\
&= c_1 \times \frac{n(n+1)}{2} + (c_0 + c_2)n + c_0 \\
&= \frac{c_1}{2} n^2 + (c_0 + \frac{c_1}{2} + c_2)n + c_0 \\
&= O(n^2)
\end{aligned}
$$

퀵 정렬의 최선의 시간복잡도

- 파티션 된 두 배열의 크기가 거의 같을 경우
- T(n) = O(n log n).

$T(n) \le cn + 2T(\frac{n}{2})$ for some constant c

$\le cn + 2(c \cdot \frac{n}{2} + 2T(\frac{n}{2^2})) = 2cn + 2^2 T(\frac{n}{2^2})$

.....

$\le \alpha \cdot cn + 2^\alpha T(\frac{n}{2^\alpha})$ (Let α be the number such that $\frac{n}{2^\alpha} = 1$.)

$= cn \log_2 n + nT(1)$

$= O(n \log n)$

평균 시간복잡도

- $T_{avg}(n) = O(n \log n)$.

$$T_{avg}(n) = \begin{cases} c_0 & \text{if } n = 0 \\ \frac{1}{n}\sum_{j=0}^{n-1}\left(c_1 n + T_{avg}(j) + T_{avg}(n-j-1) + c_2\right) & \text{if } n \geq 1 \end{cases}$$

$$\begin{aligned} T_{avg}(n) &= \frac{1}{n}\sum_{j=0}^{n-1}\left(c_1 n + T_{avg}(j) + T_{avg}(n-j-1) + c_2\right) \\ &= c_1 n + c_2 + \frac{1}{n}\sum_{j=0}^{n-1}\left(T_{avg}(j) + T_{avg}(n-j-1)\right) \\ &= c_1 n + c_2 + \frac{2}{n}\sum_{j=0}^{n-1} T_{avg}(j) \\ &\leq cn + \frac{2}{n}\sum_{j=0}^{n-1} T_{avg}(j), \quad n \geq 2. \end{aligned}$$

귀납법을 이용하여 증명

$$T_{avg}(n) \le cn + \frac{2}{n}\sum_{j=0}^{n-1} T_{avg}(j),\ n \ge 2.$$

We should prove that $T_{avg}(n) \le kn\log_e n$ for some constant k and for $n \ge 2$.

Assume that $T_{avg}(0) \le b$ and $T_{avg}(0) \le b$ for some constant b.

(1) Induction Base : For $n = 2$,

$$T_{avg}(n) \le c \cdot 2 + \frac{2}{2}\sum_{j=0}^{2-1} T_{avg}(j) = 2c + 2b \le k \cdot 2\log_e 2$$

(2) Induction Hypothesis : Assume $T_{avg}(n) \le kn\log_e n$ for $1 \le n < m$.

(3) Induction Step :

$$T_{avg}(m) \le cm + \frac{2}{m}\sum_{j=0}^{m-1} T_{avg}(j) \le cm + \frac{4b}{m} + \frac{2}{m}\sum_{j=2}^{m-1} T_{avg}(j) \le cm + \frac{4b}{m} + \frac{2k}{m}\sum_{j=2}^{m-1} j\log_e j$$

Since $j\log_e j$ is an increasing function of j,

$$T_{avg}(m) \le cm + \frac{4b}{m} + \frac{2k}{m}\int_2^m x\log_e x dx$$

Note that $\int x\log_e x dx = \left(\frac{1}{2}x^2\log x - \frac{1}{4}x^2\right) + C.$

$$T_{avg}(m) \le cm + \frac{4b}{m} + \frac{2k}{m}\left[\frac{m^2\log_e m}{2} - \frac{m^2}{4}\right] = cm + \frac{4b}{m} + km\log_e m - \frac{km}{2} \le km\log_e m$$

최악의 공간복잡도: 최대 호출 깊이=10

R_0	R_1	R_2	R_3	R_4	R_5	R_6	R_7	R_8	R_9	R_{10}	l	r
(99	10	11	22	33	44	55	66	77	88)	+∞	0	9
(88	10	11	22	33	44	55	66	77)	99()		0	8
(77	10	11	22	33	44	55	66)	88()	99()		0	7
(66	10	11	22	33	44	55)	77()	88()	99()		0	6
(55	10	11	22	33	44)	66()	77()	88()	99()		0	5
(44	10	11	22	33)	55()	66()	77()	88()	99()		0	4
(33	10	11	22)	44()	55()	66()	77()	88()	99()		0	3
(22	10	11)	33()	44()	55()	66()	77()	88()	99()		0	2
(11	10)	22()	33()	44()	55()	66()	77()	88()	99()		0	0
(10)	11()	22()	33()	44()	55()	66()	77()	88()	99()		3	6
10	11()	22()	33()	44()	55()	66()	77()	88()	99()		2	1
10	11	22()	33()	44()	55()	66()	77()	88()	99()		3	2
10	11	22	33()	44()	55()	66()	77()	88()	99()		4	3
10	11	22	33	44()	55()	66()	77()	88()	99()		5	4
10	11	22	33	44	55()	66()	77()	88()	99()		6	5
10	11	22	33	44	55	66()	77()	88()	99()		7	6
10	11	22	33	44	55	66	77()	88()	99()		8	7
10	11	22	33	44	55	66	77	88()	99()		9	8
10	11	22	33	44	55	66	77	88	99()		10	9
10	11	22	33	44	55	66	77	88	99			

공간복잡도

- 최악의 경우: $O(n)$
- 최선의 경우: $O(1)$
- 평균적인 경우: $O(\log n)$
- 작은 배열 먼저 처리: $O(\log n)$

요약

- 퀵 정렬은 안정적(stable) 이지 않다

- 피봇을 택하는 다른 방법
  - 셋 중의 중앙값 (median)
    - pivot = 중앙값 { K_l , $K_{(l+r)/2}$, K_r }
  - 예
    - median {10, 5, 7} = 7
    - median { 0, 6, 6} = 6

“정렬” [끝]