# 15. 정렬

# 개요

- 이장의 대부분 정렬 알고리즘은 *비교 정렬(comparison sort)*이다. 이것은 재배열 결정이 배열의 원소 쌍 간의 비교에 기반을 두고 있다는 것을 의미한다.
- 비교 정렬은 $\Theta(n \lg n)$이라는 시간 한계를 넘지 못한다.
- 많은 비교 정렬은 배열에 있는 원소 쌍을 서로 교환하고 있다. 이 연산을 위해 최적화된 swap() 메소드를 사용한다.

# 15.1 버블 정렬

- *버블 정렬(bubble sort)*
  - 배열에서 인접한 원소를 비교한 다음 제 위치에 있지 않을 경우 그것들을 서로 교환한다.
  - 이 알고리즘은 순서에 따라 처음부터 마지막까지 원소 쌍 사이의 비교를 수행하는 것이다.
  - 각각의 비교가 끝나면 큰 원소는 한 칸씩 앞으로 이동하기 때문에 가장 큰 원소는 "거품이 올라오듯이(bubble up)" 배열의 끝으로 이동하게 된다.
  - 하나의 완전한 과정이 끝나면 배열이 오름차순을 유지하는 경우 가장 큰 원소는 배열의 가장 끝으로 이동한다. 이러한 과정을 n-1 개의 정렬되지 않은 원소의 부분배열에 대해 반복해서 수행하면 두 번째로 큰 원소가 제 자리로 이동한다.

- LISTING 15.1: The Bubble Sort

```
1 void sort(int[] a) {
2     for (int i = a.length-1; i > 0; i--)
3         for (int j = 0; j < i; j++)
4             if (a[j] > a[j+1]) swap(a, j, j+1);
5 }
```

실행시간 ; $\Theta(n^2)$

버블 정렬을 수행하라

Array = {66, 33, 99, 88, 44, 55, 22, 77}

# 15.2 선택 정렬

- *선택 정렬(selection sort)*
  - n 원소 시이퀀스에 대해 n-1 패스를 수행하는데, 매번 나머지 정렬되지 않은 원소들 중에서 가장 큰 원소를 올바른 위치로 이동시킨다.
  - 그러나 이것은 각 패스에서 교환을 한 번만 수행하기 때문에 버블 정렬에 비해 약간 더 효율적이다.
  - 이것을 "선택(selection)" 정렬이라고 부르는 이유는 각 패스마다 정렬되지 않은 원소들 중에서 가장 큰 원소를 선택하여 그것을 올바른 위치로 이동시키기 때문이다.

- 선택 정렬 알고리즘

  1. i=n-1에서 1까지 내려가면서 단계 2를 반복.
  2. $a_i$와 max{$a_0$, …., $a_i$}를 교환.

# 선택 정렬 리스팅

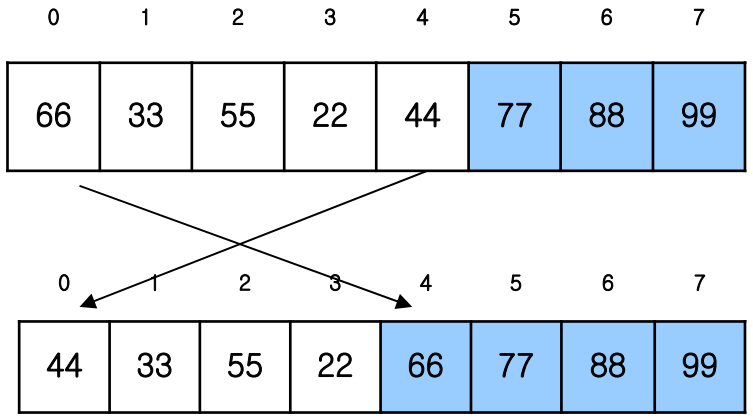
- LISTING 15.4: The Selection Sort

```
1 void sort(int[] a) {
2     for (int i = a.length-1; i > 0; i--) {
3         int m=0;
4         for (int j = 1; j <= i; j++)
5             if (a[j] > a[m]) m = j;
6         swap(a, i, m);
7     }
8 }
```

실행시간 ; $\Theta(n^2)$

# 선택 정렬의 수행 과정

Array ={66,33,99,88,44,55,22,77}




선택 정렬에서 네 번째 패스

| | a[0] | a[1] | a[2] | a[3] | a[4] | a[5] | a[6] | a[7] |
|---|---|---|---|---|---|---|---|---|
| 0 | 66 | 33 | 99 | 88 | 44 | 55 | 22 | 77 |
| 1 | | | 77 | | | | | 99 |
| 2 | | | | 22 | | | 88 | |
| 3 | | | 55 | | | 77 | | |
| 4 | 44 | | | | 66 | | | |
| 5 | | | 22 | 55 | | | | |
| 6 | 22 | | 44 | | | | | |
| 7 | 22 | 33 | | | | | | |

# 15.3 삽입 정렬

- *삽입 정렬(insertion sort)*
  - n 원소 시이퀀스에 대해 n-1 패스를 수행한다.
  - 각각의 패스에서 이것은 왼쪽에 있는 부분배열을 정렬된 상태로 두면서 다음 원소를 이 부분배열에 삽입한다.
  - 마지막 원소가 이러한 방법으로 "삽입되면(inserted)" 전체 배열이 정렬된 것이다.

- 삽입 정렬 알고리즘

  1. i=1에서 n-1까지 올라가면서 단계 2-5를 반복.
  2. 원소 $a_i$를 임시 기억장소에 저장.
  3. $a_j \geq a_i$에 대해 $j \leq i$인 최대 인덱스에 위치함.
  4. 서브시이퀀스 $\{a_j,\ldots,a_{i-1}\}$을 $\{a_{j+1},\ldots,a_i\}$가 되도록 위로 하나씩 이동.
  5. $a_i$의 저장된 값을 $a_j$로 복사.

# 삽입 정렬 리스팅

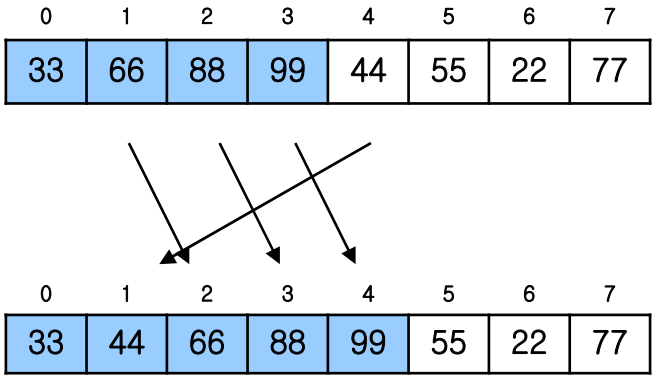
- LISTING 15.5: The Insertion Sort

```
1 void sort(int[] a) {
2     for (int i = 1; i < a.length; i++) {
3         int ai = a[i], j = i;
4         for (j = i; j > 0 && a[j-1] > ai; j--)
5                 a[j] = a[j-1];
6         a[j] = ai;
7     }
8 }
```

실행시간 ; $\Theta(n^2)$

# 삽입 정렬의 수행 과정

**Array = {66,33,99,88,44,55,22,77}**




삽입 정렬에서 네 번째 패스

| | a[0] | a[1] | a[2] | a[3] | a[4] | a[5] | a[6] | a[7] |
|---|---|---|---|---|---|---|---|---|
| 0 | 66 | 33 | 99 | 88 | 44 | 55 | 22 | 77 |
| 1 | 33 | 66 | | | | | | |
| 2 | | | 88 | 99 | | | | |
| 3 | | 44 | 66 | 88 | 99 | | | |
| 4 | | | 55 | 66 | 88 | 99 | | |
| 5 | 22 | 33 | 44 | 55 | 66 | 88 | 99 | |
| 6 | | | | | | 77 | 88 | 99 |

# stability

- 정렬 알고리즘이 stable하다
- 같은 값을 가지고 있는 두 원소가 정렬알고리즘을 수행한 후에 그 위치를 바꾸지 않을 때 정렬알고리즘이 stable하다고 한다
- 어떤 알고리즘이 이에 해당하는가?
  - Bubble sort
  - selection sort
  - insertion sort

# 15.5 비교 정렬의 속도 한계

- 공식 15.5 : 비교 정렬의 속도 한계
  – 어떤 비교 정렬도 최악의 경우 실행 시간이 $\Theta(n\lg n)$보다 좋을 수 없다.

- 분할정복법을 이용하는 정렬 알고리즘,
  합병정렬 (Merge sort)과 퀵정렬(Quick sort)는
  평균 실행 시간이 $\Theta(n\lg n)$

- 분할정복법
  – 분할: 크기가 작은 문제들로 나눈다
  – 정복: 크기가 작은 문제를 해결한다
  – 합병: 크기가 작은 문제의 답을 이용하여 큰 문제를 해결한다

# 15.6 합병 정렬

- 합병 정렬 알고리즘

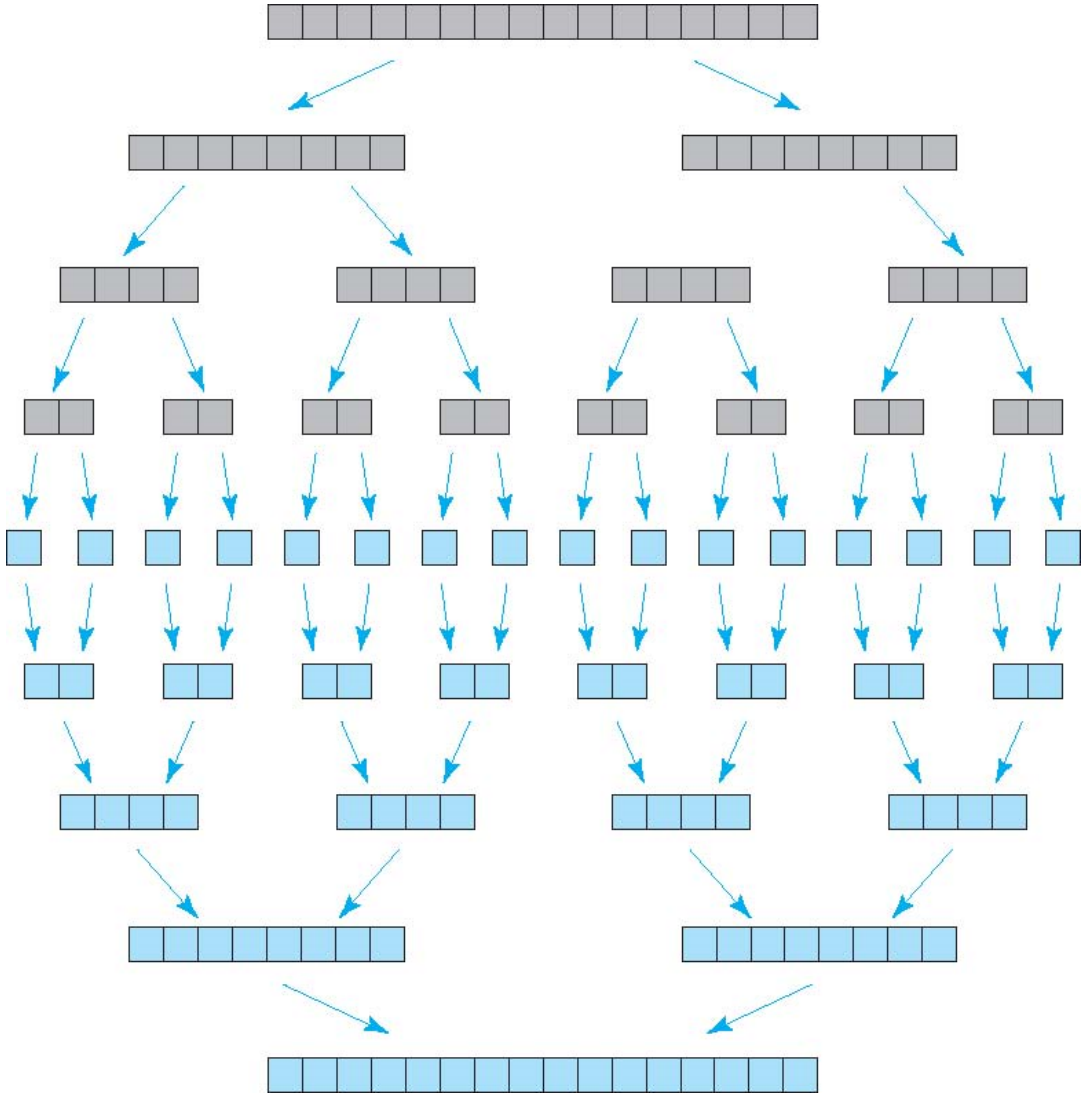
1. 만일 시이퀀스가 두 개의 원소보다 적으면, 리턴.
2. $a_m$을 시이퀀스의 중간 원소로 설정.
3. $a_m$보다 이전에 있는 원소들의 서브시이퀀스를 정렬.
4. 다른 모든 원소들의 서브시이퀀스를 정렬.
5. 두 개의 정렬된 서브시이퀀스를 합병.

# 합병 정렬 리스팅

- LISTING 15.8: The Merge Sort

```
1 void sort(int[] a, int p, int q) {
2         // PRECONDITION: 0 < p < q <= a.length
3         // POSTCONDITION: a[p...q-1] is in ascending order
4     if (q-p < 2) return;
5     int m = (p+q)/2;
6     sort(a, p, m);
7     sort(a, m, q);
8     merge(a, p, m, q);
9 }
```

# 합병 정렬



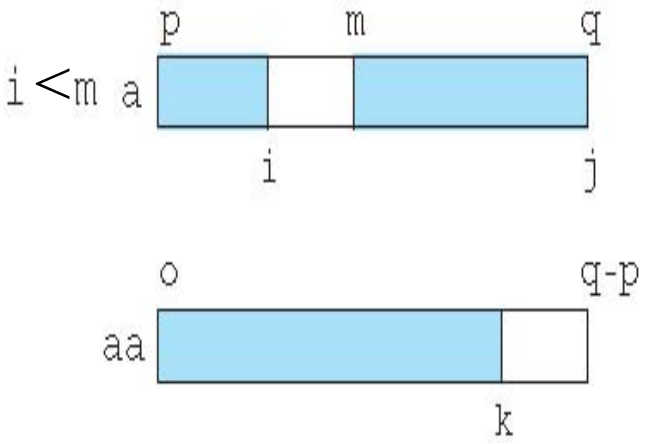
$\Theta(n\lg n)$

# merge() 메소드

```
1 void merge(int[] a, int p, int m, int q) {
2       // PRECONDITIONS: a[p...m-1] and a[m...q-1] are in
                    ascending order;
3       // POSTCONDITION: a[p...q-1] is in ascending order;
4      if (a[m-1] <= a[m]) return; // a[p...q-1] is already sorted
5      int i = p, j = m, k = 0;
6      int[] aa = new int[q-p];
7      while (i < m && j < q)
8          if (a[i]<a[j]) aa[k++] = a[i++];
9          else aa[k++] = a[j++];
10     if (i < m)  System.arraycopy(a, i, a, p+k, m-i);
                                          // shift a[i...m-1]
11     System.arraycopy(aa, 0, a, p, k);
                                      // copy aa[0...k-1] to a[p...p+k-1];
12 }
```

# 합병 정렬의 수행 과정

**Array = {66,33,99,**
**55,88,22,44,77}**




| | a[0] | a[1] | a[2] | a[3] | a[4] | a[5] | a[6] | a[7] |
|---|---|---|---|---|---|---|---|---|
| 0 | 66 | 33 | 99 | 55 | 88 | 22 | 44 | 77 |
| 1 | 33 | 66 | | | | | | |
| 2 | | | 55 | 99 | | | | |
| 3 | 33 | 55 | 66 | 99 | | | | |
| 4 | | | | | 22 | 88 | | |
| 5 | | | | | 22 | 44 | 77 | 88 |
| 6 | 22 | 33 | 44 | 55 | 66 | 77 | 88 | 99 |

# 15.7 퀵 정렬

- *퀵 정렬(Quick Sort)*
  - 평균적으로 $\Theta(n\lg n)$ 시간에 실행되는 또 하나의 순환 분할-정복 알고리즘이다.
  - 1962년 Anthony Hoare에 의해 발견되었다.
  - 두 단계로 이루어져 있다.
  - *분할(partition)*이라고 부르는 첫 번째 단계에서는 시이퀀스를 하나의 원소 $a_k$에 의해 분리되는 두 개의 세그먼트로 나눈다. 따라서 모든 $i<k$에 대해 $a_i \le a_k$이고, 모든 $j>k$에 대해 $a_j \ge a_k$이다. 원소 $a_k$를 분할을 위한 피봇 (pivot)이라고 부른다.
  - 두 번째 단계에서는 두 세그먼트 각각을 순환적으로 정렬한다.

# 퀵 정렬 알고리즘

1. 만일 시이퀀스가 두 개의 원소보다 적으면, 리턴.
2. 피봇으로 삼은 첫 번째 원소 $a_p$를 사용하여, 시이퀀스 $\{a_p, \ldots, a_{q-1}\}$를 분할하는데, 서브시이퀀스 X에 있는 모든 원소는 $a_p$보다 작거나 같고, 서브시이퀀스 Z에 있는 모든 원소는 $a_p$보다 크거나 같고, Y는 단독 서브시이퀀스 $Y=\{a_p\}$가 되도록 세 개의 서브시이퀀스 {X, Y, Z}로 분할한다.
3. 서브시이퀀스 X를 정렬한다.
4. 서브시이퀀스 Z를 정렬한다.

   – 이 알고리즘은 순환적이기 때문에, 크기 q-p의 서브시이퀀스 $a_{p\ldots q-1}$에 적용되는 것을 가정하고 있다.

# 퀵 정렬 리스팅

- LISTING 15.10: The Quick Sort

```
1 void sort(int[] a, int p, int q) {
2     // PRECONDITION: 0<= p < q <= a.length
3     // POSTCONDITION: a[p,...,q-1] is in ascending order
4      if (q-p < 2) return;
5      int j = partition(a, p, q);     //j는 pivot의 위치
6      sort(a, p, j);
7      sort(a, j+1, q);
8 }
```

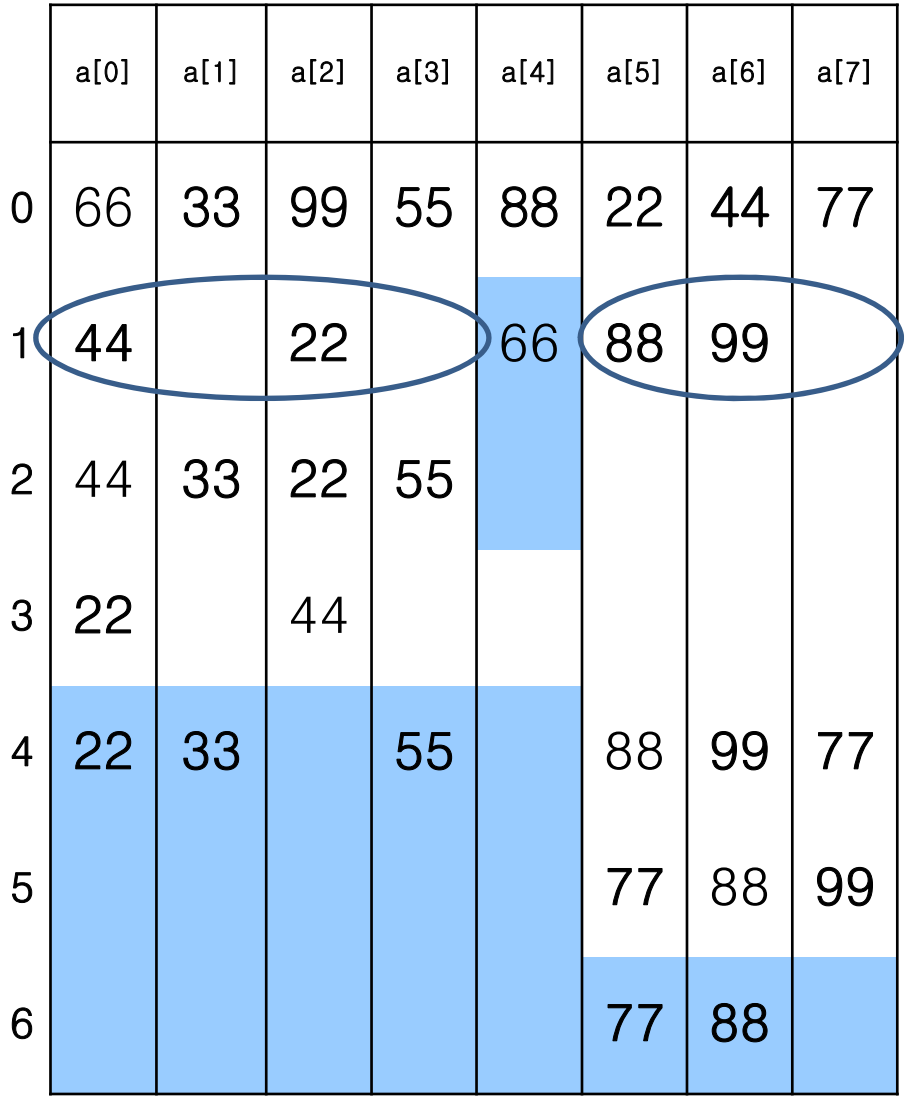
# LISTING 15.11: partition() 메소드

```
  int partition(int[] a, int p, int q) {
// RETURNS: index j of pivot element a[j];
// POSTCONDITION: a[i] <= a[j] <= a[k] for p<= i <= j <= k < q;
4      int pivot=a[p], i = p, j = q;
5      while (i < j) {
6          while (j > i && a[--j] >= pivot);        // empty loop
8          if (j > i) a[i] = a[j];
9          while (i < j && a[++i] <= pivot);        // empty loop
11         if (i < j) a[j] = a[i];
12     }
13     a[j] = pivot;
14     return j;
15 }
```

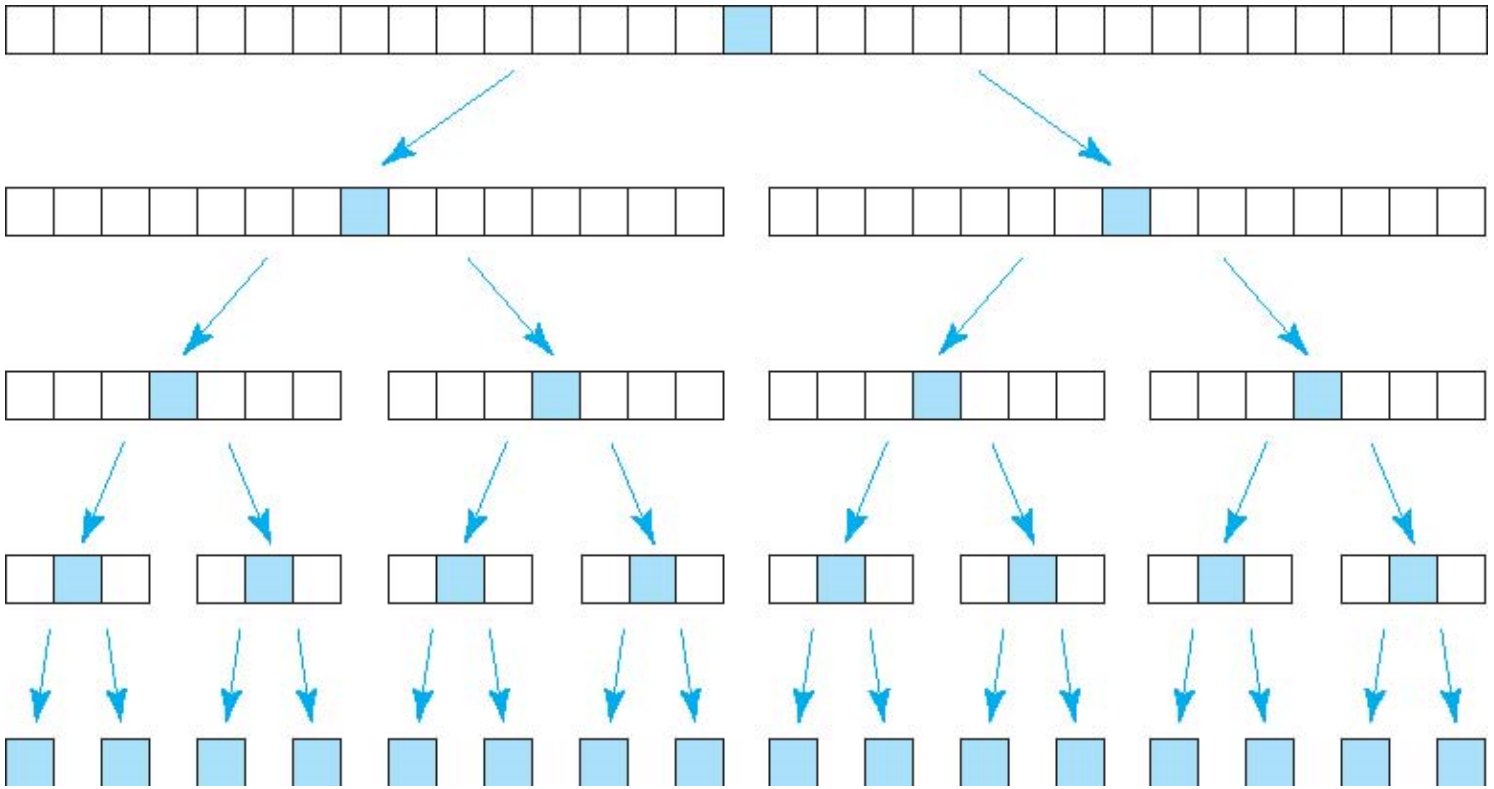
# 퀵 정렬의 수행 과정

**Array = {66,33,99,55,88,22,44,77}**

퀵 정렬에서 첫 번째 분할 →



# 최선의 경우 퀵 정렬



평균적인 경우, 최선의 경우: $\Theta(n\lg n)$

최악의 경우: $\Theta(n^2)$