정렬

정렬

- 정렬(sorting)
  - 자료 모음에 순서를 부여하는 작업
  - 예
    - 9, 1, 3, 7, 3, 4, 2, 1, 5 → 1, 1, 2, 3, 3, 4, 5, 7, 9

정렬 알고리즘

- 정렬 알고리즘
  - Comparison vs. non-comparison
    - 정렬 과정에서 자료의 순서(대소) 비교를 사용하는 여부
  - In-place
    - 입력 자료가 저장된 공간 내에서 교환 등을 통해 정렬 수행
  - Stable
    - 동일 키 자료들의 정렬 전 순서가 정렬 후에도 유지
      - **김**철수, **이**철수, **김**동수 → 성씨 정렬 → 김철수, 김동수, 이철수

정렬 알고리즘

- 정렬 알고리즘
  - 비교 기반 정렬 (comparison sorting)
    - 거품 정렬 (bubble sort)
    - 선택 정렬 (selection sort)
    - 삽입 정렬 (insertion sort)
    - 퀵 정렬 (quick sort)
    - 합병 정렬 (merge sort)
    - 힙 정렬 (heap sort)
  - 정수 정렬 (integer sorting)
    - 계수 정렬 (counting sort)
    - 기수 정렬 (radix sort)

정렬 알고리즘 시간복잡도

- 정렬 알고리즘 시간복잡도
  - n → 자료 크기, r → 수의 범위, k → 키 크기, d → digit 크기

분류	알고리즘	Best	Average	Worst
비교 정렬 $\Omega(n \log_2 n)$	거품정렬	n	n^2	n^2
	선택정렬	n^2	n^2	n^2
	삽입정렬	n	n^2	n^2
	퀵정렬	$n \log n$	$n \log n$	n^2
	합병정렬	$n \log n$	$n \log n$	$n \log n$
	힙정렬	$n \log n$	$n \log n$	$n \log n$
정수 정렬	계수정렬		$n + r$	$n + r$
	기수정렬		$n\frac{k}{d}$	$n\frac{k}{d}$

거품정렬

https://en.wikipedia.org/wiki/Bubble_sort, CC-BY-SA

					오름차순 정렬 가정
5	4	3	2	1	최초 상태
5	4	3	2	1	5,4 비교 후 교환
4	5	3	2	1	5,3 비교
4	3	5	2	1	5,2 비교 후 교환
4	3	2	5	1	5,1 비교 후 교환
4	3	2	1	**5**	n=5인 경우, 총 4번 비교 후 최대값(5) 위치 결정
3	2	1	**4**	**5**	총 3회 비교 후 두번째 큰 값(4) 위치 결정
2	1	**3**	**4**	**5**	총 2회 비교 후 세번째 큰 값(3) 위치 결정
1	**2**	**3**	**4**	**5**	총 1회 비교 후 네번째 큰 값(2) 위치 결정

총 비교 횟수(n=5): 4+3+2+1
총 비교 횟수(n=1000): 999+998+ … +2+1

$$1 + 2 + \cdots (n-1) = \frac{n(n-1)}{2} = O(n^2)$$

거품정렬


```
public class Test {
	public static void main(String[] args) {
		int	v[]=new int[]{5,8,1,9,3,5,1,5};
		bubbleSort(v);
		System.out.println(Arrays.toString(v));
	}
	private static void bubbleSort(int[] v) {
		for (int i = 0; i < v.length; i++) {
			for (int j = 1; j < v.length; j++) {
				if(v[j-1]>v[j]) {
					int	temp=v[j];
					v[j]=v[j-1];
					v[j-1]=temp;
				}
			}
		}
	}
}
```

실습: k번째 반복인 경우 이미 k개 값들의 최종 위치가 결정되었으므로 매번 전체 자료에 대한 검사는 비효율적

실습: 특정 내부 루프 수행 시 교환이 전혀 발생하지 않았다면 자료는 이미 정렬된 상태임

선택정렬

					오름차순 정렬 가정
5	4	3	2	1	최초 상태, 비정렬 목록 [5 4 3 2 1]
5	4	3	2	**1**	비정렬 목록에서 최소값 1 결정
1	4	3	2	5	비정렬 목록의 첫 위치 자료 5와 교환, 비정렬 목록 [4 3 2 5]
1	4	3	**2**	5	비정렬 목록에서 최소값 2 결정
1	**2**	3	4	5	비정렬 목록의 첫 위치 자료 4와 교환, 비정렬 목록 [3 4 5]
1	**2**	**3**	4	5	비정렬 목록에서 최소값 3 결정
1	**2**	**3**	4	5	비정렬 목록의 첫 위치 자료 3과 교환, 비정렬 목록 [4 5]
1	**2**	**3**	**4**	5	비정렬 목록에서 최소값 4 결정
1	**2**	**3**	**4**	5	비정렬 목록의 첫 위치 자료 4와 교환, 비정렬 목록 [5]

$$1 + 2 + \cdots (n-1) = \frac{n(n-1)}{2} = O(n^2)$$

선택정렬


```
public class Test {
    public static void main(String[] args) {
        int    v[]=new int[]{5,8,1,9,3,5,1,5};
        selectionSort(v);
        System.out.println(Arrays.toString(v));
    }
    private static void selectionSort(int[] v) {
        for (int i = 0; i < v.length-1; i++) {
            int    minPos=i;
            for (int j = i+1; j < v.length; j++) {
                if(v[minPos]>v[j]) minPos=j;
            }
            int    x=v[i];
            v[i]=v[minPos];
            v[minPos]=x;
        }
    }
}
```

i++ → 매 단계 후 비정렬 목록 크기 1 감소

비정렬 목록 전체를 검사하여 최소값 위치 결정

비정렬 목록의 첫 위치 자료와 최소값 교환

실습: 첫 위치 자료가 최소값인 경우 교환 작업 불필요

삽입정렬

					오름차순 정렬 가정
5	4	3	2	1	최초 상태, 정렬 목록 [5], 비정렬 목록 [4 3 2 1]
5	**4**	3	2	1	비정렬 목록의 첫 자료 4를 정렬 목록 내에 삽입
4	**5**	3	2	1	삽입 후 비정렬 목록 [3 2 1]
4	**5**	**3**	2	1	비정렬 목록의 첫 자료 3을 정렬 목록 내에 삽입
3	**4**	**5**	2	1	삽입 후 비정렬 목록 [2 1]
3	**4**	**5**	**2**	1	비정렬 목록의 첫 자료 2를 정렬 목록 내에 삽입
2	**3**	**4**	**5**	1	삽입 후 비정렬 목록 [1]
2	**3**	**4**	**5**	**1**	비정렬 목록의 첫 자료 1을 정렬 목록 내에 삽입
1	**2**	**3**	**4**	**5**	삽입 후 비정렬 목록 []

$$1 + 2 + \cdots (n-1) = \frac{n(n-1)}{2} = O(n^2)$$

삽입정렬


```
public class Test {
	public static void main(String[] args) {
		int	v[]=new int[]{5,8,1,9,3,5,1,5};
		insertionSort(v);
		System.out.println(Arrays.toString(v));
	}
	private static void insertionSort(int[] v) {
		for (int i = 1; i < v.length; i++) {
			for (int j = i; j>0 && v[j-1]>v[j]; j--) {
				int	x=v[j];
				v[j]=v[j-1];
				v[j-1]=x;
			}
		}
	}
}
```

실습: 3회 대입 연산 필요
보다 효율적으로 개선해 보시오

선택정렬 vs. 삽입정렬

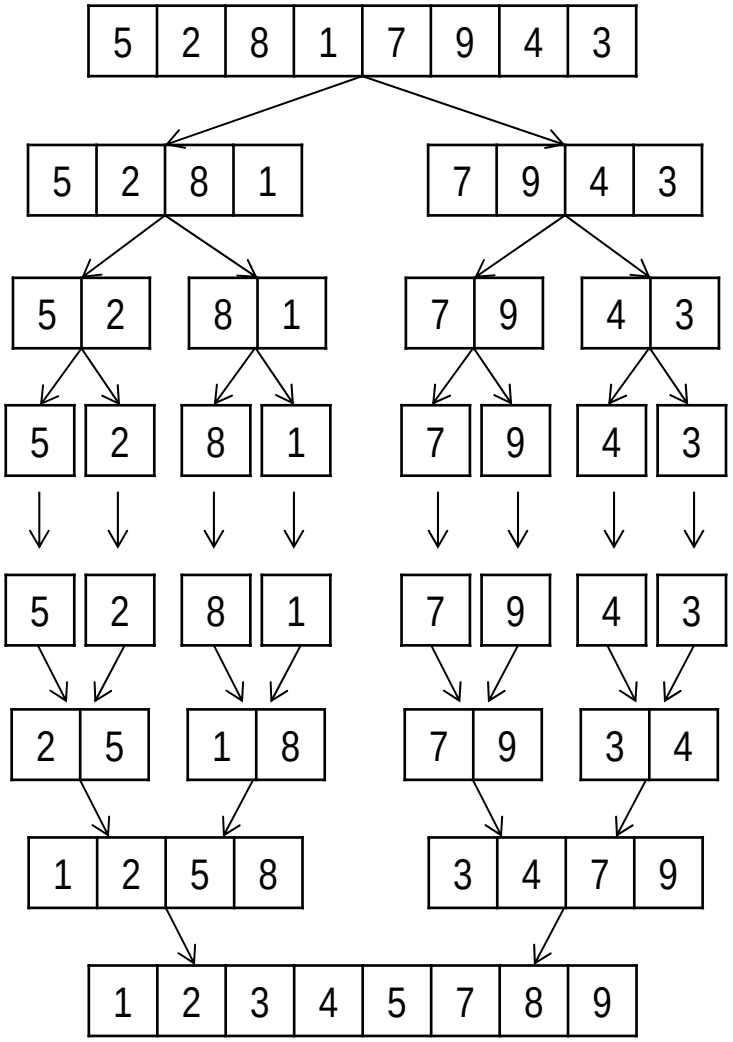
					정렬된 자료 모음에 대한 선택정렬
1	2	3	4	5	초기 상태, 비정렬 목록 [1 2 3 4 5]
1	2	3	4	5	최소값 1 탐색 위해 비정렬 목록 전체 [1 2 3 4 5] 검사
1	**2**	3	4	5	최소값 2 탐색 위해 비정렬 목록 전체 [2 3 4 5] 검사
1	**2**	**3**	4	5	최소값 3 탐색 위해 비정렬 목록 전체 [3 4 5] 검사
1	**2**	**3**	**4**	5	최소값 4 탐색 위해 비정렬 목록 전체 [4 5] 검사

					정렬된 자료 모음에 대한 삽입정렬
1	2	3	4	5	초기 상태, 정렬 목록 [1], 비정렬 목록 [2 3 4 5]
1	2	3	4	5	정렬된 자료의 경우 자료 2 삽입 시 정렬 목록 내 위치 이동 불필요
1	**2**	3	4	5	정렬된 자료의 경우 자료 3 삽입 시 정렬 목록 내 위치 이동 불필요
1	**2**	**3**	4	5	정렬된 자료의 경우 자료 4 삽입 시 정렬 목록 내 위치 이동 불필요
1	**2**	**3**	**4**	5	정렬된 자료의 경우 자료 5 삽입 시 정렬 목록 내 위치 이동 불필요

선택정렬 vs. 삽입정렬

- 선택정렬
  - 매 단계마다 최소값 탐색 위해 **비정렬 목록 전체 검사**(read) 필요
  - 매 단계 최소값의 위치는 1회 교환(write)으로 가능
    - Write 시간이 read보다 월등히 큰 경우 선택정렬이 삽입정렬보다 유리
- 삽입정렬
  - 정렬된 목록 내 임의 자료의 삽입 위치 결정 위해 **평균적으로 정렬 목록의 절반 탐색**(read) 및 자리 이동(write) 필요
    - 일반적으로 삽입정렬이 선택정렬보다 효율적
  - 거의 정렬된 자료 모음의 경우 삽입 시간 최소화 가능

합병정렬




```
public class Test {
	public static void main(String[] args) {
		int	v[]=new int[]{5,2,8,1,7,9,4,3};
		mergeSort(v, 0, v.length-1);
		System.out.println(Arrays.toString(v));
	}
	private static void mergeSort(int[] v, int left, int right) {
		if(left==right) return;
		int	m=(left+right)/2;
		mergeSort(v, left, m);
		mergeSort(v, m+1, right);
		mergeArray(v, left, m, m+1, right);
	}
	private static void mergeArray(int[] v, int l1, int r1, int l2, int r2) {
	}
}
```

합병정렬

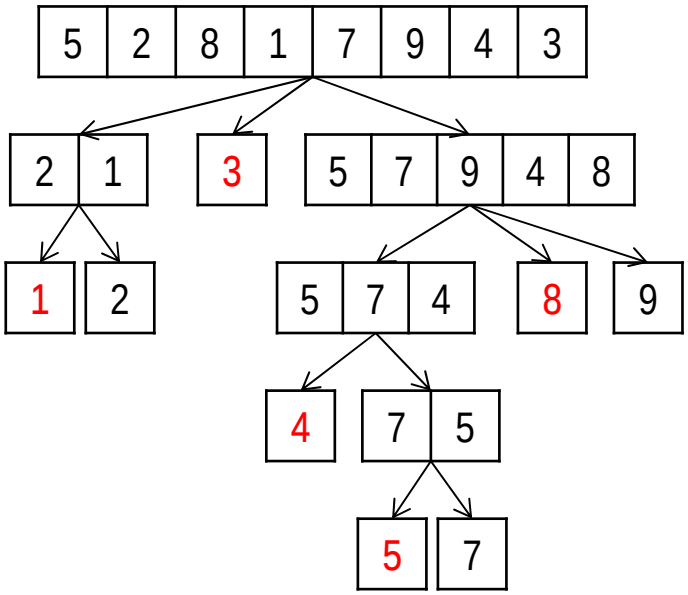
								오름차순 정렬 가정
5	2	8	1	7	9	4	3	mergeSort(v,0,7)
5	2	8	1					mergeSort(v,0,3)
5	2							mergeSort(v,0,1)
5								mergeSort(v,0,0) & return
	2							mergeSort(v,1,1) & return
2	5							mergeArray(v,0,0,1,1)
		8	1					mergeSort(v,2,3)
		8						mergeSort(v,2,2) & return
			1					mergeSort(v,3,3) & return
		1	8					mergeArray(v,2,2,3,3)
1	2	5	8					mergeArray(v,0,1,2,3)
				3	4	7	9	mergeSort(v,4,7)
1	2	3	4	5	7	8	9	mergeArray(v,0,3,4,7)

퀵정렬: Lomuto's partition

5	2	8	1	7	9	4	**3**	Pivot=3, LT=[], GTEQ=[5 2 8 1 7 9 4], GTEQ 첫 위치 p=0
5	2	8	1	7	9	4	**3**	5>Pivot
5	**2**	8	1	7	9	4	**3**	2<Pivot → 2를 GTEQ에서 LT로 이동 → GTEQ 첫 위치 p=1
2	5	8	1	7	9	4	**3**	LT=[2], GTEQ=[5 8 1 7 9 4]
2	5	**8**	1	7	9	4	**3**	8>Pivot
2	5	8	**1**	7	9	4	**3**	1<Pivot → 1을 GTEQ에서 LT로 이동 → GTEQ 첫 위치 p=2
2	**1**	8	5	7	9	4	**3**	LT=[2 1], GTEQ=[8 5 7 9 4]
2	**1**	8	5	**7**	9	4	**3**	7>Pivot
2	**1**	8	5	7	**9**	4	**3**	9>Pivot
2	**1**	8	5	7	9	**4**	**3**	4>Pivot
2	**1**	**3**	5	7	9	4	8	LT=[2 1], Pivot, GTEQ=[5 7 9 4 8]

- GTEQ → Pivot 값보다 크거나 같은 값들의 리스트
- LT → Pivot 보다 작은 값들의 리스트, 최초 LT=[]
- 최초 분할 대상 배열 전체를 GTEQ로 가정하고 배열 내 각 원소를 검사하여 Pivot보다 적은 값이 발견되면 LT로 이동 (이동 방법: 해당 값을 GTEQ의 첫 원소와 교환하고 GTEQ의 첫 위치를 오른쪽으로 하나 이동)
- GTEQ 검사 완료 후 GTEQ 첫 원소와 피봇 교환
- Lomuto 분할 이후 LT <= Pivot <= GTEQ

퀵정렬: Lomuto's partition 기반



GTEQ 리스트 첫 위치 (반복 종료 후 pivot 위치임)

GTEQ 불만족 자료 LT로 이동, GTEQ 첫 위치 수정

실습: p와 k가 다를 경우만 swap하도록 개선

GTEQ 리스트 첫 자료와 pivot 교환

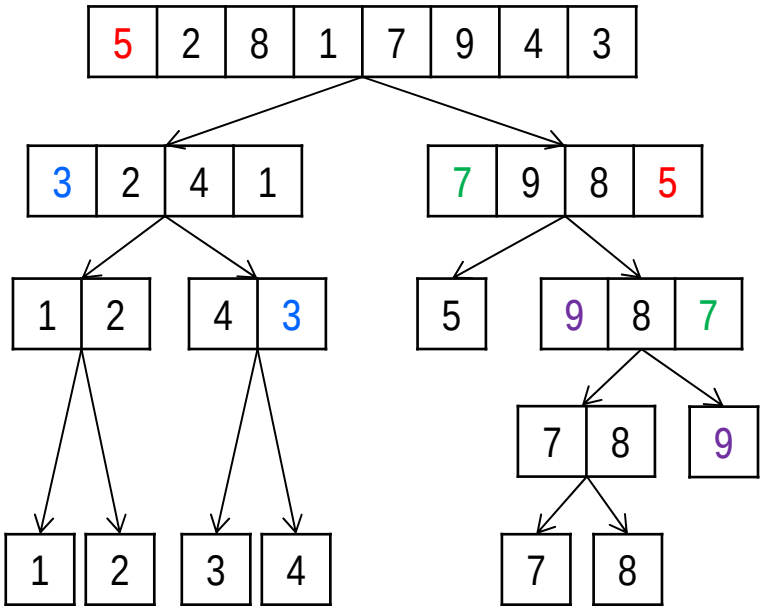
```
public class Test {
    public static void main(String[] args) {
        int    v[]={8,5,9,1,5,3,5,1};
        quickSort(v, 0, v.length-1);
        System.out.println(Arrays.toString(v));
    }
    private static void quickSort(int[] v, int low, int high) {
        if(low>=high) return;
        int    p=partition(v, low, high);
        quickSort(v, low, p-1);
        quickSort(v, p+1, high);
    }
    private static int partition(int[] v, int low, int high) {
        int    pivot=v[high];
        int    p=low;
        for (int k = low; k < high; k++) {
            if(v[k]<pivot) swap(v, p++, k);
        }
        swap(v, p, high);
        return p;
    }
    private static void swap(int[] v, int i, int j) { … }
}
```

퀵정렬: Hoare's partition 기반

0	1	2	3	4	5	6	7	
5	2	8	1	7	9	4	3	Pivot=5
5 →	2	8	1	7	9	4	3 ←	5≥Pivot, 3≤Pivot → i=0, j=7, i<j → 5, 3 교환
3	2	8 →	1	7	9	4 ←	**5**	8≥Pivot, 4≤Pivot → i=2, j=6, i<j → 8, 4 교환
3	2	4	1 ←	7 →	9	8	**5**	7≥Pivot, 1≤Pivot → i=4, j=3, j≤i → j 반환 → [3 2 4 1], [7 9 8 5]의 두 리스트로 분할 → 반환되는 j 기준으로 [3 2 4], 1, [7 9 8 5]로 분할 불가
3	2	4	1					Pivot=3
3 →	2	4	1 ←					3≥Pivot, 1≤Pivot → i=0, j=3, i<j → 3, 1 교환
1	2 ←	4 →	**3**					4≥Pivot, 2≤Pivot → i=2, j=1, j≤i → j 반환 → [1 2], [4 3]의 두 리스트로 분할
1	2							Pivot=1
1 ← →	2							1≥Pivot, 1≤Pivot → i=0, j=0, j≤i → j 반환 → [1], [2]의 두 리스트로 분할

퀵정렬: Hoare's partition 기반

https://en.wikipedia.org/wiki/Quicksort, CC-BY-SA



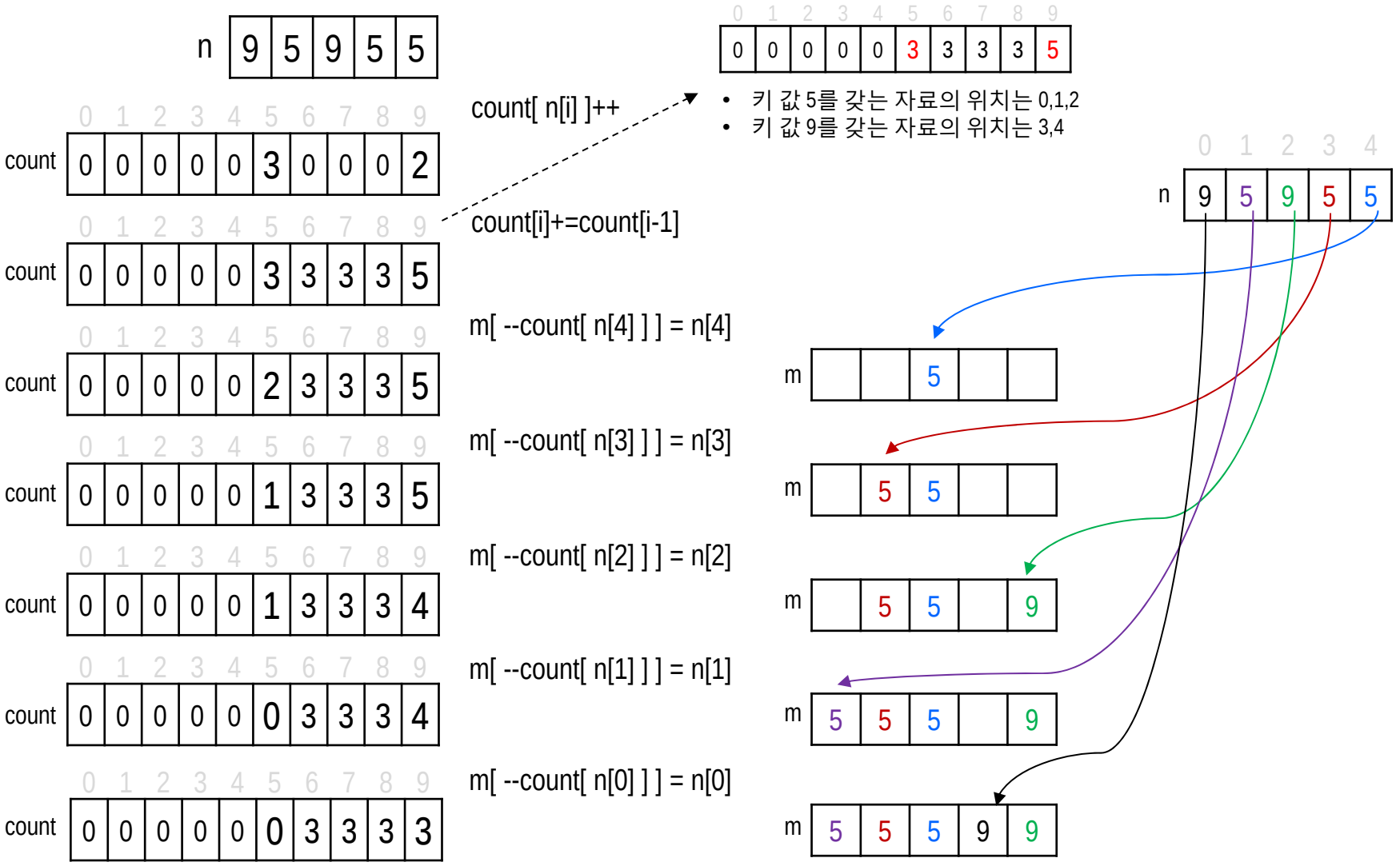

```
public class Test {
    public static void main(String[] args) {
        int    v[]={8,5,9,1,5,3,5,1};
        quickSort(v, 0, v.length-1);
        System.out.println(Arrays.toString(v));
    }
    private static void quickSort(int[] v, int low, int high) {
        if(low>=high) return;
        int    p=partition(v, low, high);
        quickSort(v, low, p);
        quickSort(v, p+1, high);
    }
    private static int partition(int[] v, int low, int high) {
        int    i=low-1, j=high+1, pivot=v[low];
        while(true){
            while(v[++i]<pivot);
            while(v[--j]>pivot);
            if(i<j) swap(v, i, j);
            else return j;
        }
    }
    private static void swap(int[] v, int k, int i) { … }
}
```

v[i] ≥ pivot이면 stop → `while(v[++i]<pivot);`

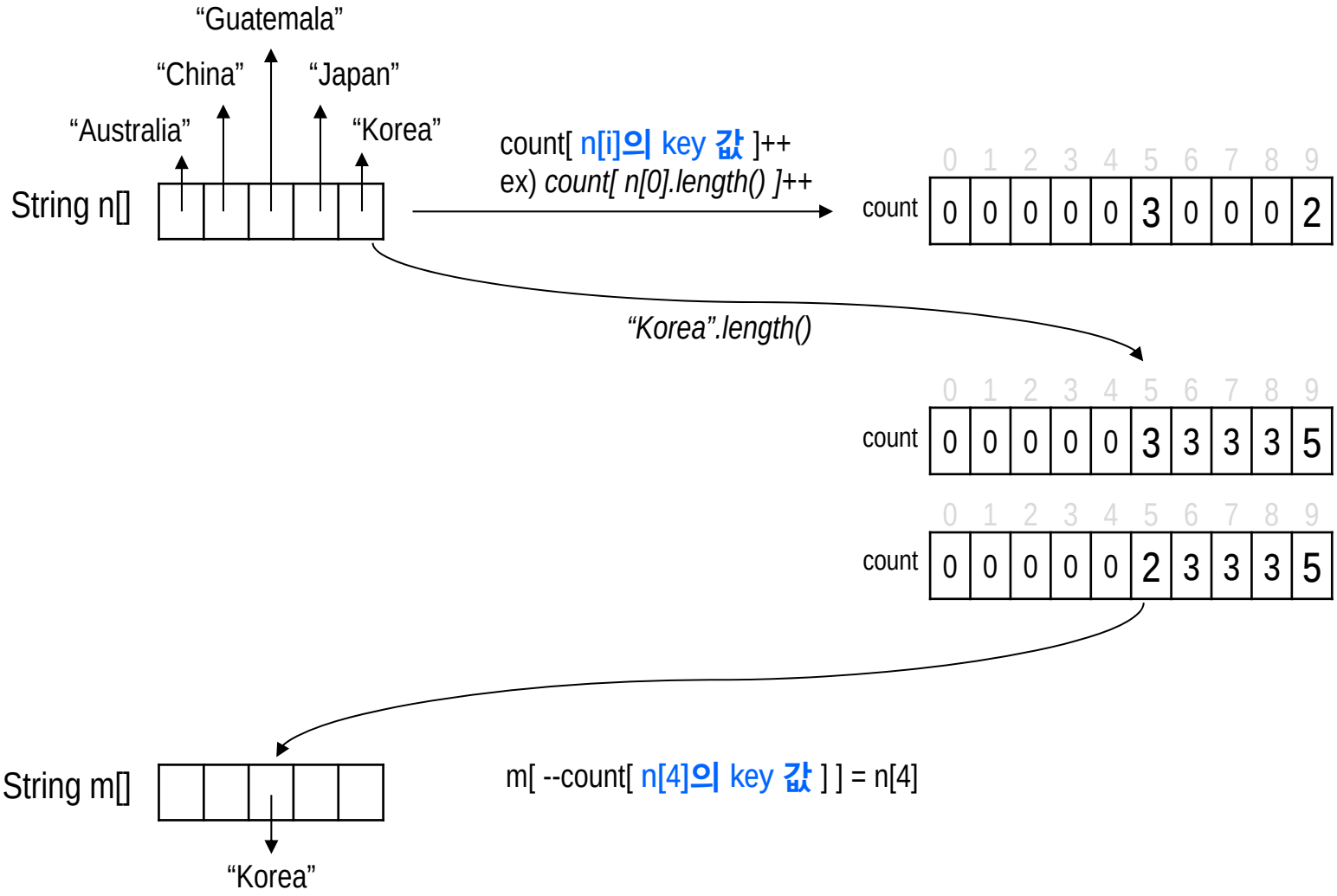
v[j] ≤ pivot이면 stop → `while(v[--j]>pivot);`

j<=i → j: right end of LTEQ list, i: left end of GTEQ list → `if(i<j) swap(v, i, j); else return j;`

계수정렬 (counting sort)



계수정렬 (counting sort)



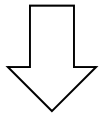
계수정렬 (counting sort)


```
public class Test {
	public static void main(String[] args) {
		int   n[] = {5,8,1,9,3,5,1,5};
		countingSort(n, 10); // 최대값 10
		System.out.println(Arrays.toString(n));
	}
	private static void countingSort(int[] n, int max) {
		int   m[]=new int[n.length], count[]=new int[max+1];
		for (int i = 0; i < n.length; i++) count[n[i]]++;
		for (int i = 1; i < count.length; i++) count[i]+=count[i-1];
		for (int i = n.length-1; i >=0; i--) m[--count[n[i]]]=n[i];
		for (int i = 0; i < m.length; i++) n[i]=m[i];
	}
}
```

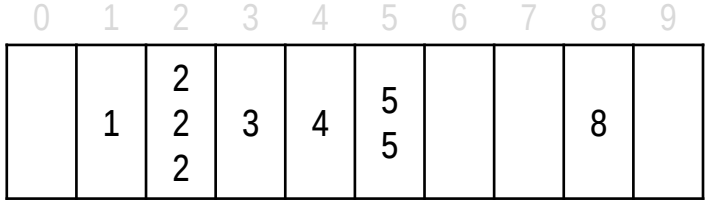
기수정렬 (Radix Sort)

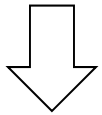
한자리 십진수 기수정렬

5, 8, 2, 1, 5, 2, 4, 2, 3



각 수를 대응하는 큐(큐0, 큐1, 큐2, ..., 큐9)에 삽입
- 숫자 0은 큐 0에 삽입, 숫자 1은 큐 1에 삽입, ..., 숫자 9는 큐 9에 삽입





큐 0부터 큐 9의 순서로
큐 내의 자료를 추출

1, 2, 2, 2, 3, 4, 5, 5, 8

```
public class Test {
	public static void main(String[] args) {
		int	n[]={5, 8, 2, 1, 5, 2, 4, 2, 3};
		LinkedList<Integer>		queue[]=new LinkedList[10];
		for (int q = 0; q < queue.length; q++){
			queue[q]=new LinkedList<>();
		}
		for (int i = 0; i < n.length; i++) queue[n[i]].add(n[i]);
		for (int q = 0, i=0; q < queue.length; q++){
			while(!queue[q].isEmpty()){
				n[i++]=queue[q].remove();
			}
		}
		System.out.println(Arrays.toString(n));
	}
}
```

기수정렬 (Radix Sort)

세자리 십진수 기수정렬

004, 423, 312, 034, 043, 002, 223, 042

1의 자리 값으로 정렬

재정렬

0	1	2	3	4	5	6	7	8	9
		312 002 042	423 043 223	004 034					

312, 002, 042, 423, 043, 223, 004, 034

10의 자리 값으로 정렬

재정렬

0	1	2	3	4	5	6	7	8	9
002 004	312	423 223	034	042 043					

002, 004, 312, 423, 223, 034, 042, 043

100의 자리 값으로 정렬

재정렬

0	1	2	3	4	5	6	7	8	9
002 004 034 042 043		223	312	423					

002, 004, 034, 042, 043, 223, 312, 423

기수정렬 (Radix Sort)

실습: 최대자리수를 파라미터로 전달받아 동작하도록 radixSort 함수 수정

```
public class Test {
	public static void main(String[] args) {
		int	n[]={4,423,312,34,43,2,223,42};
		radixSort(n);
		System.out.println(Arrays.toString(n));
	}
	private static void radixSort(int[] n) { // 세자리 십진수 대상
		int	Radix=10;
		LinkedList<Integer>		queue[]=new LinkedList[Radix]; // 십진수 0~9 대응 큐 생성
		for (int q = 0; q < queue.length; q++) queue[q]=new LinkedList<>(); // 십진수 0~9 대응 큐 생성

		for (int i = 0; i < n.length; i++) queue[(n[i]/1)%Radix].add(n[i]); // 1의 자리 값 기준 큐 삽입
		for (int q = 0, i=0; q < queue.length; q++) while(!queue[q].isEmpty()) n[i++]=queue[q].remove(); // 큐 자료 재정렬

		for (int i = 0; i < n.length; i++) queue[(n[i]/10)%Radix].add(n[i]); // 10의 자리 값 기준 큐 삽입
		for (int q = 0, i=0; q < queue.length; q++) while(!queue[q].isEmpty()) n[i++]=queue[q].remove(); // 큐 자료 재정렬

		for (int i = 0; i < n.length; i++) queue[(n[i]/100)%Radix].add(n[i]); // 100의 자리 값 기준 큐 삽입
		for (int q = 0, i=0; q < queue.length; q++) while(!queue[q].isEmpty()) n[i++]=queue[q].remove(); // 큐 자료 재정렬
	}
}
```

References

- C로 쓴 자료구조론 (Fundamentals of Data Structures in C, Horowitz et al.). 이석호 역. 사이텍미디어. 1993.
- 쉽게 배우는 알고리즘: 관계 중심의 사고법. 문병로. 한빛아카데미. 2013.
- C언어로 쉽게 풀어 쓴 자료구조. 천인국 외 2인. 생능출판사. 2017.
- 프로그래밍 콘테스트 챌린징, Akiba 등 공저, 로드북, 2011.
- https://introcs.cs.princeton.edu/
- Introduction to Algorithms, Cormen et al., 3rd Edition (The MIT Press)