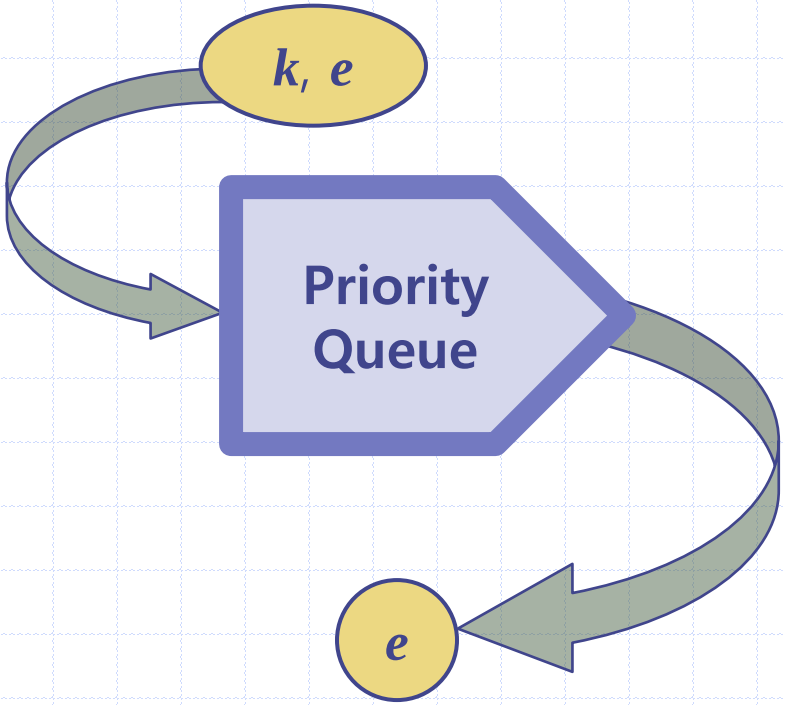
우선순위 큐

Outline

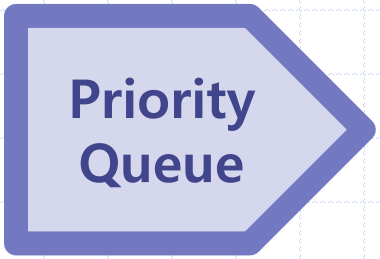
- 5.1 우선순위 큐 ADT
- 5.2 우선순위 큐를 이용한 정렬
- 5.3 제자리 정렬
- 5.4 선택 정렬과 삽입 정렬 비교
- 5.5 응용문제

우선순위 큐 ADT

- 우선순위 큐 ADT는 항목들을 저장
- 각 항목: (**키**, **원소**) 쌍
- **응용**
  - 탑승 대기자
  - 옥션
  - 주식시장

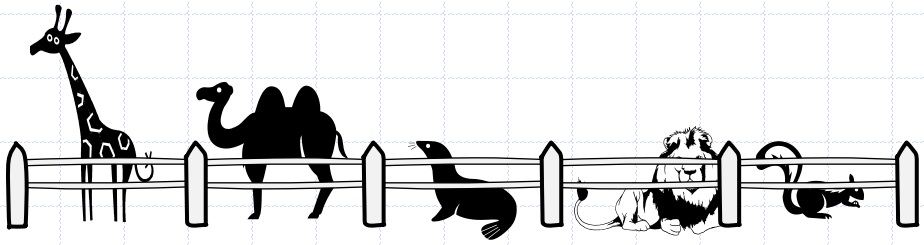


우선순위 큐 ADT 메쏘드



- 주요 메쏘드
  - insertItem(k, e): 키 k인 원소 e를 큐에 삽입
  - element removeMin(): 큐로부터 최소 키를 가진 원소를 삭제하여 반환
- 일반 메쏘드
  - integer size(): 큐의 항목 수를 반환
  - boolean isEmpty(): 큐가 비어 있는지 여부를 반환
- 접근 메쏘드
  - element minElement(): 큐에서 최소 키를 가진 원소를 반환
  - element minKey(): 큐에서 최소 키를 반환
- 예외
  - emptyQueueException(): 비어 있는 큐에 대해 삭제나 원소 접근을 시도할 경우 발령
  - fullQueueException(): 만원 큐에 대해 삽입을 시도할 경우 발령

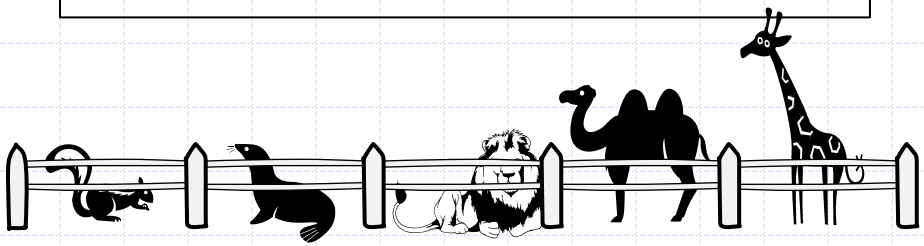
우선순위 큐를 이용한 정렬



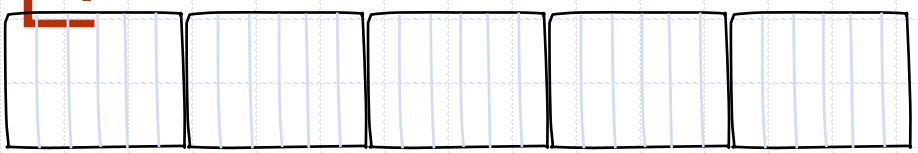
- 비교 가능한 원소 집합을 정렬하는데 **우선순위 큐** 이용 가능
  1. 연속적인 insertItem(e, e) 작업을 통해 원소들을 하나씩 삽입 (key = e로 전제)
  2. 연속적인 removeMin() 작업을 통해 원소들을 정렬 순서로 삭제
- **실행시간:** 우선순위 큐의 **구현**에 따라 다르다
- ***L***, ***P***: **일반**(generic)

Alg ***PQ-Sort(L)***
  input list ***L***
  output sorted list ***L***

1. ***P*** ← *empty priority queue*
2. **while** (!***L.isEmpty***())
   e ← ***L.removeFirst***()
   P.insertItem(***e***)
3. **while** (!***P.isEmpty***())
   e ← ***P.removeMin***()
   L.addLast(***e***)
4. **return**



리스트에 기초한 우선순위 큐



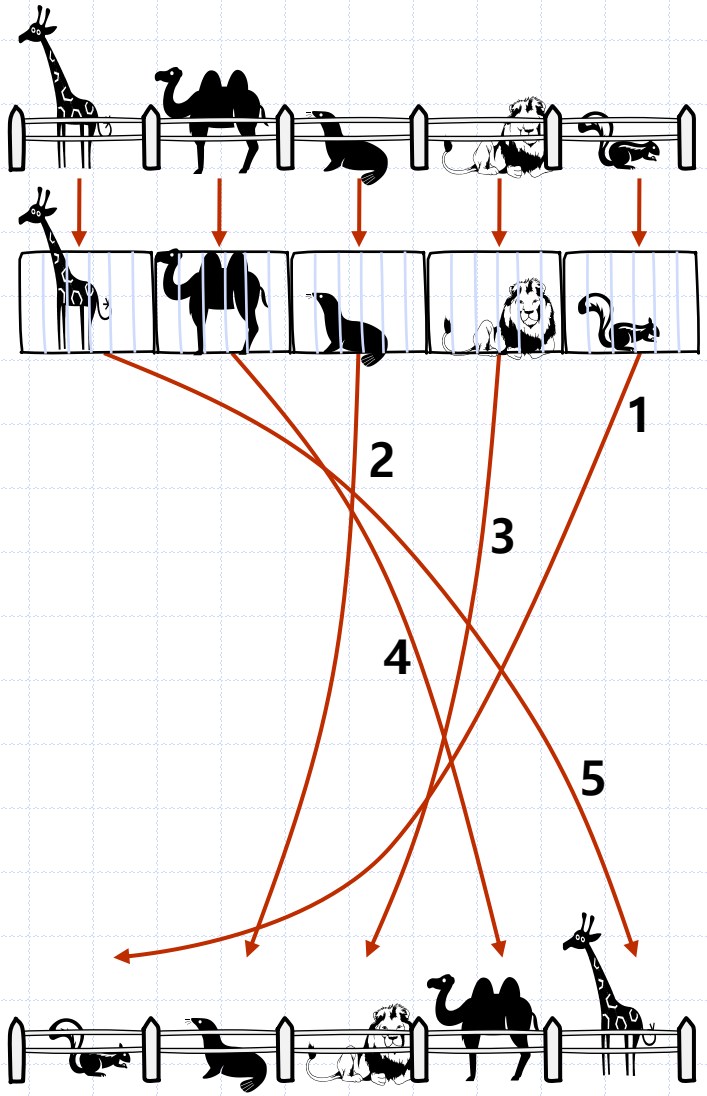
- **무순리스트**로 구현
  - 우선순위 큐의 항목들을 리스트에 임의 순서로 저장
- 성능
  - insertItem: $O(1)$ 시간 소요 – 항목을 리스트의 맨앞 또는 맨뒤에 삽입할 수 있으므로
  - removeMin, minKey, minElement: $O(n)$ 시간 소요 – 최소 키를 찾기 위해 전체 리스트를 순회해야 하므로

- **순서리스트**로 구현
  - 우선순위 큐의 항목들을 리스트에 키 정렬 순서로 저장
- 성능
  - insertItem: $O(n)$ 시간 소요 – 항목을 삽입할 곳을 찾아야 하므로
  - removeMin, minKey, minElement: $O(1)$ 시간 소요 – 최소 키가 리스트의 맨앞에 있으므로

선택 정렬

- **선택 정렬**(selection-sort)
  - PQ-Sort의 일종
  - 우선순위 큐를 **무순리스트**로 구현
- 실행시간
  - n회의 insertItem 작업을 사용하여 원소들을 **우선순위 큐**에 삽입하는데 $O(n)$ 시간 소요
  - n회의 removeMin 작업을 사용하여 원소들을 **우선순위 큐**로부터 정렬 순서로 삭제하는데 다음에 비례하는 시간 소요

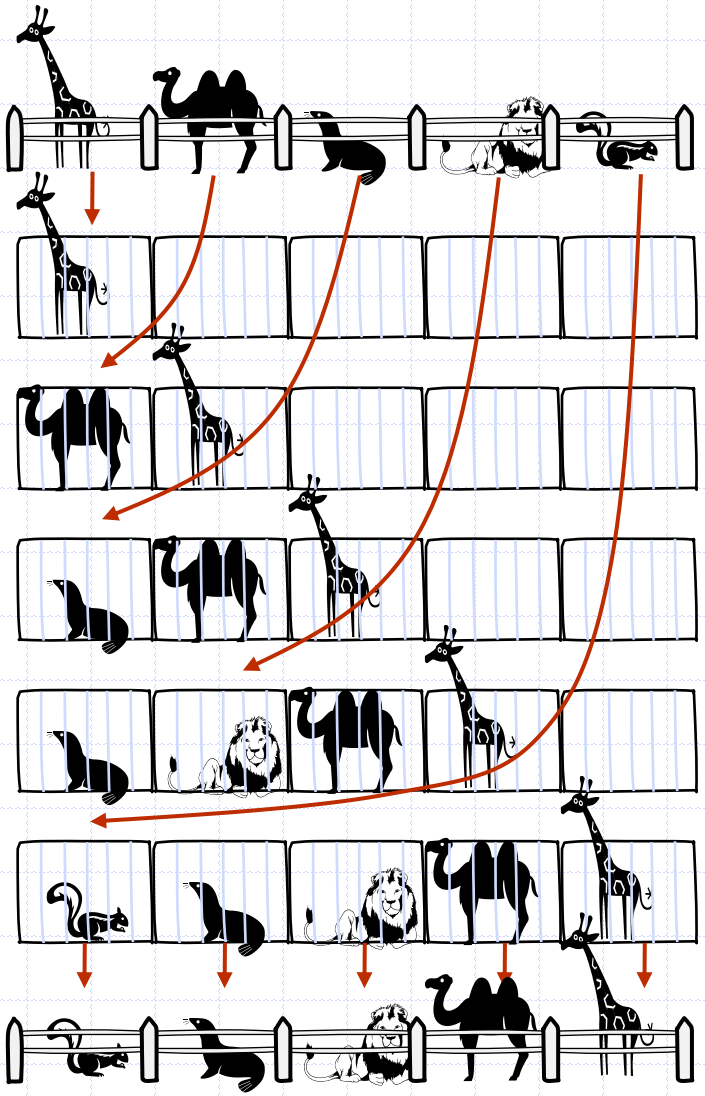
    $n + (n-1) + (n-2) + \ldots + 2 + 1$
  - **Total**: $O(n^2)$



삽입 정렬

- **삽입 정렬**(insertion-sort)
  - PQ-Sort의 일종
  - 우선순위 큐를 **순서리스트**로 구현
- 실행시간
  - n회의 insertItem 작업을 사용하여 원소들을 **우선순위 큐**에 삽입하는데 다음에 비례하는 시간 소요

    $1 + 2 + \ldots + (n-2) + (n-1) + n$
  - n회의 removeMin 작업을 사용하여 원소들을 **우선순위 큐**로부터 정렬 순서로 삭제하는데 $O(n)$ 시간 소요
  - **Total**: $O(n^2)$

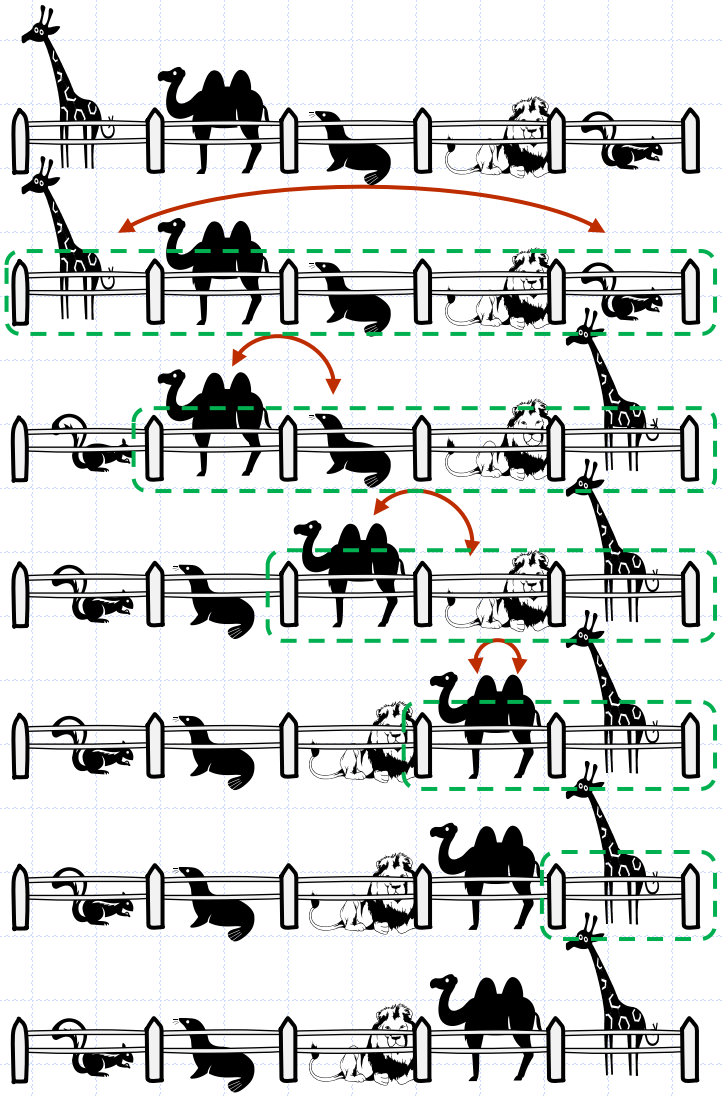


"제자리"에서 할 수 있나?

- selection-sort, insertion-sort 모두 $\Theta(n)$ 공간을 차지하는 **외부의** 우선순위 큐를 사용하여 리스트를 정렬
- 만약, 원래 리스트 자체를 위한 공간 이외에 $\mathbf{O}(1)$ 공간만을 사용한다면, 이를 "**제자리**(in-place)"에서 수행한다고 말한다
- 일반적으로, 어떤 정렬 알고리즘이 정렬 대상 개체를 위해 필요한 메모리에 추가하여 오직 상수 메모리만을 사용한다면, 해당 정렬 알고리즘이 **제자리**에서 수행한다고 말한다

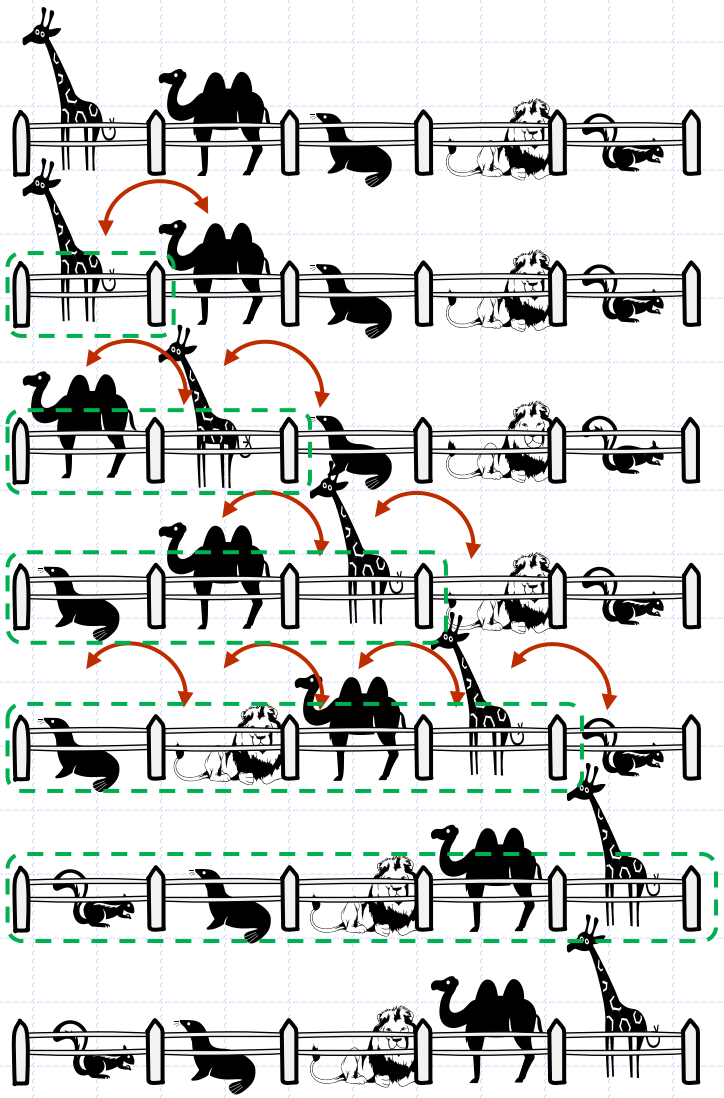
제자리 선택 정렬

- selection-sort가 외부 데이터구조를 사용하는 대신, **제자리**에서 수행하도록 구현 가능
- **우선순위 큐:** 입력 리스트의 일부(점선 부분)
- In-place selection-sort를 위해서는:
  - 리스트의 앞부분을 정렬 상태로 유지
  - 리스트를 변경하는 대신 swapElements를 사용



제자리 삽입 정렬

- insertion-sort가 외부 데이터구조를 사용하는 대신, **제자리**에서 수행하도록 구현 가능
- **우선순위 큐:** 입력 리스트의 일부(점선 부분)
- in-place insertion-sort를 위해서는:
  - 리스트의 앞부분을 정렬 상태로 유지
  - 리스트를 변경하는 대신 swapElements를 사용



제자리 정렬 알고리즘

Alg *inPlaceSelectionSort*(*A*)
    input array *A* of *n* keys
    output sorted array *A*

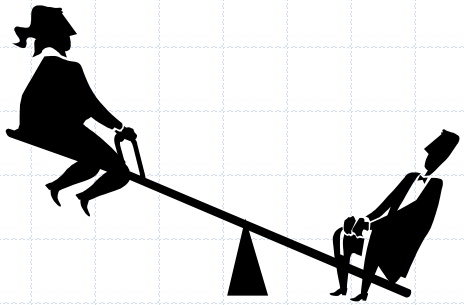
1. **for** *pass* ← 0 **to** *n* – 2
        minLoc ← *pass*
        for *j* ← (*pass* + 1) **to** *n* – 1
            if (*A*[*j*] < *A*[*minLoc*])
                minLoc ← *j*
        A[*pass*] ↔ *A*[*minLoc*]
2. **return**

Alg *inPlaceInsertionSort*(*A*)
    input array *A* of *n* keys
    output sorted array *A*

1. **for** *pass* ← 1 **to** *n* – 1
        save ← *A*[*pass*]
        j ← *pass* – 1
        while ((*j* ≥ 0) & (*A*[*j*] > *save*))
            A[*j* + 1] ← *A*[*j*]
            j ← *j* – 1
        A[*j* + 1] ← *save*
2. **return**

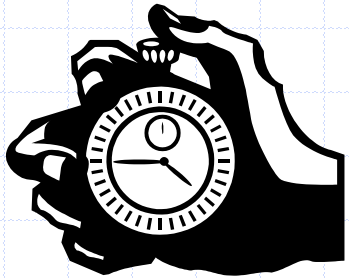
선택 정렬 vs. 삽입 정렬

- 공통점
  - 전체적으로 $O(n^2)$ 시간
    - 내부 반복문: $O(n)$ **선형 탐색**
    - 외부 반복문: $O(n)$ 패스
  - 제자리 버전은 $O(1)$ 공간 소요
  - 구현이 단순
  - 작은 n에 대해 유용



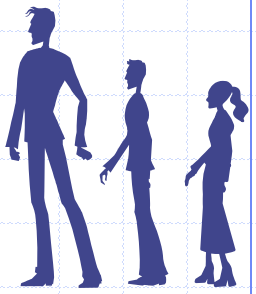
- 초기 리스트가 **완전히** 또는 **거의** 정렬된 경우
  - in-place insertion-sort가 더 빠르다
  - 내부 반복문이 $O(1)$ 시간 소요 – 따라서 전체적으로 $O(n)$ 시간에 수행되므로
- swapElements 작업이 비싼 경우
  - in-place selection-sort가 더 빠르다
  - swapElements 작업이 패스마다 $O(1)$ 시간 수행되는데 반해, in-place insertion-sort에서는 동일 작업이 패스마다 최악의 경우 $O(n)$ 시간 수행되므로

성능 요약



우선순위 큐	작업 수행시간			정렬 방식
	insertItem	removeMin	minKey, minElement	
무순리스트	$O(1)$	$O(n)$	$O(n)$	선택 정렬
순서리스트	$O(n)$	$O(1)$	$O(1)$	삽입 정렬

응용문제: 역치와 삽입 정렬



- L을 n개의 원소로 이루어진, 전체순서 관계가 정의된 리스트로 가정
- L 내의 **역치**(inversion)란 x가 y 앞에 나타나지만 $x > y$인 원소 쌍 x, y를 말한다

A. 정수 $[0, n-1]$ 범위의 유일한 원소로 이루어진 크기 n의 리스트 L 내에 **최대 가능한 역치의 수**와 이때의 L의 원소 배치를 구하라

B. 만약 모든 원소가 바른 자리(즉, 정렬 시점에 있어야 할 자리)에서 k칸 이내에 위치한다면, L에 대한 **삽입 정렬** 수행에 $O(nk)$ 시간이 소요됨을 설명하라
   - **힌트:** 삽입 정렬 알고리즘을 검토하여 우선, I가 리스트 L 내의 총 역치의 수라고 할 때 삽입 정렬이 $O(n+I)$ 시간에 수행함을 설명하라

해결

- $L = (n-1, n-2, \ldots, 2, 1, 0)$처럼 역정렬 상태인 경우며, 이때 역치의 수는 $n(n-1)/2$개다
- 리스트에 대한 삽입 정렬의 수행을 검토함으로써, 우선, I가 리스트 L 내의 총 역치의 수라고 할 때 삽입 정렬이 $O(n+I)$ 시간에 수행함을 설명하자
  - 외부 반복문에 대해 $O(n)$ 작업을 수행한다
  - 내부 반복문의 한 회전은 한 칸씩 왼쪽으로 진행하면서 딱 한 개의 역치를 교정한다
  - 알고리즘이 종료하면 역치는 남지 않는다
  - 그러므로 I가 리스트 L 내 역치의 수라고 할 때, 내부 반복문은 정확히 총 I회전을 수행해야 한다
  - 따라서 내부 반복문에 $O(I)$ 시간이 소요된다
  - 그러므로 알고리즘은 합계 $O(n+I)$ 시간에 수행한다

해결 (conti.)

- 이제 모든 원소가 바른 자리에서 k칸 이내에 위치하는 경우 총 역치의 수를 계산해보자
  - 총 역치의 수에 대해 상한을 설정하기로 한다
  - 리스트의 특정 원소 i에 대해, 리스트 내 최대 $4k$개의 원소들이 원소 i와 역치 상태에 있을 수 있다
  - 이들은 $i - 2k$에서 $i + 2k$ 범위에 있는 원소들이다
  - 따라서 최대 총 $4nk$개의 역치가 있을 수 있다
  - 그러므로 알고리즘은 전체적으로 $O(nk)$ 시간에 수행한다