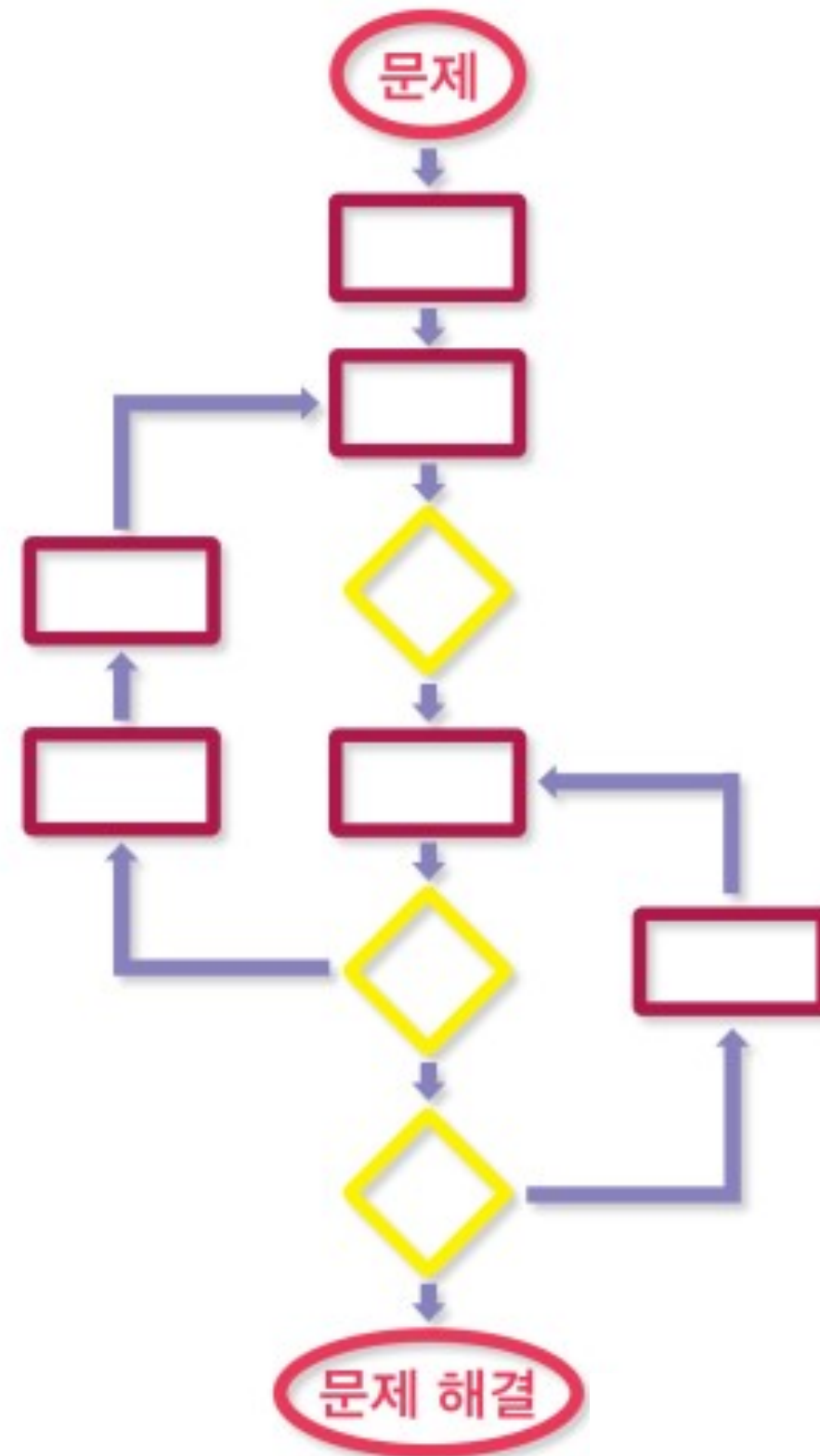


Algorithm

Algorithm [ælgərɪðəm]

- 앨거리덤, 알고리듬 / 알고리즘(X)
- 어떤 문제를 해결하기 위해 정의된 절차와 방법, 명령어의 집합
- 9세기 페르시아 수학자, 무하마드 알콰리즈미(Muhammad al-Kwarizmi) 의 이름에서 유래
'콰라즘에서 온 사람이 가르쳐 준 수 (알 콰라즘)' -> 알고리즘

Flowchart



Flowchart

기호	명칭	설명
 시작/종료	단말	순서도의 시작과 끝을 나타냄.
	흐름선	순서도 기호 간의 연결 및 작업의 흐름을 표시함.
 준비	준비	작업 단계 시작 전 해야 할 작업을 명시함.
 처리	처리	처리해야 할 작업을 명시함.
 입출력	입출력	데이터의 입출력 시 사용함.
 판단	의사 결정	비교 및 판단에 의한 논리적 분기를 나타냄.
 표시	표시	화면으로 결과를 출력함.

알고리즘은 다음의 조건을 만족해야 함

- 입력 : 외부에서 제공되는 자료가 0개 이상 존재한다.
- 출력 : 적어도 1개 이상의 결과물을 출력해야 한다.
- 명확성 : 수행 과정은 명확해야 하고 모호하지 않은 명령어로 구성되어야 한다.
- 효율성 : 모든 과정은 명백하게 실행 가능(검증 가능)한 것이어야 한다.
- 유한성(종결성) : 알고리즘의 명령어들은 계산을 수행한 후 반드시 종료해야 한다.

Examples

- 택배를 가장 빠르게 배달할 수 있는 루트
- 로봇 청소기의 움직임
- 자동 주식 거래 시스템
- 최적의 검색 결과
- 얼굴 / 지문 인식
- Siri

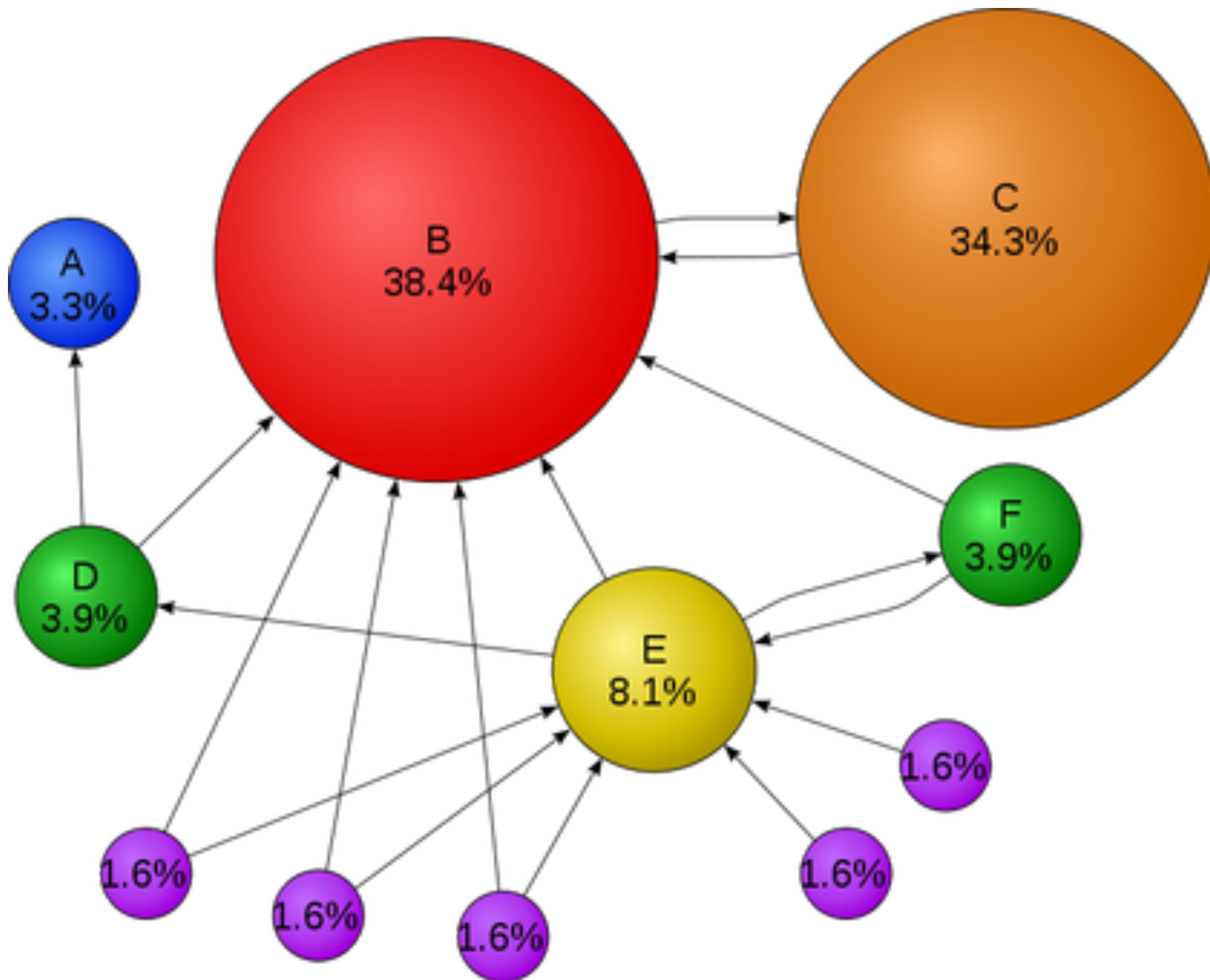
Google Search Algorithm

세르게이 브린, 래리 페이지 논문

- The Anatomy of a Large-Scale Hypertextual Web Search Engine ([링크](#))
- PageRank : 특정 페이지를 인용하는 다른 페이지가 얼마나 많이 있는지를 통해 랭킹 반영



PageRank



Why Algorithm

[알고리즘을 공부해야 하는 이유]

- 문제 해결력 상승
- 평상시 기능 구현 집중. 시간 복잡도 / 공간 복잡도 / 메모리 관심 X
알고리즘 학습을 통해 해당 부분에 대한 관심 + 고민 + 접근법 향상
- 관련된 케이스를 알고 있으면 다음에 유사한 문제를 쉽게 해결 가능
- 취업.

Complexity

Complexity

시간복잡성 - 데이터 연산 시간은 가능한 작아야 함

공간복잡성 - 데이터 연산 및 저장에 필요한 메모리 공간은 가능한 작아야 함

Worst Case

: 빅-오 (O , Big-Oh) 표기법.

: 최악의 경우를 가정. 시간 복잡도를 말할 때 가장 일반적으로 사용

Average Case

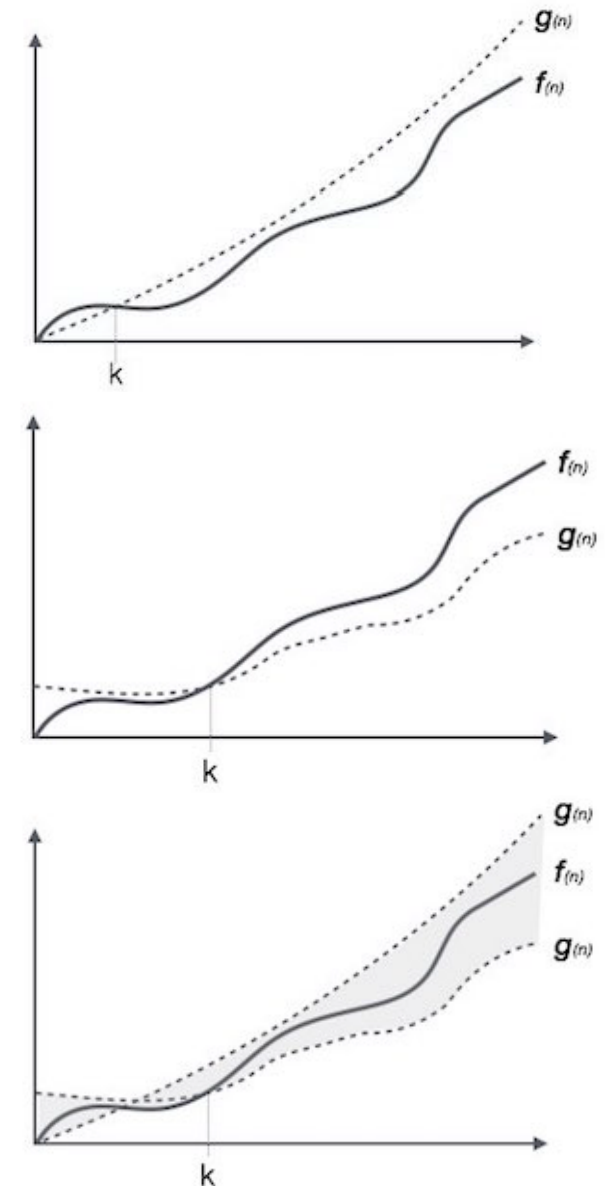
: 세타 (Θ , Theta) 표기법

: 평균적인 경우를 가정

Best Case

: 빅-오메가 (Ω , Big-Omega) 표기법

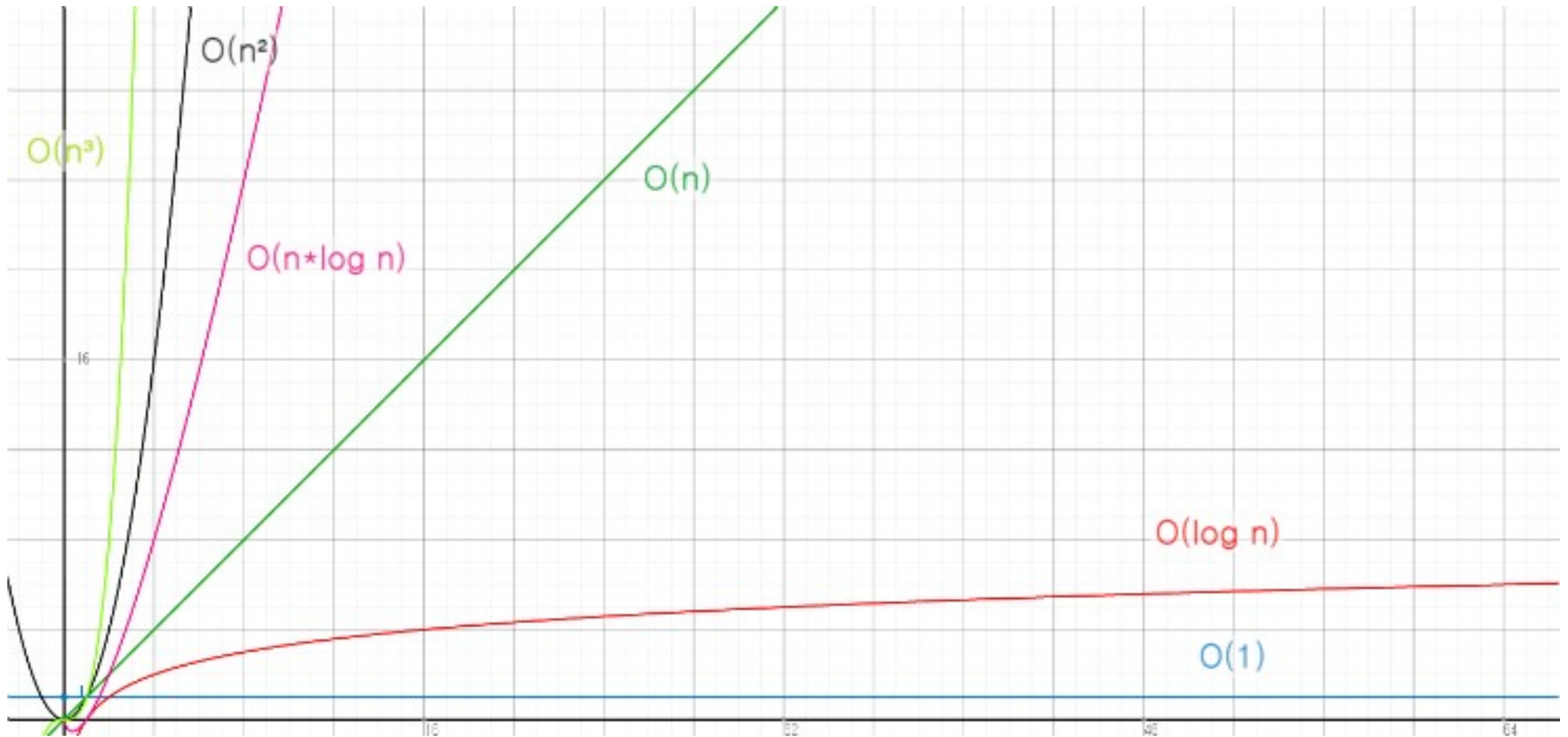
: 최선의 경우를 가정



Common Asymptotic Notations

constant	—	$O(1)$
logarithmic	—	$O(\log n)$
linear	—	$O(n)$
$n \log n$	—	$O(n \log n)$
quadratic	—	$O(n^2)$
cubic	—	$O(n^3)$
polynomial	—	$n^{O(1)}$
exponential	—	$2^{O(n)}$

Complexity



Sorting Algorithm

Sorting Algorithms

정렬 알고리즘 - 알고리즘을 소개할 때 가장 대표적으로 소개되는 케이스

worst, average, best case 등을 이해하기 쉽고 시각적으로 표현하기 좋음

다양한 곳에서 자주 쓰이기 때문에 이미 다양한 알고리즘이 나와있고 현재도 계속 연구중

[참고 링크]

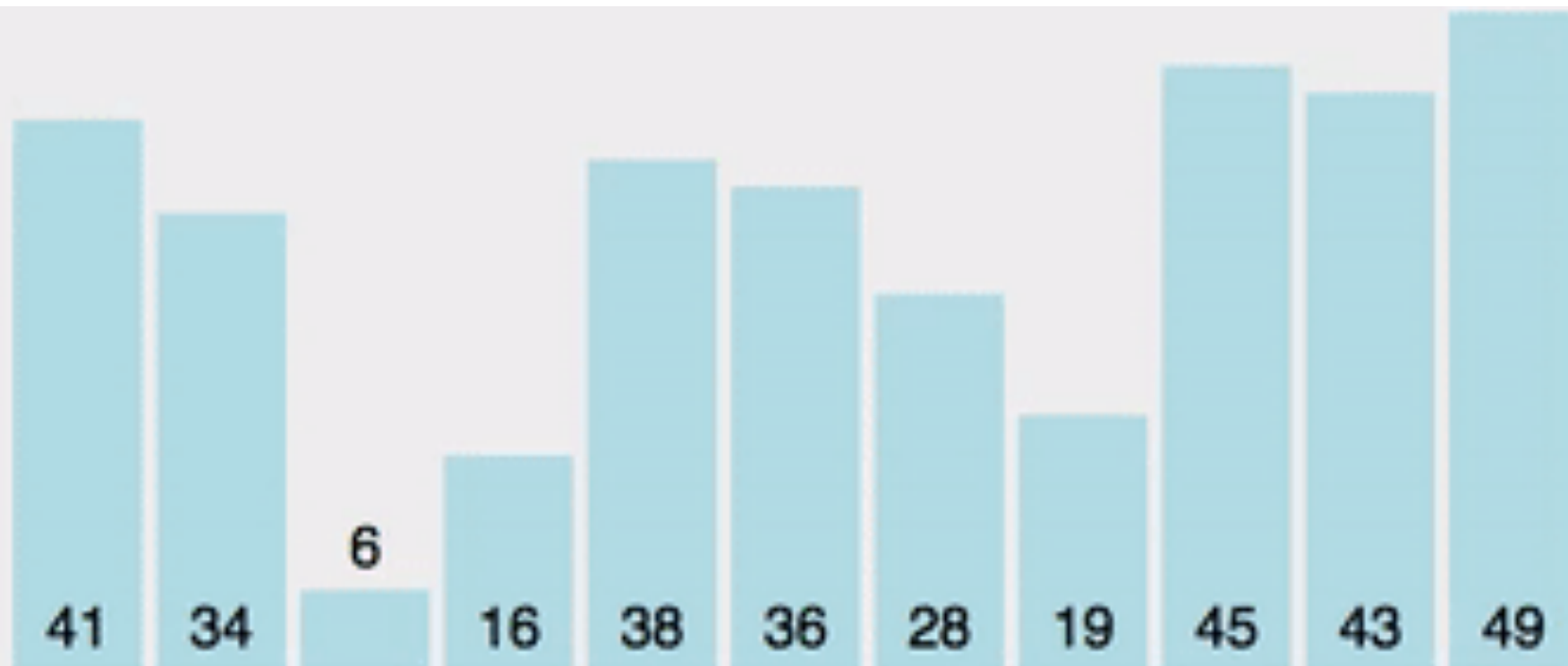
- 15 Sorting Algorithms - <https://goo.gl/fG3TJ1>
- Visualization and Comparison of Sorting Algorithms - <https://goo.gl/ayJvBG>
- Bubble-sort with Hungarian ("Csángó") folk dance - <https://goo.gl/Lwk7ya>
- Merge Sort vs Quick Sort - <https://goo.gl/U1sL9w>
- Sorting - <http://sorting.at>
- swift-algorithm-club - <https://github.com/raywenderlich/swift-algorithm-club>

Bubble Sort

인접한 두 원소의 크기를 비교하여 큰 값을 배열의 오른쪽으로 정렬해 나가는 방식

이미 대부분 정렬되어 있는 자료에서는 좋은 성능을 보이지만 그 외에는 매우 비효율적인 알고리즘
단, 직관적이어서 쉽고 빠르게 구현 가능하여 많이 알려져 있음

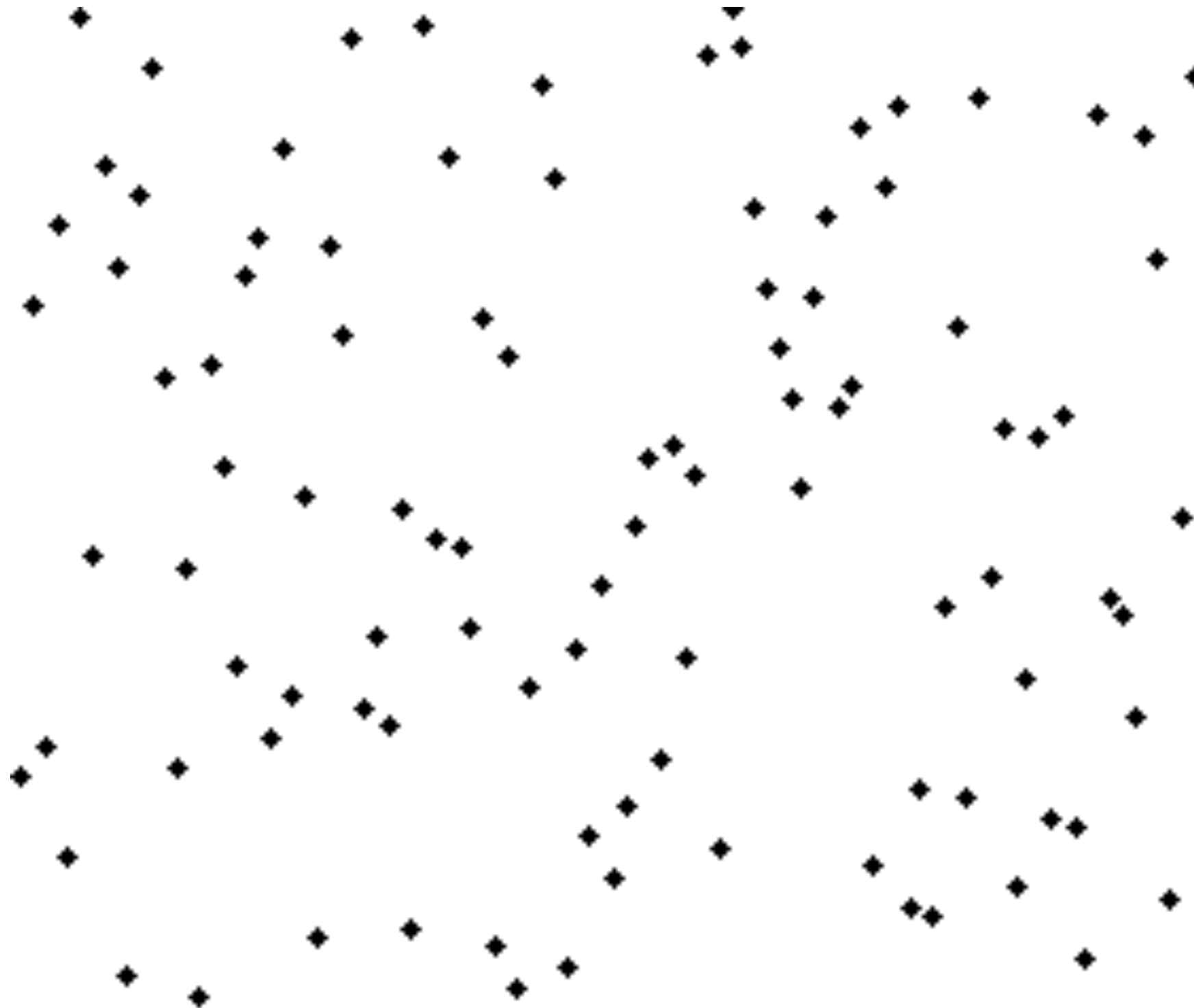
시간복잡도 - $O(n^2)$ ※ $(n-1) + (n-2) + \dots + 2 + 1 \Rightarrow n(n-1)/2$



Bubble Sort

5	3	7	1	6	9	2	5와 3 교환
3	5	7	1	6	9	2	교환 없음
3	5	7	1	6	9	2	7와 1 교환
3	5	1	7	6	9	2	7와 6 교환
3	5	1	6	7	9	2	교환 없음
3	5	1	6	7	9	2	9와 2 교환
3	5	1	6	7	2	9	

Bubble Sort

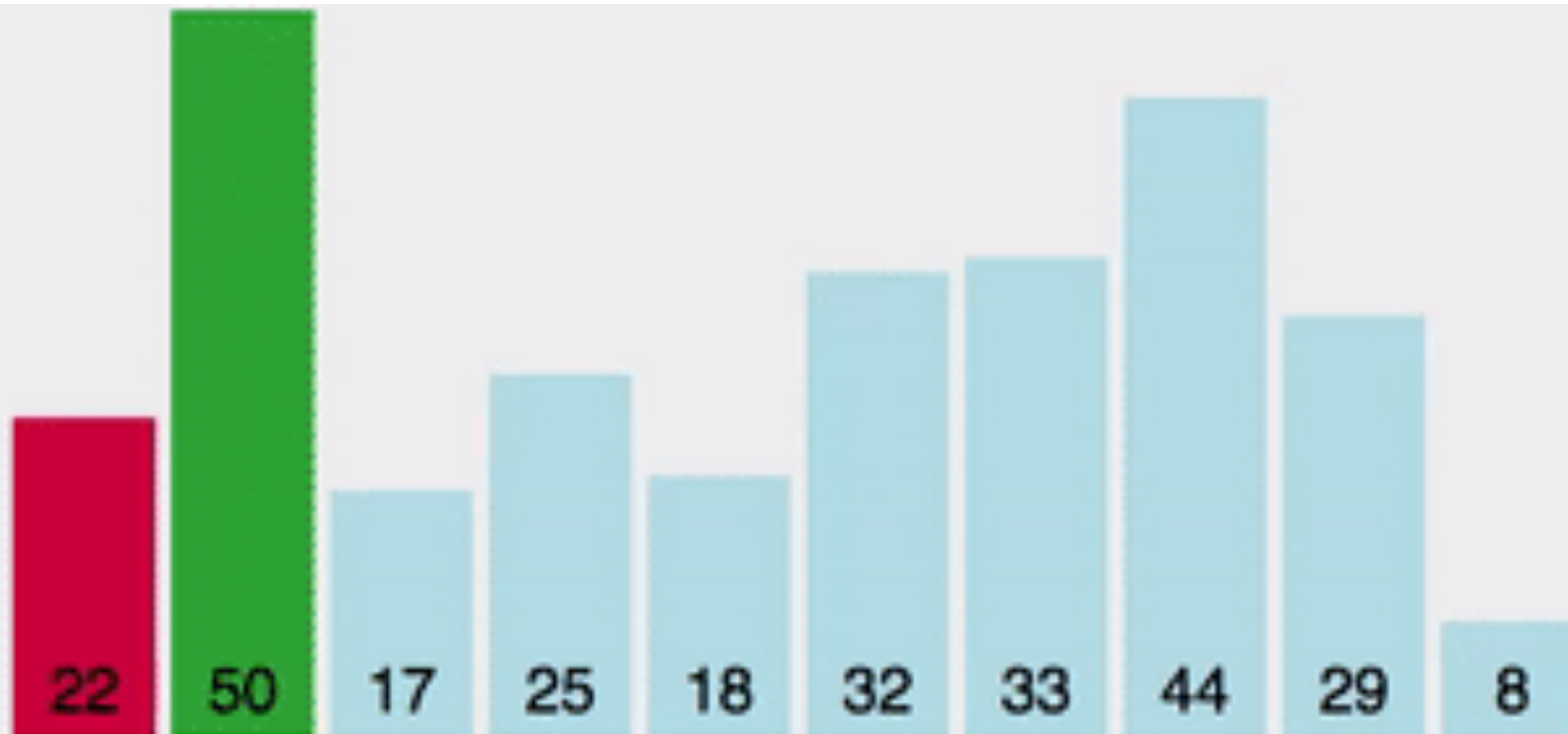


Selection Sort

데이터를 반복 순회하며 최소값을 찾아 정렬되지 않는 숫자 중 가장 좌측의 숫자와 위치 교환하는 방식


- 최소값 선택 정렬 (Min-Selection Sort) : 가장 작은 값을 기준으로 정렬 (오름차순)
- 최대값 선택 정렬 (Max-Selection Sort) : 가장 큰 값을 기준으로 정렬 (내림차순)

시간복잡도 - $O(n^2)$

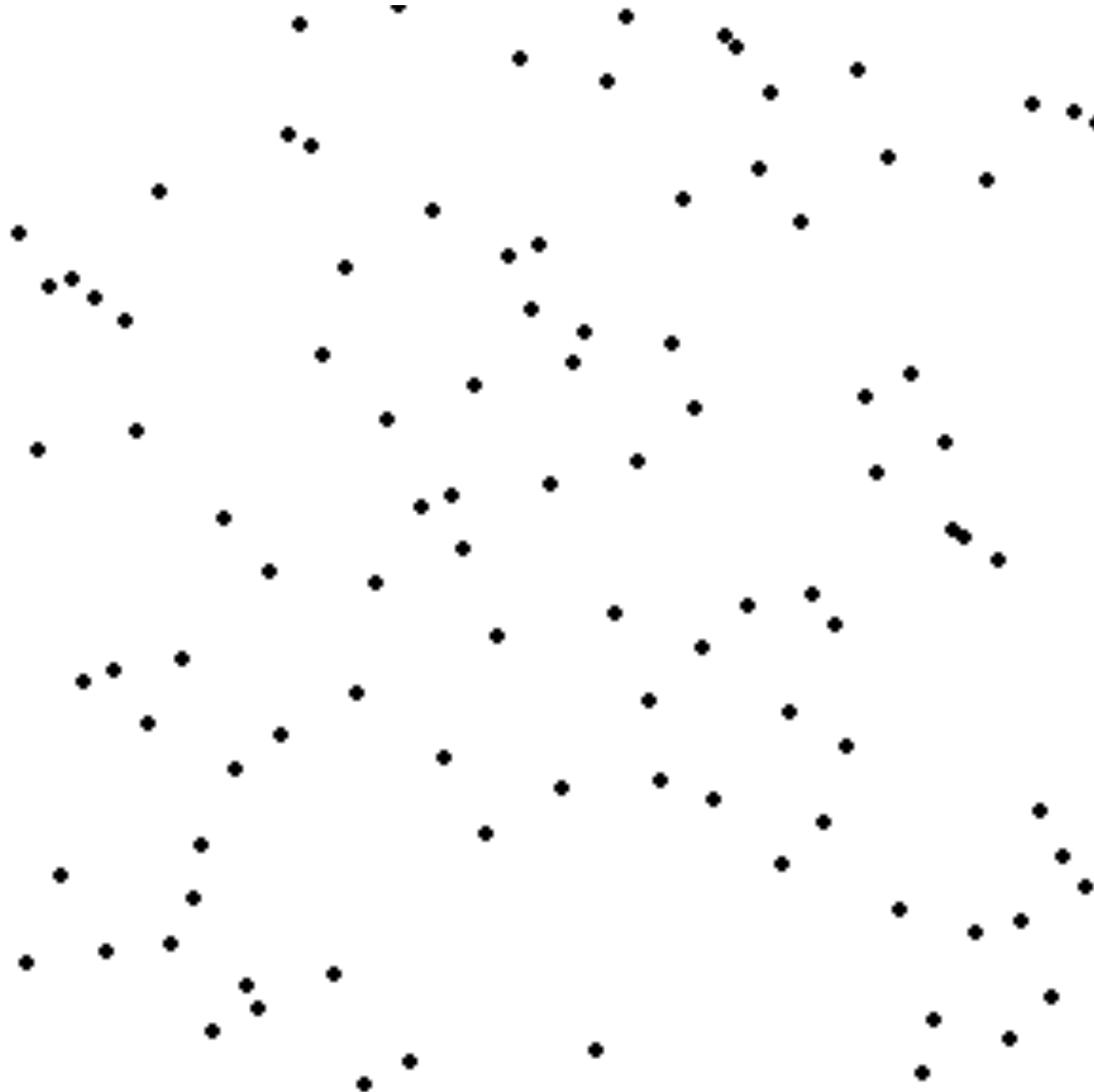


Selection Sort

Min Selection	0	1	2	3	4	5
Step 0	5	2	4	6	1	3
Step 1	1	2	4	6	5	3
Step 2	1	2	4	6	5	3
Step 3	1	2	3	6	5	4
Step 4	1	2	3	4	5	6
Step 5	1	2	3	4	5	6

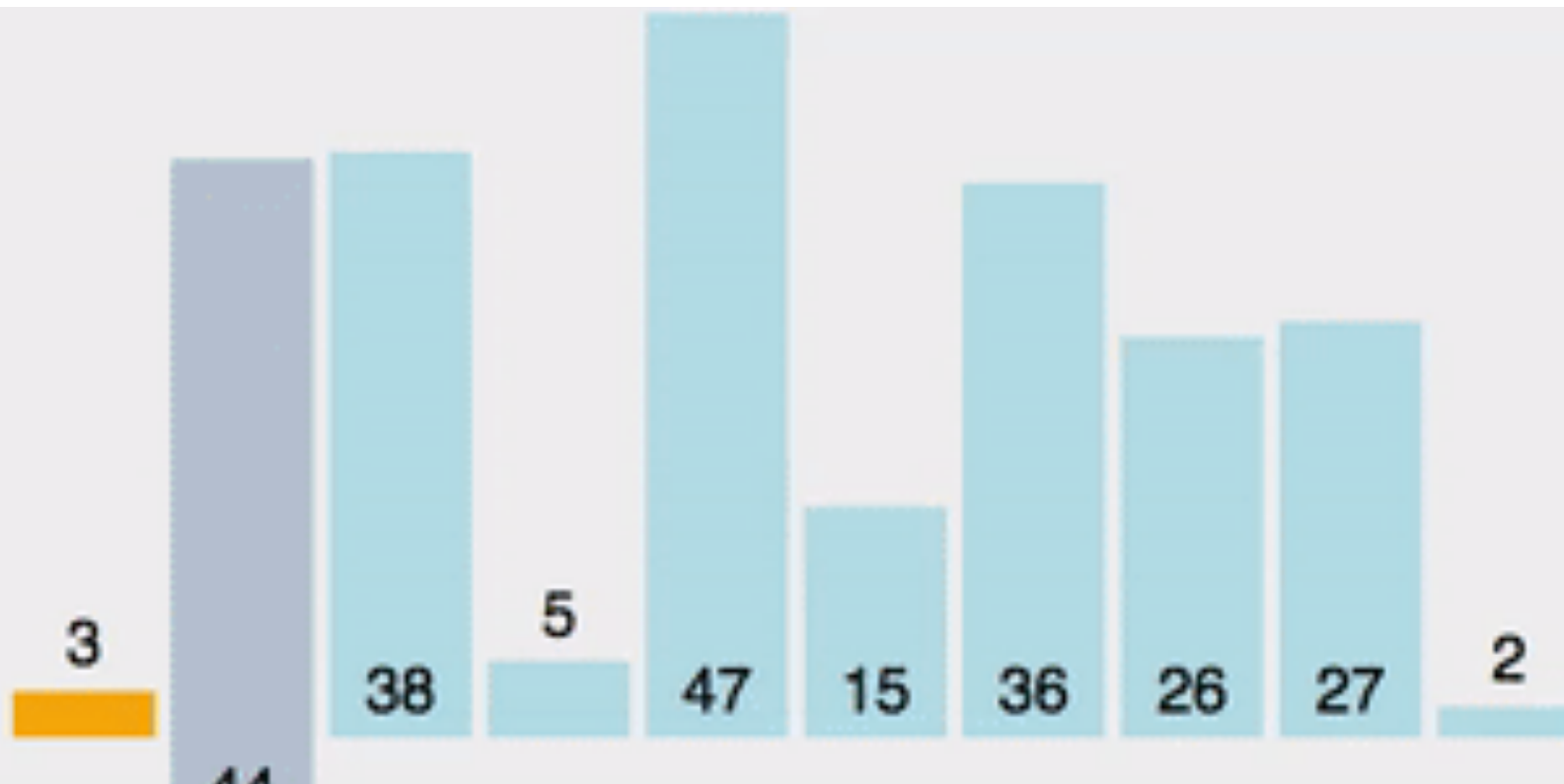


Selection Sort



Insertion Sort

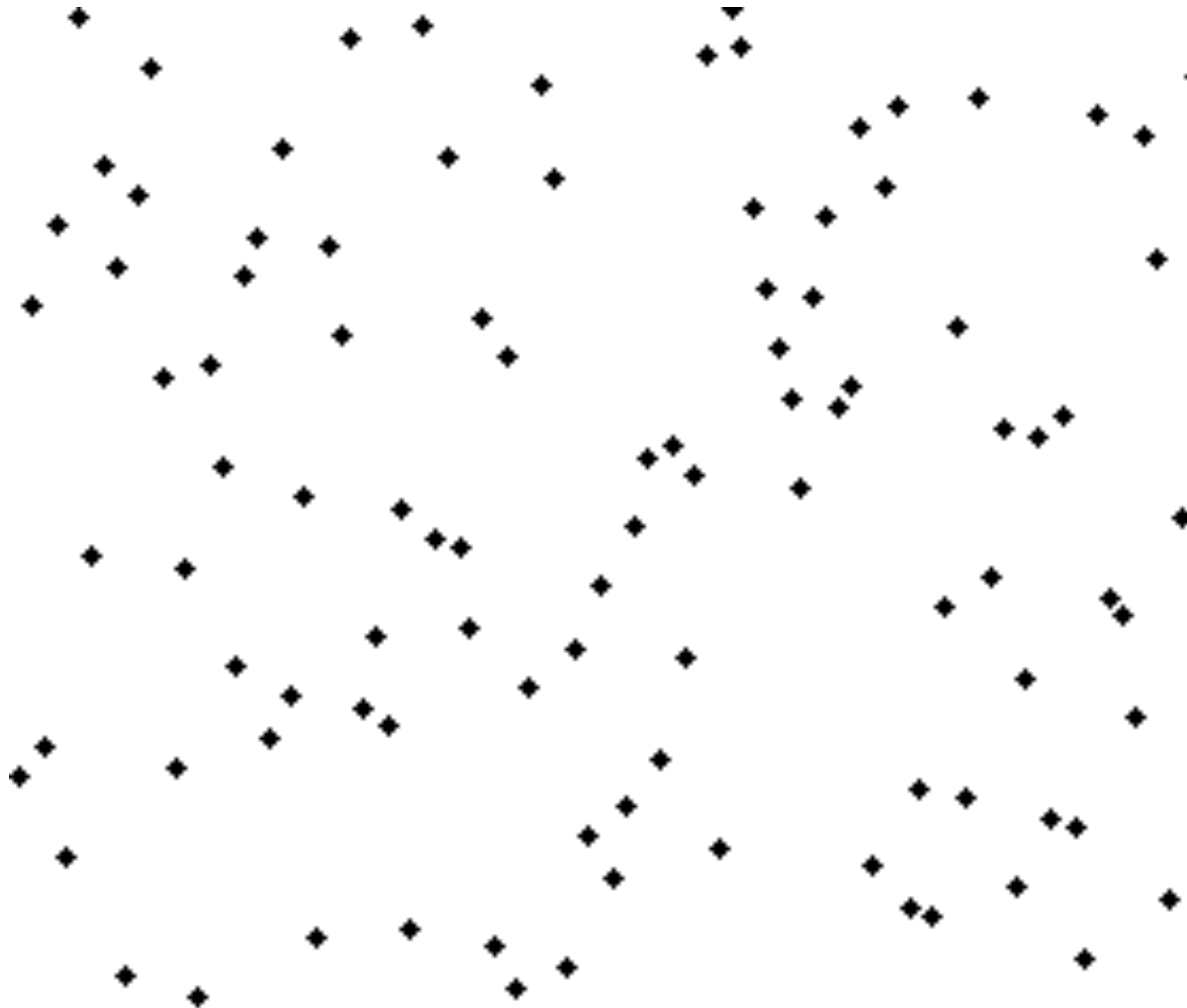
배열을 순회하며 현재 위치보다 작은 인덱스에 현재 값보다 더 작은 값이 있을 경우 해당 위치에 삽입하는 방식
 $O(n^2)$ 정렬 알고리즘 삼대장 (버블, 선택, 삽입) 중 일반적으로 가장 빠른 알고리즘
시간복잡도 - $O(n^2)$



Insertion Sort

Insertion	1	2	3	4	5	6
Step 1	5	2	4	6	1	3
Step 2	2	5	4	6	1	3
Step 3	2	4	5	6	1	3
Step 4	2	4	5	6	1	3
Step 5	1	2	4	5	6	3
Step 6	1	2	3	4	5	6

Insertion Sort



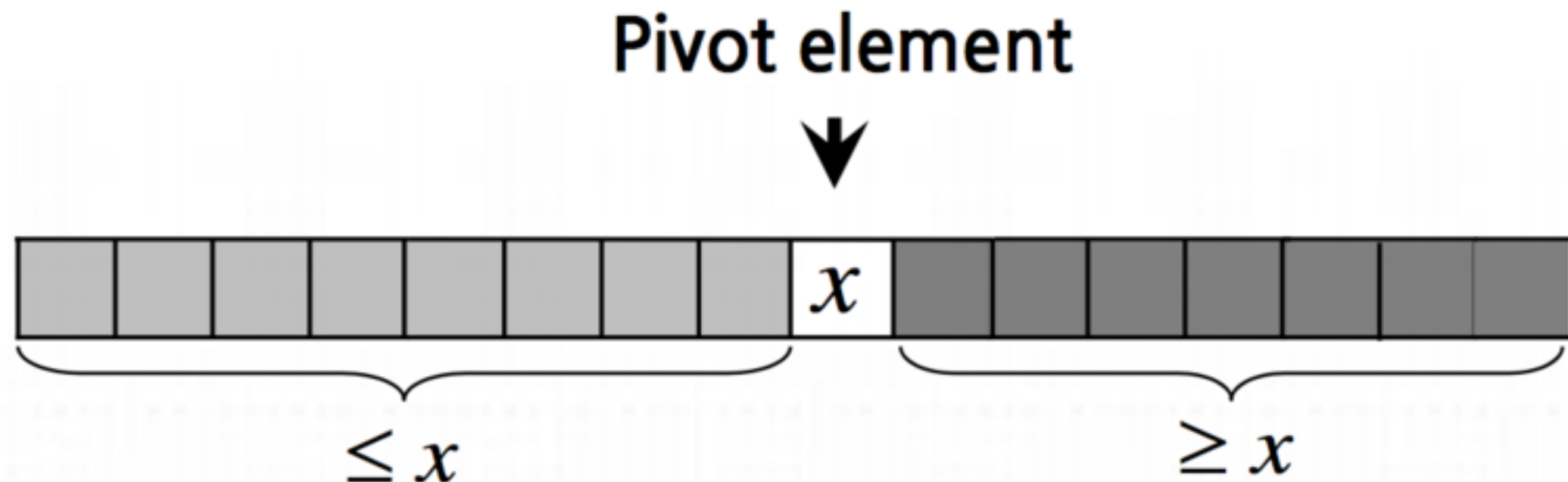
Quick Sort

기준이 되는 pivot 을 선정해 이를 기준으로 작은 값은 좌측, 큰 값을 우측에 재배치하는 것을 반복하는 방식

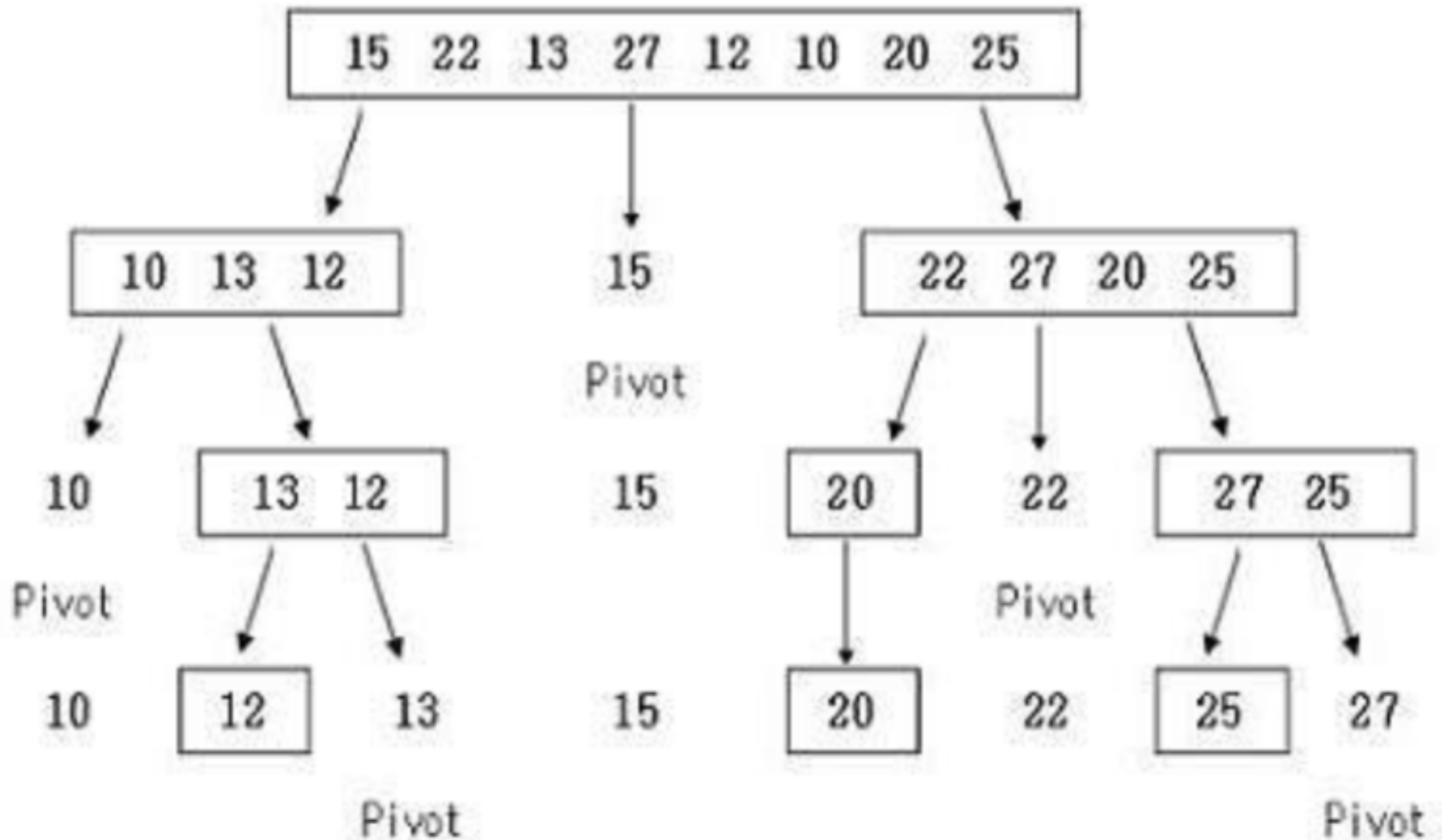
분할 정복 방식 (Divide and Conquer) 사용

일반적으로 실세계 데이터에 대입했을 때 가장 빠르다고 알려져있고 가장 많이 활용되는 정렬 알고리즘

시간복잡도 - 평균적으로 $O(n \log n)$, 최악의 경우 $O(n^2)$



Quick Sort



Quick Sort

QUICKSORT(A, p, r)

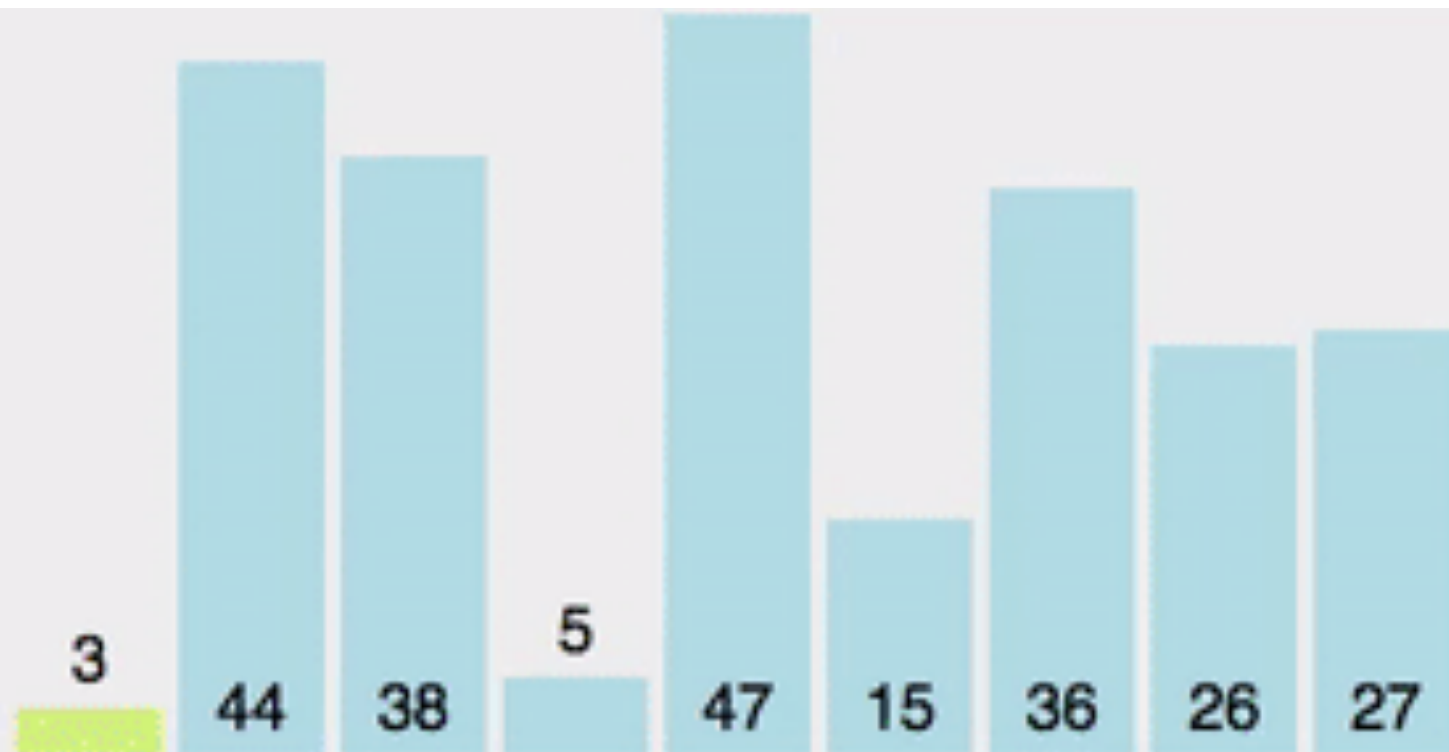
if $p < r$

$q = \text{PARTITION}(A, p, r)$

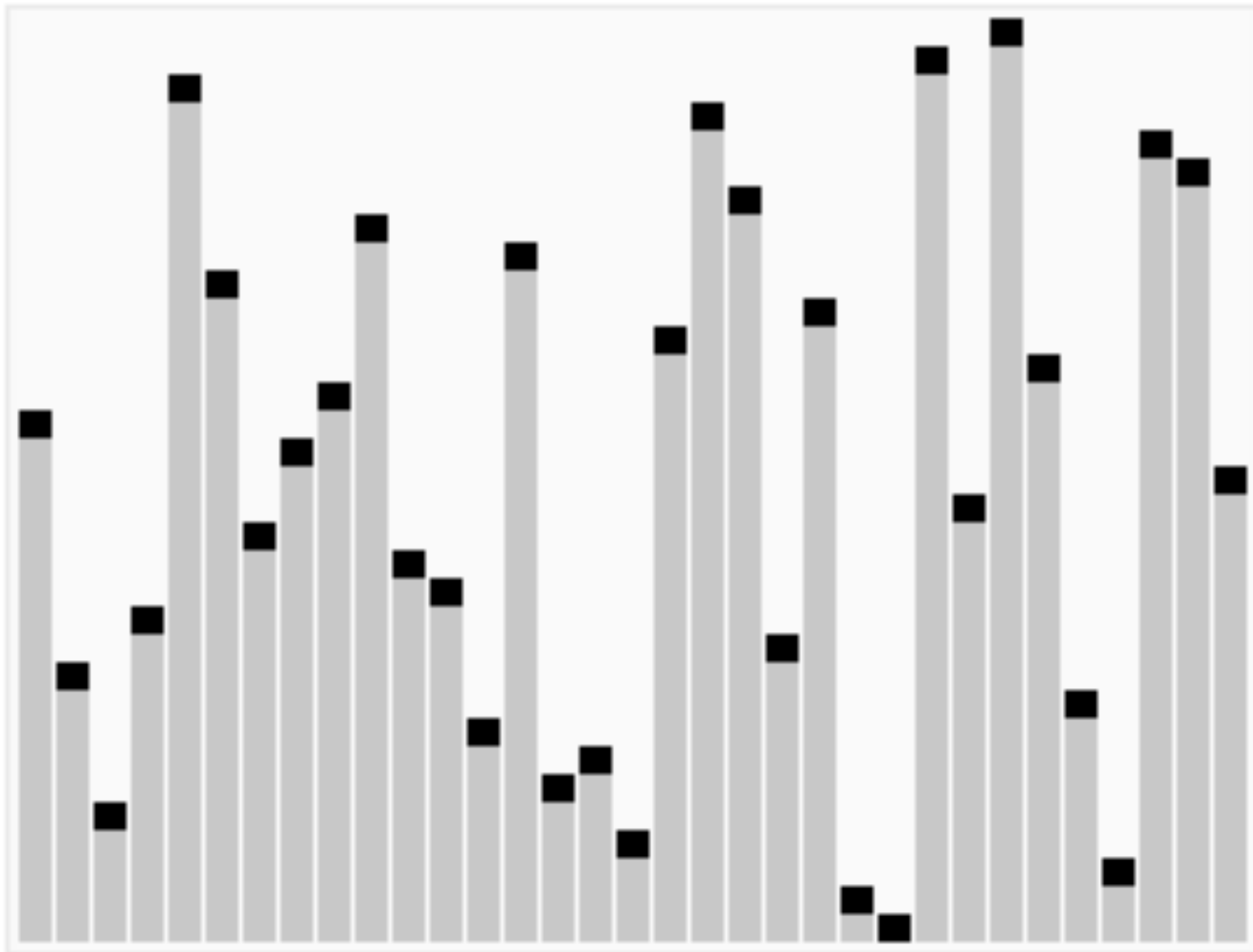
QUICKSORT($A, p, q - 1$)

QUICKSORT($A, q + 1, r$)

Quick Sort

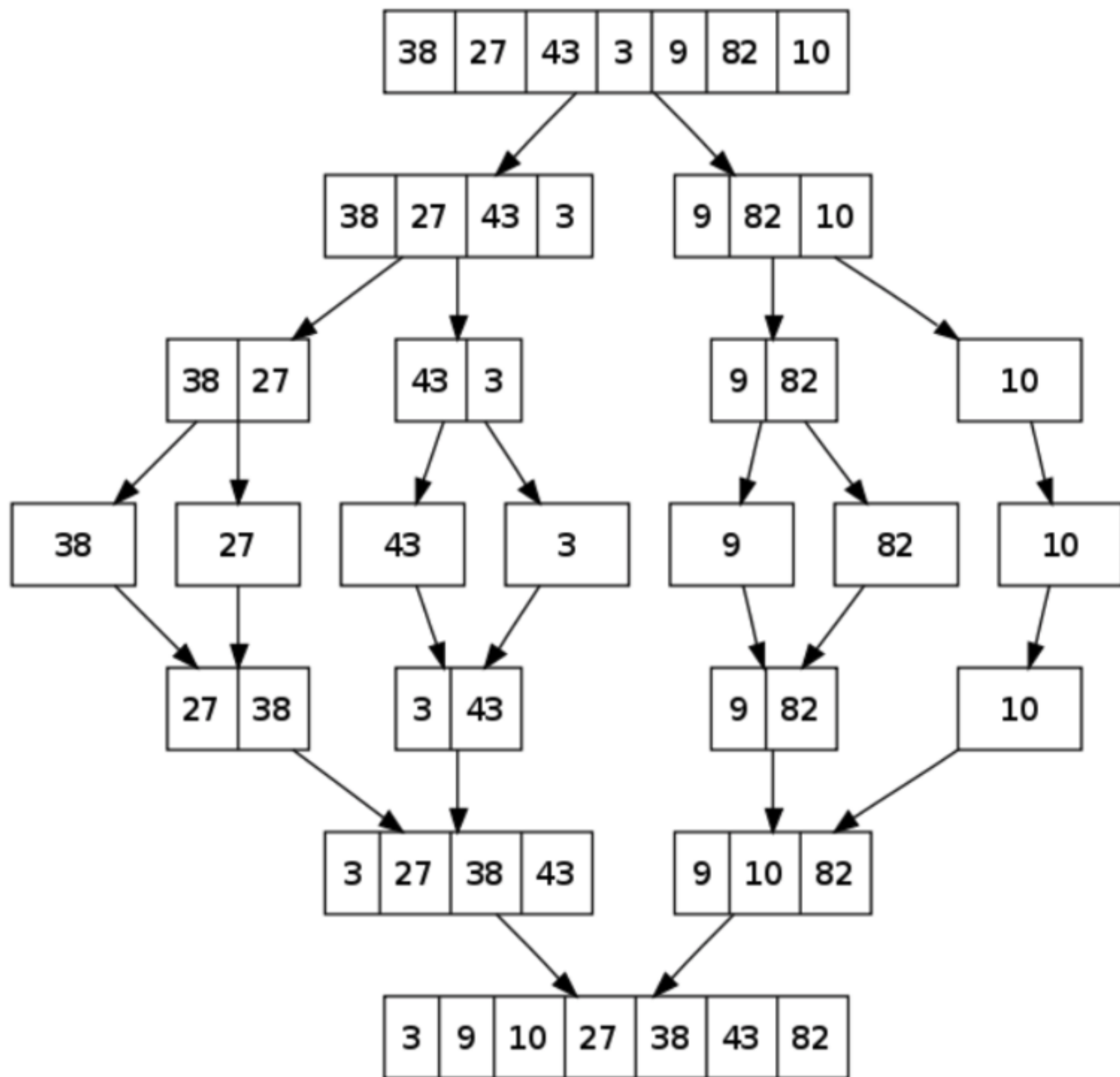


Quick Sort



Merge Sort

재귀함수를 통해 큰 데이터를 더 이상 나눌 수 없는 단위까지 잘게 쪼갬 후 다시 합치면서(merge) 정렬하는 방식
병합된 부분은 이미 정렬되어 있으므로 전부 비교하지 않아도 정렬 가능
분할 정복 방식 (Divide and Conquer) 의 대표적인 케이스 중 하나
데이터를 분해하고 합치는 작업에 필요한 (데이터 크기와 동일한) 추가 메모리 공간 필요
평균적으로 퀵 정렬에 비해 느리지만 퀵이나 힙과 달리 Stable 한 정렬이며 데이터 상태에 영향을 거의 받지 않음
시간복잡도 - 항상 $O(n \log n)$



Merge Sort

MERGE-SORT(A, p, r)

if $p < r$

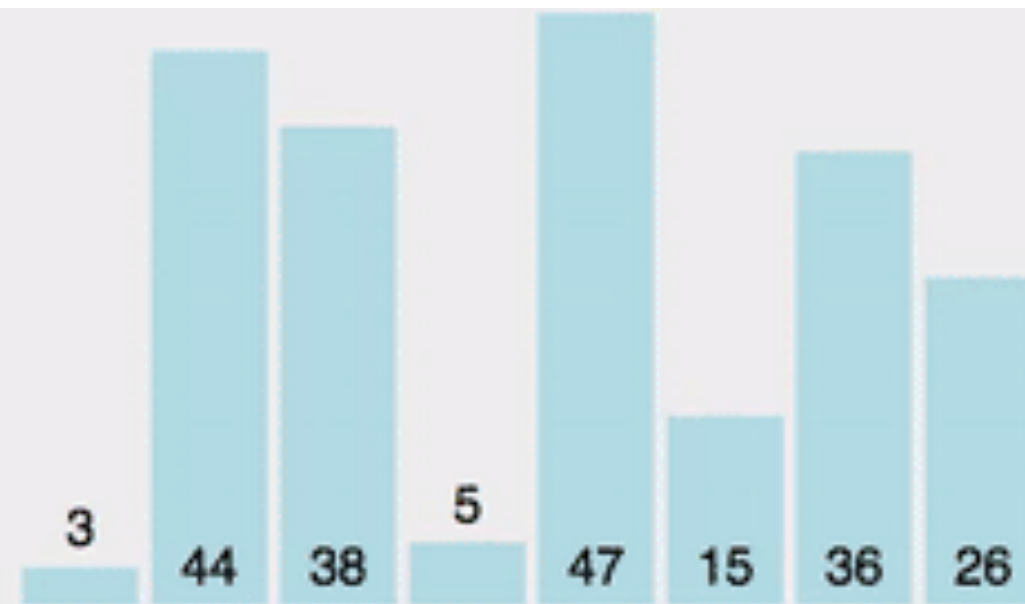
$q = \lfloor (p + r)/2 \rfloor$

 MERGE-SORT(A, p, q)

 MERGE-SORT($A, q + 1, r$)

 MERGE(A, p, q, r)

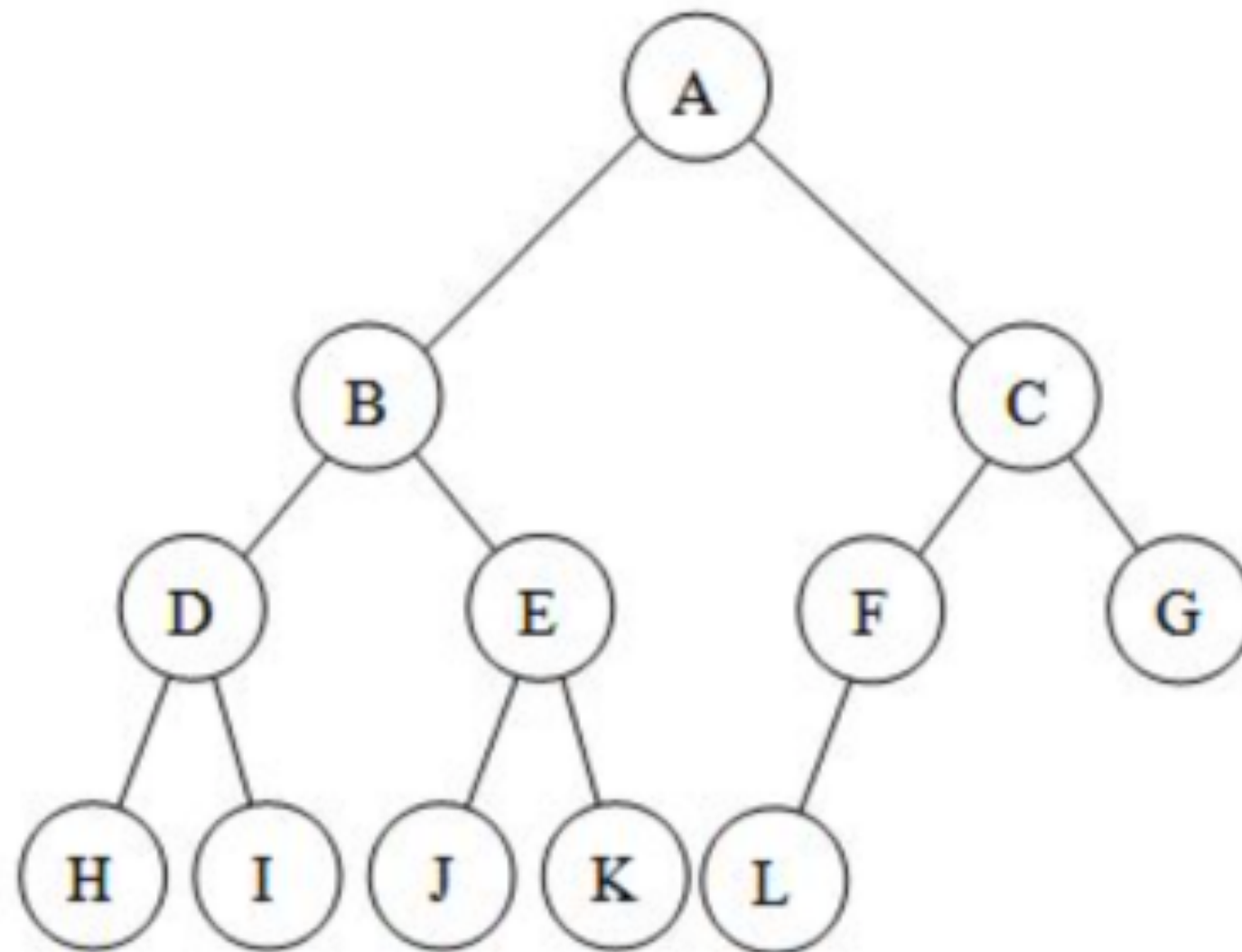
Merge Sort



Heap Sort

완전 이진 트리 (Complete Binary Tree)

- 각 노드의 자식 수가 2 이하면서 Root 노드부터 Leaf 노드까지 왼쪽부터 차례대로 채워져 있는 트리

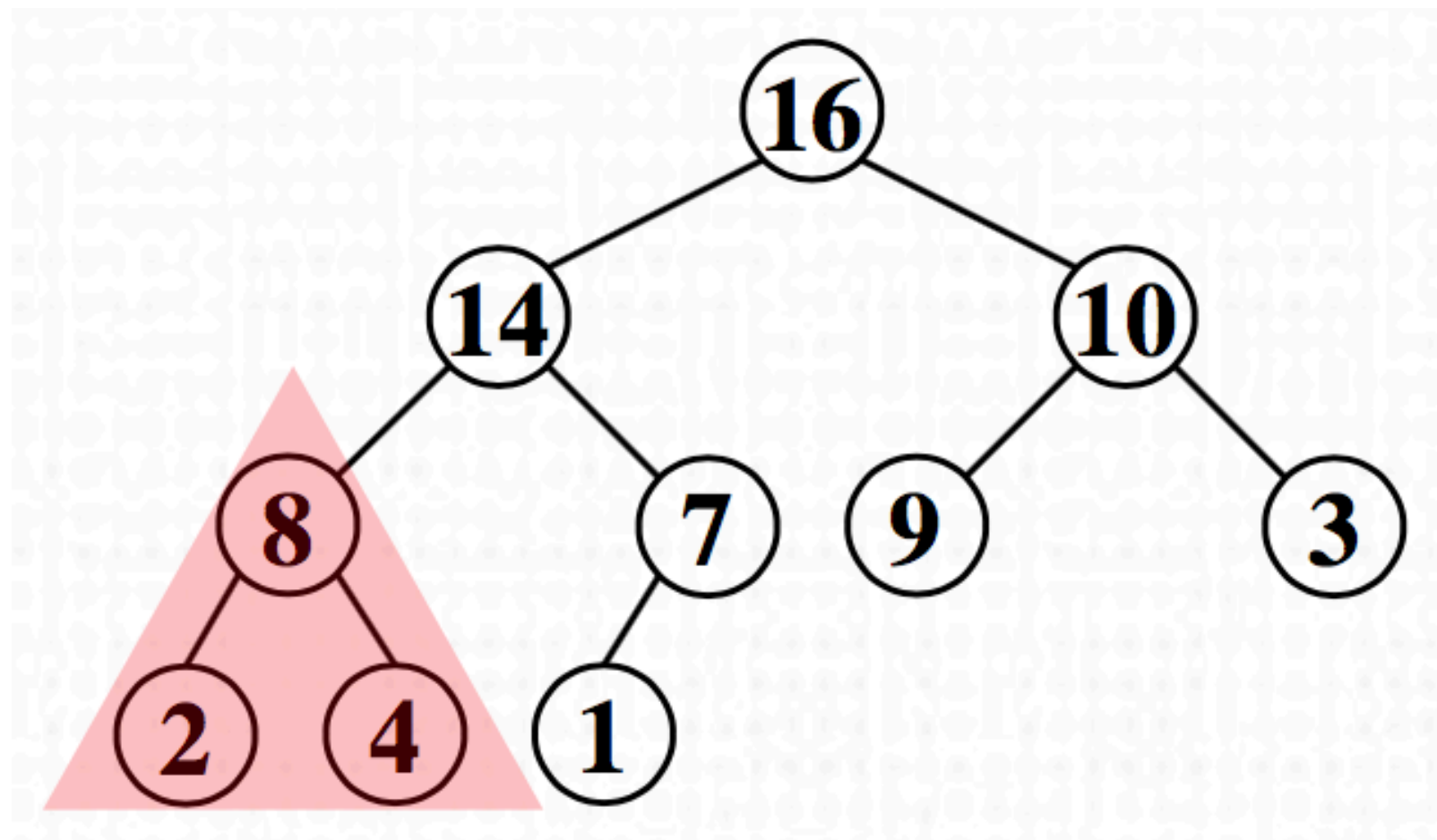


Heap Sort

완전 이진 트리에서 Max Heap, Min Heap 방식에 따라 저장한 뒤 하나씩 꺼내면서 정렬하는 방식

Max Heap의 특성

- 부모 노드의 값은 항상 자식 노드의 값보다 크다. 따라서 전체 트리의 Root 노드 값이 가장 크다
- 서브 트리 구조 각각이 모두 Max Heap의 특성을 따른다



Heap Sort

- **PARENT(i)**

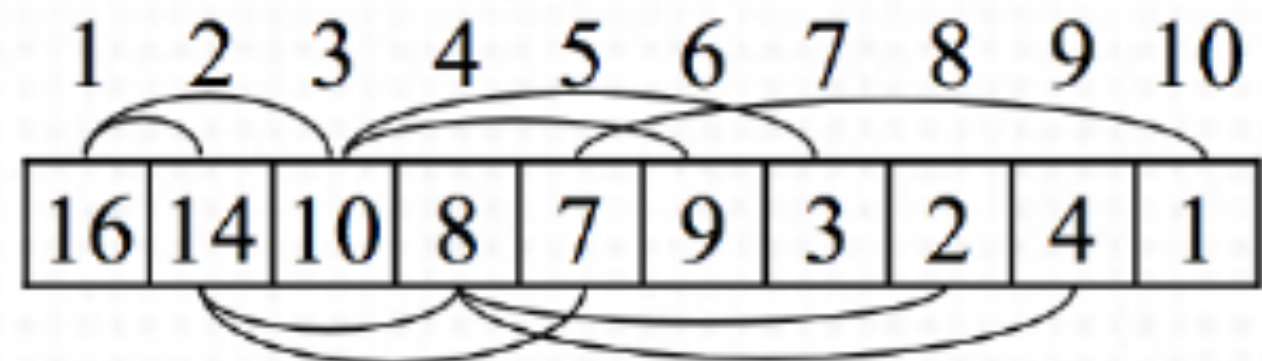
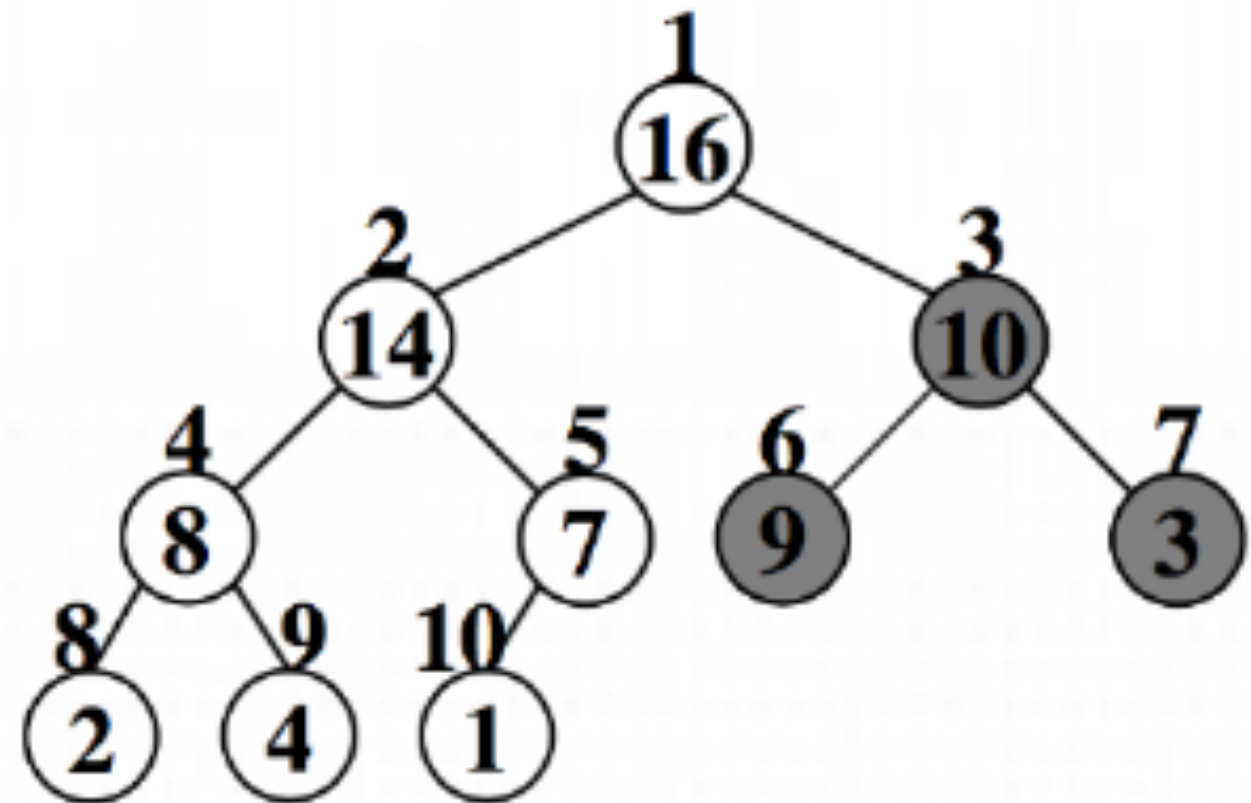
return $\left\lfloor \frac{i}{2} \right\rfloor$

- **LEFT(i)**

return $2i$

- **RIGHT(i)**

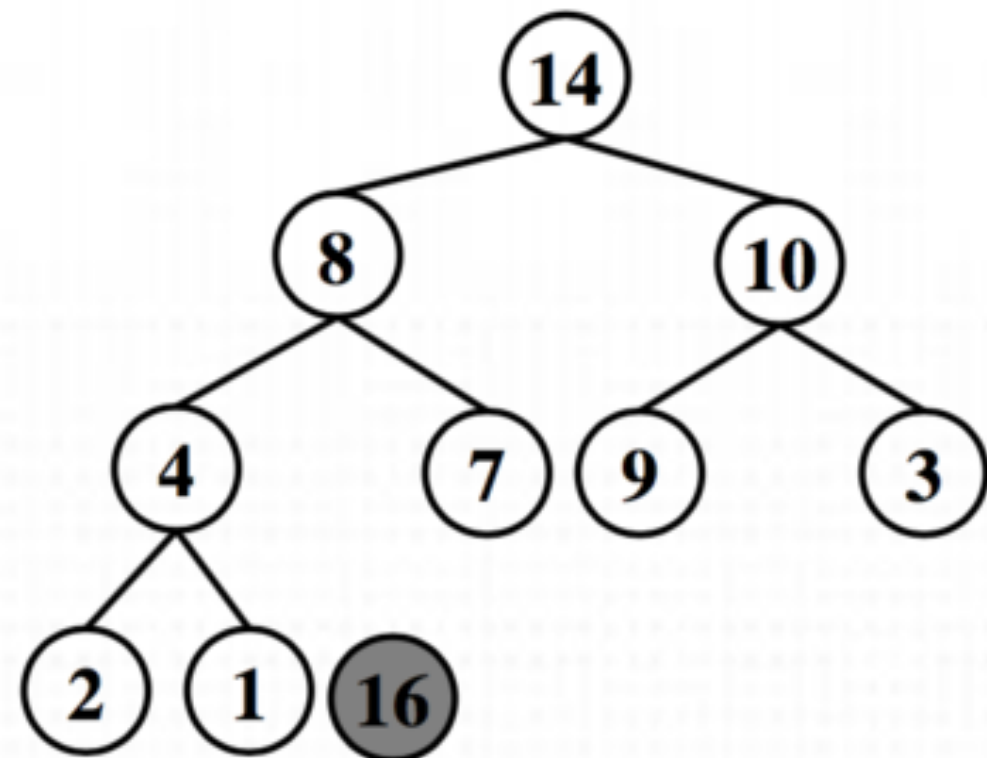
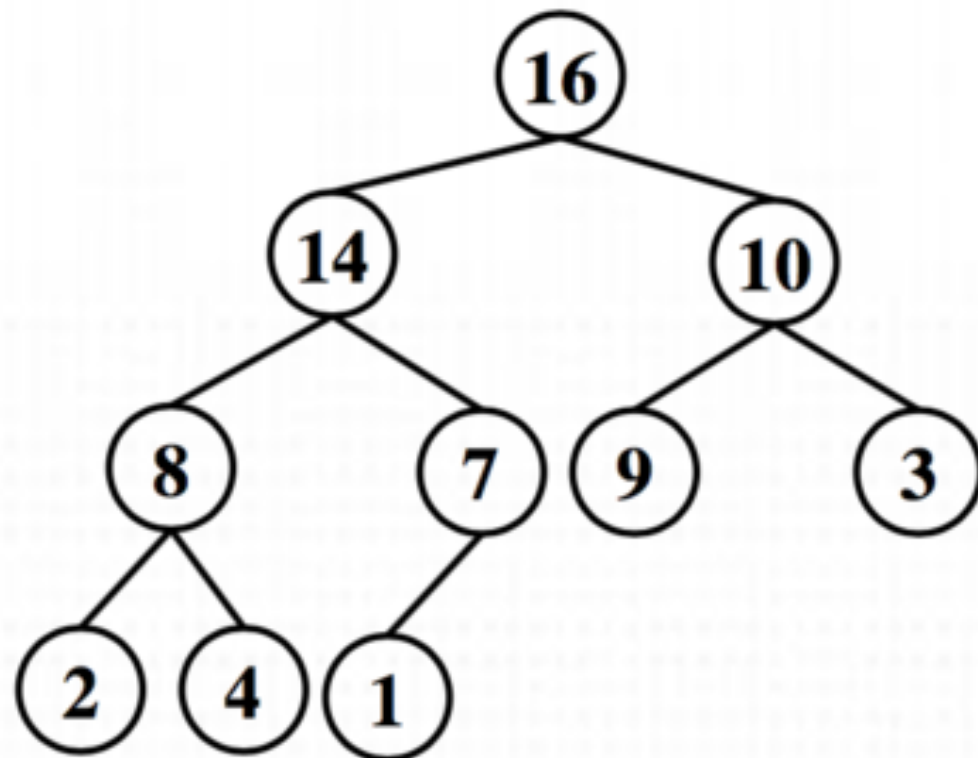
return $2i + 1$



Heap Sort

Max-Heap 구조를 이용한 Heap Sort 의 정렬 과정

1. 배열을 Max Heap 구조로 변환 (루트 노드가 최대값)
2. Root Node 를 배열의 마지막 Node와 교체한 후 해당 노드는 정렬된 값으로 간주하고
나머지 배열에 대해서만 (현재 배열 길이 -1) 1번 과정에 전달
3. 전체 데이터가 정렬될 때까지 1번, 2번 과정 반복



Heap Sort (Min Heap)

HEAPSORT(A)

BUILD-MAX-HEAP(A)

for $i = \lfloor A.length / 2 \rfloor$ downto 2

 exchange $A[1]$ with $A[i]$

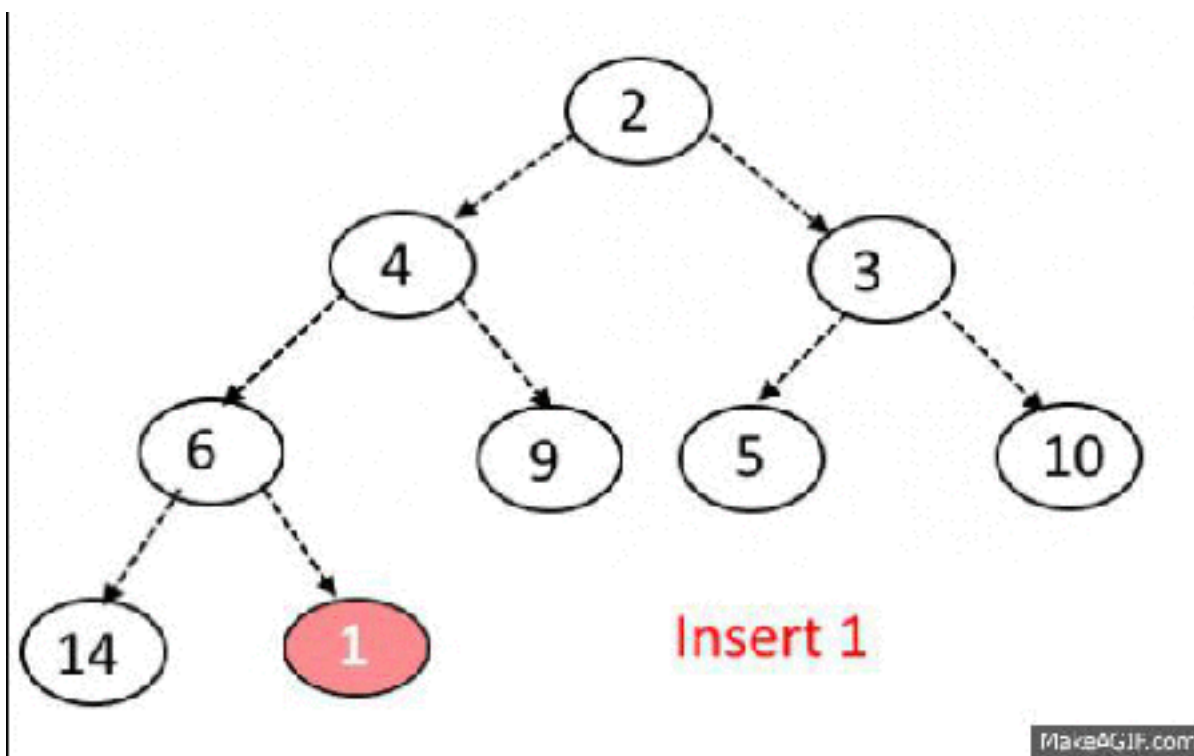
$A.heap-size = A.heap-size - 1$

 MAX-HEAPIFY(A, 1)

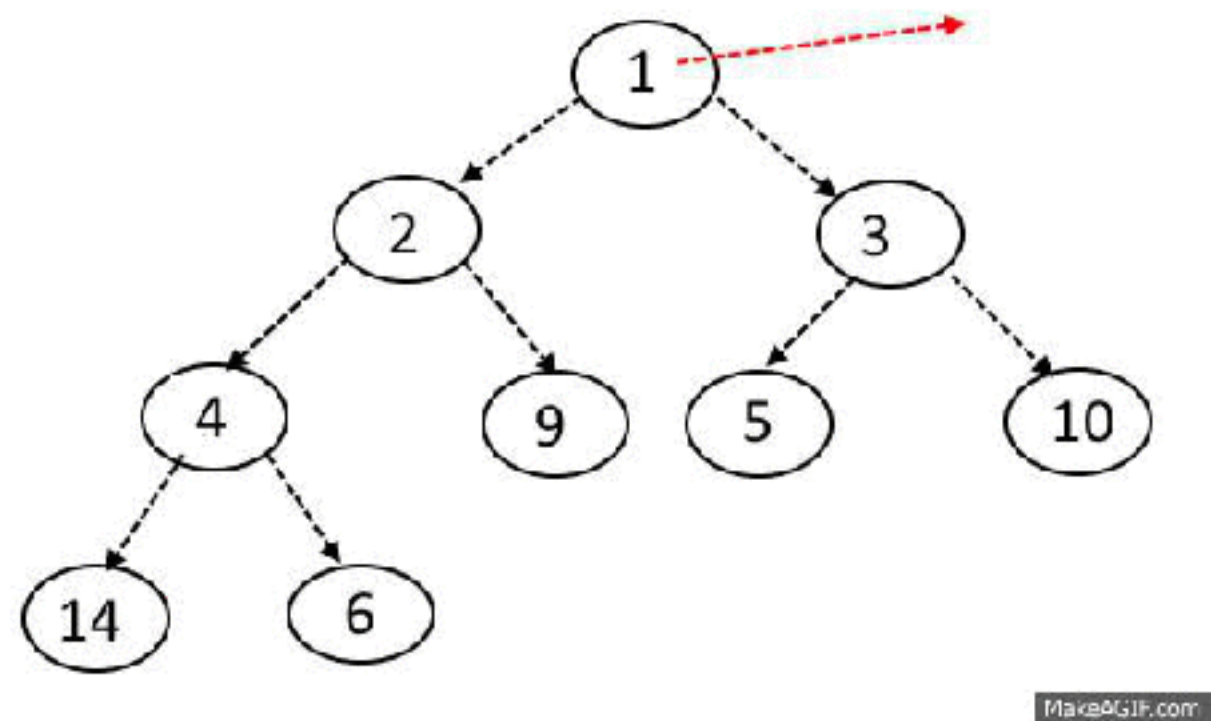
Heap Sort

10	4	8	5	12	2	6	11	3	9	7	1
----	---	---	---	----	---	---	----	---	---	---	---

Heap Sort (Min Heap)



[Build Min Heap]



[Extract & Down Heap]

Comparison

 Play All	 Insertion	 Selection	 Bubble	 Shell	 Merge	 Heap	 Quick	 Quick3
 Random								
 Nearly Sorted								
 Reversed								
 Few Unique								

Comparison

Name	Best	Average	Worst
Bubble Sort	n	n^2	n^2
Selection Sort	n^2	n^2	n^2
Insertion Sort	n	n^2	n^2
Quick Sort	$n \log n$	$n \log n$	n^2
Merge Sort	$n \log n$	$n \log n$	$n \log n$
Heap Sort	$n \log n$	$n \log n$	$n \log n$

Hybrid Sorting Algorithm

Python, Java - Timsort (Insertion Sort + Merge Sort)

Swift, C++ - Intro Sort (Insertion Sort + Quick Sort + Heap Sort)

- <https://github.com/apple/swift/blob/master/stdlib/public/core/Sort.swift.gyb>

```
// Insertion sort is better at handling smaller regions.
if elements.distance(from: range.lowerBound, to: range.upperBound) < 20 {
    ${try_} _insertionSort(
        &elements,
        subRange: range
        ${", by: areInIncreasingOrder" if p else ""})
    return
}
if depthLimit == 0 {
    ${try_} _heapSort(
        &elements,
        subRange: range
        ${", by: areInIncreasingOrder" if p else ""})
    return
}
```