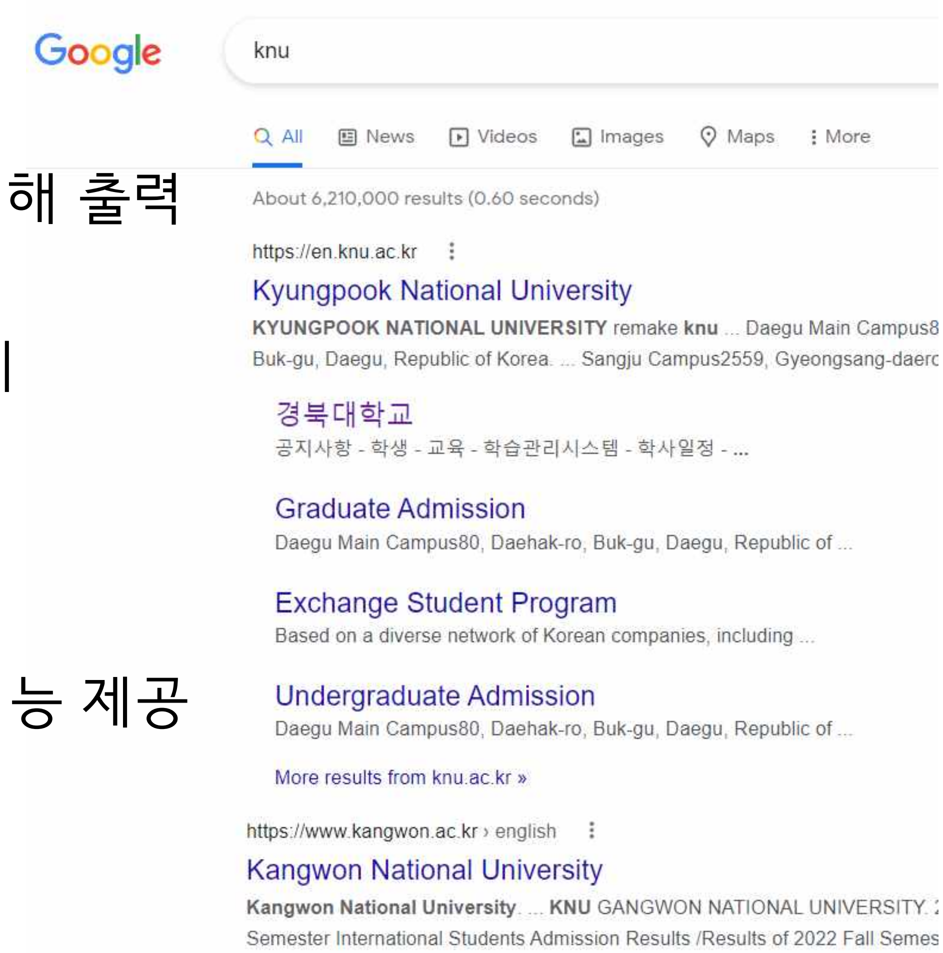
Sorting (Merge Sort, Quick Sort)

(1) 널리 활용되는 정렬 알고리즘으로부터 파생된 다양한/유용한 아이디어 보기
(2) 이러한 알고리즘을 기본 형태로부터 개선해가는 과정 보기

정렬은 다양한 application에서 필수적 역할 함

- 지원자, 인명 정보를 이름, 지역 순 정렬
- 음악 파일을 artist, 제목 순 정렬
- Google 검색 결과를 PageRank에 따라 정렬해 출력
- 온라인 뉴스 기사를 시간 역순으로 정렬
- 데이터의 중간값(median) 찾기, Outlier 찾기
- Mailing list에서 같은 주소 찾아 제거하기
- …



- 따라서, 대부분의 프로그래밍 언어는 정렬 기능 제공

프로그래밍 언어의 정렬 기능은 주로 Merge Sort/Quick Sort 기반

- 예: Java의 경우 primitive type (int, float, bool 등)에 대한 정렬은 Quick Sort로,
- 그 외 임의의 class type (object)에 대한 정렬은 Merge Sort로 구현됨
- 다양한 형태 입력 데이터에 대한 특성이 여러 각도로, 깊이 연구됨
- 거의 항상 좋은 성능 보장됨 보여짐

- **예습 자료**
- Merge Sort, Quick Sort의 기본 형태
- 성능 분석

현재 사용되는 Merge Sort와 Quick Sort는 기본 형태가 아니라 이러한 개선점 적용한 형태임

- **오늘 수업**
- Merge Sort, Quick Sort의 개선점
- 이들 활용한 정렬과 유사한 기능 (예: k번째로 큰 원소 찾기)

- **실습**
- 정렬 활용한 기능 구현 (Collinear Points)
- 정렬 사용하지 않는 경우와 비교

Sorting (Merge Sort, Quick Sort)
널리 활용되는 정렬 알고리즘을 기본 형태로부터 개선해가는 과정 보기

01. 퀴즈 풀이 & 예습 내용 복습 (이번 주 #1~3차 문제 공개)

02. Bottom-up Merge Sort: **Merge Sort에 대한 개선**

merge sort 에서 시작

03. Sorting Complexity

04. Stability of Sorting

05. Quick Select

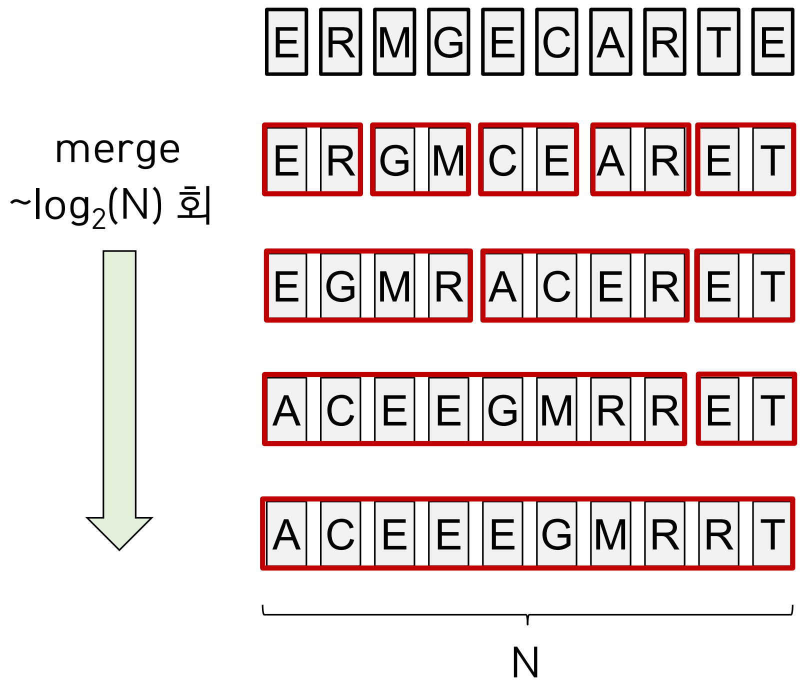
06. Duplicate Keys and 3-way Partitioning

07. 실습: Collinear Points 구현

Merge Sort: 인접한 두 조각끼리 Merge(정렬된 순서로 병합) 반복

- 총 N개 원소 병합한다면
- 이들을 순서에 맞게 차례로 결과 배열에 옮겨 담으므로
- ~N 회 작업 필요
- 이러한 작업을 ~log(N) 회 반복

- 입력 데이터 크기 N이라면
- 결과 옮겨 담을 ~N의 추가 공간 필요



입력 배열을 나누어 가는 과정에서 재귀 호출 계속됨 → 시간/공간 overhead 큼

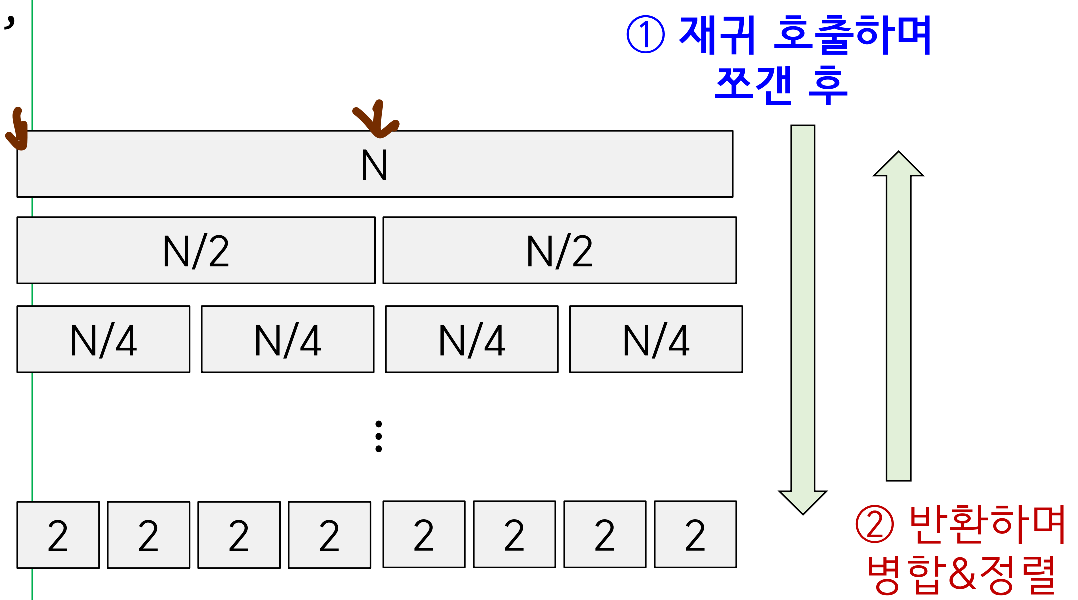
재귀호출 (쪼개기 위함)

```
# Halve a[lo ~ hi], sort each half and merge,
# using the extra space aux[]
def divideNMerge(a, aux, lo, hi):
    if (hi <= lo): return a
    mid = (lo + hi) // 2
    divideNMerge(a, aux, lo, mid)
    divideNMerge(a, aux, mid+1, hi)
    merge(a, aux, lo, mid, hi)

def mergeSort(a):
    # Create the auxiliary array once and
    # re-use for all subsequent merges
    aux = [None] * len(a)
    divideNMerge(a, aux, 0, len(a)-1)
```

①

②



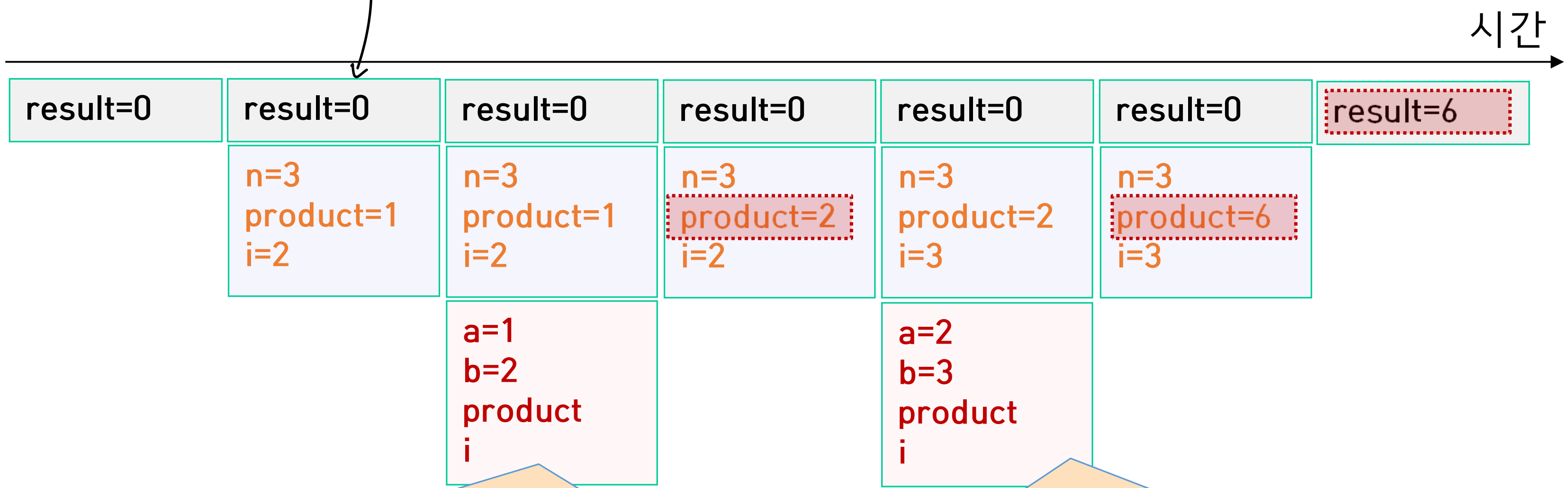
함수 호출/반환 과정에서 함수 인자, 지역변수 등을 메모리에 저장/삭제 반복 필요


```
void main() {
   int result=0;
   …
   result = result + factorial(3);
   …
}
```

```
int factorial(int n) {
   int product = 1;
   for (int i=2; i<=n; i++) {
      product = mult(product,i);
   }
   return product;
}
```

```
int mult(int a, int b) {
   int product = 0;
   for int(i=1; i<=b; i++) {
      product = product + a;
   }
   return product;
}
```

리턴주소



같은 이름의 변수라도 다른 함수에서 정의되었다면 다른 변수로 취급됨에 유의 (예: product, i)

product, i, a, b 등의 변수 값이 언제 변화되는지 잘 확인해 보세요.

쪼개는 걸 생략

Bottom-up Merge Sort

Divide 위해 재귀 호출하는 과정 생략, 아래와 같이 merge만 수행

sz = 1에서 시작

크기 sz인 인접한 부분집합끼리 병합

sz 2배 하고 sz>=N될 때까지 반복

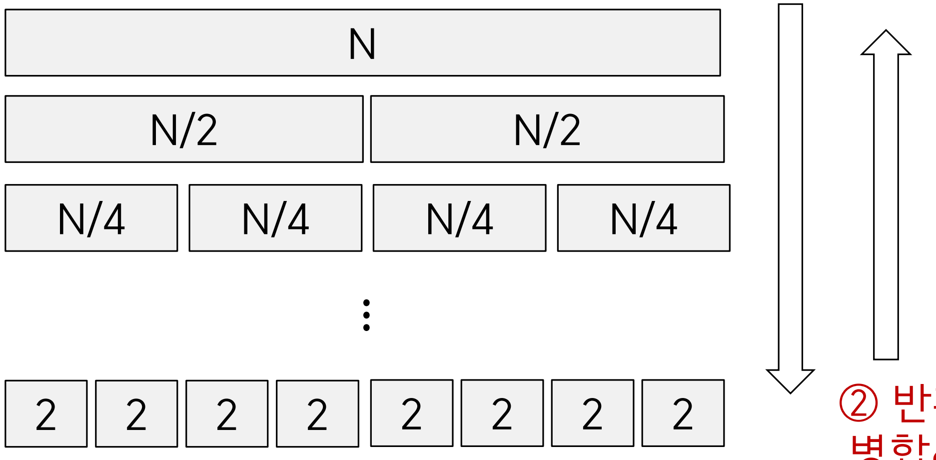
aux는 ~N 크기 추가공간

merge() 함수 인자	sz	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
merge(a, aux, lo, mid, hi)	입력	M	E	R	G	E	S	O	R	T	E	X	A	M	P	L	E
merge(a, aux, 0, 0, 1)	1	**E**	**M**	R	G	E	S	O	R	T	E	X	A	M	P	L	E
merge(a, aux, 2, 2, 3)		E	M	**G**	**R**	E	S	O	R	T	E	X	A	M	P	L	E
merge(a, aux, 4, 4, 5)		E	M	G	R	**E**	**S**	O	R	T	E	X	A	M	P	L	E
merge(a, aux, 6, 6, 7)		E	M	G	R	E	S	**O**	**R**	T	E	X	A	M	P	L	E
merge(a, aux, 8, 8, 9)		E	M	G	R	E	S	O	R	**E**	**T**	X	A	M	P	L	E
merge(a, aux, 10, 10, 11)		E	M	G	R	E	S	O	R	E	T	**A**	**X**	M	P	L	E
merge(a, aux, 12, 12, 13)		E	M	G	R	E	S	O	R	E	T	A	X	**M**	**P**	L	E
merge(a, aux, 14, 14, 15)		E	M	G	R	E	S	O	R	E	T	A	X	M	P	**E**	**L**
merge(a, aux, 0, 1, 3)	2	**E**	**G**	**M**	**R**	E	S	O	R	E	T	A	X	M	P	E	L
merge(a, aux, 4, 5, 7)		E	G	M	R	**E**	**O**	**R**	**S**	E	T	A	X	M	P	E	L
merge(a, aux, 8, 9, 11)		E	G	M	R	E	O	R	S	**A**	**E**	**T**	**X**	M	P	E	L
merge(a, aux, 12, 13, 15)		E	G	M	R	E	O	R	S	A	E	T	X	**E**	**L**	**M**	**P**
merge(a, aux, 0, 3, 7)	4	**E**	**E**	**G**	**M**	**O**	**R**	**R**	**S**	A	E	T	X	E	L	M	P
merge(a, aux, 8, 11, 15)		E	E	G	M	O	R	R	S	**A**	**E**	**E**	**L**	**M**	**P**	**T**	**X**
merge(a, aux, 0, 7, 15)	8	**A**	**E**	**E**	**E**	**E**	**G**	**L**	**M**	**M**	**O**	**P**	**R**	**R**	**S**	**T**	**X**

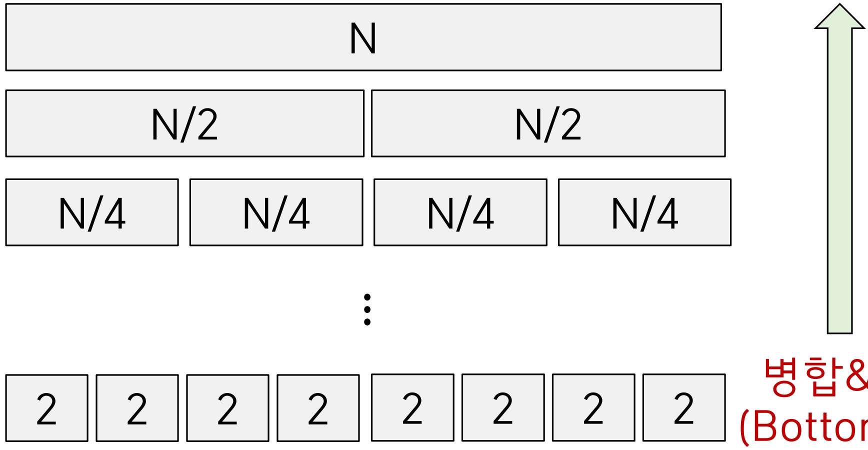
Merge Sort 기본 버전

Bottom-up Merge Sort

① 재귀 호출하며
쪼갠 후 (Top-down)



② 반환하며
병합&정렬
(Bottom-up)



병합&정렬
(Bottom-up)

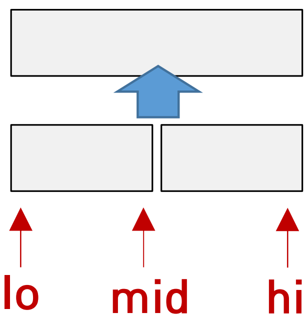
<merge 코드 - bottom-up version에서도 그대로 사용>

```
# Merge a[lo ~ mid] with a[mid+1 ~ hi],
# using the extra space aux[]
def merge(a, aux, lo, mid, hi):
    # Copy elements in a[] to aux[]
    for k in range(lo, hi+1):
        aux[k] = a[k]

    i, j = lo, mid+1
    for k in range(lo, hi+1):
        if i>mid: a[k], j = aux[j], j+1
        elif j>hi: a[k], i = aux[i], i+1
        elif aux[i] <= aux[j]: a[k], i = aux[i], i+1
        else: a[k], j = aux[j], j+1
```

재귀호출파가 없어짐.

정렬된 조각 1(a[lo]~a[mid])과
정렬된 조각 2(a[mid+1]~a[hi]) 병합
aux는 ~N 크기 추가공간



<재귀호출하는 Top-down version>

```
# Halve a[lo ~ hi],
# sort each of the halves,
# and merge them,
# using the extra space aux[]
def divideNMerge(a, aux, lo, hi):
    if (hi <= lo): return a
    mid = (lo + hi) // 2
    divideNMerge(a, aux, lo, mid)
    divideNMerge(a, aux, mid+1, hi)
    merge(a, aux, lo, mid, hi)

def mergeSort(a):
    # Create the auxiliary array once
    # re-use for all subsequent merges
    aux = [None] * len(a)
    divideNMerge(a, aux, 0, len(a)-1)
```

```
def mergeSort(a):
    # Create the auxiliary array
    aux = [None] * len(a)  ↪ size 결정

    sz = 1
    while(sz<len(a)):
        for lo in range(0, len(a)-sz, sz*2):
            merge(a, aux, lo, lo+sz-1, min(lo+sz+sz-1, len(a)-1))
        sz += sz  # Multiply by 2
```

[Q] for (lo=0; lo<len(a)-sz; lo += sz*2)
에서 lo<len(a)-sz 한 이유?

aux는 ~N 크기 추가공간

[Q] merge함수 마지막 인자에 min() 함수 사용한 이유?

2

merge() 함수 인자	sz	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
merge(a, aux, lo, mid, hi)	입력	M	E	R	G	E	S	O	R	T	E	X	A	M	P	L	E
merge(a, aux, 0, 0, 1)	1	E	M	R	G	E	S	O	R	T	E	X	A	M	P	L	E
merge(a, aux, 2, 2, 3)		E	M	G	R	E	S	O	R	T	E	X	A	M	P	L	E
merge(a, aux, 4, 4, 5)		E	M	G	R	E	S	O	R	T	E	X	A	M	P	L	E
merge(a, aux, 6, 6, 7)		E	M	G	R	E	S	O	R	T	E	X	A	M	P	L	E
merge(a, aux, 8, 8, 9)		E	M	G	R	E	S	O	R	E	T	X	A	M	P	L	E
merge(a, aux, 10, 10, 11)		E	M	G	R	E	S	O	R	E	T	A	X	M	P	L	E
merge(a, aux, 12, 12, 13)		E	M	G	R	E	S	O	R	E	T	A	X	M	P	L	E
merge(a, aux, 14, 14, 15)		E	M	G	R	E	S	O	R	E	T	A	X	M	P	E	L
merge(a, aux, 0, 1, 3)	2	E	G	M	R	E	S	O	R	E	T	A	X	M	P	E	L
merge(a, aux, 4, 5, 7)		E	G	M	R	E	O	R	S	E	T	A	X	M	P	E	L
merge(a, aux, 8, 9, 11)		E	G	M	R	E	O	R	S	A	E	T	X	M	P	E	L
merge(a, aux, 12, 13, 15)		E	G	M	R	E	O	R	S	A	E	T	X	E	L	M	P
merge(a, aux, 0, 3, 7)	4	E	E	G	M	O	R	R	S	A	E	T	X	E	L	M	P
merge(a, aux, 8, 11, 15)		E	E	G	M	O	R	R	S	A	E	E	L	M	P	T	X
merge(a, aux, 0, 7, 15)	8	A	E	E	E	E	G	L	M	M	O	P	R	R	S	T	X

1

↪ 만약 size가 13이라면?

8

Sorting (Merge Sort, Quick Sort)

널리 활용되는 정렬 알고리즘을 기본 형태로부터 개선해가는 과정 보기

01. 퀴즈 풀이 & 예습 내용 복습 (이번 주 #1~3차 문제 공개)
02. Bottom-up Merge Sort
03. Sorting Complexity: **정렬 알고리즘은 얼마나 빨라질 수 있는가?**
04. Stability of Sorting
05. Quick Select
06. Duplicate Keys and 3-way Partitioning
07. 실습: Collinear Points 구현

정렬 알고리즘은 어느 정도까지 빠를 수 있을까?

- 왜 이런 분석 하는가? 최적의 알고리즘을 이미 찾았는지, 혹은 **더 개선할 여지가 알기 위함**
- Merge Sort, Quick Sort는 이러한 최적의 속도에 다다랐는가?

$\sim N \log N$

정렬 알고리즘은 어느 정도까지 빠를 수 있을까? 가정

← 가장 약한 상태①

- N개의 **서로 다른 원소**가 있고, 이들의 **대소 관계 정해야 한다**고 가정 (예: [a, b, c] 주어졌다면, a<b<c, a<c<b, b<a<c, b<c<a, c<a<b, c<b<a 중 하나 가능)
- 같은 원소 많은 경우는 더 빠르게 정렬 가능 (예: 0/1 두 값만 있는 경우, 3-way Partitioning)

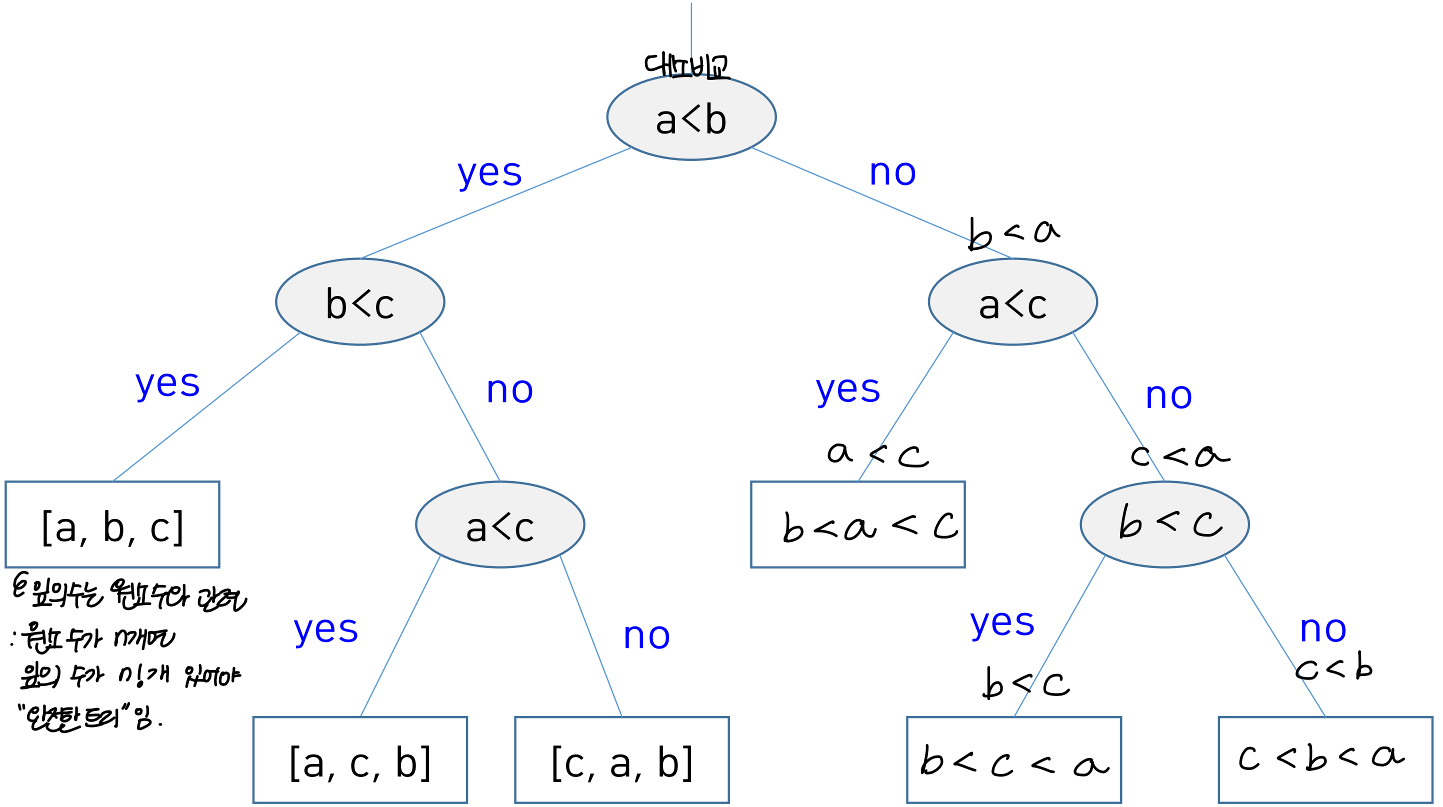
← 가장 약한 상태②

- **원소 간 대소 관계 전혀 모르는 상태에서 시작**한다고 가정
- 일부 혹은 전체 원소 간 대소 관계를 미리 안다면 더 빠르게 정렬 가능 (예: 입력이 모두 정렬되었음을 알고 있다면 정렬 안 해도 됨)

- 원소 간 대소 관계 정하기 위해서는 '대소 비교' 해야 하며
- k개 원소 간 대소 비교는 결국 2개 원소 간 대소 비교로 분해되므로
- '**2개 원소 간 대소 비교 횟수**'를 세어 보겠음
- 대소 관계 정한 후 메모리에서 값을 이동하는 작업 필요하나, 이는 대소 비교 횟수에 비례

대소 비교 결과로 나올 수 있는 모든 가능한 경우를 트리 형태(Decision Tree)로 그려 보기
N=3 인 경우

e a, b, c



e 잎의 수는 원소 수와 관련
: 원소 수가 n개면
잎의 수가 n!개 있어야
"완전한 트리"임.

잎(=자식이 없는 끝노드)에 와야
대소 관계를 알게 됨.

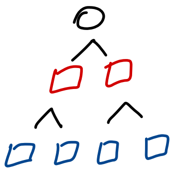
[Q] Tree ~~깊이~~depth가 나타내는 것은? 비교횟수

각 가능한 정렬이 잎(leaf, 자식이 없는 노드)에 최소한 한 번은 나와야 함

[Q] why?

대소 비교 결과로 나올 수 있는 모든 가능한 경우를 트리 형태(Decision Tree)로 그려 보기
임의의 N에 대해 일반화

[Q] 깊이 h인 이진 트리에는 (한 번 분기했을 때 깊이가 1이 된다고 가정) 잎이 최대 몇 개인가?



깊이	잎 개수
1	2
2	2^2

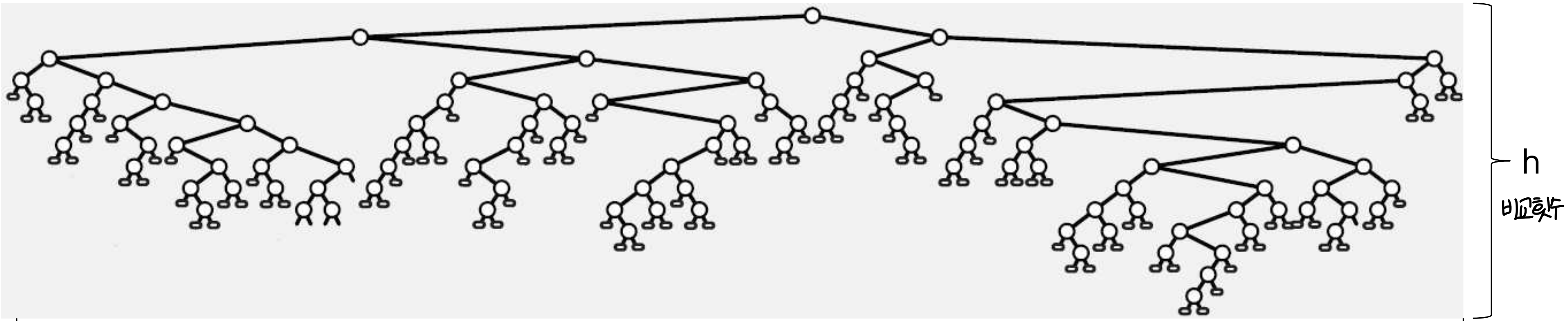
⋮

잎 개수 $\leq 2^h$

비교가 빨리 끝나는 경우도 있기 때문

정렬해야 하는 원소 수

[Q] N개 원소에 대한 모든 가능한 정렬이 잎에 한 번은 나오려면, 잎이 최소 몇 개 있어야 하나? N!



각 가능한 정렬이 잎에 최소 한 번은 나와야 함

대소 비교 결과로 나올 수 있는 모든 가능한 경우를 트리 형태(Decision Tree)로 그려 보기
임의의 N에 대해 일반화

[Q] 앞 페이지 두 문제의 답으로부터 N(정렬할 원소 개수)과 h(worst-case 비교 횟수) 간 관계를 구하시오.

$$N! < 2^h$$

$$\boxed{\log_2 N!} < h$$ (비교횟수)

$$\frac{N}{2}\log\frac{N}{2} \le \log N! \le N\log N$$

$$\log N + \log(N-1) + \log(N-2) + \cdots + \log 1$$

$$\le \log N + \log N + \log N + \cdots + \log N$$

Stirling's approximation:
log(N!) = log(N×(N−1)×⋯×2×1) = log(N) + log(N−1) + ⋯ + log2 + log1 = $\sim\int_1^n bgx \;\; dx$ 임 활용해
log(N!) = ~Nlog(N) 임 보임
https://en.wikipedia.org/wiki/Stirling%27s_approximation

- N개의 서로 다른 원소 있는 경우, 최적의 정렬 방법은 ~N log(N)회 대소 비교 필요

- Merge Sort (worst case), Quick Sort (average case)는 이러한 최저치에 가까움

[Q] Quick Sort의 Worst Case는?

① 모든 구간에서 기준값보다 큰 원소만 존재하는 경우

② " 작은 "

- 하지만 계속해서 개선되었으며, 아직 개선 여지 있음. Why?
- (1) 서로 같은 원소 있는 경우도 있음 (예: 3-way partition)
- (2) 정렬 외 다른 만족시켜야 할 성질이 존재하기도 함 (예: stability, 추가 공간 필요 여부)
- …

Sorting (Merge Sort, Quick Sort)

널리 활용되는 정렬 알고리즘을 기본 형태로부터 개선해가는 과정 보기

01. 퀴즈 풀이 & 예습 내용 복습 (이번 주 #1~3차 문제 공개)
02. Bottom-up Merge Sort
03. Sorting Complexity
04. Stability of Sorting: **정렬 시 추가로 만족시켜야 할 성질**
05. Quick Select
06. Duplicate Keys and 3-way Partitioning
07. 실습: Collinear Points 구현

ε 여러 값으로 이뤄진 원소일 때 stability 중요

Stability of Sorting

- key: 여러 값으로 이루어진 원소의 경우 (예: class 객체) 정렬에 사용하는 값
- 정렬 전후 key가 같은 원소 간 상대 순서 보존하는 성질

이름순

```
sorted(a, key=lambda x:x[0])
```

이름	학년	성적	전화번호	주소
Andrews	3	A	664-480-0023	097 Little
Battle	4	C	874-088-1212	121 Whiteman
Chen	3	A	991-878-4944	308 Blair
Fox	3	A	884-232-5341	11 Dickinson
Furia	1	A	766-093-9873	101 Brown
Gazsi	4	B	766-093-9873	101 Brown
Kanaga	3	B	898-122-9643	22 Brown
Rohde	2	A	232-343-5555	343 Forbes

학년순

```
sorted(a, key=lambda x:x[1])
```

이름	학년	성적	전화번호	주소
Furia	1	A	766-093-9873	101 Brown
Rohde	2	A	232-343-5555	343 Forbes
Andrews	3	A	664-480-0023	097 Little
Chen	3	A	991-878-4944	308 Blair
Fox	3	A	884-232-5341	11 Dickinson
Kanaga	3	B	898-122-9643	22 Brown
Battle	4	C	874-088-1212	121 Whiteman
Gazsi	4	B	766-093-9873	101 Brown

알파벳순

알파벳순

Stable한 정렬 결과

∵ 앞에 정렬한 게 남아있다
(학년순 → 알파벳순)

Stability of Sorting

- key: **여러 값으로 이루어진 원소**의 경우 (예: class 객체) 정렬에 사용하는 값
- 정렬 전후 **key가 같은 원소 간 상대 순서 보존**하는 성질

`sorted(a, key=lambda x:x[0])`

이름	학년	성적	전화번호	주소
Andrews	3	A	664-480-0023	097 Little
Battle	4	C	874-088-1212	121 Whiteman
Chen	3	A	991-878-4944	308 Blair
Fox	3	A	884-232-5341	11 Dickinson
Furia	1	A	766-093-9873	101 Brown
Gazsi	4	B	766-093-9873	101 Brown
Kanaga	3	B	898-122-9643	22 Brown
Rohde	2	A	232-343-5555	343 Forbes

`sorted(a, key=lambda x:x[1])`

이름	학년	성적	전화번호	주소
Furia	1	A	766-093-9873	101 Brown
Rohde	2	A	232-343-5555	343 Forbes
Chen	3	A	991-878-4944	308 Blair
Fox	3	A	884-232-5341	11 Dickinson
Andrews	3	A	664-480-0023	097 Little
Kanaga	3	B	898-122-9643	22 Brown
Gazsi	4	B	766-093-9873	101 Brown
Battle	4	C	874-088-1212	121 Whiteman

Stable하지 **않은** 정렬 결과

Stable한 정렬 방법이 필요한 예 (비행기/기차 목적지/출발 시간)

- Stable한 정렬 방법: 정렬 전후에 **key가 같은 원소 간 상대 순서 보존**하는 정렬 방식
- key: 여러 값으로 이루어진 원소의 경우 (예: class 객체) 정렬에 사용하는 값

시간 순으로 정렬 → 장소 순으로 정렬

시간 순으로 정렬

Chicago	09:00:00
Phoenix	09:00:03
Houston	09:00:13
Chicago	09:00:59
Houston	09:01:10
Chicago	09:03:13
Seattle	09:10:11
Seattle	09:10:25
Phoenix	09:14:25
Chicago	09:19:32
Chicago	09:19:46
Chicago	09:21:05
Seattle	09:22:43
Seattle	09:22:54
Chicago	09:25:52
Chicago	09:35:21
Seattle	09:36:14
Phoenix	09:37:44

장소 순으로 정렬

(1)

Chicago	09:25:52
Chicago	09:03:13
Chicago	09:21:05
Chicago	09:19:46
Chicago	09:19:32
Chicago	09:00:00
Chicago	09:35:21
Chicago	09:00:59
Houston	09:01:10
Houston	09:00:13
Phoenix	09:37:44
Phoenix	09:00:03
Phoenix	09:14:25
Seattle	09:10:25
Seattle	09:36:14
Seattle	09:22:43
Seattle	09:10:11
Seattle	09:22:54

(2)

Chicago	09:00:00
Chicago	09:00:59
Chicago	09:03:13
Chicago	09:19:32
Chicago	09:19:46
Chicago	09:21:05
Chicago	09:25:52
Chicago	09:35:21
Houston	09:00:13
Houston	09:01:10
Phoenix	09:00:03
Phoenix	09:14:25
Phoenix	09:37:44
Seattle	09:10:11
Seattle	09:10:25
Seattle	09:22:43
Seattle	09:22:54
Seattle	09:36:14

[Q] (1)과 (2) 중 어느 쪽이 stable한 방식으로 정렬되었나?

(2)

Stable한 정렬 방법이 필요한 예 (iTunes)

- Stable한 정렬 방법: 정렬 전후에 **key가 같은 원소 간 상대 순서 보존**하는 정렬 방식
- key: 여러 값으로 이루어진 원소의 경우 (예: class 객체) 정렬에 사용하는 값

	✓	Name		Time	Artist	Track No
1	✓	Age Of Loneliness (Enigmatic Clu...		6:20	Enigma	3 of
2	✓	Arriving		4:08	Kammarheit	1 of
3	✓	Bip Bop Bip		1:56	Don Covay	1 of 2
4	✓	Breakthrough		4:52	Madhavi Devi	4 of
5	✓	Come On		4:09	Itchy & Scratchy	5 of 1
6	✓	Daddy Loves Baby		2:22	Don Covay	7 of 2
7	✓	Eastern Vibes (Original Mix)		7:32	Mathar	12 of 1
8	✓	Everybody Party		5:06	D. Enrico	14 of 1
9	✓	Everybody Shake Your Body		5:37	Maltese Massive	8 of 1
10	✓	Falling From The Sky Like A Flock...		11:40	Jasper TX	7 of
11	✓	Feel So High (Freaky Frenzy Mix)		5:00	Sound Environment	7 of 1

노래 **제목** 순으로 정렬한 후 **Artist 순**으로 정렬할 때,
Artist가 같은 노래 간에는 제목 순서로 정렬되어 있기를 원함

Insertion Sort는 Stable함: Key 같은 원소 간 순서 보존

- Iteration i에
  - 이미 정렬된 a[0] ~ a[i-1]에
  - a[i]를 적절한 위치 (정렬되었을 때의 위치) 찾아 추가
  - 그 결과 a[0] ~ a[i]까지 정렬된 상태 됨

E1, E2, E3 모두 같은 key ‘E’인데, 상대 순서 보여주기 위해 숫자 1, 2, 3 추가해 썼음 (key 이외 값은 생략)

Key 같은 원소 (E1, E2, E3) 간 상대 위치 변하지 않음 (key 같은 원소 넘어 이동하지 않음)

- 오름차순 정렬 예
- (‘ ’는 정렬된 부분, ‘**O**’는 새로 insert된 원소)
- 다걍 E임
- X L E1 E2 A T S R E3 M O
- **L** X E1 E2 A T S R E3 M O
- **E1** L X E2 A T S R E3 M O
- E1 **E2** L X A T S R E3 M O
- **A** E1 E2 L X T S R E3 M O
- A E1 E2 L **T** X S R E3 M O
- A E1 E2 L **S** T X R E3 M O
- A E1 E2 L **R** S T X E3 M O
- A E1 E2 **E3** L R S T X M O
- A E1 E2 E3 L **M** R S T X O
- A E1 E2 E3 L M **O** R S T X

Insertion Sort는 Stable함: Key 같은 원소 간 순서 보존

- Iteration i에
  - 이미 정렬된 a[0] ~ a[i-1]에
  - a[i]를 적절한 위치 (정렬되었을 때의 위치) 찾아 추가
  - 그 결과 a[0] ~ a[i]까지 정렬된 상태 됨

```
def insertionSort(a):
    for i in range(1, len(a)):
        key = a[i]
        j = i-1
        while j>=0 and a[j] > key:
            a[j+1] = a[j]
            j -= 1
        a[j+1] = key
```

[Q] a[j] **>=** key로 수정해도 stable한가?

NO..

- 오름차순 정렬 예
- (' '는 정렬된 부분, '**O**'는 새로 insert된 원소)
- X L E1 E2 A T S R E3 M O
- **L** X E1 E2 A T S R E3 M O
- **E1** L X E2 A T S R E3 M O
- E1 **E2** L X A T S R E3 M O
- **A** E1 E2 L X T S R E3 M O
- A E1 E2 L **T** X S R E3 M O
- A E1 E2 L **S** T X R E3 M O
- A E1 E2 L **R** S T X E3 M O
- A E1 E2 **E3** L R S T X M O
- A E1 E2 E3 L **M** R S T X O
- A E1 E2 E3 L M **O** R S T X

Selection Sort는 Stable하지 않음

- Iteration i에
  - a[i] ~ a[N-1] 중 최솟값 찾아 a[i]와 swap

```
def selectionSort(a):
    for i in range(len(a)-1):
        # Find the minimum in a[i]~a[N-1]
        min_idx = i
        for j in range(i+1, len(a)):
            if a[j] < a[min_idx]:
                min_idx = j

        # Swap the found minimum with a[i]
        a[i], a[min_idx] = a[min_idx], a[i]
```

[Q] Selection sort 코드의 어느 부분 때문에 stable하지 않은가?

'원거리' swap을 하기 때문

→ 중간에 같은 원소를 swap할때 이상

- 오름차순 정렬 예
- ('O'는 선택 범위, '**O**'는 선택된 최솟값)
- A1 B1 B2 A2
- **A1** B1 B2 A2
- A1 **A2** B2 B1
- A1 A2 **B2** B1
- A1 A2 B2 **B1**

Key 같은 원소 (B1, B2) 간 위치 변함
(key 같은 원소 넘어 이동 가능)

Shell Sort: h를 점진적으로 1까지 감소시켜 가며 h-sort 수행

input

S	O	R	T	E	X	A	M	P	L	E

7-sort : 원거리 swap : ~~Stable~~

S	O	R	T	E	X	A	M	P	L	E
M	O	R	T	E	X	A	S	P	L	E
M	O	R	T	E	X	A	S	P	L	E
M	O	L	T	E	X	A	S	P	R	E
M	O	L	E	E	X	A	S	P	R	T

3-sort

M	O	L	E	E	X	A	S	P	R	T
E	O	L	M	E	X	A	S	P	R	T
E	E	L	M	O	X	A	S	P	R	T
E	E	L	M	O	X	A	S	P	R	T
A	E	L	E	O	X	M	S	P	R	T
A	E	L	E	O	X	M	S	P	R	T
A	E	L	E	O	P	M	S	X	R	T
A	E	L	E	O	P	M	S	X	R	T
A	E	L	E	O	P	M	S	X	R	T

1-sort

A	E	L	E	O	P	M	S	X	R	T
A	E	L	E	O	P	M	S	X	R	T
A	E	L	E	O	P	M	S	X	R	T
A	E	E	L	O	P	M	S	X	R	T
A	E	E	L	O	P	M	S	X	R	T
A	E	E	L	O	P	M	S	X	R	T
A	E	E	L	M	O	P	S	X	R	T
A	E	E	L	M	O	P	S	X	R	T
A	E	E	L	M	O	P	S	X	R	T
A	E	E	L	M	O	P	R	S	X	T
A	E	E	L	M	O	P	R	S	T	X

result

A	E	E	L	M	O	P	R	S	T	X

[Q] Shell Sort는 stable할까? NO

h-sort

Shell Sort: Stable하지 않음. Why?

- 4-sort 수행 예
- ('O'는 이동한 값)
- C1 B1 B2 B3 B4
- B4 B1 B2 B3 C1
- …

- 4-sort 수행 예
- ('O'는 이동한 값)
- B1 B2 B3 B4 A1
- A1 B2 B3 B4 B1
- …

Merge Sort: 크기 1인 조각에서 시작 → 인접한 조각끼리 merge 반복

- Merge Sort 수행 예
- 붉은 사각형은 merge한 조각 의미
- E1 R1 M G E2 C A R2 T E3
- E1 R1 | G M | C E2 | A R2 | E3 T
- ① E1 G M R1 | ② A C E2 R2 | E3 T
- A C E1 E2 G M R1 R2 | E3 T
- A C E1 E2 E3 G M R1 R2 T

작은애 먼저 드감.
&
같은 값이면 왼쪽 것을 먼저 넣으므로 stable

```python
# Merge a[lo ~ mid] with a[mid+1 ~ hi],
# using the extra space aux[]
def merge(a, aux, lo, mid, hi):
    # Copy elements in a[] to aux[]
    for k in range(lo, hi+1):
        aux[k] = a[k]

    i, j = lo, mid+1
    for k in range(lo, hi+1):
        if i>mid: a[k], j = aux[j], j+1
        elif j>hi: a[k], i = aux[i], i+1
        elif aux[i] <= aux[j]: a[k], i = aux[i], i+1
        else: a[k], j = aux[j], j+1
```

[Q] Merge Sort는 stable한가? 이유는 무엇인가?

Stable하다

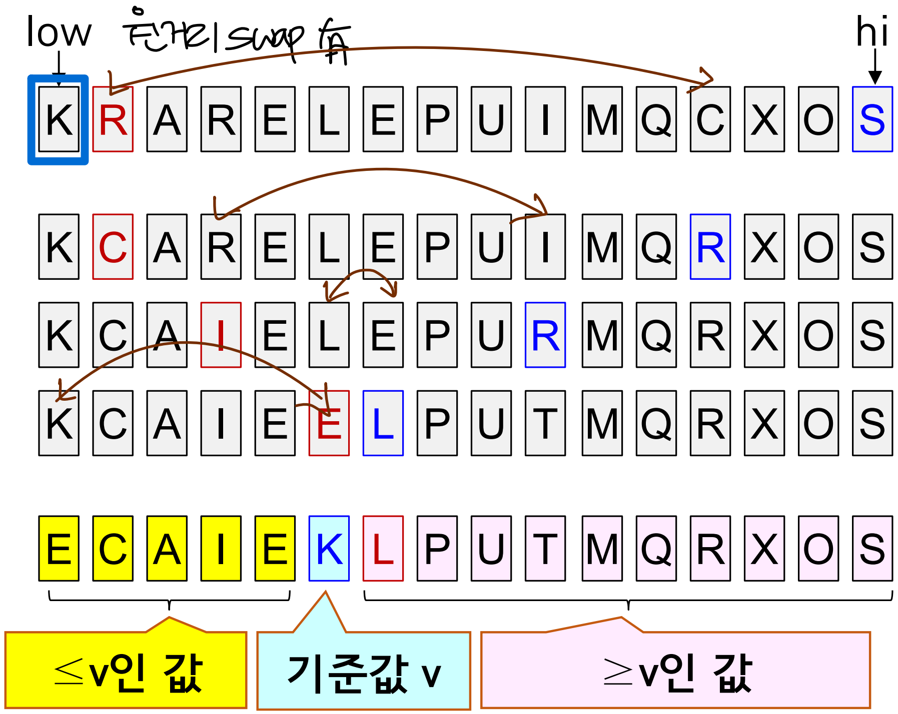
(같은 값이면 왼쪽 블록의 값을 먼저 넣기 때문)

← merge sort보다 우수↑

Quick Sort: 크기 1인 조각에서 시작 → 인접한 조각끼리 merge 반복

(안바꿔도 되는 애는 안옮겨줘서)

- Quick Sort에서 Partition 하는 과정 예
- 붉은 사각형: I, 푸른 사각형: j




```
# Partition a[lo~hi] using a[lo] as pivot
def partition(a, lo, hi):
    i, j = lo+1, hi

    while True:
        while i<=hi and a[i]<a[lo]: i = i+1
        while j>=lo+1 and a[j]>a[lo]: j = j-1
        if (j <= i): break     # Pointers cross
        a[i],a[j] = a[j],a[i] # Swap(a[i],a[j])
        i,j = i+1,j-1

    a[lo],a[j] = a[j], a[lo]   # Swap a[j] with pivot
    return j     # Return the index of item in place
```

[Q] Quick Sort는 stable한가? 이유는 무엇인가?

stable하지 않음

(먼거리 swap이 있기 때문)

- **[Q]** (x,y) 좌표로 구성된 점들의 배열 a[]가 입력으로 주어졌다. **(x,y) 좌표가 모두 같은 점들은 하나만 남기고 제거**하기 위해 배열을 정렬하였다. 다음 중 잘못된 정렬 방법은 (좌표가 같은 점을 찾기 어려운 정렬 방법은)?

두번째가 stable해야함 (insertion or Merge)

두 값 함께 key로 사용해 정렬하되,
x[0] 먼저 비교해 같으면 x[1] 비교하라

x[0] → x[1]

① quickSort(a, key=lambda x: (x[0], x[1]))

② quickSort(a, key=lambda x: x[0]) → mergeSort(a, key=lambda x: x[1])

두번째가 stable하지 않으면 앞에 애들 다 깨짐

③ mergeSort(a, key=lambda x: x[0]) → quickSort(a, key=lambda x: x[1])

④ mergeSort(a, key=lambda x: x[0]) → mergeSort(a, key=lambda x: x[1])

merge sort로 x좌표 는 → quick sort로 y좌표 는

0	0
0	0
0	1
0	5
1	0
1	1
1	1
1	1
1	2
...	

단거리 swap

Insertion sort → stable

원거리 swap

Selection sort → Unstable

13→4→1 … (원거리 swap)

Shell sort → Unstable

(같은 값이면 왼쪽 먼저 넣음)

Merge sort → stable

원거리 swap 있음

Quick Sort → Unstable

- 정리: 정렬의 Stability는 많은 application에서 중요
- Merge Sort는 효율적일 뿐 아니라 (worst case ~N $\log_2 N$) stable 하기도 하므로 'stability' 요구하는 application에서는 Quick Sort보다 선호됨

- Java의 경우 primitive type (int, float, bool 등)에 대한 정렬은 Quick Sort로,
- 그 외 임의의 class type (object)에 대한 정렬은 Merge Sort로 구현됨
- primitive type은 하나의 값만 key로 사용하지만
- class type은 여러 다른 값이 하나의 객체(object) 구성하며,
- 따라서 여러 다른 key로 정렬할 수 있으므로

Sorting (Merge Sort, Quick Sort)

널리 활용되는 정렬 알고리즘을 기본 형태로부터 개선해가는 과정 보기

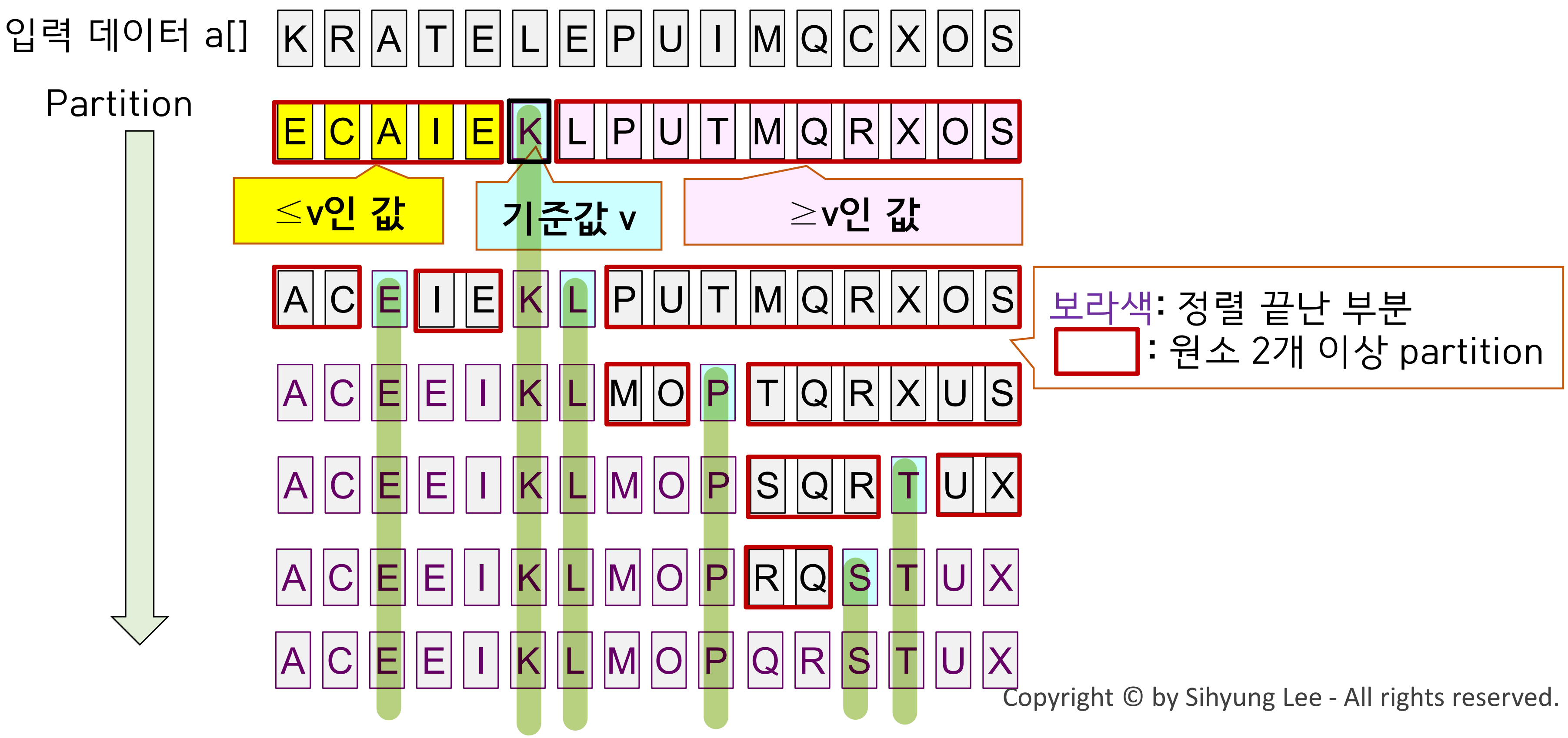
01. 퀴즈 풀이 & 예습 내용 복습 (이번 주 #1~3차 문제 공개)
02. Bottom-up Merge Sort
03. Sorting Complexity
04. Stability of Sorting
05. Quick Select
06. Duplicate Keys and 3-way Partitioning
07. 실습: Collinear Points 구현

Quick Sort에서 파생된 2가지 아이디어

Quick Sort: 기준 값 v 좌우로 partition후 각 조각 다시 partition

ε 이동할 필요X

- Partition했을 때 **기준 값은 정렬된 위치에 있게 되며** 곧 활용할 중요한 사실
- N=2개의 원소를 partition하면 정렬이 완료되므로
- 좌우 조각을 계속 partition 하다 보면 모든 원소가 정렬됨



오름차순으로 k번째 원소 (**k=0부터 시작**)

Selection: 크기 N의 배열 주어졌을 때, k번째로 작은 원소 찾기

k=10번째로 작은 원소

K	R	A	R	E	L	E	P	U	I	M	Q	C	X	O	S

k=4번째로 작은 원소

- 0≤k≤N-1
- **k=0인 경우: 최솟값(min) 찾기**
- k=N-1인 경우: 최댓값(max) 찾기
- k=N/2인 경우: 중간값(median) 찾기

- 언제 활용되는가? 통계 처리에 많이 사용됨. 예: 실험의 측정값 중 "top k" 찾기, i-th percentile 찾기(25% 구간, 50% 구간, 75% 구간 등)

- 왜 지금 배우는가? 정렬과 관련됨 (정렬 방법 그대로 활용하거나 변형해 해결 가능)

Selection: 크기 N의 배열 주어졌을 때, k번째로 작은 원소 찾기

- **[Q]** (지금 배우는 내용을 생각해 볼 때) 가장 먼저 떠오르는 ~NlogN인 해결 방법은 무엇인가?

  정렬 후 k번째 인덱스 반환

- **[Q]** k가 0에 가깝거나 (min 찾기) N-1에 가까운 경우 (max 찾기)에는 더 빠른 해법 있나?

- **[Q]** k가 0 ~ N-1 사이 임의의 값을 갖는 경우에도 ~NlogN보다 빠른 해법 있는가?

  있을듯? → 퀵소트

Quick Sort: 양쪽 조각 모두 partition

- Partition 하면 기준 값 a[j]는 정렬된 위치에 있음
- 기준 값 v **좌우 조각 모두**에 다시 **partition** 적용
- 좌우 조각을 계속 partition 하다 보면 모든 원소가 정렬됨

입력 데이터 a[] | K | R | A | T | E | L | E | P | U | I | M | Q |

Partition 후 | E | I | A | E | K | L | T | P | U | R | M | Q |

왼쪽 partition을
같은 방식으로 계속 분할

오른쪽 partition도
같은 방식으로 계속 분할

Quick Select: k 속한 쪽만 partition

- Partition 하면 기준 값 a[j]는 정렬된 위치에 있음
- j==k면 k번째 원소 찾았으므로 a[j] 반환
- j<k 라면 **오른쪽 partition만** 분할
- k<j 라면 **왼쪽 partition만** 분할

| K | R | A | T | E | L | E | P | U | I | M | Q |

| E | I | A | E | K | L | T | P | U | R | M | Q |

k=4라면,
'K 반환하고 종료

k>4라면,
오른쪽 partition만
같은 방식으로 분할

k<4라면,
왼쪽 partition만
같은 방식으로 분할

Quick Select: k 속한 쪽만 partition

k=6인 경우

index	0	1	2	3	4	5	6	7	8	9	10	11
입력	K	R	A	T	E	L	E	P	U	I	M	Q
	E	I	A	E	K	L	T	P	U	R	M	Q
	E	I	A	E	K	L	T	P	U	R	M	Q
	E	I	A	E	K	L	M	P	Q	R	T	U
	E	I	A	E	K	L	M	P	Q	R	T	U

k가 속한 쪽만 partition

기준 값 위치 j == k이므로
'M' 반환하고 종료

- Partition 하면 기준 값 a[j]는 정렬된 위치에 있음
- j==k면 k번째 원소 찾았으므로 a[j] 반환
- j<k 라면 **오른쪽 partition만** 분할
- k<j 라면 **왼쪽 partition만** 분할

Quick Sort보다 depth도 얕지만
각 depth에서 partition 대상인 원소도 훨씬 작음
(또한 더 빠르게 줄어듦)도 유의해 보시오.

- **Quick Sort 코드**

```
# Partition a[lo~hi] and
# continue to partition each half recursively
def divideNPartition(a, lo, hi):
    if (hi <= lo): return
    j = partition(a, lo, hi)
    divideNPartition(a, lo, j-1)
    divideNPartition(a, j+1, hi)

def quickSort(a):
    # Randomly shuffle a,
    # so that the partitioning item is chosen randomly
    random.shuffle(a)

    divideNPartition(a, 0, len(a)-1)
```

양쪽 partition 모두
다시 분할

둘 이상 가지로 분기하므로
재귀호출로 작성하면 편리

Quick Select: k가 속한 쪽만 partition

```
# Find k-th smallest element,
# where k = 0 ~ len(a)-1
def quickSelect(a, k):
    # Randomly shuffle a, so that two
    # partitions are equal-sized
    random.shuffle(a)

    lo, hi = 0, len(a)-1
    while (lo < hi):
        j = partition(a, lo, hi) ·· ①
        if j<k: lo = j+1 ·· ③
        elif k<j: hi = j-1 ·· ④
        else: return a[k] ·· ②

    return a[k]
```

조정

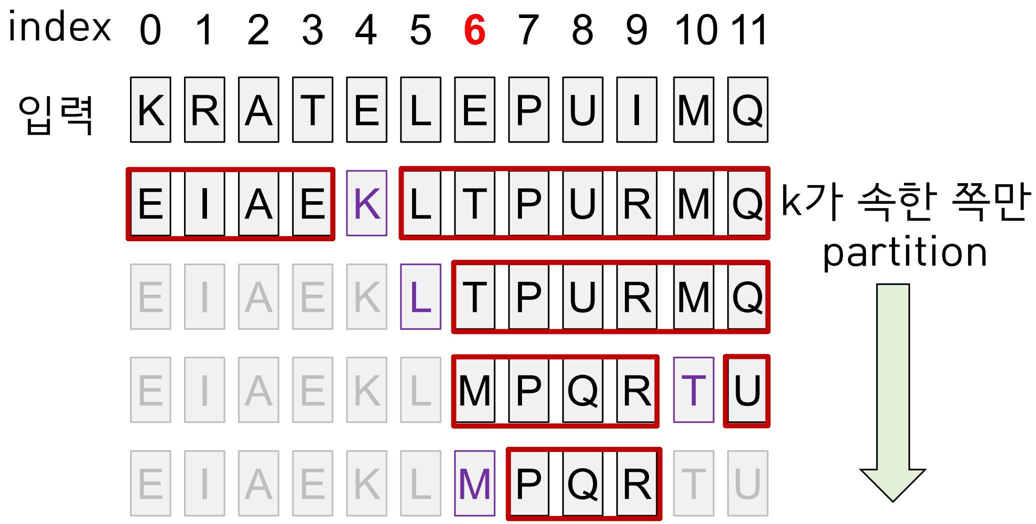
shuffle 해야 더 균등하게 분할해 성능에 좋음

partition()은 Quick Sort와 같은 함수로 기준 값의 index 반환

[Q] 코드의 어느 부분이 ①②③④에 대응되는가?

① Partition 하면 기준 값 a[j]는 정렬된 위치에 있음
② j==k면 k번째 원소 찾았으므로 a[j] 반환
③ j<k 라면 **오른쪽 partition만** 분할
④ k<j 라면 **왼쪽 partition만** 분할

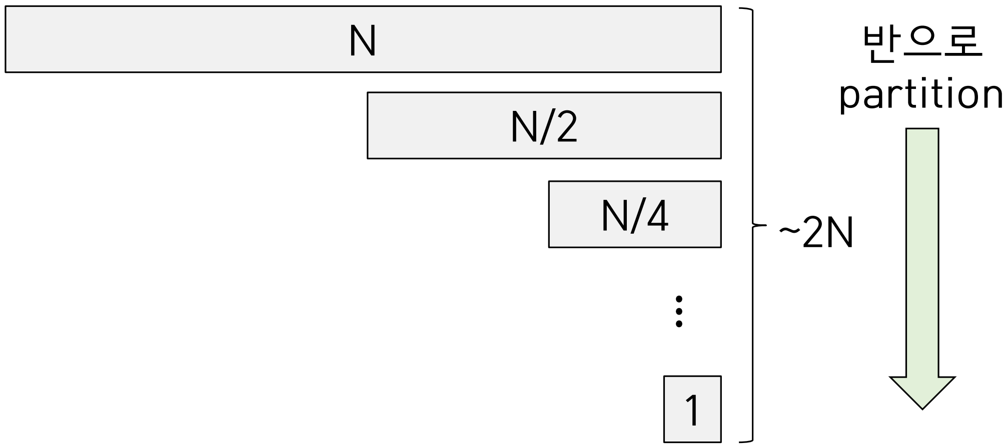
k=6인 경우

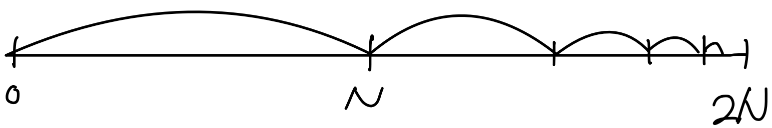


기준 값 위치 j == k이므로 'M' 반환하고 종료

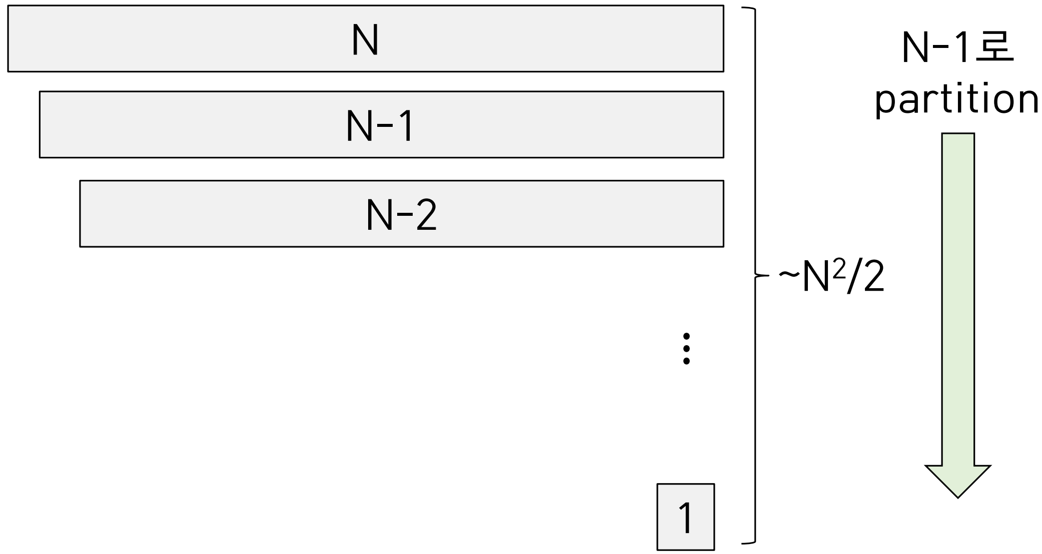
Quick Select의 성능: 평균 ~2N 회, Worst case ~$N^2/2$회 작업 필요

- N개 값 partition 위해 N에 비례한 횟수의 비교/메모리 접근 수행
- N개 원소를 크기 1이 될 때까지 **대략 반으로 나누어 간다면** (random shuffle하면 이에 가까움)
- $N + N/2 + N/4 + \cdots + 1 = \sim 2N$에 비례한 횟수의 비교/메모리 접근 수행





- N개 값 partition 위해 N에 비례한 횟수의 비교/메모리 접근 수행
- N개 원소를 partition해서 **크기 N-1인 조각**이 계속해서 나오는 경우
- $N + N-1 + N-2 + \cdots + 1 = \sim N^2/2$에 비례한 횟수의 비교/메모리 접근 수행



Sorting (Merge Sort, Quick Sort)

널리 활용되는 정렬 알고리즘을 기본 형태로부터 개선해가는 과정 보기

01. 퀴즈 풀이 & 예습 내용 복습 (이번 주 #1~3차 문제 공개)
02. Bottom-up Merge Sort
03. Sorting Complexity
04. Stability of Sorting
05. Quick Select
06. Duplicate Keys and 3-way Partitioning: **Quick Sort의 개선**
07. 실습: Collinear Points 구현

Duplicate key 많은 경우 Quick Sort의 정렬 속도 개선 가능

- Duplicate **key** 많은 경우? (**Key: 정렬 시 비교하는 값**)
  - 북구 주민들을 **나이** 순으로 정렬
  - 지원자를 **전공** 순으로 정렬
  - 메일을 **보낸 주소** 순으로 정렬
  - …

비행기/기차 시간표를
'**목적지**'를 **key**로 정렬한 예

```
Chicago 09:25:52
Chicago 09:03:13
Chicago 09:21:05
Chicago 09:19:46
Chicago 09:19:32
Chicago 09:00:00
Chicago 09:35:21
Chicago 09:00:59
Houston 09:01:10
Houston 09:00:13
Phoenix 09:37:44
Phoenix 09:00:03
Phoenix 09:14:25
Seattle 09:10:25
Seattle 09:36:14
Seattle 09:22:43
Seattle 09:10:11
Seattle 09:22:54
```

- Duplicate key 많은 경우 Quick Sort로 partition하면 **기준 값과 같은 원소도 여럿** 나오게 됨

C	C	C	C	C	C	H	H	P	P	P	S	S	S	S	S

지금까지 배운 Quick Sort (Quick Sort with 2-way partition)

- 기준 값을 중심으로 ~N 시간에 2개 partition으로 분할
- 기준 값 v 좌우 조각에 다시 partition 적용
- 좌우 조각을 계속 partition 하다 보면 모든 원소가 정렬됨

입력 데이터 a[] B C A A B C C A B C B A

Partition 후 A A A A B B | B | C C C B C

≤기준 값인 원소 / 기준 값 / ≥기준 값인 원소

Quick Sort 개선책 (Quick Sort with 3-way partition)

- 기준 값을 중심으로 ~N 시간에 3개 partition으로 분할
- 기준 값이 속한 조각은 제외하고, 좌우 조각에 다시 partition 적용
- 이를 계속하다 보면 모든 원소가 정렬됨

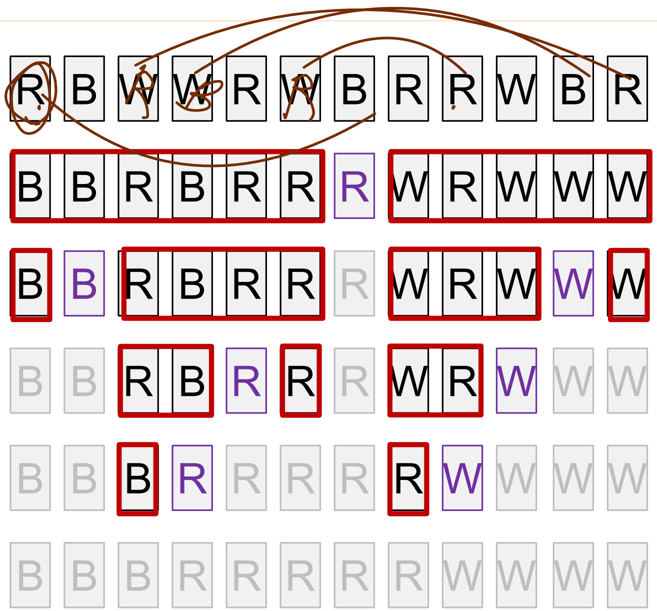
입력 데이터 a[] B C A A B C C A B C B A

Partition 후 A A A A | B B B B | C C C C

<기준 값인 원소 / =기준 값인 원소 / >기준 값인 원소

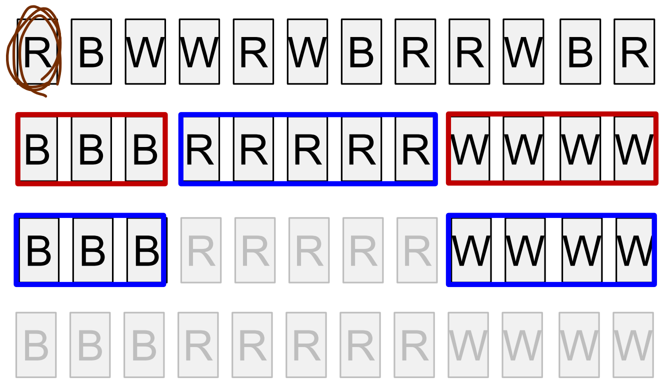
지금까지 배운 Quick Sort
(Quick Sort with 2-way partition)

- 기준 값 중심으로 **~N 시간에 2개 partition**으로 분할하고, **좌우 조각에 다시 partition** 적용



Quick Sort 개선책
(Quick Sort with 3-way partition)

- 기준 값 중심으로 **~N 시간에 3개 partition**으로 분할. **기준 값 속한 조각 제외**하고, **좌우 조각에 다시 partition** 적용

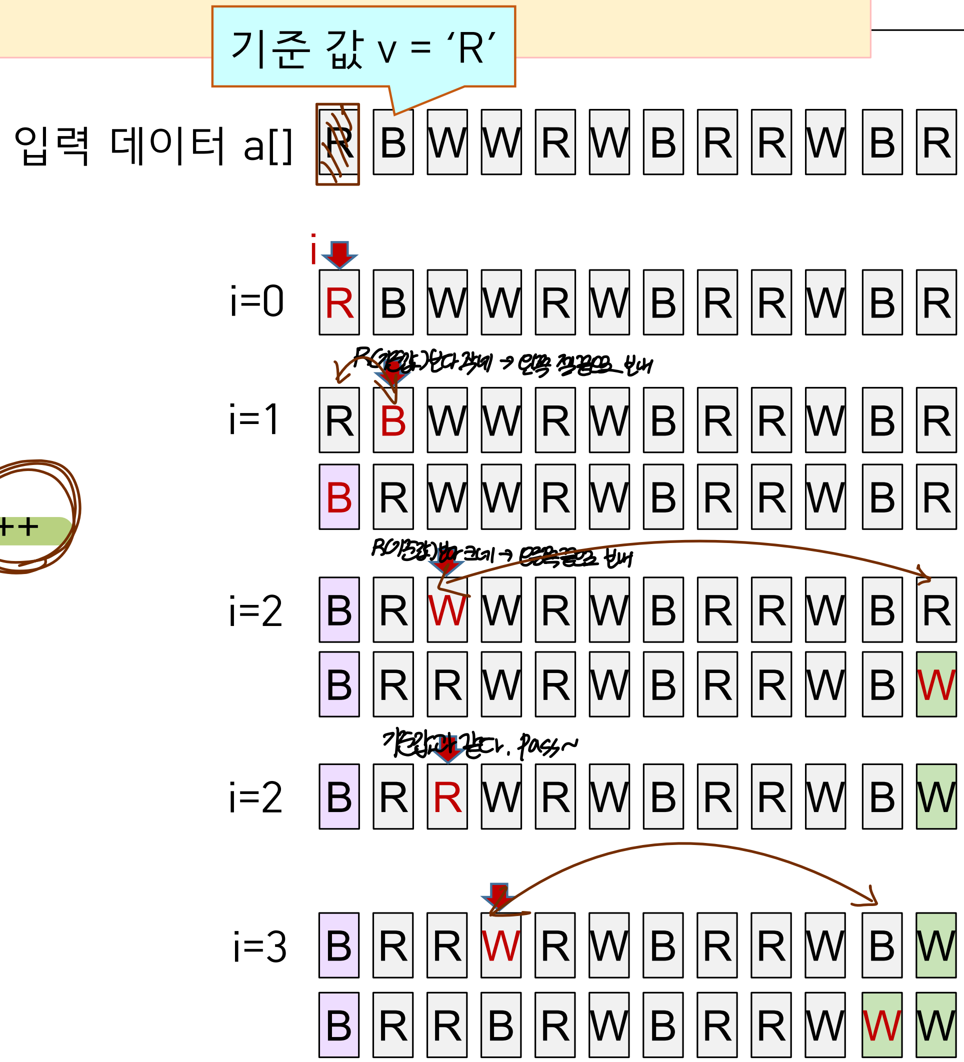


(Edsger Dijkstra's) **3-Way Partitioning**

- a[low ~ hi]를 3-way partition하기 위해
- a[low]를 기준 값 v로 정함
- 최종적으로 3개 영역으로 나누고자 함

<v 인 영역	=v 인 영역	>v 인 영역

- 왼쪽→오른쪽 방향으로 한 번에 한 원소 a[i] 검사
- a[i] < v이면 왼쪽 영역(의 오른쪽 끝)으로 이동(swap). i++
- a[i] > v이면 오른쪽 영역(의 왼쪽 끝)으로 이동(swap)
- a[i]==v 이면 이동시키지 않음. i++
- i가 오른쪽 영역에 들어서면 검사 중단



(Edsger Dijkstra's) 3-Way Partitioning

기준 값 v = 'R'

- a[low ~ hi]를 3-way partition하기 위해
- a[low]를 기준 값 v로 정함
- 최종적으로 3개 영역으로 나누고자 함

<v 인 영역	=v 인 영역	>v 인 영역

- 왼쪽→오른쪽 방향으로 한 번에 한 원소 a[i] 검사
- a[i] < v이면 왼쪽 영역(의 오른쪽 끝)으로 이동(swap). i++
- **a[i] > v이면 오른쪽 영역(의 왼쪽 끝)으로 이동(swap)**
- a[i]==v 이면 이동시키지 않음. i++
- i가 오른쪽 영역에 들어서면 검사 중단

바꿨으니까 다시 봐야함

[Q] v보다 작은 값을 왼쪽 영역으로 이동시킬 때는 i++ 하지만,
v보다 큰 값을 오른쪽 영역으로 이동시킬 때는 i++ 하지 않는 이유?

작은 값 내미 값이 더 들어오면 i 위치가 하나 옮겨지지만,
큰 값이 " , " 변하지 않는다~.

	0	1	2	3	4	5	6	7	8	9	10	11
i=3	B	R	R	B	R	W	B	R	R	W	W	W
	B	B	R	R	R	W	B	R	R	W	W	W
i=4	B	B	R	R	R	W	B	R	R	W	W	W
i=5	B	B	R	R	R	W	B	R	R	W	W	W
	B	B	R	R	R	W	B	R	R	W	W	W
i=5	B	B	R	R	R	W	B	R	R	W	W	W
	B	B	R	R	R	R	B	R	W	W	W	W
i=5	B	B	R	R	R	R	B	R	W	W	W	W
i=6	B	B	R	R	R	R	B	R	W	W	W	W
	B	B	B	R	R	R	R	R	W	W	W	W
i=7	B	B	B	R	R	R	R	R	W	W	W	W

(Edsger Dijkstra's) **3-Way Partitioning**

- a[low ~ hi]를 3-way partition하기 위해
- a[low]를 기준 값 v로 정함
- 최종적으로 3개 영역으로 나누고자 함

<v 인 영역	=v 인 영역	>v 인 영역

① 왼쪽→오른쪽 방향으로 한 번에 한 원소 a[i] 검사

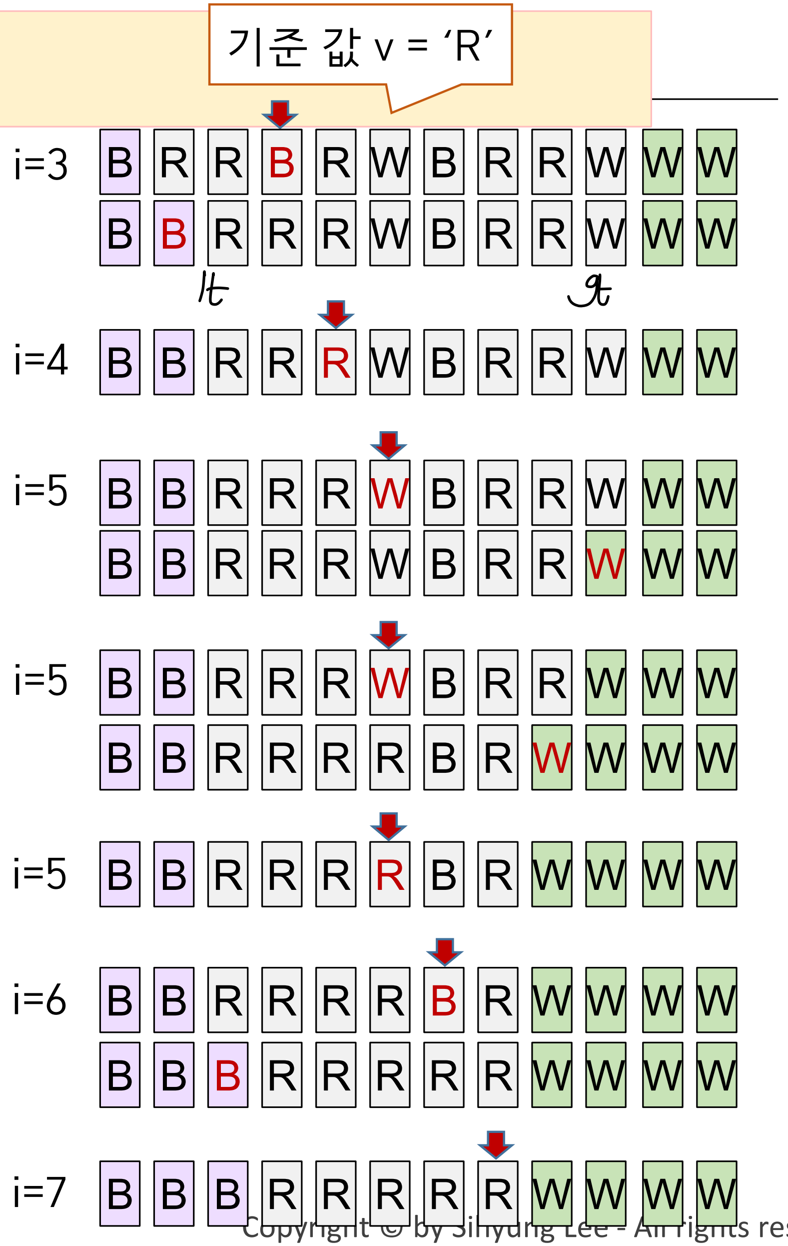
② a[i] < v이면 왼쪽 영역(의 오른쪽 끝)으로 이동(swap). i++

③ a[i] > v이면 오른쪽 영역(의 왼쪽 끝)으로 이동(swap)

④ a[i]==v 이면 이동시키지 않음. i++

⑤ **i가 오른쪽 영역에 들어서면 검사 중단**

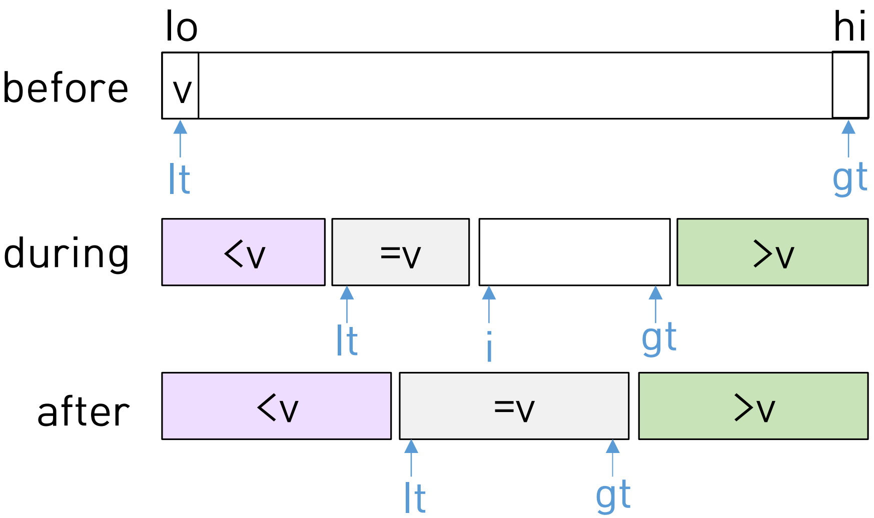
[Q] ①~⑤를 총 몇 회 수행하는가? N에 비례한 횟수인가?



(Edsger Dijkstra's) 3-Way Partitioning

- a[low ~ hi]를 3-way partition하기 위해
- a[low]를 기준 값 v로 정함
- 최종적으로 3개 영역으로 나누고자 함

① 왼쪽→오른쪽 방향으로 한 번에 한 원소 a[i] 검사
② a[i] < v이면 왼쪽 영역(lt)으로 이동(swap). i++
③ a[i] > v이면 오른쪽 영역(gt)으로 이동(swap)
④ a[i]==v 이면 이동시키지 않음. i++
⑤ i가 오른쪽 영역에 들어서면 검사 중단




```
def partition3Way(a, lo, hi):
    if (hi <= lo): return
    v = a[lo]
    lt,gt = lo,hi # Pointers to <v and >v sections
    i = lo
    while i <= gt:
        if a[i] < v:
            a[lt], a[i] = a[i], a[lt]  # Swap
            lt, i = lt+1, i+1
        elif a[i] > v:
            a[gt], a[i] = a[i], a[gt]  # Swap
            gt = gt-1
        else: i = i+1

    print(a)

    partition3Way(a, lo, lt-1)
    partition3Way(a, gt+1, hi)
```

가운데로 가는중

3-Way Quick Sort의 성능

$N=18$

5개 6개 7개

$n=3$일때 Unique key: x_1 x_2 x_3 : $-5\log\frac{5}{18} - 6\log\frac{6}{18} - 7\log\frac{7}{18}$

- N개 원소 중
- n개 서로 다른 key가 있고
- i번째 key가 x_i개 존재한다면
- 다음과 같은 횟수의 작업 필요

$$\log \frac{N!}{x_1!x_2!\ldots x_n!} = \sim -\sum_{i=1}^{n} x_i \log \frac{x_i}{N}$$

- 모든 key가 다르다면 (N개 key가 1번씩 나옴): ~N log(N)
- n개 key가 N/n번씩 나온다면: ~Nlog(n)
- 따라서 n이 작아질수록 (duplicate key가 많을수록) ~N에 가까워짐

$-\sum x \log \frac{x}{N}$

$-5\log\frac{5}{18} - 6\log\frac{6}{18} - 7\log\frac{7}{18}$

다음 사이트에서 'Few Unique' 데이터셋이 duplicate key 많은 경우에 해당
http://www.sorting-algorithms.com/

정리: Merge Sort, Quick Sort, 개선점, Applications

	Merge Sort	Quick Sort	3-Way Quick Sort	???
방법	작은 조각 → 큰 조각 순으로 merge(병합)해 가며 정렬	큰 조각 → 작은 조각 순으로 partition(분할)해 가며 정렬	3조각으로 분할(<v, =v, >v) 후 왼쪽, 오른쪽 조각만 이어서 분할	
Best	N log(N)	N log(N)	**N**	**N**
Average	**N log(N)**	**N log(N)**	**N log(N)**	**N log(N)**
Worst	**N log(N)**	$N^2/2$	$N^2/2$	**N log(N)**
추가 공간 필요?	~N 추가 공간 필요	**불필요**(원배열 안에서 순서만 변경)	**불필요**	**불필요**
Stable?	**Yes**	No	No	Yes
Java	Class type (Object) 정렬		Primitive type (int, float, …) 정렬	

Stability 필요하며 (Class 객체는 여러 field 가지므로) 추가공간 있을 것이라 가정

Stability 불필요하며, 공간 부족할 가능성 높다고 가정

Sorting (Merge Sort, Quick Sort)

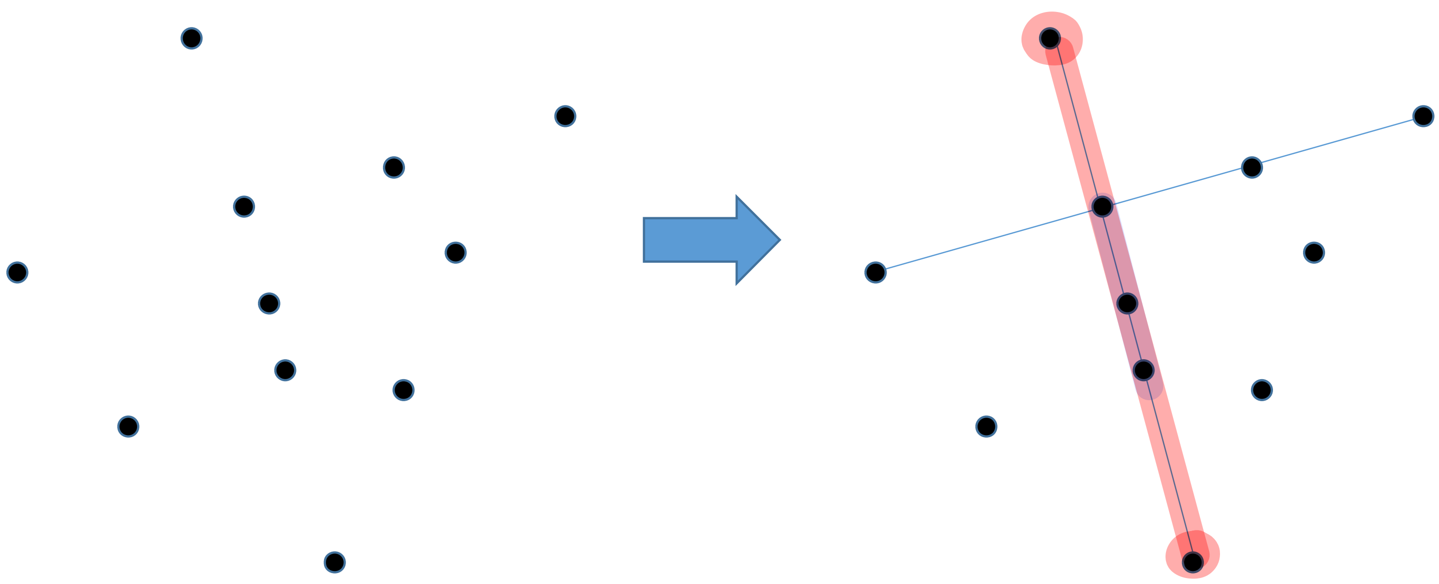
널리 활용되는 정렬 알고리즘을 기본 형태로부터 개선해가는 과정 보기

01. 퀴즈 풀이 & 예습 내용 복습 (이번 주 #1~3차 문제 공개)
02. Bottom-up Merge Sort
03. Sorting Complexity
04. Stability of Sorting
05. Quick Select
06. Duplicate Keys and 3-way Partitioning
07. 실습: Collinear Points 구현

목표: 정렬 활용하는 어플리케이션 구현해 보며,
어떤 경우 정렬 사용하면 좋을지 느껴보도록 함

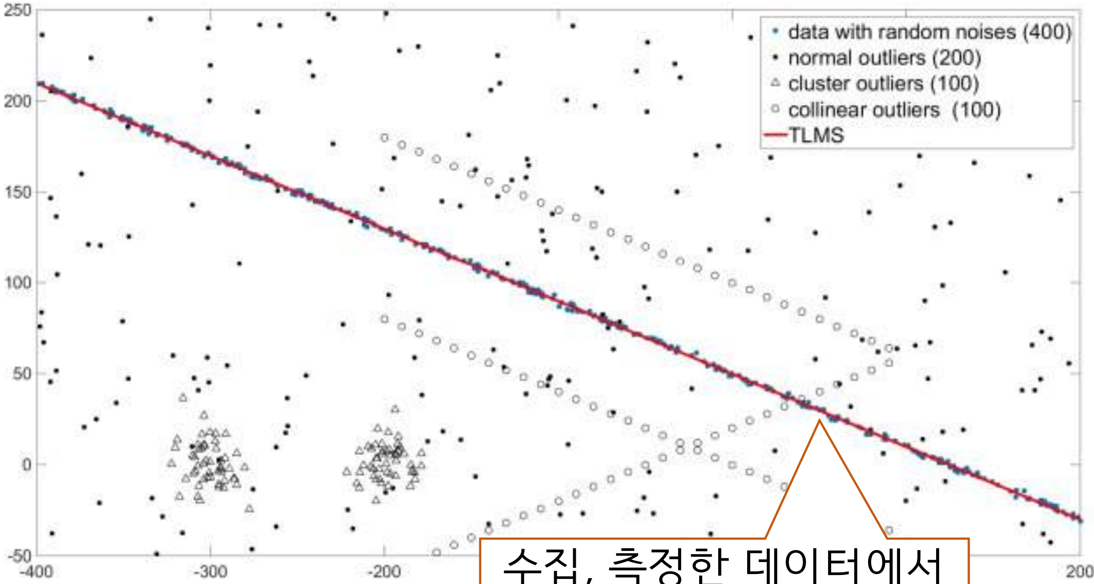
Collinear Points: 4개 이상 점 연결하는 maximal한 직선 모두 찾기

- N개 점의 (x,y) 좌표가 입력으로 주어졌을 때
- 4개 이상 (≥4) 점 연결하는 maximal한 직선 모두 찾기



[Q] 4개 이상 (≥4)의 점을 연결하는 **maximal 하지 않은** 직선의 예를 드시오.

Collinear Points의 활용 예:



수집, 측정한 데이터에서
서로 연관된 값 파악

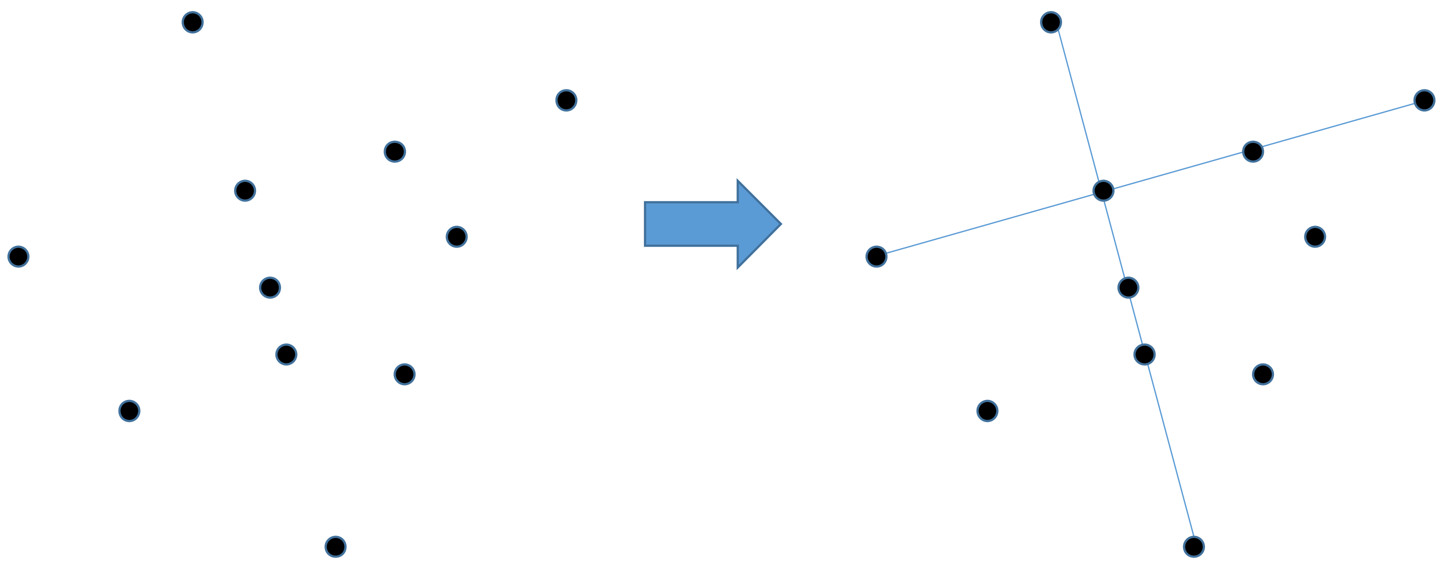
Regression(회귀분석)

이미지에서 다각형 물체의
경계선 파악

Collinear Points 탐지 방법

- p0=(x0,y0) 와 pi=(xi, yi)가 이루는 **기울기**를 si = $\frac{(yi-y0)}{(xi-x0)}$ 라 할 때
- s1 = s2 = s3 라면, p0, p1, p2, p3은 같은 직선 상에 있음

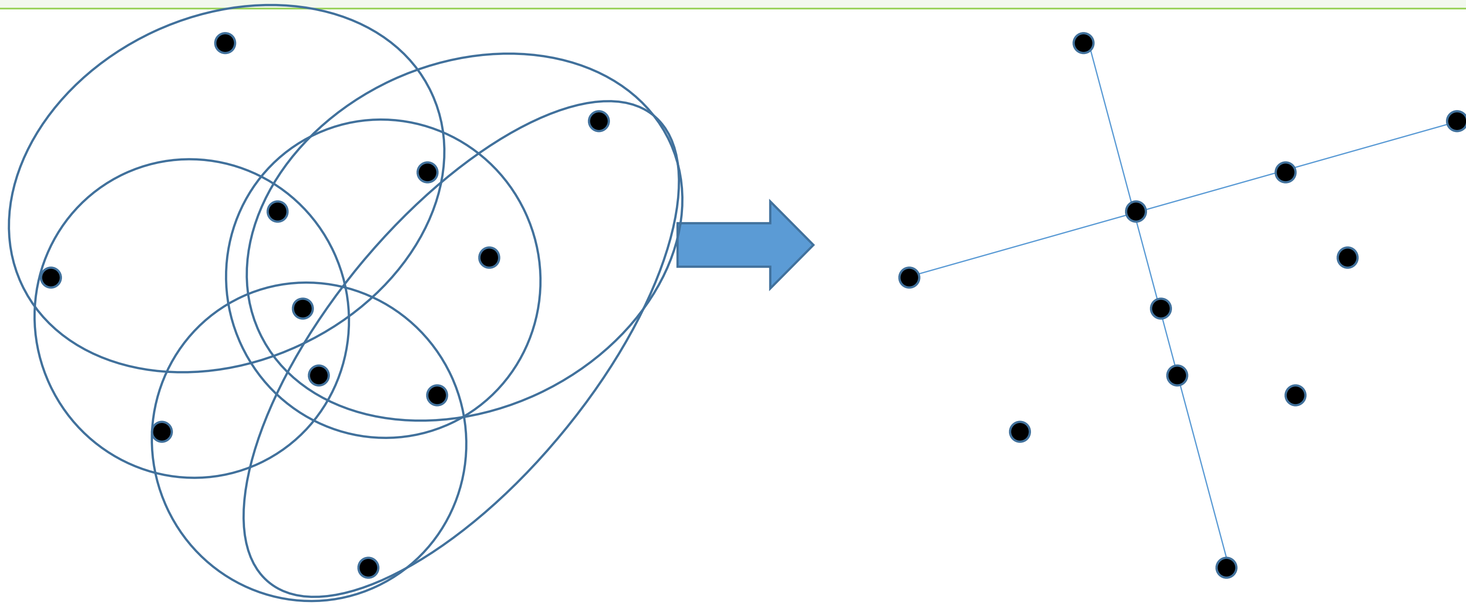
기울기가 같다면



[Q] 위에서 제시한 조건이 맞는지 검증하시오.

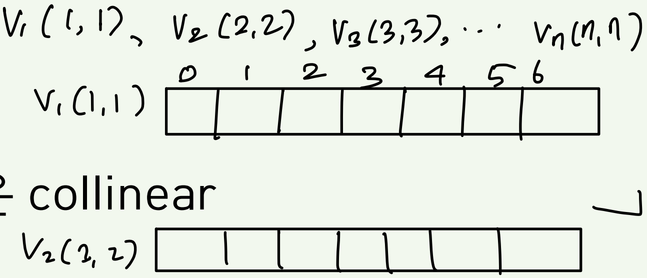
Collinear Points 탐지 방법: Brute Force

- (재귀 호출 혹은 중첩 for loop 사용해)
- 모든 가능한 **4**개 점 p0, p1, p2, p3에 대해 s1 = s2 = s3인지 확인
- 모든 가능한 **5**개 점 p0, p1, p2, p3, p4에 대해 s1 = s2 = s3 = s4인지 확인
- …
- 모든 가능한 **N**개 점 … 에 대해 … 확인
- 너무 많은 가능성 있음 ~$N^4 + N^5 + \cdots$

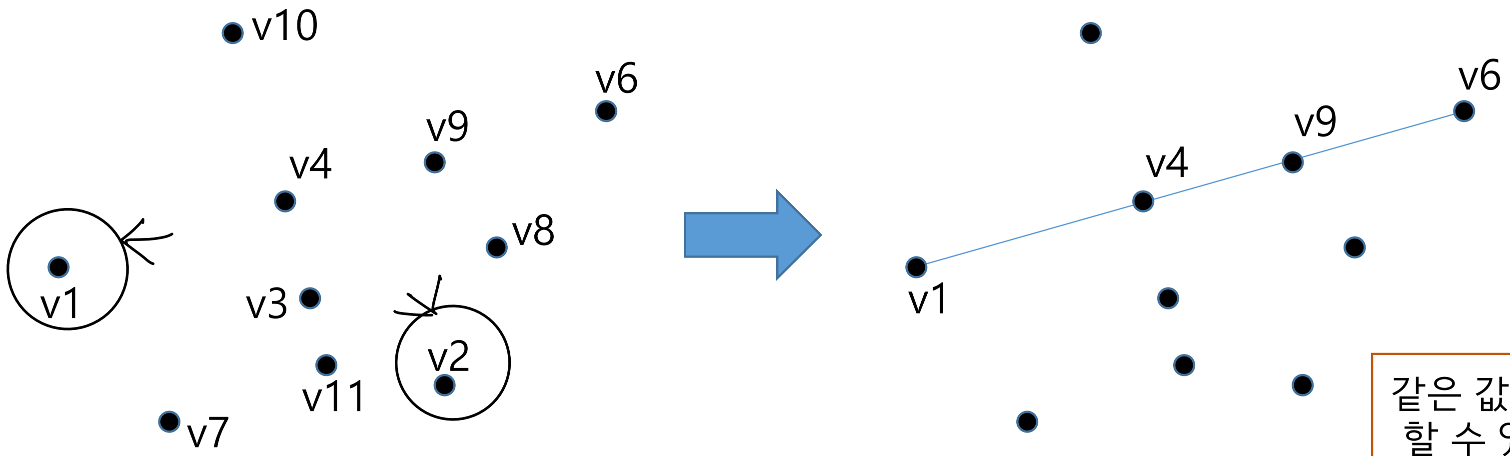


Collinear Points 탐지 방법: 정렬 활용한 더 효율적인 방법

- 각 점 p에 대해
  - p가 다른 모든 점과 이루는 기울기 계산해, 기울기를 key로 정렬
  - 정렬 결과에서 3개 이상의 인접한 점이 같은 기울기 갖는다면 이들은 collinear



- N개의 점 각각에 대해 다른 모든 점을 정렬(~NlogN)하므로 ~N^2logN



같은 값끼리 모여 있을 때 더 빠르게 할 수 있는 일은 '정렬'해 수행하는 것 고려해 보기. 정렬하는 시간 필요하지만, 종합적으로는 더 빠르게 일을 마칠 수도 있음

점	v7	v11	v2	v3	v8	v4	v9	v6	v10
v1과의 기울기	-1.5	-0.5	-0.48	-0.1	0.1	0.3	0.3	0.3	1.2

같은것끼리

프로그램 구현 조건

- Collinear Points 찾는 함수 구현

  def collinearPoints(points):

- 입력 **points**: xy 좌표계에 속한 점의 list
  - points는 tuple (x,y)의 리스트임 (예: [(3,2), (4,-1), (0,0), (-2,2)])
  - points에 속한 점은 모두 좌표가 서로 다름 x/y 값 둘 다 일치하는 점은 입력으로 주어지지 않음

- 반환 값: 4개 이상 (≥4) 점 연결하는 maximal한 직선 **모두** 리스트에 담아 반환
  - 양 끝이 p, q인 직선(p<q)은 양 끝점 좌표를 4-tuple (px,py,qx,qy) 형식으로 나타내며
  - 두 점 중 더 작은 점이 p, 더 큰 점이 q임. 두 점의 대소 관계 따질 때는 x좌표 먼저 비교하고, x좌표 같으면 y좌표 비교함
  - 둘 이상 직선을 반환할 때는 px 작은 직선 먼저 반환하며, px 같다면 py→qx→qy 순으로 비교

- 정렬 위해서는 sorted() 혹은 list.sort() 함수 활용

프로그램 입출력 예

```
>>> collinearPoints([(19000,10000), (18000,10000), (32000,10000), (21000,10000), (1234,5678), (14000,10000)])
[(14000, 10000, 32000, 10000)]


>>> collinearPoints([(10000,0), (0,10000), (3000,7000), (7000,3000), (20000,21000), (3000,4000), (14000,15000),
(6000,7000)])
[(0, 10000, 10000, 0), (3000, 4000, 20000, 21000)]

>>> collinearPoints([(0,0), (1,1), (3,3), (4,4), (6,6), (7,7), (9,9)])
[(0, 0, 9, 9)]

>>> collinearPoints([(1,0), (2,0), (3,0), (4,0), (5,0), (6,0), (8,0)])
[(1,0,8,0)]

>>> collinearPoints([(7,0), (14,0), (22,0), (27,0), (31,0), (42,0)])
[(7,0,42,0)]

>>> collinearPoints([(12446,18993), (12798,19345), (12834,19381), (12870,19417), (12906,19453), (12942,19489)])
[(12446,18993,12942,19489)]

>>> collinearPoints([(1,1), (2,2), (3,3), (4,4), (2,0), (3,-1), (4,-2), (0,1), (-1,1), (-2,1), (-3,1), (2,1), (3,1),
(4,1), (5,1)])
[(-3, 1, 5, 1), (1, 1, 4, -2), (1, 1, 4, 4)]
```

프로그램 구현 조건

- 최종 결과물로 CollinearPoints.py 파일 하나만 제출하며, 이 파일만으로 코드가 동작해야 함

- import는 사용할 수 없음

- 각자 테스트에 사용하는 모든 코드는 반드시 if __name__ == "__main__": 아래에 넣어
- 제출한 파일을 import 했을 때는 실행되지 않도록 할 것

프로그램 구현 조건 - 성능

- 100개 점 좌표가 입력으로 주어졌을 때 기준 1초 내로 답 나오지 않는다면
- 해당 테스트 케이스는 통과하지 못한 것으로 보고 다음 케이스로 넘어감

자주 하는 실수 (유의 사항)

- 5개 점이 p→q→r→s→t 순서로 같은 직선상에 나타나며 p<t라면 (px,py,tx,ty)를 반환하며
- (px,py,sx,sy), (qx,qy,tx,ty) 등 **maximal하지 않은 직선은 반환하면 안 됨**
- 또한, **같은 직선은 단 한 번만 반환**해야 하며 여러 번 반환하면 안 됨

- 한 점이 여러 다른 직선의 일부일 수 있음
- 특히 한 점 x가 여러 다른 직선의 양 끝점일 수 있으며 (x-a, x-b, x-c, d-x, …)
- 그러한 경우 출력에 같은 점 x가 여러 번 등장할 수도 있음

- x축과 평행인 직선의 기울기는 0임
- **y축과 평행인 직선의 기울기**는 무한대로 표현하며, Python에서는 **float('inf')**로 나타냄
- 점 p 자신에 대한 기울기는 (필요하다면) 음의 무한대로 표현하며, float('-inf')로 나타냄

스마트 출결

수업 전

문서 형태
개요/이론 학습

수업 전

개요 관련 퀴즈 수행
스스로 학습 & 시행착오

40~50분

수업(화/수)

퀴즈 풀이
개요 복습

수업(화/수)

응용/심화 내용 학습
질의 응답

수업(화/수)

실습 과제 (프로그래밍)
질의 응답

~3시간

실습과제 수업 중 완료 못한 경우
당일 저녁 or 다음날까지
각자 수행해 제출

이론 학습 (1.5~2시간) + 실습(1~1.5 시간) = 총 ~3시간

12:00까지 실습 & 질의응답

- 작성한 코드는 lms > 강의 콘텐츠 > 오늘 수업 > 실습 과제 제출함에 제출

- 시간 내 제출 못한 경우 내일 11:59pm까지 제출 마감
- 마감 시간 후에는 제출 불가하므로 그때까지 작성한 코드 꼭 제출하세요.