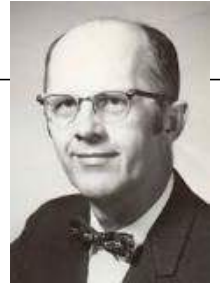




Sorting (Shell Sort, Shuffle Sort, Convex Hull)

알려진 정렬 방법에 기반한 진보된 정렬 방법 이해, 정렬의 적용 예 보기

01. 퀴즈 풀이 & 예습 내용 복습 (이번 주 #1~3차 답안 공개)
02. Shell Sort
03. Shuffle Sort (Shuffling에 정렬 방법 적용)
04. Convex Hull (Tight boundary 찾기 위해 정렬 방법 적용)
05. 실습: Convex Hull 구현



<Donald Shell>

Shell Sort의 중요성 (Shell's Sorting Method)

- 기존 정렬 방법 (Insertion Sort) 활용 간단한 아이디어가 성능 향상 이뤄낸 예
- 코드 사이즈 작아 메모리가 한정적인 embedded system 등에 사용하기 좋음
- 아직 연구 진행 중
 - 실험 상으로는 $\sim N \log N$ 인 기존 정렬 방식보다 빠를 수 있음을 보였으나
 - 아직 수학적 모델 사용해 모든 경우에 대해 정확히 비교하지는 못하였음
 - (수학적 모델 만들기 어려웠음)
 - h-sort의 h 줄여 나가는 최적의 방식도 아직 합의하지는 못함

Insertion Sort 복습

Iteration i에

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

[Q] Iteration i에서 $a[0] \sim a[i-1]$ 에 새로 추가되는 원소 $a[i]$ 는 항상 일정한 거리만큼 왼쪽으로 이동하는가? *NO.*


- 오름차순 정렬 예
- (**'I'**는 정렬된 부분, **'O'**는 새로 insert된 원소)

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



Insertion Sort: 비교 & swap 횟수가 성능에 영향 미침

- 입력 데이터의 상태에 따른 Insertion Sort의 성능 비교



입력 데이터의 상태	대소 비교 횟수	swap 횟수
(best case) 이미 정렬된 상태 아무 원소도 이동시키지 않아도 됨	$N-1$	0
(worst case) 반대 방향으로 정렬된 상태 각 원소를 왼쪽 끝까지 이동시켜야 함	$\sim N^2/2$	$\sim N^2/2$
(average case) 각 원소를 절반 정도 이동시켜야 하는 상태	$\sim N^2/4$	$\sim N^2/4$

- 순서 바꾸어야 할 원소의 쌍이 N 에 비례한 개수($\leq cN$ 개)라면 (이 조건을 만족하는 경우를 partially ordered 상태라 함)
- $\sim N$ 회 작업이 필요하므로 Insertion Sort가 다른 정렬 방법에 비해 유리

인접한 원소끼리 swap 하므로 Inversion 수만큼 비교 & swap 수행

Iteration i에

- 이미 정렬된 $a[0] \sim a[i-1]$ 에
- $a[i]$ 를 적절한 위치 (정렬되었을 때의 위치) 찾아 추가
 - 적절한 위치까지 **비교 & swap**하며 이동
- 그 결과 $a[0] \sim a[i]$ 까지 정렬된 상태 됨

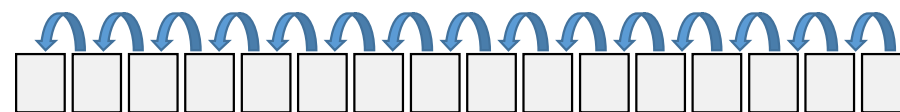
Inversion

- 정렬 순서 어긋난 쌍
- A E E L M O T R X P S 에는 6개의 inversion 있음 (T-R, T-P, T-S, R-P, X-P, X-S) $\rightarrow C_2$
- Insertion Sort는 inversion 수만큼 swap 수행
- Inversion 없다면 (0개) 정렬된 상태

inversion 개수를
swap이 대체함 \rightarrow

따라서 Inversion 개수 $\sim N$ ($\leq cN$ 개) 인 경우
정렬 시간 $\sim N$
이때를 partially ordered 상태라 함

- 오름차순 정렬 예
- ('I'는 정렬된 부분, 'O'는 새로 insert된 원소)
- A E E L M O T R X P S
- A E E L M O T R X P S (0회 swap)
- A E E L M O R T X P S (T-R swap)
- A E E L M O R T X P S (0회 swap)
- A E E L M O P R T X S (X-P, T-P, R-P swap)
- A E E L M O P R S T X (X-S, T-S swap)



한 번에 한 원소씩 왼쪽으로 이동해가며 비교&swap 함도 기억



[Q] 다음 데이터에는 몇 개의 inversion이 있나?

■ A E E L M P X: 정렬된 상태임 $\rightarrow \therefore$ inversion 0개

■ A M E E L P X: inversion 3개

\therefore 모든 쌍을 비교해야 함
(= swap이 3번 필요함)

[Q] sorting-algorithms.com에서 insertion sort 실행해 보기. 아래 3 경우 비교해 보기
Random (모든 쌍 중 반 정도가 inversed)
Nearly sorted (Inversion 개수 $\sim N$)
Reversed (모든 쌍이 inversed. Inversion 개수 최대)

Insertion Sort는 인접한 원소끼리 swap 하므로 한 swap이 한 inversion만 해결
한 swap이 여러 inversion 한 번에 해결하게 할 수 없을까?

↗ insertion sort

h-sort: 모든 h만큼 떨어진 원소끼리 (insertion sort 방식으로) 정렬된 상태로 만들

▪ h=4, 즉 4-sort 수행한 예. 4칸 떨어진 원소 간에는 모두 정렬됨

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

L M P T → sorting된 상태

E		H		S		S
	E		L		O	
		A		E		L
						R

이번 시간에 배울 Shell Sort에서 사용하는 기본 operation이 h-sort

1칸 떨어진 원소 간에는 모두 정렬되지 않는 것을 유의해 보세요.

[Q] A F C H G T X Z 는 2-sort된 상태인가? 만약 그렇지 않다면 어느 쌍이 정렬되지 않았나?

A	C	G	X	
	F	H	T	Z

Yes

[Q] 1 -1 5 7 4 4 9 8 는 3-sort된 상태인가? 만약 그렇지 않다면 어느 쌍이 정렬되지 않았나?

1		7		9	
	-1		4		8

No

5			4		
---	--	--	---	--	--

sorting 안된 상태

→ 아직 h-sort가 필요한 상태

h가 클수록 대략 정렬이 된 상태
 " 작을수록 안정정렬에 가까운 상태

h-sort: 모든 h만큼 떨어진 원소끼리 (insertion sort 방식으로) 정렬된 상태로 만들

■ h=4, 즉 4-sort 수행한 예. 4칸 떨어진 원소 간에는 모두 정렬됨

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

L				M				P				T			
---	--	--	--	---	--	--	--	---	--	--	--	---	--	--	--

E H S S

E L O X

A E L R

이번 시간에 배울 Shell Sort에서 사용하는 기본 operation이 h-sort

[Q] 1-sort는 우리가 지금까지 해온 정렬과 같은가? 왜 그러한가? *yes*

6 $h < n$ (원소 개수보다 h 값이 작아야 함)

h-sort: 모든 h 만큼 떨어진 원소끼리 (insertion sort 방식으로) 정렬된 상태로 만들

[Q] a-sort, b-sort ($a < b$) 중 어느 쪽이 할 일이 더 많을까? 왜 그런가?

--	--	--	--	--	--	--	--

0	1	2	3	4	5	6	7

1sort: 7

0	1	2	3	4	5	6	7

2sort: 6

0	1	2	3	4	5	6	7

3sort: 5

h-sort 수행 방법: Insertion sort를 h칸 간격으로 수행

- Insertion sort 사용해 3-sort 수행하는 과정

입력: M O L E E X A S P R T



a[h]부터 왼쪽으로 이동시킴

h-sort 코드 vs. Insertion sort

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

h-sort 코드

h칸 씩
왼쪽으로 이동하며
비교&swap

```
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
```

a[1]부터 왼쪽으로 이동시킴

1칸 씩
왼쪽으로 이동하며
비교&swap

- Insertion sort 사용해 3-sort 수행하는 과정

입력: 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

A E L E O P M S X R T

[Q] h-sort 코드를 보고 답하시오.
h=1일 때 (즉 1-sort 일 때) 오른쪽의
Insertion sort와 같아지는가? *yes*

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

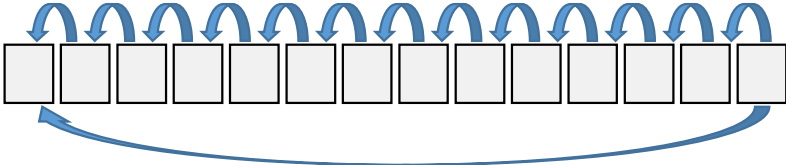
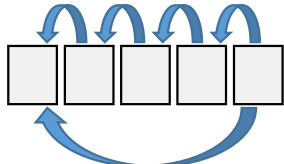
- 0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15
- input: S H E L L S O R T E X A M P L E
 - 13-sort: P H E L L S O R T E X A M S L E
 - 4-sort: L E E A M H L E P S O L T S X R
 - 1-sort: A E E E H L L L M O P R S S T X

최종적으로 1-sort 수행해 완전히 정렬된 상태로 만들

- h 는 1씩 감소시키지 않고 **그보다 큰 간격으로 감소**시킴. 감소시키는 방법은 곧 알아보겠음

데모 보기: Shell Sort vs. Insertion Sort 간 **swap 횟수 비교**

Shell Sort: 왜 Insertion Sort보다 전반적으로 비교 & Swap 횟수 줄어드나?

- input: S H E L L S O R T E X A M P L E
- 13-sort: P H E L L S O R T E X A M S L E

- 4-sort: L E E A M H L E P S O L T S X R

- 1-sort: A E E E H L L L M O P R S S T X

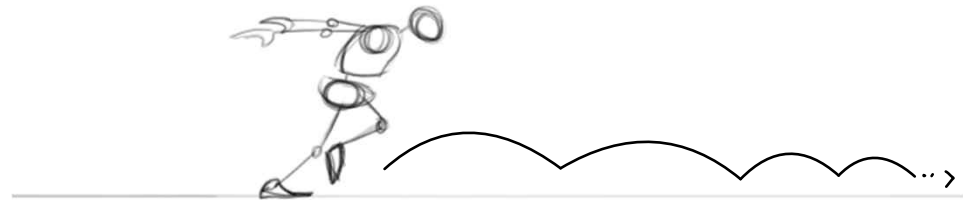
최종적으로 1-sort 수행해 완전히 정렬된 상태로 만듦

(여러 차례 비교 & swap 통해) 여러 칸 앞으로 가야 할 것들을 한 번에 h칸씩 앞으로 이동시켜 비교 & swap 횟수 줄임
(한 swap으로 여러 inversion 한 번에 해결)

Shell Sort: 왜 Insertion Sort보다 전반적으로 비교 & Swap 횟수 줄어드나?

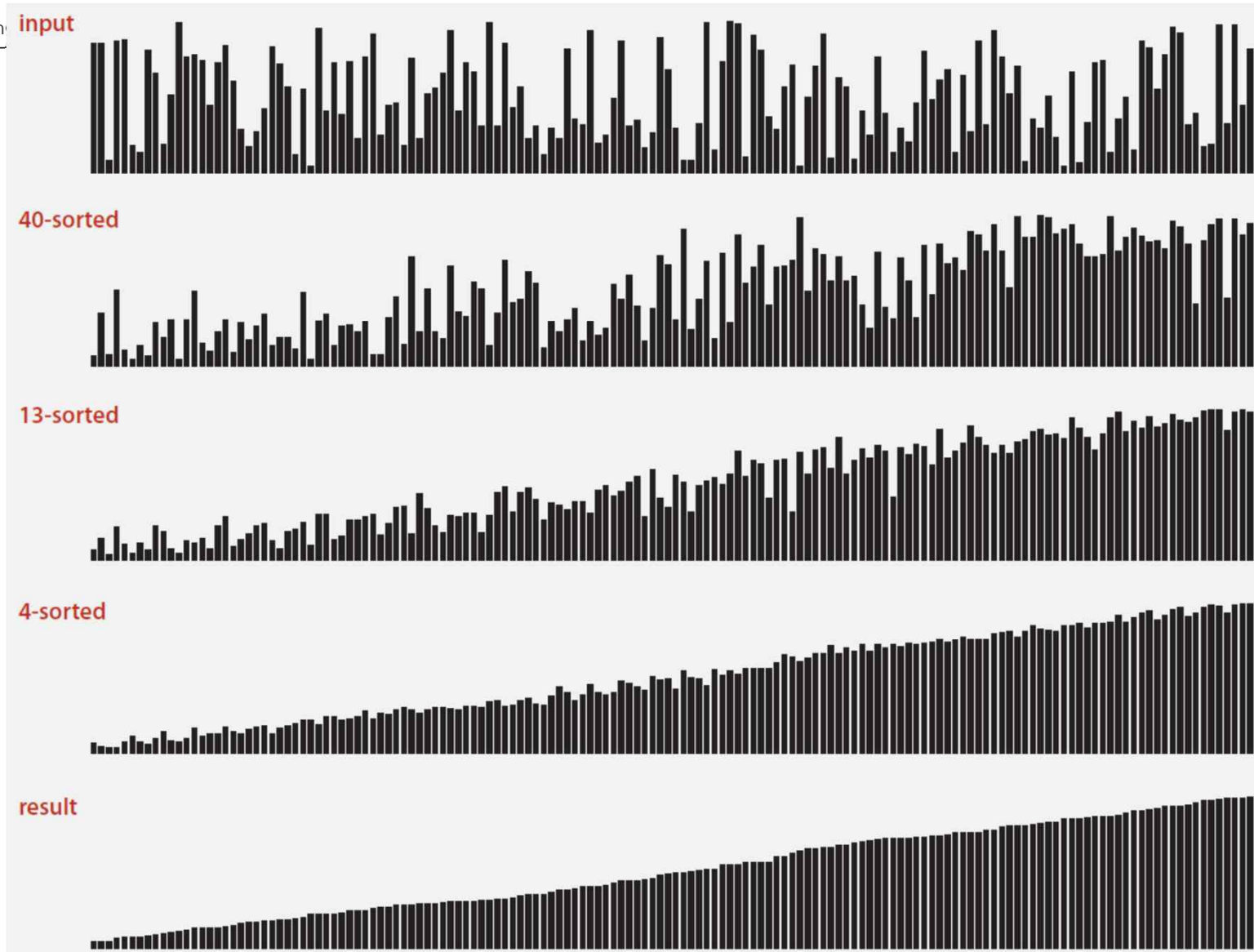


Insertion sort: 1-sort



Shell sort: 13-sort → 4-sort → 1-sort

<https://i.gifer.com/>



Shell Sort: 여러 번의 h-sort 해도 괜찮은가?

각 h-sort는 Insertion sort($\sim N^2$)만큼 오래 걸리지 않는가?

- input: S H E L L S O R T E X A M P L E
- 13-sort: P H E L L S O R T E X A M S L E
- 4-sort: L E E A M H L E P S O L T S X R
- 1-sort: A E E E H L L L M O P R S S T X

비교하진

여러 칸($h > 1$) 떨어진 원소끼리만
비교하므로 비교&swap 횟수 적음

$h=1$ 로 Insertion sort와 같지만,
앞의 h-sort들에 의해 거의 정렬된 상태이므로 (inversion 수가 $\sim N$
에 가깝게 줄어드는 상태이므로)
 $\sim N$ 에 가까운 비교 & swap 만으로 완료



Algorithm 2,

6 h-sort 하는 횟수를 $\log n$ 에 비례하게

7→3→1 sort한 예

input

S O R T E X A M P L E
0 1 2 3 4 5 6 7 8 9 10

7-sort

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
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

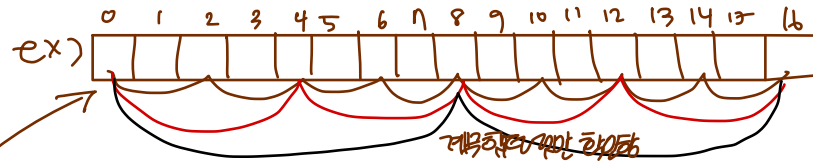
1-sort

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 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 T X

result

A E E L M O P R S T X

각 h-sort에서 각각 0~2회만 이동했음에 유의



h-sort의 h 값은 어떻게 선정해야 하나?

- 2^k : $1 \leftarrow 2 \leftarrow 4 \leftarrow 8 \leftarrow 16 \leftarrow 32 \leftarrow \dots$
- No... Why? *작은 인덱스만 활용하므로 효과없는 방법임*

h를 어떤 방식으로 선택하더라도 최종적으로 $h=1$, 즉 1-sort 수행해
완전히 정렬된 상태로 만들어야 함에 유의

- $2^k - 1$: $1 \leftarrow 3 \leftarrow 7 \leftarrow 15 \leftarrow 31 \leftarrow 63 \leftarrow \dots$
- 2^k 와 같은 문제는 없음

또

- $3x + 1$: $1 \leftarrow 4 \leftarrow 13 \leftarrow 40 \leftarrow 121 \leftarrow 364 \leftarrow \dots$

가장 많이 사용

- Donald Knuth가 제안한 방법. 다음 h를 **계산하기 쉬운 편**이며, **성능**도 괜찮음

- $1 \leftarrow 5 \leftarrow 19 \leftarrow 41 \leftarrow 109 \leftarrow 209 \leftarrow \dots$

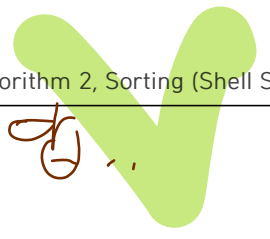
- Princeton 대학 R. Sedgewick 교수가 제안한 수열.

- 지금까지 보인 수열 중에 (실험적으로) **성능** 가장 나은 편

어떤 h 값이 최적인지는
아직 open problem

$$h = \begin{matrix} 1 \\ 2 \\ 4 \\ 8 \\ \vdots \\ N \end{matrix} \left. \vphantom{\begin{matrix} 1 \\ 2 \\ 4 \\ 8 \\ \vdots \\ N \end{matrix}} \right\} \log_2 N$$

$$h = \begin{matrix} 1 \\ 3 \\ 9 \\ 27 \\ \vdots \\ N \end{matrix} \left. \vphantom{\begin{matrix} 1 \\ 3 \\ 9 \\ 27 \\ \vdots \\ N \end{matrix}} \right\} \log_3 N$$



N 넘지 않는 h의 최댓값 구하기

```
def shellSort(a):
    N = len(a)
    h = 1
    while h < N/3:
        h = 3*h + 1 # Knuth's Sequence 1, 4, 13, 40, ...
```

h를 1까지 변경하며 h-sort 수행

```
while h >= 1:
    hInsertionSort(a, h) 13 sort → 4 sort → 1 sort
    h = h//3 (과정 생략)
```

[Q] N = 30이라면 h의 최댓값은 어떤 값이 되는가?

[Q] 두 번째 while loop에서 $h = h//3$ (3으로 나눈 후 소수점 아래 제외) 하면 Knuth의 수열 $3x+1$ 을 따르는 이유는?

[Q] 첫 while loop에서는 왜 “N을 넘지 않는” h의 최댓값을 구하는가? (h가 이보다 커지면 안되나?)

0	1	2	3	4	...	27	28	29
---	---	---	---	---	-----	----	----	----

$h = 1 \rightarrow 4 \rightarrow 13$

Shell Sort의 성능 분석: 아직 정확한 수학적 모델을 찾지는 못함

- 각 h-sort가 실험적으로 $\leq cN$ 회 (c는 작은 정수) 비교/swap 함 보여졌으나
- 정확한 수학적 모델은 찾지 못함
- $h = 3x + 1$ 을 사용하는 경우 $\sim \log N$ 회의 h-sort 수행 $N \times (N \log N)$
- 따라서 $\sim N \log N$ 회의 비교/swap을 수행할 것으로 예측됨

N	비교 횟수 (실험)	$2.5 N \log_e N$
5,000	93K	106K
10,000	209K	230K
20,000	467K	495K
40,000	1022K	1059K

input

S O R T E X A M P L E

7-sort

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

result

A E E L M O P R S T X

[Q] Quick Sort와의 비교? sorting-algorithms.com 에서 nearly-sorted 경우 보기



Sorting (Shell Sort, Shuffle Sort, Convex Hull)

진보된 정렬 방법 이해하고 정렬의 적용 예 보기

01. 퀴즈 풀이 & 예습 내용 복습 (이번 주 #1~3차 문제 공개)
02. Shell Sort
03. Shuffle Sort (Shuffling에 정렬 방법 적용)
04. Convex Hull (Tight boundary 찾기 위해 정렬 방법 적용)
05. 실습: Convex Hull 구현

Shuffle이란? 입력 데이터의 순서를 임의로 섞음

- 특히 대부분의 경우 ^{확률의 일정한 배율을 부여} **uniformly random** 하게 섞음
- 예: 입력이 [1, 2, 3, 4]라면 shuffle 결과 **같은 확률**로 아래 중 하나가 됨
 [1, 2, 3, 4] [1, 2, 4, 3] [1, 3, 2, 4] [1, 3, 4, 2] [1, 4, 2, 3] [1, 4, 3, 2]
 [2, 1, 3, 4] [2, 1, 4, 3] [2, 3, 1, 4] [2, 3, 4, 1] [2, 4, 1, 3] [2, 4, 3, 1]
 [3, 1, 2, 4] [3, 1, 4, 2] [3, 2, 1, 4] [3, 2, 4, 1] [3, 4, 1, 2] [3, 4, 2, 1]
 [4, 1, 2, 3] [4, 1, 3, 2] [4, 2, 1, 3] [4, 2, 3, 1] [4, 3, 1, 2] [4, 3, 2, 1]

[Q] Uniform Random Shuffle 결과 순서 안 바뀔 수도 있는가? (예: shuffle([1,2,3,4]) = [1,2,3,4])

yes (같은 확률 ↓)

[Q] 입력이 N개의 원소를 갖고 있다면 shuffle 결과 나올 수 있는 경우는 총 몇 가지인가? $N!$ 개
 또한 Uniformly random하게 섞는다면, 각각이 나올 확률은 무엇인가? $\frac{1}{N!}$

Shuffle (Random Permutation) 방법 배우는 이유?

Sorting 활용해 구현 가능 + 다양한 활용도

- Sorting 그대로 적용해 or Sorting과 유사한 방법으로
- Shuffle 기능 구현 가능

SW 테스트 위해
다양한 입력 데이터 발생시키고 주입하여
성능이 잘 나오는지, 오류는 없는지 등 관찰

게임 등에서 random sequence 발생시키기



다양한 랜덤 데이터 주입



암호화에 필요한 키 생성,
다양한 상황에 대한 시뮬레이션 등



>>> 이 외에도 많음 <<<

Naïve한 Shuffling 방법: 모든 가능한 Permutation 발생시킨 후 하나 선정

■ 예: 입력이 [1, 2, 3, 4]라면 아래 경우를 발생시킨 후 하나 선정

[1, 2, 3, 4] [1, 2, 4, 3] [1, 3, 2, 4] [1, 3, 4, 2] [1, 4, 2, 3] [1, 4, 3, 2]

[2, 1, 3, 4] [2, 1, 4, 3] [2, 3, 1, 4] [2, 3, 4, 1] [2, 4, 1, 3] [2, 4, 3, 1]

[3, 1, 2, 4] [3, 1, 4, 2] [3, 2, 1, 4] [3, 2, 4, 1] [3, 4, 1, 2] [3, 4, 2, 1]

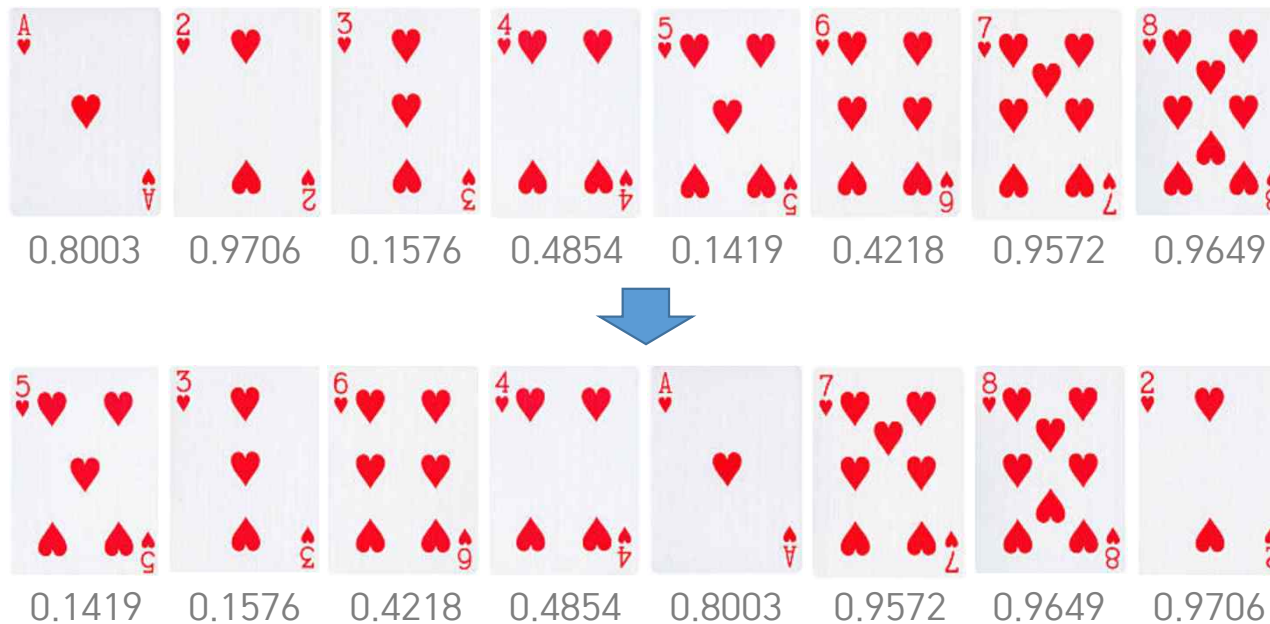
[4, 1, 2, 3] [4, 1, 3, 2] [4, 2, 1, 3] [4, 2, 3, 1] [4, 3, 1, 2] [4, 3, 2, 1]

[Q] 길이 N인 데이터를 shuffle해야 한다면 위 방법의 성능은 무엇에 비례하나? 이 방식은 효율적인가?

$N! \rightarrow$ ~~비효율적~~..

Sort를 직접 활용한 해결책(Shuffle Sort)

- 입력 데이터의 각 원소에 대해 random 실수 발생
- 발생한 실수 값 key로 사용해 정렬



N 개의 난수 생성: $\sim N$

N 개 정렬 (난수 5개당): $\sim N \log N$

Uniformly random 분포로부터 난수를 발생시킨다면, shuffle 결과도 uniformly random함

Shuffle Sort의 실제 활용 예

- MS Windows의 사용자가 브라우저를 선택하는 화면에서
- IE만 한 위치에 고정해 잘 보이도록 하자
- EU에서(유럽 연합) 여러 웹 브라우저 개발사에게 공정한 기회 주기 위해
- 브라우저 목록을 random하게 shuffle해 제공하도록 강제함
- 이를 위해 Shuffle Sort 사용

Select your web browser(s)



A fast new browser
from Google. Try it
now!



Safari for Windows
from Apple, the world's
most innovative
browser.



Your online security is
Firefox's top priority.
Firefox is free, and
made to help you get
the most out of the



The fastest browser on
Earth. Secure, powerful
and easy to use, with
excellent privacy
protection.



Designed to help you
take control of your
privacy and browse
with confidence. Free
from Microsoft.



```
import random

def shuffleSort(a):
    # Generate a random real number for each entry in a[]
    r = []
    for _ in range(len(a)):
        r.append(random.random())

    # Sort according to the random real number
    return [i for i, _ in sorted(zip(a, r), key=lambda x: x[1])]
```

(1) 입력 데이터 a의 길이만큼
난수 발생

(4) 정렬 결과에서 난수는 제거하
고 원래 a에 속한 원소만 반환

(2) a의 각 원소와 난수를
tuple로 결합 (예: [(1,0.1),
(2,0.8), (3,0.2), ...])

(3) 발생시킨 난수를
key로 사용해 정렬

[Q] 난수 발생에 constant time이 걸린다고 하자. 길이 N인 데이터를 Shuffle할 때, Shuffle Sort의 성능은?

↳ $N \log N$

Knuth's Shuffle (Uniformly Random, ???-time Shuffle)

Insertion Sort

- Iteration i 때 $a[i]$ 를 왼쪽 원소와 비교/swap해 가되, 크지 않은 원소 나올 때 까지 진행
- 그 결과 $a[0] \sim a[i]$ 가 정렬된 상태 됨

Knuth's Shuffle

- Iteration i 때 $a[0] \sim a[i]$ 중 하나를 uniformly random 하게 선정해 $a[i]$ 와 swap
- 그 결과 $a[0] \sim a[i]$ 가 uniformly random하게 shuffle 된 상태 됨

'O'는 정렬된 부분
'O'는 새로 주입된 원소

Iteration	Data
0	A L G O R I T H M
1	A L G O R I T H M
2	A G L O R I T H M
3	A G L O R I T H M
4	A G L O R I T H M
5	A G I L O R T H M
6	A G I L O R T H M
7	A G H I L O R T M
8	A G H I L M O R T

'O'는 shuffle된 부분
'O'는 새로 주입된 원소

칸안에서
드는 원소가
관용

Iteration	Data
0	A L G O R I T H M
1	L A G O R I T H M
2	L G A O R I T H M
3	L G A O R I T H M
4	L G A R O I T H M
5	L I A R O G T H M
6	L I A T O G R H M
7	L I A T O H R G M
8	M I A T O H R G L

Knuth's Shuffle은 결과는 왜 Uniformly Random한가?

- Knuth's Shuffle
- Iteration i 때 $a[0] \sim a[i]$ 중 하나를 uniformly random하게 선정해 $a[i]$ 와 swap
- 그 결과 $a[0] \sim a[i]$ 가 uniformly random하게 shuffle된 상태 됨

Iteration	Data
0	A L G O R I T H M
1	L A G O R I T H M
2	L G A O R I T H M
3	L G A O R I T H M
4	L G A R O I T H M
5	L I A R O G T H M
6	L I A T O G R H M
7	L I A T O H R G M
8	M I A T O H R G L

'O'는 shuffle된 부분
'O'는 새로 주입된 원소

[Q] $a[0]$ 는 uniformly random 한가? *yes*

[Q] $a[0] \sim a[1]$ 은 uniformly random 한가?

Knuth's Shuffle의 성능은 무엇에 비례하나?

- Insertion Sort
- Iteration i 때 $a[i]$ 를 왼쪽 원소와 비교/swap해 가되, 크지 않은 원소 나올 때 까지 진행
- 그 결과 $a[0] \sim a[i]$ 가 정렬된 상태 됨

- Knuth's Shuffle
- Iteration i 때 $a[0] \sim a[i]$ 중 하나를 uniformly random 하게 선정해 $a[i]$ 와 swap
- 그 결과 $a[0] \sim a[i]$ 가 uniformly random하게 shuffle 된 상태 됨

'O'는 정렬된 부분
'O'는 새로 주입된 원소

[Q] Insertion sort는 왜 한 번에 갈 자리까지 jump해 가지 못하고, 여러 차례 비교/swap 해야 하나?

Iteration	Data
0	A L G O R I T H M
1	A L G O R I T H M
2	A G L O R I T H M
3	A G L O R I T H M
4	A G L O R I T H M
5	A G I L O R T H M
6	A G I L O R T H M
7	A G H I L O R T M
8	A G H I L M O R T

Iteration Data

0	A L G O R I T H M
1	L A G O R I T H M
2	L G A O R I T H M
3	L G A O R I T H M
4	L G A R O I T H M
5	L I A R O G T H M
6	L I A T O G R H M
7	L I A T O H R G M
8	M I A T O H R G L

'O'는 shuffle된 부분
'O'는 새로 주입된 원소

[Q] 각 iteration에서 몇 번의 비교 필요한가?

110 비교, 대신 11번 swap

[Q] 각 iteration에서 몇 번의 swap 필요한가? 11번



▪ Knuth's Shuffle

- Iteration i 때 $a[0] \sim a[i]$ 중 하나를 uniformly random하게 선정해 $a[i]$ 와 swap
- 그 결과 $a[0] \sim a[i]$ 가 uniformly random하게 shuffle된 상태 됨

```
import random
```

```
def knuthShuffle(a):
```

```
    for i in range(1, len(a)):
```

```
        j = random.randint(0, i) # Randomly select a position among  $0 \sim i$ 
```

```
        a[i], a[j] = a[j], a[i] # Swap  $a[i]$ ,  $a[j]$ 
```

[Q] i 값의 범위가 $0 \sim N-1$ 이 아닌 $1 \sim N-1$ 인 이유는 무엇인가?

0번째를 이미 정렬되어 있기 때문

[Q] 실행 결과 데모 보기



Sorting (Shell Sort, Shuffle Sort, Convex Hull)

진보된 정렬 방법 이해하고 정렬의 적용 예 보기

01. 퀴즈 풀이 & 예습 내용 복습 (이번 주 #1~3차 문제 공개)
02. Shell Sort
03. Shuffle Sort (Shuffling에 정렬 방법 적용)
04. Convex Hull (Tight boundary 찾기 위해 정렬 방법 적용)
05. 실습: Convex Hull 구현

첫 인상은 정렬과 관련 없어 보일 수 있지만
정렬로 해결되며
그 성능 또한 정렬에 의해 결정되는 문제

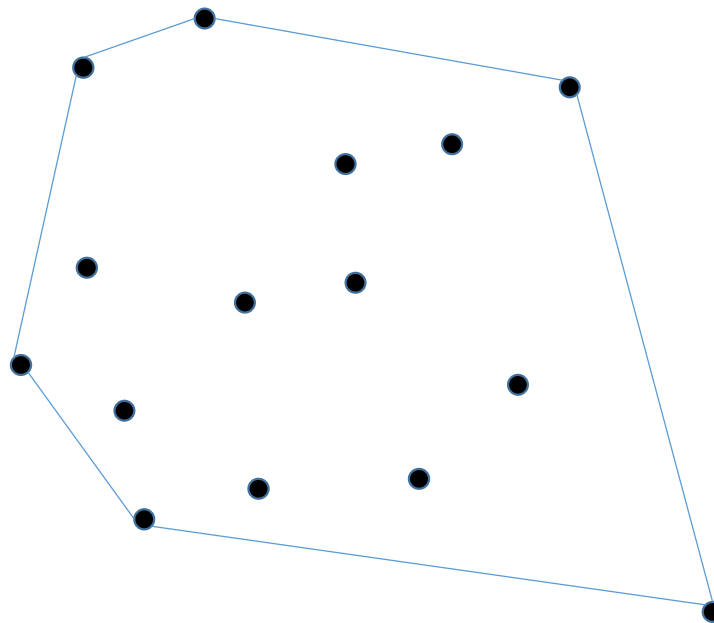
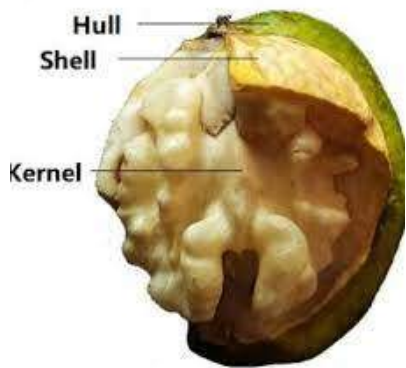
오목해지지않음.
6 ∴ 볼록한 껍데기

Convex Hull: 가장 작은 경계 찾기

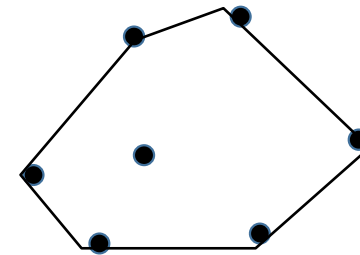
33

N개 정점 전체 혹은 일부 사용해 만들 수 있는 **다각형** 중 **정점 모두를 내부에 포함**하면서 **최소 정점**으로 만들 수 있는 다각형

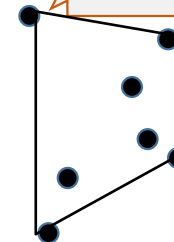
- Convex (볼록한) Hull (껍데기) of a set of N points:
- Smallest polygon that covers all given points** whose vertices are points in the set



[Q] convex hull을 찾으시오.



[Q] convex hull을 찾으시오.

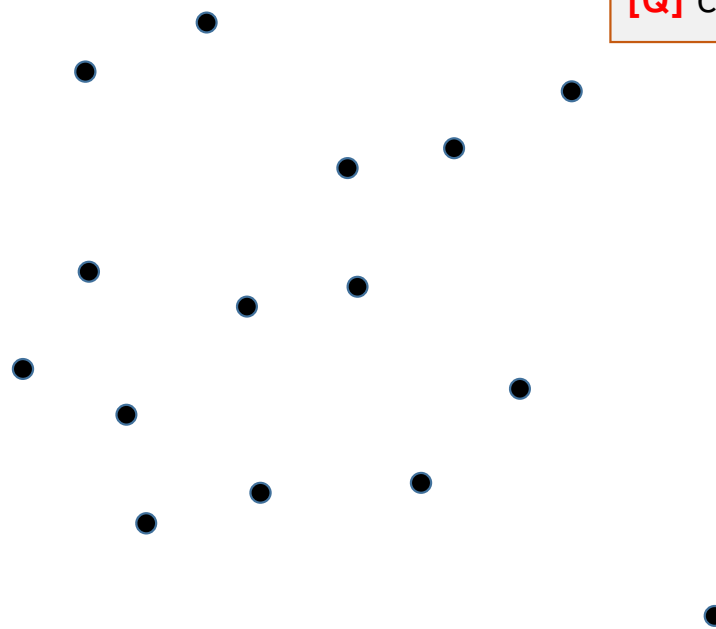
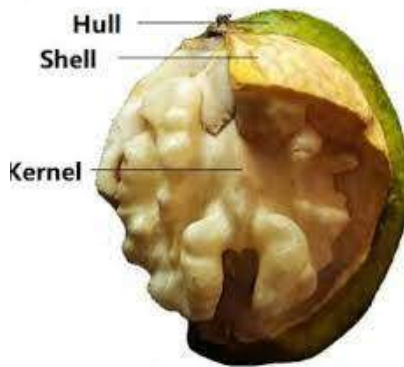


[Q] N개 점이 있을 때 convex hull에 포함되는 점의 개수는 최대 몇 개인가?

Convex Hull: 가장 작은 경계 찾기

N개 정점 전체 혹은 일부 사용해 만들 수 있는 **다각형** 중 **정점 모두를 내부에 포함**하면서 **최소 정점**으로 만들 수 있는 다각형

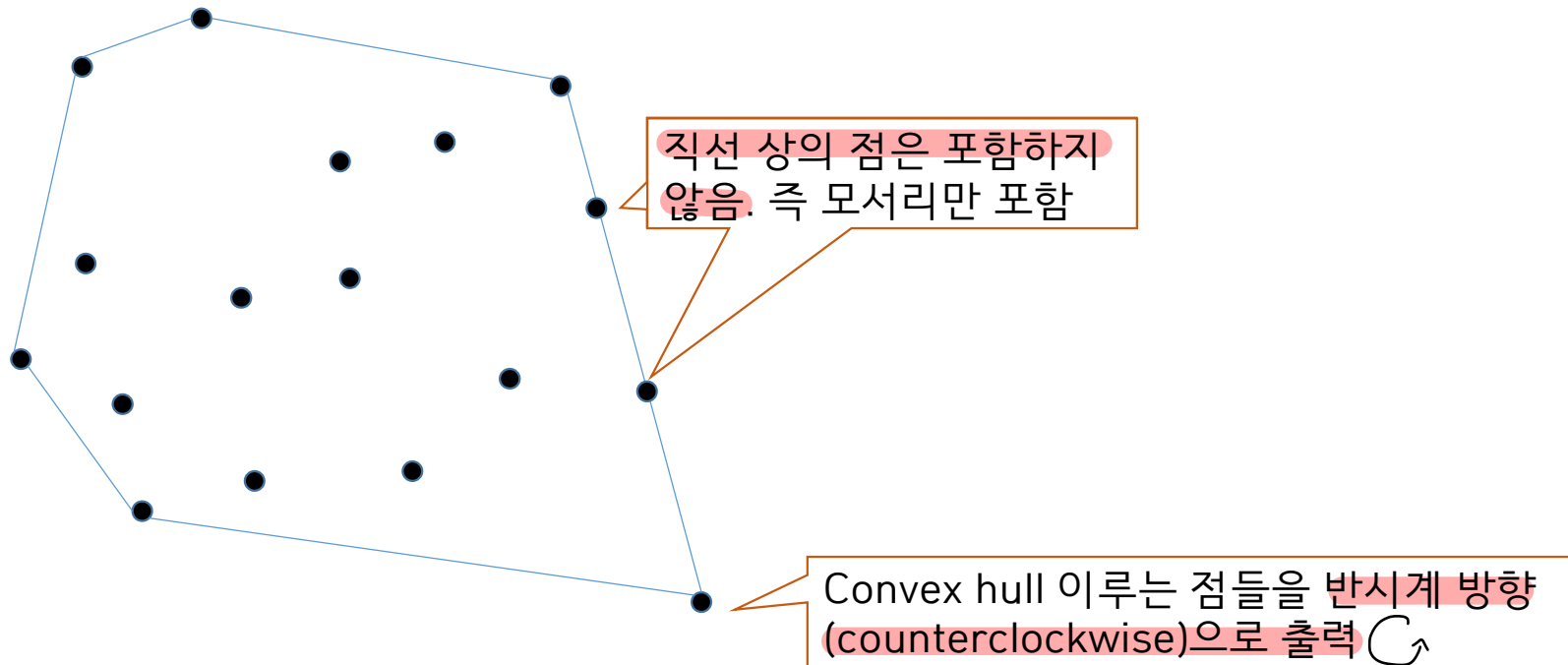
- Convex (볼록한) Hull (껍데기) of a set of N points:
- Smallest polygon that covers all given points** whose vertices are points in the set



[Q] convex hull이 일부라도 오목(concave)할 수 있을까?

Convex Hull 정의 & 출력 관련 유의사항

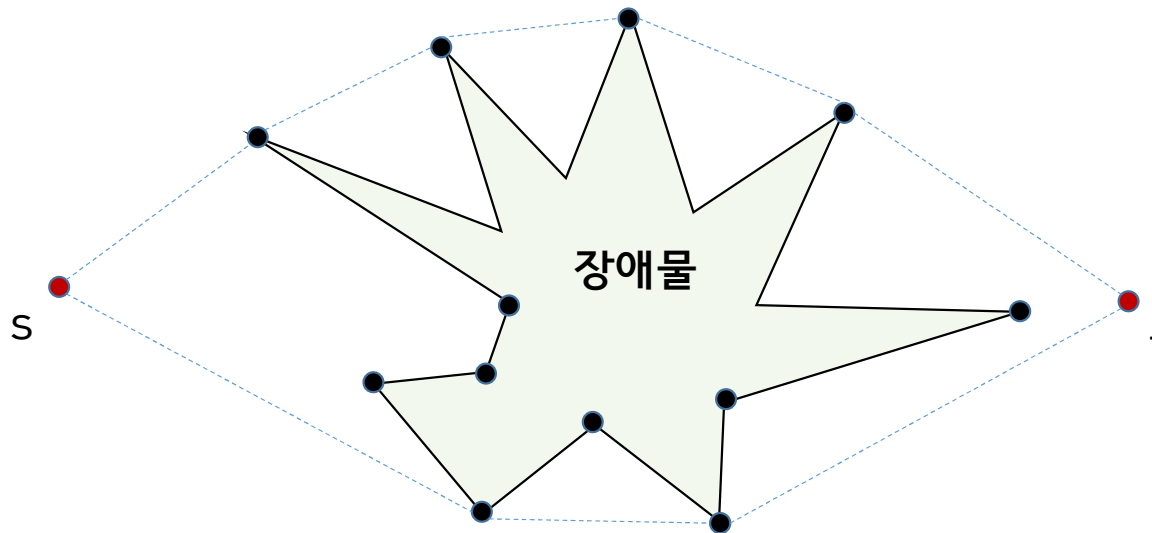
- Convex (볼록한) Hull (껍데기) of a set of N points
 - N 개 정점 전체 혹은 일부 사용해 만들 수 있는 **다각형** 중 **정점 모두를 내부에 포함**하면서 **최소 정점**으로 만들 수 있는 다각형
- Smallest polygon that covers all given points** whose vertices are points in the set



Convex Hull 활용 예: Robot Motion Planning

36

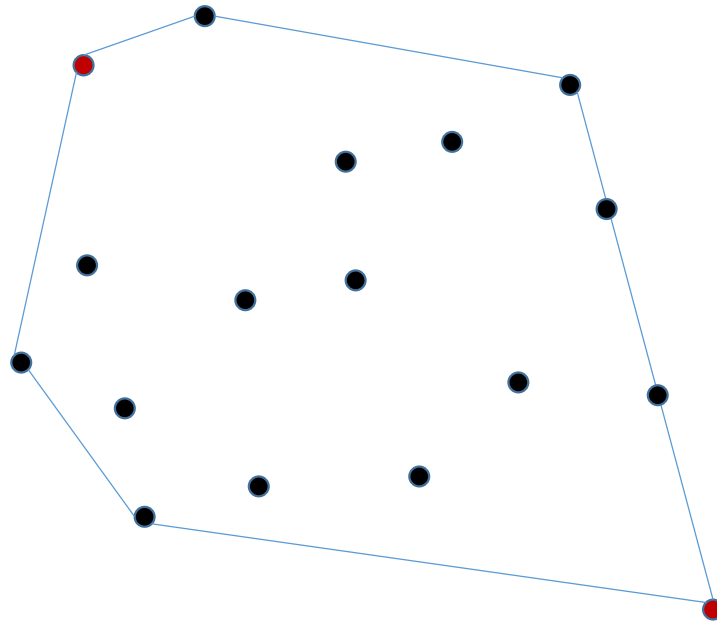
- Polygon(다각형) 형태 장애물 있을 때 두 지점 s와 t 간 최단경로 찾기



Convex Hull 활용 예: Farthest Pair 찾기

37

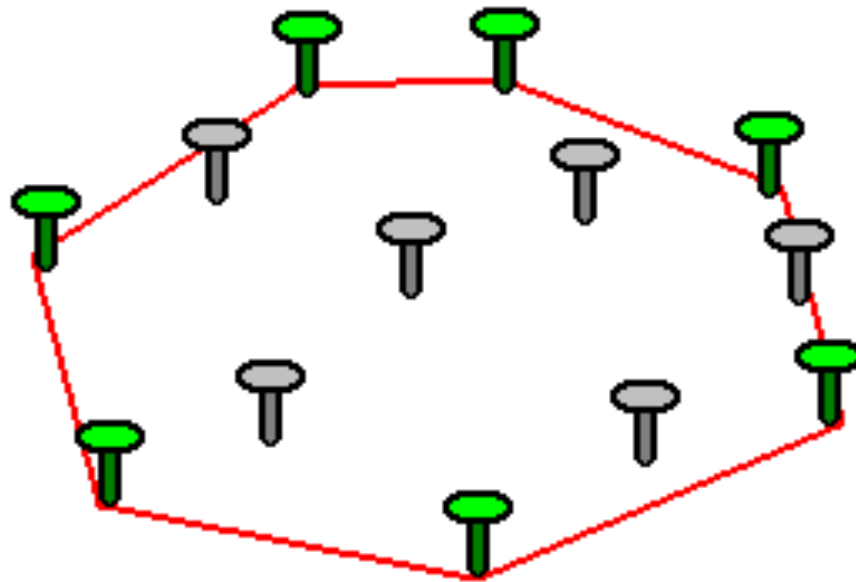
- 평면 상에 N개의 점이 있을 때, 서로 거리가 가장 먼 두 점 찾기





Convex Hull에 대한 기계적인 해결책

- N개의 점 위치에 못을 박고, 고무 밴드 두르기 ☺
- 컴퓨터 프로그램으로 구현하기에는 복잡함 (앞으로 배울 알고리즘이 더 간단)

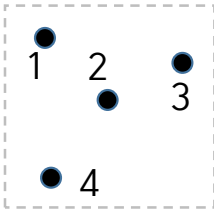


http://www.idlcoyote.com/math_tips/convexhull.html

무식한 st

Naïve한 Convex Hull 찾는 방법: N개 점에 대한 모든 가능한 부분집합 찾아 다른 점들 포함하는지 검사

39



다음 각 부분집합은 다른 모든 점을 포함하는가?

...

{1,2,3}

{1,2,4}

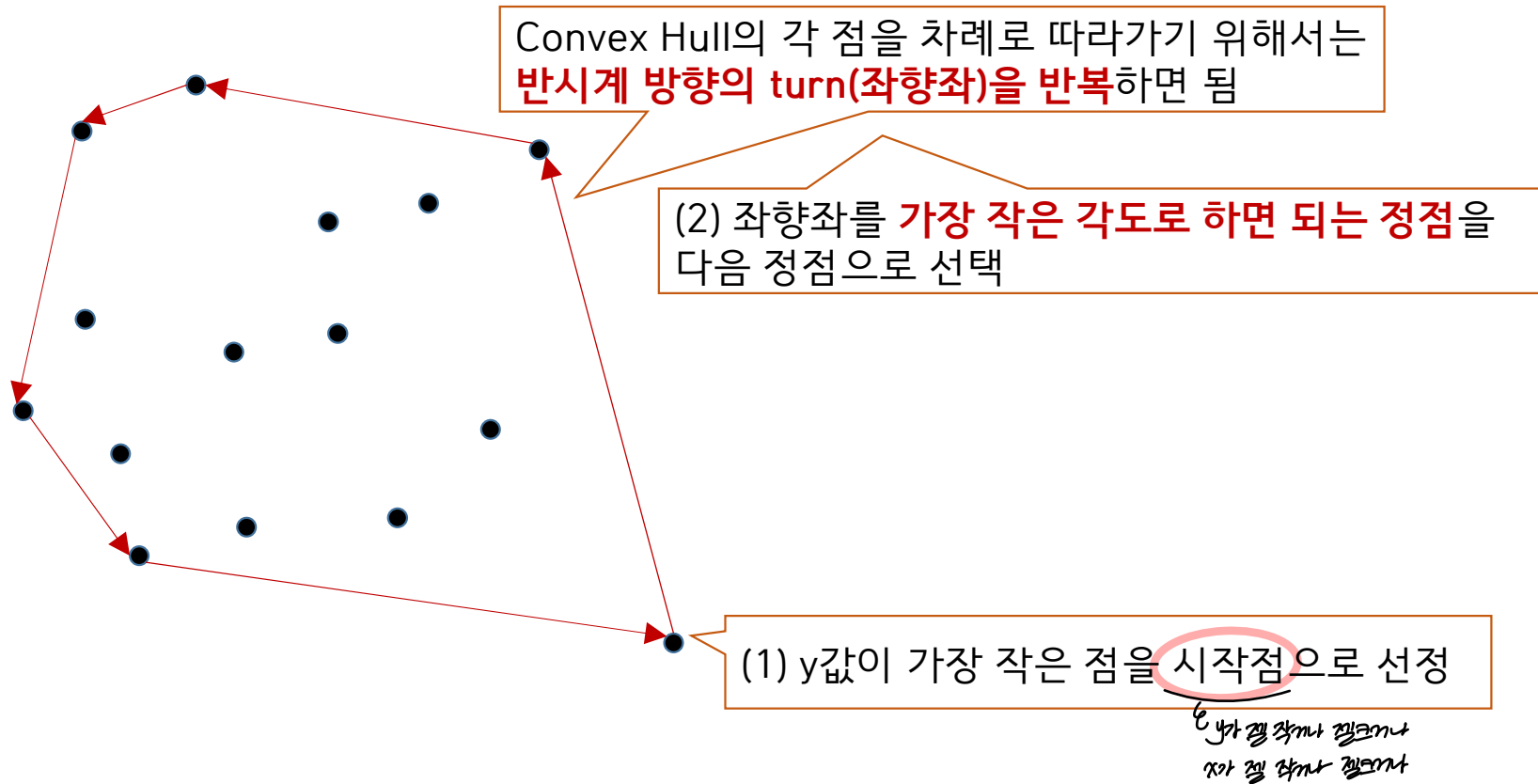
{2,3,4}

{1,2,3,4}

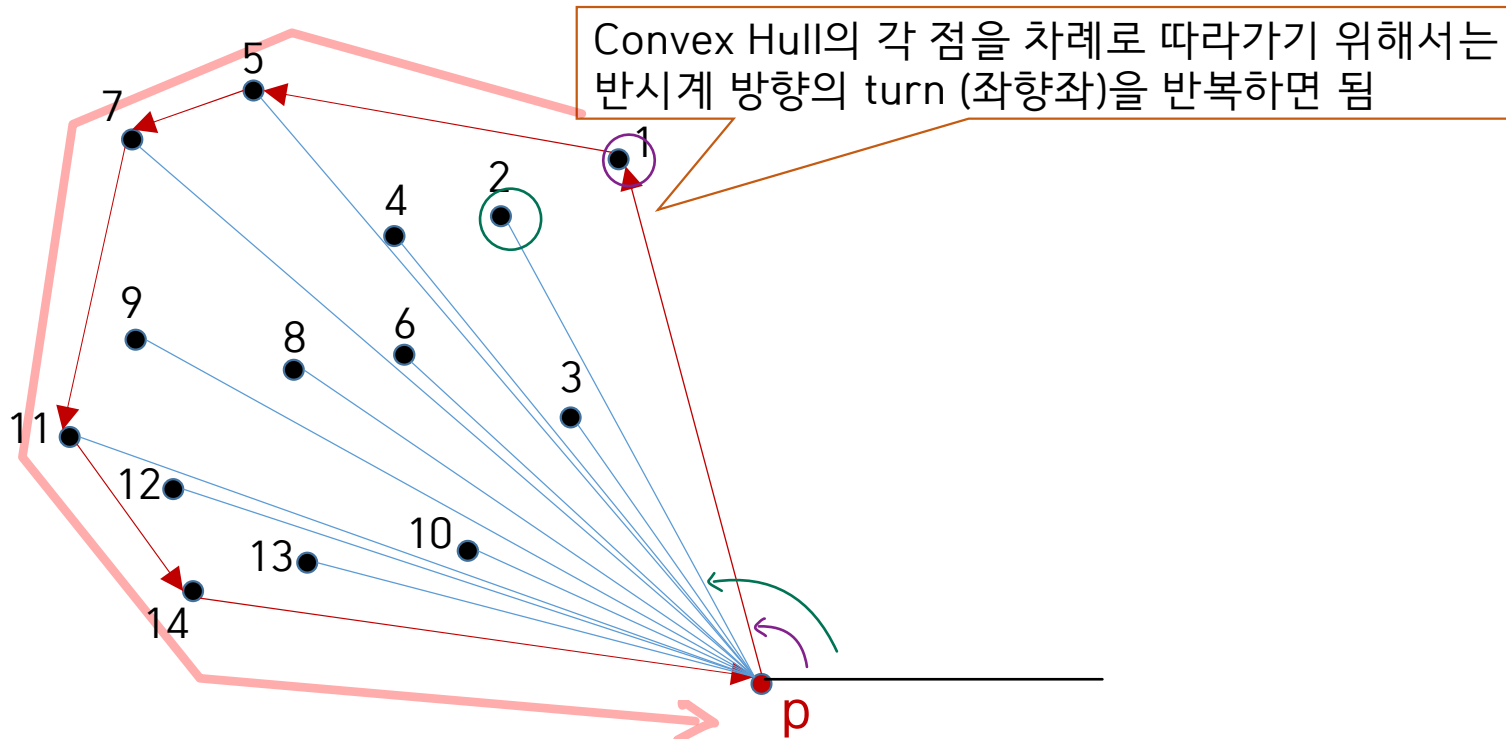
[Q] N개 점이 주어졌다면, 위 방법의 성능은 무엇에 비례하나? 이 방식은 효율적인가?

점 $\begin{cases} 포함 O \\ 포함 X \end{cases}$

Convex Hull에 대해 알아야 할 성질: 반시계 방향으로 포함할 점 선택



Convex Hull에 대해 알아야 할 성질: 반시계 방향 순서로 선택 = 최저점과 각도 순으로 선택



y값이 가장 작은 점을 p라 할 때,
convex hull 상의 점들을 **p로부터의 각도가 증가하는 순**으로 선택하면
convex hull을 반시계 방향으로 따라가는 순서가 됨

앞의 두 성질 활용한 첫 번째 방법: 반시계 방향 turn 반복 + 각도 순으로 선택

- y값 가장 작은 점 p에서 시작해
- 반시계방향 각도 가장 작은 점으로 연결
- 마지막에 a → b 순서로 선택했다면
- b로부터 a-b 라인과 반시계방향 각도 가장 작은 점으로 연결 반복



[Q] 왜 y값 가장 작은 점에서 시작하나?
y값이 가장 작은 점은 convex hull에 포함되나?

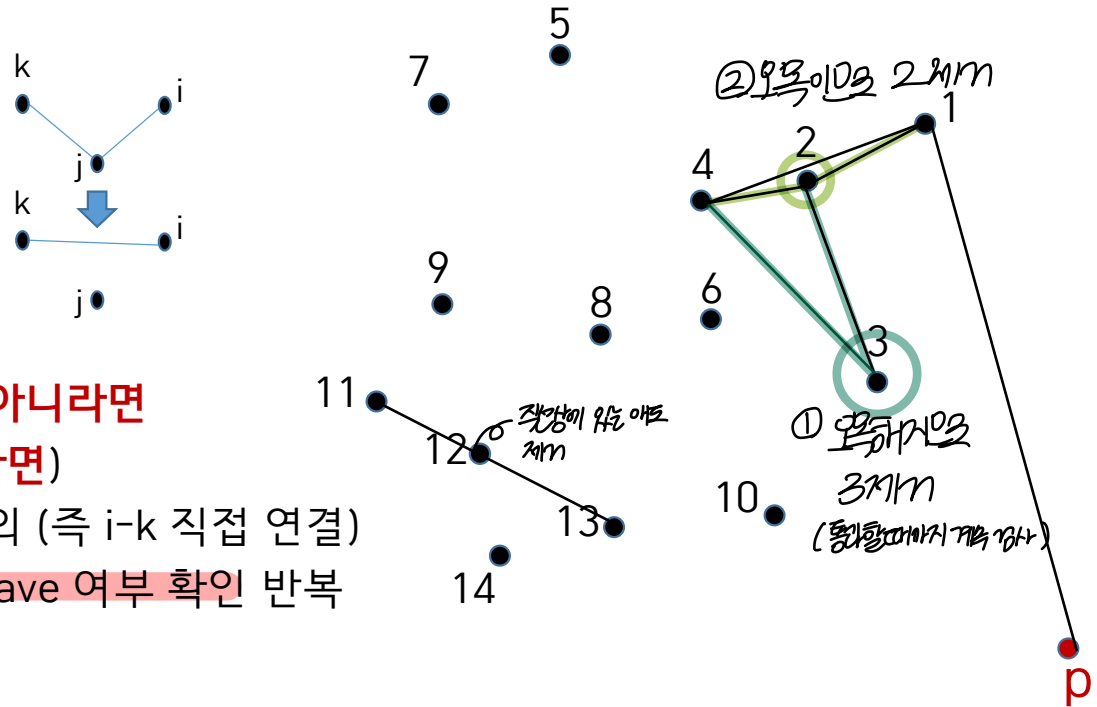
[Q] N개의 점 있을 때, 이 방법의 성능은 무엇에 비례하나?

$$\underbrace{N}_{\text{선택}} + (N-1) + (N-2) \dots = \frac{N(N-1)}{2} = \sim N^2$$

앞의 두 성질 활용한 두 번째 방법: Graham's Scan

- y값 가장 작은 점 p에서 시작해
- 다른 모든 점과의 반시계방향 각도 계산
- 각도가 작은 점 순으로 차례로 연결해 보며
- 새로운 점 연결할 때 마다 아래 수행
 - 마지막에 $i \rightarrow j \rightarrow k$ 순으로 연결했다면
 - $i-j$ 선 기준으로 볼 때 $j-k$ 가 반시계방향의 turn이 아니라면
 - (즉 $i-j-k$ 가 convex 아닌 concave angle을 만든다면)
 - j 는 잘못 연결된 것으로 보고 convex hull에서 제외 (즉 $i-k$ 직접 연결)
 - 더는 제외할 점 없을 때까지 마지막 세 점의 concave 여부 확인 반복

↳ 뿔 Test!

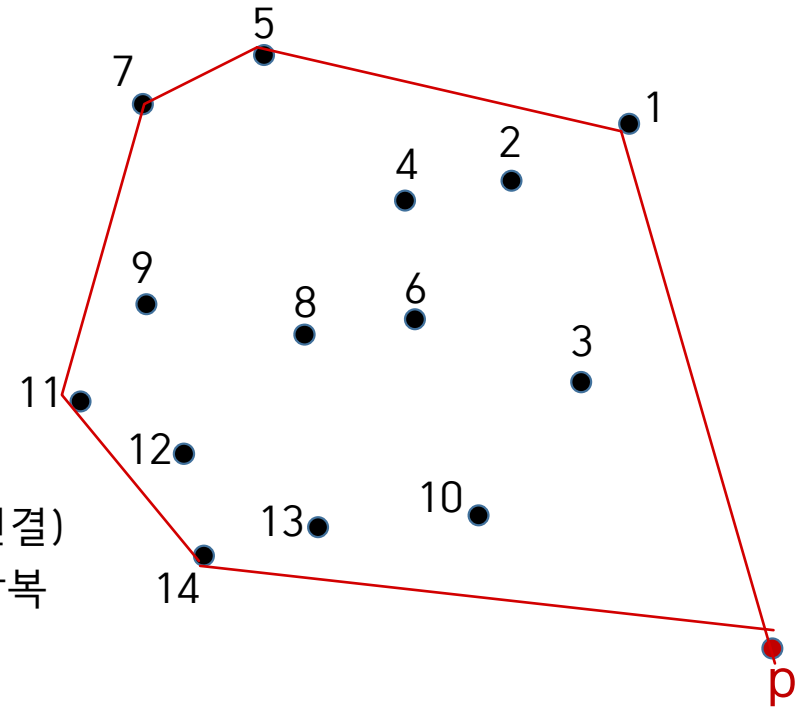
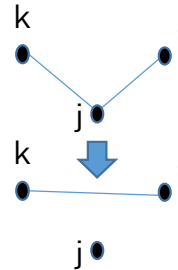


뿔이든 2차원
오목이면 버리기

Graham's Scan:

최저점으로부터 반시계방향 각도 순 연결 → concave angle 만드는 점은 제외

- y값 가장 작은 점 p에서 시작해
- 다른 모든 점과의 반시계방향 각도 계산
- 각도가 작은 점 순으로 차례로 연결해 보며
- 새로운 점 연결할 때 마다 아래 수행
 - 마지막에 $i \rightarrow j \rightarrow k$ 순으로 연결했다면
 - $i-j$ 선 기준으로 볼 때 $j-k$ 가 반시계방향의 turn이 아니라면
 - (즉 $i-j-k$ 가 convex 아닌 concave angle을 만든다면)
 - j 는 잘못 연결된 것으로 보고 convex hull에서 제외 (즉 $i-k$ 직접 연결)
 - 더는 제외할 점 없을 때까지 마지막 세 점의 concave 여부 확인 반복

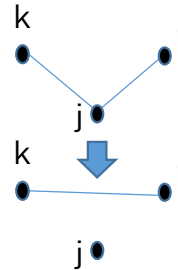


[Q] 이 방법에서 정렬은 언제 사용되나?

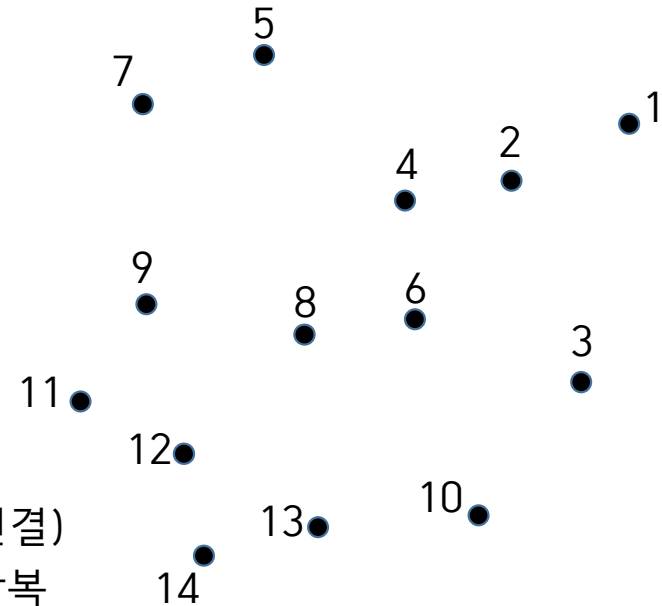
Graham's Scan:

최저점으로부터 반시계방향 각도 순 연결 → concave angle 만드는 점은 제외

- y값 가장 작은 점 p에서 시작해
- 다른 모든 점과의 반시계방향 각도 계산



- 각도가 작은 점 순으로 차례로 연결해 보며
- 새로운 점 연결할 때 마다 아래 수행
 - 마지막에 $i \rightarrow j \rightarrow k$ 순으로 연결했다면
 - $i-j$ 선 기준으로 볼 때 $j-k$ 가 반시계방향의 turn이 아니라면
 - (즉 $i-j-k$ 가 convex 아닌 concave angle을 만든다면)
 - j 는 잘못 연결된 것으로 보고 convex hull에서 제외 (즉 $i-k$ 직접 연결)
 - 더는 제외할 점 없을 때까지 마지막 세 점의 concave 여부 확인 반복



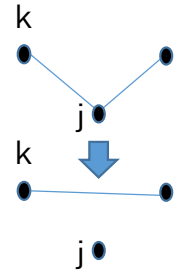
[Q] N개의 점 있을 때, 이 방법의 성능은 무엇에 비례하나?

$\sim N$

\because (추가 1회
제거 1회 \leftarrow 비교 1회

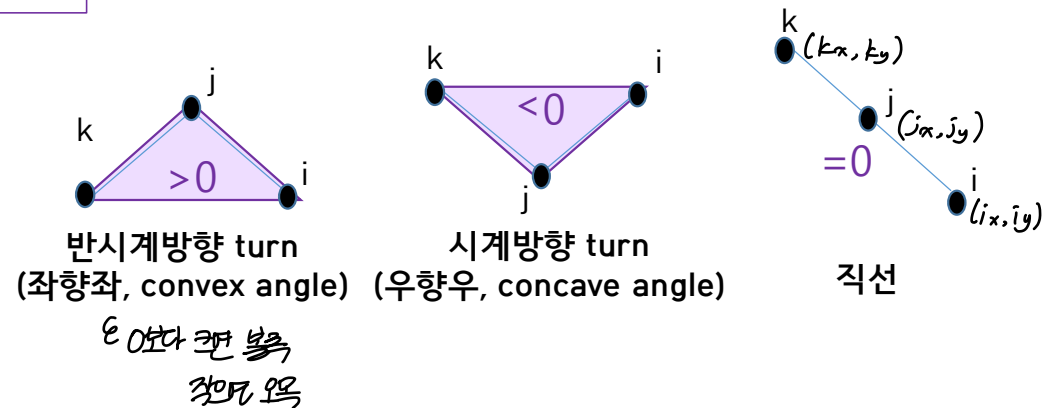
Graham's Scan: 반시계방향 turn (convex angle) 확인 방법

- y값 가장 작은 점 p에서 시작해
- 다른 모든 점과의 반시계방향 각도 계산
- 각도가 작은 점 순서로 차례로 연결해 보며
- 새로운 점 연결할 때 마다 아래 수행
 - 마지막에 $i \rightarrow j \rightarrow k$ 순으로 연결했다면
 - $i \rightarrow j$ 선 기준으로 볼 때 $j \rightarrow k$ 가 **반시계방향의 turn이 아니라면**
 - (즉 $i \rightarrow j \rightarrow k$ 가 **convex가 아닌 concave angle을 만든다면**)
 - j 는 잘못 연결된 것으로 보고 convex hull에서 제외 (즉 $i \rightarrow k$ 를 직접 연결)
 - 더 이상 제외할 점 없을 때까지 마지막 세 점의 concave 여부 확인 반복



$$2 \times \text{면적}(i, j, k) = \begin{vmatrix} i_x & i_y & 1 \\ j_x & j_y & 1 \\ k_x & k_y & 1 \end{vmatrix}$$

$$= (j_x - i_x)(k_y - i_y) - (j_y - i_y)(k_x - i_x)$$

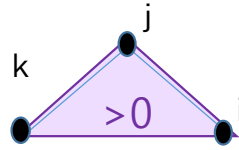


Non-trivial한 알고리즘은 거의 수학 활용 (다른 예: 기계학습) 수학의 중요성 잊지 마세요.

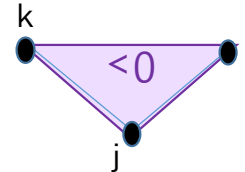


$$2 \times \text{면적}(i,j,k) = \begin{vmatrix} i_x & i_y & 1 \\ j_x & j_y & 1 \\ k_x & k_y & 1 \end{vmatrix}$$

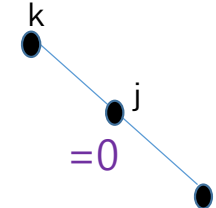
$$= (j_x - i_x)(k_y - i_y) - (j_y - i_y)(k_x - i_x)$$



반시계방향 turn
(좌향각, convex angle)



시계방향 turn
(우향각, concave angle)

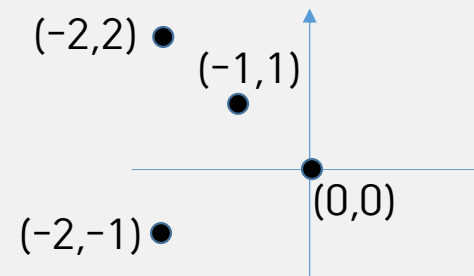


직선

```
# Determine if the line i->j->k is a counter-clockwise turn
# Each of i, j, and k is a 2-tuple (x coordinate, y coordinate)
def ccw(i, j, k):
    area2 = (j[0] - i[0]) * (k[1] - i[1]) - (j[1] - i[1]) * (k[0] - i[0])
    if area2 > 0: return True
    else: return False

if __name__ == "__main__":
    # ccw turns
    print(ccw((0,0), (-1,1), (-2, -1)))

    # non-ccw turns
    print(ccw((0,0), (-2, -1), (-1,1)))
    print(ccw((0,0), (-1, 1), (-2, 2))) # Straight line
```



[Q] 계산한 값이 0 일 때도 왜 False 반환하나? *같은 좌상점*

정리: Sort & Applications

- Shell **Sort**: Insertion Sort를 넓은 간격 → 좁은 간격 순서로 적용함으로써 성능을 향상시킨 예
 - 이미 잘 정렬된 알고리즘이라도 개선 여지 없는지 생각해 보자.
- Shuffle
 - 원소 수만큼 난수 발생 후 **Sort**
 - Insertion **Sort**와 유사한 방식으로 한 원소 씩 추가해 가되, 랜덤한 위치에 추가
- Convex Hull
 - **Sort**를 활용해 tight한 boundary 찾기
 - 처음에는 정렬과 관련 없어 보이지만, 정렬로 해결되는 문제
 - 앞으로도 정렬이 문제를 더 효율적으로 해결하는데 도움 되는 상황인지 생각해 보자.
- Sort의 적용 예 상당히 많음
- 이들은 Sort를 직접 활용하거나, Sort와 유사한 방법 활용하며
- **Sort의 성능이 방법의 성능에 큰 영향** 미침



Sorting (Shell Sort, Shuffle Sort, Convex Hull)

진보된 정렬 방법 이해하고 정렬의 적용 예 보기

01. 퀴즈 풀이 & 예습 내용 복습 (이번 주 #1~3차 문제 공개)
02. Shell Sort
03. Shuffle Sort (Shuffling에 정렬 방법 적용)
04. Convex Hull (Tight boundary 찾기 위해 정렬 방법 적용)
05. 실습: Convex Hull 구현

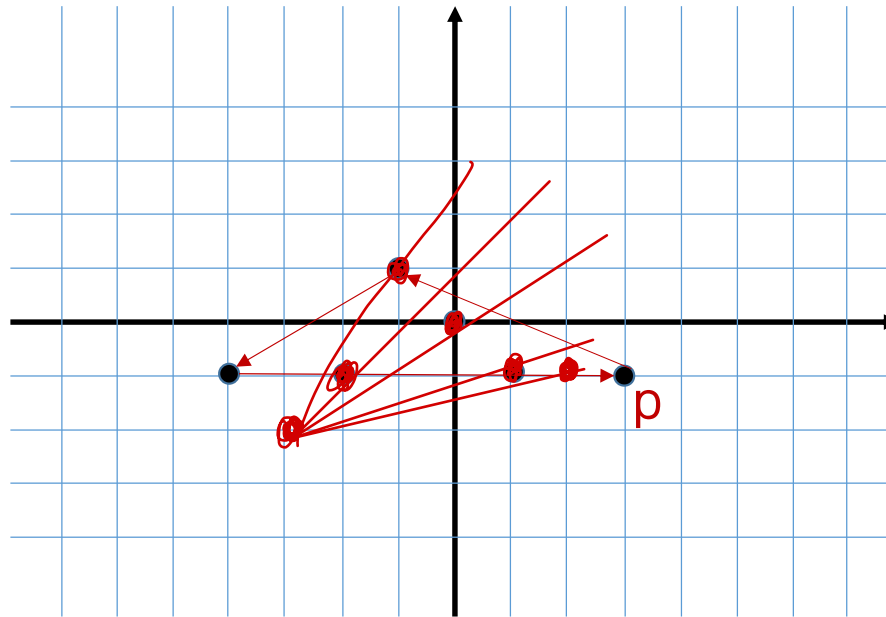
프로그램 입출력 조건

- xy 좌표계에 속한 점의 list(points)를 입력으로 받는 함수 정의
- points는 tuple (x,y)의 리스트임 (예: [(3,2), (4,-1), (0,0), (-2,2)]). 리스트 길이 ≥ 3 이며, 이 조건에 대한 검사&예외 처리를 할 필요는 없음
- points에 속한 점은 모두 좌표가 서로 다름
- ★ **def grahamScan(points):**
 - 위 함수는 Graham's Scan을 사용해 convex hull에 속한 점의 좌표를 구한 후 입력과 같은 형식으로 반환
 - 최초로 convex hull에 포함하는 점 p는 **y 값이 가장 작은 점 중 x 값이 가장 큰 점**
 - 이후에 convex hull에 포함하는 점은 p로부터 **반시계방향으로** 거쳐가는 차례로 포함되어야 함: *꼭 지켜야 함!*
- Hint: convex hull에 포함하는 점은 stack에 push, pop 하면 편리함 (optional)

프로그램 입출력 조건

- 최초로 convex hull에 포함하는 점 p 는 y 값이 가장 작은 점 중 x 값이 가장 큰 점
- 이후에 convex hull에 포함하는 점은 p 로부터 반시계방향으로 거쳐가는 차례로 포함되어야 함

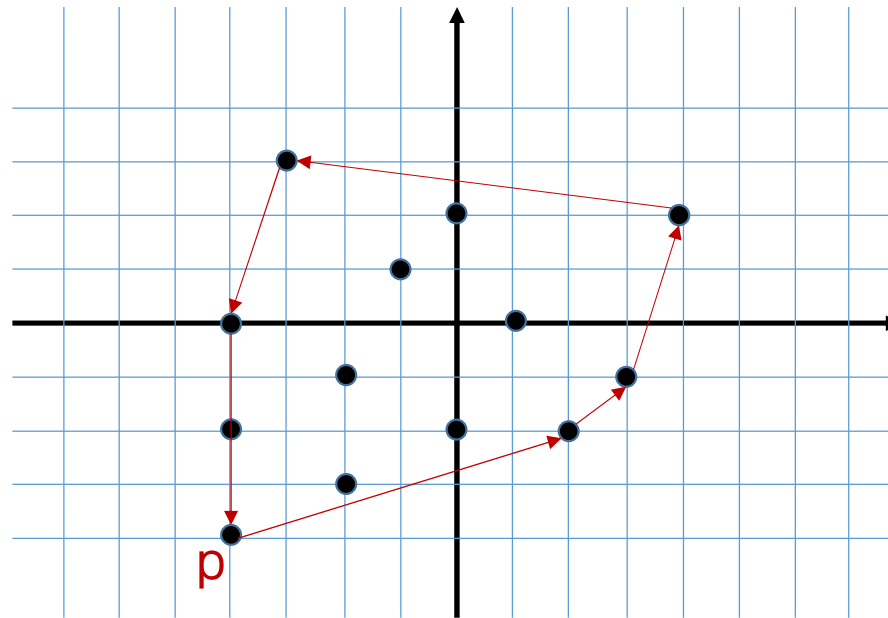
```
>>> print(ghamScan([(0,0),(-2,-1),(-1,1),(1,-1),(3,-1),(-3,-1)]))
[(3, -1), (-1, 1), (-3, -1)]
```



프로그램 입출력 조건

- 최초로 convex hull에 포함하는 점 p 는 y 값이 가장 작은 점 중 x 값이 가장 큰 점
- 이후에 convex hull에 포함하는 점은 p 로부터 반시계방향으로 거쳐가는 차례로 포함되어야 함

```
>>> print(gramhamScan([(4,2),(3,-1),(2,-2),(1,0),(0,2),(0,-2),(-1,1),(-2,-1),(-2,-3),(-3,3),(-4,0),(-4,-2),(-4,-4)]))
[(-4, -4), (2, -2), (3, -1), (4, 2), (-3, 3), (-4, 0)]
```

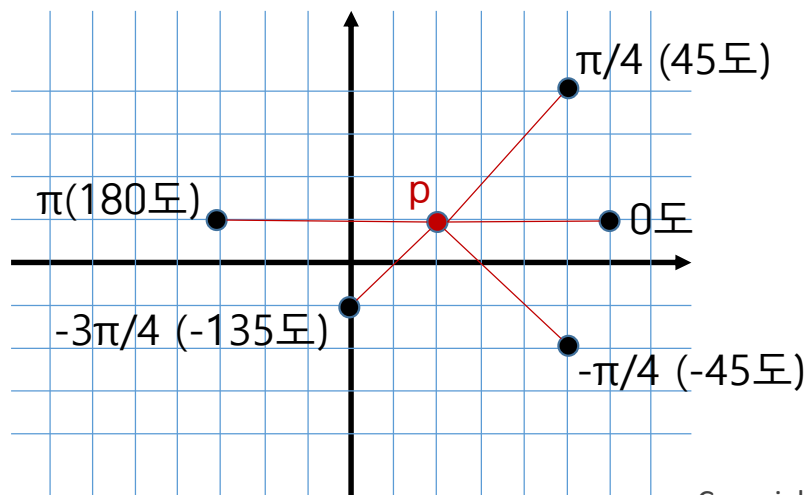




그 외 프로그램 구현 조건

53

- 정렬 기능 필요할 때: Python에서 제공하는 `sorted()` 함수 사용해도 되고, 수업에서 배운 정렬 함수 사용해도 됨
- 점 (px, py) 로부터 점 (ax, ay) 가 이루는 각도 계산
 - (px, py) 를 영점으로 보았을 때 (ax, ay) 가 이루는 각도 의미
 - 아래 그림의 예 참조
- `math.atan2()` 함수 사용 (arc tangent, tangent의 역함수 의미)
- `math.atan2(ay - py, ax - px)`가 radian 단위로 $(-\pi(-180\text{도}) \sim +\pi(180\text{도}))$ 사이의 각도 반환
좌표 평면에 점 p를 표시





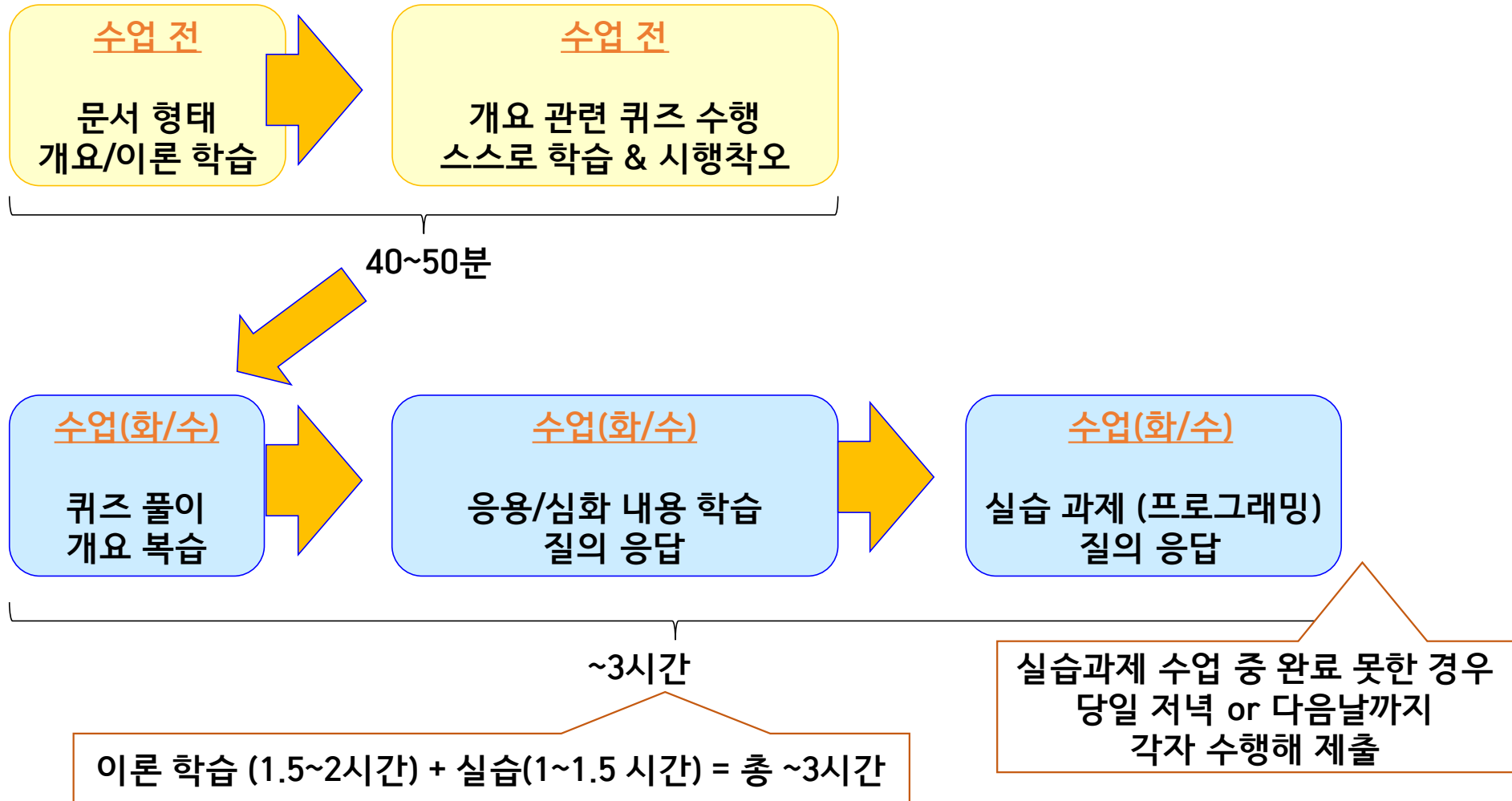
그 외 프로그램 구현 조건

54

- 작성한 코드는 파일 이름 **C**onvex**H**ull.py에 저장
- 최종 결과물로는 **C**onvex**H**ull.py 파일 하나만 제출하며, 이 파일만으로 코드가 동작해야 함
- import는 math만 할 수 있음
- 구현해야 하는 함수 def grahamScan() 이외에
- ✱ 각자 테스트에 사용하는 모든 코드는 if `__name__ == "__main__"`: 아래에 넣어서
 - 제출한 파일을 import 했을 때는 실행되지 않도록 할 것



스마트 출결





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
- **제출하면 기본 점수** 있으므로
- 그때까지 작성한 코드 **꼭 제출**하세요.
- 마감 시간 후에는 제출 불가능합니다.