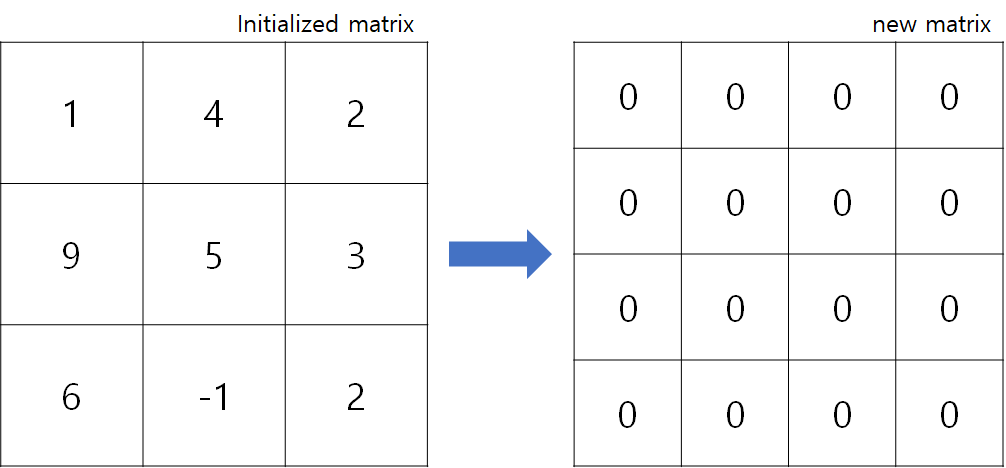
**고급소프트웨어실습I 보고서 3**

**20211584 장준영**

**A) Matrix Addition**

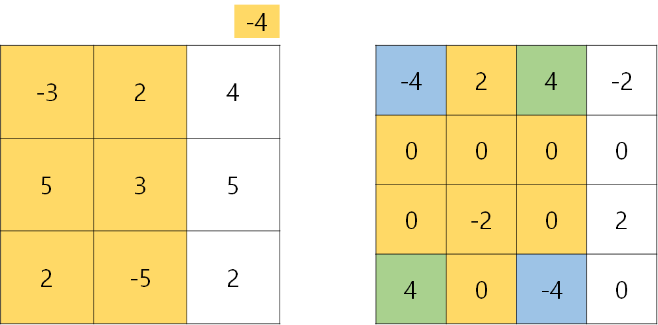
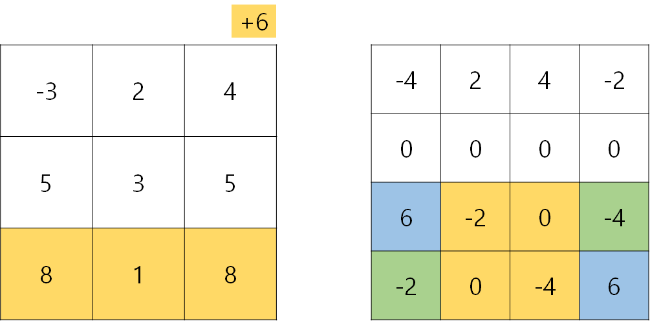
[1] 알고리즘 단순 개요



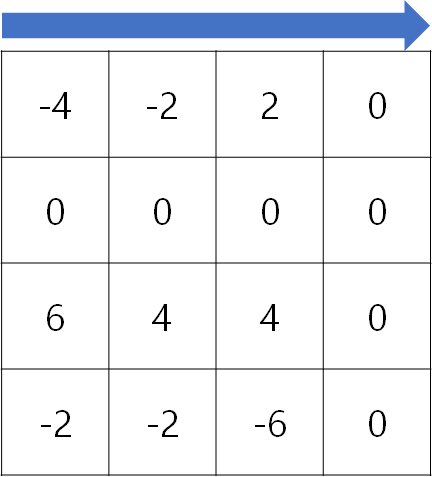
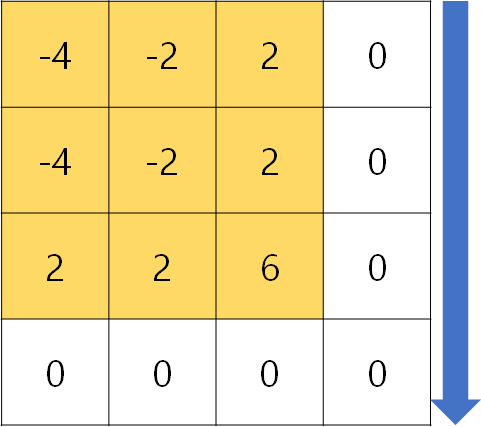
문제에서 주어진 행렬보다 행과 열이 각각 하나씩 더 많고 값이 0으로 초기화된 행렬을 생성해준다. 이곳에 더해질 값에 대한 정보를 저장할 것이다.



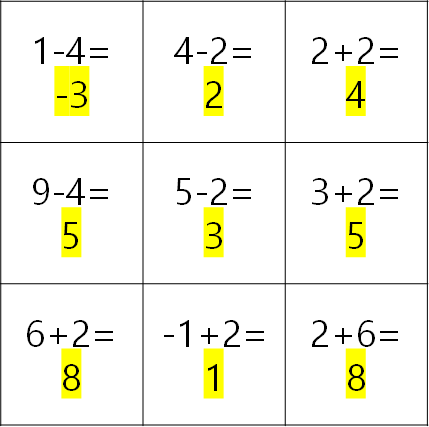
(1, 2, 2, 3, 2)의 질의가 주어진 상황이다. 이 경우 새로운 행렬의 [1][2], [3][4]는 +2를, [1][4], [3][2]에는 -2를 더해준다.

추가적으로 두 개의 질의가 주어진 상황이다. 일반화하면, (a, b, c, d, v)의 질의가 주어졌을 때 [a][c], [b+1][d+1]는 +v를, [b+1][c], [a][d+1]는 -v를 더해주면 된다.

모든 질의를 처리하고 완성된 새로운 행렬에서, 행 기준으로 누적합을 구한 이후 열 기준으로도 누적합을 구한다. 그럼 다시 크기의 행렬을 얻을 수 있다.



기존 행렬에 얻은 행렬을 더해주면 원하는 행렬을 구할 수 있다.

- 질의 처리 :

- 구해진 크기 행렬에 대해 행, 열 기준 누적합 계산 :

- 두 행렬을 덧셈 :

**>> 총 시간 복잡도 :**

[2] 설명

모든 행과 열에 대해 직접 덧셈하지 않고, 정보를 저장한 후 일괄적으로 처리해야 한다고 생각했다. 특정 구간에 덧셈을 이어가기 때문에 누적합을 역으로 이용해 의 연산을 에 처리할 수 있겠다는 아이디어를 떠올렸고, 처음에는 2차원에서의 일반화를 떠올리지 못해 행에 대해서만 처리해보았다. 행렬 중 하나의 행에 대해서, 그 행을 A[N]이라는 수열로 정의하면 새로운 수열 B[N]을 다음과 같이 정의할 수 있다.

이 수열 B[N]은 다음과 같은 성질이 있다.

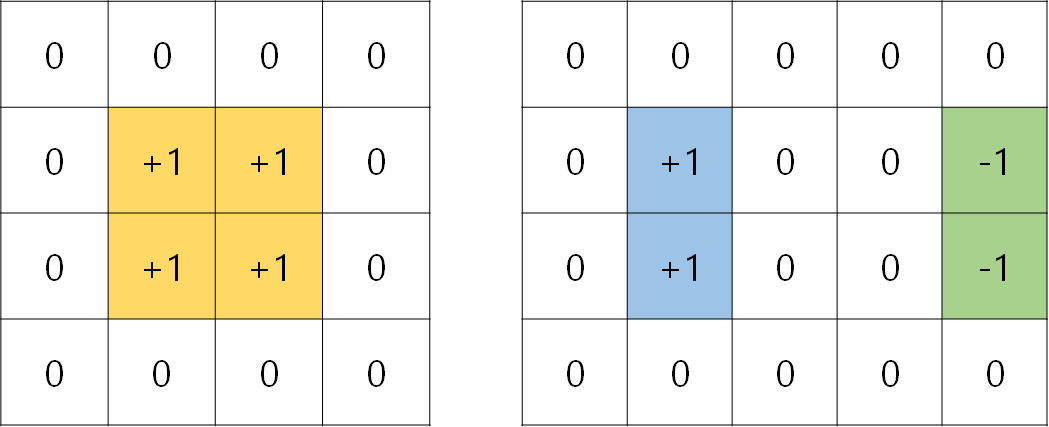
(1) A[i], A[i+1], …, A[j-1], A[j]에 k를 더하는 update를 B[i]+k, B[j+1]-k로 치환할 수 있다.

pf)

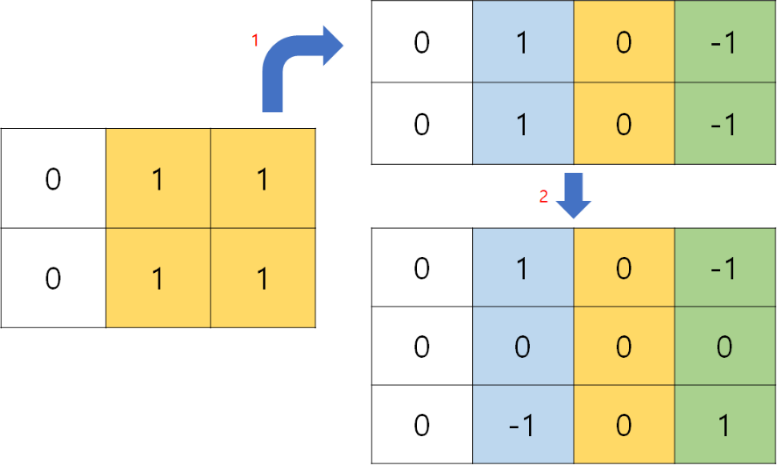
(2) A[i]의 값은 B[1]+B[2]+…+B[i-1]+B[i]이다.

pf)

따라서, A[N]에 직접 모든 값을 더하면 번 반복되는 루프에서 번의 덧셈을 해 총 번의 연산을 해야 하지만 B[N]에 정보를 저장하면 번의 연산만으로 모든 질의를 처리할 수 있다. 이후, 모든 행에 대해 구한 B[N]에서 각각의 누적합을 구해주면 번의 연산에 질의 정보가 복원된다.



하지만, 이를 2차원에 적용해보았더니 B[N]을 수열이 아닌 행렬로 일반화할 수 있다는 것을 알게 되었다.

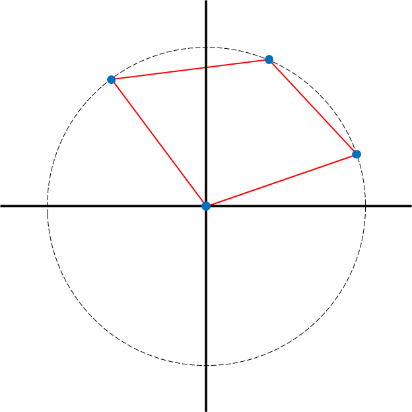
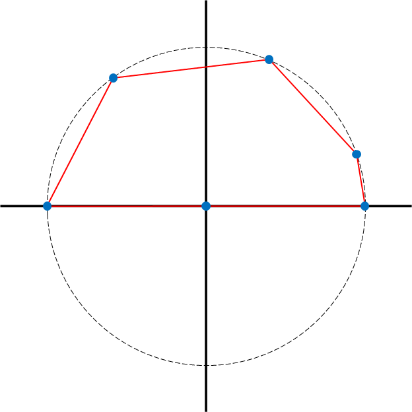
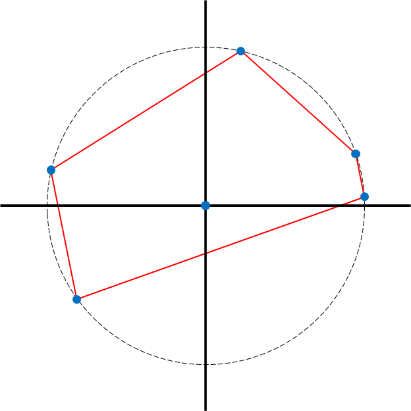


1번 화살표는 기존 행렬에서 행 기준 B[N]을 만든 것이다. 이를 확인해보면, 2차원 구간에 대해 열 기준으로 연속되는 값을 확인할 수 있다. 모든 행이 같은 열 구간동안 값이 변화하기 때문에, 생성된 B[N] 행들의 모임 또한 열 기준 누적합으로 표현할 수 있는 형태가 된다. 이 방식을 사용하면 질의 처리를 번의 연산으로 마치고, 번의 연산으로 질의 정보를 복원할 수 있다.

**B) Half-Circle Property**

**(※ B번에 대한 최적의 알고리즘은 [3]에 작성되어 있습니다. [1], [2]는 사고 과정의 일부이니 읽지 않고 넘어가셔도 괜찮습니다.)**

[1] 알고리즘 단순 개요

S의 점과 원점 O에 대해 볼록 껍질Convex Hull을 만든다. 만들어진 볼록 껍질을 구성하는 점 중 원점이 존재한다면 half-circle property를 만족하고, 그렇지 않다면 만족하지 않는다. 예를 들어, 위 그림의 1, 2번 S는 half-circle property를 만족하고, 3번 S는 만족하지 않는다. (2번 그림은 기준에 따라 결과가 달라질 수 있다.) 총 시간 복잡도는 다음과 같다.

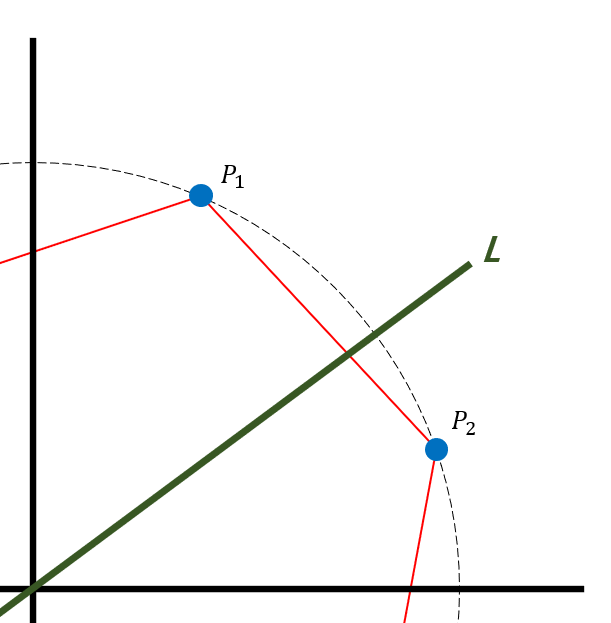
- 볼록 껍질을 만들기 위해 점을 기준점 기준 정렬 :

- Graham’s Scan :

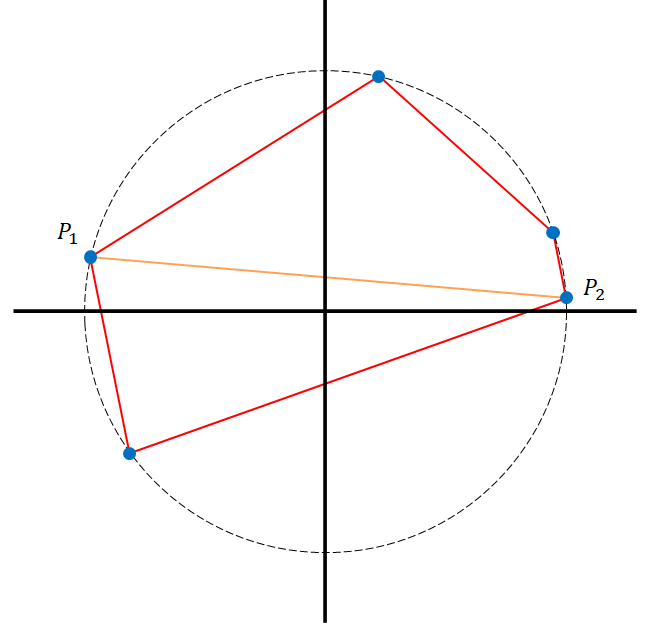
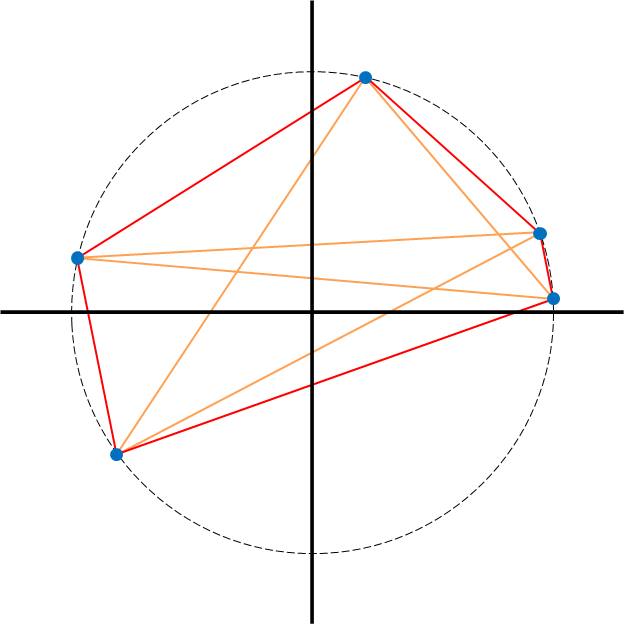
- 원점이 포함되는지 확인 :

**>> 총 시간 복잡도 :**

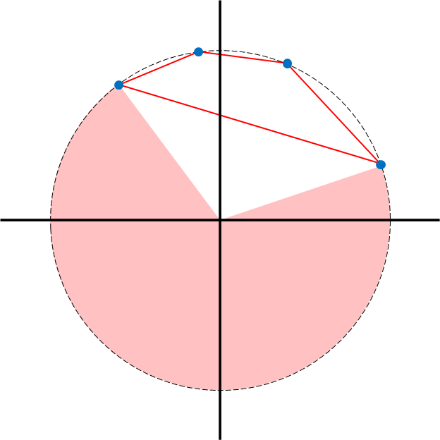
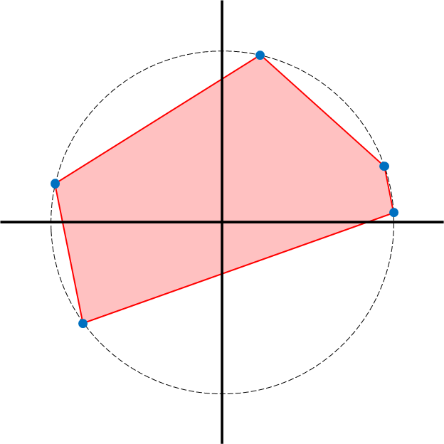
[2] 설명



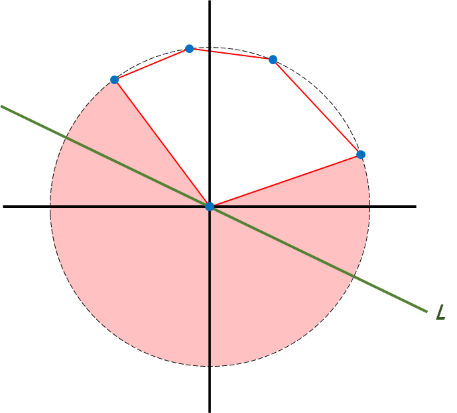
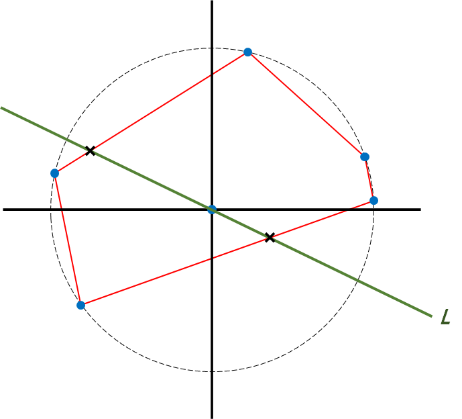
S의 점들이 L 기준 한 쪽으로 몰리지 않는 경우는 위와 같다. S에 포함된 임의의 두 점 , 를 이은 선과 L이 교차하는 경우, 그 두 점은 L 기준 한 쪽으로 몰 수 없다. L이 S의 모든 점을 한 쪽으로 몰기 위해선, S에 포함된 임의의 두 점을 이은 어떠한 선분도 L과 교차하지 않아야 한다. S는 고정값이고 L이 가변적인 요소이므로, 원점 O에서 어떠한 기울기로 직선 L을 그려도 포함된 임의의 두 점을 이은 선분과 교차하게 되는 S는 half-circle property를 만족하지 않고, 그렇지 않으면 만족한다. 하지만 S의 모든 점의 조합으로 선분 교차 확인을 하면 번의 확인을 해야하므로, 범위를 줄여 확인해야 하는 선분을 최소화해야 한다.

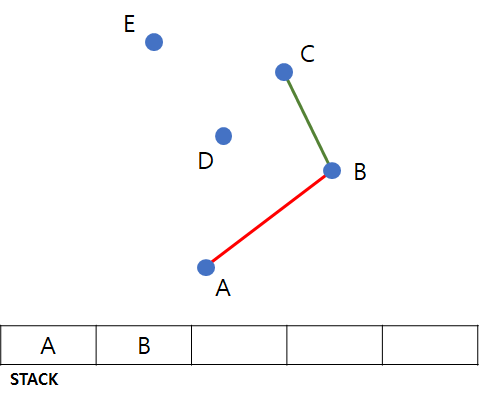
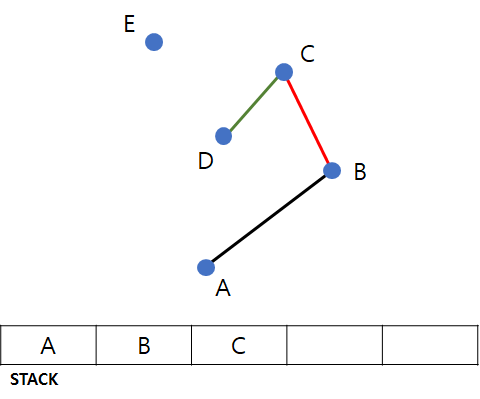
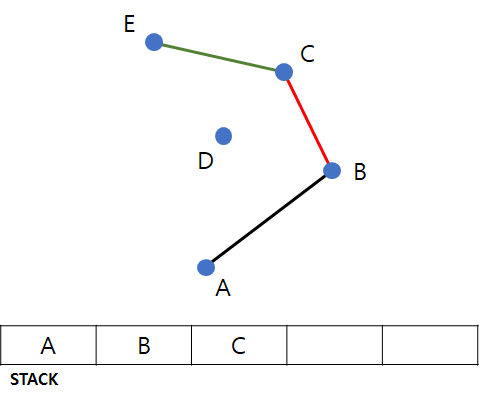
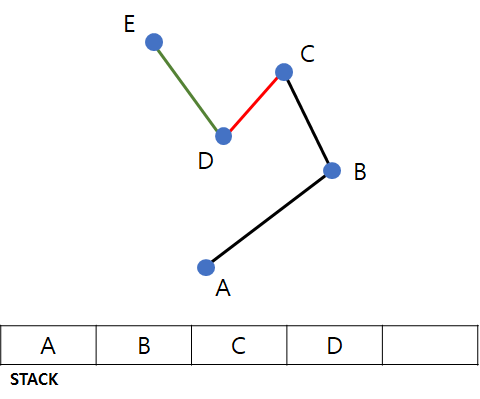
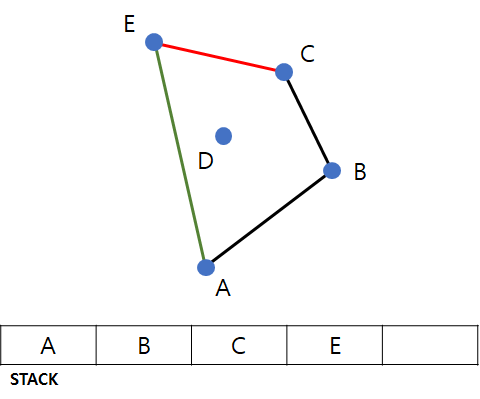
빨간 선분은 볼록 껍질에 해당하는 선분이다. 좌표평면의 모든 직선 중 점 과 를 이은 주황색 선분과 교차하는 직선은 반드시 빨간 선분과도 교차한다. S의 어떤 점의 조합도 이 성질을 만족하므로, 어차피 확인해야 하는 볼록 껍질의 선분만 확인해도 다른 조합의 선분 또한 확인할 수 있다.

원점이 S로 만든 볼록 껍질의 외부에 있다면 기울기를 조절해 모든 점을 한 쪽으로 모는 L을 그을 수 있고, 내부에 있다면 어떻게 그어도 선분과 교차하기 때문에 그을 수 없다. 우리가 잇는 직선 L의 고정점은 원점이고 S에는 원점을 원의 중심으로 갖는 점만이 포함되므로, 내부인지 외부인지 확인하기 위해 원점까지 포함시켜 볼록 껍질을 만들면 쉽다. 원점이 볼록 껍질 외부의 점이 된다면, 그 점까지 포함하기 위해 볼록 껍질의 점에 포함시킬 것이기 때문이다.

볼록 껍질을 찾기 위해 의 시간 복잡도를 갖는 Graham’s scan을 사용한다. 대략적인 방법은 다음과 같다.

모든 점은 A를 기준으로 반시계 방향으로 정렬되었다. 우선 처음의 두 점 A, B를 스택에 넣는다. 다음 점인 C와 스택의 top의 두 점 A, B가 반시계 방향으로 볼록한지 확인한다. 이 경우 그렇기 때문에 스택에 C를 넣는다. 다음 점인 D에 대해서도 마찬가지로 B, C, D가 반시계 방향으로 볼록하므로 스택에 D를 넣는다. 하지만 E의 경우 C, D, E가 시계 방향으로 볼록하므로 D를 스택에서 뽑는다. 이후 스택에 남은 B, C에 대해선 B, C, E가 반시계 방향으로 볼록하므로 이 상태에서 스택에 E를 넣는다. 마지막 A에 도달했을 때 여전히 C, E, A가 반시계 방향으로 볼록하므로 스택에 볼록 껍질의 모든 점이 구성된다. 점마다 최대 2번의 탐색만 하므로 의 시간 복잡도로 볼록 껍질을 찾을 수 있다.

**[3] 시간 및 공간 복잡도 개선**

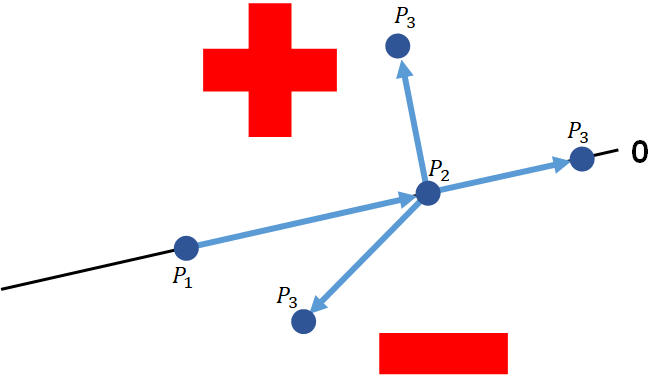
볼록 껍질을 만들어 해결하는 방식에는 다음과 같은 문제가 있었다.

1. 점을 정렬해야 하기 때문에 의 시간 복잡도가 필요하다. 정렬 자체의 시간 복잡도도 크고, 탐색하지 않아도 될 점을 탐색해야 하기 때문에 연산이 더 필요하다.

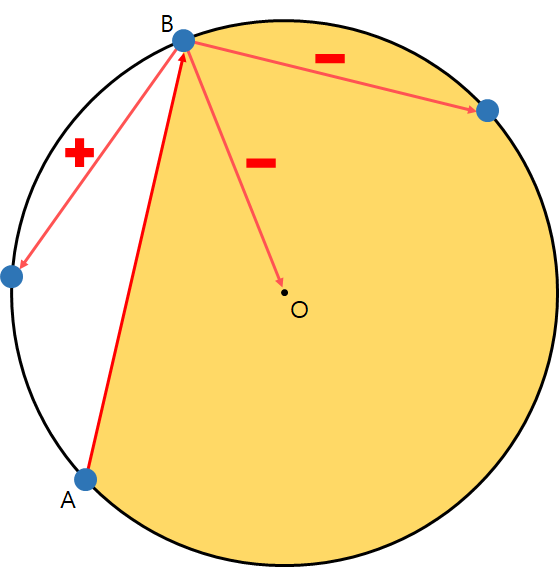
2. 점을 정렬해둔 vector, Graham’s scan에 이용할 stack 등의 자료구조를 새로 선언해야 하기 때문에 공간 복잡도가 많이 필요하다.

3. 직선 L이 존재하는지 체크하기 위해 볼록 껍질을 사용하기엔 구현이 너무 복잡하다. 선분 교차의 차원에서 바라보지 않고, 중요한 점만 확인하면 휴리스틱하게 탐색량을 줄일 수 있을 것이라고 생각했다.

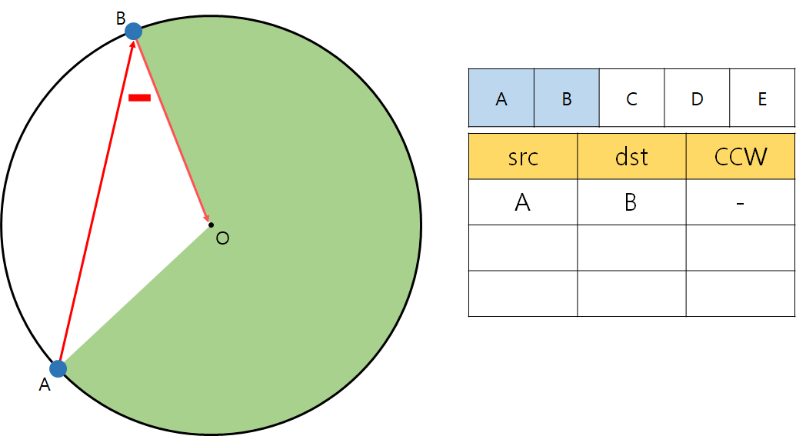
이 문제를 해결하기 위해 더 간결한 알고리즘을 생각하다가, 볼록 껍질 아이디어에서 얻은 인사이트를 사용하기로 하였다. 볼록 껍질에 원점이 포함되는 경우는, 가장 많이 벌어진 두 점 쌍의 각도가 이상 벌어져있는 경우이다. 다시 말해, **원의 의 중심각을 갖는 어떠한 호 위에는 점이 하나도 없어야 한다. 이러한 호를 찾기 위해 CCW 알고리즘을 사용할 것이다.** 투 포인터 알고리즘과 비슷하게, 사이에 점이 없는 가장 큰 각을 알기 위해 그 호의 양 끝 점을 저장하고 점점 좁히는 방식으로 문제를 해결할 것이다.



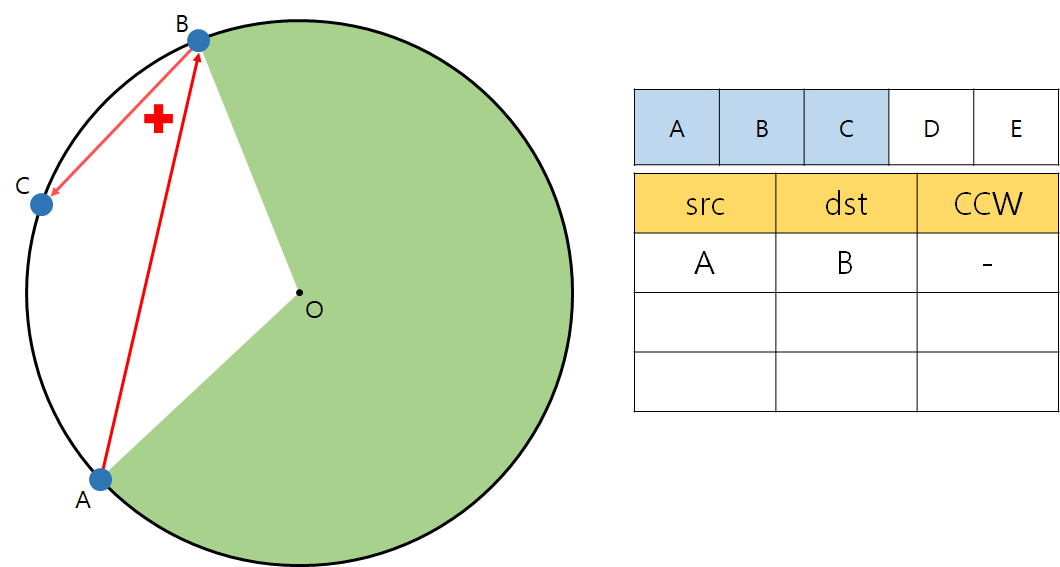
CCW의 증명과 원리 대신 동작을 대략적으로 설명하면, 점 세 개를 이은 선의 방향이 반시계 방향일 때 양수 값을, 시계 방향일 때 음수 값을, 직선일 때 0을 반환한다.



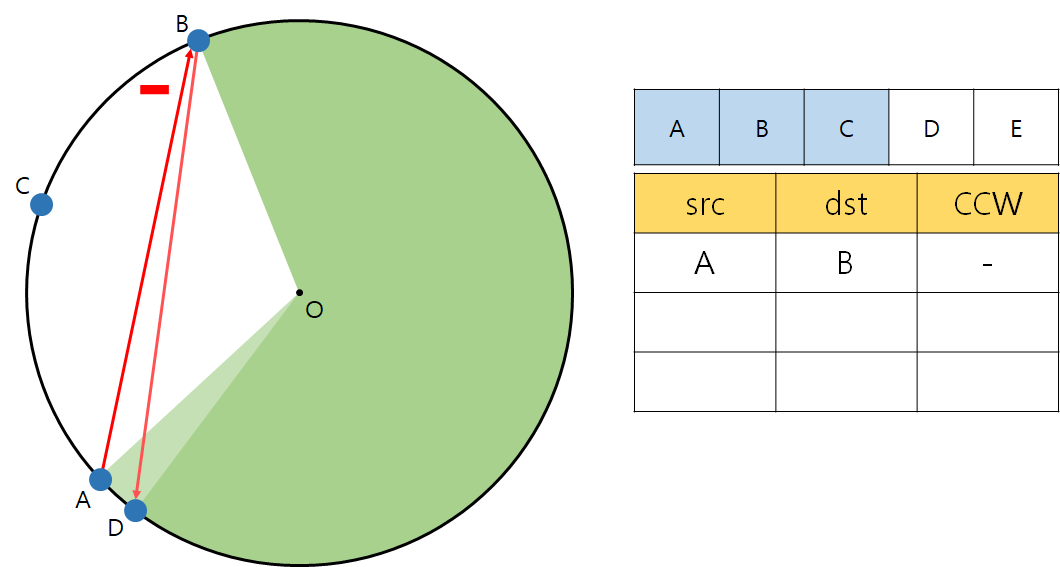
만약 원 위의 두 점 A, B가 존재한다면, 그 두 점으로 인해 생기는 호는 두 가지이다. 원 위의 다른 점 X에 대해, CCW(A, B, X)가 양수인 경우와 음수인 경우 두 개로 영역을 반으로 나눌 수 있다. 우린 두 영역 중 각의 크기가 큰 영역에서 시작하여 조일 것이기 때문에, 큰 영역에 대한 정보를 저장해주어야 한다. 그 큰 영역에는 반드시 원점이 포함되어 있으므로, 점 A, B, 그리고 CCW(A, B, O) 이 세 가지 정보만 있으면 점을 포함하지 않는 가장 큰 내각의 호를 저장할 수 있다.



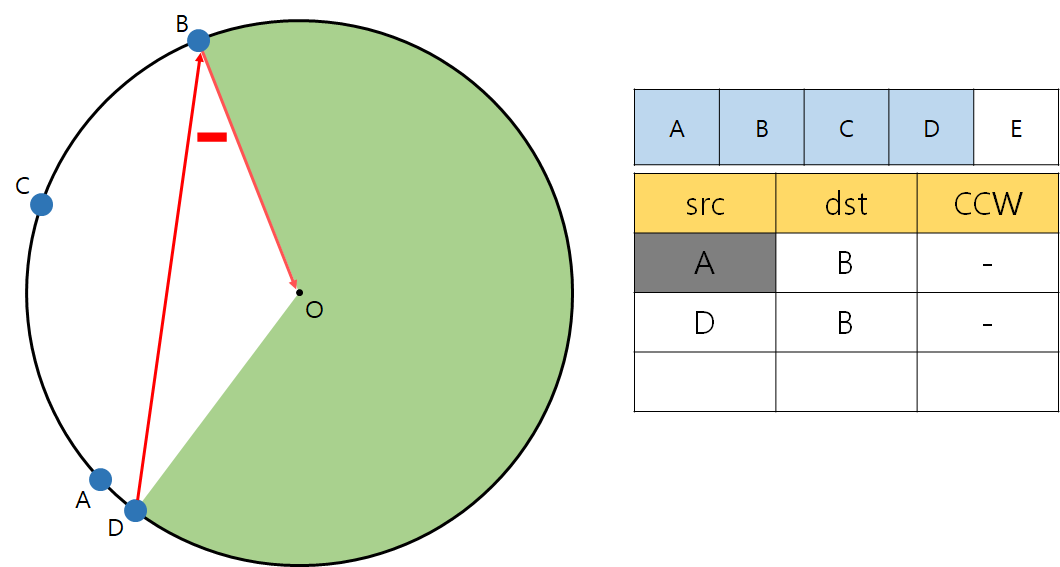
점의 좌표를 순차적으로 입력 받는다고 가정하였다. 먼저, 가장 먼저 입력되는 두 점은 첫 번째를 src, 두 번째를 dst에 저장하고, CCW(src, dst, O)의 초기값을 정한다. 그림의 예시에선 -이다. 그렇게 되면, 현재 초록색으로 칠해진 영역에 대해서만 탐색하면서 가장 큰 각도를 찾으면 된다.



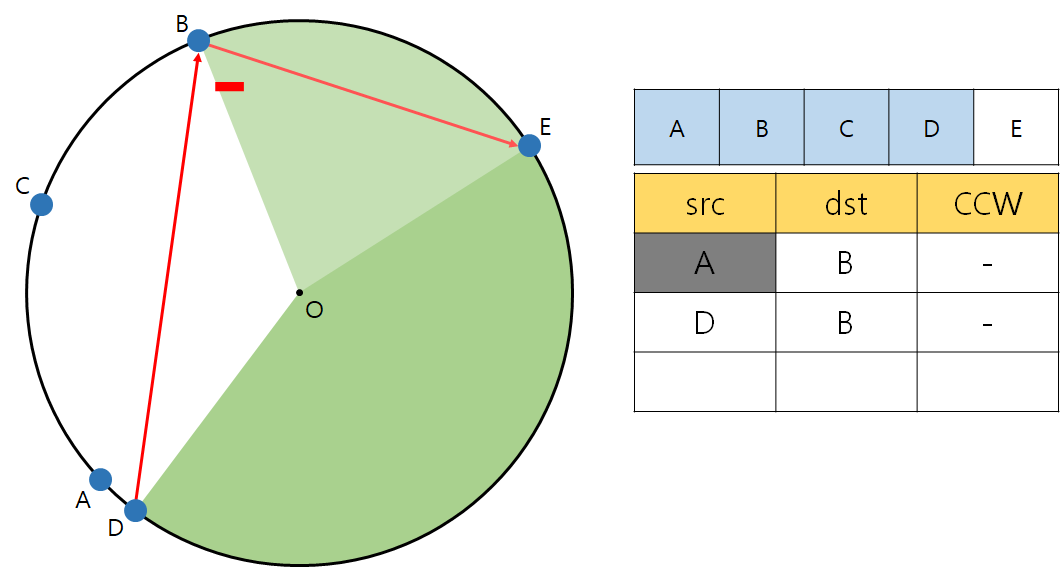
다음 점으로는 C가 입력되었다. 우선 탐색 영역 내부의 점인지 확인하기 위해 CCW(src, dst, C)가 초기 CCW와 부호가 같은지 확인한다. 예시의 경우 다르기 때문에 C는 더 이상 탐색하지 않는다.



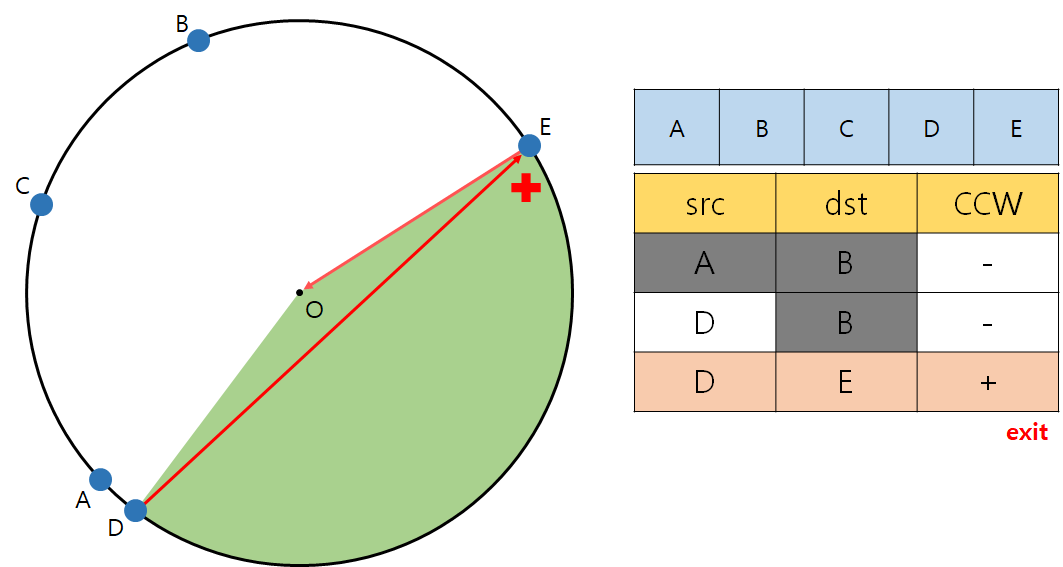
다음 점으로 D가 입력되었다. CCW(src, dst, D)와 초기 CCW의 부호가 같으므로 탐색 영역 내부에 있다. 내부의 점에 대해서는, src와 dst 중 한 가지 값을 바꿔 탐색 영역을 좁혀주어야 한다. 이때 단순히 두 점 사이의 거리 공식을 사용하여, src와 dst 중 새로 추가된 점과 더 가까운 점을 변경해주면 된다. 예시의 경우 src(A)가 더 가깝기 때문에 src가 D로 변경된다.



src와 dst 중 하나가 update 됐다면, CCW(src, dst, O)의 부호가 초기 CCW의 부호와 달라졌는지 확인해야 한다. 이 경우 여전히 음수로, 바뀌지 않았기 때문에 A, B, C, D에 대해서는 half-circle property가 유지된다고 할 수 있다.



마지막으로 E 점이 추가된 경우이다. 우선 CCW(src, dst, E)가 음수이기 때문에 탐색 영역 내부의 점이다. dst(B)가 E와 더 가까우므로 dst가 E로 변경된다.



변경된 dst에 대해, CCW(src, dst, O)를 해보면 양수가 나온다. 초기 CCW는 음수였기 때문에, E가 추가된다면 half-circle property가 깨지게 된다.

맨 처음 입력되는 두 점을 기준으로 이상의 각을 갖는 영역부터 시작하고, 그 지점에서 원점과 CCW를계산하기 때문에 이 알고리즘은 정해를 도출한다. 점이 존재하지 않는 영역의 내각(원점과의 각도)이 를 분기로 부호가 변화하기 때문에, 이상에서 미만으로 변할 때 CCW의 부호가 변하면서 half-circle property가 위배됨을 알 수 있다. 이 알고리즘의 시간 복잡도는 다음과 같다.

- 점을 입력받고, 필요한 과정을 처리 :

**>> 총 시간 복잡도 :**

시간 복잡도 외에도 다음과 같은 장점이 있다.

1. src, dst, CCW 세 가지 값만 저장하며 사용하기 때문에 크기 N의 vector와 stack을 사용하는 볼록 껍질 알고리즘보다 공간 복잡도가 훨씬 작다.

2. 볼록 껍질 알고리즘은 볼록 껍질을 완성하고 나서야 답을 낼 수 있지만, 이 경우 worst case가 아니라면scan 중에 발견하여 답을 낼 가능성이 있다. 또, greedy하게 영역 밖의 점은 아예 탐색하지 않기 때문에 미세하게 더 빠른 연산을 할 수 있다.