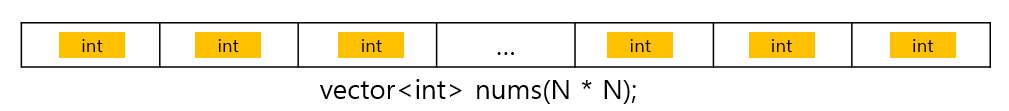
**고급소프트웨어실습I 보고서 2**

**20211584 장준영**

**A) Largest N numbers**

포함된 수가 모두 다른 N\*N 행렬에서 N번째로 큰 값을 찾는 문제이다. 가장 먼저 떠올린 naive한 알고리즘은 행렬의 모든 값을 저장한 일차원 vector를 만들어 정렬한 후 address hashing하는 방식이다.



값을 기준으로 대소 비교를 하여 nums vector에 내림차순 정렬하고, N-1번째 값을 확인하면 그것이 N번째로 큰 값이 될 것이다. 이 방식의 시간 복잡도는 다음과 같다.

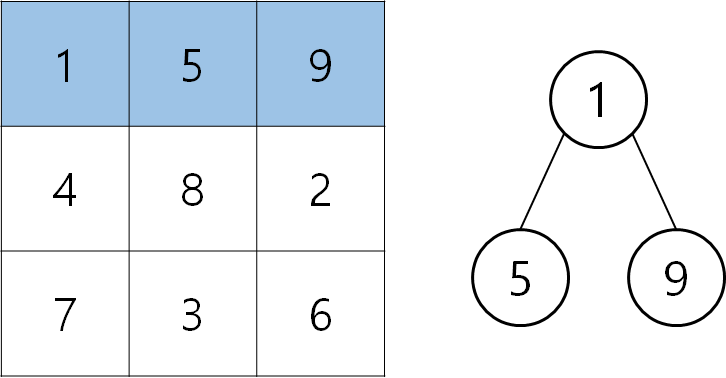
행렬을 순차 탐색하여 배열에 저장 : O(N^2)

배열을 정렬 : O((N^2)log(N^2)) = O((N^2)logN)

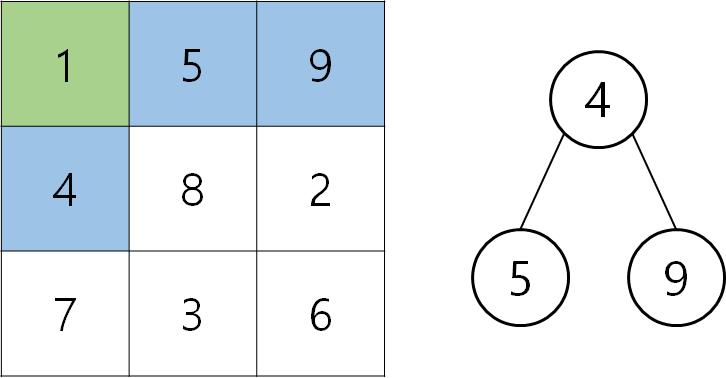
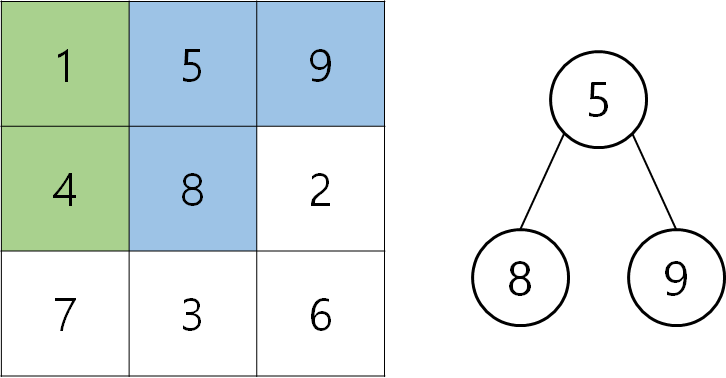
N번째로 큰 값 찾기 : O(1)

**>> 총 시간 복잡도 : O((N^2)logN)**

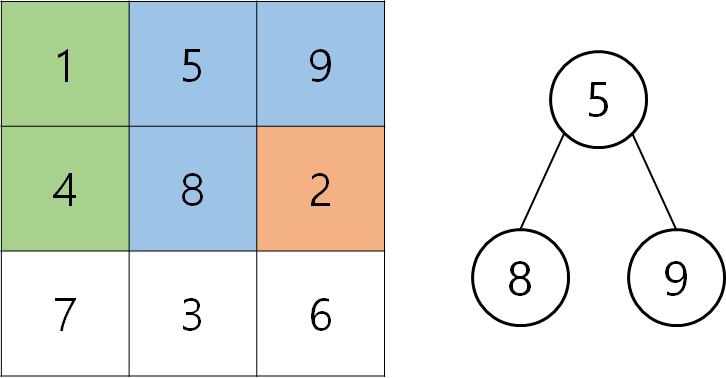
하지만 이 방법은 worst case부터 best case까지의 시간 복잡도가 모두 O((N^2)logN)으로 상당히 크고, N^2 크기의 일차원 vector를 새로 선언해야 하는 등 비효율적인 알고리즘이다. 이 문제를 조금 더 효율적으로 해결하기 위해, ‘N번째’ 숫자를 찾는다는 것에 집중하였다. N\*N보다 N이 훨씬 작은 숫자이므로, 배열을 탐색하면서 큰 값 N개만 저장해도 된다는 아이디어를 떠올릴 수 있었다. 배열에서 가장 큰 N개의 값을 모아두고 탐색을 마치면 그중 가장 작은 값이 N번째로 큰 수일 것이다. 따라서, 값의 대소에 따라 삽입 및 삭제가 용이한 자료구조인 최소 힙을 사용하기로 하였다. 최소 힙의 루트 노드는 저장중인 숫자 중 가장 작은 숫자이므로, 그보다 작다면 삽입하지 않고, 크다면 삽입하면 된다. 또, 공간 복잡도 및 시간 복잡도를 개선하기 위해 숫자를 넣을 때마다 가장 작은 값인 루트 노드의 값을 빼준다. 동작 방식은 다음과 같다.



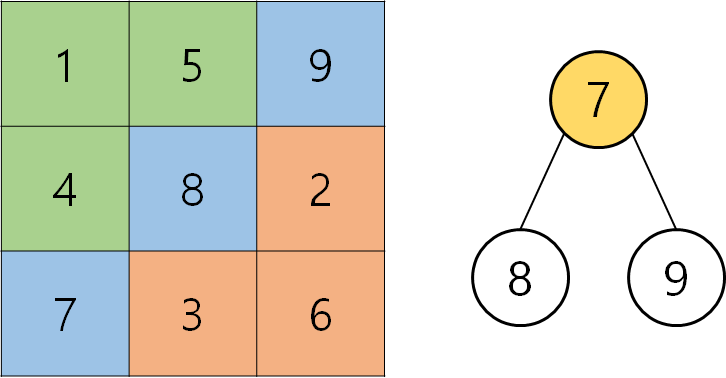
N이 3인 경우의 예시이다. 우선 첫번째 행의 모든 원소를 최소 힙에 넣어준다. 힙의 원소가 N개가 되므로, 이제부터는 힙에 삽입하지 않거나, 삽입 후 삭제하거나 하여 N개의 원소를 유지한다.

다음 행으로 진행한다. 4는 힙의 최소 원소인 1보다 크므로 삽입하고, 최소값인 1을 삭제한다. 이렇게 하면 지금까지 탐색한 원소들 {1, 5, 9, 4} 중 가장 큰 N개의 원소만이 힙에 존재한다. 마찬가지로 다음 원소인 8은 힙의 최소 원소인 4보다 크므로 삽입하고 4를 삭제한다.



다음 원소인 2의 경우는 힙의 최소 원소인 5보다 작다. 따라서, 삽입하지 않는다.



모든 행렬 원소에 대해 탐색한 결과는 위와 같다. 최소 힙의 루트 노드인 7이 N번째로 큰 값이 된다.

이 알고리즘이 항상 정해를 도출하기 위해선, 1~N번째로 큰 수만 힙에 삽입하여 마지막까지 남겨야 하고, 그보다 작은 수를 힙에서 적절히 제거해주어야 한다. 행렬의 모든 수를 탐색하면서, 저장중인 N개의 수 중 가장 작은 숫자를 행렬의 수와 비교하여 삽입 및 삭제하므로 힙에는 항상 탐색한 범위 내에서 가장 큰 N개의 수만 저장된다. 그 과정에서 1~N번째로 큰 수는 삽입 이후 삭제되지 않고, 나머지 숫자는 루트 노드에서 삭제되므로 이 알고리즘은 항상 정해를 도출한다.

그렇다면, 가장 먼저 제시한 naive algorithm과 비교해 얼마나 효율적일까? 우선 이 알고리즘의 시간 복잡도는 다음과 같다.

행렬을 탐색하면서 힙에 삽입 및 삭제 : O(N^2 \* logN) = O((N^2)logN)

탐색을 마치고 루트 노드의 값 확인 : O(1)

**>> 총 시간 복잡도 : O((N^2)logN)**

worst case의 시간 복잡도만 보면 큰 차이가 없는 것을 확인할 수 있다. 하지만, 이 알고리즘은 naive algorithm과 비교해 다음과 같은 장단점이 있다.

1. 정렬을 위해 N\*N 크기의 새로운 1차원 배열을 할당할 필요가 없다. 물론 추가적인 최소 힙을 선언하지만, 이 힙의 크기는 N이기 때문에 공간 복잡도 측면에서 훨씬 좋다. N이 1,000이라면 999,000의 차이이기 때문에 N이 커질수록 유의미하게 효율성의 차이를 낼 수 있다.

2. naive algorithm은 best case, average case, worst case 모두 T((N^2)log(N^2))의 시간 복잡도를 갖는다. 하지만, 이 알고리즘의 worst case의 경우 힙의 크기가 N으로 고정되기 때문에 T((N^2)log(N))의 시간 복잡도를 갖는다. Average case의 경우 루트 노드의 값보다 작아 O(logN)의 삽입 과정을 거치지 않는 노드가 N/2개는 존재하기 때문에 T((N/2)logN+N/2)의 시간 복잡도를 갖는다. Best case의 경우 N개의 노드를 삽입한 이후 아무런 노드도 삽입하지 않기 때문에 T(NlogN+N^2-N) = O(N^2)의 시간 복잡도를 갖는다. 모든 경우에 보다 적은 연산량으로 해결할 수 있다. 물론 big-O notation에 의해서는 같은 값을 갖지만, N이 커지면 커질수록 이 역시 유의미한 차이가 된다.

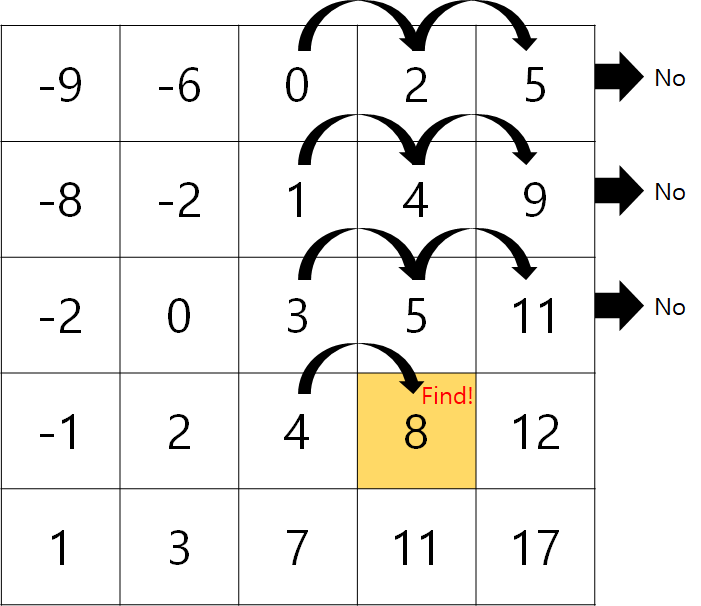
3. 효율성 면에서는 장점 뿐이지만, 최소 힙의 삽입 및 삭제를 구현해야 하기 때문에 구현이 훨씬 어렵다는 단점이 있다. 또, 혹시 문제에서 ‘Q개의 질의에 대해 찾으려는 k largest number가 바뀐다’면, 정렬 알고리즘에선 모든 질의를 O(Q)에 처리할 수 있지만 힙 알고리즘에선 매번 새로운 힙을 선언해 O(Q(N^2)logk)의 과정을 거쳐야 한다. 이 경우에는 naive algorithm의 효율이 훨씬 좋을 수 있다.

따라서, 문제 상황에 맞는 알고리즘을 선택하는 것이 좋다.

**A) Position of k**

행과 열에 대해 모두 정렬된 행렬에 대하여, 어떤 요소 k의 위치를 찾는 알고리즘이다. N이 4인 행과 열에 대하여 정렬된 행렬의 예시는 다음과 같다.

어떤 행이나 열을 보아도 오름차순으로 정렬되어 있다. 이 특징을 효율적인 알고리즘에 이용하기 위하여, 특정한 요소를 찾기 위한 방법으로 가장 먼저 떠올린 naive한 방법은 모든 행에 대해 binary search를 진행하는 것이었다.



이 경우 모든 행 N개에 대해 O(logN)의 탐색을 진행하므로 O(NlogN)의 시간 복잡도를 갖는다. 하지만 여기서 의문점이 들었다.

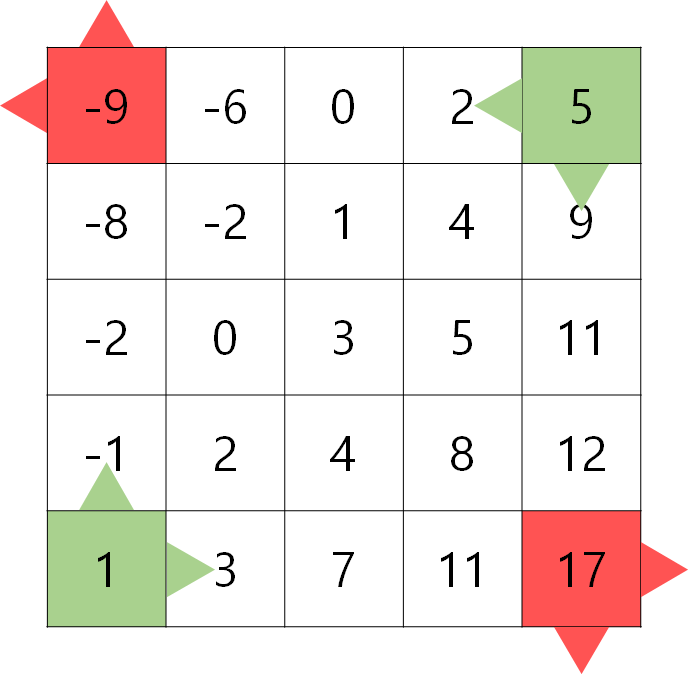
위와 같은 행렬에서 8을 찾고 있고, 2행까지 이분 탐색을 모두 마쳤다고 가정하자. 3행에서 이분 탐색을 시작하면, 8보다 작은 값인 3에서 시작하는 것을 볼 수 있다. 선형 이분 탐색을 하고 있었다면 왼쪽 값인 -2와 0만 탐색 영역에서 제외하겠지만, 행과 열이 모두 정렬되었다는 특징 덕에 3의 왼쪽 위에 있는 모든 값을 탐색에서 제외할 수 있다.

마찬가지로 11은 8보다 크므로, 11의 오른쪽 아래에 있는 모든 값을 탐색에서 제외할 수 있다.



따라서, 위의 그림처럼 일반화할 수 있다. 만약 찾으려는 값이 중심의 3이라는 값보다 크다면 파란색으로 칠해진 영역을 모두 제외할 수 있고, 작다면 노란색으로 칠해진 영역을 모두 제외할 수 있다. 이전에 떠올린 행마다 이분 탐색을 하는 방식은 결국 2차원성을 유지하는 방식이므로 시간 복잡도가 크다. 정렬된 행렬이라는 특징을 이용하여, 행과 열 모두의 탐색 영역을 줄이는 방식을 찾아내야 한다. 위 그림을 떠올리고 나서 분할 정복 및 재귀를 이용하여 탐색 영역을 줄여보려 하였지만, N이 짝수일 때와 홀수일 때의 경우가 복잡하게 나뉘고 탐색해야 할 영역을 제한하는 부분의 구현도 너무 복잡하여 다른 방법을 떠올려보았다. 구현의 용이성을 위해, 한번의 탐색이 하나의 행 또는 열만 제외할 수 있다면 복잡한 구현 없이 탐색의 수를 줄일 수 있을 것이라고 판단하였다. 그러기 위해선, 탐색을 중심이 아닌 모서리부터 시작해야 했다.



행렬에선 시작할 수 있는 모서리가 네 개가 있다. 빨간 색으로 칠해진 모서리는 무조건 행렬의 최솟값/최댓값이므로 제거할 수 있는 행과 열이 없다. 하지만, 초록 색으로 칠해진 모서리는 값이 k보다 클 때와 작을 때 모두 제거할 수 있는 행과 열이 있으므로 탐색은 초록 색으로 칠해진 모서리에서 시작해야 한다.

5를 찾는 탐색을 좌측 하단 1에서 시작한다. 1은 5보다 작으므로, 1보다 작은 수만 모여있는 1열을 탐색 영역에서 제거하기 위해 포인터를 한 칸 오른쪽으로 움직인다. 3 또한 5보다 작기 때문에 마찬가지로 동작한다.

7은 5보다 크다. 7보다 큰 수만 모여있는 우측 행을 탐색 영역에서 제거하기 위해, 포인터를 위로 옮긴다. 4는 5보다 작으므로, 포인터를 오른쪽으로 옮긴다. 8은 5보다 크기 때문에, 포인터를 위로 옮긴다. 이렇게 하면 5를 찾게 된다.

이 풀이가 항상 정해를 도출해내려면, 포인터를 옮기는 것이 탐색 영역을 올바르게 제한해야 하고, 제한된 탐색 영역에 찾고자 하는 숫자가 없어야 한다. 이는 원리에 따라 자명한 이야기인데, 포인터의 이동 방향이 오른쪽과 위쪽밖에 없기 때문이다. 어떤 특정한 지점에 도달하기 위해서는 그 지점 기준 왼쪽과 아래쪽의 모든 지점을 훑으면서 왔다는 의미가 되는데, 그 과정에서 포인터가 항상 가능 영역의 좌측 하단 모서리에 위치하게 된다. 그 과정에서 절대로 후보가 될 수 없는 값들이 ㄴ자 모양으로 제거되기 때문에, 탐색 영역의 일관성이 보장된다. 따라서, 이 풀이는 정렬된 행렬에서 항상 정해를 도출한다. 이 방법의 시간 복잡도는 다음과 같다.

Worst case : 찾는 수가 우측 상단에 있어 모든 행과 열이 제외될 때까지 탐색

**>> 총 시간 복잡도 : O(2N-1) = O(N)**

이분 탐색을 N번 하는 방법의 O(NlogN)에서 O(N)으로 시간 복잡도를 감소시킬 수 있었다. 또, 하나의 포인터만 사용해 값을 단순 비교하는 연산만 하기 때문에 풀이가 직관적이다.