CSE2011 Problem Solving, 2016 Spring

2013312343 이상헌

**Homework 1**

**1. 문제 이해**

(1) 문제

- There is an N\*N matrix. All elements in each row and column are sorted in an ascending order.

① Find a single k efficiently (minimum access of elements). If there are multiple k, you need to find **ANY SINGLE** k.

② Find all k efficiently (minimum access of elements). If there are multiple k, you need to find **ALL** k.

(2) 중요 정보

- 0 이상의 정수를 원소로 갖는 N\*N Matrix.

- 같은 행이나 열에 위치하는 원소들은 오름차순으로 저장되어 있다.

(3) 문제 정의

- 주어진 Matrix에서 원소 k를 가장 효율적으로(access를 최소화하여) 찾는 방법. (k가 여러 개인 경우, 아무 k 한 개만 찾는다.)

- 주어진 Matrix에서 원소 k를 가장 효율적으로(access를 최소화하여) 찾는 방법. (k가 여러 개인 경우, Matrix 내의 모든 k를 찾는다.)

**2. 문제 해결**

(1) 해결 방법 결정

*① 재귀적 방법*

- 처음 생각했던 방법은 수업에서 배운, 재귀적으로 접근하는 방법이다. 재귀(Recursive)란, 어떠한 함수를 그 자체의 함수를 이용하여 정의하는 방법으로, 함수 내에 다시 함수를 call하는 것을 의미한다. N\*N Matrix는 정사각형으로 되어 있고, 이를 여러 개의 작은 정사각형으로 나눈 후, 각각의 정사각형에서 주어진 원소 k를 찾을 수 있다고 생각하였다.

- 아이디어

작은 정사각형으로 나눌 때, 기준 점이 있어야 한다. 즉, 기준점을 중심으로 좌표의 사분 면과 같이 네 구역으로 나눌 수 있다. 기준 점은 N\*N Matrix가 정사각형이기 때문에, 대각선의 중심으로 잡을 수 있다. 이 후에 한 구역에서 k를 검색하는 함수를 구현한다면, 처음에는 나누어진 네 구역에서, 그리고 각각의 구역에서 또 다시 네 구역으로 나누어 함수를 이용한다.

Figure 재귀적 방법. find() 함수를 재귀한다.

find()

find()

find()

find()

find()

- 이점

위와 같이 재귀적(Recursive) 방법을 이용한다면, 코드를 간편화 할 수 있다. 또한 기준이 되는 함수만 정확히 구현된다면, 예외적인 상황에서도 정확한 결과 값을 나타낼 것이다. 그리고 재귀적 방법은 문제를 가장 효율적으로 푸는 방법들 중 하나이다. (실제로 문제의 O(n)을 구할 때도, Divide-And-Conquer 와 Recursive Call을 이용하여 계산한다.)

- 한계

재귀적 방법으로 본 문제를 해결하는 과정에서, 여러 한계를 느꼈다. 첫 번째로, N\*N Matrix 자체는 정사각형이 맞지만, 이를 네 영역으로 나눌 때, 일정한 틀을 유지하지 못한다는 점이다. 특히 N이 홀수인 경우, 네 정사각형으로 나누기 위해선 한 개의 행 또는 열이 남고, N이 짝수인 경우, 정확한 중심이 되는 기준점을 잡기 어렵다. 물론 구역을 나누면서 N의 홀,짝에 따라 하나의 변을 버리거나, 중심과 가까운 점을 중심으로 잡는 등의 방법으로 정사각형 형태를 만들 수 있다. 하지만 여러 가지 경우를 생각해야 하므로, 일반적인 재귀적 방법과는 어긋난다.

가장 크게 느낀 한계는 두 번째이다. 코드를 구현할 때, call\_single()과 call\_all() 함수 이외의 다른 함수를 만들지 못하는 규칙이 있다. 하나의 함수 내에서 다른 함수를 정의하는 것은 C 프로그래밍 상에서 불가능하기 때문에, 재귀 함수를 이용하기에는 큰 어려움이 있다고 판단하였다.

*② 랜덤 함수(rand()) 이용*

- N\*N Matrix에 저장되는 원소들은 각 같은 행과 열에서 오름차순을 이룬다는 성질은 있지만, 결국 랜덤 하게 저장된다. 즉, 행렬 내의 원소들의 범위, 수의 종류 등은 전혀 알 수 없다. 때문에, 일반적인 N과 모든 수에 대해 문제를 해결할 수 있는, 일반적인 알고리즘을 기반으로 코드를 구현해야 한다. 기존 행렬 내의 수 배치가 랜덤 하다는 것에서 착안하여, 원소를 찾는 방법을 랜덤 함수를 이용하여 구현할 수 있다는 생각을 하였다.

- 아이디어

랜덤 함수(rand())는 현재 시각에 따라, 제한되지 않은 범위 내의 random한 수를 제공한다. 범위를 설정하기 위해서는 % (나머지) 기호를 이용한다. 랜덤 함수를 처음부터 이용하는 것은 항상 효율적인 값을 나타낼 수 없다. 그러므로, 원소 값을 access해야 하는 영역의 범위를 최대한 줄인다. N\*N Matrix에서 대각선 위의 원소들((0,0)부터)을 비교하여, k와 가장 근접하고, k보다 큰 원소를 찾는다. 이 원소를 중심으로 왼쪽 위의 영역 내에는 k가 없다는 것을 알 수 있다. 이는 모든 행과 열이 오름차순으로 정리되어 있기 때문이다.

나머지 세 구역 내의 원소들에 대해서 rand() % LENGTH 로 임의의 원소를 뽑아 k와 비교한다. 만일 k라면, 좌표를 따로 저장한다. k와 같지 않다면 저장하지 않고, 그 원소의 왼쪽에 위치하는 원소들을 검색 대상에서 제외시킨다. 한 번 검사된 원소는 따로 표시하여, 중복 검사를 막는다. 이와 같은 방법으로 나머지 세 구역의 원소들을 검사한다.

X

O

O

O

Figure 랜덤 함수 이용 방법. 대각선을 따라 기준점을 찾고, 왼쪽 위의 구역을 검색에서 제외시킴으로써 access 수를 줄인다.

- 이점

이 방법은 코드를 가장 간단하게 구현할 수 있다는 장점이 있다. 다양한 경우를 생각하지 않아도 일반적이고 정확한 결과 값을 나타낼 것이다. 또한 제외한 구역이 존재하기 때문에, 모든 수를 검사하는 방법(두 번의 for문을 이용)에 비하여 적은 access 수로 문제를 해결할 수 있다.

- 한계

결국 이 방법은 모든 수를 검사하는 경우이기 때문에, 효율적인 부분에서 큰 문제가 발생한다. 본 문제의 핵심은 “얼마나 효율적으로, 정확한 답을 찾아내는가” 이다. 이 아이디어는 정확한 답을 찾아낼 수는 있지만, 가장 효율적인 알고리즘이라고 말할 수는 없다.

*③ 구역 할당 및 기준점 검색 방법.*

- 여러 방법을 생각하면서, N\*N Matrix의 대각선을 이용하여 기준점을 잡고, 네 구역으로 나누어 하나의 구역(왼쪽 위 구역)을 검색에서 제외시키는 방법이 일단 효율적이라고 생각하였다. 이를 바탕으로, 보다 일반적이고 효율적으로 나머지 세 구역을 검색하는 알고리즘을 생각하였다. 특히, 문제에서 가장 중요한 요소인, 같은 행과 열에서 원소들이 오름차순으로 정리되어 있다는 점을 고려하였다.

- 아이디어

(0,0)에서 시작하여 대각선을 따라 내려오면서, 처음으로 k보다 크거나 같은 수가 나타나는 좌표를 기준점으로 잡았다. 경우를 크게 두 가지로 나눈다.

만일, 기준점이 k와 같다면, 오른쪽 아래에도 k와 같은 수의 원소가 있을 가능성이 있다. 이는 같은 행과 열에서 중복되는 값을 갖는 원소가 존재할 수 있기 때문이다. 오른쪽 아래의 원소들은 다음과 같은 방법으로 조사한다.

A. 기준점에서 오른쪽 방향으로, k보다 큰 수가 나타나는 원소까지 같은 행의 원소들을 k와 비교한다. 만일 원소의 값이 k보다 크다면, 그 원소의 오른쪽에 위치하는 모든 원소들은 k보다 크기 때문에 비교할 필요가 없다. 비교 중에 k와 같은 원소가 있다면, 좌표를 저장한다.

B. 다음 (아래의) 행에서 같은 방법을 반복하여 비교한다. 만일 밑의 행의 첫 번째 원소가 k보다 크다면, 그 아래 행과 오른쪽 열에 존재하는 모든 원소들은 k보다 크기 때문에 비교할 필요가 없다. 비교 중에 k와 같은 원소가 있다면, 따로 좌표를 저장한다.

만일, 기준점이 k와 같지 않다면 기준점 오른쪽 아래의 모든 원소들은 k보다 크다. 따라서 조사할 필요가 없다.

이후, 기준점의 오른쪽 위 구역의 원소들을 조사한다. 이는 기준점이 k와 같던 다르던, 반드시 행해야 한다. 오른쪽 위 구역의 원소들은 다음과 같은 방법으로 조사한다.

A. 기준점의 열에서 위쪽 방향으로 k보다 작거나 같은 원소가 나타날 때까지 조사한다. 만일 원소의 값이 k보다 크다면, 그 원소의 오른쪽 행에 위치하는 모든 원소들은 k보다 크기 때문에 비교할 필요가 없다.

B. k보다 작거나 같은 원소가 나타난다면, 그 원소의 오른쪽 방향으로, k보다 큰 원소가 나타날 때까지 같은 행의 원소들을 비교한다. 만일 원소의 값이 k보다 크다면, 그 오른쪽에 위치하는 모든 원소들은 k보다 크기 때문에 비교할 필요가 없다. 비교 중에 k와 같은 원소가 있다면, 좌표를 따로 저장한다.

마지막으로, 기준점의 왼쪽 아래 구역의 원소들을 다음과 같은 방법으로 조사한다.

A. 기준점의 행에서 왼쪽으로 가면서 k보다 작은 원소가 나타날 때까지 조사한다. 만일 원소의 값이 k보다 크다면, 그 원소의 아래 열에 위치하는 모든 원소들은 조사할 필요가 없다.

B. k보다 작거나 같은 원소가 나타난다면, 그 원소의 아래 방향으로, k보다 큰 원소가 나타날 때까지 같은 열의 원소들을 비교한다. 만일 원소의 값이 k보다 크다면, 그 아래에 위치하는 모든 원소들은 k보다 크기 때문에 비교할 필요가 없다. 비교 중에 k와 같은 원소가 있다면, 좌표를 따로 저장한다.

- 이점

아이디어의 가장 큰 특징은 생각한 다른 아이디어들보다 더욱 효율적이라는 것이다. 처음 큰 정사각형을 네 구역으로 나눈 후, 한 구역을 조사할 필요가 없도록 하였다. 또한, 나머지 세 구역에서의 조사도 모든 원소를 조사하지는 않는다. 같은 행 또는 열의 원소들은 오름차순이라는 성질을 가지고 있다. k보다 큰 값이 나타난 경우에서 같은 행 또는 열에 위치한 원소들을 더 이상 비교할 필요가 없다는 점을 착안하여, 비교 횟수를 줄였다.

뿐만 아니라 모든 구역에서의 꼼꼼한 비교가 이루어지기 때문에, 예외적인 상황 혹은 극단적인 상황에 대해서도 정확한 결과 값을 도출할 수 있다.

- 한계

본 아이디어는 네 구역으로 나눈 후, 한 구역을 조사하지 않고 나머지 세 구역을 조사한다. 세 구역을 조사하는 과정에서, 오름차순의 성질을 최대한 활용하여 원소들을 검사한다. 하지만 조사하는 방법은 역시 각각의 원소의 크기 비교 및 k와의 등호로, 하나하나 조사한다. 조사하는 방법에서 다른 아이디어를 복합하여 새로운 알고리즘을 짠다면 (예를 들어, 조사하는 경우에서 재귀적 방법을 이용하여 여러 직사각형의 꼴에서도 조사할 수 있도록 만들 수 있다.) 더욱 효율적으로 결과 값을 도출하는 프로그램을 작성할 수도 있다. 일단 직접 생각한 여러 가지 아이디어들 중에서 가장 효율적이라고 생각하여, 본 아이디어를 선택하였지만, 더욱 효율적, 일반적으로 결과를 도출하는 다른 알고리즘이 분명히 있을 것이라 생각한다.

**3. 코드 작성**

- 본 아이디어 및 설명을 바탕으로 코드를 작성하였다.

① call\_single()

- call\_single() 함수는 matrix 내에서 하나의 k값만 찾으면 되는 함수이다. 이 함수는 먼저 call\_all() 함수를 구현한 뒤에, 하나의 k값을 찾은 경우에, 원소의 좌표를 출력하고 함수를 바로 return 하도록 설정하였다.

void cal\_single(int k) {

int Kx, Ky, numconst, mid = LENGTH, i=0, j=0, c1=0;

Kx, Ky는 k값을 갖는 원소의 x, y좌표이다. numconst는 원소를 비교할 때, 같은 원소를 두번 access해야하는 경우를 막기 위해 임의로 설정한 상수 변수이다. mid는 기준점의 x, y 좌표이다. 기준점은 대각선 상에 존재하기 때문에 x와 y의 좌표 값이 같다. I, j, c1 은 for문, while문에서 사용되는 상수 변수이다.

while(c1<LENGTH){

numconst = access\_value(c1,c1);

if (k <= numconst){

mid = c1;

break;

}

c1++;

}

위의 코드는 (0,0)을 기준으로 대각선 방향으로 원소들을 검사하면서, 최초로 k보다 크거나 같은 원소를 찾는 코드로, 기준점의 좌표를 설정한다.

if(mid == LENGTH){

return;

}

만일 기준점의 좌표가 처음 설정한 대로라면 (즉, k보다 크거나 같은 원소가 대각선 상에 존재하지 않는다면) 함수를 그대로 return 한다.

if(k == numconst){

Kx = c1;

Ky = c1;

printf("(%d,%d)\n",Kx,Ky);

return;

}

기준점이 k와 같다면, x와 y 좌표를 출력하고 함수를 return 한다.

i=0; j=1;

while( mid-j >= 0){

if(access\_value(mid-j,mid) <= k){

i=0;

while( mid+i < LENGTH){

numconst = access\_value(mid-j,mid+i);

if(k == numconst){

Kx = mid-j;

Ky = mid+i;

printf("(%d,%d)\n",Kx,Ky);

return;

}

else if(k < numconst){

break;

}

i++;

}

}

j++;

}

기준점을 중심으로 오른쪽 위의 원소들을 조사하는 코드이다. 하나의 k값만 찾으면 되기 때문에, 만일 k값과 같은 원소를 찾았다면, 이의 좌표를 출력하고 함수를 return한다.

i=1; j=0;

while( mid - i >= 0){

if(access\_value(mid,mid-i) <= k){

j=0;

while( mid +j < LENGTH){

numconst = access\_value(mid+j,mid-i);

if(k == numconst){

Kx = mid+j;

Ky = mid-i;

printf("(%d,%d)\n",Kx,Ky);

return;

}

else if(k<numconst){

break;

}

j++;

}

}

i++;

}

}

기준점의 왼쪽 아래 구역의 원소들을 조사하는 코드이다. 단 하나의 k값을 찾으면 되기 때문에 만일 k와 같은 원소를 찾았다면, 이의 좌표를 출력하고 함수를 return한다.

② call\_all()

- matrix 내의 모든 k 값을 갖는 원소의 좌표를 출력하는 코드이다. call\_single() 함수와 다른 점은, k값을 갖는 원소의 개수를 count하는 int형 변수를 설정하였고, 기준점의 값이 k인 경우에 기준점의 오른쪽 아래 구역도 조사해야하기 때문에, 이에 대한 코드가 추가된다. 또한 여러 개의 k 값을 가질 수 있기 때문에, k 값을 갖는 원소의 좌표를 저장하는 변수를 동적 할당을 이용한 1차원 배열 두개로 정의하였다.

void cal\_all(int k) {

int \*Kx, \*Ky, Knum=0, numconst, mid = LENGTH, i=0, j=0, c1=0;

Kx, Ky는 1차원 배열로, k 값을 갖는 원소의 좌표를 저장한다. Knum은 배열 내의 k값을 갖는 원소의 개수를 저장한다. 나머지 변수들은 call\_single()의 변수와 의미가 같다.

Kx=(int\*)calloc(LENGTH\*LENGTH,sizeof(int));

Ky=(int\*)calloc(LENGTH\*LENGTH,sizeof(int));

for(i=0;i<LENGTH\*LENGTH;i++){

Kx[i] = 0;

Ky[i] = 0;

}

Kx, Ky는 동적 할당을 이용하여 각각을 1차원 배열로 정의하고, 모든 값을 0으로 초기화하였다. k의 최대 개수는 LENGTH\*LENGTH 개이므로 이를 배열의 크기로 정하였다.

while(c1<LENGTH){

numconst = access\_value(c1,c1);

if (k <= numconst){

mid = c1;

break;

}

c1++;

}

대각선을 따라 기준점을 잡고, 기준점의 좌표를 mid 변수에 저장한다.

if(mid == LENGTH){

return;

}

기준점이 없다면 함수를 바로 return한다.

i = 1; j = 0;

if(k == numconst){

Kx[0] = c1;

Ky[0] = c1;

Knum++;

while( mid+i < LENGTH){

numconst = access\_value(mid+j,mid+i);

if(k == numconst){

Kx[Knum] = mid+j;

Ky[Knum] = mid+i;

Knum++;

}

else if(k < numconst){

break;

}

i++;

}

j++;

while(mid+j <LENGTH){

if(access\_value(mid+j,mid) > k)

break;

i=0;

while( mid+i < LENGTH){

numconst = access\_value(mid+j,mid+i);

if(k == numconst){

Kx[Knum] = mid+j;

Ky[Knum] = mid+i;

Knum++;

}

else if(k < numconst){

break;

}

i++;

}

j++;

}

}

기준점의 원소의 값이 k라면, 기준점의 오른쪽 아래 영역의 원소들을 k와 비교한다. 이들 중, k와 같은 값을 갖는 원소의 좌표를 Kx, Ky에 저장한다.

i=0; j=1;

while( mid-j >= 0){

if(access\_value(mid-j,mid) <= k){

i=0;

while( mid+i < LENGTH){

numconst = access\_value(mid-j,mid+i);

if(k == numconst){

Kx[Knum] = mid-j;

Ky[Knum] = mid+i;

Knum++;

}

else if(k < numconst){

break;

}

i++;

}

}

j++;

}

기준점의 오른쪽 위 영역의 원소들을 k와 비교, 저장한다.

i=1; j=0;

while( mid - i >= 0){

if(access\_value(mid,mid-i) <= k){

j=0;

while( mid +j < LENGTH){

numconst = access\_value(mid+j,mid-i);

if(k == numconst){

Kx[Knum] = mid+j;

Ky[Knum] = mid-i;

Knum++;

}

else if(k<numconst){

break;

}

j++;

}

}

i++;

}

기준점의 왼쪽 아래 영역의 원소들을 k와 비교, 저장한다.

if(Knum == 0)

return;

만일 k값을 갖는 원소가 하나도 없다면, 함수를 바로 return한다.

for(i=0;i<Knum-1;i++)

printf("(%d,%d), ",Kx[i],Ky[i]);

printf("(%d,%d)",Kx[Knum-1],Ky[Knum-1]);

k값을 갖는 모든 원소들의 좌표를 형식에 맞게 출력한다.

return;

}

**4. 확인 및 분석**

- 본 아이디어를 이용하여 프로그램을 작성하였다. 이 프로그램이 정확한 결과값을 도출하는지를 알아보기 위해서 간단한 10\*10 matrix를 임의로 만들었다. 아래는 프로그램 작동 확인을 위해 임의적으로 만든 10\*10 matrix이다.

1 2 3 4 5 6 7 22 23 24

2 3 4 5 6 7 8 22 23 24

3 6 8 9 10 11 12 23 23 24

4 6 9 9 10 12 23 24 24 25

6 6 9 10 10 12 24 25 25 25

9 9 10 10 12 12 24 32 32 38

9 10 10 10 23 23 25 32 33 38

12 12 12 23 24 24 25 33 38 39

23 23 24 24 25 25 32 38 38 39

24 24 24 24 32 32 32 39 41 43

- 이 matrix를 이용하여 코드를 실행시켜 보았다.

 - Figure 3은 k에 24를 대입하고 프로그램을 실행한 결과이다. call\_all() 함수의 결과를 보면 예시의 matrix 내 24를 갖는 모든 원소들의 좌표를 정확하게 출력했다는 것을 알 수 있다. count는 61로, 최대 100의 count를 할 수 있는 환경에서 많이 줄였다고 생각할 수 있다. 또한 call\_single()의 함수의 count수는 9로, 매우 효율적으로 결과를 도출한 것이라고 볼 수 있다.

Figure k=24에 대한 결과

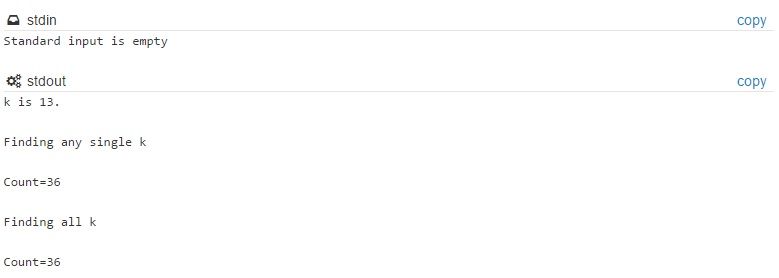


Figure k=13에 대한 결과

- Figure 4는 k에 13을 대입하고 프로그램을 실행한 결과이다. 예시의 matrix 내에는 13을 갖는 원소는 존재하지 않는다. 프로그램 실행 결과, 존재하지 않는 원소까지 정확하게 포착하여 아무것도 출력하지 않는 결과를 나타내었다. count 수도 어느 정도 존재하여, 프로그램 실행 중의 오류로 인해 아무것도 출력하지 않은 것은 아님을 알려준다.

**6. 고찰**

- 문제는 오름차순으로 정리된 임의의 배열 내에서 특정 k값의 좌표를 출력하는 것이다. 사실 문제만 놓고 보았을 때, 성질과 결과 등 매우 간단하다. 처음 문제를 접할 때에도 매우 간단하다고 생각하였고, 보자마자 여러 가지 알고리즘과 해결 방안이 떠올랐다. 하지만 이를 구체화, 일반화하는 과정에서 많은 한계와 시행착오를 겪었다. 또한 코드로 옮기는 과정에서도 실력의 미숙함을 느꼈다. 하지만 문제를 푸는 과정에서 아이디어 맵핑 및 새로운 방향의 시도를 함으로써, 주어진 문제를 어떠한 방향으로 풀어야 하는가의 전체적인 구조를 설계하는 방법은 보다 숙달되고 익숙해진 것 같다. 앞으로 주어진 문제에 대한 푸는 방법을 수업시간과 과제를 통해 배울 수 있게 되어 흥미롭고, 실력 향상에 큰 도움이 될 것 같다.

**7. 참고문헌 및 사용**

- [www.ide.com](http://www.ide.com) : C언어 구현 및 실행.

- 이진규 교수님, [문제해결기법] CSE2011\_2016spring\_Lecture\_Note01, 2016