6주차 결과보고서

전공: 수학/컴퓨터공학 학년: 3학년 학번: 20181288 이름: 윤성호

<CheckToMove 함수>

정해진 실습시간 이전에 집에서 미리 실습을 진행하여, 예비보고서의 pseudo code와 실제 코드가 크게 달라진 점은 없다. 4\*4 block prototype에서 블록이 있는 지점에 대해서 필드바깥으로 나가거나, 이미 쌓여있는 다른 블록과 중복이 되면 이동이 불가능하므로 0을 리턴하고, 이외의 경우에는 이동이 가능하므로 1을 리턴하는 함수이다.

<<시간복잡도>>

4\*4 이중 loop를 돌며 currentBlock의 모양을 탐색하여, 먼저 블록이 있는 지점인지를 체크해준다. 블록이 있다면 필드 바깥으로 나가는지를 판별하는 조건문과 필드에 쌓인 블록과 겹치는지를 판별하는 조건문을 거치게 되고, 이러한 조건문을 이중 loop로 탐색중인 모든 블록에 대해 통과한다면 이동가능한 위치라고 판단하고 1을 리턴한다. 1을 리턴하는 이 상황이 곧 함수가 최대의 시간이 걸리게 되는 순간이고, 이때는 블럭이 저장되어있는 이중 배열의 크기인 BLOCK\_WIDTH와 BLOCK\_HEIGHT에 시간복잡도가 영향을 받게된다. 따라서, Big o notation으로 시간복잡도를 표현하면 O(BLOCK\_WIDTH \* BLOCK\_HEIGHT) 이다.

<<공간복잡도>>

I, j 등의 반복문을 위한 변수들과, 배열의 참조값정도만 사용되므로 공간복잡도는 O(1)이다.

아래는 실제 실습시간에 제출한 CheckToMove 함수의 코드이다.

int CheckToMove(char f[HEIGHT][WIDTH],int currentBlock,int blockRotate, int blockY, int blockX){

// user code

/////////////////////////////////////////////////////////////////

// 현재 블록을 움직일 수 없는 경우 //

// 1. 예상된 현재 블록이 필드 바깥으로 나갈 경우 //

// 2. 예상 현재 블록이 쌓여 있는 블록과 겹치는 경우//

/////////////////////////////////////////////////////////////////

// 각각의 블록에 대하여

for (int i = 0 ; i < 4; i++){

for (int j = 0; j < 4 ; j++){

// 4X4 block prototype에서 블록이 있는 지점에 대해서

if ( block[currentBlock][blockRotate][i][j] == 1 ){

// 예상된 현재 블록이 필드 바깥으로 나갈 경우 움직일 수 없으므로 0을 리턴한다.

if ((0 > i + blockY) || (HEIGHT <= i + blockY) || (0 > j + blockX) || (WIDTH <= j + blockX))

return 0;

// 예상 현재 블록이 쌓여 있는 블록과 겹치는 경우 움직일 수 없으므로 0을 리턴한다.

if (f[i+blockY][j+blockX] == 1)

return 0;

}

}

}

// 예상된 현재 블록이 모두 필드 안에 있고 쌓여 있는 블록과 겹치지 않은 경우 움직일 수 있으므로 1을 리턴한다.

return 1;

}

<DrawChange 함수>

실습시간 이전에 제출한 pseudo code에서 다른부분은 모두 같지만 이전 블록의 정보 중 회전 횟수 정보를 받아오는 데 약간의 오류가 있었다. 이전 회전 횟수를 불러오려면 (blockRotate + 3)%4의 형태로 구해야 한다. 여기서 +3을 해주는 이유는 270도를 돌려야 하기 때문이다. 그러나 실습을 진행하기 전에는 (blockRotate - 1)%4의 형태로 코드를 작성하여서 오류가 났다. 이러한 오류가 나타난 이유는 C언어에서 modulo 연산자가 작동하는 원리에 대한 이해부족 때문이었다. Modulo연산은 피연산자의 부호를 따라가게 되어있다. 즉 a%b의 결과의 부호는 a의 부호를 따라간다는 의미이다. 실수한 코드에서는 blockRotate가 1, 2, 3일때는 아무런 문제가 없지만 0이 되는 순간 오류가 발생한다. 이 때문에 실제 게임화면에서 위쪽 방향키를 3번째까지 누를때는 오류가 없지만 4번째로 누르게 되면 이전 블록이 제대로 지워지지 않는 것을 확인할 수 있었다. DrawChange함수는 블록의 변경정보를 인수로 받아, 변경정보에 알맞은 블록의 모양을 화면에 출력해주는 함수이다.

<<시간복잡도>>

알고리즘의 과정을 세단계로 나뉜다. 이전 블록의 정보를 찾는 부분, 이전 블록의 정보를 지우는 부분, 그리고 새로운 블록정보를 그리는 부분이다. 먼저 이전 블록의 정보를 찾는 부분은 반복문이 없고, 이전 정보를 담는 변수 3개(이전 x좌표, 이전 y좌표, 이전 회전횟수)에 대입해주는 연산정도가 필요하다. 이 부분은 뒤에 두 과정에서 때문에 시간복잡도 측면에서 크게 의미가 있는 부분은 아니다. 두번째로 이전 블록의 정보를 지우는 부분이다. 이때는 4\*4 block prototype을 이중 loop으로 확인하여 블록이 있고, 그리고 y값이 0이상이라면 “.”로 바꿔주어야 한다. 이때 이중 loop는 BLOCK\_WIDTH와 BLOCK\_HEIGHT 두 변수에 의존한다. 따라서 이부분의 시간복잡도를 계산하면 BLOCK\_WIDTH \* BLOCK\_HEIGHT에 비례하는 값을 얻게된다. 그리고 마지막으로 새로운 블록정보를 그리는 부분에서는 DrawBlock 함수를 사용하였는데, 이 함수에서도 이중 loop으로 블록이 존재하는 부분에 대해 그려주는 것이기 때문에 BLOCK\_WIDTH와 BLOCK\_HEIGHT 두 변수에 의존한다. 결과적으로 DrawChange에서 진행하는 세가지 부분을 모두 고려하게 되면 big o notation으로 시간복잡도 O(BLOCK\_WIDTH \* BLOCK\_HEIGHT)가 된다.

<<공간복잡도>>

이전 블록의 회전횟수, x좌표, y좌표등의 정보를 저장하기 위한 변수들과, 반복문을 위한 변수인 i, j, 그리고 배열의 참조값이 사용되므로 공간복잡도는 O(1)이다.

아래는 실제 실습시간에 제출한 DrawChange 함수의 코드이다.

void DrawChange(char f[HEIGHT][WIDTH],int command,int currentBlock,int blockRotate, int blockY, int blockX){

// user code

//1. 이전 블록 정보를 찾는다. ProcessCommand의 switch문을 참조할 것

//2. 이전 블록 정보를 지운다. DrawBlock함수 참조할 것.

//3. 새로운 블록 정보를 그린다.

////////////////////////////// 1. 이전 블록 정보를 찾는다. ////////////////////////////

// 이전 블록의 위치, 회전횟수를 얻기 위해 이전 command를 살펴본다.

// preBlockRotate : 이전 블록의 회전횟수

// preBlockX : 이전 블록의 x좌표

// preBlockY : 이전 블록의 y좌표

int preBlockRotate; // 이전 블록의 회전 횟수

int preBlockX; // 이전 블록의 x좌표

int preBlockY; // 이전 블록의 y좌표

switch(command){

// 윗 방향키이면 블록이 반시계방향으로 90도 회전했다는 것을 의미하고,

// 이전 방향은 시계방향으로 90도 회전 시키면 된다.

case KEY\_UP:

preBlockRotate = (blockRotate+3) % 4;

preBlockX = blockX;

preBlockY = blockY;

break;

// 아래 방향키이면 블록이 아래로 한칸 내려왔다는 것을 의미하고,

// 이전 위치는 y좌표가 1 줄어든 위치이다. x좌표는 변함이 없다.

case KEY\_DOWN:

preBlockRotate = blockRotate;

preBlockX = blockX;

preBlockY = blockY - 1;

break;

// 오른쪽 방향키이면 블록이 오른쪽으로 한칸 이동했다는 것을 의미하고,

// 이전 위치는 x좌표가 1 줄어든 위치이다. y좌표는 변함이 없다.

case KEY\_RIGHT:

preBlockRotate = blockRotate;

preBlockX = blockX - 1;

preBlockY = blockY;

break;

// 왼쪽 방향키이면 블록이 왼쪽으로 한칸 이동했다는 것을 의미하고,

// 이전 위치는 x좌표가 1 늘어난 위치이다. x좌표는 변함이 없다.

case KEY\_LEFT:

preBlockRotate = blockRotate;

preBlockX = blockX + 1;

preBlockY = blockY;

break;

// 다른것이 입력되었다면 따로 변화는 없다.

default:

break;

}

//////////////////////////// 2. 이전 블록 정보를 지운다. ///////////////////////////

// DrawBlock 함수를 참조했는데, 바꾼 점은 tile parameter가 "."로 고정이 되어있다는 점이다. 공백을 출력해야 하기 때문이다.

int i, j;

for (i = 0 ; i < 4 ; i++){

for (j = 0; j < 4 ; j++){

if (block[currentBlock][preBlockRotate][i][j] == 1 && i + preBlockY >= 0){ // 이전 블록의 모양은 공백으로 바꿔준다.

move(i+preBlockY+1, j+preBlockX+1); // 커서를 이전의 블럭이 있던 위치로 옮겨준다.

printw("."); // 공백의 의미로 ."."을 그려준다.

}

}

}

//////////////////////////// 3. 새로운 블록 정보를 그린다. /////////////////////////

DrawBlock (blockY, blockX, currentBlock, blockRotate, ‘ ‘);

}

<BlockDown 함수>

1초마다 한번씩 실행되며, 아래방향으로 내려갈 수 있는 지 없는 지에 따라서 코드의 실행이 달라지는 함수이다. 실습 이전의 pseudo code와 실제 실습 코드와의 차이점은 아래방향으로 내려갈 수 있는 상황에서 발생하였다. 실습 이전에는 timed\_out = 0이라는 과정을 생략하고 코드를 진행하였다. 써주지 않아도 작동할 거라고 생각했기 때문이다. 그러나 이 문구를 써주지 않는다면 play함수에서 BlockDown함수와 ProcessCommand함수가 독립적으로 작동하지 않게 되고, 입력이 주어지면 BlockDown함수가 작동하지 않고, 입력이 주어지지 않을때만 BlockDown함수가 동작하는 오류가 생기게 되었었다. Timed\_out를 다시 0으로 초기화해주어 매 입력이 BlockDown함수에 영향을 끼치지 않는 방향으로 코드를 수정하여 실습시간에 제출하였다.

<<시간복잡도>>

먼저 블록이 한 칸 내려갈 수 있는 경우를 생각해보자. CheckToMove 함수를 통해 아래로 내려갈 수 있는지를 판단하는데 아래방향으로 내려갈 수 있게 된다는 말은 CheckToMove함수의 리턴값이 1이라는 말과 같고, 이 상황에서 시간복잡도는 BLOCK\_WIDTH와 BLOCK\_HEIGHT에 영향을 받게 됨을 알 수 있다. 두번째 상황으로 블록이 더 이상 내려갈 수 없는 경우를 생각해보자. 그 경우 gameover 변수를 변경하는 것은 크게 영향이 미치지 않지만, AddBlockToField 함수는 이중 loop를 돌기때문에 시간복잡도에 영향을 미치게 된다. 이때 이중 loop는 BLOCK\_WITDH와 BLOCK\_HEIGHT로 구성되어 있으므로 이 두 변수가 시간복잡도를 결정한다. 그 다음 DeleteLine함수에서는 이중 loop를 이용해 필드를 탐색한다. 그 이후 완전히 채워진 줄을 발견하게 되면 그 줄을 기준으로 위로 올라가며 블록들을 한칸씩 내리게 된다. 이때의 시간복잡도는 필드의 가로와 세로를 의미하는 WIDTH, HEIGHT변수에 의존하게 되는데, 최악의 경우를 생각하면 제일 아래 4개의 줄이 완전히 채워진 경우이다. 이 줄들을 지울 때는 HEIGHT의 4배정도의 시간이 걸린다고 생각할 수 있다. 결과적으로 WIDTH와 HEIGTH의 제곱에 비례하는 시간복잡도를 가진다고 생각할 수 있다. 따라서, 블록이 한칸 내려갈 수 있는 경우에는 O(BLOCK\_WIDTH \* BLOCK\_HEIGTH) 이고, 블록이 더 이상 내려갈 수 없는 경우에는 O((WIDTH \* HEIGTH)^2)의 시간복잡도가 걸린다.

<<공간복잡도>>

배열 값의 할당 없이 timed\_out, blockY, score, nextBlock 등의 변수에 값을 저장해주는 정도의 공간이 필요하므로, 공간복잡도는 O(1)이다.

void BlockDown(int sig){

// user code

//강의자료 p26-27의 플로우차트를 참고한다.

// 블록이 한 칸 내려갈 수 있는 경우

if (CheckToMove(field, nextBlock[0], blockRotate, blockY+1, blockX) == 1){

// 블록을 한 칸 내리고

blockY++;

// 한 칸 아래로 내려온 블록을 콘솔에 그려준다.

DrawChange(field, KEY\_DOWN, nextBlock[0], blockRotate, blockY, blockX);

// 시간을 초기화해준다.

timed\_out = 0;

return;

}

// 블록이 한 칸 내려갈 수 없는 경우

// 블록의 y좌표가 -1 이면 gameOver = 1 로 변경해준다.

if (blockY == -1) gameOver = 1;

// 더 이상 내려갈 수 없음이 보장되었으므로, 현재의 블럭 정보를 필드에 반영해준다.

score += AddBlockToField(field, nextBlock[0], blockRotate, blockY, blockX);

// 추가점수를 반영해준다.

score += DeleteLine(field);

// 점수를 화면상에 그려준다.

PrintScore(score);

// 다음블럭을 현재블록으로 만들어주고, 다음블록은 0부터 6사이의 임의의 수를 저장해준다.

nextBlock[0] = nextBlock[1];

nextBlock[1] = rand() % 7;

// 다음 블록을 화면에 그려준다.

DrawNextBlock(nextBlock);

// 새롭게 내려올 블럭의 정보를 초기화해준다.

blockRotate = 0; // 회전횟수 : 0

blockY = -1; // y값 : -1

blockX = WIDTH/2 - 2; // x값 : 화면의 중간값

// 현재 필드정보를 블럭에 그려준다.

DrawField();

// 커서를 화면 바깥으로 빼준다.

move(HEIGHT, WIDTH+10);

// 시간을 초기화해준다.

timed\_out = 0;

return;

}

<AddBlockToField 함수>

정해진 실습시간 이전에 집에서 미리 실습을 진행하여, 예비보고서의 pseudo code와 실제 코드가 크게 달라진 점은 없다. 현재 블록의 x좌표, y좌표, 회전횟수, 블럭모양을 인자로 받은 후, 4\*4 block prototype을 이중 loop로 탐색하며 블록이 존재하는 부분을 필드에 반영해주는 함수이다.

<<시간복잡도>>

현재 block prototype에서 어느부분에 블록이 존재하는지를 알아내기 위해 이중 loop를

이용하여 탐색한다. 이때, BLOCK\_WIDTH, BLOCK\_HEIGHT가 이중 loop의 시간복잡도에 영향을 미치게 된다. 따라서, 시간복잡도는 O(BLOCK\_WIDTH \* BLOCK\_HEIGHT) 이다.

<<공간복잡도>>

반복문을 위한 I, j와, 배열값 할당을 위한 부분이 있으므로 공간복잡도는 O(1)이다.

Void AddBlockToField(char f[HEIGHT][WIDTH],int currentBlock,int blockRotate, int blockY, int blockX){

// user code

//Block이 추가된 영역의 필드값을 바꾼다.

int score;

// block이 있는 곳을 field에도 반영해준다.

for (int i = 0; i < 4; i++){

for (int j = 0; j < 4; j++){

if (block[currentBlock][blockRotate][i][j] == 1){

f[blockY+i][blockX+j] = 1; // 1은 블럭이 존재함을 의미한다.

}

}

}

}

<DeleteLine 함수>

정해진 실습시간 이전에 집에서 미리 실습을 진행하여, 예비보고서의 pseudo code와 실제 코드가 크게 달라진 점은 없다. 필드 전체를 탐색하며, 완전히 채워진 줄을 발견하면 deletedLines변수를 1 증가시키고, 그 줄 위쪽에 쌓여있는 블록을 모두 한칸씩 내려주는 과정을 진행한다. 결과적으로 deletedLines^2 \* 100을 리턴값으로 주고 이는 추가해야할 점수를 나타낸다.

<<시간복잡도>>

기본적으로 필드 전체를 탐색해야하기 때문에 필드의 가로와 세로를 나타내는 WIDTH와 HEIGHT 변수에 걸리는 시간이 의존하게 된다. 이 과정이 진행되면서, 만약 완전히 채워진 줄을 발견했다면 그 줄을 기준으로 필드의 위쪽부분을 전체적으로 탐색하며 모든 줄을 한칸씩 내려주어야한다. 최대 4개까지 한번에 지울 수 있으며 하나의 행의 모든 열을 지워야하기 때문에 WIDTH 변수에 비례하게 된다. 결국, 최대로 걸리는 시간은 약 4\* WIDTH \* HEIGHT 가 된다. 따라서 시간복잡도는 O(WIDTH \* (HEIGHT)^2)가 된다.

<<공간복잡도>>

isFull, deletedLines, addedScore 변수를 이용하고, 완전히 채워진 블록이 존재한다면 한칸씩 내려주는 과정이 필요한데, 여기서는 배열의 값을 참조하고, 그 참조값을 배열에 다시 할당해주는 형태로 프로그램이 진행된다. 따라서 공간복잡도는 O(1)이다.

int DeleteLine(char f[HEIGHT][WIDTH]){

// user code

//1. 필드를 탐색하여, 꽉 찬 구간이 있는지 탐색한다.

//2. 꽉 찬 구간이 있으면 해당 구간을 지운다. 즉, 해당 구간으로 필드값을 한칸씩 내린다.

int isFull; // 현재 탐색중인 줄이 완전히 채워져 있는지의 여부를 담는 변수

int deletedLines = 0; // 지운 line의 개수를 저장하는 변수. 0으로 초기화해준다.

int addedScore; // 추가 될 점수를 저장하는 변수

// 각각의 행을 탐색하여, 완전히 채워진 줄이 있으면 isFull 가 1이므로 그 줄을 삭제해주고, 채워지지 않은 줄은 isFull 변수가 0이므로 다음줄을 탐색하러 간다.

for (int i =0; i < HEIGHT; i++){

isFull = 1; // 기본적으로 isFull 변수는 1로 설정해준다. 1이면 삭제를 진행해준다.

for (int j = 0; j < WIDTH ; j++){

if (f[i][j] == 0){ // 완전히 채워지지 않은 줄이면

isFull = 0; // 채워지지 않았다는 의미로 isFull 변수를 0으로 설정해준다.

break; // 루프를 탈출한다.

}

}

// 현재 탐색 중인 줄이 완전히 채워져 있다면

if (isFull){

// 지운 line의 수를 하나 늘려준다.

deletedLines++;

// k는 i부터 0까지 변화하는 변수이고,

// k번째 줄에k-1번째 줄의 내용을 덮어씌워준다.

for (int k = i; k >= 0; k--){

if (k == 0){ // 가장 윗줄은 더 윗줄에서 가져올 정보가 없으므로 0(빈칸으로 정보를 초기화해준다.)

for (int l = 0; l < WIDTH; l++)

f[k][l] = 0; // 빈칸의 의미로 0을 넣어준다.

}

else{ // 가장 윗줄이 아니라면

for (int l = 0 ; l < WIDTH ; l++)

f[k][l] = f[k-1][l]; // 윗 행의 정보를 아랫줄에 덮어씌워준다.

}

}

}

}

// (지워진라인)^2 \* 100 를 추가점수로 부여해준다.

addedScore = deletedLines\*deletedLines\*100;

return addedScore; // 추가점수를 반환해준다.

}

1. 1주차 숙제 문제는 아래의 3문제로 이루어져 있다.

* 블록이 필드에 쌓이게 될 부분을 나타내주는 그림자를 그리는 기능 구현
* 다음블록의 개수를 1개에서 2개로 늘리기
* 점수 배점 방식을 필드와 블록의 아랫부분이 닿는 면적도 포함하여 계산

< 1. 그림자 기능 >

🡪 DrawShadow 함수를 새롭게 구현.

CheckToMove 함수를 이용해 블록이 가장 아랫부분까지 내려갔을 때의 블럭정보를 받아온 후, 그 정보를 바탕으로 DrawBlock함수를 이용하여 ‘/’ 모양을 그려준다.

<시간복잡도>

CheckToMove함수의 시간복잡도는 O(BLOCK\_WIDTH \* BLOCK\_HEIGHT)이고, while loop는 최대 필드의 높이, 즉 HEIGHT에 의존하는 시간복잡도를 갖는다. 그리고 아래 DrawBlock 함수 또한 O(BLOCK\_WIDTH \* BLOCK\_HEIGHT)의 시간복잡도를 가지므로, 이들을 모두 합하면 시간복잡도는 O(BLOCK\_WIDTH \* BLOCK\_HEIGHT \* HEIGHT)가 된다.

<공간복잡도>

변수 y의 값을 조건이 충족된다면 1씩 증가시켜주는 과정 말고 다른 메모리는 필요로 하지 않는다. 따라서 공간복잡도는 O(1)이다.

아래는 실제 DrawShadow 함수의 코드이다. 자세한 코드의 설명을 주석을 통해 확인할 수 있다.

void DrawShadow(int y, int x, int blockID,int blockRotate){

// 더 이상 내려갈 수 없을 때의 위치를 찾기 위한 반복문이다.

while (CheckToMove(field, blockID, blockRotate, y+1, x)) y+= 1;

// 더 이상 내려갈 수 없는 위치를 찾았다면 그 지점에 '/'로 그림자를 그려준다.

DrawBlock(y, x, blockID, blockRotate, '/');

}

🡪 DrawBlockWithFeatures 함수를 새롭게 구현.

DrawBlock함수와 DrawShadow함수를 호출하는 함수이다. 그리고 이 함수는 이전의 BlockDown함수의 블록이 한칸 더 내려갈 수 있다고 판단하는 조건문안의 DrawBlock함수 대신 입력되는 함수이다.

<<시간복잡도>>

DrawBlock함수는 O(BLOCK\_WIDTH \* BLOCK\_HEIGHT) 이고, DrawShadow함수는 O(BLOCK\_WIDTH \* BLOCK\_HEIGHT \* HEIGHT) 가 되므로 이 둘을 합하면 시간복잡도는 O(BLOCK\_WIDTH \* BLOCK\_HEIGHT \* HEIGHT) 가 된다.

<<공간복잡도>>

새로운 변수없이 DrawBlock함수와 DrawShadow함수를 호출해주는 함수인데, 두 함수 모두 공간복잡도가 O(1)이므로 공간복잡도가 O(1)이다.

아래는 실제 DrawBlockWithFeatures 함수의 코드이다. 자세한 코드의 설명을 주석을 통해 확인할 수 있다.

void DrawBlockWithFeatures(int y, int x, int blockID, int blockRotate){

// 블록의 새로운 위치를 그려준다.

DrawBlock(y, x, blockID, blockRotate, ' ');

// 블록의 새로운 그림자를 그려준다.

DrawShadow(y, x, blockID, blockRotate);

}

< 2. 다음블록개수 1개에서 2개로 늘리기 >

🡪 tetris.h의 nextBlock 배열

현재 nextBlock배열의 크기는 2다. 현재 게임에서 사용중인 블럭과 바로 그 다음 블럭을 저장하기 위해 만든 배열이다. 이때, 크기를 3으로 바꿔주게되면 현재 사용중인 블럭과 그 다음으로 사용할 두개의 블럭을 저장할 수 있게 된다.

🡪 InitTetris 함수의 일부분 수정

nextBlock[2] = rand() % 7 부분을 추가해줌으로써 다다음 차례에 등장할 블럭의 모양을 저장해준다. 추가된 부분의 시간복잡도는 하나의 대입연산이 진행된 것이므로 O(1)이고, 추가된 부분의 공간복잡도는 하나의 배열값을 할당해 주는 것이므로 이 또한 O(1)이라고 생각할 수 있다.

🡪 DrawNextBlock 함수의 일부분 수정

원래는 nextBlock[1]만 그려주었지만, nextBlock[2]를 그려주는 부분을 추가하여 다음블록과 다다음블록을 모두 그려주도록 함수를 수정하였다.

<<시간복잡도>>

시간복잡도는 다음블록과 다다음 블록을 그려주는 과정에서 4\*4 block prototype를 탐색해야하는데, 이때 걸리는 시간복잡도는 O(BLOCK\_WIDTH \* BLOCK\_HEIGHT) 이다.

<<공간복잡도>>

반복문을 위한 I, j와 배열값 참조를 통해 조건문을 확인하는 과정이 포함되어 있으므로 공간복잡도는 O(1)이다.

아래는 실제 DrawNextBlock 함수의 코드이다. 자세한 코드의 설명을 주석을 통해 확인할 수 있다.

void DrawNextBlock(int \*nextBlock){

int i, j;

for( i = 0; i < 4; i++ ){

// nextBlock[1] 그리기

move(4+i,WIDTH+13); // 커서를 y축방향을 4+i만큼, x축 방향으로 WIDTH+13만큼 떨어진 지점으로 옮긴다.

for( j = 0; j < 4; j++ ){

if( block[nextBlock[1]][0][i][j] == 1 ){

attron(A\_REVERSE);

printw(" ");

attroff(A\_REVERSE);

}

else printw(" ");

}

// nextBlock[2] 그리기

move(10+i, WIDTH+13); // 커서를 y축방향을 10+i만큼, x축 방향으로 WIDTH+13만큼 떨어진 지점으로 옮긴다.

for (j = 0; j < 4 ; j++){

if ( block[nextBlock[2]][0][i][j] == 1 ){

attron(A\_REVERSE);

printw(" ");

attroff(A\_REVERSE);

}

else printw(" ");

}

}

}

🡪 BlockDown함수의 일부분 수정

블록을 더 이상 내릴 수 없는 조건문안으로 들어왔을때, 기존에는 다음블록을 랜덤하게 선택하는 부분이 하나 있었다. 그러나 이 부분을 아래와 같이 수정해주어야 한다.

nextBlock[0] = nextBlock[1];

nextBlock[1] = nextBlock[2];

nextBlock[2] = rand()%7;

추가된 부분의 시간복잡도는 한줄의 값 대입부분이 추가된 것이므로 O(1)이라고 생각할 수 있고, 추가된 부분의 공간복잡도는 하나의 배열공간이 함수의 진행과정에서 더 필요하므로 이 또한 O(1)이라고 생각할 수 있다.

🡪 DrawOutline 함수의 일부분 수정

이 함수의 기존의 역할은 nextBlock을 출력해줄 네모칸과, 점수를 출력해줄 네모칸을 그려주는 것이었다. 그러나 현재 nextBlock이 두개가 그려져야 되므로, 바로 다음블록을 그리고 조금 아래에 다다음블록을 그리고, 조금 아래에 점수를 출력해줄 네모칸을 그려주도록 수정하였다.

다음블럭이 그려질 테두리를 그려주는 부분이 하나 추가된 것이라 원래 시간은 두배정도가 걸릴 것이지만, 시간복잡도 측면에서 본다면 변화가 없다. 공간복잡도 또한 원래 공간의 두배정도가 필요하지만 공간복잡도 측면에서 본다면 변화가 없다.

아래는 실제 DrawOutline 함수의 코드이다. 자세한 코드의 설명을 주석을 통해 확인할 수 있다.

void DrawOutline(){

int i,j;

/\* 블럭이 떨어지는 공간의 태두리를 그린다.\*/

DrawBox(0,0,HEIGHT,WIDTH);

/\* next block1을 보여주는 공간의 태두리를 그린다.\*/

move(2,WIDTH+10);

printw("NEXT BLOCK");

DrawBox(3,WIDTH+10,4,8);

/\* next block2를 보여주는 공간의 테두리를 그린다.\*/

DrawBox(9, WIDTH+10, 4, 8);

/\* score를 보여주는 공간의 태두리를 그린다.\*/

move(15,WIDTH+10);

printw("SCORE");

DrawBox(16,WIDTH+10,1,8);

}

🡪 printScore 함수의 일부분 수정

새롭게 등장할 블록을 표시해주는 부분이 1개에서 2개로 늘어나서, 점수 표시해주는 부분도 조금 아래로 내려야 한다. 따라서, 커서를 move(17,WIDTH+11); 을 이용하여 올바른 위치에 이동시켜준 후 점수를 출력해준다. 시간복잡도와 공간복잡도의 변화는 없다.

< 3. 새로운 점수 계산 방식>

🡪 AddBlockToField 함수의 일부분 수정

Touched변수를 새롭게 생성하여 필드와 블록의 바닥면이 몇부분 닿았는지 체크하여 저장한다. 이를 위해서는 현재 블록을 이중 loop를 이용해서 탐색하며, 블록이 존재하고, 블록의 아래부분의 필드가 범위 바깥이거나, 블록이 있다면 touched변수를 1증가시킨다. 이러한 과정을 통해 얻은 touched변수는 새로운 점수 계산 방식에 의해 Touched \* 10을 함수의 리턴값을 주게한다.

<<시간복잡도>>

블록을 이중 loop를 이용하여 탐색해야하기 때문에 O(BLOCK\_WIDTH \* BLOCK\_HEIGHT)의 시간복잡도가 걸린다. Touched변수의 증가여부를 판단하기 위한 조건문들의 시간복잡도는 모두 O(1)으로 생각해도 무방하다.

<<공간복잡도>>

Touched와 score변수를 추가로 선언하여, 필드와 블록의 아랫부분이 맞닿은 면적을 저장하고, 점수를 저장한다. 이를 통해 공간복잡도는 O(1)임을 알 수 있다.

아래는 실제 AddBlockToField 함수의 코드이다. 자세한 코드의 설명을 주석을 통해 확인할 수 있다.

int AddBlockToField(char f[HEIGHT][WIDTH],int currentBlock,int blockRotate, int blockY, int blockX){

// user code

//Block이 추가된 영역의 필드값을 바꾼다.

int touched = 0; // 맞닿은 부분의 개수를 저장하는 변수

int score;

// block이 있는 곳을 field에도 반영해준다.

for (int i = 0; i < 4; i++){

for (int j = 0; j < 4; j++){

if (block[currentBlock][blockRotate][i][j] == 1){

f[blockY+i][blockX+j] = 1; // 1은 블럭이 존재함을 의미한다.

if ( blockY+i+1 >= HEIGHT) touched += 1;

else if (f[blockY+i+1][blockX+j] == 1) touched += 1;

}

}

}

score = touched \* 10;

return score;

}

🡪 BlockDown 함수의 일부분 수정

AddBlockToField 함수를 통해 필드와 블록의 아랫부분과 맞닿은 개수에 비례하는 점수를 얻어, score에 추가해준다. 시간복잡도는 원래의 코드의 그것과 다르지 않은 O(WIDTH \* (HEIGTH)^2)이다. 공간복잡도는 변화가 없다.