**Artificial Intelligence**

Assignment1 : IDS, GBS, ASS

2016025532 컴퓨터전공 심수정

**1. code compile version**

모든 코드는 python3.6 version을 기준으로 작성되었습니다. 따라서 python3 version이 설치되어 있지 않다면 이를 설치할 필요가 있습니다.

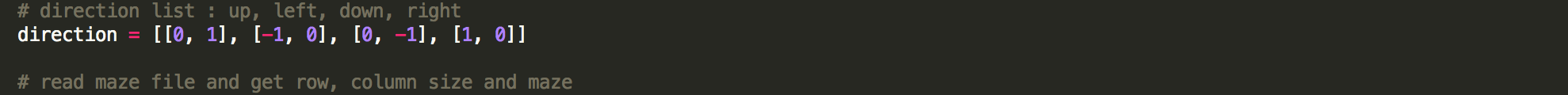
“**python3 [filename]**”을 입력한다면 해당 코드를 컴파일, 실행할 수 있습니다.

**2. code review**

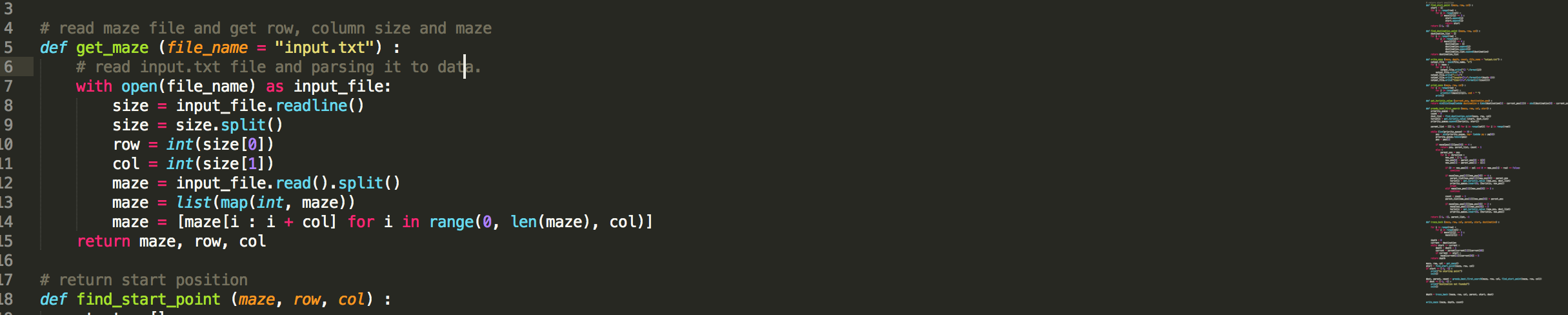
code는 function 단위로 캡쳐되었고, 설명되었습니다. IDS, GBS, ASS의 공통된 함수, GBS, ASS의 공통된 함수, IDS에만 존재하는 함수, GBS에만 존재하는 함수, ASS에만 존재하는 함수 순으로 나옵니다.

**1) common**

- IDS, GBS, ASS



탐색이 가능한 방향을 direction list에 넣어줍니다. 차례대로 위, 왼, 아래, 오른 방향입니다.



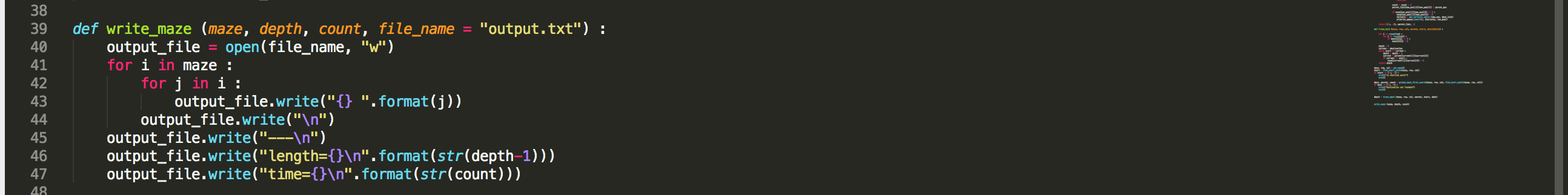
input file(input.txt)에서 row, column의 크기를 가져온 후 나머지 maze를 list of list의 형태로 저장해줍니다. 이 때 maze안의 element들은 int로 저장합니다.



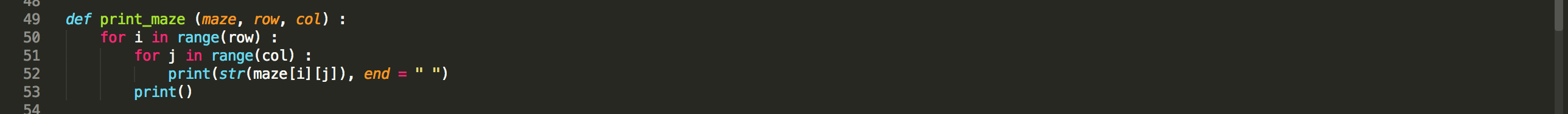
starting position의 위치를 찾아 list = [x, y]의 형태로 반환합니다. 만약 maze 내에 starting position이 없다면 [-1, -1]의 형태로 반환합니다.



destination position의 위치를 찾아 list = [x, y]의 형태로 destination list에 넣어줍니다. 모든 maze의 element들을 확인한 후 destination list를 반환합니다.

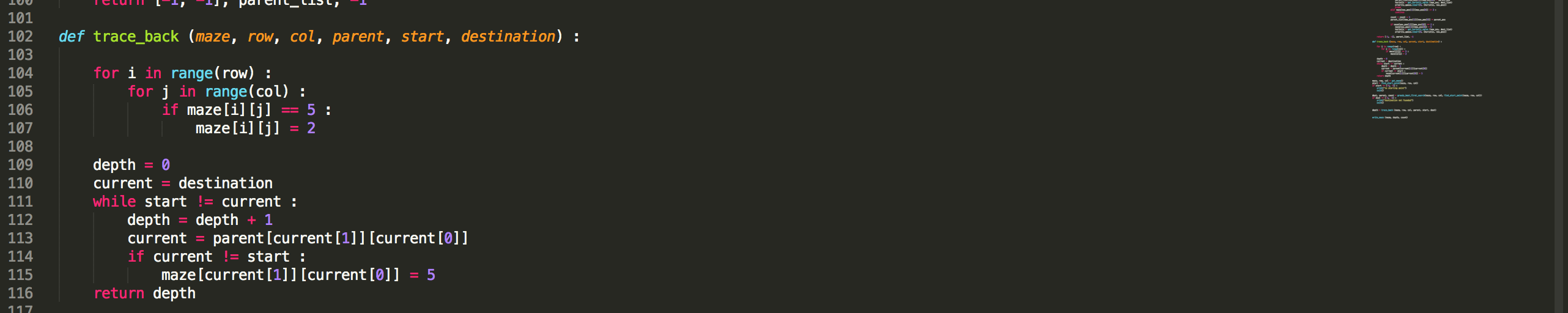


maze, depth, count를 받아 output file(output.txt)에 maze를 쓰고, “---\n”을 출력한 후 length=($depth)\n, time=($count)\n을 씁니다.

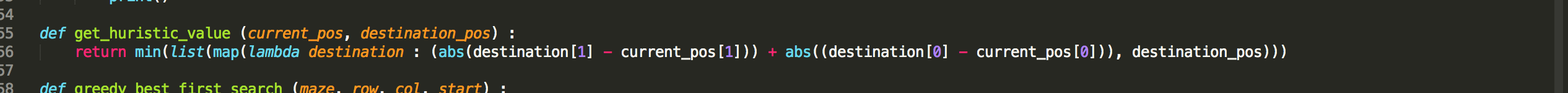


maze를 행렬의 형태로 출력합니다.

- GBS, ASS

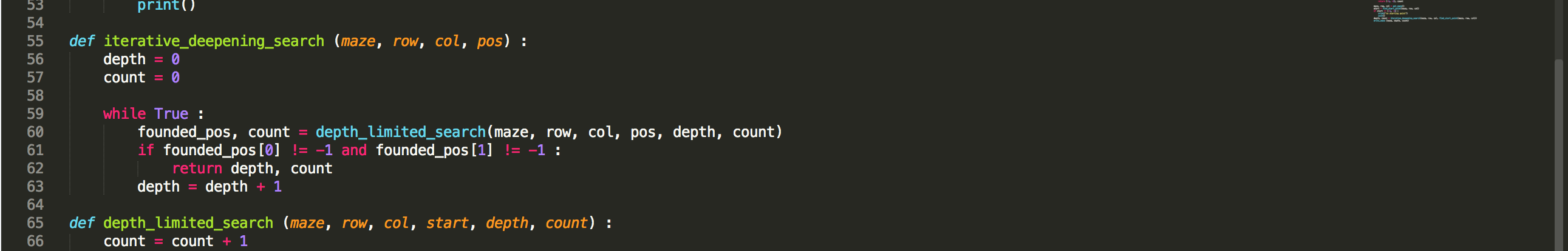


algorithm을 수행하는 동안 visited, 즉 5로 표시된 것을 다시 unvisited, 즉 2로 다시 돌립니다. 그리고 도착한 목적지에서부터 그 전의 경로를 추적하며 start position의 위치를 찾아갑니다. 동시에 최종 depth가 몇이였는지 추적합니다.



destination position list의 element들과 맨해튼 거리를 계산하여 그 중 가장 작은 값을 return합니다.

**2) IDS**



탐색 level을 세기 위한 depth, 총 time을 세기 위한 count를 0으로 지정한 후 depth limited search하며 매 count를 저장하고, 찾아진 위치가 [-1, -1]이 아니면 현재의 depth, count를 리턴합니다. 만약 [-1, -1], 즉 찾아지지 않았다면 depth를 증가시킵니다.

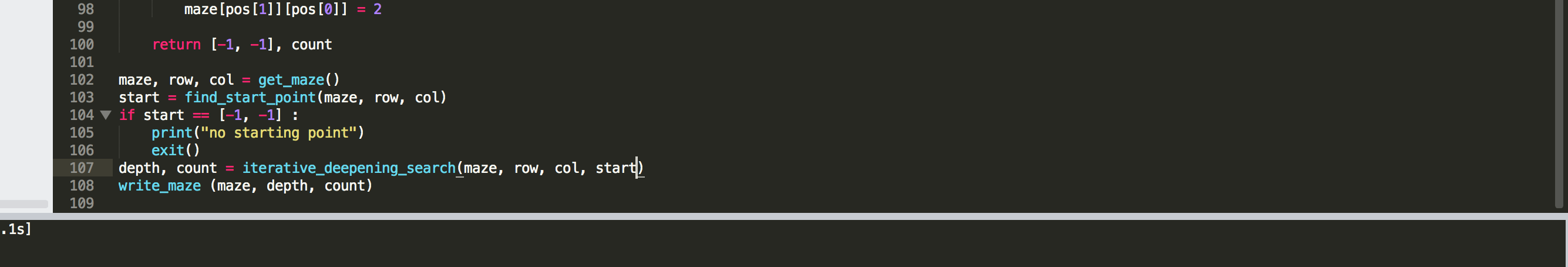


우선 count를 증가시키고, 새로 받은 start position이 도착점이라면 현 position, count를 return해줍니다. 그렇지 않은데 최대 depth가 0이거나 그보다 작아졌다면 [-1, -1]과 count를 return해줍니다.

Return되지 않았다면 Direction list를 통하여 각 방향별 위치를 구한 후 맵 사이즈를 넘어서는 범위가 아닌지 확인한 후에 4나 2가 아니면, 즉 도착점이 아니거나 방문하지 않은 길이 아니면 이 경우는 끝내고, 도착점이나 방문하지 않은 길이라면 계속 실행합니다. 이 때 만약 2였다면 visited 표시, 즉 5로 표시해주고, 이 위치에 대해 depth limited search를 실행합니다.

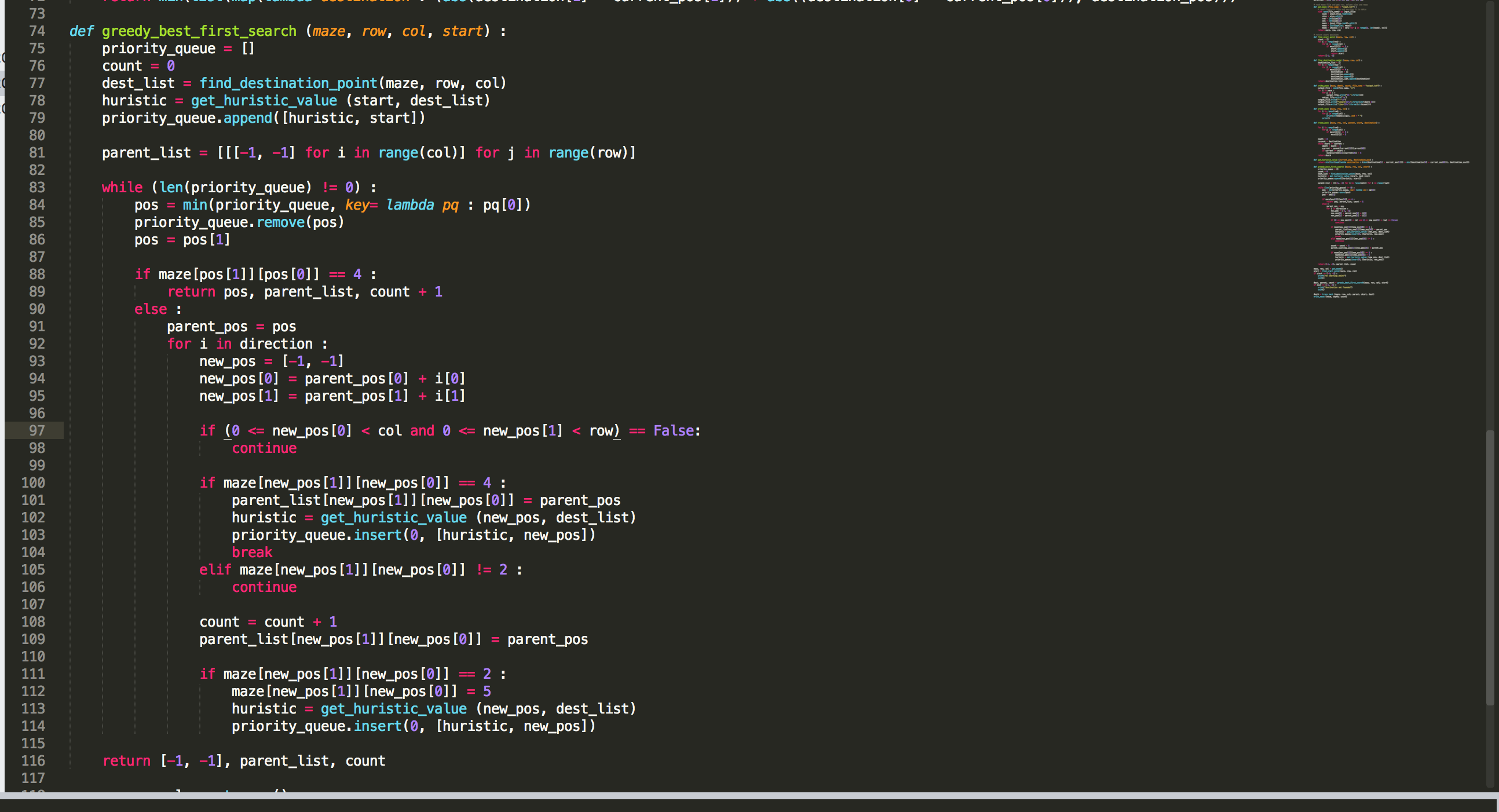
만약 도착점 4가 찾아졌다면 이는 종료점의 위치와 count의 정상적인 값, [-1, -1]이 아닌 값을 받게 되는데 이 경우 이 함수를 종료하게 되고, 그렇지 않다면 다시 unvisited 표시, 즉 2로 바꿔줍니다.

정상적인 값이 내 위치에서 찾아지지 않았다면 정상적이지 않은 위치 [-1, -1]과 현재 count를 return합니다.



maze와 maze의 크기를 받은 후, 출발점을 찾고 이 값이 존재하는지 확인합니다. 후에 iterative deepening search를 통해 depth, count를 찾고 이를 maze와 함께 output file에 써주게 됩니다.

**3) GBS**



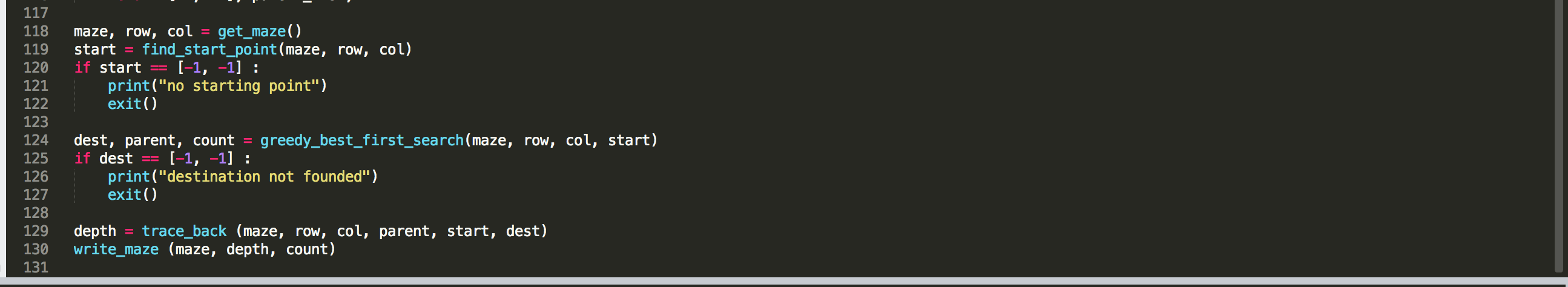
우선 priority queue를 list로 생성하고, destination position list를 구한 후 현재 start position의 huristic을 구해 start position과 함께 priority queue에 넣어줍니다. 또한 parent list도 maze의 크기만큼 [-1, -1]로 채워 생성합니다.

Priority queue가 비지 않은 동안 이 중 가장 huristic이 가장 작은 값을 골라 제거하고, pos에 위치만 넣어줍니다.

만약 pos가 도착점이였다면 pos, parent\_list, count+1을 return하여주고, 그렇지 않다면 parent\_pos에 현재 pos를 저장한 후 각 방향을 구합니다. 각 각 방향별 위치로 map size를 넘지 않았는지 확인하고 넘지 않았으면 이가 4인지, 즉 도착점인지 확인합니다. 도착점이라면 parent list에 parent\_pos를 넣고, 현재 위치의 huristic을 구해 현재 위치와 함께 priority queue의 가장 첫 위치에 넣어줍니다. 그렇지 않다면 2인지 확인하고 2라면, 즉 방문하지 않은 길이라면 다음 명령어들을 실행합니다.

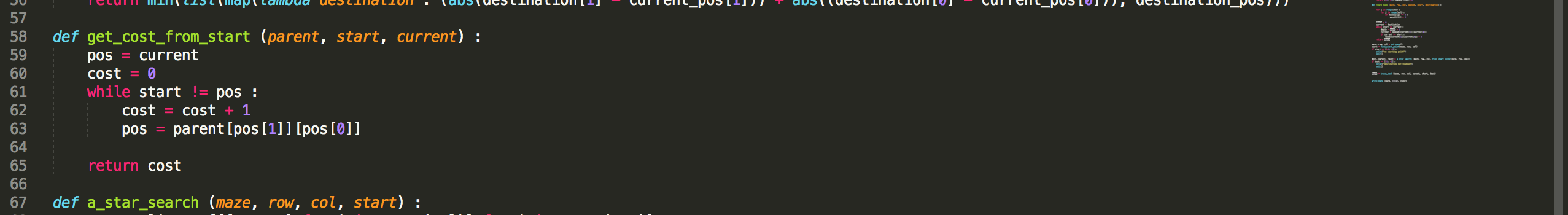
Count의 값을 1만큼 올리고, 현 위치의 parent를 parent list에 갱신합니다. 또한 현재 길이 방문하지 않은 길이라면 방문한 길, 즉 5로 표시하고, 현 위치의 휴리스틱을 구해 현 위치와 함께 priority queue의 가장 첫 위치에 넣어줍니다.

만약 priority queue의 길이가 0이 된다면 이 maze의 도착점은 찾아지지 않으므로 [-1, -1]의 유효하지 않은 위치와 parent list, 현재 count를 return합니다.

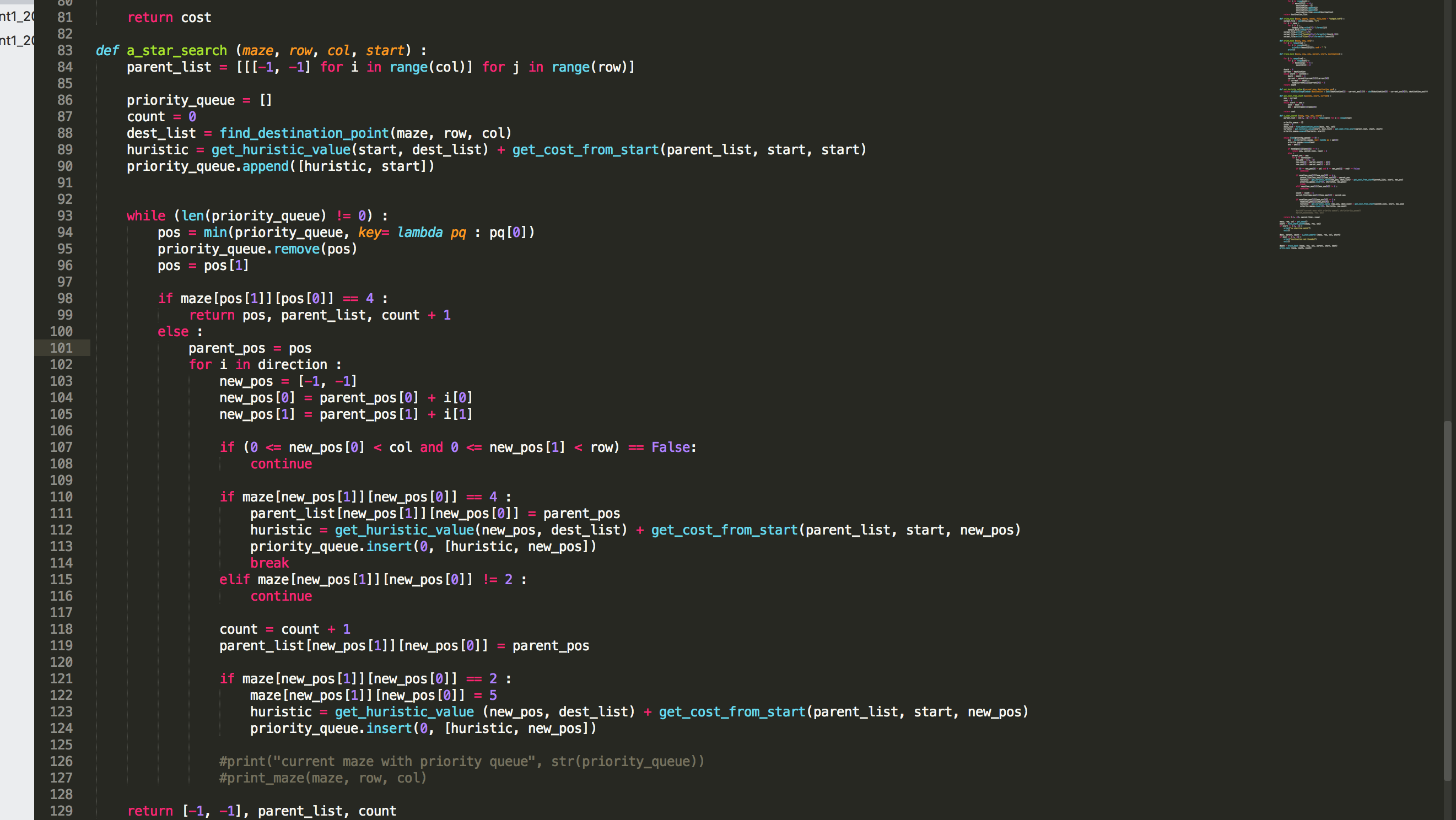
****

maze와 maze의 크기를 받은 후, 출발점을 찾고 이 값이 존재하는지 확인합니다. 후에 greedy best first search를 통해 최종 목적지의 위치, maze element들의 parent list, count를 받습니다. 만약 최종 목적지가 안찾아진 경우라면 종료합니다. 아니라면 depth를 trace back을 통해 찾고 이를 maze, count와 함께 output file에 써주게 됩니다.

**4) ASS**



parent list를 통한 traceback으로 start position부터 현재까지 몇 칸을 왔는지 계산하여 return합니다.



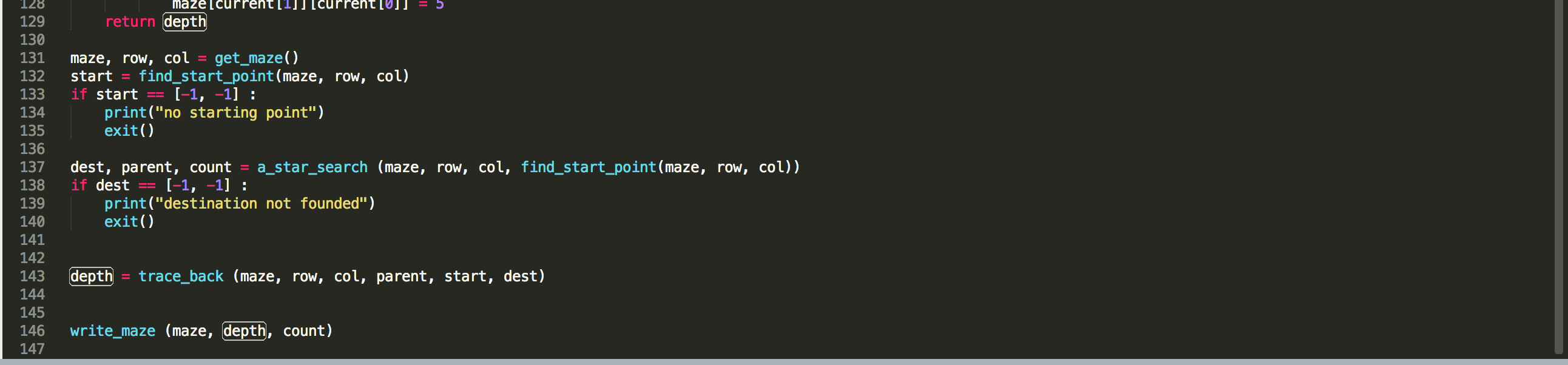
우선 priority queue를 list로 생성하고, destination position list를 구한 후 현재 start position의 huristic과 현재까지의 cost를 구해 더하여 start position과 함께 priority queue에 넣어줍니다. 또한 parent list도 maze의 크기만큼 [-1, -1]로 채워 생성합니다.

Priority queue가 비지 않은 동안 이 중 가장 huristic 과 현재까지의 cost의 합이 가장 작은 값을 골라 제거하고, pos에 위치만 넣어줍니다.

만약 pos가 도착점이였다면 pos, parent\_list, count+1을 return하여주고, 그렇지 않다면 parent\_pos에 현재 pos를 저장한 후 각 방향을 구합니다. 각 각 방향별 위치로 map size를 넘지 않았는지 확인하고 넘지 않았으면 이가 4인지, 즉 도착점인지 확인합니다. 도착점이라면 parent list에 parent\_pos를 넣고, 현재 위치의 huristic과 현재까지의 cost의 합을 구해 현재 위치와 함께 priority queue의 가장 첫 위치에 넣어줍니다. 그렇지 않다면 2인지 확인하고 2라면, 즉 방문하지 않은 길이라면 다음 명령어들을 실행합니다.

Count의 값을 1만큼 올리고, 현 위치의 parent를 parent list에 갱신합니다. 또한 현재 길이 방문하지 않은 길이라면 방문한 길, 즉 5로 표시하고, 현 위치의 huristic과 현재 위치까지의 cost의 합을 구해 현 위치와 함께 priority queue의 가장 첫 위치에 넣어줍니다.

만약 priority queue의 길이가 0이 된다면 이 maze의 도착점은 찾아지지 않으므로 [-1, -1]의 유효하지 않은 위치와 parent list, 현재 count를 return합니다.

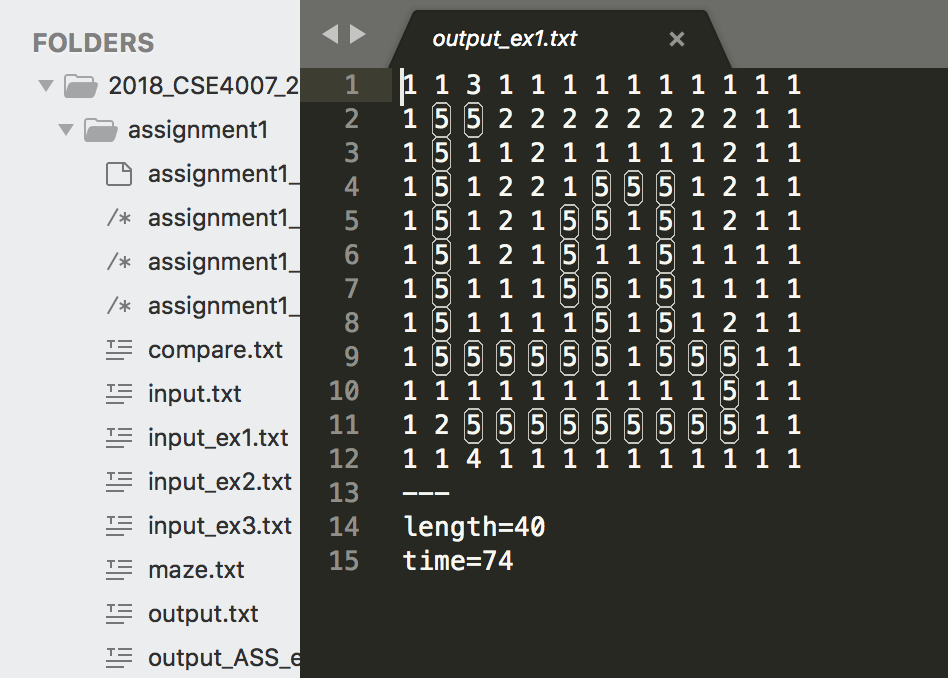


maze와 maze의 크기를 받은 후, 출발점을 찾고 이 값이 존재하는지 확인합니다. 후에 greedy best first search를 통해 최종 목적지의 위치, maze element들의 parent list, count를 받습니다. 만약 최종 목적지가 안찾아진 경우라면 종료합니다. 아니라면 depth를 trace back을 통해 찾고 이를 maze, count와 함께 output file에 써주게 됩니다.

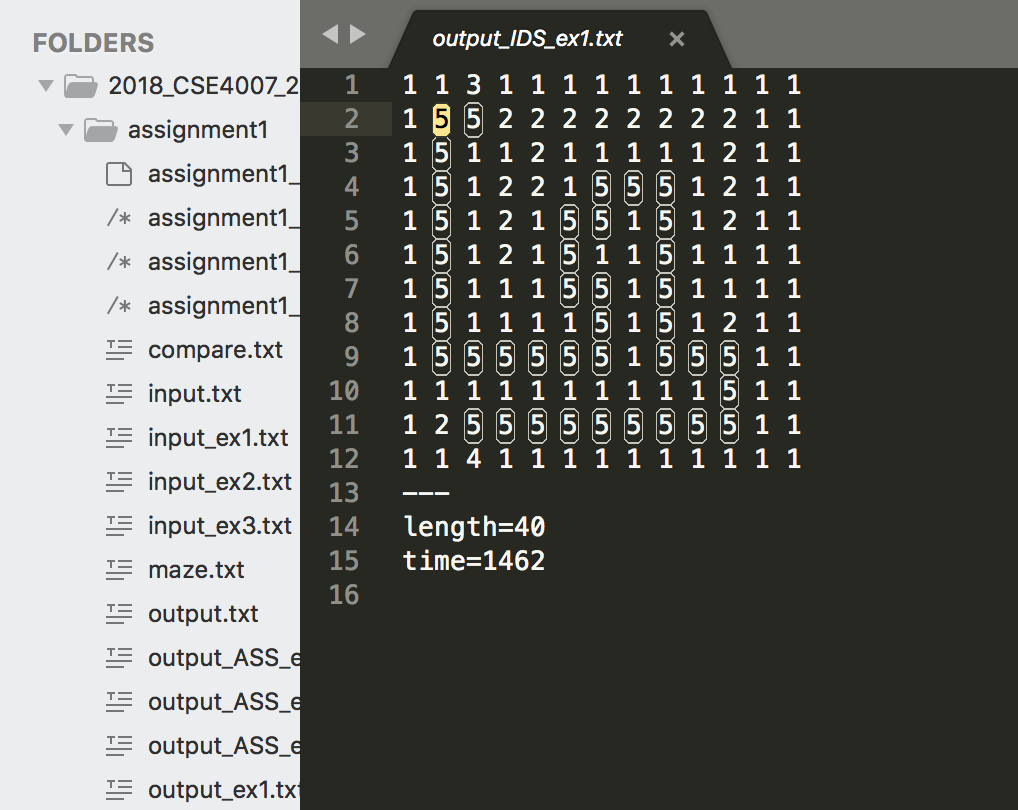
**3. result**

**1) example1**

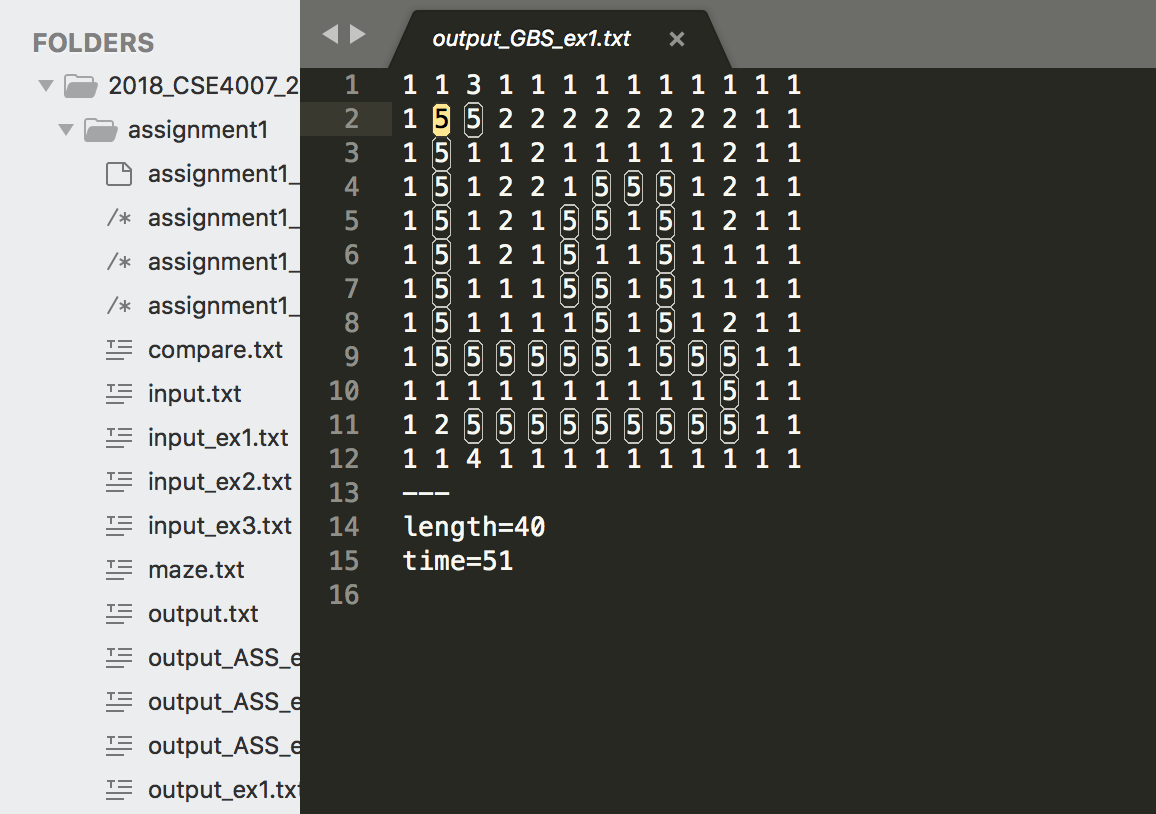
(0) Answer



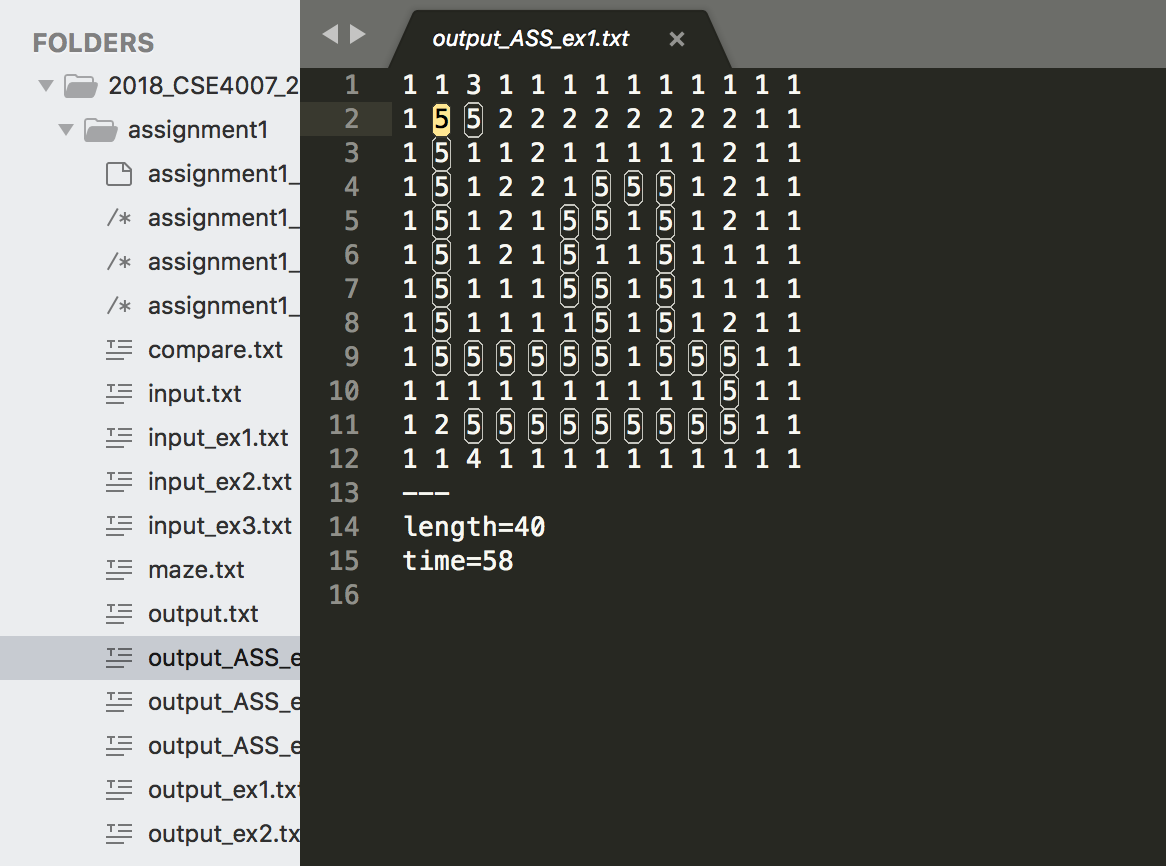
(1) IDS



(2) GBS



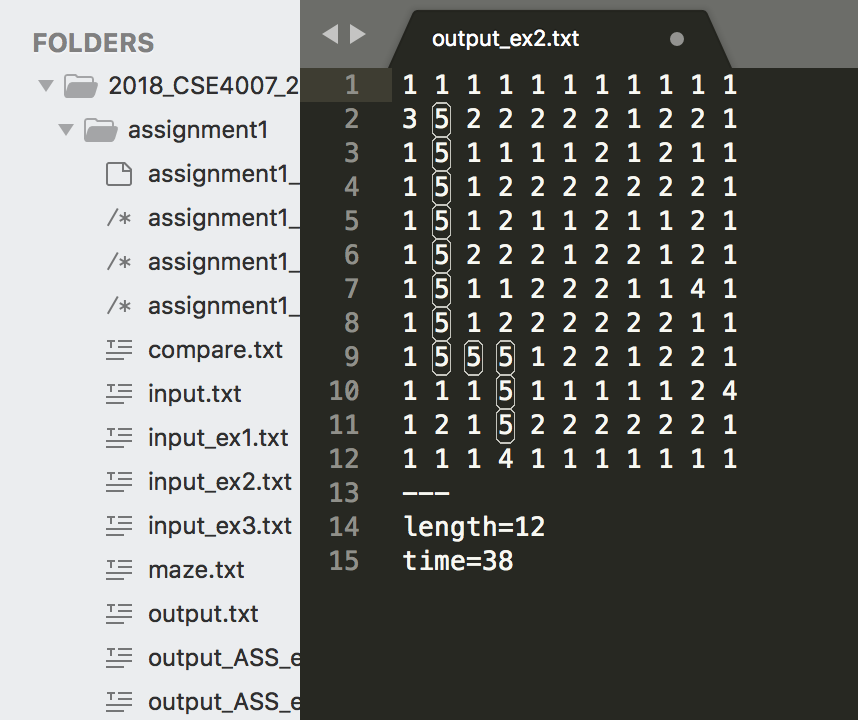
(3) ASS



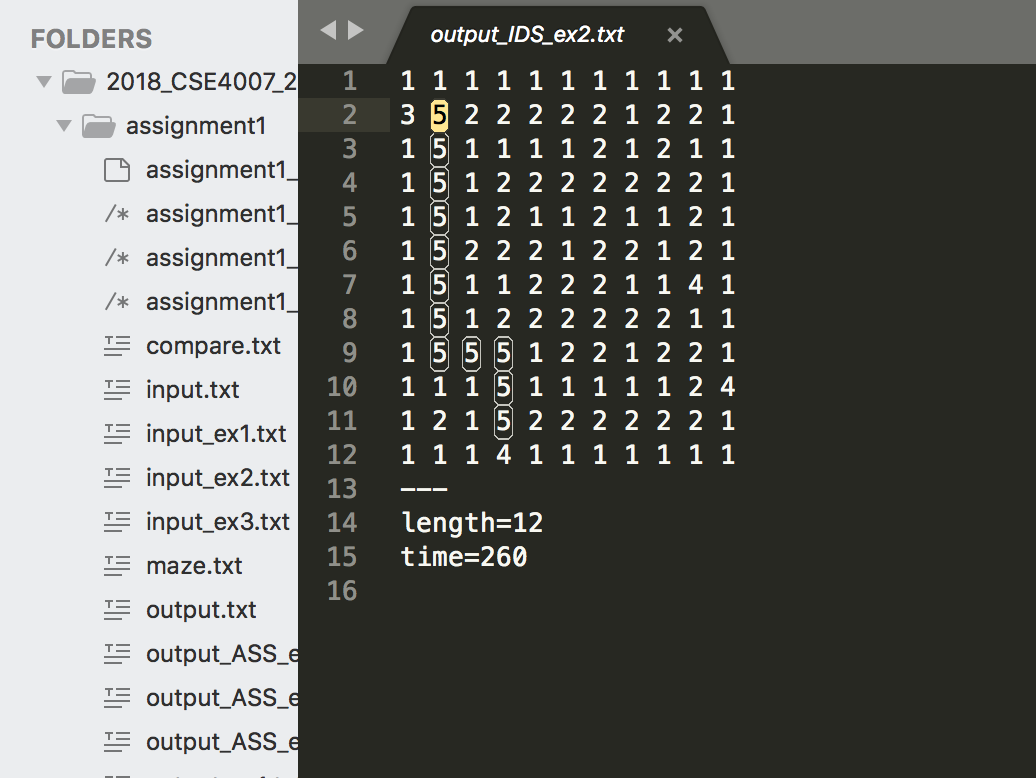
이 경우 IDS, GBS, ASS 모두 optimal한 solution을 찾는 것을 확인할 수 있습니다. 하지만 총 시간은 GBS > ASS > IDS 순으로 빠른 것을 확인할 수 있습니다.

**2) example2**

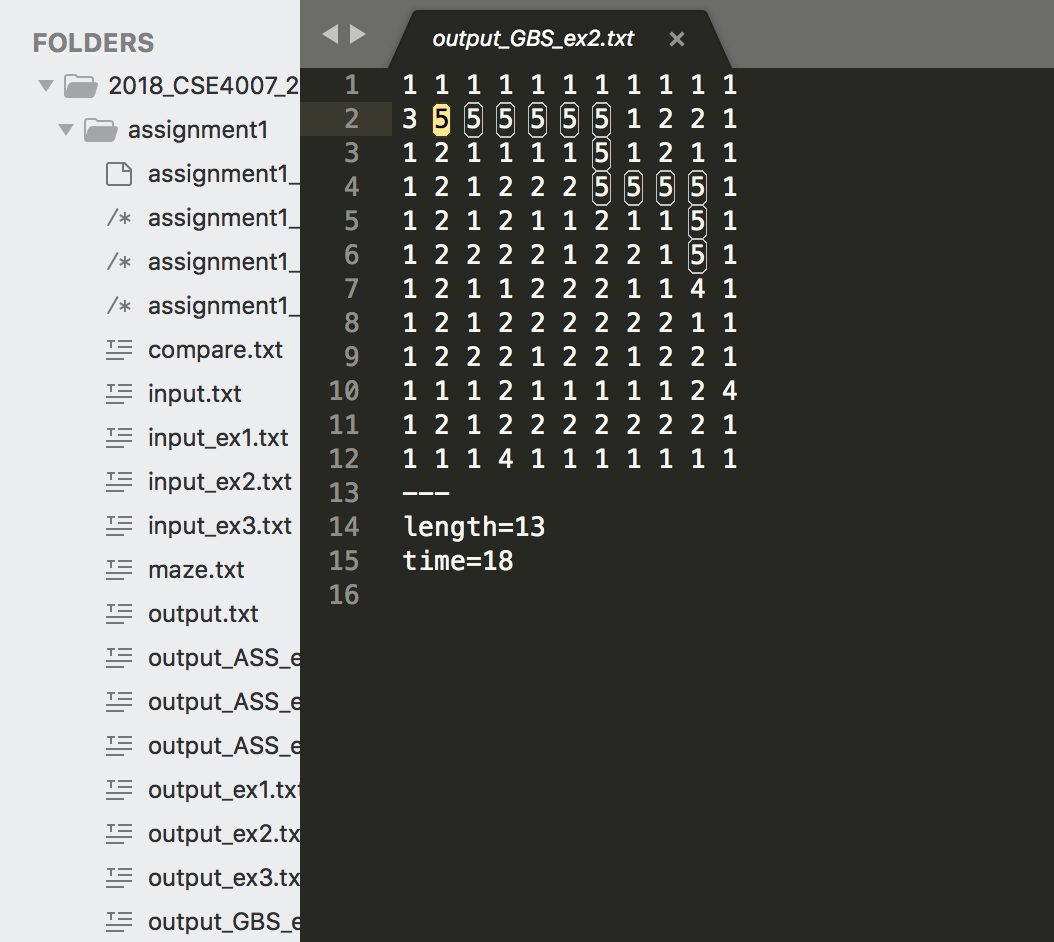
(0) Answer



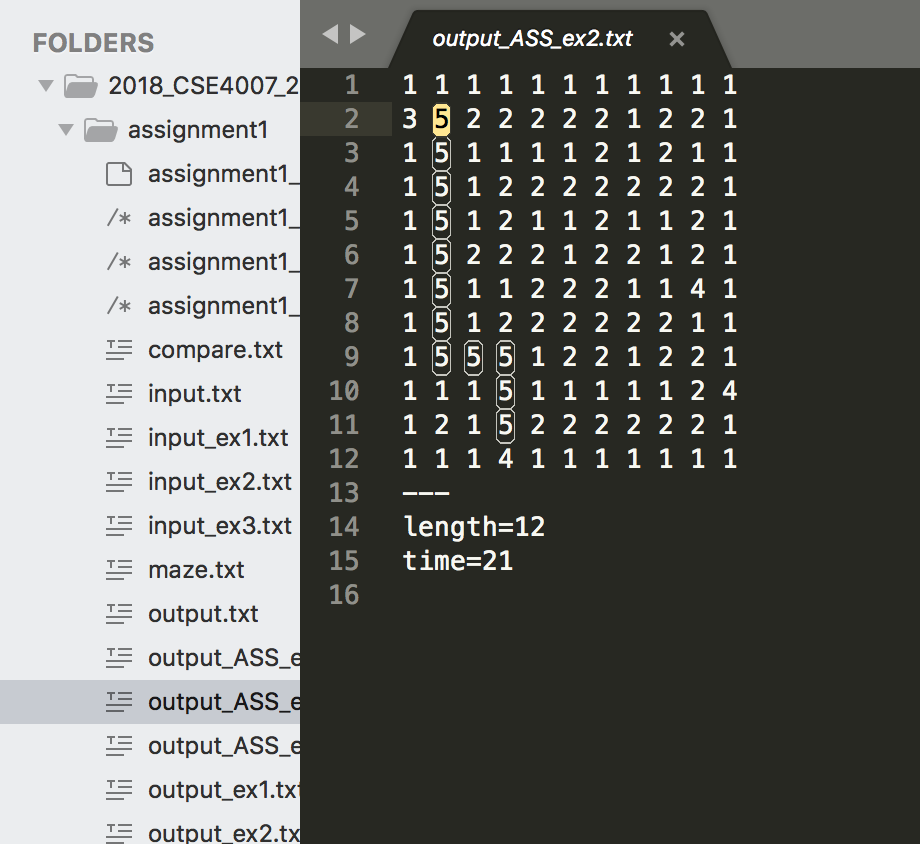
(1) IDS



(2) GBS



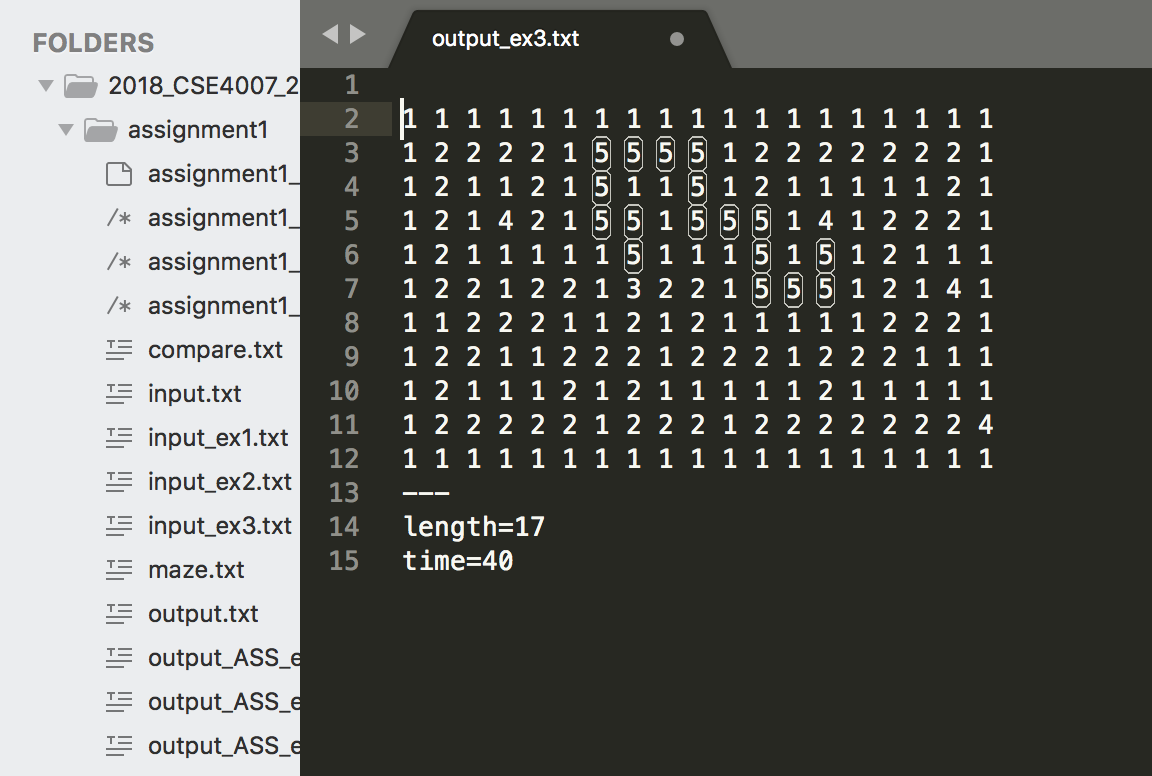
(3) ASS



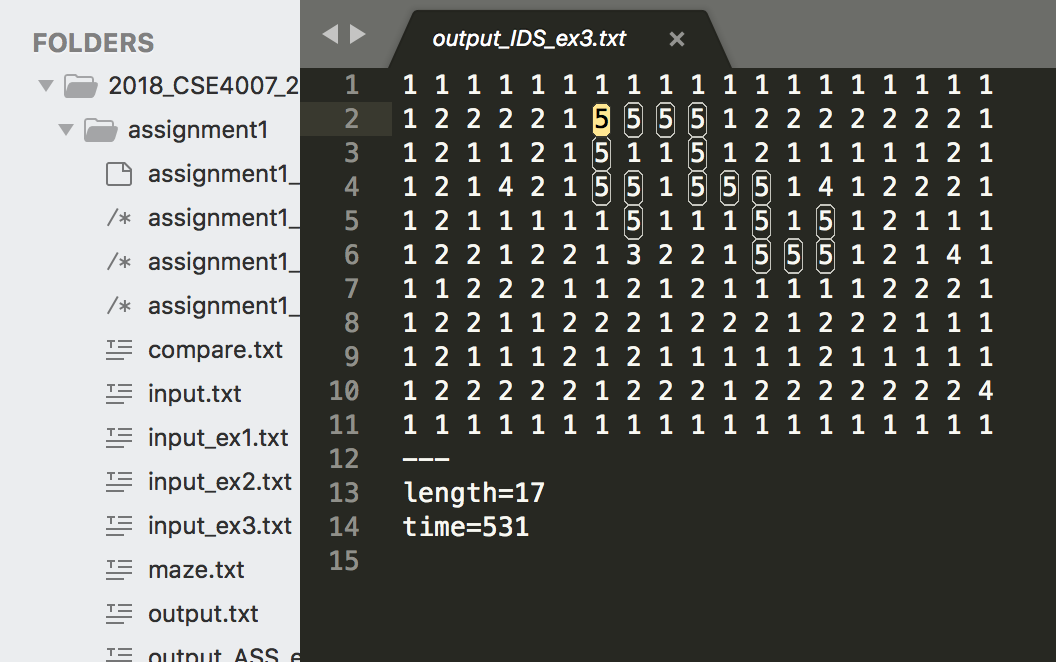
이 경우 IDS, ASS는 optimal한 solution을 찾으나 GBS는 optimal한 solution을 찾지 못하는 경우를 볼 수 있습니다. 하지만 전체 time은 여전히 GBS > ASS > IDS 순인 것을 확인할 수 있습니다.

**3) example3**

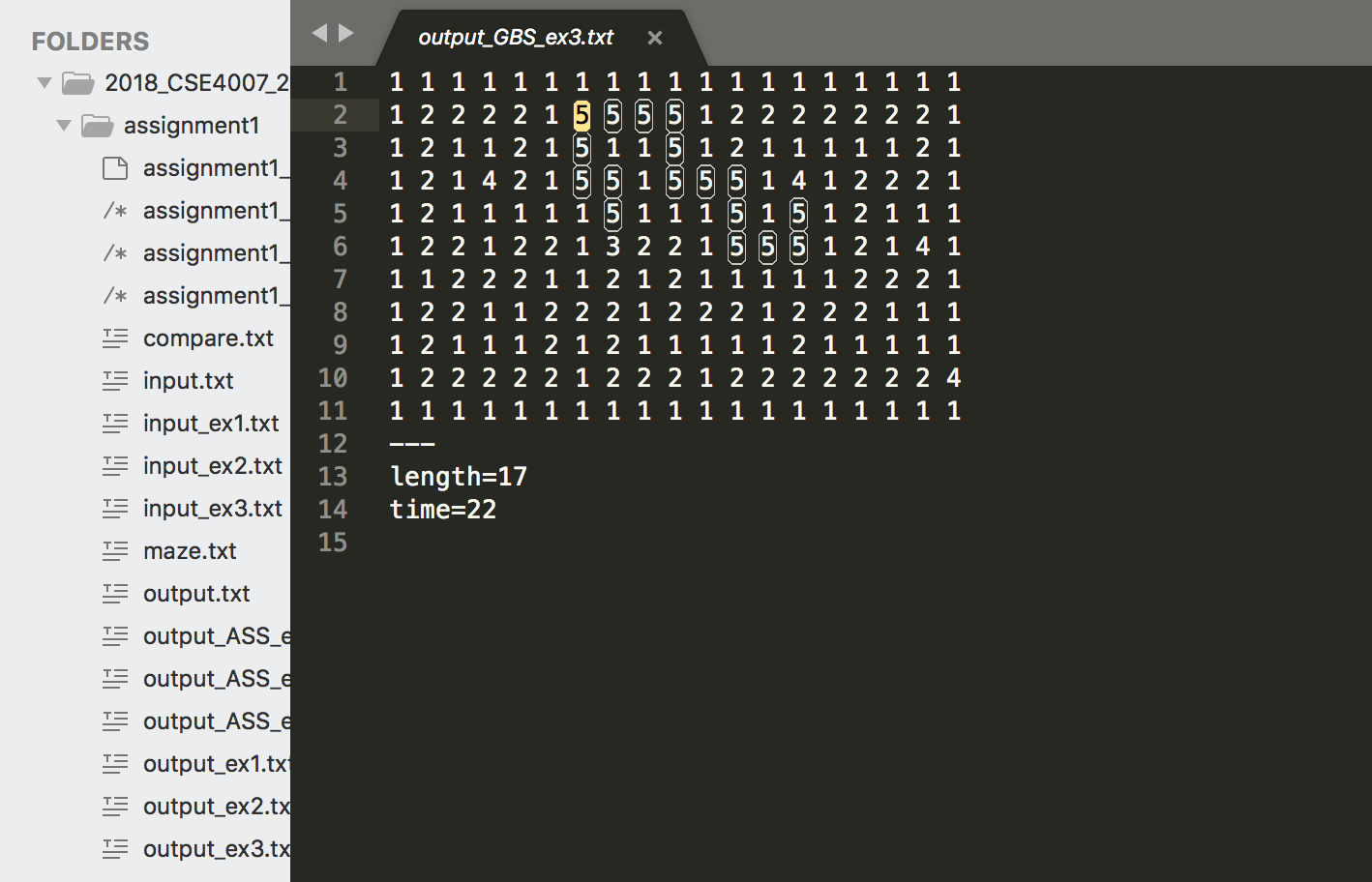
(0) Answer



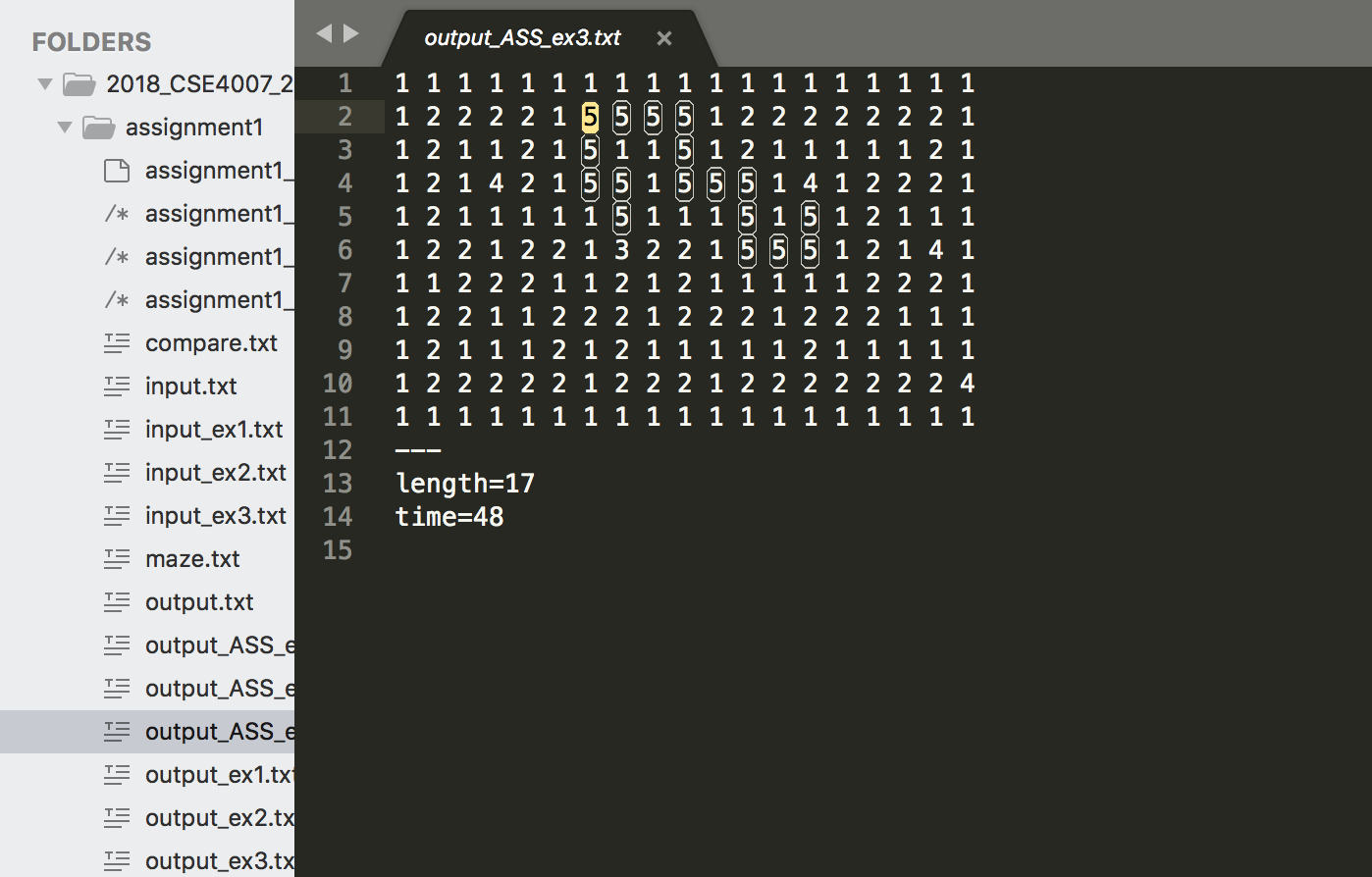
(1) IDS



(2) GBS



(3) ASS



이 경우 IDS, GBS, ASS 모두 optimal한 solution을 찾는 경우를 확인할 수 있습니다. 또한 총 시간은 여전히 GBS > ASS > IDS 순으로 빠른 것을 확인할 수 있습니다.

**4) analysis**

전체적으로 과제를 수행하면서 어떤 방향을 찾는 것에 우선순위를 두느냐에 따라 총 time에 차이가 생길 수 있다는 것을 확인할 수 있었습니다. 또한 특히 GBS에서는 이에 따라 optimal한 solution을 찾는지, 못찾는지가 결정될 수가 있다는 것도 확인할 수 있었습니다.

그리고 IDS, ASS는 항상 optimal한 solution을 찾고, GBS는 항상 현재 상황에서 빠른 길을 고르기 때문에 진행한 방향에 도착점이 없는 경우가 아니라면 가장 빠르게 길을 찾게 된다는 것을 알 수 있었습니다. 하지만 이는 optimal하는 보장이 없음도 알 수 있었습니다. 또한 ASS도 도착점과 빠른 지점을 찾기 때문에 전체 경로를 모두 검색하는 IDS보다 빠른 성능을 내면서 optimal한 solution을 찾는다는 것을 알 수 있었습니다. 그리고 IDS는 도착점에 대한 정보 없이, 모든 경로를 검색하여 optimal한 solution을 찾기에 전체 시간은 긴 경우가 대부분임을 확인할 수 있었습니다.