Elad Ohayon

The simple explanation of the solution is to use an array to keep track of mined paths (invalid location) and a custom object to represent the location of the "person". The algorithm then returns the minimum number of moves required to reach g, or 0 if it impossible. My algorithm's approach to computing the minimum number of steps needed to "reach" g, consists of a bfs on said array. The BFS is done on a queue of *Move* objects which store a position, and the number of moves need to reach the position. Beginning with a couple of observations, the minimum number of steps needed to reach **g** for any pair of g and m is the ceiling of g/m since each move forward is **m.** This means that if we can never do less moves than the ceiling of g/m. To find the most optimal solution, we use a bfs since we must explore all possible jumps as any choice now will affect our position in the future and may cause more moves than necessary. With these considerations in mind, we can begin a through explanation of the solution.

Upon initialization, the constructor creates an array of size MAX_FORWARD+1+m, the max length of the 1-dimensional world for the purposes of this algorithm. We also initialize a Boolean named unsolvable which determines whether the current setup is solvable; its semantics will be explained in addMinedLineSegment.

The method addMinedLineSegment enables the user to specify a range of [x,y], y≥x which the "pointer" cannot be located in. To reduce runtime of the algorithm, a Boolean variable unsolvable is used to determine instantly whether a given setup is unsolvable due to a certain condition. There are two positions for an added minefield, it must be placed in a range [X,Y] X≥G or [I,J] J>G. The pointer will encounter all minefields in the range [X,Y]. If |minefield| ≥ **m,** then the pointer will never be able to get across this minefield. In this case we set unsolvable equal to true; in turn, when unsolvable is set to true, any further calls to addMinedLineSegment instantly return and solveIt instantly returns 0, both of which are O(1) operations. On the other hand, if a minefield is valid, meaning it is between [0,**G+M+1**] and its length is smaller than m then we update the array representing the one dimensional world. We set every array index in the range of the minefield equal to -1. This also enables us to check with O(1) time if there is overlap, an illegal operation; The cost to add a valid minefield is O(length of minefield) for each minefield. Lastly, if the start of the minefield is greater than g+m+1 (to add some leeway :), the minefield is never added since the pointer will never encounter it, and the method simply returns. The cost is not being able to detect overlapping minefields. This is worth it as say for example, the user sets g=11, and adds a minefield with range (50,1,000,000), it would be a waste to set all those fields to -1.

Lastly, we look at the implementation of solveIt; as mentioned earlier, any move chosen now will affect future solutions so we must consider all possibilities through a BFS. The way the algorithm works is we instantiate a Move object where position, move=0 and add it to the queue. We also create a variable of type int which stores the min number of moves and set it equal to Integer_MAX_VALUE. In a while loop that runs until the queue is empty, we first remove the first element and then check if we can move it m spaces forwards and do so if possible, updating and enqueuing it afterwards. We then mark this position as visited and don't add any Move objects with the same position to prevent cycles. We don't check anything past m steps since a jump forwards will not lead to any progress, and we only add unmarked states to our queue to avoid an infinite loop. After each loop we check if a Move object's position has exceeded g and if so, we take the min of its moves and the current lowest amount of moves required to reach the final state. On the other hand, if there is a mine at Move.position +m then we add to the queue new Move objects with position-1 to position-(m-1) that haven't been explored yet (meaning not marked).We repeat the above processes, until the queue is empty and if moves is still Integer_Max_Value then that means there is no solution and we return 0.

The runtime of solveit depends on whether we must jump backwards or forwards. As such, the runtime varies from case to case, and we can only make an observation. Since we don't need to move beyond **g**, our max number of forward jumps without moving backwards is Math.ceil(O(g/m)). Every computation to move backwards is O(m). Furthermore, for every m units moved backwards, we need another jump forwards. Putting this together, our official estimate of solveIt is $\mathrm{O}(\ (\mathrm{Math.ceil}((\mathrm{G}/M)) + \mathrm{Math.floor}(H/M))*\mathrm{X})$, where **G** represents the location of the goal, **M:** the number of steps in each jump, **H** the number of units we moved backwards for this move object, and **X** being the number of Move Objects we added to the queue (1 or more). As everything except X is a constant which depends on G/m the average runtime is linear.