COMP3121 Assignment 3 – Q2

2.1

Given the 2D array, $B[1\ldots m][1\ldots n]$ representing the grid of a warehouse where $B[i][j]$ is either $TRUE$ or $FALSE$ determining whether a box exists in (i,j).

To determine whether a path in the warehouse is available from the initial location at cell $[1,1]$ to the exit at cell $[m,n]$, consider an algorithm which will use a dynamic programming approach to determine whether the warehouse meets this requirement.

To begin, consider the function $opt(i,j)$ which returns the number of possible paths to get from the given cell $[i,j]$ to the exit at the final cell $[m,n]$. Additionally, the algorithm will initialize an empty 2D array to store the values of $opt(i,j)$ into $DP[i][j]$.

The function $opt(i,j)$ also has the base cases that:

- $opt(m,n) = 1$, as long as $B[m][n] = FALSE$ as there is exactly one path from $[m,n]$ to $[m,n]$ when there is not a box.
- $opt(i,j) = 0$, if $i > m$ or $j > n$.

With a dynamic programming approach, the algorithm will start with a nested loop iteration starting from the exit $[m,n]$.

To begin the iterations, the algorithm will use a nested for loop. For each row value i from m to 1, loop through each column value j from n to 1. This nested loop will iterate through every cell $[i,j]$ starting from $[m,n]$ and ending at $[1,1]$.

Since there can be a box in each cell which will block the potential paths, the algorithm will consider that for each iteration, checking whether for each i and j if $B[i,j]$ is $TRUE$ or $FALSE$.

If $B[i,j]$ is $TRUE$, then the cell $[i,j]$ contains a box, meaning there are 0 valid paths from $[i,j]$ to the exit $[m,n]$. Thus, since there is no path at $[i,j], opt(i,j) = 0$. $DP[i][j] = 0$.

If $B[i,j]$ is $FALSE$, then the cell $[i,j]$ does not contain a box. Thus, the cell is not blocked and there is at least 1 path from the given cell. To calculate the number of paths available from the given cell is the sum of the number of paths from the cell below and to the right of the given cell $[i,j]$, these cells being $[i,j+1]$ and $[i+1,j]$. Thus, the calculation of $opt(i,j)$ is as follows:

$$opt(i,j) = opt(i,j+1) + opt(i+1,j).$$

Additionally, as the base case states, $opt(i,j)$ will equal 0 if $i > m$ or $j > n$ as it implies that the given cell is outside of the warehouse.

With this recursive relation, the algorithm will start at $[m,n]$ with the calculation of $opt(m,n) = 1$ as stated by the base case. The next iteration being $opt(m-1,n) = opt(i,j) = opt(i,j+1) + opt(i+1,j) = 0 + 1 = 1$ and so on. As such, $DP[i][j] = opt(i,j)$.

After iterating through each cell in the grid, the algorithm can compute the final cell $DP[1][1] = opt(1,1)$ which will determine how many valid paths are available from the starting cell $[1,1]$.

Based on the result of $DP[1][1]$ the algorithm will determine whether it is possible to reach the exit by only moving down or right one cell.

If $DP[1][1]$ is greater than 0, there is at least one possible way. If $DP[1][1]$ equals to 0, there is no valid path to reach the exit from the starting point.

The time complexity of any dynamic programming algorithm can be summarized as the time complexity of the subproblem multiplied by the number of subproblems. This algorithm starts off by initializing the 2D array DP which will at most run in $O(mn)$. Additionally, the nested for loop which has a time complexity of $O(nm)$ as the initial loop iterates through all m row values and during each iteration, loop through each n column value. The time complexity of each subproblem is $O(1)$ as the computation will be at most summing 2 values. As such, this algorithm will run in $(O(mn) * O(1) + O(nm))$ time which can be simplified to $O(nm)$.

2.2

Consider an algorithm which is identical to that of question 2.1. Some small adjustments can be made to determine the smallest number of boxes that must be removed to meet the requirements laid out in 2.1. The algorithm will initialize an empty 2D array to store the values of $opt(i,j)$ into $DP[i][j]$.

In this modified version of the 2.1 algorithm, $opt(i,j)$ will instead compute and $DP[i][j]$ will instead store the number of boxes $(nBoxes)$ of the path that goes through the minimum number of boxes from (i,j) to (m,n). That is, after considering every path from (i,j) to (m,n), the algorithm will choose the path with the smallest number of boxes encountered and store that value into $DP[i][j]$.

This algorithm will run very similarly to 2.1 where the recursion will begin at (m,n) and conclude at (1,1), returning $DP[1][1]$ as the final answer.

The algorithm will loop for each row value i from m to 1, loop through each column value j from n to 1. This nested loop will iterate through every cell $[i,j]$ starting from $[m,n]$ and ending at [1,1].

If $B[i][j]$ is TRUE, then the cell contains a box. The calculation of $opt(i,j) = \min\big(opt(i,j+1), opt(i+1,j)\big) + 1$.

If $B[i][j]$ is FASLE, then the cell does not contain a box. The calculation of $opt(i,j) = \min\big(opt(i,j+1), opt(i+1,j)\big)$

This recurrence relation will have the same base case that $opt(i,j)$ will equal 0 if $i > m$ or $j > n$ as it is out of the grid area. Thus, $opt(i,j)$ will be stored in $DP[i][j]$.

Additionally, in this modified version $opt(m,n)$ has a different base case that, $opt(m,n) = 0$ if $B[m][n] = FALSE$ **or** $opt(m,n) = 1$ if $B[m][n] = TRUE$.

This recursion of taking the minimum count of boxes encountered will allow $opt(1,1)$ to return the minimum number of boxes encountered after the algorithm considers every possible path from (1,1) to (m,n).

This algorithm will have a time complexity of $O(mn)$ as initializing the array and filling it up will run in $O(mn)$, the calculation of the number of boxes in each cell will run in $O(1)$ and the nested for loop used to calculate each cell will run in $O(mn)$. Thus, the overall time complexity is $(O(mn) * O(1)) + O(nm)$ time which can be simplified to $O(nm)$.

2.3

Any warehouse which passes the safety requirements from 2.1 has a path from (1,1) to (m,n) that includes a shortcut.

Any layout of the warehouse grid (without considering the box positions yet) will have at least one shortcut available. This is because of the specification that $m, n \geq 2$ which makes the minimum warehouse size a $2 * 2$ grid. The only grid layouts which will be impossible to have shortcuts are if either m or n are equal to 1.

A shortcut involves going diagonally to the cell located right and down from the current cell. Thus, without considering any boxes yet, any layout of the warehouse will include a shortcut.

Additionally, when considering that there may be boxes obstructing the path in the warehouse, as long as the warehouse layout passes the safety requirement in 2.1, there is always a shortcut from $(1,1)$ and (m, n). This is because if the warehouse passes the 2.1 requirement, there will always be at least one instance where the path includes an L shape path, either moving down and right, or right and down from the initial cell. This is where the shortcut can occur.

This is also a requirement of the 2.1 as a valid path must only be movements either going a cell down or right.

Thus, any warehouse which passes the safety requirements from 2.1 has a path from $(1,1)$ to (m, n) that includes a shortcut.

2.4
Consider an algorithm that runs in $O(nm)$ time and determines the minimum total hazard rating achievable using one shortcut.

To begin, consider the function $opt(i, j)$ which will calculate the minimum total hazard rating the path from $(1,1)$ to (i, j) which includes one shortcut.

The algorithm will initialise a 2D array DP, which will store the values $opt(i, j)$ in $DP[i][j]$.

As base case, $DP[1][1] = H[1][1]$ as there is only one rating for the path at the first cell.

The remaining base cases include, $DP[1][j] = DP[1][j-1] + H[1][j]$ as the minimum hazard rating for a path traversing to the right is the sum of the hazards in the first row.

And $DP[i][1] = DP[i-1][j] + H[i][1]$ as the minimum hazard rating for a path traversing only downwards is the sum of the hazards in the first column.

The recursive relation in this algorithm is that $DP[i][j]$ is calculated by finding sum of $H[i][j]$ and the minimum value of either $DP[i-1][j]$ or $DP[i][j-1]$. This computation will take the minimum hazard rating for the given path from above and from the left, adding the hazard rating at cell (i, j). This is so the algorithm can find the path with the minimum hazard rating for each path when extending it each cell.

In equation form, this can be expressed as:

$$DP[i][j] = H[i][j] + \min(DP[i-1][j], DP[i][j-1]).$$

The algorithm will use a nested for loop to traverse through the entire grid (2D array) to apply the given formula to every combination of i and j. Once the entire array DP is filled with the values of the subproblems, the final minimum hazard rating for the path with exactly one shortcut can be determined, being $DP[m][n]$. Thus, the algorithm will return $DP[m][n]$ as the final answer.

The time complexity of this algorithm is dominated by the nested for loop used to calculate the values to be stored in array DP. This has a run time of $O(mn)$ as the maximum width and height of the 2D array is $m * n$ dimensionally and thus the overall algorithm has a time complexity of $O(mn)$.