Kick Start 2020 - Round G

Analysis: Maximum Coins

Terminology: In what follows, when we refer to a *diagonal* starting from (x, y), we mean all cells (p, q) such that x - y = p - q. Only cells satisfying these conditions are considered, because Mike can only move diagonally.

Test Set 1

Mike is allowed to go from cell (i,j) to cell (i+1,j+1) or to cell (i-1,j-1). We can consider each possible starting point and try calculating the value of each possible path Mike can traverse. Initialize the answer as 0. For each starting cell (i,j), Mike can either go diagonally upwards or diagonally downwards. First consider all the paths which start at cell (i,j) and end diagonally above it. We keep on adding the value of coins by traversing upwards diagonally and updating the answer. Similarly, we consider all paths which start at cell (i,j) and end diagonally below it and update the answer. Each path can contain at most O(**N**) elements. Thus, it takes O(**N**) time for each starting position. There can be O($\mathbf{N}^2$) starting positions. Thus, the overall complexity of the solution is O($\mathbf{N}^3$).

Sample Code(C++)

```
int GetMaximumCoins(const vector< vector< int > >& coins) {
  int answer = 0;
  for(int i = 0; i < coins.size(); i++) {
    for(int j = 0; j < coins[0].size(); j++) {
    int currx = i, curry = j, currval = 0;
    // Go above.
    while(currx >= 0 and curry >= 0) {
      currval += coins[currx][curry];
      answer = max(answer, currval);
      currx--;
      curry--;
    }
    currx = i;
    curry = j;
    currval = 0;
    // Go below.
    while(currx < coins.size() and curry < coins[0].size()) {
      currval += coins[currx][curry];
      answer = max(answer, currval);
      currx++;
      curry++;
    }
   }
  }
  return answer;
}
```

Test Set 2

An important observation here is that all the numbers are positive. Thus, it is always optimal to collect the coins for each cell present on a particular diagonal instead of choosing some of the cells on the diagonal. This can be done by starting on the top left of each diagonal and traversing down and adding the coins collected. For each diagonal it would take O(**N**) time to calculate the coins present in that diagonal. There are O(**N**) diagonals present in the matrix. Thus, the overall time complexity of the solution is O($\mathbf{N}^2$).

Sample Code(C++)

```
long long int GetDiagonalSum(const vector< vector< int > >& coins, int i, int j) {
   long long int currval = 0;
   int currx = i, curry = j;
   while(currx < coins.size() 0 && curry < coins[0].size()) {
     currval += coins[currx][curry];
     currx++;
     curry++;
   }
  return currval;
}

long long int GetMaximumCoins(const vector< vector< int > >& coins) {
  long long int answer = 0;
  // Top row.
  for(int i = 0; i < coins[0].size(); i++)
    answer = max(answer, GetDiagonalSum(coins, 0, i));
  // Left column.
  for(int i = 0; i < coins.size(); i++)
    answer = max(answer, GetDiagonalSum(coins, i, 0));
  return answer;
}
```