SID: 1323782
SName: Taylor Tang

1. Longest Skiing Path

(a) Pseudocode:

```
Note: M [0..n −1][0..n −1] showing as matrix

function LongestSkiingPath(matrix)

    if matrix.length == 0 do                            # Sanity check
       return 0

    create number m = matrix.length
    create number n = matrix[0].length

    create array of array cache [m][n]                  # create a memoization matrix with same dimension
                                                           as M
    create number ans = 0

    for i <-- 0 to m do
        for j <-- 0 to n do
            ans = max(ans, dfs(matrix, i, j, cache))# iterating through each cell with dfs()

    return ans;

function dfs(matrix i, j, cache)

    if cache[i][j] != 0 then                            # check cache for value first
        return cache[i][j]                              # if exist, no need to calculate again

    create array dirs = {{0, 1}, {0, -1}, {1, 0}, {-1, 0}}

    for dir in dirs do                                  # check all four directions
        x = i + dir[0]
        y = j + dir[1]
        if 0 <= x && x < m && 0 <= y && y < n && matrix[x][y] > matrix [i][j] then
            cache[i][j] = max(cache[i][j], dfs(matrix, x, y, cache))
                                                        # within matrix boundaries, recursively calling dfs()
                                                        # to find the largest cache value
    return cache[i][j] + 1                              # add 1 to the largest cache value found in dirs
```

(b) Time complexity of this algorithm is $O(n^2)$.

For this algorithm, we first iterate every cell (that is n x n = n^2 cells) and run the depth-first search function (dfs() for short below).

A standard dfs() time complexity is $O(n^2)$ in this case. To improve the time complexity, a cache for memoization has been created to store the value, so the value can be subsequently retrieved without repeating the computation. Therefore, for all the cells that has been searched by the dfs(), the next time we need to retrieve the value, the time complexity reduced to $O(1)$.

During the iteration of all cells, we first check the cache, if not exist then we use dfs(). Each cell will be calculated once and only once. Hence, the time complexity of the dfs() part with cache is $O(1)$. The overall time complexity for this algorithm is $O(n^2) * O(1) = O(n^2)$

2. Combination of molecules

To analysis question, first we need to make a few assumptions:

I. One can't make sure of a result unless testing it with robot. As we don't have a rigid conclusion or observations that giving two molecules A & B, if the combination AB will be more effective that individual A or individual B.
II. We expect more molecules combination to work as good or better than less molecules combination, such as A + B > A or B. **However** there is a chance molecules combination could performance worse than molecules combination.
III. Giving molecules and efficacy A > B > C, we expect that A + B > A + C. **However** there is a chance that A + B < A + C.

(a) Based on above assumption, algorithm and pseudocode are proposed below:

1) For $O(n)$ time complexity:

- Step 1, test all molecules for individual efficacy, sort them in descending order, call it candidates.
- Step 2, take the highest efficacy molecule add to curr_stack, its efficacy is the curr_eff.
- Step 3, take the next highest molecule add it to curr_stack, test the effectiveness of the combination, get the result after_eff.
- Step 4, compare curr_eff with after_eff, if curr_eff < after_eff, meaning the chosen molecules are a good combination, we keep both.
  If curr_eff > after_eff, meaning they are not cope with each other well, we discard the recent added molecule from both currSet.
- Step 5, repeat step 3 & 4 until iterated all molecules, return the highest efficacy and molecule combination.
- Note: Although in step 1 we use sorting algorithm that is exceeding O(n) time complexity, we only call the most concerning operation test_wet_lab_robot() n times. Hence, the overall time complexity is not exceeding O(n).

```
function BestEfficacyCombinationN(mols)

    create dictionary mol_eff
    for i <-- 0 to mols.length do                      # test each mol, record its efficacy
        mol_eff.update(mols[i], test_wet_lab_robot(mols[i]))

    sort mol_eff by values                             # sorting detail omitted, refer to note*

    create linked list mols_sorted
    for key in mol_eff do
        mols_sorted.add(key)                           # add all molecules in list by sorted order

    create stack curr_stack
    curr_stack.push(mols_sorted[0])                    # add the highest efficacy mol to currSet
    mols_sorted.remove(mols_sorted[0])                 # remove it from candidate molecules
    create number curr_eff = mol_eff.get(mol_eff[0])   # get the efficacy from the added mol

    for i <-- 0 to mols_sorted.length do

        curr_stack.push(mols_sorted[i])                # add a new molecule and test efficacy
        create after_eff = test_wet_lab_robot(curr_stack)

        if curr_eff <= after_eff then                  # if result good, keep and update
            curr_eff = after_eff
        else                                           # if not, discard the recent added molecule
            curr_stack.pop()

    return {curr_stack, curr_eff}                      # return the best combination and efficacy
                                                       (in dictionary structure)
```

(Note: sorting algorithm omitted, assuming using quick sort or merge sort to achieve efficiency O(nlog(n)). Detailed implementation omitted; details can differ based on coding languages. For example, in python, this can be achieved by using sorted(mol_eff.items(), key=lambda x: x[1]), and mols_dict.reverse() for descending order.)*

2) For $O(n^2)$ time complexity:

- Step 1, test all molecules for individual efficacy, sort them in order.
- Step 2, take the highest efficacy molecule add to curr_stack, its efficacy is the curr_eff.
- Step 3, test the combination of curr_stack with each of candidate molecules, pick the combination with highest efficacy as the curr_stack. Remove the added molecules from candidate molecules.
- Step 4, repeat step 3 until no better candidate molecules can be found.
- Note: Step 3 is O(n) time complexity, it is nested in step 4 which is also O(n) time complexity. Hence, the overall time complexity is O(n^2).

```
function BestEfficacyCombinationN2(mols)

    create dictionary mol_eff
    for i <-- 0 to mols.length do                      # test each mol, record its efficacy
        mol_eff.update(mols[i], test_wet_lab_robot(mols[i])

    sort mol_eff by values                             # sorting detail omitted, refer to note*

    create linked list mols_sorted
    for key in mol_eff do
        mols_sorted.add(key)                           # add all molecules in list by sorted order

    create stack curr_stack
    curr_stack.push(mols_sorted[0])                    # add the highest efficacy mol to currSet
    mols_sorted.remove(mols_sorted[0])                 # remove it from candidate molecules
    create number curr_eff = mol_eff.get(mol_eff[0])   # get the efficacy from the added mol

    create boolean need_more = true
    while need_more do
        create mol curr_best = null
        create boolean found_better = false

        for i <-- 0 to mols_sorted.length do           # find best matching molecule with currSet

            after_eff = test_wet_lab_robot(curr_stack.push(mols_sorted[i]))
                                                       # test new combination
            if curr_eff <= after_eff then              # if combination efficacy higher
                curr_eff = after_eff                   # update current best efficacy
                curr_best = mols_sorted[i]             # update current best matching molecule
                found_better = true
                curr_stack.pop()                       # remove the recent added mole for next
                                                         testing
            else
                curr_stack.pop()

        if found_better then
            curr_stack.push(curr_best)                 # add best matching molecule
            mols_sorted.remove(curr_best)              # and remove it from candidate set

        else
            need_more = false                          # break point from them loop

    return {currSet, currEff}                          # return both currSet & currEff (in dictionary
                                                       structure)
```

Greedy algorithms analyse:

The greedy part of these two algorithms is that at each iteration, we always decide whether to add more molecules (or keep looking for better molecules) based on current situation that is if it can cope well with the molecules set we already have. There is a chance that one molecule might not cope well with current molecules set but will cope well future adding molecules. However, in a greedy algorithm we only concern with locally best choice, not globally optimal choice.

(b) The above algorithms do not always find the optimal solution. The nature of greedy algorithms is that when we are making a decision, we only concern with locally best choice, not globally optimal choice. Hence, there is always possible that the yield result is not the global optimal result.

When we decide of adding or discarding molecule x to our current set, we only considered the effect of x with the current set and exclude the consideration of x with potential future molecules. For example, we have molecule [A] and just about to test molecules [A, B]. After testing we find that [A, B] doesn't make a good combination so we discard [B]. However, [B, C] might be a very strong efficacy combination but since we don't test it, we will never know. Hence, the above algorithms do not provide optimal solution.

Although the $O(n^2)$ can provide better result than $O(n)$ one in term of combination efficacy, as it tests more combination.

To solve this problem optimally, we will need to literately check all possible combinations, that is given n molecules, there is nC1 + nC2 + … + nCn possible combinations (2^n – 1 after simplify). This problem is like travel salesman problem but even more difficult as we do not have preview to the cost (efficacy) of each city (molecule).

However, it would be too costly too check all combination. Hence, we could use greedy algorithm to achieve a relatively high efficacy result with much less cost.

3. Stickify

```
function BSTInsert(root, new)
    if new.value < root.value then
        if root.left = NULL then
            root.left ←new
            Stickify(new, root)
        else
            BSTInsert(root.left, new)
    if new.value > root.value then
        if root.right = NULL then
            root.right ←new
            Stickify(new, root)
        else
            BSTInsert(root.right, new)

function Stickify(new, root)
    if new.value < root.value then
        RotateRight(root)
```