Project 2 Report
Dynamic Programming

Trevor Ammons,
Alexander Cook,
Thiago Lopes,
Anthony Peer

1. This problem can be approached by considering a 2D array between each set of teeth n and m. Each cell of the array is the sum of n_i and m_i.

The recursive function will be called on the two original arrays. The first call is represented at the 1-1 cell. From there the function makes 3 recursive calls shown by the arrows below. At each recursive level n and m are reduced in size until it reaches the end of both arrays (In this case [6],[2], highlighted below).

	2	7	4	5	3	1	4	6
6	8	13	10	11	9	7	10	12
4	6	11	8	9	7	5	8	10
1	3	8	5	6	4	2	5	7
3	5	10	7	8	6	4	7	9
5	7	12	9	10	8	6	9	11
4	6	11	8	9	7	5	8	10
7	9	14	11	12	10	8	11	13
2	4	9	6	7	5	3	6	8

(Question 1 Cont.) This chart shows one possible path out of many that the recursive calls will take. At each cell the height is updated to the current lowest that has been found in that path.

	2	7	4	5	3	1	4	6
6	.8	13	10	11	9	7	10	12
4	8	11	8	9	7	5	8	10
1	3	8	5	6	4	2	5	7
3	5	10	8	8	6	4	7	9
5	7	12	9	10	8	6	9	11
4	6	11	8	9	7	8	8	10
7	9	14	11	12	10	8	11	13
2	4	9	6	7	5	3	8	.8

After all paths are completed, the 1-1 cell will contain the minimum possible height for the sets of teeth. The path taken represents which teeth need to be extended to obtain that height. For example, starting from 1-1, moving down a cell extends 2, while extending horizontally would extend 6.

2. The base cases of this recurrence are the outer edges of the array. The recursive function checks the indices i, and j to not continue recursive calls out of bounds.

	2	7	4	5	3	1	4	6
6	8	13	10	11	9	7	10	12
4	6	11	8	9	7	5	8	10
1	3	8	5	6	4	2	5	7
3	5	10	7	8	6	4	7	9
5	7	12	9	10	8	6	9	11
4	6	11	8	9	7	5	8	10
7	9	14	11	12	10	8	11	13
2	4	9	6	7	5	3	6	8

3. The data structure suggested for this problem is a 2D array of size $(n \times n)$ where n is the length of the largest array between the two initial arrays from input.txt. Each index of the array (x, y) is initialized to the sum of Array1[x] + Array2[y]. The implementation of this data structure is a 2D vector.

4. The pseudocode below recursively traverses and alters the 2D array, which is populated with the minimum value possible that index could be, when referencing the sum of its associated teeth heights, compared the sum of the associated teeth heights if one array was to be extended.

Input: Reference to 2D Array, *input*, initialized in function which takes in values from an input file.
Input: Reference to a 2D array, *memVec*, which stores data for memoization, initialized to -1.
Input: Two integers, *i* and *j*, which represent the current location in the 2D array, *input*.

```
AlignTeeth(Reference to 2D Array input, int i, int j, Reference to 2D
Array memVec)
    if memVec[i][j] != -1 (Check for found value to skip recursive call)
        Input[i][j] = memVec[i][j]
    else if i = input.length - 1 and j = input.height - 1
        memVec[i][j] = input[i][j]
    else if i+1 > input.length -1
        Input[i][j] = max(input[i][j], AlignTeeth(input, i, j+1, memVec))
        memVec[i][j] = input[i][j]
    else if j+1 > input.length - 1
        Input[i][j] = max(input[i][j], AlignTeeth(input, i+1, j, memVec))
        memVec[i][j] = input[i][j]
    else
        Int o1 = max(input[i][j], AlignTeeth(input, i+1, j+1, memVec))
        Int o2 = max(input[i][j], AlignTeeth, i, j+1, memVec))
        Int o3 = max(input[i][j], AlignTeeth, i+1, j, memVec))
        Input[i][j] = min(o1, o2, o3)
        memVec[i][j] = input[i][j]

return input[i][j]
```

5. The worst-case time complexity of the memoized algorithm is $O(n^2)$ because there are n by n function calls, one for each cell of the 2d array. There is a constant time operation for each function call.

6. The pseudocode below is used to fill a 2D array with values that are used by a separate function, FindPath. The FindPath function (pseudocode found in question 8) uses this 2D array to iteratively track the best path (path being the least number of teeth extended while maintaining the minimum height possible) from the last index (Array.length, Array.height) to the first index (0,0).

(Cont. Next page)

Input: Two integers, *xLen* and *yLen*, which represent the lengths of input vectors
Input: Two vectors, x and y, representing height of teeth that need to be aligned
Output: A 2D Array, *input*, which is populated with the minimum value possible that index could be, when referencing the sum of its associated teeth heights, compared the sum of the associated teeth heights if one array was to be extended.

```
Iterative-Align-Teeth(int xLen, int yLen, vector<int> x, vector<int> y)
Arr1 = Array(xLen)
Arr2 = Array(yLen)

If (Arr1.size != Arr2.size)
        Extend shorter array by its minimum value until Arr1.size =
Arr2.size

Input = 2D Array (Arr1.size + 1 by Arr2.size + 1)
Initialize all values of 2Darr to -1

for i = xLen down to 1 do
        for j = yLen down to 1 do
                int currSize = Arr1[i] + Arr2[j]
                int sum1 = max(currSize, 2Darr[i+1, j+!1])
int sum2 = max(currSize, 2Darr[i+1, j])
                int sum3 = max(currSize, 2Darr[i, j+1])
                input[i, j] = min(sum1, sum2, sum3)
        end
end

return input
```

7. Using the iterative solution, we can improve the space complexity by only storing two columns of the 2D array. This is because a column's values are only dependent on the values of the column to the right. This would improve the space complexity from $O(n^2)$ to $O(2n)$. The code above would also need to be edited to allow for the path to be tracked within the for loops which iterate through the 2D array. All that would need to be added is a check within the loop searching for the last point in the path, and if that point is found, check the surrounding required indices for the next point. If you check the index, $i-1, j-1$ (cell diagonally to the south east), all required indices to check for the next point in the path will already be found.

8. The pseudocode below represents the algorithm which takes in the 2D array created from either the memoized, or iterative algorithms above (questions 4 and 6) and returns the optimal height for the teeth as well as both extended arrays which produce the optimal height.

Input: 2D Array, *input*, which is populated with the minimum value possible that index could be, when referencing the sum of its associated teeth heights, compared the sum of the associated teeth heights if one array was to be extended.
Input: Two integers, i and j, representing the index of the 2D array, *input*.
Input: Two references to vectors, *sol1* and *sol2*, which represent the output vectors after the input vectors have been extended.
Input: Two maps, m and n, which allow for the loop up of the teeth height associated with a specific index in the input Arrays.
Output: Original input array. The function also alters the arrays *sol1*, and *sol2*, which are passed into another function to be exported to an output file.

(Cont. next page)

```
FindPath(2D Array input, int i, int j, int max, Reference to Array sol1,
Reference to Array sol2, map if ints m, map of ints n )

if i = input.length - 1 OR  j = input.height - 1 (if the point is on the
bottom or right edge of the 2D array)
        if i = input.length - 1 AND j = input.height - 1 (Most south east
corner of 2D array)
                Add m[i] to sol1
                Add n[j] to sol2
                return input (End of 2D Array)
        else if i = input.length - 1 (East most column)
                if input[i][j + 1] <= max
                        Add m[i] to sol1
                        Add n[j] to sol2
                        return FindPath(input, i, j+1, max, sol1, sol2, m, n)
        else if j = input.height - 1 (South most row)
                if input[i+1][j] <= max
                        Add m[i] to sol1
                        Add n[j] to sol2
                        return FindPath(input, i+1, j, max, sol1, sol2, m, n)

if input[i + 1][j + 1] <= max
        Add m[i] to sol1
        Add n[j] to sol2
        return FindPath(input, i+1, j+1, max, sol1, sol2, m, n)
else if input[i + 1][j] <= max
        Add m[i] to sol1
        Add n[j] to sol2
        return FindPath(input, i+1, j, max, sol1, sol2, m, n)
else if input[i][j + 1] <= max
        Add m[i] to sol1
        Add n[j] to sol2
        return FindPath(input, i, j+1, max, sol1, sol2, m, n)

return input
```