COMP 352: Data Structures & Algorithms

Assignment 2: Part 2

Fall 2022

Submitted by: Karina Sanchez-Duran (40189860)
and Sofia Valiante (40191897)

Part 2: Pseudocodes

1) Recursive algorithm for StrikeEdgeZero

Algorithm recStrikeEdgeZero (initial, size, A, seen)
Input: An integer array A of size *size*, an integer *initial* which indicates the starting index of the marker, and an Array List seen which contains a list of indexes already visited.
Output: Returns 1 if you win the game (if a solution exists). Otherwise, returns 0.

```
$newInitial \leftarrow 0$
$goRight \leftarrow initial + A[initial]$
$goLeft \leftarrow initial - A[initial]$

if $A[initial] = 0$ then
    return 1        //you win
if $(goRight >= size)$ && $(goLeft < 0)$ then
    return 0        //you lose
if $seen.contains(initial)$ then
    return 0        //you lose

$seen.add(initial)$

if $goRight < size$ then
    $newInitial \leftarrow goRight$
    if $A[goRight] = A[initial]$ && $goRight + A[goRight] >= size$
     && $goLeft > 0$ then
        $newInitial \leftarrow goLeft$
    return $recStrikeEdgeZero(newInitial, size, A, seen)$
else
    $newInitial \leftarrow goLeft$
    return $recStrikeEdgeZero(newInitial, size, A, seen)$
```

2) Stack algorithm for StrikeEdgeZero

Algorithm stackStrikeEdgeZero(gameStack, tempStack)

Input: A stack object gameStack that will be the stack played during the game, a stack object tempStack that will be used to store the values of the gameStack pops.

Output: Returns 1 if you win the game (if a solution exists). Otherwise, returns 0.

```
while gameStack.top() != 0 do
    top ← gameStack.top()
    symmetricalTop ← gameStack.top()

    if gameStack.size() - 1 then
        if gameStack.top() = symmetricalTop ∧ top != 1 ∧ top =
        gameStack.size()-2 then
            top ← gameStack.top()
            if top ≤ tempStack.size() then
                for j < top do
                    gameStack.push(tempStack.pop())
                    j ← j+1
            return 0
        for j < top do
            tempStack.push(gameStack.pop())
            j ← j+1
    else if top ≥ gameStack.size() ∧ top ≤ tempStack.size() then
        for j < top do
            gameStack.push(tempStack.pop())
            j ← j+1
    else
        return 0
return 1
```

<u>Part 2: Questions:</u>

a)

i) The time complexity for *recStrikeEdgeZero* is $O(n)$ because it uses linear recursion which means it only makes one recursive call each time the function is called. The array takes up the most memory space and each element in the array takes 4 bytes * N space where N is the size of the array. Thus, the space complexity for *recStrikeEdgeZero* is $O(n)$.

ii) The time complexity for *recStrikeEdgeZero* is $O(n^2)$ because it employs two nested loops, one being the main while loop that controls when the game ends, and the for loops inside that pop the top amount of elements in the stacks. Though there are many for loops within the algorithm, they are parallel to each other, and therefore also yield a time complexity of $O(n^2)$, and not $O(n^3)$ or higher. The space complexity is O(n) as both stacks, gameStack and tempStack, are implemented with an array of size N and N elements. Given that these array are not nested and are simply parallel to each other, they are not $O(n^2)$. A more accurate space complexity would be $O(2n)$, but dropping the constant leaves $O(n)$

b) The recursive implementation (i.e: version A) uses linear recursion. More specifically, it uses tail recursion because the recursive call is the last statement every time the program is executed.

c) A stack was chosen as the second part of the solution (version B) because it was the data structure that aligned itself more with the logic of the game. The only value that is being kept track of during the game is the value landed on, which equates to the top of the stack. Thus, you would pop the stack as many times as the value you had previously landed on it. To know whether to go left or right, depending on the direction you had previously gone, going left simply equates to popping the stack and going right equates to pushing the stack.

d)

i) There are essentially two ways one can detect an unsolvable array configuration. If one cannot go to the right nor to the left without violating the boundaries of the array, then the configuration is unsolvable. If one has already visited the index at which the marker is currently held, then one is going in circles (going on an infinite loop) and the configuration is unsolvable. One can check these possibilities at the beginning of the function in order to speed up execution time.

ii) An unsolvable array can be detected by determining if the currentTop of the stack is equal to the top of the stack if you were to pop it currentTop number of times, in addition to there not being enough values left for the currentTop to instead move in the opposite direction. Determining this condition, allows for the program to break the game's loop and output that a solution will never exist, thus speeding up execution time.