COMP 352 Fall Semester 2017

Assignment 2

Rui Zhao

40018813

Programming Questions:

<u>Pseudo code (Recursive version):</u>

Algorithm recursiveRightMagnetic (A, x)

Input: an array A of integers, an start marker x

Output: Boolean value that indicates if the right magnetic game is successful or not(true for success)

```
for i ←0 to A.length-1 do
    tempArray[i] ← A[i]                    {create a temporary array to store elements of A}
end for
if x> tempArray.length-1 then
   return false                            {given start maker is out of bound}
else if x= tempArray.length-1 then
    return true                            {success}
else
    temp ← ceiling(tempArray[x]/2)
    if (x+temp) <= A.length and x!=x+temp then          {move to right}
        tempArray[x]  ← 0                  {if not visited, change the element to 0 then recursion}
        return recursiveRightMagnetic (tempArray, x+temp)
```

```
    else if (x-temp)>0 and x!= x-temp then                    {move backwards}
        tempArray[x]  ← 0          {if not visited, change the element to 0 then recursion}
        return recursiveRightMagnetic (tempArray, x-temp)
    else return false
    end if
end if
```

Pseudo code (Stack version):

Algorithm stackRightMagnetic (A, x)

Input: an array A of integers, an start marker x

Output: Boolean value that indicates if the right magnetic game is successful or not (true for success)

```
if x> tempArray.length-1 then
    return false                              {given start maker is out of bound}
end if
```

Stack s1: holds the visited indices

```
s1.push(x)                    {move to the start maker}
while x!=A.length-1           {while not solved}
    if s1.size>a.length then    {exceed the maximum moves for a successful game}
        return false
    end if
    temp ← ceiling(A[s1.top]/2)
    rightIndex ← x+temp      {potential index to the right}
```

```
        leftIndex ← x-temp        {potential index to the left}
        if rightIndex < A.length then   {check if right index is in bound}
            s1.push(rightIndex)       {move to right}
            x ← rightIndex            {now the marker is at the new position}
        else if leftIndex>0 then      {when it's impossible to move to right and left index is in bound
            s1.push(leftIndex) {move to left}
            x ← leftIndex  {now the marker is at the new position}
        else return false {out of bound both for left and right}
        end if
end while
return true  {array is solvable}
```

a) For the recursive version, the time and space complexity are both $O(n^2)$. Because in each recursion, we create an array with length n, in total it's 1+2+…+n, which is $O(n^2)$. For the stack version, the time complexity is O(n), because for a successful game, the slowest process is to visit every index once, that's n in total. After that, there will be repeated visit, which means the array is unsolvable. The space complexity is O(n), because we created one stacks and one new variables, in total is n+1, which is O(n).

b) The recursive version is in the form of linear recursion (however not tail recursion), because the function makes only one recursive call in its body and wait for the result returned by this recursive call. And for recursive version, the reason why time and space complexity is $O(n^2)$ is not because it's a linear recursion, instead, it's because the creation of the array.

c) For the algorithm that I use, there's no difference of using stack or queue or list or vector. To determine if the given array is solvable, I let the process go to right when it's possible, and go to left when the potential right move is out of bound. And store the visited indices in s1. Since for a successful game, the slowest process is to visit every index once, that's n in total. After that, there will be repeated visit, which means the array is unsolvable. In all these ADTs, the complexity is the same for add a new element at the end of the ADT (O(1)) and for querying the size of the ADT (for good implementation). Therefore, adding n of them will all require O(n).

d) **<u>Recursive version:</u>**

```
The right magnetic result for array
8 16 10 4 6 10 2 12 8 0
with size 10 and start marker 0 is true

The right magnetic result for array
1 10 16 4 6 2 10 0
with size 8 and start marker 0 is false

The right magnetic result for array
11 8 3 2 1 6 10 0
with size 8 and start marker 0 is false

The right magnetic result for array
8 16 10 4 6 10 2 12 8 11 8 3 2 1 6 10 0
with size 17 and start marker 0 is false

The right magnetic result for array
2 5 1 5 7 4 6 4 8 9 14 27 0
with size 13 and start marker 0 is true

The right magnetic result for array
20 6 38 17 41 22 49 41
with size 8 and start marker 5 is false

The right magnetic result for array
14 18 36 25 36 37 48 33 28 6 49 28 41
with size 13 and start marker 7 is false

The right magnetic result for array
38 16 10 45 10 2 33 4 37 35 30
with size 11 and start marker 4 is false

The right magnetic result for array
```

```
9 31 9 47 18 50
with size 6 and start marker 0 is true

The right magnetic result for array
28 20 15 49 38 37 34 6 31
with size 9 and start marker 6 is false

The right magnetic result for array
15 8 25 21 11 26 38 36 23 46 17
with size 11 and start marker 7 is false

The right magnetic result for array
13 15 2
with size 3 and start marker 1 is false

The right magnetic result for array
22 25 5 15 24 33 14 28 49 30 4 50 36 2 13 36 29 43 47
with size 19 and start marker 8 is false

The right magnetic result for array
26 40 31 50 4 49 13 36 20 32 10
with size 11 and start marker 5 is false

The right magnetic result for array
31 1 17 50 43 1 34 28 1 48 49 20 50 48 40 18
with size 16 and start marker 7 is false

The right magnetic result for array
46 30 20 32 49 26 42 47 18 26 41 12 49 14 16 47 17 26 23 14 48 22 39 41 26 18 32 27 12 8 17
18 29 48 16 31 27 6 19 3 27 32 19 13 3 19 27 40
with size 48 and start marker 20 is true

The right magnetic result for array
16 13 31 34 28 34 43 21 29 33 5 23 18 49 13 1 32 16 7 42 21 28 9 37 16 42 13 13 21 36 49 29
19
with size 33 and start marker 28 is true

The right magnetic result for array
46 31 8 27 2 33 42 23 7 33 17 26 46 31
with size 14 and start marker 13 is true

The right magnetic result for array
44 15 3 34 41 36 37 33 7 49 41 29 40 38 13 37 15 13 46 36 8 21 12 1 12 4 20 10
with size 28 and start marker 2 is true

The right magnetic result for array
23 18 9 34 32 8 46 43 26 35
with size 10 and start marker 5 is true

The right magnetic result for array
…
with size 43892 and start marker 0 is false (output is too long, only paste the result)
```

Stack version:

The right magnetic result for array
8 16 10 4 6 10 2 12 8 0
with size 10 and start marker 0 is true

The right magnetic result for array
1 10 16 4 6 2 10 0
with size 8 and start marker 0 is false

The right magnetic result for array
11 8 3 2 1 6 10 0
with size 8 and start marker 0 is false

The right magnetic result for array
8 16 10 4 6 10 2 12 8 11 8 3 2 1 6 10 0
with size 17 and start marker 0 is false

The right magnetic result for array
2 5 1 5 7 4 6 4 8 9 14 27 0
with size 13 and start marker 0 is true

The right magnetic result for array
30 31 32 32 35 4 9 20 31 17 20 9 15 37 44 25 2 7 22 4
with size 20 and start marker 8 is false

The right magnetic result for array
21 21 18 33 50 2 18 1 33 33 34 35 20 40 41 49 37 20 34 15
with size 20 and start marker 2 is false

The right magnetic result for array
14 24 42 27 27 20 3 44 38 32 2 16 37 45 18 50
with size 16 and start marker 3 is false

The right magnetic result for array
12 3 1 24 14 35 24 39 15 9 40 1 48 2 2 16 27 37 27 48
with size 20 and start marker 11 is false

The right magnetic result for array
44 15 47 9 50 31 15 2 4 2
with size 10 and start marker 0 is false

The right magnetic result for array
4 48
with size 2 and start marker 1 is true

The right magnetic result for array
11 20 45 48 23 13 21 22 6 30 31 14 8 6 8 24 35 33 25
with size 19 and start marker 11 is true

The right magnetic result for array
50 6 8 49 27
with size 5 and start marker 4 is true

The right magnetic result for array
7 29 38 40 23 11 41 8 47 14 25 32 32 43 39 35 7 25 48
with size 19 and start marker 16 is false

The right magnetic result for array

```
7 32 23
with size 3 and start marker 2 is true

The right magnetic result for array
40
with size 1 and start marker 0 is true

The right magnetic result for array
15 37 19 41 39
with size 5 and start marker 2 is false

The right magnetic result for array
29 14 30 46 39 17 11 50 40 30 14 49 14 6 34 21 17 22 34 44 12 9 29 19 26 26 4 37 34 48 9 36
2 30 1 29 21 18 18
with size 39 and start marker 6 is false

The right magnetic result for array
48 23 15 4 6 22 1 5 40 47 8 18 43 14 41 21 47 7 8 45
with size 20 and start marker 14 is false

The right magnetic result for array
25 3 3 15 13 35 3 26 38 24 9 17 45 49 35 13 8 1 22 24 42 43 11 22 31 24 12 4
with size 28 and start marker 1 is false

The right magnetic result for array
…
with size 441823 and start marker 0 is false (output is too long, only paste the result)
```

e) There are three characteristics that indicate the given array is unsolvable, one is the array has an element, whose value is bigger than the last index multiply by 2. Under this situation, the process will highly likely (if the process will visit that element) gets stuck on that element (because it cannot move forwards or backwards). The second main characteristic is that if we let the process go forwards whenever it is possible, and backwards only when it cannot go forwards. Then after the process goes backwards, once we are going to visit an index that has been visited before, the game will fail. Another characteristic is that since the worst case for solvable array is an array of 1's or 2's, in which case, the process will take n times to succeed. Then we can make a counter and each time we visit an index, increment that counter. Once the counter is bigger than n, game will fail. In my stack version algorithm, I used all these characteristics.