**Strobogrammatic Number II**

Given an integer n, return all the **strobogrammatic numbers** that are of length n. You may return the answer in **any order**.

A **strobogrammatic number** is a number that looks the same when rotated 180 degrees (looked at upside down).

**Example 1:**

**Input:** n = 2

**Output:** ["11","69","88","96"]

**Example 2:**

**Input:** n = 1

**Output:** ["0","1","8"]

**Constraints:**

* 1 <= n <= 14

Hint:

Try to use recursion and notice that it should recurse with *n* - 2 instead of *n* - 1.

Solution

Overview

We need to return all strobogrammatic numbers of length N*N*.  
A strobogrammatic number is a number that looks the same when rotated 180 \degree180° (looked at upside down).

There are two possibilities, the strobogrammatic number can have:

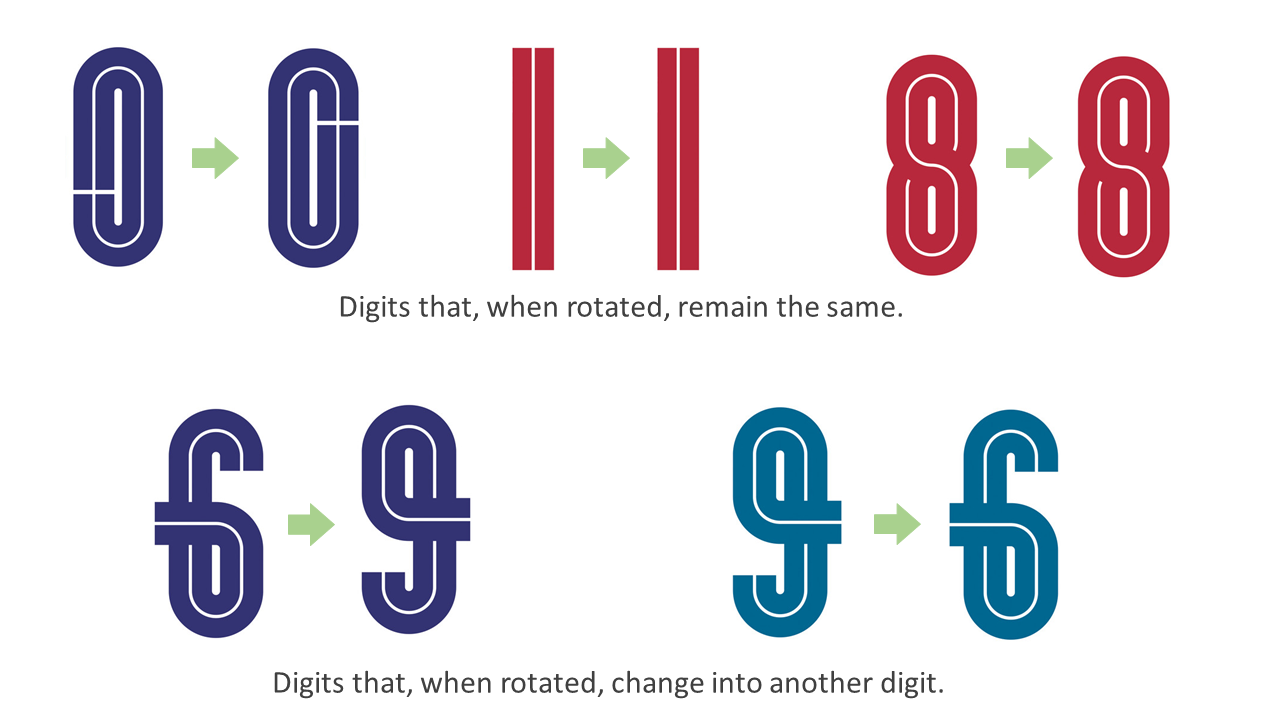
* An even number of digits: Then each digit will swap and rotate with some other digit.
* Or, an odd number of digits: Then all digits will swap and rotate with some other digit except the center one which will remain at the same place but rotate 180 \degree180°.

We can visualize the rotation process using the following slideshow.

**First, let's see which digits can be used in strobogrammatic numbers.**

Two types of digits are useful here:

* One, the digits which after rotation of 180 \degree180° remains the same, like 0, 1 (here 1 is a straight line only), and 8.
* Other is, the digits which after rotation converts into a different digit, like 6 changes to 9 and vice versa.



**Second, let's see which digits can occupy which places in the strobogrammatic number.**

* As demonstrated in the slideshow, when the length is odd, the middle digit remains at the same position. Thus this digit must be something that remains the same when rotated (e.g. 0, 1, 8).
* And when two positions interchange, we can use both types of digits, either we put two digits which change into one another (e.g. 6 and 9), or we put two same digits which remain same when rotated (e.g. 8 and 8).

Approach 1: Recursion

**Intuition**

Now, we know which digits can be used so let's try solving some sample cases.  
We will try sample cases for N*N* equals 11 up to N*N* equals 44 which might be a bit lengthy, but by observing these sample cases, we should be able to come up with a recursive relation for this problem.

**If the length of the number is 11:**  
The digit will remain at the same position and rotate 180 \degree180°.  
Thus, the numbers can only be, 00, 11, and 88.

**If the length is 22:**  
We have two positions (say indices 0 and 1), which will interchange.

So we can use,

* Either two same digits which remain the same after rotation, 1111 and 8888.  
  Take note that we can't use 00 here because 0000 is not a 2 digit number. So we can't place 00 at the 0^{th}0*th* index of any number, except for the number 00 itself.
* Or, two digits that change into one another, 6969 and 9696.

Thus, 2-digit strobogrammatic numbers will be, 1111, 6969, 8888, and 9696.

**If the length is 33:**  
We have three positions (say indices 0, 1, and 2) and out of these three positions, index 0 and index 2 will interchange, but index 1 will remain in the same place.

Suppose we have all of the 1-digit strobogrammatic numbers. To find the 3-digit strobogrammatic numbers, we just need to append one extra digit at the beginning and at the end which are reversible. Here, reversible means it will keep the number strobogrammatic. We know that we can use both types of digits (those that change into each other or remain the same after rotation) for this position. But keep in mind that we can't append 00 at the beginning.

=> '0' + (1-digit strobogrammatic numbers) + '0'

'0' + '0' + '0' = '000' (1-digit number) (not valid)

'0' + '1' + '0' = '010' (2-digit number) (not valid)

'0' + '8' + '0' = '080' (2-digit number) (not valid)

=> '1' + (1-digit strobogrammatic numbers) + '1'

'1' + '0' + '1' = '101'

'1' + '1' + '1' = '111'

'1' + '8' + '1' = '181'

=> '6' + (1-digit strobogrammatic numbers) + '9'

'6' + '0' + '9' = '609'

'6' + '1' + '9' = '619'

'6' + '8' + '9' = '689'

=> '8' + (1-digit strobogrammatic numbers) + '8'

'8' + '0' + '8' = '808'

'8' + '1' + '8' = '818'

'8' + '8' + '8' = '888'

=> '9' + (1-digit strobogrammatic numbers) + '6'

'9' + '0' + '6' = '906'

'9' + '1' + '6' = '916'

'9' + '8' + '6' = '986'

Thus, these are all of the 3-digit strobogrammatic numbers: 101101, 111111, 181181, 609609, 619619, 689689, 808808, 818818, 888888, 906906, 916916, and 986986.

**If the length is 44:**  
We have four positions (say indices 0, 1, 2, and 3), and out of these four positions, (index 0, index 3) and (index 1, index 2) will interchange.

Suppose we have all of the 2-digit strobogrammatic numbers. To find the 4-digit strobogrammatic numbers, we just need to append one extra digit at the beginning and the end which are reversible (just like previously discussed).

But here, in 2-digit strobogrammatic numbers, we can include 0000 because after appending a nonzero digit to the beginning and the end, '0000' becomes a valid 4-digit number.

=> '1' + (2-digit strobogrammatic numbers) + '1'

'1' + '00' + '1' = '1001'

'1' + '11' + '1' = '1111'

'1' + '69' + '1' = '1691'

'1' + '88' + '1' = '1881'

'1' + '96' + '1' = '1961'

=> '6' + (2-digit strobogrammatic numbers) + '9'

'6' + '00' + '9' = '6009'

'6' + '11' + '9' = '6119'

'6' + '69' + '9' = '6699'

'6' + '88' + '9' = '6889'

'6' + '96' + '9' = '6969'

=> '8' + (2-digit strobogrammatic numbers) + '8'

'8' + '00' + '8' = '8008'

'8' + '11' + '8' = '8118'

'8' + '69' + '8' = '8698'

'8' + '88' + '8' = '8888'

'8' + '96' + '8' = '8968'

=> '9' + (2-digit strobogrammatic numbers) + '6'

'9' + '00' + '6' = '9006'

'9' + '11' + '6' = '9116'

'9' + '69' + '6' = '9696'

'9' + '88' + '6' = '9886'

'9' + '96' + '6' = '9966'

Thus, 4-digit strobogrammatic numbers will be, 10011001, 11111111, 16911691, 18811881, 19611961, 60096009, 61196119, 66996699, 68896889, 69696969, 80088008, 81188118, 86988698, 88888888, 89688968, 90069006, 91169116, 96969696, 98869886, and 99669966.

Following this pattern, we can conclude that to find all strobogrammatic numbers with N-digits, we first need to find all strobogrammatic numbers with (N - 2) digits and then append reversible digits to the beginning and the end.

Now, instead of strobogrammatic numbers of length N, we need to find strobogrammatic numbers of length N - 2.  
This is the same problem but with a smaller input. So we can approach this problem using **recursion**.

If you are not familiar with recursion, you can check out our [Recursion Explore Card](https://leetcode.com/explore/featured/card/recursion-i/).

generateStroboNumbers(N) = List("digit1" + "number" + "digit2"

for each number in generateStroboNumbers(N - 2)

for each (digit1, digit2) in reversiblePairs

)

This will be our recursive logic, which is called as **recurrence relation**.

Now, to terminate the recursive calls, we need to think about base cases. A base case is some input for which we can determine the result without making further recursive calls. With each recursive call, we decrease N*N* by 22. Therefore, if N*N* is initially even, we will make recursive calls until N*N* is 00. Otherwise, if N*N* is odd, we will make recursive calls until N*N* is 11. We can determine the result for input N = 0*N*=0 and input N = 1*N*=1 without using recursion (as shown above). Thus, we can make our base cases generateStroboNumbers(0) = [''] and generateStroboNumbers(1) = ['0', '1', '8'].

And the empty string can be considered as a 0-digit strobogrammatic number.

**Note:** We need to take N == 0 as our base case, not N == 2, because 2-digit numbers will not consider '00' as a valid number. Thus, if we generate 4-digit numbers using N == 2 as the base case, we will skip some numbers. However, if we start with N == 0, we will have an option to generate '00' as a valid number because we know the current position is not the first digit of the 4-digit number, so we can append 0.

**Algorithm**

1. Initialize a data structure reversiblePairs, which contains all pairs of reversible digits.
2. Call and return the recursive function, generateStroboNumbers(n, finalLength), where the first argument indicates that the current call will generate all n-digit strobogrammatic numbers. The second argument indicates the length of the final strobogrammatic numbers that we will generate and will be used to check if we can add '0' to the beginning and end of a number.
3. Create a function generateStroboNumbers(n, finalLength) which will return all strobogrammatic numbers of n-digits:
   * Check for base cases, if n == 0 return an array with an empty string [""], otherwise if n == 1 return ["0", "1", "8"].
   * Call generateStroboNumbers(n - 2, finalLength) to get all the strobogrammatic numbers of (n-2) digits and store them in subAns.
   * Initialize an empty array currStroboNums to store strobogrammatic numbers of n-digits.
   * For each number in prevStroboNums we append all reversiblePairs at the beginning and the end except when the current reversible pair is '00' and n == finalLength (because we can't append '0' at the beginning of a number) and push this new number in ans.
   * At the end of the function, return all the strobogrammatic numbers, i.e. currStroboNums.

**Implementation**

let reversiblePairs = [

['0', '0'], ['1', '1'],

['6', '9'], ['8', '8'], ['9', '6']

];

let generateStroboNumbers = (n, finalLength) => {

if (n == 0) {

// 0-digit strobogrammatic number is an empty string.

return [""];

}

if (n == 1) {

// 1-digit strobogrammatic numbers.

return ["0", "1", "8"];

}

let prevStroboNums = generateStroboNumbers(n - 2, finalLength);

let currStroboNums = [];

prevStroboNums.forEach((prevStroboNum) => {

reversiblePairs.forEach((pair) => {

if (pair[0] != '0' || n != finalLength) {

currStroboNums.push(pair[0] + prevStroboNum + pair[1]);

}

});

});

return currStroboNums;

};

let findStrobogrammatic = function(n) {

return generateStroboNumbers(n, n);

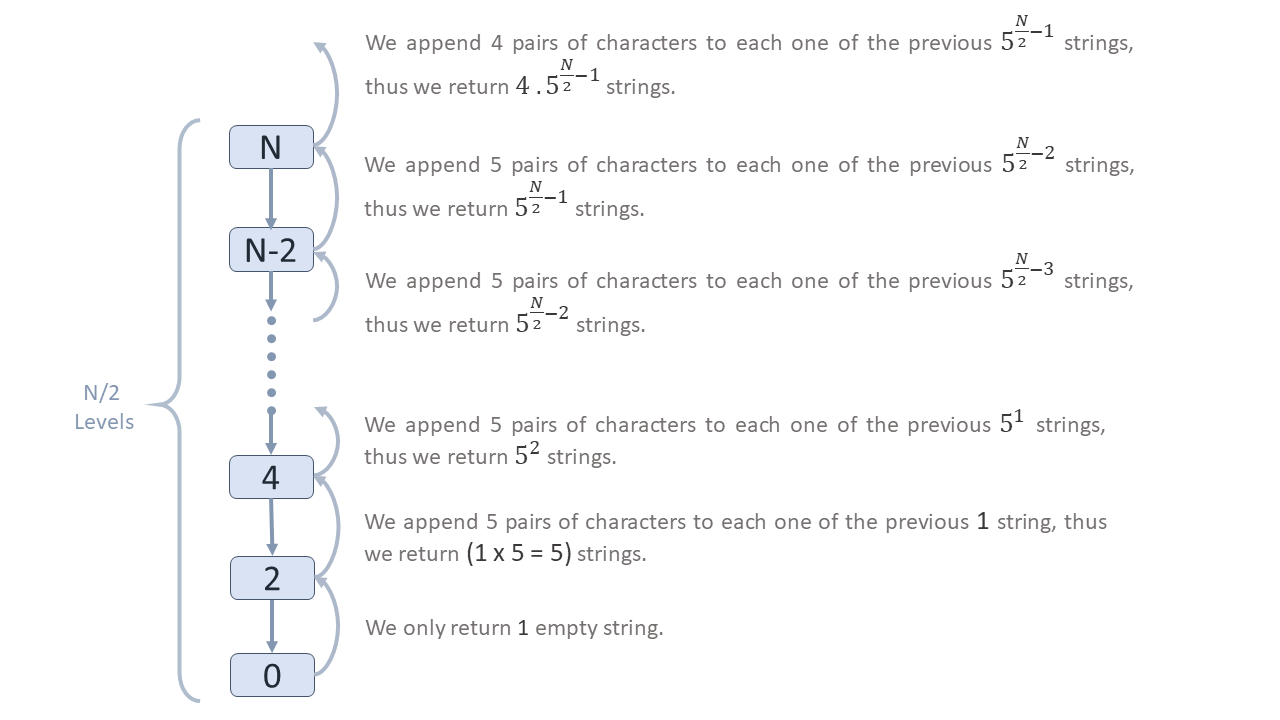
};

**Complexity Analysis**

Here, N*N* is the length of strobogrammatic numbers we need to find.

Let's try visualizing the recursion tree for our approach. Our recursive function makes another recursive call and with each recursive call we decrease N*N* by 22.  
In series, [[ N*N*, N-2*N*−2, N-4*N*−4, N-6*N*−6, .........., 00 ]] using arithmetic progression (AP) formula we can say we have N / 2*N*/2 elements.  
Hence, in our recursion tree, we will have at most N / 2*N*/2 levels.

* Time complexity: N \cdot 5^{\lfloor N/2 \rfloor + 1}*N*⋅5⌊*N*/2⌋+1.
  + In each recursive level, we iterate over all prevStroboNums strings and append 55 pairs of characters to each one of them; thus, we increase currStroboNums array size by a factor of 55 with each level. Except for the last level, where it increases by a factor of 44 because we omit 00.



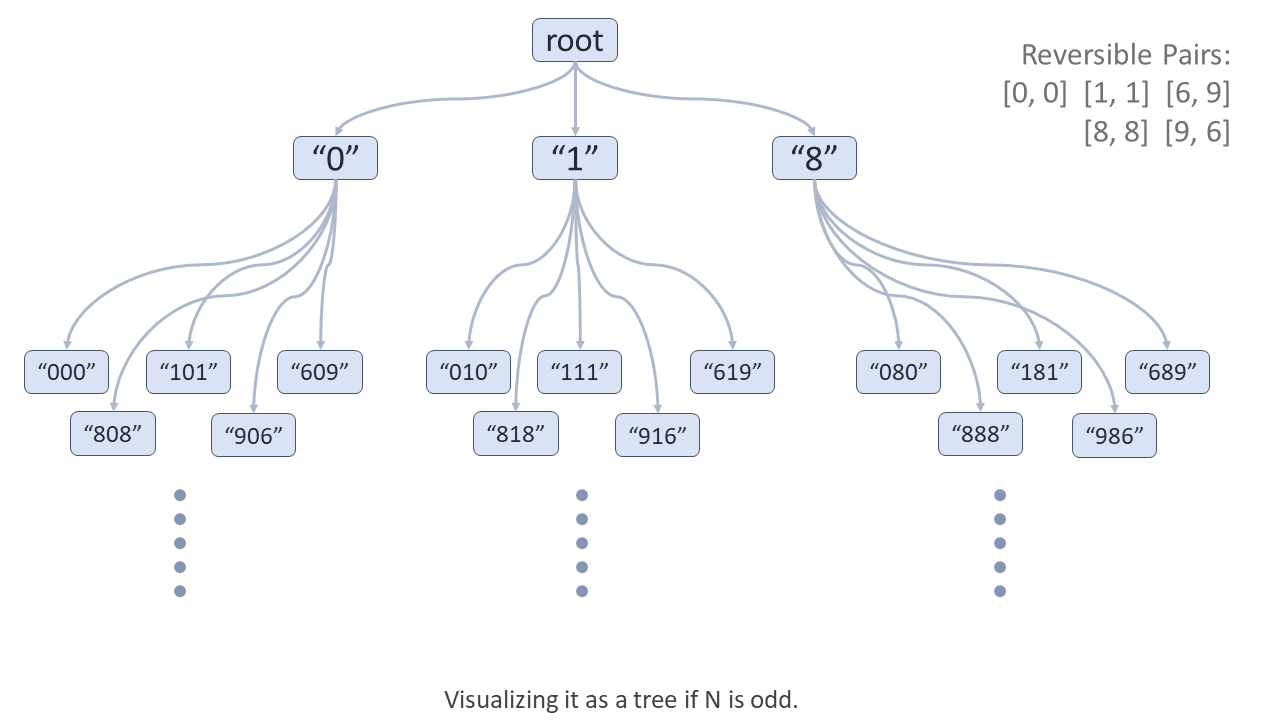
* + So, when N*N* is even total iterations done will be, 5 + 5^{2} + 5^{3} + .... + 5^{(N/2) - 1} + 4 \cdot 5^{(N/2) - 1} \approx \bf{5^{N/2}}5+52+53+....+5(*N*/2)−1+4⋅5(*N*/2)−1≈**5N/2**.
  + Similarly, when N*N* is odd we start with 33 strings, and append 55 pairs of characters to each string in each level except the last level. So, here total iterations done will be, 3 \cdot 5 + 3 \cdot 5^{2} + 3 \cdot 5^{3} + .... + 3 \cdot 5^{(N-3)/2} + 3 \cdot 4 \cdot 5^{(N-3)/2} \approx 5^{((N-1)/2) + 1} = \bf{5^{\lfloor N/2 \rfloor + 1}}3⋅5+3⋅52+3⋅53+....+3⋅5(*N*−3)/2+3⋅4⋅5(*N*−3)/2≈5((*N*−1)/2)+1=**5**⌊**N/2**⌋+**1**.
  + And appending of a character at the end of a string takes O(1)*O*(1) time, but appending at the beginning takes O(N)*O*(*N*) time. Thus, we will perform the O(N)*O*(*N*) operation in each iteration.
  + Thus, the overall time taken will be N \cdot 5^{\lfloor N/2 \rfloor + 1}*N*⋅5⌊*N*/2⌋+1.
* Space complexity: O(N \cdot 5^{\lfloor N/2 \rfloor})*O*(*N*⋅5⌊*N*/2⌋).
  + We make at most N/2*N*/2 recursive calls. Thus, the call stack will use O(N/2)*O*(*N*/2) space.
  + currStroboNums is the output array, thus in the last recursion, it will not be considered as auxiliary space. However, in earlier recursions currStroboNums array will be considered as auxiliary space.
  + We store at most \bf{5^{(N/2) - 1}}**5**(**N/2**)−**1** strings when N*N* is even, otherwise 3 \cdot 5^{((N-1)/2) - 1} \approx \bf{5^{\lfloor N/2 \rfloor}}3⋅5((*N*−1)/2)−1≈**5**⌊**N/2**⌋ when N*N* is odd in our currStroboNums array and each string is of length N-2*N*−2.
  + Thus, overall space used will be (N-2) \cdot 5^{\lfloor N/2 \rfloor} + N/2 \approx N \cdot 5^{\lfloor N/2 \rfloor}(*N*−2)⋅5⌊*N*/2⌋+*N*/2≈*N*⋅5⌊*N*/2⌋.

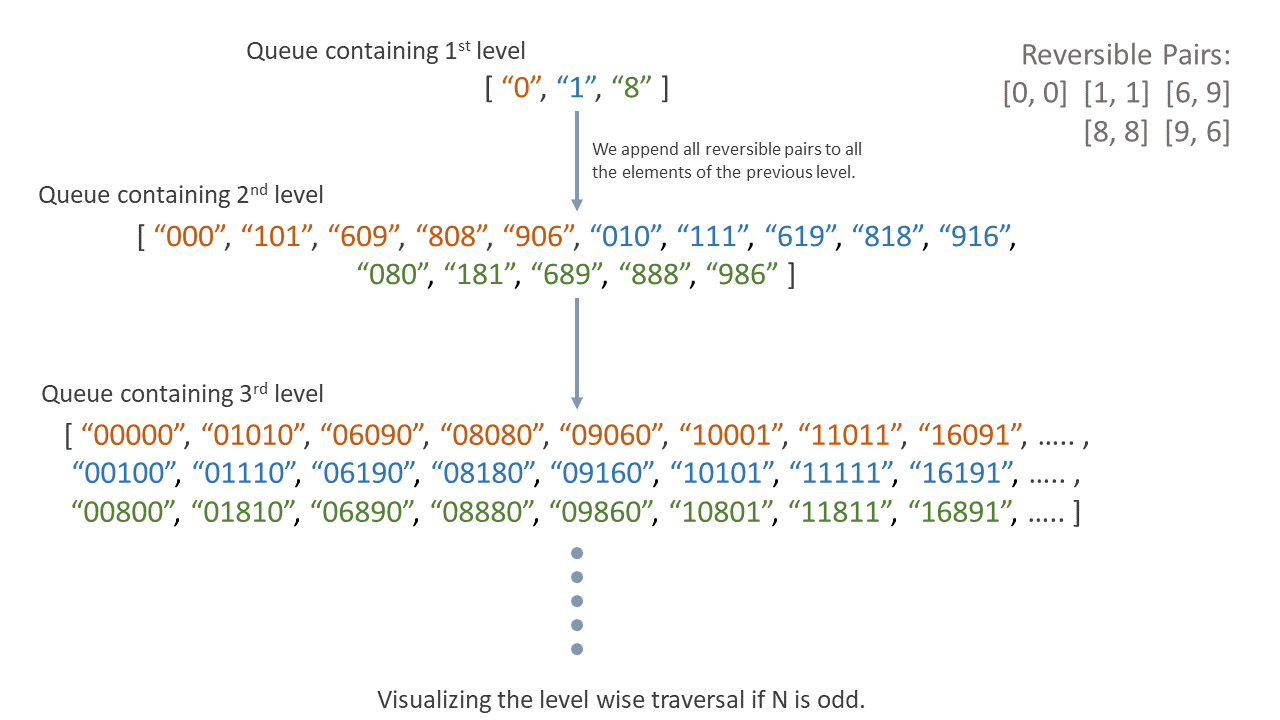
Approach 2: Iteration (Level Order Traversal)

**Intuition**

We can also implement the above approach iteratively by using a queue instead of recursion. This process will be the same as doing a level order traversal. While traversing all elements of one level, we will push the next level's elements into the queue and repeat this process for \lfloor N/2 \rfloor⌊*N*/2⌋ levels.

This process can be visualized using the following image. Here, we can see the first level has 1-digit strobogrammatic numbers. Then in the next level, after appending reversible digits, we get 3-digit strobogrammatic numbers, then 5-digit strobogrammatic numbers, and so on.





We keep all strobogrammatic numbers of (N-2) digits in the queue and then append all reversible pairs to all the numbers in the queue to get strobogrammatic numbers of N digits.

We also know we can't append the '00' pair at the beginning of the number.  
So here, we will use one variable, currStringsLength, which will denote the length of the strobogrammatic numbers in the current level. When currStringsLength equals N*N*, the added number will be in the first position; thus, we cannot append '0'.

**Algorithm**

1. Initialize:
   * reversiblePairs as a data structure that stores all pairs of reversible digits.
   * q as a queue for doing level order traversal.  
     In javascript and python, we will use an array as a queue for easier implementation.
   * currStringsLength as 0 if n is even or 1 if n is odd, to denote the number of digits in each string, in the current level strobogrammatic numbers.
2. If n is even, we push [""] in the queue, and initialize currStringsLength with 0, because n will decrease till 0. Thus for this case, our starting case will be 0-digit strobogrammatic numbers.  
   Otherwise, if n is odd, we push ["0", "1", "8"] in the queue and initialize currStringsLength with 1, because for odd n the starting case will be 1-digit strobogrammatic numbers.
3. We will iterate over one whole level stored in the queue until currStringsLength becomes equal to n.
   * In each level, we will append two characters, thus increasing currStringsLength by 2.
   * For each number in the current level (present in the queue) we pop it and, append all reversiblePairs at the beginning and the end except when the current reversible pair is '00' and currStringsLength == n (because we can't append '0' at the beginning of a number) and push this new number in the queue again.
4. After traversing all levels, the queue will contain all n digit strobogrammatic numbers, thus we push them in a stroboNums array and return it.

**Implementation**

let reversiblePairs = [

['0', '0'], ['1', '1'],

['6', '9'], ['8', '8'], ['9', '6']

];

let findStrobogrammatic = function(n) {

let currStringsLength;

let q = [];

// When n is even, it means when decreasing by 2 we will go till 0.

if (n % 2 == 0) {

// We will start with 0-digit strobogrammatic numbers.

currStringsLength = 0;

q = [""];

} else {

// We will start with 1-digit strobogrammatic numbers.

currStringsLength = 1;

q = ["0", "1", "8"];

}

while (currStringsLength < n) {

currStringsLength += 2;

let nextLevel = []

q.forEach((number) => {

reversiblePairs.forEach((pair) => {

if (currStringsLength != n || pair[0] != '0') {

nextLevel.push(pair[0] + number + pair[1]);

}

});

});

q = nextLevel;

}

return q;

};

**Complexity Analysis**

Here, N*N* is the length of strobogrammatic numbers we need to find.

We start with strings of length 0 or 1 and go until they are N*N* digits long. At each level, we increment the string length by 2. Thus, we must traverse N/2*N*/2 levels.

* Time complexity: O(N \cdot 5^{\lfloor N/2 \rfloor + 1})*O*(*N*⋅5⌊*N*/2⌋+1).
  + In each level we iterate over all queue elements and append 55 pairs of characters to each one of them; thus, we increment our queue size by 5 times (approximately) with each level.
  + So, when N*N* is even we start with 11 empty string and the total number of elements traversed using the queue will be, 1 + 5 + 5^{2} + 5^{3} + .... + 5^{(N/2) - 1} + 4 \cdot 5^{(N/2) - 1} \approx \bf{5^{N/2 + 1}}1+5+52+53+....+5(*N*/2)−1+4⋅5(*N*/2)−1≈**5N/2**+**1**.
  + Similarly, when N*N* is odd we start with 33 strings. So, here the total number of elements traversed will be, 3 + 3 \cdot 5 + 3 \cdot 5^{2} + 3 \cdot 5^{3} + .... + 3 \cdot 5^{(N-3)/2} + 3 \cdot 4 \cdot 5^{(N-3)/2} \approx 5^{(N-1)/2 + 1} = \bf{5^{\lfloor N/2 \rfloor + 1}}3+3⋅5+3⋅52+3⋅53+....+3⋅5(*N*−3)/2+3⋅4⋅5(*N*−3)/2≈5(*N*−1)/2+1=**5**⌊**N/2**⌋+**1**.
  + And appending of a character at the end of a string takes O(1)*O*(1) time, but appending at the beginning takes O(N)*O*(*N*) time. Thus, we will perform the O(N)*O*(*N*) operation in each iteration.
  + Thus, the overall time taken will be N \cdot 5^{\lfloor N/2 \rfloor + 1}*N*⋅5⌊*N*/2⌋+1.
* Space complexity: O(N \cdot 5^{\lfloor N/2 \rfloor})*O*(*N*⋅5⌊*N*/2⌋).
  + In the last level, we store at most 4 \cdot 5^{(N/2) - 1} \approx \bf{5^{N/2}}4⋅5(*N*/2)−1≈**5N/2** strings when N*N* is even, otherwise 3 \cdot 4 \cdot 5^{(N-3)/2} \approx 5^{(N-1)/2} = \bf{5^{\lfloor N/2 \rfloor}}3⋅4⋅5(*N*−3)/2≈5(*N*−1)/2=**5**⌊**N/2**⌋ strings when N*N* is odd in the queue and each string is of length N*N*.
  + Thus, overall space used will be N \cdot 5^{\lfloor N/2 \rfloor} + N/2 \approx N \cdot 5^{\lfloor N/2 \rfloor}*N*⋅5⌊*N*/2⌋+*N*/2≈*N*⋅5⌊*N*/2⌋.

**Note:** In javascript and python, in the last iteration the arrary is the output array thus it will not be considered in auxiliary space. But still the overall order of the complexity remains the same.