



Problem Set 1—Algorithmic Strategies

Important notes:

- Watch [this video](#) recorded by Joram Erbarth (M23) with advice on how to prepare for CS110 assignments. Most of the suggestions will also apply to other CS courses, so make sure to bookmark this video for future reference. You will also notice that the video refers to submitting primary and secondary resources, but for this specific assignment, since you will answer questions directly in the notebook, you won't need to worry about uploading your work.
- Make sure to include your work whenever you see the labels `###YOUR DOCSTRING HERE` or `###YOUR CODE HERE` (there are several code cells per question you can use throughout the notebook, but you need not use them all).
- Please refer to the CS110 course guide on how to submit your assignment materials.
- If you have any questions, do not hesitate to reach out to the TAs in the Slack channel `#cs110-algo`, or come to the instructors' OHs.

Question 1 of 10

Setting up:

Start by stating your name and identifying your collaborators. Please comment on the nature of the collaboration (for example, if you briefly discussed the strategy to solve problem 1, say so, and explicitly point out what you discussed). Example:

Name: Ahmed Souza

Collaborators: Lily Shakespeare, Anitha Holmes

Details: I discussed the iterative strategy of problem 1 with Lily, and asked Anitha to help me design an experiment for problem 2.

Normal ⌵ **B** *I* U A

Name : Uladzislau Andreichuk

Problem 1

A strobogrammatic number is a number that looks the same when rotated 180 degrees (i.e., upside down). For instance, **8** stays the same after rotating 180 degrees; therefore, it is a strobogrammatic number. Here, we will define **1** as a strobogrammatic number, though this will depend on the font used.

Your goal is to define two different strategies to return all the strobogrammatic numbers of length n , where n is a positive integer. The questions below will guide you through the necessary work to achieve this.

Question 2 of 10

A. Explain how you would design two approaches (an iterative and a recursive one) to solving this computational problem in plain English to determine whether a number is strobogrammatic. Avoid using technical jargon (such as "indices" and while/for loops), and focus on explaining in as simple terms as possible how the algorithm works and how it is guaranteed, upon termination, to return the correct answer.

Normal ⌵ **B** *I* U A

Iterative approach:

First, I would check if the requested length of strobogrammatic numbers is valid, that is, if it is a negative number or a string, then the algorithm should terminate and state that the requested length is invalid. Then, I would start by checking if the given length is odd or even. If it is even, then I would set the middle of future strobogrammatic numbers of even length to be empty, while if it is odd, then I would set the possible middle numbers to be 0, 1, or 8 (0, 1, and 8 are the only strobogrammatic numbers on their own). After I outlined all potential middle values, I would start adding all possible pairs of strobogrammatic numbers to the previously identified middle values: one to the left and one to the right. Such pairs are 0-0, 1-1, 6-9, 8-8, and 9-6. We can see that although 6 and 9 are not strobogrammatic on their own, they can produce strobogrammatic numbers if added on opposing sides from each other. Note that I would only add a 0-0 pair to the middle value if the newly obtained number after the 0-0 addition is at least 2 digits short of the target length. Otherwise, the newly obtained number would be of the desired length and start with 0, which is not a valid number. We can also only add pairs of numbers, one from each side, because if we add an odd number of numbers, some

identified pairs to the middle values until the numbers grow to the desired length. Once the numbers reach the desired length, the algorithm terminates. The algorithm is guaranteed to return the correct solution since it checks all possible strobogrammatic numbers for the middle and all possible pairs of strobogrammatic numbers for both sides, which in the end produces all possible strobogrammatic numbers of the requested length.

For example,

The desired length of the strobogrammatic words is 3 digits. It is a valid length as it is a positive integer. Since it is odd, the initial middle values are set to 0, 1, and 8. Then, to each of the middle values, I add identified pairs and get 101, 609, 808, 906, 111, 619, 818, 916, 181, 689, 888, and 986. Note that I did not add the 0-0 pair to the middle values because the newly obtained numbers, such as 000 or 080, would be already of desired length but start with 0, which is not valid. Since all the obtained numbers are of length 3, the algorithm terminates as it found the solution: all possible strobogrammatic numbers of length 3 digits.

Recursive approach:

Just like before, the first thing I would do is to check if the requested length is valid, which is a positive integer. If not, then the algorithm can not run. Otherwise, I would start by setting up a couple of conditions that, if satisfied, would terminate the algorithm. Such conditions are:

1. If the desired length of the strobogrammatic numbers is 0, then the algorithm should immediately produce a solution, which is 'no numbers'
2. If the desired length of the strobogrammatic numbers is 1, then the algorithm should immediately produce a solution, which is numbers 0, 1, and 8, the only strobogrammatic numbers of length 1 digit.

These set of conditions can be viewed as the smallest subproblems for which we can easily find the solution. The idea of this algorithm then is to take the problem and break it down into smaller subproblems until the subproblems are the ones we identified above. To reach these subproblems, we would repeat the algorithm over and over, each time decreasing the desired length of strobogrammatic numbers by 2 until the length becomes either 0 (if the original length was an even number) or 1 (if the original length was an odd number). This way, we are able to reach the subproblems we know how to solve. When we do reach them, we would be able to combine the solutions for these subproblems back to find the answer to the original problem. by performing a series of operations on the answers to the subproblems. When we get the answer to the smallest subproblems, that is either 'no number' (nothing) or numbers [0,1,8], we assign these answers to be the middle digits for the future numbers as we build our way back up to the original problem. After the middle digits are set, we would add all possible pairs of strobogrammatic numbers to them: one to the left and one to the right. Such pairs are 0-0, 1-1, 6-9, 8-8, and 9-6. Note that I would only add a 0-0 pair to the middle value if the newly obtained number after the 0-0 addition is at least 2 digits short of the target length. Otherwise, the newly obtained number would be of the desired length and start with 0, which is not a valid number. This procedure is the same as the one that was performed in the previous algorithm. After we added these pairs of numbers to the middle digits, we set these newly obtained numbers as new middle digits for future numbers. This is how we combine the solutions for the smallest subproblems to eventually arrive at the solution for the original problem. We do so until we reach the proposal's desired length of strobogrammatic numbers. This would be the final solution to the problem. The solution is guaranteed to be correct because we use the solutions of the subproblems that we know to be true to derive the solution for the whole problem by adding all possible combinations of strobogrammatic numbers until we reach the requested length.

For example,

The desired length of the strobogrammatic words is 3 digits. It is a valid length as it is a positive integer. We set the solutions for the smallest subproblems: if the requested length is 0, then the solution is 'no strobogrammatic numbers of such length', or empty, and if the requested length is 1, then the solution is numbers [0,1,8]. Then, according to the algorithm, we want to identify the middle values by running the algorithm again, but with the requested length decreased by 2: $3-2=1$. Now that we reached length 1, we know the answer - [0,1,8]. We build our way back up and use each of these numbers as middle values for future numbers. To each of the middle values, I add identified pairs and get 101, 609, 808, 906, 111, 619, 818, 916, 181, 689, 888, and 986. Note that I did not add the 0-0 pair to the middle values because the newly obtained numbers, such as 000 or 080, would be already of desired length but start with 0, which is not valid. Since all the obtained numbers are of length 3, the algorithm terminates as it found the solution: all possible strobogrammatic numbers of length 3 digits.

Question 3 of 10

B. Using your descriptions from question 1 A, produce two flowcharts that correctly describe each approach. Please be particularly careful about describing the termination condition for both algorithms. Upload your flowcharts below.

Question 1 Flowchart.pdf (311 kB) ×

Drop or [upload](#) a file here

C. Provide both recursive and iterative Python implementations that return all the strobogrammatic numbers of length n as a sorted array.

Remember to add at least 3 test cases to demonstrate that your function is correctly implemented in Python.

Code Cell 1 of 22

```
In [1] 1 def strobogrammatic_iterative(n):
2     '''
3     Return all the strobogrammatic numbers that are of length n through iterative
4     approach.
5     -----
6     Parameters:
7     n: int
8         The targeted length of the digit.
9     -----
10    Returns: list
11        All strobogrammatic numbers that are of the targeted length.
12    '''
13    #check that the input is a positive integer
14    if type(n)!=int or not n>=0:
15        return 'Invalid input'
16
17    #check if the number is odd or even to determine the middle values
18    if n%2!=0:
19        result=["0", "1", "8"]
20    else:
21        result=['']
22
23    #keep adding numbers from the sides untill we reach the required length n
24    for i in range(n//2):
25        mid=[]
26        #add numbers to each middle value stored in result list
27        for j in result:
28            #add 0s on the sides only if the number will not end up starting with 0
29            if i<(n//2)-1:
30                mid.append("0" + j + "0")
31            #adding the rest of strobogrammatic pairs of numbers from the sides and storing in mid list
32            mid.append("1" + j + "1")
33            mid.append("6" + j + "9")
34            mid.append("8" + j + "8")
35            mid.append("9" + j + "6")
36
37        result=mid #update the results list with new middle values
38
39    return sorted(result)
40
41
42
43 def strobogrammatic_recursive(n):
44     '''
45     Return all the strobogrammatic numbers that are of length n through recursive
46     approach.
47     -----
48     Parameters:
49     n: int
50         The targeted length of the digit.
51     -----
52    Returns: list
53        All strobogrammatic numbers that are of the targeted length.
54    '''
55    #check that the input is a positive integer
56    if type(n)!=int or not n>=0:
57        return 'Invalid input'
```

```

59 def rec_help(n,length): # define a helper function for recursion
60     '''
61     Return all the strobogrammatic numbers that are of length n through recursive
62     approach.
63     -----
64     Parameters:
65     n: int
66         The targeted length of the digit.
67     length: int
68         The targeted length of the digit.
69     -----
70     Returns: list
71         All strobogrammatic numbers that are of the targeted length.
72     '''
73     #define base cases to stop recursion
74     if n==0:
75         return ['']
76     if n==1:
77         return ["0", "1", "8"]
78
79     result=[]
80     mid=rec_help(n-2,length) #create middle values by recursing the function for n-2 until we hit a base case
81
82     #for each middle value in the list, add defined pairs of strobogrammatic numbers and store in result list
83     for j in mid:
84         #add 0s on the sides only if the number will not end up starting with 0
85         if len(j)<length-2:
86             result.append("0" + j + "0")
87
88             result.append("1" + j + "1")
89             result.append("6" + j + "9")
90             result.append("8" + j + "8")
91             result.append("9" + j + "6")
92
93     return sorted(result)
94
95 return rec_help(n,n)

```

Run Code

Code Cell 2 of 22

```

In [2] 1 #test cases for the recursive algorithm using various types of inputs
2 assert(strobogrammatic_recursive(4)==
3 ['1001','1111','1691','1881','1961','6009','6119','6699','6889','6969','8008','8118','8698','8888','8968','9006','9116','969
4 6','9886',
5 '9966'] )
6 assert(strobogrammatic_recursive(3)==['101','111','181','609','619','689','808','818','888','906','916','986'])
7 assert(strobogrammatic_recursive(1)==['0','1','8'])
8 assert(strobogrammatic_recursive(0)==[''])
9 assert(strobogrammatic_recursive(-10)=='Invalid input')
10 assert(strobogrammatic_recursive('hello')=='Invalid input')
11 print('Passed all cases')

```

Run Code

```
Out [2]    Passed all cases
```

Code Cell 3 of 22

```
In [3]: 1 #test cases for the iterative algorithm using various types of inputs
2 assert(strobogrammatic_iterative(4)==
3 ['1001','1111','1691','1881','1961','6009','6119','6699','6889','6969','8008','8118','8698','8888','8968','9006','9116','9696','9886',
4 '9966'] )
5 assert(strobogrammatic_iterative(3)==['101','111','181','609','619','689','808','818','888','906','916','986'])
6 assert(strobogrammatic_iterative(1)==['0','1','8'])
7 assert(strobogrammatic_iterative(0)==[''])
8 assert(strobogrammatic_iterative(-10)=='Invalid input')
9 assert(strobogrammatic_iterative('hello')== 'Invalid input')
10 print('Passed all cases')
```

Run Code

```
Out [3]    Passed all cases
```

Code Cell 4 of 22

```
In [4] 1 ### YOUR CODE HERE
        2
        3
```

Run Code

Code Cell 5 of 22

```
In [5] 1 ### YOUR CODE HERE
```

Run Code

Question 4 of 10

Why are your test cases appropriate or possibly sufficient?


Normal **B** *I* U “ ” </> A

My test cases are appropriate and possibly sufficient because they encompass possibly all potential input categories. In particular, the test cases include inputs which are a negative odd number, negative even number, 0, 1, positive odd number, and positive even number. If the algorithms pass all of these test cases, it would give us a lot of confidence that the algorithm is correct and would produce the correct solution for every possible input instance.

Please run the following code cell to check if your code passes some corner cases.

Code Cell 6 of 22 - Hidden Code

Run Code

Out [6] Testing your code...
 All tests have completed successfully! Excellent work!

Code Cell 7 of 22

In [7] 1 **### PLEASE DO NOT USE THIS CELL, IT WILL BE USED FOR FURTHER TESTING**

Run Code

Question 5 of 10

D. Design experiments to determine the average run time and number of steps that both implementations take for increasing values of n . Consider n to be smaller than 20. Which implementation is better and why? (Please refrain from quoting/deriving bigO results as the question merely asks you to focus on the experimental analysis from your specific implementations).

Normal  **B** *I* U        A

I have chosen average runtime and step counter as my main metrics to evaluate the performance of each algorithm. In this experiment, I defined a step as a mathematical operation (add, subtract, multiply, and divide) or addition of numbers to the sides of the middle values. This can be a reasonable choice given the purpose and structure of the algorithms (adding numbers from the sides is the method the algorithms used to produce strobogrammatic numbers). For both approaches, I ran both iterative and recursive algorithms 10 times for each of the input sizes (from 0 to 19), averaging the runtimes and number of steps for each input size to mitigate the potential effect of the outlier results. I visualized these results and used the plots to compare the efficiency of both algorithms.

For the runtime, we can see that for smaller input sizes both algorithms perform similarly, but as the input size grows larger (after $n=16$) the runtime for the iterative algorithm starts to grow faster than for the recursive one, suggesting that the recursive algorithm is more efficient and a better implementation based on the runtime metric. However, we can also see that the order of growth of the runtime as the input size grows larger is similar for both algorithms (exponential), suggesting they have similar efficiency.

For the step counter, we can see that the number of steps required to solve a problem is very similar for both algorithms for all input sizes, with the recursive algorithm taking slightly fewer steps on average. This shows that the efficiency of both algorithms is very similar.

Code Cell 8 of 22

```
In [8] 1 import time
      2 import numpy as np
      3 av_times_it=[]
      4 av_times_rec=[]
      5 #run the experiment for each of the input size n from 0 to 19
      6 for i in range(20):
      7     times_it=[]
      8     times_rec=[]
      9     #repeat the same experiment for each input size 10 times to average the time
     10     for j in range(10):
     11
     12         start=time.time() #record start time
     13         strobogrammatic_iterative(i)
     14         end=time.time() #record end time
     15         times_it.append(end-start) #store the runtime of the iterative algorithm each time we run it for the same input size
     16
     17
```

Python 3 (384MB RAM) | Edit

Run All Cells

Kernel Ready |

```

18     start=time.time() #record start time
19     strobogrammatic_recursive(i)
20     end=time.time() #record end time
21     times_rec.append(end-start) #store the runtime of the recursive algorithm each time we run it for the same input
size
22
23     #store the average runtime for each input size
24     av_times_it.append(np.mean(times_it))
25     av_times_rec.append(np.mean(times_rec))
26

```

Run Code

Code Cell 9 of 22

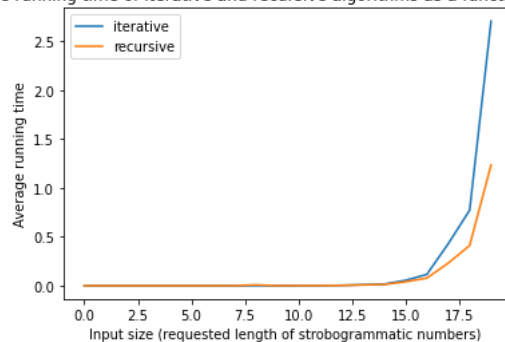
```

In [9] 1 import matplotlib.pyplot as plt
2 #plot both recursive and iterative algorithms runtimes (from previous code cell) on the same graph
3 plt.plot([i for i in range(20)],av_times_it, label='iterative')
4 plt.plot([i for i in range(20)],av_times_rec,label='recursive')
5 plt.xlabel('Input size (requested length of strobogrammatic numbers) ')
6 plt.ylabel('Average running time')
7 plt.title('Average running time of iterative and recursive algorithms as a function of input size')
8 plt.legend()
9 plt.show()

```

Run Code

Out [9] Average running time of iterative and recursive algorithms as a function of input size



Code Cell 10 of 22

```

In [10] 1 def strobogrammatic_iterative_steps(n):
2     '''
3     Return the number of steps it takes for the iterative algorithm to reach the solution
4     -----
5     Parameters:
6     n: int
7         The targeted length of the digit
8     -----
9     Returns: steps
10         The number of steps performed
11     '''
12     steps=0 #initialize the step counter
13

```

```

14     #check that the input is a positive integer
15     if type(n)!=int or not n>=0:
16         return 'Invalid input'
17
18     #check if the number is odd or even to determine the middle values
19     if n%2!=0:
20         steps+=1 #count the deviation by 2 as a step
21         result=["0", "1", "8"]
22     else:
23         result=['']
24
25     #keep adding numbers from the sides untill we reach the required length n
26     for i in range(n//2):
27         mid=[]
28         #add numbers to each middle value stored in result list
29         for j in result:
30             #add 0s on the sides only if the number will not end up starting with 0
31             if i<(n//2)-1:
32                 steps+=1 #count addition of 0s on the sides as a steps
33                 mid.append("0" + j + "0")
34
35             #adding the rest of strobogrammatic pairs of numbers from the sides and storing in mid list
36             steps+=4 #count addition of the rest of number pairs as 4 steps
37             mid.append("1" + j + "1")
38             mid.append("6" + j + "9")
39             mid.append("8" + j + "8")
40             mid.append("9" + j + "6")
41
42         result=mid #update the results list with new middle values
43
44     return steps
45
46
47
48 def strobogrammatic_recursive_steps(n):
49     '''
50     Return the number of steps it takes for the recursive algorithm to reach the solution
51     -----
52     Parameters:
53     n: int
54         The targeted length of the digit
55     -----
56     Returns: steps
57         The number of steps performed
58     '''
59     #check that the input is a positive integer
60     if type(n)!=int or not n>=0:
61         return 'Invalid input'
62
63     def helper(n,length,steps=0):#initialize steps as an argument in the helper function to avoid resetting it to 0 every
recursion
64         '''
65         Return the number of steps it takes for the recursive algorithm to reach the solution
66         -----
67         Parameters:
68         n: int
69             The targeted length of the number
70         length : int
71             The targeted length of the number
72         -----
73         Returns: steps

```



```

75     '''
76
77     #define base cases to stop recursion
78     if n==0:
79         return ([''], steps)
80     if n==1:
81         return (["0", "1", "8"], steps)
82
83     result=[]
84     #create middle values by recursing the function for n-2 until we hit a base case
85     mid=helper(n-2,length,steps)[0] #assign to mid variable only the middle values list, not the step counter
86
87     #for each middle value in the list, add defined pairs of strobogrammatic numbers and store in result list
88     for j in mid:
89         #add 0s on the sides only if the number will not end up starting with 0
90         if len(j)<length-2:
91             steps+=1 #count addition of 0s on the sides as a steps
92             result.append("0" + j + "0")
93
94             steps+=4 #count addition of the rest of number pairs as 4 steps
95             result.append("1" + j + "1")
96             result.append("6" + j + "9")
97             result.append("8" + j + "8")
98             result.append("9" + j + "6")
99
100         return (sorted(result),steps) #return both the result list and steps variable
101
102     return helper(n,n)[1] #return only the last element in the helper function output - steps counter
103
104
105
106 av_steps_it=[]
107 av_steps_rec=[]
108 #run the experiment for each of the input size n from 0 to 19
109 for i in range(20):
110     steps_it=[]
111     steps_rec=[]
112     #repeat the same experiment for each input size 10 times to average the number of steps
113     for j in range(10):
114
115         #store the number of steps of both algorithms each time we run them for the same input size
116         steps_it.append(strobogrammatic_iterative_steps(i))
117         steps_rec.append(strobogrammatic_recursive_steps(i))
118
119     #store the average number of steps for each input size
120     av_steps_it.append(np.mean(steps_it))
121     av_steps_rec.append(np.mean(steps_rec))
122

```

Run Code

Code Cell 11 of 22

```

In [11] 1 #plot the average number of steps for each algorithm for each inout size up to 19
2 plt.plot([i for i in range(20)],av_steps_it, label='iterative')
3 plt.plot([i for i in range(20)],av_steps_rec,label='recursive')
4 plt.xlabel('Input size (requested length of strobogrammatic numbers)')
5 plt.ylabel('Average number of steps')
6 plt.title('Average number of steps of iterative and recursive algorithms as a function of input size')

```

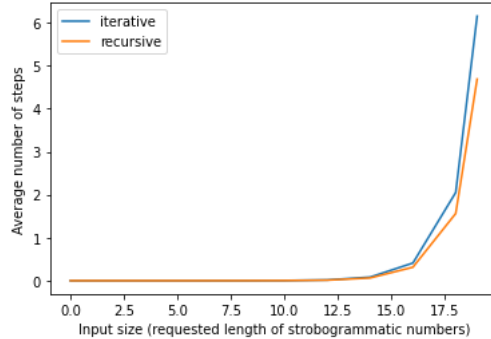
```

7 plt.legend()
8 plt.show()

```

Run Code

Out [11] Average number of steps of iterative and recursive algorithms as a function of input size



Question 2

You are writing an algorithm for a post office of your rotation city that will send trucks to different houses in the city. The post office wants to minimize the number of trucks needed, so all the homes with the same postal code should only require one truck.

The houses' postal codes are stored in a sorted array. Many places can share the same postal code. To simplify the database, you have implemented a dictionary that maps the postal codes (that can also contain letters) to more specific numbers. For example, 94102 has been mapped to 1.

Given a **target** postal code, your task is to return the beginning and end index of this **target** value in the array (inclusive) and the total number of postal codes. If this postal code is not in the database, return **None**.

For example, given the postal array `[1, 2, 2, 3, 4, 4, 4, 5, 5, 5, 5, 7]`, and our target 4, the answer should be `[4, 6, 3]`, where 4 is the start-index, 6 is the end-index, and 3 is the number of postal codes of this value. If our target is 1 for the same postal array, the answer should be `[0, 0, 1]`. If the target is 6, the answer should be **None**.

Question 6 of 10

A. Explain how you would design two approaches (an iterative and a recursive one) to solve this computational problem in plain English. Avoid using technical jargon (such as "indices" and while/for loops), and focus on explaining in as simple terms as possible how the algorithm works and how it is guaranteed, upon termination, to return the correct answer.

Normal ⌵ **B** *I* U **A**

Iterative approach:

We start by checking if there is a target number in the given list of numbers at all. If not, there is no point in doing any operations and we stop. Otherwise, we would go through each number in the list and check if it is a target number. If it is, we record its index. As we go through the whole list, we count how many target numbers we found and keep recording their corresponding indexes. Since we are going through the list from left to right, that is in the correct order, the first index recorded would correspond to the first occurrence of the target number in the list, and the last index to the last target number. Thus, we are guaranteed to find the correct solution: the first index, the last index, and the number of target numbers in the list.

For example,

we have the list `[1,2,3,3,4]` and the target number is 3. We start by checking each number in the list. The first number is 1, which is not a target number 3, so we go to the next number. The next number is 2, which is again not 3, meaning we move to the next number. This number is 3, which is a target number, so we record its index - 3 (or 2 in python notation), and record that we found 1 target number so far. Moving to the next number, we find that it is again 3. We record its index - 4 (or 3 in python notation), making the index list `[3,4]`, and record that we already found 2 target numbers. Finally, we move to the last number, which is 4 and not the target number. We finish the algorithm because there is no more number in the list to check. To find the first index, we take the first number in the index list, which is 3, and for the last index, we take the last

Recursive approach:

We start by checking if there is a target number in the given list of numbers at all. If not, there is no point in doing any operations and we stop. Otherwise, we define a condition that would terminate the algorithm if satisfied:

1. If the lower bound of our search interval becomes bigger than the upper bound, then stop the algorithm.

This condition can be viewed as the smallest subproblem to which we know the solution. The idea of this algorithm is then to take the problem and break it down into smaller subproblems until the subproblem is the one we identified above. To reach this subproblem, we would repeat the algorithm over and over, each time updating the upper or lower search interval to narrow it down. This way, we are able to reach the subproblem for which we can produce a solution. When we do reach them, we would be able to combine the solutions for these subproblems back to find the answer to the original problem by performing a series of operations on the answers to the subproblems.

We start by establishing the lower and upper bounds for the search interval, which are the beginning and end of the list we are searching in. If the lower bound is smaller than the upper bound, we can proceed further. We then start picking numbers in the middle of the search interval. After picking the number, we have 5 conditions to check:

1. Is our choice less than (to the left of) the target number?
2. Is our choice bigger than (to the right of) the target number?
3. Is our choice a target number?
 - a. If yes, is the number to the left of our choice also a target number?
 - b. If yes, is there still a number to the right of our choice and is this number also a target number?

If the number we picked is smaller than the target, we repeat the guessing process above on the narrower interval by setting the lower bound to be the next number after (to the right of) our previous guess. If not, move to the next step.

If the number we picked is bigger than the target, we repeat the guessing process above on the narrower interval by setting the upper bound to be the number before (to the left of) our previous guess. If not, move to the next step.

If the number we picked is the target, and the number before (to the left of) our guess is also the target, we repeat the guessing process above on the narrower interval by setting the upper bound to be the number before (to the left of) our previous guess. If not, move to the next step.

If the number we picked is the target, the number after (to the right of) our guess is also the target, and the number after our guess exists in the list, we repeat the guessing process above on the narrower interval by setting the lower bound to be the number after (to the right of) our previous guess.

If the number we picked is the target, but none of the last 2 conditions above are satisfied, we record the position of the number we picked in the list.

After we recorded it, we produce the output, which is the smallest recorded position of the target number, the biggest, and the number of target number occurrences we found, which is the same as the number of recorded positions.

As we can see, we immediately move to the next step if the previous is not satisfied. If it is satisfied, however, one should note that we still move to the next step but only after we repeat the algorithm as many times as possible until either the lower bound becomes larger than the upper one or we produce an output. Then we use this result (either nothing or the recorded positions and a number of target number occurrences) to move to the next step. The algorithm stops when all the steps are completed.

It is guaranteed to return the correct solution because it decomposes the problem into subproblems by narrowing down the search interval and combining the solutions for subproblems into the solution for the whole problem. It also contains conditions that would stop the algorithm from repeating indefinitely.

For example,

1. The list is [1,2,2,3] and the target number is 2. We check if the target number is in the list. Since there is a 2 in the list, we set the lower and upper bounds to be 0 and 3 respectively (positions start from 0). Since the lower bound is smaller than the upper bound ($0 < 3$), we pick a number in the middle, which is in position 1 (we sum the lower and upper bounds, divide by 2, and take the integer of the result : $(0+3)/2=1.5$, use 1 as a middle position). The number we picked at position 1 is 2. We first check if it is smaller than the target number. Since it is not, we check if it is bigger than the target number. Since it is not, we check if it is a target number. It is. We then check if the number to the left of it, which is 1, is a target number. It is not, which means we check if the number to the right of it is the target number and if it exists in the list. It does exist and it is also a target number, which means we repeat the algorithm with the higher bound staying the same - 3, and the lower bound being updated to the next number after the middle number we picked before - the next number after 2 (position 1) is 2 (position 2).

2. Since the lower bound is still smaller than the upper bound ($2 < 3$), we pick a number in the middle, which is in position 2: $(2+3)/2=2.5$, using 2 as a middle position. The number we picked at position 2 is 2. We first check if it is smaller than the target number. Since it is not, we check if it is bigger than the target number. Since it is not, we check if it is a target number. It is. We then check if the number to the left of it, which is 2, is a target number. It is, and we repeat the algorithm with the lower bound 2 and the upper bound 1 (middle position 2 is 1).
3. We check that the upper bound is now smaller than the lower bound, which means the output for this subproblem is nothing. We go back to the previous time we ran the algorithm (step 2 above) with nothing in the output and continue with the rest of the steps. Since the last thing we did was check if the previous number was a target, now what we have left is to check if the next number after our choice (2 at position 2) exists and is a target. It exists, but it is not the target, which means we completed all steps, can record the current middle position, which is 2, and produce the output: smallest index is 2, biggest is 2, and we found 1 target number so far.
4. We use this solution to this subproblem to return back to the first time we started the algorithm (step 1). Since the last thing we did was check if the next number from the choice number was the target, there are no more steps left, and we recorded the middle index at that step, which was 1, and produced the output: smallest index is 1, biggest is 2, and we found 2 target number so far.

Since we completed all the steps and subproblems, our algorithm terminates with the correct solution - [1,2,2].

Question 7 of 10

B. Using your descriptions from question 2 A, produce two flowcharts that correctly describe each approach. Please be particularly careful about describing the termination condition for both algorithms. Upload your flowcharts below.

Question 2 Flowchart.pdf (251 kB) ×

Drop or [upload](#) a file here

C. Provide recursive and iterative Python implementations that return a list with three elements (start-index, end-index, and number of matches found in the sorted array).

Remember to add at least 3 test cases to demonstrate that your function is correctly implemented in Python.

You will need to justify why those specific test cases are appropriate and possibly sufficient.

Code Cell 12 of 22

```
In [12] 1 def iterative_binary_search_extension(nums, target):
2     '''
3     Return the start and end index (inclusively)
4     of a target element in a sorted array
5     -----
6     Parameters:
7     nums: list
8         a sorted array
9     target: int
10        the target value to be found
11    -----
12    Returns:
13    list/ None
14        list of the beginning index, end index and total number of indexes
15        if found, else None
16    '''
17    #checks if the target number is in the list at all
18    if target not in nums:
19        return None
20    else:
21        count=0
22        idx_list=[]
23        #iterates through every number in the list to count how many target numbers there are and what their indexes
        (ordered) are
```

```

25         #if we find the target number,count it and store its index
26         if nums[i]==target:
27             count+=1
28             idx_list.append(i)
29     return [idx_list[0], idx_list[-1],count] #return first index, last index, and number of target numbers found
30
31 def recursive_binary_search_extension(nums, target):
32     '''
33     Return the start and end index (inclusively)
34     of a target element in a sorted array
35     -----
36     Parameters:
37     nums: list
38         a sorted array
39     target: int
40         the target value to be found
41     -----
42     Returns:
43     list/ None
44         list of the beginning index, end index and total number of indexes
45         if found, else None
46     '''
47     #checks if the target number is in the list at all
48     if target not in nums:
49         return None
50
51 def binarySearch(nums,target,low,high,indices=[]):
52     '''
53     Return the start and end index (inclusively)
54     of a target element in a sorted array
55     -----
56     Parameters:
57     nums: list
58         a sorted array
59     target: int
60         the target value to be found
61     low:int
62         lower index of our search interval
63     high:int
64         higher index of our search interval
65     indices: list
66         stores indexes of found target numbers
67     -----
68     Returns:
69     list/ None
70         list of the beginning index, end index and total number of indexes
71         if found, else None
72     '''
73     #define a base case to stop recursion when lower index is bigger than higher index
74     if low > high:
75         return None
76     else:
77         #define the middle index for binary search to always make guesses in the middle of the interval
78         mid = (low + high) // 2
79
80         if target > nums[mid]: #if target is on the right side from the guess, update the lower bound of the interval
81             binarySearch(nums, target, mid + 1, high,indices) #lower bound becomes the number bigger than our guess by
82             one
83
84         elif target < nums[mid]: #if target is on the left side from the guess, update the higher bound of the interval
85             binarySearch(nums, target, low, mid - 1,indices) #upper bound becomes the number smaller than our guess by

```

```

85
86         else: #if our guess is the target number
87             #if the number to the left of our guess is also a target number, run the algorithm again with updated higher
bound
88             if target==nums[mid-1]:
89                 binarySearch(nums, target, low, mid - 1, indices) #upper bound becomes the number smaller than our guess
by one
90
91             #if the number to the right of our guess is also a target number, run the algorithm again with updated lower
bound
92             if mid+1 <= high and target==nums[mid+1]:
93                 binarySearch(nums, target, mid+1, high, indices) #lower bound becomes the number bigger than our guess by
one
94
95             indices.append(mid) #add the index of the target number to the list
96
97         return [min(indices),max(indices),len(indices)]
98     return binarySearch(nums,target,0, len(nums)-1)

```

Run Code

Code Cell 13 of 22

```

In [13] 1 #test cases for the iterative algorithm using various types of inputs
2 assert(iterative_binary_search_extension([], 5)==None )
3 assert(iterative_binary_search_extension([1], 1)==[0,0,1])
4 assert(iterative_binary_search_extension([1,1], 1)==[0,1,2])
5 assert(iterative_binary_search_extension([1,2,3,4,5], 6)==None)
6 assert(iterative_binary_search_extension([1,2,3,4,4,5], 4)==[3,4,2])
7 assert(iterative_binary_search_extension([-1,2,3,4,5], -1)==[0,0,1])
8 assert(iterative_binary_search_extension([-1,2,3,4,5,6,7,7,7,8,9,23,54,54,60], 7)==[6,8,3])
9 print('Passed all cases')

```

Run Code

Out [13] Passed all cases

Code Cell 14 of 22

```

In [14] 1 #test cases for the recursive algorithm using various types of inputs
2 assert(recursive_binary_search_extension([], 5)==None )
3 assert(recursive_binary_search_extension([1], 1)==[0,0,1])
4 assert(recursive_binary_search_extension([1,1], 1)==[0,1,2])
5 assert(recursive_binary_search_extension([1,2,3,4,5], 6)==None)
6 assert(recursive_binary_search_extension([1,2,3,4,4,5], 4)==[3,4,2])
7 assert(recursive_binary_search_extension([-1,2,3,4,5], -1)==[0,0,1])
8 assert(recursive_binary_search_extension([-1,2,3,4,5,6,7,7,7,8,9,23,54,54,60], 7)==[6,8,3])
9 print('Passed all cases')

```

Run Code

Out [14]

Passed all cases

Code Cell 15 of 22

In [14]

1

Run Code

Code Cell 16 of 22

In [15]

1

YOUR CODE HERE

Run Code

Question 8 of 10

Why are your test cases appropriate or possibly sufficient?

Normal

⌵

B

I

U

🔗

”

⌵

☰

☰

☰

☰

A

My test cases are appropriate and possibly sufficient because they encompass possibly all potential input categories. In particular, the test cases include inputs where the target number is present in the array, not present in the array, the length of the array is 0, 1, 2, or bigger, as well as arrays including negative numbers. If the algorithms pass all of these test cases, it would give us a lot of confidence that the algorithm is correct and would produce the correct solution for every possible input instance.

Please run the following code cell to check if your code passes some corner cases.

Code Cell 17 of 22 - Hidden Code

Run Code

Out [16]

Testing your code...

🔍 All tests have completed successfully! Excellent work!

Code Cell 18 of 22

In [17]

1

PLEASE DO NOT USE THIS CELL, IT WILL BE USED FOR FURTHER TESTING

Run Code

Question 9 of 10

D. Design several experiments to determine which of these formulations are better and why. You may want to consider:

- different edge cases for the input, and
- different metrics to determine which algorithm seems better

Be as creative as you would like!

Normal  **B** *I* U        

I have chosen average runtime and step counter as my main metrics to evaluate the performance of each algorithm. In this experiment, I defined a step as a mathematical operation (add, subtract, multiply, and divide) or comparison of a selected number to a target number. This can be a reasonable choice given the purpose and structure of the algorithms: the goal is to find the target numbers from the list of numbers. For both approaches, I ran both iterative and recursive algorithms 10000 times for each of the input sizes (from 0 to 49), averaging the runtimes and number of steps for each input size to mitigate the potential effect of the outlier results. For each input size, I generated random lists of the corresponding lengths and a random target number. I used the same inputs for both iterative and recursive approaches to be able to better compare their performance. I visualized these results and used the plots to compare the efficiency of both algorithms.

For the runtime, we can see that for smaller input sizes both algorithms perform similarly, but as the input size grows larger (after $n=20$) the runtime for the iterative algorithm starts to grow faster than for the recursive one, suggesting that the recursive algorithm is more efficient and a better implementation based on the runtime metric. However, we can also see that the order of growth of the runtime as the input size grows larger is similar for both algorithms (linear), suggesting they have similar efficiency.

For the steps count, we can see that for smaller input sizes both algorithms perform similarly, but as the input size grows larger (after $n=10$) the number of steps for the iterative algorithm starts to grow faster than for the recursive one, suggesting that the recursive algorithm is more efficient and a better implementation based on the steps count metric.

Code Cell 19 of 22

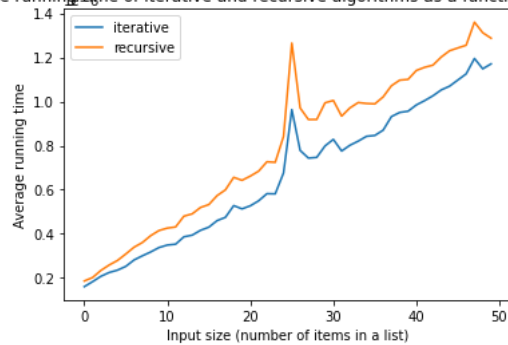
```
In [18] 1 import random
        2 import time
        3 import matplotlib.pyplot as plt
        4 import numpy as np
        5
        6 av_times_it=[]
        7 av_times_rec=[]
        8 #run the experiment for each of the input size n from 0 to 19
        9 for i in range(50):
       10     times_it=[]
       11     times_rec=[]
       12     #repeat the same experiment for each input size 10000 times to average the time
       13     for j in range(10000):
       14         randomlist = sorted(random.choices(range(-50, 50), k=i))
       15         target=random.randint(-50,50)
       16
       17         start=time.time() #record start time
       18         iterative_binary_search_extension(randomlist, target) #run the iterative algorithm
       19         end=time.time() #record end time
       20         times_it.append(end-start) #store the runtime of the iterative algorithm each time we run it for the same input size
       21
       22         start=time.time() #record start time
       23         recursive_binary_search_extension(randomlist, target) #run the recursive algorithm
       24         end=time.time() #record end time
       25         times_rec.append(end-start) #store the runtime of the recursive algorithm each time we run it for the same input
       26         size
       27
       28     av_times_it.append(np.mean(times_it)) #store the average runtime for each input size
       29     av_times_rec.append(np.mean(times_rec)) #store the average runtime for each input size
       30
       31
       32 #plot both recursive and iterative algorithms runtimes on the same graph
       33 plt.plot([i for i in range(50)],av_times_it, label='iterative')
       34 plt.plot([i for i in range(50)],av_times_rec,label='recursive')
       35 plt.xlabel('Input size (number of items in a list)')
       36 plt.ylabel('Average running time')
       37 plt.title('Average running time of iterative and recursive algorithms as a function of input size')
```



```
39 plt.show()
```

Run Code

Out [18] Average running time of iterative and recursive algorithms as a function of input size



Code Cell 20 of 22

```
In [19] 1 def iterative_binary_search_steps(nums, target):
2     '''
3     Return the number of steps it takes to reach the solution
4     -----
5     Parameters:
6     nums: list
7         a sorted array
8     target: int
9         the target value to be found
10    -----
11    Returns:
12    steps: integer
13        total number of steps to run an algorithm and reach a solution
14    '''
15
16    steps=0
17
18    #checks if the target number is in the list at all, if not - terminates the algorithm
19    if target not in nums:
20        return steps
21    else:
22        count=0
23        idx_list=[]
24        #iterates through every number in the list to count how many target numbers there are and what their indexes
25        # (ordered) are
26        for i in range(len(nums)):
27            steps+=1 #count checking one number as one step
28            #if we find the target number, count it and store its index
29            if nums[i]==target:
30                count+=1
31                idx_list.append(i)
32
33    return steps + len(nums) #return the total number of steps performed, including checking if the target is in the list
34
35 def recursive_binary_search_steps(nums, target):
36     '''
37     Returns the number of steps it takes to reach the solution
38     -----
39     Parameters:
40         a sorted array
41     target: int
42         the target value to be found
43     -----
44     Returns:
45     steps: integer
46         total number of steps to run an algorithm and reach a solution
47     '''
48
49     steps=0
50
51     #checks if the target number is in the list at all, if not - terminates the algorithm
52     if target not in nums:
53         return steps
54     else:
55         count=0
56         idx_list=[]
57         #iterates through every number in the list to count how many target numbers there are and what their indexes
58         # (ordered) are
59         for i in range(len(nums)):
60             steps+=1 #count checking one number as one step
61             #if we find the target number, count it and store its index
62             if nums[i]==target:
63                 count+=1
64                 idx_list.append(i)
65
66     return steps + len(nums) #return the total number of steps performed, including checking if the target is in the list
```

```

39     nums: list
40         a sorted array
41     target: int
42         the target value to be found
43     -----
44     Returns:
45     steps: integer
46         total number of steps to run an algorithm and reach a solution
47     '''
48
49     #checks if the target number is in the list at all
50     if target not in nums:
51         return 0
52
53     def binarySearch(nums,target,low,high,indices=[],steps=0):
54         '''
55         Returns the number of steps it takes to reach the solution
56         -----
57         Parameters:
58         nums: list
59             a sorted array
60         target: int
61             the target value to be found
62         low:int
63             lower index of our search interval
64         high:int
65             higher index of our search interval
66         indices: list
67             stores indexes of found target numbers
68         steps : int
69             count the number of steps it takes for the algorithm to reach the solution
70         -----
71         Returns:
72         steps: integer
73             total number of steps to run an algorithm and reach a solution
74         '''
75
76         #define a base case to stop recursion when lower index is bigger than higher index
77         if low > high:
78             return None
79         else:
80             #define the middle index for binary search to always make guesses in the middle of the interval
81             mid = (low + high) // 2
82             steps+=1 #count finding the middle as a step
83
84             if target > nums[mid]: #if target is on the right side from the guess, update the lower bound of the interval
85                 steps+=1 #count a comparison as a step
86                 binarySearch(nums, target, mid + 1, high,indices,steps) #lower bound becomes the number bigger than our
guess by one
87
88             elif target < nums[mid]: #if target is on the left side from the guess, update the higher bound of the interval
89                 steps+=1 #count a comparison as a ste
90                 binarySearch(nums, target, low, mid - 1,indices,steps) #upper bound becomes the number smaller than our
guess by one
91
92             else: #if our guess is the target number
93                 #if the number to the left of our gues is also a target number, run the algorithm again with updated higher
bound
94                 if target==nums[mid-1]:
95                     steps+=1 #count a comparison as a step
96                     binarySearch(nums, target, low, mid - 1,indices,steps) #upper bound becomes the number smaller than our
guess by one

```

```

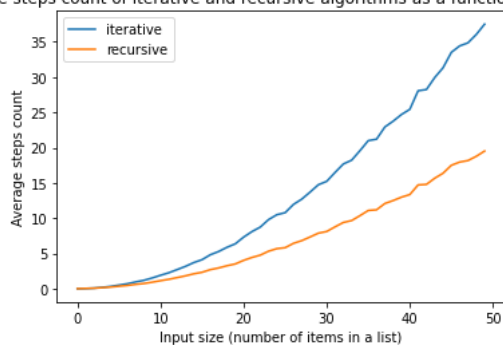
98         #if the number to the right of our guess is also a target number, run the algorithm again with updated lower
bound
99         if mid+1 <= high and target==nums[mid+1]:
100             steps+=1 #count a comparison as a step
101             binarySearch(nums, target, mid+1, high, indices, steps) #lower bound becomes the number bigger than our
guess by one
102
103             indices.append(mid) #add the index of the target number to the list
104
105         return steps
106     #return the steps (add the steps it takes to check the very first condition)
107     return binarySearch(nums, target, 0, len(nums)-1) + len(nums)
108
109
110 av_steps_it=[]
111 av_steps_rec=[]
112 #run the experiment for each of the input size n from 0 to 49
113 for i in range(50):
114     steps_it=[]
115     steps_rec=[]
116     #repeat the same experiment for each input size 10000 times to average the number of steps
117     for j in range(10000):
118         #for each experiment, generate random lists of integers of a given size and random target number
119         randomlist = random.choices(range(-50, 50), k=i)
120         target=random.randint(-50,50)
121
122         #run the iterative algorithm and store its steps
123         steps_it.append(iterative_binary_search_steps(sorted(randomlist), target))
124
125         #run the recursive algorithm and store its steps
126         steps_rec.append(recursive_binary_search_steps(sorted(randomlist), target))
127
128     av_steps_it.append(np.mean(steps_it)) #store the average step count for each input size
129     av_steps_rec.append(np.mean(steps_rec)) #store the average step count for each input size
130
131
132 #plot both recursive and iterative algorithms step count on the same graph
133 plt.plot([i for i in range(50)],av_steps_it, label='iterative')
134 plt.plot([i for i in range(50)],av_steps_rec,label='recursive')
135 plt.xlabel('Input size (number of items in a list)')
136 plt.ylabel('Average steps count')
137 plt.title('Average steps count of iterative and recursive algorithms as a function of input size')
138 plt.legend()
139 plt.show()

```

Run Code

Out [19]

Average steps count of iterative and recursive algorithms as a function of input size



Code Cell 21 of 22

In [19] 1

Run Code

Code Cell 22 of 22

In [20] 1 ### YOUR CODE HERE

Run Code

Question 10 of 10

References

Please write here all the references you have used for your work

Normal ⌵ B I U 🔗 “ ” </> ☰ ☷ ≡ ≡ A

🏁 You are all done! Congratulations on finishing your first CS110 assignment!

🎉