Tommy Betz - Algorithm Problem - May 30, 2020

1) Given an array of non-negative numbers, give a naive algorithm with time complexity $O(n^2)$ to find a continuous subarray with sum of S.

Since we are trying to achieve a runtime of $O(n^2)$ this means an algorithm with a nested loop is required. Because the values of the subarray are continuous sums of S, there is no need to sum non-neighboring values of S. This limits the possible number of combinations tremendously. Furthermore, because we do not include negative numbers, we can stop calculating the sum once the sum becomes larger than the target sum.

Assumptions

- the algorithm is supposed to stop once S is found, therefore eliminating the need to find all subarrays adding up to S.
- values in array can be non-integers <-- does not matter for this algorithm
- values in array are not sorted <-- does not matter for this algorithm
- Duplicate values within the array are allowed

Instead of a pseudocode, I have written the algorithm in Python3 along with comments. I hope that is ok.

Input

- array = [10, 2, 13, 3, 15, 5, 6, 18]
- S = 11

Output

- subarray = [5, 6]

Runtime

- $O\left(n^2\right)$

Code

```
array = [10, 2, 13, 3, 15, 5, 6, 18] # example array
S = 11 # example target sum

# first and outer loop through n
# we run through each value of the array as the start of the subarray
for index_start in range(len(array)):
    index_end = index_start # at first we set the ending index to be the same as the starting index which will be changed in the second loop below

    # float_sum contains the sum of the current subarray and will change with each iteration of the outer and inner loop
    # the value of float_sum starts off with current starting index of the array
    # float_sum = current sum
    float_sum = array[index_start]

    # we run this loop as long as the ending index is within a valid range, and the current sum is smaller to the target sum
    while (index_end < len(array)-1) and (float_sum < S): # second loop through n
        index_end += 1 # since the current sum is not equal to the target sum, we include one more value of the array
        float_sum += array[index_end] # updating the current sum to include one more value of the array

    if float_sum == S: # once the current sum is equal to the target sum, we can exit the loop as we have found a valid subarray
        break

if float_sum == S:
    print("subarray:", array[index_start:index_end+1])
else:
    print("no subarray with sum of S found")
```

2) Can you find a more efficient solution? What is the time complexity of this solution?

The above algorithm is inefficient because we start the calculations all over again after each iteration of the outer loop. A more efficient approach would be to move over the sum from the previous iteration, and subtract the first value of the array, thus creating a sliding window.

Assumptions

- the same assumptions as in 1)

Input

- array = [10, 2, 13, 3, 15, 5, 6, 18]
- S = 26

Output

- subarray = [15, 5, 6]

Runtime

1. At first the algorithm will run through all values of the array (n) until the sum of the subarray is larger than S. So far, the runtime is $O(n)$
2. Then the sliding window logic will pick up at the previous iteration but will shift the starting index by 1 and increase the end index until the sum of the subarray is larger again then S.
3. The previous step 2 will be repeated until either a valid subarray is found, or we reached the end of the array.
4. Step 2 and 3 add up to a runtime of $O(n)$
5. Total runtime of step 1, 2, and 3: $O(n) + O(n) = O(2n) = O(n)$

Code

On the next page

2) Code

```
array = [10, 2, 13, 3, 15, 5, 6, 18] # example array
S = 1050 # example target sum
index_start = 0 # starting index
index_end = 0 # ending index

# the value of the current sum (float_sum) starts off with the first value of the array
float_sum = array[index_start]

# we run this loop as long as the index_end is within valid range, and float_sum is not equal to the target sum (S)
while (index_start <= index_end < len(array)) and (float_sum != S):

    # if the current sum is larger than the target sum
    if float_sum > S:
        # if the starting index is the same as the ending index, meaning the subarray consists of only 1 value
        if index_start == index_end:
            # we shift the starting and ending index by 1, and slide the current sum 1 over in the array
            index_start += 1
            index_end += 1
            float_sum = array[index_start]

        # if the starting index is not the same as the ending index, meaning the subarray consists of more than 1 value
        else:
            # we subtract the most left value in the subarray of the current sum, and shift the starting index 1 to the left
            float_sum -= array[index_start]
            index_start += 1

    # if the current sum is smaller than the target sum
    elif float_sum < S:
        # if the ending index is within valid range, we increase the subarray by 1, and we increase the current sum by the new value of the subarray
        if (index_end + 1) < len(array):
            index_end +=1
            float_sum += array[index_end]
        # if the ending index is no longer withing valid range, that means we were not able to find a subarray, and will exit the loop
        else:
            break

if float_sum == S:
    print("subarray:", array[index_start:index_end+1])
else:
    print("no subarray with sum of S found")
```

3) What approach would you take to the problem if we removed the constraints of the numbers being non-negative and the subarray being continuous? What would be the time complexity of a solution using this approach?

Since the subarray no longer has to be continuous, this means that we can no longer use a purely sliding method. Furthermore, because we now include negative numbers, we can no longer stop the sum calculation if the sum becomes larger than the target sum. The sum calculation has to continue as any negative element could decrease the sum and eventually be equal to the target sum.

A naive approach could be to run through all possible combinations of the array and sum all elements in each combination.

Runtime

- Run through each combination: $O\left(2^n\right)$ <-- approximately
- Sum all elements in each combination: at most N
- Total: $O\left(2^n N\right)$

However, there is no need to sum each combination separately. Instead, whenever we increase a combination by one element, we can just add the new element to the sum of the previous combination. The sum for each combination could be stored in a dictionary with the individual elements' index concatenated as a string, ready to be looked up whenever a new element is added. In my code below I did not store any sums but kept passing the results within a list to the function. A recurssive function would have worked as well.

I am not sure how to translate these efficiencies in O-notation but there would be many fewer iterations which could decrease the overall runtime considerably.

Most of my code below is about making sequential combinations. However, in a regular script I would just use a library such as itertools and customize it to check whether the current sum is equal to the target sum. This way we can stop the script early in case a subarray with the target sum is found.

Since I explained the overall logic of my code above, I did not fully comment the following code. I hope that is ok.

Assumptions

- the same assumptions as in 1)

Input

- array = [4, -3, 6, -2, 7]
- S = 8

Output

- subarray = [4, -3, 7]

Code

On the next page

3) Code

```
def get_combinations(array_original, array, array_elements, int_arr_len, S, bol_S_found):

    # helper variables/lists
    arr_combinations = []
    arr_combinations_elements = []
    int_count_sitout = 0
    bol_sitout = False
    int_run_len = 0

    # if the maximum length of the combination is 1, then just return the list as is
    if int_arr_len == 1:
        for element in array:
            arr_combinations.append(element)
            arr_combinations_elements.append(str(element))

            if element == S:
                bol_S_found = True
                break

        return arr_combinations, arr_combinations_elements, bol_S_found

    # defining the settings for the additional elements to add to the previous combinations
    array_help = array_original.copy()[int_arr_len-1:]
    len_help = len(array_help)
    help_remaining = len_help

    # depending on the length of the possible combinations, different settings apply
    if int_arr_len == len(array):
        if len(array_original) < 3:
            int_run_len = -1
        else:
            int_run_len = -2
    elif int_arr_len == 1:
        int_run_len = len(array)
    else:
        int_run_len = (len(array_original)-len(array_help)) *-1
```

3) Code - continued

```
    # run through each element of the array
    for index_element in range(len(array[:int_run_len])):

        # add new elements to the previous combinations in the array
        for element_help in array_help:
            if (int_count_sitout == 0) and (help_remaining>0) and (bol_S_found==False):
                result = array[:int_run_len][index_element] + element_help

                arr_combinations.append(result)
                arr_combinations_elements.append(str(array_elements[:int_run_len][index_element]) + ", " +
str(element_help))

                if result == S:
                    bol_S_found = True
                    break

            else:
                bol_sitout = True

            if len(array_help) == 1:
                if bol_sitout == True:
                    int_count_sitout = 0
                else:
                    int_count_sitout = 1
                    bol_sitout = False

            help_remaining -= 1
            help_remaining = max(help_remaining, 1)

        if len(array_help) > 1:
            array_help = array_help[1:]

    return arr_combinations, arr_combinations_elements, bol_S_found

array_original = [12, 42, -3, -10, 24, 2, 4, -3, 9, 1] # example array
S = 18

# helping variables/lists
array = array_original.copy() # contains the array that is being modified in each run of the function below
lst_combinations = []
lst_combinations_elements = []
array_elements = []
bol_S_found = False

for int_arr_len in range(1, len(array_original)+1):
    array, array_elements, bol_S_found = get_combinations(array_original, array, array_elements, int_arr_len, S,
bol_S_found)

    lst_combinations += array
    lst_combinations_elements += array_elements

    if bol_S_found == True:
        break

if bol_S_found == True:
    print('subarray', [lst_combinations_elements[-1]])
else:
    print('no subarray with sum of S found')
```