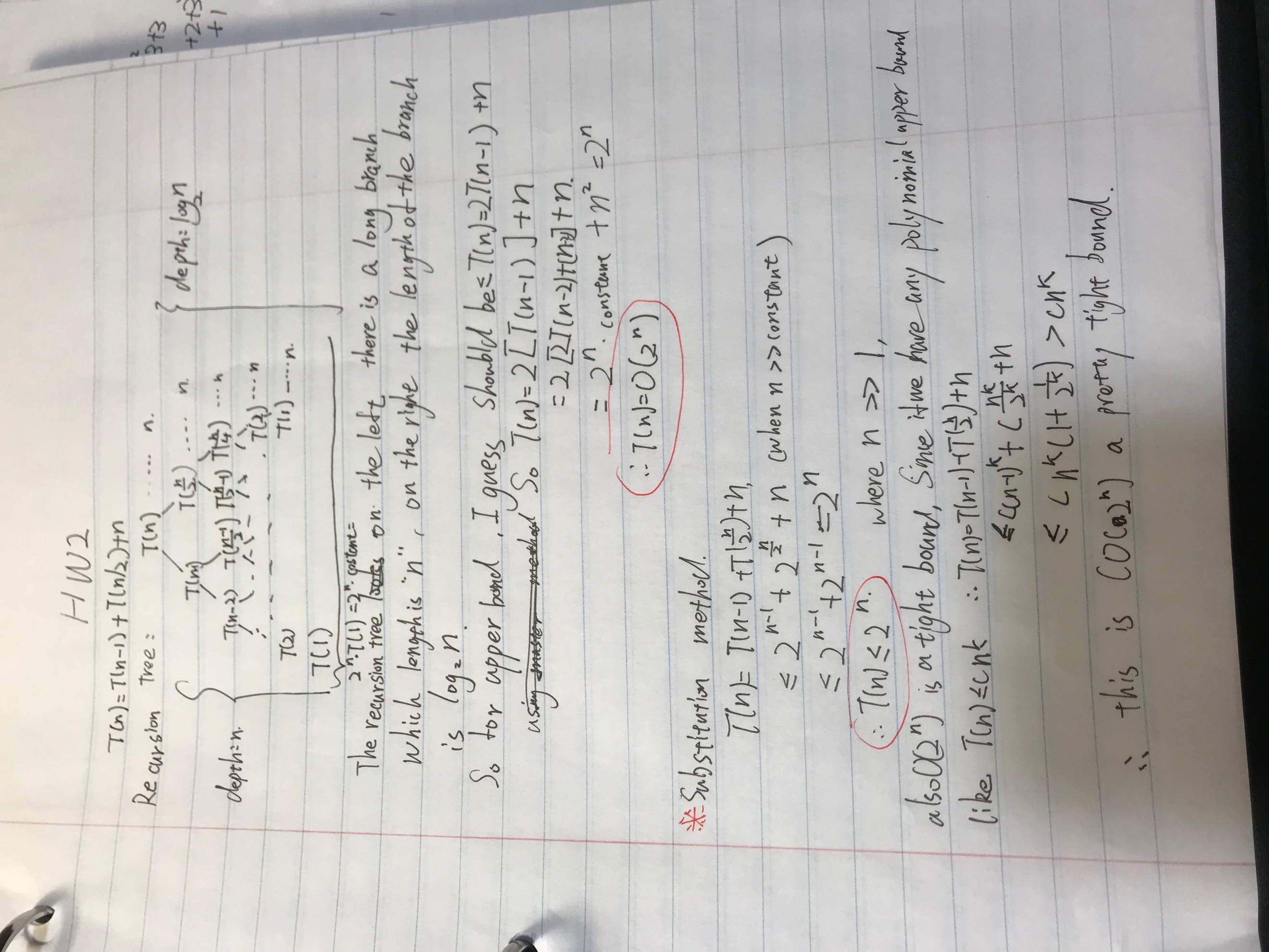
CSE541 HW2

Q1,



Q2,

A.

1. Divide the n elements of the input array into [n/9] groups of 9 elements each and at most one group made up of the remaining n mod 9 elements.
2. Find the median of each of the [n/9] groups by first insertion-sorting the elements of each group (of which there are at most 9) and then picking the median from the sorted list of group elements.
3. Use SELECT recursively to find the median x of the [n/9] medians found in step 2. (If there are an even number of medians, then by our convention, x is the lower median.)
4. Partition the input array around the median-of-medians x using the modified version of PARTITION. Let k be one more than the number of elements on the low side of the partition, so that x is the kth smallest element and there are n-k elements on the high side (or we can say, right side)of the partition.
5. If i = k, then return x. Otherwise, use SELECT recursively to find the ith smallest element on the low side if i < k. or the (i-k)th smallest element on the high side if i > k.

B.

For groups size of 9, we know at least half of medians found in step 2 are greater than or equal to the median-of-medians x. Thus, at least half of the [n/9] groups contribute at least 5 elements that are greater than x, except for the one group that has fewer than 9 elements if 9 does not divide n exactly, and the one group containing x itself. Discounting these two groups, it follows that the number of elements greater than x is at least:

*5\*([1/2\*[n/9]]-2) ≥ 5/18\*n -10= ((5n/18 )-10)*

Similarly, at least 5/18\*n-10 elements are less than x. thus, in the worst case, step 5 calls select recursively on at most 13/18\*n+10 elements.

C.

T(n) ≤ ⎨ O(1) if n < constant ( like: 140)

T(n/9) + T(13n/18+10) +O(n) if n ≥ constant ( like: 140)

D.

We show that the running time is linear by substitution. More specifically, we will show that T(n) ≤ c\*n for some suitably large constant c and all n>0. We begin by assuming that T(n) ≤ c\*n for some suitably large constant c and all n < 140; this assumption holds if c is large enough. We also pick a constant a such that the function described by the O(n) term above (which describes the non-recursive component of the running time of the algorithm) is bounded above by *a\*n* for all n>0. Substituting this inductive hypothesis into the right-hand side of the recurrence yields:

T(n) ≤ c[n/9]+ c[13n/18+10]+ an

≤ c\*n/9 + 13c\*n/18+10c +an

= 5c\*n/6 +10c+a\*n

= c\*n + ( -c\*n/6 +10c+a\*n)

which is at most c\*n if

*( -c\*n/6 +10c+a\*n) ≤ 0*  - 1.1

Inequality (1.1) is equivalent to the inequality c ≥ 6a (n /(n-60)) when n > 60. Because we assume that n ≥ 140, so n’s value will satisfy inequality (1.1). (Note that there is nothing special about the constant 140; we could replace it by any integer strictly greater than 70 and then choose c accordingly.) The worst-case running time of SELECT is therefore linear.

So T(n) = O(n).

Q3.

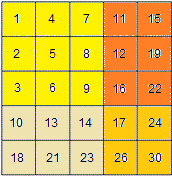
a.

Step 1: we can divide the n x n matrix into four (*n/2* x *n/2*) smaller matrices. For example, the center element **9** partitions the matrix into four matrices as shown in the picture below. Since the four smaller matrices are also sorted both row and column-wise, the problem can naturally be divided into four sub-problems.

Step 2: Assume the target number is 18, which is greater than the center element 9, we can eliminate the upper left matrix since all element in this matrix is smaller than it rightest-lowest element which is 9. We also can assume our target number is 8, which is smaller than 9.

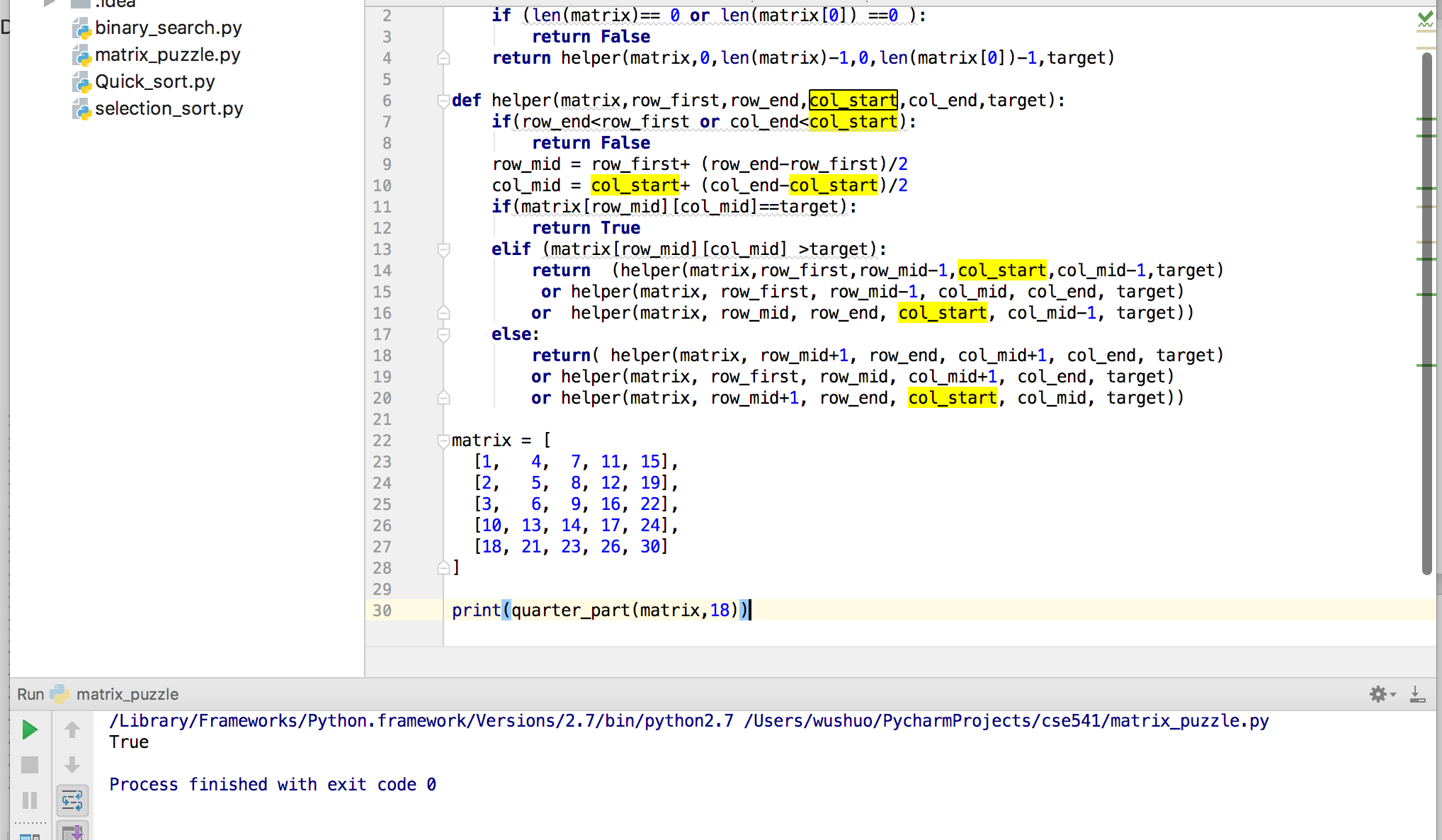
Similarly, we can drop the bottom right matrix for future calculation since the elements in this matrix are all bigger than 9. Please note however, we still need to search the upper right and bottom left quadrant, even though the example below seems to show all elements in the two mentioned quadrants are greater than 9.

Step3: Of course, if the center element is our target element, we have found the target and stop searching. If not, we proceed by searching the rest of three quadrants.



pseudo-code:

**def** quarter\_part( matrix, target):  
 **if** (len(matrix)== 0 **or** len(matrix[0]) ==0 ):  
 **return False  
 return** helper(matrix,0,len(matrix)-1,0,len(matrix[0])-1,target)  
  
**def** helper(matrix,row\_first,row\_end,col\_start,col\_end,target):  
 **if**(row\_end<row\_first **or** col\_end<col\_start):  
 **return False** row\_mid = row\_first+ (row\_end-row\_first)/2  
 col\_mid = col\_start+ (col\_end-col\_start)/2  
 **if**(matrix[row\_mid][col\_mid]==target):  
 **return True  
 elif** (matrix[row\_mid][col\_mid] >target):  
 **return** (helper(matrix,row\_first,row\_mid-1,col\_start,col\_mid-1,target)  
 **or** helper(matrix, row\_first, row\_mid-1, col\_mid, col\_end, target)  
 **or** helper(matrix, row\_mid, row\_end, col\_start, col\_mid-1, target))  
 **else**:  
 **return**( helper(matrix, row\_mid+1, row\_end, col\_mid+1, col\_end, target)  
 **or** helper(matrix, row\_first, row\_mid, col\_mid+1, col\_end, target)  
 **or** helper(matrix, row\_mid+1, row\_end, col\_start, col\_mid, target))



B,

T(n) = 3T(n/2) + c,

where n is the dimension of the matrix.

C,

Using master method,

We know that

T(*n*) = 3T(*n*/2) + *c*,

= 3 [ 3T(*n*/4) + *c* ] + *c*

= 3 [ 3 [ 3T(*n*/8) + *c* ] + *c* ] + *c*

= 3*k* T(*n*/2*k*) + *c* (3k - 1)/2

= 3*k* ( T(n/2*k*) + *c* ) - c/2

Setting *k* = lg *n*,

T(*n*) = 3lg *n* ( T(1) + *c* ) - c/2

= *O*(3lg *n*)

= *O*(*n*lg 3) <== 3lg *n = n*lg 3

= O(*n*1.58)

Q4,

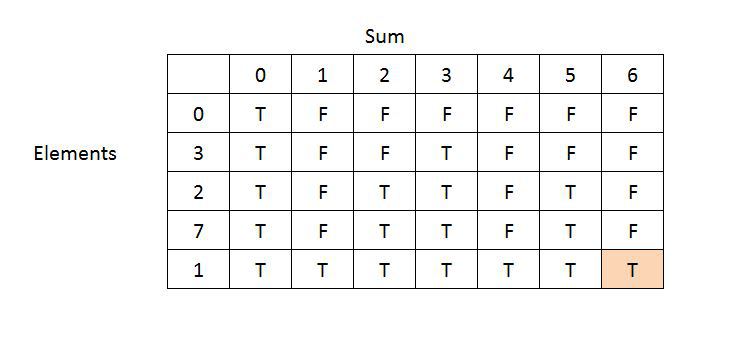
Pseudo-code:

Step 1: check if there is a subsequence of the given sequence with sum equal to given sum:

**def** isSubSum(st, n, sm):  
 *# The value of subset[i][j] will be  
 # true if there is a subset of  
 # set[0..j-1] with sum equal to i*

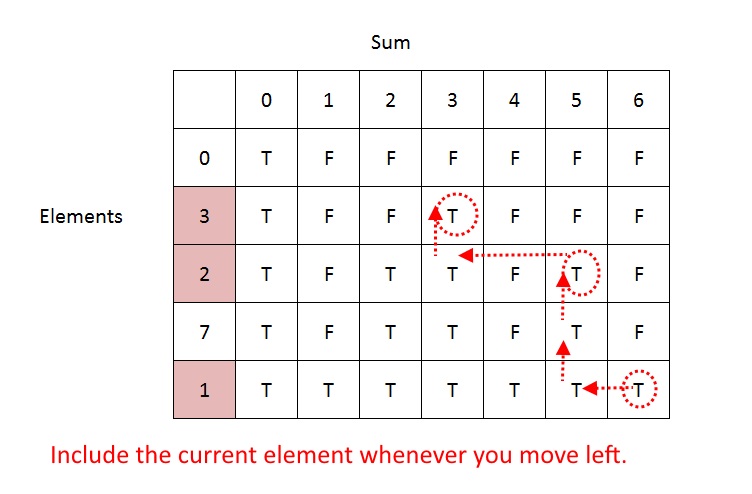
boolean[][] solution = new boolean[A.length + 1][sum + 1]

*# If sum is 0, then answer is true* **for** i **in** range(0, n + 1):  
 subset[i][0] = **True** *# If sum is not 0 and set is empty,  
 # then answer is false* **for** i **in** range(1, sm + 1):  
 subset[0][i] = **False** *# Fill the subset table in botton  
 # up manner* **for** i **in** range(1, n + 1):  
 **for** j **in** range(1, sm + 1):  
 **if** (j < st[i - 1]):  
 subset[i][j] = subset[i - 1][j]  
 **if** (j >= st[i - 1] and subset[i][j] == False):  
 subset[i][j] = subset[i - 1][j] **or** subset[i - 1][j - st[i - 1]]  
  
 **return** subset[n][sm];

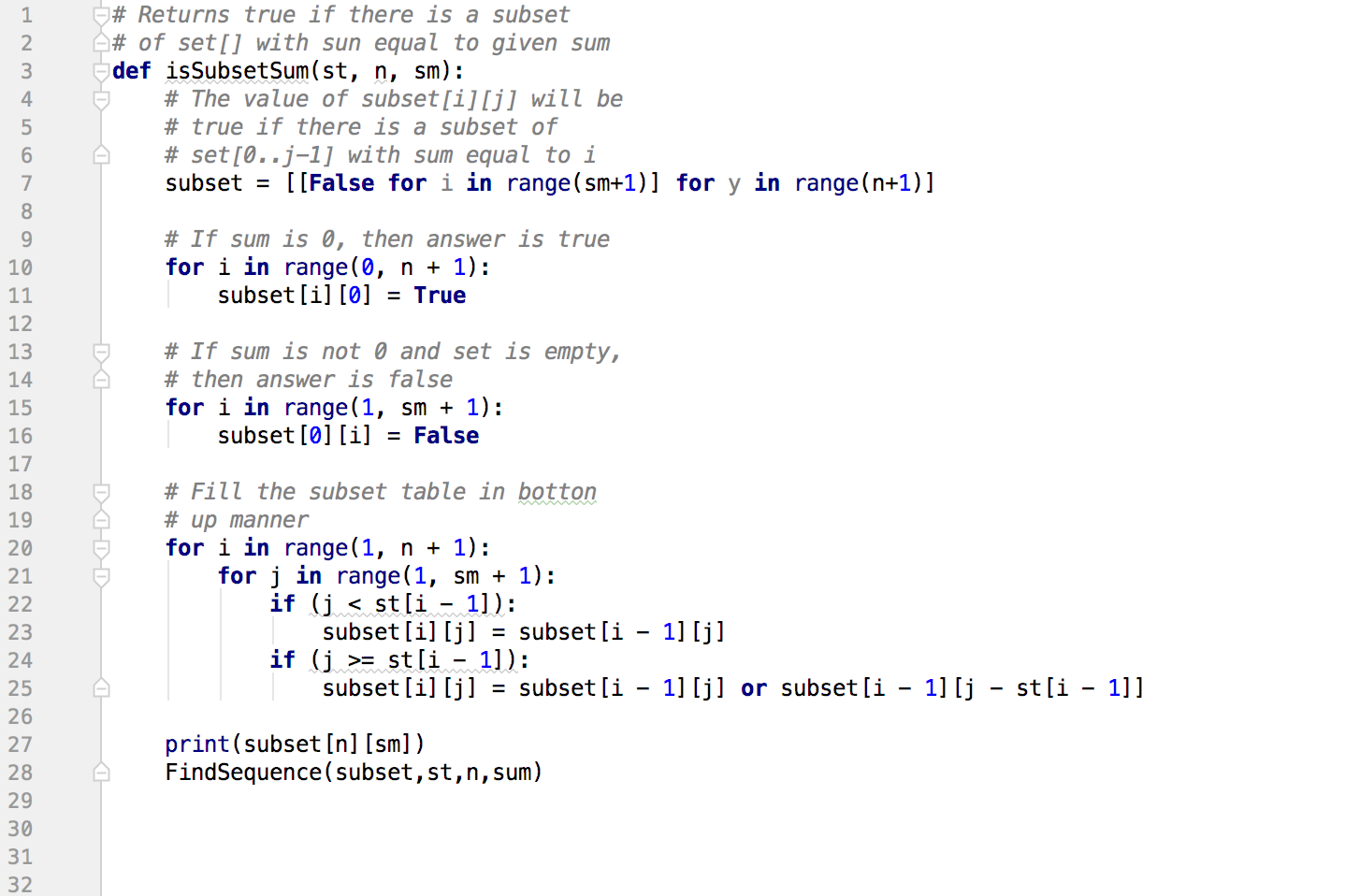


Step 2: if subset == true, return the sequence; else, return false

So I just include the current element whenever I move left:



Final code:





output:



input: set = [3,2,7,1] sum =6

output answer = [1,2,3]

worst case: T(n) = Θ(sum\*n)+ Θ(n+sum)= Θ(sum\*n) since we need to fill the table with size sum\*n.

where n is the length of the sequence. Sum is the number we want to find.