1. Report B:

There is an array A [1, …, N], which include integers between and . We wrote a code, which finds possible non-empty contiguous subarrays of A that has sum in the range [, ], and it finds valid counts of the subarrays. There are three algorithms: , and .

Firstly, a pseudo code below is algorithm.

for to in A: // start pointer

for to in A: // end pointer

for to : // calculating sum of subarray

calculate sum of subarray in the range [, ]

if sum of subarray is in [, ]

count += 1

The code above is to count the number of sum of subarrays, which are in the range [, ]. By using start pointer ( and end pointer (), the first for loop set a start pointer to beginning of a subarray. The second for loop set end pointer to the end of subarray by increasing index one by one until it reaches the end of entire array. The last for loop is to calculate the sum of the subarray from start pointer to end pointer.

Next, a pseudo code below is algorithm.

for to in A: // start pointer

for to in A: // end pointer

sum of subarray += A []

if sum of subarray is in [, ]

count += 1

The code above is also to count the number of sum of subarrays, which are in the range [, ]. This method is quite similar to algorithm. However, we can optimize previous algorithm by moving end pointer and adding the value each step instead of using the for loop to calculate sum of subarray.

Lastly, a pseudo code below is algorithm.

def dac (A[1, …, n], ,

if n < 2: // base case

return 1 if <= A[1] <= else 0

count\_left = dac A[1, …, n/2-1], , // half\_left of array

count\_right = dac A[n/2, …, n], , // half\_right of array

count\_mid = dac\_mid (A[1, …, n], , ) // check across cases

def dac\_mid (A[1, …, n], ,

mid = len(A) // 2

idx\_left, idx\_right = mid-1, mid

right\_prefix\_sum = A[idx\_right]

mid\_sum = right\_prefix\_sum

while idx\_left >= 0 and idx\_right < n:

mid\_sum += A[idx\_left] // add right and left values

if sum of subarray is in [, ]

count += 1

decrese idx\_left

if idx\_left is A[1]:

move idx\_left to mid-1

increase idx\_right and add the value to right\_prefix\_sum

The code above uses Divide-And-Conquer algorithm to count the number of the sum of subarrays without sorting, which are in the range [, ]. There are two functions: ‘dac’ and ‘dac\_mid’. The ‘dac’ function divides a given array into two subarray by using mid-point, which is a center of the given array. The subarray of the given array can be A[1, …, 2/n-1] and A[2/n, …, n], and they will be used to find subarrays fulfilling the requirement in recursive ways repeatedly until they become base cases. The ‘dac\_mid’ function is for checking and counting the subarrays if there are some exceptional cases such as the elements from right side to left side of the array. While the ‘dac\_mid’ function is working, the program stores from mid-point values and add the values from left side of array step by step. When the index is equal to zero, it is initialized by (mid-1) for next calculation. This loop will continue until the index of right side of array is equal to n. After all calculation, the count will return.

1. Calculating Running Time of Problem B

The first code uses three nested loops. The first loop moves from A[1] to A[n], which means it depends on the array size, so it would be . Likewise, the second loop also moves from A[1] to A[n] many times, following the first loop, which leads to a time complexity of . The last loop moves from the pointer of the first loop to the pointer of the second loop. It also follows the first and second loop pointers, and its worst case would be . Therefore, all loops work simultaneously, which means \* \* . A total time complexity of this algorithm is .

The second code uses two nested loops. The first loop moves from A[1] to A[n], which means it depends on the array size, so it would be . Unlike the first code, the second loop plays two role in the algorithm as becoming pointer and gathering the sum of subarrays. It also moves from A[1] to A[n] gradually, which leads to a time complexity of . The calculation work simultaneously, which means \* . Thus, a total time complexity of this algorithm is .

The last code uses Divide-And-Conquer algorithm (DAC) to find the subarrays that fulfill the requirements of the problem without sorting. This method takes , where n is the number of elements in the given input array. By using DAC algorithm, the program recursively divides input array into two subarrays until reaching the base case, finds the subarrays fulfilling requirements (left, right and both) and applies DAC again to each part in a recursive way. This results with at most many stages in the recursion tree. At every stage, it takes operation when the program calls the function to check the subarrays that are across the midpoint. In particular, ‘idx\_left’ and ‘idx\_right’ in the code are not incremented in every iteration, and in the worst case, the while loop might iterate over all elements in the middle part of the array, resulting in a time complexity of . The overall time complexity of DAC algorithm consists of the ‘dac’ function and the ‘dac\_mid’. Especially, the ‘dac’ function has recursion like which leads to a time complexity of . Therefore, a total time complexity is .

1. Proof of Correctness Of B

To prove both first and second algorithm, the start and end pointers always search from A[1] to A[n] in A[1, …, n]. For any given array, these pointers are guaranteed to reach the last index of the array because whether certain subarrays near the last index fulfill the requirement or not, pointers continue to move until reaching the end of the array. For instance, a sum of array has elements , which means . The value of () should be in the array X, which indicates . Therefore, both linear search algorithm finds all possible cases fulfilling the requirements.

The proof that the proposed algorithm solves the problem needs to show that a base case, inductive step, midpoint subarray count and termination in ‘dac’ algorithm and searching valid sum in ‘dac\_mid’. In the base case, subarray should have one element, which can be as an example. If , it means is not valid. Then, the base case leads to termination of the recursion in divide and conquer algorithm, which directly computes and returns the result when In the inductive step, the algorithm correctly counts subarrays with sums with specified range for smaller subproblems. For example, in the larger subarray A[1, …, n], the count of smaller subarray B[1,..n/2] A can be ‘’, the count of smaller subarray C[n/2+1, …, n] can be ‘’, and the count of smaller subarray D[1, …, n] can be ‘’ because the whole array can be the valid case, which means the value of should be return value.

During this process, midpoint subarray count can also predict because of the case above. Linear search for a value of right\_prefix\_sum should be the sum of subarray of larger array. For example, right side subarray is , and its elements are , which means . Left side subarray is , and its elements are , which means . Then, the sum of each values of both sides should be the sum of its elements.