Additional Exercise (pre-partial exam) - Count Subsequences problem

You will be asked to implement several solutions of the following problem: given as an input a list `arr` with **positive** integers (i.e. all elements of `arr` are greater than zero), and a positive integer `k`, output the number of pair of indices (i, j), s.t. the sum of all elements $arr[i] + arr[i+1] + \ldots arr[j]$ is equal to k.

Example
`count_subsequences([1,3,2,1, 6, 2, 2, 2], 6) == 4` because $1+3+2=6, 3+2+1=6, 6=6$, and $2+2+2=6$, so pairs $(0,2), (1,3), (4,4)$ and $(5,8)$ all satisfy the conditions.

We will go over several, progressively faster algorithms solving this problem. Implement each of those.

Exercise 1
Write a function `count_subsequences_slow` which iterates over all pairs of indices $i < j$ (consider using nested loop), for each pair compute the sum of the sub-array starting at the index i and ending at the index j. If the sum is equal to k, increment the counter, counting the number of valid pairs. Return the value of the counter at the end. This algorithms runs in time $O(n^3)$.

(Hint 1: to calculate a sum of all elements in a given list, you can either write a separate function, or use the Python built-in function `sum`)
(Hint 2: to get a sub-array containing all elements with indices from i to j, you can use Python slicing syntax `arr[i:j+1]`)

Exercise 2
a) Write first a function `cumulative_sum(arr)`, which runs in time $O(n)$, and -- given a list of length n of integers -- produces a new list of length $n+1$. Element with the index i of the output list should be the sum of the first i elements of the original list. In particular, the element with the index 0 of the output list is a sum of the first 0 elements of the original list --- it is always zero; similarly, the last element of the output list is just a sum of all elements in the original list.

Example:

```
cumulative_sum([5, 3, 1, 2, 1]) == [0, 5, 8, 9, 11, 12]
cumulative_sum([1,2,3,4]) == [0, 1, 3, 6, 10]
```

b) Using this function write a function `count_subsequences_faster`, solving the original problem in time $O(n^2)$.

Your code should implement the following algorithm: given an array `arr` as input, call the `cumulative_sum(arr)` function to compute sums of all prefixes of the array `arr`. Assign result of this call to the variable `cs`. Observe that `arr[i] + arr[i+1] + ... + arr[j] == k` happens exactly when `cs[j+1] - cs[i] == k`. As such, it is enough to iterate over all pairs of indices i, j, and check if `cs[j] - cs[i] == k`, increasing a counter each time this condition is satisfied. This can be done with nested loops in time $O(n^2)$.

Exercise 3
a) Write a function `binary_search(arr, k)`, where `arr` is an increasing list of integers, and `k` is an integer. The function `binary_search` should output `True` if the element `k` is present in the (increasing) array `arr`. Write this even if you already wrote binary search as a part of some previous exercise. This function should implement a `binary_search` algorithm and run in time $O(\log n)$.
b) Using the `cumulative_sum` function from the Exercise 2, and the `binary_search` function from the previous sub-problem, write a function `count_subsequences_fast` solving the original problem in time $O(n \log n)$.

Specifically, note that for a given index i, there is at most one different index j s.t. `cs[j+1] == cs[i] + k` (since all elements of the array `arr` where positive, the array `cs` produced by the function `cumulative_sum` is strictly increasing). Using the function `binary_search` from part (a) check if there is indeed such an index. Concretely, your function `count_subsequences_fast` should implement the following algorithm:

- First, compute the cumulative sum of the input array `arr`.
- Then, iterating over all elements from the array `cs`, call the `binary_search` function to check if the elements `cs[i] + k` is present in the array `cs` -- if it is, increase the counter.
- Return the value of the counter.
  Note that this code calls $O(n)$ times a function `binary_search` which has running time $O(\log n)$, for the total complexity of $O(n \log n)$.

Exercise 4
Write a function that solves the original problem stated in the beginning in time $O(n)$. Your function can, but do not have to, use the `cumulative_sum` function from the Exercise 2. It should implement the following algorithm:

1. At all times we keep two indices `head` and `tail`, with `tail <= head`, pointing at two locations in the array `arr`, as well as a variable `current_sum` keeping the current sum of a sub-array between the index `tail` and `head`.

2. In each iteration of the loop:
   - If the `current_sum` is exactly `k`, increase the variable `result` counting how many solutions we found.
   - If the `current_sum` is at least `k` decrease the `current_sum` by `arr[tail]` and increase the counter `tail`.
   - If the `current_sum` is smaller than `k`, increase the counter `head`, and increase the `current_sum` accordingly, so that the property in (1.) is satisfied: the `current_sum` variable should keep the sum of all elements between the index `tail` and `head`.
   Make sure to implement proper conditions for when to finish the loop, and that you correctly handle the situations where either of the counters reaches the end of the array, or the `tail` catches up with the `head`.