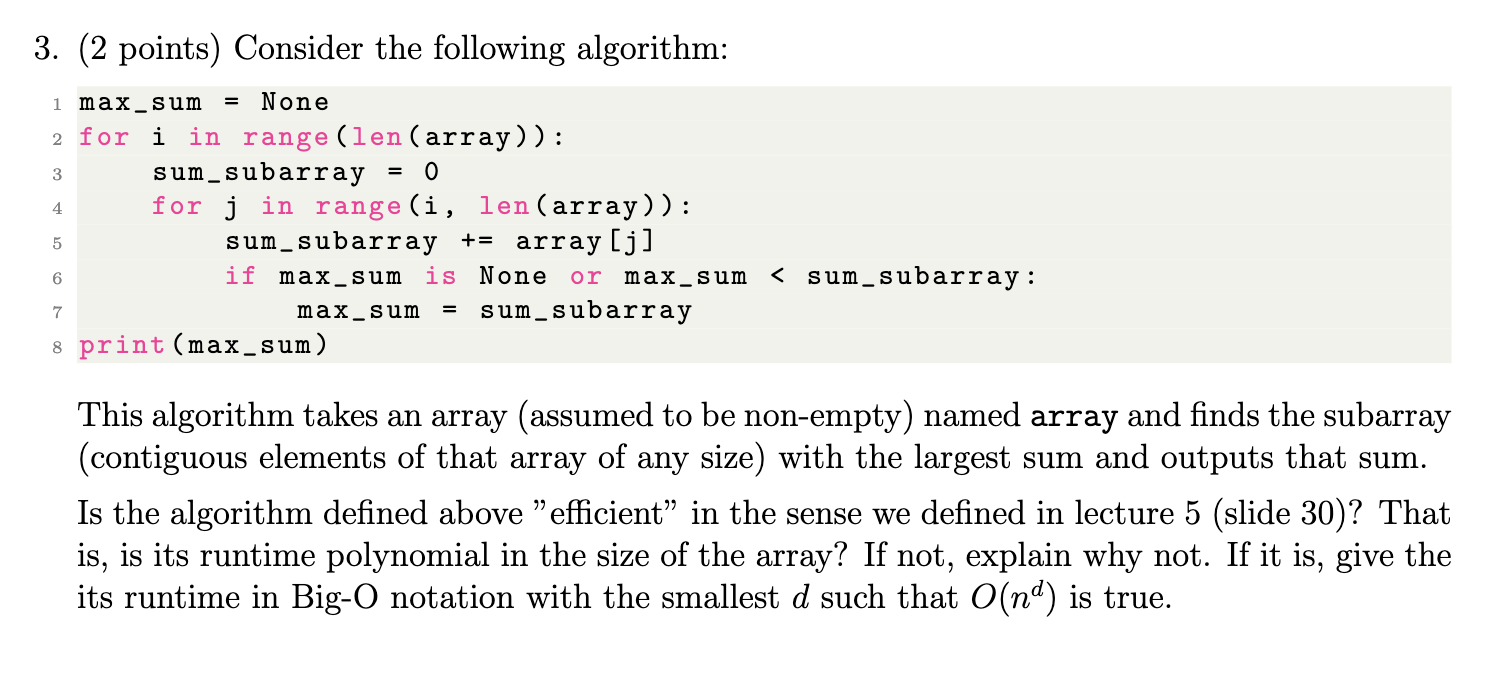


* Function
  + takes a parameter (‘input’)
  + ‘input’ is presumably a list array
  + it iterates over every element in ‘input’
  + if the first character/value of the element is ‘1’. then this function transforms the whole element to be ‘None’.
  + It then returns the newly modified array
* given that the ‘original\_array’ first value is a ‘1’, when it is passed to the function, the ‘new array’ will output as [None, 5\_2]
  + so the output given from this code chunk is   
    ["1\_3", "5\_2"]

[None, "5\_2"]

* considering this from the perspective of pointers,
  + A pointer is when a memory location is encoded and stored in memory itself
  + variables hold pointers to memory locations
  + so when we are passing ‘original array’ to ‘check array’ function, we are passing a reference to the list’s memory location, where the function then directly modifies the list
  + this is directly edited because both the parameter inside the function and the original variable outside the function point to the same memory location
  + as our python function iterates over each element according to its index reference, it is like passing a pointer to the array
  + slide 31 also considers ‘garbage collection’ – here if an object in python no longer has any references pointing to it, it becomes eligible for garbage collection (free up memory). But that is NOT happening here, as bot ‘original array’ var, and ‘new array’ var are pointing to it, so it remains an active variable

variables can act as pointers to data structures in memory, allowing for in-place modifications that are reflected across all references to that data structure.



3:

* this is inefficient, brute-force approach
* line 2: outer loop, that iterates through each element of the array
  + each selected index of the array then serves as the starting point of the contiguous subarray
* line 4: inner loop, starts from current element of outer loop (i) to end of the array
  + The variable j represents the ending index of the subarray being considered. This loop expands the subarray one element at a time. ??????????
  + it adds the value of the current element (j) to the sum\_subarray variable.
  + This therefore accumulates total sum a all the elements in the inner loop between i and end of array. ????
* lines 6 & 7 just update the max sum to the largest value found so far
* intuitively it is highly inefficient, because the very first calculation will be the largest sum (where we start at i and therefore have the maximum number of elements between i and the end of the list). All subsequent calculations are unnecessary
  + wait, but if it has negative numbers
  + then a larger sum could come from smaller array
  + so Without prior knowledge of the array's contents, it's impossible to know ahead of time which subarray will yield the largest sum. Therefore, the algorithm performs an exhaustive search to ensure it finds the maximum sum
  + there are ways to make this more efficient, but they are beyond my pay grade
* as always, we take the worst case (seeking upper bound):
  + outer loop: runs n times
  + for each of these n iterations, the inner loop runs
  + inner loop depends on position of outer loop
    - worst case: when i = 0:
    - first iteration (i=0): runs n times
    - second iteration (i=1): n-1 times
    - third iteration (i=2): n-2 times
    - ..
    - final iteration (i=n-1): runs 1 time
* here is where it is bruite force – it could take a running sum and work out where there are negative numbers, to be able to adapt and discard as required. Could also know apriori if there are only pos integers (and 0) then it only needs to take first sum
* to find upper bound on total operations in nested loops: sum iterations of each inner loop for each possible value of outer loop
  + n + (n-1) + (n-2)… +1
  + well combinatorics known series: (what’s the specific name?)
  + n\*(n+1)/2
* in big O: we ignore constants: just focus on polynomials of n
  + n \* n
  + n^2
  + O(n^2)
  + = complexity quadratically increases with size or input array