Data Structures & Algorithms: Assignment 2

Henry Baker

March 13, 2024

1 Question 2

Going through the algorithm step-by-step:
(NB here I am consciously using bullet points without full sentences for clarity)

- The function `check_array` does the following:
  - takes a parameter (here: `input`)
  - `input` is presumably a list array (given function name `check_array`)
  - it iterates over every element in `input` (here: `idx`)
  - if the first character of the element (`[0]`) is `1` then this function transforms the whole element to be `None`.
  - It then returns the newly modified array.
- This code chunk then assigns a set of values to `original_array` variable.
- The code then feeds `original_array` to the defined function and assigns the output to `new_array`.
- Given that the `original_array`'s 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 here we first store the function's algorithm operations in memory and assign a pointer with `check_array` variable. We then store `["1_3" "5_2"]` in memory and a pointer is held to that location by `original_array` variable. When we pass `original_array` to the `check_array`

function we are passing a reference to the list's memory location - as the function iterates over each element according to its index reference it is like passing a pointer to the array. The function then modifies the list and stores those results at a different location in memory which is in turn pointed to by `new_array` variable.

NB there's no opportunity for 'garbage collection' to free up memory as both `original_array` variable and `new array` variable continue pointing at their memory locations so they remain active variables.

2 Question 3

This represents a brute-force approach. There are ways to make it more efficient. However given our definition of efficiency where *'an algorithm is efficient if its runtime is polynomials'* it is efficient.

The algorithm does as follows:

1. **line 2**: sets up an outer loop that iterates through each element of the array:
   - each selected index of the array is then the starting point of the contiguous subarray (whose sum we will be taking).
2. **line 4:** sets up an inner loop that iterates through the subarray starting from current element of outer loop (i) to the end of the array itself.
   - The variable j is the end index of the subarray being considered. This loop expands the subarray one element at a time as it iterates through the subarray.
   - it adds the value of the current element (j) to the `sum_subarray` variable (line 5).
3. **Lines 6 & 7**: checks if the sum of the currently considered subarray between i and j is the largest value found so far (line 6). If so it saves it as `max_sum` (line 7)

This therefore accumulates all the possible sums of all possible contiguous permutations of the elements in the array and saves the iteratively largest sum encountered at that point in the algorithm. The algorithm halts when the outer loop has completed (i.e., it has indexed through the entire length of the array).

To determine whether it is efficient or not, we take the worst case (we are seeking an upper bound). The outer loop: runs n times, for each of these n iterations, the inner loop runs. The inner loop depends on the position of the outer loop. In the

first iteration ($i = 0$), the inner loop runs n times; in the second iteration ($i = 1$), it runs $n-1$ times; in the third iteration ($i = 2$), it runs $n-2$ times, etc., until the final iteration ($i = n-1$), where it runs 1 time. Thus, to find the upper bound on the total operations in nested loops, we sum the iterations of each inner loop for each possible value of the outer loop. Formally: $n + (n-1) + (n-2) + \ldots + 1$. This combinatorics series can be simplified to $\frac{n \cdot (n+1)}{2}$.

In big O notation, we ignore constants to focus on the polynomials of n. In this case, to determine the order, we have an $n \times n$ element, giving us n^2. In big O notation, this becomes $O(n^2)$, denoting that its complexity quadratically increases with the size of the input array. Here we are in polynomial time where $d = 2$. An algorithm is efficient if its runtime is polynomial; thus, it is efficient.

Nevertheless, this represents a brute force approach because there is unexploited structure here. Unless there are negative numbers, the greatest sum would simply be the very first iteration. If we wanted to be more efficient, we could take a running sum to determine the location of negative values within the array, to be able to build our sum to maximise around these negative values.