Lab 01

Fill in the missing constant n in the given code. The procedure uvw() should be called exactly 49 times.

```
for (i = 0; i < 7; i++) {
  j = i;
  while (j < n) {
    uvw();
    j++;
  }
}
```

Outer loop executes exactly 7-times. So the question is how to tweak the inner loop to satisfy the constraint. Expanding the loop, we see that inner loop executes $n + n - 1 + n - 2 + \ldots + n - 6 = 7n - 21$ times. Thus, $49 = 7n - 21 \Rightarrow n = 10$

A sequence $S = s_0, s_1, \ldots, s_n$ contains positive integers, negative integers and zeros. Find a contiguous subsequence of S whose sum of elements is the maximum possible among all contiguous subsequences of S. Is it possible to scan S only once to complete the task? Do not create any additional sequences or big data structures.

Yes, it is possible to do it in $O(n)$. The algorithm iterates over the field and computes the "cumulative sum" $C_i = \sum_{j=0}^{i} S_j$, i.e., the sum of elements so far. With the cumulative sum, we can query a sum of arbitrary sub-sequence $S_{a,b}$ as $C_b - C_a$. As we are interested in the biggest partial sum, we just keep track of the so-far biggest and lowest value of the cumulative sum $s_a = \min(S)$ and $s_b = \max(S)$, such that $b > a$.

When function f grows asymptotically faster than function g (ie. $f(x) \notin O(g(x))$) some of the following is true:

- If both f and g are defined in x then $f(x) > g(x)$.

Not necessarily. E.g., 2^x grows faster than x^2 ($2^x \notin O(x^2)$), but $2^3 = 8 \ngtr 3^2 = 9$.

- The difference $f(x) - g(x)$ is always positive.

Not necessarily. In the example above, for $x = 3$, the difference is -1.

- The difference $f(x) - g(x)$ is positive for each $x > x_0$, where x_0 is some sufficiently big real number.

Yes. $f \notin O(g) \Rightarrow f \in \theta(g)$. Recall definition. For the example above, the x_0 we seek is 4.

- Both f and g are defined only for non-negative arguments.

No. Both exemplar functions are also defined on negative numbers.

- none of the previous

No. We found one true statement.

Algorithm A processes all elements of a 1D array of size n . Processing of an element with index k consists of a subroutine call which asymptotic complexity is $\Theta(k + n)$. Asymptotic complexity of A is therefore:

$$\Theta\left(\underbrace{0 + n}_{k=0} + \underbrace{1 + n}_{k=1} + \underbrace{2 + n}_{k=2} + \dots + \underbrace{n + n}_{k=n}\right) = \Theta\left(n \sum_{i=1}^{n} i\right) = \Theta(n^3)$$

Determine the asymptotic complexity of the code with respect to the value of N.

```
int a [N];
for(i = 0; i < N; i++)
  a[i] = i;
}
for (i = 0; i < N; i++){
  while (a[i] > 0) {
    print(a[i]);
    a[i] = a[i]/2; // integer division
  }
}
```

Let's first understand the inner loop: It halves a given number $a[i]$ until it is zero. We need $\log_2(a[i])$ divisions to reach zero. (Printing can be regarded as a single operation together with the division and thus safely ignored.)

Now remember how the array is initialized. The outer loop thus expands into $\log_2(1) + \log_2(2) + \log_2(3) + \dots + \log_2(n) = \log_2(n!) \in \Theta(n \log n)$. (See e.g., Stolz-Cezaro proof. Or at least find an upper bound by comparing against "worst case" $\sum_{i}^{n} \log(n) = n \log n$.)

If you are tempted to include the initialization such as $\Theta(n + n \log n)$, do not be. The "weaker" additive function is not significant, recall the definition of asymptotic complexity.