Exercise 1

As a warm-up for analyzing runtime, consider the pseudo-code for SUM-EVEN-NUMBERS which takes a sequence of integers and returns the sum of all even numbers:

```
SUM-EVEN-NUMBERS(A)
sum = 0
for i = 1 to A.length
    if A[i] mod 2 == 0
        sum = sum + A[i]
return sum
```

Assume that each line i takes c_i time to execute.

1. What is the *exact* running time $T(n)$ of SUM-EVEN-NUMBERS?
2. What is the *worst case* running time and what input would cause this?

Exercise 2

Consider the problem of finding the two smallest numbers in a non-empty sequence of numbers, not necessarily sorted.

1. Formalise the above as a computational problem (be careful to precisely define the input, the output and their relationship).
2. Write the pseudo-code of an algorithm that solves the above computational problem, assuming the sequence is given as an array $A[1 \ldots n]$.
3. Assume that line i of your pseudo-code takes constant time c_i to execute. What is the worst case running time of your algorithm?

Exercise 3

Consider the problem of reversing an array of numbers, ie. so the last element becomes the first, the second last becomes the second, etc. For example, the reverse of $[1, 2, 3, 4, 5]$ is $[5, 4, 3, 2, 1]$.

1. Write the pseudo-code for an algorithm that solves this problem. *Hint:* In pseudo-code, you can just create a new array of size n by writing '$B =$ new array of size n'
2. What is the worst case running time of your algorithm?

We measure space almost like we measure time, except instead of assuming a constant cost of each line, we assume a constant cost c of each new variable (since the value associated with the variable is stored in memory). An array of size n then takes $c \cdot n$ space, since we are essentially creating n variables, one for each slot in the array.

3. Assuming you created another array for the reversed sequence in the above question, try now to propose an algorithm that only uses a contant amount of extra space. What is the worst case running time of your algorithm?

Exercise 4

Consider the following star-printing algorithms:

```
PRINT-LOGARITHMIC(n)
i = n
while i ≥ 1
    print *
    i = i/2
```

```
PRINT-LINEAR(n)
for i = 1 to n
    print *
```

```
PRINT-SQUARED(n)
for i = 1 to n
    for j = 1 to n
        print *
```

1. How many stars are printed with by each algorithm, when you call PRINT-LOGARITHMIC(n) with $n = 32, 16, 8, 4, 2$ and PRINT-LINEAR(n) and PRINT-SQUARED(n) with $n = 1, 2, 3, 4, 5$?
2. Is there a relation between the number of stars printed and the running time of the algorithms?