# DSA – Seminar 2 –
# Complexity (Algorithm Analysis)

1. TRUE or FALSE?
   a. $n^2 \in O(n^3)$ – True
   b. $n^3 \in O(n^2)$ – False
   c. $2^{n+1} \in \Theta(2^n)$ – True
   d. $2^{2n} \in \Theta(2^n)$ - False
   e. $n^2 \in \Theta(n^3)$ – False
   f. $2^n \in O(n!)$ - True
   g. $\log_{10}n \in \Theta(\log_2 n)$ - True
   h. $O(n) + \Theta(n^2) = \Theta(n^2)$ - True
      $\Theta(n) + O(n^2) = O(n^2)$ – True also $\Theta(n) + O(n^2) = \Omega(n)$
   i. $O(n) + O(n^2) = O(n^2)$ – True
   j. $O(n) + \Theta(n) = O(n)$ – True, but $\Theta(n)$ should be used
   k. $(n + m)^2 \in O(n^2 + m^2)$ – True - because $(n+m)^2 < 3*(n^2+m^2)$
   l. $3^n \in O(2^n)$ – False
   m. $\log_2 3^n \in O(\log_2 2^n)$ – True
2. Complexity of search and sorting algorithms

| Algorithm | Time Complexity | | | | Extra Space Complexity |
|---|---|---|---|---|---|
| | Best C. | Worst C. | Average C. | Total | |
| Linear Search | $\Theta(1)$ | $\Theta(n)$ | $\Theta(n)$ | $O(n)$ | $\Theta(1)$ |
| Binary Search | $\Theta(1)$ | $\Theta(\log_2 n)$ | $\Theta(\log_2 n)$ | $O(\log_2 n)$ | $\Theta(1)$ |
| Selection Sort | $\Theta(n^2)$ | $\Theta(n^2)$ | $\Theta(n^2)$ | $\Theta(n^2)$ | $\Theta(1)$ – in place |
| Insertion Sort | $\Theta(n)$ | $\Theta(n^2)$ | $\Theta(n^2)$ | $O(n^2)$ | $\Theta(1)$ – in place |
| Bubble Sort | $\Theta(n)$ | $\Theta(n^2)$ | $\Theta(n^2)$ | $O(n^2)$ | $\Theta(1)$ – in place |
| Quick Sort | $\Theta(n \log_2 n)$ | $\Theta(n^2)$ | $\Theta(n \log_2 n)$ | $O(n^2)$ | $\Theta(1)$ – in place |
| Merge Sort | $\Theta(n \log_2 n)$ | $\Theta(n \log_2 n)$ | $\Theta(n \log_2 n)$ | $\Theta(n \log_2 n)$ | $\Theta(n)$- out of place |

3. Analyze the time complexity of the following two subalgorithms:

```
subalgorithm s1(n) is:
      for i ← 1, n execute
            j ← n
            while j ≠ 0 execute
                  j ← [j/2]
            end-while
      end-for
end-subalgorithm
```

- The *for* loop is repeated n times.
- The *while* loop is repeated $\log_2 n$ times, independent of the value of i. (how many times can we divide n to get to 0)
- $T(n) \in \Theta(n * \log_2 n)$

```
subalgorithm s2(n) is:
      for i ← 1, n execute
            j ← i
            while j ≠ 0 execute
                  j ← [j/2]
            end-while
      end-for
end-subalgorithm
```

- The *for* loop is repeated n times.
- The *while* loop is repeated $\log_2 i$ times.
- $T(n) = \log_2 1 + \log_2 2 + \log_2 3 + \ldots + \log_2 n = \log_2 n! \Rightarrow n \log_2 n$ (Stirling's approximation)
- $T(n) \in \Theta(n * \log_2 n)$

4. Analyze the time complexity of the following two subalgorithms:

```
subalgorithm s3(x, n, a) is:
      found ← false
      for i ← 1, n execute
            if xi = a then
                  found ← true
            end-if
      end-for
end-subalgorithm
```

$$\left.\begin{matrix} BC: \theta(n) \\ WC: \theta(n) \end{matrix}\right\} \Rightarrow \Theta(n)$$

```
subalgorithm s4(x, n, a) is:
      found ← false
      i ← 1
      while found = false and i ≤ n execute
            if xi = a then
                  found ← true
            end-if
            i ← i + 1
      end-while
end-subalgoritm
```

BC: $\Theta(1)$
WC: $\Theta(n)$

AC: there are n+1 possible cases (element is found on one of the n positions and the case when element is not found. We suppose that all of these cases have equal probability – even if this might not always be the case in real life).

$$T(n) = \sum_{I \in D} P(I) * E(I) = \frac{1}{n+1} + \frac{2}{n+1} + \dots + \frac{n}{n+1} + \frac{n}{n+1} = \frac{n*(n+1)}{2*(n+1)} + \frac{n}{n+1} \in \Theta(n)$$

Total Complexity: O(n)

5. Analyze the time complexity of the following algorithm (x is an array, with elements $x_i$ <= n):

```
Subalgorithm s5(x, n) is:
      k← 0
      for i ← 1, n execute
            for j ← 1, xi execute
                  k ← k + xj
            end-for
      end-for
end-subalgorithm
```

   a. if every $x_i > 0$

When we have for loops (and the loop variable changes by 1), computing the complexity can be done by writing the for loop as a sum (limits of the sum are limits of the for and the content of the sum if the number of instructions in the for loop).

$$T(x,n) = \sum_{i=1}^{n} \sum_{j=1}^{xi} 1 = \sum_{i=1}^{n} x_i = s \ (sum\ of\ all\ elements)$$

T(n) ϵ Θ (s)

   b. if $x_i$ can be 0

- Does the complexity change if we allow values of 0 in the array?

Think about an array *x* defined in the following way:

Let $x_i = \begin{cases} 1, if\ i\ is\ a\ perfect\ square \\ 0, otherwise \end{cases}$

In this case: s = √n, but the complexity is Θ (n), because of the first for loop which will be executed n times, no matter what.

T(x, n) ϵ Θ (max {n, s}) = Θ (n + s)

6. Consider the following problems and find an algorithm (having the required time complexity) to solve them :
    a. Given an arbitrary array with numbers $x_1...x_n$, determine whether there are 2 equal elements in the array. Show that this can be done with Θ (n $\log_2$ n) time complexity.
    b. Given an arbitrary array with numbers $x_1...x_n$, determine whether there are two numbers whose sum is *k* (for some given k). Show that this can be done with Θ (n $\log_2$ n) time complexity. What happens if *k* is even and *k/2* is in the array (once or multiple times)?
    c. Given an ordered array $x_1...x_n$, in which the elements are distinct integers, determine whether there is a position such that A[i] = i. Show that this can be done with O($\log_2$ n) complexity.

7. Analyze the time complexity of the following algorithm:

```
subalgorithm s6(n) is:
      for i ← 1,n execute
            @elementary operation
      end-for
      i ← 1
      k ← true
      while i <= n – 1 and k execute
            j ← i
            k₁ ← true
            while j <= n and k₁ execute
                  @ elementary operation (k₁ can be modified)
                  j ← j + 1
            end-while
            i ← i + 1
            @elementary operation (k can be modified)
      end-while
end-subalgorithm
```

Best Case: k, $k_1$ can become false after one iteration, but we still have the for loop from the beginning => Θ (n)

Worst Case: k, $k_1$ never becomes false, the while loops will behave as 2 for loops, going from I to n-1 and i to n.

$$T(n) = n + \sum_{i=1}^{n-1} \sum_{j=i}^{n} 1 = n + \sum_{i=1}^{n-1} n - i + 1 = n + \sum_{i=1}^{n-1} n - \sum_{i=1}^{n-1} i + \sum_{i=1}^{n-1} 1 =$$

$$n + n * (n-1) - \frac{n * (n-1)}{2} + n - 1 \,\epsilon\, \Theta\,(n^2)$$

Average case:

Let's consider first the inner while loop (the one with j and $k_1$). The number of operations depends on *i*, but let's assume that *i* is fixed (like a parameter). The while loop is executed until $k_1$ becomes false (or j becomes greater than n). This can mean 1,2, ..., n-i+1 iterations =>

Probability: $\frac{1}{n-i+1}$

$$\frac{1}{n-i+1} + \frac{2}{n-i+1} + \ldots + \frac{n-i+1}{n-i+1} = \frac{(n-i+1)*(n-i+2)}{2\,(n-i+1)} = \frac{(n-i+2)}{2}$$

So this is the average number of operations of the inner while for a fixed *i*.

Let's see now the external while loop. This while loop runs until *k* becomes false or *j* becomes equal to *n*. This means 1, 2, …, n-1 iterations => Probability: $\frac{1}{n-1}$

Remember, formula for average case was:

$$\sum_{I \in D} P(I) * E(I)$$

E(I) – number of instructions for input I – is made of two parts:

- The average number of instructions of the inner while loop (marked with green), but now with the value of i is no longer fixed (we will know that i is 1, 2, 3, …, n-1)
- The number of times the instructions in the first while loop, but not in the second (marked with blue) are executed.

```
while i <= n - 1 and k execute
      j ← i
      k₁ ← true
      while j <= n and k₁ execute
            @ elementary operation (k₁ can be modified)
            j ← j + 1
      end-while
      i ← i + 1
      @elementary operation (k can be modified)
end-while
```

$$T(n) = \frac{1}{n-1} * \frac{n-1+2}{2} + \frac{2}{n-1} * \frac{n-2+2}{2} + \ldots + \frac{n-1}{n-1} * \frac{n-(n-1)+2}{2} =$$

$$\frac{1}{2*(n-1)} * \sum_{i=1}^{n-1} i*(n-i+2)$$
$$= (do\ the\ multiplication\ in\ the\ sum\ and\ split\ in\ 3\ different\ sums) \ldots$$
$$= \frac{1}{2*(n-1)} * \left(\frac{n*(n-1)*n}{2} - \frac{(n-1)*n*(2n-1)}{6} + 2*\frac{(n-1)*n}{2}\right)$$
$$= \frac{1}{2} * \left(\frac{n^2}{2} - \frac{2*n^2 - n}{6} + n\right) = \frac{1}{2} * \left(\frac{3n^2 - 2n^2 + 7n}{6}\right) \in \Theta(n^2)$$

Total complexity: $O(n^2)$

8. Analyze the time complexity of the following recursive algorithm:

```
subalgorithm p(x,s,d) is:
   if s < d then
         m ← [(s+d)/2]
         for i ← s, d-1, execute
               @elementary operation
         end-for
         for i ← 1,2 execute
               p(x, s, m)
         end-for
   end-if
end-subalgorithm
```

```
Initial call for the subalgorithm: p(x, 1, n)
```

- In case of recursive algorithms, the first step of the complexity computation is to write the recurrence relation.

$$T(n) = \begin{cases} 2*T\left(\frac{n}{2}\right) + n, if\ n > 1 \\ 0, \quad else \end{cases}$$

$$T(n) = 2T\left(\frac{n}{2}\right) + n$$

Assume: $n = 2^k$

$$T(2^k) = 2*T(2^{k-1}) + 2^k$$
$$2*T(2^{k-1}) = 2^2*T(2^{k-2}) + 2^k$$
$$2^2*T(2^{k-2}) = 2^3*T(2^{k-3}) + 2^k$$
$$\ldots$$
$$2^{k-1}*T(2) = 2^k*T(1) + 2^k$$

Add them up (many terms will simplify, because they appear on the left hand side of one equation and right hand side of another equation):

$$T(2^k) = 2^k * T(1) + k * 2^k = k * 2^k = n * log_2 n \rightarrow T(n) \in \Theta(n \, log_2 n)$$

9. Analyze the time complexity of the following algorithm:

```
Subalgorithm s7(n) is:
      s ← 0
      for i ← 1, n² execute
            j ← i
            while j ≠ 0 execute
                  s ← s + j
                  j ← j - 1
            end-while
      end-for
end-subalgorithm
```

While loops can be written as sum as well, if the loop variable changes by 1 in every iteration.

$$T(n) = \sum_{i=1}^{n^2} \sum_{j=1}^{i} 1 = \sum_{i=1}^{n^2} i = \frac{n^2 * (n^2 + 1)}{2} \in \Theta(n^4)$$

10. Analyze the time complexity of the following algorithm:

```
Subalgorithm s8(n) is:
      s ← 0
      for i ← 1, n² execute
            j ← i
            while j ≠ 0 execute
                  s ← s + j - 10 * [j/10]
                  j ← [j/10]
            end-while
      end-for
end-subalgorithm
```

- The *while* loop is repeated $log_{10} i$ times (but we report logarithmic complexities in base 2)
- So we will have: $log_2 1 + log_2 2 + log_2 3 + \ldots + log_2 n^2 = log_2 (n^2)!$
- Striling's approximation tells us that: $log_2 x! = x * log_2 x$
- $log_2 (n^2)! = n^2 * log_2 n^2 = 2 * n^2 * log_2 n$ – constants are ignored
- $T(n) \in \Theta (n^2 \, log_2 n)$