

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  $\leftarrow$  1, n execute
        j  $\leftarrow$  n
        while j  $\neq$  0 execute
            j  $\leftarrow$   $\left[ \frac{j}{2} \right]$ 
        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  $i \leftarrow 1, n$  execute
         $j \leftarrow i$ 
        while  $j \neq 0$  execute
             $j \leftarrow \lfloor \frac{j}{2} \rfloor$ 
        end-while
    end-for
end-subalgorithm
```

- The *for* loop is repeated n times.
- The *while* loop is repeated $\log_2 i$ times.
- $T(n) = \log_2 1 + \log_2 2 + \log_2 3 + \dots + \log_2 n = \log_2 n! \Rightarrow n \log_2 n$ (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  $\leftarrow$  false
    for  $i \leftarrow 1, n$  execute
        if  $x_i = a$  then
            found  $\leftarrow$  true
        end-if
    end-for
end-subalgorithm
```

$BC: \theta(n)$
 $WC: \theta(n)$

```
subalgorithm s4( $x, n, a$ ) is:
    found  $\leftarrow$  false
     $i \leftarrow 1$ 
    while found = false and  $i \leq n$  execute
        if  $x_i = a$  then
            found  $\leftarrow$  true
        end-if
         $i \leftarrow i + 1$ 
    end-while
end-subalgorithm
```

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

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 $x_i \leq n$):

Subalgorithm s5(x, n) **is:**

```

k ← 0
for i ← 1, n execute
    for j ← 1,  $x_i$  execute
        k ← k +  $x_j$ 
    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 is the number of instructions in the for loop).

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

$$T(n) \in \Theta(s)$$

- 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:

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

In this case: $s = \sqrt{n}$, but the complexity is $\Theta(n)$, because of the first for loop which will be executed n times, no matter what.

$$T(x, n) \in \Theta(\max\{n, s\}) = \Theta(n + s)$$

6. Consider the following problems and find an algorithm (having the required time complexity) to solve them :

- Given an arbitrary array with numbers $x_1 \dots x_n$, determine whether there are 2 equal elements in the array. Show that this can be done with $\Theta(n \log_2 n)$ time complexity.
- Given an arbitrary array with numbers $x_1 \dots x_n$, determine whether there are two numbers whose sum is k (for some given k). Show that this can be done with $\Theta(n \log_2 n)$ time complexity. What happens if k is even and $k/2$ is in the array (once or multiple times)?
- Given an ordered array $x_1 \dots x_n$, 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  $\leftarrow 1, n$  execute
    @elementary operation
  end-for
  i  $\leftarrow 1$ 
  k  $\leftarrow$  true
  while i  $\leq n - 1$  and k execute
    j  $\leftarrow i$ 
    k1  $\leftarrow$  true
    while j  $\leq n$  and k1 execute
      @ elementary operation (k1 can be modified)
      j  $\leftarrow j + 1$ 
    end-while
    i  $\leftarrow i + 1$ 
    @elementary operation (k can be modified)
  end-while
end-subalgorithm
```

Best Case: k, k₁ can become false after one iteration, but we still have the for loop from the beginning => $\Theta(n)$

Worst Case: k, k₁ never becomes false, the while loops will behave as 2 for loops, going from 1 to n-1 and i to n.

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

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

$$T_{\text{inner}}(n, i) = \frac{1}{n-i+1} + \frac{2}{n-i+1} + \dots + \frac{n-i+1}{n-i+1} = \frac{(n-i+1)*(n-i+2)}{2(n-i+1)} = \frac{(n-i+2)}{2}$$

So $T_{\text{inner}}(n, i)$ is the average number of operations of the inner while for a fixed i .

We will see three different ways of computing the complexity of the outer while loop (obviously, they will all lead to the same result). Let's see the loop again:

```
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
```

The outer while loop is made of two *types* of instructions (colored in two different colors): the inner while loop (the green part) and the instructions which are part of the first while loop, but not the second (the blue ones). For the green part we will work with the average case computed before. The blue instructions are all elementary instructions, so we have a constant number of instructions, so we will consider them in the computations as 1 instruction/repetition of the outer while loop.

V1

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:

- $T_{\text{inner}}(n, i)$, the average number of instructions of the inner while loop (marked with green), but now 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.

$$\begin{aligned}
T_{outer}(n) &= \frac{1}{n-1} * \frac{n-1+2}{2} + \frac{2}{n-1} * \frac{n-2+2}{2} + \dots + \frac{n-1}{n-1} * \frac{n-(n-1)+2}{2} = \\
&= \frac{1}{2 * (n-1)} * \sum_{i=1}^{n-1} i * (n-i+2) \\
&= (\text{do the multiplication in the sum and split in 3 different sums}) \dots \\
&= \frac{1}{2 * (n-1)} * \left(\frac{n * (n-1) * n}{2} - \frac{(n-1) * n * (2n-1)}{6} + 2 * \frac{(n-1) * n}{2} \right) \\
&= \frac{1}{2} * \left(\frac{n^2}{2} - \frac{2 * n^2 - n}{6} + n \right) = \frac{1}{2} * \left(\frac{3n^2 - 2n^2 + 7n}{6} \right) \in \Theta(n^2)
\end{aligned}$$

-----V2-----

Let's look at the outer while loop. It can be executed once, twice, 3 times, ..., a total of $n-1$ times. This means that the probability of one case is $\frac{1}{n-1}$

Now, assume that the outer while loop will be executed p times (where p is a value between 1 and $n-1$, inclusive). This means that variable i will have the value 1, 2, 3, ..., p in these iterations. In every iteration we will execute:

- The code written with blue once
- The code written with green, $T(n,i)$ times on average (where this time we have values for i).

So, for an outer loop executed p times, we will have the following number of instructions:

- The code written with blue a total of p times (and since that is a constant number of instructions, we can simply consider them as p).
- The code written with green, $T(n,1) + T(n,2) + \dots + T(n, p)$ times, which is:

$$\begin{aligned}
\sum_{i=1}^p T(n, i) &= \sum_{i=1}^p \frac{n-i+2}{2} = \frac{1}{2} * (p * n - \frac{p * (p+1)}{2} + 2 * p) = \frac{1}{4} * (2 * p * n - p^2 - p + 4 * p) = \\
&= \frac{p * (2 * n + 3) - p^2}{4}
\end{aligned}$$

If we add the p instructions in blue as well, we will get:

$$\frac{p * (2n + 7) - p^2}{4}$$

as the number of instructions, when the outer while loop is executed p times.

p can have a value between 1 and $n-1$ and the probability of having either of them is: $\frac{1}{n-1}$ so:

$$T_{outer}(n) = \frac{1}{n-1} \sum_{p=1}^{n-1} \frac{p * (2n+7) - p^2}{4} =$$

(do the multiplication and split into 3 different sums) =

$$\begin{aligned} \frac{1}{4 * (n-1)} \left(\frac{2n * (n-1) * n}{2} + \frac{7 * (n-1) * n}{2} - \frac{(n-1) * n * (2n-1)}{6} \right) &= \frac{6n^2 + 21n - 2n^2 + n}{24} \\ &= \frac{4n^2 + 22n}{24} \in \theta(n^2) \end{aligned}$$

V3

Let's look again at what happens in the outer while loop. This loop can be executed the following number of times (value on the left) and for a given number of executions i will have the following values (on the right side):

1	i: 1	and we execute the blue instructions once
2	i: 1, 2	and we execute the blue instructions twice
3	i: 1, 2, 3	and we execute the blue instructions three times
...		
n-1	i: 1, 2, 3, ..., n-1	and we execute the blue instructions (n-1) times

Now look at it by columns. Considering all executions together (what we need for average case), we see that i will have the value 1 a total of n-1 times (once for every execution), so we will have $T_{inner}(n, 1)$ n-1 times. Similarly, i will have the value 2 n-2 times (once for every execution except for the first), we will have $T_{inner}(n, 2)$ n-2 times. In general, we will have $T_{inner}(n, k)$ n-k times (where k goes from 1 to n-1).

Regarding the blue instructions, we also have +k instructions for every case.

So, adding the probability of one case, on average we will have:

$$\begin{aligned} T_{outer}(n) &= \frac{1}{n-1} \sum_{k=1}^{n-1} (T(n, k) * (n - k) + k) = \\ \frac{1}{n-1} \sum_{k=1}^{n-1} \left(\frac{(n - k + 2)(n - k)}{2} + k \right) &= \frac{1}{2 * (n-1)} \sum_{k=1}^{n-1} (n^2 - nk + 2n - nk + k^2 - 2k + 2k) = \end{aligned}$$

$$\begin{aligned}
&= \frac{1}{2(n-1)} * \sum_{k=1}^{n-1} (n^2 - 2nk + 2n + k^2) \\
&= \frac{1}{2 * (n-1)} * (n^2 * (n-1) - \frac{2n * (n-1) * n}{2} + 2n * (n-1) \\
&\quad + \frac{(n-1) * n * (2n-1)}{6}) = \frac{1}{2} * \left(n^2 - \frac{2n^2}{2} + 2n + \frac{n * (2n-1)}{6} \right) = \frac{11n + 2n^2}{12} \\
&\in \theta(n^2)
\end{aligned}$$

No matter how we compute the average complexity, we have the same result.

Consequently, the total complexity is $O(n^2)$.

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.

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

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

Assume: $n = 2^k$

$$\begin{aligned}
T(2^k) &= 2 * T(2^{k-1}) + 2^k \\
2 * T(2^{k-1}) &= 2^2 * T(2^{k-2}) + 2^k \\
2^2 * T(2^{k-2}) &= 2^3 * T(2^{k-3}) + 2^k \\
&\dots
\end{aligned}$$

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

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: $\log_2 1 + \log_2 2 + \log_2 3 + \dots + \log_2 n^2 = \log_2 (n^2)!$
- Stirling'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
- $T(n) \in \Theta(n^2 \log_2 n)$