HOMEWORK #2

Q1) a) $\lim_{n\to\infty} \frac{f(n)}{g(n)} = \frac{(n^2-3n)^2}{5n^3+n} = \frac{\infty}{\infty}$

L'hospital $\rightarrow \lim_{n\to\infty} \frac{(n^4-6n^3+9n^2)'}{(5n^3+n)'} = \frac{4n^3+18n^2+18n}{15n^2+1} = \frac{\infty}{\infty}$

L'hospital $\rightarrow \lim_{n\to\infty} \frac{(4n^3+18n^2+18n)'}{(15n^2+1)'} = \frac{12n^2-36n+18}{30n} = \frac{\infty}{\infty}$

L'hospital $\rightarrow \lim_{n\to\infty} \frac{(12n^2-36n+18)'}{(30n)'} = \frac{24n-36}{30} = \infty$

$f(n) \in \Omega(g(n))$

b) $\lim_{n\to\infty} \frac{f(n)}{g(n)} = \frac{n^3}{\log_2 n^4} = \frac{\infty}{\infty}$

L'hospital $\rightarrow \lim_{n\to\infty} \frac{(n^3)'}{(\log_2(n^4))'} = \frac{3n^2}{\frac{4n^3}{n^4}\log_2 e} = \frac{3n^2 \cdot n}{4\log_2 e} = \frac{3n^3}{4\log_2 e} = \infty$

$f(n) \in \Omega(g(n))$

c) $\lim_{n\to\infty} \frac{f(n)}{g(n)} = \frac{5n\log_2(4n)}{n\cdot\log_2(5^n)} = \frac{\infty}{\infty}$

L'hospital $\rightarrow \lim_{n\to\infty} \frac{(5n\log_2(4n))'}{(n\cdot\log_2(5^n))'} = \frac{5\log_2(4n) + 5n\cdot\frac{4}{4n}\cdot\log_2 e}{\log_2(5^n) + n\frac{5^n\ln 5}{5^n}\cdot\log_2 e} = \frac{\infty}{\infty}$

L'hospital $\rightarrow \lim_{n\to\infty} \frac{(5\log_2(4n) + 5\log_2 e)'}{(\log_2(5^n) + n\cdot\ln 5\log_2 e)'} = \frac{5\cdot\frac{4}{4n}\cdot\log_2 e}{\frac{5^n\ln 5}{5^n}\cdot\log_2 e + \ln 5\cdot\log_2 e}$

$= \frac{\frac{5}{n}}{2\ln 5} = \frac{5}{2\ln 5\cdot n} = 0$ //

$f(n) \in O(g(n))$

d) $\lim_{n\to\infty} \frac{f(n)}{g(n)} = \frac{n^n}{10^n} = \frac{\infty}{\infty}$

L'hospital → $\lim_{n\to\infty} \left(\left(\frac{n}{10}\right)^n\right)' = \frac{1}{10} \cdot \left(\frac{n}{10}\right)^n \ln\left(\frac{n}{10}\right)$

$= \frac{n^n \cdot \ln\left(\frac{n}{10}\right)}{100} = \infty$

$f(n) \in \Omega(g(n))$

e) $\lim_{n\to\infty} \frac{f(n)}{g(n)} = \frac{8n \cdot \sqrt[5]{2n}}{n \cdot \sqrt[3]{n}} = \frac{\infty}{\infty}$

Simplification → $\frac{8n \cdot \sqrt[5]{2n}}{n \cdot \sqrt[3]{n}} = \frac{8 \cdot 2^{1/5} \cdot n^{1/5}}{n^{1/3}} = 8 \cdot 2^{1/5} \cdot n^{1/5} \cdot n^{-1/3}$

$= 8 \cdot 2^{1/5} \cdot n^{-2/15}$

$= \frac{8 \cdot 2^{1/5}}{n^{2/15}}$

$\lim_{n\to\infty} \frac{8 \cdot 2^{1/5}}{n^{2/15}} = \frac{8 \cdot 2^{1/5}}{\infty} = 0$ //

$f(n) \in O(g(n))$

Q2 a) In this method, we iterate over each element of the array "str_array" and assign an empty string to each element. The time complexity of this operation is O(1).

```
str_array[i] = "";  → O(1)
```

However, we are performing this operation "str_array.length" times since we iterate over the entire array. Therefore, the overall time complexity of this method is O(n), where n is the length of the array "str_array".

```
str_array[i] = "";  → O(1)
str_array[i] = "";  → O(1)
        ⋮
str_array[i] = "";  → O(1)
```

} n times (n is str_array length)

time complexity is O(n)

So, the worst-case time complexity is O(n).

b) The first loop iterates over the "str_array" and for each iteration, it calls "methodA". The time complexity of "methodA" is O(n), and this loop iterates "n" times. Therefore, the time complexity of this part is $O(n^2)$.

```
methodA(str_array);  → O(n)
methodA(str_array);  → O(n)
        ⋮
methodA(str_array);  → O(n)
```

} n times (n is str_array.length)

time complexity is $O(n^2)$

The second loop iterates over the "str_array" and prints each elements. Printing each element takes constant time, so the time complexity of this operation is O(1). However we are performing this operation "n" times, where n is str_array.length. Hence, the overall time complexity of this loop is O(n).

```
System.out.println(str_array[i]);  → O(1)
System.out.println(str_array[j]);  → O(1)
        ⋮
System.out.println(str_array[j]);  → O(1)
```

} n times

time complexity is O(n)

Therefore, the overall worst-case time complexity of "methodB" is dominated by the first loop, making it $O(n^2)$.

c) The outer loop iterates over the "str_array", and for each iteration, the inner loop iterates over the "str_array" as well. Inside the inner loop, "methodB" is called. The time complexity of "methodB" is $O(n^2)$, as analyzed previously.

The nested loops iterate over the entire array "str_array" for each element in "str_array". Since there are n elements in "str_array" and each loop iterates "n" times, the total number of iterations is $n \cdot n = n^2$.

```
methodB(str_array); → O(n^2)
methodB(str_array); → O(n^2)
        ⋮                      } n^2 times
methodB(str_array); → O(n^2)   } time complexity is O(n^4)
```

Therefore, the overall worst-case time complexity of "methodC" is $O(n^4)$.

d) The crucial point to notice here is the decrementing of the loop variable "i" inside the loop. This effectively makes the loop iterate over the same element again, leading to an infinite loop. While it might seem like the loop variable "i" is being decremented within the loop, it actually doesn't change the behavior because it gets incremented again in the loop header. This results in the loop running indefinitely, printing the same element infinitely.

Therefore, the worst-case time complexity of this method cannot be determined using conventional analysis methods because it results in an infinite loop, which is not a valid scenario for time complexity analysis. However, it's important to understand that this method would result in a runtime error or crash due to an infinite loop rather than providing any meaningful output.

e) The conditional statement "`if (str_array[i] == "")`" involves comparing the current element of the array to an empty string. This comparison is done in constant time, denoted by $O(1)$, because it's a simple comparison operation.

The loop iterates over each element of the array once, from index 0 to "`str_array.length - i`". This operation has a time complexity of $O(n)$, where n is the length of the "str_array".

```
if(str_array[i] == ' ') break; → O(1)
if (str_array[i] == " ") break; → O(1)
        ⋮
if(str_array[i] == " ") break; → O(1)
```

n times

time complexity is O(n)

The worst-case time complexity of this method is O(n).

Q3 a) Function maxDifferenceSorted(A):

Return A[length(A)-1] - A[0]

This algorithm simply calculates the difference between the last and first elements of the array, which gives the maximum difference since the array is sorted in ascending order.

The time complexity of this algorithm is O(1) because it performs a constant number of operations regardless of the size of the array.

A[length(A)-1] - A[0]

b) Function maxDifference(A)

```
min_element = A[0]
max_difference = A[1] - A[0]

For i from 1 to length(A)-1:
    max_difference = max(max_difference, A[i] - min_element)
    min_element = min(min_element, A[i])

Return max_difference
```

This algorithm maintains two variables: "min_element" to track the minimum element encountered so far, and "max_difference" to track the maximum difference between any two elements encountered so far. It iterates through the array once, updating these variables as necessary. At each step, it calculates the difference between the current element and "min_element" and updates "max_difference" if this difference is greater than the current "max_difference". Finally, it returns "max_difference", which represents the maximum difference between two elements in the array.

The time complexity of this algorithm is O(n), where n is the number of elements in the array. This is because the algorithm iterates through the array only once, performing constant time operations at each iteration.

```
max_difference = max(....) } O(1)
min_element    = min(....) }

max_difference = max(....) } O(1)
min_element    = min(....) }
        :                    } n times
max_difference = max(....) } O(1)
min_element    = min(....) }
```

Time complexity is O(n).