Exercise Session for Introduction
into Computer Science
(for non Informatics studies, TUM BWL)
(IN8005), WS 2021/2022

Sheet 12

- Homework are to be worked out independently.
- Submission is done via moodle.

  `https://www.moodle.tum.de/course/view.php?id=69095`
- Deadline for submission is 24:00 o clock on 13.2.2022.
- If you have any questions regarding your solution after comparison with the sample solutions, you may post them in a comprehensive way in the moodle forum.
- Each assignment is marked to give an estimation of its difficulty and/or necessary effort.
  - * Simple and little effort,
  - ** Difficult or at least requires some work,
  - *** Very Difficult or extensive.

Complexity

- Goal: Evaluate the execution time of an algorithm X on a conceptual level!

  Example: how long does it take to execute the BubbleSort algorithm?
- Problem: Actual execution time depends on
  - computer,
  - programming language, and
  - problem instance.
- Solution: Measure number of elementary operations (i.e. statements), instead of time for each instance.

  Let I denote the set of problem instances.

  The number of statements for instance $i \in I$ is denoted as $T_X(i)$.
- Problem: T_X is hard to determine
- Solution: Aggregation:
  - Group instances by their size and

    Let I_n denote the set of problem instances of size n.
  - Use aggregated measures (because the execution time of the instances in a set I_n is not identical.)
    - $T_X^{\text{worst}}(n) = \max_{i \in I_n} T_X(i)$
    - $T_X^{\text{average}}(n) = \frac{1}{|I_n|} \sum_{i \in I_n} T_X(i)$
    - $T_X^{\text{best}}(n) = \min_{i \in I_n} T_X(i)$

Most of the time, we are interested in the worst case, but not always.

Hence, if nothing else is said, the time complexity of X refers to T_X^{worst}.

- Problem:
  - Exact formulas for $T_X(n)$ are sophisticated
  - We are interested in the execution time of difficult problem instances.
- Solution
  - estimation, i.e. asymptotic analysis with Landau-Symbols.

Landau Symbols

- Definition:

  Let $f, g : \mathbb{N} \to \mathbb{R}^+$

  $O(f(n)) = \{g(n) \mid \exists c > 0, n_0 > 0 : \forall n \geq n_0 : g(n) \leq cf(n)\}$

  $\Omega(f(n)) = \{g(n) \mid \exists c > 0, n_0 > 0 : \forall n \geq n_0 : g(n) \geq cf(n)\}$

  $\Theta(f(n)) = O(f(n)) \cap \Omega(f(n))$

  Notation: $f(n) \in O(g(n))$ or $f(n) = O(g(n))$ (mathematically not correct, but very common).

Calculation Rules for Landau Symbols

- $c \cdot f(n) \in \Theta(f(n))$ for all $c > 0$
- $\mathcal{O}(f(n)) + \mathcal{O}(g(n)) = \mathcal{O}(f(n) + g(n))$
- $\mathcal{O}(f(n)) \cdot \mathcal{O}(g(n)) = \mathcal{O}(f(n) \cdot g(n))$
- $\mathcal{O}(f(n) + g(n)) \in \mathcal{O}(f(n))$, if $g(n) \in \mathcal{O}(f(n))$

Similar for Ω.

Dynamic Arrays

- If the required capacity of an array is not known when it is created, an array that can grow dynamically is needed.
- Simply creating huge arrays wastes storage.
- In the Java class library a dynamic array is called `ArrayList.`
- Each time an element is added and the capacity does not suffice,
  - a new array of double length is created, and
  - the new array replaces the old one.

  This step is called `reallocate`.
- The logical array length, i.e. the last element's index in the array + 1, is called size.

  Example:

0	1	2	3	4	5	6	7
5	4	2	3				

  length = 8, size = 4

Tutorial 12.1: Time Complexity *

Determine the time complexity of the following code snippets using the Landau Symbol $\mathcal{O}$

1.

```
System.out.println("Hi")
```

$O(1)$

2.

```
System.out.println("Hi")
System.out.println("Hi")
```

$O(1) + O(1) = O(2) \in O(1)$

3.

```
for(int i = 1; i <= n; i = i * 2) {
  System.out.println("Hi");
}
```

$O(1 + \log_2 n) \in O(\log n)$

4.

```
for(int i = 1; i <= n; i = i * 3) {
  System.out.println("Hi");
}
```

$O(1 + \log_3 n) = O(\log n)$

5.

```
for(int i = 0; i < n; i++) {
  System.out.println("Hi");
}
```

$O(1 + \sum_{i=0}^{n} i) = O(n)$

6.

```
for(int i = 0; i < n; i = i + 2)
  System.out.println("Hi");
}
```

$\in O(n)$

7.

```
for(int i = 0; i < n; i++) {
  for(int j = 1; j < n; j = j * 7) {
    System.out.println("Hi");
  }
}
```

$O(n \log n)$

8.

```
for(int i = 0; i < n; i = i + 3) {
  for(int j = 1; j < n; j = j * 5) {
    System.out.println("Hi");
  }
}
```

$O(\log_5 n) \cdot O(n)$
$\in O(n \log n)$

9.

```
for(int i = 0; i < n; i++) {
  for(int j = 0; j < n; j++) {
    System.out.println("Hi");
  }
}
```

$O(n^2)$

10.

```
for(int i = 0; i < n; i++) {
  for(int j = 0; j < n; j++) {
    for(int k = 0; k < n; k++) {
      System.out.println("Hi");
    }
  }
}
```

$O(n^3)$

Tutorial 12.2: Dynamic Arrays **

Implement a simple dynamic array for integers with a constructor that receives the initial capacity and a method to add elements. If the new size exceeds the inner capacity, double the inner capacity.

Homework 12.1: Time Complexity * (2P)

Determine the time complexity of the following code snippets using the Landau Symbol $\mathcal{O}$

1.

```
for(int i = 0; i < n; i++) {
    int j=1;
    while ( j < n ) {
        System.out.println("Hello");
        j = j * 3;
    }
    for (int k = 1; k < n; k++) {
        System.out.println("Hi");
    }
}
```

$O(n)$ $O(n)$ $O(n)$ $\in O(n^3)$

2.

```
for(int i = 0; i < n; i++) {
    for(int j = 1; j < n; j++) {
        if (j-i < n) {
            for (int k = 1; k < n; k = k * 2) {
                System.out.println("Hello");
            }
        } else {
            System.out.println("Hi");
        }
    }
}
```

$O(n)$ $O(n)$ $O(\log n)$ $O(1)$

if $\Rightarrow \in O(n^2 \log n)$

else $\Rightarrow \in O(n^2)$

$\in O(n^2 \log n + n^2)$

Homework 12.2: Dynamic Arrays - Downsizing ** (2P)

Extend the dynamic array of Tutorial 12.2 (you can use your implementation or copy tutorial's solution ☺) and add a method `downsize()`.
If the size of the array is less than half of the inner capacity, the method resizes the array halving the inner capacity. If the size of the array is greater or equal than the inner capacity, nothing has to be done.

HW 12.1

1. $\in O(n^3)$

2. $\in O(n^2 \log n + n^2)$

Homework 12.3: Dynamic Arrays - Remove ** (2P)

Extend the dynamic array of Homework 12.2 add a method `removeElem(int elem)`.
The method removes every occurrence of `elem` in the array. When removing an element, you must shift the elements after `it` by 1 to fill the empty space. All unallocated values must be 0 (including the one left free because of the shifting after a removal). If the size becomes less than half of the inner capacity, you must resize the array halving the inner capacity. You can call the `downsize()` method defined in Homework 12.2.
Test your code as follows:

1. create a new dynamic array,
2. filling it with `[1,1,1,2,3,1]`,
3. print the array,
4. call `removeElem(int 1)`on the array,
5. print the array again.