

Datastructures, Oefeningen Algoritmes en Complexity

1. Geef aan welke time en space complexity deze algoritmes hebben. Ga voor de time complexity ervan uit dat elke instructie evenveel tijd in beslag neemt. Is het combinatie van meerdere verlopen, neem dan steeds de hoogste complexity.

Algoritme	tijdsverloop	Time complexity	Geheugen verloop	Space complexity
<pre> 0 references long CalculateSum(int[] list) { long sum = 0; for (int f = 0; f < list.Length; f++) { sum += list[f]; } return sum; } </pre>	<p>Lineair</p> <p>(for lus duurt langer als in de input vergroot)</p>	O(n)	<p>Constant</p> <p>(2 var , type int)</p>	O(1)
<pre> 0 references bool IsSumGreaterThan100(int[] list) { long sum = 0; for (int f = 0; f < list.Length; f++) { sum += list[f]; if (sum > 100) return true; } return false; } </pre>	<p>Lineair</p> <p>(vergeet niet dat O handelt over "worst case", in het slechtste geval wordt telkens de volledige lijst doorlopen)</p>	O(n)	Constant	O(1)

<pre> 0 references double CalculateAverage(int[] list) { long sum = 0; for (int f = 0; f < list.Length; f++) { sum += list[f]; } long numberOfElements = 0; for (int g = 0; g < list.Length; g++) { numberOfElements++; } var average = (double)sum / (double)numberOfElements; return average; } </pre>		<p>Lineair</p> <p>(er zijn 2 for lussen die vergroten in functie van de input, maar het blijft een lineair verloop omdat ze na elkaar komen)</p>	$O(n)$	<p>Constant</p> <p>(5 variabelen van een basic type, de hoeveelheid geheugen verbruik wordt bepaald door het type en staat los van de input)</p>	$O(1)$
<pre> long CalculateSpecialSum(int[] list) { long specialSum = 0; for (int f = 0; f < list.Length; f++) { for (int g = 0; g < list.Length; g++) { specialSum += list[g]; } } return specialSum; } </pre>		<p>Kwadratisch</p> <p>(er zijn 2 for lussen in elkaar-nested loops – dus de tijd loopt $length * length$ op)</p>	$O(n^2)$	<p>Constant</p>	$O(1)$
<pre> long CalculateSpecialSumVersion2(int[] list) { long specialSum = 0; for (int f = 0; f < 100; f++) { for (int g = 0; g < list.Length; g++) { specialSum += list[g]; } } return specialSum; } </pre>		<p>Lineair</p> <p>(2 nested for loops, maar de buitenste heeft een constante tijd, ze gaat steeds 100 iteraties doen, dus enkel de binnenste gaat verhogen)</p>	$O(n)$	<p>Contant</p>	$O(1)$

<pre> 0 references bool TestIfEnoughMemoryIsPresent(int size) { double[] temp = new double[size]; int counter = 0; for (int i = 0; i < temp.Length; i++) { temp[i] = counter++; } return true; } </pre>		Lineair	$O(n)$	Lineair (er wordt een array aangemaakt die groter wordt als de input waarde verhoogt)	$O(n)$
<pre> bool TestIfEnoughMemoryIsPresentV2 (int size) { double[,] temp = new double[size, size]; int counter = 0; for (int j = 0; j < temp.GetLength(0); j++) { for (int i = 0; i < temp.GetLength(1); i++) { temp[j, i] = counter++; } } return true; } </pre>		Kwadratisch (2 nested for loops)	$O(n^2)$	Kwadratisch (de temp array is een 2-dim array die een grootte (size*size) krijgt en dus kwadratisch groter wordt in functie van de input)	$O(n^2)$
<pre> bool TestIfEnoughMemoryIsPresentV3 (int size) { double[,,,] temp = new double[size, size, size]; int counter = 0; for (int j = 0; j < temp.GetLength(0); j++) { for (int i = 0; i < temp.GetLength(1); i++) { for (int k = 0; k < temp.GetLength(2) - temp.GetLength(1) + 1; k++) { temp[j, i, k] = counter++; } } } return true; } </pre>		Kwadratisch verloop (3 nested loops, waarvan er echter maar 2 verlengen in functie van de input, de 3 ^e heeft een constant tijdsverloop)	$O(n^2)$	Derde machtsverloop(cubic) Aangezien er een 3-dim array wordt aangemaakt die vergroot met de input volgens size*size*size	$O(n^3)$

<pre> 0 references double TakeRandomElement(double[] list) { double[] temp = new double[list.Length * list.Length]; var r = new Random().Next(list.Length); return list[r]; } </pre>		Constant verloop	$O(1)$	Kwadratisch verloop	$O(n^2)$
Bekijk de complexity van jouw stack versie 1, <ul style="list-style-type: none"> • Methode push • Methode pop 		Beide een constant verloop	$O(1)$	Beide een constant verloop	$O(1)$
Bekijk de complexity van jouw stack versie 2 <ul style="list-style-type: none"> • Methode push • Methode pop 		Push: lineair Pop: constant	$O(n)$ $O(1)$	Push: lineair Pop: constant	$O(n)$ $O(1)$
Bekijk de complexity van jouw lineaire queue versie 1 <ul style="list-style-type: none"> • Enqueue • Dequeue 		Enqueue: constant Dequeue: lineair	$O(1)$ $O(n)$	Beide Constant	$O(1)$ $O(1)$
Bekijk de complexity van jouw lineaire queue versie 2 <ul style="list-style-type: none"> • Enqueue • Dequeue 		Enqueue: lineair Dequeue: lineair	$O(n)$ $O(n)$	Enqueue: lineair Dequeue: constant	$O(n)$ $O(1)$
Bekijk de complexity van jouw circulaire queue versie 1 <ul style="list-style-type: none"> • Enqueue • Dequeue 		Beide Constant	$O(1)$	Beide Constant	$O(1)$
Bekijk de complexity van jouw circulaire queue versie 2 <ul style="list-style-type: none"> • Enqueue • Dequeue 		Enqueue : Lineair Dequeue: Constant	$O(n)$ $O(1)$	Enqueue: Lineair Dequeue: Constant	$O(n)$ $O(1)$