Aufgabenblatt 3 - O-Notation

Aufgabe 1: Komplexitätsklassen

Füllen Sie die folgende Tabelle mit den unterschiedlichen Funktionswerten für die angegebenen Werte für n aus,
mindestens so weit Ihr Taschenrechner reicht. Welche Schlussfolgerungen können Sie aus den Werten in der Tabelle auf
die Laufzeit von Algorithmen ziehen?

N	log_2 (n)	√n	n* log_2 (n)	n^2	n^3	2^n	3^n
16	4	4	64	256	4096	65536	43046721
64	6	8	384	4096	262144	18446744073709551616	343368382029251248465784908 9281
256	8	16	2048	65536	16777216	115792089237316195423570985008687907853269984665640564039457584007913129639936	139008452377144732764939786789661303114218850808529137991604824430036072629766435941001769154109609521811665540548899435521

N	log_2 (n)	√n	n* log_2 (n)	n^2	n^3	2^n	3^n
1024	10	32	10240	1048576	1073741824	179769313 486231590 772930519 078902473 361797697 894230657 273430081 157732675 805500963 132708477 322407536 021120113 879871393 357658789 758814416 622492847 430639474 124377767 893424865 485276302 219601246 094119453 082952085 005768838 150682342 462881473 913110540 827237163 350510684 586298239 947245938 479716304 835356329 624224137 216	373391848 741020043 532959754 184866588 225409776 783734007 750636931 722079040 617265251 229993688 938803977 220468765 065431475 158108727 054592160 858581351 336982809 187314191 748594262 580938807 019951956 404285571 818041046 681288797 402925517 658012340 617298396 574731619 152386723 046235125 934896058 590588284 654793540 505936202 376547807 442730582 144527058 988756251 452817793 413352141 920744623 027518729 185432862 375737063 985485319 476416926 263819972 887006907 013899256 524297198 527698749 274196276 811060702 333710356 481

N	log_2 (n)	√n	n* log_2 (n)	n^2	n^3	2^n	3^n
2^20	20	1024	20971520	1099511627	1152921504606846976	∞	∞

Die Laufzeit des Algorithmus ist eine Kombination aus linearem und logarithmischen Aufwand und steigt etwas stärker als linear mit der Eingabegröße an. Moderateraufwand

Algorithmen mit logarithmischem Zeitaufwand (O(log n)) sind auch für große *n* effizient.

Algorithmen mit quadratischem Zeitaufwand (O(n^2)) sind für moderate *n* noch anwendbar, aber sie werden schnell unpraktikabel, wenn *n* wächst.

Algorithmen mit kubischem Zeitaufwand (O(n^3)) sind bereits für relativ kleine *n* ineffizient.

Exponentielle Algorithmen (O(2^n)) und (O(3^n)) sind extrem ineffizient und praktisch nicht anwendbar, sogar für kleine n.

Aufgabe 2: O-Notation

Zeigen Sie, dass die folgenden Aussagen wahr sind oder widerlegen Sie sie. Begründen Sie Ihre Entscheidung und geben Sie
bei wahren Aussagen ein geeignetes c und n0 an.

https://share.goodnotes.com/s/6nYE4EdnefcNgm0Z3eDaBH

1. $27 \in O(1)$
2. $\frac{n(n-1)}{2} \in O(n^2)$
3. $max(n^3, 10n^2) \in O(n^2)$
4. $logn \in \Omega(n)$
5. $2n + 4 \in \Theta(n)$

1.

27 ist eine Konstante daher war

2. $\frac{n(n-1)}{2} \in O(n^2)$

$\frac{n(n-1)}{2} = \frac{n^2 - n}{2}$ | *2

$n^2 - n \leq 2 * c * a^2$ | c= 1/2

$n^2 - n \leq n^2$ gilt wenn $n \geq 1$ und c = 1/2

Wahr da c und n_0 positiv sind

3. $max(n^3, 10n^2) \leq c * n^2$

$10n^2$ *Obergrenze*

$max(n^3, 10n^2) \leq 10n^2$

$n \geq 1$
$n_0 = 1$

Wahr

4. $\log n \in \Omega(n)$

$n \geq n_0$
$\log n \geq c \cdot 4$

N	$\log_2 n$
1	0
2	1
3	1,58496
4	2
4000	11,9658

nicht wahr da Bedingung nicht erfüllt

5. $2n + 4 \in \theta(n)$

$O(n)$
$2n + 4 \leq c \cdot n$ | -4
$2n \leq c \cdot n - 4$ | c = 3
$2n \leq 3 \cdot n - 4$

$n \geq 4$
$c = 3$

Wahr

$\Omega(n)$
$2n + 4 \geq c \cdot n$ | c = 1
$2n + 4 \geq 1 \cdot n$

$c = 1$
$n \geq 1$

Wahr

$\theta(n)$ = Wahr

Aufgabe 3: Codeanalyse

Bestimmen Sie zunächst die Anzahl elementarer Rechenschritte wie Vergleiche, Zuweisungen und arithmetischer Operationen
für die folgenden Codestücke in Abhängigkeit von der Anzahl n „beteiligter“ Elemente. Die zu betrachtenden
Elementarschritte sind jeweils angegeben. Geben Sie dann den Aufwand in O-Notation an.

1. Vertauschen zweier Feldelemente: (Anzahl der Zuweisungen?)

```
class CodeAnalysis {
   static void exchange(int[] array, int index1, int index2) {
       array[index1] = array[index1] + array[index2];
       array[index2] = array[index1] - array[index2];
       array[index1] = array[index1] - array[index2];
   }
}
```

Zuweisungen = 3
Anzahl elementarer Rechenschritte = 6

$O(n)$

2. Suche des Minimums in einem Array: (Anzahl der Vergleiche?)

```
class CodeAnalysis {
   static int minimum(int[] array) {
      int minimum = array[0];
      int index = 1;
      while (index < array.length) {
         if (array[index] < minimum) {
            minimum = array[index];
         }
         index++;
      }
      return minimum;
   }
}
```

Zuweisungen = 3
Vergleiche = 2
Anzahl elementarer Rechenschritte = 6

$O(n)$

3. Suche eines Wertes in einem sortierten Array: (Anzahl der Vergleiche?)

```
class CodeAnalysis {
   static int indexOfValue(int[] array, int value) {
      int from = 0;
      int to = array.length - 1;
      int middle;
      while (from <= to) {
         middle = (from + to) / 2;
         if (value < array[middle]) {
            to = middle - 1;
         } else if (value > array[middle]) {
            from = middle + 1;
         } else {
            return middle;
         }
      }
      return -1;
   }
}
```

Zuweisungen = 5
Vergleiche = 3
Anzahl elementarer Rechenschritte = 13

$O(logn)$

Aufgabe 4: O-Notation

Die folgende Funktion implementiert die Suche eines Wertes in einem sortierten Array. Bestimmen Sie zunächst die rekursive Aufwandsfunktion für die Anzahl der erforderlichen Vergleiche und bestimmen Sie dann die Komplexität der Funktion in O-Notation.

```
class CodeAnalysis {
   static int indexOfRecursive(int[] array, int value, int from, int to) {
      int middle;
      if (from <= to) {
         middle = (from + to) / 2;
         if (value < array[middle]) {
            return indexOfRecursive(array, value, from, middle - 1);
         } else if (value > array[middle]) {
            return indexOfRecursive(array, value, middle + 1, to);
         } else {
            return middle;
         }
      }
      return -1;
   }
}
```

$$t_n = at(\frac{n}{b}) + f(n)$$

$$t_n = 1t(\frac{n}{2}) + 1$$

a = 1
b = 2
f(n) = 1

$$O(logn)$$

N	t(n)	Log n
1	1	0
2	2	1
4	3	2
8	4	3
16	5	4
32	6	5
64	7	6