**NitscheSahoo - AD Praktikum für 17.10.13 – Aufgabe 3 – Lauzzeitanalyse**

Aufgabe 3.1:

In den Klammern hinter den Bezeichnungen in den Zeilen geben dar wie wir den Zeitaufwand dieser Operation im Code interpretieren. Eine Zuweisung braucht z.B. konstante Zeit. Eine Zuweisung für die Laufariable in einer For-schleife braucht so-viel Zeitaufwand, wie oft die Iteration emacht wird.

Algorithmus 1: Quersumme

1:Zuweisung(1)

2: Zuweisung für For-schleife (n)

3: Zuweisung(1) und Addition(1) und Dereferenzierung(1)

5: return(1)

Summe: 1 + n + n \* (1 + 1 + 1) + 1

= 2 + 4 \* n

Algorithmus 2: (namenslos)

1: Zuweisung für For-schleife (n)

2: Zuweisung(1) und Dereferenzierung(1)

4: Zuweisung für For-schleife (n)

5: Zuweisung(1) und Dereferenzierung(1)

6: Zuweisung für For-schleife(n)

7: If-Abfrage(1) und Dereferenzierungen(2)

8: Zuweisung(1) und Dereferenzierungen(2)

12:return(1)

Summe: ( n \* (1 + 1) ) + ( n \* ( 1 + 1 + n \* ( (1 + 2) + (1 + 2) ) ) ) + 1

= 2\*n + n\*(2+n\*6) + 1

= 2\*n + 6\*n² + 2\*n + 1

= 6\*n² + 4\*n + 1

Algorithmus 3: Matrixmultiplikation

1: Zuweisung für For-schleife (n)

2: Zuweisung für For-schleife (n)

3: Zuweisung(1) und Dereferenzierungen(1)\*

4: Zuweisung für For-schleife (n)

5: Zuweisung(1), Multiplikation(1) und Dereferenzierungen(3)

9: return(1)

\*Interpretation des Zugriffs auf einen Element eines 2dim-Arrays als einzelne Dereferenzierung trotz zwei Indizes

Summe: n \* ( n \* ( (1 + 1) + n \* ( 1 + 1 + 3) ) ) + 1

= n \* ( n \* ( 2 + n \* 5 ) ) + 1

= n \* ( 2\*n + 5\*n² ) + 1

= 5\*n³ + 2\*n² + 1

Algorithmus 4: Allgemeines Beispiel

1: Zuweisung für For-schleife (n)

2: Zuweisung für For-schleife (i)\*

3: Zuweisung(1), Addition(1) und Dereferenzierungen(1)

6: return(1)

\* Wir nehmen an, dort sollte „to“ stehen statt einem „downto“, da sonst der Algorithmus sinnlos wäre.

Summe: (1 + 2 + 3 + ... + n) \* 3

= 3 \* ( n \* (n + 1) ) / 2 (Gaussche Summenformel)

= 1.5\*n² + 1.5\*n

Die Drei in der ersten Zeile dieser Formeln kommt von den drei Operationen in der dritten Codezeile. Die aufsteigenden Zahlen der Summe sind eine Konsequenz der inneren For-shleiffe die i-viele Iterationen macht.

Aufgabe 3:

Erwartung: Wir erwarten, dass bei wachsender Potenz k, die Laufzeit(bzw. die Anzahl der Rekursiven Aufrufe) der herkömmlichen iterativen Variante linear steigt. Bei der alternativen/rekursiven Variante erwarten wir eine logarithmische Komplexität da bei jedem Rekursionsaufruf das Problem in zwei geteilt wird, wobei beide Teilprobleme identisch sind und daher nur einmal brechnet werden müssen.

Ergebnisse: Siehe Excel Tabelle Sheet „Aufgb3“ -> Aufgabe 3

Interpretation: Unsere Erwartung hat sich bestätigt. Die Laufzeit des iterativen Variant ist liner, und die der alternativen ist logarithmisch. Z.B. ist bei k = 32 die iterative Variante auch bei 32 Aufrufen, während die alternative Variante bei 7 liegt. (Hinweis: log(32) = 8) Feststellbar ist auch die Tatsache, dass die Anzahl der Aufrufe der alternativen Variante nur bei allen Zweierpotenzen steigt während die iterative Variante bei allen k steigt.

Aufgabe 4:

Erwartung: Wir erwarten, analog zu Aufgabe 3, dass der AccessCount bei der herkömmlichen Variante linear und bei der neuen Variante logarithmisch verläuft.

Ergebnisse: Siehe Excel Tabelle Sheet „Aufgb3“ -> Aufgabe 3 // TODO:

Interpretation: //TODO:

Aufgabe 3.3: O-Notation