Algorithmus Analyse

Ein Seminarbericht von Johann Basnakowski

Arbeitsbereich Wissenschaftliches Rechnen
Fachbereich Informatik
Fakultät für Mathematik, Informatik und Naturwissenschaften
Universität Hamburg

Name: **Johann Basnakowski**
Seminar: **Effiziente Programmierung in C**
Thema: **Algorithmus Analyse**
Betreuer: **Nathanael Hübbe**
Datum: **14.03.2014**

Inhaltsverzeichnis

- 1.0 Einleitung 3
- 2.0 Algorithmus Analyse 3
  - 2.1 Zeitkomplexität 4
  - 2.2 Platzkomplexität 5
  - 2.3 Große O-Notation 5-6
  - 2.4 Häufige Wachstumsraten 6-7
  - 2.5 Kostenarten 7-8
  - 2.6 Beispiel Analyse 8-10
    - 2.6.1 Insertion Sort 8-9
    - 2.6.2 Binary Search 9-10
  - 2.7 Zeitkomplexität vs. Platzkomplexität 11
  - 2.8 Nachteil 11-12
- 3.0 Abschlusswort 12
- 3.1 Quellen 13

1.0 Einleitung

Der nachfolgende Bericht behandelt die Algorithmus Analyse und ihre Methoden zur Bestimmung der Laufzeit verschiedener Algorithmen. In der Programmierung findet man häufig verschiedenste Algorithmen um das selbe Problem zu lösen. Aber welcher Algorithmus ist schneller oder besser für eine Situation geeignet. Es wird behandelt, wie die Algorithmus Analyse verwendet werden kann, um bestehendes Optimierungspotential an den eigenen Programmen zu erkennen und diese zu verbessern.
Wie genau bewertet man einen Algorithmus? Ist Algorithmus A immer besser als Algorithmus B? Welche Möglichkeiten habe ich meinen Code zu optimieren.
Diese und andere Fragen versucht der nachfolgende Bericht zu beantworten.

2.0 Algorithmus Analyse

Die Algorithmus Analyse ist die Bestimmung der Menge der Ressourcen, welche benötigt werden, um einen Algorithmus auszuführen. Die Algorithmus Analyse betrachtet einen Algorithmus unabhängig von der verwendeten Programmiersprache, in welcher ein Algorithmus implementiert wurde, unabhängig von der Geschwindigkeit des Computers und unabhängig von der Qualität des verwendeten Compilers. Somit wird sichergestellt, dass ein Algorithmus generell bewertet werden kann und es nicht zu unterschiedlichen Ergebnissen kommt, nur weil der Algorithmus anders implementiert wurde oder andere Hardware verwendet wurde. Hierbei wird angenommen das elementare Operationen wie Addition, Subtraction, Division und Multiplikation jeweils immer eine konstante Zeit benötigen, welche unabhängig von der Größe der Zahlen ist. Zusätzlich wird bei der Algorithmus Analyse zwischen der Zeitkomplexität und der Platzkomplexität unterschieden. Die Zeitkomplexität beschreibt die Anzahl der benötigten Rechensschritte während die Platzkomplexität den benötigten Speicherverbrauch beschreibt. Beides wird in der Abhängigkeit der Größe , der Eingabe angegeben, den ein Algorithmus braucht bekanntlich länger bei der Behandlung von 1000 Elementen, als bei der Behandlung von 10 Elementen.

2.1 Zeitkomplexität

Wie bereits oben erwähnt, bezeichnet die Zeitkomplexität die Anzahl der benötigten Rechenschritte, in abhängigkeit der Eingabe n. Um die Zeitkomplexität zu bestimmen, zählt man im ersten Schritt die Anzahl der Operationen und drückt im zweiten Schritt die Effizienz mittels Wachstumsraten aus. Zum zweiten Schritt kommen später noch, im nachfolgenden wollen wir und zunächst mit dem ersten Schritt beschäftigen. Zur Bestimmung der benötigten Rechenschritte betrachten wir nun einmal folgenden Beispiel Code:

```
11
12      int count = 3, sum = 2;
13
14      count = count + 1;          // Cost: c1
15      sum = sum + count;          // Cost: c2
16
```

Die Anzahl der Operationen ist hier zwei. Die benötigte Laufzeit dieses Programmes ist damit T(n) = c1 + c2.

T(n) ist hierbei die mathematische Beschreibung der benötigten Laufzeit und c1 bzw. c2 beschreiben die Kosten, welche benötigt werden, um die Operationen auszuführen.

Sehen wir uns ein weiteres Beispiel an:

```
 9
10
11      if (n < 0) {            // Cost: c1
12          absval = -n;        // Cost: c2
13      } else {
14          absval = n;         // Cost: c3
15      }
16
17
```

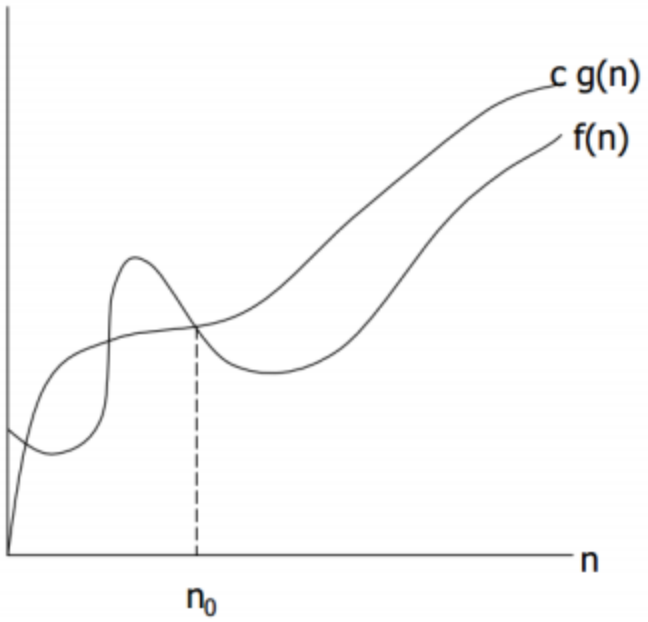
Hierbei beträgt die Zeitkomplexität T(n) = c1 + max(c2, c3). Die Bedingung der If-Bedingung wird nur einmal überprüft, danach wird einer der beiden Zweige ausgeführt. Angenommen die Kosten c2 und c3 sind unterschiedlich, dann würden, im schlimmsten Fall, die Kosten c1 + max(c2,c3) betragen.

2.2 Platzkomplexität

Wie bereits oben beschrieben, beschreibt die Platzkomplexität den benötigten Speicherbedarf , abhängig von der Eingabe n, eines Algorithmuses. Dies drücken wir mathematisch mit S(n) aus.
Die Platzkomplexität aber meist von geringem Interesse und wird daher hier auch nicht weiter ausformuliert. Dies liegt daran, dass die meisten Algorithmen nicht mehr Speicher benötigen, als ihre Eingabe groß ist, zusätzlich werden Speicherbausteine immer größer und immer preiswerter und deswegen spielt die Platzkomplexität eine immer weniger entscheide Rolle.

2.3 Große O-Notation

Kommen wir nun zum zweiten Teil der Zeikomplexität, der Zuordnung unseren Funktion zu einer Wachstumsrate. Hierzu bedienen wir uns eines Hilfsmittels, nämlich der Großen O-Notation. Die Große O-Notation besagt, dass die Funktion f zur Menge O(g) gehört, wenn es eine positive Konstante c und eine positive Konstante n0 gibt, so dass f(n) <= c * g(n) gilt.
Dies nochmal grafisch verdeutlicht:



- G. Zachmann

Vereinfacht ausgedrückt bedeutet dies, dass wir nur den dominanten Term aus unseren T(n) Funktion betrachten und alle Konstanten, welche nicht von n abhängen, ignorieren.

Betrachten wir dies nochmal an einem Code Beispiel:

```
 9
10      int k = 0;
11      for (int i = 1; i <= n/2; i++) {
12          for (int j = 1; j <= n; j++) {
13              k = k + i + j;
14          }
15      }
16
```

Bestimmen wir die Zeitkomplexität für diesen Code, so erhalten wir T(n) = (n/2) * (n*c1). c1 drückt hier die Kosten für die Additionszeile aus. Betrachten wir hiervon nur noch den dominanten Term, so erhalten wir O(n^2), da die /2 und das c1 konstanten Werten gleich kommt und diese bei der Großen O-Notation vernachlässigt werden. Somit ist die Zeitkomplexität dieses Algorithmuses O(n^2).

2.4 Häufige Wachstumsraten

Bei der Analyse von Algorithmen kommen manche Wachstumsraten häufiger vor als andere, die am häufigsten auftretenden Wachstumsraten sind:

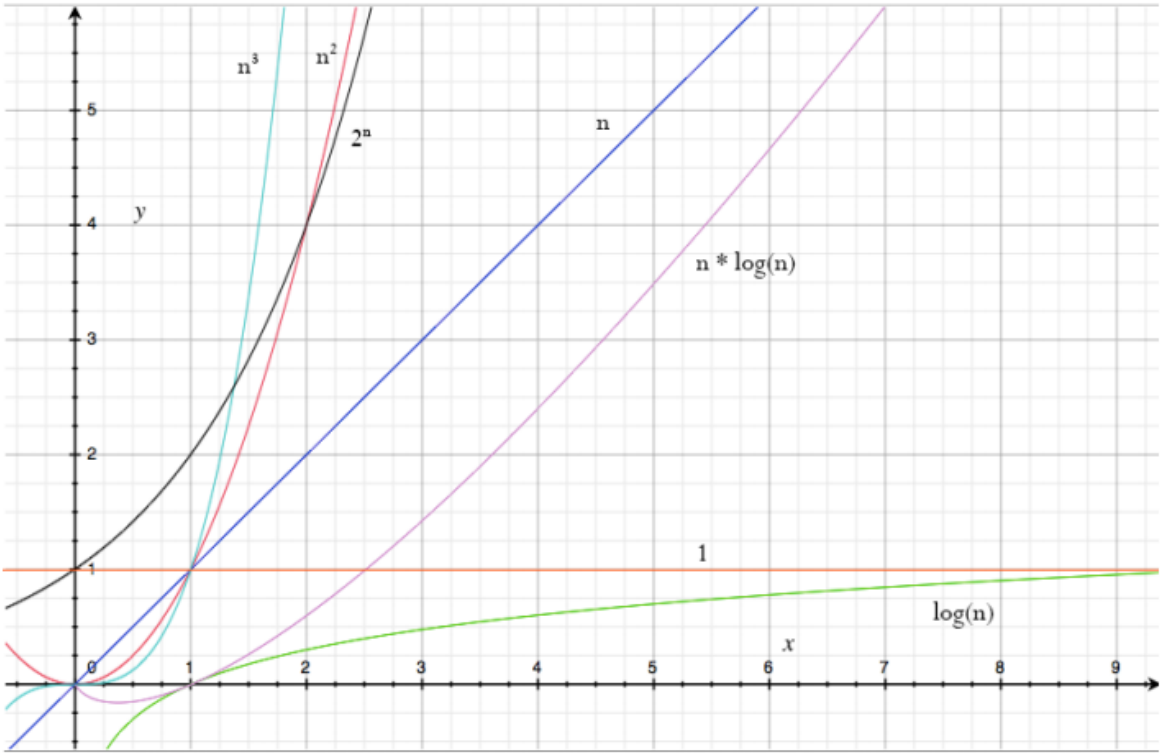
O(1)
O(log(n))
O(n)
O(n * log(n))
O(n^2)
O(n^3)
O(k^n)

O(1) bezeichnet Algorithmen , welche eine konstante Laufzeit, unabhänhig von der Eingabe, haben. Die nachfolgende Grafik zeigt die häufigsten Wachstumsraten nochmal auf. Offensichtlich zeigt die Grafik auf, dass die Laufzeit n^2 offensichtlich schlechter ist, als die von n, dies ist aber nicht immer der Fall. Betrachten wir zum Beispiel dieses stark vereinfachte
Beispiel:

A :	T(n) = 1000 + n	B:	T(n) = 2 * (n * n)
	T(n) = O(n)		T(n) = O(n^2)

Betrachten wir nun A und B so könnte man meinen, dass A besser ist als B. Für riesige Eingaben

wie 10000 ist durchaus wahr, aber für Eingabe wir zum Beispiel 5 ist dies nicht der Fall.
Es ist somit durchaus mit Vorsicht vorzugehen und sich zu überlegen, mit welchen Eingabegrößen arbeitet.



2.5 Kostenarten

Wir können einen Algorithmus auf verschiedene Fälle analysieren. Wir können den besten Fall betrachten, den schlechtesten Fall und den durchschnittlichen Fall und hierzu die Laufzeit bestimmen. Dies ist manchmal nötig, da die Laufzeit mancher Algorithmen von der Eingabe abhängt, z.b wie weit eine Zahlenkette sortiert ist. Offensichtlich wird die Laufzeit einiger Sortieralgorithmen besser sein, wenn man ihnen eine bereits sortierte Liste als Eingabe gibt.

Worst-Case Analyse: Die Worst-Case Analyse betrachtet Eingaben, welche die Laufzeit maximiert. Die Worst-Case Analyse stellt somit die obere Schranke der Laufzeit des Algorithmus dar. Die Worst-Case Analyse ist die am häufigste angewandte Analyse.

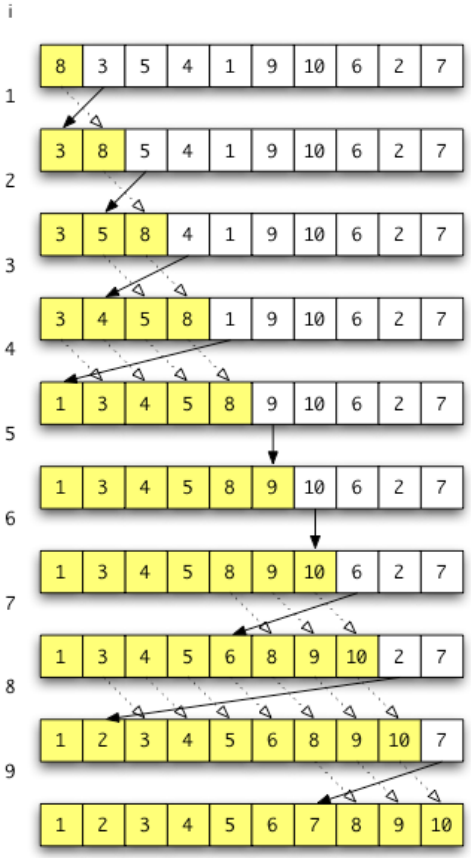
Average-Case-Analyse: Die Average-Case Analyse betrachtet die durchschnittliche Laufzeit für alle Eingaben der Größe n. Dabei wird eine Gleichverteilung auf alle Eingaben der Größe n vorausgesetzt. Die Average-Case Analyse ist in der Praxis sehr nützlich, aber meist technisch sehr schwer durchzuführen.

Best-Case-Analyse:
Die Best-Case Analyse betrachtet die bestmögliche Laufzeit für einen Algorithmus. Sie stellt damit die untere Schranke für einen Algorithmus dar. Die Best-Case Analyse wird eher selten angewendet.

2.6 Beispiel Analyse

Im nachfolgenden wollen wir eine Algorithmus Analyse auf zwei bekannte Algorithmen anwenden und deren Laufzeiten analysieren.

2.6.1 Insertion Sort



Betrachten wir nun einmal den bekannten Sortieralgorithmus Insertion Sort. Die Funktionsweise diese Algorithmuses ist schnell erklärt. Insertion Sort iteriert über sämtliche Einträge einer Liste und guckt dann im zweiten Schritt, ob die Zahlen die Links von ihm stehen, kleiner sind, als die Zahl, an dessen Position man sich befindet. Ist dies der Fall so werden die beiden Werte getauscht und die jetzige Position um 1 nach links verschoben. Dies wird so lange wiederholt, bis man wieder am Anfang der Liste ist, oder die Werte links von unserem Wert nicht größer ist, als unser Wert. Danach wird der nächste Eintrag in der Liste genommen und die ganze Prozedur wird wiederholt und zwar solange bis man jedes Element in der Liste einmal durch hat. Danach ist die Liste sortiert. Dies ist an dem Bild links nochmal einmal an einem Beispiel verdeutlicht.

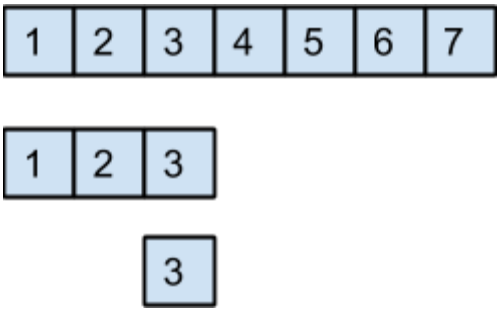
```
12
13      for(int i = 1; i < n; i++) {
14          temp = array[i];                        // Cost: c1
15          j = i;                                  // Cost: c2
16          while(j > 0 && array[j-1] > temp) {
17              array[j] = array[j-1];              // Cost: c3
18              j = j - 1;                          // Cost: c4
19          }
20          array[j] = temp;                        // Cost: c5
21      }
22
```

Hier ist nochmal die Hauptroutine des Insertion Sort in C - Code zu sehen.
Analysieren wir nun diesen Code, so stellen wir fest das zum einen die Liste mindestens einmal durchlaufen wird, also selbst wenn die Liste bereits sortiert ist, wird die Liste mindestens einmal durchlaufen, was uns zu einer Best-Case Laufzeit von O(n) bringt. Im Schlimmsten Fall ist die Liste genau verkehrt herum sortiert, somit muss bei jedem Durchlauf das Element von hinten nach vorne geschoben werden, was uns zu einer Worst-Case Laufzeit von O(n^2) bringt.
Der Average-Case ist etwas schwieriger zu ermitteln, er beträgt aber ebenfalls O(n^2).

Zusammenfassend betragen alle Laufzeiten dieses Algorithmuses:

T(n) =	n * (c1 + c2+ (n * (c3+ c4)) + c5)
Best-Case:	O(n)
Worst-Case:	O(n^2)
Average-Case:	O(n^2)

2.6.2 Binary Search



Betrachten wir nun einmal den Suchalgorithmus Binary Search. Binary Search benötigt immer eine sortierte Liste als Eingabe, andernfalls funktioniert der Algorithmus nicht. Der Algorithmus ermittelt zuerst den Mittelpunkt der Liste und betrachtet den Wert an dieser Position, ist der gesuchte Wert, der Wert der an dieser Position steht? Wenn ja, dann sind wir fertig und die Position wird zurückgegeben. Ist dies nicht der Fall, so gucken wir ob der gesuchte Wert größer oder kleiner ist und betrachten dementsprechend die Linke oder Rechte Seite der Liste von

unseren Position. Hiervon wird wieder der Mittelwert angesehen und die Prozedur wiederholt, so lange bis die Liste leer ist oder der Wert gefunden wurde. Dies ist auf dem linken Bild nochmal an einem Beispiel zu sehen.

```
11
12      int first = 0, last = n - 1;                            // Cost: c1
13      int middle = ( first+last ) / 2;                        // Cost: c2
14
15      while(first <= last) {
16          if (arrayToSearch[middle] == search) {              // Cost: c3
17              printf("Wert gefunden an position : %d\n", middle);
18              break;
19          } else if (arrayToSearch[middle] < search) {        // Cost: c4
20              first = middle + 1;                             // Cost: c5
21          } else {
22              last = middle - 1;                             // Cost: c6
23          }
24          middle = ( first+last ) / 2;                        //Cost: c7
25          if(first > last) {                                  //Cost: c8
26              printf("%d konnte nicht gefunden werden\n", search);
27              break;
28          }
29      }
30
```

Hier ist nochmal die Hauptroutine als C - Code zu sehen. Die Bestimmung der Laufzeit für diesen Algorithmus ist schon ein wenig schwerer. Der Algorithmus teilt im Prinzip die Liste immer in zwei Teile und diese Teile wieder in zwei und so weiter. Die Anzahl der Teilungsschritte ist das, was uns hier interessiert. Mit anderen Worten es gilt 2^h = n , es werden h Schritte benötigt bei einer Eingabe von n. Mit Log erhalten wir hier h = log(n). Mit anderen Worten es wird im Schlimmsten Fall log(n) Schritte benötigt, was uns eine Laufzeit von O(log(n)) gibt. Im besten Fall ist der Wert den wir suchen direkt in der Mitte der Liste, dann beträgt die Laufzeit O(1). Und der durchschnittliche Fall ist genauso wie der schlechteste Fall O(log(n)).

Zusammenfassend betragen alle Laufzeiten dieses Algorithmuses:

T(n) = c1+ c2+ (log(n) * (c3+ c4+ c5+ c6+ c7))
Best-Case: O(1)
Worst-Case: O(log n)
Average-Case: O(log n)

2.7 Zeitkomplexität vs. Platzkomplexität

```c
int fibTime (int x) {
    int a = 0, b = 1;
    int current = 1;
    for(int i = 0; i < x; i++) {
        current = a + b;
        a = b;
        b = current;
    }
    return current;
}
```

```c
int fib[10] = {1,1,2,3,5,8,13,21,34,55};
int fibSpace(int x) {
    return fib[x];
}
```

Eine Sache die definitiv noch angesprochen werden sollte, ist die Möglichkeit der Optimierung auf Kosten der Zeitkomplexität oder der Platzkomplexität. Der oben gezeigte Code ist ein sehr vereinfachtes Beispiel, eine Funktion noch anderweitig implementiert werden kann, um die Laufzeit seines Programms zu erhöhen. Falls mehr Speicher zur Verfügung steht, so können Werte zwischen gespeichert werden um Sie bei einem erneuten Aufruf nicht erneut berechnen zu müssen. Das selbe geht auch in die andere Richtung. Es ist also immer zu beachten, ob einem die Zeikomplexität oder die Platzkomplexität wichtiger ist.

2.8 Nachteil

Neben dem einen Nachteil, welcher bereits im Kapitel Wachtstumsraten angesprochen wurde, gibt es noch einen großen Nachteil bei der Algorithmus Analyse und zwar dem nicht erkennen von potentiellen Optimierungspotential. Betrachten wir hierzu folgendes Beispiel:

```c
double* foo (int n) {
    double (*table)[n][n] = malloc(sizeof(*table));
    for (int y = 0; y < n; y++) { for (int x = 0; x < n; x++) {
        (*table)[y][x] = sqrt(x*y); }
    }
    return &(*table)[0][0];
}
```

- Nathanael Hübbe

Analysieren wir diesen Code, so würden wir eine Laufzeit von O(n^2) ermitteln, da wir eine

Liste der Größe O(n^2) füllen, könnte man denken, dass die Code optimal ist, dies ist aber nicht so. Die meiste Zeit geht bei der Berechnung der Wurzel von X*Y drauf, mit anderen Worten, der folgende Code ist schneller.

```
20
21  double* foo2 (int n) { double sqrtLUT[n];
22      for (int i = 0; i < n; i++) {
23          sqrtLUT[i] = sqrt(i); }
24      double (*table)[n][n] = malloc( sizeof (*table) );
25      for (int y = 0; y < n; y++) { for (int x = 0; x < n; x++) {
26          (*table)[y][x] = sqrtLUT[x] * sqrtLUT[y]; }
27      }
28      return &(*table)[0][0];
29  }
30
```

- Nathanael Hübbe

Dieser Code ist etwa um einen Faktor 10 schneller. Leider erkennt die Algorithmus Analyse dieses Potential nicht. Jede Analyse ist somit mit Vorsicht zu genießen.

3.0 Abschlusswort

Dieser Bericht sollte einen guten Überblick über die Stärken und Schwächen der Algorithmus Analyse gegeben haben. Nur weil ein Algorithmus nach der Analyse schneller zu sein scheint, so heißt das nicht, dass dieser es auch für alle jede Eingabe ist. Code kann optimiert werden auf Kosten von mehr Speicher und vice versa und leider wird nicht jedes Optimierungspotenzial erkannt.

3.1 Quellen

Wikipedia
Analysis of algorithms
http://en.wikipedia.org/wiki/Analysis_of_algorithms
15:34, 05.01.2014

Unbekannt
Analysis of Algorithms
http://www.csd.uwo.ca/courses/CS1037a/notes/topic13_AnalysisOfAlgs.pdf
10:45, 05.01.2014

O. Bittel
Komplexitätsanalyse
http://www-home.fhkonstanz.de/~bittel/prog2/Vorlesung/05KomplexitaetsAnalyse.pdf
11:4 , 04.01.2014

G. Zachmann
Komplexität von Algorithmen
http://cgvr.cs.uni-bremen.de/teaching/info2_06/folien/11_komplexitaet_4up.pdf
19:23, 06.01.2014

Hr. Schlinger
InsertSort
http://www.sn.schule.de/~gym-ehrenberg-dz/info/12_history.htm
20:13, 14.03.2014