Algorithmus Analyse

Johann Basnakowski

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

Gliederung

- Algorithmus Analyse
- Zeitkomplexität
- Große O-Notation
- Kostenarten
- Häufige Wachstumsraten
- Insertion-Sort
- Binary Search
- Zeitkomplexität vs. Platzkomplexität
- Nachteile
- Zusammenfassung

Algorithmus Analyse

Definition

“ The analysis of algorithms is the determination of the amount of resources (such as time and storage) necessary to execute them. ”

http://en.wikipedia.org/wiki/Analysis_of_algorithms

Algorithmus Analyse

Zeitkomplexität:

- Benötigte Laufzeit
- Mathematische Funktion T(n) abhängig von der Eingabe n

Platzkomplexität :

- Benötigter Speicherbedarf
- Mathematische Funktion S(n) abhängig von der Eingabe n

Algorithmus Analyse

Unabhängig von:

- Der benutzten Programmiersprache
- Der Qualität des Compilers
- Der Geschwindigkeit des Computers

Zeitkomplexität

1. Zählen der Operationen
2. Angabe der Effizienz mittels Wachstumsraten

Zeitkomplexität

Code

```
int count = 3, sum = 2;
count  =  count  + 1;          //      Cost: c1
sum = sum + count;             //      Cost: c2
```

$T(n) = c_1 + c_2$

Zeitkomplexität

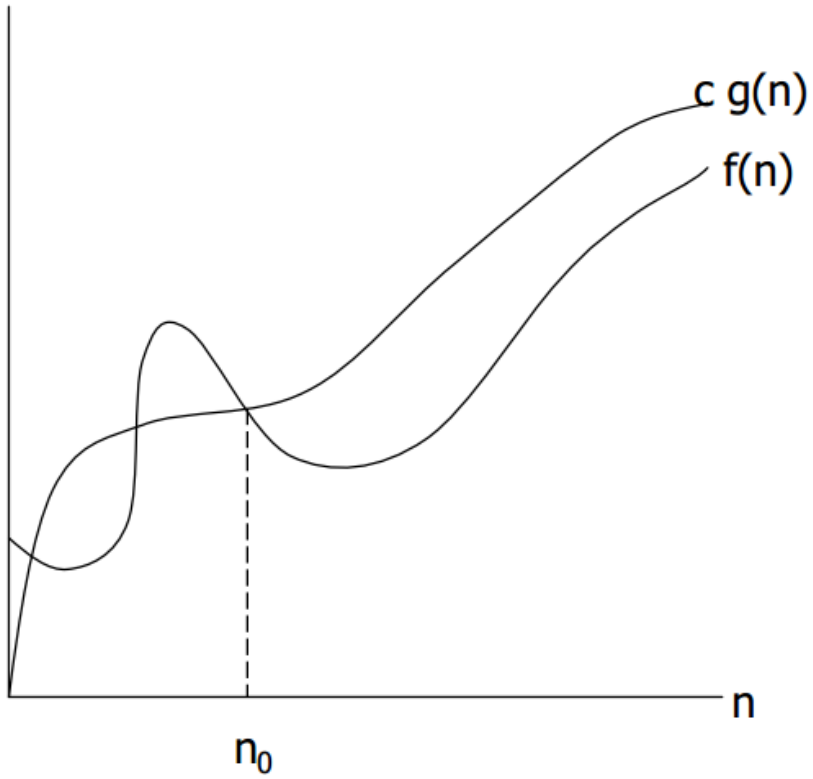
Code

```
if (n < 0) {          //    Cost: c1
    absval = -n;      //    Cost: c2
} else {
    absval = n;       //    Cost: c3
}
```

$T(n) = c_1 + max(c_2, c_3)$

Big O-Notation

“ Die Funktion f gehört zur Menge O(g), wenn es positive Konstanten c und n0 gibt, so dass f(n) ,ab n0 ,unterhalb cg(n) liegt ”



G.Zachmann

Big O-Notation

- *Nur der dominante Term wird betrachtet*
- *Alle Konstanten, welche nicht von n abhängen, werden ignoriert*

Big O-Notation

Code

```
int k = 0;
for (int I = 1; I <= n/2; i++) {
    for (int j = 1; j <= n; j++) {
        k = k + I + j;                    // Cost: c1
    }
}
```

$T(n) = (n/2) * (n * c_1)$

$O(n^2)$

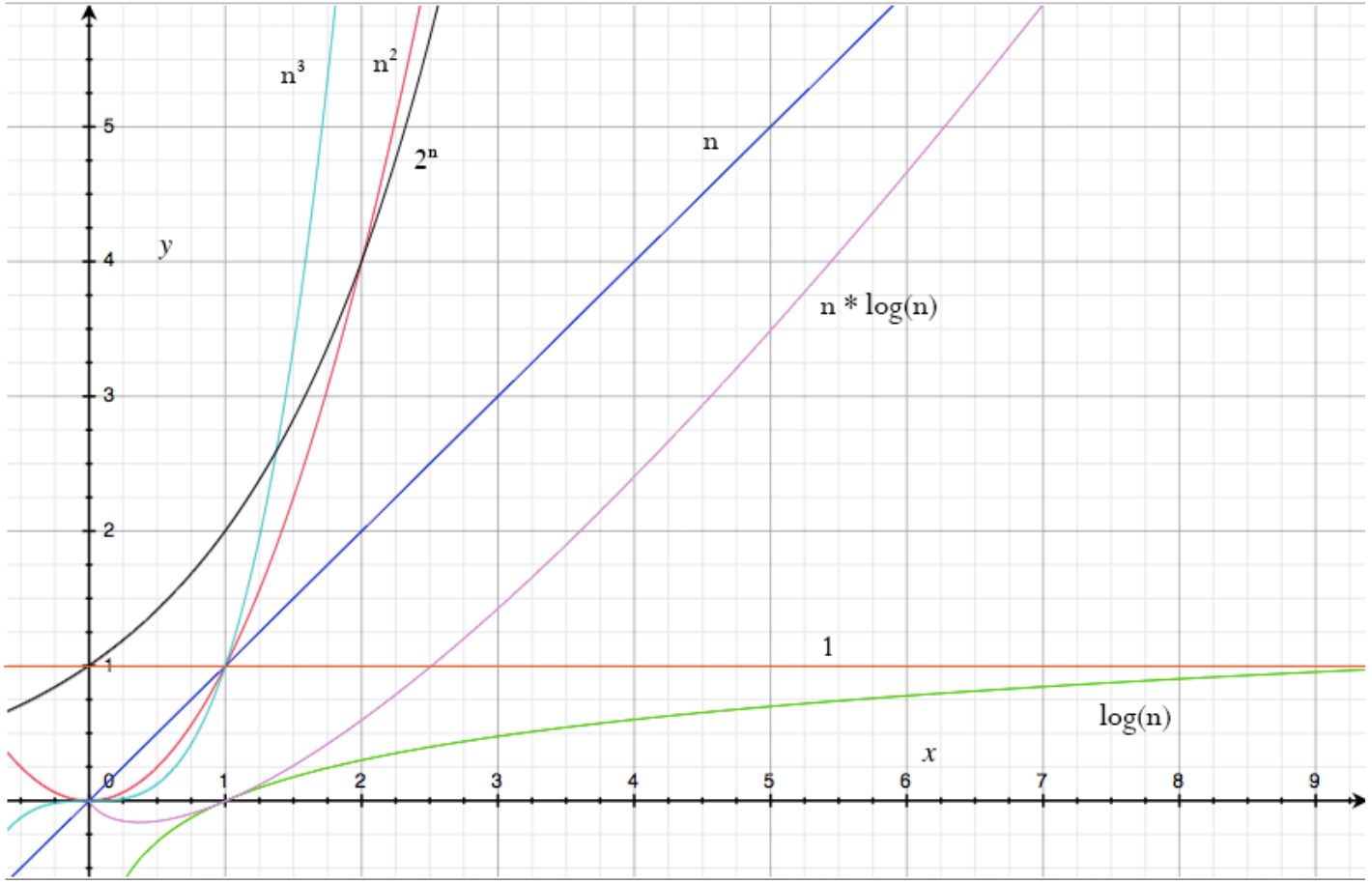
Kostenarten

- Worst-Case Analyse
- Average-Case Analyse
- Best-Case-Analyse

Häufige Wachstumsraten

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

Häufige Wachstumsraten



Insertion-Sort

Code

```
for(int i = 1; i < n; i++) {
    temp = array[i];                          // Cost: c1
    j = i;                                    // Cost: c2
    while(j > 0 && array[j-1] > temp) {
        array[j] = array[j-1];                // Cost: c3
        j = j - 1;                            // Cost: c4
    }
    array[j] = temp;                          // Cost: c5
}
```

Insertion-Sort

- $T(n) = n * (c_1 + c_2 + (n * (c_3 + c_4)) + c_5)$
- Best-Case: O(n)
- Worst-Case: O(n^2)
- Average-Case: O(n^2)

Binary Search

Code

```
int first = 0, last = n - 1;                          // Cost: c1
int middle = ( first+last ) / 2;                      // Cost: c2
while(first <= last) {
   if (arrayToSearch[middle] == search) {             // Cost: c3
      //Wert gefunden, gib Ergebnis aus
   } else if (arrayToSearch[middle] < search) {       // Cost: c4
      first = middle + 1;                             // Cost: c5
   } else {
      last = middle - 1;                              // Cost: c6
   }
    middle = ( first+last ) / 2;
    if(first > last) {                                //Cost: c7
       //meldung, element nicht gefunden
    }
}
```

Binary Search

- $T(n) = c_1 + c_2 + (\log(n) * (c_3 + c_4 + c_5 + c_6 + c_7))$
- Best-Case: O(1)
- Worst-Case: O(log n)
- Average-Case: O(log n)

Zeitkomplexität vs. Platzkomplexität

Code

```
int fib(x) {
    int a = 0;
    int b = 1;
    int current = 1;

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

Code

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

- Speicherplatz gegen Geschwindigkeit
- Geschwindigkeit gegen Speicherplatz

Nachteile

Code

```
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];
}
```

Quelle : Nathanael Hübbe

Nachteile

Code

```
double* foo (int n) {
    double sqrtLUT[n];
    for (int i = 0; i < n; i++) {
        sqrtLUT[i] = sqrt(i);
    }
    double (*table)[n][n] = malloc( sizeof (*table) );

    for (int y = 0; y < n; y++) {
        for (int x = 0; x < n; x++) {
            (*table)[y][x] = sqrtLUT[x] * sqrtLUT[y];
        }
    }
    return &(*table)[0][0];
}
```

Zusammenfassung

- T(n) und O(n) zur Performanceanalyse
- O(n^2) ist nicht immer schlechter als O(n)
- Laufzeiten von Binary Search und Insertion-Sort
- Zeitkomplexität ⇔ Platzkomplexität
- Nicht jedes Optimierungspotenzial wird erkannt

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.fh-konstanz.de/~bittel/prog2/Vorlesung/05KomplexitaetsAnalyse.pdf
11:4 , 04.01.2014

Quellen

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