14 Such- und Sortieralgorithmen

Datensätze bestehen aus verschiedenen Teilen (vgl. Strukturen). Ein Teil speichert den Schlüssel des Datensatzes, über den die Identifikation des Datensatzes erfolgt. Die weiteren Teile beinhalten Zusatzinformationen.

Beispiel:
Personendaten (Name, Adresse, Telefonnummer)
Schlüssel: Name

Die Anordnung der Datensätze kann in unterschiedlichen Datenstrukturen erfolgen.
Beispiele für Datenstrukturen

- **Feld**
  Zugriff auf ein Element über einen Index
- **Liste**
  Element besitzt Verweis auf das folgende Element
  - **Stapel (Stack)**
    Elemente können nur in umgekehrter Reihenfolge des Einfügens gelesen oder gelöscht werden
  - **Warteschlange (Queue)**
    Elemente können nur in gleicher Reihenfolge des Einfügens gelesen oder gelöscht werden
- **Graphen, Bäume**
  Elemente besitzen variable Anzahl von Verweisen auf weitere Elemente

14.1 Laufzeitkomplexität von Algorithmen

Beispiele für Suchalgorithmen:

- lineare/sequentielle Suche
- binäre Suche

Beispiele für Sortierverfahren

- Bubble-Sort
- Selection-Sort
- Insertion-Sort
- Quick-Sort
- Merge-Sort

Die unterschiedlichen Verfahren unterscheiden sich u.a. nach der Effizienz, d.h. nach Bedarf an Speicherplatz und Rechenzeit u. dem Wachstum der Rechenzeit bei steigender Anzahl n der Eingabeelemente Laufzeitkomplexität)

Beispiel:
Sortieralgorithmus A benötigt $\mathbf{n^2}$ Schritte für **n** Elemente
Sortieralgorithmus B benötigt $\mathbf{n*\log_2(n)}$ Schritte für **n** Elemente

Zur Abschätzung der Aufwandsfunktion f(n) von Algorithmen wird sehr häufig die **O-Notation** verwendet:

$$f(n) \in O(g(n)) : \leftrightarrow \exists c, n_0 \, \forall n \geq n_0 : f(n) \leq c\, g(n)$$

Das bedeutet, dass ab einem bestimmten n das f(n) nicht stärker als ein g(n) wächst und somit g als Abschätzung verwendet werden kann.

Daraus ergeben sich zum Beispiel die folgenden Abschätzungsarten:

O(1)	konstanter Aufwand
O(log n)	logarithmischer Aufwand
O(n)	linearer Aufwand
O(n*log n)	n*log n Aufwand
$O(n^2)$	quadratischer Aufwand
$O(n^k)$	polynomialer Aufwand
$O(2^n)$	exponentieller Aufwand

n	log n	n * log(n)	n^2	2^n
2	1	2	4	4
4	2	8	16	16
8	3	24	64	256
16	4	64	256	65536
32	5	160	1024	4,3E+09

Beispiele für die Zusammensetzung von Funktionen und die sich dadurch ergebene Aufwandsabschätzung:

f(n) dominiert g(n) falls f(n) bei großen n-Werten stärker wächst als g(n), d.h.

2^n dominiert n^k für jedes feste k
n^k dominiert n^r, falls k>=r
n*log(n) dominiert n
n dominiert $\log_b(n)$ für jede Basis b

Wird g(n) von f(n) dominiert, dann gilt

O(f(n) + g(n)) = O (f(n))

Beispiele:

$O(n^2-n) = O(n^2)$
$O(5*n^2+88n+1) = O(n^2)$
O (123) = O(1)
$O(n^2+2^n) = O(2^n)$

Wie bestimmt man die Komplexität der Zahl der Schritte, wie schätzt man die Laufzeitkomplexität ab?

zB Maschinenmodell:

- Schritte eines Algorithmus sind zeitkonstante Anweisungen
- Laufzeitunterschiede einzelner Schritte werden vernachlässigt
- Addieren, Runden, Kopieren, bedingte Verzweigung, Aufruf eines Unterprogramms sind jeweils Schritte
- Sortieren ist kein Schritt (sinnvolle Definition von Schritten!)

Laufzeitkomplexität für einige Sprachkonstrukte:

elementare Operation

`i1=0;`	**O(1)**

Sequenz elementarer Operationen

`i1=0;`
`i2=0;`
`...`
`i123=0;`
	`O(1)`
	`O(1)`
	`...`
	`O(1)`
		`123*O(1)=`**O(1)**

Schleifen

`for (i=0;i<n;i++)` `  a[i]=0;`	`O(n)` `O(1)`	`O(1)*O(n) =` **O(n)**

`for (i=0;i<n;i++)`
`{`
`  a1[i]=0;`
`  ...`
`  a23[i]=0;`
`}`
	`O(n)`	`O(n)`	`O(1)*O(n)=`**O(n)**

	`O(1)`
	`...`
	`O(1)`
		`23*O(1) = O(1)`	

`for (i=0;i<n;i++)`
`  for (j=0;j<n-1;j++)`
`  {`
`  a1[i][j]=0;`
`  ...`
`  a34[i][j]=0;`
`  }`
	`O(n)`	`O(n)*O(n-1) =`
		$O(n)*O(n)=O(n^2)$
			$O(1)*O(n^2)=$**$O(n^2)$**

	`O(n-1)`		

	`O(1)`
	`...`
	`O(1)`
		`34*O(1) = O(1)`	

bedingte Anweisung

`if (x<100)`
`  y=x;`
`else`
`  for (i=0;i<n;i++)`
`  if (a[i]>y)`
`  y=a[i];`
	`O(1)`	`O(1)`	`O(max{1,n})=`**O(n)**

	`O(1)`		

	`O(n)`
	`O(1)`
	`O(1)`
		`O(n)*O(1) = O(n)`	

Die Laufzeit hängt nicht immer ausschließlich von der Größe des Problems ab, sondern auch von der Beschaffenheit der Eingabemenge.

Daraus ergeben sich

- **beste Laufzeit (best case)**
  beste Laufzeitkomplexität für eine Eingabeinstanz der Größe n

- **schlechteste Laufzeit (worst case)**
  schlechteste Laufzeitkomplexität für eine Eingabeininstanz der Größe n

- **mittlere oder erwartete Laufzeit (average case)**
  gemittelte Laufzeitkomplexität für alle Eingabeinstanzen der Größe n

Beispiel:

Eingabe:
Feld a mit n Elementen mit 1 <= a[i] <= n

Programmcode:

<table>
<tr><td rowspan="3"><pre>if (a[0]==1)
      a[0]=1;
else
      for (i=0;i<n;i++)
            a[i]=2;</pre></td><td>O(1)</td><td>O(1)</td><td rowspan="3">?</td></tr>
<tr><td>O(1)</td><td>O(1)</td></tr>
<tr><td>O(n)
O(1)</td><td>O(n)</td></tr>
</table>

beste Laufzeit: O(1) + O(1) = O(1)

schlechteste Laufzeit: O(1) + O(n) = O(n)

mittlere Laufzeit:
Mittelung der notwendigen Schritte für **alle** Eingabeinstanzen der Größe **n**
jedes Element a[i] kann n Werte annehmen
n Elemente: n*n*...*n **= n^n verschiedene Eingabeinstanzen**

a[0]=1 in n^{n-1} Instanzen
a[0]!=1 in allen anderen Fällen, also $n^n - n^{n-1} = n^{n-1} *(n-1)$ Instanzen

Gesamtschritte:
n^{n-1}Instanzen * 1 Schritt + n^{n-1} *(n-1) Instanzen * n Schritte
= $n^{n-1} + n^n (n-1)$ Schritte

mittlere Laufzeit: Schritte / Instanzen

$$\frac{n^{n-1}+n^n\cdot(n-1)}{n^n} = \frac{1}{n} + n - 1$$ = O(n)

14.2 Suchalgorithmen

Beispiele für Suchalgorithmen:

- **lineare/sequentielle Suche**
- **binäre Suche**
- weitere Suchverfahren auf sortierten Feldern
    - Fibonacci-Suche
    - Sprung-Suche
    - Exponentielle Suche

Aufgabenstellung:
In einem Telefonbuch sollen Einträge gesucht werden. Die Datensätze bestehen aus
Nachname
Vorname
Adresse und
Telefonnummer,
Sie sind geordnet nach Nachname, bei Gleichheit nach Vorname.
Nach- und Vorname sind Schlüssel.

Aufgabe 1: Wenn man den Namen kennt und die Telefonnummer wissen möchte, findet man den Eintrag relativ rasch:

- Aufschlagen des Telefonbuchs in der Mitte
- Dann wird entschieden, ob man in der linken oder in der rechten Hälfte weitersucht
  →binäre Suche

Aufgabe 2: Das Telefonbuch von Wien ist 100-mal so dick, wie das vorhin durchsuchte. Dauert das Suchen auch 100-mal so lange?

- Nein- für die Suche werden nur 6-7 weitere Namen gelesen/verglichen werden.
  →binäre Suche ist effizient
- Der **Aufwand der binären Suche** ist proportional **$\log_2(n)$,** dem binären Logarithmus von n

Aufgabe 3: Wer hat die Telefonnummer 52575 ?

- Die Information ist auch im Telefonbuch, aber nur schwer zu finden
- jeder Datensatz muss durchsucht werden.
  →lineare/sequentielle Suche

Aufgabe 4: Wie lange dauert die Suche in einem 100-mal dickeren Telefonbuch?

- Die Suche dauert 100-mal so lange.
- Der **Aufwand der linearen Suche** ist proportional der Anzahl **n** der Datensätze

14.2.1 sequentielle Suche

Prinzip: überprüfe sequentiell alle Elemente des Feldes (brute force search)

```
int seqSearch(int k, int a[], int n)
{
      int i=0;
      while (i<n && a[i]!=k)        //Zwei Abfragen pro Durchlauf
      {     i++;
      }
      if (i<n)
            return i;               //Element gefunden
      else
            return -1;              //Element nicht gefunden
}
```

optimierte Variante:
Man fügt das gesuchte Element an die erste (oder letzte) Stelle des Feldes ein. Das spart eine Abfrage pro Durchlauf:

```
int seqSearch(int k, int a[], int n)
{
      int i = n;              //durchsucht a[1] .. a[n] nach k
      a[0] = k;               //a[0] gehört nicht zur durchsuchenden Menge
                              //(= Stopp-Element)
      do {
        i--;
      } while (a[i]!=k);      //Eine Abfrage pro Durchlauf
      if (i!=0)
            return i;
      else
            return -1;
}
```

Laufzeit:

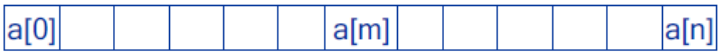
- bester Fall: O(1)
- schlechtester Fall: O(n)
- mittlerer Fall: $\frac{1}{n}\sum_{i=1}^{n} i = \frac{1}{n}\frac{n\cdot(n+1)}{2} = \frac{n+1}{2} = O(n)$

14.2.2 binäre Suche

Ist die zu durchsuchende Menge von Datensätzen sortiert, kann die Suche beschleunigt werden, indem das gesuchtes Element mit dem mittleren Element des Feldes verglichen und bei Ungleichheit nur jeweils die linke oder die rechte Hälfte des Feldes rekursiv weiter betrachtet wird.

Prinzip:

Suche von k auf einem sortierten Feld a



Vergleiche k mit a[m] wobei m=(0+n)/2

Fall 1: k==a[m] → fertig

Fall 2: k<a[m] → rekursive Suche von k in a[0] ... a[m-1]

Fall 3: k>a[m] → rekursive Suche von k in a[m+1] ... a[n]

iterative Variante:

```
int binSearch1(int k, int a[], int l, int r)
{
      int m;

      while(l<=r)
      {
            m=(l+r)/2;
            if (k==a[m]) return m;
            if (k<a[m])  r=m-1;
            if (k>a[m])  l=m+1;
      }
      return -1;
}
```

rekursive Variante:

```
int binSearch2(int k, int a[], int l, int r)
{
      int m;

      if (l>r) return -1;

      m=(l+r)/2;
      if (k==a[m]) return m;
      if (k<a[m])  return binSearch2(k, a, l  , m-1);
      if (k>a[m])  return binSearch2(k, a, m+1, r  );
}
```

Laufzeit:

- jeder Vergleich reduziert die zu durchsuchende Menge um den Faktor 2
- schlechtester Fall: O(log n)
- bester Fall: O(1)
- mittlerer Fall: $\frac{\log(n+1)\cdot(n+1)-(n+1)+1}{n} \approx \log(n+1)-1$ ergibt O(log n)
  Im Durchschnitt 1 Vergleich weniger als die Maximalzahl möglicher Vergleiche (intuitiv durch Baumrepräsentation nachzuvollziehen)