Effizienz von Algorithmen und Datenstrukturen

Software Entwicklung 1

Annette Bieniusa, Mathias Weber, Peter Zeller

Im letzten Kapitel haben wir verschiedene Algorithmen zum Sortieren von Datensätzen betrachtet. Welcher Algorithmus soll nun bei einem konkreten Anwendungsfall einsetzt werden?
In diesem Kapitel werden wir Algorithmen analysieren, um sie bezüglich ihrer Laufzeit und ihres Speicherbedarfs vergleichen zu können. Diese Analyse kann sowohl durch Messungen bei der Ausführung auf verschiedenen Eingaben als auch durch theoretische Überlegungen erfolgen.
Zum Einstieg in die Algorithmenanalyse betrachten wir zunächst einen einfacheren Algorithmus, die binäre Suche auf sortierten Daten.

Kapitel 4.1 aus *Sedgewick, Wayne: Einführung in die Programmierung mit Java*

1 Binäre Suche

Frage 1: Ein kleines Ratespiel: Ich stelle mir eine Zahl zwischen 0 und 1000 vor; Sie müssen diese Zahl erraten und dürfen mir dazu Fragen stellen. Die einzige erlaubte Frage ist dabei "Ist die Zahl kleiner als _ ?"; die Antwort ist entweder "Ja" oder "Nein".

- Wie gehen Sie vor?
- Wie viele Fragen müssen Sie maximal stellen, um die Zahl zu ermitteln?

Wir betrachten im Folgenden eine Verallgemeinerung dieser Fragestellung:

Wie kann man mit möglichst wenigen Fragen eine Zahl $x \in \mathbb{N}$ zwischen 0 und N erraten?

Wir betrachten folgende algorithmische Idee zur Lösung des Ratespiels:

- Wir verwenden ein Intervall $[l, h)$ mit $x \in [l, h)$, das in jedem Schritt halbiert wird. Dabei gilt für die gesuchte Zahl x in jedem Schritt: $l \leq x$ und $x < h$
- Als Anfangsintervall wählen wir das Intervall $[0, N + 1)$.
- Rekursive Strategie:

  Basisfall: Enthält das Intervall nur noch eine Zahl ($h - l = 1$), dann ist die gesuchte Zahl $x = l$

  Rekursiver Schritt:

  * Frage, ob die Zahl kleiner ist als die Mitte des Intervalls, $m = l + (h - l)/2$.
  * Falls ja, suche die Zahl im linken Teilintervall $[l, m)$.
  * Andernfalls, suche die Zahl im rechten Teilintervall $[m, h)$.

Man kann dieses Ratespiel folgendermaßen implementieren:

```c
#include<stdio.h>

// Sucht die Zahl im Interval [lo,hi)
int search(int lo, int hi) {
  // Basisfall
  if ((hi-lo) == 1) {
    return lo;
  }
  // Rekursiver Schritt
  int mid = lo + (hi -lo) / 2;
  printf("Ist die Zahl kleiner als %d ? (j/n)\n", mid);
  char answer;
  scanf("%c", &answer);
  getchar(); // liest das naechste Zeichen, d.h. den Zeilenumbruch
  if (answer == 'j') {
    return search(lo,mid);
  } else {
    return search(mid,hi);
  }
}

int main(void)
{
  // Erwartet als Befehlzeilenparameter eine positive ganze Zahl (obere Grenze)
  printf("Rate eine Zahl zwischen 0 und 100!\n");
  int x = search(0, 100);
  printf("Die gesuchte Zahl ist %d\n", x);
}
```

Frage 2: Wie kann man das Programm erweitern, damit in einem beliebigen Intervall gesucht werden kann?

Korrektheit des Verfahrens Wie können wir *beweisen*, dass unser Suchverfahren das richtige Ergebnis liefert?
Wir machen hierzu die vereinfachende Annahme, dass $N = 2^n$ für ein $n \in \mathbb{N}$ ist. Diese Annahme erleichtert das Halbieren der Intervallgröße im rekursiven Fall. Für jeden Aufruf von `search(h,l)` gilt (Beweis durch Induktion):

- Das Intervall enthält immer die gesuchte Zahl.
- Die Intervallgrößen, d.h. die Differenz $h - l$, sind Zweierpotenzen, die mit jedem rekursiven Aufruf kleiner werden, beginnend mit 2^n und endend mit $2^0 = 1$.

Damit ist sichergestellt, dass das Verfahren terminiert, da der Basisfall mit Intervallgröße 1 immer erreicht wird. Außerdem enthält das Intervall immer, also auch im Basisfall, die gesuchte Zahl.

Analyse Wie viele Fragen werden benötigt, um die gesuchte Zahl zu ermitteln?

- Nach Annahme ist $N = 2^n$ für ein $n \in \mathbb{N}$, d.h. $n = \log_2(N)$.
- Sei $T(N)$ die Anzahl der Fragen, die für die Suche im Intervall der Größe N gestellt werden müssen.
- In jedem Rekursionsschritt wird eine Frage gestellt und das Problem auf einem Intervall halber Größe gelöst:

$$T(N) = 1 + T(N/2)$$

- Im Basisfall ist keine Frage mehr nötig:

$$T(1) = 0$$

- Insgesamt erhalten wir:

$$\begin{aligned} T(N) &= T(2^n) \\ &= T(2^{n-1}) + 1 = T(2^{n-2}) + 2 = \cdots = T(2^{n-n}) + n \\ &= T(2^0) + n = T(1) + n \\ &= n \end{aligned}$$

Man muss also $n - 1$-mal fragen; beim letzten Aufruf von `search` befindet man sich im Basisfall. Insgesamt benötigt man $n - 1 + 1 = n = \log_2(N)$ Aufrufe von `search`.

1.1 Binäre Suche in sortiertem Array

Die Idee der binären Suche ist auch beim Suchen in sortierten Datensammlungen anwendbar, die einen direkten Zugriff auf einen Eintrag erlauben. Im Alltag verwendet man den Algorithmus beispielsweise bei der Suche in Wörterbüchern oder Telefonbüchern. Wir wenden nun die gleiche algorithmische Idee wie im Ratespiel an, um Einträge in einem sortierten Array zu suchen.

Problembeschreibung Gegeben sei eine Folge $s_0, \ldots, s_{N-1}$ von sortierten *Datensätzen*. Jeder Datensatz s_j besteht aus einem Schlüssel k_j und einem zugehörigen Wert (hier: String). Wir gehen hier davon aus, dass die Schlüssel Integer sind. Wir repräsentieren einen Datensatz durch die Struktur `dataset_t`:

```
5   typedef struct dataset {
6     int key;
7     char *data;
8   } dataset_t;
```

Für sortierte Datensätze gilt:

$$k_0 \leq k_1 \leq \cdots \leq k_{N-1}$$

```
12  /* Hilfsfunktion fuer die rekursive Variante */
13  /* Sucht den Datensatz mit Schluessel x im Indexbereich [lo,hi)
       */
14  dataset_t *search(int x, dataset_t *f, int lo, int hi) {
15    if (hi <= lo) {
16      // Element nicht gefunden
17      return NULL;
18    }
19    int mid = lo + (hi-lo) /2;
20    if (x < f[mid].key) {
21      // Suche rekursiv im Intervall [lo, mid)
22      return search(x, f, lo, mid);
23    }
24    if (x > f[mid].key) {
25      // Suche rekursiv im Intervall [mid+1, hi)
26      return search(x, f, mid+1, hi);
27    }
28    // Hier gilt nun: x == f[mid].key, d.h. Element gefunden
29    return &f[mid];
30  }
31
32  /* Liefert NULL, wenn Datensatz mit Schluessel x nicht in f
       enthalten ist; andernfalls die Referenz auf den gefundenen
       Datensatz */
33  /* rekursive Variante */
34  dataset_t *binsearch_recursive (int x, dataset_t *f, int n) {
35    return search(x, f, 0, n);
```

```
}
```

Hinweis: Dieser Algorithmus kann auch in der Implementierung von `searchIndex` für die Maps basierend auf Arrays in leicht abgewandelter Form angewendet werden, um den passenden Array-Index für einen Schlüssel zu ermitteln (vgl. Hinweis in Kapitel 14). Dabei muss sichergestellt sein, dass die Einträge im Array sortiert verwaltet werden (d.h. neue Einträge werden nicht am Ende, sondern sortiert am richtigen Index eingefügt, damit die Schlüssel in der Map immer aufsteigend (bzw. absteigend) sortiert sind).

Wir zeigen hier außerdem zum Vergleich die iterative Variante:

```
/* Liefert null, wenn Datensatz mit Schluessel x nicht in f
  enthalten ist; andernfalls die Referenz auf den gefundenen
  Datensatz */
/* iterative Variante */
dataset_t *binsearch_iterative(int x, dataset_t *f, int n) {
  int lo = 0;
  int hi = n - 1;

  // fuer alle int i < lo gilt elems[i].key < x
  // fuer alle int i > hi gilt elems[i].key > x
  while (lo <= hi) {
    int mid = lo + (hi-lo)/2;
    if (x < f[mid].key) {
      hi = mid - 1;
    } else if (x > f[mid].key) {
      lo = mid + 1;
    } else {
      return &f[mid];
    }
  }
  // lo == hi + 1
  // fuer alle int i < lo gilt elems[i].key < x
  // fuer alle int i >= lo gilt elems[i].key > x

  return NULL;
}
```

2 Einführung in die Algorithmenanalyse

Programme müssen oft gewisse nicht-funktionale Anforderungen erfüllen. Dazu gehören die Geschwindigkeit oder eine Beschränkung der verfügbaren Ressourcen wie Arbeitsspeicher. Daher kommen Fragen auf, wie:

- Wie lange braucht das Programm um eine Berechnung auszuführen?

- Reicht die Größe meines Arbeitsspeichers für das Programm aus?

Die Antwort auf diese Fragen ist von vielen Faktoren abhängig und sehr schwierig präzise zu beantworten. In der Regel hängt die Laufzeit und der Speicherbedarf von der konkreten Eingabedaten ab.
Wir wollen in diesem Kapitel mit Hilfe von Messungen und Analyse des Quelltexts Modelle entwickeln, mit denen sich Laufzeit (bzw. Speicherbedarf) in Abhängigkeit der Eingabe abzuschätzen lassen. Messungen sind dabei nötig, um das Modell zu validieren und wichtige Kenngrößen zu ermitteln. Die Analyse des Quelltexts ist wichtig, da sie tiefere Einblicke gibt und eventuell auch Fälle aufzeigt, die nur selten auftreten und durch Tests evtl. nicht abgedeckt werden.
Wir betrachten hier zunächst die Laufzeit von Programmen. Die Ermittlung des Speicherbedarfs behandeln wir im nächsten Kapitel.

2.1 Wissenschaftliche Vorgehensweise

Um ein Modell zu ermitteln und zu validieren, gehen wir folgendermaßen vor:

1. *Beobachte* bestimmte Eigenschaften des Programms (hier: quantitativ durch Messungen von Laufzeit).
2. Erstelle eine *Hypothese*, die mit den Beobachtungen übereinstimmt.
3. *Prognostiziere* mit Hilfe dieser Hypothese Ereignisse.
4. *Validiere* die Vorhersagen durch weitere Beobachtungen. Falls die Vorhersagen nicht beobachtet werden können, verwerfe die Hypothese und erstelle eine neue, angepasste Hypothese.

Die Experimente zur Beobachtung müssen reproduzierbar sein. Hypothesen müssen außerdem falsifizierbar sein (d.h. müssen prinzipiell durch Experimente widerlegbar sein).

2.2 Beispiel: Selectionsort

Schritt 1: Beobachtungen Zunächst messen wir die Laufzeit des Programms für verschiedene Eingaben. Messung der Laufzeit ist auf verschiedene Arten möglich.
Eine einfache Möglichkeit die Laufzeit eines Programms zu messen ist das Programm `time`, welches auf den meisteen Linux- und Mac-Systemen bereits vorinstalliert ist. (Unter Windows kann alternativ zum Beispiel der `Measure-Command`-Befehl in der PowerShell verwendet werden.)
Das Programm `time` nimmt als Programm-Argumente einen Befehl, der ausgeführt und dessen Laufzeit gemessen wird. Um die Laufzeit des Aufrufs `./a.out 30000` zu testen, kann `time` beispielsweise wie folgt aufgerufen werden:

```
$ time ./a.out 30000
```

```
real    0m1.167s
user    0m1.160s
sys     0m0.000s
```

Wir können `time` zusammen mit einer `main`-Funktion verwenden, welche ein Array mit gegebener Größe erstellt, mit zufälligen Werten füllt und dann mit `selectionsort` sortiert:

```
1  // Erstellt ein Array der Größe size mit zufälligen Werten
2  int *random_array(int size)
3  {
4      int *ar = malloc(size*sizeof(int));
5      if (ar == NULL)
6      {
7          printf("OUT of memory, size = %d\n", size);
8          abort();
9      }
10     for (int i=0; i<size; i++)
11     {
12         ar[i] = rand();
13     }
14     return ar;
15 }
16
17 int main(int argc, char **argv) {
18     int size = 10000;
19     if (argc > 1)
20     {
21         // Größe von Programmparameter lesen
22         size = atoi(argv[1]);
23     }
24     int *ar = random_array(size);
25     selectionsort(ar, size);
26     free(ar);
27     return 0;
28 }
```

Beim Entwerfen von Programmen zum Messen der Laufzeit sollte man bedenken, dass Optimierungen, die der Compiler anwenden kann, eventuell das Ergebnis verfälschen könnten. Zum Beispiel könnte ein hinreichend intelligenter Compiler erkennen, dass das sortierte Array nie gelesen wird und daher das Sortieren aus dem Programm entfernen.

Ein Nachteil des Messens mit `time` ist die geringe Genauigkeit und die fehlende Möglichkeit das Generierens der Eingabe vom eigentlich zu messenden Sortieren zu trennen. Alternativ zum Programm `time` kann die Zeit auch im Programm selbst gemessen wer-

den. Dazu bietet C beispielsweise die Funktion `clock_t clock(void)` [1] an, welche die Prozessor-Zeit für den aktuellen Prozess angibt. Diese Zeit ist unabhängig von der realen Zeit und läuft mal schneller oder langsamer, je nachdem wieviel Prozessorzeit der aktuelle Prozess verwendet. Mit Hilfe dieser Funktion können wir ein Programm schreiben, welches den SelectionSort-Algorithmus mit verschiedenen Eingabegrößen testet und jeweils die Größe und Laufzeit in Millisekunden (ms) ausgibt. Wir verdoppeln hier die Eingabegröße so lange, bis ein Test 5000ms oder länger benötigt.

```
23 int main(int argc, char **argv) {
24     int size = 1;
25     double max_time = 5000;
26     double time;
27     do {
28         int *ar = random_array(size);
29         // Zeit vor Sortieren:
30         clock_t start_time = clock();
31         // Sortieren:
32         selectionsort(ar, size);
33         // Zeit nach Sortieren:
34         clock_t end_time = clock();
35         // Verbrauchte Zeit in ms berechnen:
36         time = (end_time - start_time)*1000.0 / CLOCKS_PER_SEC;
37         printf("%d, %lf\n", size, time);
38         free(ar);
39         size = size * 2;
40     } while (time < max_time);
41     return 0;
42 }
```

Ausführen des Programms ergibt z.B. die folgende Ausgabe:

```
1, 0.000000
2, 0.000000
4, 0.001000
8, 0.001000
16, 0.000000
32, 0.001000
64, 0.004000
128, 0.010000
256, 0.033000
512, 0.114000
1024, 0.412000
2048, 1.562000
4096, 10.494000
8192, 27.875000
16384, 102.111000
32768, 366.685000
65536, 1450.968000
131072, 5901.001000
```

Durch weitere Experimenten und Messungen ergeben sich die folgenden Beobachtungen:

[1] Siehe auch `http://en.cppreference.com/w/c/chrono/clock`. Zum Verwenden der Funktion wird der Import `#import <time.h>` benötigt.

- Die Laufzeit ist abhängig von der Größe N des zu sortierenden Arrays (insbesondere für große N).
- Die Laufzeit variiert leicht für *gleiche* Eingabe bei verschiedenen Durchläufen.
- Es gibt keine offensichtliche Abhängigkeit von den zu sortierenden Zahlen, insbesondere scheint der Algorithmus insensitiv in Bezug auf Vorsortierung des Arrays zu sein.
- Die Laufzeit ändert sich, wenn andere Hardware verwendet wird, der Trend bleibt aber gleich.

Schritt 2: Hypothese Typischerweise beeinflusst eine (bisweilen auch mehrere) ***Problemgröße*** die Komplexität eines Programms. Die Laufzeit sollte mit wachsender Problemgröße zunehmen. Typischerweise ist dies die Größe der Eingabe oder auch der Wert von Befehlszeilenargumenten. Im Fall von Selectionsort ist die Größe N des zu sortierenden Arrays die Problemgröße.
Um die Frage zu beantworten, wie viel tatsächlich die Laufzeit mit wachsender Arraygröße zunimmt, wird häufig die ***Verdopplungshypothese*** verwendet.
Man erhält diese Hypothese als Antwort auf die folgende Frage:

Welche Auswirkungen hat es auf die Laufzeit,
wenn wir die Problemgröße verdoppeln?

Zum Beantworten dieser Frage betrachten wir verschiedene Messungen, mit jeweils verdoppelter Eingabegröße und die Veränderung der Laufzeit im Vergleich zum vorherigen Durchlauf.

N	Laufzeit (ms)	Veränderung
1	0	
2	0	
4	0.001	
8	0.001	
16	0	
32	0.001	
64	0.004	4.00
128	0.01	2.50
256	0.033	3.30
512	0.114	3.45
1024	0.412	3.61
2048	1.562	3.79
4096	10.494	6.72
8192	27.875	2.66
16384	102.111	3.66
32768	366.685	3.59
65536	1450.968	3.96
131072	5901.001	4.07

Wir beobachten, dass sich bei großen Eingabegrößen die Laufzeit ungefähr um den Faktor 4 erhöht, wenn sich die Eingabegröße verdoppelt.

Schritt 3: Prognostiziere mit Hilfe der Hypothese Ereignisse Mit der Hypothese, dass ein Verdoppeln der Eingabegröße die Laufzeit um den Faktor 4 erhöht, können wir Vorhersagen treffen.
Da ein Array mit 131072 Elementen $5901ms$ zum Sortieren benötigte, erwarten wir für ein Array mit doppelter Größe eine Laufzeit von etwa $4 \cdot 5901ms = 23604ms$. Entsprechend können wir die 3 nächsten Verdoppelungsschritte prognostizieren:

N	Vorhersage der Laufzeit (ms)
262144	23604
524288	94416
1048576	377664

Schritt 4: Validieren der Hypothese Zum Validieren der Hypothese führen wir das gleiche Experiment mit den vorhergesagten Eingabegrößen durch. Das Ergebnis zeigt, dass die Vorhersage nah an der tatsächlichen Laufzeit liegt.

N	Laufzeit (ms)	Vorhersage
262144	24600.95	23604 (-4.05%)
524288	99026.04	94416 (-4.66%)
1048576	435795.29	377664 (-13.34%)

Weitere Messungen und Experimente sind nötig, um die Hypothese weiter zu stützen oder zu widerlegen.

2.3 Mathematisches Modell zum Laufzeitverhalten

Wie kann die Hypothese nun theoretisch validiert werden?
Nach D. E. Knuth ist die Laufzeit eines Programms bestimmt durch (1.) die (Laufzeit-)Kosten für die Ausführung der verschiedenen Operationen und (2.) die Häufigkeit der Ausführung der verschiedenen Operationen.
Es muss also zunächst geklärt werden, welche Operationen konkret betrachtet werden und welche Kosten bei diesen Operationen entstehen.

Kostenmodell für die Sortierung auf Integer-Arrays Zur Analyse von Sortieralgorithmen betrachten wir hier die folgenden Operationen:

- Vergleich
- Zuweisung bzw. Inkrement / Dekrement von Indizes
- Addition / Subtraktion

Wir nehmen vereinfachend an, dass alle diese Operationen die gleiche Zeit benötigen.
Alle anderen Aspekte (z.B. Speicherhierarchien bzw. Caches) vernachlässigen wir zunächst.

2.4 Laufzeitanalyse von Sortieren durch Auswahl

Zunächst wandeln wir die beiden `for`-Schleifen zu äquivalenten `while`-Schleifen um und ersetzen den Aufruf von `swap` durch entsprechende Anweisungen. Dies verdeutlicht, wann welche Anweisungen ausgeführt werden.

```
1  void selectionsort2(int *f, int N) {
2    int i = 0;
3    while (i < N - 1) { // Schleife A
4      int imin = i;
5      int j = i+1;
6      while (j < N) { // Schleife B
7        if (f[j] < f[imin]) {
8          imin = j;
9        }
10       j++;
11     }
12     int temp = f[i];
13     f[i] = f[imin];
14     f[imin] = temp;
15     i++;
16   }
17 }
```

Wir betrachten nun den Code und zählen die verschiedenen Operationen. Dazu betrachten wir die Schleifen gesondert, da die Anzahl der von der Schleife ausgeführten Operationen von der Anzahl der Schleifendurchläufe abhängt (und damit auch von der Eingabegröße N).

- In jedem Schleifendurchlauf von Schleife B:
  - ein Vergleich (Abbruchbedingung)
  - ein Vergleich (if-Bedingung)
  - eine Zuweisung (zu `imin`, falls die `if`-Bedingung sich zu `true` ergibt)
  - ein Inkrement (von `j`)

  Insgesamt: 4 Operationen
- In jedem Schleifendurchlauf von Schleife A:
  - eine Subtraktion (in Abbruchbedingung)
  - ein Vergleich (Abbruchbedingung)
  - eine Zuweisung (`imin`)
  - eine Addition (`i+1`)
  - eine Zuweisung (`j`)
  - $N-(i+1)$ mal der Rumpf der B-Schleife (jeweils 4 Operationen im schlimmsten Fall)

  - ein Vergleich (Abbruchbedingung der Schleife)
  - eine Zuweisung (`temp`)
  - eine Zuweisung (`f[i]`)
  - eine Zuweisung (`f[imin]`)
  - ein Inkrement (`i`)

  Insgesamt: $10 + 4 \cdot (N - (i + 1))$ Operationen

Für die gesamte Prozedur `selectionsort2` ergibt sich damit:

- eine Zuweisung (`i`)
- $N - 1$ mal der Rumpf der A-Schleife (je $10 + 4 \cdot (N - (i + 1))$ Operationen für Werte $i = 0, 1, \ldots, N - 2$)
- eine Subtraktion (Abbruchbedingung der Schleife)
- ein Vergleich (Abbruchbedingung der Schleife)

Wir gehen nun vereinfachend davon aus, dass immer der schlimmste Fall eintritt und die `if`-Bedingung sich zu `true` ergibt. Die Beobachtungen oben lassen sich dann in folgender Formel zusammenfassen, welche die Anzahl der Operationen insgesamt in Abhängigkeit von der Länge N des Eingabe-Arrays angibt:

$$T_A(N) = 3 + \sum_{i=0}^{N-2} (10 + 4 \cdot (N - (i + 1)))$$

Diese Formel lässt sich mathematisch wie folgt vereinfachen:

$$\begin{aligned}
T_A(N) &= 3 + \sum_{i=0}^{N-2} (10 + 4 \cdot (N - (i + 1))) \\
&= 3 + \sum_{i=1}^{N-1} (10 + 4 \cdot (N - i)) \\
&= 3 + \sum_{i=1}^{N-1} (10 + 4 \cdot N - 4 \cdot i)) \\
&= 3 + \sum_{i=1}^{N-1} 10 \quad + \sum_{i=1}^{N-1} 4 \cdot N \quad - \sum_{i=1}^{N-1} 4 \cdot i \\
&= 3 + (N-1) \cdot 10 + (N-1) \cdot 4 \cdot N - 4 \cdot \sum_{i=1}^{N-1} i \\
&\overset{\text{für } N>1}{=} 3 + (N-1) \cdot 10 + (N-1) \cdot 4 \cdot N - 4 \cdot \frac{N \cdot (N-1)}{2} \\
&= 2 \cdot N^2 + 8 \cdot N - 7
\end{aligned}$$

Die Umformungen verwenden die Gaußsche Summenformel:

$$\sum_{k=1}^{n} k = \frac{n \cdot (n+1)}{2}$$

Wir können nun die mathematische Analyse mit der zuvor aufgestellten Hypothese vergleichen, dass ein Verdoppeln der Array-Länge zu einer Verlängerung der Laufzeit um den Faktor 4 führt. Wir berechnen dazu $T_A(2 \cdot N)$:

$$\begin{aligned} T_A(2N) &= 2 \cdot (2N)^2 + 8 \cdot (2N) - 7 \\ &= 8 \cdot N^2 + 16 \cdot N - 7 \\ &= 4 \cdot (2 \cdot N^2 + 8 \cdot N - 7) - 16 \cdot N + 21 \\ &= 4 \cdot T_A(N) - 16 \cdot N + 21 \end{aligned}$$

Da für große N der Teil $-16 \cdot N + 21$ vernachlässigt werden kann, bestätigt sich hier die Hypothese: Für große N ist $T_A(2N) \approx 4 \cdot T_A(N)$.

Bemerkungen Wie das Beispiel zeigt, kann eine exakte Analyse bisweilen sehr komplex werden. Es ist beispielsweise schwierig genau zu analysieren, wie oft die Zuweisung `imin = j;` im Allgemeinen erfolgt.
Häufig genügt es die ***Größenordnung*** des Wachstums einer ***Aufwandsfunktion*** $A(N)$ (im obigen Beispiel $T_A(N)$) zu betrachten, die den Zeit- bzw. Speicheraufwand in Abhängigkeit von der Problemgröße N beschreibt. Die Größenordnung ist unabhängig vom tatsächlichen Rechner und erlaubt es insbesondere Laufzeiten verschiedener Algorithmen für große Problemgrößen zu vergleichen. Um Laufzeitgarantien zu geben, wird dabei oft konservativ der schlimmste Fall (engl. *worst case*) betrachtet.

2.5 O-Notation

Eine Funktion $f : \mathbb{N} \to \mathbb{R}^+$ heißt ***asymptotische obere Schranke*** einer Aufwandsfunktion A, wenn gilt:
Es gibt $c, d \in \mathbb{N}$, sodass für alle $N \in \mathbb{N}$ gilt:

$$A(N) \leq c * f(N) + d$$

Man sagt auch, A wächst wie f bzw. ist von der Größenordnung f.
Die Menge aller Funktionen von der Größenordnung f bezeichnet man mit $O(f)$:

$$O(f) = \{g \mid \exists c, d \in \mathbb{N} : \forall N \in \mathbb{N} : g(N) \leq c * f(N) + d\}$$

Diese Notation wird oft O-Notation ("Groß O Notation") oder auch Bachmann–Landau Notation genannt. Oft wird auch folgende äquivalente Definition verwendet:

$$O(f) = \{g \mid \exists c, N_0 \in \mathbb{N} : \forall N \geq N_0 : g(N) \leq c * f(N)\}$$

Bei Verwendung der O-Notation wird die Funktion f meist durch einen Ausdruck angegeben, der abhängig von N ist. So bezeichnet $O(N^2)$ beispielsweise die Menge $O(f)$ für $f(N) = N^2$.
Die O-Notation lässt sich außerdem auf Funktionen mit mehreren Eingabegrößen verallgemeinern. Beispielsweise hat die Multiplikation einer $m \times n$ Matrix mit einer $n \times k$ Matrix einen Aufwand in $O(n \cdot m \cdot k)$.

Beispiel Der Zeitaufwand T vom Sortieren durch Auswahl ist für $N \geq 2$ nach oben abschätzbar durch:

$$T_A(N) \leq 2N^2 + 8N - 7$$

und damit in $O(N^2)$; denn mit $c = 3$ und $d = 9$ gilt:

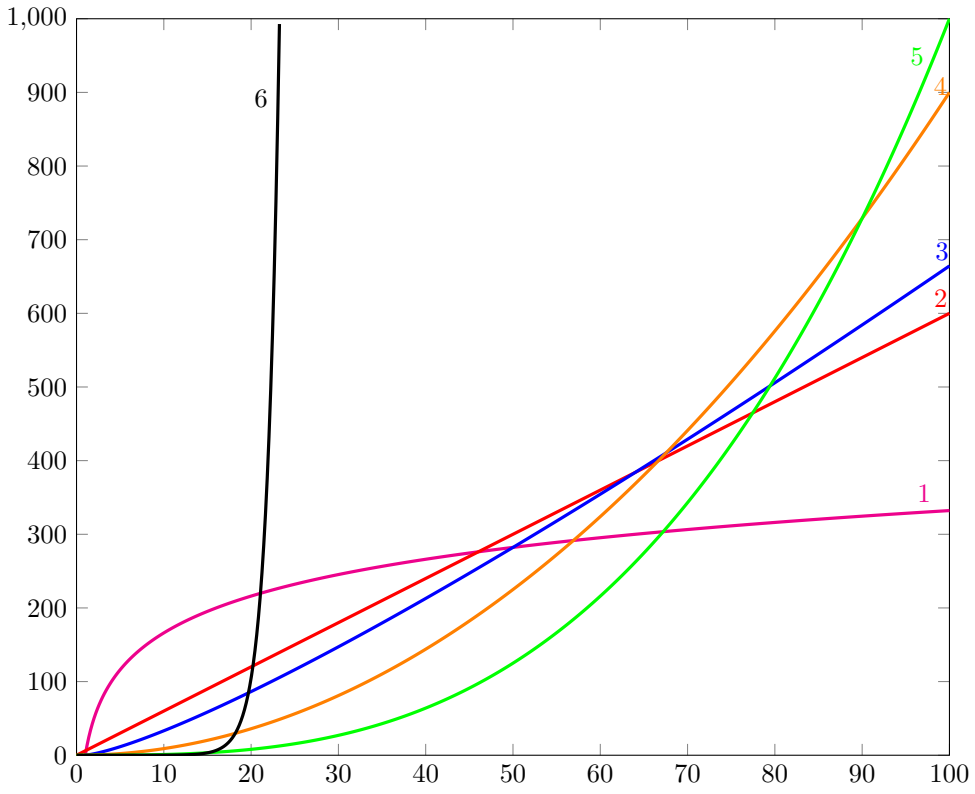
$$T_A(N) \leq 3N^2 + 9$$

Wichtige Komplexitätsklassen Die folgende Tabelle gibt eine Übersicht über wichtige Komplexitätsklassen und ihr Verhalten im Bezug auf ein Verdoppeln der Eingabegröße.

Klasse	Verdoppeln	Bezeichnung	Beispiel
$O(1)$	*1	*konstant*	Schaltjahrberechnung, Hashverfahren
$O(\log(N))$	-	*logarithmisch*	binäre Suche in Bäumen
$O(N)$	*2	*linear*	Mittelwerts von Arrayeinträgen
$O(N \cdot \log(N))$	-	*linearithmetisch*	Mergesort
$O(N^2)$	*4	*quadratisch*	Sortieren durch Auswahl
$O(N^3)$	*8	*kubisch*	Matrixmultiplikation
$O(2^N)$	-	*exponentiell*	diverse Optimierungsverfahren

Algorithmen mit einem Aufwand in $O(N^k)$ für ein festes k nennt man ***polynomiell*** (engl. *polynomial*).
In der Regel wird versucht eine möglichst präzise obere Schranke zu finden. Die Aufwandsfunktion für den Selectionsort-Algorithmus liegt Beispielsweise auch in $O(N^3)$, die Aussage, dass der Aufwand in $O(N^2)$ enthält jedoch mehr Informationen.

Frage 3: Ordnen Sie die Funktionsgraphen in folgender Abbildung den oben gezeigten Komplexitätsklassen zu.



2.6 Laufzeitabschätzung von Quicksort

Die Laufzeit von Quicksort hängt essentiell vom Partitionierungsschritt und der Wahl des Pivotelements ab, da dies über die Anzahl der rekursiven Aufrufe entscheidet. Wir betrachten für Quicksort den günstigsten und den ungünstigsten Fall getrennt voneinander. Sei n die Länge des betrachteten Arrays.

Günstigster Fall Im besten Fall wird in jedem Partitionierungsschritt das aktuelle Teilarray in zwei gleichgroße Teile partitioniert (das Pivotelement ist dann Median des aktuellen Teilarrays). Der rekursive Aufruf erfolgt dann auf Arrays halber Länge, genauer $\frac{n-1}{2}$, da das Pivotelement nicht Teil der rekursiven Aufrufe ist. Für den Aufrufbaum ergibt sich daher ein balancierter Baum der Tiefe von $O(\log n)$.
Bei der Partitionierung wird mit Hilfe von `left` und `right` das Teilarray von 0 bis $n-2$ abgelaufen. Im besten Fall sind keine Vertauschungen notwendig, allerdings wird jedes Element (genau) einmal mit dem Pivot verglichen. Damit benötigt man für diesen Schritt des Algorithmus $O(n)$ Operationen.
Auf jedem Level des Aufrufbaums wird der Scan für (fast) alle Elemente durchgeführt (Ausnahme sind die bisher ermittelten Pivotelemente). Insgesamt ergibt sich als Größenordnung für die Laufzeit im günstigsten Fall daher $O(n \cdot \log(n))$.

Das folgende Beispiel zeigt eine günstige Ausführung bei der Sortierung von 15 Zahlen. Die Pivot-Elemente sind jeweils rot markiert und fallen immer in die Mitte des aktuellen Teilarrays. Damit hat der Aufrufbaum eine Höhe von 4. Auf Level 1 wird ein Array der Größe 15 bearbeitet, auf Level 2 zwei Teilarrays der Größe 7, auf Level 3 vier Teilarrays der Größe 3 und auf dem vierten Level acht Teilarrays der Größe 1.

0	2	1	5	4	6	3	11	8	10	9	13	12	14	7

0	2	1	5	4	6	3	7	8	10	9	13	12	14	11

0	2	1	3	4	6	5	7	8	10	9	11	12	14	13

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14

Ungünstigster Fall Im ungünstigsten Fall wird das maximale (oder minimale) Element in jedem Partitionierungsschritt gewählt. Für den rekursiven Aufruf ergibt es sich dann, dass einmal `quicksort` auf einem Teilarray der Größe 0 und einmal auf einem Teilarray der Größe $k-1$ aufgerufen wird. Damit ist der Aufrufbaum zu einer Liste degeneriert und hat eine Tiefe von $O(n)$.
Auch im ungünstigsten Fall wird beim Scan mit `left` und `right` der Abschnitt von `ug` bis `og-1` einmal abgelaufen. Dabei wird im ungünstigsten Fall jedes Element (genau einmal) mit dem Pivot verglichen und mit einem anderen Element vertauscht. Damit benötigt man auch im ungünstigsten Fall für diesen Schritt des Algorithmus $O(k)$ Operationen.
Auf jedem Level wird der Scan auch im ungünstigen Fall für (fast) alle Elemente durchgeführt. Insgesamt ergibt sich als Größenordnung für Laufzeit im ungünstigsten Fall daher $O(n \cdot n)$, also $O(n^2)$.

Im folgenden Beispiel wird ein absteigend sortiertes Array mit Quicksort sortiert, was dazu führt, dass immer abwechselnd das kleinste und größte Element als Pivot gewählt wird. Dies führt zu 14 Ebenen im Aufrufbaum.

14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	13	12	11	10	9	8	7	6	5	4	3	2	1	14
0	13	12	11	10	9	8	7	6	5	4	3	2	1	14
0	1	12	11	10	9	8	7	6	5	4	3	2	13	14
0	1	12	11	10	9	8	7	6	5	4	3	2	13	14
0	1	2	11	10	9	8	7	6	5	4	3	12	13	14
0	1	2	11	10	9	8	7	6	5	4	3	12	13	14
0	1	2	3	10	9	8	7	6	5	4	11	12	13	14
0	1	2	3	10	9	8	7	6	5	4	11	12	13	14
0	1	2	3	4	9	8	7	6	5	10	11	12	13	14
0	1	2	3	4	9	8	7	6	5	10	11	12	13	14
0	1	2	3	4	5	8	7	6	9	10	11	12	13	14
0	1	2	3	4	5	8	7	6	9	10	11	12	13	14
0	1	2	3	4	5	6	7	8	9	10	11	12	13	14
0	1	2	3	4	5	6	7	8	9	10	11	12	13	14

2.7 Lösbarkeit großer Probleme

Angenommen, ein Programm benötigt für ein Problem der Größe N auf einem Rechner wenige Sekunden. Wie verhält sich die Laufzeit des Programm, wenn man die Problemgröße erhöht? Um wie viel kann ein schnellerer Rechner die Berechnungszeit verringern?
Wenn man die Problemgröße um den Faktor 100 erhöht, wächst die Laufzeit von einigen Sekunden auf ...

Klasse	Laufzeitabschätzung
linear	einige Minuten
linearithmetisch	einige Minuten
quadratisch	mehrere Stunden
kubisch	einige Wochen
exponentiell	Milliarden von Jahren

Hilft uns die Investition in besserer Hardware? Wenn man einen Computer verwendet, der um den Faktor 10 schneller rechnet, kann in der gleichen Zeit eine Probleminstanz gelöst werden, die um den Faktor ... größer ist.

Klasse	Faktor der Erhöhung der Problemgröße
linear	10
linearithmetisch	10
quadratisch	3-4
kubisch	2-3
exponentiell	1

Bemerkungen Die O-Notation erlaubt eine (vergleichsweise einfache) theoretische Einordnung und Vergleich von Algorithmen. Sie liefert aber nur eine grobe Klassifizierung durch eine obere Schranke. Der Fokus bei der Verwendung der O-Notation liegt auf dem asymptotischen Verhalten bei Programmen bei großen Eingaben.
In der Praxis können konstante Faktoren und programmiersprachenspezifische Aspekte entscheidend sein; diese sollten dann auch entsprechend im Kostenmodell reflektiert werden.

Beispiel: Stringrepräsentation in Java Die Repräsentierung von Strings hat großen Einfluss auf Laufzeit (und auch Speicher!) von Stringoperationen. Typischerweise wurden Strings vor Java 1.7 in JDK als `char[]` mit Startposition im Array und Länge repräsentiert. Dies erlaubte es Substring-Extraktion in konstanter Zeit durchzuführen ($O(1)$):

```
1 String fullName = "Bill Gates";
2 String firstName = fullName.substring(0, 3);
```

Seit Java 1.7 teilen sich Substrings nicht mehr das gleiche `char[]`-Array, sondern erhalten jeweils eine eigene Kopie des Sub-Arrays. Die Substring-Extraktion ist erfordert nun das Kopieren von N Charactern ($O(N)$), wobei N die Länge des Substrings ist.

3 Warnung!

> We should forget about small efficiencies, say about 97% of the time: premature optimization is the root of all evil. Yet we should not pass up our opportunities in that critical 3%.
> [Donald Knuth, 1974]

Es ist essentiell das Laufzeitverhalten eines Programms bei der bereits bei der Planung und Gestaltung im Blick zu behalten. Die Wahl des richtigen Algorithmus' kann entscheidend sein, auch große Probleminstanzen mit vetretbarer Laufzeit und Speicherverbrauch lösen zu können.
Die meisten Optimierungen auf Code-Ebene (z.B. "Einsparen" von Variablen) sind allerdings kontraproduktiv:

- Sie sind meist komplex und schwer zu verstehen.
- Sie liefern nur geringe Verbesserungen.
- Sie sind oft inkorrekt.

Unser Ziel ist zunächst klarer und korrekter Code!

Hinweise zu den Fragen

Hinweise zu Frage 3: Die gezeigten Funktionen sind:

#	Farbe	Funktion	Komplexitätsklasse
1	Magenta	$50 \cdot log_2(n)$	$O(log(n))$
2	Rot	$6 \cdot n$	$O(n)$
3	Blau	$n \cdot log_2(n)$	$O(n \cdot log(n))$
4	Orange	$0,09 \cdot n^2$	$O(n^2)$
5	Grün	$0,001 \cdot n^3$	$O(n^3)$
6	Schwarz	$0,0001 \cdot 2^n$	$O(2^n)$