Universität des Saarlandes
FR 6.2 Informatik
Prof. Dr. Hans-Peter Lenhof
Dipl. Inform. Andreas Hildebrandt

Programmierung II, SS 2003
Musterlösung zur 11. Übung

Aufgabe 1: Wissenstest (0 Punkte)

(a) Eine *Template-Klasse* kann in C++ dazu benutzt werden, Datenstrukturen und Algorithmen so unabhängig wie möglich vom Typ der Eingabedaten u.Ä. zu implementieren. So können z.B. in einem `vector<T>` Elemente eines beliebigen Typs `T` gespeichert werden, z.B. `int` - Variablen in einem `vector<int>`.

(b) Sei $F := \{f | f : \mathbb{N} \to \mathbb{R}_0^+\}$. Dann definieren wir für $g \in F$:

$$\Theta(g) := \{f \in F | \exists c_1, c_2 > 0 \wedge \exists n_0 \in \mathbb{N} : \forall n \geq n_0 : c_1 g(n) \leq f(n) \leq c_2 g(n)\}$$

(c) **Doppelt verkettete sortierte Liste:**

- *Suche* geht bei der doppelt verketteten sortierten Liste in $\mathcal{O}(n)$, denn wir müssen uns von vorne oder von hinten an durchhangeln bis wir das Element gefunden haben und können dabei die Ordnungseigenschaft nicht ausnutzen
- wenn *vorne anfügen* die Sortierung zerstören darf, dann geht es offenbar in $\mathcal{O}(1)$. Wenn wir die Eigenschaft erhalten wollen, dann müssen wir zunächst den richtigen Einfügeort suchen ($\mathcal{O}(n)$, wenn n die Länge der Liste bezeichnet) und dann an der gefunden Stelle in $\mathcal{O}(1)$ einfügen $\Rightarrow \mathcal{O}(n)$.

Unsortiertes Array:

- Da wir keine Informationen über die gespeicherten Elemente haben, müssen wir sie bei der *Suche* u.U. alle überprüfen bis wir das gesuchte Element gefunden haben $\Rightarrow \mathcal{O}(n)$
- Da wir das gesamte bisher gespeicherte Array nach hinten verschieben müssen um Platz für das neue Element zu schaffen, läuft hier *vorne anfügen* in $\mathcal{O}(n)$.

(d) (i) linear Probing: wenn eine Kollision auftritt, wird so lange ein neuer Hashwert mit einer Linearkombination des alten Hashwertes und einem Laufindex i berechnet, bis man auf ein freies Feld stößt. $h(k,i) = (h_{\text{alt}}(k) + i) \mod m$.

(ii) quadratic Probing: wie linear Probing, allerdings mit Polynom zweiten Grades: $h(k,i) = (h_{\text{alt}}(k) + c_1 i + c_2 i^2) \mod m$.

(iii) double hashing: hier wird eine zweite Hashfunktion verwendet, um neue Hashwerte zu berechnen bis ein freies Feld gefunden wird $h(k,i) = (h_{\text{alt}}(k) + i h_2(k)) \mod m$

(e) Eine endliche Menge $\mathcal{H}$ von Hashfunktionen $h : \mathcal{U} \to \{0, 1, \ldots, m-1\}$ heißt *universelle Klasse von Hashfunktionen*, wenn für jedes Paar k_i, k_j von verschiedenen Schlüsseln gilt: die Zahl der Funktionen $h \in \mathcal{H}$ mit $h(k_i) = h(k_j)$ ist höchstens $\frac{|\mathcal{H}|}{m}$.

(f) In einem Binärbaum definiert man die *Balance eines Knotens* `v` durch $\text{bal}(v) = h_l - h_r$, wobei h_l die Höhe seines linken Unterbaumes, h_r die Höhe seines rechten Unterbaumes bezeichnet. Dann ist ein Knoten `v` offenbar *balanciert*, wenn sich die Höhe der beiden Teilbäume um maximal 1 unterscheidet[1] ($\Rightarrow \text{bal}(v) \in \{-1, 0, 1\}$). Ein *AVL–Baum* ist ein Binärbaum, in dem alle Knoten balanciert sind.

(g) Ein *Circuit* oder *Rundgang* in einem Graphen ist ein Pfad, bei dem der erste und der letzte Knoten identisch sind. Ein *Euler-Circuit* ist ein solcher Rundgang, der *jede Kante* des Graphen *genau einmal* enthält.

(h) (i) Berechne die finishing_time (FT) für jeden Knoten mit DFS(G).

(ii) Berechne den transponierten Graphen G^t.

(iii) Berechne DFS(G^t), wobei die Knoten in der Reihenfolge ihrer FT's aus der DFS(G) - Berechnung im ersten Schritt (fallend) in der Hauptschleife von DFS(G^t) abgearbeitet werden.

(iv) Gib die Knoten eines jeden Baumes des DF-Waldes, der im dritten Schritt berechnet wurde, als eine starke Zusammenhangskomponente aus.

Aufgabe 2: C++ (10 Bonuspunkte)
Zunächst spezifizieren wir das Klasseninterface:

```
template <typename T>
class Baum24
{
  // Wir brauchen eine Klasse f"ur die Baumknoten, die wir
  // direkt in unseren Namensraum einf"ugen
  class Node24
  {
    public:
      // Wir brauchen keinen Defaultkonstruktor und basteln nur einen Konstruktor
      // der direkt die Knotenbeschriftung mit uebernimmt
      Node24(const T& neuer_wert);

      // Der Destruktor
      ~Node24();

      // Wir brauchen hier nicht zu kapseln, da die Klasse eh nur vom 2-4 - Baum
      // verwendet wird
      T wert;
```

[1] Wir müssen einen Unterschied von 1 erlauben, um z.B. einen balancierten Baum mit 4 Knoten überhaupt definieren zu können!

```
    // Die Kinder
    Node24 *kind[4];

    // Der Vater
    Node24 *vater;

    // Die Kindanzahl
    int grad;
};

public:
  // Der Default-Konstruktor des 2-4 - Baumes
  Baum24();

  // Ein praktischer Konstruktor, der ein Array nimmt und daraus einen Baum macht
  Baum24(const vector<T>& v);

  // Der Copy-Konstruktor
  Baum24(const Baum24<T>& b);

  // Der Destruktor
  ~Baum24();

  // Der Assignmentoperator
  const Baum24& operator = (const Baum24<T>& b);

  // Ein paar Operationen auf dem Baum

  // Einfuegen
  void einfuegen(const T& e);

  // Loeschen
  void loesche(const T& e);

  // Maximum
  T maximum() const;

  // Minimum
  T minimum() const;

  // Suche nach einem Element
  bool suche(const T& e) const;

  // Elementanzahl
  int size() const;

protected:
```

```
    // Hier kommen auch die Rebalancierungsoperationen hin

    // veraendert n und erzeugt einen zusaetzlichen Knoten
    Node24* spalten_(Node24* n);

    // n1 und n2 werden verschmolzen, das Ergebnis in n1 gespeichert
    // und n2 geloescht
    void verschmelzen_(Node24 *n1, Node24 *n2);

    // n1 und n2 werden veraendert
    void stehlen_(Node24* n1, Node24* n2);

    Node24* wurzel_;
};


// Wir implementieren noch schnell die Suche im 2-4-Baum
template <typename T>
bool Baum24<T>::suche(const T& e) const;
{
  // der aktuelle Knoten
  Node24* k = wurzel_;

  // wir iterieren so lange bis wir unten angekommen sind oder
  // schon feststeht das wir den Knoten auf keinen Fall im Baum
  // finden k"onnen

  // wenn der e groesser als der groesste
  // Wert in unserem Teilbaum ist, dann kann es nicht im Baum
  // enthalten sein
  while ( (k.wert >= e) )
  {
    // haben wir den Wert schon gefunden?
    if (k.wert == e)
    {
      return true;
    }

    // Wenn nicht, sind wir schon bei den Blaettern angekommen?
    if (k.grad == 0)
    {
      return false;
    }

    // Sonst muessen wir das Kind suchen, in dessen Teilbaum
    // wir uns umsehen muessen
    int i=0;
```

```
    while ( k.kind[i] && (i < 4) && (k.kind[i].wert < e) )
    {
      i++;
    }

    k = k.kind[i];
  }

  // wenn wir hier ankommen, haben wir den Knoten nicht gefunden
  return false;
}
```

Aufgabe 3: $\mathcal{O}$-Notation (10 Bonuspunkte)

- Es sei $f(n) = \sum_{i=0}^{k} g_i(n)$, und es gelte für einen Index $i \in \{0, \ldots, k\} : \forall j \neq i, j \in \{0, \ldots, k\} : g_j(n) \in \mathcal{O}(g_i(n))$.
  zu Zeigen: $f(n) \in \Theta(g_i(n))$
  Beweis:

  $$g_j(n) \in \mathcal{O}(g_i(n)) \Leftrightarrow \exists\, c_j > 0, n_{0j} > 0, n_0 \in \mathbb{N} : \forall n \geq n_{0j} : 0 \leq g_j(n) \leq c_j g_i(n)$$

  Sei $\mathbb{N} \ni n_0 := \max\{n_{00}, \ldots, n_{0k}\}$, $c := \max\{c_0, \ldots, c_k\}$. Dann gilt offenbar für alle Funktionen g_j: $\forall n \geq n_0 : 0 \leq g_j(n) \leq c g_i(n)$, und damit natürlich auch

  $$\forall n \geq n_0 : 0 \leq \sum_{j=0}^{k} g_j(n) \leq \underbrace{kc}_{:=c'} g_i(n) = c' g_i(n) \Rightarrow f(n) \in \mathcal{O}(g_i(n))$$

  Andererseits gilt natürlich auch, da $g_j(n) \geq 0 \forall j$, dass

  $$f(n) = \sum_{j=0}^{k} g_j(n) \geq g_i(n) \geq 0 \Rightarrow f(n) \in \Omega(g_i(n))$$

  . Damit haben wir also $f(n) \in \Omega(g_i(n))$ und $f(n) \in \mathcal{O}(g_i(n))$, und daraus folgt $f(n) \in \Theta(g_i(n))$■

- Anordnung:

  $$\log_2 \log_2(n), (\log_2(n))^2, (\sqrt{2})^{\log_2(n)}, n, (\frac{3}{2})^n$$

  Begründung:

  (a) Es gilt $\frac{\frac{d}{dn} \log_2 \log_2(n)}{\frac{d}{dn} (\log_2(n))^2} = \frac{\ln(2)}{2(\ln(n))^2}$, also $\lim_{n\to\infty} \frac{\log_2 \log_2(n)}{(\log_2(n))^2} \underset{\text{L'H}}{=} 0$ und damit auch das $\log_2 \log_2(n) \in \mathcal{O}((\log_2(n))^2)$■.

  (b) Zuerst formen wir um: $(\sqrt{2})^{\log_2(n)} = 2^{\frac{1}{2} \log_2(n)} = 2^{\log_2 n^{\frac{1}{2}}} = n^{\frac{1}{2}} = \sqrt{n}$. Damit können wir dann bilden: $\lim_{n\to\infty} \frac{(\log_2(n))^2}{\sqrt{n}} \underset{\text{L'H}}{=} \lim_{n\to\infty} \frac{4 \ln(n)}{\ln(2)^2 \sqrt{n}} = 0$ und damit folgt $(\log_2(n))^2 \in \mathcal{O}(\sqrt{(}n))$■.

(c) Das $\sqrt{n} \in \mathcal{O}(n)$ folgt sofort mit $\lim_{n\to\infty} \frac{\sqrt{n}}{n} = \lim_{n\to\infty} \frac{1}{\sqrt{n}} = 0$■.

(d) Es gilt: $\lim_{n\to\infty} \frac{n}{(\frac{3}{2})^n} \underset{\text{L'H}}{=} \lim_{n\to\infty} \frac{1}{(\frac{3}{2})^n \ln(\frac{3}{2})} = 0$ und damit folgt sofort $n \in \mathcal{O}((\frac{3}{2})^n)$■.

Hinweis: Bei dieser Aufgabe werden auch ein wenig informellere Begründungen akzeptiert...

Aufgabe 4: Rekurrenzgleichungen, Substitutionsmethode (10 Punkte)
Gegeben ist die Rekurrenz:

$$T(n) = 3T(\frac{n}{5}) + cn^2$$

Wir vermuten (z.B. durch den Rekursionsbaum), dass $T(n) \in \Theta(n^2)$, und beweisen dies durch vollständige Induktion:
zu Zeigen: $T(n) \in \Theta(n^2)$
Beweis:

(a) **zu Zeigen:** $T(n) \in \mathcal{O}(n^2)$
Beweis: I.A. (n=1): $T(1) = 1 \leq d1^2 \forall d \geq 1$
I.S.:

$$\begin{aligned} T(n) &= 3T(\frac{n}{5}) + cn^2 \\ &\underset{\text{I.A.}}{\leq} 3d\frac{n^2}{25} + cn^2 \\ &= dn^2\left(\frac{3}{25} + \frac{c}{d}\right) \\ &\leq dn^2 \forall d \geq \frac{c}{1-\frac{3}{25}} \blacksquare \end{aligned}$$

(b) **zu Zeigen:** $T(n) \in \Omega(n^2)$
Beweis: I.A. (n=1): $T(1) = 1 \geq d1^2 \forall d \leq 1$
I.S.:

$$\begin{aligned} T(n) &= 3T(\frac{n}{5}) + cn^2 \\ &\underset{\text{I.A.}}{\geq} 3d\frac{n^2}{25} + cn^2 \\ &= dn^2\left(\frac{3}{25} + \frac{c}{d}\right) \\ &\geq dn^2 \forall d \leq \frac{c}{1-\frac{3}{25}} \blacksquare \end{aligned}$$

Aufgabe 5: Mastertheorem (10 Bonuspunkte)

- $T(n) = 49T(\frac{n}{7}) + cn^2 + d$
  Hier gilt $n^{\log_b(a)} = n^{\log_7(49)} = n^2$, und damit $f(n) := cn^2 + d \in \Theta(n^{\log_b(a)})$. Also ist Fall 2 des Mastertheorems anwendbar, und es folgt $T(n) \in \Theta(n^2 \log_2(n))$

- $T(n) = 63T(\frac{n}{4}) + 17n^3$
  Hier gilt $2 < \log_4(63) < 3$, und mit $f(n) := 17n^3$ und $\varepsilon := 3 - \log_4(63)$ können wir folgern: $f(n) \in \Omega(n^{\log_4(63)+\varepsilon})$. Außerdem gilt: $af(\frac{n}{b}) = 63 \times 17\frac{n^3}{64} = \underbrace{\frac{63}{64}}_{c \leq 1} 17n^3$ Daher ist Fall 3 des Mastertheorems anwendbar, und es folgt: $T(n) \in \Theta(n^3)$.

- $T(n) = 64T(\frac{n}{8}) + n\ln(n) + n$
  Hier ist $n^{\log_8(64)} = n^2$ und $f(n) := n\ln(n) + n$. Es gilt

$$\lim_{n\to\infty} \frac{n\ln(n)}{n^{2-\varepsilon}} = \lim_{n\to\infty} \frac{n^{1+\varepsilon}\ln(n)}{n^2}$$
$$\underset{\text{L'H}}{=} \lim_{n\to\infty} \frac{\ln(n) + 1 + \varepsilon\ln(n)}{2n^{1-\varepsilon}} \underset{0<\varepsilon<1}{=} 0$$

  Und damit folgt $f(n) \in \mathcal{O}(n^{2-\varepsilon})$ für ein $\varepsilon > 0$. Also können wir Fall 1 des Mastertheorems anwenden, und es folgt: $T(n) \in \Theta(n^2)$.

Aufgabe 6: Heaps (10 Bonuspunkte)
Unter einem (Max-)Heap verstehen wir einen Baum mit der sogenannten *Heap-Eigenschaft:* der in einem Knoten gespeicherte Wert ist immer größer oder gleich (bei einem Min-Heap: kleiner oder gleich) den in den Kindern gespeicherten Werten. Ein binärer Heap ist natürlich ein Binärbaum mit dieser Eigenschaft.
Da wir schon zwei korrekte Heaps A und B als Eingabe bekommen, stellt sich die Frage, wie sich diese zu einem binären Heap C vereinigen lassen, der alle Elemente von A und B enthält. Wir geben einen Algorithmus in $C++$-Syntax an, der dieses Problem in $\mathcal{O}(\log(m+n))$ löst (hierbei seien A und B o.B.d.A. Max-Heaps, A enthalte m Elemente, B enthalte n Elemente):

```
BinHeap joinHeaps(BinHeap& A, BinHeap& B)
{
  // zuerst klauen wir dem Heap A die Wurzel
  Node& n = A.root;
  A.delete_max(); // Da das Maximum in der Wurzel stand, haben wir nun
                  // die urspruengliche Wurzel aus A geloescht und einen
                  // neuen korrekten Heap gebastelt. Dies geht in O(log(m))

  // Nun basteln wir einen neuen binaeren Baum, der die urspruengliche Wurzel
  // von A als Wurzel hat und den aus A durch delete_max hervorgegangenen
  // (korrekten) binaeren Heap als linkes, B als rechtes Kind der Wurzel
  n.left_child  = A.root;
  n.right_child = B.root;
```

```
  BinHeap C;
  C.root = n;

  // Die beiden Teilbaeume unter der Wurzel unseres neuen Heaps sind nun korrekte
  // Heaps. Nur die Wurzel verletzt u.U. noch die Heap-Eigenschaft, was wir aber
  // durch ein push_down einfach reparieren koennen. Da in C (m+n) Elemente
  // gespeichert sind, liegt die Hoehe von C (und damit auch die Laufzeit von
  // push_down) in O(log(m+n))
  C.push_down(C.root);

  // Und damit sind wir fertig. C ist ein korrekter binaerer Heap der alle m+n
  // Elemente von A und B enthaelt, und wir haben log(m) + log(m+n) = O(log(m+n))
  // Operationen dafuer benoetigt.
  return C;
}
```

Aufgabe 7: Eulergraphen (10 Punkte)
Um einen Kreis in einem ungerichteten Eulergraphen $G = (V, E)$ zu identifizieren könnten wir trivialerweise alle Kanten ausprobieren. Dieser Algorithmus liefe in $\mathcal{O}(E)$, und da im worst-case jeder Knoten mit jedem verbunden sein könnte, hätten wir eine Laufzeit von $\mathcal{O}(n^2)$ zu erwarten. Andererseits macht man sich leicht klar, dass wir auf jeden Fall einen Kreis finden können der nicht mehr als $n+1$ Knoten[2] hat: angenommen wir haben einen Kreis mit $m > n+1$ Knoten gefunden. Dann gibt es ausser dem Start- und Endknoten noch mindestens einen Knoten der mehr als einmal in dem Kreis vorkommt (denn der Graph enthält ja nur n verschiedene Knoten). Also enthält der Kreis einen kleineren Teilkreis. Dies können wir solange fortführen bis wir einen Kreis mit $\leq n+1$ Knoten gefunden haben. Also werden wir versuchen einen Algorithmus zu basteln, der in $\mathcal{O}(n)$ läuft.
Dazu wählen wir nun einfach einen beliebigen Startknoten und folgen einer beliebigen Kante. Wir gelangen zu einem zweiten Knoten, von dem wir überprüfen ob wir ihn schon besucht haben. Falls ja, dann haben wir einen Kreis gefunden. Falls nein, dann verlassen wir diesen Knoten, wobei wir aufpassen müssen nicht die selbe Kante nochmal zu verwenden. Dies stellt aber kein Problem dar, da ja jeder Knoten einen geraden Grad besitzt, also auf jeden Fall noch mindestens eine andere Kante zur Verfügung steht um ihn wieder zu verlassen. Nach spätestens nach $n+1$ Schritten müssen wir nun auf einen Knoten stoßen den wir schon besucht hatten, da ja keine noch nicht besuchten Knoten mehr übrig geblieben sein können.
Eine Beispielimplementierung sieht folgendermaßen aus:

```
list<node> kreis_im_eulergraph(const graph& G)
{
  list<node> kreis; // Hier speichern wir das Ergebnis

  node_array<bool> besucht(G, false); // Hiermit markieren wir, ob wir den Knoten sch
```

[2] +1, da ja der Startknoten auf jeden Fall doppelt vorkommt

```
  // Wir brauchen einen Startknoten
  node n = G.first_node();

  // Und eine Kante mit der wir ihn verlassen
  edge e = *(G.adj_edges(n).begin());

  while ( besucht[n] == false )
  {
    besucht[n] = true; // Wir waren schonmal hier
    kreis.push_back(n);

    // Wir verlassen n ueber die Kante e
    n = target(e);

    // Und finden eine neue Kante
    edge e2 = *(G.adj_edges(n).begin());

    if (e2 == e)
    {
      e = *(++(G.adj_edges(n).begin())); // wir nehmen einfach die naechste Kante
    }
    else
    {
      e = e2;
    }
  }

  // Nun haben wir einen Knoten gefunden, den wir schon mal besucht haben. Dieser ist
  // der Endpunkt unseres Kreises
  kreis.push_back(n);

  // Wir muessen aber noch die Knoten loswerden, die vor dem ersten Auftreten von n
  // besucht wurden
  while (*(kreis.begin()) != n)
  {
    kreis.pop_front();
  }

  // Fertig.
  return kreis;
}
```

Aufgabe 8: Erwartungswerte (10 Punkte)
Wir berechnen zunächst den Erwartungswert $E[X]$ für eine Variable X mit $p(\{X = i\}) =$

$(1-p)^{i-1}p = q^{i-1}p.$

$$\begin{aligned}
E[X] &= \sum_{i=0}^{\infty} ip(\{X=i\}) = \sum_{i=0}^{\infty} ipq^{i-1} \\
&= p\sum_{i=0}^{\infty} iq^{i-1} \\
&= p\sum_{i=0}^{\infty} \frac{d}{dq}q^i \\
&= p\frac{d}{dq}\sum_{i=0}^{\infty} q^i \\
&= p\frac{d}{dq}\frac{1}{1-q} \\
&= p\frac{1}{(1-q)^2} = p\frac{1}{p^2} \\
&= \frac{1}{p}\blacksquare
\end{aligned}$$

Für die Varianz $\text{Var}[X] = E[X^2] - (E[X])^2$ berechnen wir nun $E[X^2]$:

$$\begin{aligned}
E[X^2] &= \sum_{i=0}^{\infty} i^2 p q^{i-1} \\
&= p \sum_{i=0}^{\infty} i \left(\frac{d}{dq} q^i \right) \\
&= p \sum_{i=0}^{\infty} \frac{d}{dq} (i q^i) \\
&= p \frac{d}{dq} \sum_{i=0}^{\infty} i q^i \\
&= p \frac{d}{dq} \sum_{i=0}^{\infty} q i q^{i-1} \\
&= p \frac{d}{dq} \left(q \sum_{i=0}^{\infty} i q^{i-1} \right) \\
&= p \frac{d}{dq} \left(q \sum_{i=0}^{\infty} \frac{d}{dq} q^i \right) \\
&= p \frac{d}{dq} \left(q \frac{d}{dq} \sum_{i=0}^{\infty} q^i \right) \\
&= p \frac{d}{dq} \left(q \frac{d}{dq} \frac{1}{1-q} \right) \\
&= p \frac{d}{dq} \left(q \frac{1}{(1-q)^2} \right) \\
&= p \frac{q+1}{(1-q)^3} \\
&= p \frac{q+1}{p^3} \\
&= \frac{q+1}{p^2}
\end{aligned}$$

Und damit folgt:

$$\text{Var}[X] = E[X^2] - (E[X])^2 = \frac{q}{p^2} + \frac{1}{p^2} - \frac{1}{p^2} = \frac{q}{p^2} \blacksquare$$

Aufgabe 9: Suche (10 Punkte)
Es sei $A = \{a_0, \ldots, a_{n-1}\}$ ein `vector<int>` der Länge n. A habe zusätzlich die Eigenschaft, dass $|A[i] - A[i+1]| \leq 1 \forall i \in \{0, \ldots, n-2\}$. Ausserdem gelte $x := A[0] < A[n-1] =: y$. Überlegen wir uns zunächst, was die Abstandsbedingung die wir als Voraussetzung mitbekommen bedeutet. Offenbar unterscheiden sich zwei in dem Array aufeinanderfolgende Zahlen maximal um eins. Da es sich um Integerzahlen handelt, können wir dies als eine Art

Stetigkeit auf den ganzen Zahlen auffassen und offenbar eine Art *Zwischenwertsatz* verwenden: wenn für eine Zahl z gilt: $A[i] \leq z \leq A[j]$ für zwei Indizes $i < j \in \{0, \ldots, n-1\}$, dann gibt es auch einen Index k mit $i \leq k \leq j$ und $A[k] = z$, denn aufgrund der Abstandsbedingung muss im Array jede ganze Zahl zwischen $A[i]$ und $A[j]$ mindestens einmal durchlaufen werden. Zwar kann es mehrere solcher Indizes k zwischen i und j geben (wir haben ja keinerlei Voraussetzungen an die Monotonie gestellt), da wir aber nur *irgendeinen* solchen Index finden sollen, bereitet uns dies keine Probleme.

Diese Feststellung ermöglicht es uns, trotz der fehlenden Monotonie eine *binäre Suche* durchzuführen:

```
int suche(const vector<int>& A, int z, int i, int j)
{
  // ist das Element ueberhaupt enthalten?
  if ((z < A[i]) || (z > A[j]))
  {
    return -1;
  }

  // haben wir das Element schon gefunden?
  if (A[i] == z)
  {
    return i;
  }
  if (A[j] == z)
  {
    return j;
  }

  int mitte = (i+j)/2; // bestimme die Mitte des Intervals

  if (z <= A[mitte])
  {
    return suche(A, z, i, mitte);
  }
  else
  {
    return suche(A, z, mitte, j);
  }
}
```

Dies unterscheidet sich nicht wesentlich von der bekannten binären Suche, und wir erhalten für die worst-case Laufzeit die Rekurrenz:

$$T(n) = T(\frac{n}{2}) + \Theta(1)$$

welche nach dem Mastertheorem, Fall 2, die Lösung $T(n) \in \Theta(\log_2(n))$ besitzt.
