Informatik-Seminar zum Thema Algorithmen

Veranstalter: Prof. Dr. Sebastian Iwanowski

Decrease-and-Conquer

Theresa Brandt von Fackh

Matr.-Nr.: winf 2823

Fachbereich: Wirtschaftsinformatik

Fachsemester: 5. Semester

Mai/Juni 2008

Inhaltsverzeichnis

1 Einführung

1.1 Decrease-and-Conquer Technik

Die Decrease-and-Conquer Technik wurde als solches von Prof. Dr. Anany Levitin benannt. Man findet diese Technik aber auch unter anderen Namen, wie z. B. „Chip-and-Conquer“ ([4]; S.133) oder „Induction approach“ (Udi Manber: *Algorithms - A Creative Approach*). Oft wird diese Technik als Sonderform von Divide-and-Conquer betrachtet. Decrease-and-Conquer heißt übersetzt soviel wie „Reduziere-und-Erobere“.

Die Idee hinter dieser Art von Algorithmus ist, dass man die Beziehung zwischen der Lösung eines Problems und der Lösung eines Subproblems, welches eine kleinere Instanz des Problems ist, ausnutzt.
Das Problem der Größe n wird zu einem Subproblem der Größe m reduziert, wobei m < n ist. Die Lösung des Subproblems der Größe m wird zu der Lösung des Problems der Größe n erweitert mit Hilfe der Beziehung, die zwischen dem Problem und dem Subproblem besteht. Abbildung 1 veranschaulicht diese Vorgehensweise.

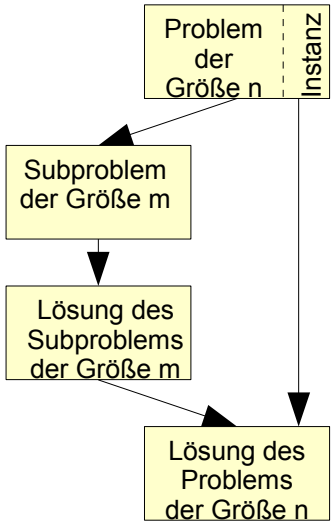


Abb. 1: Decrease-and-Conquer
Es gilt: m < n

Decrease-and-Conquer-Algorithmen können sowohl top-down als auch bottom-up implementiert werden. Top-down würde eine rekursive Umsetzung bedeuten, d.h. mit dem Subproblem wird genauso verfahren, wie beim Problem höherer Instanz, um es zu lösen. Bottom-up verwirklicht eine iterative Variante, die die Lösung der gegebenen Instanz solange anwendet, bis man zur Lösung des Problems kommt.

1.2 Varianten

Es gibt drei wichtig zu unterscheidende Varianten der Decrease-and-Conquer-Technik. Sie unterscheiden sich darin, wie die Größe der Instanz, die in jedem Iterationsschritt das Problem reduziert, festgelegt wird.

1.2.1 Decrease by a constant

Die Decrease-by-a-constant-Variante reduziert die Größe des Problems bei jedem Iterationsschritt um dieselbe Konstante. Oft hat die Konstante den Wert 1.

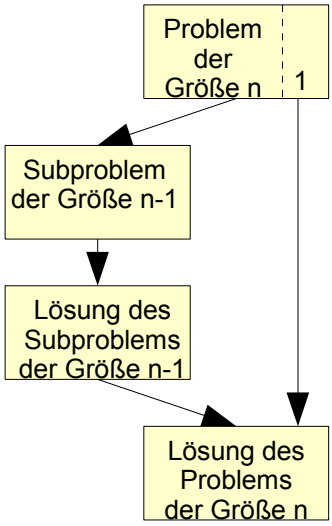


Abb. 2: Decrease-by-one

Beispiel: Algorithmus für Exponential-Problem a^n für positive ganzzahlige Exponenten n

Das Problem lautet a^n, wobei die Größe des Problems durch den Exponenten n angegeben wird. Die Konstante, um die das Problem reduziert werden soll, soll 1 betragen. Die Beziehung zwischen der Lösung des Problems und der Lösung des Subproblem ist durch folgende Formel beschrieben:

$$a^n = a^{n-1} * a^1$$

Das Subproblem würde in diesem Falle a^{n-1} lauten.

Die Formel kann top-down berechnet werden mit der folgenden rekursiven Definition:

$$f(n) = \begin{cases} f(n-1) * a & \text{für } n > 1 \\ a & \text{für } n = 1 \end{cases}$$

Die Formel kann aber auch bottom-up berechnet werden, indem man a (n-1)-mal mit sich selbst multipliziert. (Anany Levitin, Kap. 5.1)

Ein Fall, wo die Konstante den Wert 2 hat, wäre wenn ungerade und gerade Größen unterschiedlich behandelt werden sollen.
Die Decrease-by-a-constant-Variante wird z. B. bei InsertionSort, beim topologischen Sortieren oder bei der Permutationsgenerierung verwendet. Auf diese drei Verfahren wird später noch näher drauf eingegangen.

1.2.2 Decrease by a constant factor

Die Decrease-by-a-constant-factor-Variante reduziert die Größe des Problems bei jedem Iterationsschritt um denselben konstanten Faktor. Üblicherweise hat der konstante Faktor den Wert 2.

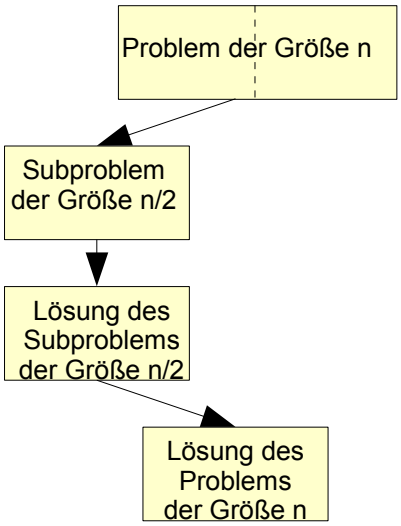


Abb. 3: Decrease-by-half

Es gibt nicht viele Beispiele für diese Variante, da diese Algorithmen normalerweise logarithmisch, somit sehr schnell, sind und nicht sehr häufig vorkommen. Ein anderer Wert als 2 für den konstanten Faktor ist höchst selten. Die Decrease-by-a-constant-factor-Variante wird z. B. bei der binären Suche, beim Fake-Coin Problem, bei der Multiplikation à la Russe oder beim Josephus Problem verwendet.

Beispiel: Fake-Coin Problem

Es gibt verschiedene Versionen dieses Problems. Um die Decrease-by-a-constant-factor-Variante zu verdeutlichen, soll das Problem folgendes sein:
Das Problem ist, eine gefälschte Münze unter einer Menge von identischen Münzen festzustellen. Mit einer Waage lässt sich das Gewicht zweier Mengen von Münzen vergleichen. Es lässt sich herausstellen, ob die Mengen von

Münzen gleich schwer sind oder welche Menge die schwerere ist. Es lässt sich jedoch nicht feststellen wie viel schwerer die Menge ist.
In dieser einfachen Version des Problems, nehmen wir an, dass die gefälschte Münze leichter sei als eine Original-Münze. Die zu untersuchende Menge von Münzen hat die Größe n. Man müsste nun die Menge von Münzen in zwei gleichgroße Mengen zerlegen, so dass zwei Mengen der Größe $\lfloor n/2 \rfloor$ entstehen. Sollte n ungerade sein, so wird eine Münze beiseite gelegt. Nun werden die beiden Mengen auf der Waage gewogen. Sind die Mengen gleich schwer, so ist die beiseite gelegte Münze die gefälschte Münze. Ist eine Menge leichter als die andere, so wird nur noch diese näher betrachtet, indem man die gleiche Prozedur auf diese Menge anwendet. Dies geschieht solange bis man die gefälschte Münze gefunden hat.
Dieser Algorithmus ist jedoch nicht der effizienteste. Man könnte die Menge auch in 3 Mengen zerlegen, d.h. der konstante Faktor beträgt nicht 2, sondern 3.. Darauf soll an dieser Stelle nicht näher drauf eingegangen werden. (Anany Levitin, Kap. 5.5)

1.2.3 Variable-size decrease

Bei der Variable-size-decrease-Variante unterscheidet sich die Reduktion der Größe des Problems bei jedem Iterationsschritt.
Die Variable-size-decrease-Variante wird z. B. beim Euklidischen Algorithmus (größter gemeinsamer Teiler), bei der Mediansuche oder bei der Interpolationssuche verwendet. Auf die letzten beiden Probleme wird später noch näher drauf eingegangen.

1.3 Vergleich Divide-and-Conquer und Decrease-and-Conquer

Bei der Decrease-by-a-constant-factor-Variante lässt sich eine Ähnlichkeit zu der Divide-and-Conquer-Technik feststellen. Nämlich dass beide Techniken ein Problem lösen indem sie kleinere Instanzen des Problems lösen. Es bestehen aber wichtige Unterschiede.
Bei der Divide-and-Conquer-Technik wird das Problem in gleichartige Teilprobleme aufgeteilt. Diese werden rekursiv gelöst und zur Lösung des Gesamtproblems zusammengeführt (siehe Abb. 4). Es werden gleich mehrere Subprobleme gelöst.
Bei der Decrease-and-Conquer-Technik wird das Problem in eine kleinere Instanz reduziert, welche dann rekursiv gelöst wird (siehe Abb. 3). Es wird also nur ein Subproblem gelöst.

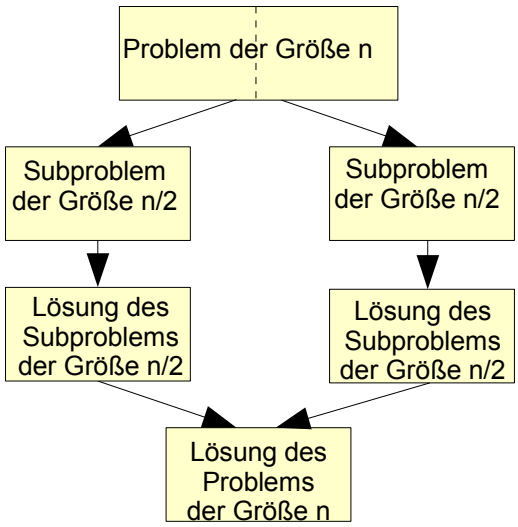


Abb 4: Divide-and-Conquer

Beispiel: Algorithmus für Exponential-Problem a^n für positive ganzzahlige Exponenten n

Decrease-and-Conquer:

Das Problem lautet a^n, wobei die Größe des Problems durch den Exponenten n angegeben wird. Der konstante Faktor, um den das Problem reduziert werden soll, soll 2 betragen. Die Beziehung zwischen der Lösung des Problems und der Lösung des Subproblem wäre dann durch folgende Formel beschrieben:

$$a^n = (a^{n/2})^2$$

Das Subproblem würde in diesem Falle $a^{n/2}$ lauten. Da n ganzzahlig sein soll, gilt für ungerade n die Beziehung:

$$a^n = (a^{(n-1)/2})^2 * a$$

Das Subproblem würde in diesem Falle $a^{(n-1)/2}$ lauten.

Die Formel kann top-down berechnet werden mit der folgenden rekursiven Definition:

$$f(n)= \begin{cases} f(n/2)^2 & \text{für } n > 1 \text{ und gerade} \\ f((n-1)/2)^2 * a & \text{für } n > 1 \text{ und ungerade} \\ a & \text{für } n = 1 \end{cases}$$

Ein Divide-and-Conquer Algorithmus hätte folgende rekursive Definition:

$$f(n) = \begin{cases} a^{\lfloor n/2 \rfloor} * a^{\lceil n/2 \rceil} & \text{für } n > 1 \\ a & \text{für } n = 1 \end{cases}$$

Der Decrease-and-Conquer-Algorithmus ist schneller als der Divide-and-Conquer-Algorithmus, da dieser logarithmisch ist. (Anany Levitin, Kap. 5.1)

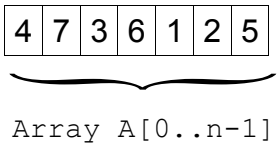
2 InsertionSort

2.1 Einführendes Beispiel

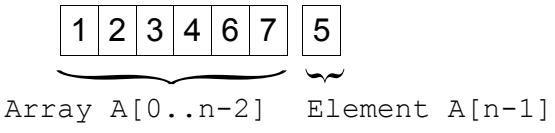
InsertionSort meint soviel wie „sortieren durch direktes Einfügen“. Das beste Beispiel aus dem Alltag für diese Methode ist das Sortieren von Spielkarten. Man kann oft bei Kartenspielern beobachten, dass sie ihre Karten auf der Hand nach folgender Methode einsortieren:
Jede neu aufgenommene Karte wird verglichen mit den schon bisher aufgenommenen Karten und an entsprechender Stelle eingefügt. Das Blatt auf der Hand ist somit immer sortiert. Der Haufen, von dem die einzusortierenden Karten aufgenommen werden, ist unsortiert. Alle sich auf der Hand befindenden Karten waren ursprünglich auch auf dem Aufnahmehaufen. ([2], S.15 ff.)

2.2 Algorithmus

Es liegt folgendes Problem vor: Ein Array A mit n sortierbaren Elementen liegt unsortiert vor.



Mit der Decrease-by-a-constant-Variante wird das Problem um die Konstante 1 reduziert, so dass wir ein Subproblem erhalten, welches ein zu sortierendes Array der Größe n-1 umfasst. Das Subproblem ist dann gelöst, wenn das Array der Größe n-1 sortiert ist, also wenn A[0] ≤ ... ≤ A[n-2].
Um das Problem zu lösen wird das Element A[n-1] in das bereits sortierte Array, welches durch Lösen des Subproblems entstanden ist, einsortiert.



Es gibt nun drei Alternativen, um das Element an entsprechender Stelle einzufügen:

1. Das sortierte Array A[0..n-2] der Größe n-1 wird **von links nach rechts** durchsucht bis das erste Element gefunden wurde, welches ≥ **A[n-1]** ist, also dem einzufügenden Element. Das Element A[n-1] wird dann **links von** diesem gefundenen Element

eingefügt.

2. Das sortierte Array A[0..n-2] der Größe n-1 wird **von rechts nach links** durchsucht bis das erste Element gefunden wurde, welches ≤ **A[n-1]** ist, also dem einzufügenden Element. Das Element A[n-1] wird dann **rechts von** diesem gefundenen Element eingefügt.
3. Es wird die Binäre Suche angewandt, um die richtige Stelle im sortierten Array für das einzufügende Element A[n-1] zu finden. Diese Alternative wird auch als „binary insertion sort“ bezeichnet.

Die Alternativen 1 und 2 werden „(straight) insertion sort“ genannt. Die dritte Alternative außen vorgelassen, wird meistens bei InsertionSort die Alternative 2 gewählt, da diese schneller bei sortierten und fast-sortierten Arrays arbeitet. Dies hängt damit zusammen, dass bei der Alternative 2 immer nur zwei benachbarte Elemente im Array verschoben werden müssen.

InsertionSort basiert auf einer rekursiven Idee, es ist jedoch effizienter diesen Algorithmus bottom-up, d.h. iterativ, zu implementieren.

```
ALGORITHMUS InsertionSort (A)
 // Sortiert ein Array der Größe n mit InsertionSort
 // Input: Ein Array A[0..n-1] mit n sortierbaren
           Elementen
 // Ouput: Ein aufsteigend sortiertes Array A[0..n-1]
 // Variablen: j = Laufvariable im sortierten Array
 //            i = Laufvariable im gesamten Array
 //            element = einzufügendes Element
 for i ← 1 to länge(A) do
   element ← A[i]
   j ← i - 1
   while j ≥ 0 and A[j] > element do
     A[j + 1] ← A[j]
     j ← j - 1
   A[j + 1] ← element
```

Abb. 5: Pseudocode InsertionSort (Alternative 2)

Das einzufügende Element wird zunächst in einer lokalen Variablen 'element' zwischengespeichert, so dass der ursprüngliche Platz im Array überschrieben werden kann und somit eine Lücke darstellt. Dann wird wiederholt 'element' mit dem Element, das links von der Lücke steht, verglichen. Solange 'element' kleiner als das zu vergleichende Element ist, wird das Element in die Lücke verschoben, wo bei es dann eine Lücke

hinterlässt. ([4], S.151 ff.) Ein konkretes Beispiel wird in Abbildung 6 gezeigt:

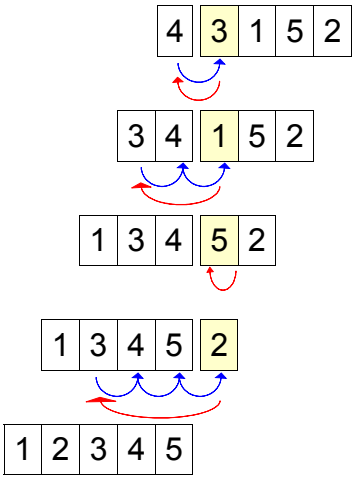


Abb. 6: InsertionSort eines nicht sortierten Arrays A (4, 3, 1, 5, 2)

2.3 Analyse

Die Basis-Operation des Algorithmus' ist der Elementenvergleich A[j] > element. Die Anzahl der Vergleiche hängt von dem Aufbau des Inputs ab. Es ist zu beachten, dass in der i-ten Iteration des InsertionSorts alle vorstehenden Elemente von A[i] die ersten i Elemente im Input sind.

Worst Case: Der Elementenvergleich wird für jedes links von der Lücke stehende Element j = i-1,...,0 ausgeführt. Dies würde dann eintreffen, wenn das jeweils einzufügende Element immer ganz nach vorn im Array muss. Somit wäre der worst case ein absteigend sortiertes Array als Input.

Best Case: Der Elementenvergleich wird nur einmal in jeder Iteration (der äußeren Schleife) angewandt, also wenn $A[i-1] \leq A[i]$ für jedes i = 1, ..., n – 1. Somit wäre der best case ein aufsteigend sortiertes Array als Input.

Average Case: Die Analyse des average cases basiert auf die Erforschung der Anzahl von Element-Paaren, die in falscher Reihenfolge stehen (i < j , A[i] > A[j]). Bei nicht-sortierten Arrays macht InsertionSort im Durchschnitt halb so viele Vergleiche wie bei einem absteigend sortiertem Array.

Durch diese doppelt-so-schnell-Durchschnittsfall-Leistung und der hohen Effizienz bei fast sortierten Arrays, ist InsertionSort der beste Algorithmus unter den elementaren Sortieralgorithmen.

3 Topologisches Sortieren

3.1 Einführendes Beispiel

In vielen Situation wird topologisch sortiert, z. B. beim Kochen, bei Projekten oder bei der Modellierung einer Fertigungsstraße, wo gewisse Handlungen vor Anderen ausgeführt werden müssen. Ein ebenfalls gutes Beispiel dafür ist, wie sich ein Student einen Studienplan selbst zusammenstellt.
Im Studium sind gewisse Veranstaltungen Voraussetzung für andere Veranstaltungen, deshalb müssen diese gewissen Veranstaltungen vor den anderen Veranstaltungen besucht werden. Der Student soll sich einen Studienplan zusammenstellen, indem er die Veranstaltungen so zusammenstellt, dass keine Veranstaltung eine später aufgeführte Veranstaltung voraussetzt. ([1], Kap. 5.3; [6], S.211)

Veranstaltung	Voraussetzung
A	-
B	A
C	-
D	B,C
E	B
F	D,E

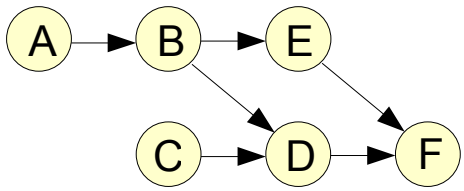


Abb. 7: Studienplan und gerichteter Graph des Studienplans

Ein Student hat die Information in Abbildung 7 vor sich liegen. Der Student würde zunächst die Veranstaltung A und C besuchen. Im nächsten Semester könnte er dann Veranstaltung B hören. Im dritten Semester hat er dann auch die Voraussetzungen erfüllt, um Veranstaltung E und D zu besuchen. Im darauf folgenden Semester könnte er dann Veranstaltung F besuchen. Diese Art die Veranstaltungen zu sortieren, nennt man topologisches Sortieren.

3.2 Definition

Das topologische Sortieren ist ein sehr wichtiges Problem für gerichtete Graphen. Knoten sollen in eine topologische Reihenfolge gebracht werden, d.h. die Knoten sollen in eine Reihenfolge angebracht werden, so dass Knoten, von denen Kanten ausgehen, vor den Knoten stehen, zu denen diese Kanten führen.
Abstrakt formuliert bedeutet topologisches Sortieren, dass eine teilweise Ordnung (mit den Eigenschaften: Transitivität, Asymmetrie und Irreflexivität) in eine lineare Ordnung eingebettet werden soll. ([6], S.210 ff.)

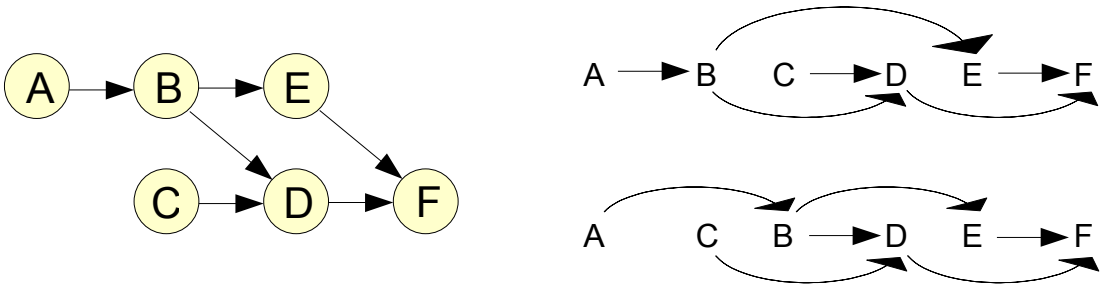


Abb. 8: Partiell geordnete Menge (links) und 2 Möglichkeiten der linearen Anordnung der Menge (rechts)

Beim topologischen Sortieren kann man zu mehreren Lösungen kommen (siehe Abb. 8). Der Graph muss ein gerichteter azyklischer Graph sein, um diesen topologisch sortieren zu können. Andernfalls könnte man nicht zu einer Lösung kommen, da bei einem Zirkelschluss die Bedingung, dass ein Knoten, von denen Kanten ausgehen, vor den Knoten stehen, zu denen diese Kanten führen, nicht erfüllbar wäre.
In der abstrakten Definition garantieren Transitivität und Asymmetrie der teilweisen Ordnung, dass der Graph keine Schleifen enthält, was eine Einbettung in eine lineare Ordnung überhaupt erst möglich macht. ([6], S.210 ff.)

3.3 Algorithmus

Bei einem kleinen Beispiel wird das Problem nicht so bewusst, da man dieses schnell mit bloßem Auge lösen kann. Aber bei größeren Projekten ist ein Algorithmus unausweichlich. Es gibt einen Algorithmus der zugleich prüft, ob es sich um einen gerichteten azyklischen Graphen handelt und die Knoten dieses Graphen in eine topologische Reihenfolge bringt.
Der Algorithmus basiert auf der Decrease-(by-one)-and-Conquer-Technik. Eine Quelle sei ein Knoten der keine eingehenden Kanten hat. Es wird wiederholt eine Quelle im restlichen gerichteten Graphen gesucht und diese wird dann mit allen ausgehenden Kanten gelöscht bis es keine Knoten mehr gibt. Gibt es mehrere Quellen, so werden diese in beliebiger Reihenfolge gelöscht. Gibt es keine Quelle, so gibt es auch keine Lösung, da in diesem Fall ein zyklischer Graph vorliegen muss. Die Reihenfolge in der die Knoten gelöscht werden, geben die topologische Reihenfolge des gerichteten Graphen an. Eine Veranschaulichung dieses Algorithmus' ist in Abbildung 9 gegeben.

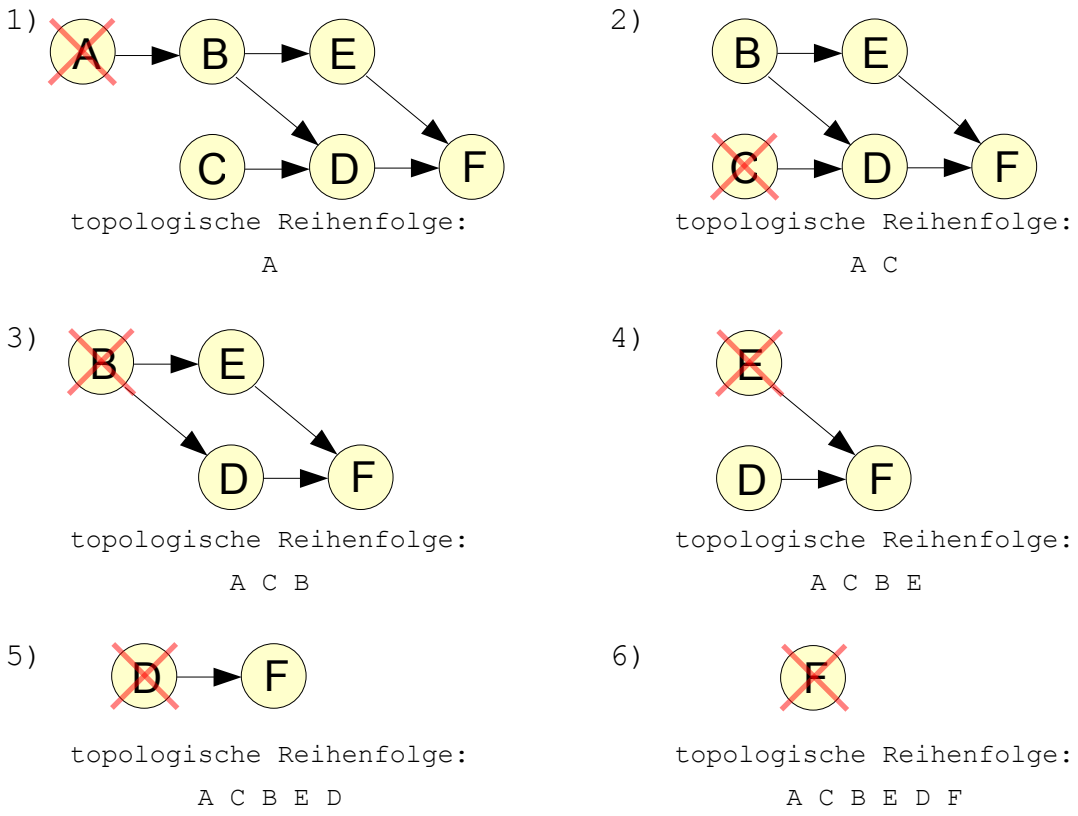


Abb. 9: Veranschaulichung des Algorithmus' für topologische Sortierung

Bei Projekten müssen die Voraussetzungen zunächst auf Widersprüchlichkeit geprüft werden. Dies kann man, indem man das Problem des topologischen Sortierens löst. Erst dann kann man sich Pläne zusammenstellen, um z. B. die Projektdauer zu optimieren mit Hilfe von OR-Algorithmen, CPM (Critical Path Method) oder PERT (Program Evaluation and Review Technique). (→ Vorlesung Operations Research und Projektmanagement)

Lässt man den Algorithmus „rückwärts“ durchlaufen, so ergibt sich eine umgekehrte topologische Reihenfolge für den Graphen. Man würde zum gleichen Ergebnis kommen, wenn man alle Kanten im Graphen umkehrt und den Algorithmus („vorwärts“) anwendet. ([3], S.545)

Der „rückläufige“ Algorithmus soll an Hand eines Beweises dafür, dass jeder azyklischer gerichteter Graph immer topologisch sortiert werden kann, gezeigt werden:

Die Abwesenheit von Zirkelschlüssen impliziert, dass ein Knoten mit keinen ausgehenden Kanten existiert (→ Endknoten). Dieser Knoten wird mit allen eingehenden Kanten eliminiert. Die zuvor ausgeführten Schritte werden mit den nun entstandenen Graphen erneut durchgeführt, bis alle Knoten eliminiert wurden. ([7], S.108)

4 Permutationsgenerierung

4.1 Einführung

Bei der Permutationsgenerierung geht es darum, alle möglichen Anordnungen von einer Menge aus unterschiedlichen Objekten zu erzeugen; weniger darum die Anzahl von Anordnungen zu berechnen. Die Anzahl von Permutationen einer Menge der Größe n errechnet man mit n!.
Permutationen zu generieren, ist zum Beispiel für das Traveling Salesman Problem von Interesse. Zu einer gegebenen Menge von n Orten, ist die kürzeste/günstigste Tour zu finden, die alle Orte verbindet. Mit der Permutationsgenerierung lassen sich alle möglichen Touren ermitteln. Im folgenden kann man dann die Kosten für jede Tour berechnen und somit die günstigste Tour herausfinden.

4.2 Minimal-Change Algorithmus

Der Minimal-Change Algorithmus um Permutationen zu generieren nutzt die Decrease-by-one Technik:
Das Problem lautet n! Permutationen zu generieren. Man gehe davon aus, dass das Subproblem, nämlich (n-1)! Permutationen zu generieren, bereits gelöst ist. Wenn nun das n-te Element in alle n möglichen Positionen unter den Elementen von jeder bereits generierten Permutation von n-1 Elementen einmal eingefügt wird, so erhält man die Lösung. Alle Permutationen wären eindeutig und deren Gesamtzahl wäre n*(n-1)! = n!.

Ein Pseudocode für einen bottom-up Algorithmus:
Solange noch unbearbeitete Permutationen in der Sublösung sind, werden folgende Schritte ausgeführt: Für die erste Permutation der Sublösung ist die Richtung von rechts nach links angegeben.

1) Füge das n-te Element in alle n möglichen Positionen in der Permutation (vor bzw. hinter den einzelnen Elementen der Permutation) in der angegebenen Richtung einmal ein. Immer wenn das n-te Element an einer Position eingefügt wurde, wird diese neu entstandene Permutation in eine Liste geschrieben.
2) Gehe zur nächsten Permutation der Sublösung, verdrehe die Richtung und starte wieder bei Schritt 1

Ein Beispiel ist in Abbildung 10 dargestellt.

Aktion	Permutationen
Unterste Sublösung	{1}
Nächst höhere Sublösung	{1,2} {2,1} 2. Element von rechts nach links einfügen
Lösung	{1,2,3} {1,3,2} {3,1,2} {3,2,1} {2,3,1} {2,1,3} 3. Element von rechts nach links einfügen — 3. Element von links nach rechts einfügen

Abb. 10: Darstellung des Minimal-Change Algorithmus mit der Menge M = {1,2,3}

Diese Reihenfolge des Einfügens erfüllt die „Minimal-Change“ Anforderung, die besagt, dass jede Permutation von seinem unmittelbaren Vorgänger erlangt werden kann, indem nur zwei benachbarte Elemente vertauscht werden. Diese Anforderung ist für die Geschwindigkeit des Algorithmus' von Vorteil.
Beispiel für den Vorteil der Minimal-Change-Anforderung: Es werden Permutationen von Städten generiert, um das Traveling Salesman Problem zu lösen. Wenn diese Permutationen mit einem Minimal-Change Algorithmus generiert werden, kann man die Länge einer neuen Tour von der Länge der „Vorgänger“-Tour in konstanter Zeit errechnen.

	a	b	c	d	e	f
a	-	2	6	5	5	3
b	2	-	3	1	2	2
c	6	3	-	4	2	3
d	5	1	4	-	4	6
e	5	2	2	4	-	5
f	3	2	3	6	5	-

Tour	Kosten
a-b-c-d-**e-f-**a	2 + 3 + 4 + **4** + 5 + **3**
a-b-c-d-**f-e-**a	2 + 3 + 4 + **6** + 5 + **5**

Abb. 11: Matrix mit Angabe der Kosten zwischen zwei Orten (links) und zwei Touren, die sich darin unterscheiden, dass zwei benachbarte Elemente vertauscht wurden, mit Markierung der Änderung (rechts)

4.3 Johnson-Trotter Algorithmus

Der Johnson-Trotter Algorithmus (bzw. Steinhaus-Johnson-Trotter Algorithmus) generiert ebenfalls Permutationen, jedoch ohne kleinere Subprobleme zu lösen.
Beim Johnson-Trotter-Algorithmus erhält jedes Element in einer Permutation einen Richtungspfeil. Ein Element k in einer Permutation wird als mobil bezeichnet, wenn sein

Richtungspfeil in die Richtung zeigt, in der eine kleinere Größe benachbart ist. Voraussetzung hierfür ist natürlich, dass die Elemente ordinal sein müssen.

Beispiel: Menge M = {1,2,3}

→ ← →
1 2 3

Element 2 ist mobil
Elemente 1 und 3 sind nicht mobil

Pseudocode des Johnson-Trotter Algorithmus:
Zunächst erhalten alle Elemente einer beliebigen Permutation einen Richtungspfeil, der nach links zeigt. Solange die Permutation ein mobiles Element enthält, werden die folgenden Schritte ausgeführt:

1) Finde das größte mobile Element in der Permutation. Wenn keines gefunden werden kann, wird der Algorithmus abgebrochen
2) Tausche dieses Element mit dem benachbarten Element, auf den das gefundene Element zeigt
3) Drehe die Richtung der Pfeile aller Elemente, die größer als das gefundene Element sind
4) Füge die somit neu entstandene Permutation der Liste hinzu und starte wieder bei Schritt 1

Beispiel:

← ← ←	← ← ←	← ← ←	→ ← ←	← → ←	← ← →
1 2 **3**	1 **3** 2	3 1 **2**	**3** 2 1	2 **3** 1	2 1 3

Der Algorithmus kann so implementiert werden, dass er proportional zu der Anzahl der Permutationen läuft. Das würde bedeuten, dass der Algorithmus sehr langsam bei größeren Mengen ist. Die Schuld liegt jedoch nicht am Algorithmus, sondern am Problem, dass einfach zu viele Permutationen generiert werden.
Die Reihenfolge der Permutationen, die mit dem Johnson-Trotter Algorithmus generiert wurde, ist nicht in lexikographischer Ordnung.

Johnson-Trotter:	123	132	312	321	231	213
lexikographische Ordnung:	123	132	213	231	312	321

Dies kann man aber mit einem weiteren Algorithmus, der die Permutationen in lexikographischer Ordnung bringt, beheben.

5 Mediansuche

5.1 Einführung

Der Median ist ein wichtiges Lagemaße in der mathematischen Statistik und bezeichnet in einer Zahlenreihe von quantitativen Elementen genau den Wert, der in der Mitte der Zahlenreihe liegt, d. h. dass eine Hälfte der Zahlenreihe kleiner und die andere Hälfte größer als dieser Wert sind. Hat die Zahlenreihe eine ungerade Anzahl von Elementen, so kann man den Median genau bestimmen mit ⌈Anzahl der Elemente / 2⌉. Liegt eine gerade Anzahl von Elementen vor, so gibt es zwei Mediane: der untere Median liegt bei (Anzahl der Elemente / 2) und er obere Median bei ((Anzahl der Elemente / 2) + 1). Im folgenden soll immer der obere Median gemeint sein.
Die Mediansuche ist ein spezieller Fall des Auswahlproblems. Das Auswahlproblem behandelt die Problematik das k-kleinste Element in einer Liste zu finden. Um in einer Liste der Größe n den Median zu finden, muss man das (⌈n/2⌉)-kleinste Element in der Liste suchen.

5.2 Algorithmus

Um in einem Array das k-kleinste Element zu finden, könnte man natürlich das Array sortieren und dann das k-ste Element des Arrays ausgeben. Die Laufzeit wäre so groß, je nachdem welcher Sortieralgorithmus verwendet wird. Jedoch ist es gar nicht notwendig das komplette Array zu sortieren, wie der folgende Algorithmus zeigt.

Ein effizienterer Algorithmus wäre das Array in zwei Subarrays aufzuteilen. Man nehme ein Element als Pivot p. Alle links stehenden Elemente von p sind kleiner und alle rechts stehenden Elemente sind größer als p.

$$a_{i_1} \dots a_{i_{s-1}} \quad p \quad a_{i_{s+1}} \dots a_{i_n}$$

Pivot-Element p befindet sich an der Position s, wo auch die Aufspaltung des Arrays erfolgt. k gibt an, nach dem wievielten kleinsten Element im Array gesucht wird.
Ist s = k, so wurde die Lösung für das Auswahlproblem gefunden (Wenn das Array mit dem Index 0 beginnt, muss es s = k-1 sein). Wenn s > k gilt, so muss sich das k-kleinste Element als k-kleinstes Element im linken Subarray befinden. Ist s < k, so wurden schon die ersten s kleinsten Elemente gefunden, deshalb muss im rechten Subarray nur noch nach dem (k-s)-kleinsten Element gesucht werden, um auf die Lösung zu kommen.

Das Problem wird solange zu einem Subproblem, welches auf gleiche Weise gelöst wird, reduziert bis eine Lösung gefunden wurde, also bis s = k. Die Größe der Reduktion des Problems unterscheidet sich von mal zu mal, was der variable-size-decrease-Technik entspricht.
Der rekursive Ansatz teilt das Problem immer in ein kleinere Instanz, welche mit dem gleichem Ansatz gelöst wird jedoch mit angepasstem k. Ein nicht-rekursiver Ansatz wäre, den Wert k nicht zu verändern, sondern den Algorithmus so lange wiederholen zu lassen bis man zu einer Lösung, also s = k, kommt.

Beispiel: Folgende Liste von 9 Zahlen liegt vor: 2, 7, 15, 4, 19, 13, 1, 3, 10

Es wird nach dem k-kleinsten Element in der Liste gesucht, wobei k = $\lceil 9/2 \rceil = 5$, wenn nach dem Median gefragt wird. Die **fett** gedruckte Zahl ist das Pivot Element. Die unterstrichenen Zahlen werden links vor dem Pivot-Element eingefügt.

2	8	15	4	19	13	1	3	7
1	**2**	8	15	4	19	13	3	7

Das Pivot-Element befindet sich nach der Sortierung an Position s=2.
Da s=2 < k=5, wird der Bereich rechts vom Pivot-Element bearbeitet.

1	*2*	**8**	15	4	19	13	3	7
1	*2*	7	3	4	**8**	15	19	13

Da s=6 > k=5, wird der Bereich links vom Pivot-Element bearbeitet.

1	*2*	**7**	3	4	*8*	*15*	*19*	*13*
1	*2*	4	3	**7**	*8*	*15*	*19*	*13*

Jetzt wo s = k = 5, wird der Algorithmus beendet. Der Median befindet sich in der letzten Liste an Stelle 5 und hat somit den Wert 7.

Der Algorithmus ist im average case linear und löst neben dem Problem, das k-kleinste Element zu finden, auch die Frage der k-kleinsten und (n-k) größten Elemente im Array.
Der Algorithmus ähnelt dem Sortieralgorithmus Quicksort, der ebenfalls mit Aufteilung eines Arrays in zwei Subarrays arbeitet. Allerdings werden bei Quicksort beide Subarrays weiter verarbeitet, was der Divide-and-Conquer-Methodik entspricht.

5.3 Implementierung

Für die Implementierung werden zwei Funktionen benötigt. Man braucht zum einen eine Funktion, die das Array in zwei Teilbereiche aufspaltet, nämlich in einen Bereich mit den Elementen, die kleiner gleich einem ausgewähltem Pivot, und einem Bereich mit den Elementen, die größer diesem Pivot sind. Zum Anderen braucht man eine rekursive Funktion, die immer wieder einen Teilbereich des Arrays mit der anderen Funktion in zwei Bereiche teilt und sich selbst mit einem dieser Bereiche erneut aufruft, bis die Lösung gefunden wurde.

Partition-Funktion

```
function partition(var List: arr; left, right: integer): integer;
var element, pivot, i, temp: integer;
begin
   pivot:= left;
   element:= pivot+1;
   while ( element <= right) do
   begin
       temp:= List[element];
       if (List[element] <= List[pivot]) then
       begin
           for i:= element downto left+1 do
              List[i]:= List[i-1];
           List[left]:= temp;
           pivot:= pivot+1;
       end;
       element:= element+1;
   end;
   partition:= pivot;
end;
```

Die Funktion hat als Eingabeparameter das Array, in welchem nach dem Median gesucht werden soll, und eine Angabe von wo bis wo das Array bearbeitet/betrachtet werden soll. Das Pivot-Element befindet sich zu Beginn der Funktion immer am links äußeren Rand des zu bearbeitenden/betrachtenden Bereich des Arrays:

```
function partition(var List: arr; left, right: integer): integer;
...
   pivot:= left;
   ...
```

Nun werden alle rechts vom Pivot-Element stehenden Elemente mit dem Pivot-Element verglichen:

```
...
element:= pivot+1;
while ( element <= right) do
begin
        temp:= List[element];
        if (List[element] <= List[pivot]) then
        ...
```

Ist ein Element kleiner gleich dem Pivot-Element, so wird dieses am Anfang des Subarrays eingefügt, indem es den ersten Platz einnimmt, während alle anderen Elemente bis zum einschließlich ursprünglichen Platz des Elementes um einen Platz nach rechts verschoben werden:

```
    ...
    begin
        for i:= element downto left+1 do
            List[i]:= List[i-1];
        List[left]:= temp;
        pivot:= pivot+1;
    end;
    ...
```

Die Funktion gibt, nachdem alle Elemente vor oder hinter dem Pivot platziert wurden, die aktuelle Position des Pivots wieder.

Selection-Funktion

```
function selection (var List: arr; left,right,k: integer): integer;
var s: integer;
begin
   s:= partition(List,left,right);
   if (s = k) then
       selection:= s
   else   if (s > k) then
              selection:= selection(List,left,s-1,k)
          else   if (s < k) then
                     selection:= selection(List,s+1,right,k);
end;
```

Je nachdem, ob die Stelle s des aktuellen Pivots kleiner, größer oder gleich der Größe k ist, ruft sich die Funktion mit neuer Bereichseinschränkung des Arrays auf oder es wird die Stelle s des aktuellen Pivots als Lösung zurückgegeben. Wenn s größer k ist, werden die links stehenden Elemente des Pivots als Bereich angegeben; wenn s kleiner k ist, dann werden die Elemente, die rechts vom Pivot stehen, als Bereich angegeben.

6 Interpolationssuche

6.1 Einführung

Die Interpolationssuche ist der binären Suche ähnlich. Es soll ein Element in einem sortierten Array gesucht werden.
Bei der binären Suche wird jeweils das mittlere Element im sortierten Array als Vergleichselement m genutzt. Ist das gesuchte Element e kleiner als das Vergleichselement m, so wird nur noch in der linken Hälfte des Arrays weiter gesucht, d.h. alle Elemente die links vom Vergleichselement stehen werden mit dem gleichen Verfahren durchsucht. Ist das gesuchte Element e größer als das Vergleichselement m, so wird in der rechten Hälfte weiter gesucht. Ist das gesuchte Element e gleich dem Vergleichselement m, so liegt die Lösung vor und das gesuchte Element liegt an der Stelle an der das Vergleichselement m vorliegt.

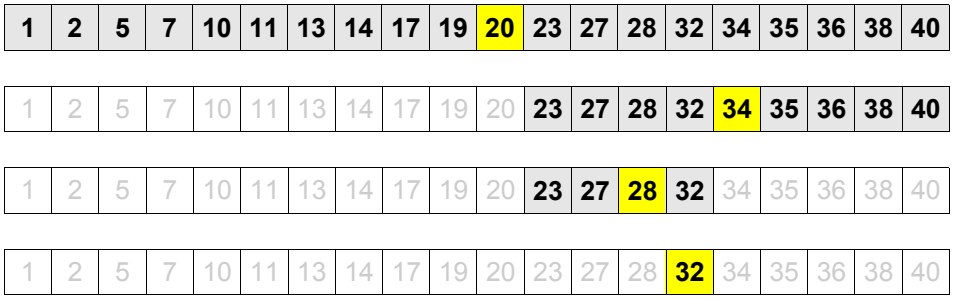


Abb. 12: Binäre Suche nach dem Element 32 in einem sortierten Array A

Im Gegensatz zu der binären Suche, bei der immer das mittlere Element als Vergleichselement genutzt wird, berücksichtigt die Interpolationssuche die Größe des gesuchten Elements um ein Vergleichselement aus der Liste zu wählen, dass dann mit dem gesuchten Element verglichen wird. Ähnlich handelt auch der Mensch, wenn er z.B. nach einem Namen im Telefonbuch sucht. Wenn der Name mit dem Buchstaben 'B' anfängt würde man das Telefonbuch zunächst weiter vorne aufschlagen, bei dem Buchstaben 'Y' weiter hinten, um dann präziser zu suchen.

6.2 Algorithmus

Um ein passendes Vergleichselement zu finden, wird mit einer geraden linearen Linie gearbeitet, die von dem am weitesten links stehenden Element und dem am weitesten rechts stehenden Element im Array A aufgespannt wird. Es handelt sich also um die Linie, die entsteht, wenn die Punkte (l,A[l]) und (r,A[r]) verbunden werden. Der Algorithmus nimmt an, dass die Werte des Arrays linear entlang der Linie Ansteigen.
Das gesuchte Element e wird verglichen mit dem Element, dessen Index errechnet wurde als x-Koordinate des Punktes auf der Geraden, wo die y-Koordinate gleich dem Wert des gesuchten Elements e ist.

Beispiel: Ein liegt ein 9-elementiges sortiertes Array A vor, dessen erstes Element A[l]=1 und letztes Element A[r]=9 ist. Das Array beginnt mit dem Index 1. Es wird eine lineare Linie zwischen den beiden äußersten Punkten aufgespannt. Das gesuchte Element e könnte sich im Intervall zwischen A[l] und A[r] befinden.

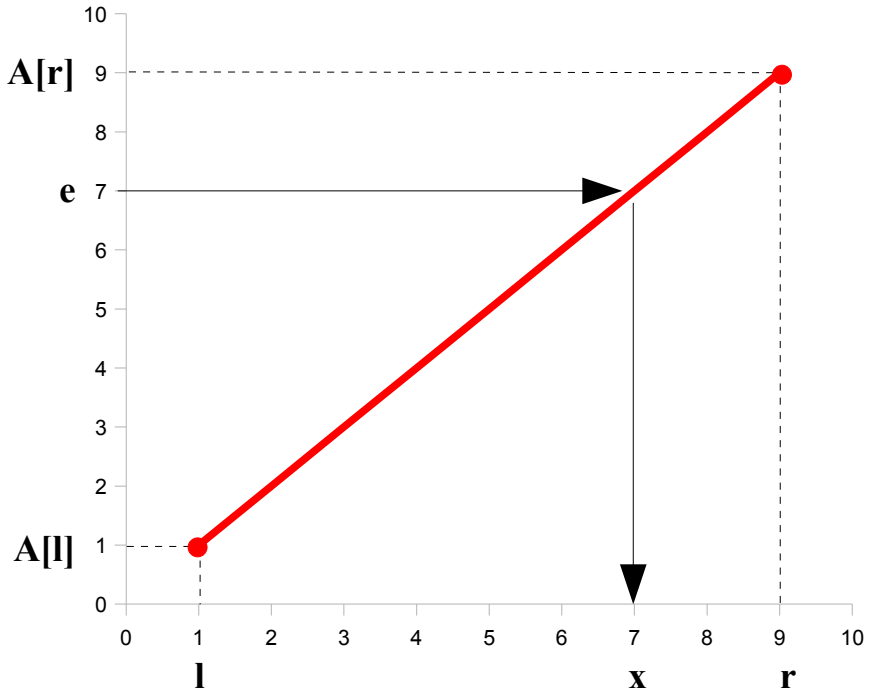


Abb. 13: Grafische Darstellung der Index-Berechnung x eines Vergleichselements bei der Interpolationssuche

Als Formel, lässt sich die Stelle x, an der das Vergleichselement im Array steht, mit folgender Formel berechnen:

$$x = l + \left\lfloor \frac{(e - A[l]) * (r - l)}{A[r] - A[l]} \right\rfloor \qquad \textit{Formel 1}$$

Es ist bekannt, dass in einem sortiertem Array A die Elemente von A[l] nach A[r] ansteigen (l = erstes Element, r = letztes Element), jedoch ist nichts über die Art dieses Anstiegs

bekannt. Bei der einfachsten Art (Werte steigen linear), kann der Index des gesuchten Elements e mit der Formel gefunden werden. Wenn die Werte nicht linear steigen, so lässt sich trotzdem eine grobe Annäherung erzielen. Je mehr die Werte linear steigen, desto schneller ist der Algorithmus.

Wenn das gesuchte Element e nicht im Intervall zwischen A[l] und A[r] liegt, muss die Formel gar nicht erst angewandt werden, da Element e in diesem Fall nicht im Array A vorkommen kann. Befindet sich Element e jedoch zwischen A[l] und A[r], so kann die Formel angewandt werden und man erhält den Index x, der den Index für das Vergleichselement im Array angibt. Das gesuchte Element e wird mit dem Element A[x] verglichen:

- Wenn e = A[x], dann liegt die Lösung vor
- Wenn e < A[x], wird der Algorithmus auf die Elemente zwischen l und x-1 angewandt, also auf die links stehenden Elemente vom Vergleichselement A[x]
- Wenn e > A[x], wird der Algorithmus auf die Elemente zwischen x+1 und r angewandt, also auf die rechts stehenden Elemente vom Vergleichselement A[x]

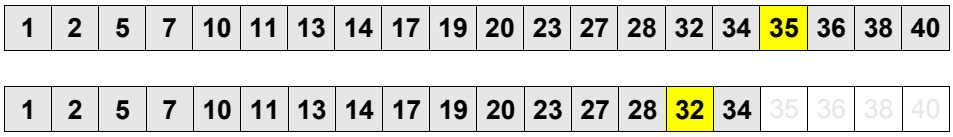


Abb. 14: Interpolationssuche nach dem Element 32 in einem sortierten Array A

Es wird also immer eine kleinere Instanz des Problems betrachtet, dessen Größe man aber erst zur Laufzeit bestimmen kann und die sich von mal zu mal ändert. Dies entspricht der variable-size-decrease Technik. Je tiefer man in die Rekursion geht, desto genauer wird die Annäherung an das gesuchte Element e.

Im average case werden weniger als $\log_2\log_2 n+1$ Vergleiche gemacht, wenn in einem Array mit n verschiedenen Werten gesucht wird. Die Funktion wächst so langsam, dass die Anzahl der Vergleiche eine sehr kleine Konstante ist für so gut wie alle möglichen Eingaben. Im worst case ist der Algorithmus jedoch nur linear, welches als schlechte Leistung betrachtet werden muss.
Es lässt sich folgern, dass die binäre Suche besser für kleinere Inputs und die Interpolationssuche besser für größere Inputs ist. Interpolationssuche ist ebenfalls besser geeignet für Anwendung, bei denen Vergleiche besonders aufwendig sind. ([3], S.240 ff.)

6.3 Implementierung

Die Interpolationssuche wird mit folgenden Eingabeparametern aufgerufen:

- List: Das Array, in welchem gesucht werden soll
- Element: das gesuchte Element
- left, right: geben die Indexgrenzen des betrachteten Bereichs des Arrays an
- x: Position des Vergleichselements

Die Funktion überprüft zuerst, ob sich das gesuchte Element im Intervall zwischen den Bereichsgrenzen des Arrays befinden kann. Sollte dies nicht der Fall sein, so wird der Wert false zurückgegeben.

Ansonsten ermittelt die Funktion einen Indexwert für das Vergleichselement mit der oben genannten Formel (Formel 1). Danach wird das Element, welches sich an Stelle x im Array befindet mit dem gesuchten Element verglichen. Je nachdem, ob das gesuchte Element kleiner, größer oder gleich dem Element an Position x ist, ruft sich die Funktion mit neuer Bereichseinschränkung des Arrays auf oder es wird der Wert true zurückgegeben, der bedeutet, dass die Position x die Lösung ist. Wenn das gesuchte Element kleiner ist, werden alle links stehenden Elemente vom Vergleichselement als Bereichseinschränkung genommen; ist das gesuchte Element größer, werden alle rechts stehenden Elemente vom Vergleichselement weiter verarbeitet.

```
function interpolation( List: arr; Element: integer;
                        left,right: integer; var x: integer): boolean;
begin
    if ((Element <= List[right]) and (Element >= List[left])) then
    begin
        x:= left + trunc(
                ((Element - List[left]) * (right - left))
                / (List[right] - List[left])
            );
        if (Element = List[x]) then
            interpolation:= true
        else if (Element < List[x]) then
                interpolation:= interpolation(List,Element,left,x-1,x)
             else if (Element > List[x]) then
                    interpolation:= interpolation(List,Element,x+1,right,x);
    end
    else interpolation:= false;
end;
```

7 Literaturverzeichnis

[1] Anany Levitin: *Introduction to the Design and Analysis of Algorithms;*
Addison-Wesley; 2006; ISBN 0-321-36413-9

[2] Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, Clifford Stein:
Algorithmen – Eine Einführung;
Oldenbourg; 2. Auflage (2007); ISBN 978-3-486-58262-8

[3] Robert Sedgewick: *Algorithmen*;
Pearson Studium; 2. Auflage (2002); ISBN 3-8273-7032-9

[4] Sara Baase, Allen van Gelder: *Computer Algorithms: Introduction to Design and Analysis*;
Pearson; 3. Auflage (1999); ISBN 0-201-61244-5

[5] Thomas Ottmann, Peter Widmayer: *Algorithmen und Datenstrukturen*;
Spektrum Akademischer Verlag; 4. Auflage (2002); ISBN 3-8274-1029-0

[6] Niklaus Wirth: *Algorithmen und Datenstrukturen. Pascal- Version*;
Teubner B.G. GmbH; 4. Auflage (1995); ISBN 3-519-22250-7

[7] Alexander Shen: *Algorithms and Programming: Problems and Solutions*;
Birkhäuser; Auflage: Reprint of the 1997 edition (2008); ISBN 978-0-8176-4760-5

Weitere Quellen:

http://www.wikipedia.com

http://www.csc.villanova.edu/~levitin/paper.html