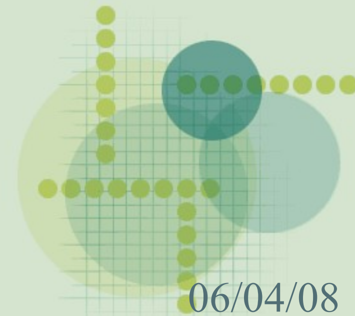
Informatik-Seminar zum Thema Algorithmen

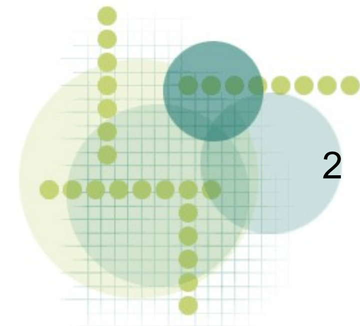
Decrease-and-Conquer

Theresa Brandt v. Fackh



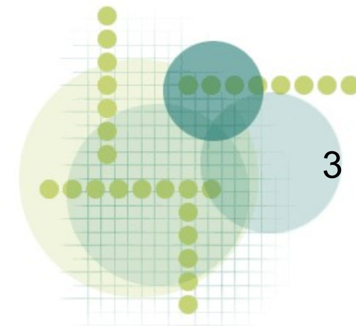
Inhalt

- Einführung
- InsertionSort
- Topologisches Sortieren
- Permutationsgenerierung
- Mediansuche
- Interpolationssuche



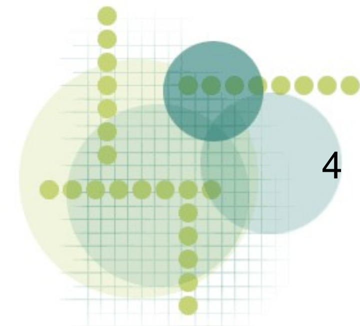
Überblick

- **Einführung**
  - **Decrease-and-Conquer Technik**
  - **Varianten**
  - **Vergleich mit Divide-and-Conquer**
- InsertionSort
- Topologisches Sortieren
- Permutationsgenerierung
- Mediansuche
- Interpolationssuche

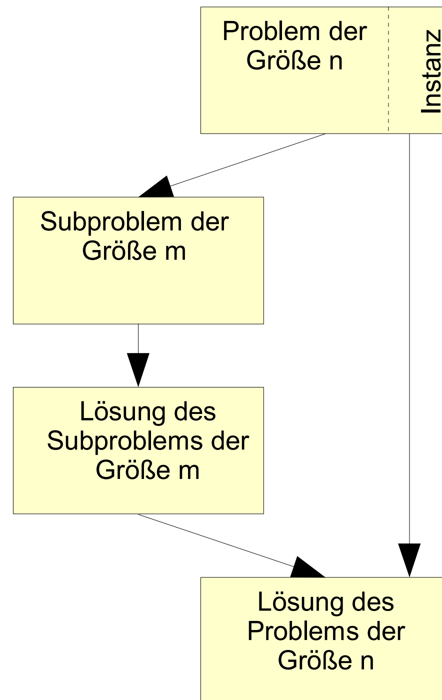


Decrease-and-Conquer Technik

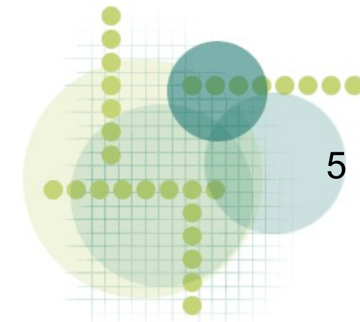
- Findet man in der Literatur unter verschiedenen Namen
  - Chip-and-Conquer (Sara Baase & Allen van Gelder)
  - Induction approach (Udi Manber)
  - Als Sonderform von Divide-and-Conquer
- “Reduziere-und-Erobere”
- Beziehung zwischen Lösung eines Problems und der Lösung einer kleineren Instanz dieses Problems
- Sowohl top-down als auch bottom-up Implementierung möglich



Decrease-and-Conquer Technik

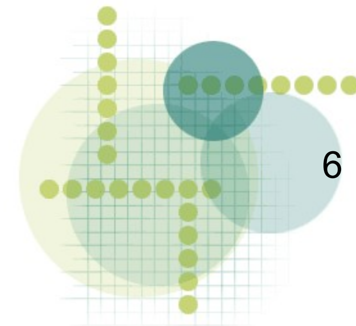


- Problem der Größe n zu Subproblem der Größe m reduzieren, wobei $m < n$
- Subproblem lösen
- Lösung des Subproblems wird zu der Lösung des Problems erweitert

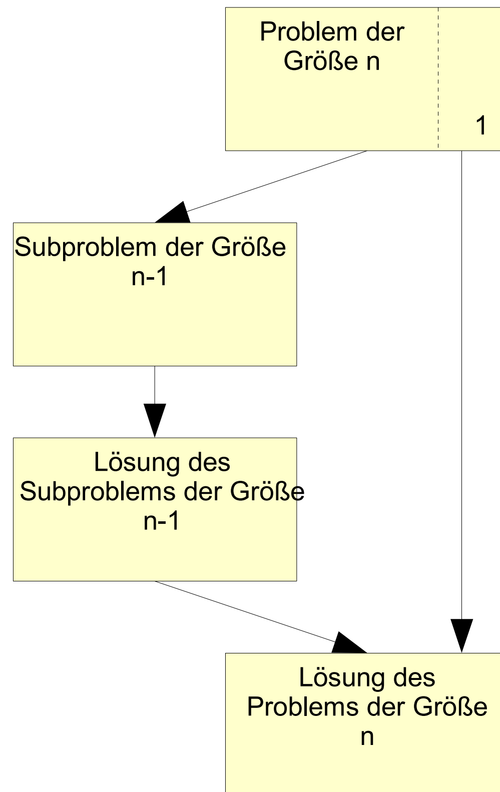


Varianten

- 3 unterschiedliche Varianten
  - Decrease-by-a-constant
  - Decrease-by-a-constant-factor
  - Variable-size-decrease
- Größe der Instanz, um die sich das Problem reduziert, ist in jeder Variante unterschiedlich



Variante: Decrease-by-a-constant



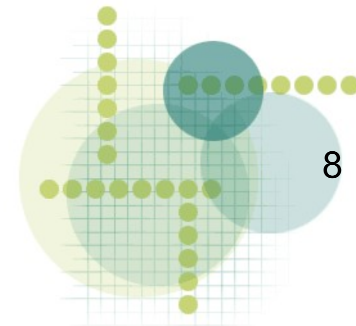
- Reduziert die Größe des Problems um dieselbe Konstante
- Oft ist die Konstante 1 ("decrease-by-one")
- z.B. InsertionSort, topologisches Sortieren, Permutationsgenerierung

Variante: Decrease-by-a-constant

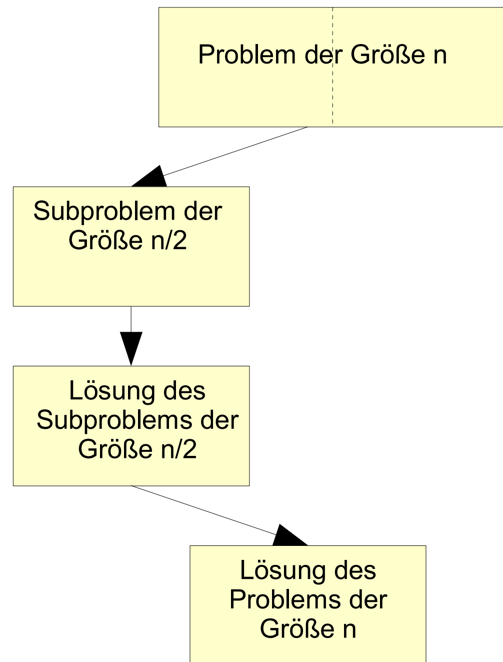
- Beispiel: Algorithmus für Exponential-Problem a^n für positive ganzzahlige Exponenten n
  - Problem: a^n
  - Konstante: 1
  - Beziehung zu Subproblem: $a^n = a^{n-1} * a^1$
  - Rekursive Definition (top-down):

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

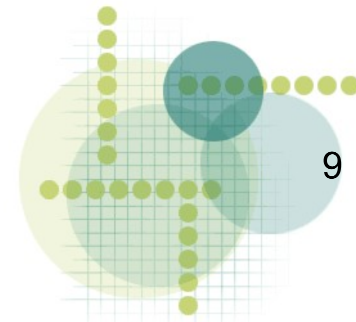
  - Bottom-up: a muss (n-1)-mal mit sich selbst multipliziert werden



Variante: Decrease-by-a-constant-factor



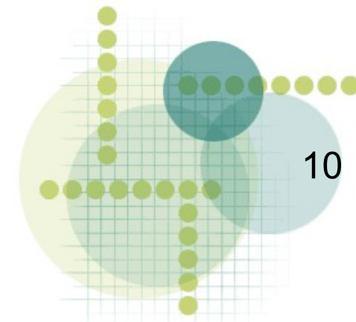
- Reduziert die Größe des Problems um denselben konstanten Faktor
- Konstanter Faktor hat meist den Wert 2 ("decrease-by-half")
- z.B. Binäre Suche, Fake-Coin-Problem, Josephus Problem
- selten



Variante: Decrease-by-a-constant-factor

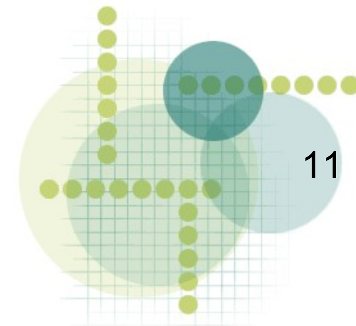
- Beispiel: Fake-Coin Problem (1/2)
  - Problem: Gefälschte Münze unter einer Menge von identischen Münzen identifizieren
  - Gewicht zweier Mengen mit einer Waage vergleichen

  - Annahme: gefälschte Münze ist leichter als eine Original-Münze



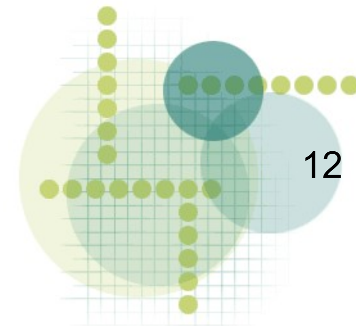
Variante: Decrease-by-a-constant-factor

- Beispiel: Fake-Coin Problem (2/2)
  - Menge von Münzen der Größe n in zwei gleichgroße Mengen $\lfloor n/2 \rfloor$ zerlegen
  - Wenn n ungerade ist, wird eine Münze beiseite gelegt
  - Mengen werden gewogen:
    - Mengen sind gleich schwer
      -> beiseite gelegte Münze ist gefälscht
    - Eine Menge ist leichter
      -> diese Menge näher betrachten mit gleichem Verfahren



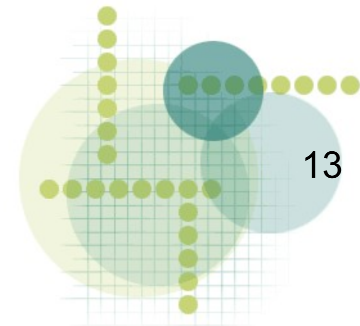
Variante: Variable-size decrease

- Reduktion der Größe des Problems unterscheidet sich in jedem Iterationsschritt
- z.B. Euklidischer Algorithmus (ggT), Mediansuche, Interpolationssuche



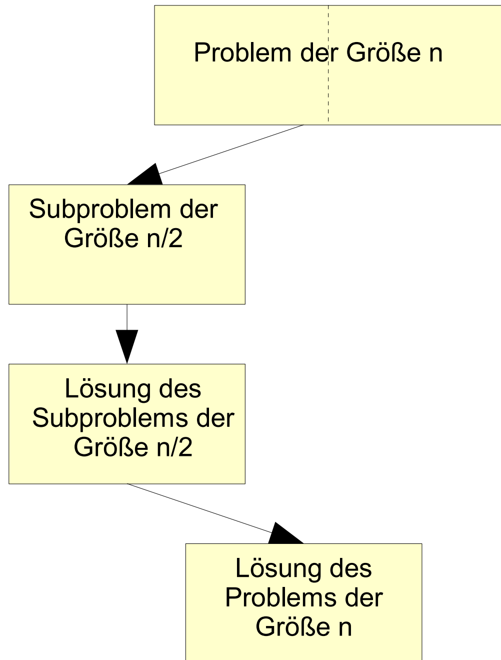
Vergleich mit Divide & Conquer

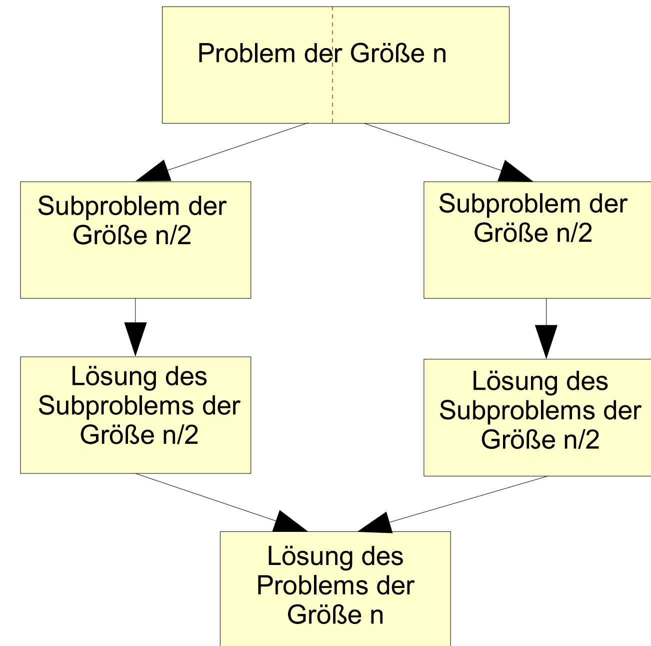
- Gemeinsamkeiten:
  - Problem lösen, indem man kleinere Instanzen des Problems löst
- Unterschiede:
  - Divide & Conquer: Problem in gleichartige Teilprobleme zerlegen, lösen und wieder zusammenführen
  - Decrease & Conquer: Problem in eine kleinere Instanz reduzieren, lösen und zu der Lösung des Problems höherer Instanz erweitern

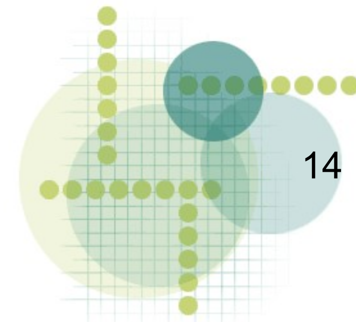


Vergleich mit Divide & Conquer

- decrease-by-half
- divide-and-conquer

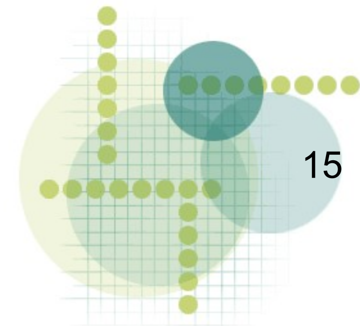






Überblick

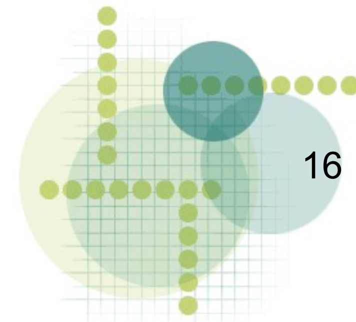
- Einführung
- **InsertionSort**
  - **Einführendes Beispiel**
  - **Algorithmus**
  - **Analyse**
- Topologisches Sortieren
- Permutationsgenerierung
- Mediansuche
- Interpolationssuche



Einführendes Beispiel

- “sortieren durch direktes Einfügen”
- Beispiel: Sortieren von Spielkarten

  - Karte neu aufnehmen, mit den Karten auf der Hand vergleichen und an entsprechender Stelle einfügen
  - Aufnahmestapel: unsortiert
  - Karten auf der Hand: sortiert



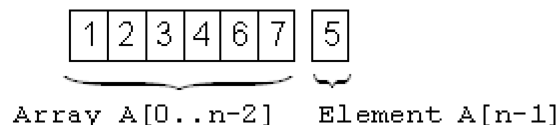
Algorithmus

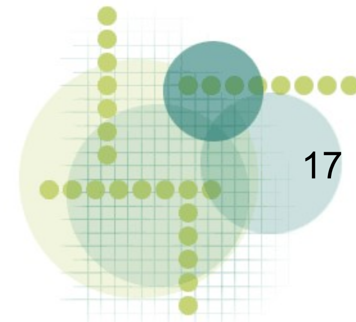
- Array A mit n sortierbaren Elementen

- Decrease-by-one: Subproblem umfasst ein zu sortierendes Array der Größe n-1

- Subproblem lösen, so dass A[0] ≤ ... ≤ A[n-2]

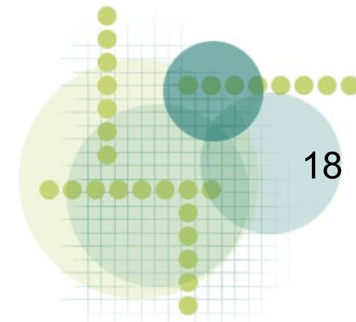
- Element A[n-1] in das bereits sortierte Array der Größe n-1 einsortieren





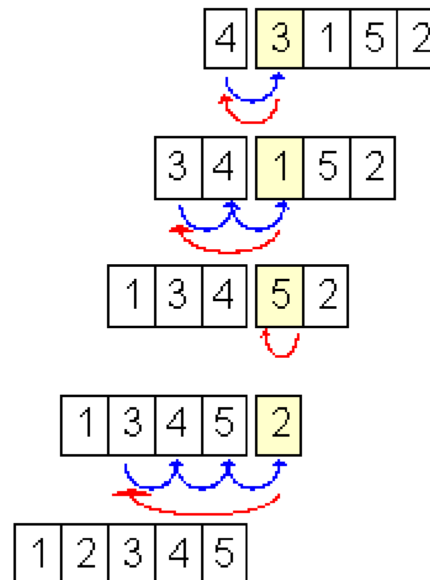
Algorithmus

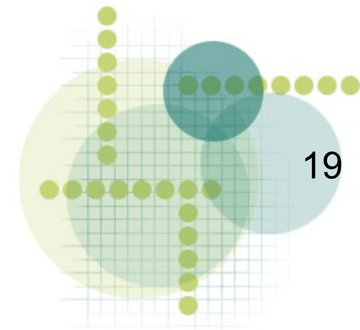
- Drei Alternativen für das Einfügen:
  - Subarray von links nach rechts nach richtiger Einfüge-Stelle durchsuchen ('straight' insertion sort)
  - Subarray von rechts nach links nach richtiger Einfüge-Stelle durchsuchen ('straight' insertion sort)
  - Binäre Suche anwenden, um die richtige Einfüge-Stelle zu finden (binary insertion sort)
- Bei 'straight' insertion sort wird Alternative 2 vorgezogen, da schneller bei sortierten und fast-sortierten Arrays



Algorithmus

- Beispiel: Array A = {4, 3, 1, 5, 2} soll sortiert werden



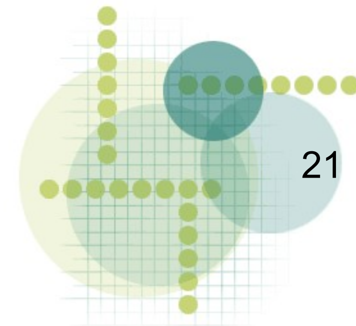


Analyse

- Basis-Operation: Elementenvergleich
- Worst-Case
  - Einzufügendes Element muss immer ganz nach vorne und muss somit mit jedem Element verglichen werden
  - Absteigend sortiertes Array
- Best-Case
  - Elementenvergleich wird jeweils nur einmal angewandt
  - Aufsteigend sortiertes Array

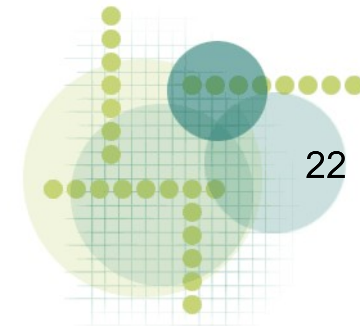
Analyse

- Average-Case:
  - Basiert auf Erforschung der Anzahl von Element-Paaren, die in falscher Reihenfolge stehen
  - Bei nicht-sortierten Arrays halb so viele Vergleiche wie bei absteigend sortiertem Array
- Fazit: InsertionSort ist bester Algorithmus unter den elementaren Sortieralgorithmen
  - Doppelt-so-schnell im Durchschnitt
  - hohe Effizienz bei fast sortierten Array



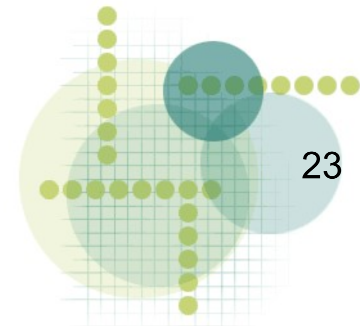
Überblick

- Einführung
- InsertionSort
- **Topologisches Sortieren**
  - **Einführendes Beispiel**
  - **Allgemeines**
  - **Algorithmus**
- Permutationsgenerierung
- Mediansuche
- Interpolationssuche



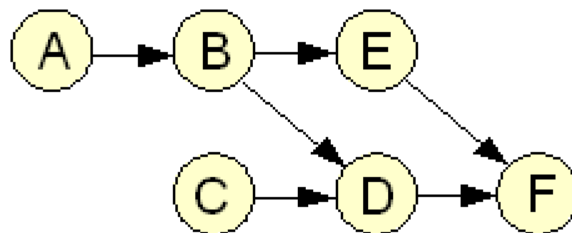
Einführendes Beispiel

- Student muss sich Studienplan selbst zusammenstellen
- Veranstaltungen sind Vorraussetzung für andere Veranstaltungen
- Studienplan so zusammenstellen, dass keine Veranstaltung eine später aufgeführte Veranstaltung vorraussetzt
- Andere Beispiele: Kochen, Projekte, Modellierung einer Fertigungsstrasse



Einführendes Beispiel

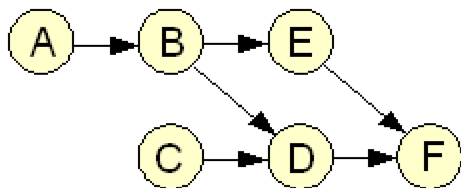
Veranstaltung	Vorausetzung
A	–
B	A
C	–
D	B,C
E	B
F	D,E

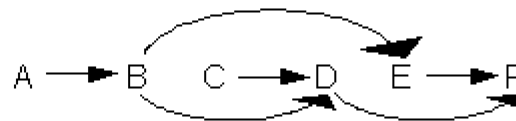


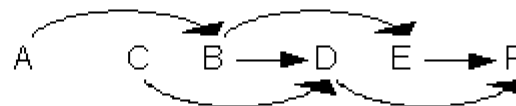
- Aus links stehender Information folgt:
  - 1. Semester: A und C
  - 2. Semester: B
  - 3. Semester: E und D
  - 4. Semester: F

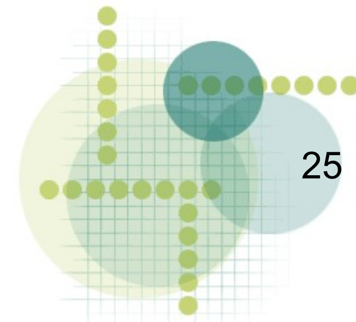
Allgemeines

- Wichtiges Problem für gerichtete Graphen
- Knoten in topologischer Reihenfolge bringen heißt, dass Knoten, von denen Kanten ausgehen, vor den Knoten stehen, zu denen diese Kanten führen
- Abstrakt: teilweise Ordnung (Transitivität, Asymmetrie, Irreflexivität) in eine lineare Ordnung einbetten
- Beispiel:



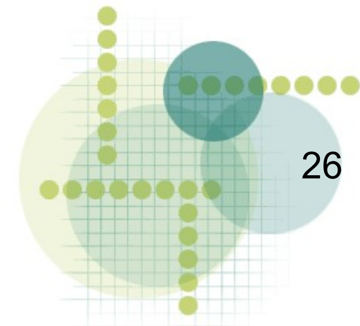






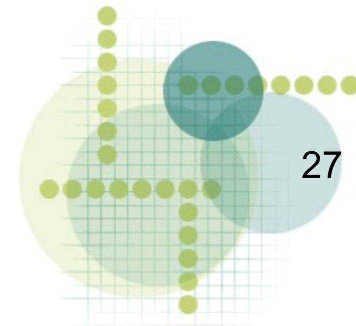
Allgemeines

- Mehrere Lösungen möglich
- Graph muss gerichteter nicht-zyklischer Graph sein, sonst keine Lösung möglich
- Abstrakte Definition berüchsichtigt dies:
  Transitivität und Asymmetrie der teilweisen Ordnung garantieren, dass der Graph keine Schleifen enthält
- Bei größeren Graphen ist ein Algorithmus unausweichlich



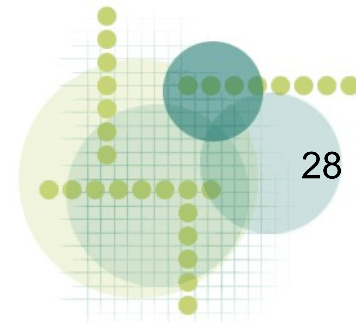
Algorithmus

- Algorithmus, der zugleich...
  - ...prüft, ob es sich um einen gerichteten azyklischen Graphen handelt
  - ...die Knoten in topologische Reihenfolge bringt
- Decrease-by-one
- Definition: Quelle
  - Knoten, der keine eingehenden Kanten hat



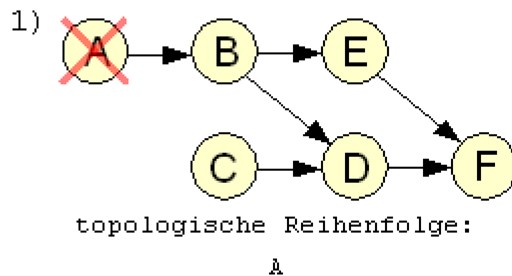
Algorithmus

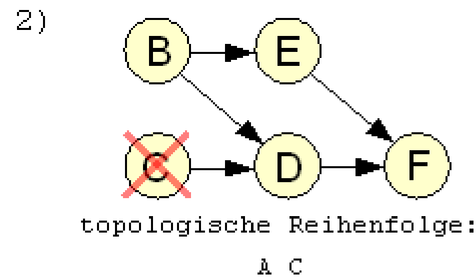
- Wiederholt Quelle im jeweils restlichen gerichteten Graphen suchen
- Quelle mit allen ausgehenden Kanten löschen
- Bei mehreren Quellen wird in beliebiger Reihenfolge gelöscht
- Gibt es keine Quelle, so gibt es auch keine Lösung
- Reihenfolge, in der die Knoten gelöscht werden, geben die topologische Reihenfolge an

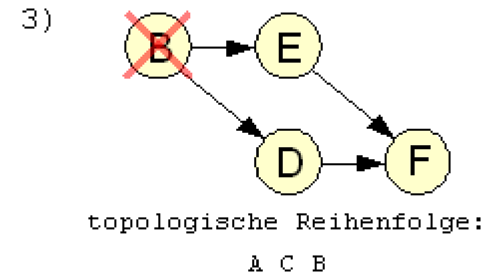


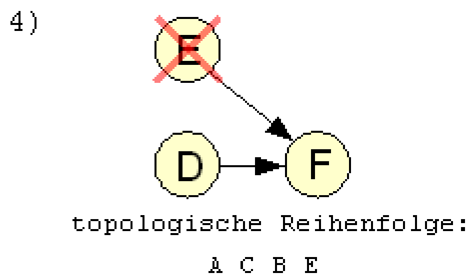
Algorithmus

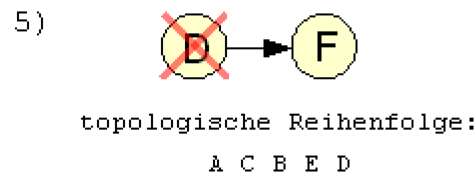
- Beispiel:



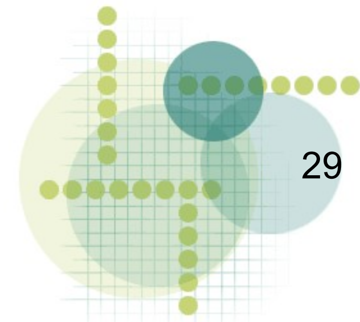






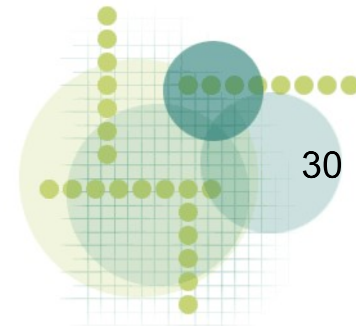


- Topologische Reihenfolge: A C B E D F



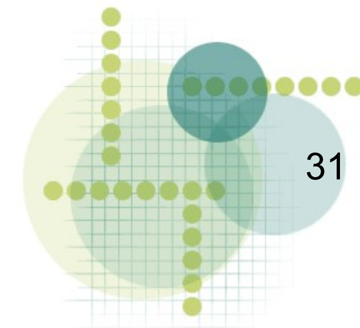
Algorithmus

- Den Algorithmus “rückwärts” anzuwenden, führt zu einer umgekehrten topologischen Reihenfolge
- Wiederholt Endknoten (mit keinen ausgehenden Kanten) mit ihren eingehenden Kanten löschen
- Reihenfolge des Entfernens ergibt die topologische Reihenfolge
- Gleiches Ergebnis bekommt man, wenn man alle Kanten im Graphen umkehrt und den Algorithmus “vorwärts” anwendet



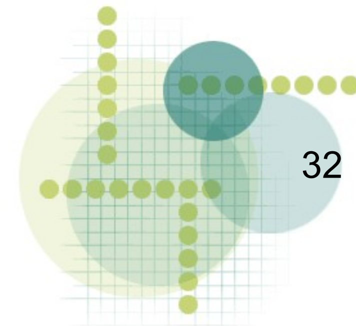
Überblick

- Einführung
- InsertionSort
- Topologisches Sortieren
- **Permutationsgenerierung**
  - **Einführung**
  - **Minimal-Change-Algorithmus**
  - **Johnson-Trotter**
- Mediansuche
- Interpolationssuche



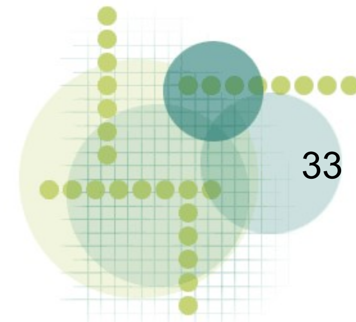
Einführung

- Alle möglichen Anordnungen von einer Menge aus unterschiedlichen Objekten erzeugen
- Nicht: Anzahl von Anordnungen berechnen
- z.B. für Traveling-Salesman-Problem brauchbar:
  - Zu gegebener Menge von n Orten die günstigste Tour ermitteln, die alle Orte verbindet
  - Alle möglichen Touren mit Permutationsgenerierung ermitteln
  - Kosten für jede Tour berechnen und somit die günstigste Tour herauskriegen



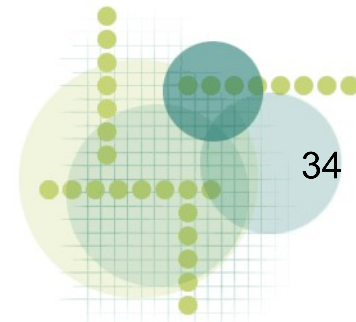
Minimal-Change Algorithmus

- Decrease-by-one
- Problem: n! Permutationen generieren
- Subproblem: (n-1)! Permutationen generieren
- Lösung des Problems:
  n-te Element in alle n möglichen Positionen unter den Elementen von jeder bereits generierten Permutation (gelöstes Subproblem) einfügen
- Alle Permutationen wären eindeutig und deren Gesamtzahl wäre n*(n-1)! = n!



Minimal-Change Algorithmus

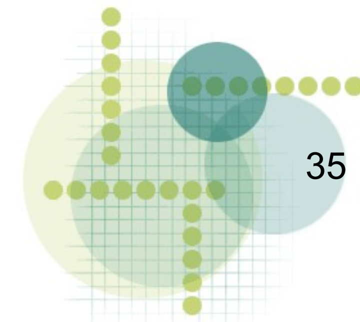
- Für die erste Permutation der Sublösung ist die Richtung von rechts nach links festgelegt
- Solange noch unbearbeitete Permutationen in der Sublösung sind:
  - n-te Element in alle n möglichen Positionen in der Permutation in der angegebenen Richtung einmal einfügen
  - Die daraus neu entstandene Permutation in eine Liste schreiben
  - Gehe zur nächsten Permutation der Sublösung und verdrehe die Richtung



Minimal-Change Algorithmus

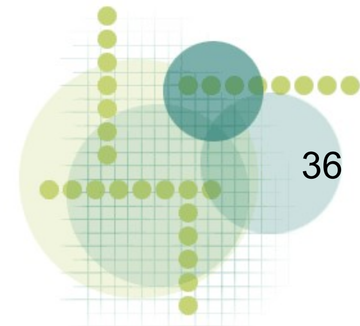
- Beispiel:

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		



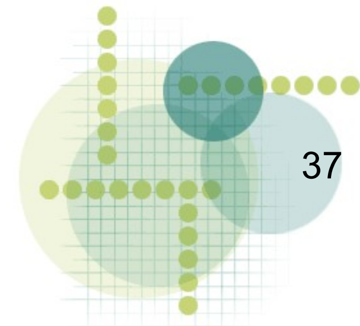
Minimal-Change Algorithmus

- Reihenfolge des Einfügens erfüllt die “Minimal-Change” Anforderung
- “Minimal-Change”-Anforderung:
  - jede Permutation kann von seinem unmittelbaren Vorgänger erlangt werden
  - es müssen nur 2 benachbahrte Elemente vertauscht werden
- Anforderung ist für die Geschwindigkeit des Algorithmus' von Vorteil



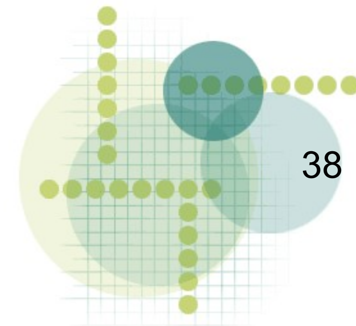
Johnson-Trotter Algorithmus

- auch: Steinhaus-Johnson-Trotter Algorithmus
- Generiert Permutationen ohne kleinere Subprobleme zu lösen
- Jedes Element in einer Permutation erhält einen Richtungspfeil
- Element k wird als mobil bezeichnet, wenn sein Richtungspfeil in die Richtung zeigt, in der eine kleinere Größe benachbart ist
- Vorraussetzung: Elemente müssen ordinal sein



Johnson-Trotter Algorithmus

- Alle Elemente einer beliebigen Permutation erhalten einen Richtungspfeil, der nach links zeigt
- Solange die Permutation ein mobiles Element enthält:
  - Finde das größte mobile Element in der Permutation; wenn keins vorhanden ist der Algorithmus beendet
  - Tausche dieses Element mit dem benachbarten Element, auf den das gefundene Element zeigt
  - Drehe alle Richtungspfeile aller Elemente, die größer als das gefundene Element sind
  - Füge die somit neu entstandene Permutation der Liste hinzu

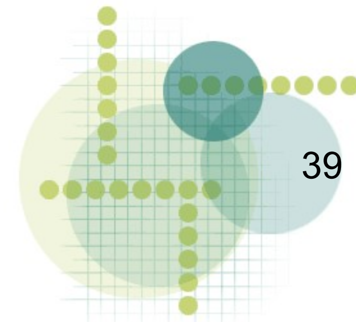


Johnson-Trotter Algorithmus

- Beispiel:

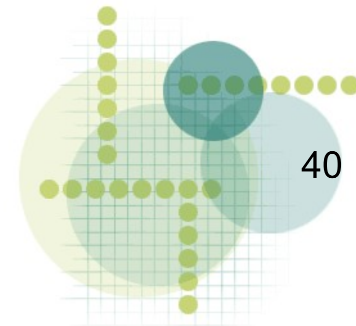
← ← ←	← ← ←	← ← ←	→ ← ←	← → ←	← ← →
1 2 **3**	1 **3** 2	3 1 **2**	**3** 2 1	2 **3** 1	2 1 3

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



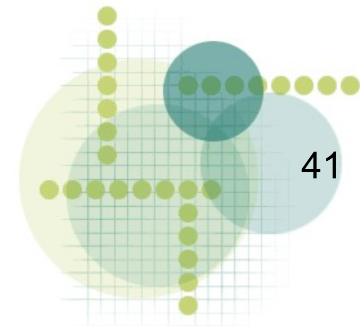
Johnson-Trotter Algorithmus

- Reihenfolge der Permutationen sind nicht in lexikographischer Ordnung
- Kann man mit einem weiteren Algorithmus beheben
- Algorithmus kann proportional zu der Anzahl der Permutationen laufen
- Langsam bei größeren Mengen
- Schuld liegt nicht beim Algorithmus, sondern am Problem, dass einfach zu viele Permutationen generiert werden



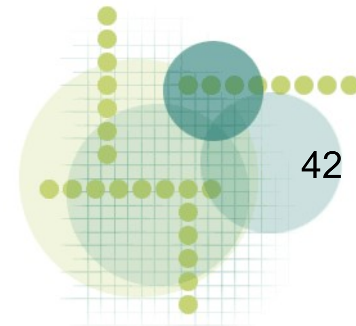
Überblick

- Einführung
- InsertionSort
- Topologisches Sortieren
- Permutationsgenerierung
- **Mediansuche**
  - **Einführung**
  - **Algorithmus**
  - **Implementierung**
- Interpolationssuche



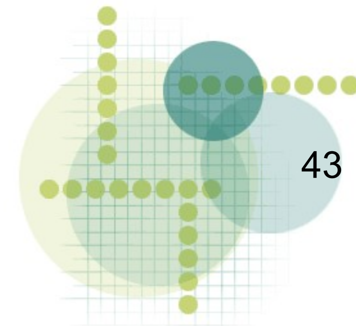
Einführung

- Median
  - Wichtiges Lagemaß in der mathematischen Statistik
  - Bezeichnet in einer Zahlenreihe von quantitativen Elementen genau den Wert, der in der Mitte der Zahlenreihe liegt
  - Untere Median: $\frac{\text{Anzahl der Elemente}}{2}$
  - Obere Median: $\frac{\text{Anzahl der Elemente}}{2} + 1$
- Mediansuche ist spezieller Fall des Auswahlproblems



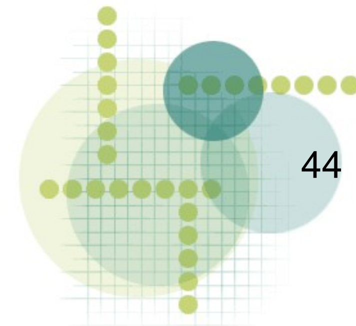
Einführung

- Auswahlproblem
  - Das k-kleinste Element in einer Liste von vergleichbaren Elementen finden
  - Median in einer Liste der Größe n finden, heißt:
    - $\lceil n/2 \rceil$-kleinstes Element in der Liste suchen



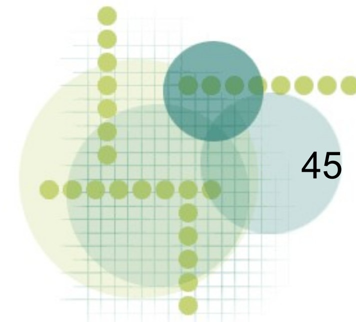
Algorithmus

- Naiver Ansatz
  - Liste sortieren und dann das k-ste Element ausgeben
  - Laufzeit, je nachdem welcher Sortieralgorithmus gewählt wurde
  - Sortierung ist nicht notwendig!



Algorithmus

- Array in 2 Subarrays teilen:
  - Pivot p befindet sich an Position s im Array, an der auch die Aufteilung des Arrays erfolgt: $a_{i_1} \ldots a_{i_{s-1}} \quad p \quad a_{i_{s+1}} \ldots a_{i_n}$
  - Links von Pivot p: Elemente, die ≤ p
    Rechts von Pivot p: Elemente, die > p
- s = k: Lösung für das Auswahlproblem gefunden
- s > k: k-kleinstes Element befindet sich als k-kleinstes Element im linken Subarray
- s < k: k-kleinstes Element befindet sich als (k-s)-kleinstes Element im rechten Subarray



Algorithmus

- Beispiel:

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

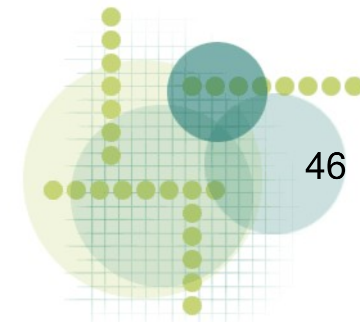
Das Pivot-Element befindet sich nach der Sortierung an Position s=2.
Da s=2 < k=5, wird der Bereich rechts vom Pivot-Element bearbeitet.

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.

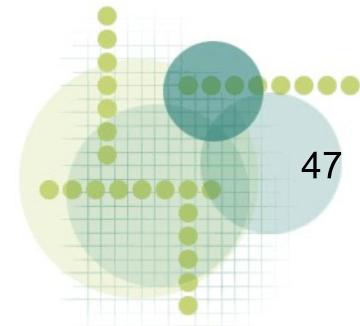
1	2	**7**	3	4	8	15	19	13
1	2	4	3	**7**	8	15	19	13

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.



Algorithmus

- Variable-size decrease
- Average Case: linear
- Liefert auch Antwort auf die Frage nach den k-kleinsten und (n-k)-größten Elemente im Array
- Ähnelt dem Sortieralgorithmus Quicksort, jedoch arbeitet dieser mit der Divide-and-Conquer Technik

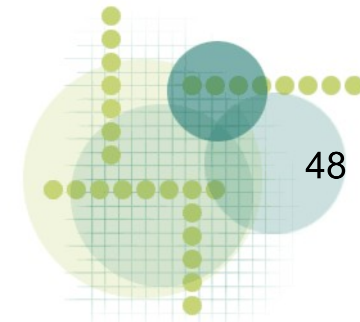


Implementierung


```
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;
```

Partition-Funktion

- Eingabeparameter:
  Array, Grenzen
- Wahl des Pivots p:
  links äußerer Rand
- Aufgabe:
  rechts von p stehende Elemente mit p vergleichen und entweder stehen lassen oder am Anfang des Array einfügen
- Rückgabe:
  Aktuelle Position des Pivots

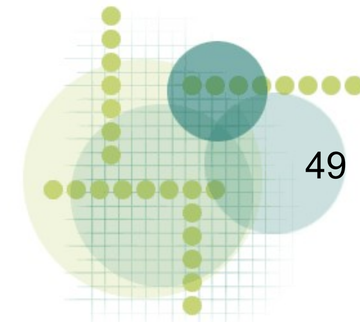


Implementierung


```
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;
```

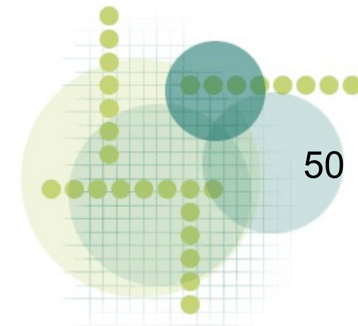
Selection-Funktion

- Eingabeparameter:
  Array, Grenzen, k
- Aufgabe:
  Je nachdem, ob Stelle s des aktuellen Pivots <, > oder = k ist, ruft sich die Funktion mit neuer Bereichseinschränkung des Arrays auf oder die Lösung wurde gefunden
- Rückgabe:
  Position des k-kleinsten Elementes



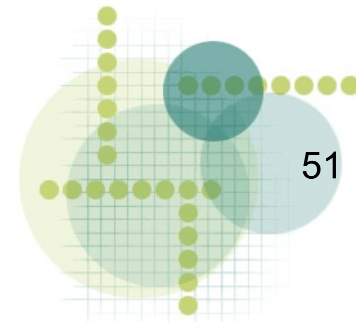
Überblick

- Einführung
- InsertionSort
- Topologisches Sortieren
- Permutationsgenerierung
- Mediansuche
- **Interpolationssuche**
  - **Einführung**
  - **Algorithmus**
  - **Implementierung**



Einführung

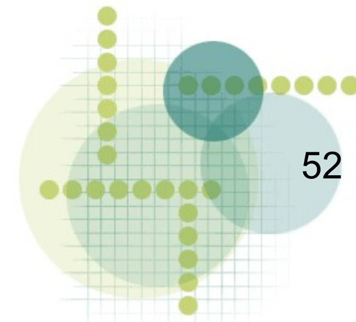
- Binäre Suche:
  - Mittleres Element im sortierten Array wird als Vergleichselement genutzt
  - Ist gesuchtes Element < Vergleichselement, so wird links vom Vergleichselement weitergesucht
  - Ist gesuchtes Element > Vergleichselement, so wird rechts vom Vergleichselement weitergesucht
  - Lösung wenn gesuchtes Element = Vergleichselement
- Interpolationssuche berücksichtigt bei Auswahl des Vergleichselements die Größe des gesuchten Elementes



Algorithmus

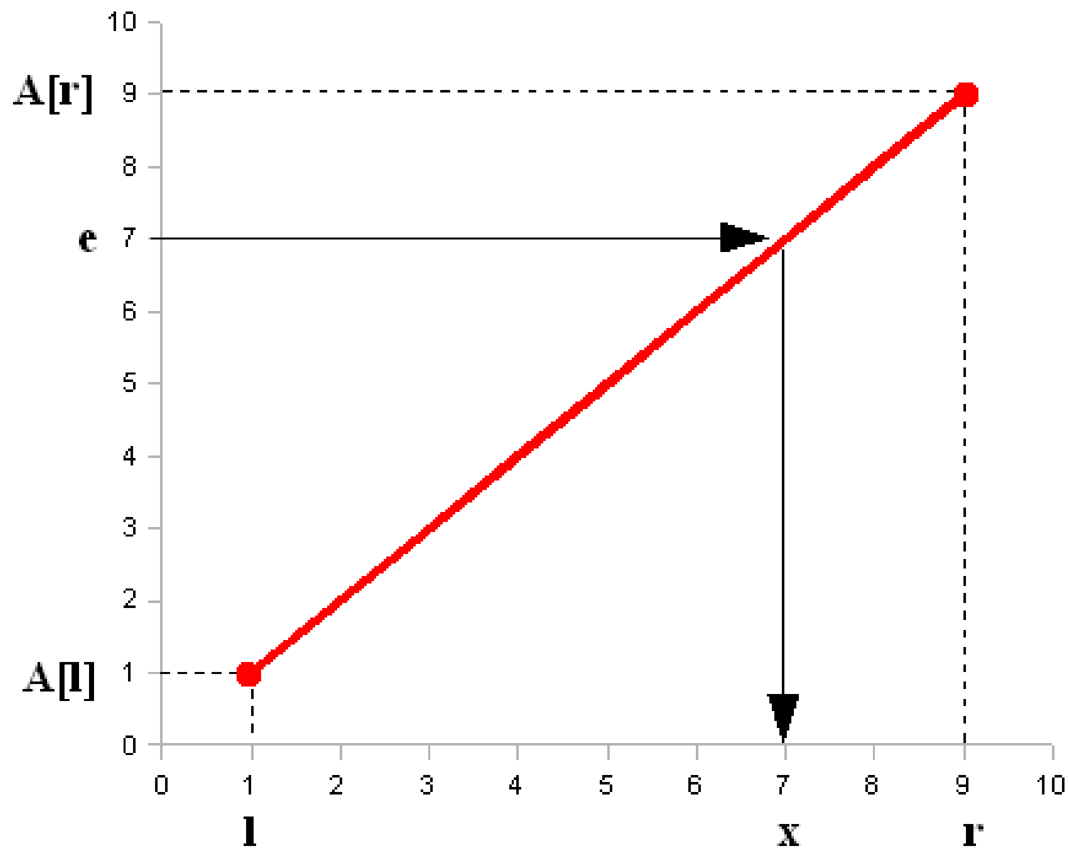
- Auswahl des Vergleichselementes:
  - mit dem am weitesten links stehenden Element und dem am weitesten rechts stehenden Element im Array A eine Gerade aufspannen
  - Algorithmus nimmt an, dass die restlichen Werte des Arrays linear entlang der Linie ansteigen
  - Vergleichselement befindet sich an der Position x, dessen Wert als x-Koordinate des Punktes auf der Geraden errechnet wurde, wo die y-Koordinate gleich dem Wert des gesuchten Elementes e ist
  - Als Formel:

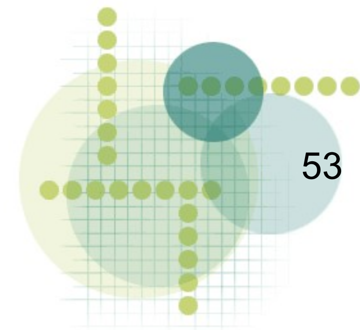
$$x = l + \left\lfloor \frac{(e - A[l]) * (r - l)}{A[r] - A[l]} \right\rfloor$$



Algorithmus

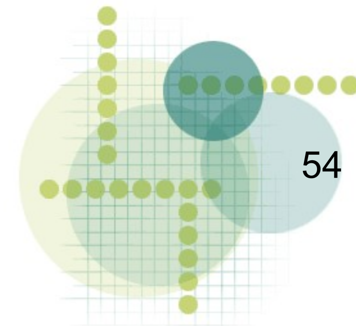
- Grafik:





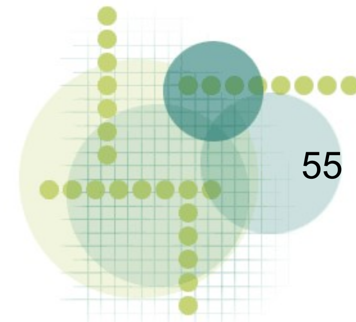
Algorithmus

- Bekannt: Elemente in einem sortierten Array steigen von A[l] nach A[r]
- Nicht bekannt: Art des Anstiegs
- Bei einfachster Art (linearer Anstieg) kann Position des gesuchten Elementes e mit der Formel gefunden werden
- Wenn der Anstieg nicht linear ist, so lässt sich trotzdem eine grobe Annäherung erzielen



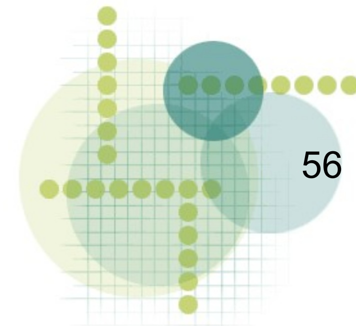
Algorithmus

- Wenn Element e nicht im Intervall zwischen A[l] und A[r], kann Element e nicht im Array vorkommen
- Ansonsten wird die Formel angewandt, um Position des Vergleichselements zu ermitteln
- Gesuchtes Element e wird mit Vergleichselement A[x] verglichen:
  - e = A[x]: Lösung liegt vor
  - e < A[x]: Algorithmus auf links von A[x] stehende Elemente (A[l] .. A[x-1]) anwenden
  - e > A[x]: Algorithmus auf rechts von A[x] stehende Elemente (A[n+1] .. A[r]) anwenden



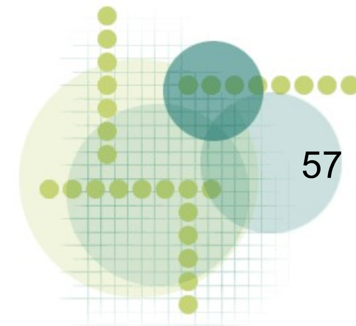
Algorithmus

- Immer kleinere Instanz des Problems wird betrachtet, dessen Größe jedoch erst zur Laufzeit bestimmt werden kann
  → Variable-size decrease
- Average Case:
  - weniger als $\log_2 \log_2 n+1$ Vergleiche bei Array mit n verschiedenen Werten
  - Wächst so langsam, dass Anzahl der Vergleiche eine sehr kleine Konstante ergibt für alle Eingaben
- Worst Case: linear



Algorithmus

- Bei kleinen Inputs: binäre Suche
- Bei großen Inputs: Interpolationssuche
- Bei Vergleichen, die besonders aufwendig sind: Interpolationssuche



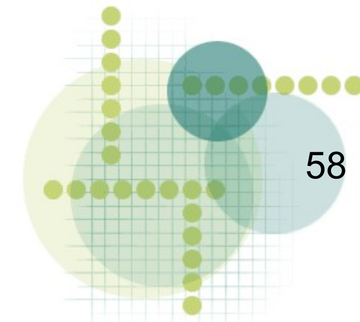
Implementierung


```
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;
```

Interpolation Funktion

- Eingabeparameter:
  Array, gesuchtes Element, Grenzen
- Aufgabe:
  ermittelt Position des Vergleichselements; Je nachdem, ob das gesuchte Element <, > oder = dem Vergleichselement, ruft die Funktion sich selbst mit neuem Bereich auf oder bricht ab

- Rückgabe:
  true und Position, wenn das gesuchte Element gefunden wurde; ansonsten false



The End...

Danke für Ihre Aufmerksamkeit!