6. Laufzeitanalyse, Sortieren u.a

funktionale Sprachen:
Reduktionen
Ein-/Ausgabe-intensive Algorithmen: **Seitenzugriffe**

Idee:

- Aufwand eines Algorithmus ist abhängig von „**Problemgröße**“ n ... und natürlich vom Algorithmus
- Aufwand ist proportional zur Anzahl **Elementaroperationen**, die zur Lösung eines Problems der Größe n ausgeführt werden müssen
- Elementaroperationen bilden das **Kostenmaß**:
  - in imperativen Sprachen: elementare Operationen **Zuweisung, Vergleich**
- Gesucht: Funktion $g(n)$, die **Anzahl der Elementaroperationen** abhängig von Problemgröße liefert
- Einfacher (und übersichtlicher): **Wachstum** von $g(n)$ mit wachsendem n untersuchen
- Technik : g mit bekannter Funktion f vergleichen.
  Ist g irgendwann (ab einem bestimmten n) immer größer / kleiner als f ?

O-Notation

Nach P. Bachmann,1894, und Edmund Landau

Obere Schranke

$O(f)$ bezeichnet eine ***Klasse*** von Funktionen $f: \mathbf{N} \to \mathbf{N}$

Def.: $O(f(n)) = \{ g(n) \mid \exists n_0, c > 0 : \forall n \geq n_0 : g(n) \leq c f(n) \}$

- statt $g \in O(f)$ schreibt man oft salopp : $g = O(f)$
- "Gleichungen" mit $O(...)$ gelten daher nur "von links nach rechts"!

Keine Gleichung!
.. und erst recht keine Zuweisung :-)

„Rechnen“ mit Mengen von Funktionen:

$cO(f) = \{ cg: g \in O(f) \}$

$O(f) + O(g) = \{ h + k : f \in O(h)$ und $k \in O(g) \}$

und analog für andere Operationen

Einige "Rechenregeln", z. B.:

$f = O(f)$

$c\, O(f) = O(f)$ für eine Konstante c

$O(O(f)) = O(f)$

$O(f) + O(g) = O(f + g)$

Untere und exakte Schranke

Für *untere Schranken* verwendet man die Ω *-Notation* (Omega):

Def.: $\Omega(f(n)) = \{ g(n) \mid \exists\, n_0, c > 0 : \forall\, n \geq n_0 : c\, f(n) \leq g(n) \}$

Als Notation für eine *exakte Schranke* definiert man die Θ *-Notation* *(Theta)* so:

Def.: $\Theta(f) = \{ g(n) \mid \exists\, n_0, c_1, c_2 > 0 : \forall\, n \geq n_0 : 0 <= c_1 f(n) \leq g(n) \leq c_2 f(n)$

Sprechweise:

$g = O(f)$: g wächst *höchstens* so schnell wie f

$g = \Omega(f)$: g wächst *mindestens* so schnell wie f

$g = \Theta(f)$: g wächst *genauso* schnell wie f

$g \in \Theta(f)$ äquivalent zu $g \in O(f)$ und $g \in \Omega(f)$

Wichtig noch die Klasse der *polynomiellen Funktionen* :

$$POLY(n) = \cup_{p>0}\ O(n^p)$$

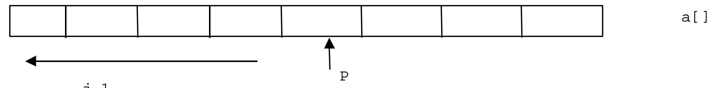
falls $f \in POLY(n)$, dann spricht man von *polynomiellem Aufwand*

Schlechtester, Bester, mittlerer Fall

- Laufzeit ist (meist) abhängig von der Reihenfolge der Eingabe z.B. Sortieren
- Offenbar gilt:
- Wenn $g = O(f)$ im schlechtesten Fall, dann auch im besten und mittleren Fall
- Wenn $g = \Omega(f)$ im besten Fall dann auch im mittleren und schlechtesten Fall
- Meist meint man mit : „*Die Laufzeit des Algorithmus ist asymptotisch* $O(f)$“ die Laufzeit im schlechtesten Fall.
- Die Bestimmung der mittleren Laufzeit ist praktisch immer viel schwieriger, als die Untersuchung des schlechtesten (besten) Falls.
- Was ist der „mittlere“ Fall?
  Annahme meist: alle Permutationen der Eingabe sind gleich wahrscheinlich,
  Darüber mitteln!

Es gibt realistischere Techniken

Schon bekannt: Sortieren durch Einfügen




```
Prüfe für  0 < j <=p ob  a[p] < a[j-1]
wenn ja: tauschen

Mache das für alle 1 < p < a.length
```

Wie effizient ist der Algorithmus?

```
public   class Sort
{     /* Simple insertion sort.
      * @param a an array of Comparable items.
      */
    public static void insertionSort( Comparable [ ] a )
       { int p = 1;
         while (p < a.length;)
         {Comparable tmp = a[p];
          int j = p;
          while (j > 0 && tmp.lessThanEq( a[j - 1]))
             {a[ j ] = a[ j - 1];    j--}
          a[ j ] = tmp;
          p++;
         }
       }
}
```


Laufzeit „Sortieren durch Einfügen“

Schlechtester Fall: $O(n^2)$ ..... warum?

Bester Fall: Feld sortiert, dann wird die Schleife:

`while (j > 0 && tmp.lessThanEq( a[j - 1]))`

nie betreten.

Also 4 Zuweisungen und ein Test für jedes der n Elemente: Laufzeit $O(n)$

Mittlerer Fall

Betrachte Inversionen: Paare a[i] > a[j], i< j

Beispiel: (6 5 1 3 7 8)

(6,5), (6,1), (6,3), (5,1), (5,3)

Algorithmus entfernt in jeder Schleife (`a[ j ] = a[ j - 1];`) eine Inversion.

--> Mittlere Anzahl von Schleifendurchläufen = mittlere Anzahl Inversionen

Betrachte Feld a mit n Elementen und `reverse`(a)

Für jedes Paar (i,j) 0<=i,j< n gilt: entweder (`a[i],a[j]`) oder (`a[j],a[i]`) ist Inversion. Es gibt $\binom{n}{2}$ = n*(n-1) / 2 Paare, im Mittel also n*(n-1) / 4 Inversionen

Sortieren durch Einfügen hat im Mittel quadratische Laufzeit!

Quicksort: Prinzip

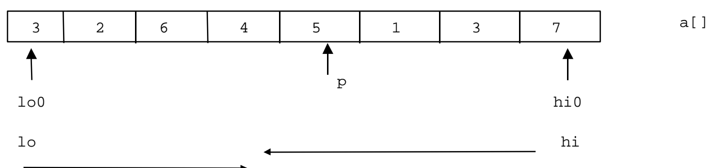
```
qs :: Ord a => [a] -> [a]
qs [] = []
qs (p:xs) = qs ( [x | x <-xs , x < p]) ++ [p]
         ++ qs ( [x | x <-xs , x >= p])
```

Pivotelement

nur rekursiver Aufruf „sichtbar", Teilen des Felds in ZF-Notation „versteckt"

Beachte Einfluss des Abstraktionsniveaus der Sprache auf Algorithmus!

Pivotelement p: wünschenswert: die Hälfte der Elemente ist kleiner, die andere größer als p. Welchen Index p hat, ist belanglos. Auf den Wert kommt es an



Quicksort: Partitionieren

1.

3 2 6 4 **5** 1 3 7

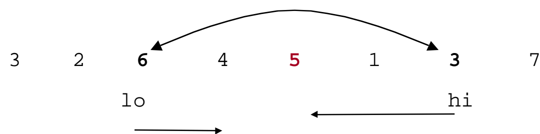
```
while( ( lo < hi0 ) && ( a[lo] < p )) ++lo;
```

3 2 **6** 4 **5** 1 3 7

2.

lo

```
while( ( hi > lo0 ) && ( a[hi] > p )) --hi;
```




```
swap();
lo++;
hi--
```

3.

3 2 **3** 4 **5** 1 **6** 7

Quicksort: Partitionieren

3.

3 2 3 4 **5** 1 6 7

lo → ← hi

```
while( ( lo < hi0 ) && ( a[lo] < p )) ++lo;
```

```
while( ( hi > lo0 ) && ( a[hi] > p )) --hi;
```

4.

3 2 3 4 **5** **1** 6 7

lo → ← hi

```
swap();
lo++;
hi--
```

```
while( lo <= hi )
```

3 2 3 4 **1** | **5** 6 7

← hi | lo →

Bedingung besagt: Partition noch nicht fertig!

< 5

Hier gilt: lo > hi: fertig!

>= 5

Quicksort: Spezifkation des Partitionierens

```
//{ min a[i] <= p <= max a[i], i=0...a.length}
  while( lo <= hi ) S;
//{ lo > hi ∧  ∀i< lo: a[i] <= p ∧ ∀j > hi: a[i] >= p}
```

```
//{ Min a[i] <= p <= max a[i], i=0...a.length}
  while( lo <= hi ) {

   // {INV =  ∀ i: lo0 <= i < lo : a[i] < p}
    while( ( lo < hi0 ) && ( a[lo] < p )) ++lo;
   // {(lo = hi0 ∨ a[lo]>= p) ∧ ∀ i: lo0 <= i < lo : a[i] < p}

   // {INV: ∀ j: hi0 >= j > hi   : a[j] > p}
   while( ( hi > lo0 ) && ( a[hi] > p )) --hi;
  // {(hi = lo0 ∨ a[hi]<= p) ∧ ∀ j: hi0 >= j > hi : a[j] > p}
  //  {(lo = hi0 ∨ a[lo] >= p) ∧ {(hi = lo0 ∨ a[hi] <= p) }
    if (lo <= hi)
     {swap(); lo++; hi--}
  // { ....}
 }
 //{ lo > hi ∧ ∀ i < lo: a[i] <= p ∧ j > hi a[i] >= p}
```

Selbst überlegen!
Terminiert der Algorithmus? Das ist nicht trivial!

Quicksort: initialisieren

```
....
* nach J. Gosling,K. Smith (java demo)
* @param a       an integer array
* @param lo0     left boundary of array partition
* @param hi0     right boundary of array partition
*/
static void  QuickSort(int a[], int lo0, int hi0)     {
  int lo = lo0;
  int hi = hi0;
  int p;
  if ( hi0 > lo0)
  { /* Arbitrarily establishing partition element as the midpoint of
     * the array.
     */
    p = a[ ( lo0 + hi0 ) / 2 ];
    // loop through the array until indices cross
```

Quicksort: partitionieren

```
while( lo <= hi )
 {  /* find the first element that is greater than or equal to
             * the partition element starting from the left Index.
             */
   while( ( lo < hi0 ) && ( a[lo] < p ))
                 ++lo;
            /* find an element that is smaller than or equal to
             * the partition element starting from the right Index.
             */
   while( ( hi > lo0 ) && ( a[hi] > p ))
               --hi;
            // if the indexes have not crossed, swap
   if( lo <= hi )
     { swap(a, lo, hi);
      ++lo; --hi;
     }
  } // while ( lo <= hi)
```

Darf es hier auch „a[lo] <= p „ heissen?

... Quicksort: rekursiver Teil

```
/* If the right index has not reached the left side of array
 * must now sort the left partition.
 */
 if( lo0 < hi )
   QuickSort( a, lo0, hi );
   /* If the left index has not reached the right side of array
    * must now sort the right partition.
    */
  if( lo < hi0 )
   QuickSort( a, lo, hi0 );
} // if (hi0 > low0)..
} // Quicksort
```


Laufzeit Quicksort

Schlechtester Fall: Behauptung: $O(n^2)$

jede Partitionierung spaltet nur ein Element ab

Partitionierung hat Laufzeit: $\Theta(n)$

Damit Zeitaufwand t(n) zum Sortieren:

$$t(n) = t(n-1) + \Theta(n)$$
$$= S_{i=1..n}\, \Theta(i)$$
$$= \Theta(S_{i=1..n}\, i)$$
$$= \Theta(n^2)$$

← Rekurrenzgleichung

Wirklich??

Genauer: $t(n) <= max_{1 <= p <= n-1}\, t(n-p) + t(p) + \Theta(n)$

$t(n) <= max_{1 <= p <= n-1}\, c*(n-p)^2 + c*p^2 + \Theta(n)$

$=c* \; max_{1 <= p <= n-1}\, (n-p)^2 + p^2 + \Theta(n)$

=> $t(n) <= \; c*(n-1)^2 + c* 1^2 + \Theta(n)$

$= c* n^2 + 2*c*(n-1) + \Theta(n) <= \; c*n^2 = \Theta(n^2)$

Lösung raten:
$t(n) <= c* n_2$
für eine Konstane c

Maximum für p=1 bzw p = n-1

..... Laufzeit Quicksort

Durchschnittlicher Fall:

```
qs (p:xs) = qs ( [x | x <-xs , x < p]) ++ [p]
         ++ qs ( [x | x <-xs , x > p])
```

Annahmen:

- Pivotelement p ist Element von Feld/Liste a[ ] mit n Elementen,
- Keine doppelten Werte
- für jeden Wert x: jeder Feldindex i mit a[i] = x gleichwahrscheinlich
- p kein Element der Partion

wie hier

Vorgehensweise: Partionierungsaufwand abschätzen,
Rekurrenzgleichung aufstellen (das ist der trickreiche Teil!)
Mit geeigneter Technik lösen

Partitionierungsaufwand: Jedes Feldelement wird mit p verglichen, ggf. Vertauschung
=> linearer Aufwand n (Konstanten weggelassen, Vereinfachung)

Partitionierung liefert Liste mit kleineren bzw größeren Elementen als p
Länge beider Listen jeweils : 0, 1, 2, ..., (n-1) , jede mit gleicher Wahrscheinlicheit

=> $t(n) = 2(t(0) + t(1) + ... + t(n-1)) / n \quad + \; n$

linearer Partitionierungaufwand

...Laufzeit

=> $t(n) = 2(t(0) + t(1) + ... + t(n-1)) / n \quad + \; n$

$t(n) - t(n-1) \quad => \quad \mathbf{t(n) / n+1 = t(n-1)/(n+1) + 2/(n+1)}$ (*)

Rekurrenzgleichung (*) lösen durch Einsetzen:

$$t(n) / (n+1) = t(1)/2 + 2(1/3 + 1/4 + ... 1/(n+1))$$
$$= 2 (1 + 1/2 + ... 1/(n+1)) - 5/2$$
$$...$$
$$= \Theta(log\ n)$$

Wegen $h(n) = \Sigma_{i=1...n}\, i$
$= \Theta(log\ n)$
(Harmonische Zahlen)

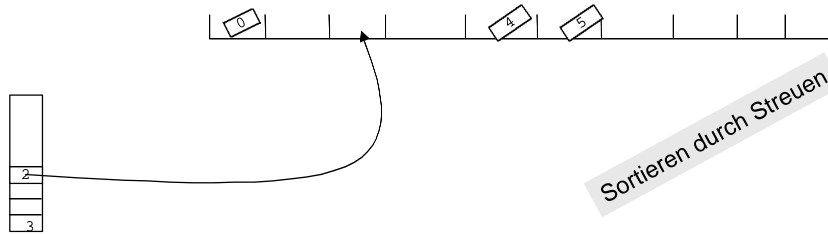
Warum verhält sich Quicksort im Durchschnitt besser als einfache Austauschverfahren?

- Austausch von Feldelementen mit großem Abstand günstiger als Tausch direkter Nachbarn
- Wahrscheinlichkeit ein ungeeignetes Pivotelement zu treffen gering
  ... aber gelten die Annahmen? Gleichverteilung, keine Duplikate, ...?

Kann man schneller sortieren ?

Ja!

Z.B. Einsortieren in „Fächer“ : gibt es für jeden möglichen Wert ein Fach,
braucht man zwei lineare Durchläufe.
1. Einsortieren, 2. sortiert auslesen => O(n) im besten, mittleren, schlechtesten Fall



Und Sortieren durch Vergleiche zwischen je zwei Werten?

Nein!

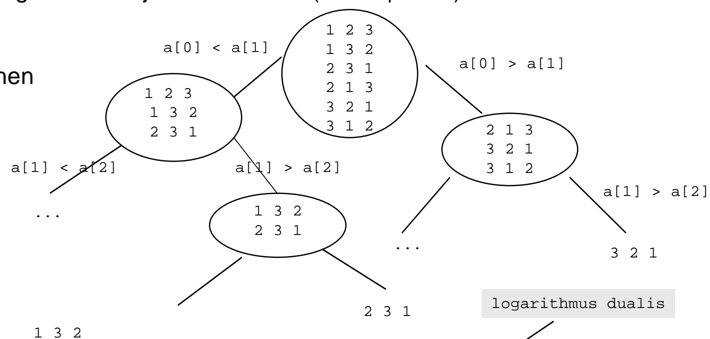
Und wie kann man das allgemein zeigen?

Sortieren durch Vergleichen: untere Schranke

Duales Problem: gleich wahrscheinlich

- suche aus allen n! Permutationen der n zu sortierenden Werte die zu sortierende Folge durch binäre Vergleiche von je zwei Feldern (a < b | a > b)

In jedem Schritt:
Einschränken der möglichen Permutationen gemäß binärer Entscheidung



Jedem **Blatt** entspricht **genau eine Permutation,** also n! Blätter

Baum der Höhe h: maximal 2^h Blätter => $n! <= 2^h$ <=> $ld\,(n!) <= h$

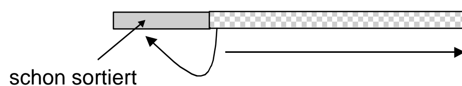
=> $h >= n\,ld\,n - n\,ld\,e = \Omega(n\,ld\,n)$

Stirlingsche Formel: $n! >= \left(\frac{n}{e}\right)^n$

e=2,7182..

Klassifizierung von Sortierverfahren

- **Einfügen**
  - zu sortierende Folge elementweise in (teil)sortierte einfügen

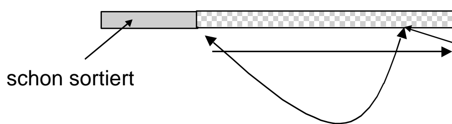


schon sortiert

schlechte Laufzeit

- **Aussuchen**
  - suche Element mit geeigneter Eigenschaft z.B. kleinstes, größtes und bringe es an die richtige Stelle

auch: Heapsort



schon sortiert

Direktes Aussuchen:

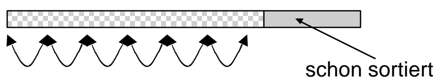
kleinstes Element im unsortierten Bereich wird mit erstem vertauscht.
Sortierte Teilfolge wächst jeweils um 1

schlecht

- **Sortieren durch Streuen**

Klassifikation der Sortierverfahren

- **Austauschen**
  - Finde Inversion : i < j und a[i] > a[j] , tauschen



schon sortiert

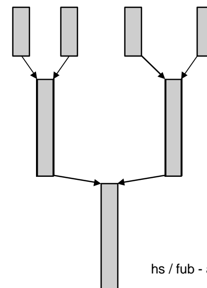
Bubble-Sort: prüfe von links nach rechts auf Inversion, größtes Element sinkt nach unten und verlängert den schon sortierten Bereich

Austauschen auch in Quicksort, Heapsort, ..

schlecht

gut!

- **Mischen**
  - Zerlegen in k sortierte Teilfolgen
  - Mischen der Folgen in Sortierfolge

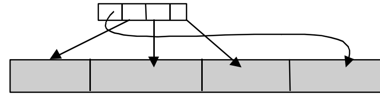


Merge-Sort (Mischsortieren),
externes Sortieren (Dateien)

gut

Sortieren: Praktische Aspekte

- Art der Elemente
  - statt Datensätze Stellvertreter sortieren
  - Stabilität nötig?
    „Reihenfolge gleicher Elemente bleibt erhalten“
    (nicht nur vom Algorithmus, sondern von Implementierung abhäöngig)
  - Sind Elemente vorsortiert?
    Kann Algorithmus das ausnutzen?



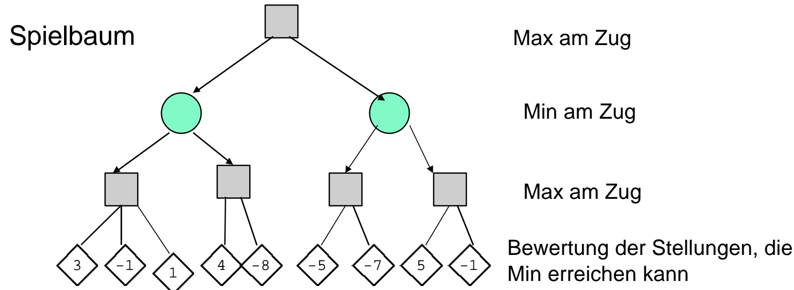
Ausführliche Abhandlung zum Sortieren:
Donald Knuth:
Fundametals of Computer Programming, Vol3
Sorting and Searching, Addison Wesley

- Anzahl der Elemente
  - Kleine Anzahl von Elementen: einfache Algorithmen schneller
  - Die meisten Verfahren nur für internes Sortieren geeignet, beachte: anderes Kostenmaß bei externen Operationen
- Wahl des Pivotelements (Quicksort) beinflusst Effizienz wesentlich

7. Spiele

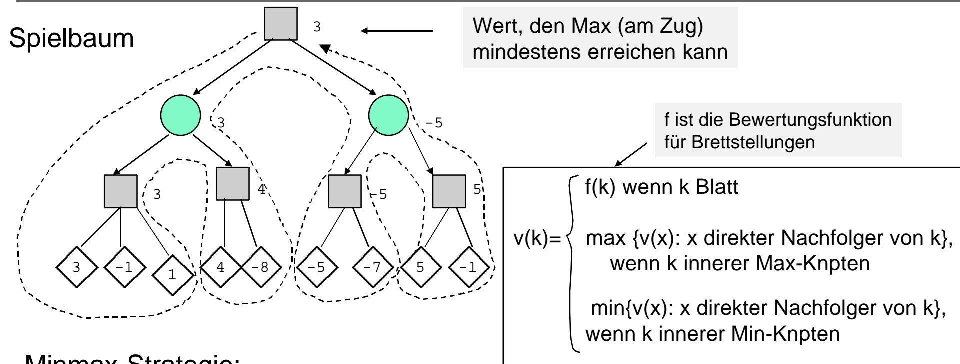
- Spiele und Wissenschaft
  - Glücksspiel ---> Wahrscheinlichkeitstheorie
  - Gesellschaftsspiele --> Mathematisch-ökonomische Spieltheorie
  - Puzzle usw. -> Heuristisches Problemlösen
  - Computerschach --> Suchverfahren, Wissensformalisierung
- Hier nur rein strategische, endliche Zweipersonen-Nullsummenspiele mit vollständiger Information
  - rein strategisch: kein Zufall, Spieler treffen alle Entscheidungen
  - endlich: nur endlich viele (aber meist viele!) Züge möglich, ggf. Zugwiederholungen ausschliessen
  - Vollständige Information: kein verdeckter Informationsvorsprung, alle wissen gleich viel,
  - Zwei Personen: keine Koalitionen, Kooperation, einfacher als n Personen
  - Nullsummenspiel: was A gewinnt, verliert B, Auszahlungsfunktion f(A)+f(B) = 0
    z.B. +1 (Gewinn), -1 (Verlust), 0 Unentschieden

Spielbäume



- Spiel wird vollständig durch Baum beschrieben
- Einzelspiel Min versus. Max entspricht Pfad im Baum
- Min, Max ziehen abwechselnd
- Min strebt minimalen, Max maximalen Wert an
- Annahme: beide verhalten sich optimal
- Aber...: Spielbäume meist SEHR groß (Schach: ca. 10^{120})

Minimax-Strategie



Minmax-Strategie:
Max wählt den Zug, der maximalen Wert verspricht, wenn Min sich optimal verhält => Max wählt immer Alternative zum größten Folgeknoten (Zug)

Min dual: wähle den Zug mit minimalem Wert, wenn Max jeweils maximalen Wert anstrebt => Min : wähle minimalen direkten (Zug)

-> rekursive Baumtraversierung

Binäre Bewertung und Schnitte

Spielbaum

Schnitt;
Max kann seinen Wert (
1 aus linkem Teilbaum)
nicht erhöhen, also
rechten ignorieren

Hierher kommen wir garnicht,
schon weggeschnitten

Binäre Bewertung: Gewinn, Verlust
Bewertung innerer Knoten: & bzw. | auf Nachfolger anwenden!

Lohnt sich das Sortieren
der Nachfolger eines Knotens?
(Wieviel) spart man maximal?

Beobachtungen:

- Spielbaum mit binärer Bewertung identisch zu Operatorbaum mit & und | als Operatoren
- „Lazy-Semantik“ von & und | führt zu Schnitten: Verschneiden gewisser Zweige möglich ohne Verlust an Information!

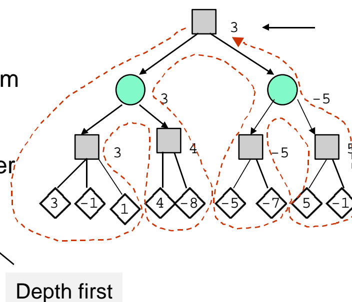
Ist das auch für allgemeinere
Bewertungsfunktionen
möglich?

Auswerten von Spielbäumen

Naiv: Erzeugen aller Knoten, dann Auswertung

Tiefensuche: Linker Teilbaum, Wurzel, rechter Teilbaum

Typischer Backtrack-Algorithmus:
hier: Knoten beim Absteigen besuchen und später dahin zurückkehren



Buchhaltung am einfachsten dem Laufzeitsystem überlassen => Rekursion

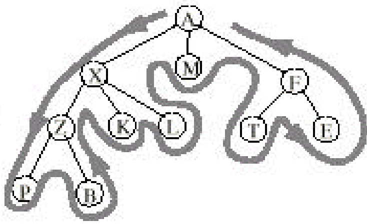
Depth first

Alternative:

Best first

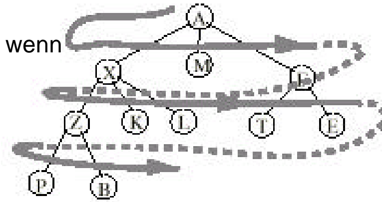
Angenommen, jeder Knoten hat einen Schätzwert.
Suchheuristik: expandiere den Knoten mit höchstem Schätzwert (Max).
Erlaubt selektive Entwicklung des Baums gemäß Schätzung.
Nachteil: explizite Buchführung nötig
Wie findet man gute Schätzung
Für große Spielbäume unerlässlich.

Tiefen- und Breitensuche



Breitensuche (breadth first): Knoten einer Ebene erst dann erzeugen, wenn alle der darüberliegenden Ebene erzeugt wurden. Speicheraufwand! In jedem Schritt verdoppelt sich die Gesamtanzahl Knoten etwa (Verzweigungsgrad 2)

Tiefensuche (depth first): Knoten erst erzeugen, wenn alle weiter links stehenden schon erzeugt und untersucht wurden.



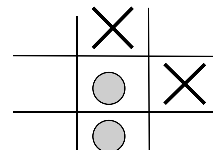
Typischer **Backtrack**-Algorithmus: zurückkehren zu „Knoten", die man schon kennt und fortsetzen mit Alternative.
Buchführung dafür explizit (mühsam!) oder im Laufzeitstapel (Rekursion!)

Bestensuche (best fit) : günstig, Effektivität hängt von guter Heuristik ab („wie weit ist es noch zum Ziel?"

Beispiel: TicTacToe

Repräsentation: `int [] [] board = new int [3] [3]`

Bewertung: HUMAN_WIN | COMPUTER_WIN, DRAW



```
class Best {int row, column, val}

        public Best chooseMove( int side )
        {   int opp;              // The other side
            Best reply;           // Opponent's best reply
            int dc;               // Placeholder
            int simpleEval;       // Result of an immediate evaluation
            int bestRow = 0;
            int bestColumn = 0;
            int value;

            if( ( simpleEval = positionValue( ) ) != UNCLEAR )
                return new Best( simpleEval );

            if( side == COMPUTER )
            {
                opp = HUMAN; value = HUMAN_WIN;
            }
            else
            {
                opp = COMPUTER; value = COMPUTER_WIN;
            }
```

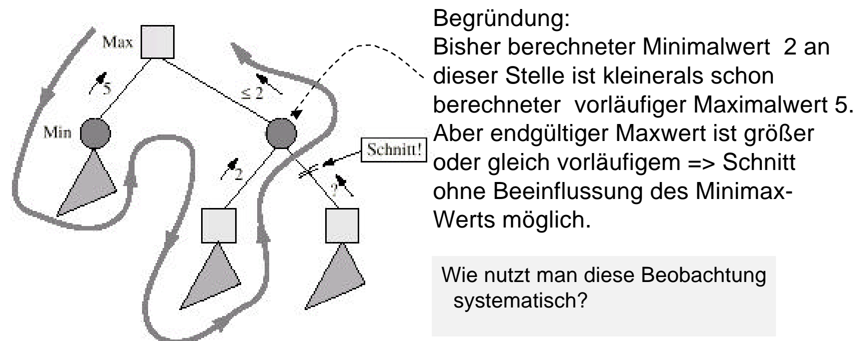
TicTacToe : Backtrack

```
Outer:
        for( int row = 0; row < 3; row++ )
            for( int column = 0; column < 3; column++ )
                if( squareIsEmpty( row, column ) )
                {
                    place( row, column, side );
                    reply = chooseMove( opp );
                    place( row, column, EMPTY );
                    if( side == COMPUTER && reply.val > value ||
                        side == HUMAN && reply.val < value )
                    {
                        if( side == COMPUTER )
                            value = reply.val;
                        else
                            value = reply.val;

                        bestRow = row; bestColumn = column;
                    }
                }

        return new Best( value, bestRow, bestColumn );
    }
```

Backtrack!
Effekt rückgängig machen und Alternative probieren (for-Schleifen)

Baumschnitte

Spielbäume sind groß! Untersuchung aller Knoten praktisch (und theoretisch?) unmöglich, etwa 10^{120} Schachstellungen
Auch Untersuchung *aller* Knoten bis zu fester Tiefe kaum durchführbar. Geschicktes Beschneiden garantiert optimalen Wert ohne Minimax-Wert zu beeinflussen!



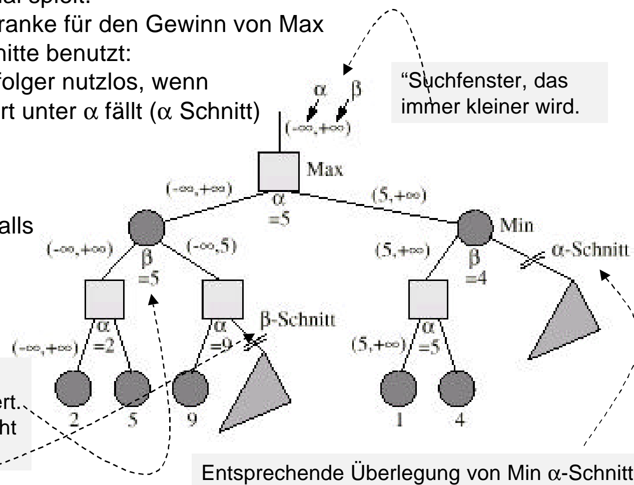
Begründung:
Bisher berechneter Minimalwert 2 an dieser Stelle ist kleinerals schon berechneter vorläufiger Maximalwert 5. Aber endgültiger Maxwert ist größer oder gleich vorläufigem => Schnitt ohne Beeinflussung des Minimax-Werts möglich.

Wie nutzt man diese Beobachtung systematisch?

Das α–β - Prinzip

α-Schranke:
- ist der Wert, den Max in jedem Fall erzielen kann, wenn er (wie angenommen) optimal spielt.
- wächst monoton, untere Schranke für den Gewinn von Max
- wird von Min-Knoten für Schnitte benutzt:
  Berechnung weiterer Nachfolger nutzlos, wenn bisher berechneter Min-Wert unter α fällt (α Schnitt)

β-Schranke symmetrisch als
Wert, den Min bisher bestenfalls erzielen kann.

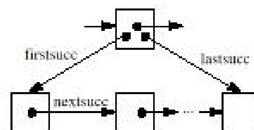


Hier ist das bisherige Maximum (9) schon größer als bisheriger Min-Wert. Also wird Min diesen Zug sicher nicht machen.

Der Algorithmus

```
int maxValue (GameState g, int α , int β )
{ if (cutofftest( g)) return eval( g);
for (GameState s = g. firstsucc;
    s != g. lastsucc; s = s. nextsucc)
 {α = max( α , minValue( s, α , β ));
  if ( α >= β ) return b ; // β -Schnitt
 }
return α ;
}

int minValue (GameState g, int α , int b )
{ if (cutofftest( g)) return eval( g);
for (GameState s = g. firstsucc;
   s != g. lastsucc; s = s. nextsucc)
 { β = min( β , maxValue( s, α , β ));
   if ( β <= α ) return α ; // α -Schnitt
 }
return β ;
}
```

Ist die value-Parameterübergabesemantik hier angemessen? Besser value-result?



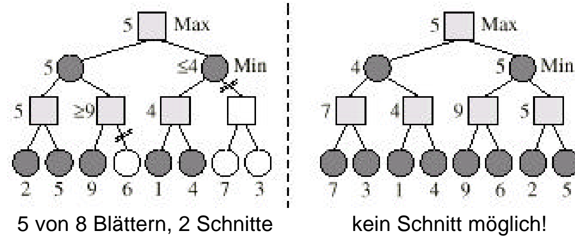
veröffentlicht von J.McCarthy, 1953

Tiefensuch-Algorithmus, aber nur die Knoten expandieren, die den Max / Min - Wert beeinflussen können.

`cuttofftest(): true,` wenn maximale Baumtiefe erreicht
`eval()` bewertet Spielstellung

Effizienz des a-b-Algorithmus

Wieviel gewinnt man durch Baumschnitte?



5 von 8 Blättern, 2 Schnitte kein Schnitt möglich!

Reihenfolge der Knoten einer Ebene **entscheidend** für Anzahl auszuwertender Blätter:
- Min-Knoten aufsteigend,
- Max-Knoten absteigend sortiert halten.

Gewinn im günstigsten Fall:
Annahme : pro Knoten w direkte Nachfolger, Tiefe d

- Minimax : w^d,
- Bester Fall bei α-β: $w^{d/2}$ (ohne Beweis)
- Durchschnitt? Experimentell: selbst ohne Sortieren der Ebenen 1/3 Knoten weniger zu untersuchen

α–β wird in praktisch allen 2-Personen-Nullsummen-spielen eingesetzt**.**
Zusätzlich wichtig: Verfahren, die Mehrfachberechnung von Zügen vermeiden.

In Alp2 behandelte Themen

- Einführung
  - Imperative Programme als Abbildungen des **Zustandsraums**
  - Historisches zu Programmiersprachen
  - Übersetzen, Interpretieren von Programmen
  - Graphische Darstellung von Algorithmen (Flussdiagramm)
  - Anweisungen: Zuweisung, Sequenz, Schleifen (Iteration), Sprünge
  - Syntax: kontextfreie Grammatik, (E)BNF; Syntaxdiagramme
- Imperative Sprachkonzepte
  - Elementare (Basis-) Typen
  - Typumwandlung
  - statische / dynamische Typisierung
  - Sprachkonstrukte zur Modellierung: Verbund („record"), Klasse
  - Referenzen, dynamische Erzeugung , Objekte

In Alp2 behandelte Themen

- Datenabstraktion: Klassen, Abstrakte Datentypen (Haskell), Module
- Objekte in Java: Konstruktoren, Zugriffsrestriktionen
- Parameterübergabe: call by value, reference, value-result, name
  copy-, Referenz-Semantik
- Felder
- Listen: einfach / doppelt verkettet, zyklisch
- Namensräume: Gültigkeit, Sichtbarkeit; Namensräume in Java

- **Rekursion und Iteration**
  - Endrekursion, Entrekursivierung
  - Rekursive Grafik: Raumfüllende Kurven u.a.
  - Darstellung und Auswertung von Formeln
    Keller, Postfix, Praefix, Infix-Darstellungen und Umwandlung
  - Formelübersetzung
  - Übersetzung / Interpretation: Virtuelle Maschine von Java (kurz)

In Alp2 behandelte Themen

- **Typsysteme**
  - Polymorphie, Generizität, Überladen
  - Typklassen in Haskell
  - Java: Interface
  - Typen und Subtypen
  - Co- / Kontravarianz in Methodenaufrufen
  - Zuweisungskompatibilität
  - (Unter-)Klassen
  - Generisches Methoden / Klassen in Java (soweit möglich)
    Sortieren, Listen
  - Ausnahmebehandlung

In Alp2 behandelte Themen

- Spezifikation und Verifikation von Programmen
  - Formale Spezifikation durch Prädikate über dem Zustandsraum
  - Voraussetzung und Effekt
  - Starke, schwache Spezifikationen
  - Zusicherungen
  - Verifkation durch Formalisierung der Semantik von imperativen Programmen: Hoare-Kalkül, schwächste Voraussetzung
  - partielle und totale Korrektheit
  - Schleifeninvarianten und wie man sie findet
  - Systematische Programmentwicklung mit Zusicherungen
  - Testen (kurz)

In Alp2 behandelte Themen

- Laufzeitanalyse
  - das Instrumentarium: O-Notation, Ω , Θ
  - Sortierverfahren und Laufzeitanalyse
    Insertion-Sort, Quicksort (durchschnittlicher Fall)
  - Klassifizierung anderer Sortierverfahren
  - untere Schranke für Sortierverfahren mit Elementvergleich

- Spiele
  - Zweipersonen-Nullsummenspiele mit vollständiger Information
  - Minimax-Algorithmus
  - α–β-Algorithmus

Nicht behandelt:

Registermaschine (-> Turing-Maschine Theor. Inf.)
Benutzeroberflächen