

Zesde college algoritmiëk

16 maart 2007

Backtracking

Verdeel en Heers

Basisidee backtracking

- bouw een oplossing stap voor stap op en controleer steeds of de deeloplossing in conflict komt met de restricties
- op elk moment kun je kiezen uit een aantal mogelijke vervolgstappen; maak een keuze en ga langs die weg verder met het opbouwen van de oplossing
- als een keuze op niets uitloopt, herzie je deze keuze en probeer je een andere mogelijkheid

Vergelijk

- het vinden van de uitgang in een doolhof: loop steeds verder en als je bij het zoeken vastloopt, ga terug op je pad om het laatste open alternatief te proberen

Knapzakprobleem

Gegeven n objecten, met gewicht w_1, \dots, w_n en waarde v_1, \dots, v_n , en een knapzak met capaciteit W . **Gevraagd:** de meest waardevolle deelverzameling der objecten die in de knapzak past (dus met totaalgewicht $\leq W$).

Voorbeeld:

object	gewicht	waarde
1	8	42
2	3	14
3	4	40
4	5	27

knapzakcapaciteit 12

- genereer de deelverzamelingen **stap voor stap**, bijvoorbeeld door steeds aan een goede deelverzameling van de objecten 1 t/m i achtereenvolgens object $i + 1$ of $i + 2$ of $\dots n$ toe te voegen (mogelijke **keuzes**)
- **controleer** of het totaalgewicht van de aldus uitgebreide verzameling nog steeds $\leq W$ is
- zo nee, **herzie** dan **je keuze** (en probeer het volgende object)
- zo ja, ga dan op **dezelfde** manier verder
- houd ook de totaalwaarde van de (deel)verzamelingen bij en de tot dusver gevonden maximale waarde

Traveling Salesman Problem (handelsreizigersprobleem)

Gegeven n steden waarvan alle onderlinge afstanden bekend zijn. **Gevraagd:** de/een kortste route die elke stad precies één keer aandoet, en weer terugkeert in het vertrekpunt.

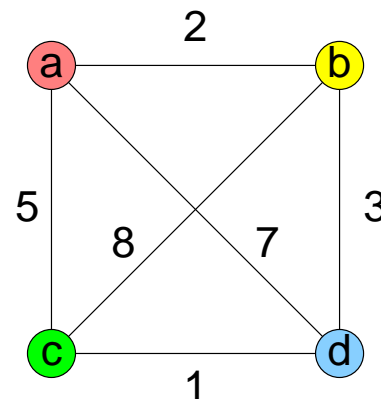
Ofwel: vind de/een kortste Hamiltonkring in een samenhangende gewogen (complete) graaf.

Voorbeeld:

minimale route

a b d c a

(of a c d b a)



Merk op: elke permutatie van de knopen is een Hamiltonkring, dus:

- genereer alle permutaties van de knopen (beginnend bij knoop a) **stap voor stap**, door deelpermutaties (= beginstuk Hamiltonkring) uit te breiden (zie hierna)
- houd de tot dusver gevonden minimale lengte min bij
- **controleer** of de lengte van de deelpermutatie kleiner is dan min
- zo ja, ga dan op **dezelfde** manier door met uitbreiden
- zo nee, dan heeft het geen zin om door te gaan, dus **herzie je meest recente keuze**

Genereer alle permutaties van 1 t/m n met backtracking.

```
void permutaties(int n, int perm[ ], int hierzo) {
    int i;
    if (hierzo == n+1)
        drukaf(perm,n); // permutatie gevonden
    else {
        for (i=1; i<=n; i++) {
            perm[hierzo] = i;
            if (!aanwezig(perm, i, hierzo)) // test of i
                // al in perm[1] t/m perm[hierzo-1] voorkomt
                permutaties(n, perm, hierzo+1);
        } // for
    } // else
} // permutaties
```

Vraag: wat heeft dit met torens op een schaakbord te maken?

Gegeven een rechthoekig doolhof. Gevraagd wordt een pad van Start naar Eind, waarbij alleen horizontaal en verticaal gelopen mag worden.



```

XXXXXXXXXXXXX
X      X      X
X  X  X  XXX  XX
XXX  X  X      X
X      X      X  XX
X  XXXXXXXX  XX
X      X      X
XXXXX  X  X  X  X
S   XXX  X  X  X
XX      X  X  X
X   X  X  X  X  X
XXXXXXXXXXEXXX
    
```



```

XXXXXXXXXXXXX
X   ***X   X
X  X*X*XXX  XX
XXX*X*X***OX
X***X***X*XX
X*XXXXXXXX*XX
X*****X*OX
XXXXOX*X*XOX
S**XXX*X*XOX
XX*****X*XOX
X   X  XOX*XOX
XXXXXXXXXXEXXX
    
```



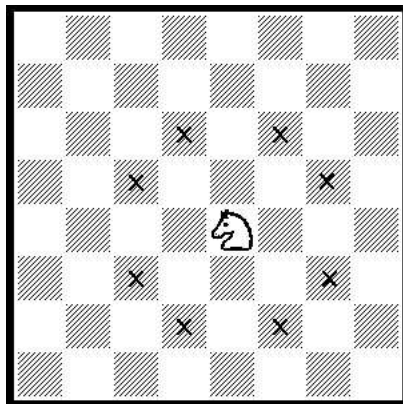
```
bool dwaal(int x, int y) {
// is er een pad van (x,y) naar (x_eind,y_eind) ?
  int richting, x_volgende, y_volgende;
  if ( ( x == x_eind ) && ( y == y_eind ) ) { // gevonden!
    doolhof[x][y] = '*';
    return true;
  }
  else if ( doolhof[x][y] != ' ' ) { // geen vrije plek
    return false; }
  else {
    doolhof[x][y] = '?'; // tijdelijk markeren; voorkomt ∞ loopen
    for ( richting = OOST; richting <= NOORD; richting++ ) {
      x_volgende = volgende_x(x,richting);
      y_volgende = volgende_y(y,richting);
      if ( dwaal(x_volgende,y_volgende) ) {
        doolhof[x][y] = '*';
        return true;
      }
    }
    doolhof[x][y] = '0'; // afgehandeld: geen pad via deze (x,y)
    return false;
  }
}
```

Laten oplossingen van de vorm $(X[1], X[2], \dots, X[n])$ zijn en zij S_i de verzameling waarden die $X[i]$ kan aannemen. De algemene vorm van een backtracking algoritme is dan:

```
backtrack( $X[1 \dots i]$ )::
//  $X[1 \dots i]$  is een veelbelovende deeloplossing, consistent met
// de restricties; we zoeken alle oplossingen
if  $X[1 \dots i]$  is een oplossing then
    print( $X[1 \dots i]$ );
else
    for elke  $x \in S_{i+1}$  consistent met  $X[1 \dots i]$  en de restricties do
         $X[i + 1] := x$ ;
        backtrack( $X[1 \dots i + 1]$ );
    od
fi
```

Opdracht:

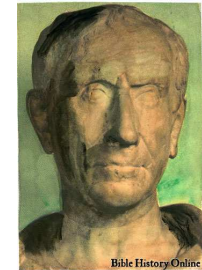
- Bepaal een serie van $m * n - 1$ sprongen van het paard op een m bij n bord, zodat elk veld precies één keer bezocht wordt.
- Idem, maar nu moet het paard weer in het startvakje terugkomen.



beweging van het paard

1	14	17	20	11	8	5
16	19	12	3	6	21	10
13	2	15	18	9	4	7

een oplossing op het 3 bij 7 bord

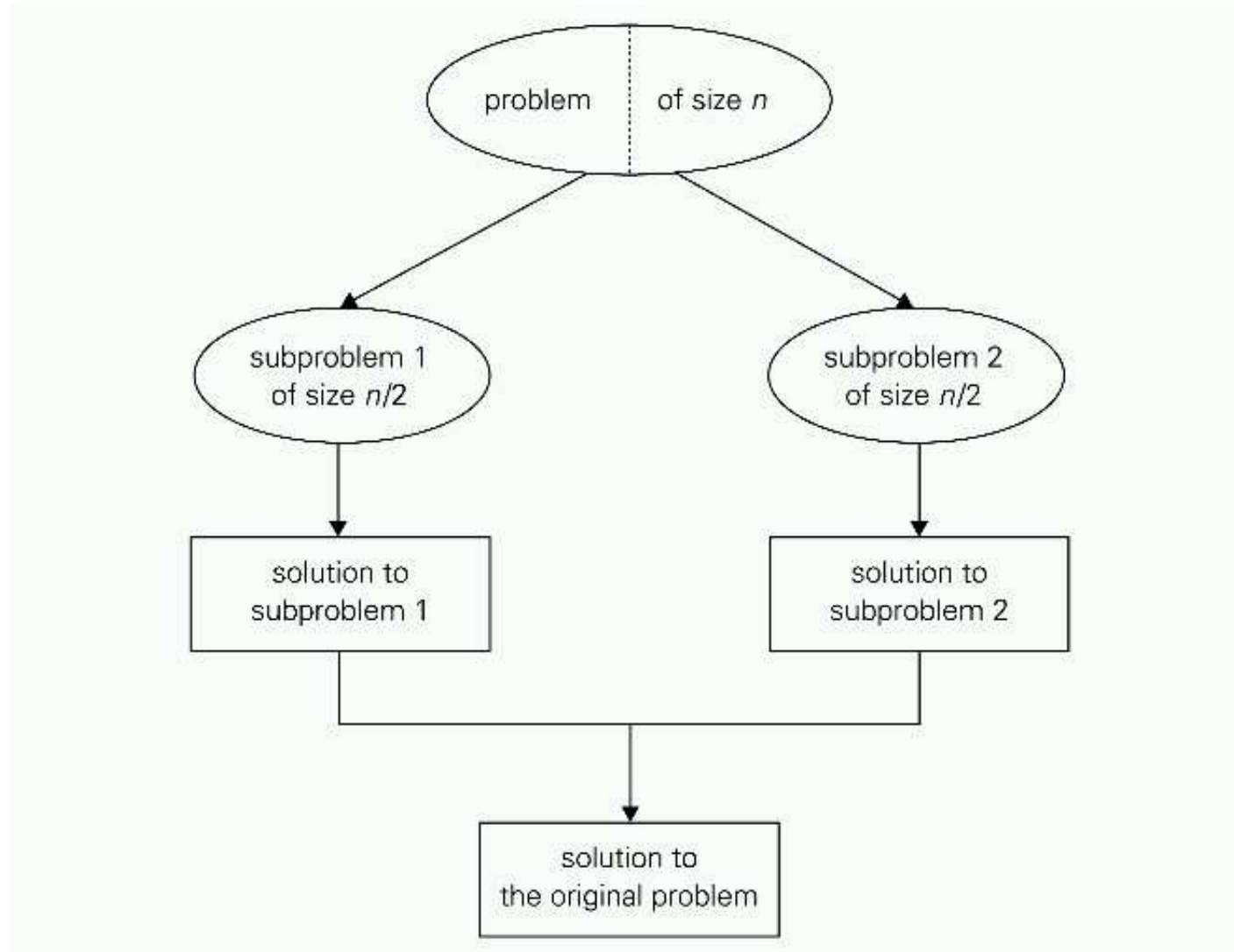


Divide and Conquer

1. Verdeel een instantie van het probleem in twee of meer kleinere instanties
2. Los de kleinere instanties op: meestal **recursief**
3. Combineer deze twee oplossingen tot een oplossing van de oorspronkelijke (grotere) instantie

Opmerking: meestal wordt een probleeminstantie in twee ongeveer gelijke delen verdeeld.

Verdeel
en heers



Decrease and Conquer

1. Reduceer een instantie van het probleem tot een kleinere instantie van hetzelfde probleem
2. Los de kleinere instantie op: vaak **recursief**
3. Breid de oplossing van de kleinere probleeminstantie uit tot een oplossing van de oorspronkelijke instantie

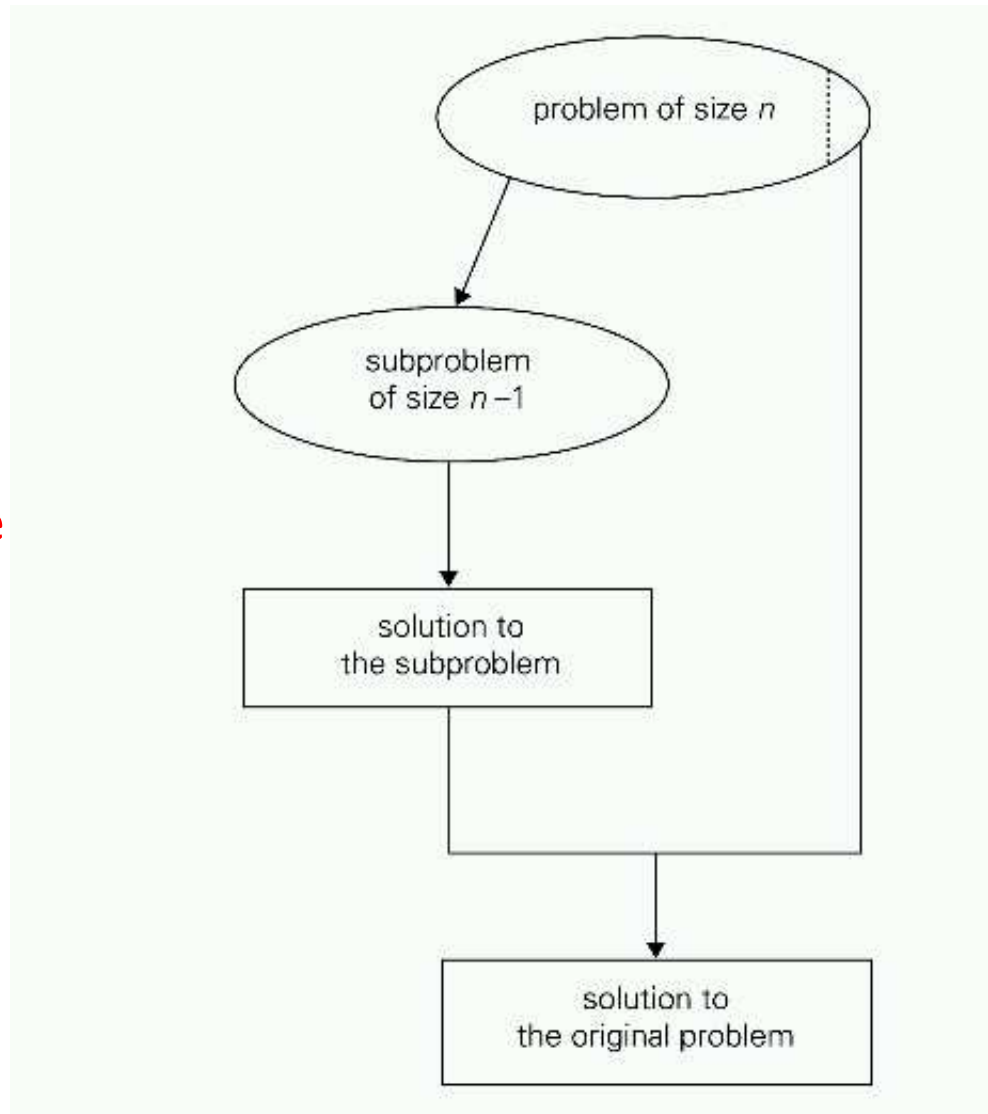
In het boek wordt onderscheid gemaakt tussen:

Decrease by one

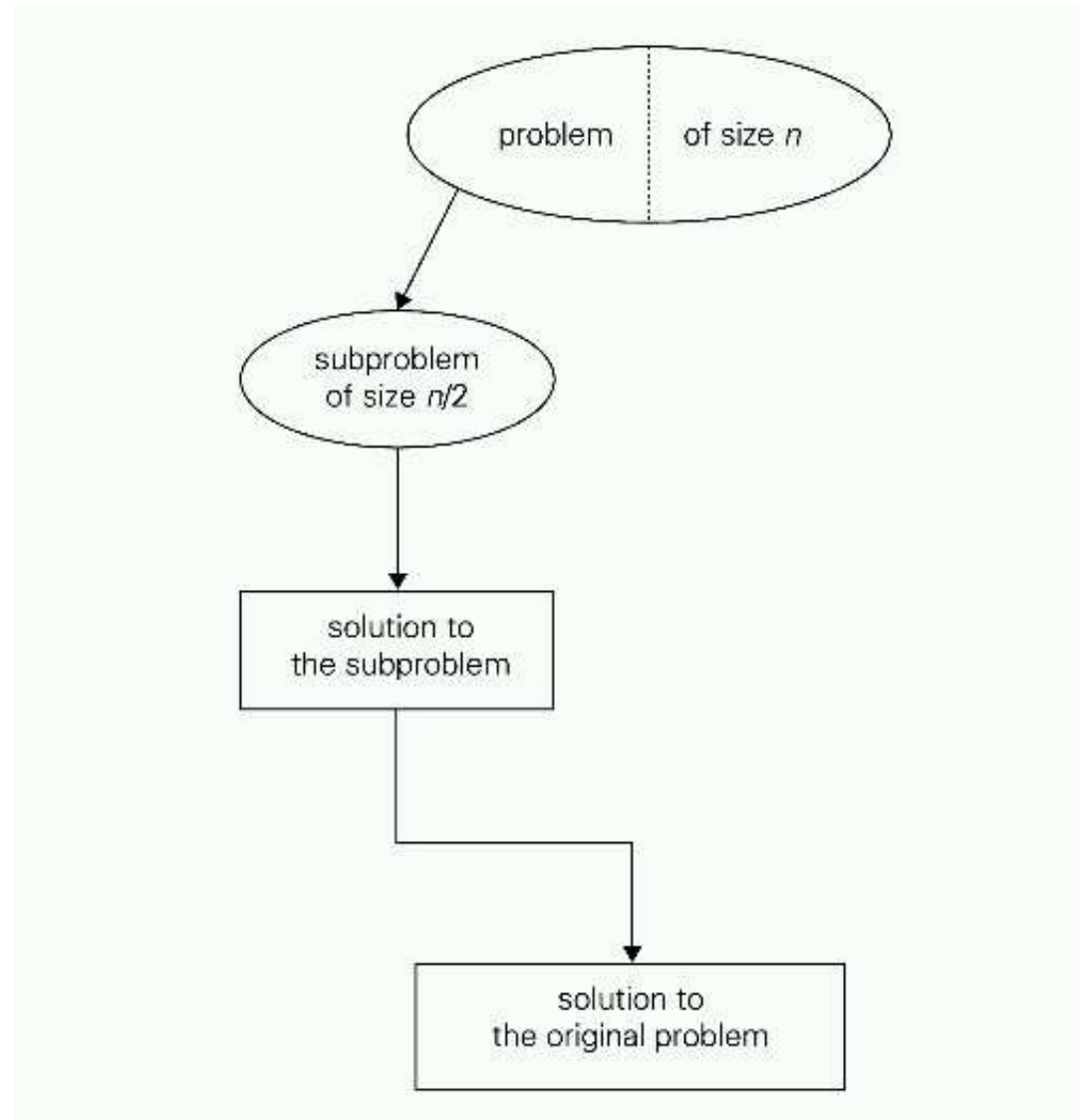
Decrease by a constant factor

Variable-size decrease

Decrease
by one



Decrease by a constant factor



Verdeel en heers en sorteren:

Sorteer(rij)::

if (de rij heeft meer dan één element) **then**

Verdeel de rij in twee stukken: linkerrij en rechterrij;

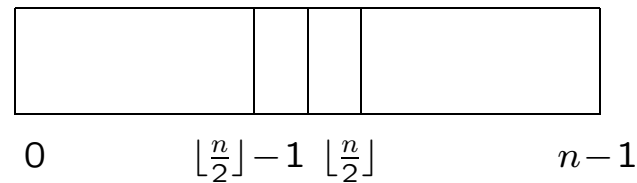
Sorteer(linkerrij);

Sorteer(rechterrij);

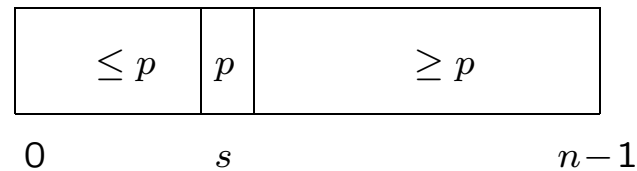
Combineer linkerrij en rechterrij;

fi .

Divide and conquer

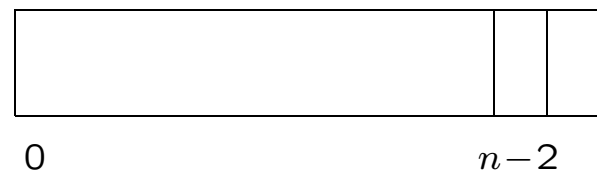


Mergesort



Quicksort

Decrease and conquer (decrease by one)



Insertion sort

```
Mergesort( $A[0 \dots n - 1]$ )::  
// sorteert het array  $A[0..n - 1]$  recursief  
// uitvoer:  $A[0..n - 1]$  oplopend gesorteerd  
if  $n > 1$   
    copieer( $A[0 \dots \lfloor \frac{n}{2} \rfloor - 1]$ ,  $B[0 \dots \lfloor \frac{n}{2} \rfloor - 1]$ );  
    copieer( $A[\lfloor \frac{n}{2} \rfloor \dots n - 1]$ ,  $C[0 \dots \lceil \frac{n}{2} \rceil - 1]$ );  
    Mergesort( $B[0 \dots \lfloor \frac{n}{2} \rfloor - 1]$ );  
    Mergesort( $C[0 \dots \lceil \frac{n}{2} \rceil - 1]$ );  
    Merge( $B, C, A$ );  
fi .
```

```
Merge( $B[0 \dots p - 1]$ ,  $C[0 \dots q - 1]$ ,  $A[0, \dots p + q - 1]$ ) ::  
// voegt 2 gesorteerde arrays  $B$  en  $C$  samen tot 1 gesorteerd array  $A$   
   $i, j, k := 0$ ;  
  // voeg samen totdat een van de twee op is: ritsen  
  while  $i < p$  and  $j < q$  do  
    if  $B[i] \leq C[j]$  then  
       $A[k] := B[i]; k := k + 1; i := i + 1$ ;  
    else  
       $A[k] := C[j]; k := k + 1; j := j + 1$ ;  
  od  
  // en de rest  
  if  $i = p$  then  
    copieer  $C[j..q - 1]$  naar  $A[k..p + q - 1]$ ;  
  else  
    copieer  $B[i..q - 1]$  naar  $A[k..p + q - 1]$ ;  
  fi .
```

```
Quicksort( $A[l \dots r]$ )::  
// sorteert het (sub)array  $A[l..r]$  recursief  
// uitvoer:  $A[l..r]$  oplopend gesorteerd  
  if  $l < r$   
     $s := \text{Partitie}(A[l \dots r]);$  //  $s$  het splitspunt  
    Quicksort( $A[l \dots s - 1]$ );  
    Quicksort( $A[s + 1 \dots r]$ );  
  fi .
```

Partitie($A[l \dots r]$) ::

// partitioneert een (sub)array, met $A[l]$ als spil (pivot)

$p := A[l];$

$i := l; j := r + 1;$

repeat

repeat $i := i + 1;$ **until** $i > r$ **or** $A[i] \geq p;$

repeat $j := j - 1;$ **until** $A[j] \leq p;$

if $i < j$ **then**

Wissel($A[i], A[j]$);

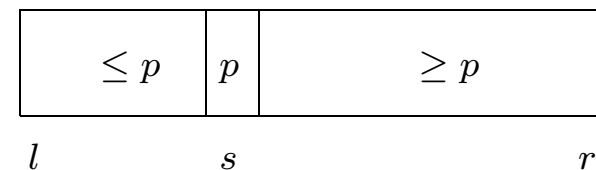
if

until $i \geq j;$

Wissel($A[l], A[j]$);

return $j;$.

Partitie



Probleem: reorganiseer de elementen van een gegeven array A zodanig dat alle negatieve elementen voorafgaan aan de positieve. Het algoritme moet lineair zijn en in situ.

```
i = 0; j = n-1;
while (i <= j) {
    if (A[i] < 0)
        i = i+1;
    else {
        wissel(A[i],A[j]);
        j = j-1;
    }
}
```

Variant (Dutch Flag): gegeven een array met 'R', 'W' en 'B'. Reorganiseer het array zodat v.l.n.r. eerst alle 'R', dan de 'W' en dan de 'B' staan. Zie Levitin, opgave 4.2.9.

Insertionsort($A[0 \dots m - 1]$)::

if $m > 1$

Insertionsort($A[0 \dots m - 2]$);

Voeg $A[m - 1]$ op de juiste plek in;

fi .

Invoegen van $A[m-1]$ in het reeds gesorteerde voorstuk $A[0] \dots A[m-2]$ door van rechts naar links $A[m-1]$ te vergelijken met $A[i]$. Deze recursieve versie komt overeen met de iteratieve versie zoals bij **Programmeermethoden** behandeld (zie ook Levitin):

$A[0] \leq A[1] \leq \dots \leq A[i] \leq A[i+1] \leq \dots \leq A[m-3] \leq A[m-2] || A[m-1] \dots$

kleiner of gelijk $A[m-1]$ \uparrow groter dan $A[m-1]$

hier invoegen

Mergesort:

- worst case complexiteit: $\Theta(n \log n)$
- extra geheugen: $O(n)$

Quicksort:

- worst case complexiteit: $\Theta(n^2)$ voor (o.a.) het reeds gesorteerde rijtje
- average case complexiteit: $\Theta(n \log n)$
- extra geheugen: in situ

Insertion sort:

- worst case/average case complexiteit: $\Theta(n^2)$
- extra geheugen: in situ

- **Werkcollege**

donderdag 22 maart 2007 in zaal 312

- **Opgaven:**

zie <http://www.liacs.nl/home/graaf/ALGO/algo2007.html>

- **Volgend college:**

vrijdag 23 maart 2007