

Zesde college algoritmiek

21 maart 2014

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/eisen, en nog wel tot een oplossing kan leiden
- 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

Exhaustive search. Genereer alle 2^{10} deelrijtjes van A volledig en controleer van elk daarvan of deze **stijgend** is en bepaal welke de **langste** van deze deelrijen is.

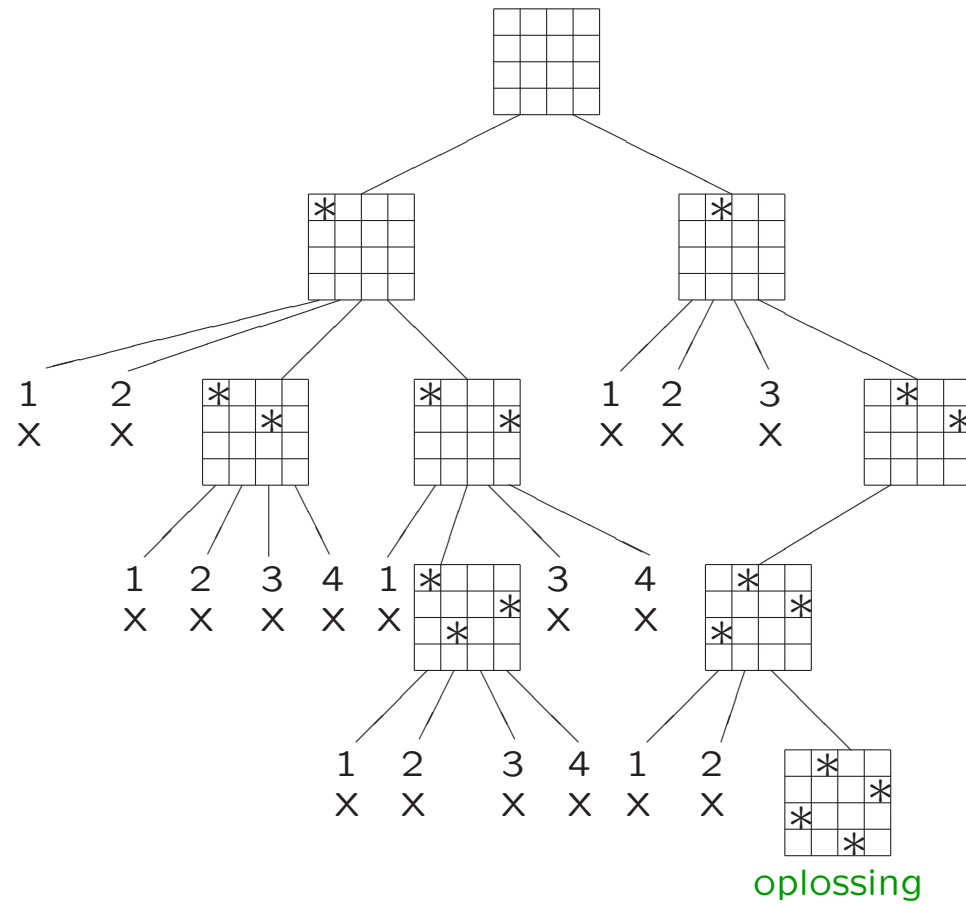
Backtracking. Bouw de deelrijtjes stap voor stap op (hoe?) en controleer na elke stap of het deelrijtje nog wel stijgend is. Zo niet, dan hoeft het rijtje niet meer uitgebreid te worden (het kan toch niets worden). Houd de lengte van het deelrijtje bij en vergelijk die met de lengte van de tot dusver gevonden langste deelrij.

Deze methode kan erg veel werk uitsparen. Bijvoorbeeld het deelrijtje 3, 1 is al niet stijgend, dus alle 2^8 deelrijtjes van A die met 3, 1 beginnen zeker ook niet. Deze hoeven bij backtracking dus niet allemaal te worden gegenereerd.

Om de *werking* van backtracking te *beschrijven* kunnen we een **state-space tree (toestand-actie-boom)** gebruiken.

- speciaal soort toestandsruimte
- toestand (=knoop) \iff deeloplossing
actie (=tak) \iff keuze uitbreiding deeloplossing
- blad \iff (kandidaat)oplossing
- pad van wortel naar blad \iff stap-voor-stap-constructie van (kandidaat)oplossing

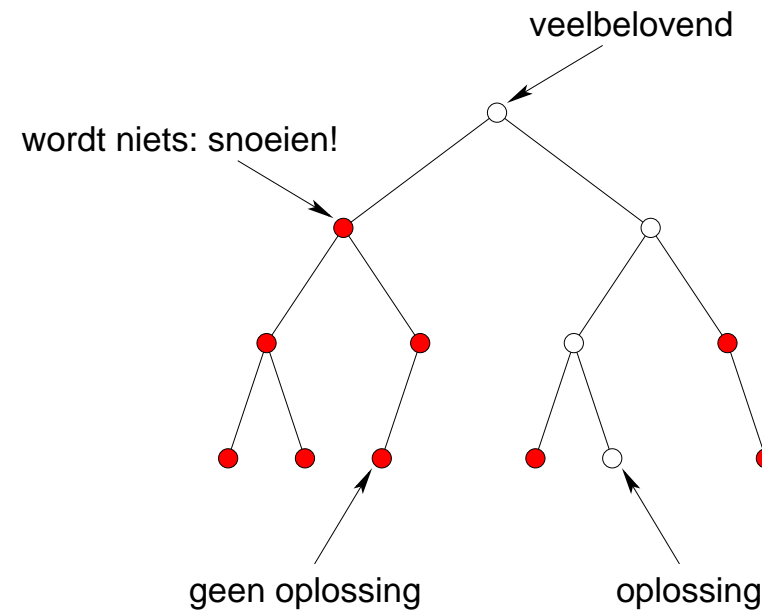
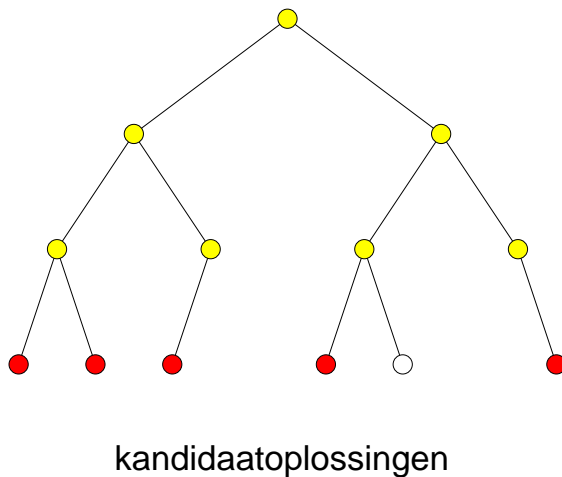
De boom beschrijft hoe een oplossing wordt opgebouwd (en uiteindelijk gevonden). De volledige state-space tree bevat in feite de hele stapsgewijze constructie van alle kandidaatoplossingen



x: deeloplossing niet verder uitbreiden, keuze herzien

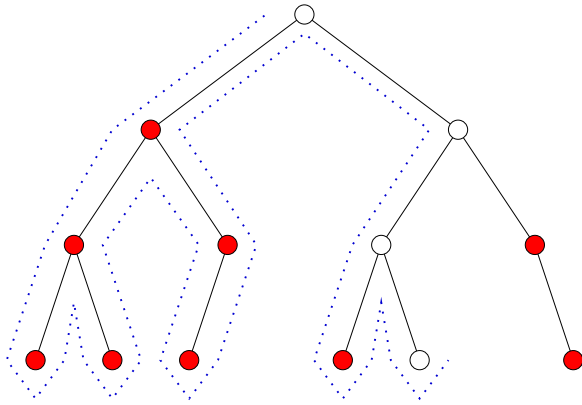
Exhaustive search (met gebruik van stap-voor-stap-constructie om kandidaatoplossingen te genereren) doorloopt de hele state-space tree en evalueert alle bladeren.

Backtracking stopt met het doorlopen van een subboom bij een knoop als de betreffende knoop nooit tot een oplossing kan leiden (en backtrackt dan naar de ouder van die knoop om van daaruit verder te zoeken). In de bijbehorende state-space tree die het backtracking algoritme beschrijft wordt zo'n niet-doorlopen subboom dan ook weggelaten.

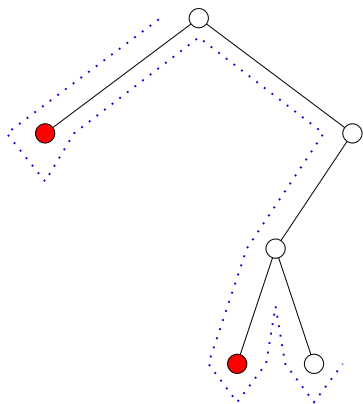


Exhaustive search:
de hele boom wordt bekeken (tot een goede oplossing is gevonden)

Backtracking: hele subbomen kunnen soms worden **gesnoeid**



Backtracking doorzoekt de state-space tree via **depth first search**.



Als het meezit wordt er flink gesnoeid.

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

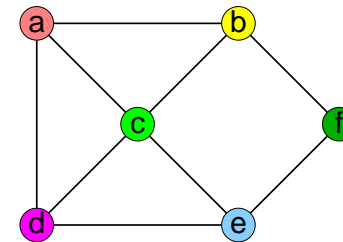
```
void permutaties(int n, int perm[ ], int hierzo) {
    int i;
    if (hierzo == n+1)
        drukaf(perm,n); // permutatie gevonden
    else {
        for (i=1; i<=n; i++) {
            perm[hierzo] = i;
            if (!aanwezig(perm, i, hierzo)) // test of i
                // al in perm[1] t/m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

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

Definitie: Een **Hamiltonkring** in een (ongerichte) graaf is een kring die elke knoop precies één keer aandoet.

Voorbeeld: a b f e c d (&) is een Hamiltonkring in nevenstaande graaf.

Echter a b c d e f is geen Hamiltonkring omdat er geen tak tussen f en a loopt.



Probleem: vind een Hamiltonkring in een gegeven ongerichte graaf.

(&) We kunnen deze kring ook aangeven met a b f e c d a. Echter bij een kring is de eindknoop hetzelfde als de beginknoop, dus kunnen we de eindknoop net zo goed weglaten: die ligt toch vast. Kortom: Hamiltonkring \leftrightarrow permutatie van de knopen.

Exhaustive search: genereer alle $(n - 1)!$ kandidaatoplossingen (permutaties van de knopen) en controleer daarna van elk of het een Hamiltonkring voorstelt.

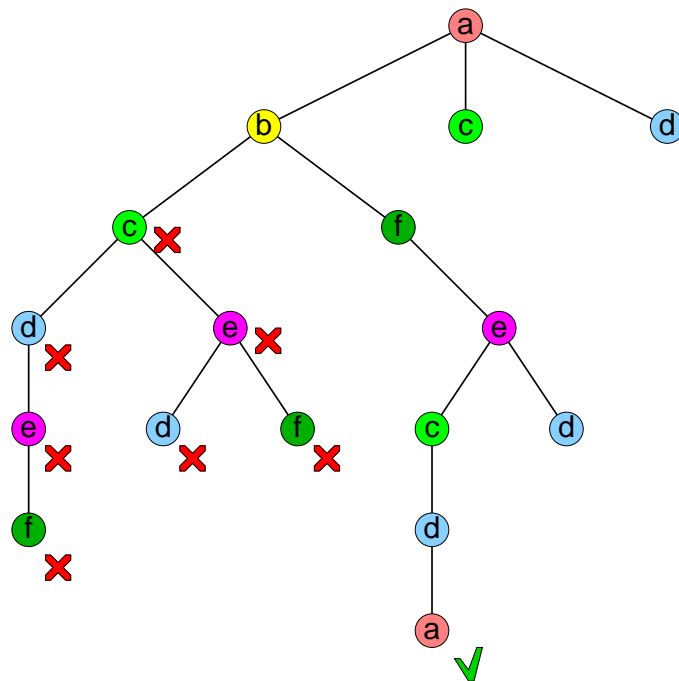
Backtracking: genereer de mogelijke Hamiltonkringen stap voor stap* en controleer tijdens de constructie al of de deeloplossing wel aan de restricties† voldoet.

Kies in elke uitbreidingsstap steeds de eerstvolgende **buurknoop** (in een of andere volgorde) en controleer hiervan of deze nog niet geweest is in het reeds geconstrueerde deelpad. Blijkt die keuze toch (hier of verderop in de constructie) op niets uit te lopen, kies dan de volgende buurknoop. Als er geen buurknopen meer zijn kan het deelpad blijkbaar niet meer uitgebreid worden en moet je je vorige keuze herzien.

*hier: tak voor tak of knoop voor knoop

†alle knopen verschillend; tak tussen opeenvolgende knopen

Een beschrijving van de werking van het backtracking-algoritme voor het vinden van een Hamiltonkring in de voorbeeldgraaf wordt gegeven door de volgende state space tree:



- . breid het pad telkens met één knoop uit (hier: keuzes van de buren in alfabetische volgorde)
- . uitbreidingen met knopen die al eerder voorkwamen in het pad zijn niet weergegeven
- . in de knopen met een rood kruis backtrackt het algoritme zodra blijkt dat die niet tot een oplossing leiden

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.

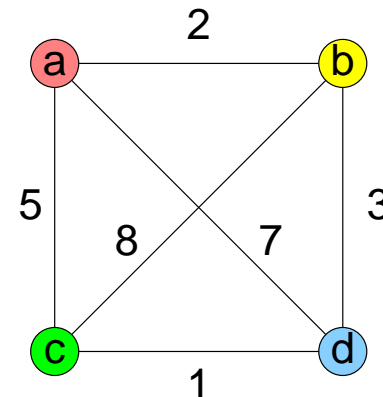
Ofwel: vind de/een kortste Hamiltonkring in een samenhangende gewogen (complete) graaf.

Voorbeeld:

minimale route:

a b d c (ofwel a b d c a)

(of a c d b (ofwel 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) verstandig uit te breiden:
- 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**

Probleem:

Gegeven een verzameling $S = \{s_1, s_2, \dots, s_n\}$ van positieve (> 0) gehele getallen en een geheel getal d . Laat S oplopend gesorteerd zijn. Vind een deelverzameling (of alle deelverzamelingen) van S waarvan de som der getallen gelijk is aan d .

Voorbeelden:

$S = \{1, 2, 5, 6, 8\}$ en $d = 9$. Er zijn twee oplossingen, namelijk $\{1, 2, 6\}$ en $\{1, 8\}$.

$S = \{3, 5, 6, 7\}$ en $d = 15$. Er is één oplossing: $\{3, 5, 7\}$.

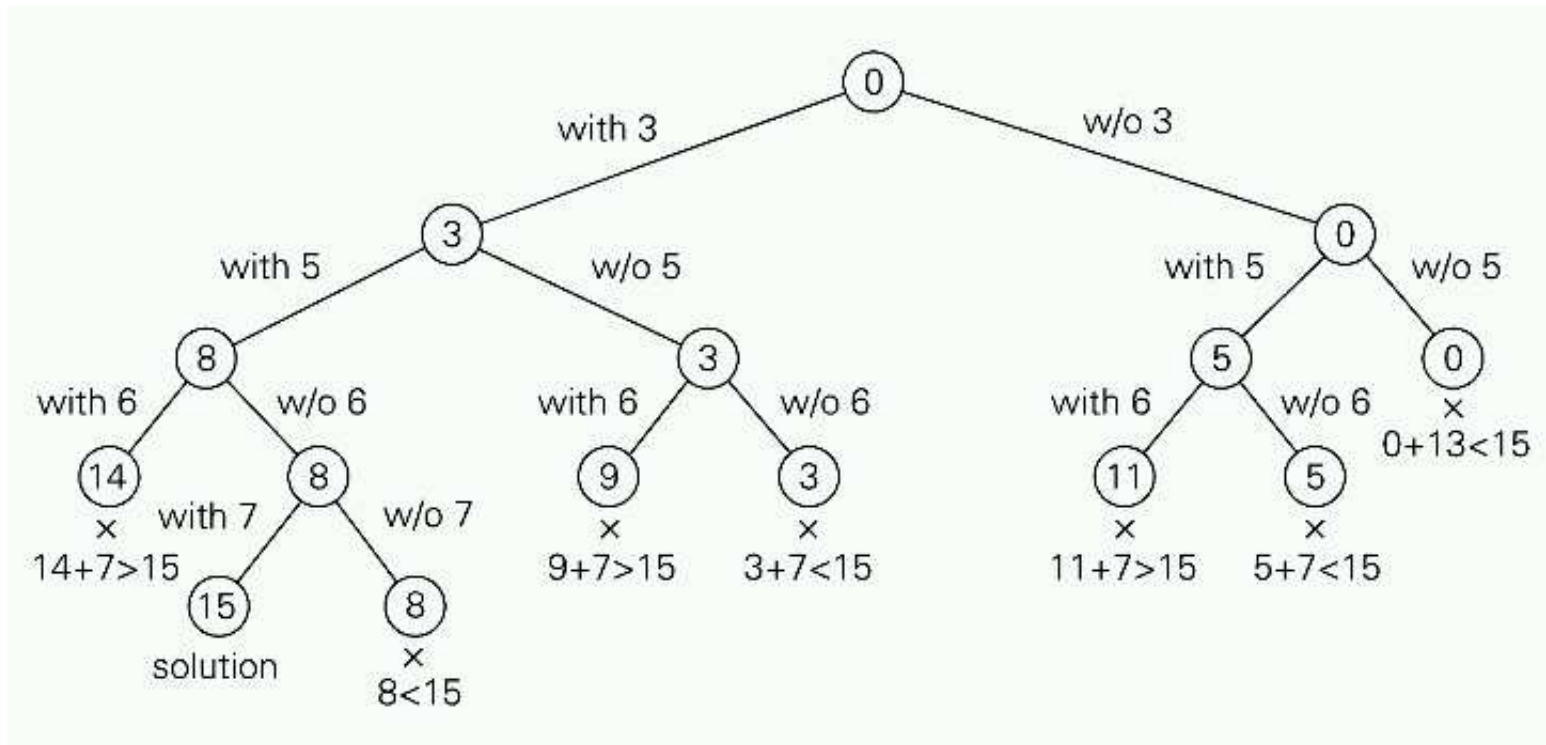
Exhaustive search: genereer alle 2^n deelverzamelingen van S en controleer of hun som gelijk is aan d .

De stap-voor-stap constructie van deeloplossingen doen we (bijvoorbeeld) zo: gegeven een deeloplossing (= een veelbelovende deelverzameling van $\{s_1, s_2, \dots, s_i\}$), dan zijn er twee mogelijke vervolgstappen: óf s_{i+1} wordt toegevoegd, óf s_{i+1} wordt niet toegevoegd.

Backtracking bekijkt ook alle deelverzamelingen, maar hoeft ze niet allemaal volledig te genereren. Een (veelbelovende) deelverzameling van $\{s_1, s_2, \dots, s_i\}$ met som s' hoeft niet verder uitgebreid te worden als $s' + s_{i+1} > d$ (*), of als $s' + \sum_{j=i+1}^n s_j < d$.

Opmerking:

Als (*) geldt levert elke uitbreiding van s' , met welke s_j ($j \geq i+1$) dan ook, een te grote totaalsom op. Dus: herzie je vorige keuze. Hier is gebruikt dat S olopend gesorteerd is.



De state-space tree voor toepassing van backtracking op probleeminstantie $S = \{3, 5, 6, 7\}$ en $d = 15$ (waarbij alle oplossingen gezocht worden). De knopen stellen deeloplossingen voor. Het getal in een knoop is de som van de s_j uit de corresponderende deelverzameling. De ongelijkheid onder een blad geeft aan waarom daar backtracking plaatsvindt.

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 of 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

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
    
```



```

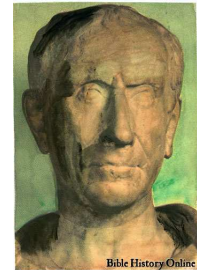
XXXXXXXXXXXXX
X   ***X   X
X  X*X*XXX  XX
XXX*X*X***OX
X***X***X*XX
X*XXXXXXXX*XX
X*****X*OX
XXXXOX*X*XOX
S**XXX*X*XOX
XX*****X*XOX
X   X  XOX*XOX
XXXXXXXXXXEXXX
    
```



```
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:
    return false;      // geen rechtstreeks pad via deze (x,y)
  }
}
```

Laten oplossingen van een bepaald probleem van de vorm $(X[1], X[2], \dots, X[m])$ 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
```

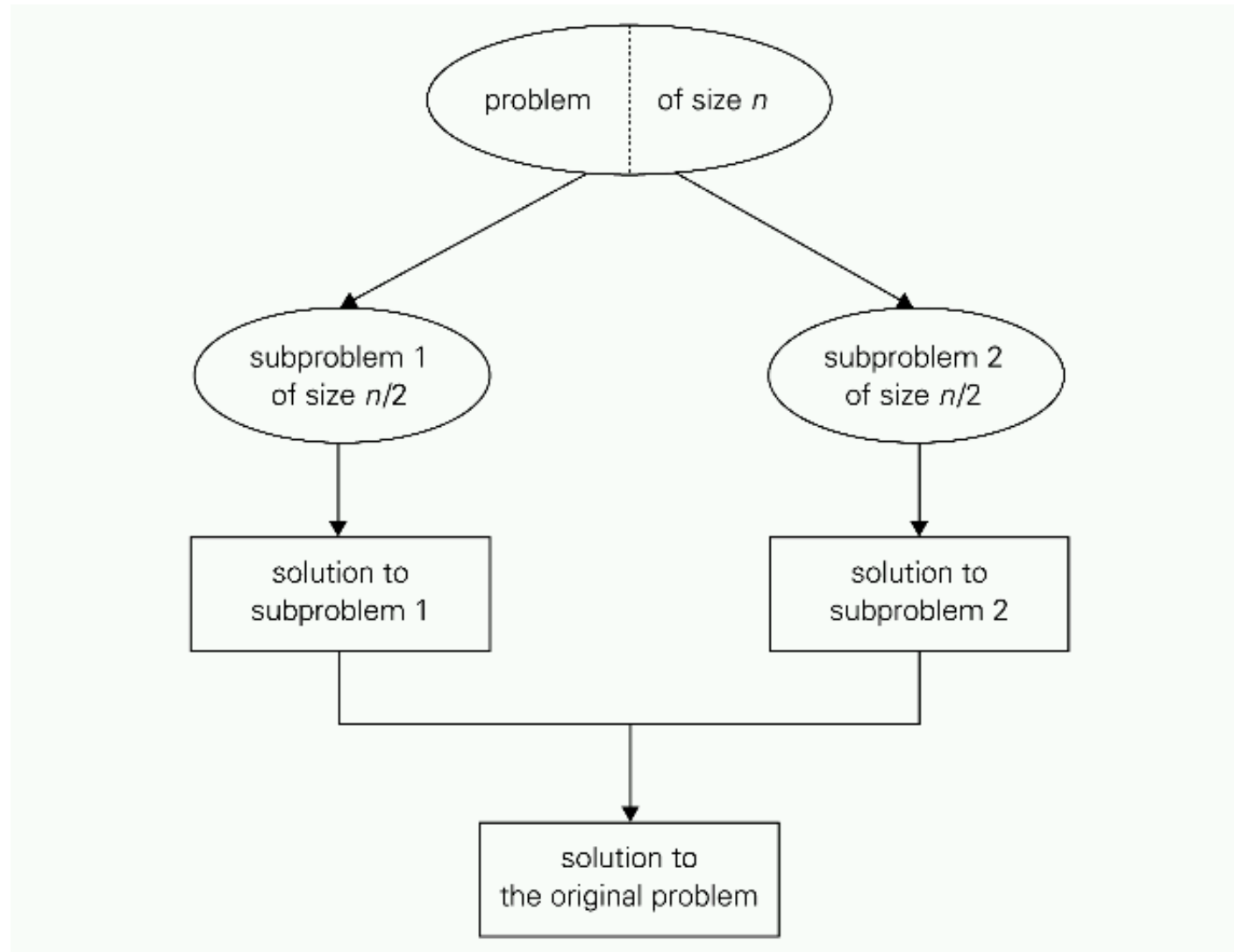


Divide and Conquer

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 (of meer) 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

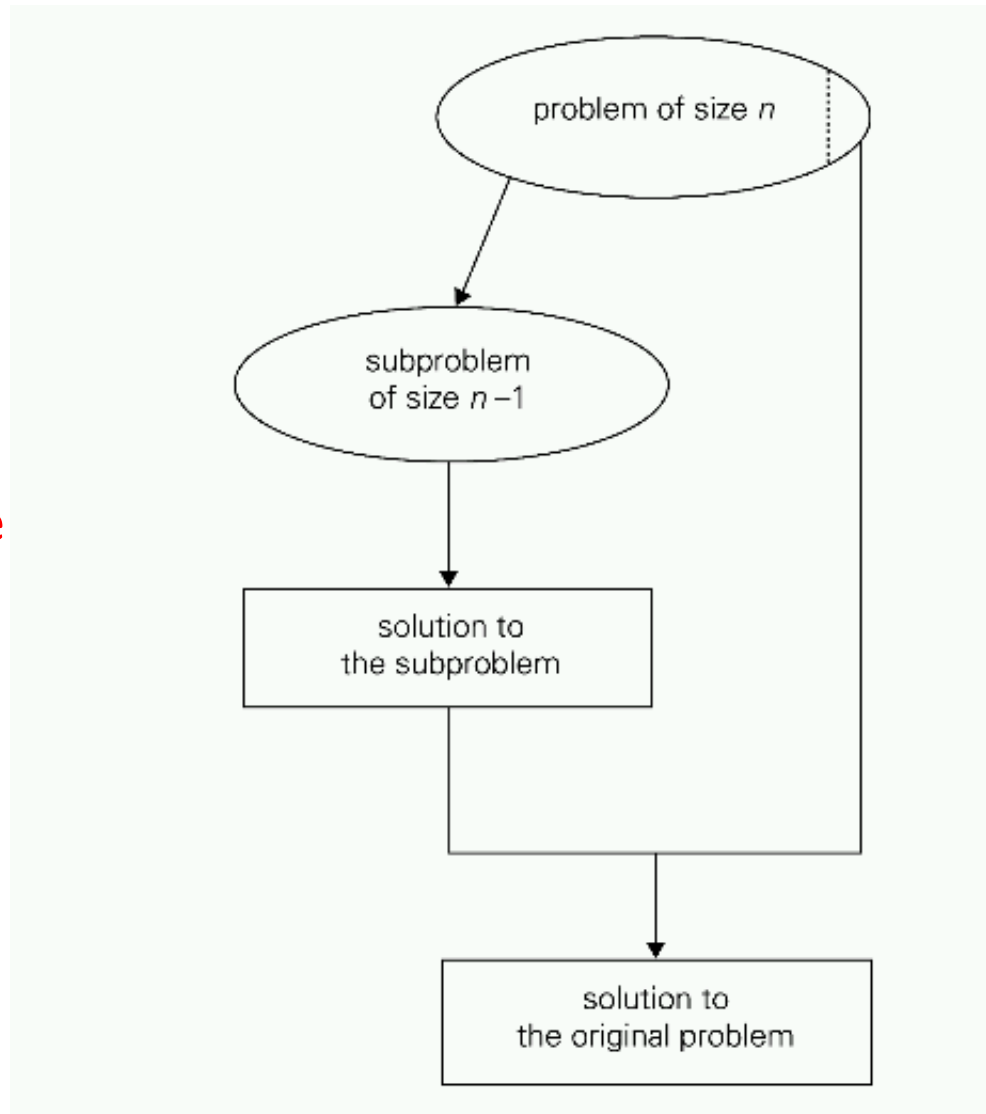
In het boek wordt onderscheid gemaakt tussen:

Decrease by one

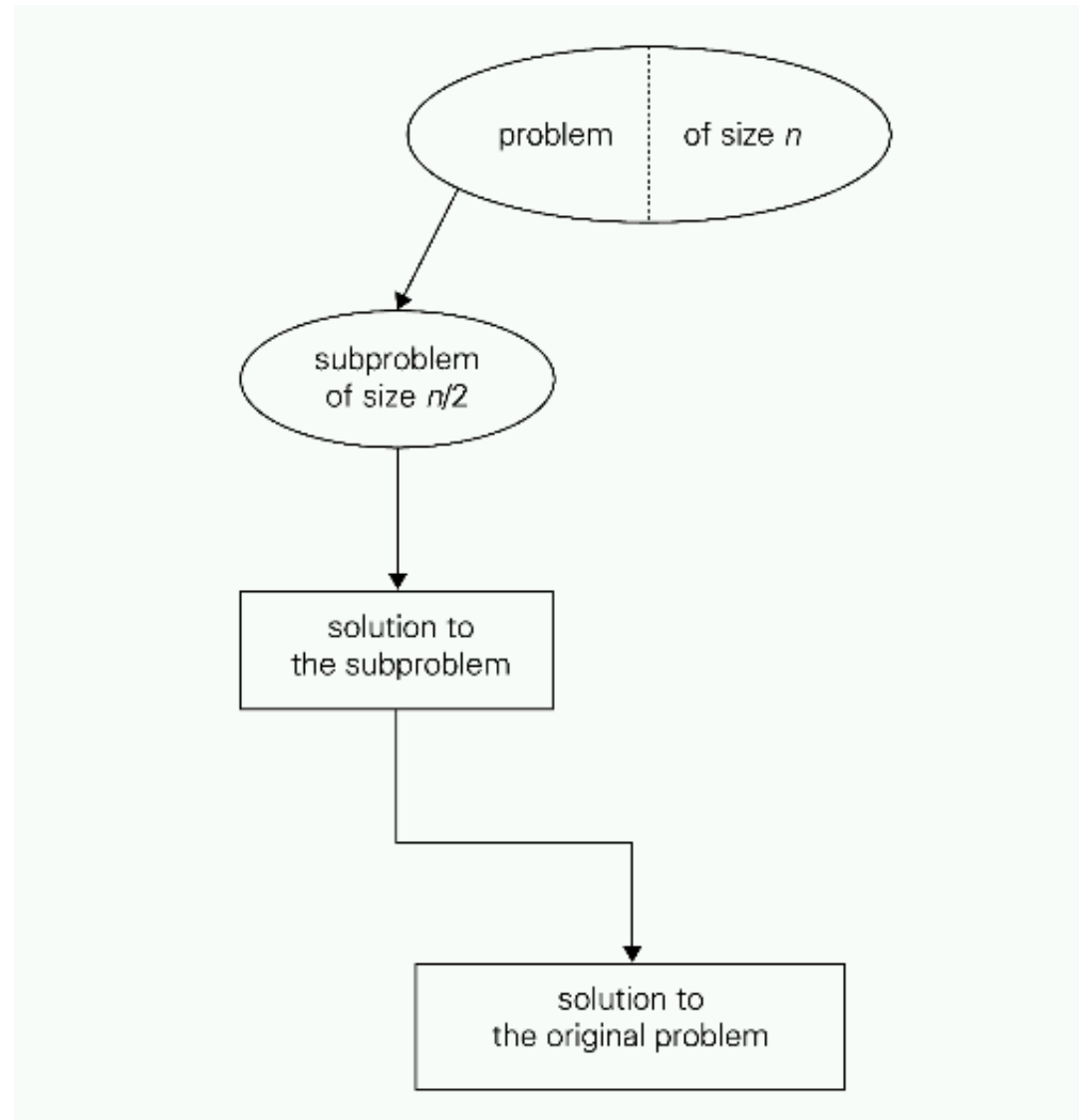
Decrease by a constant factor

Variable-size decrease

Decrease
by one



Decrease by a
constant factor
(decrease by half)



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;

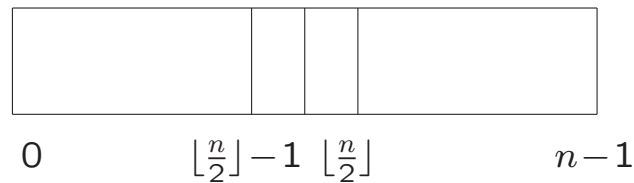
Sorteer(linkerrij);

Sorteer(rechterrij);

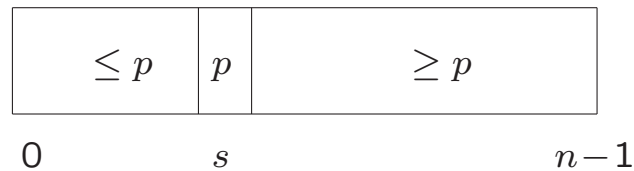
Combineer linkerrij en rechterrij;

fi .

Divide and conquer

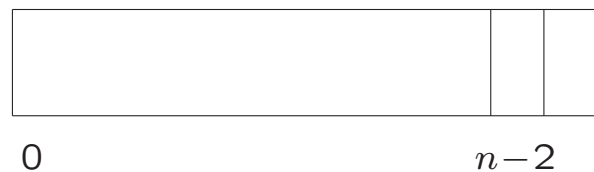


Mergesort



Quicksort

Decrease and conquer (decrease by one)



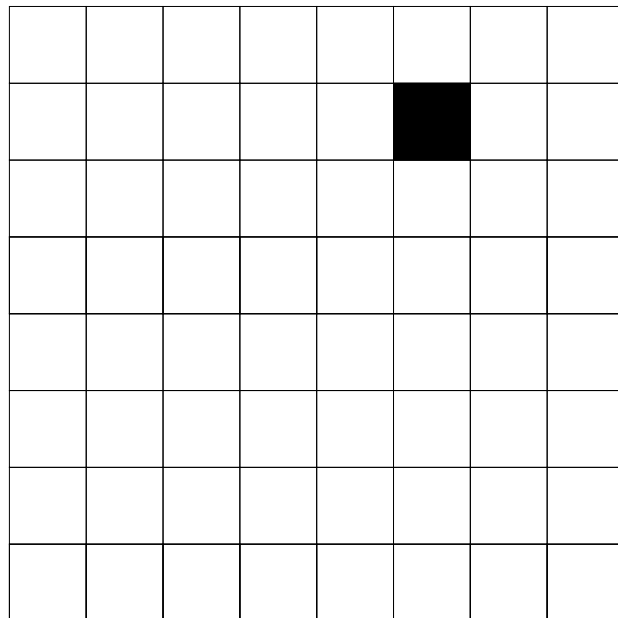
Insertion sort

```
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 .
```

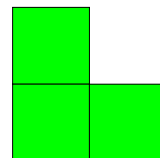
```
Merge( $B[0 \dots p - 1]$ ,  $C[0 \dots q - 1]$ ,  $A[0 \dots p + q - 1]$ ) ::  
// voegt 2 gesorteerde arrays  $B$  en  $C$  samen tot 1 gesorteerd array  $A$   
   $i, j, k := 0$ ;  
  // voeg samen totdat een van de twee op is: ritsen  
  while  $i < p$  and  $j < q$  do  
    if  $B[i] \leq C[j]$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

Nog een leuk voorbeeld van de methode Divide and Conquer is de Tromino puzzel, Levitin 5.1.11.

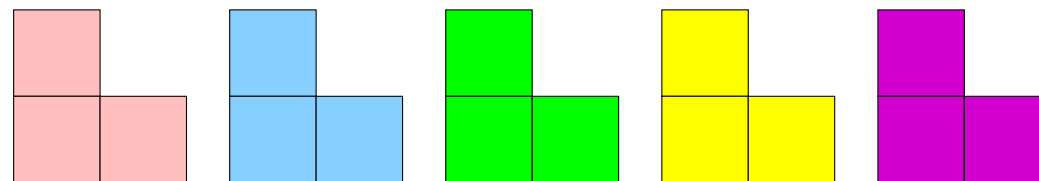
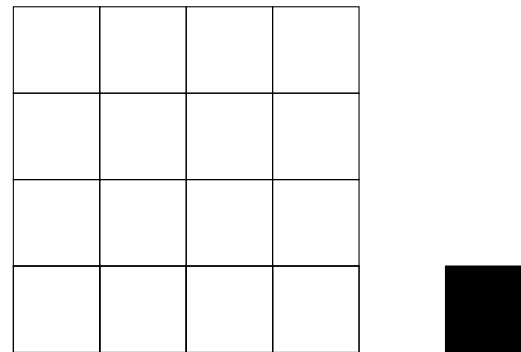
Bedek een $2^n \times 2^n$ schaakbord –waaruit één vakje mist– met tromino's.



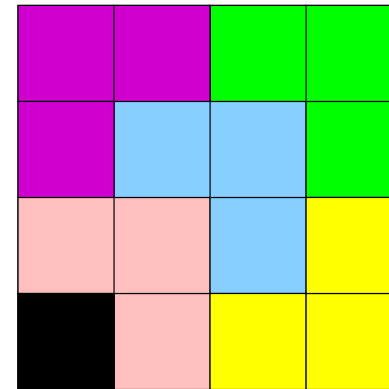
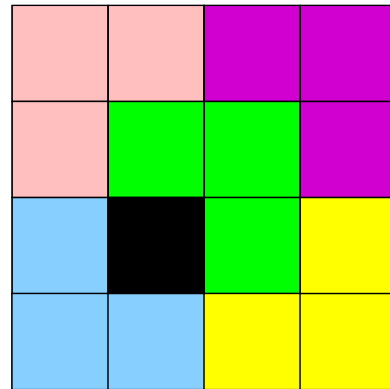
Het 8x8 geval



Het 4x4 geval: gegeven een 4x4 bord waaruit één vakje mist. Bedek het bord volledig (behalve het missende vakje), dus met 5 tromino's.

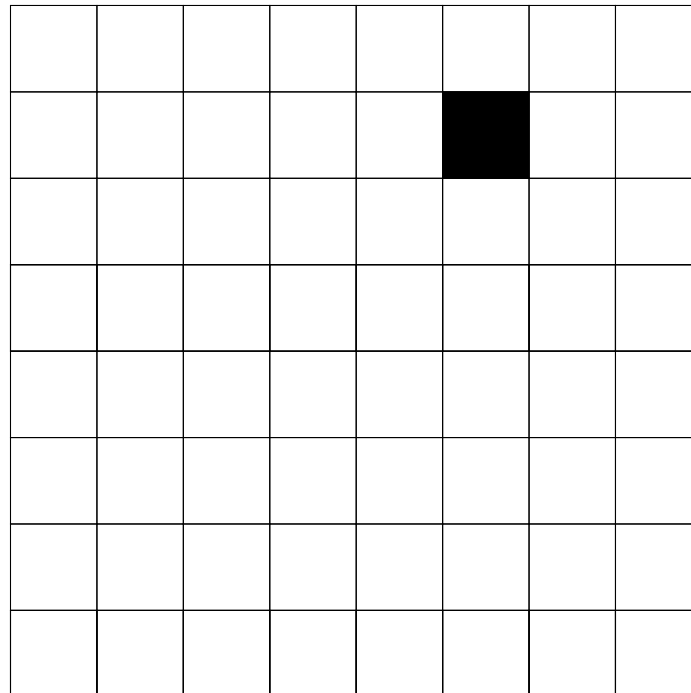


Het 4x4 geval: twee probleeminstanties met oplossing (= bedekking)

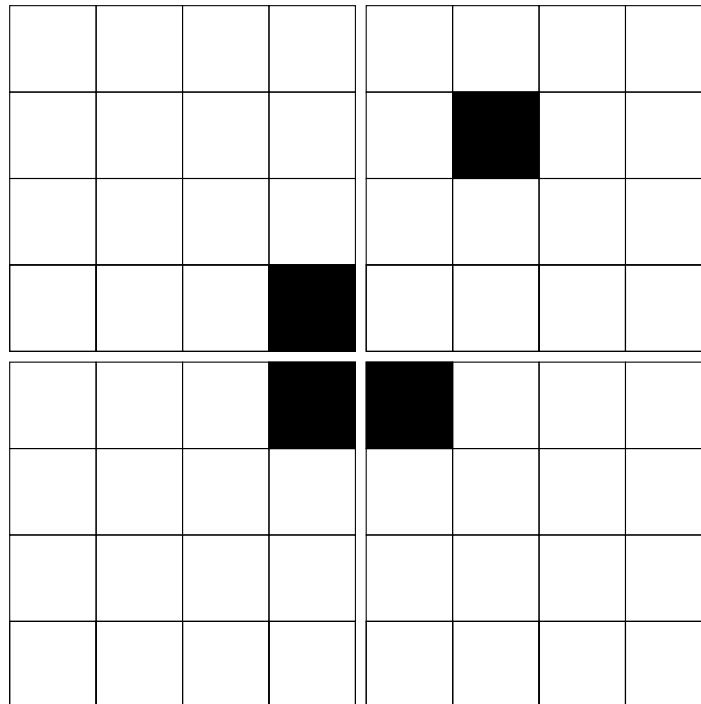


Het 8x8 geval: hoe vinden we een (de?) bedekking met tromino's?

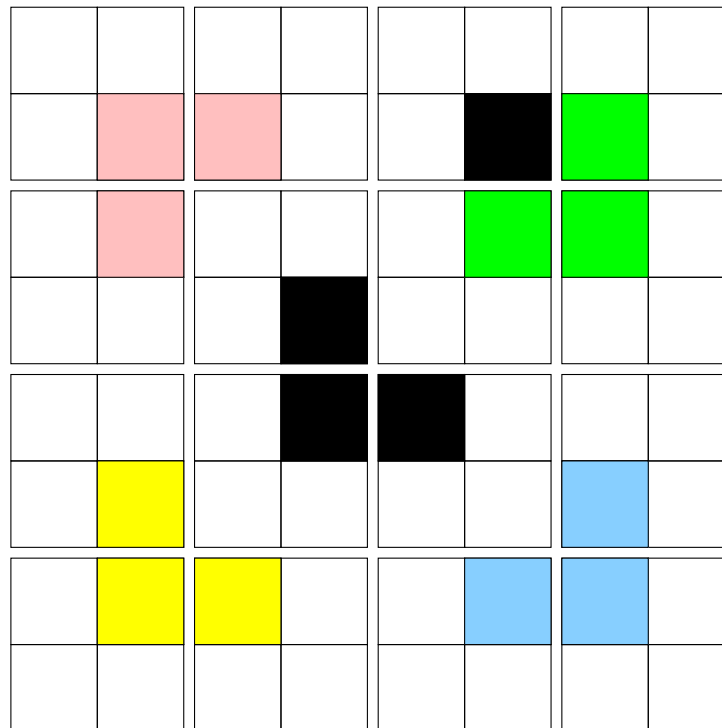
Bijvoorbeeld voor dit bord:



Leg een tromino in het midden, zodat in elk kwart één stuk mist of bedekt is. Het probleem is nu teruggebracht tot vier keer hetzelfde probleem, maar dan voor 4x4 borden.



Doe weer hetzelfde (recursie!) met de 4x4 borden.



Dit **divide and conquer** algoritme kun je op elk $2^n \times 2^n$ bord toepassen.

- **Lezen/leren bij dit college:**
Paragraaf 12.1, 5 inleiding, 5.1, 4 inleiding
- **Tweede programmeeropgave:** verschijnt dit weekend op <http://www.liacs.nl/home/graaf/ALGO/>
- **Werkcollege:**
donderdag 27 maart 2014, zaal 306/308, 303:
over de tweede programmeeropdracht
- **Volgend college:**
vrijdag 28 maart 2014