Vijfde college algoritmiek

3 maart 2020

Brute Force

Exhaustive Search

Backtracking

Opdracht 1

- partner?
- deadline: 20 maart
- vragenuur: vanmiddag, 16.15 uur (voor zover ruimte)

Gegeven een patroon (= string van m karakters) en een tekst (= string van $n \geq m$ karakters). **Gevraagd** de index van de beginpositie in de tekst waar het patroon voorkomt.

Brute force algoritme: patroon v.l.n.r. langs de tekst schuiven en steeds de overeenkomstige karakters uit tekst en patroon vergelijken

```
for i := 0 to n - m do
    j := 0;
    while j < m and patroon[j] = tekst[i + j] do
        j := j + 1;
    od
    if j = m then
        return i;
    fi
od
return -1; // geen match gevonden
```

De werking van het algoritme geïllustreerd aan de hand van het volgende voorbeeld:

```
N O B O D Y - N O T I C E D - H I M
N O T
  N O T
    N O T
      N O T
        N O T
          N O T
            N O T
              N O T
```

Het aantal vergelijkingen dat dit algoritme doet hangt af van de tekst en het patroon. In de worst case worden $m * (n - m + 1)$ vergelijkingen gedaan. Dit komt voor wanneer in elke i-stap het patroon helemaal (dus m vergelijkingen) vergeleken wordt met de tekst. De complexiteit van het algoritme is dus O($n * m$).

Opgave: geef een tekst en een patroon waarvoor het algoritme $m * (n - m + 1)$ vergelijkingen doet.

Opmerking: het kan beter (Boyer-Moore, Knuth-Morris-Prattt), maar voor "gewone-taal" teksten is het algoritme zo slecht nog niet.

Opmerking 2: `string::find (...)` gebruikt brute force

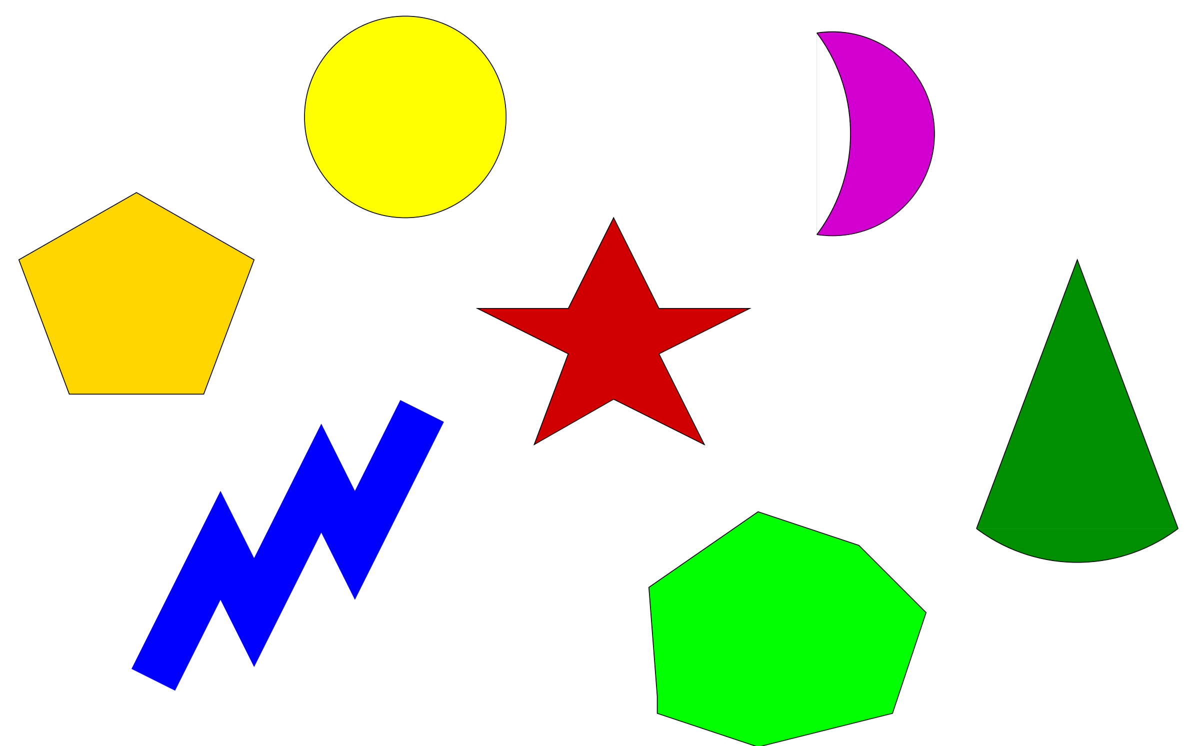
Gegeven n punten $p_1 = (x_1, y_1), \ldots, p_n = (x_n, y_n)$.
Gevraagd het/een tweetal punten dat het dichtst bij elkaar ligt. Afstands-maat: $\mathrm{d}(p_i, p_j) = \sqrt{(x_i - x_j)^2 + (y_i - y_j)^2}$.

Brute force algoritme: alle paren (p_i, p_j) (met $i < j$) aflopen en hun onderlinge afstanden $\mathrm{d}(p_i, p_j)$ vergelijken.

```
dmin := ∞;
for i := 1 to n − 1 do
    for j := i + 1 to n do
        d := (x_i − x_j)^2 + (y_i − y_j)^2;
        if d < dmin
            dmin := d; k := i; l := j;
        fi // (p_k, p_l) voorlopig closest pair
    od
od
```

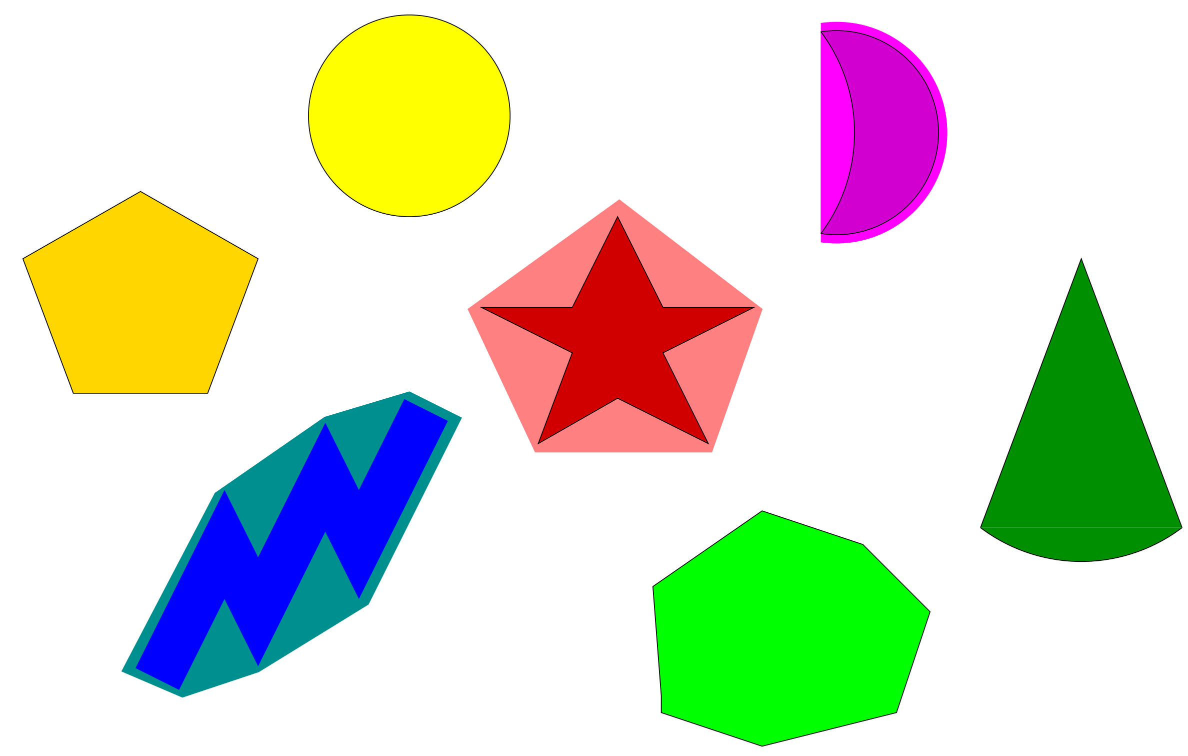
Complexiteit: $\frac{1}{2}n(n-1) = \Theta(n^2)$

Een verzameling punten in het platte vlak heet convex als voor elk tweetal punten uit die verzameling geldt dat het verbindend lijnstuk ook weer in die verzameling ligt.



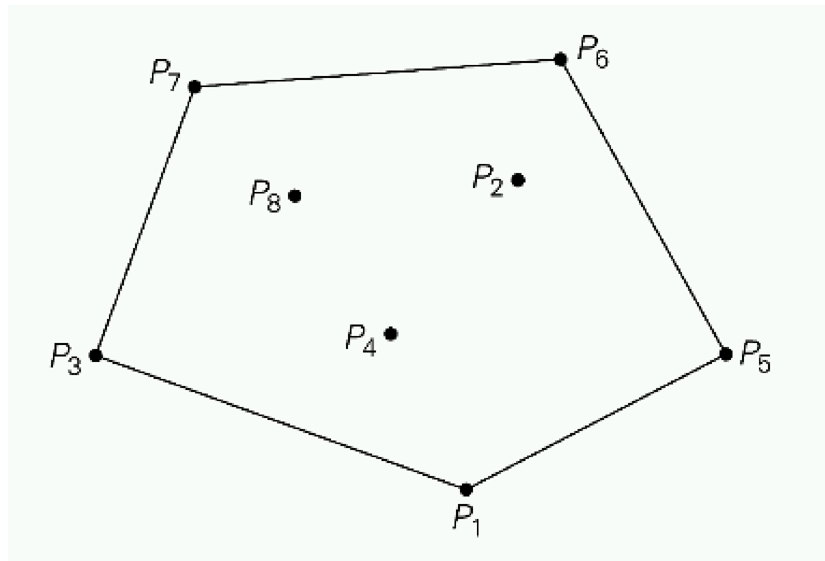
Convexe en niet-convexe vormen

De convex hull (convex omhulsel) van een verzameling S van punten in $\mathbf{R}^2$ is de kleinste convexe verzameling die S bevat.



Convexe omhulsels

Stelling: de convex hull van een verzameling S van $n > 2$ punten in $\mathbf{R}^2$ (niet alle op één lijn) is een convexe veelhoek (polygoon) met als hoekpunten enkele punten uit S.



De convex hull van de verzameling $\{P_1, P_2, \ldots, P_8\}$ is de convexe veelhoek met hoekpunten P_1, P_5, P_6, P_7 en P_3

We baseren ons brute force algoritme op de volgende observatie: een lijnstuk P_iP_j maakt deel uit van de rand van de convex hull van $\{P_1, P_2, \ldots, P_8\}$ d.e.s.d.a. alle andere punten van de verzameling aan een en dezelfde kant van de lijn door P_i en P_j liggen.

Brute force: Ga *voor elk tweetal* punten $P_i = (x_i, y_i)$ en $P_j = (x_j, y_j)$ na of *alle andere punten* aan dezelfde kant van de lijn $(y_j - y_i)x + (x_i - x_j)y = x_iy_j - y_ix_j$ liggen. Zo ja, dan is P_iP_j dus deel van de convex hull.

Complexiteit: $O(n^3)$

Opmerking: het kan veel beter, namelijk $O(n \lg n)$

Brute force:

- **Voordelen**:
  - algemeen toepasbaar
  - eenvoudig
  - levert voor een aantal belangrijke problemen (zoeken, patroonherkenning) een zeer behoorlijk algoritme op

- **Nadelen**:
  - levert meestal geen efficiënt algoritme op
  - soms onacceptabel langzaam

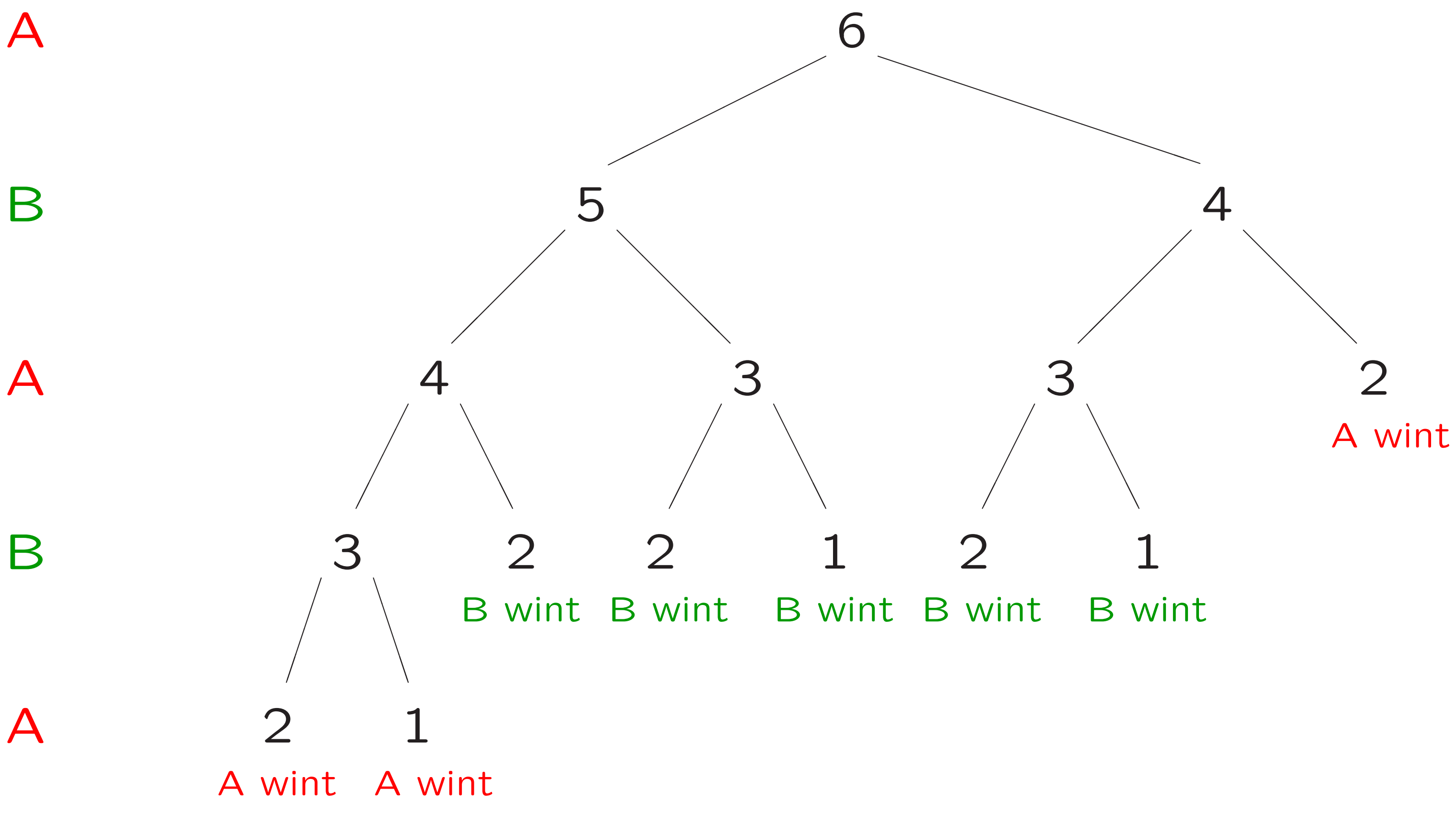
Exhaustive search: brute force benadering voor problemen die te maken hebben met het vinden van een element met een speciale eigenschap binnen een verzameling van bijv. permutaties of deelverzamelingen of toestanden of ...

Methode:

. construeer op een systematische manier alle kandidaatoplossingen

. evalueer elk van deze mogelijke oplossingen

. retourneer een/de kandidaatoplossing met de gevraagde eigenschap (als die bestaat)



Exhaustive search: doorloop (*als het ware*) de hele spelboom om te bepalen of een stand winnend is. Alle toestanden/alle spelverlopen worden zo bekeken. Je kunt stoppen zodra je een winnende zet gevonden hebt.

Belangrijke observatie:

> een stand is winnend voor degene die aan de beurt is, dan en slechts dan als ten minste één van zijn directe vervolgstanden niet winnend is voor de tegenstander

Met terugzetten:

```
winnend(stand)::

    if eindstand(stand) then
        //  makkelijk; bijv return false;
    else
        for alle mogelijke zetten i do
            doezet(stand,i);
            if not winnend(stand) then
                undoezet(stand,i);
                return true;
            undoezet(stand,i);
        od
        return false;
    fi
```

Exhaustive search: brute force benadering voor problemen die te maken hebben met het vinden van een element met een speciale eigenschap binnen een verzameling van bijv. permutaties of deelverzamelingen of toestanden of ...

Methode:

. construeer op een systematische manier alle kandidaatoplossingen, bijvoorbeeld alle permutaties van de getallen 1 t/m n

. evalueer elk van deze mogelijke oplossingen

. retourneer een/de kandidaatoplossing met de gevraagde eigenschap (als die bestaat) (*)

(*) soms, zoals bij optimalisatieproblemen, *moet* je daartoe alle kandidaatoplossingen gezien hebben

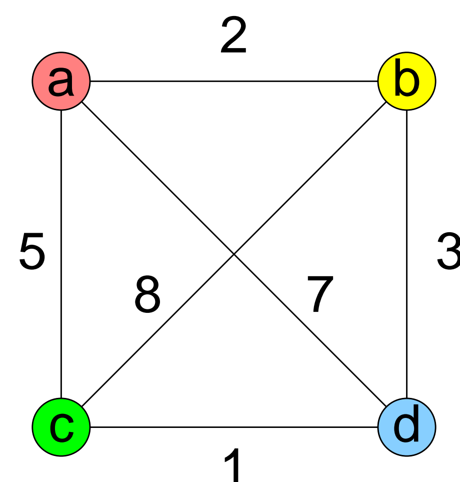
Traveling Salesman Problem (handelsreizigersprobleem)

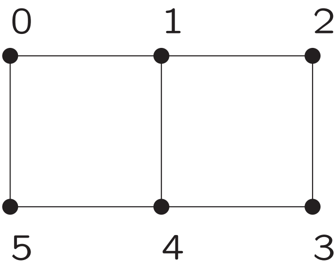
Gegeven n steden waarvan alle onderlinge afstanden bekend zijn.

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 (volledige) graaf.

Voorbeeld:

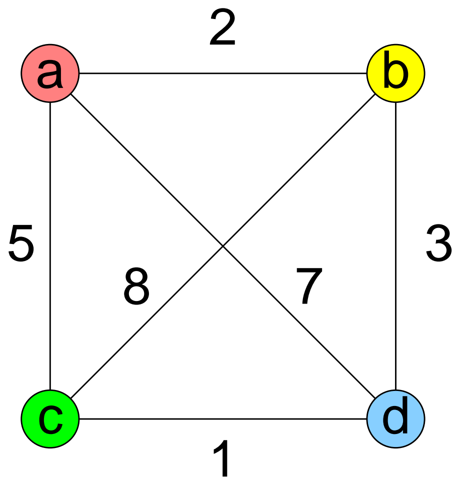




1.

$V = \{0, 1, 2, 3, 4, 5\}$;

$E = \{(0, 1), (1, 2), (2, 3), (3, 4), (4, 5), (5, 0), (4, 1)\}$



Route	Lengte
a → b → c → d → a	$2+8+1+7=18$
a → b → d → c → a	$2+3+1+5=11$
a → c → b → d → a	$5+8+3+7=23$
a → c → d → b → a	$5+1+3+2=11$
a → d → b → c → a	$7+3+8+5=23$
a → d → c → b → a	$7+1+8+2=18$

Complexiteit: $\Omega((n-1)!)$,
immers alle $(n-1)!$ mogelijke Hamiltonkringen worden bekeken.

Code voor TSP met ES, 1

```
void bepaalroute(int n, int pos, int route[], int &best) {
  int lengte;

  ...
  ...
  ...
  ...
  ...
  ...
  ...
  ...
  ...
  ...
  ...
  ...
  ...
}

int main ( ) {
  int best = MAXINT;

  route[0] = 1;
  bepaalroute (n, 1, route, best);
  cout << "Kortste afstand: " << best << endl;
}
```

Code voor TSP met ES, 2

```
void bepaalroute(int n, int pos, int route[], int &best) {
  int lengte;

  if (pos == n) {  // route afsluiten (‘eindstand’)
    route[pos] = route[0];
    lengte = berekenlengte (n, route)
    if (lengte < best)
      best = lengte;
  }
  else {  // pos < n
    ...
    ...
    ...
    ...
    ...
  }
}

int main ( ) {
  int best = MAXINT;

  route[0] = 1;
  bepaalroute (n, 1, route, best);
  cout << "Kortste afstand: " << best << endl;
}
```

Code voor TSP met ES, 3

```
void bepaalroute(int n, int pos, int route[], int &best) {
  int lengte;

  if (pos == n) {  // route afsluiten (‘eindstand’)
    route[pos] = route[0];
    lengte = berekenlengte (n, route)
    if (lengte < best)
      best = lengte;
  }
  else {  // pos < n
    for (int i=1; i<=n; i++)  // ‘for alle mogelijke zetten’
      if (‘i nog niet in route’) {
        route [pos] = i;
        bepaalroute (n, pos+1, route, best);
      }
  }
}

int main ( ) {
  int best = MAXINT;

  route[0] = 1;
  bepaalroute (n, 1, route, best);
  cout << "Kortste afstand: " << best << endl;
}
```

Code voor TSP met ES, 4

```
void bepaalroute(int n, int pos, int route[], bool gehad[], int &best) {
  int lengte;
  if (pos == n) {  // route afsluiten (‘eindstand’)
    route[pos] = route[0];
    lengte = berekenlengte (n, route)
    if (lengte < best)
      best = lengte;
  }
  else {  // pos < n
    for (int i=1; i<=n; i++)  // ‘for alle mogelijke zetten’
      if (! gehad[i]) {
        route [pos] = i;      gehad[i] = true;
        bepaalroute (n, pos+1, route, gehad, best);
        gehad[i] = false;  // ‘undoezet’
      }
  }
}

int main ( ) {
  bool gehad[n+1];  // TODO: false initialiseren
  int best = MAXINT;
  route[0] = 1;      gehad[1] = true;
  bepaalroute (n, 1, route, gehad, best);
  cout << "Kortste afstand: " << best << endl;
}
```

N	10	50	100	300	1000
$\log_2 N$	3	5	6	8	9
$5N$	50	250	500	1500	5000
$N \cdot \log_2 N$	33	282	665	2469	9966
N^2	100	2500	10.000	90.000	7 cijfers
N^3	1000	125.000	7 cijfers	8 cijfers	10 cijfers
2^N	1024	16 cijfers	31 cijfers	91 cijfers	302 cijfers
$N!$	7 cijfers	65 cijfers	161 cijfers	623 cijfers	onvoorstelbaar
N^N	11 cijfers	85 cijfers	201 cijfers	744 cijfers	onvoorstelbaar

Ter vergelijking:

het aantal protonen in het heelal is een getal met 79 cijfers
het aantal microseconden sinds de Big Bang heeft 24 cijfers

Knapzakprobleem

Gegeven n objecten, met gewicht $w_1, \ldots, w_n$ en waarde $v_1, \ldots, v_n$, 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

deelverzameling	gewicht	waarde
$\emptyset$	0	0
$\{1\}$	8	42
$\{2\}$	3	14
$\{3\}$	4	40
$\{4\}$	5	27
$\{1,2\}$	11	56
$\{1,3\}$	12	82
$\{1,4\}$	13	te zwaar
$\{2,3\}$	7	54
$\{2,4\}$	8	41
$\{3,4\}$	9	67
$\{1,2,3\}$	15	te zwaar
$\{1,2,4\}$	16	te zwaar
$\{1,3,4\}$	17	te zwaar
$\{2,3,4\}$	12	81
$\{1,2,3,4\}$	20	te zwaar

Complexiteit: $\Omega(2^n)$,
immers alle 2^n deelverzamelingen van n objecten worden bekeken.

Hoe? Zien we later: stap voor stap opbouwen

- steeds alle mogelijkheden proberen voor het eerstvolgende object
- steeds het volgende object wel-of-niet kiezen

Assignmentproblem (toewijzingsprobleem)

Gegeven n personen en n taken (jobs). Persoon i kan taak j doen voor `kosten`$[i][j]$ euro.

Gevraagd: de/een toewijzing van de personen aan de jobs (één persoon per job en één job per persoon) met minimale kosten.

Voorbeeld:

	job 1	job 2	job 3	job 4
Anna	9	2	7	8
Bob	6	4	3	7
Carla	5	8	1	8
David	7	6	9	4

$n = 4$

	job 1	job 2	job 3	job 4
Anna	9	2	7	8
Bob	6	4	3	7
Carla	5	8	1	8
David	7	6	9	4

$n = 4$

```
1,2,3,4 -> 9+4+1+4 = 18
1,2,4,3 -> 9+4+8+9 = 30
1,3,2,4 -> 9+3+8+4 = 24
1,3,4,2 -> 9+3+8+6 = 26
1,4,2,3 -> 9+7+8+9 = 33
1,4,3,2 -> 9+7+1+6 = 23
2,1,3,4 -> 2+6+1+4 = 13
2,1,4,3 -> 2+6+8+9 = 25
2,3,1,4 -> ..
2,3,4,1 -> ..
2,4,1,3 -> ..
2,4,3,1 -> ..
3,1,2,4 -> ..
3,1,4,2 -> ..
3,2,1,4 -> ..
3,2,4,1 -> ..
3,4,1,2 -> ..
3,4,2,1 -> ..
4,1,2,3 -> ..
4,1,3,2 -> ..
4,2,1,3 -> ..
4,2,3,1 -> ..
4,3,1,2 -> ..
4,3,2,1 -> ..
```

De goedkoopste toewijzing is hier `2,1,3,4`, met kosten 13.

Complexiteit: $\Omega(n!)$,
immers alle $n!$ mogelijke toewijzingen worden bekeken.

Eindconclusie Exhaustive Search

* Veel exhaustive search algoritmen werken **alleen voor kleine probleeminstanties** in acceptabele tijd

* Voor veel problemen zijn er veel efficiëntere algoritmen bekend (Euler-kring, kortste paden, toewijzingsprobleem)

* Voor andere problemen is exhaustive search (of varianten daarop) in essentie de enig bekende oplossing (handelsreizigersprobleem, knapzakprobleem)

Bij veel problemen gaat het erom een element met een speciale eigenschap te vinden binnen een ruimte die exponentieel groeit als functie van de invoergrootte. Dan wordt meestal backtracking gebruikt als goed alternatief voor ES.

Exhaustive search genereert alle kandidaatoplossingen en haalt daar het speciale element tussenuit.

Backtracking

- bouwt kandidaatoplossingen component voor component op,
- kijkt al tijdens de constructie of de deeloplossing nog tot een oplossing kan leiden en
- zo niet, breidt dan de deeloplossing niet verder uit

Op deze manier spaar je soms veel werk uit en kun je dus grotere probleeminstanties oplossen.

Backtracking versus exhaustive search

Exhaustive search bekijkt *alle* volledige kandidaatoplossingen.
Backtracking controleert telkens van deeloplossingen of ze nog aan de eisen/restricties voldoen; zo niet, dan weet je zeker dat alle uitbreidingen van deze oplossing ook niet voldoen, dus die hoef je dan niet meer expliciet te bekijken.

Basisidee backtracking

- bouw een oplossing stap voor stap op en controleer steeds of de deeloplossing in conflict komt met de restricties (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. Straks!

Het acht koninginnenprobleem luidt als volgt:

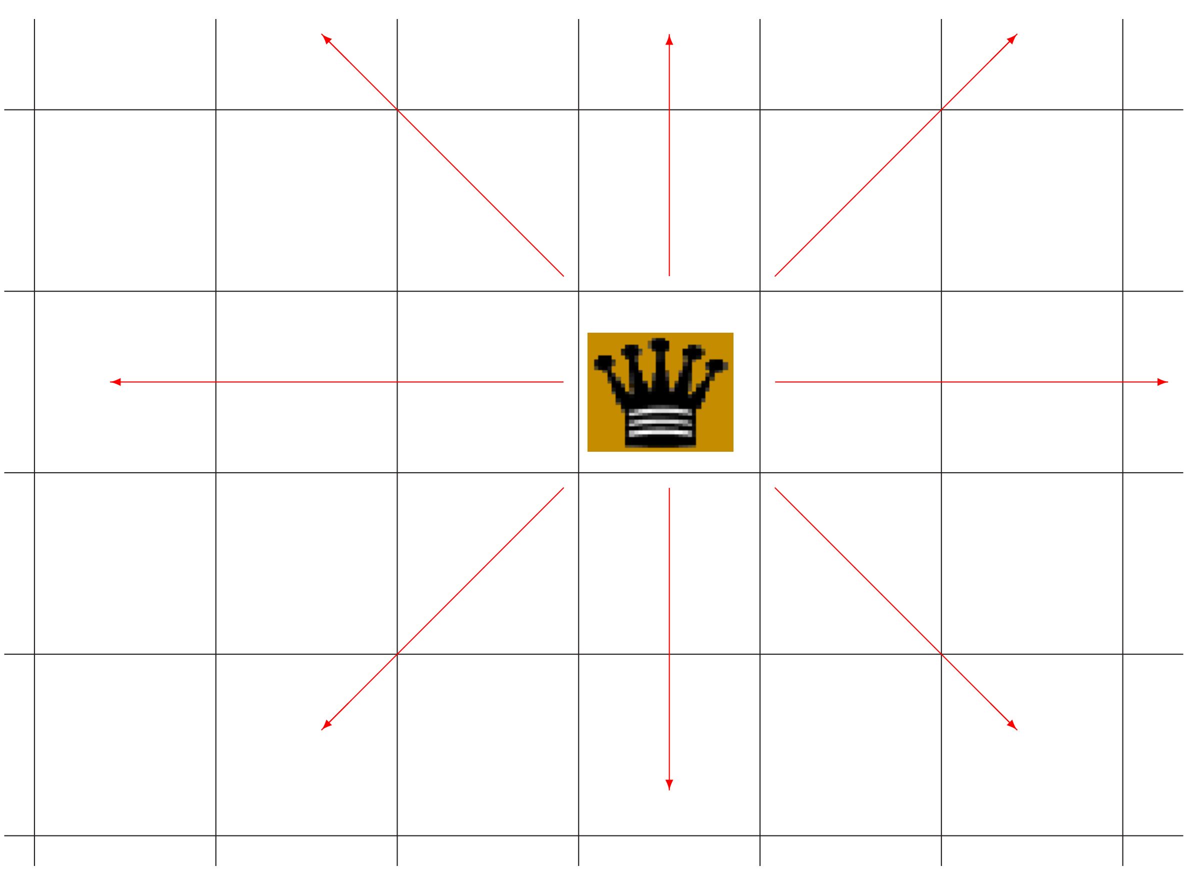
1. Kun je 8 dames (koninginnen) op een 8 bij 8 schaakbord zetten zonder dat zij elkaar aanvallen (= in één keer kunnen slaan)?
2. Zo ja, op hoeveel verschillende manieren kan dat?

En nu **algemeen**:

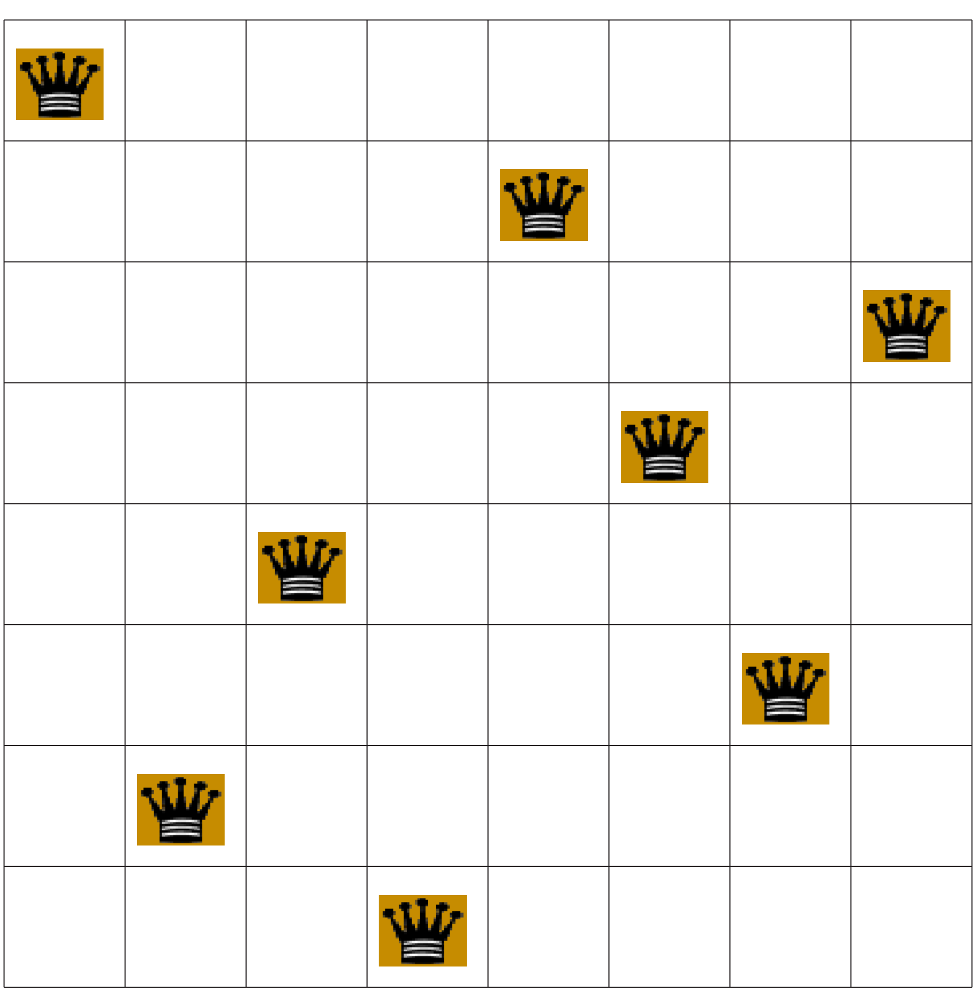
Op hoeveel manieren kun je n dames op een n bij n bord plaatsen zonder dat zij elkaar aanvallen?

Beweging van de dame

Een dame kan in één zet een willekeurig aantal vakjes naar links, rechts, onder, boven of diagonaal schuiven.

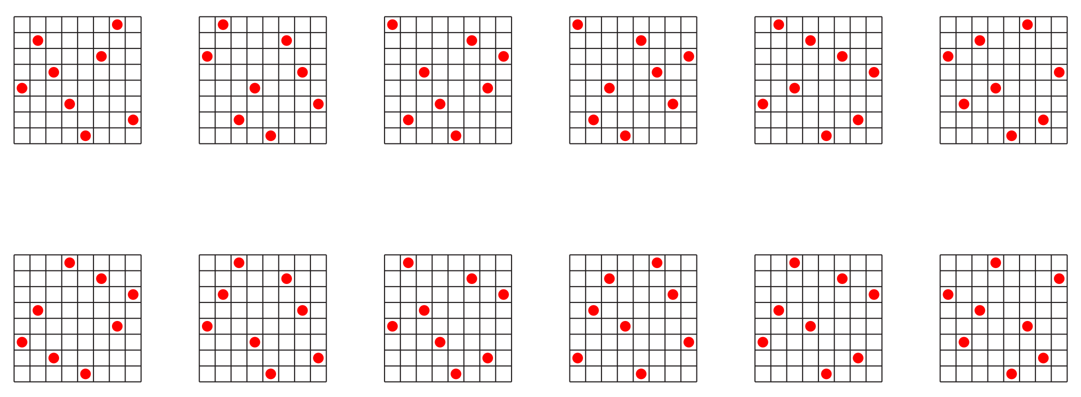


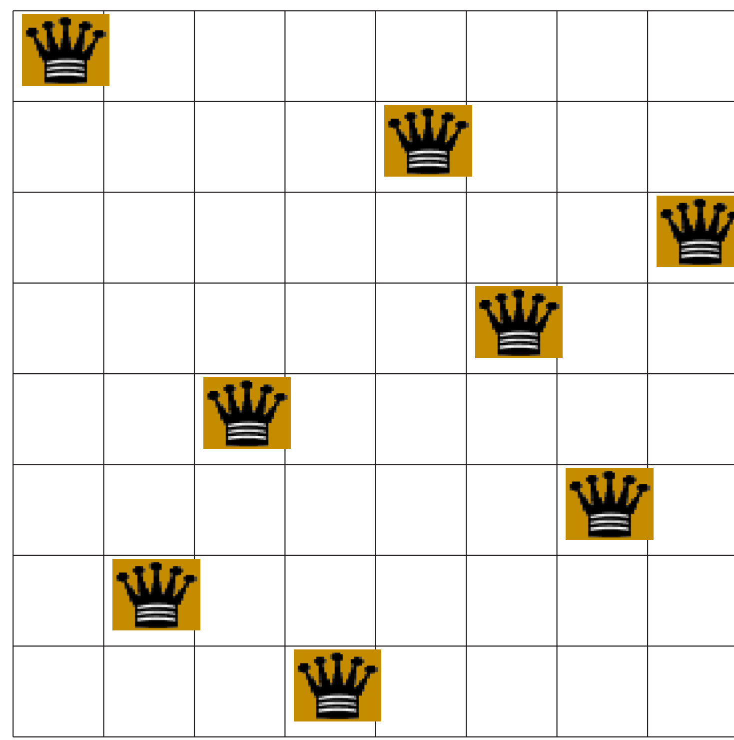
Een oplossing is onderstaande configuratie:



dit correspondeert met
de volgende permutatie:
1 5 8 6 3 7 2 4

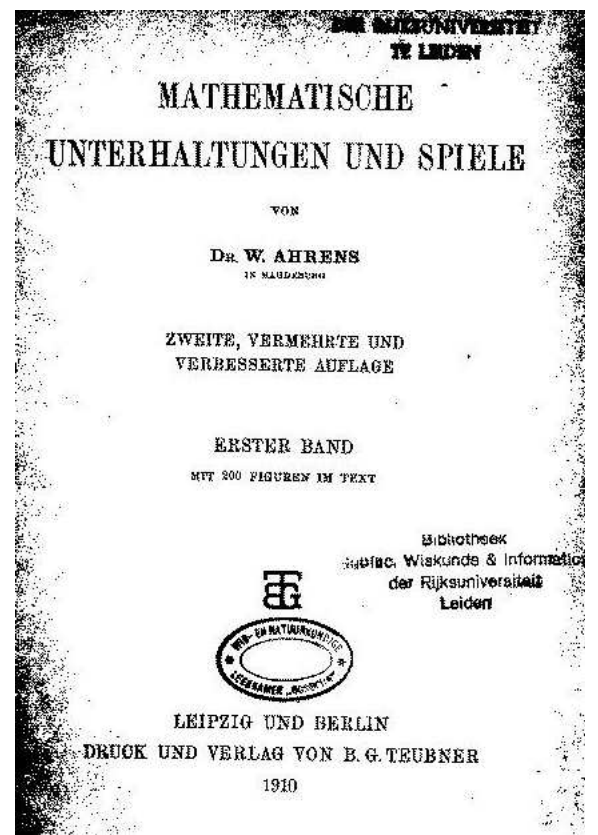
Op het 8×8 schaakbord zijn er 92 oplossingen. In essentie zijn er 12 verschillende oplossingen, waaruit je door draaien en spiegelen (8 mogelijkheden) ze allemaal kunt maken. Er is één wat meer symmetrische oplossing.





n	aantal	echt aantal	$n!$
1	1	1	1
2	0	0	2
3	0	0	6
4	2	1	24
5	10	2	120
6	4	1	720
7	40	6	5040
8	92	12	40.320
9	352	46	362.880
10	724	92	3.628.800
11	2680	341	39.916.800
12	14.200	1787	479.001.600
13	73.712	9233	
14	365.596		

W. Ahrens, 1910


MATHEMATISCHE
UNTERHALTUNGEN UND SPIELE

VON

DR. W. AHRENS
IN MAGDEBURG

ZWEITE, VERMEHRTE UND
VERBESSERTE AUFLAGE

ERSTER BAND

MIT 200 FIGUREN IM TEXT

LEIPZIG UND BERLIN
DRUCK UND VERLAG VON B. G. TEUBNER
1910

DER RIJKSUNIVERSITEIT TE LEIDEN
Bibliotheek
Subfac. Wiskunde & Informatica
der Rijksuniversiteit
Leiden

Kapitel IX.

Das Achtköniginnenproblem.

Jean Paul
„Die unsichtbare Loge". Erster Sektor.

Aus einem Gedicht des Muhammed ibn Scheref.
übers. v. Hammer-Purgstall

§ 1. Historische Einleitung.

In der „Illustrirten Zeitung" vom 1. Juni 1850 (Nr. 361, 14. Bd., p. 352) findet sich unter der Rubrik „Schach" „Eine in das Gebiet der Mathematik fallende Aufgabe von Herrn Dr Nauck in Schleusingen" folgenden Inhalts: „**Man kann 8 Schachfiguren, von denen jede den Rang einer Königin hat, auf dem Brett so aufstellen, daß keine von einer anderen geschlagen werden kann.**"[1]) In der Nummer vom 21. September

*) „Wesir" für unser „Königin". Ich entnehme dies Wort aus A. v. d. Linde, „Geschichte u. Litteratur des Schachspiels", II, p. 257.

1) Irrtümlicherweise wird diese Stelle zumeist als das erste Vorkommen unseres Problems zitiert. Die Aufgabe ist jedoch bereits in der Schachzeitung, herausgegeben von der Berliner Schachgesellschaft, Bd. III, 1848, p. 363 von einem ungenannten „Schachfreund" gestellt worden, und zwar war, wie Max Lange („Handbuch der Schachaufgaben", Leipzig 1862, p. 30. Anm. 6) „nach einer directen persönlichen Mittheilung" angibt, dieser „Schachfreund" Max Bezzel in Ansbach. — Wenn wir trotzdem oben die Geschichte des Problems an jene Nauck'sche Behandlung anknüpfen, so bestimmt uns hierbei der Umstand, daß die Fragestellung in der „Schachzeitung" zunächst nur 2 spezielle Lösungen (s. Schachzeitung IV, 1849, p. 40) gezeitigt und anscheinend überhaupt kein sonderliches Interesse für unser Problem

14*

n	Stammlösungen: doppelt-symmetrische	einfach-symmetrische	un-symmetrische	zusammen	Gesamtzahl aller Lösungen
2				0	0
3				0	0
4	1			1	2
5	1		1	2	10
6		1		1	4
7		2	4	6	40
8		1	11	12	92
9		4	42	46	352
10		3	89	92	724
11		12	329	341	2680
12	4	18	1744	1766	14032

Een brute force (exhaustive search) aanpak:

> Genereer alle mogelijke configuraties van n dames op een n bij n bord, en controleer van elk daarvan of de dames elkaar al dan niet aanvallen.

Het aantal te controleren kandidaatoplossingen is hier exponentieel:

- n^n onder de aanname: één dame per rij
- $n!$ onder de aanname: één dame per rij en één per kolom; dit zijn gewoon alle permutaties van 1 t/m n

Basisidee backtracking

- bouw een oplossing stap voor stap op en controleer steeds of de deeloplossing in conflict komt met de restricties (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

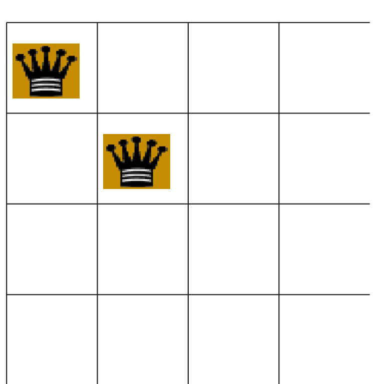
Vergelijk

- 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. Straks!

Backtracking versus exhaustive search

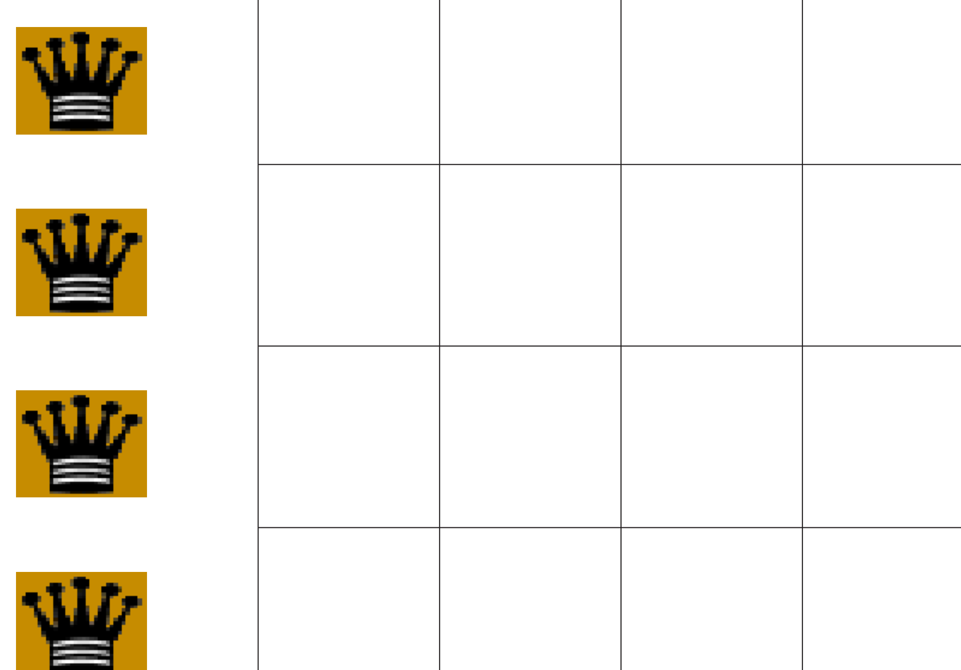
Exhaustive search bekijkt *alle* volledige kandidaatoplossingen. Dat zijn hier alle permutaties van 1 t/m n.
Backtracking controleert telkens van deeloplossingen of ze nog aan de eisen/restricties voldoen; zo niet, dan weet je zeker dat alle uitbreidingen van deze deeloplossing ook niet voldoen, dus die hoef je dan niet meer expliciet te bekijken. *Soms* spaar je zo heel veel uit.

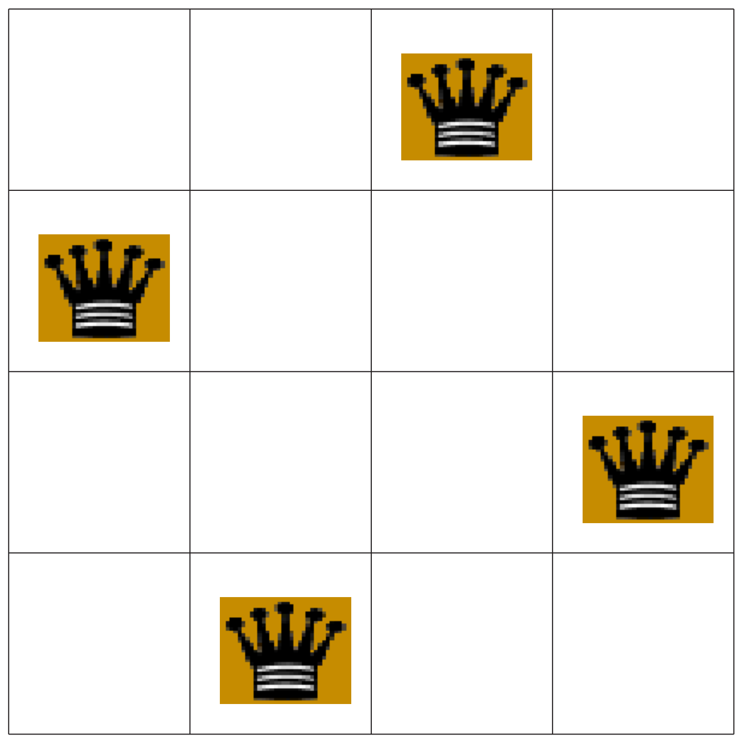
Voorbeeld:



Alle $(n-2)!$ kandidaatoplossingen met de eerste twee dames op de aangegeven posities behoeven niet verder onderzocht te worden!

Dames $n = 4$

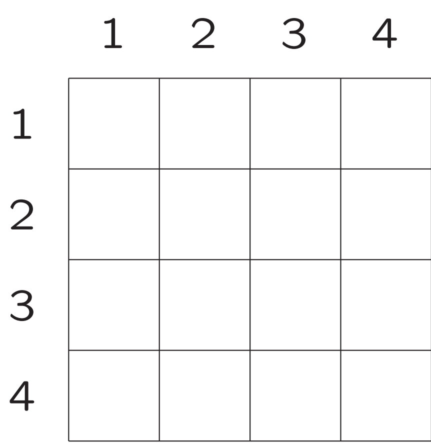




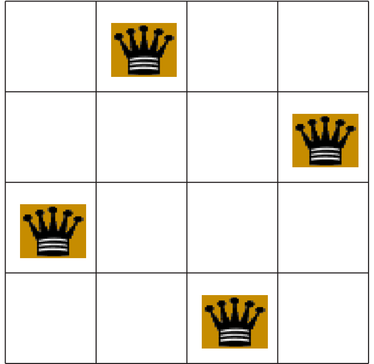
oplossing gevonden !!

zo doorgaand vinden we verder geen andere oplossingen meer

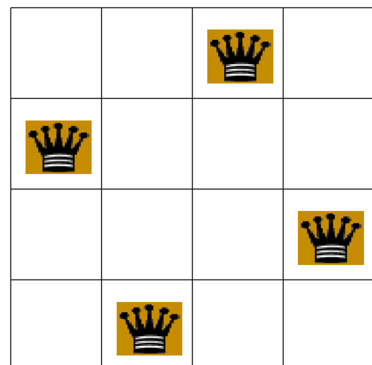
Oplossingen dames: $n = 4$



oplossing 1



oplossing 2



- **Lezen/leren bij dit college:**
  3.4, slides

- **Werkcollege** brute force en exhaustive search

  vanmiddag, 14.15–16.00, in deze zaal

- **Opgaven:**

  zie `http://www.liacs.leidenuniv.nl/~vlietrvan1/algoritmiek/`

- **Vragenuur bij programmeeropdracht 1:**

  vanmiddag, 16.15-18.00, in computerzalen Snellius

- **Volgend college:**

  dinsdag 10 maart 2020, 11.15–13.00, Havingazaal