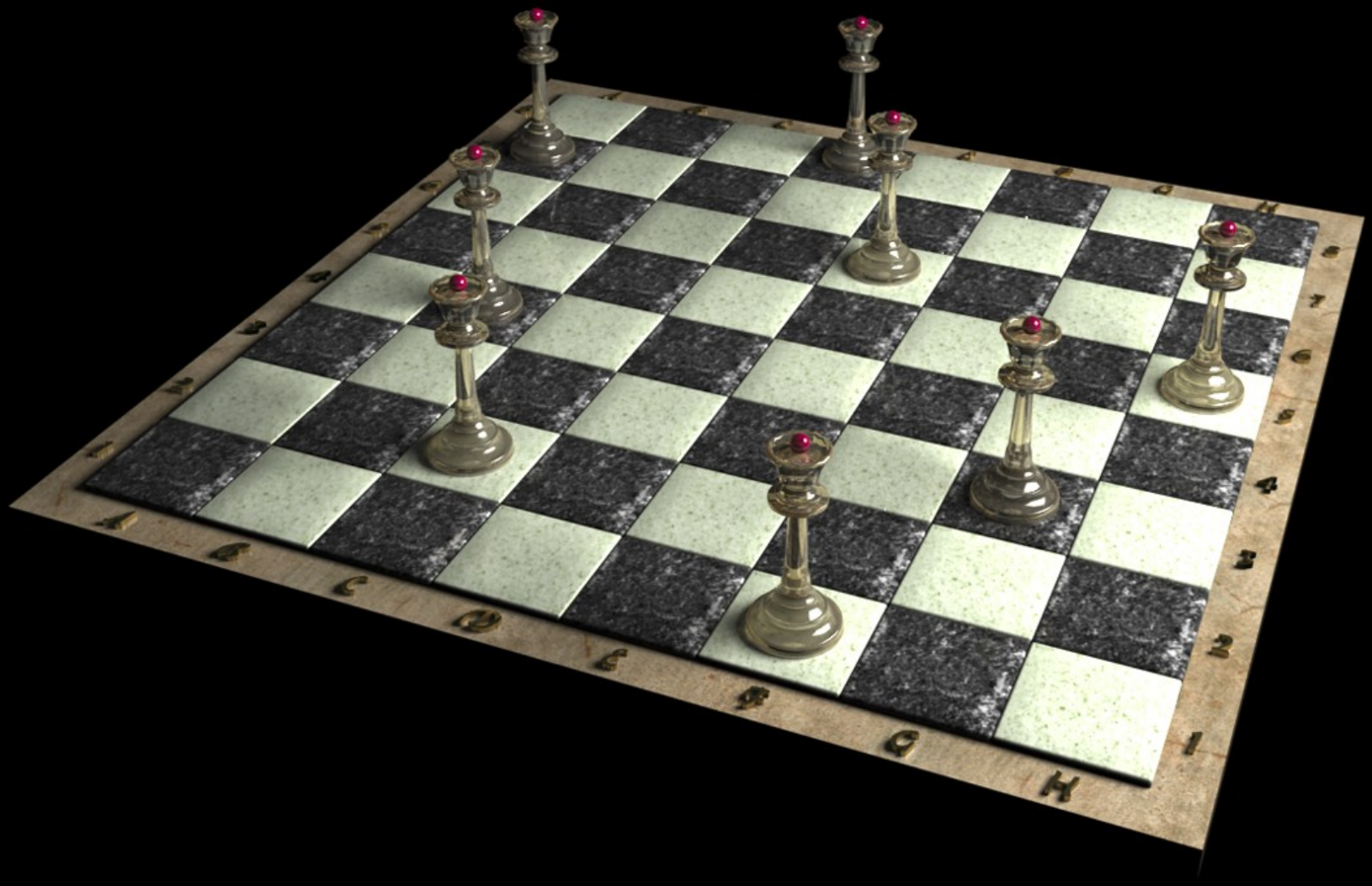
Backtracking



Backtracking som løsningsmetode

- Backtracking løser problemer der løsningene kan beskrives som en sekvens med *steg* eller *valg*
- Kan enten finne *én* løsning eller *alle* løsninger
- Bygger opp løsningen(e) *ett steg* om gangen:
  - Prøver etter hvert steg *alle mulige* steg videre
  - Hvis vi ser at et steg *ikke* kan lede til en løsning stopper vi videre utbygging av løsningen (avskjæring)
  - Tar da et steg tilbake (backtracking) og prøver i stedet *neste* mulige steg som kan lede til løsning

Backtracking vs. brute force

- "Rå kraft" (brute force) algoritmer:
  - Genererer alle mulige sekvenser av steg som *kan* være en løsning
  - For hver av de mulige løsningene: Sjekk om den er *korrekt*
  - Ressurskrevende, ubrukelig for større problemer
- Algoritmer med backtracking:
  - Er ofte basert på en "rå kraft" algoritme
  - Stopper oppbyggingen av en mulig løsning når vi ser at den ikke kan være korrekt
  - Effektiviteten avhenger av hvor smart/tidlig vi gjør denne avskjæringen av løsninger

Backtracking og rekursjon

- Backtrackingsalgoritmer er av natur rekursive:
  - Vi tar et og et steg mot en fullstendig løsning
  - I hvert steg løser vi det *samme* problemet
  - Problemstørrelsen *reduseres* for hvert steg
  - *Bunnen* i rekursjonen: Vi har bygget opp en komplett løsning og det er ikke flere steg igjen å ta

Problemeksempler

- Finne en “korrekt” rekkefølge:
  - Dronningproblemet
  - Fargelegging av kart
- Fordeling av ressurser iht. behov eller ønsker
- Løsning av kryssord, sudoku, brettspill
- Rutevalg:
  - Finne “beste” vei gjennom et nettverk
  - Labyrinter

Enkelt eksempel: En labyrint

- Kvadratisk rutenett der noen ruter er blokkerte
- Starter i øvre venstre hjørne, prøver å finne vei til nedre høyre hjørne
- Fire muligheter i hvert steg:
  - Høyre, ned, venstre, opp
- Avskjæring:
  - Går ikke ut av labyrinten
  - Går ikke til blokkerte ruter
  - Går ikke til ruter som er besøkt tidligere

Datastruktur for labyrinten

- Setter opp labyrinten som en todimensjonal tabell med heltall, med verdiene 1 (fri) og 0 (stengt)
- Mens algoritmen utføres:
  - Markerer oppsøkte ruter med verdien 2 i tabellen
  - Tar vare på veien, ved at ruter som ligger på funnet vei markeres med verdien 3 i tabellen

Representasjon av labyrint i Java

```
int[][] L = { {1,1,1,0,1,1,0,0,0,1,1,1,1},
              {1,0,0,1,1,0,1,1,1,1,0,0,1},
              {1,1,1,1,1,0,1,0,1,0,1,0,0},
              {0,0,0,0,1,1,1,0,1,0,1,1,1},
              {1,1,1,0,1,1,1,0,1,0,1,1,1},
              {1,0,1,0,0,0,0,1,1,1,0,0,1},
              {1,0,1,1,1,1,1,1,0,1,1,1,1},
              {1,0,0,0,0,0,0,0,0,0,0,0,0},
              {1,1,1,1,1,1,1,1,1,1,1,1,1} };
```

Labyrinten etter at løsning er funnet

Rekkefølge: Høyre – Ned – Venstre – Opp

```
{ {3,2,2,0,1,1,0,0,0,2,2,2,2},
  {3,0,0,1,1,0,3,3,3,2,0,0,2},
  {3,3,3,3,3,0,3,0,3,0,2,0,0},
  {0,0,0,0,3,3,3,0,3,0,2,2,2},
  {3,3,3,0,2,2,2,0,3,0,2,2,2},
  {3,0,3,0,0,0,0,3,3,2,0,0,2},
  {3,0,3,3,3,3,3,3,0,2,2,2,2},
  {3,0,0,0,0,0,0,0,0,0,0,0,0},
  {3,3,3,3,3,3,3,3,3,3,3,3,3} };
```

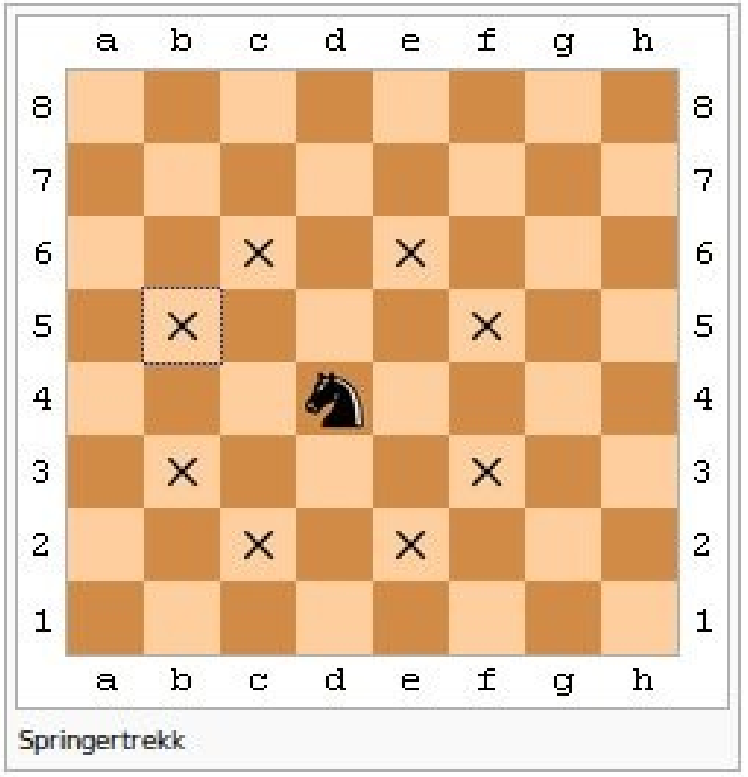
Rekursiv implementasjon

- Merker av alle besøkte ruter i labyrint-tabellen
- Bunn i rekursjonen når vi kommer til nedre høyre hjørne
- Maks. fire rekursive kall, med flytt til hver av naborutene (høyre, ned, venstre opp) som det er *lovlig* å flytte til
- Lovlig flytt: Naboruten finnes, er fri og ikke tidligere besøkt
- Stopper rekursjonen med en gang en vei er funnet
- Merker rutene på evt. vei når rekursjonen trekker seg tilbake etter at en vei er funnet
- Java-kode: `labyrint.java` *

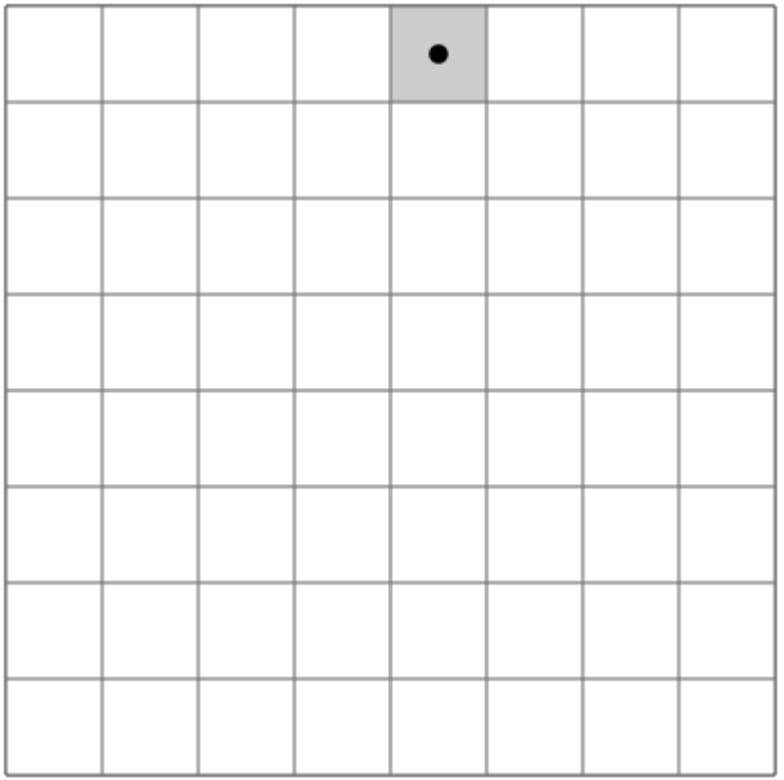
*: Her er en versjon av labyrinten som inneholder flere tips til løsningen av oblig. 2...

Oblig. 2: Springerproblemet

- Springer: “2 frem, 1 til siden”
- Starter med en springer i en gitt posisjon på et *n* x *n* brett
- Problem: Finn en måte å flytte springeren rundt på sjakkbrettet, slik at den er innom hver rute en og bare en gang – en “springertur”
- Finnes ikke alltid løsning, avhenger av startposisjon og størrelsen på brettet



Animasjon av en “springertur”



Utskrift av løsning

```
 1 10  5 18  3
14 19  2 11  6
 9 22 13  4 17
20 15 24  7 12
23  8 21 16 25
```

Springerproblemet og backtracking

Løses på en tilsvarende måte som labyrinten:

- Bruk en 2D-tabell til å lagre hvilke ruter som er oppsøkt
- Må merke en rute som *ubrukt* igjen når vi backtracker
- Prøv alle *lovlige* steg (maks. 8) videre fra hver posisjon
- Ikke gå utenfor brettet, ikke oppsøk en rute flere ganger
- Ta vare på antall steg og hvilke flytt som utføres
- Problemet er løst etter n^2 steg (alle ruter er besøkt)
- Bruk gjerne labyrintprogrammet som utgangspunkt

Springerproblemet: Tidsforbruk

- "Standardløsningen" på forrige side bruker en "brute-force" tilnærming som prøver alle veier videre
- Dette gir en vanvittig stor arbeidsmengde: Mer enn 10^{51} mulige trekksekvenser for et 8 x 8 brett
- Vil "aldri bli ferdig" for $n > 7$ på en standard PC
- Løsningen kan testes og skal virke for f.eks.:

n	5	6	7
start	(1, 1)*	(1, 1)	(2, 2)

*: (1,1) er her feltet øverst til venstre

Raskere løsninger av springerproblemet

- Problemet har vært forsket på lenge – finnes mange løsningsmetoder som er raskere enn “brute force”
- En kjent heuristisk effektivisering er Warnsdorf's regel:
  - Prøver bare én vei videre fra hvert felt
  - Springeren flyttes alltid videre til det feltet som har *færrest mulige trekk* videre
  - Finner oftest en løsning veldig mye raskere