Invarianter, Induksjon og Rekursjon

Magnus Lie Hetland

30. august 2002

Oversikt

- Invarianter er viktige for korrekthetsbevis
- Induksjon er en viktig bevisteknikk generelt
- Rekursjon er en viktig implementasjonsteknikk

Det er mange sammenhenger mellom disse…

Eksempel

La oss se på et eksempel – vi vil summere tallene i tabellen (arrayen) `nums`:

```
int nums[] = {1, 2, 3, 4, 5};
```

Invarianter

Invariant: Noe som ikke endrer seg.

Løkke-invariant: Et utsagn som er sant før/etter hver iterasjon.

Eksempel:

```
int sum=0;
for (int i=0; i!=nums.length; ++i) {
    sum += nums[i];
}
```

Invariant: `sum` er summen av de `i` første tallene av `nums`.

Til slutt er `i==nums.length`, så `sum` er summen av alle tallene.

Korrekthetsbevis for løkker

1. Bevis at løkken terminerer

2. Bevis en invariant for løkken

3. Bevis at invarianten gir riktig svar etter terminering

La oss se på 1 og 3 først. . .

Terminering

```
int sum=0;
for (int i=0; i!=nums.length; ++i) {
    sum += nums[i];
}
```

Påstand: Denne løkken terminerer.

Bevis: Til å begynne med er `i==0`. `i` endres kun ved inkrementering. `nums.length` kan ikke være negativ. Med andre ord må `i` før eller siden bli lik `nums.length`. □

Korrekt slutt-tilstand

```
int sum=0;
for (int i=0; i!=nums.length; ++i) {
    sum += nums[i];
}
```

Påstand: `sum` er summen av tallene i `nums` ved terminering.

Bevis: Vi har løkke-invarianten at `sum` til enhver tid er summen av de `i` første tallene i `nums`. Ved terminering vet vi at `i==nums.length`. □

Og så var det invarianten...

```
int sum=0;
for (int i=0; i!=nums.length; ++i) {
    sum += nums[i];
}
```

Påstand: `sum` er til enhver tid summen av de `i` første tallene i `nums`.

Bevis: Det er sant før løkken starter, siden `sum=0` og `i=0` (en definisjonssak).

Hvis det er sant for `i-1` og `sum` økes med `nums[i-1]` så er det også sant for `i`. Det er dermed sant for alle `i`, siden vi aldri "ødelegger" egenskapen. □

Men hvorfor er det slik?

Her brukte jeg induksjon…

Generelt kan vi bruke induksjon til å vise at egenskapen P gjelder for alle positive heltall i hvis:

- $P(1)$
- $P(x-1) \rightarrow P(x)$

Her kan f.eks. $P(i)$ bety:

> `sum` er summen av de i første tallene i `nums`.

Induksjon virker fordi…

$P(1)$
gir oss et sted å starte – P er sann for 1.

$P(x-1) \rightarrow P(x)$
lar oss "vandre" oppover. Hvis P er sann for 1, så er P sann for 2. Hvis P er sann for 2, så…

Et matematisk eksempel

Påstand:

$$\sum_{i=1}^{n} i = \frac{n \cdot (n+1)}{2}$$

Bevis: Vi vet at dette gjelder for $n = 1$.

Anta at det gjelder for $n = x - 1$. Vi har da:

$$\begin{aligned}
\sum_{i=1}^{x} i &= (\sum_{i=1}^{x-1} i) + x \\
&= \frac{(x-1) \cdot x}{2} + x \\
&= \frac{(x-1) \cdot x + 2x}{2} \\
&= \frac{x^2 + x}{2} \\
&= \frac{x \cdot (x+1)}{2}
\end{aligned}$$

□

Rekursjon

```
int sum(int[] nums, int i) {
    if (i==0) return 0;
    return nums[--i] + sum(nums, i);
}
```

Hvordan virker funksjonen `sum`?

For rekursive funksjoner antar du alltid at de rekursive kallene "gjør jobben sin." Du må bare passe på grensetilfellene.

Høres det kjent ut?

Induksjon og Rekursjon

```
int sum(int[] nums, int i) {
    if (i==0) return 0;
    return nums[--i] + sum(nums, i);
}
```

Påstand: Funksjonen `sum` beregner summen av de `i` første tallene i `nums`.

Bevis: Hvis `i==0` returnerer sum 0, så grunntilfellet er OK.

Hvis `sum(nums, i-1)` er OK, så er `sum(nums, i)` OK.

Dermed (per induksjon) er `sum` OK for alle $0 \leq$ `i` $\leq$ `nums.length` $\square$

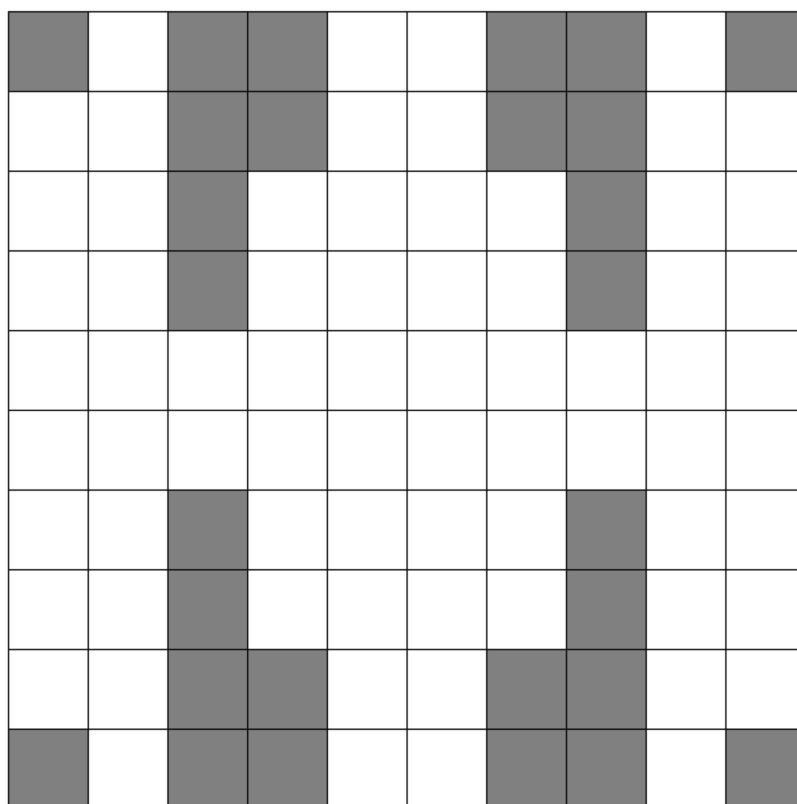
Med andre ord…

Med induksjonsprinsippet i baklomma blir rekursiv programmering mye enklere.

Vi sjekker grensetilfeller, og antar at alle rekursive kall er korrekte. Hvis resultatet vårt da blir riktig så er den rekursive funksjonen korrekt.

Et viktig eksempel: Fargelegging (1)

Vi skal fargelegge et irregulært område av skjermen (det hvite på figuren nedenfor):



Taktikk

Hvordan fargelegge med utgangspunkt i en **rute**:

a) Fargelegg **rute**.

b) For **nabo** i [**høyre**, **under**, **venstre**, **over**]:

> Hvis **nabo** ikke er fargelagt eller ulovlig: Fargelegg med utgangspunkt i **nabo**.

Korrekthet

Hvordan kan vi bevise at denne taktikken vil fargelegge hele regionen?

Svaret er som før: Induksjon.

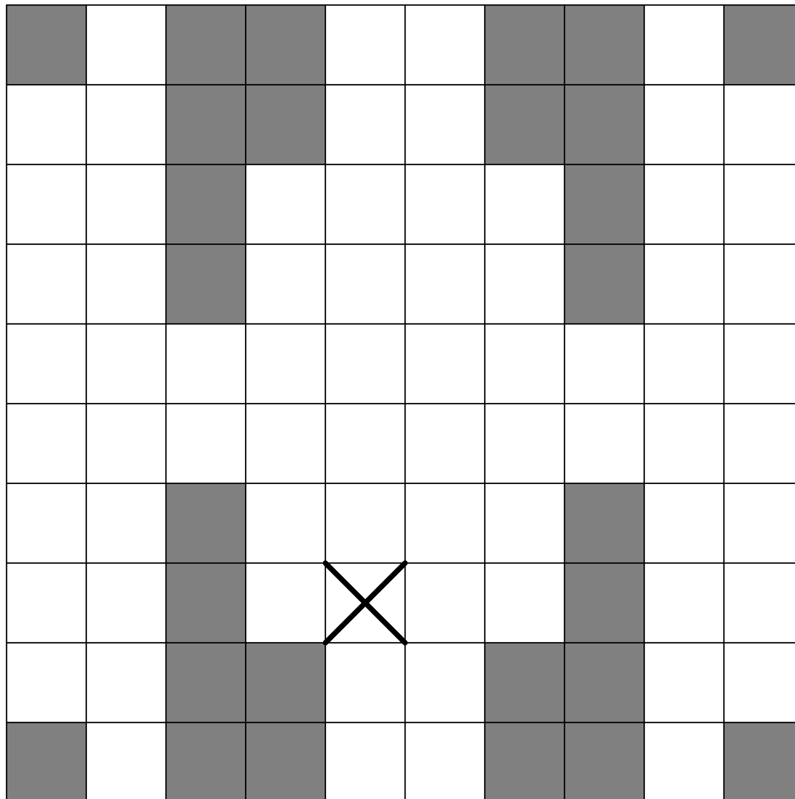
Anta at `colour` er en metode som implementerer taktikken vår, at `grid` er rutenettet, og at `point` er et vilkårlig punkt i den regionen vi skal fargelegge.

Påstand: `colour(grid,point)` fargelegger alle hvite punkter som kan nås fra `point`.

Bevis: Hvis `point` ikke har noen hvite naboer fargelegger den seg selv, og utsagnet er dermed sant.

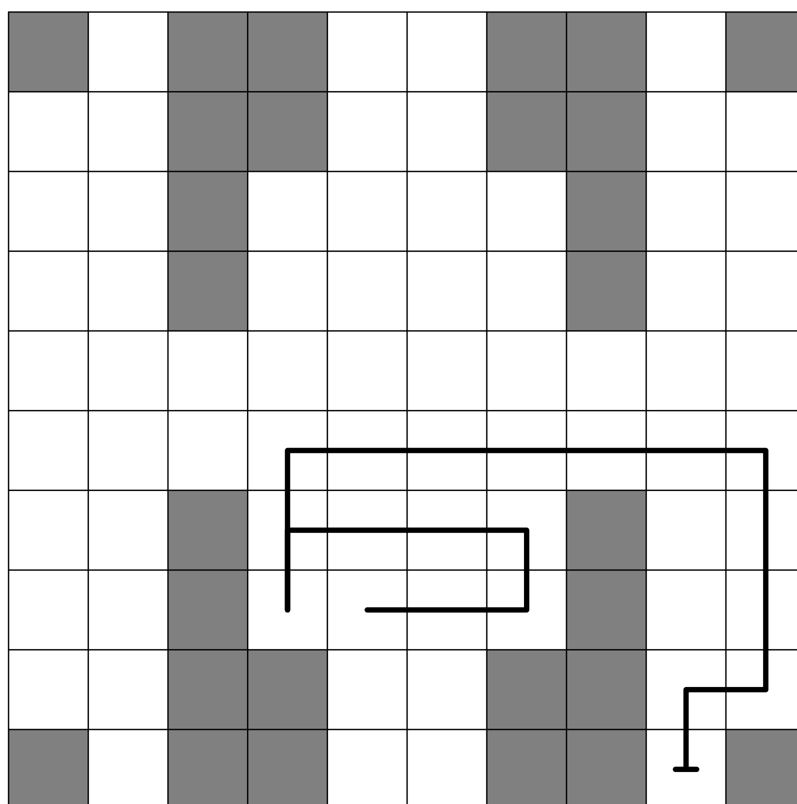
Anta at `point` har naboer. Anta at utsagnet er sant for alle punktmengder som er mindre enn den som kan nås fra `point`. Når `point` er fargelagt stemmer altså utsagnet for alle naboene til `point`. De rekursive kallene til naboene gjør dermed utsagnet sant også for `point`. □

Fargelegging (2)

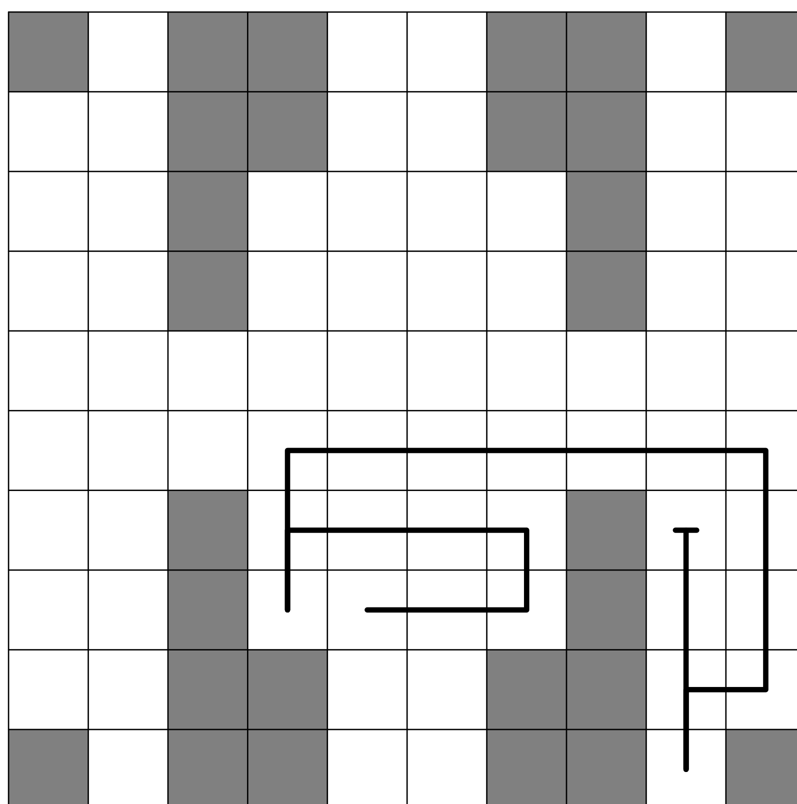


Fargelegging (3)

Fargelegging (4)

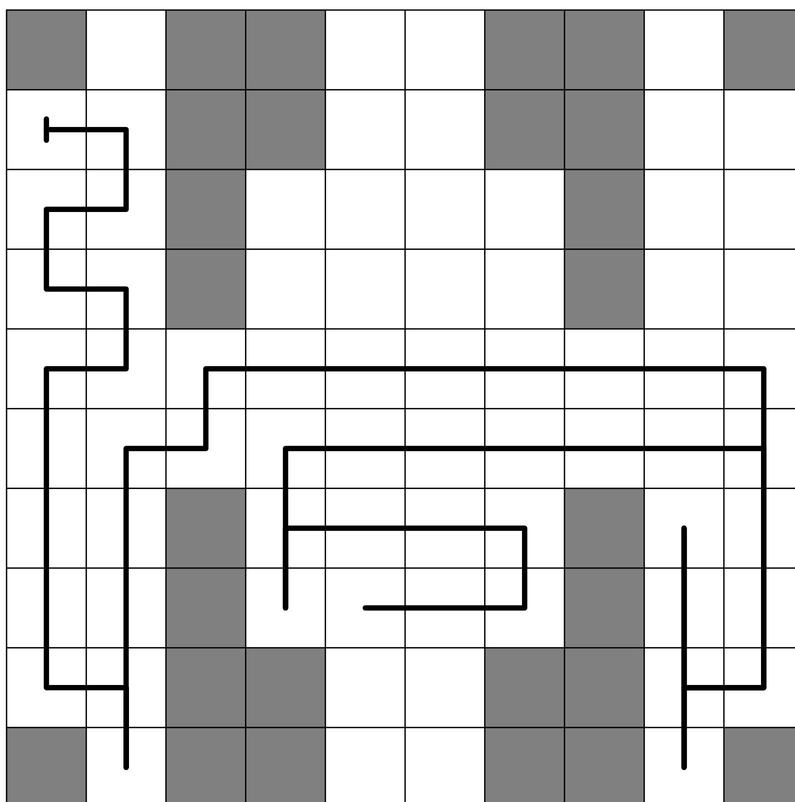


Fargelegging (5)

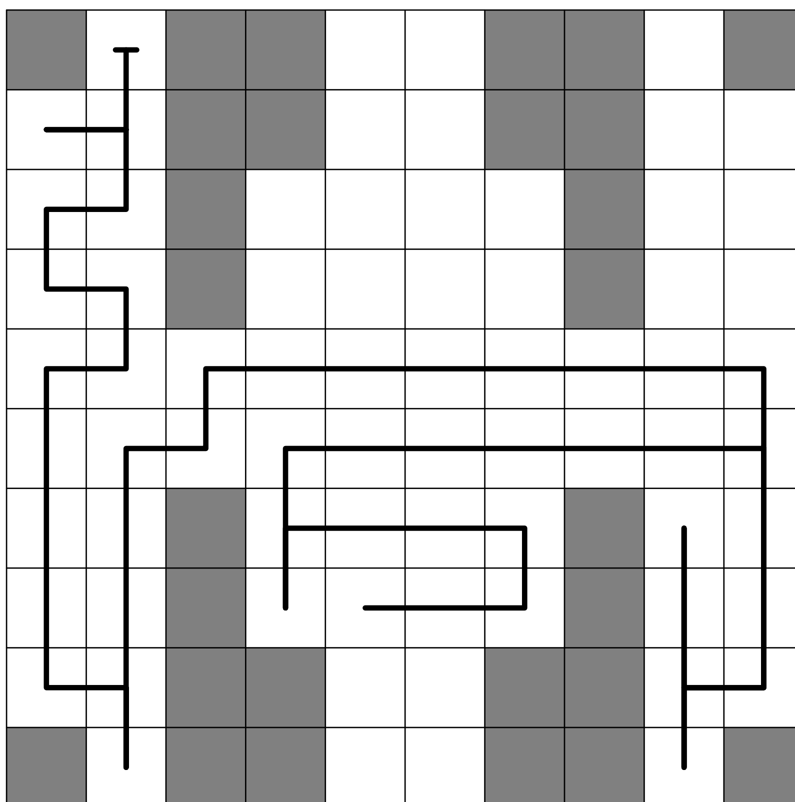


Fargelegging (6)

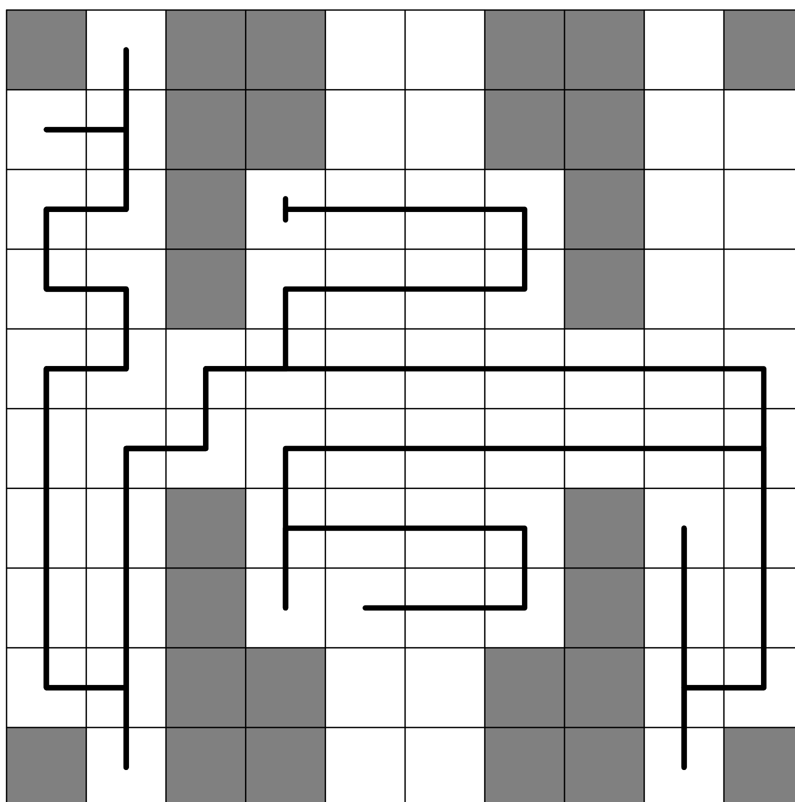
Fargelegging (7)



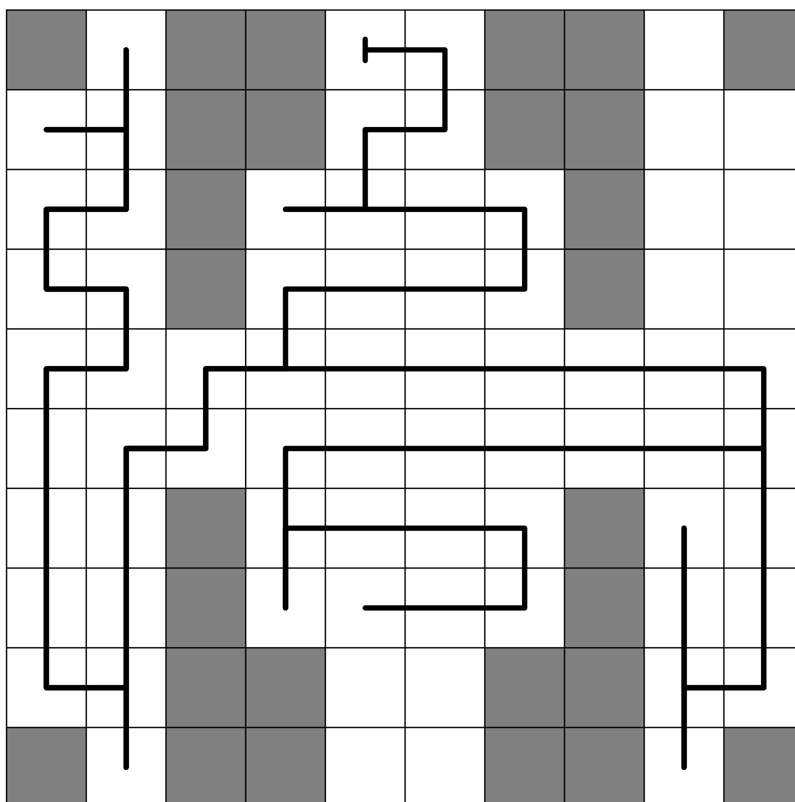
Fargelegging (8)



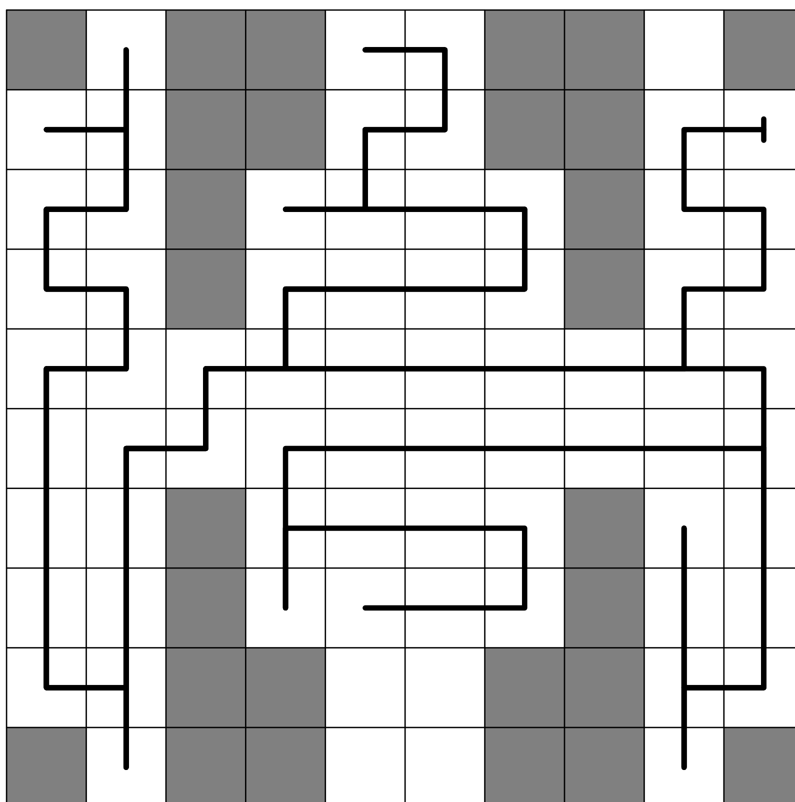
Fargelegging (9)



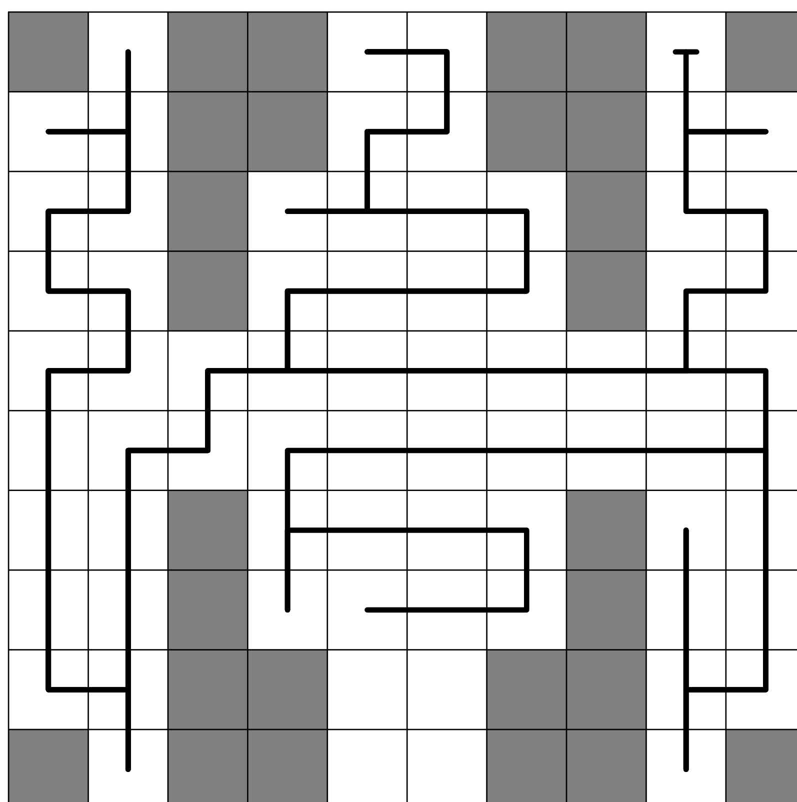
Fargelegging (10)



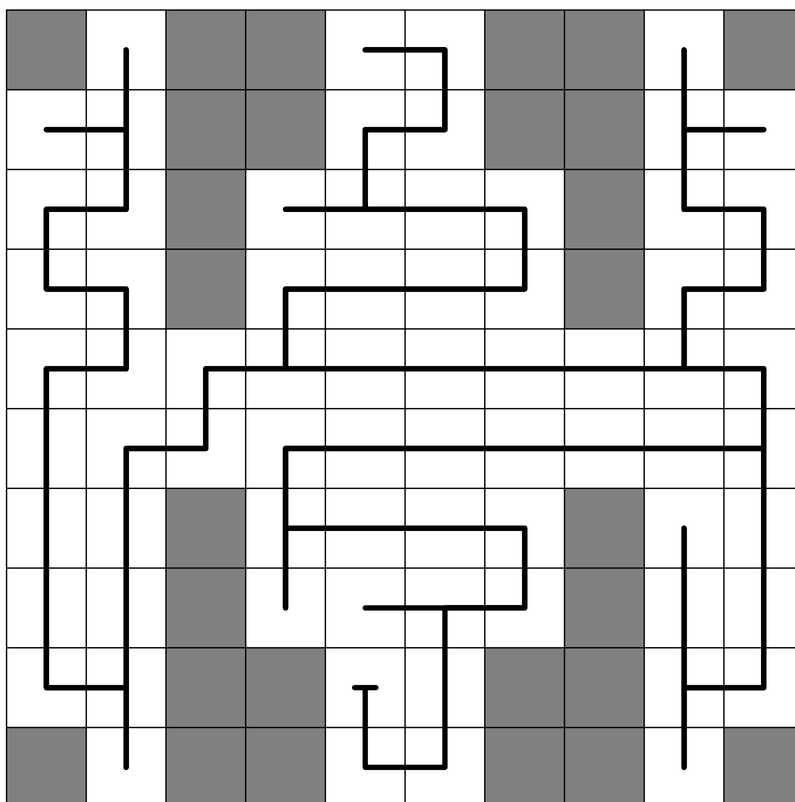
Fargelegging (11)



Fargelegging (12)



Fargelegging (13)



Men hvorfor var dette så viktig, da?

For det første: God trening i å forstå rekursjon.

For det andre: De samme prinsippene gjelder for all traversering!

De eneste antagelsene vi gjorde i korrekthetsbeviset var at:

… antall ruter var endelig

… vi kunne finne naboene til hver rute

Med disse antagelsene kan vi altså behandle ("fargelegge") alle rutene én gang. Ingen krav om rutemønster, maks 4 naboer e.l. Taktikken virker altså på trær og grafer også.

Generell taktikk
(dybde-først-traversering)

Hvordan traversere en graf med utgangspunkt i en **node**:

a) Fargelegg **node**.

b) For **nabo** i **naboliste**:

> Hvis **nabo** ikke er fargelagt eller ulovlig: Traverser med utgangspunkt i **nabo**.