

Forelesning 1

Vi starter med fagfeltets grunnleggende byggesteiner, og skisserer et rammeverk for å tilegne seg resten av stoffet. Spesielt viktig er ideen bak induksjon og rekursjon: Vi trenger bare se på *siste trinn*, og kan *anta* at resten er på plass.

Pensum

- Kap. 1. The role of algorithms in computing
- Kap. 2. Getting started:
Innledning, 2.1–2.2
- Kap. 3. Growth of functions:
Innledning og 3.1

Læringsmål

- [A₁] Forstå bokas *pseudokodekonvensjoner*
- [A₂] Kjenne egenskapene til *random-access machine*-modellen (RAM)
- [A₃] Kunne definere *problem*, *instans* og *problemstørrelse*
- [A₄] Kunne definere *asymptotisk notasjon*, O, Ω, Θ, o og ω.
- [A₅] Kunne definere *best-case*, *average-case* og *worst-case*
- [A₆] Forstå *løkkeinvarianter* og *induksjon*
- [A₇] Forstå *rekursiv dekomponering* og *induksjon over delproblemer*
- [A₈] Forstå **INSERTION-SORT**

Forelesningen filmes



Forelesning 1

Problemer og algoritmer



1. Hva og hvorfor?
2. Asymptotisk notasjon
3. Dekomponering
4. Eksempel: Sum
5. Eksempel: Insertion-sort

Hovedbudskap:

- Brute force er ofte helt ubruklig
- Dekomponer problemet i stedet:
 - Anta at du kan løse mindre instanser
 - Bruk dette til å finne en løsning

1:5

Hva og hvorfor?

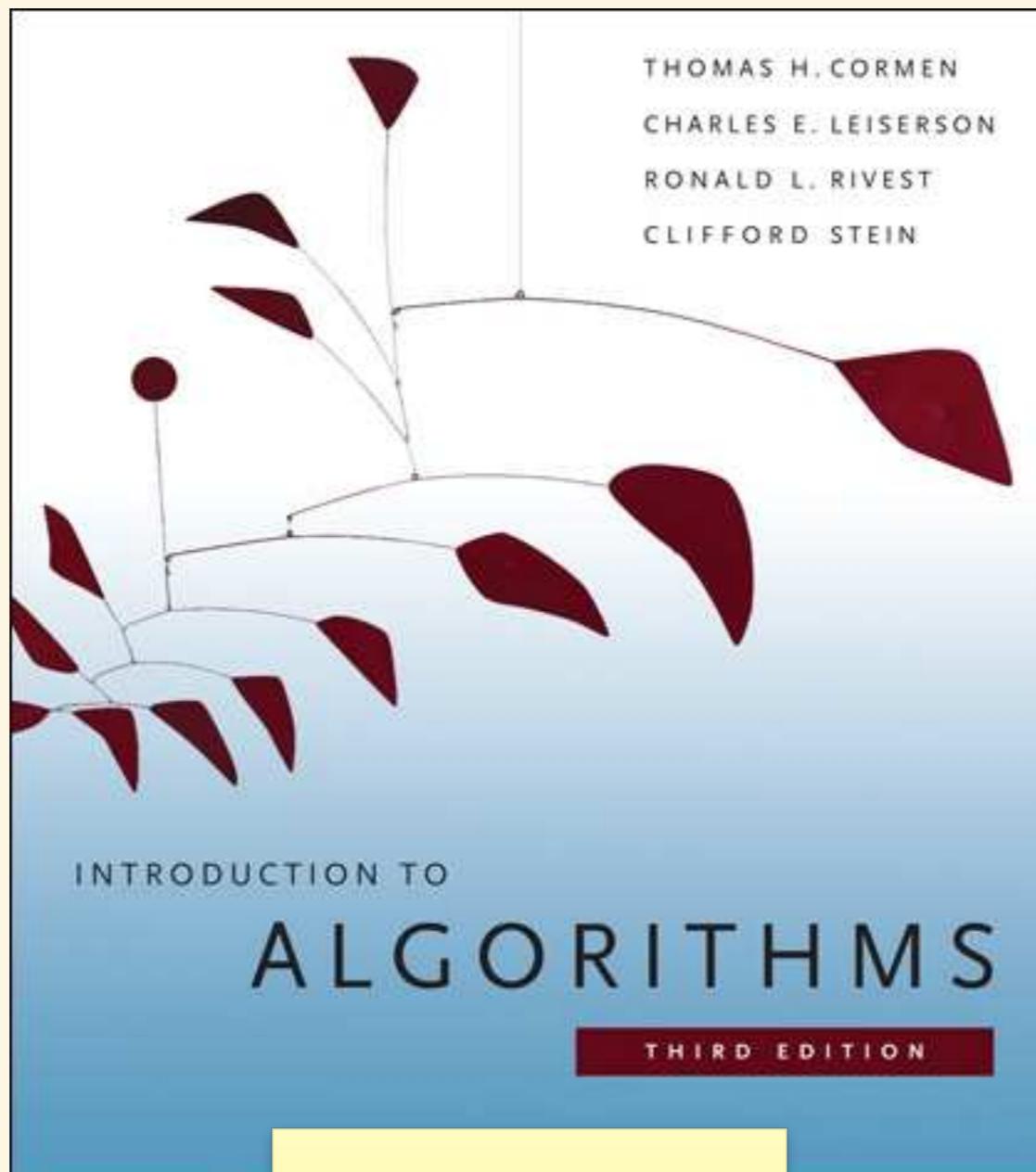


Om faget

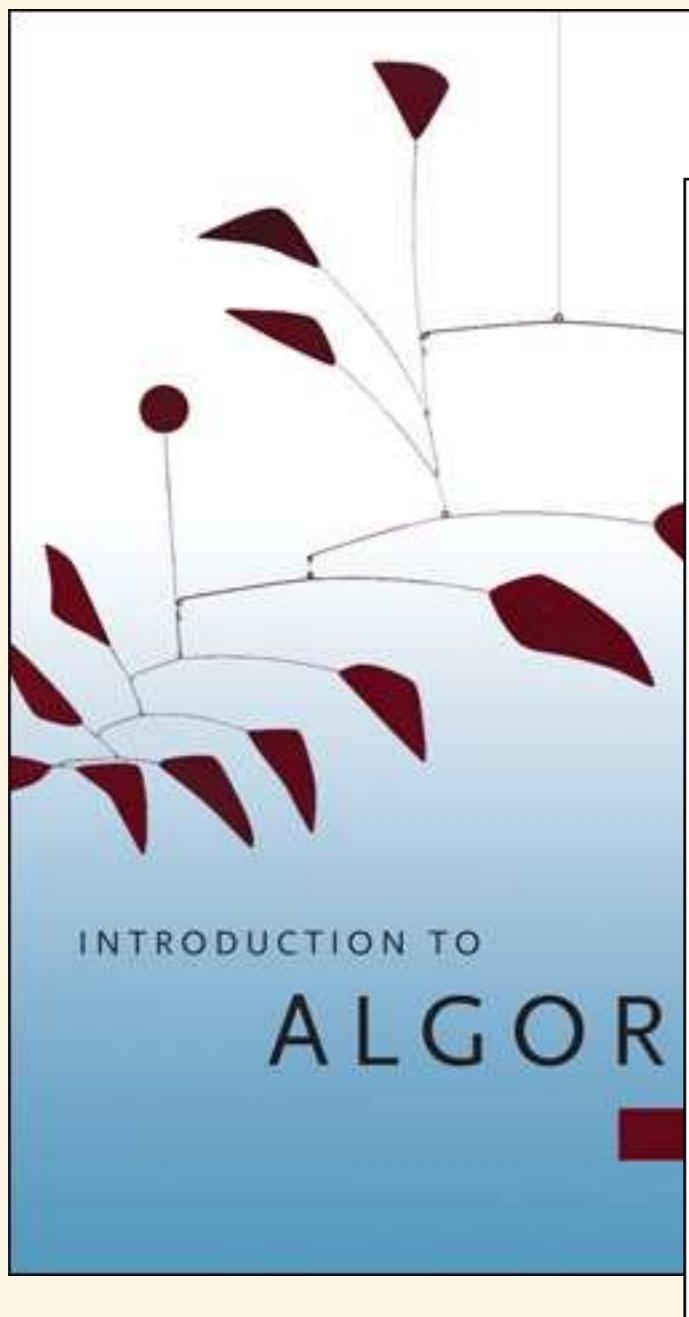
En kort orientering

Læringsmål

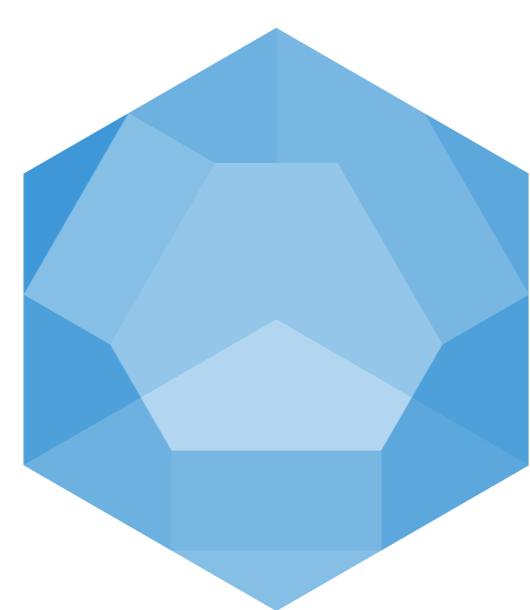
- Kjenne klassiske algoritmer
- Kjenne klassiske problemer
- Kunne analysere og designe algoritmer



Det er viktig å lese boka. Det holder ikke med YouTube-forklaringer :-)



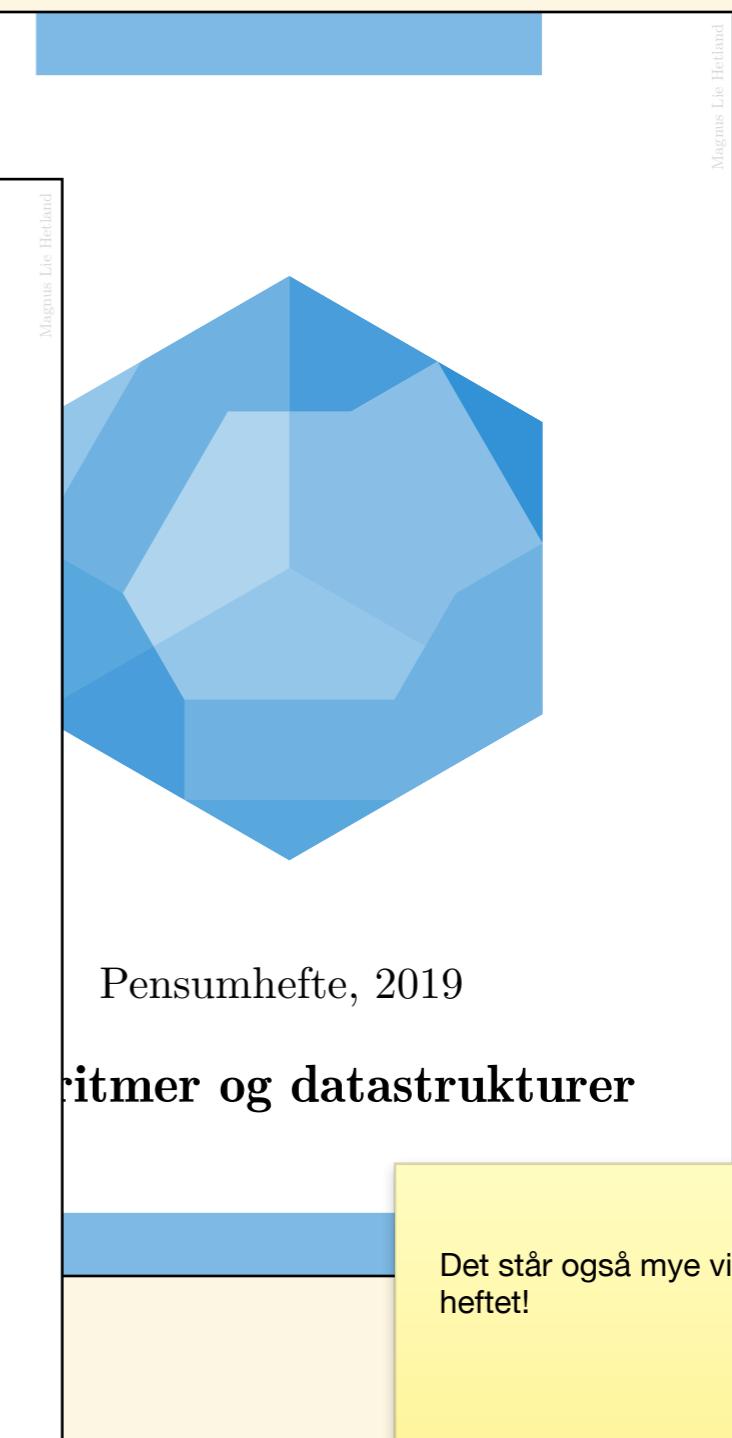
THOMAS H. CORMEN
CHARLES E. LEISERSON



Problemløsningsguide, 2019

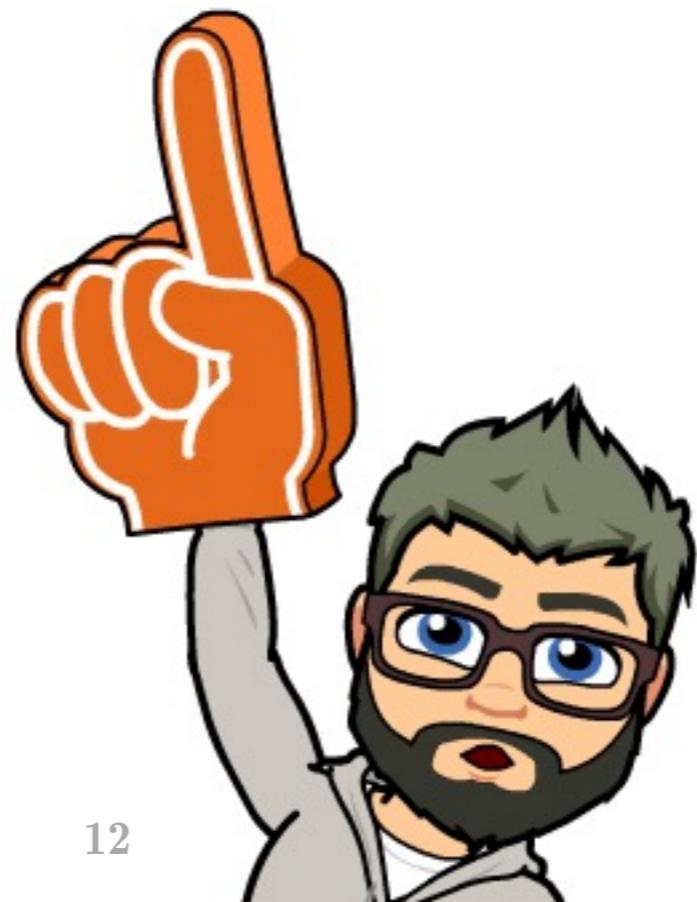
Algoritmer og datastrukturer

Denne guiden er bare ment å være til hjelp – den er ikke pensum!



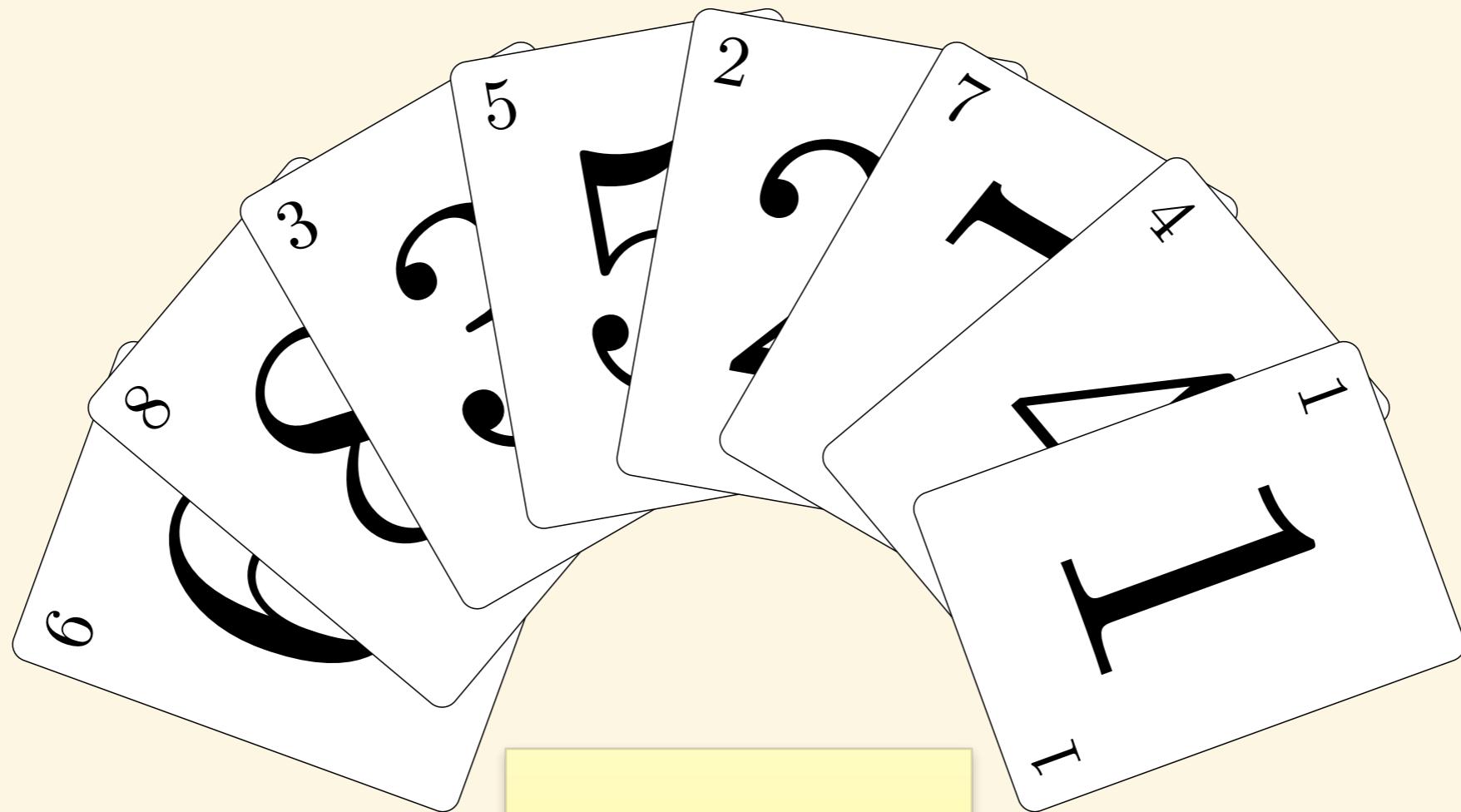
Merk: I motsetning til i noen andre fag er ikke boka bare en «anbefalt ressurs». Den er helt sentral!

algdat.idi.ntnu.no

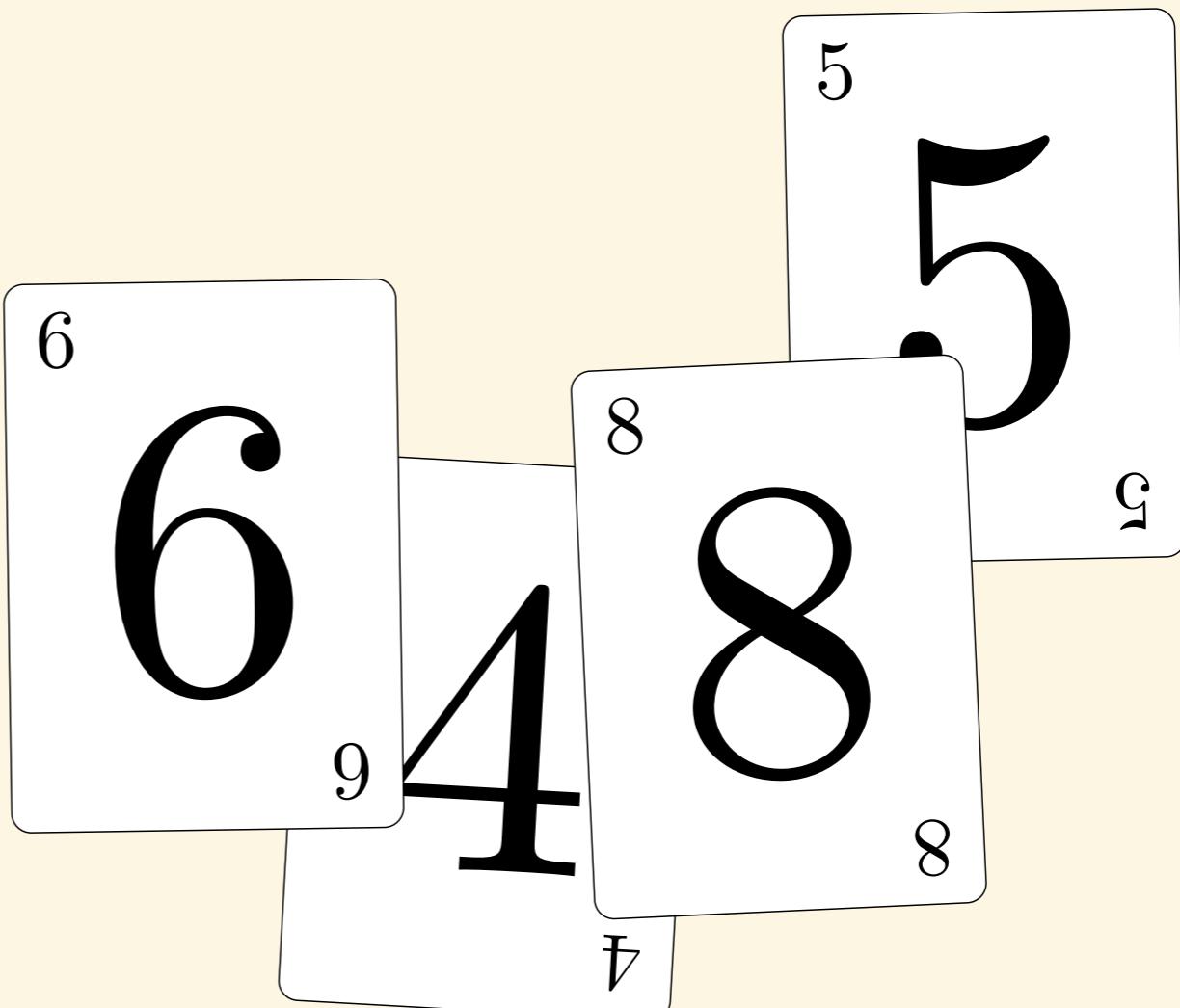
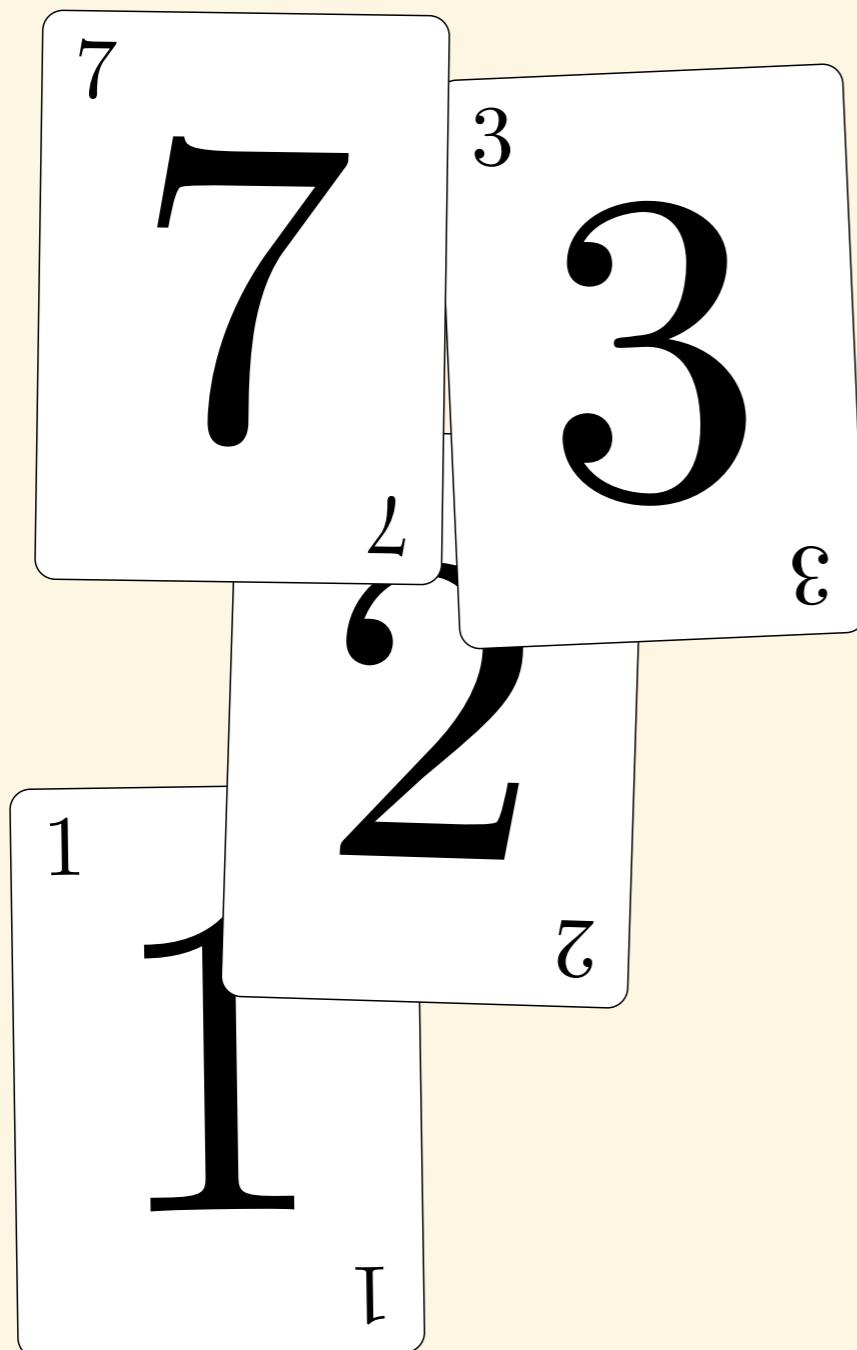


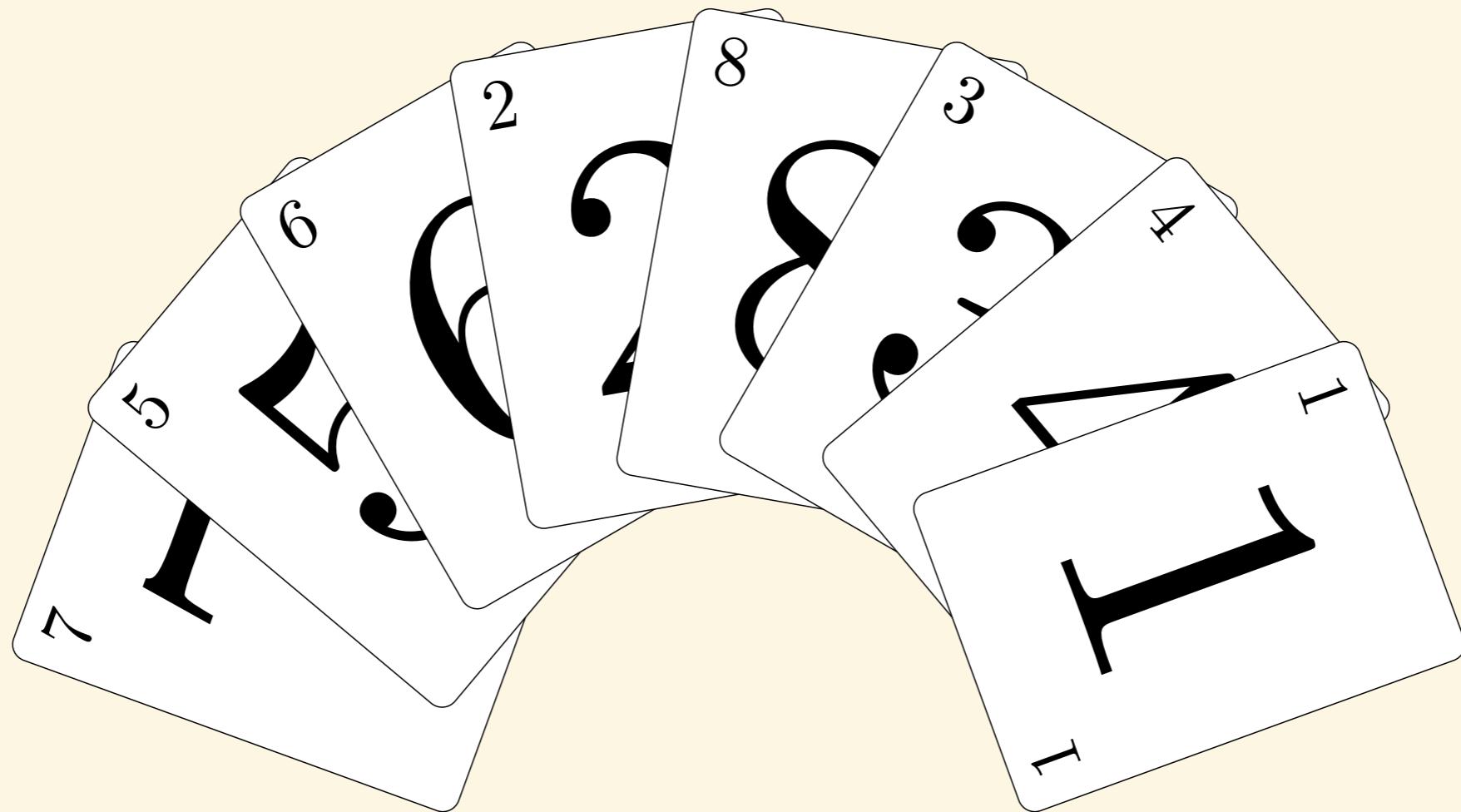
Motivasjon

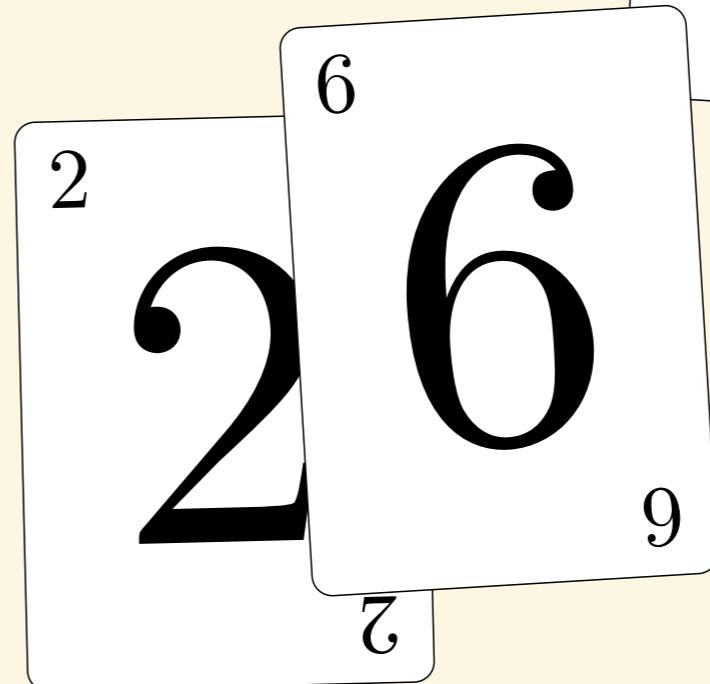
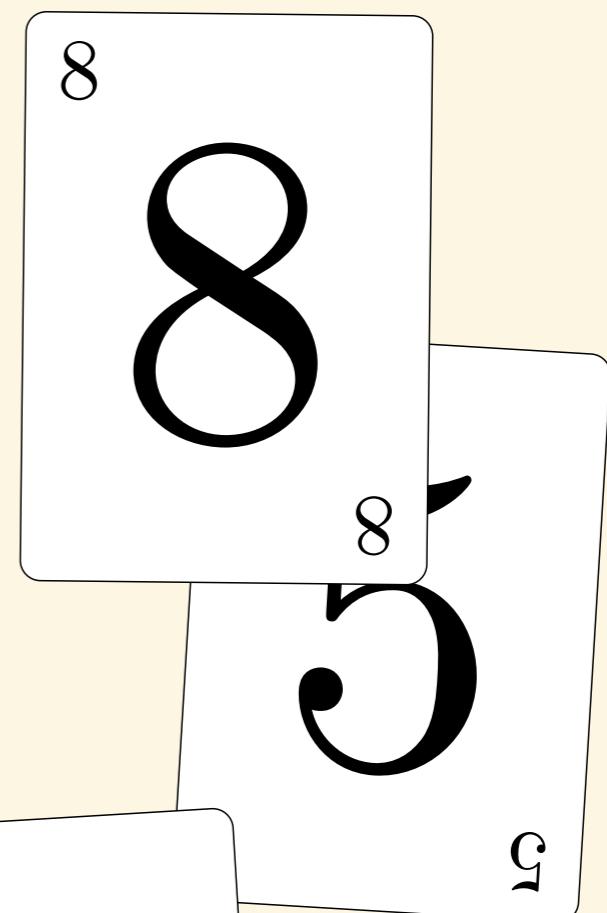
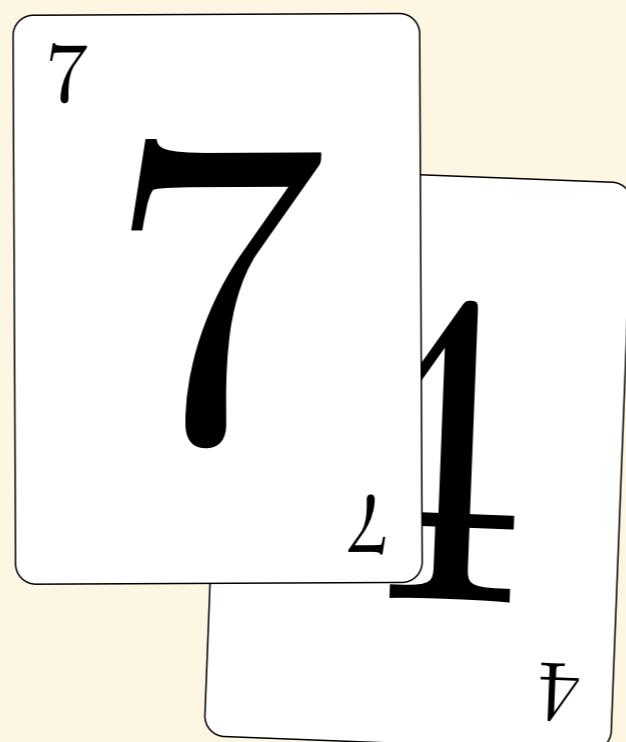
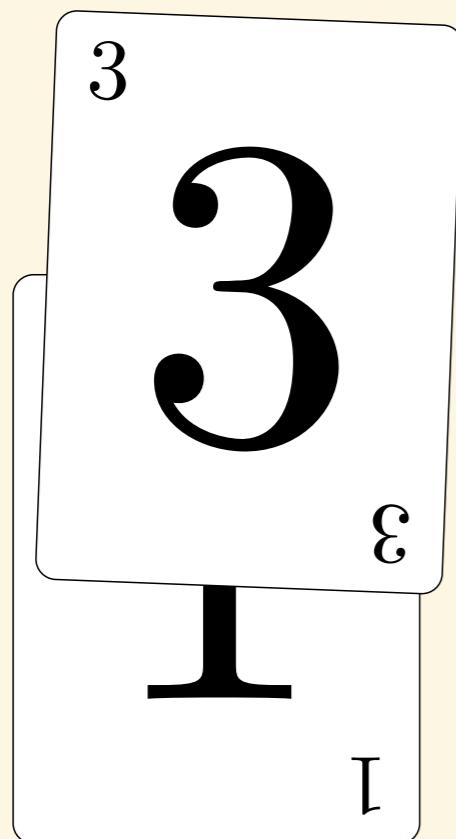
Hvorfor ikke «brute force»?

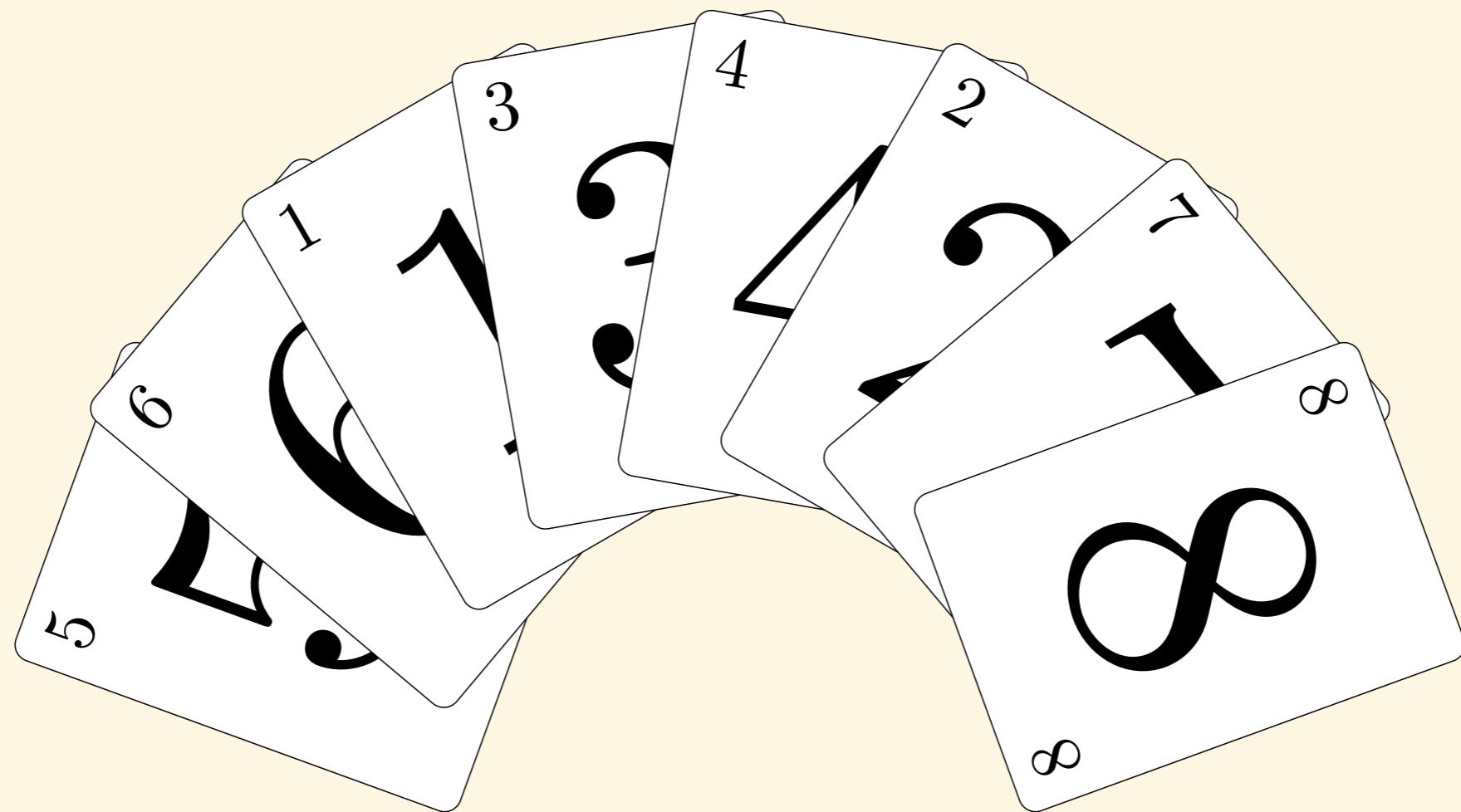


Hva om vi vil sortere noen kort
... kan vi bare skyfle dem rundt
og se om vi får rett svar?









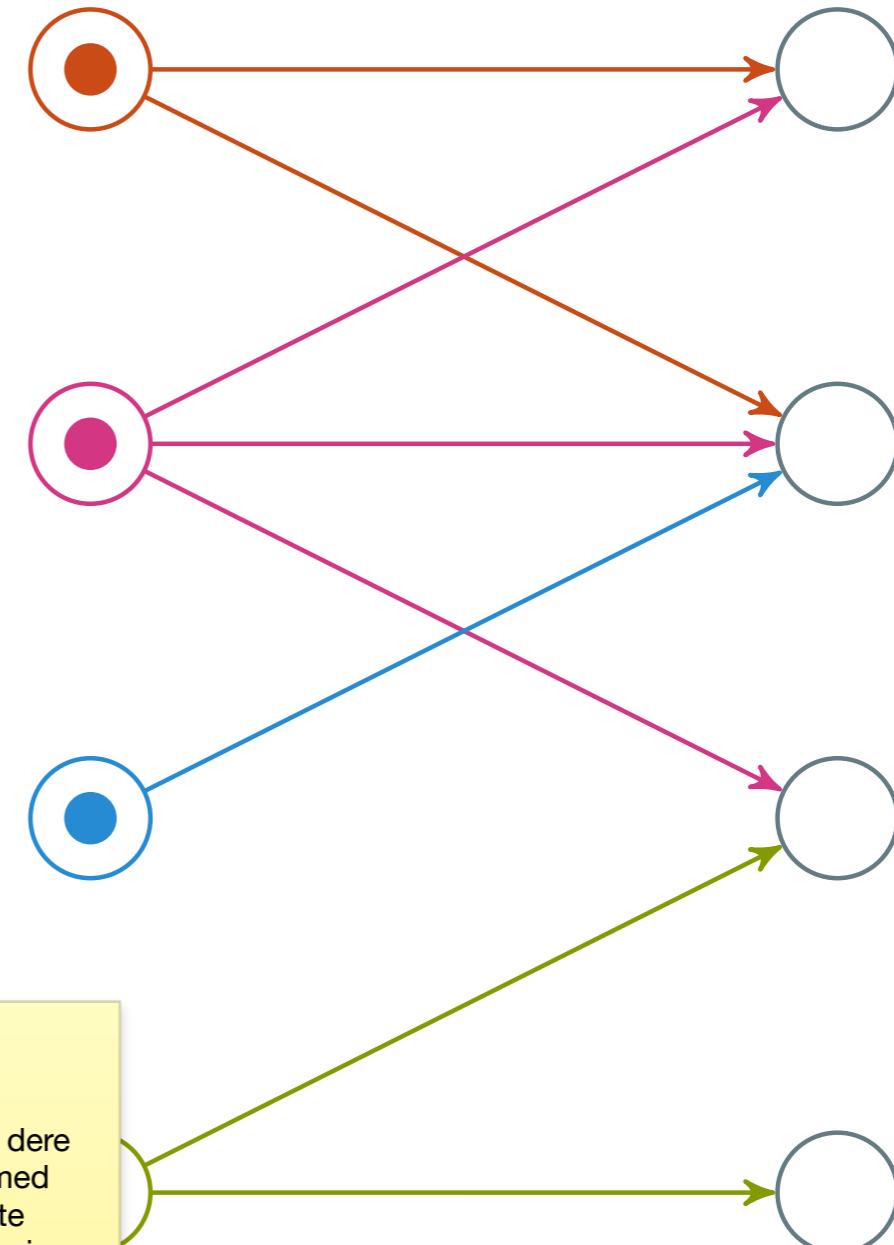
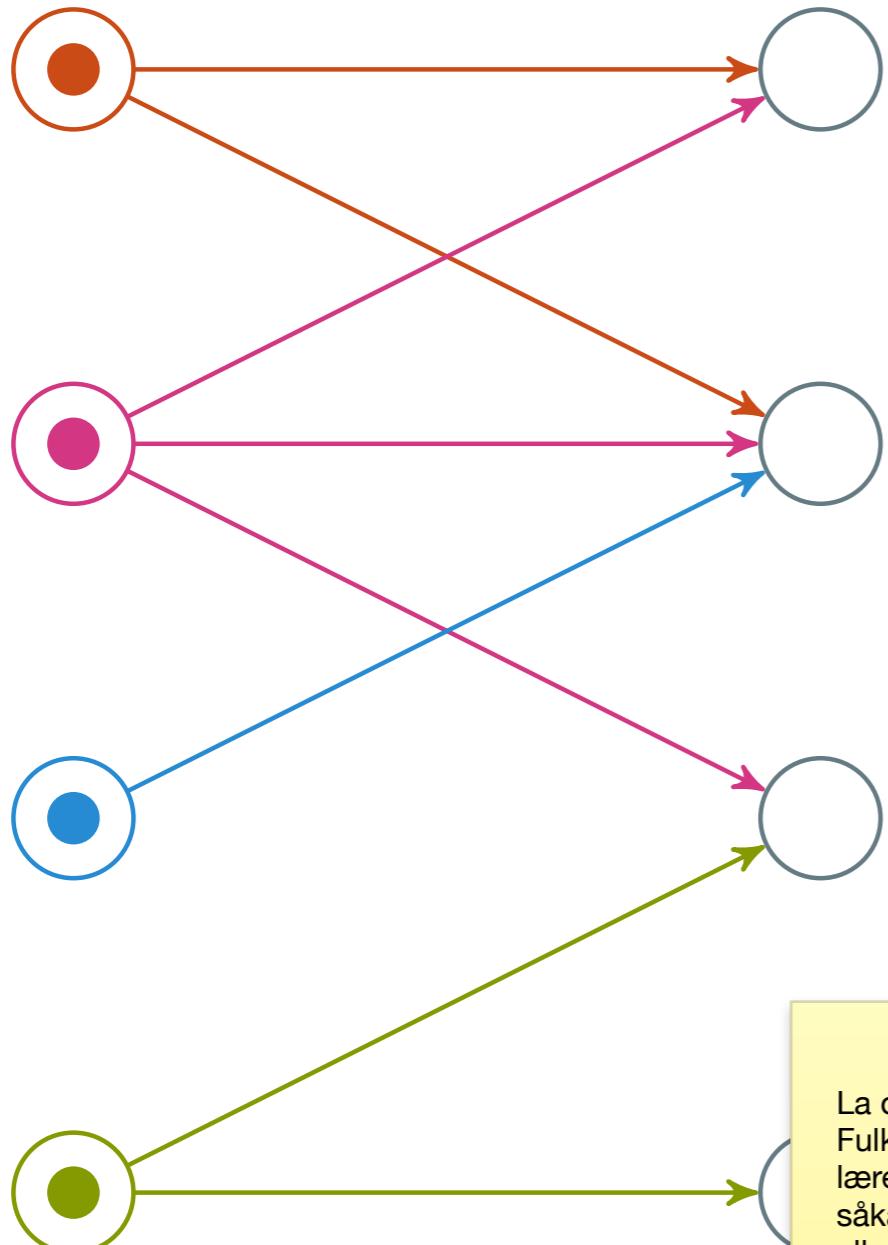
Dette kalles bogo-sort, og er noen særlig god løsning.

Og ikke er dette et så viktig problem heller, kanskje.

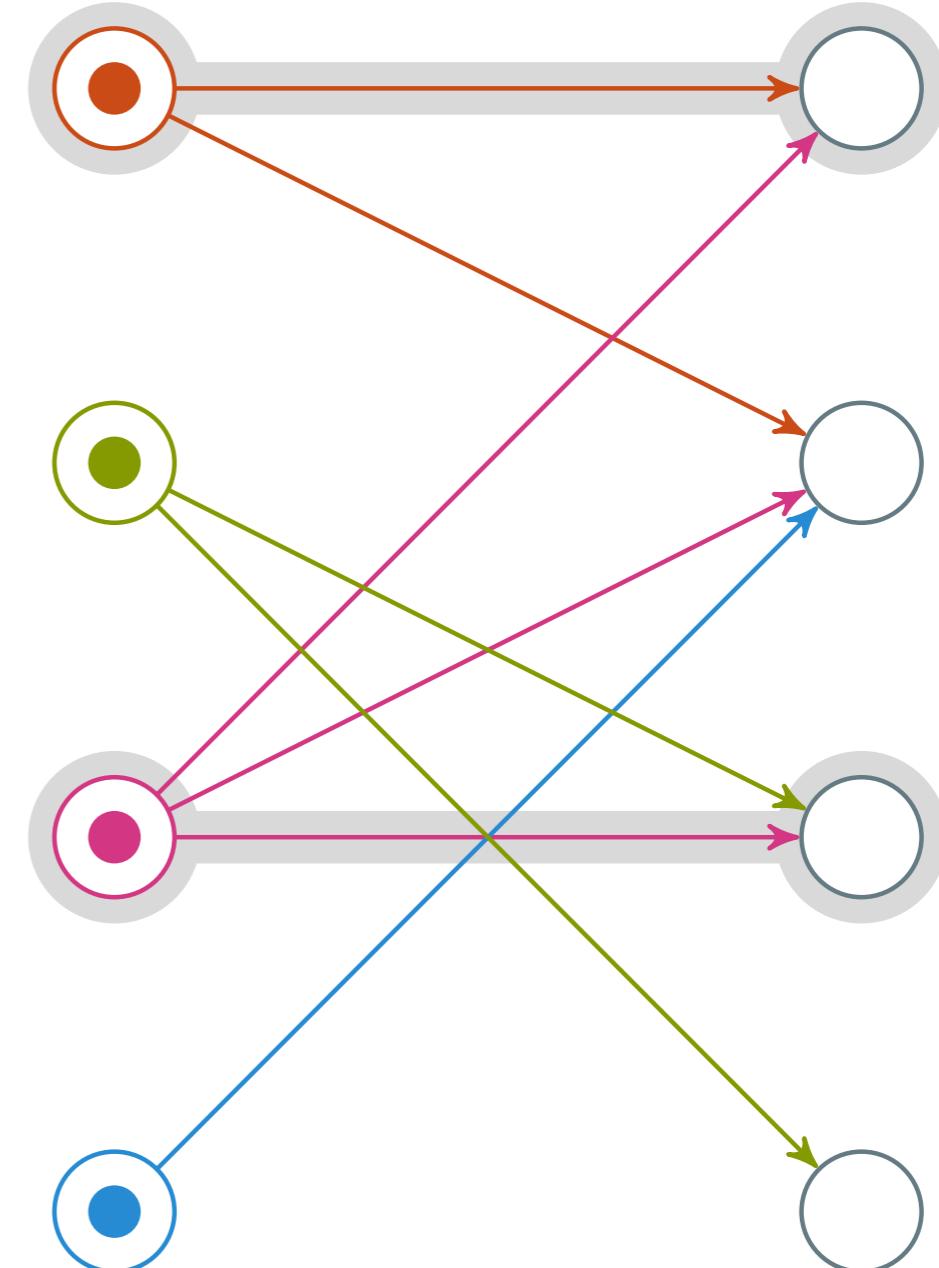
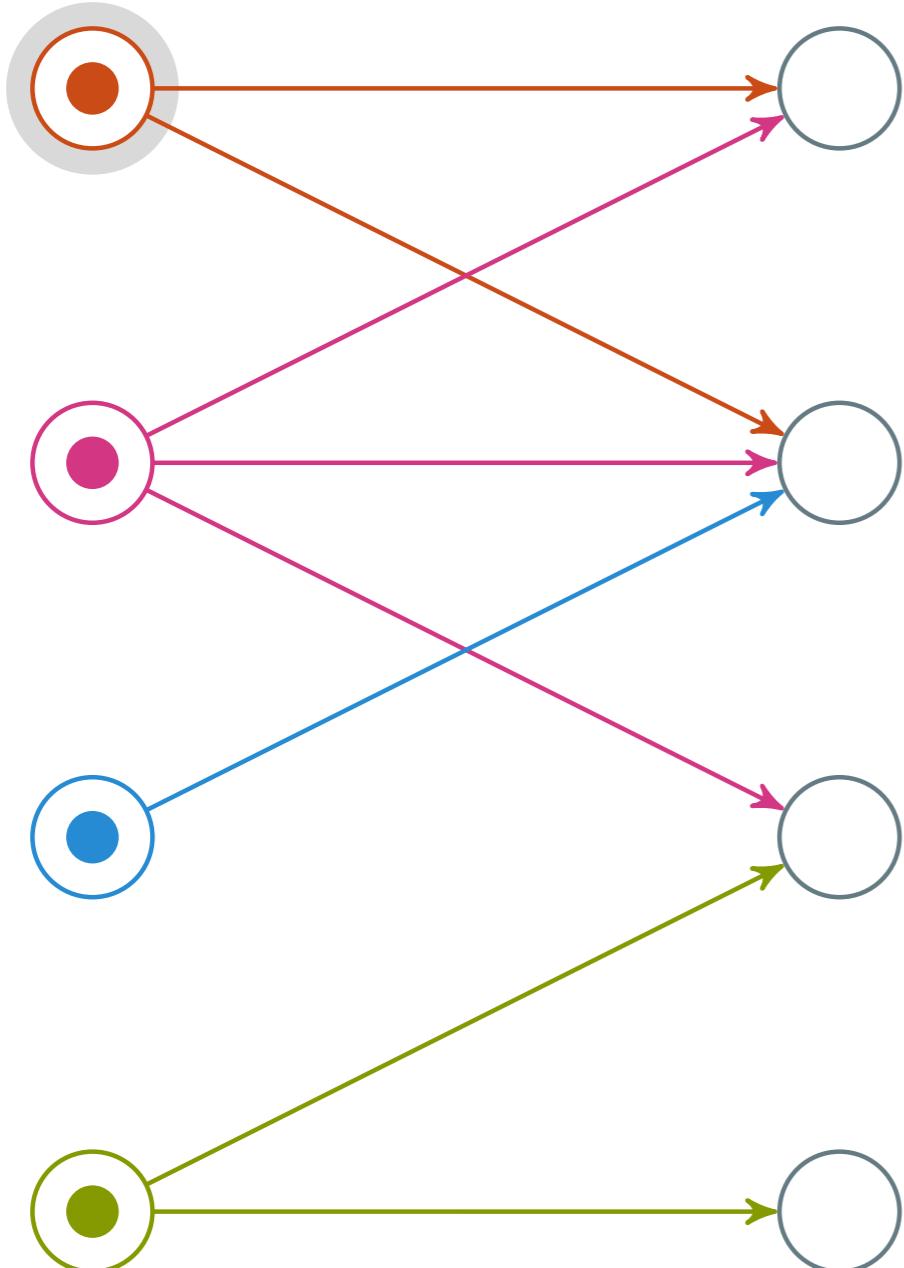
Men hva om vi for eksempel vil finne flest mulig par med kompatible donorer og resipienter for f.eks. nyretransplantasjon? En bedre løsning vil bety flere liv som reddes.

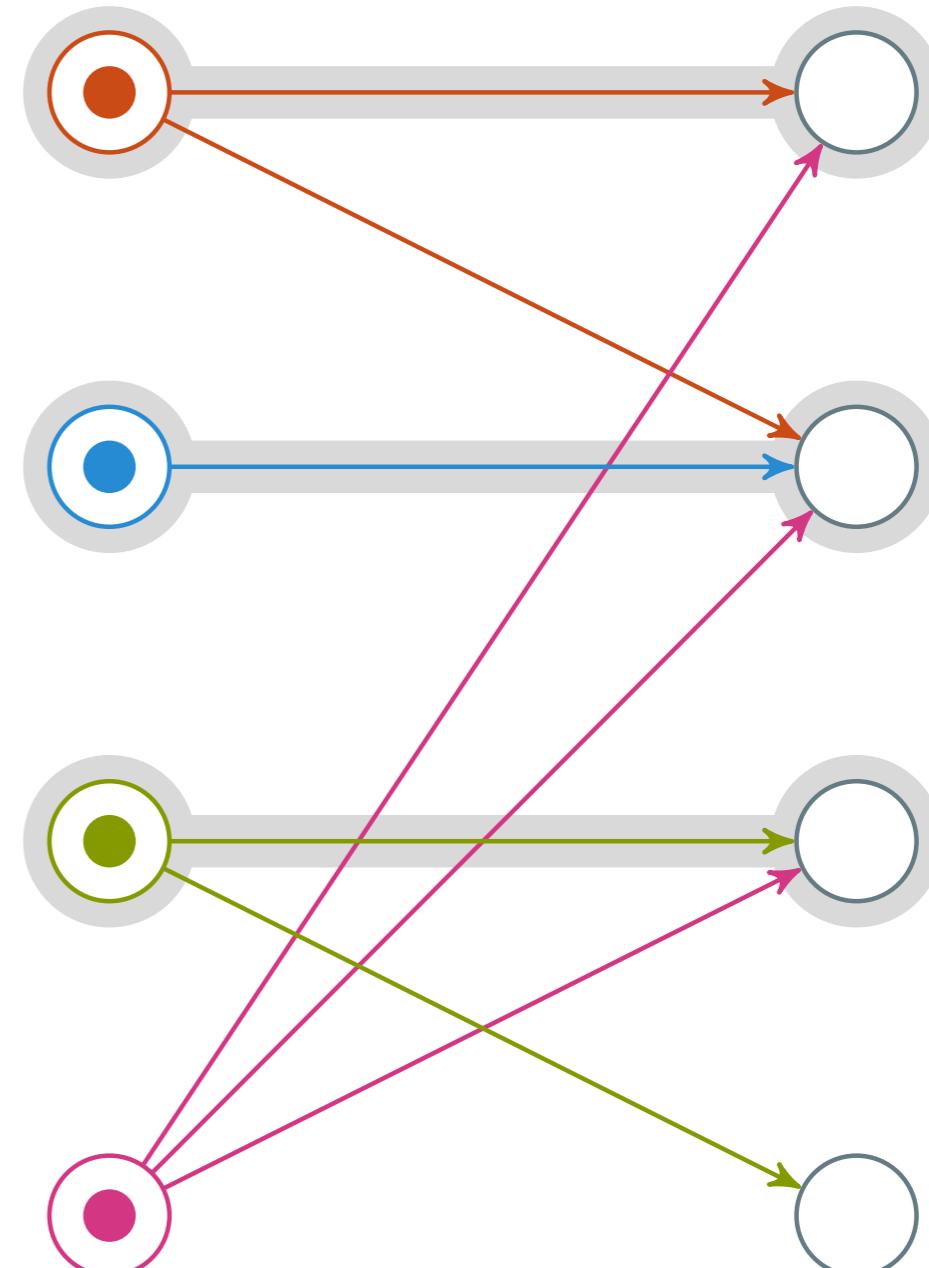
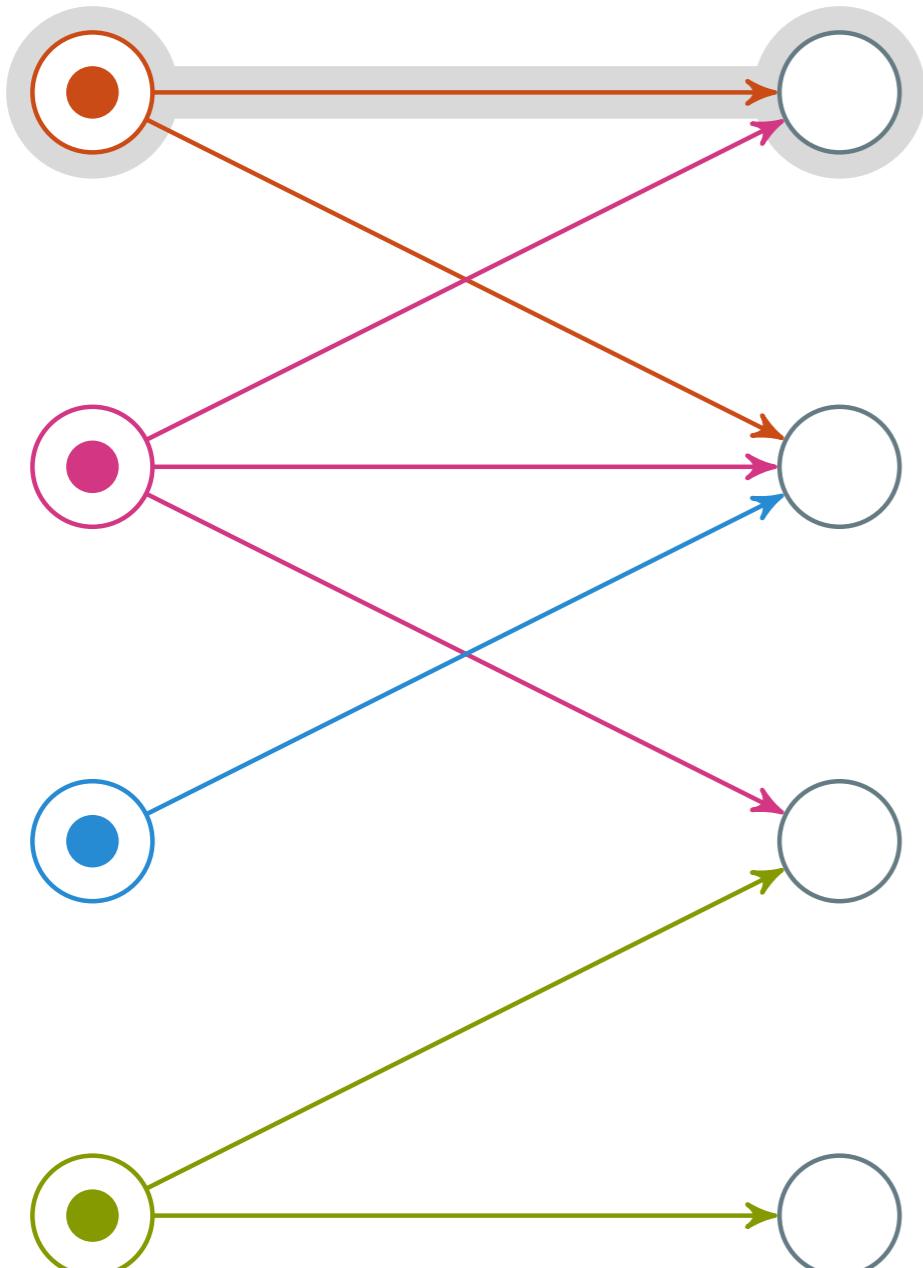
Her er det kanskje mindre opplagt hvordan vi skal gå frem – og om det er greit å prøve alle muligheter.

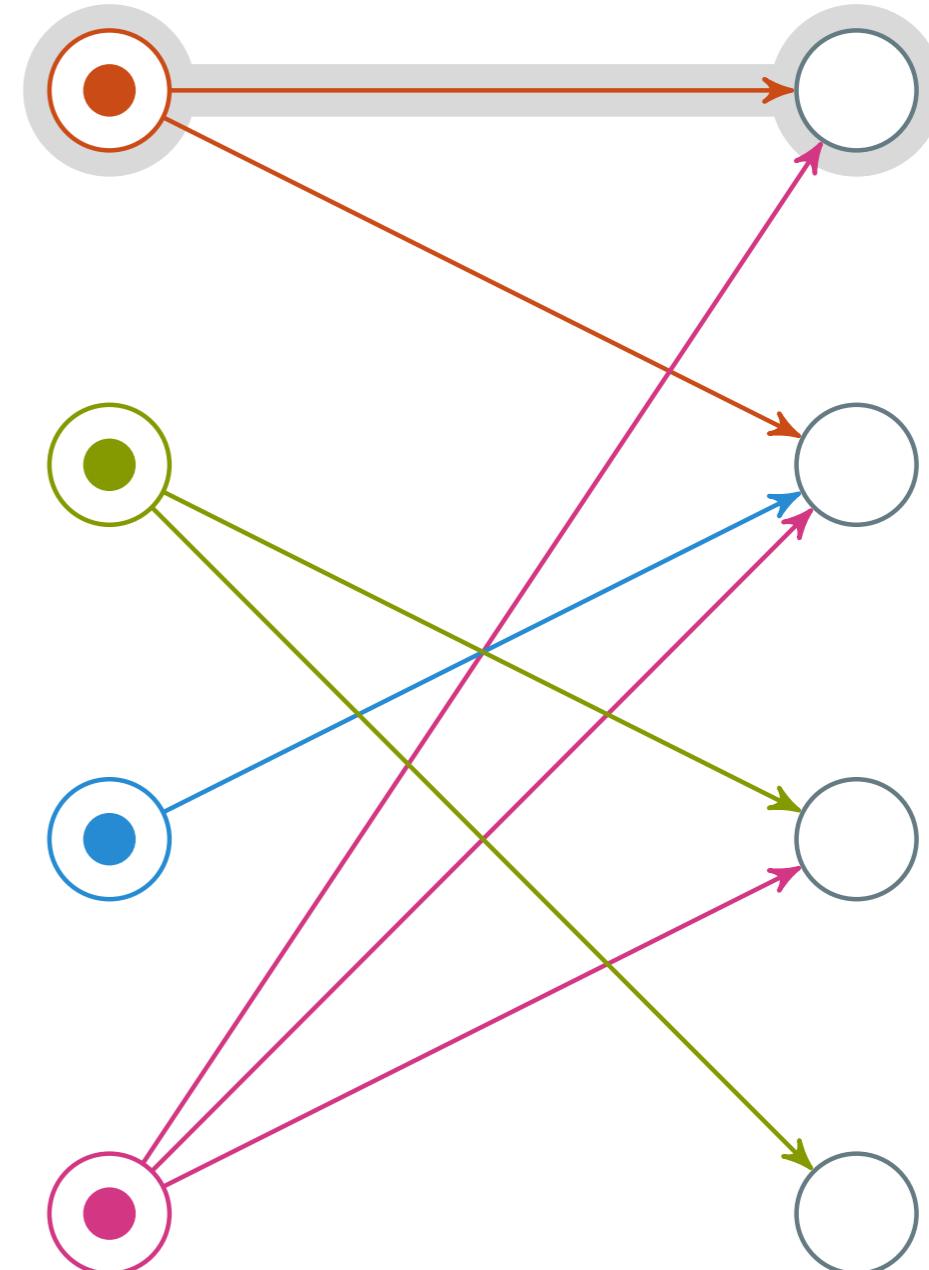
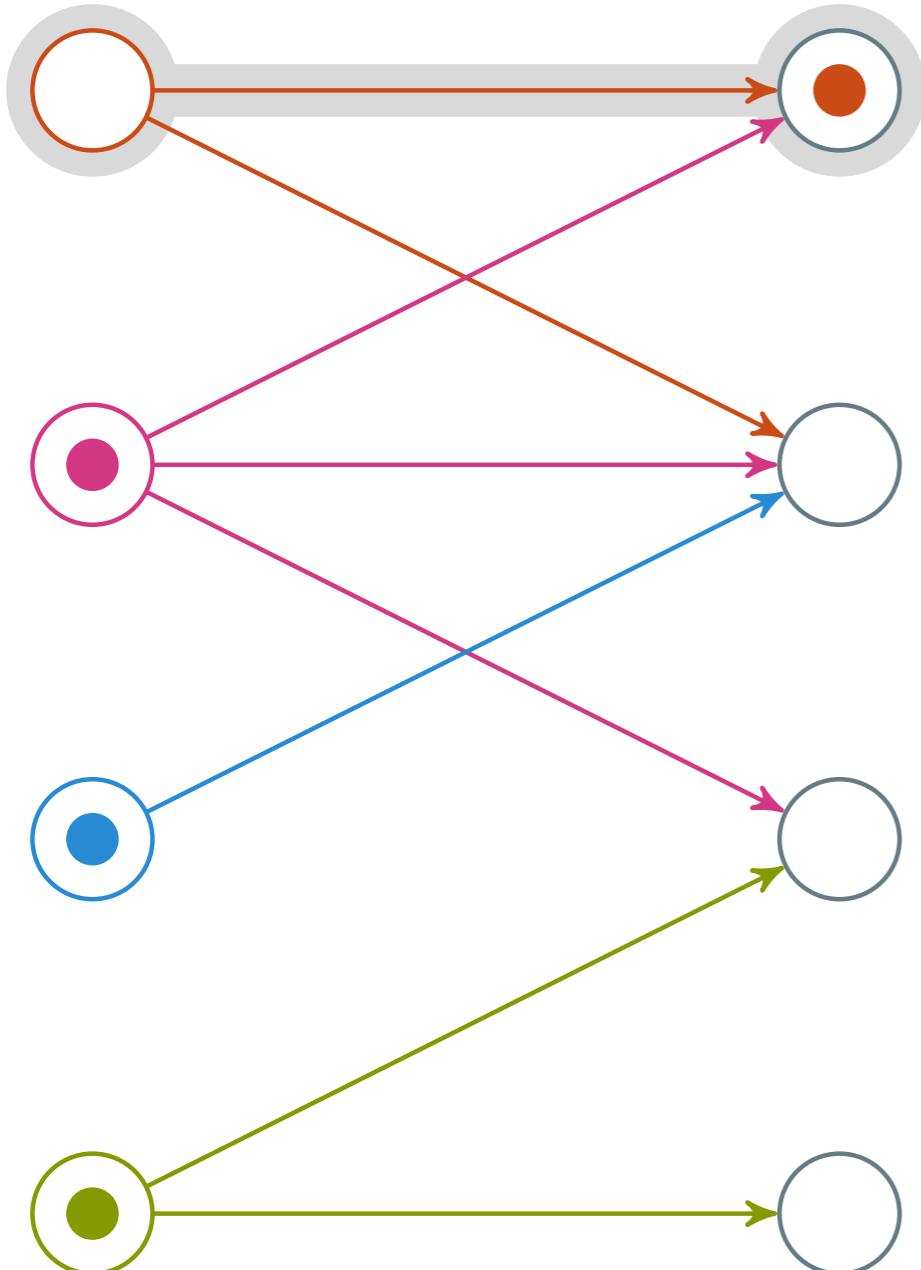


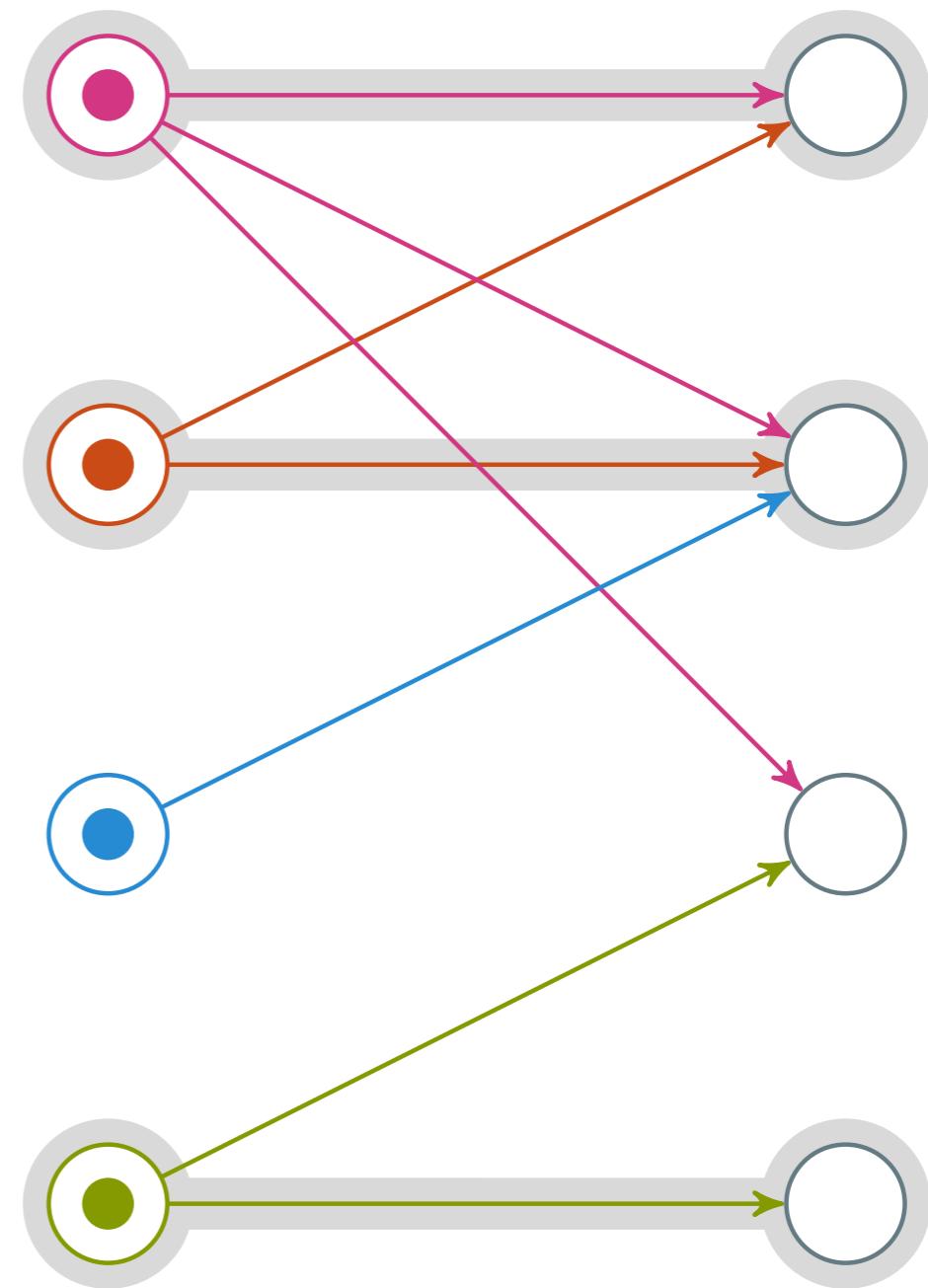
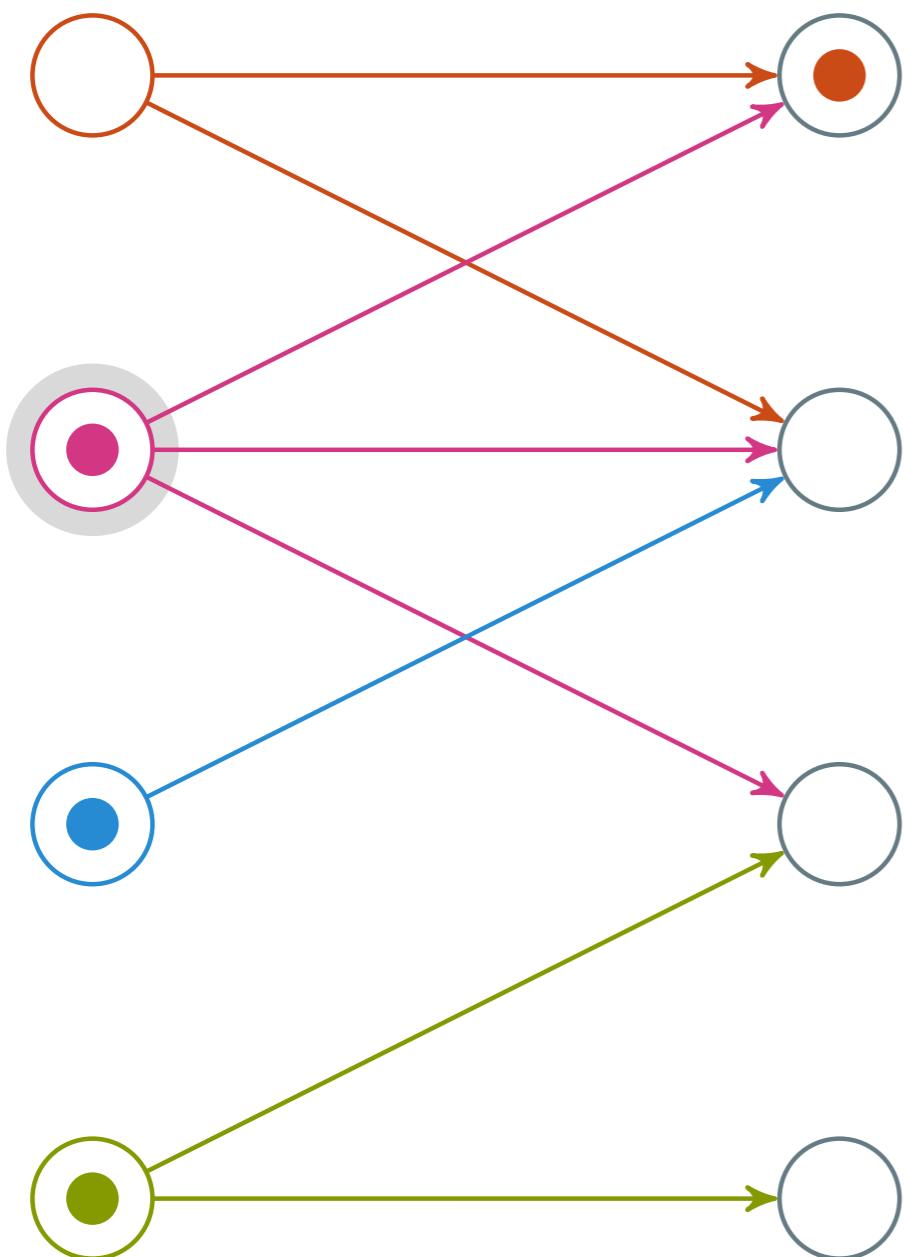


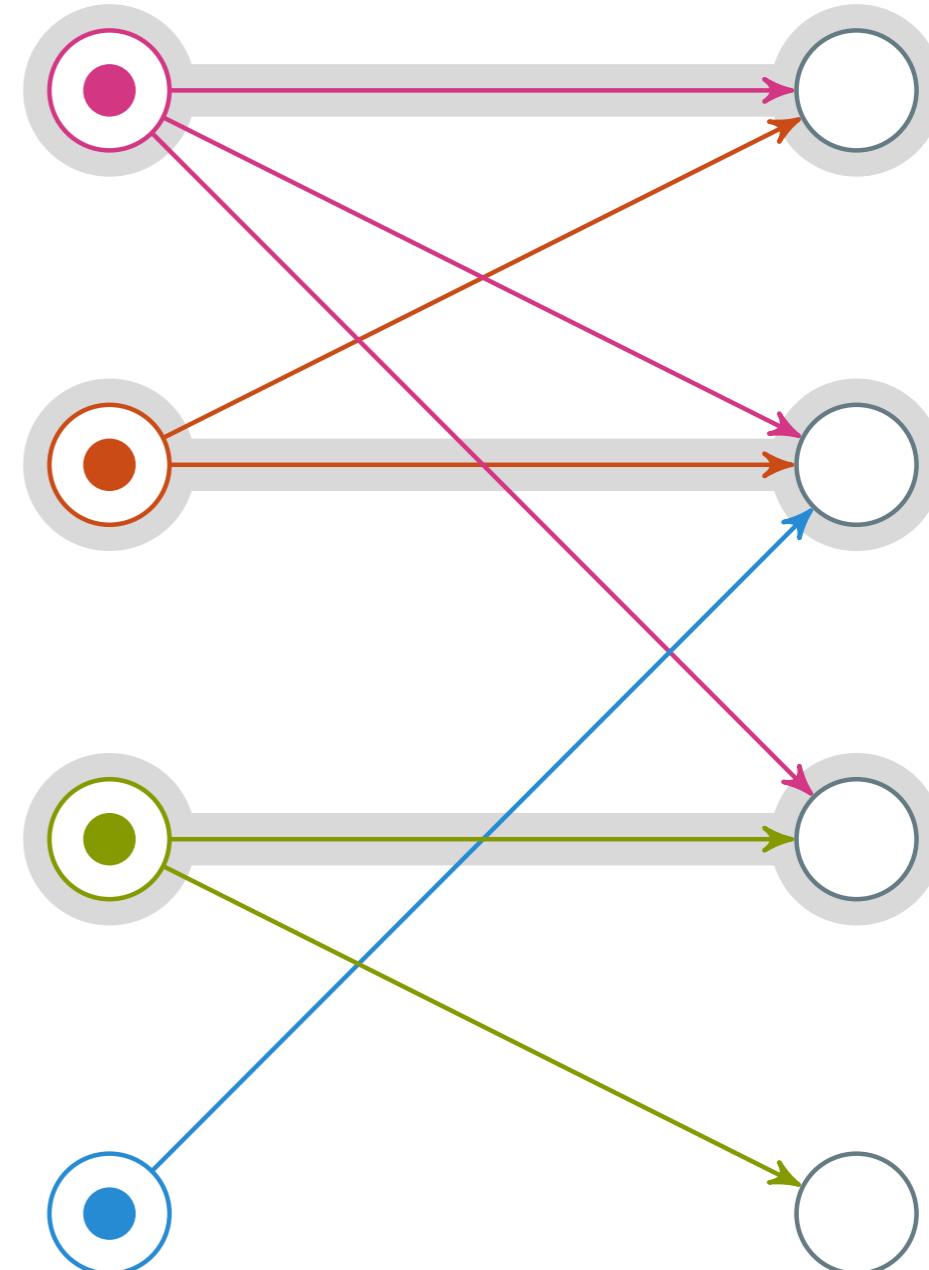
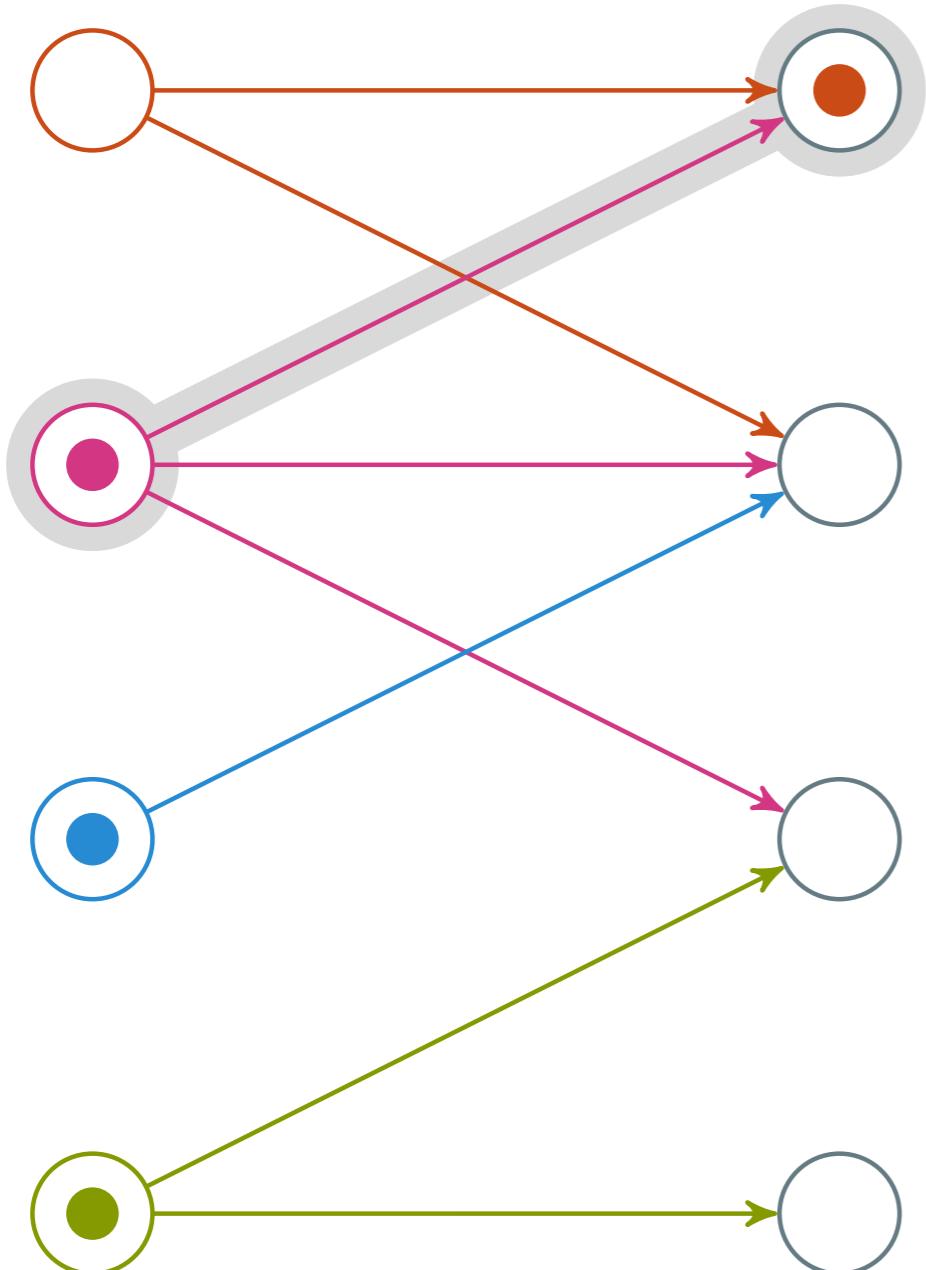
La oss sammenligne Ford-Fulkerson-algoritmen (som dere lærer om i forelesning 12) med såkalt «brute force» – å teste alle muligheter. La oss si at vi har fire donorer og fire resipienter.

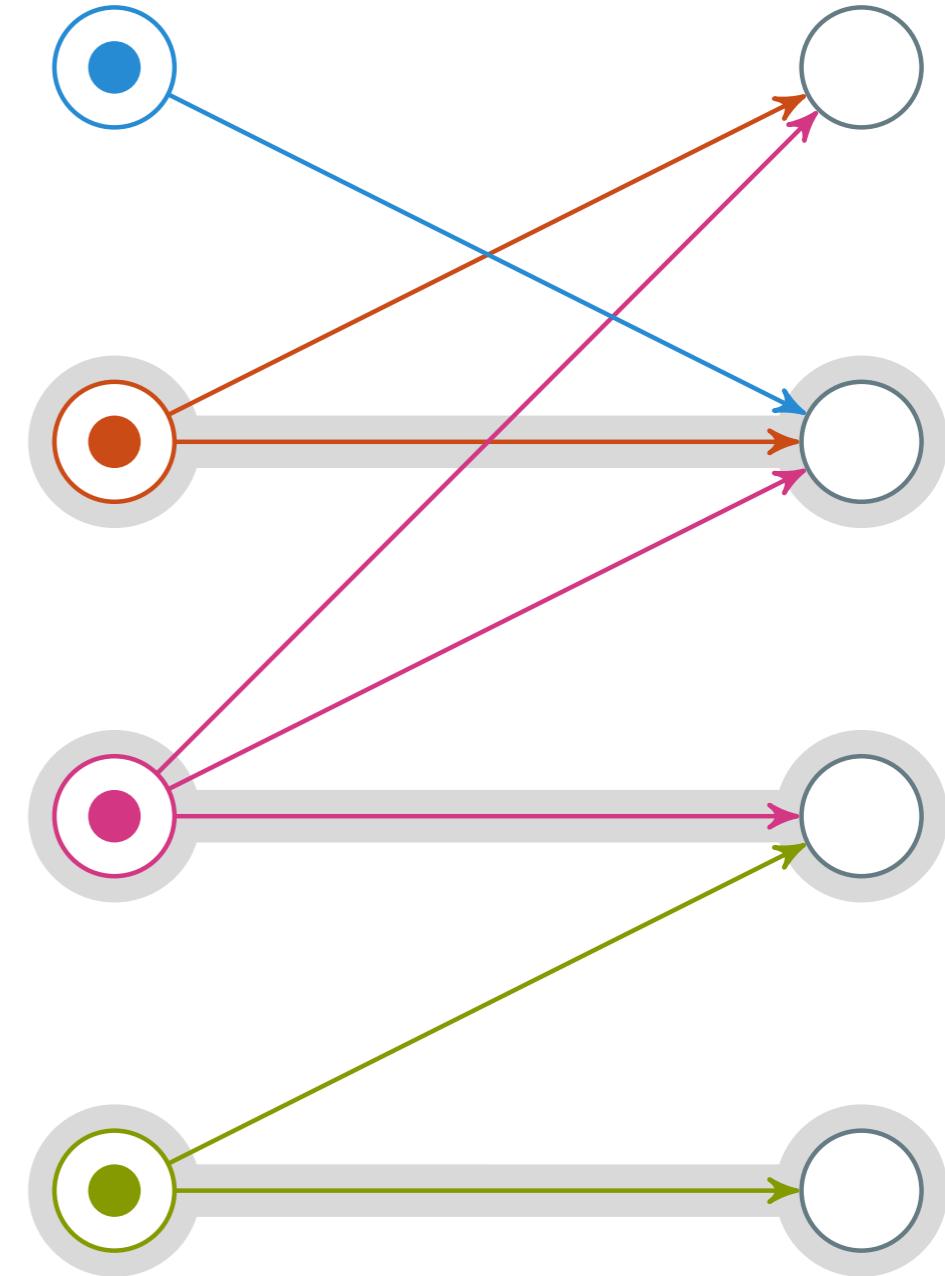
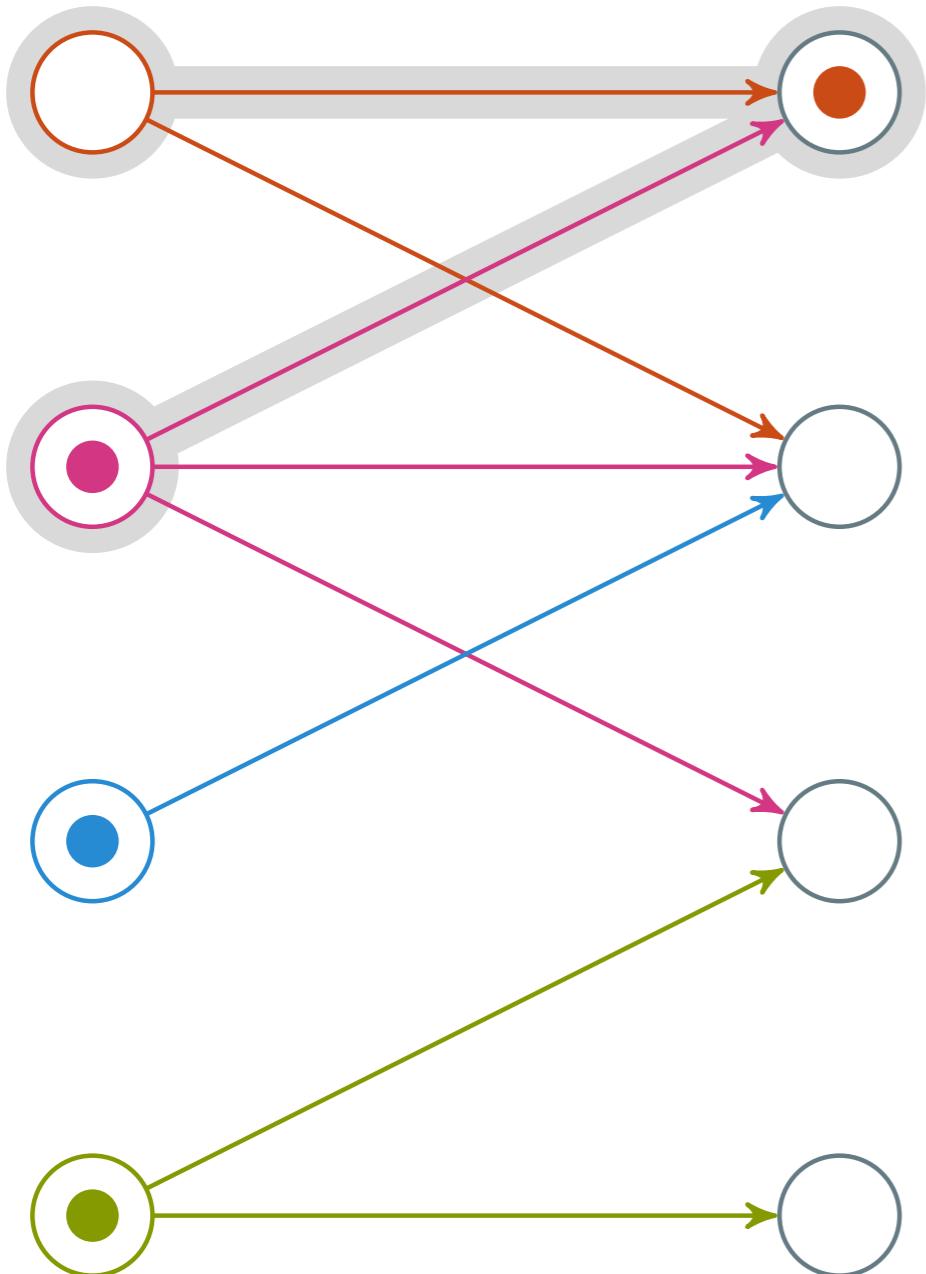


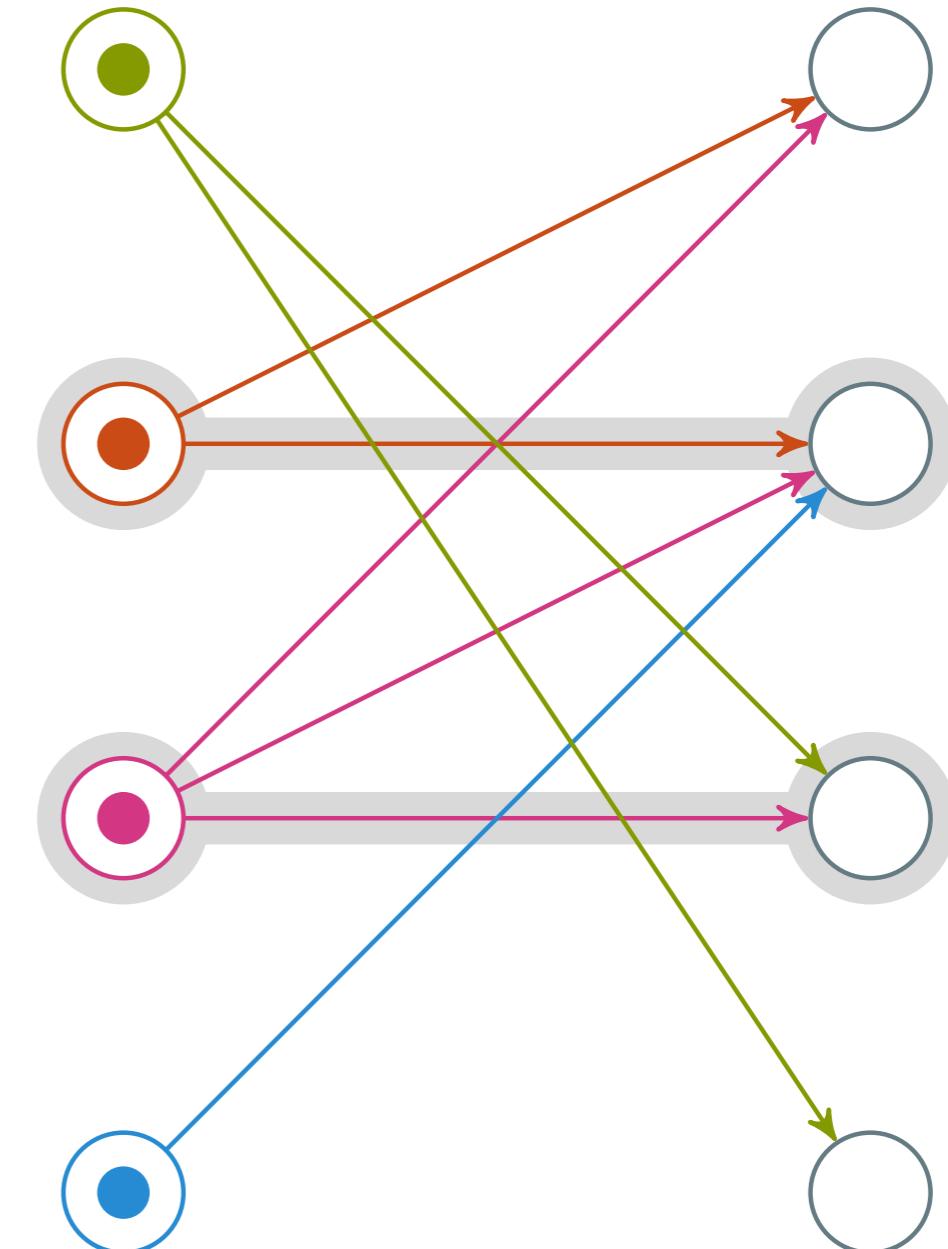
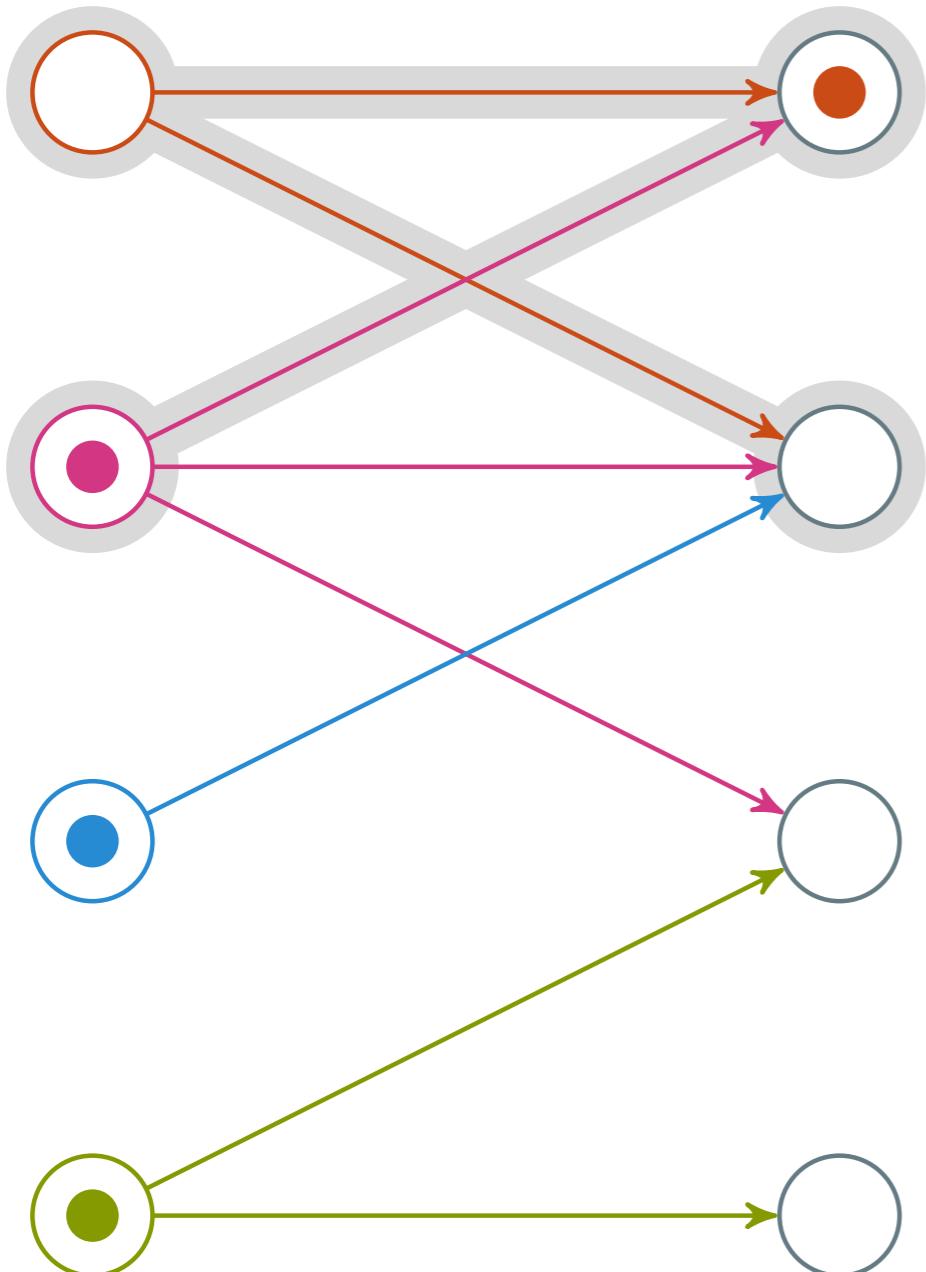


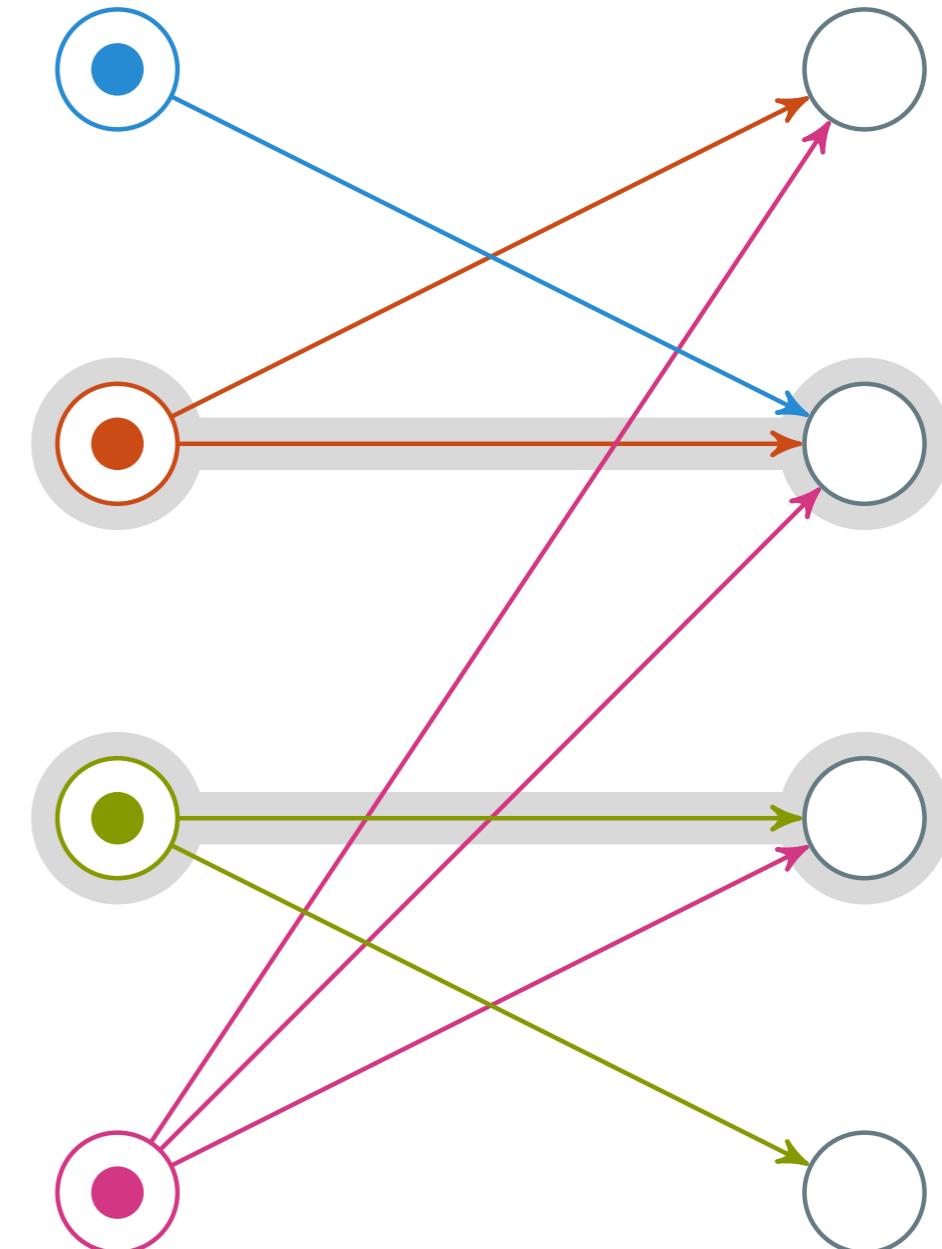
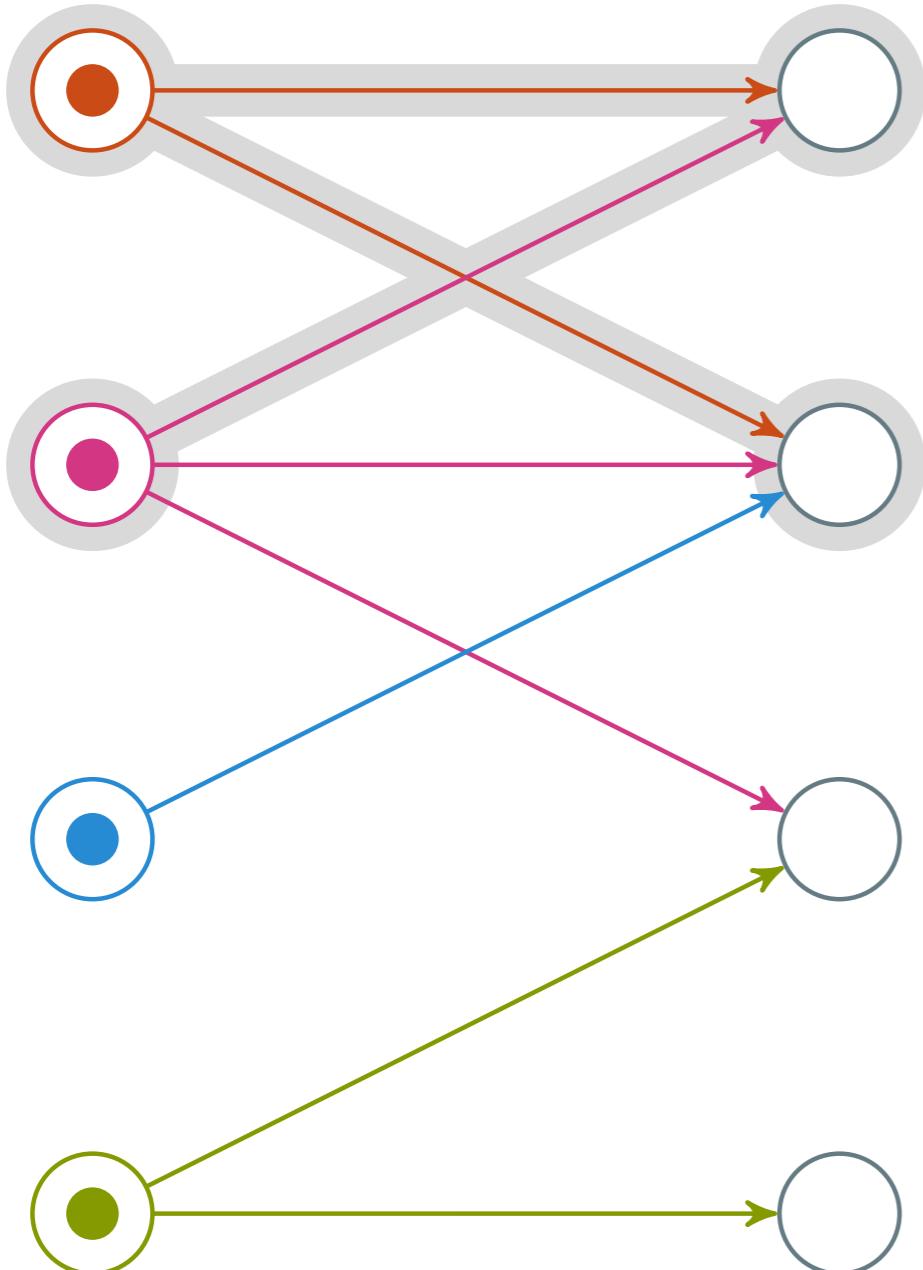


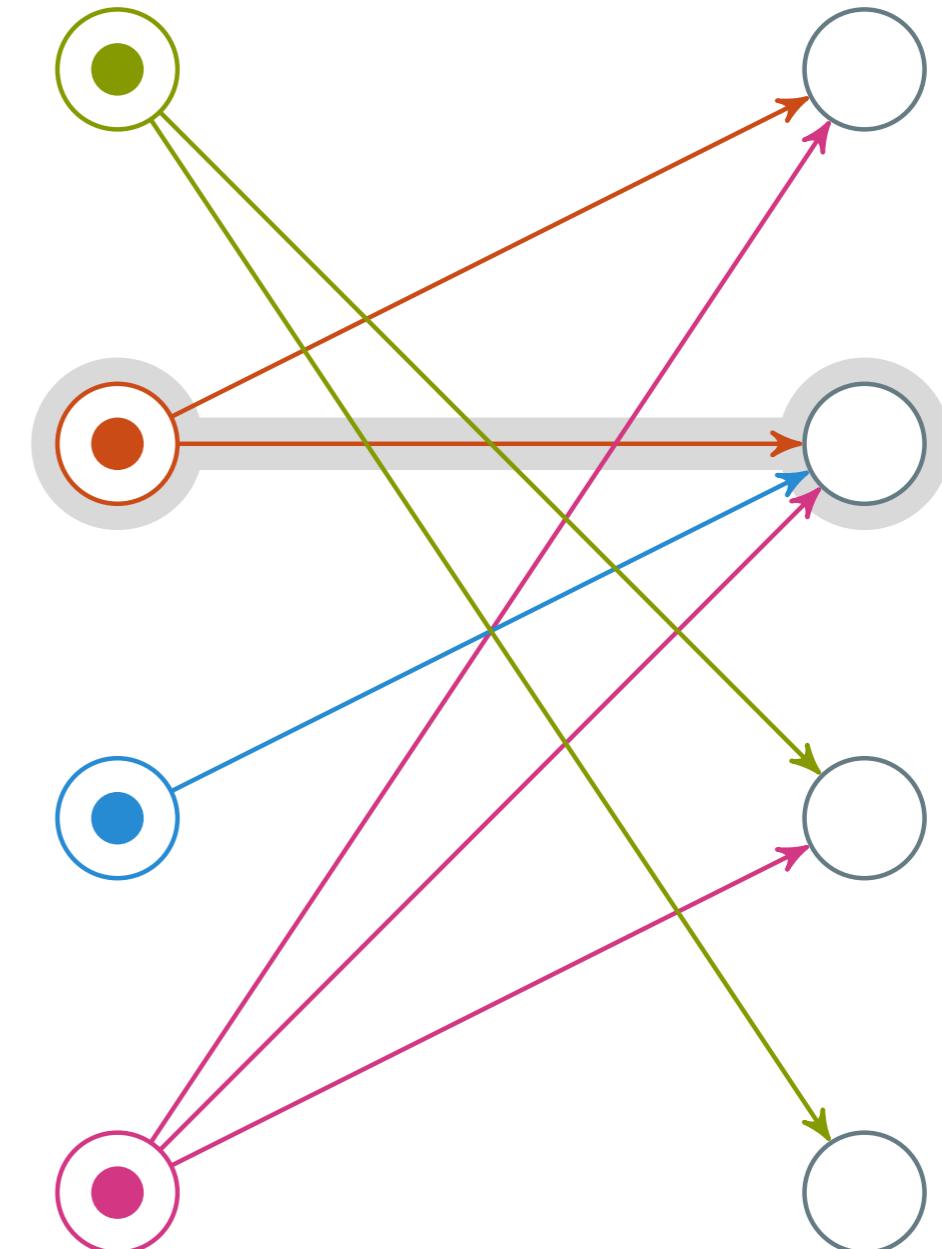
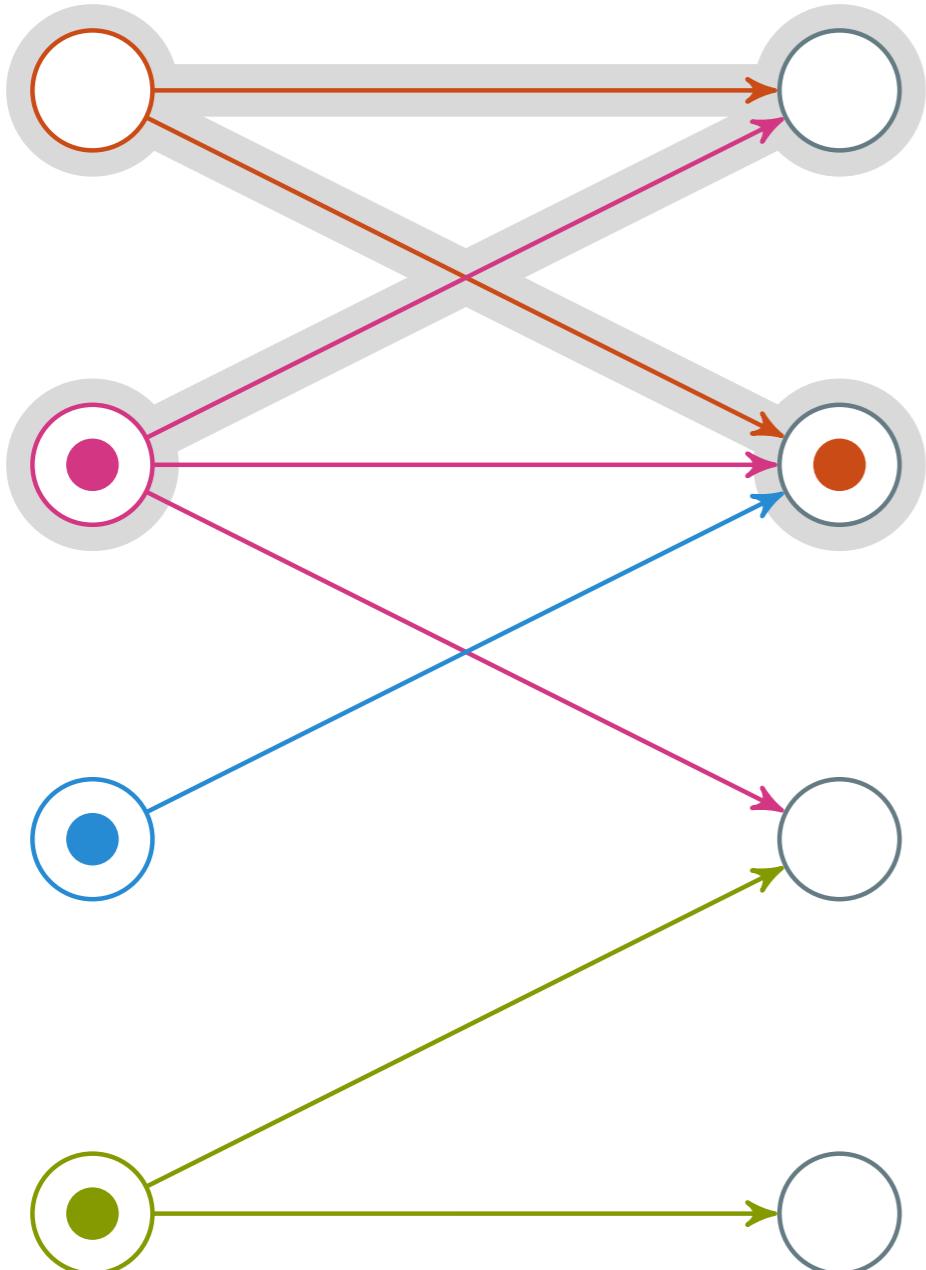


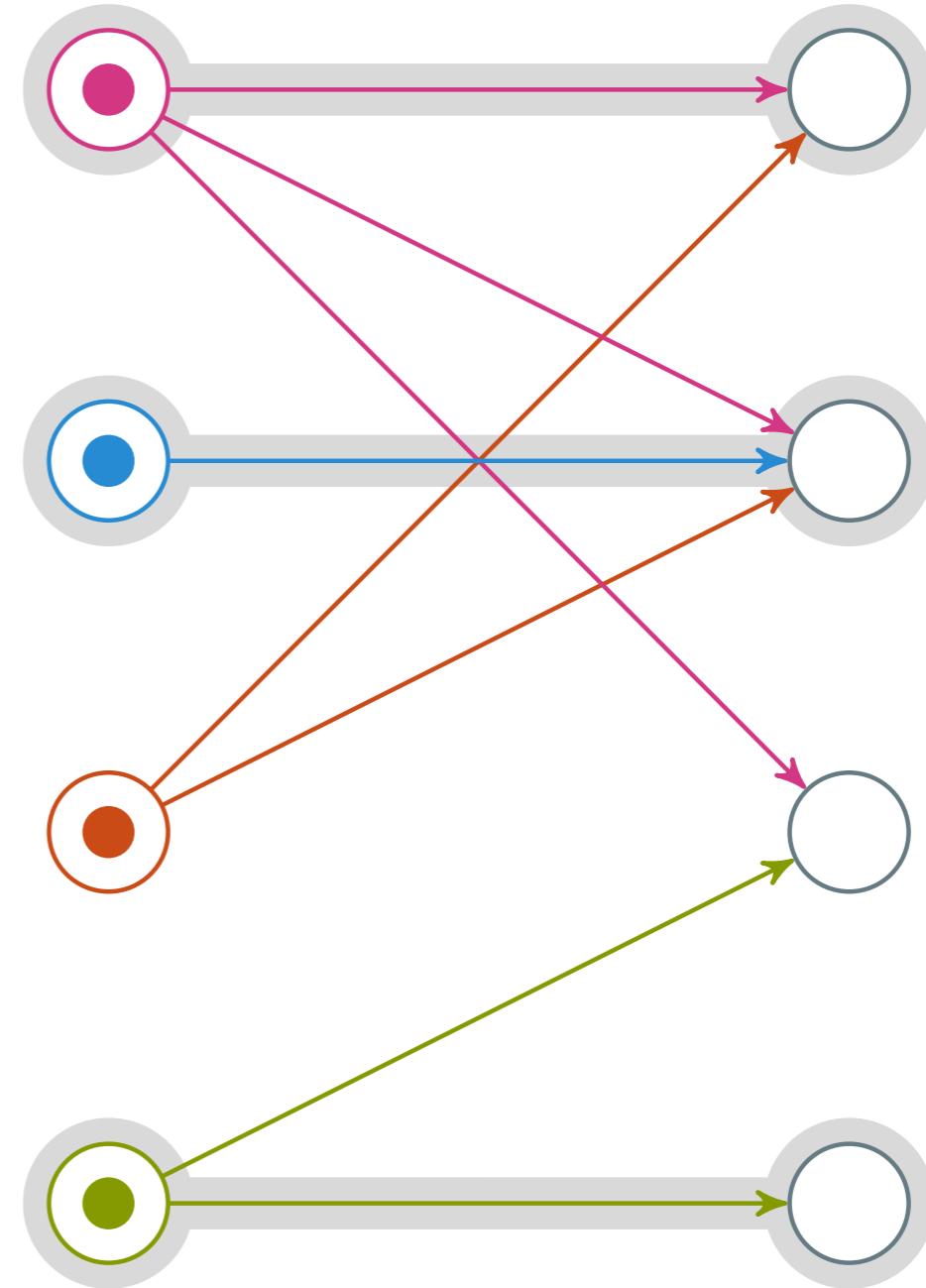
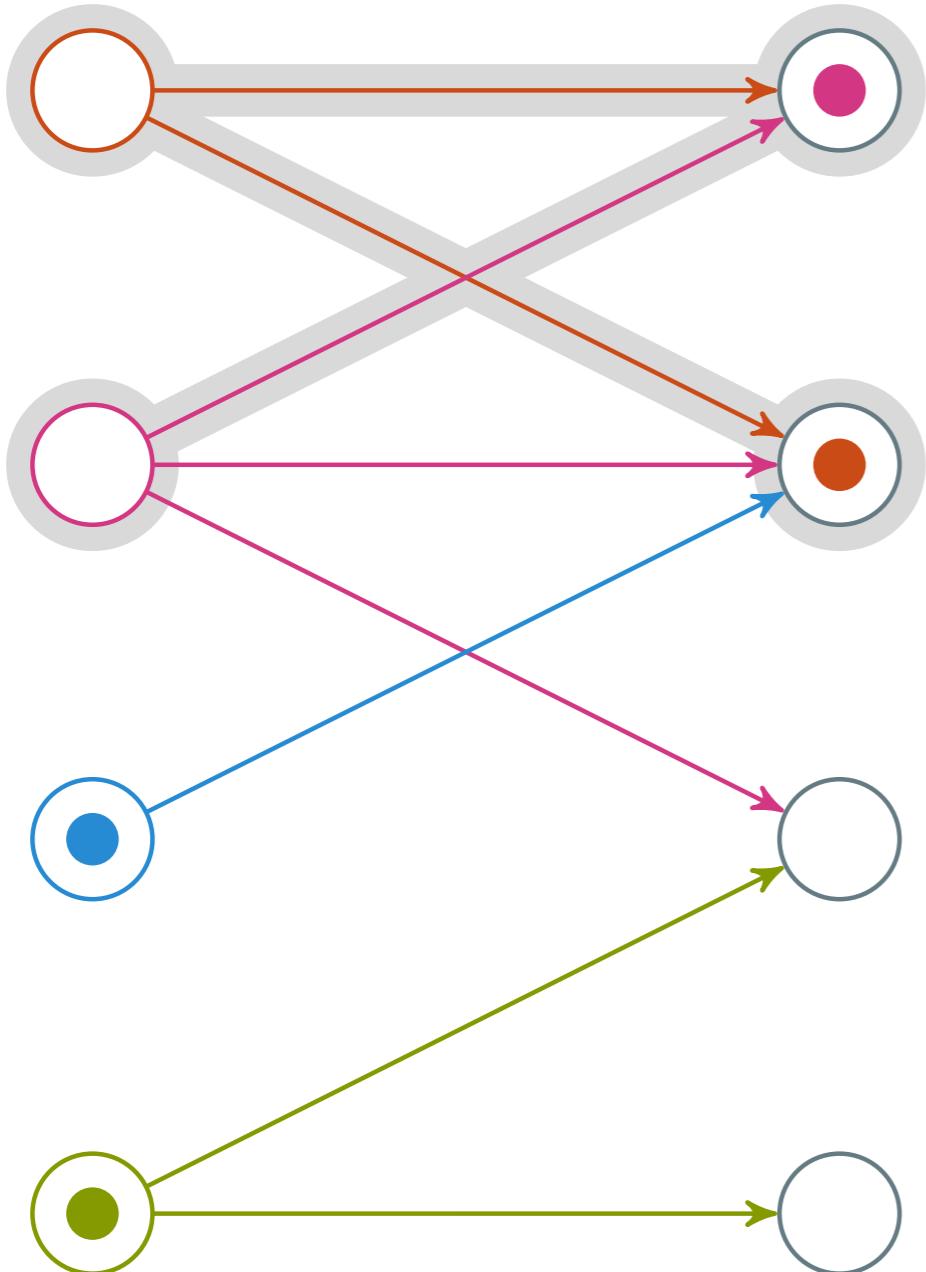


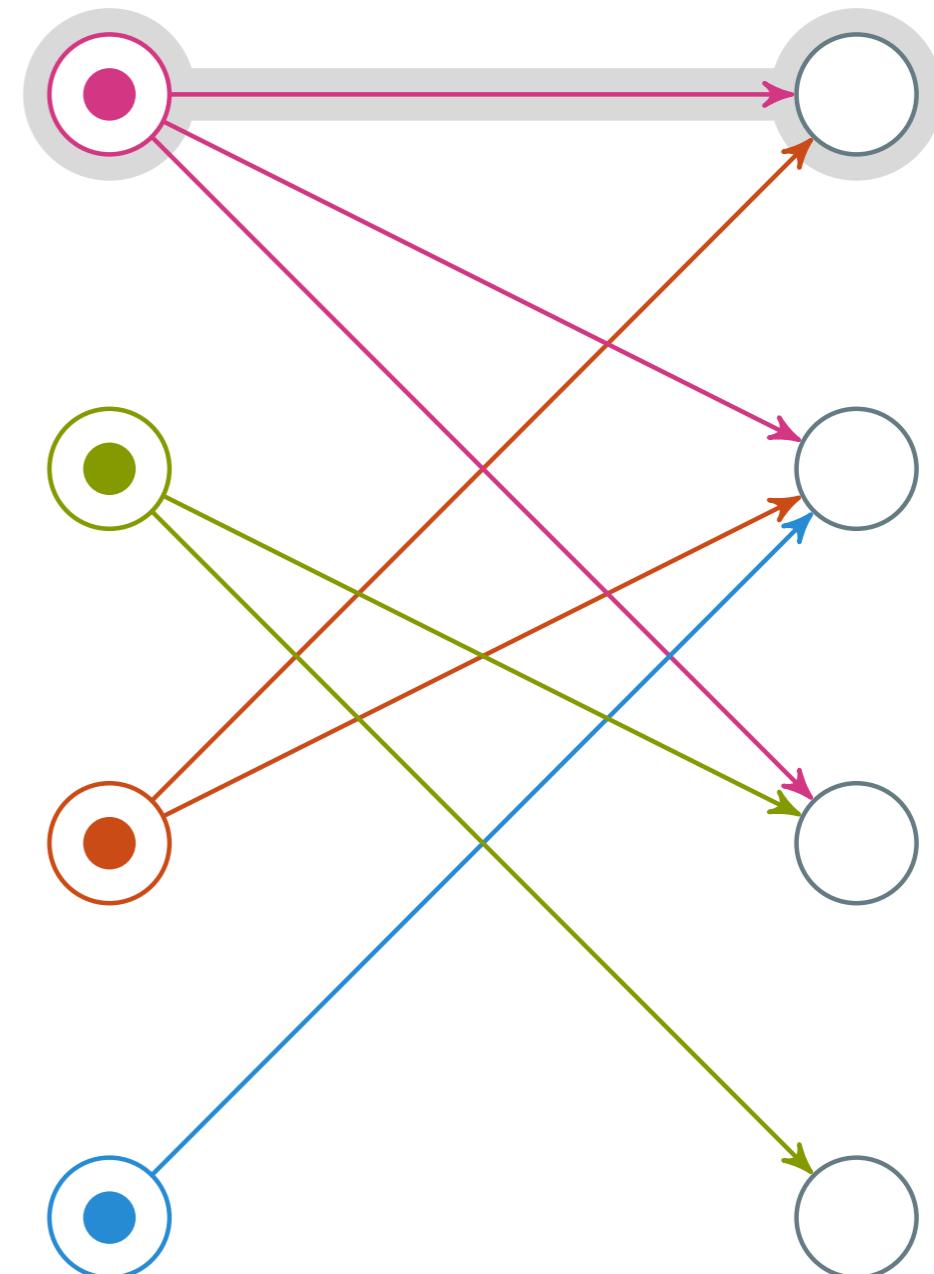
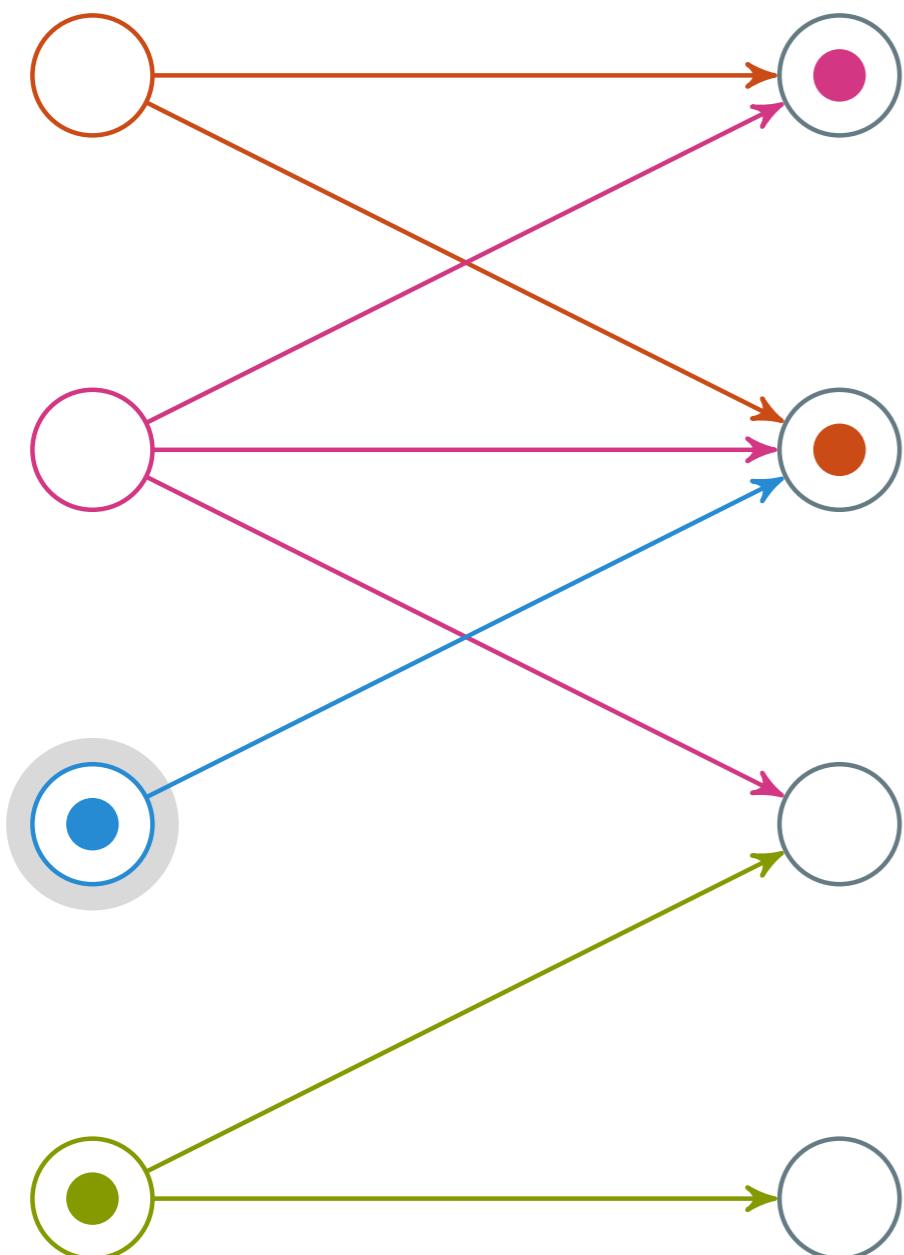


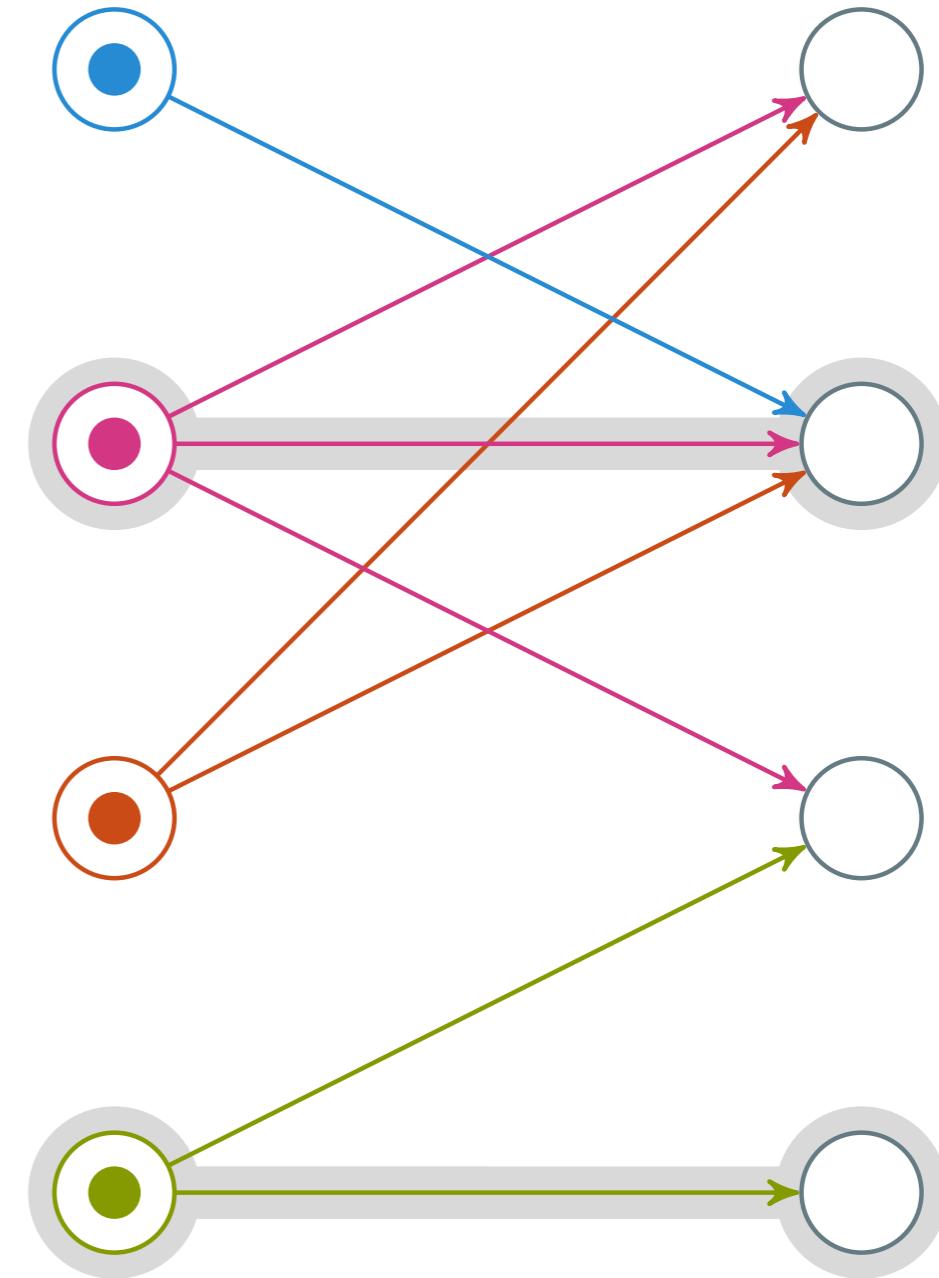
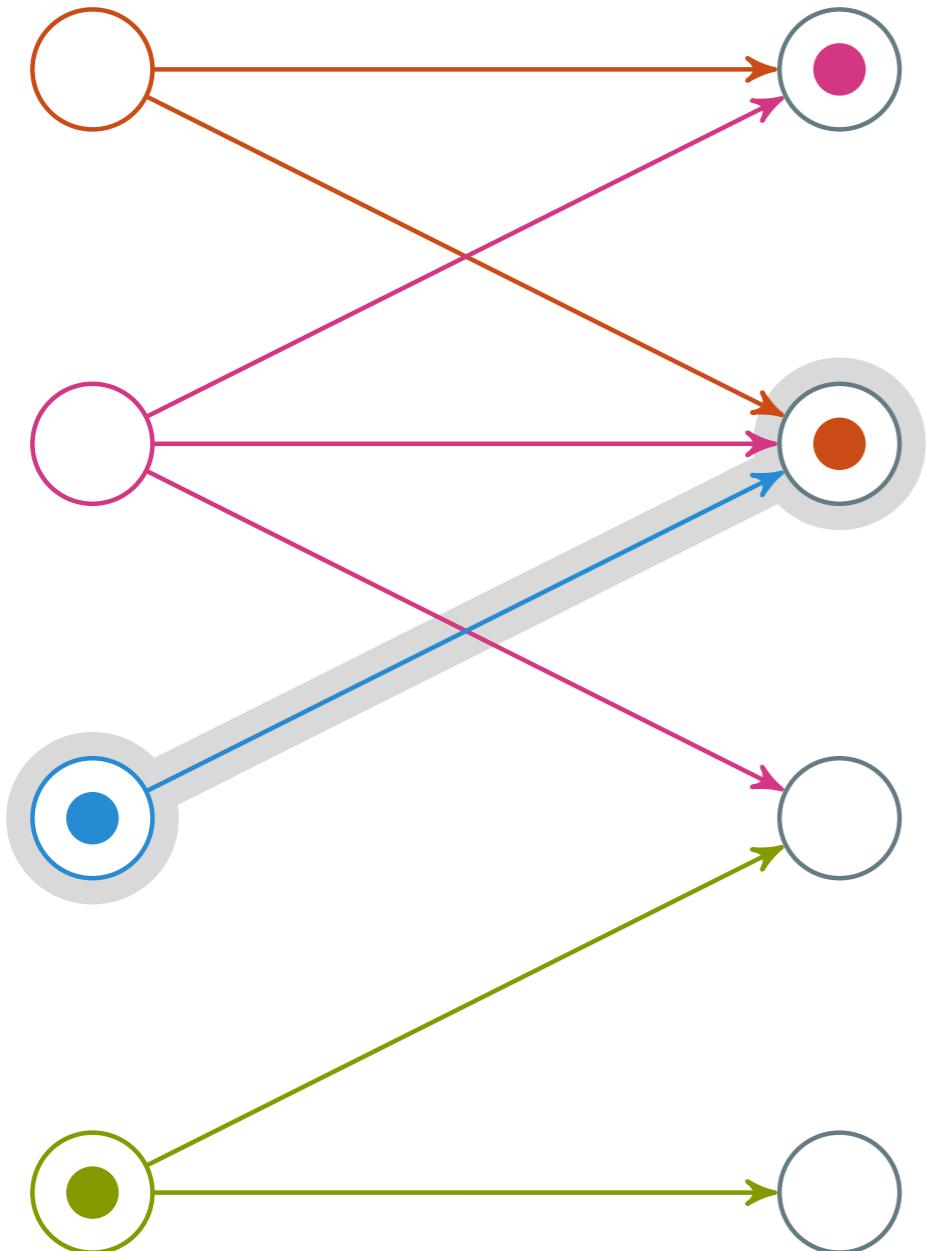


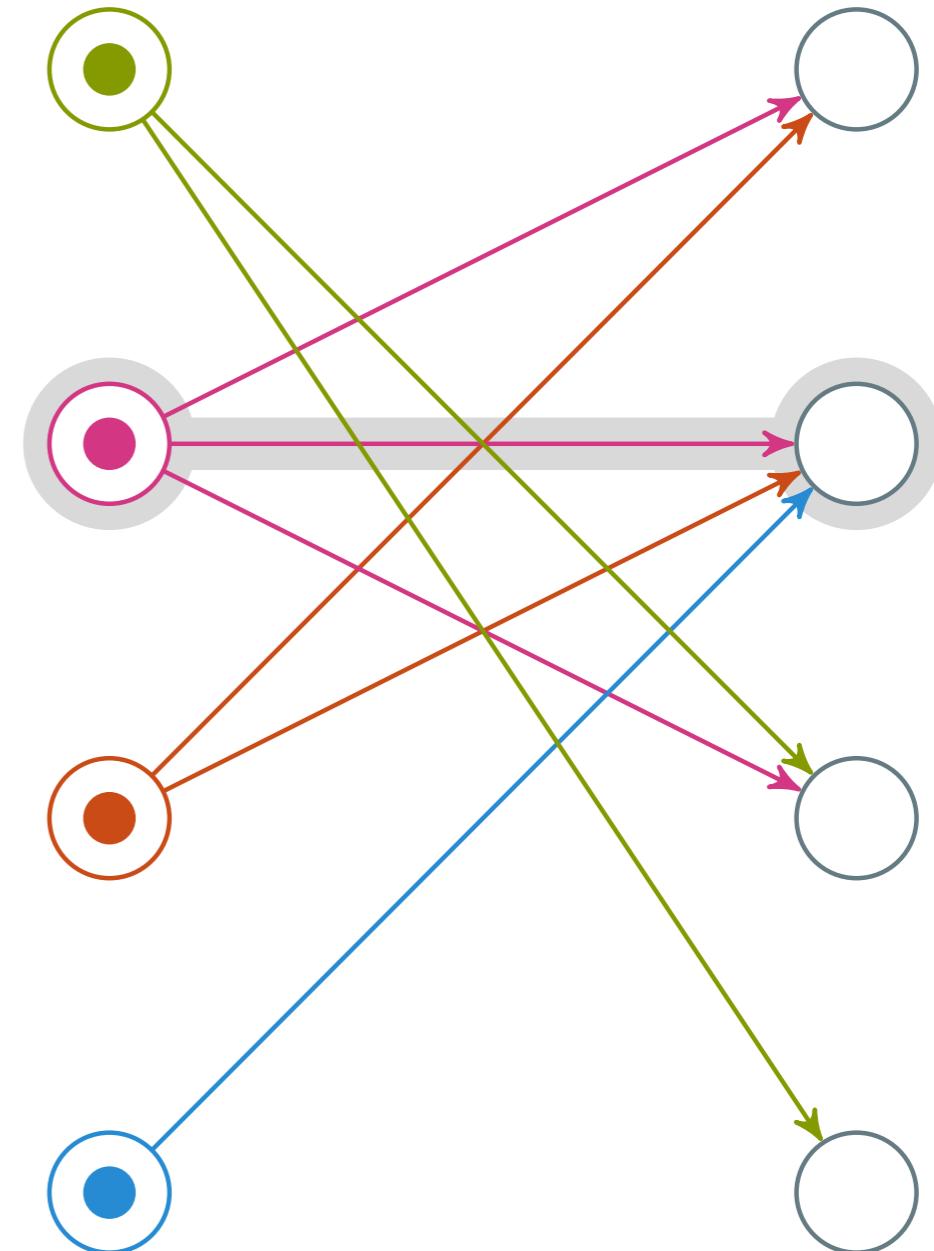
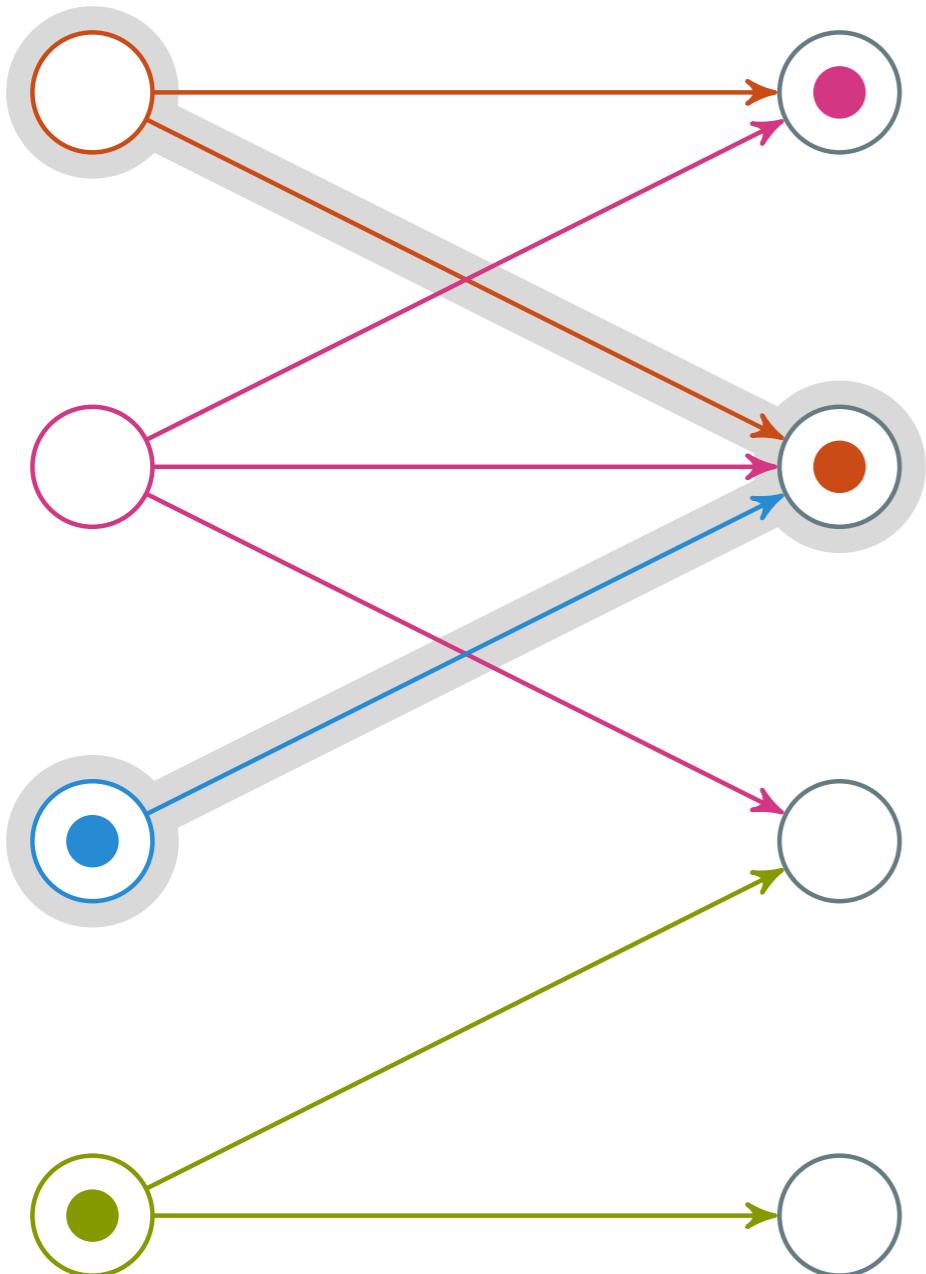


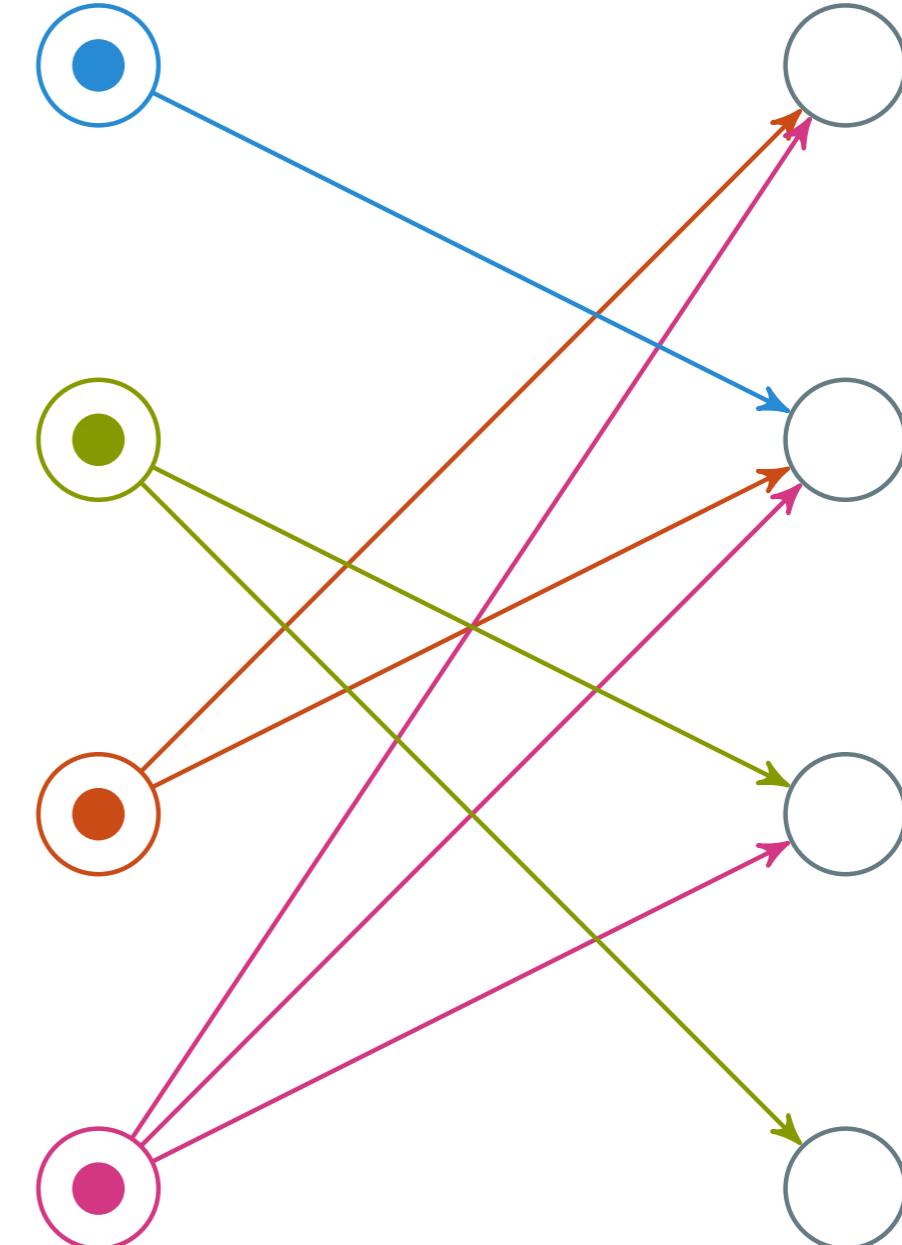
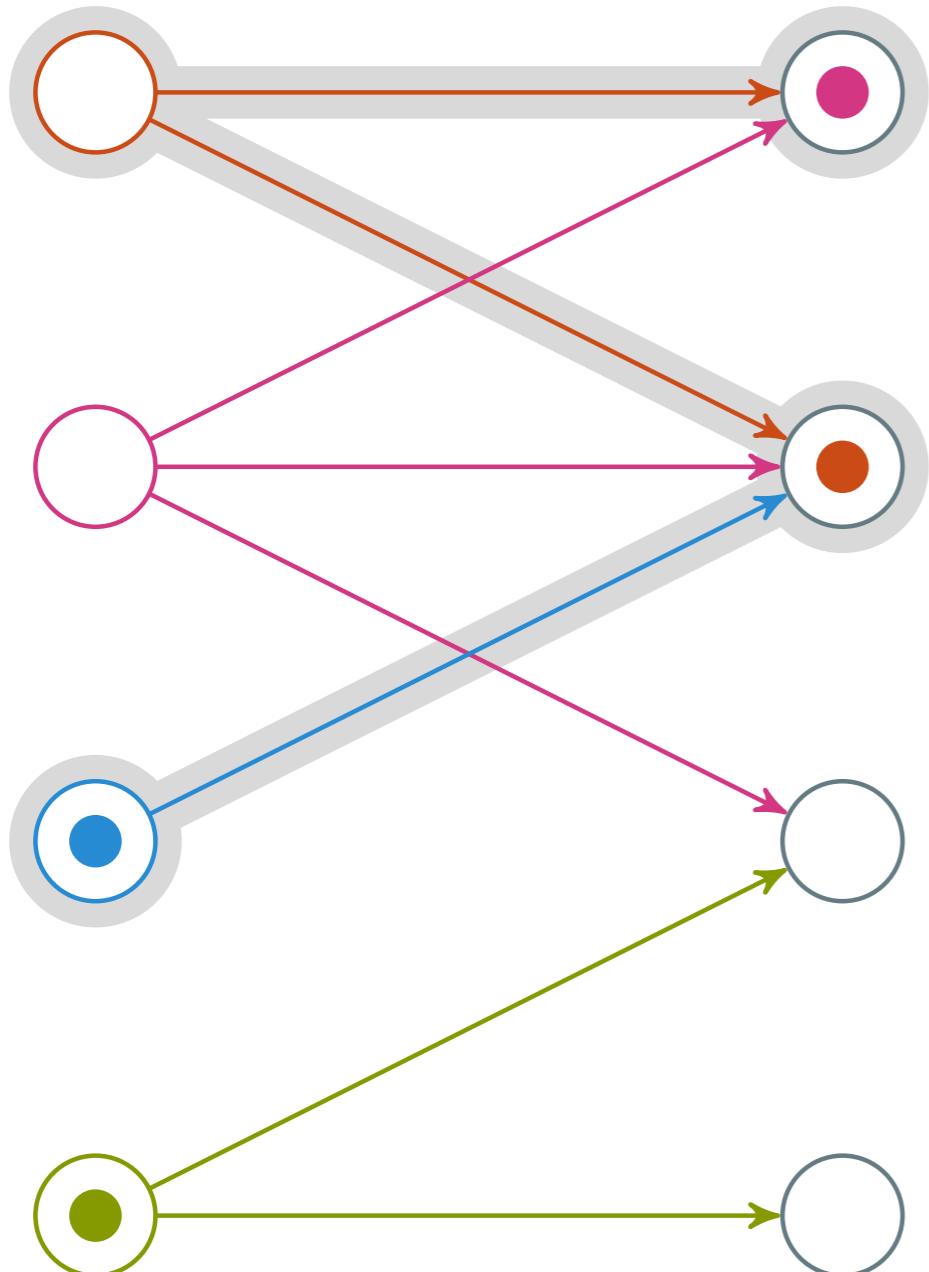


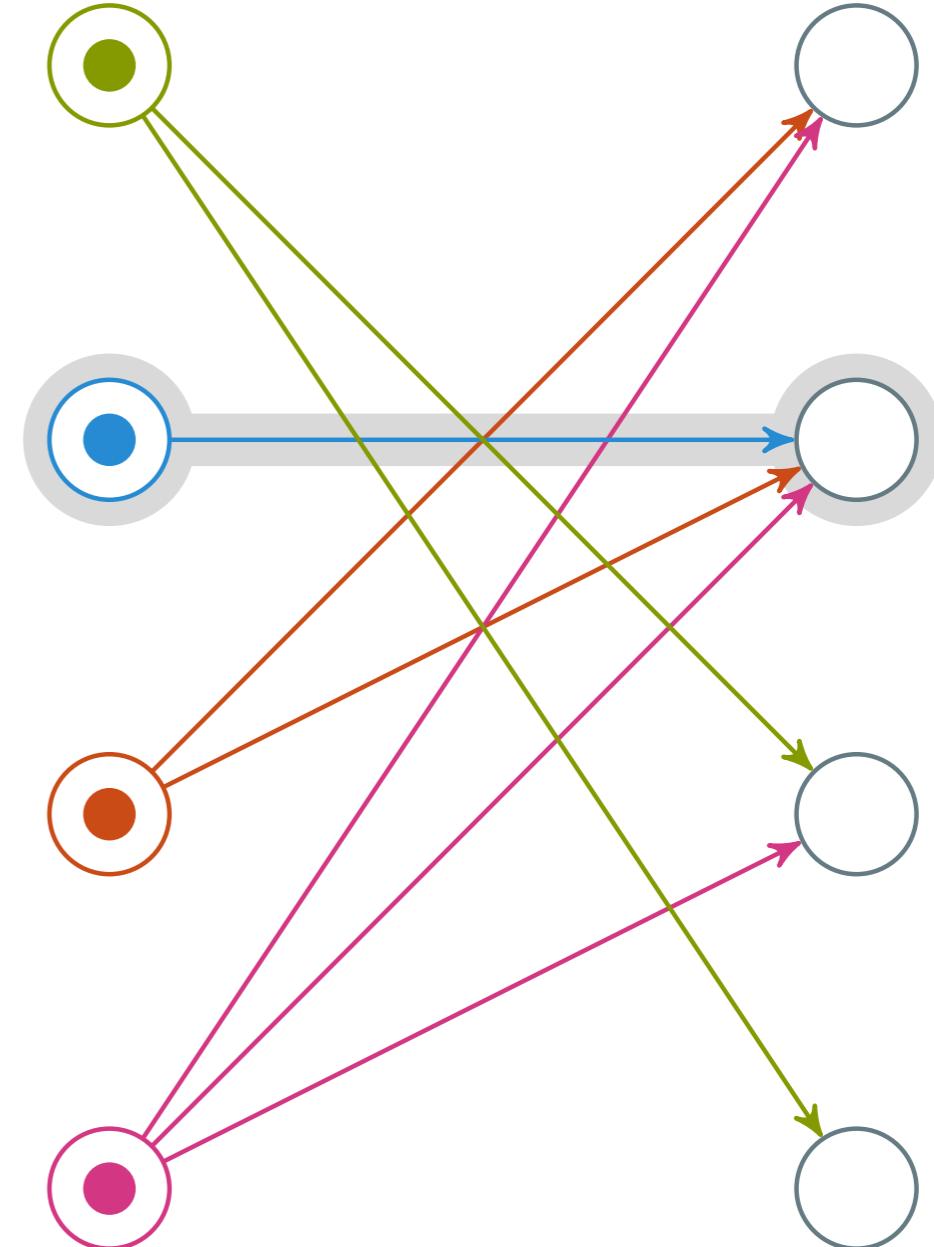
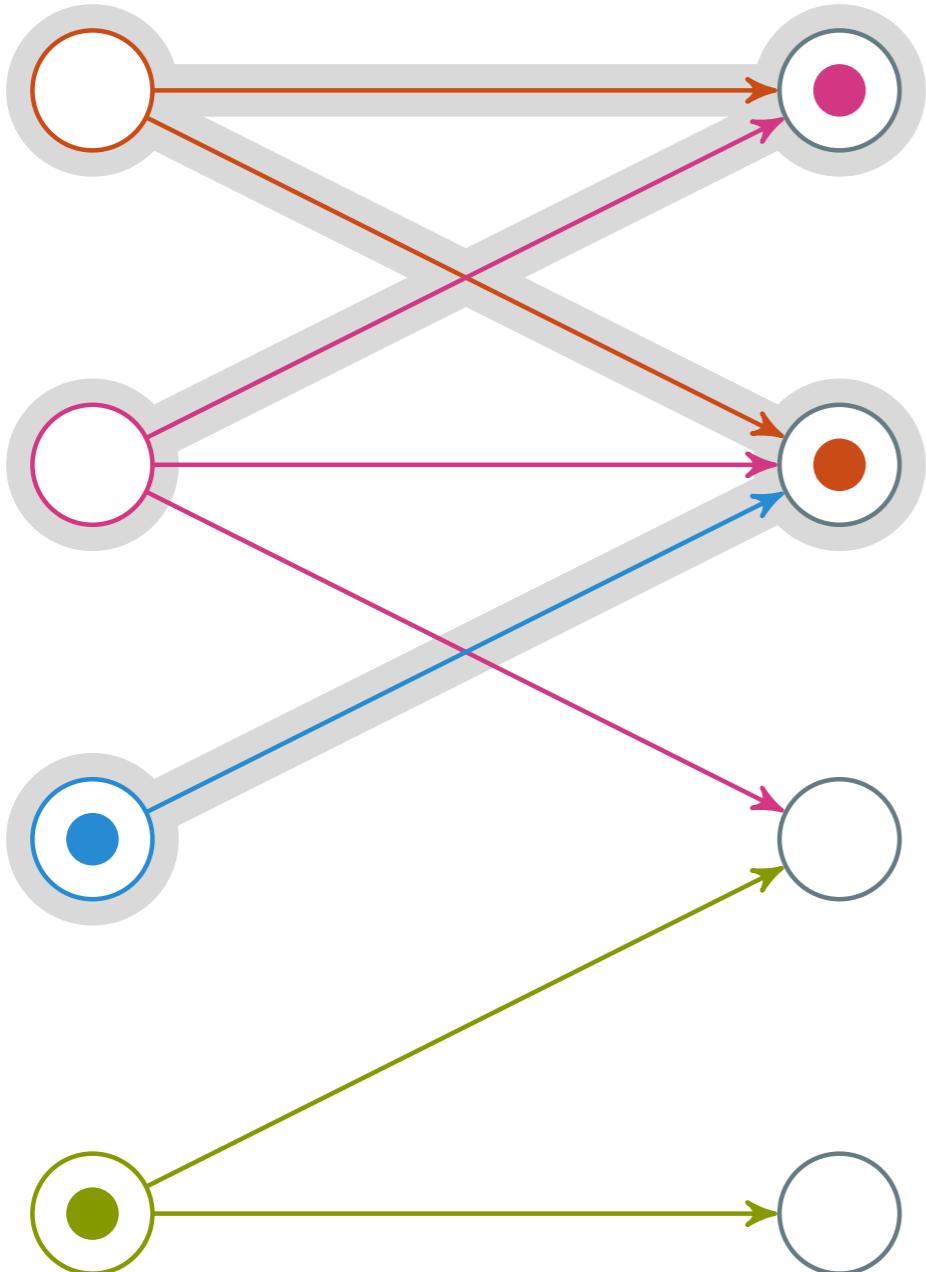


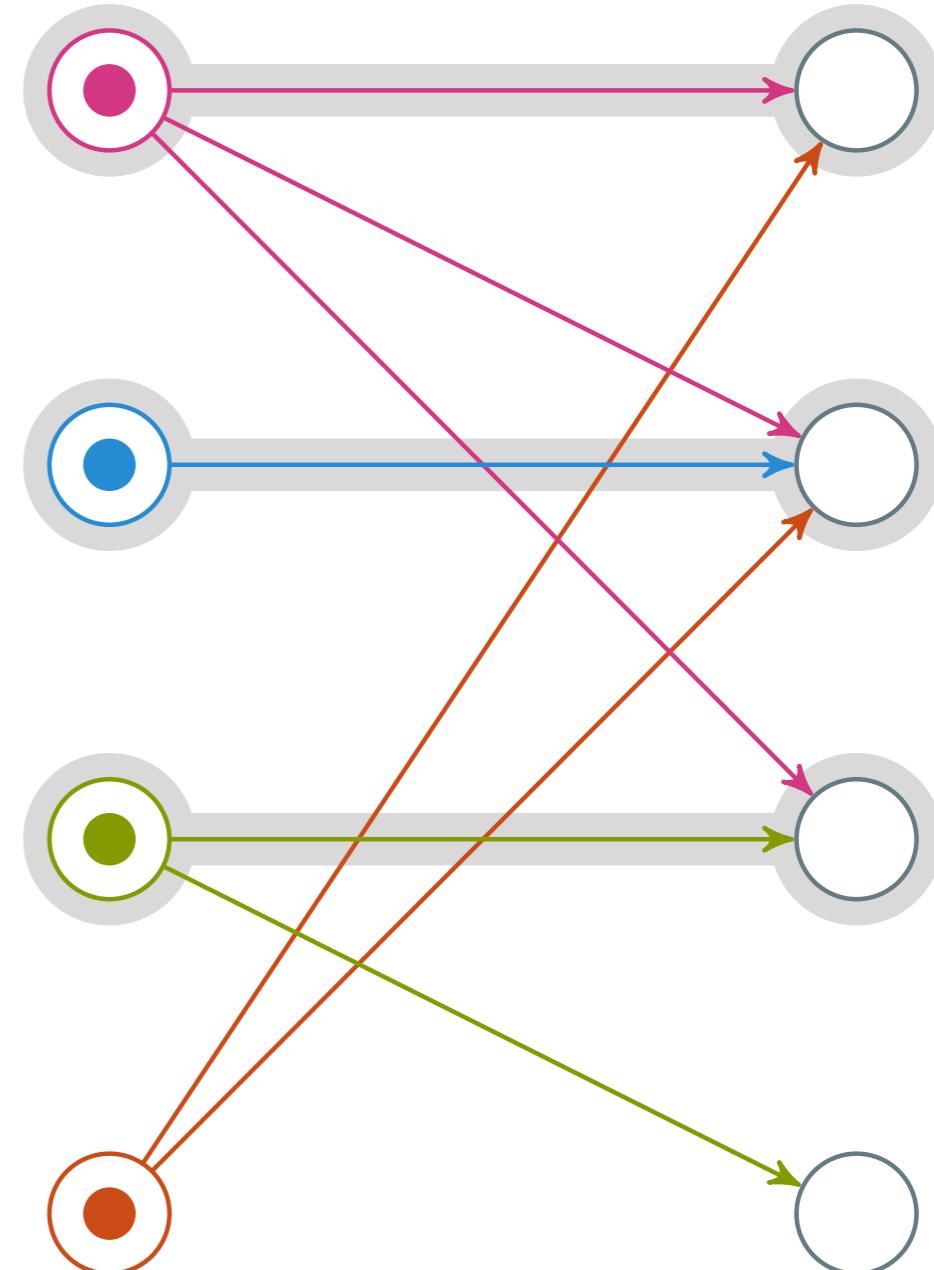
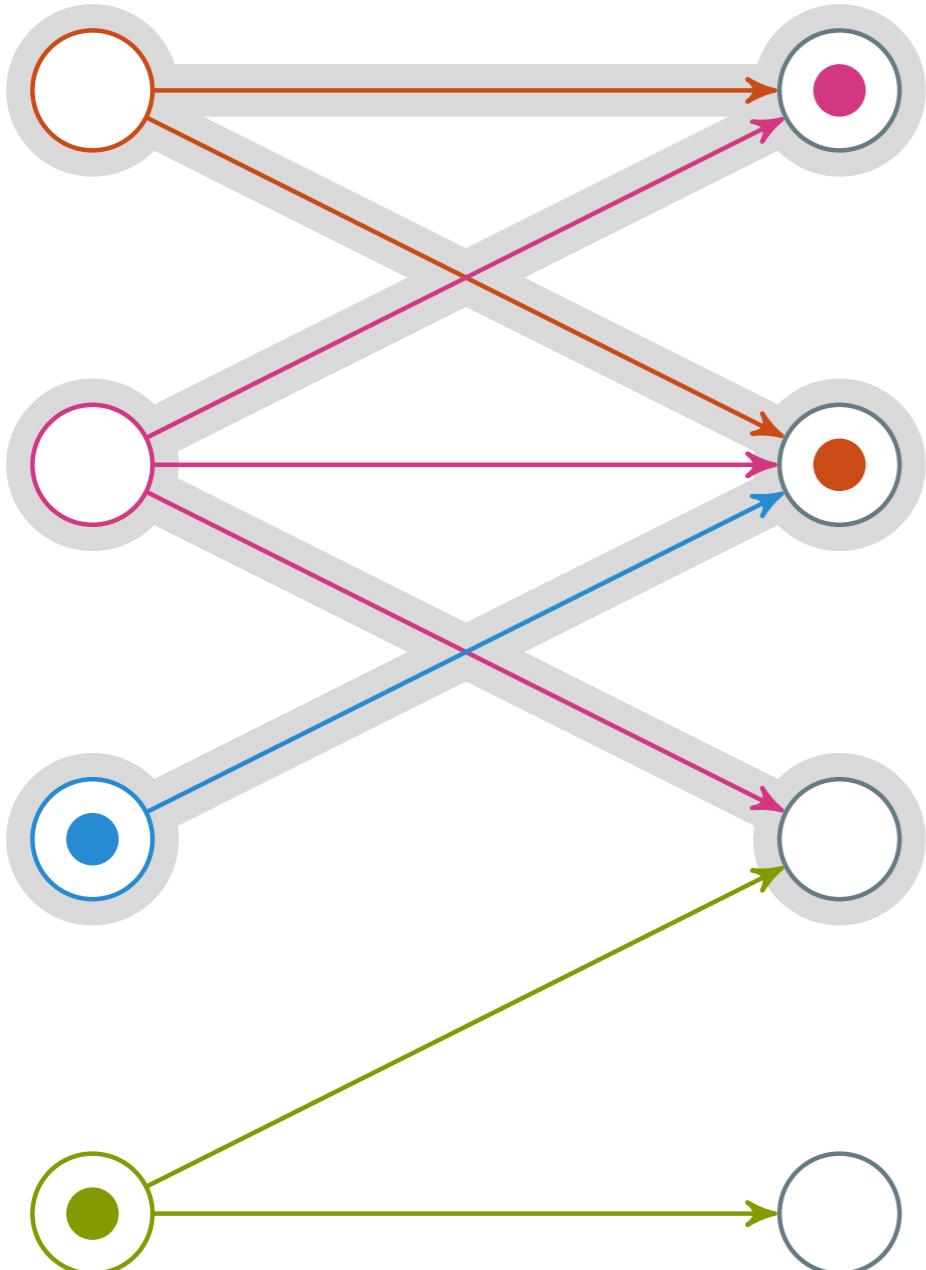


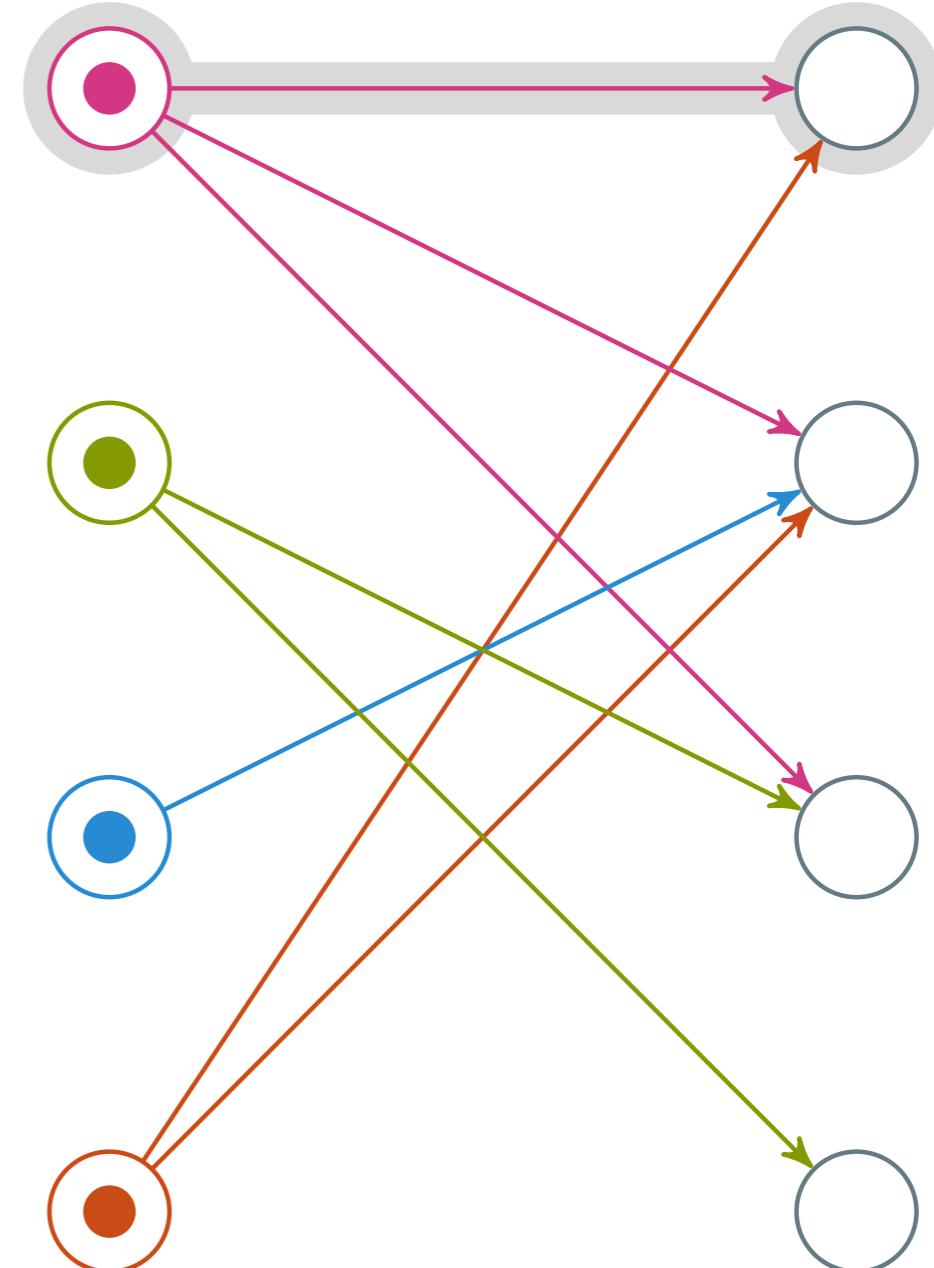
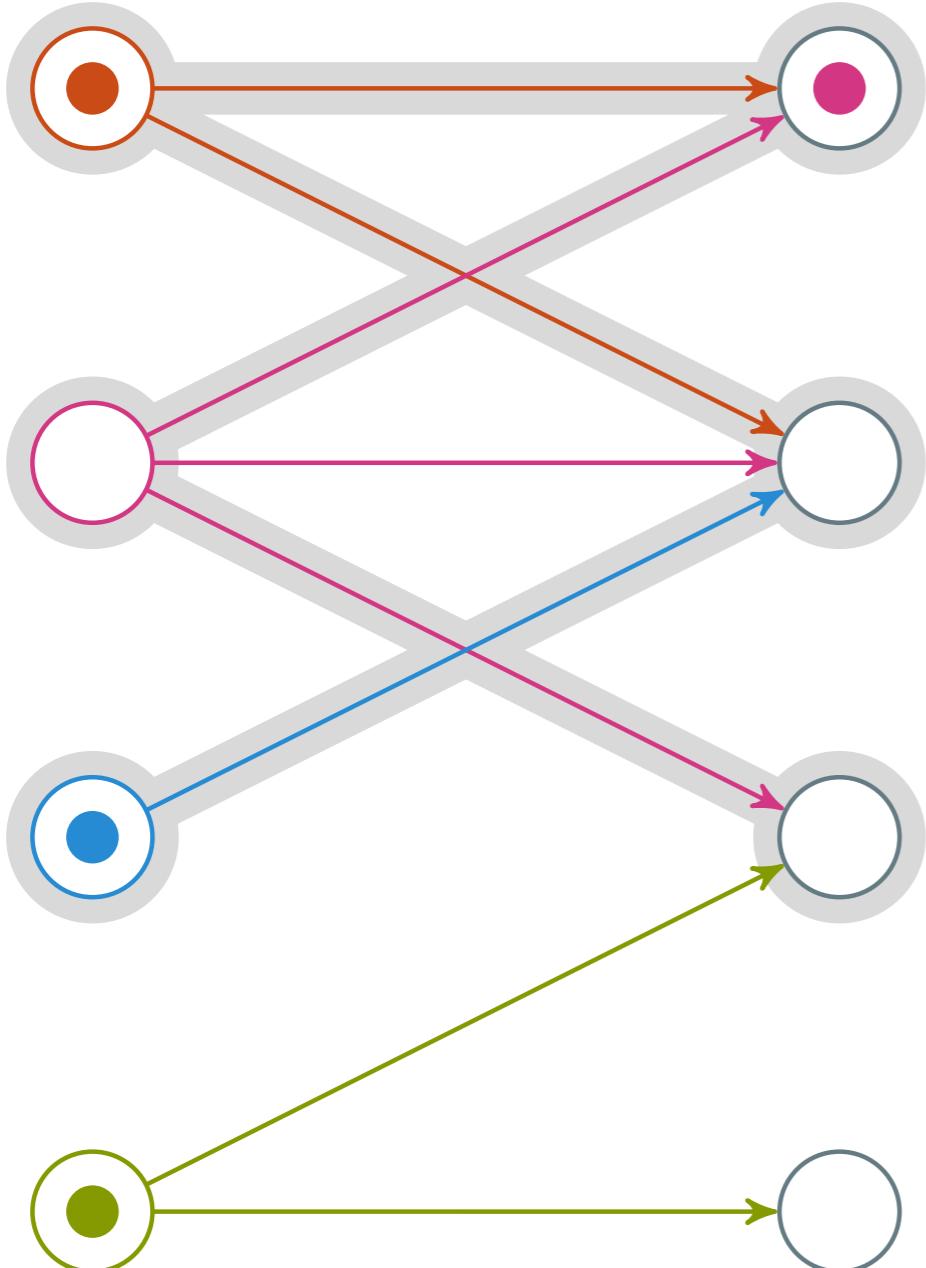


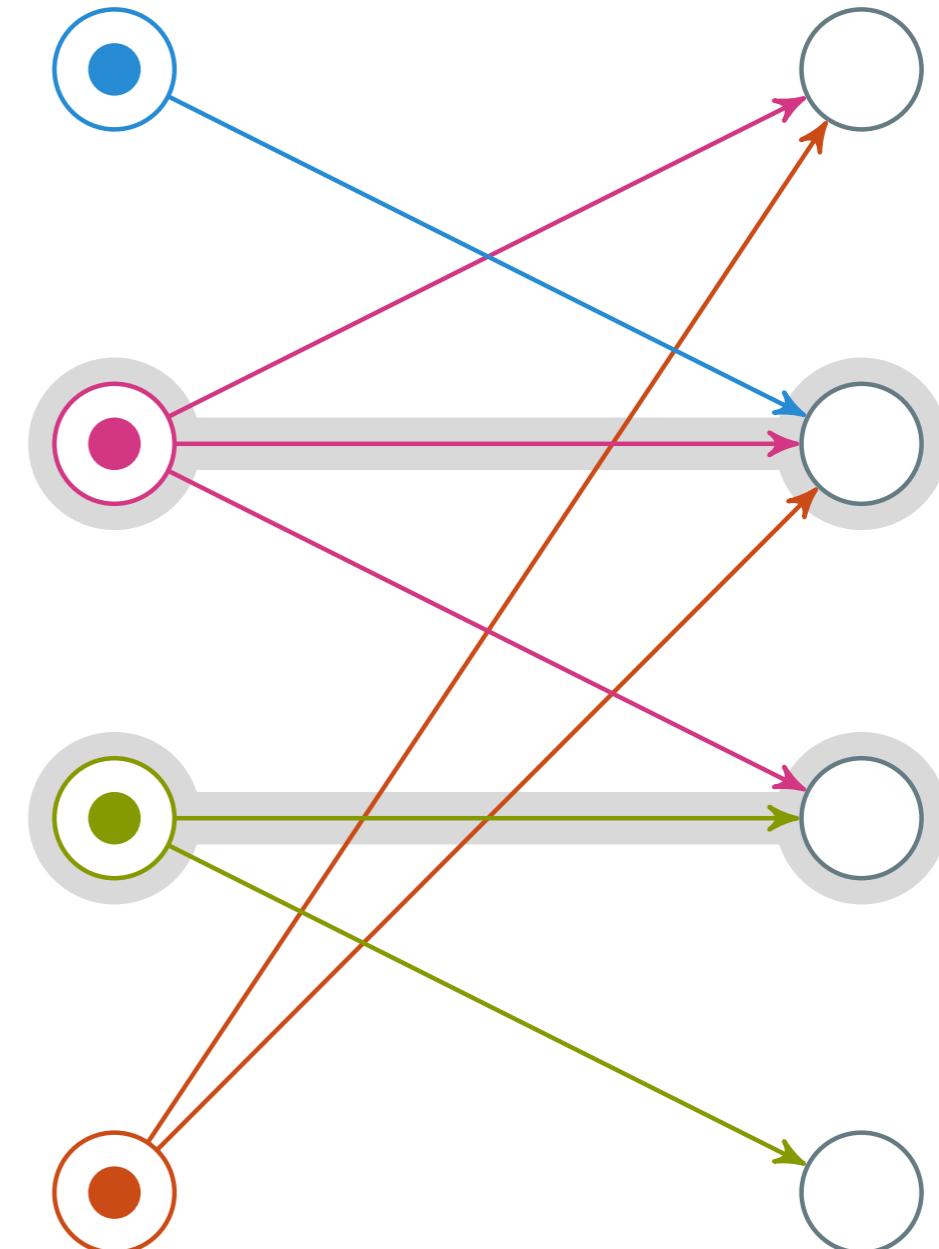
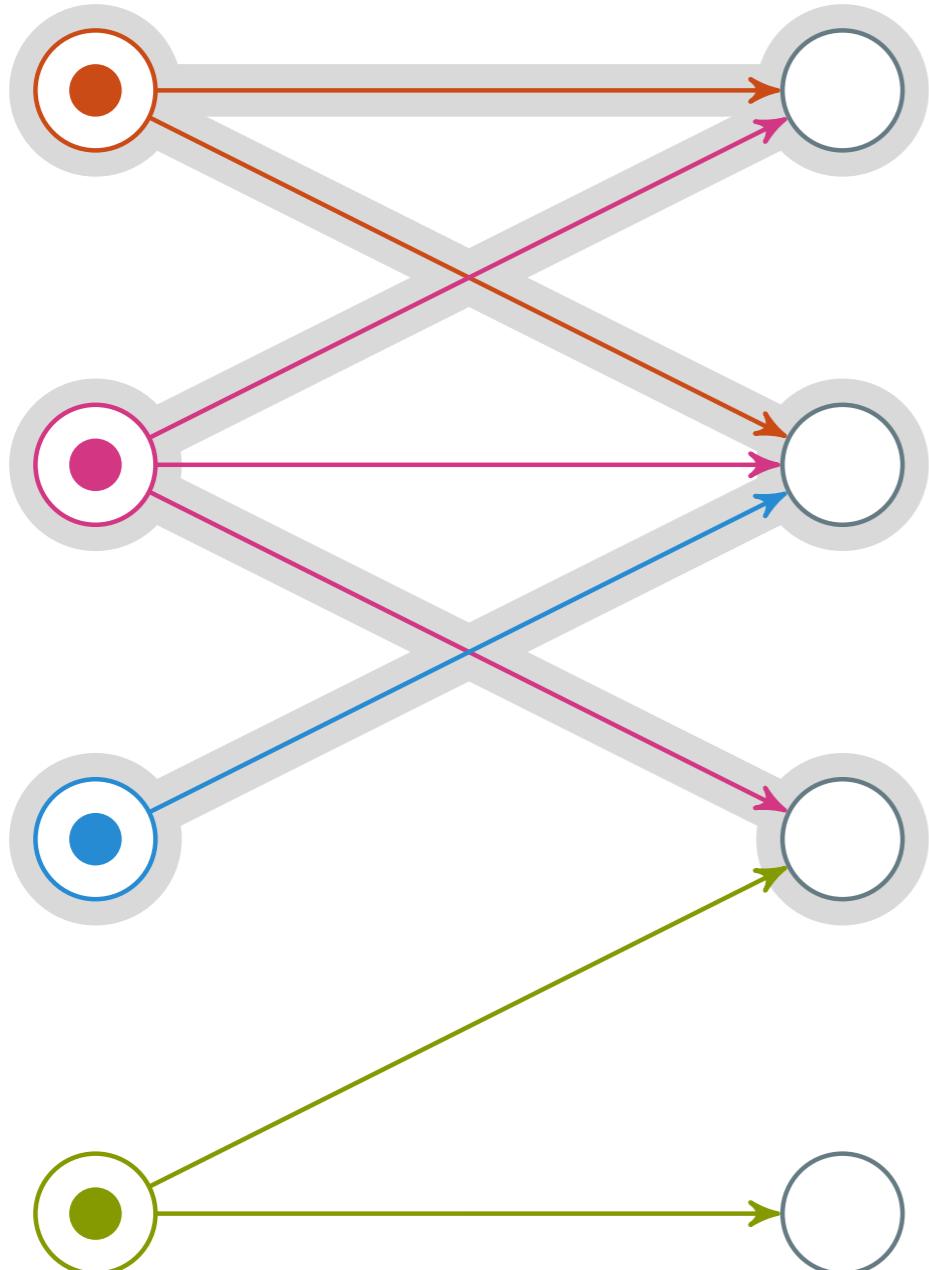


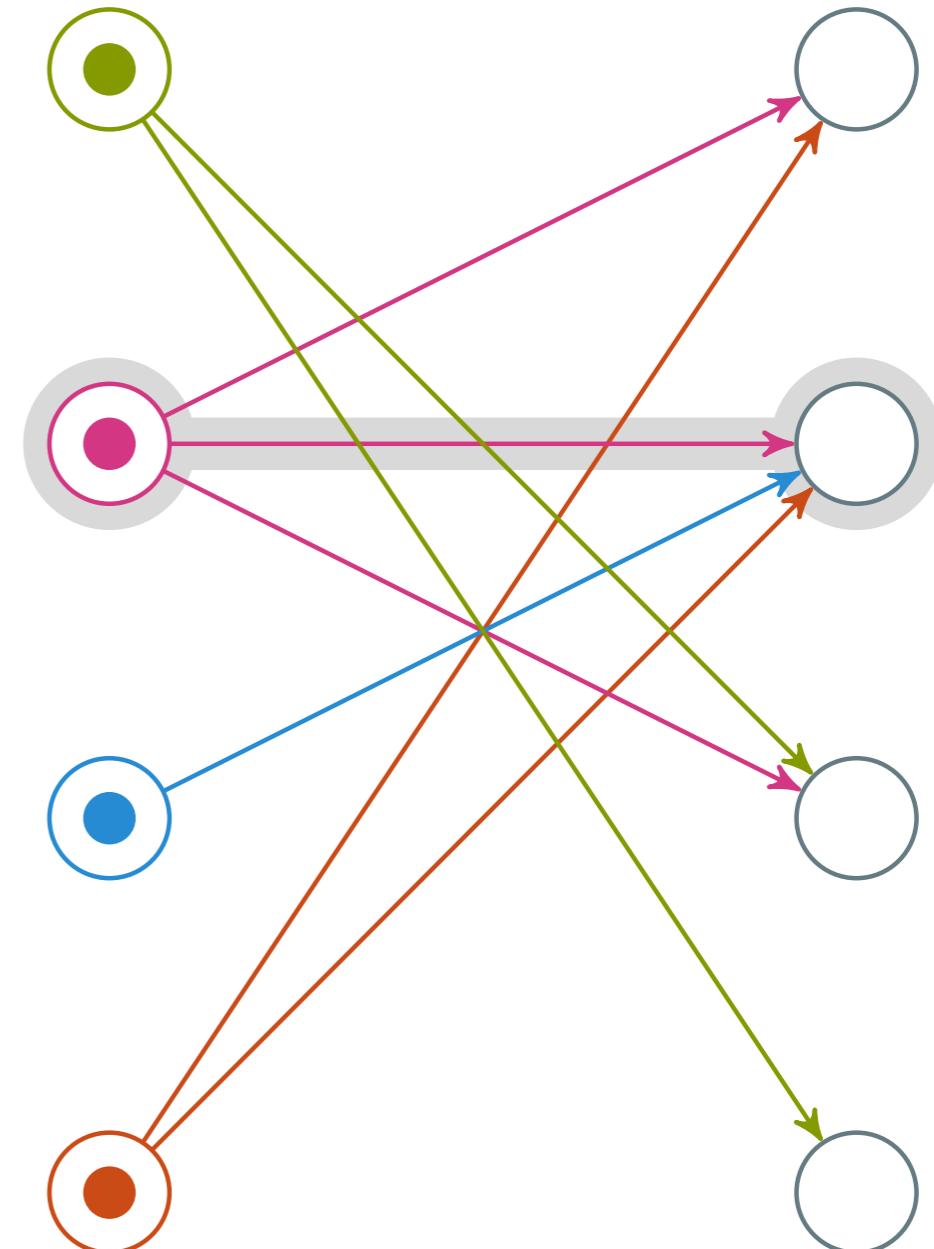
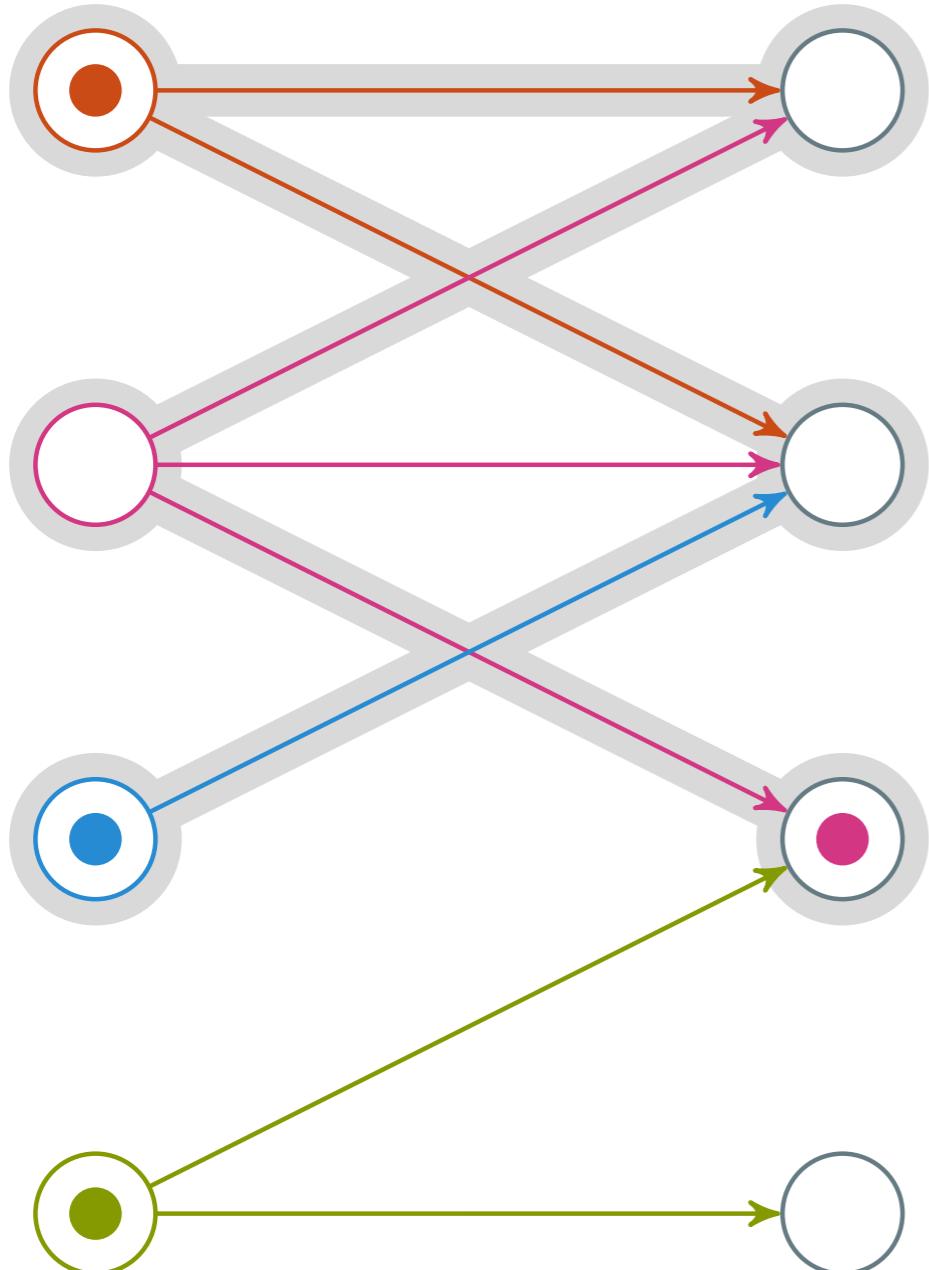


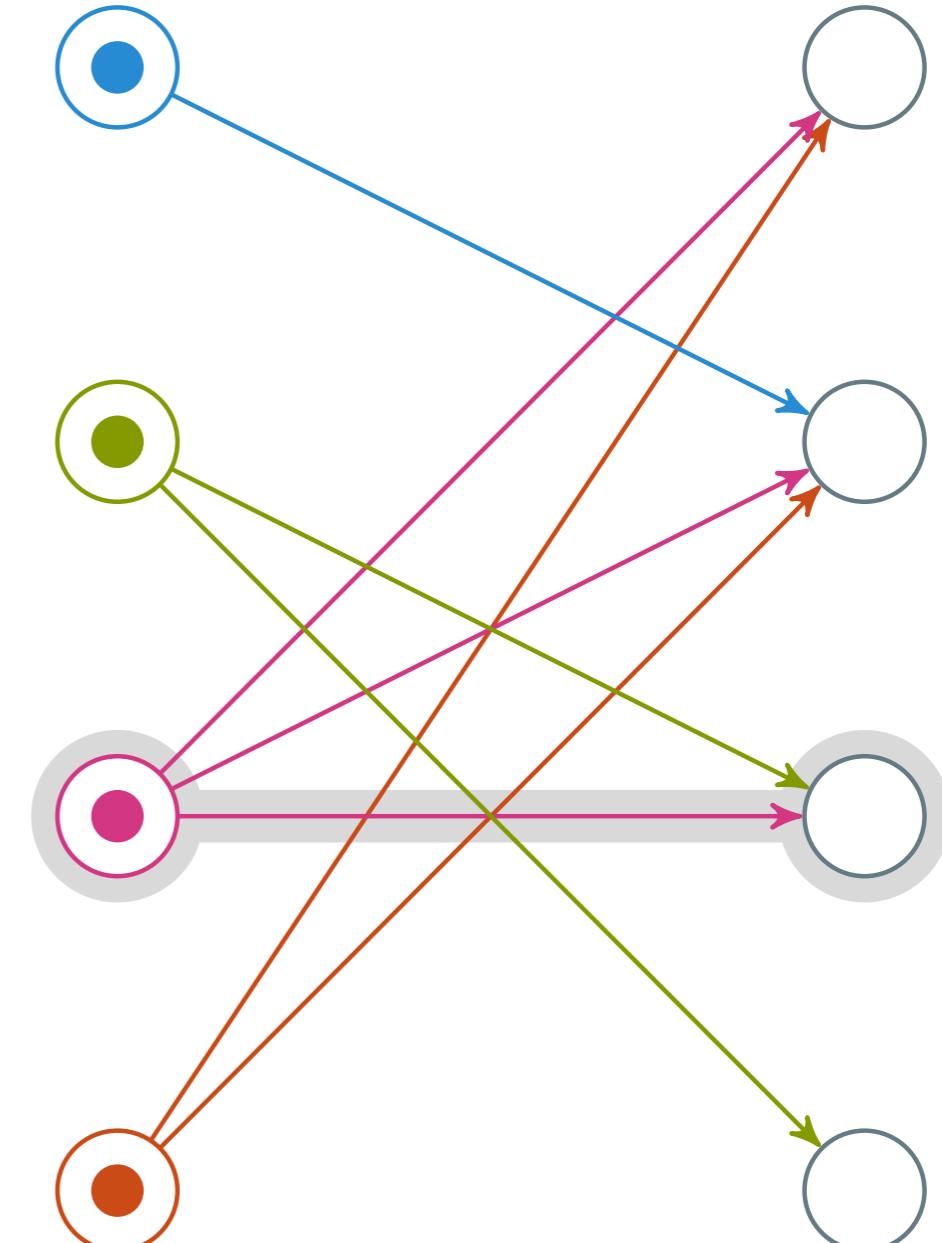
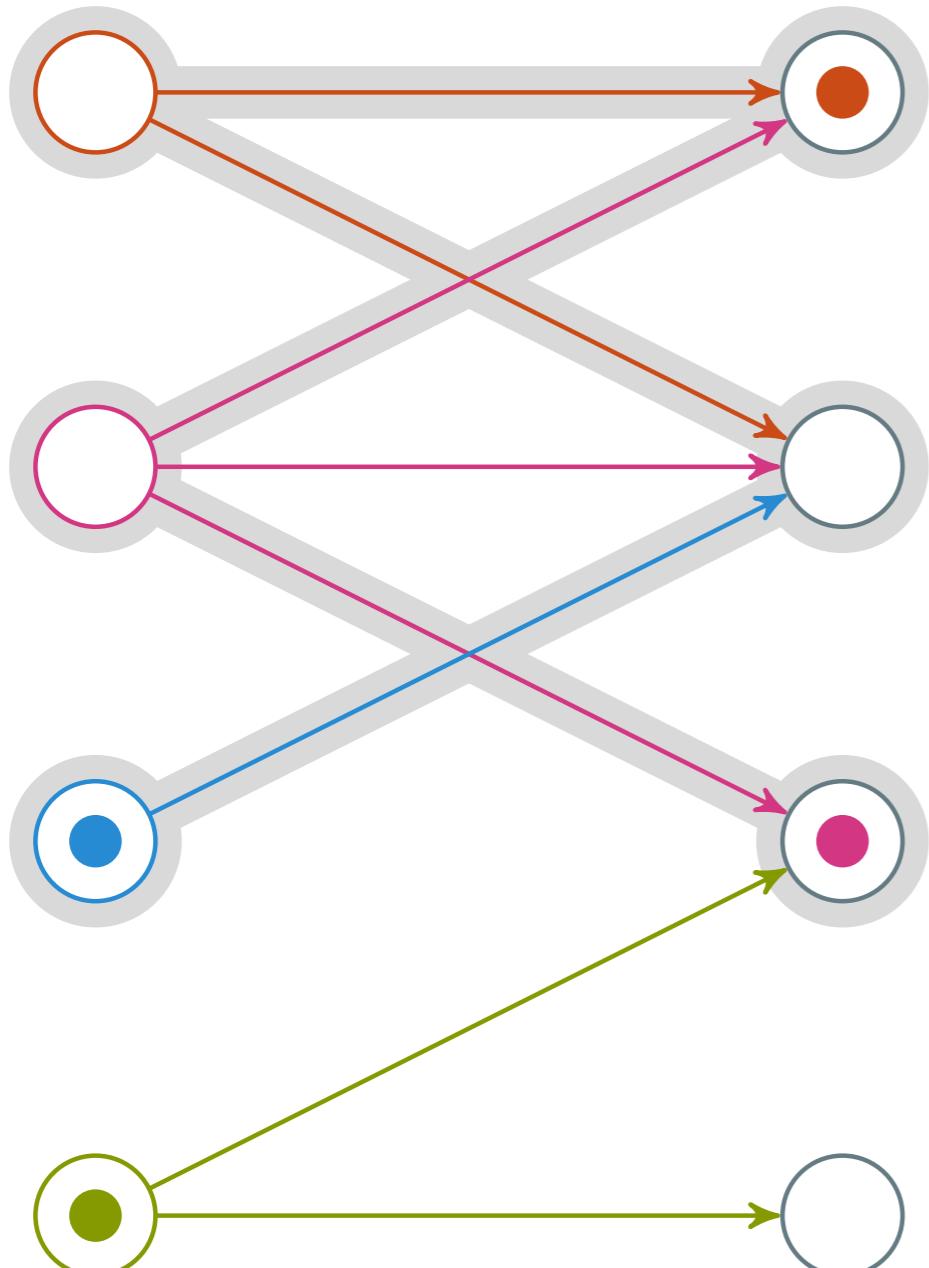


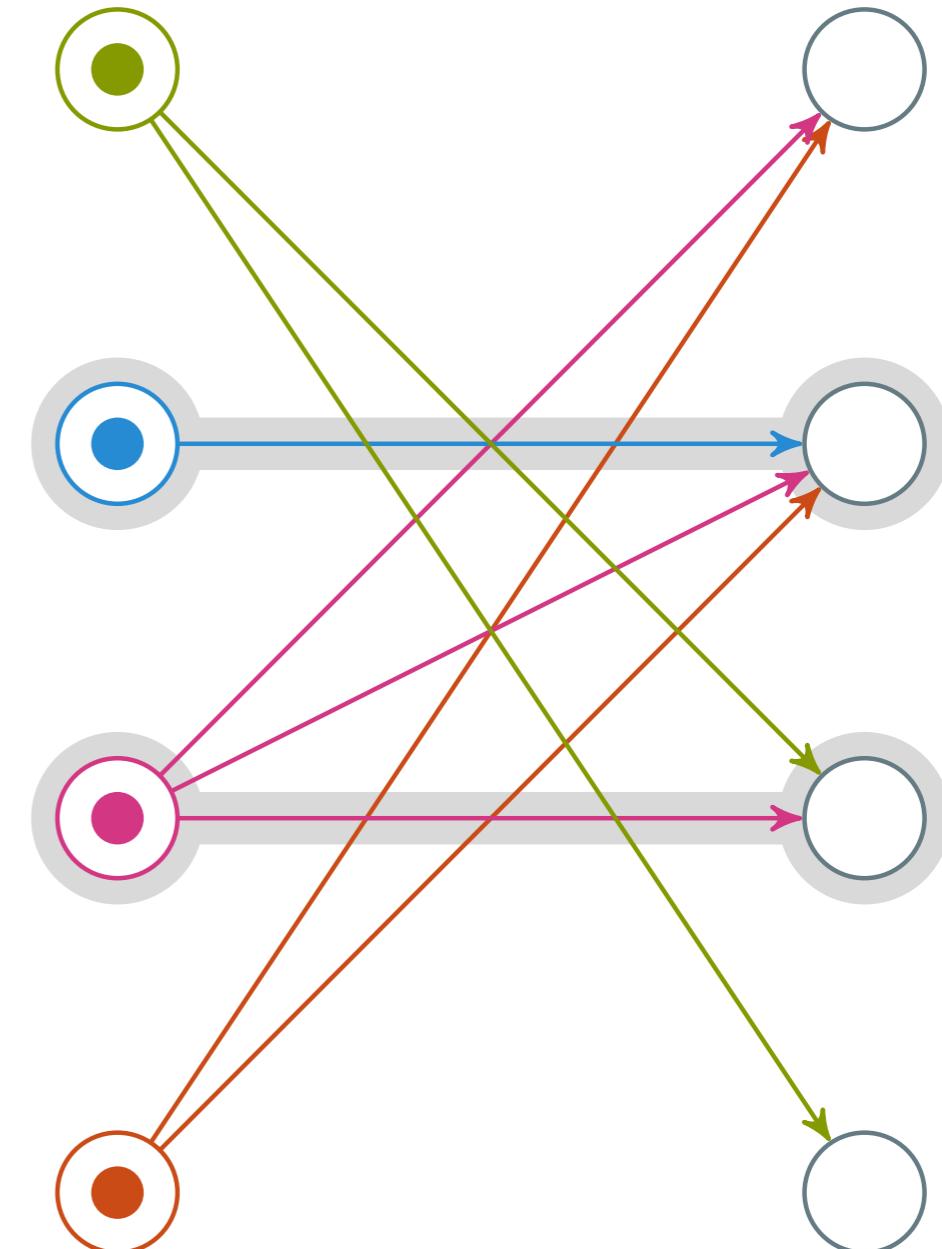
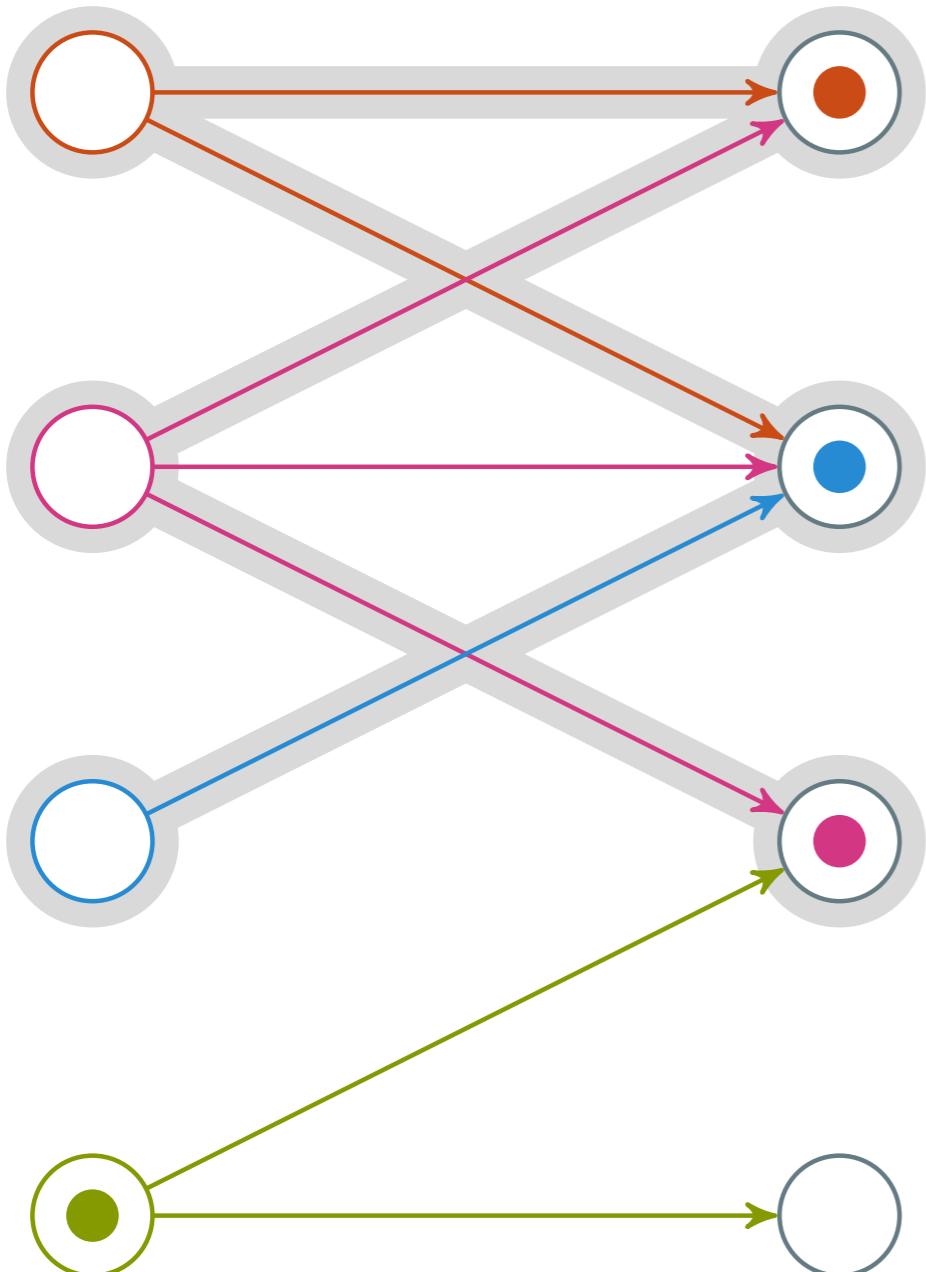


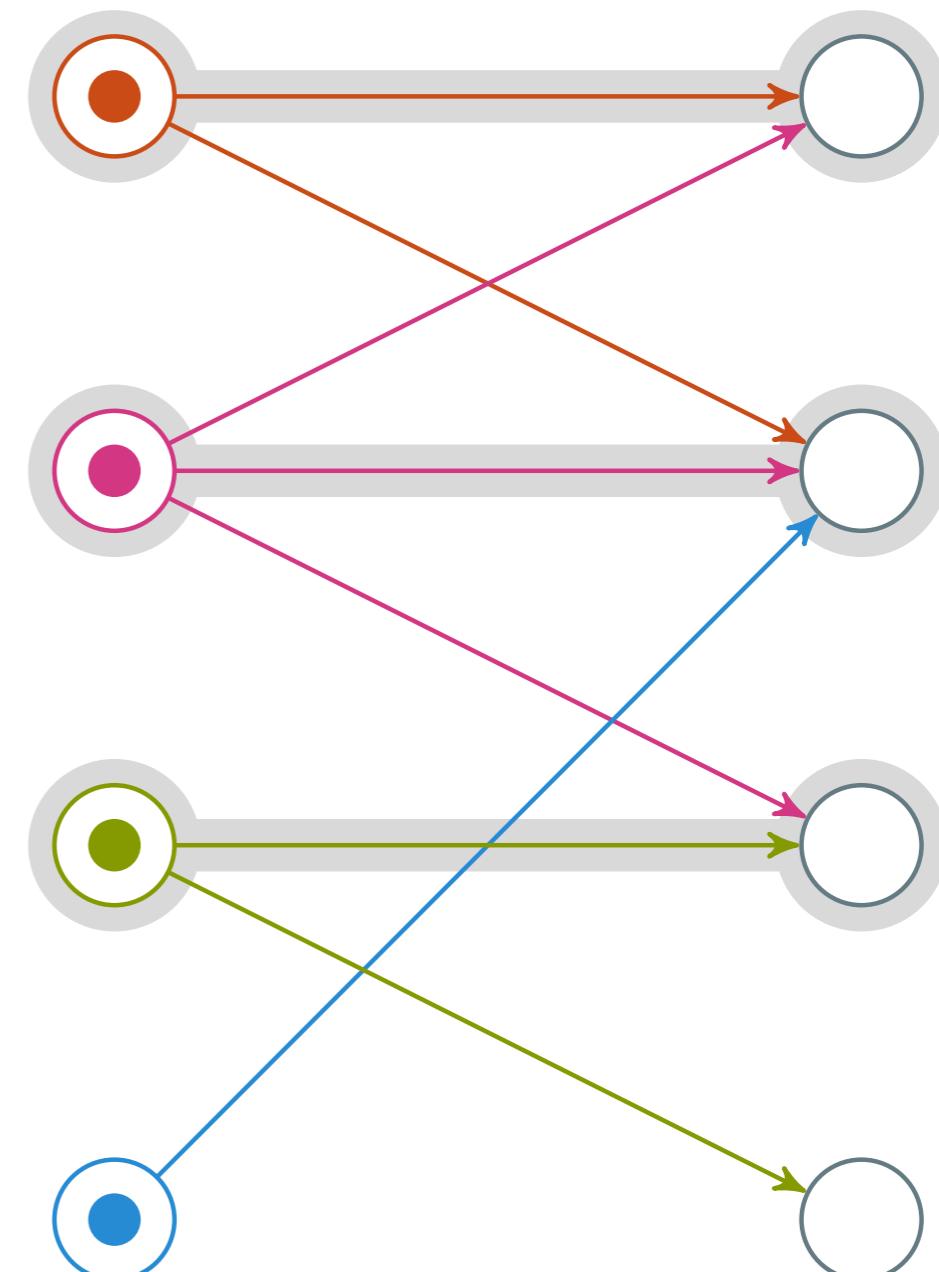
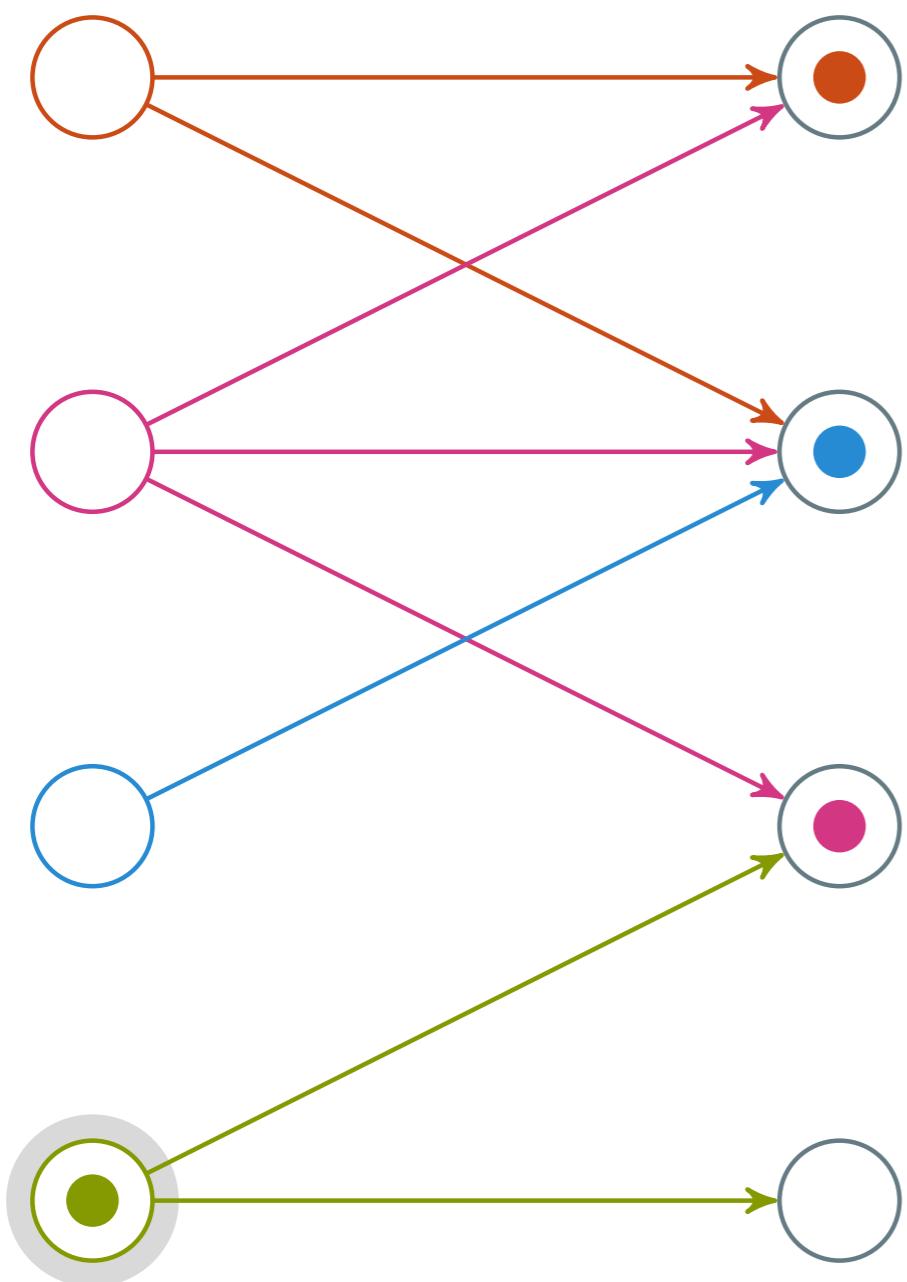


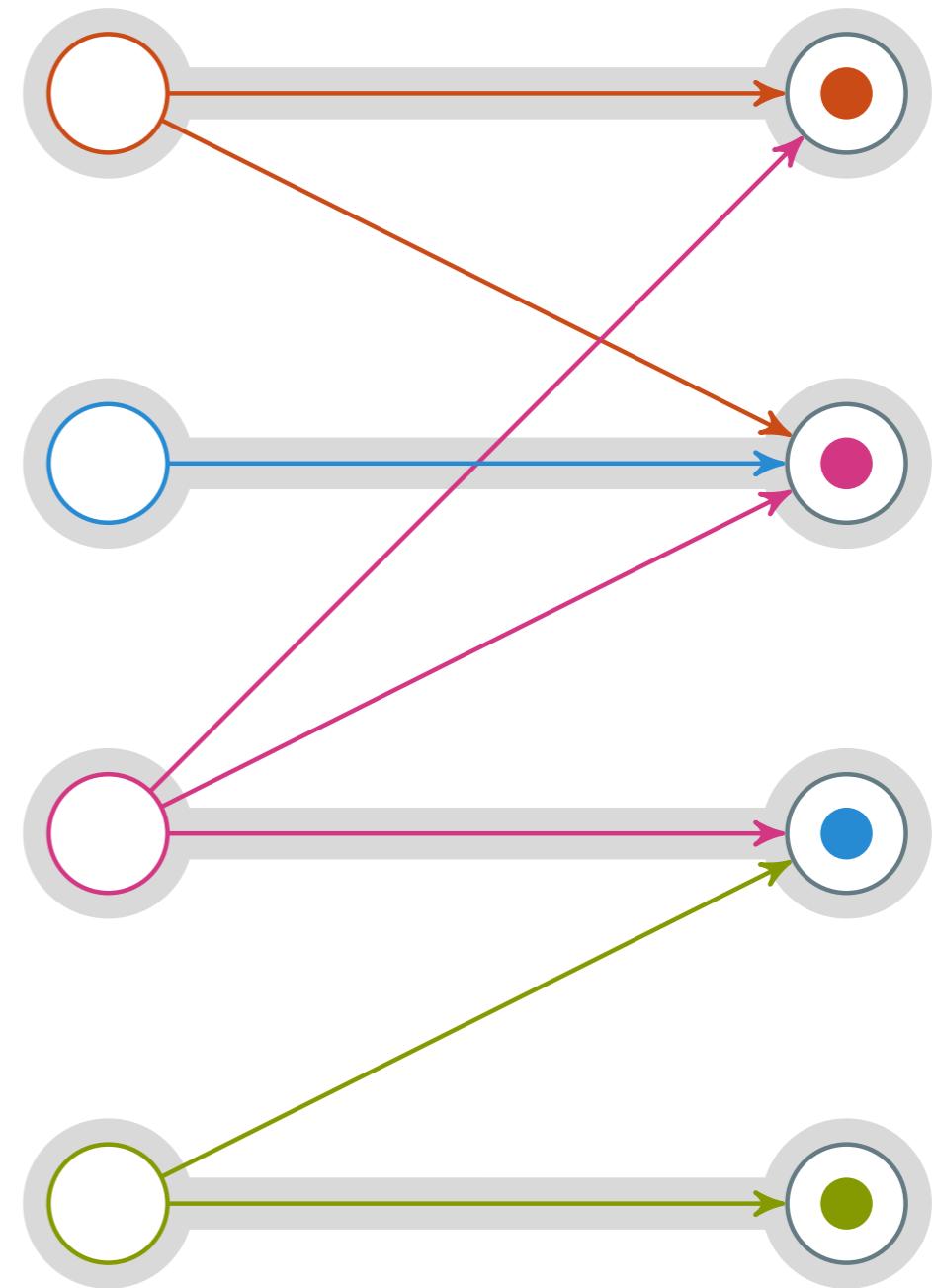
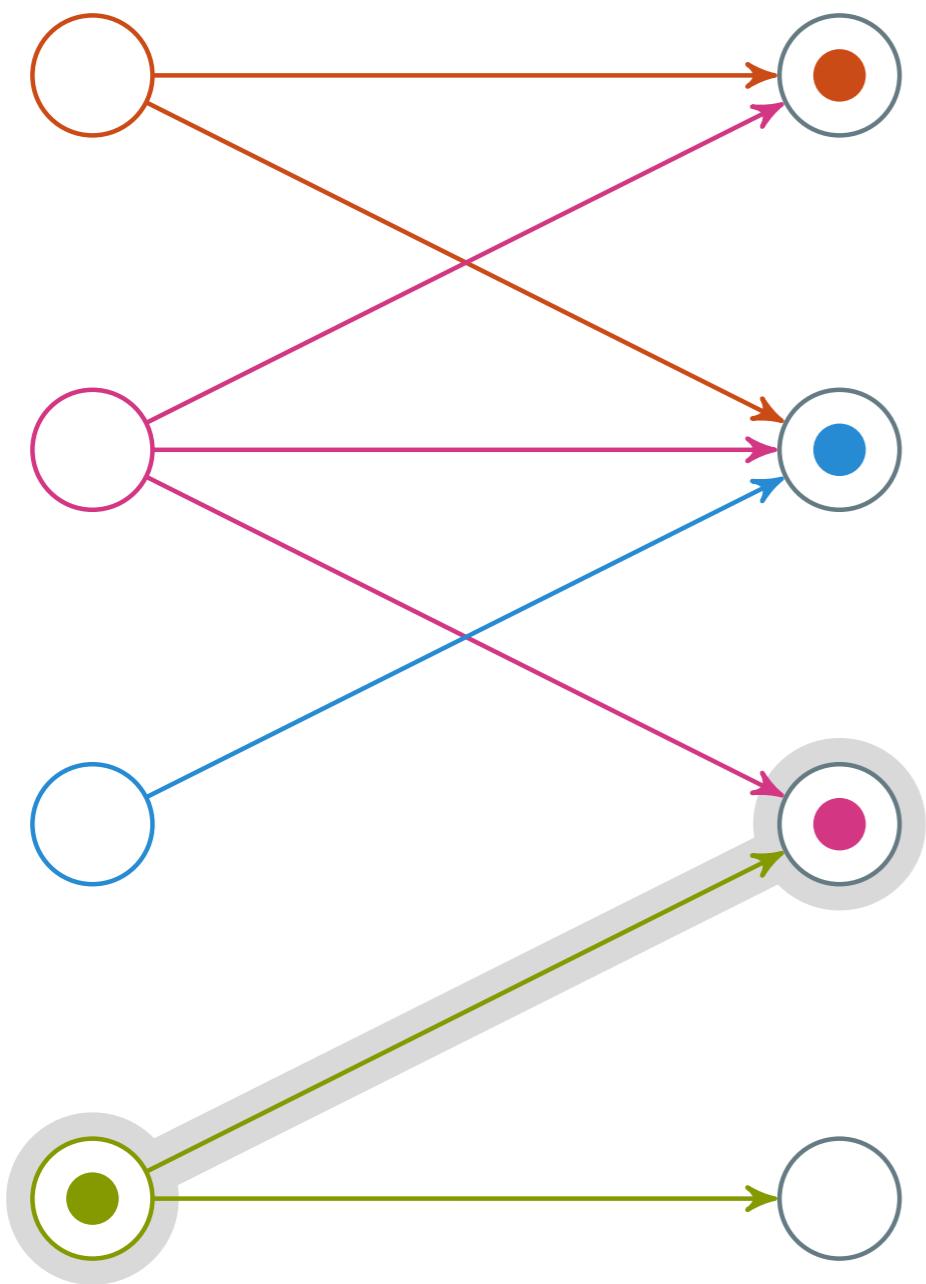


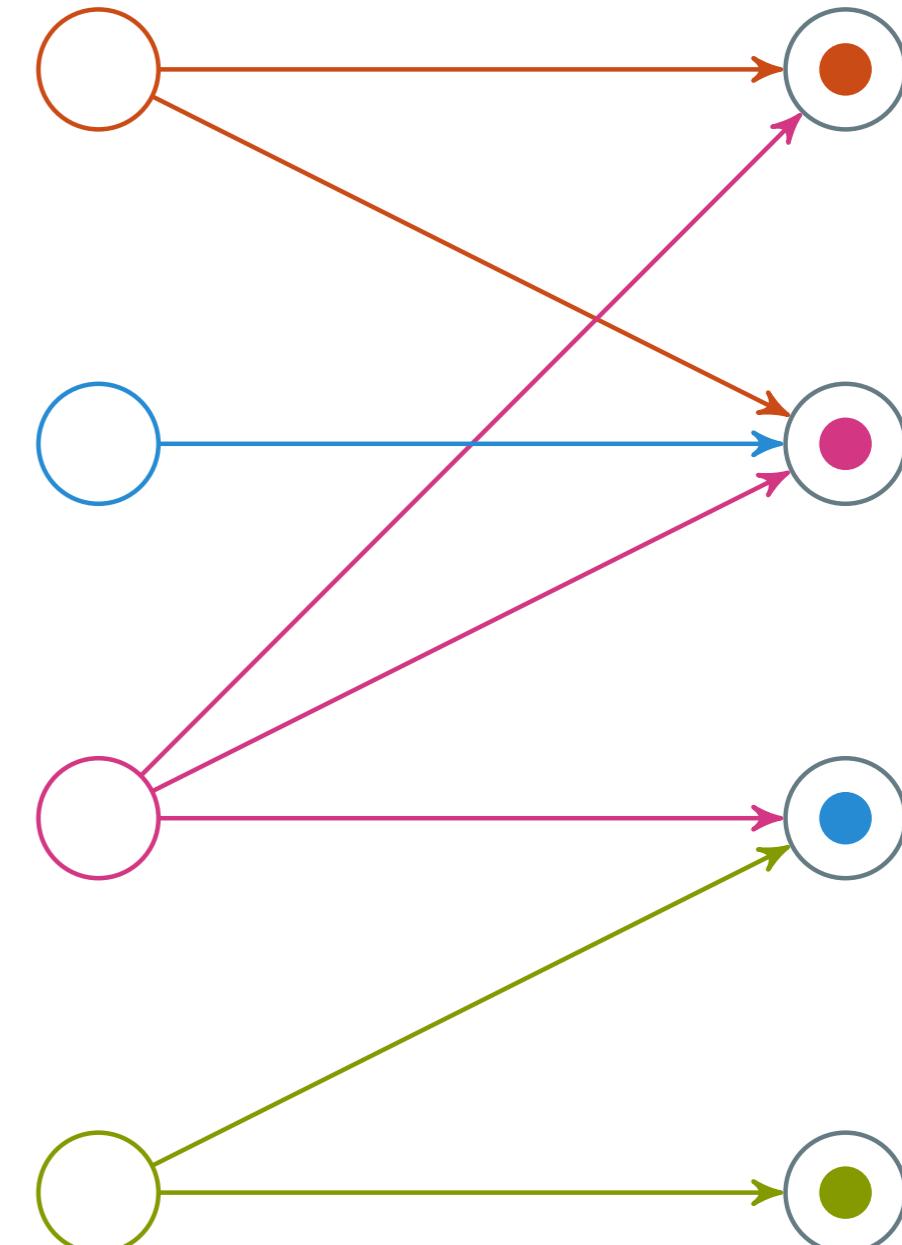
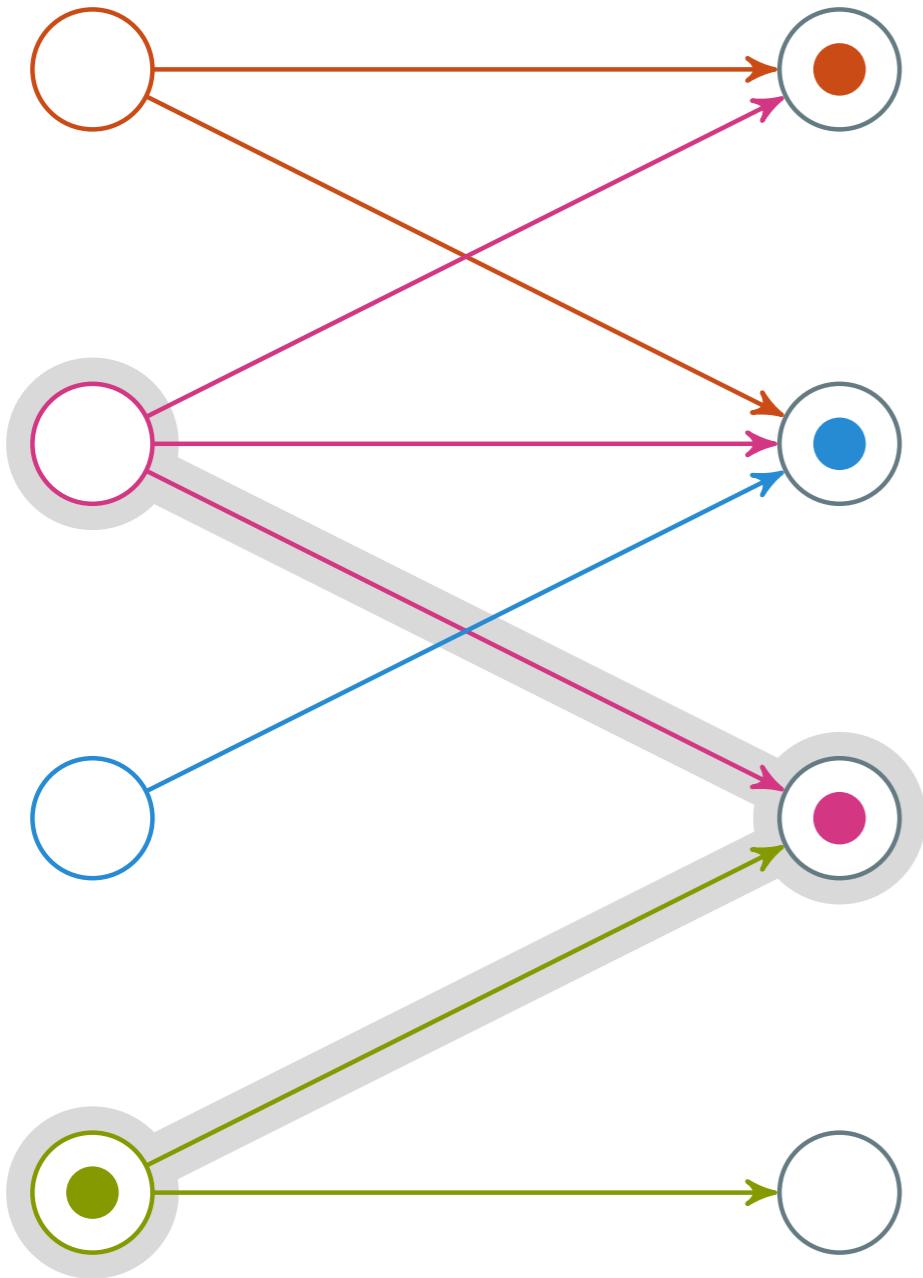


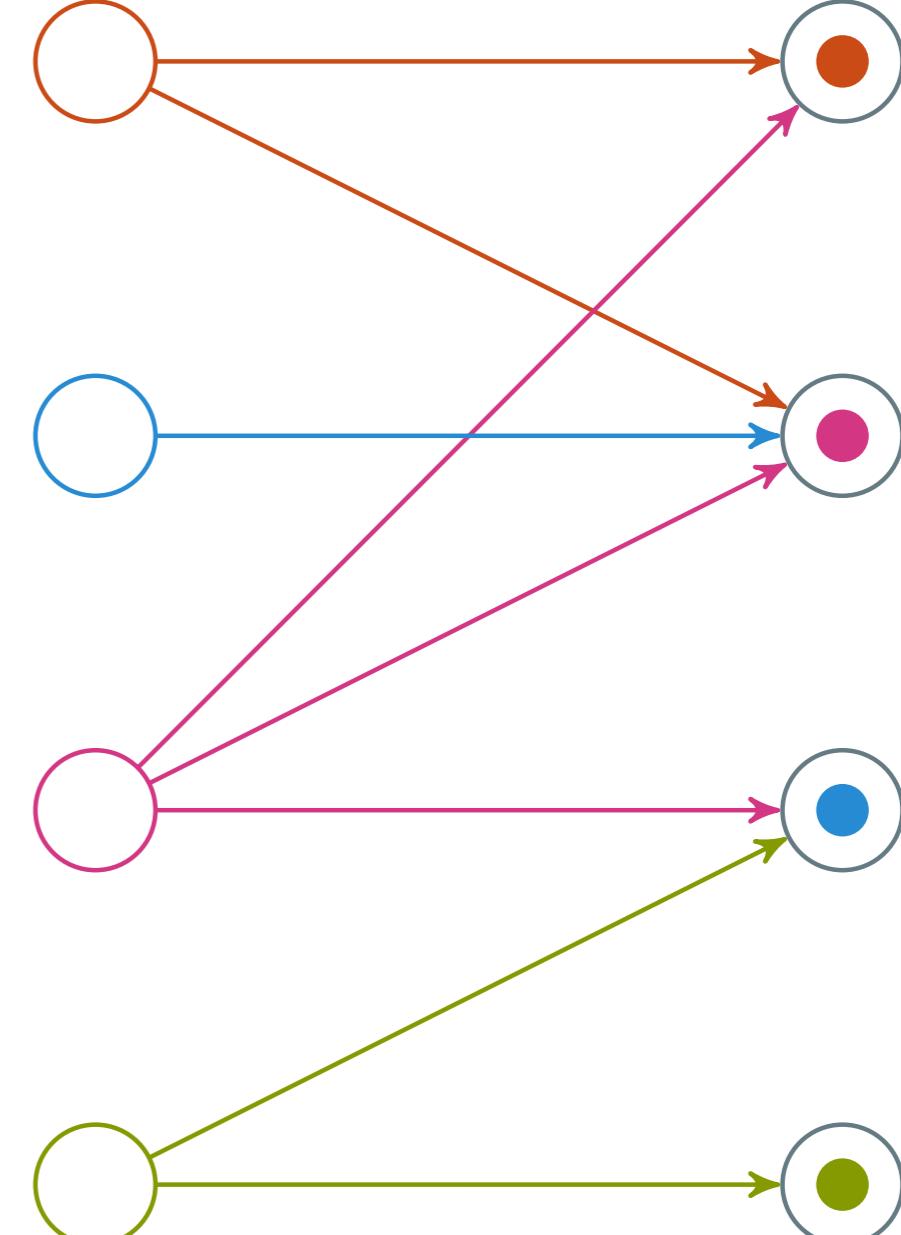
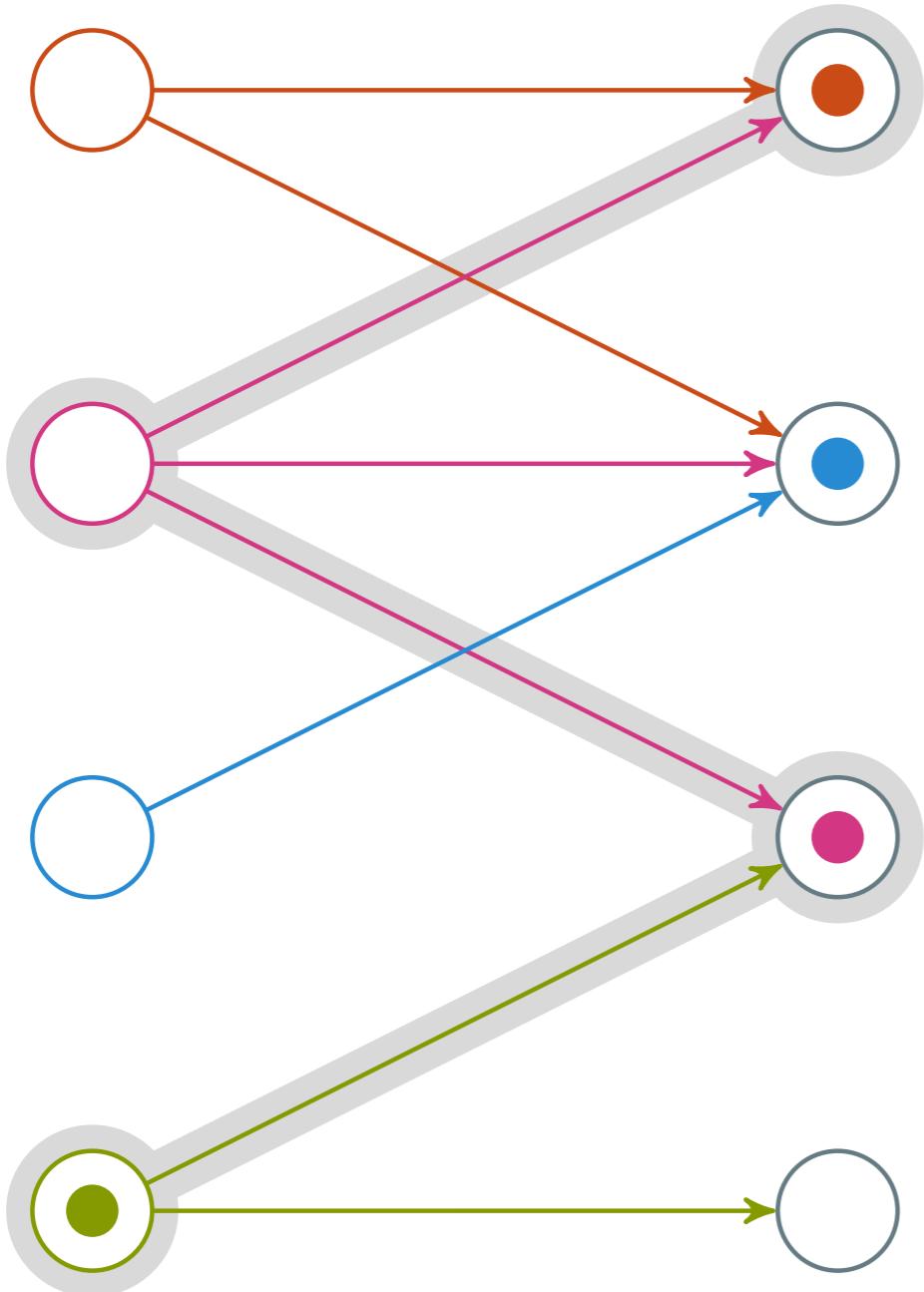


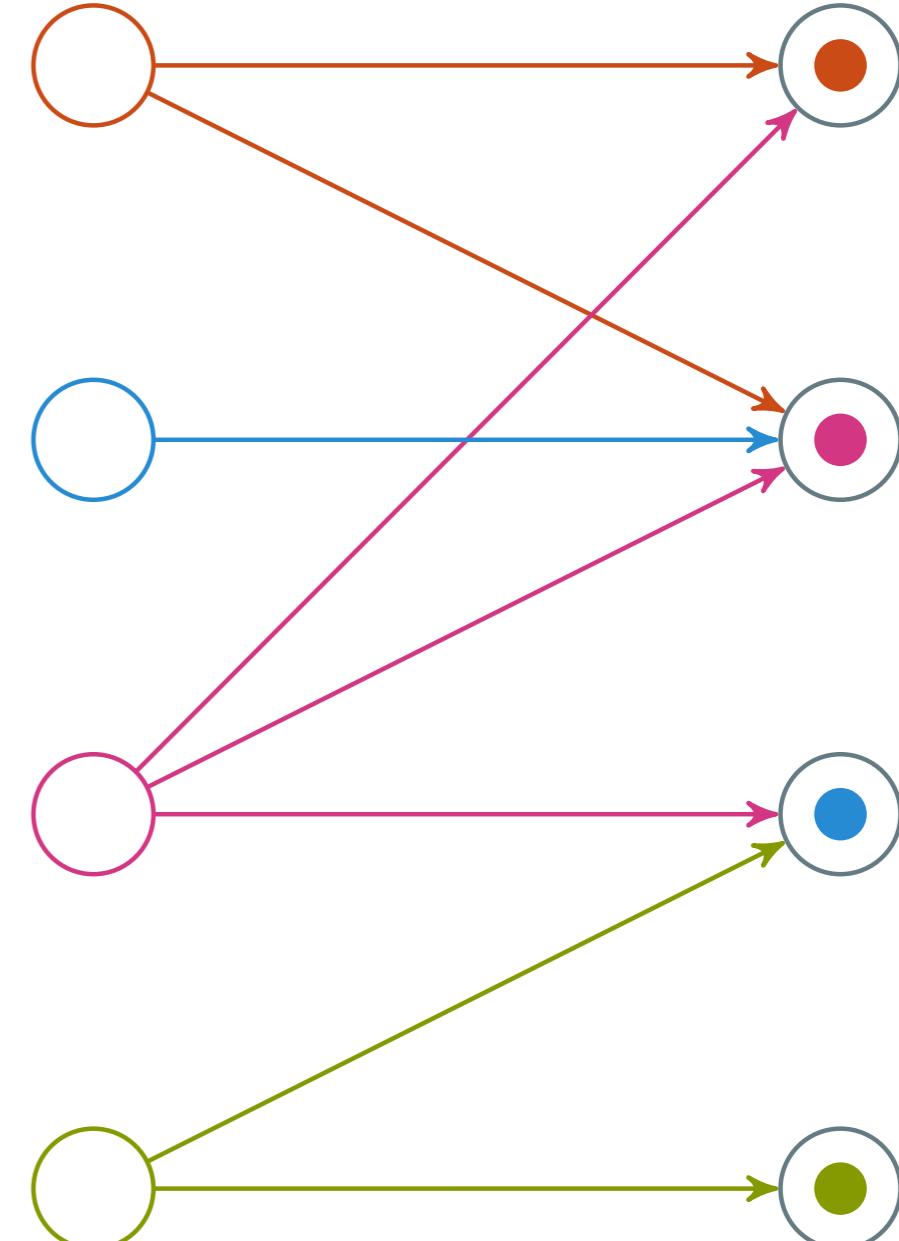
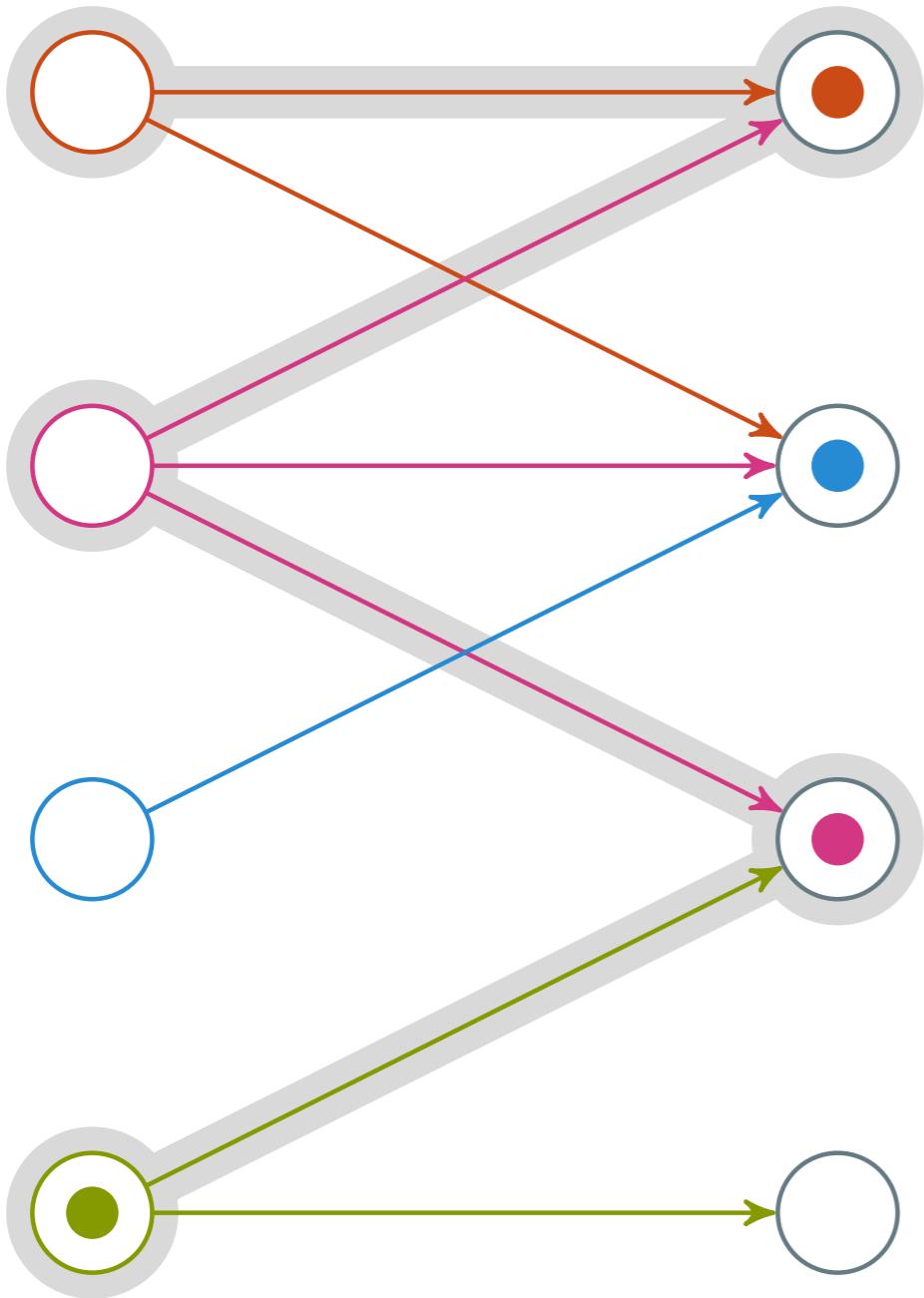


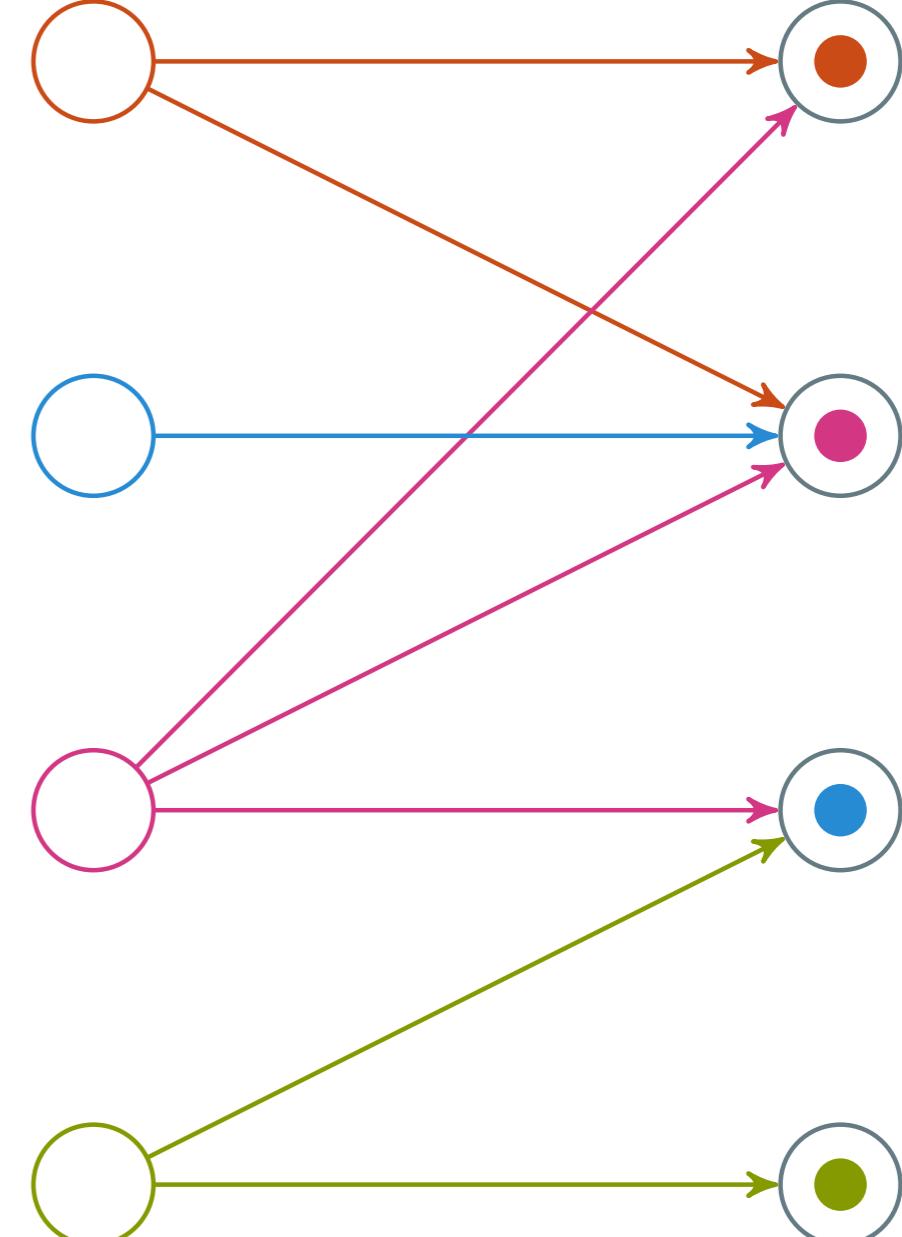
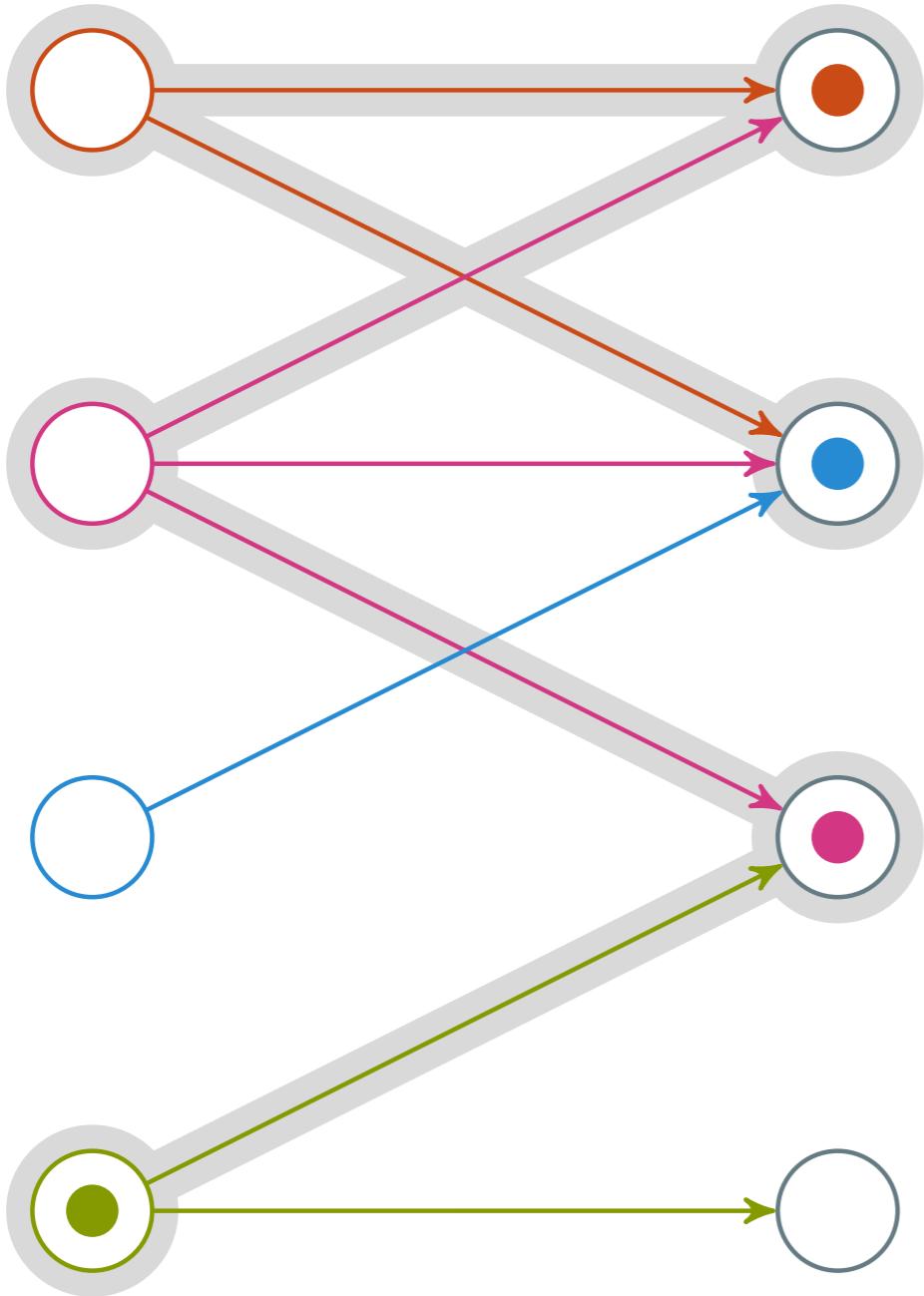


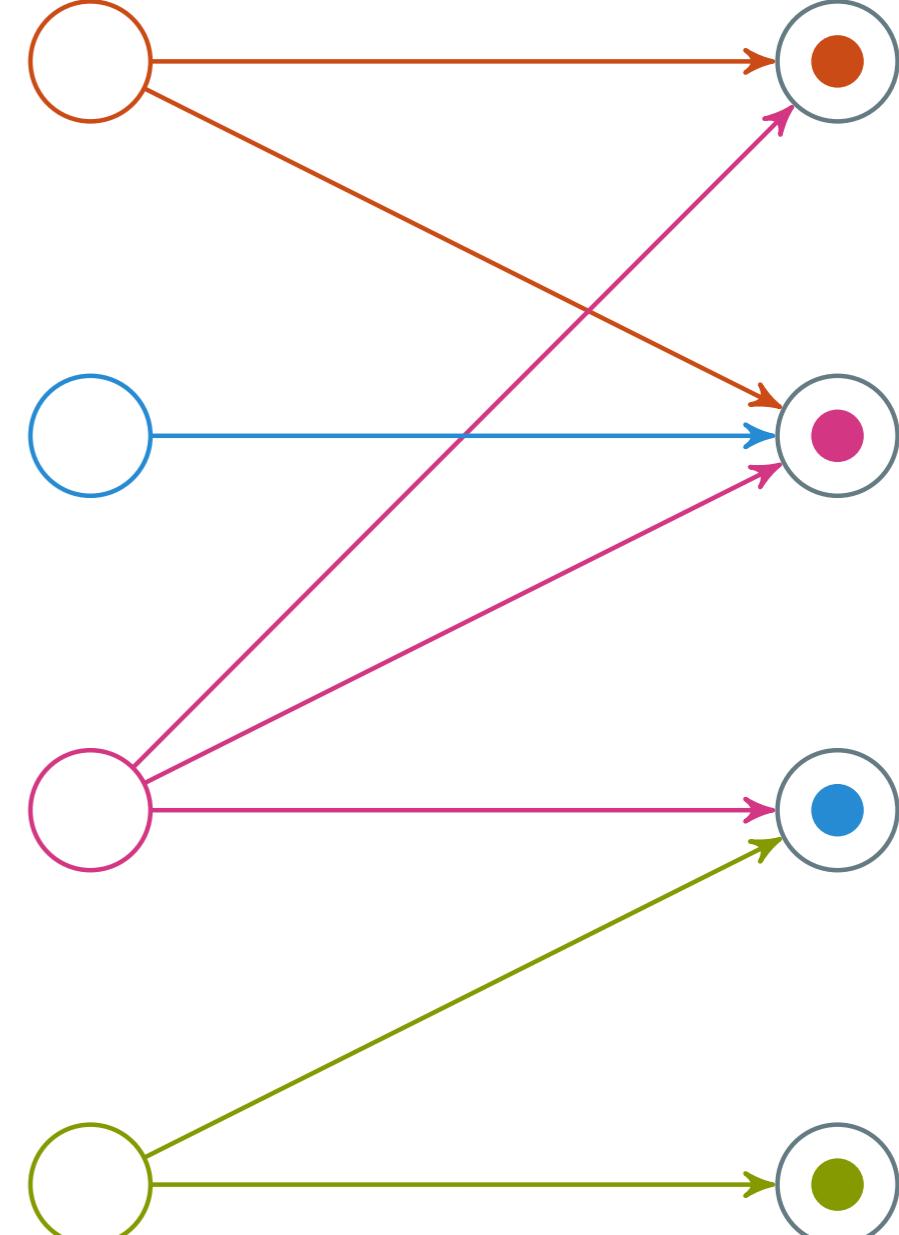
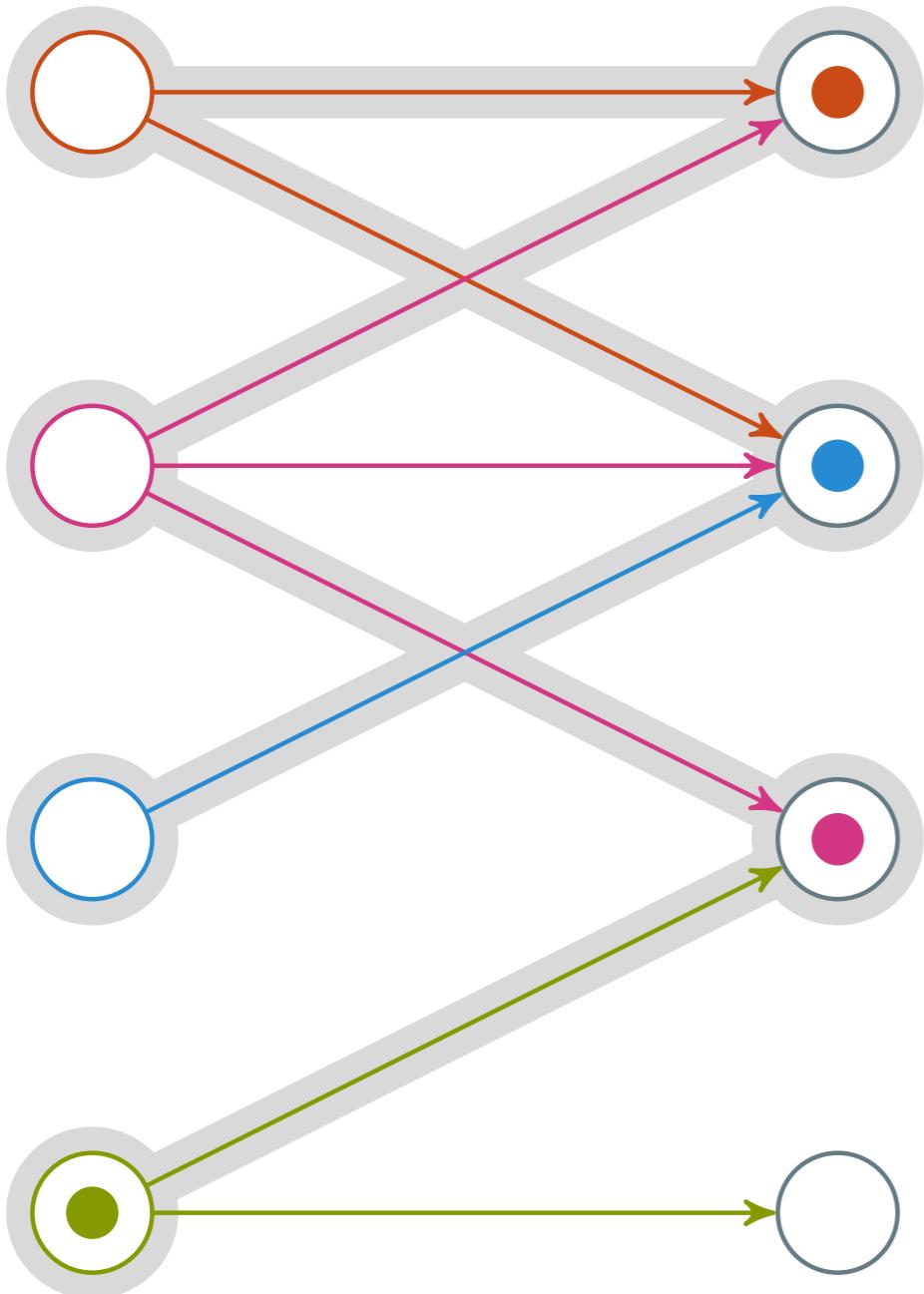


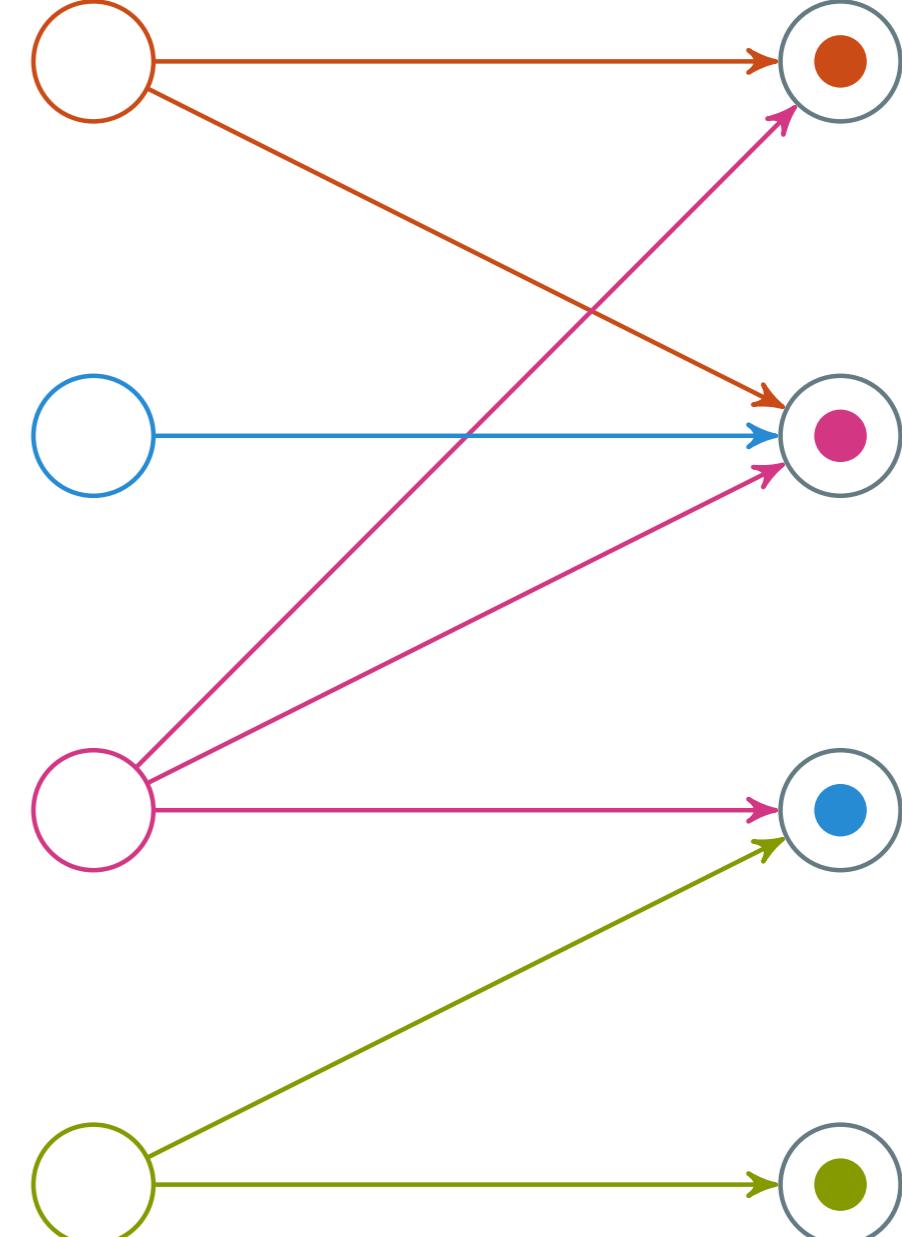
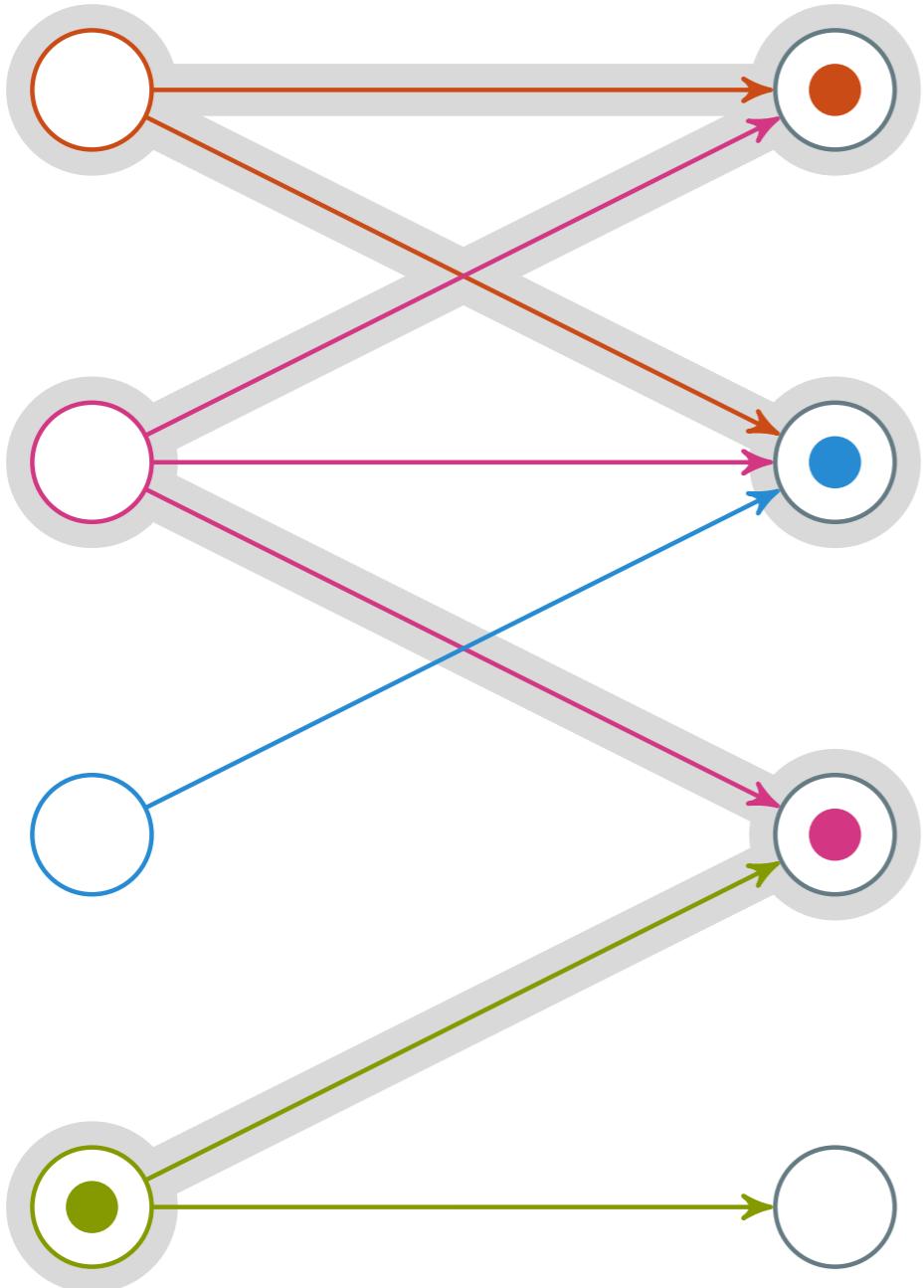


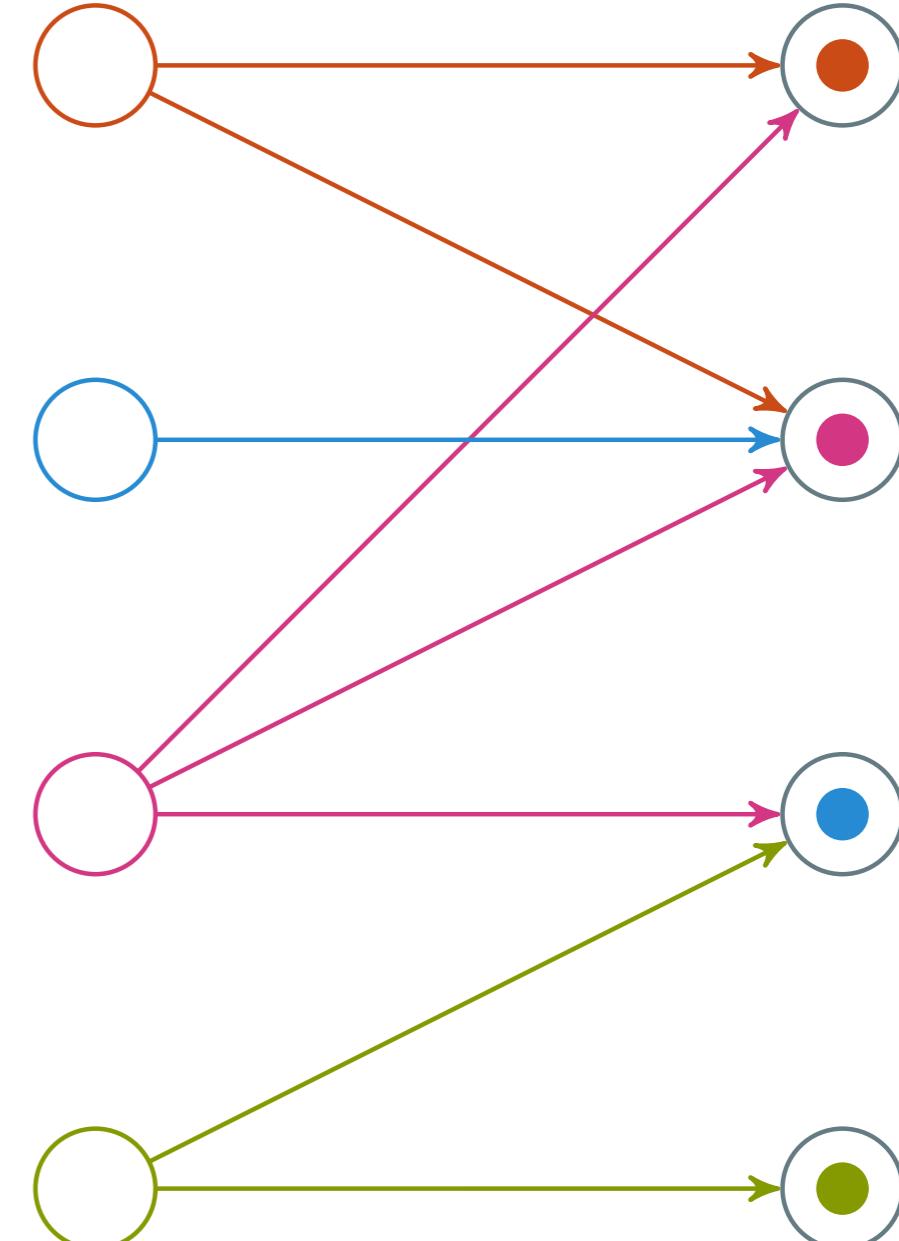
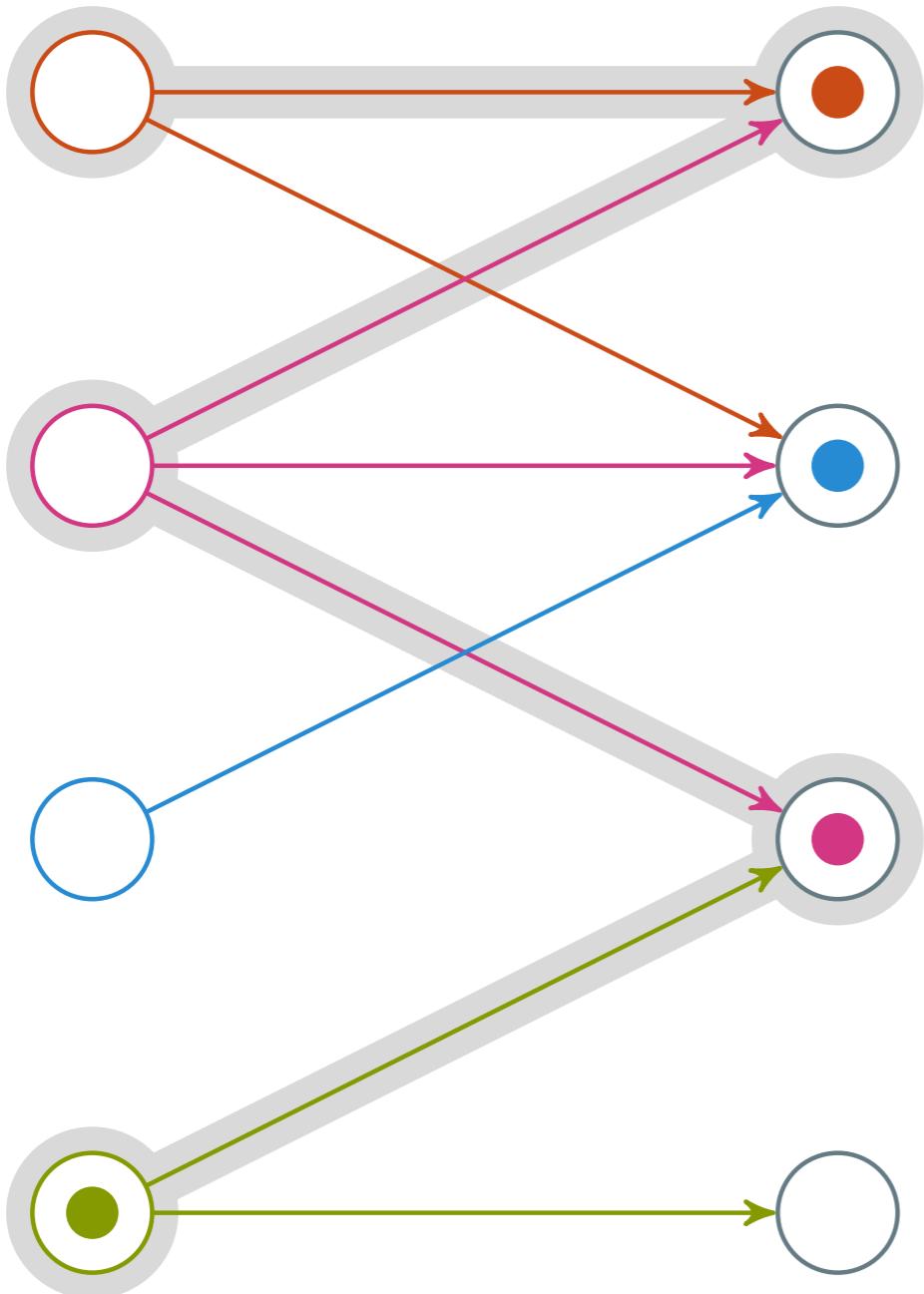


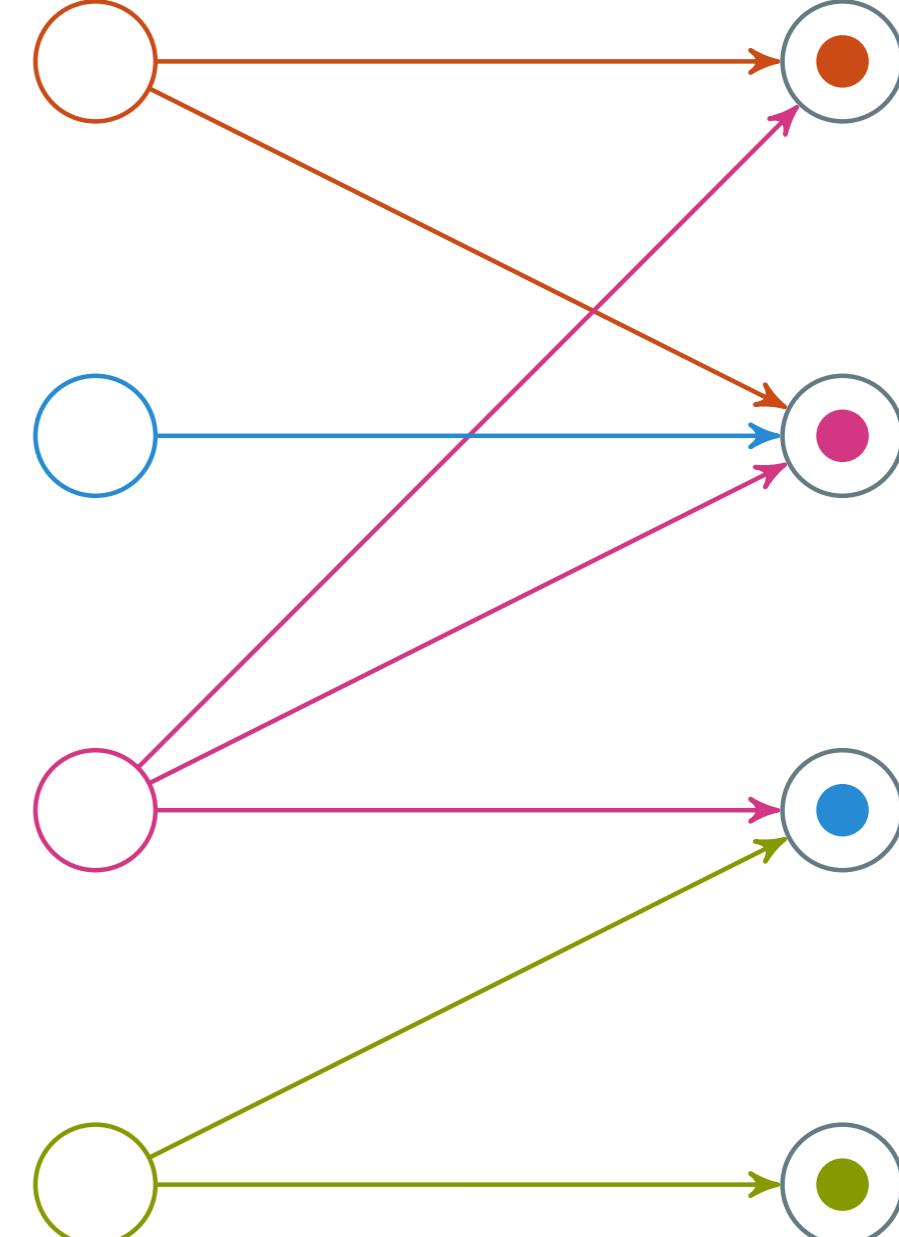
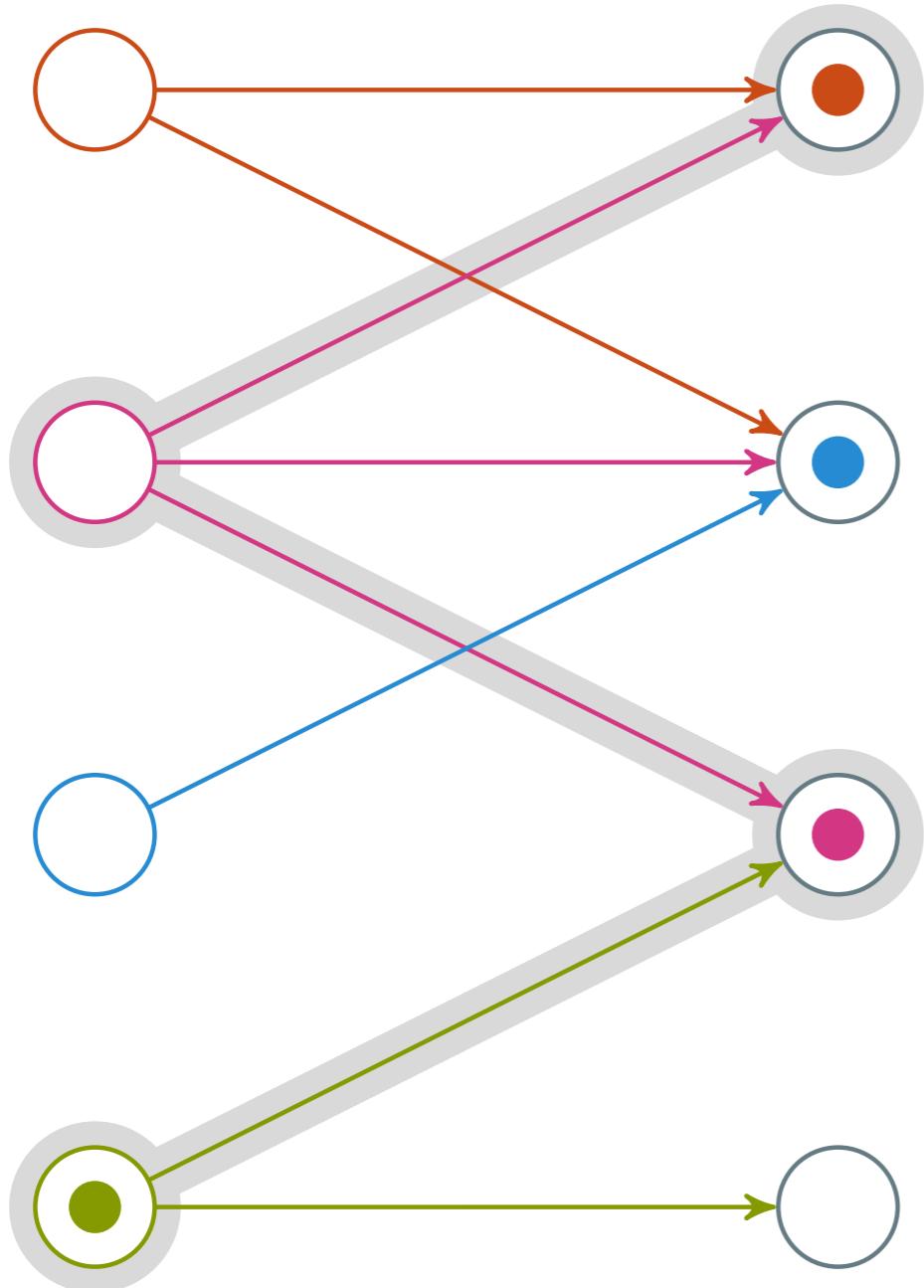


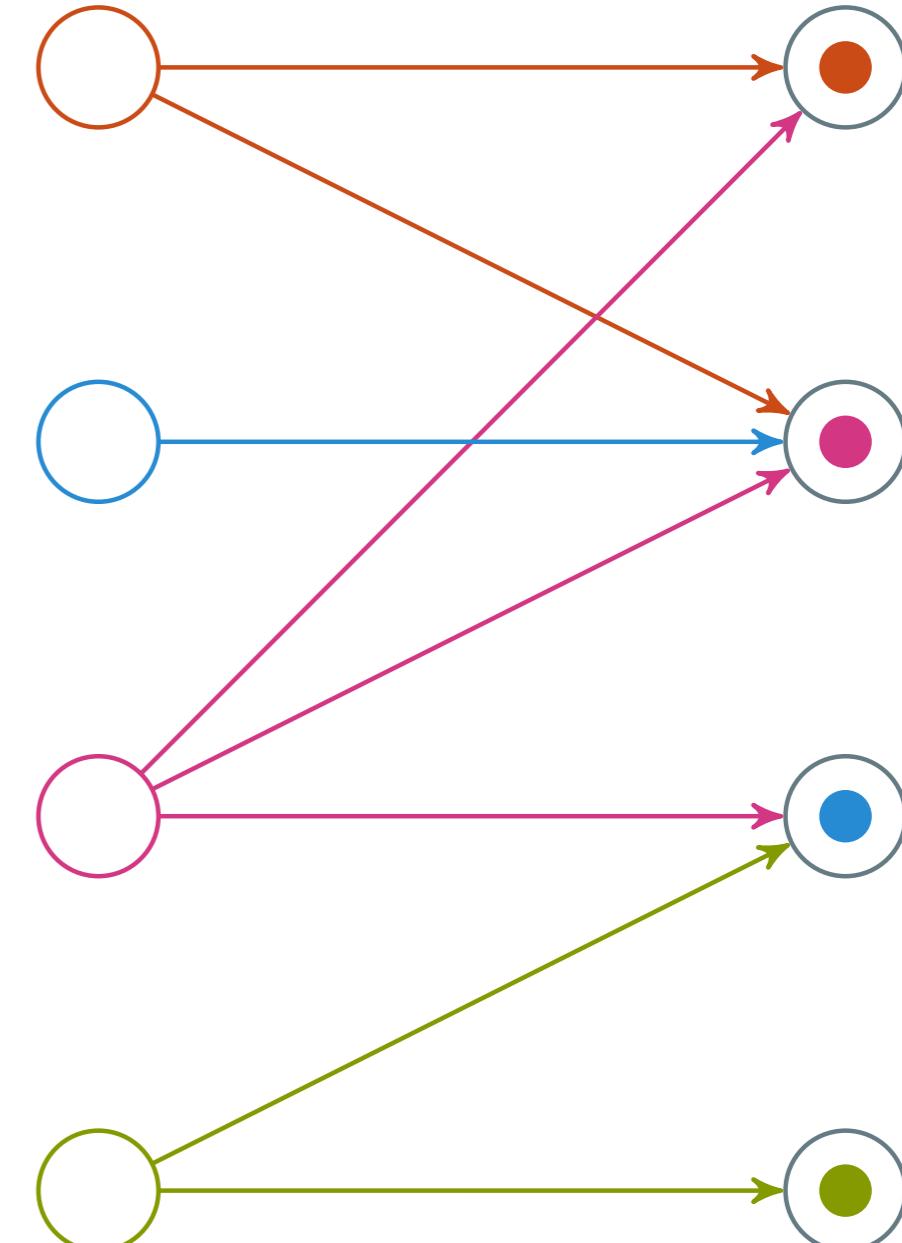
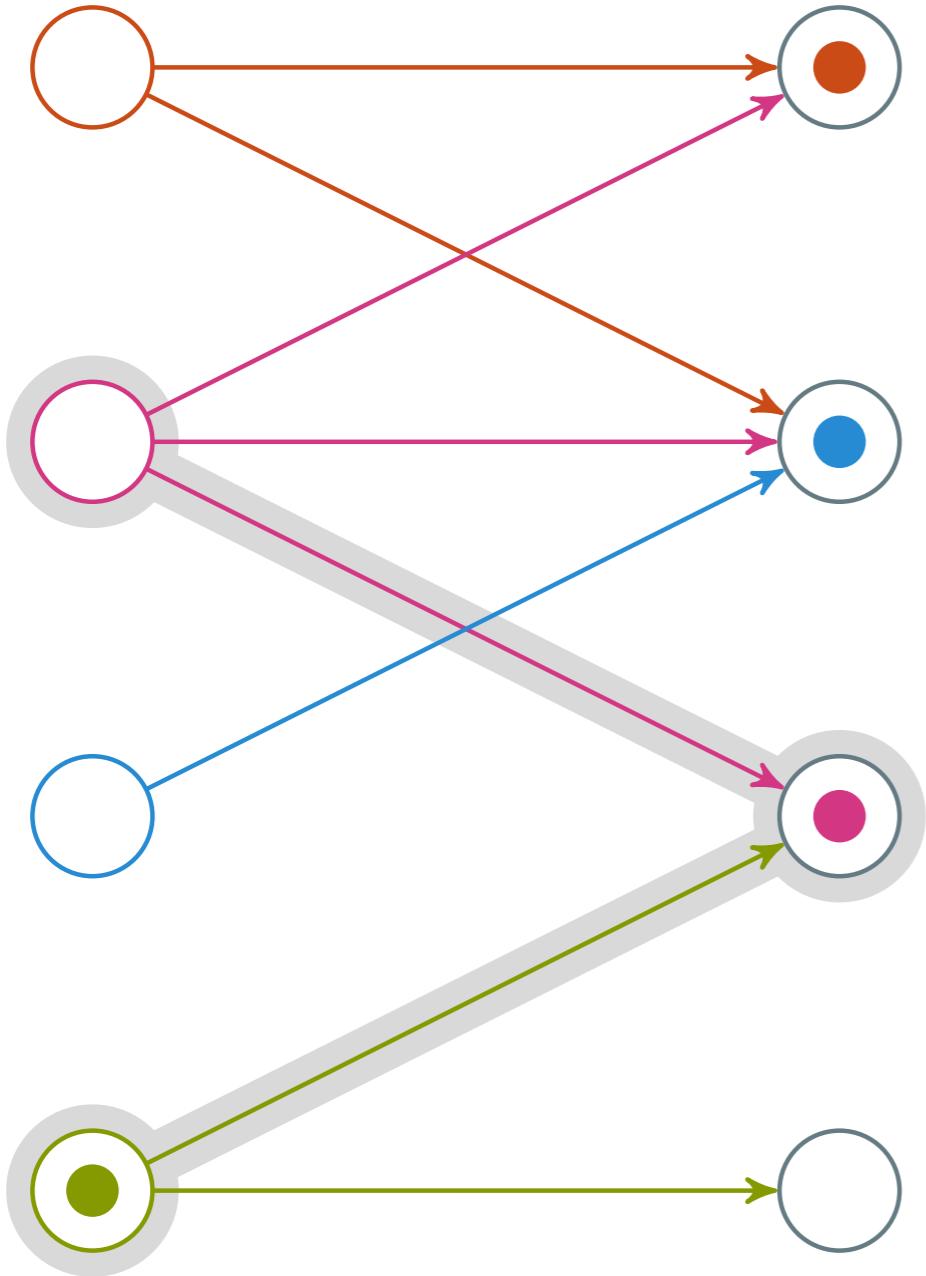


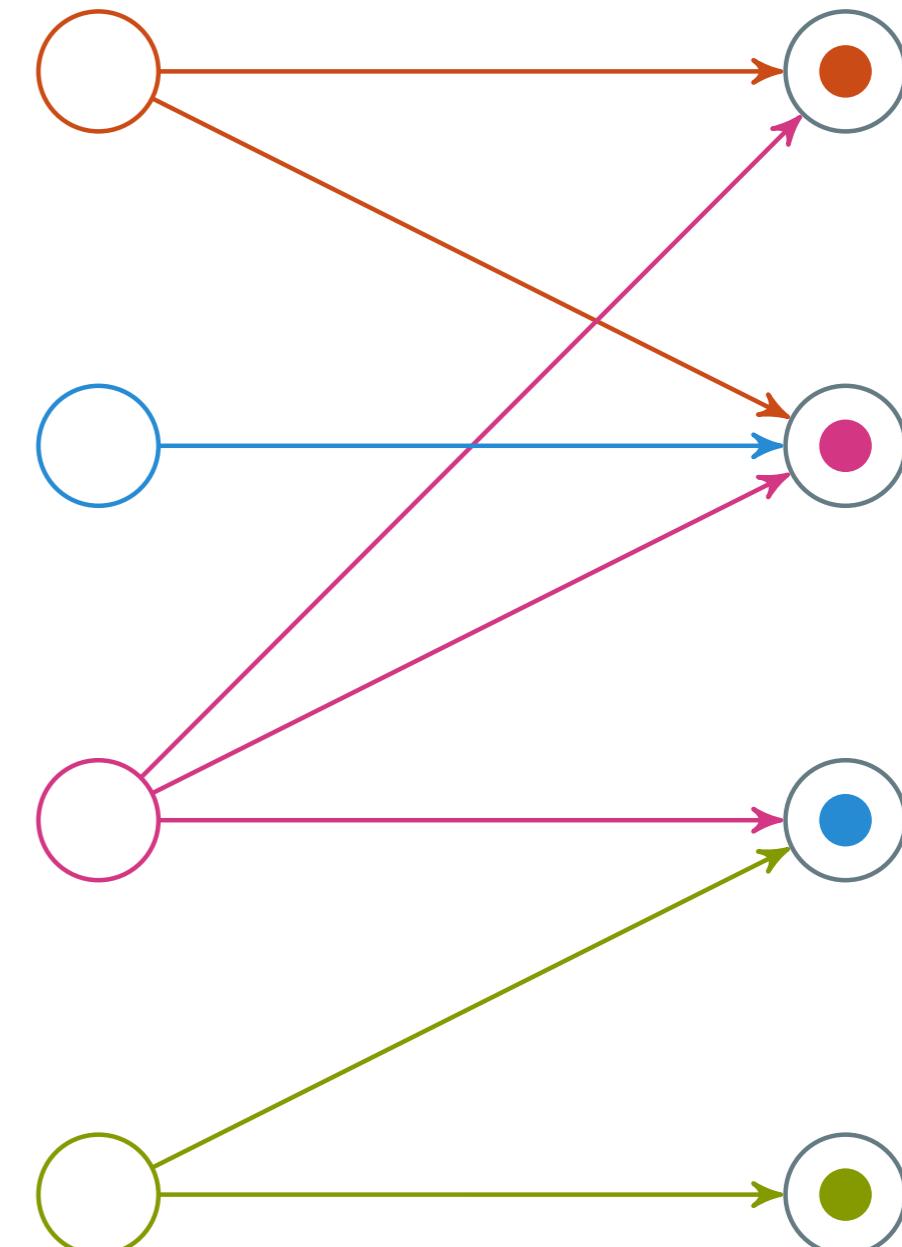
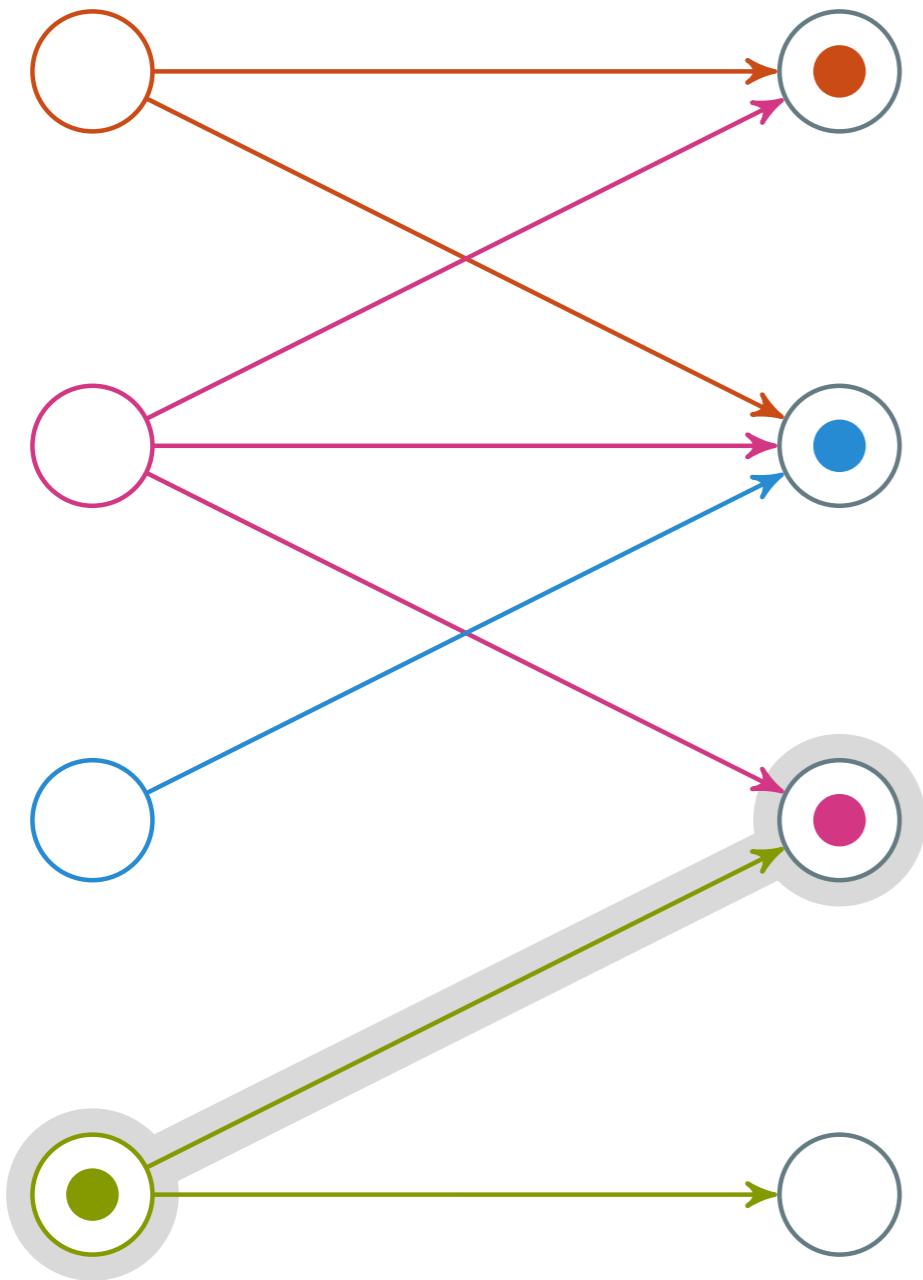


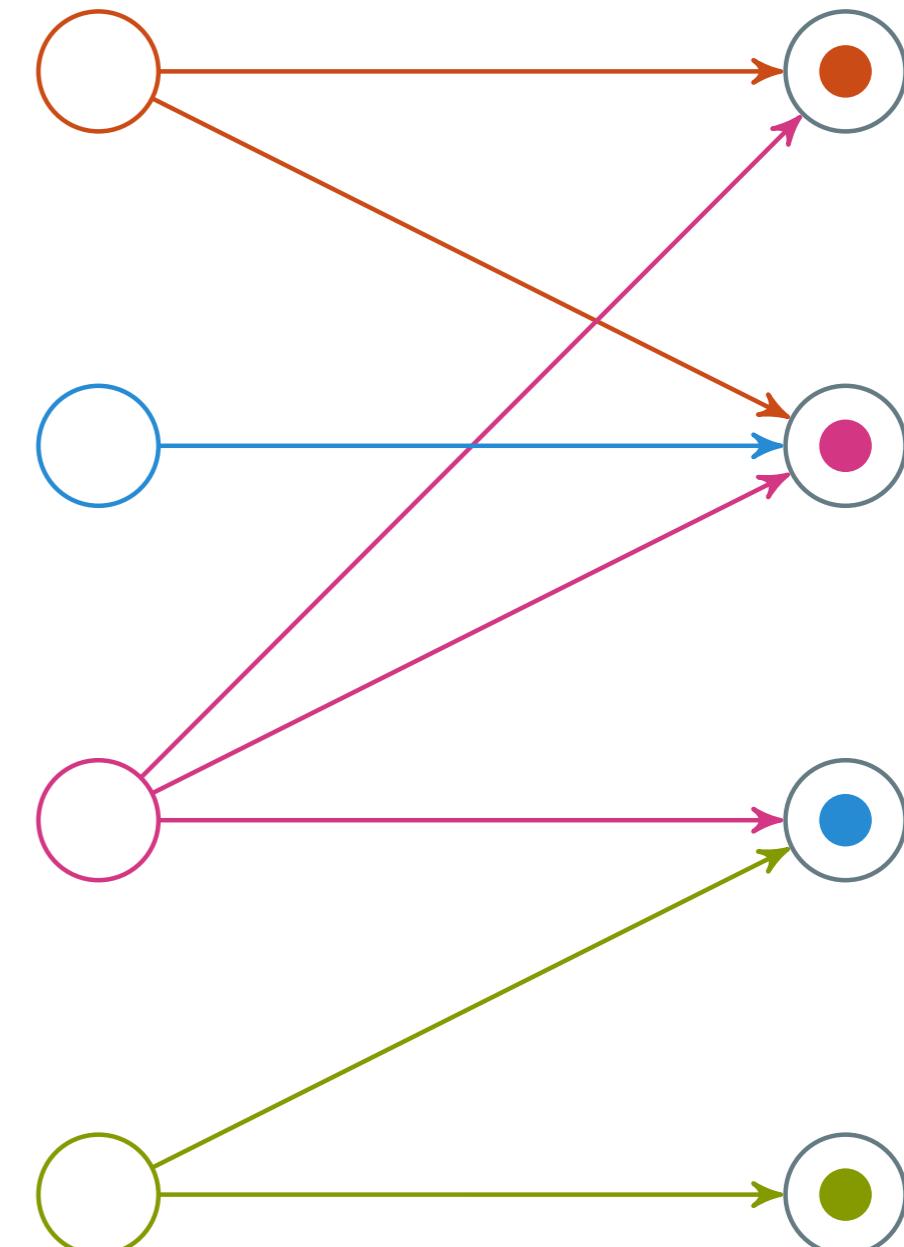
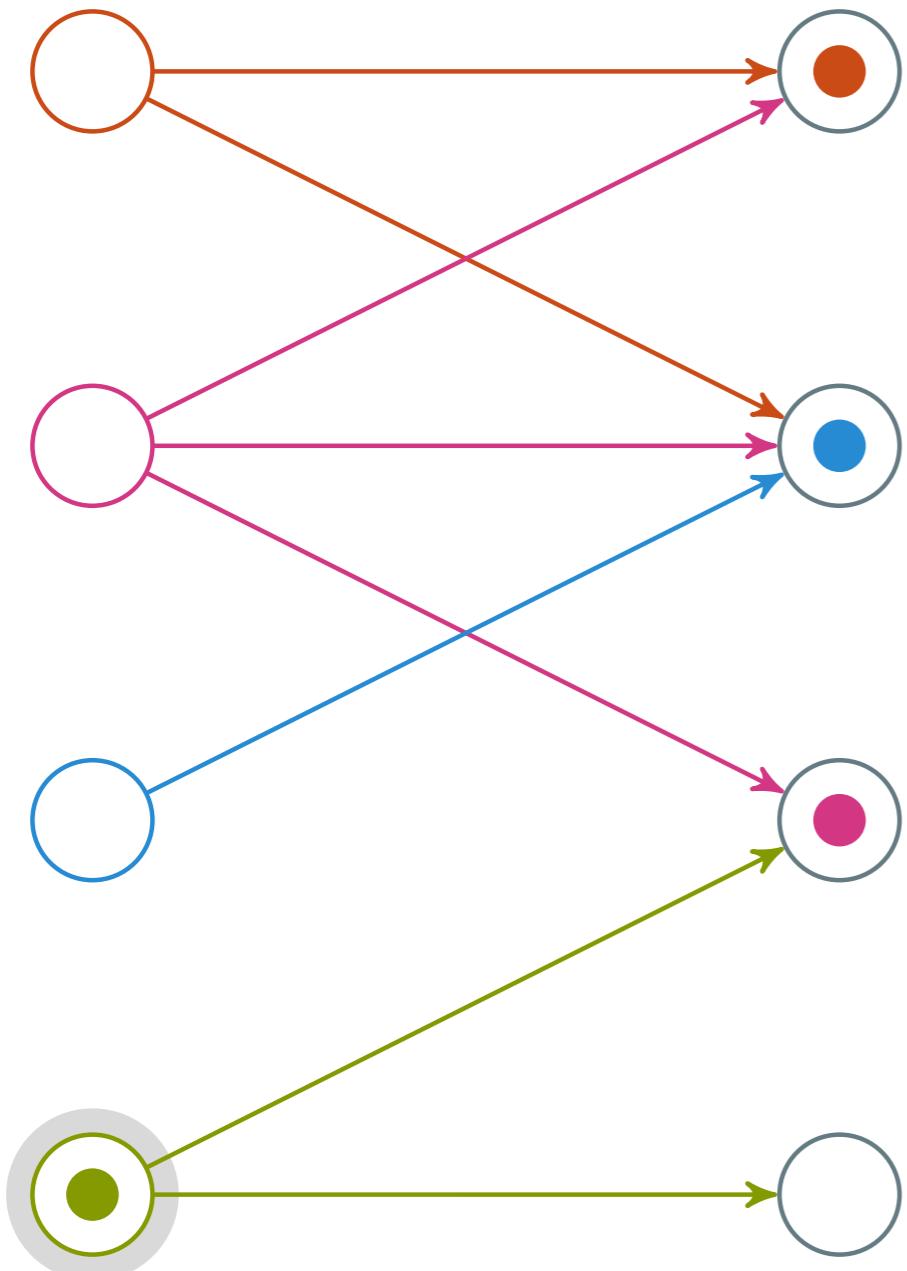


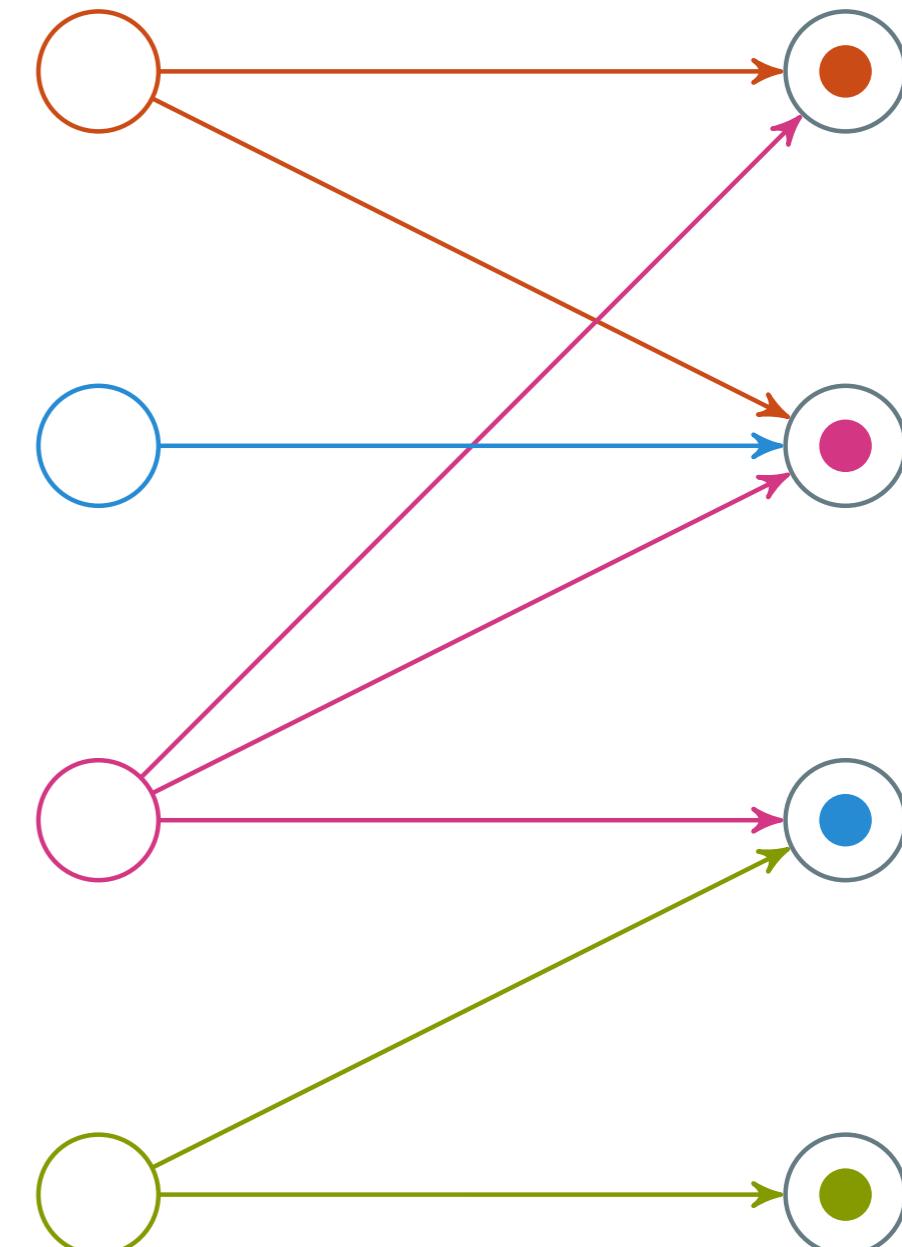
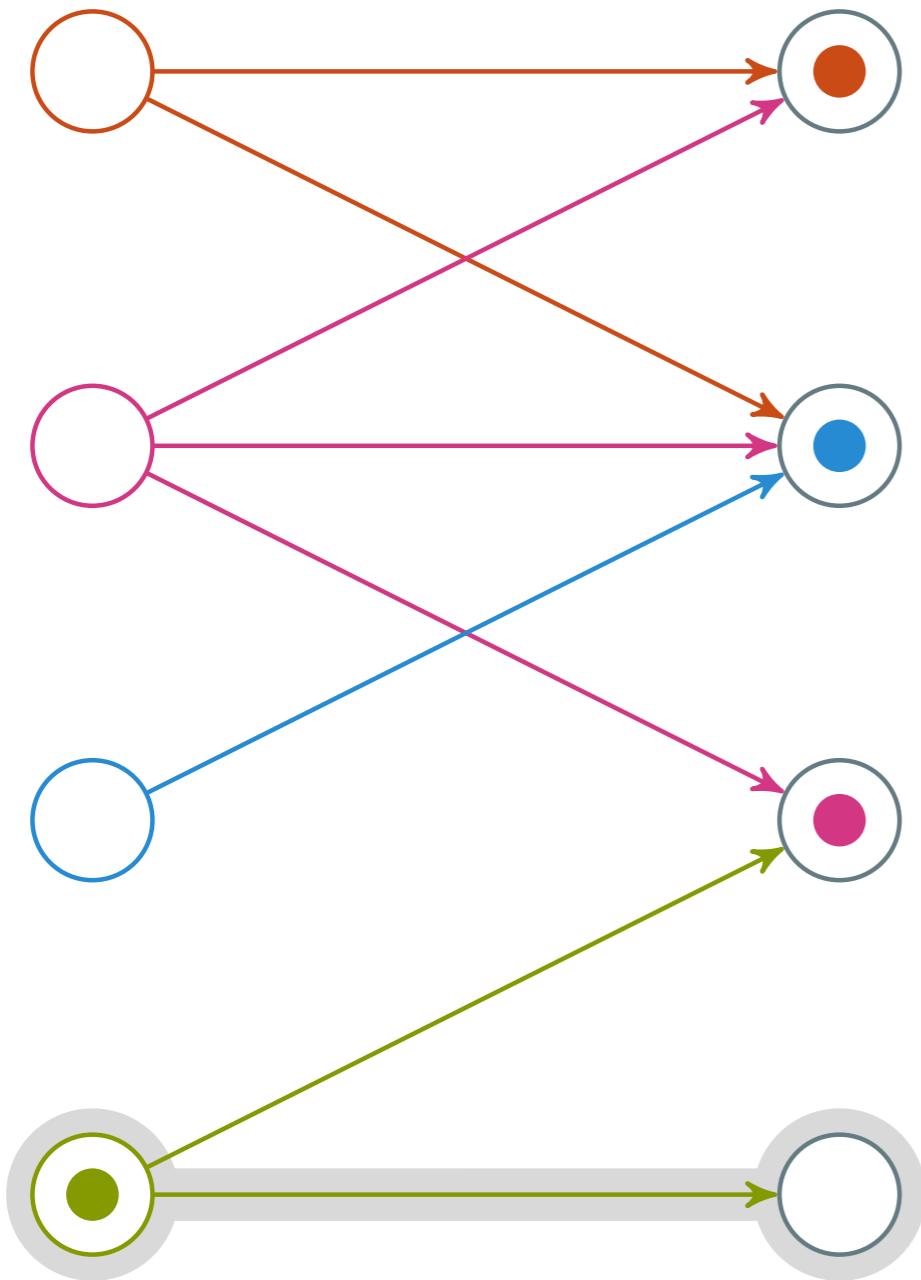


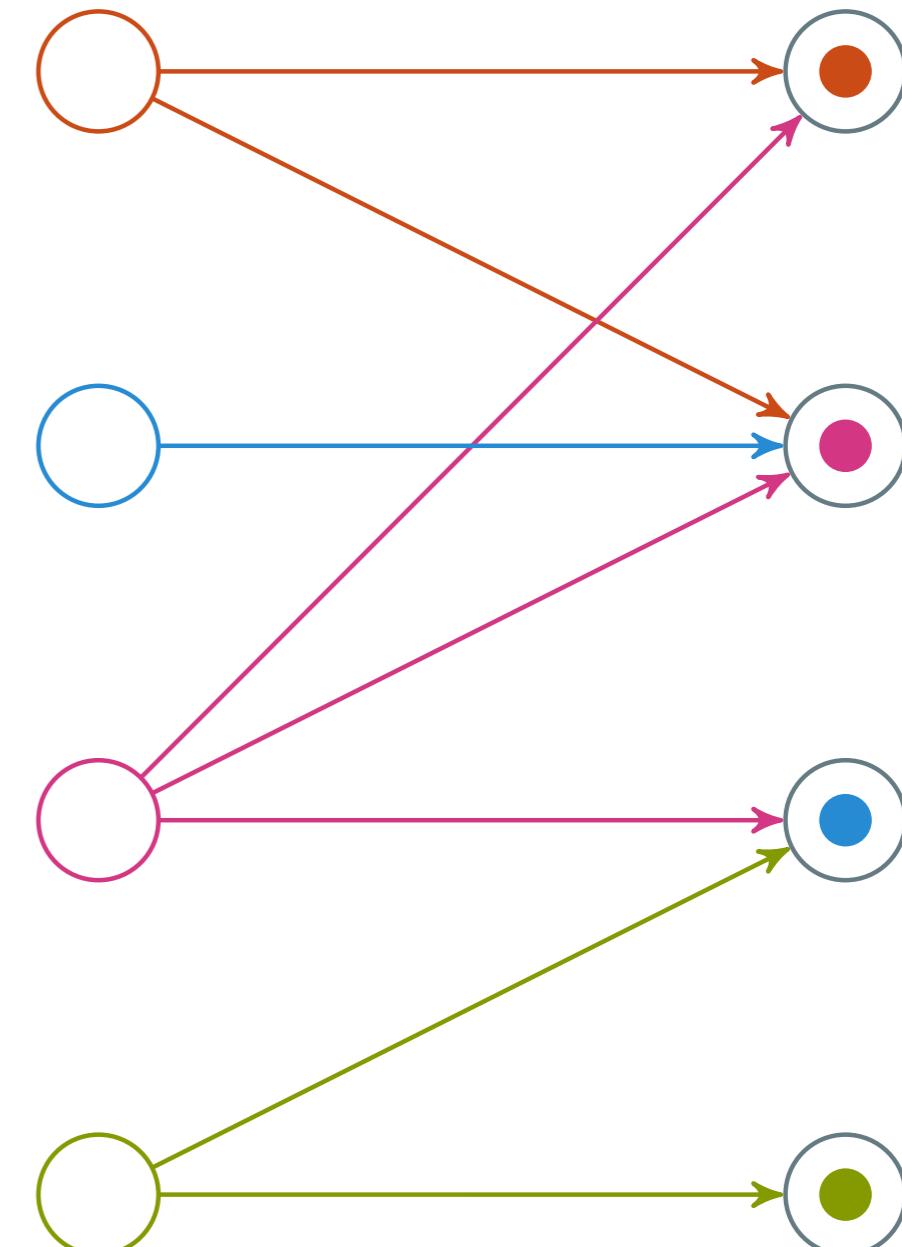
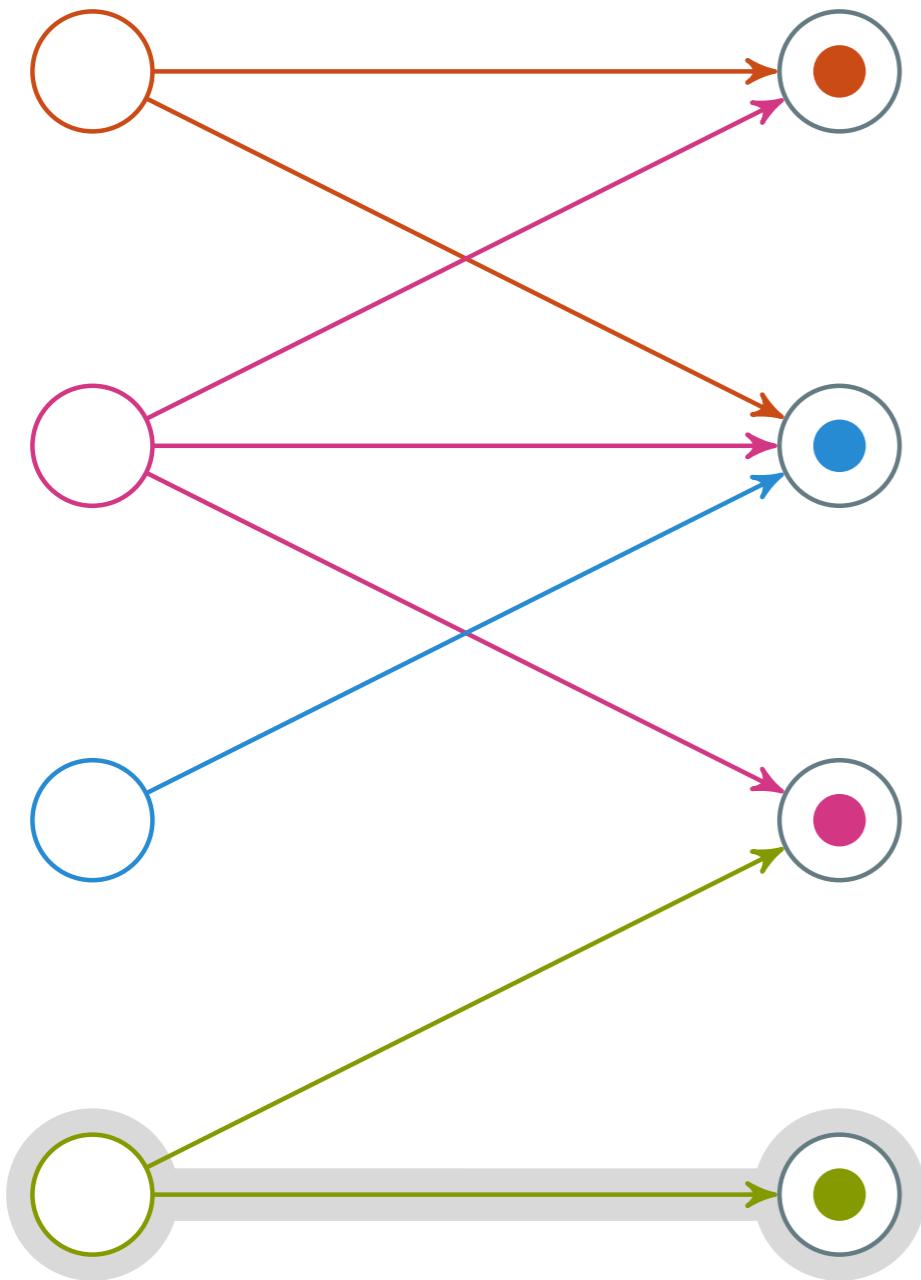


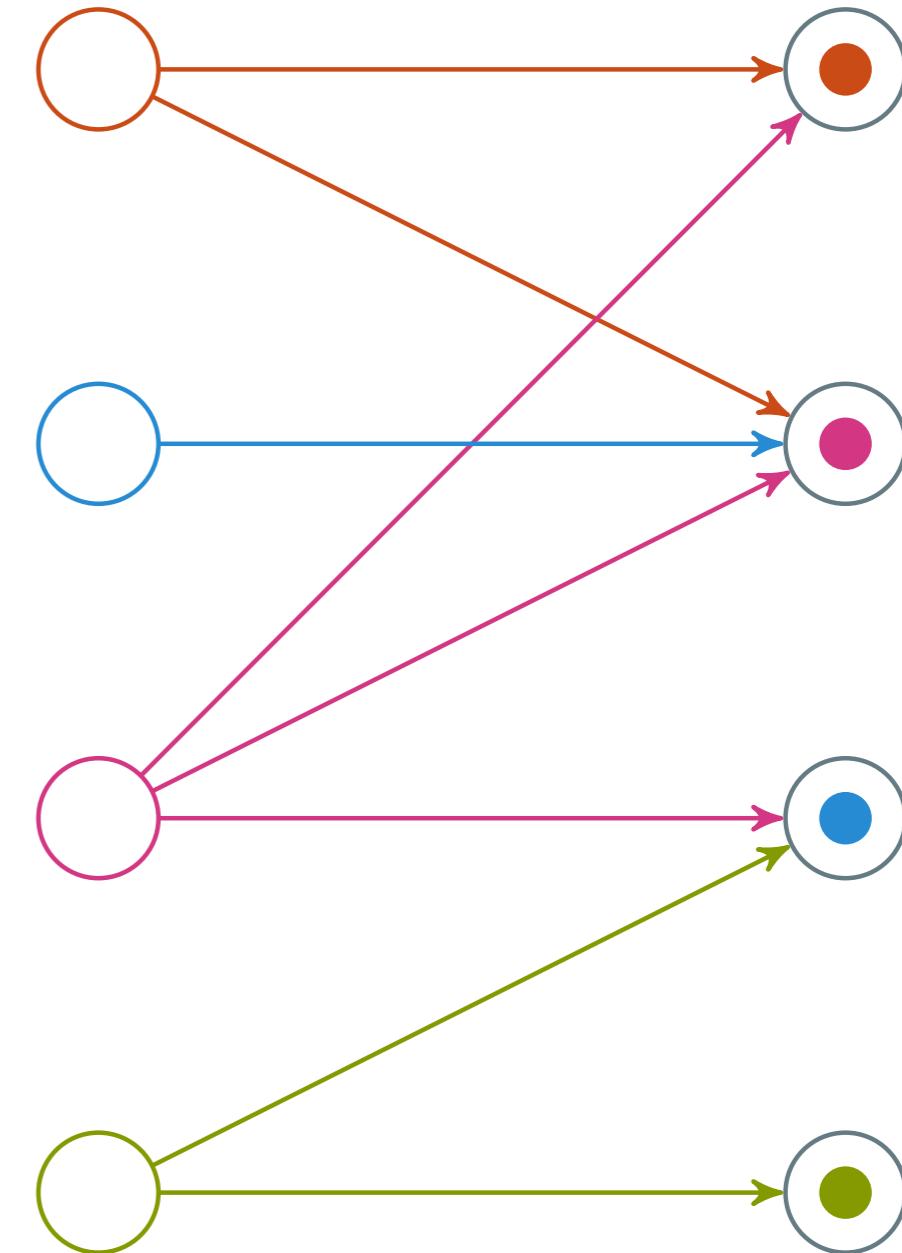
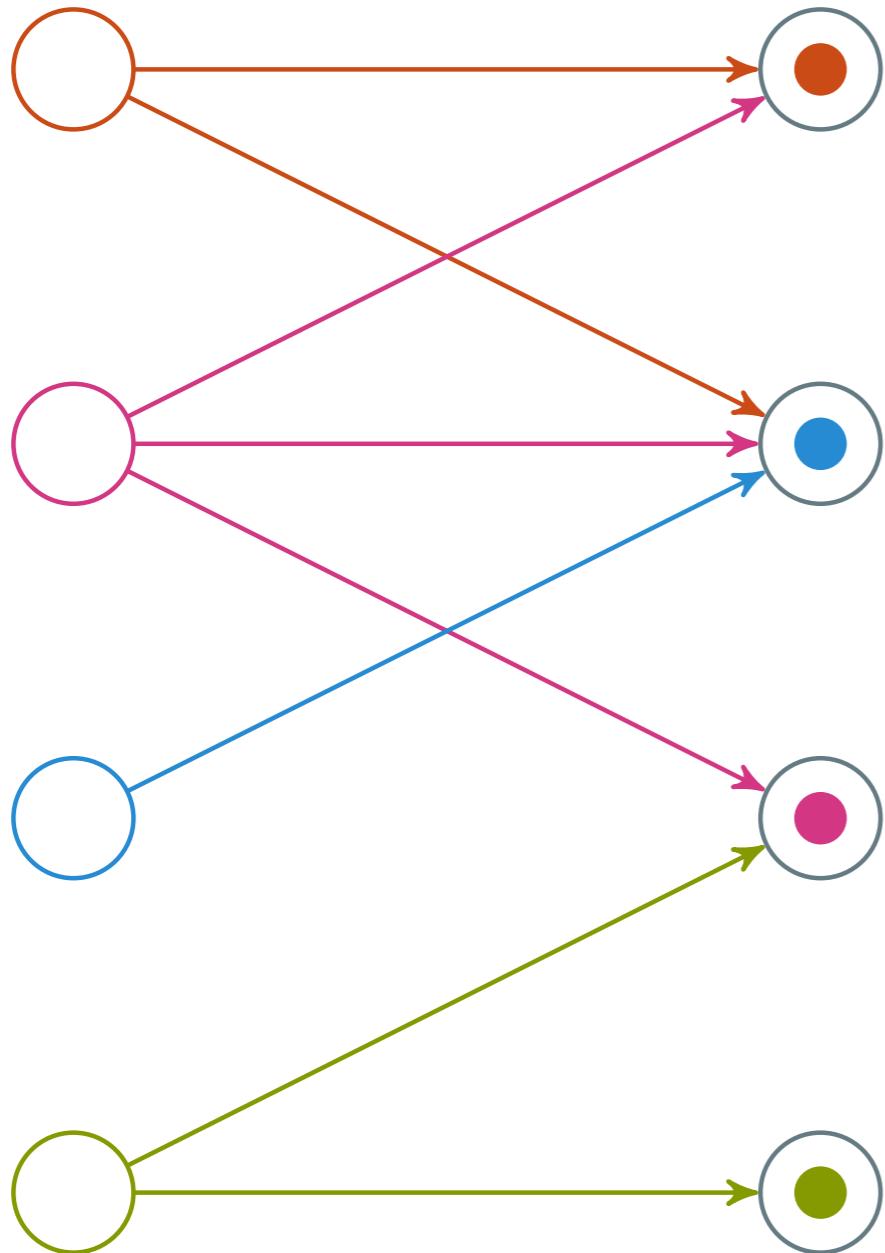












Ford-Fulkerson



Brute force



0

100 %

$$n = 4$$

Her måtte vi prøve 24 mulige permutasjoner, mens Ford-Fulkerson måtte utføre 39 «operasjoner» (ikke så presist definert her). Så idet brute-force-løsningen var ferdig, så hadde F-F fortsatt et stykke igjen:

Ford-Fulkerson



Brute force



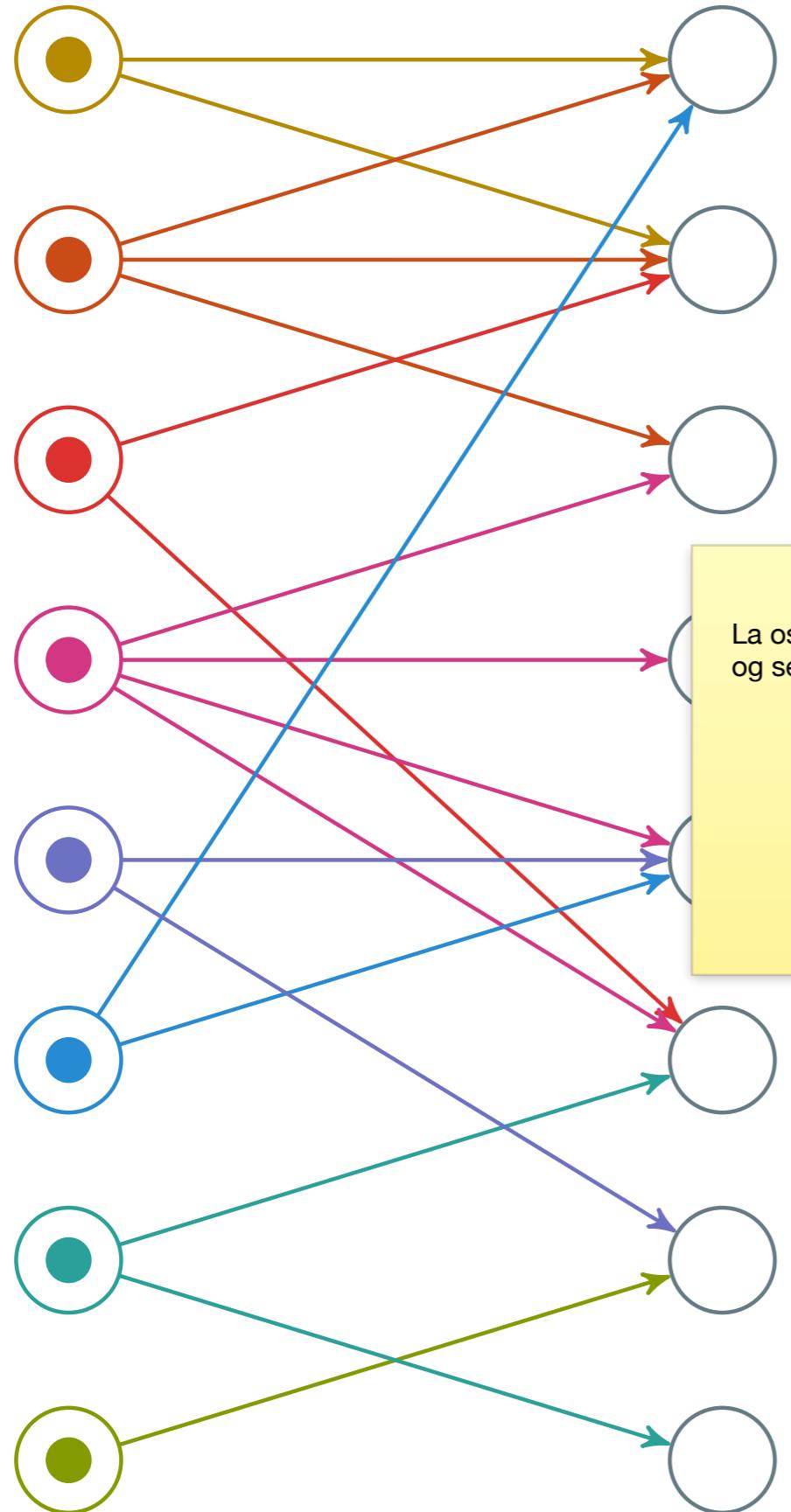
0

100 %

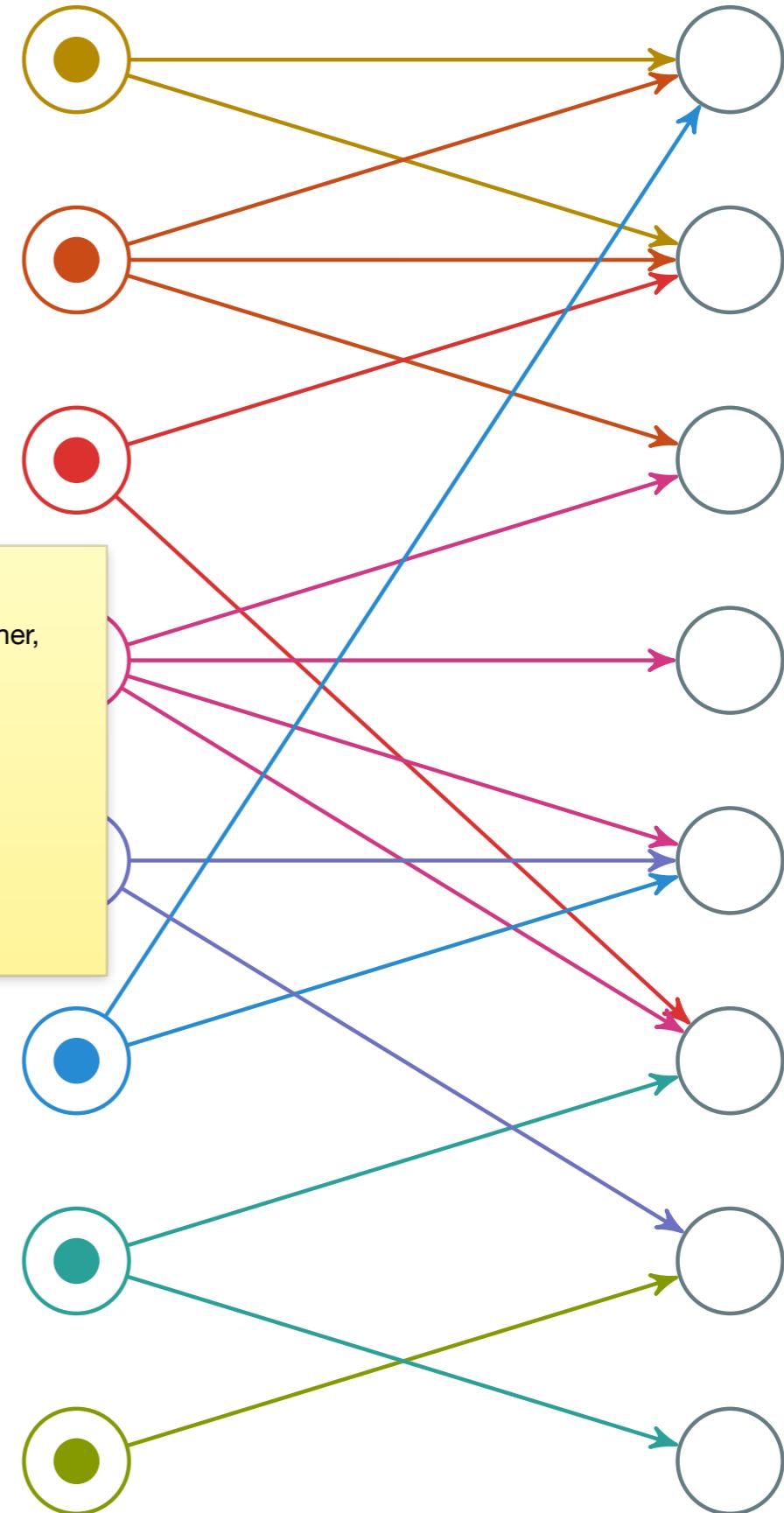
$$n = 4$$

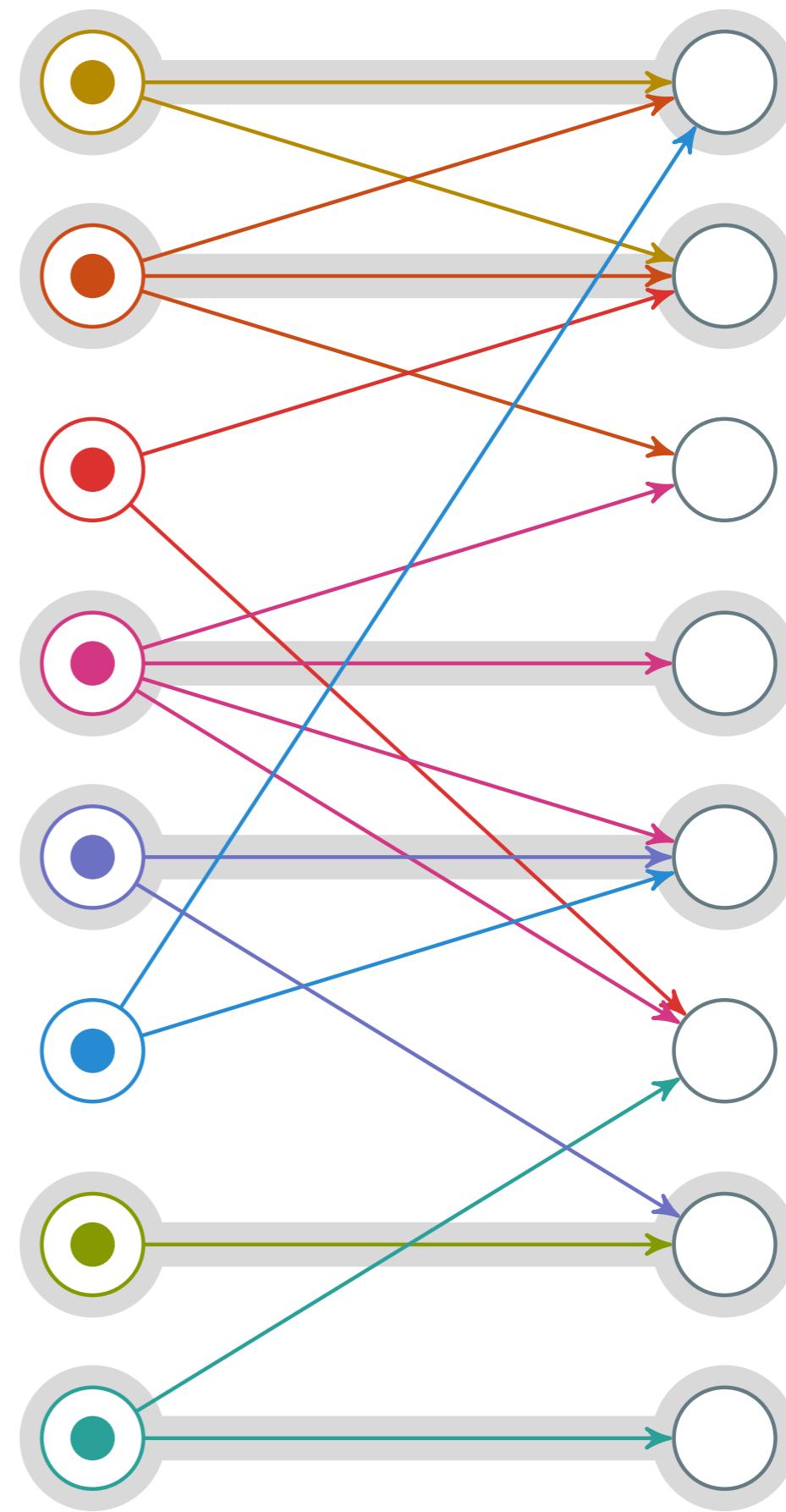
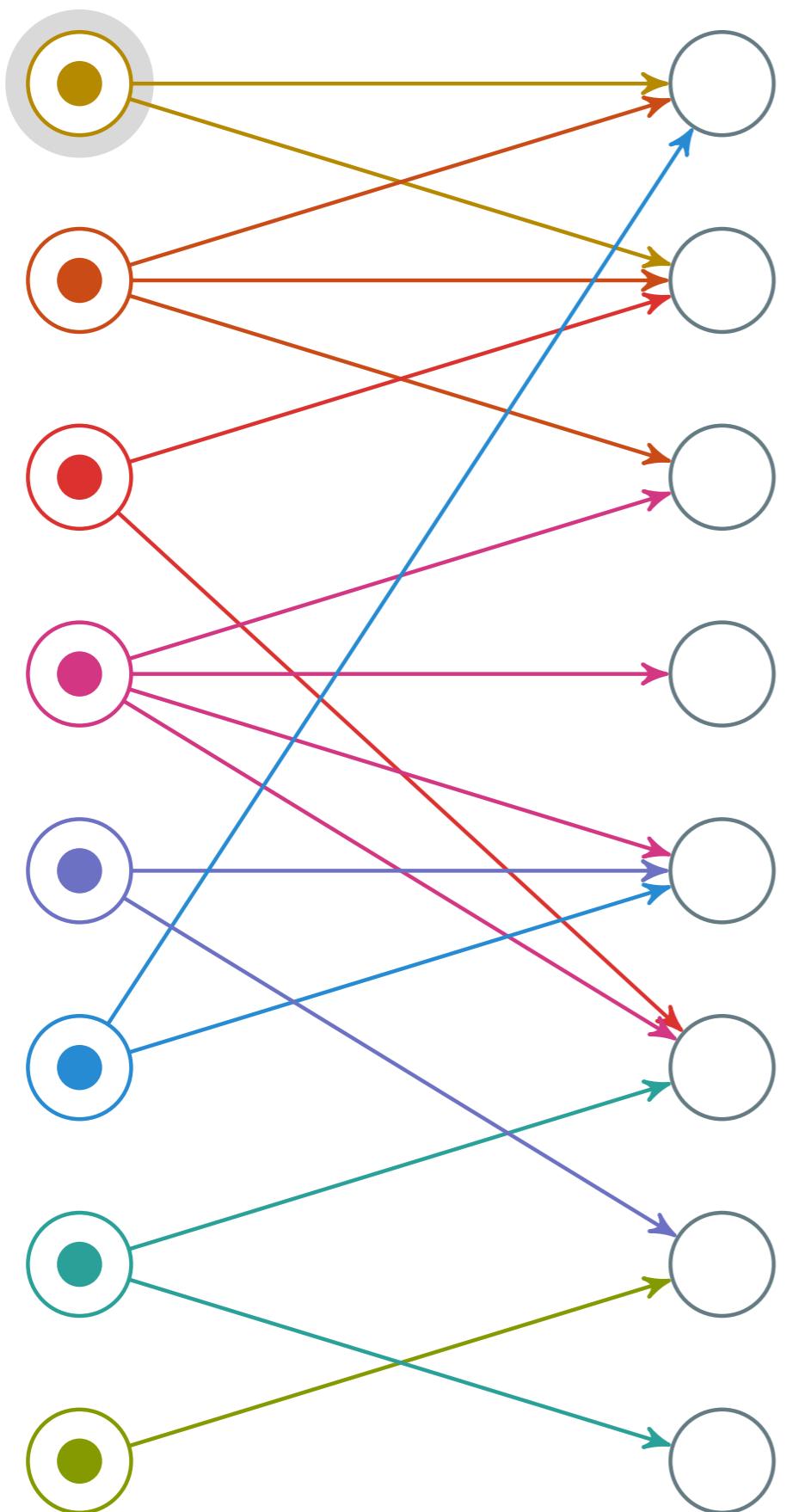
Ford-Fulkerson er litt komplisert
– kanskje en enkel brute-force-
løsning er like grei, da? Eller?

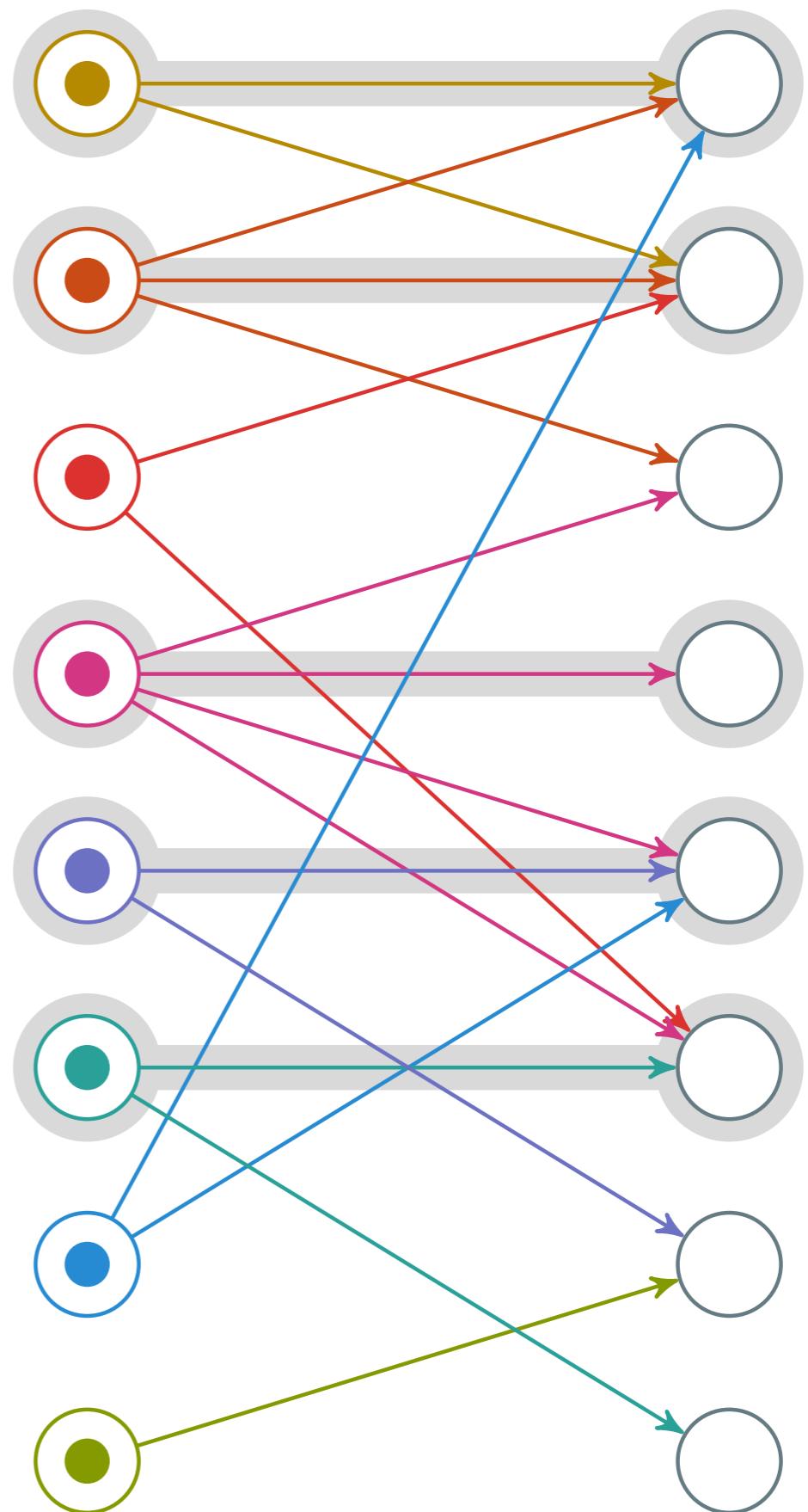
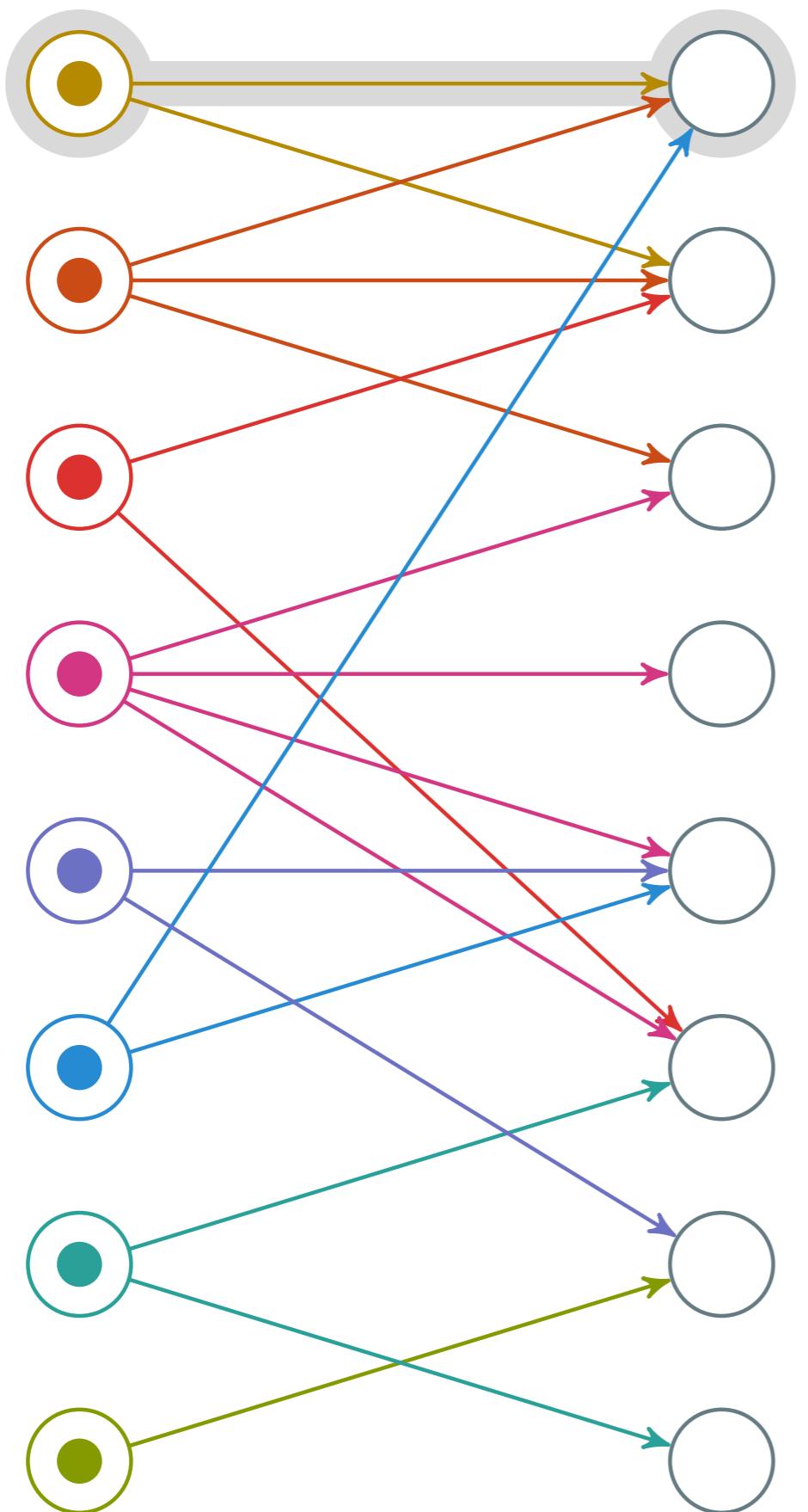
(«When in doubt, use brute
force» – Rob Pike)

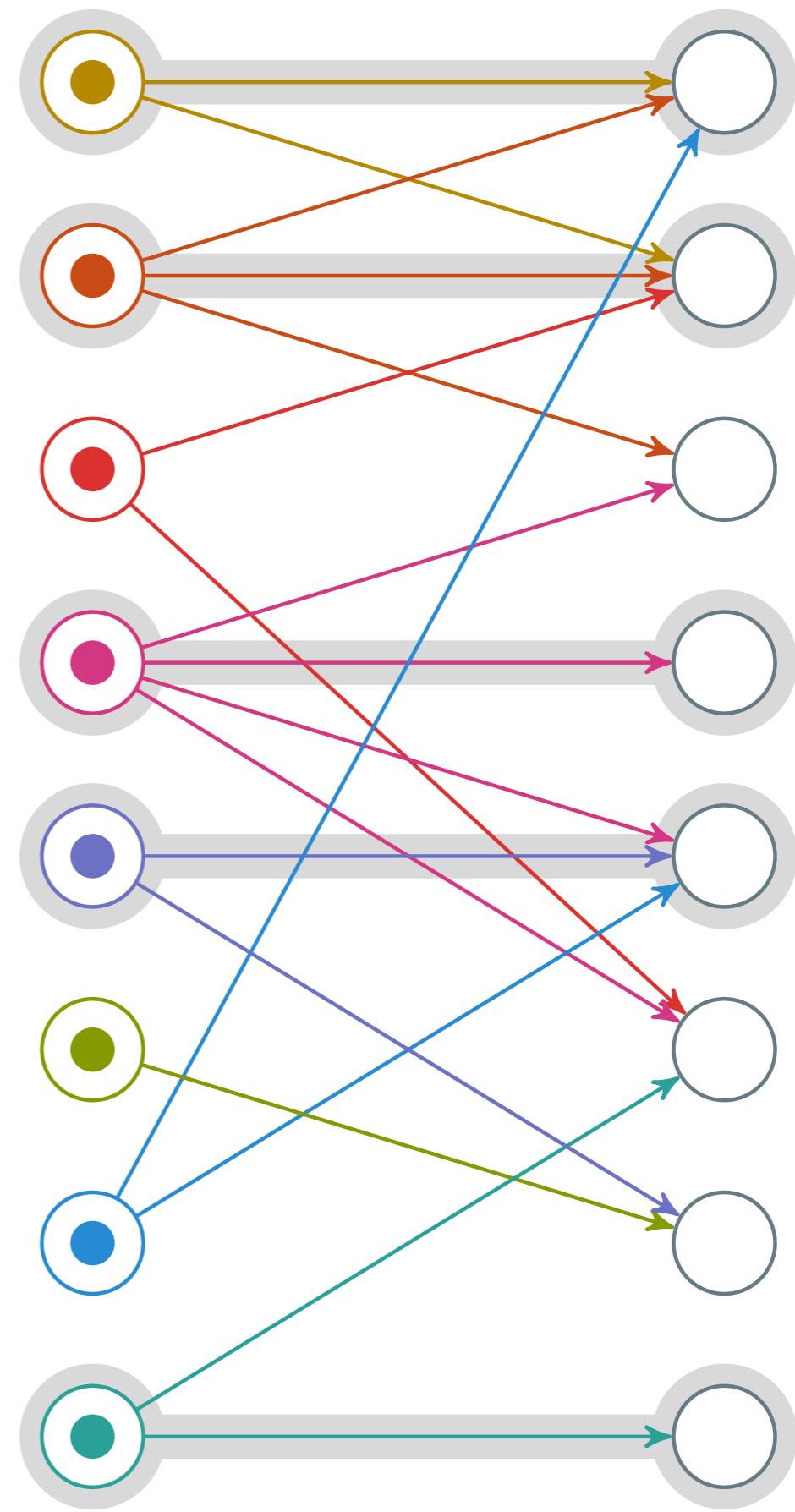
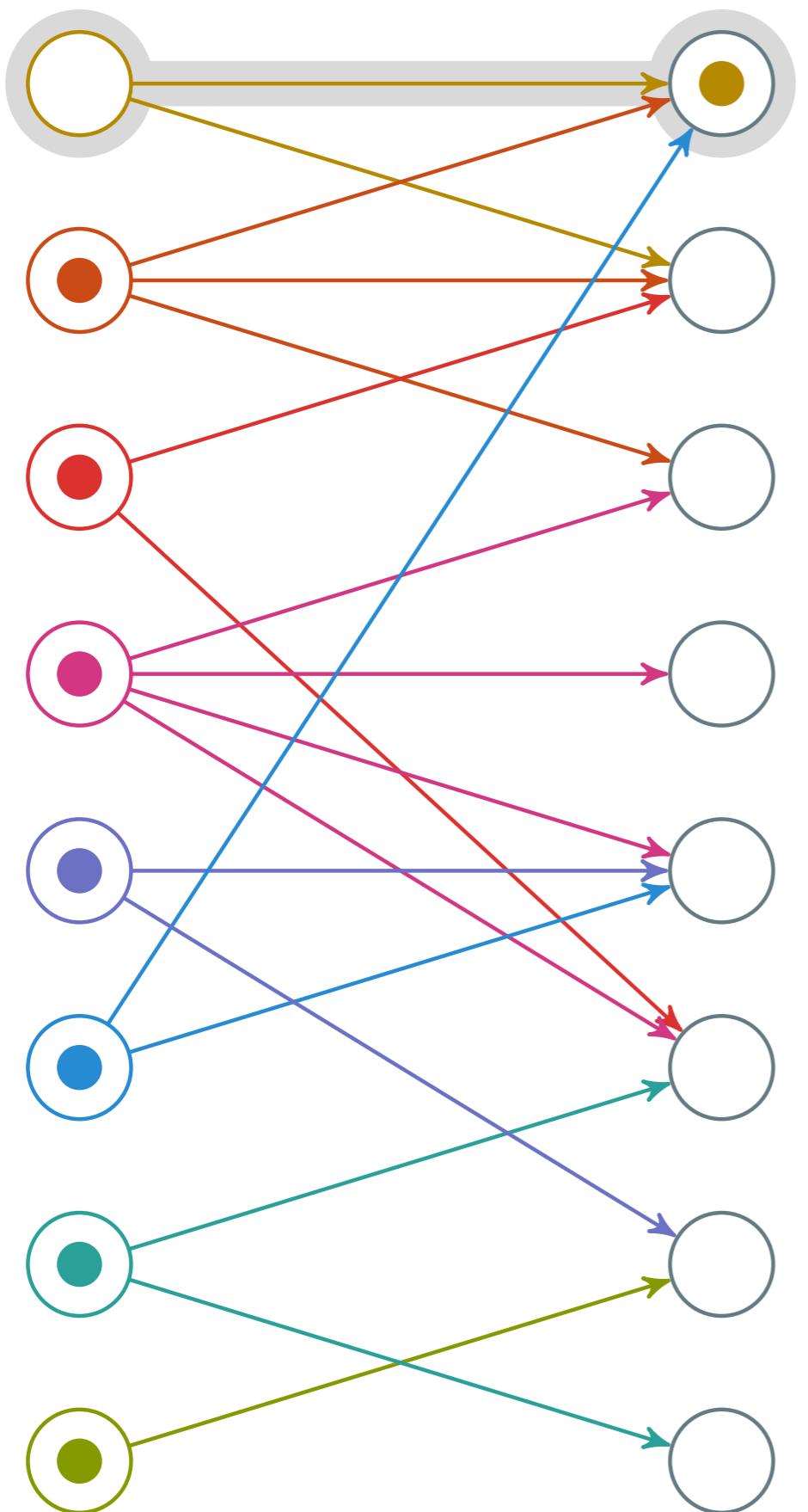


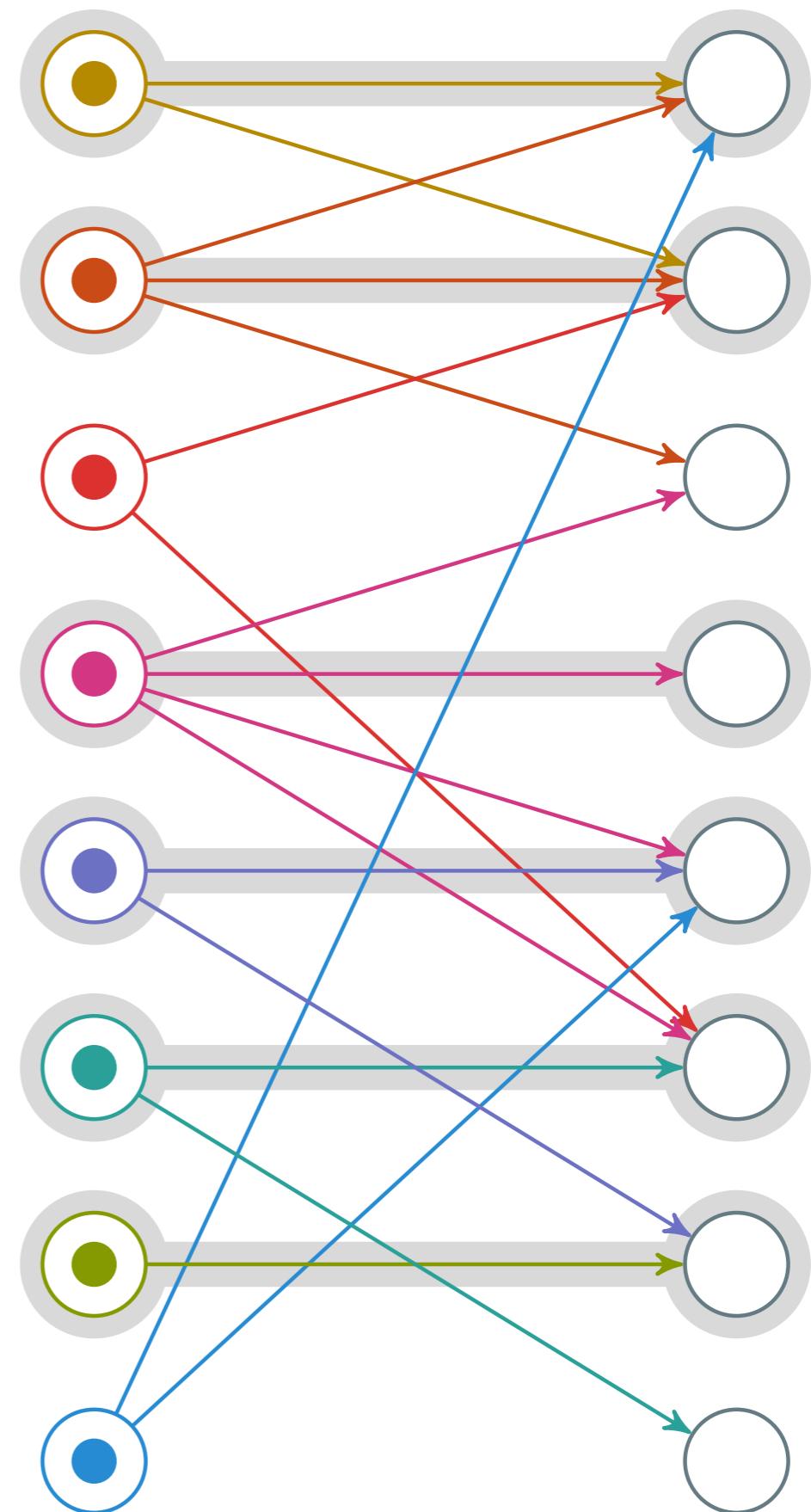
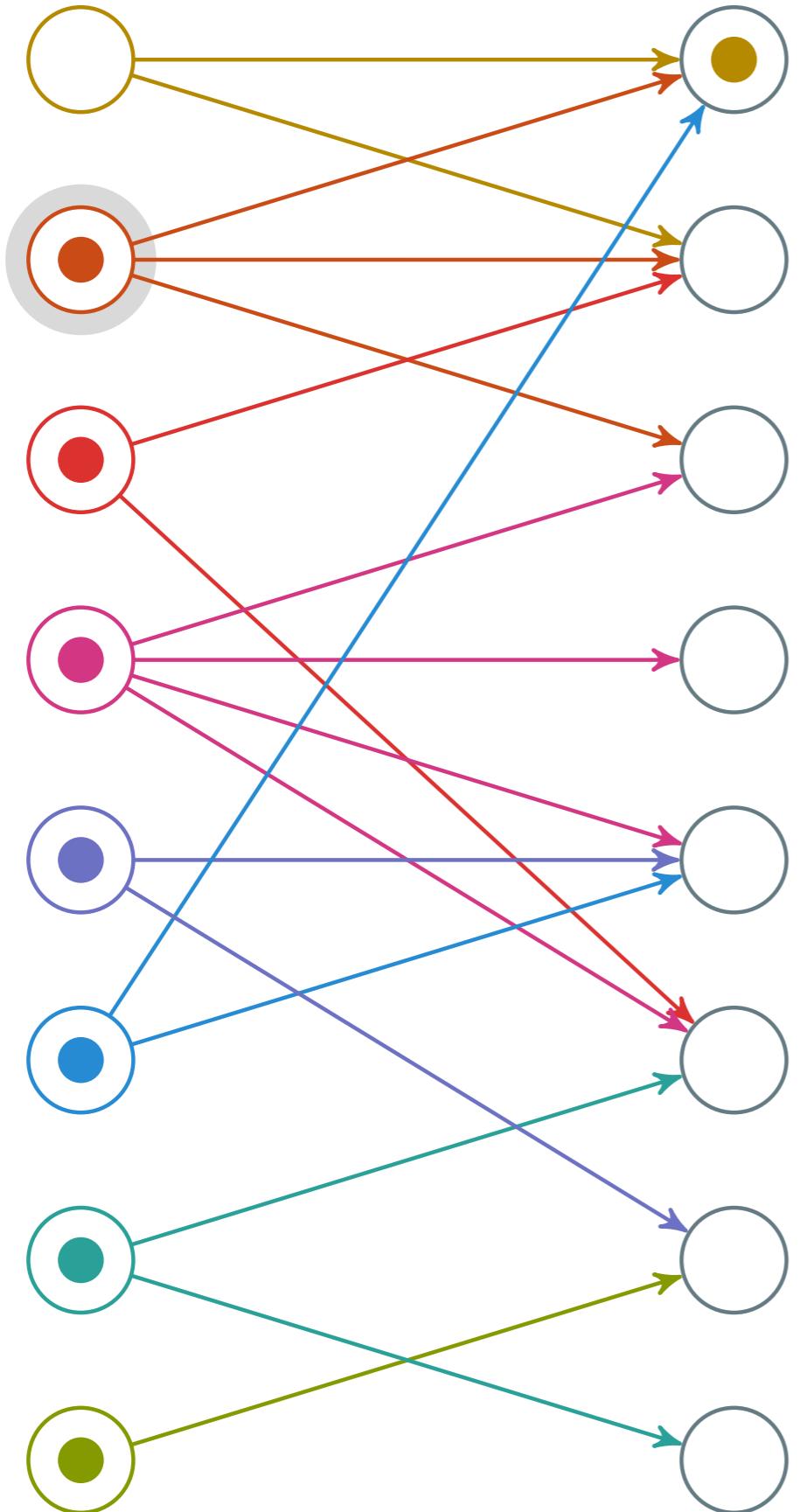
La oss doble antallet personer,
og se hva som skjer.

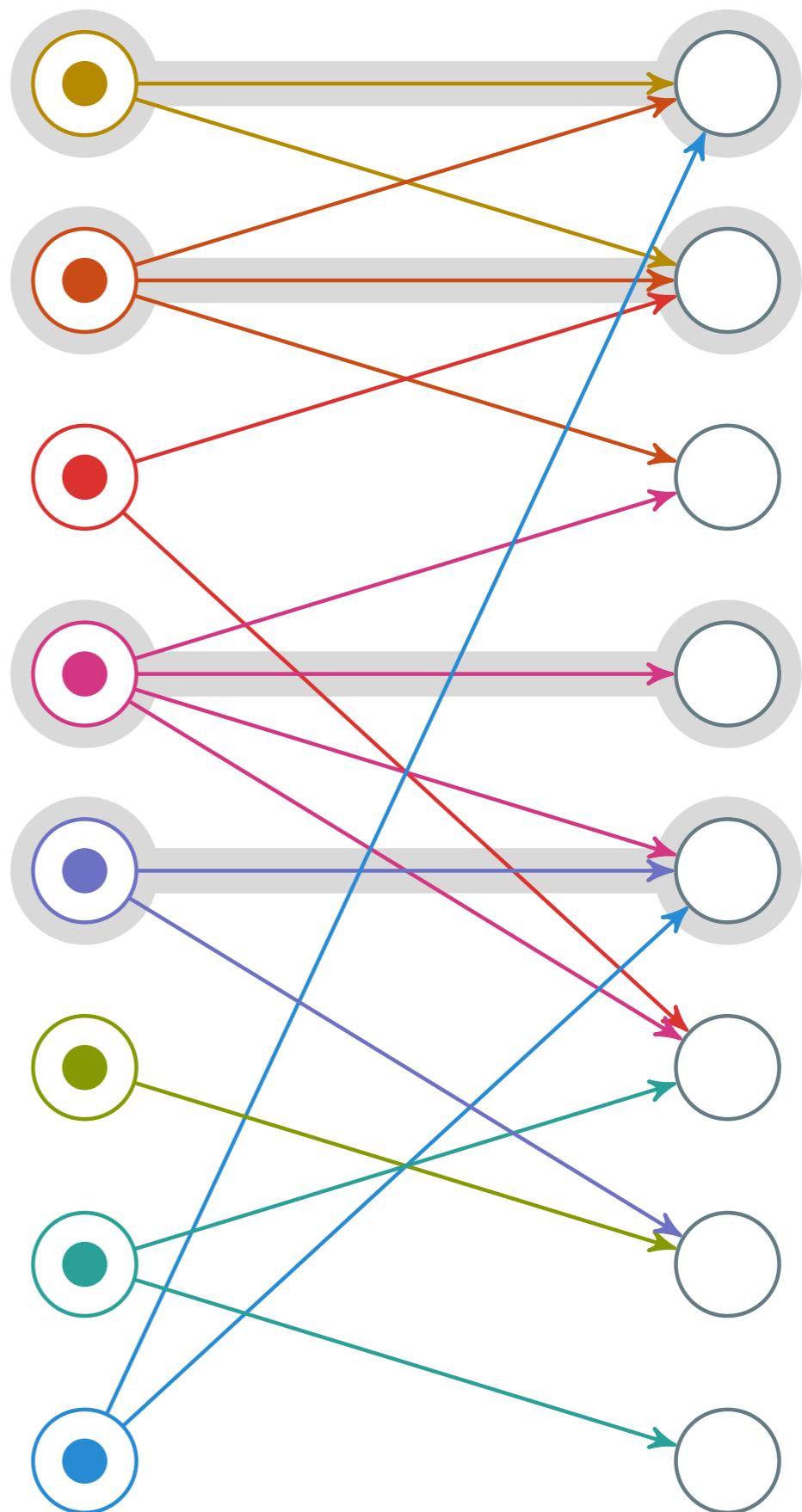
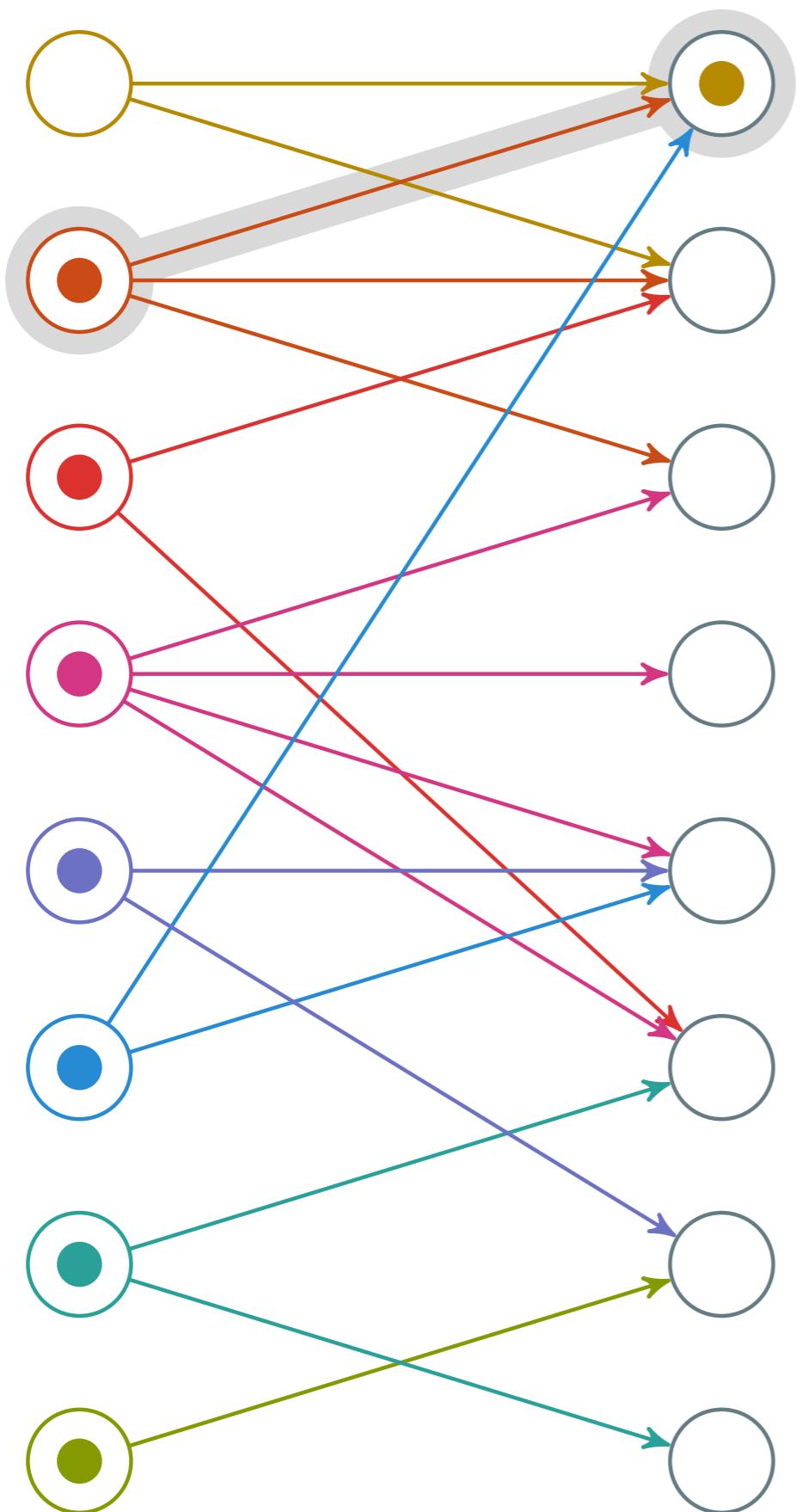


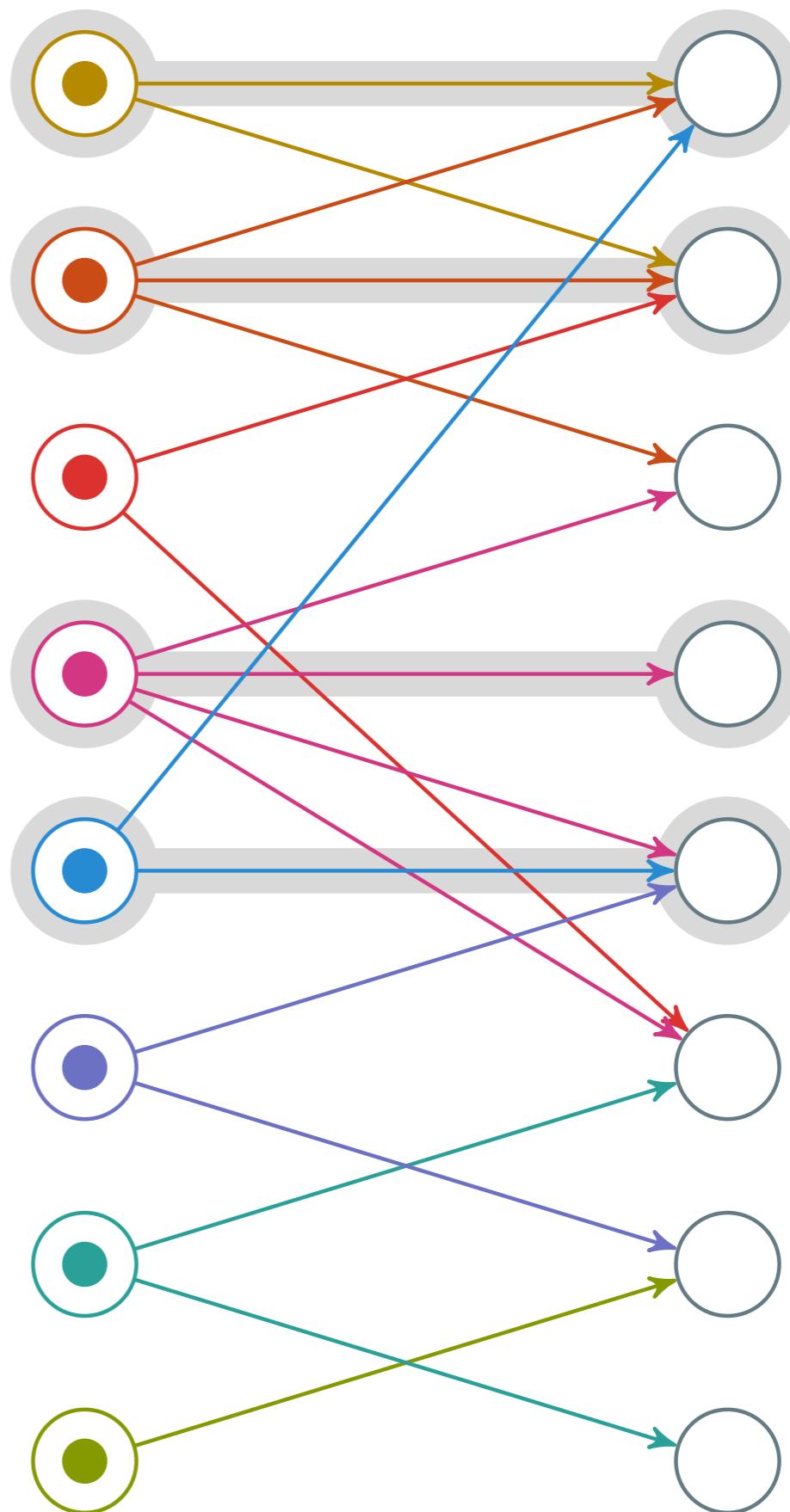
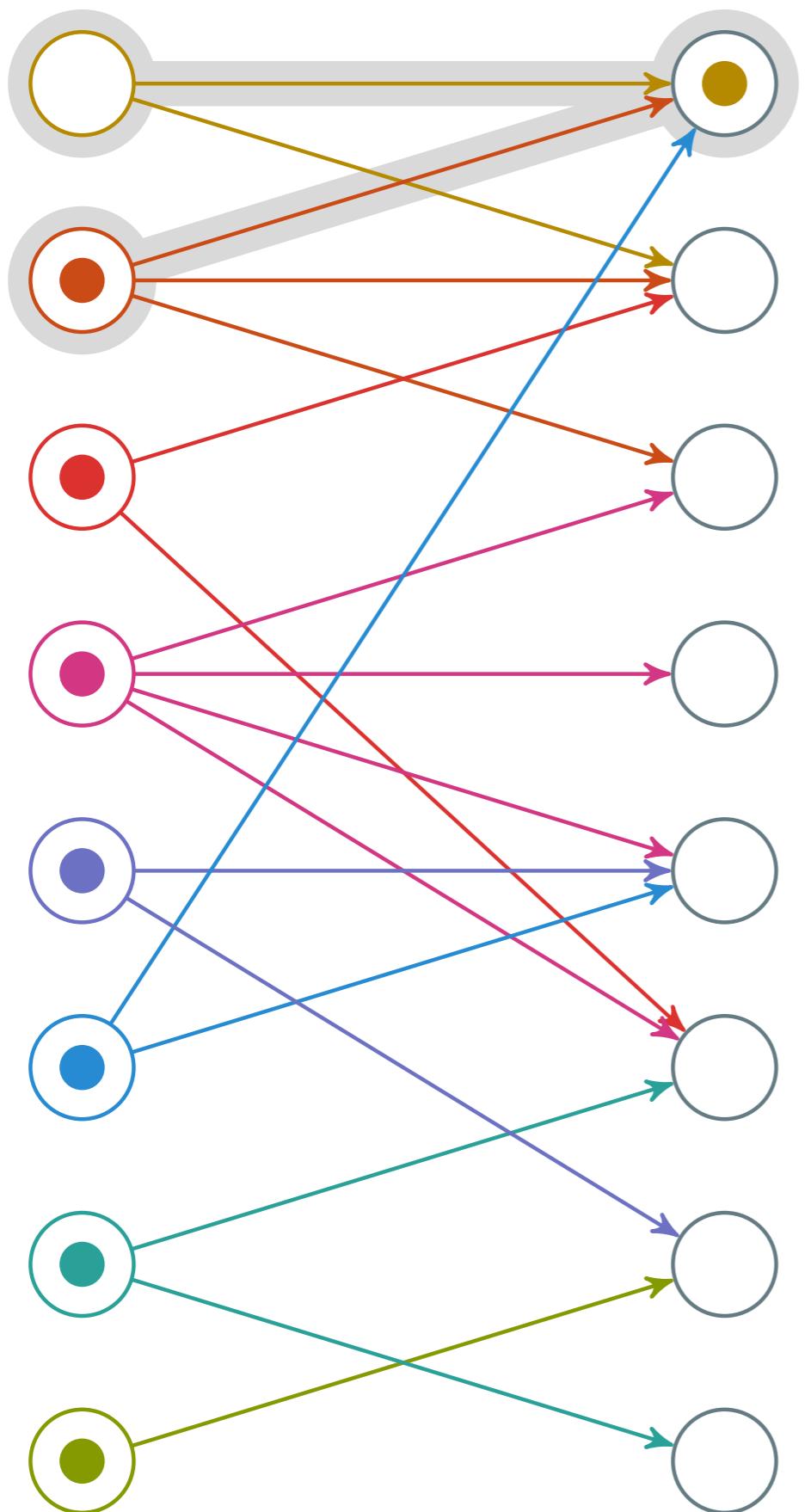


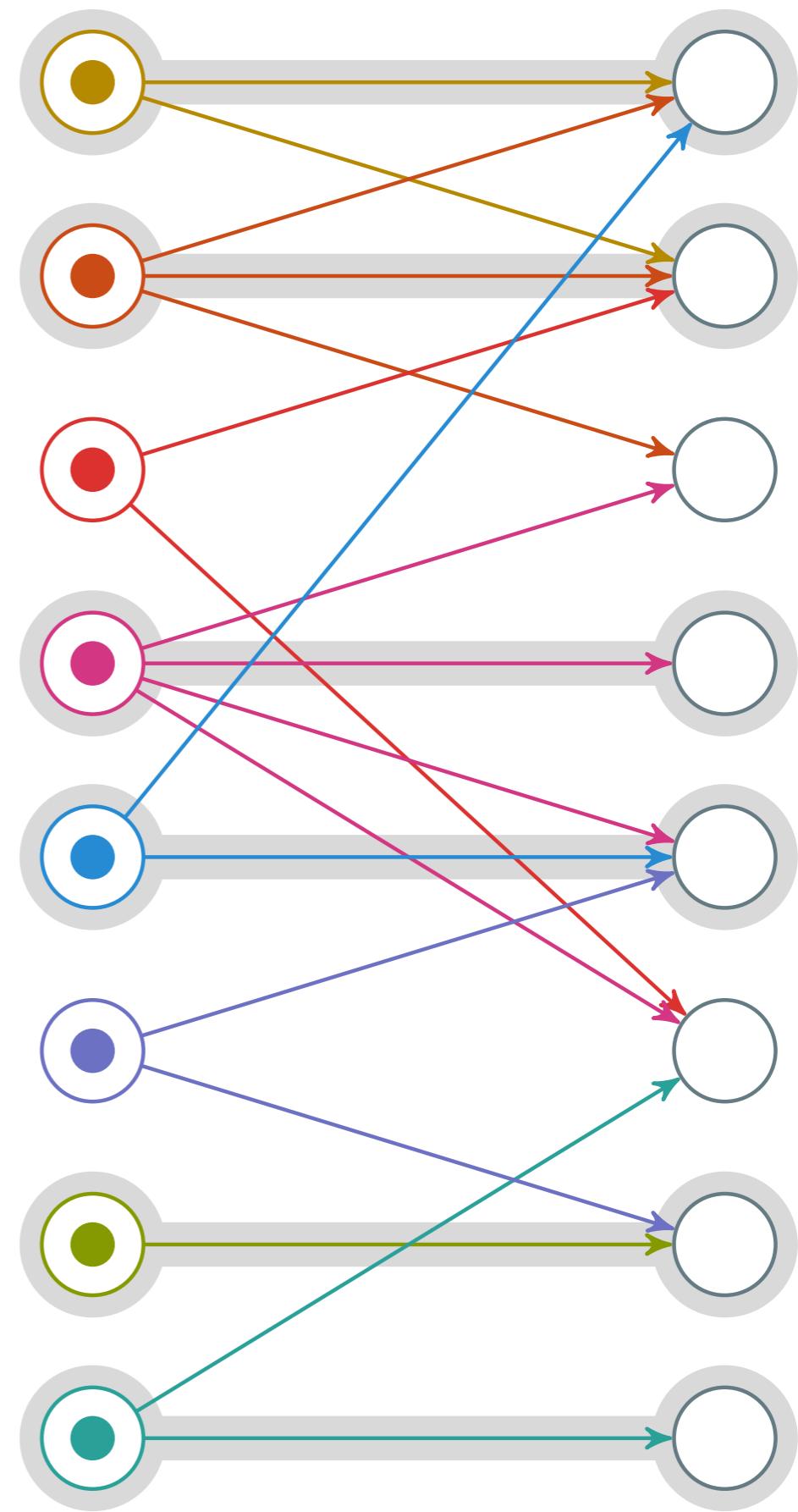
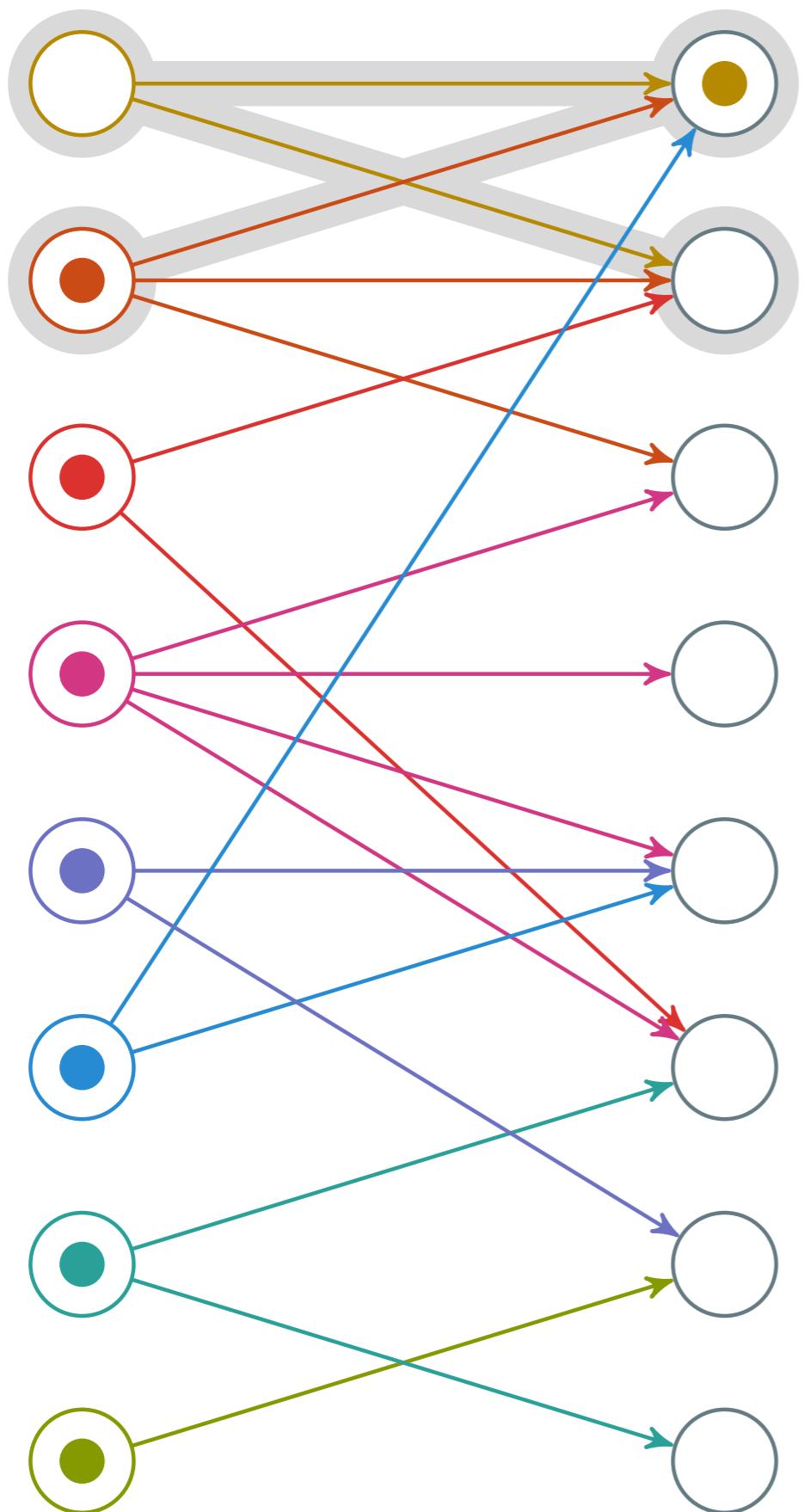


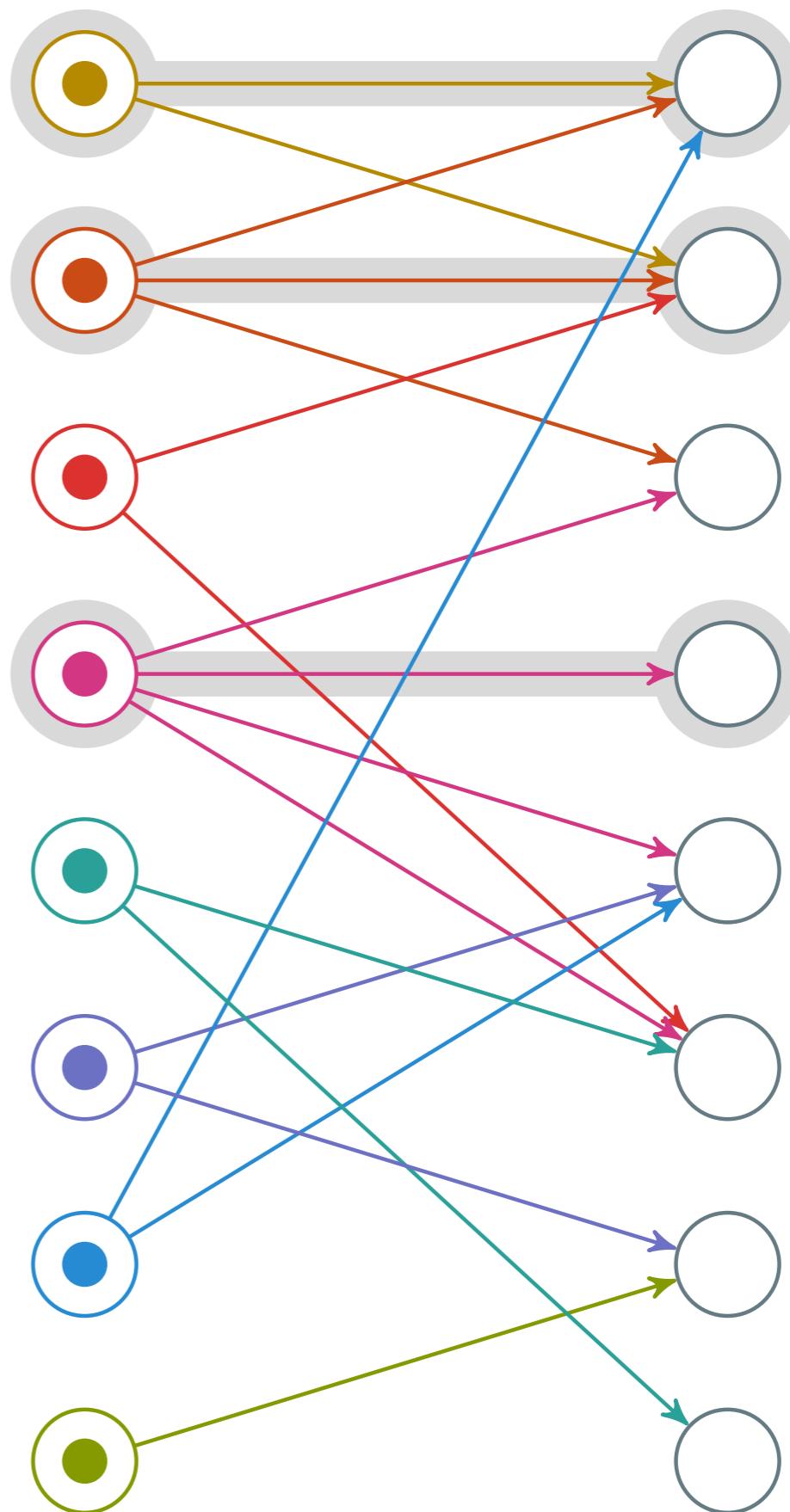
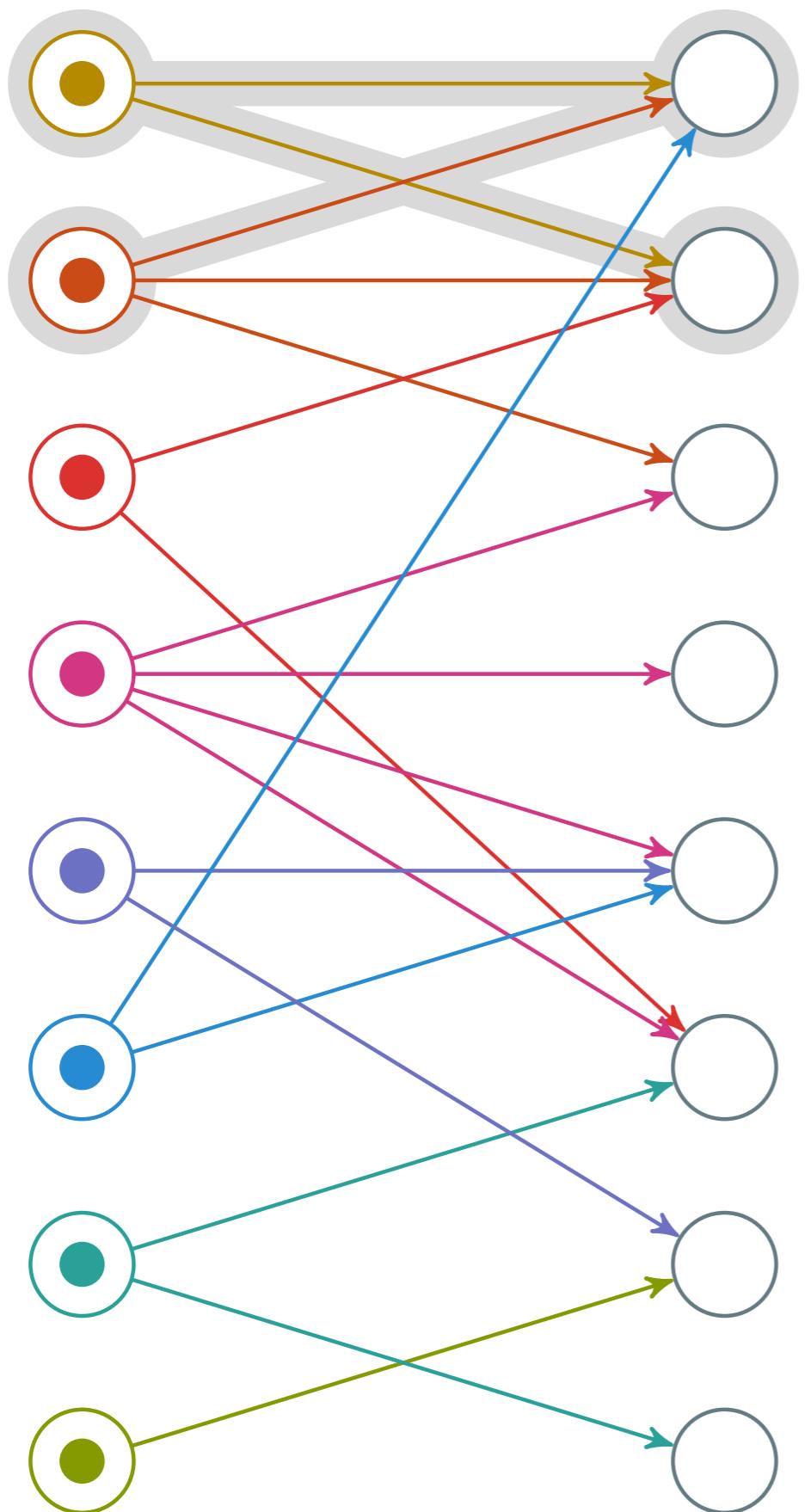


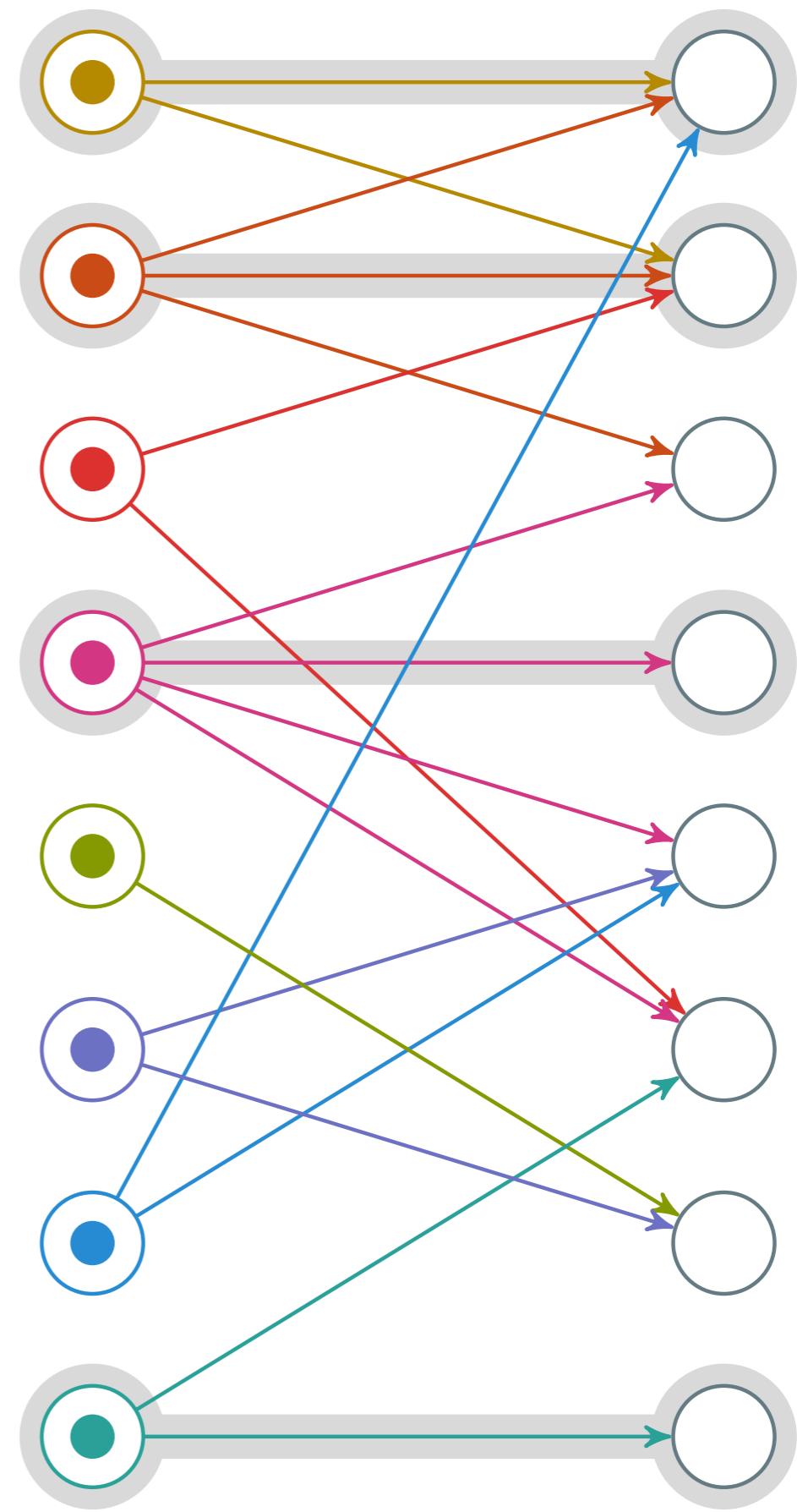
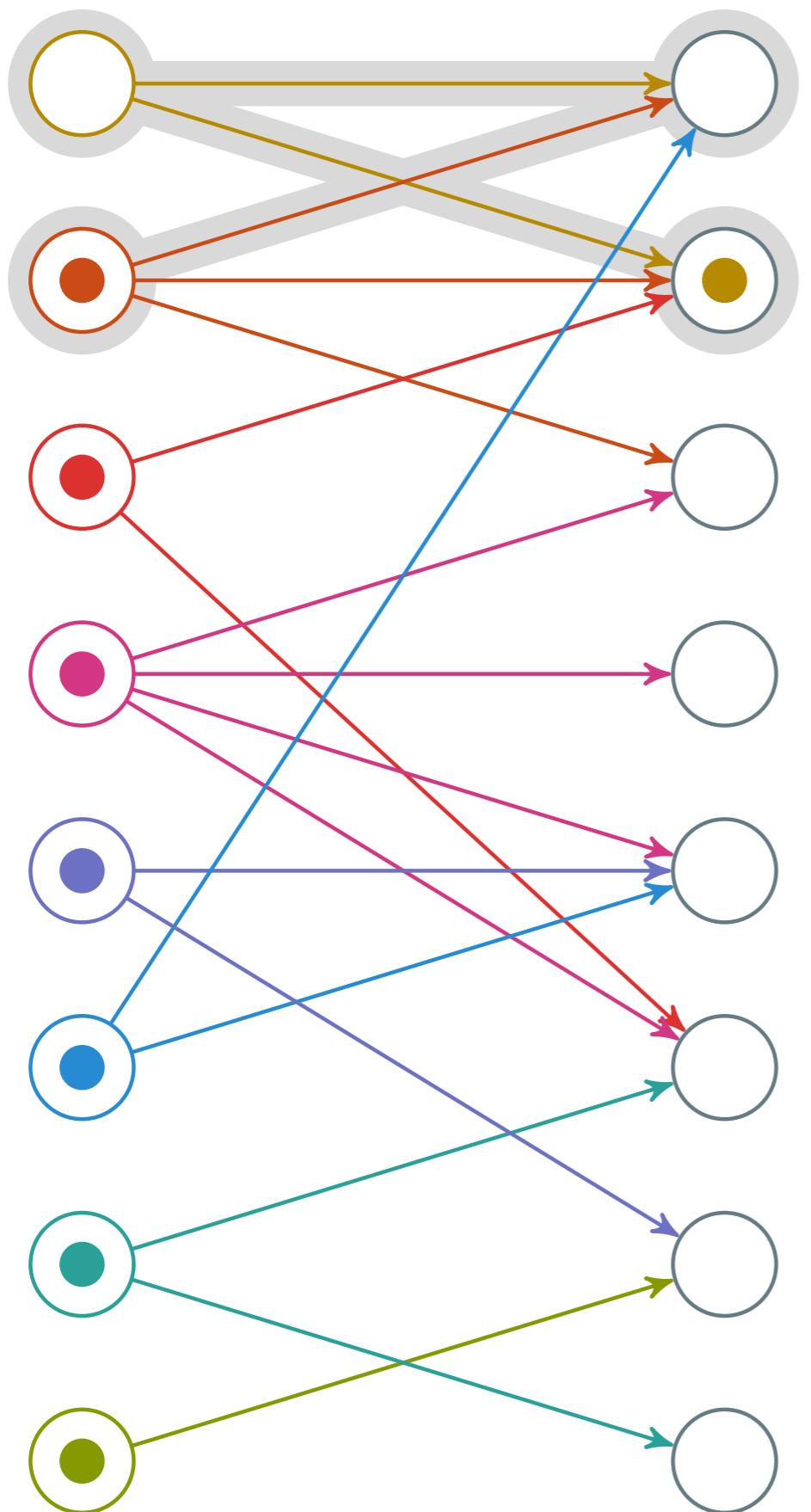


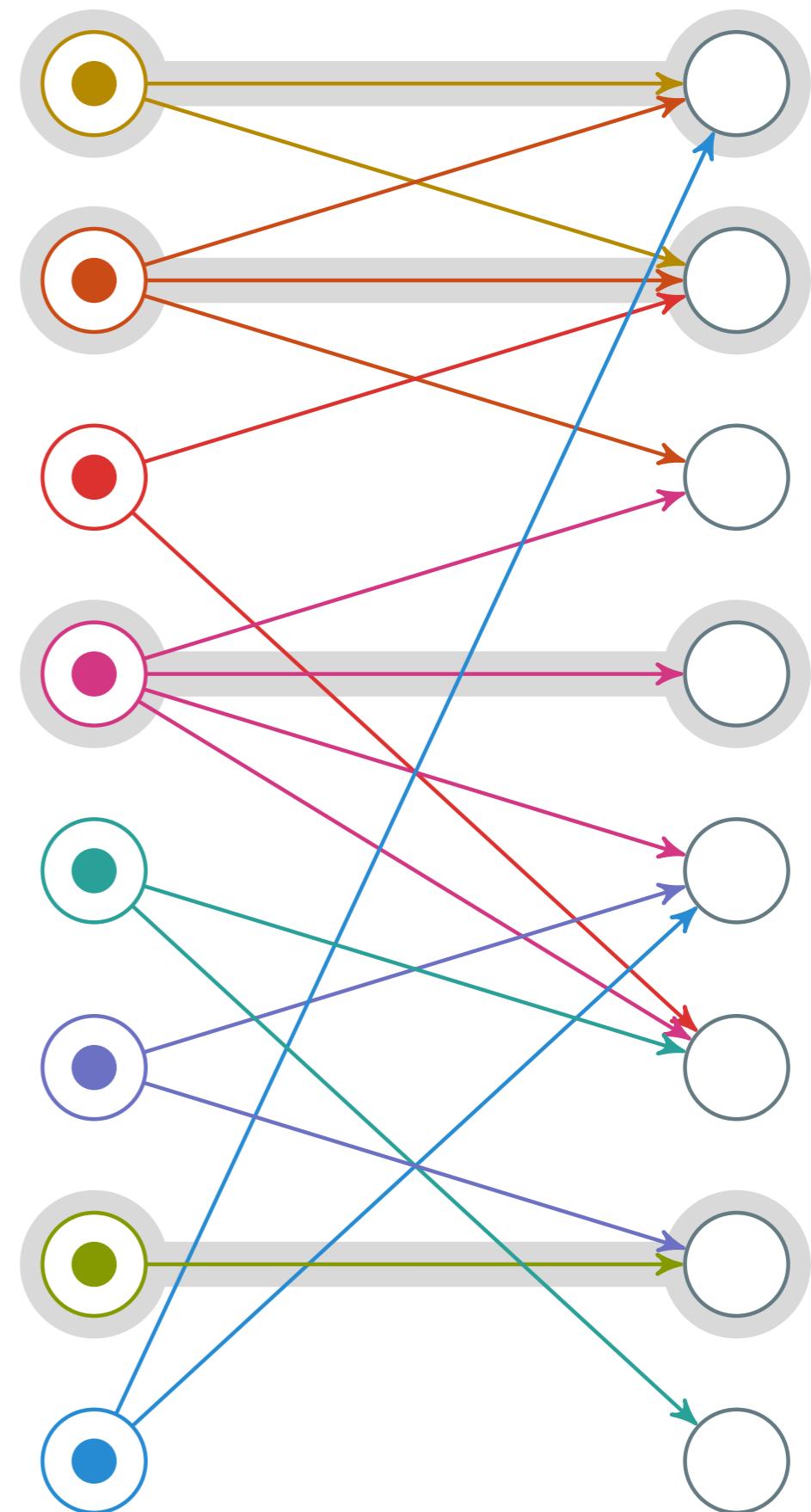
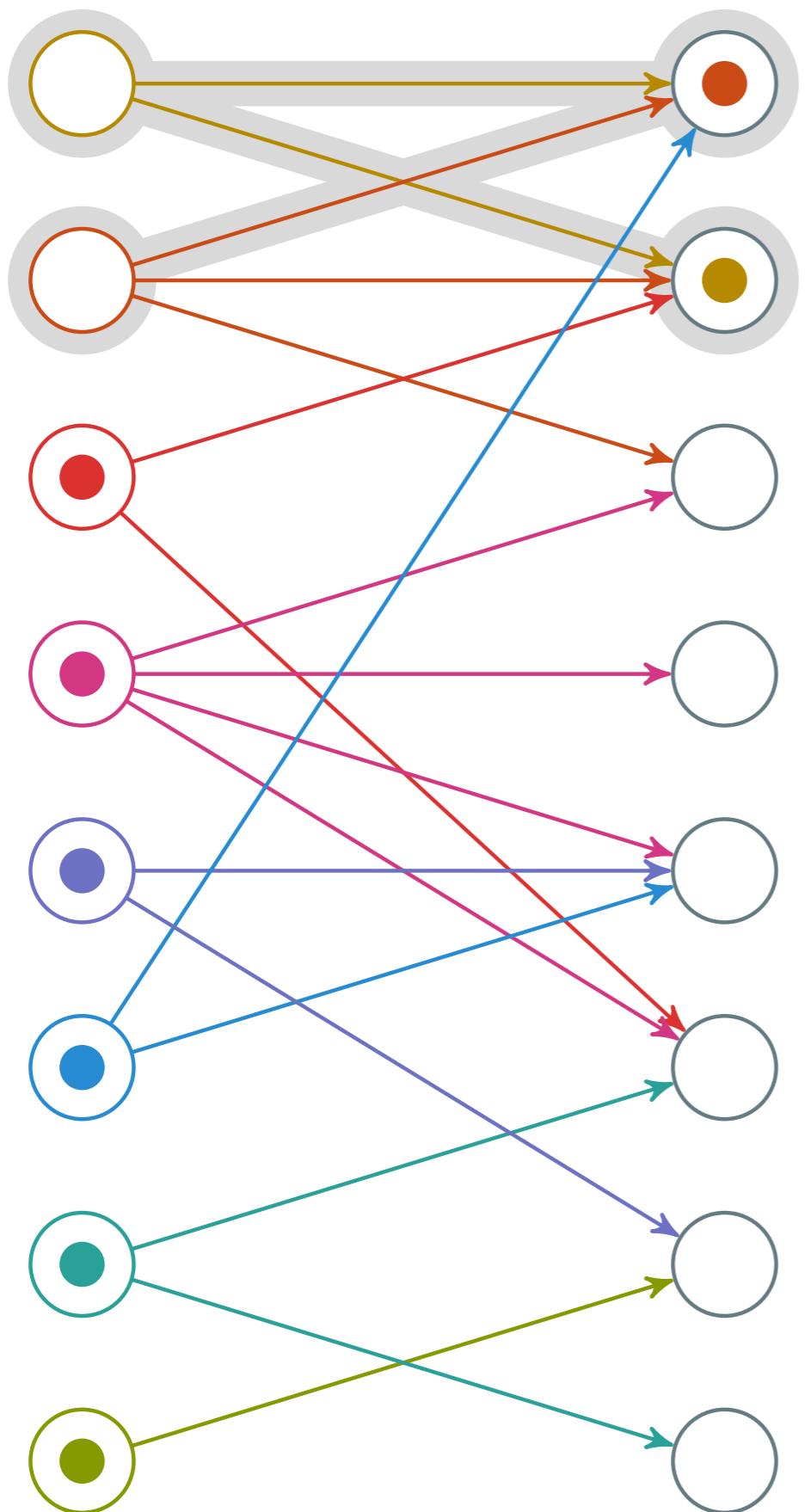


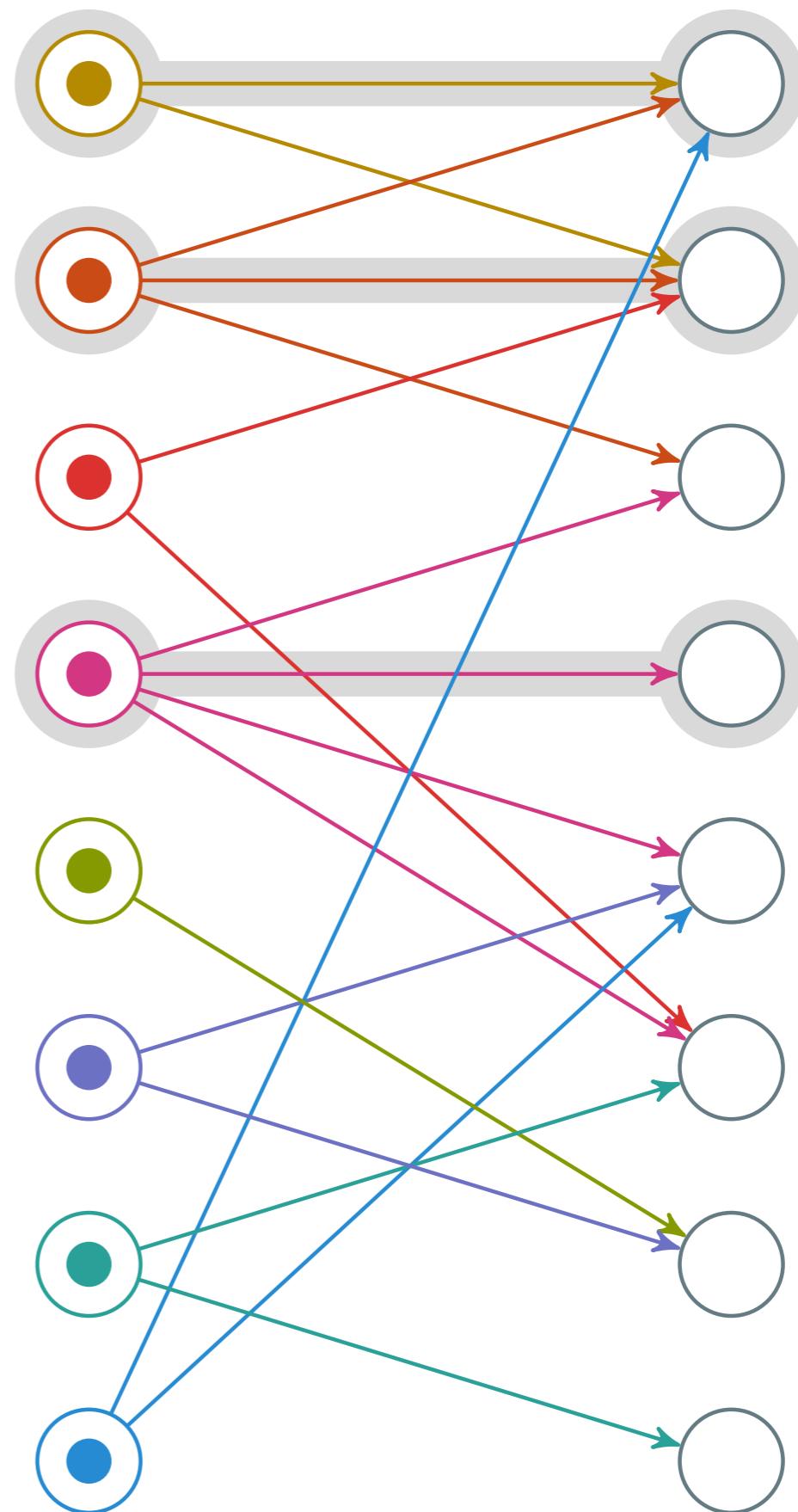
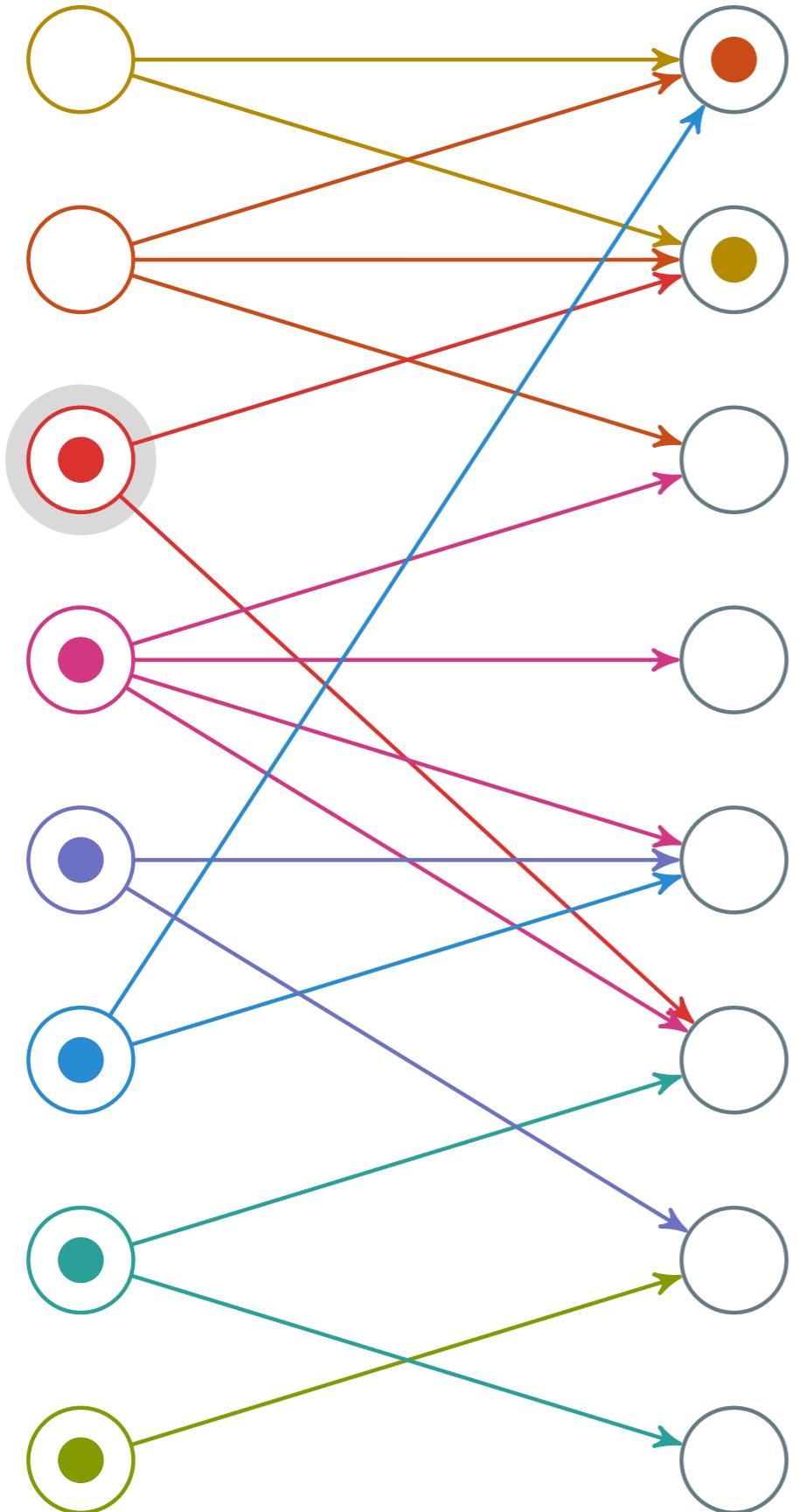


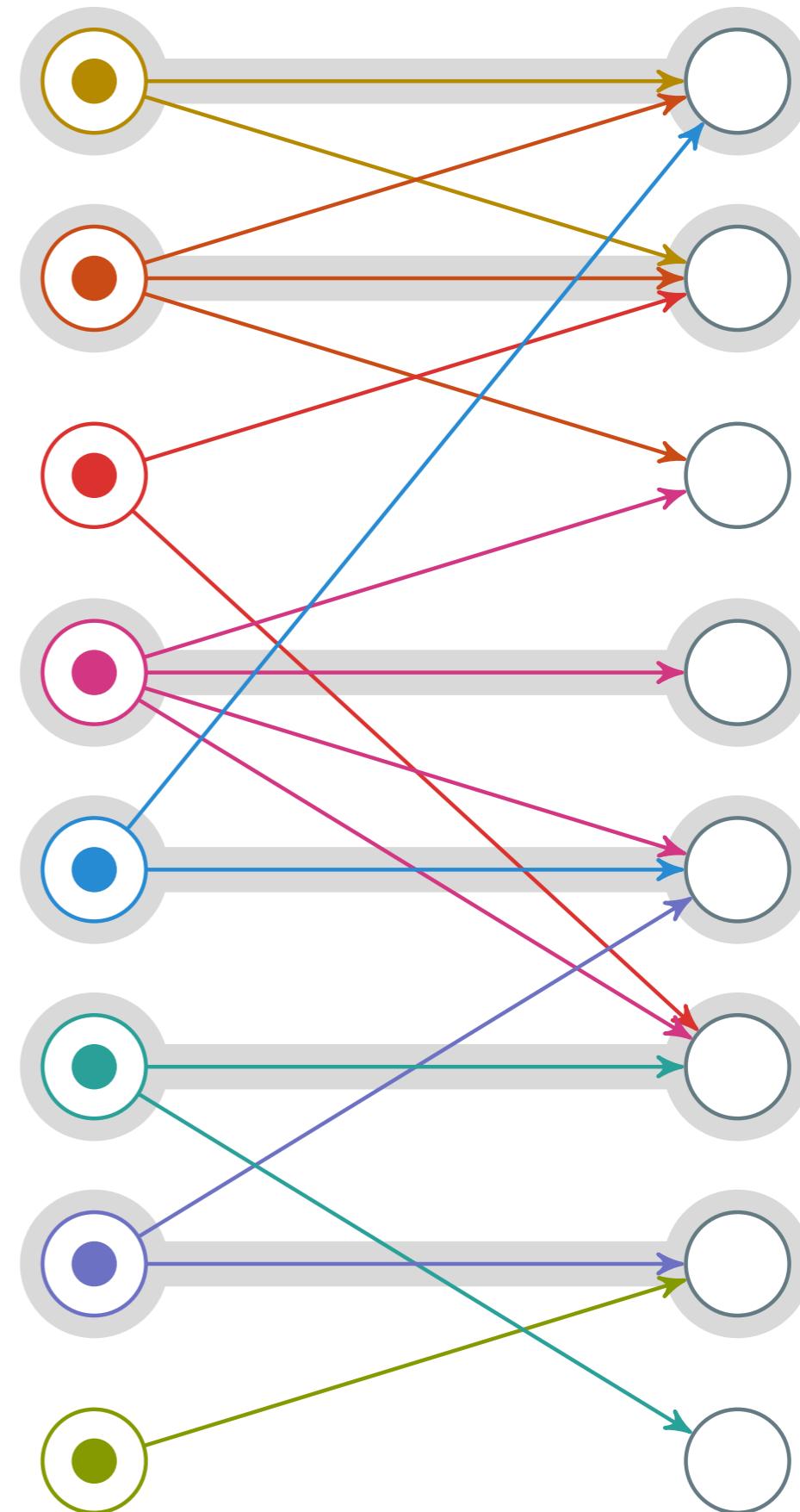
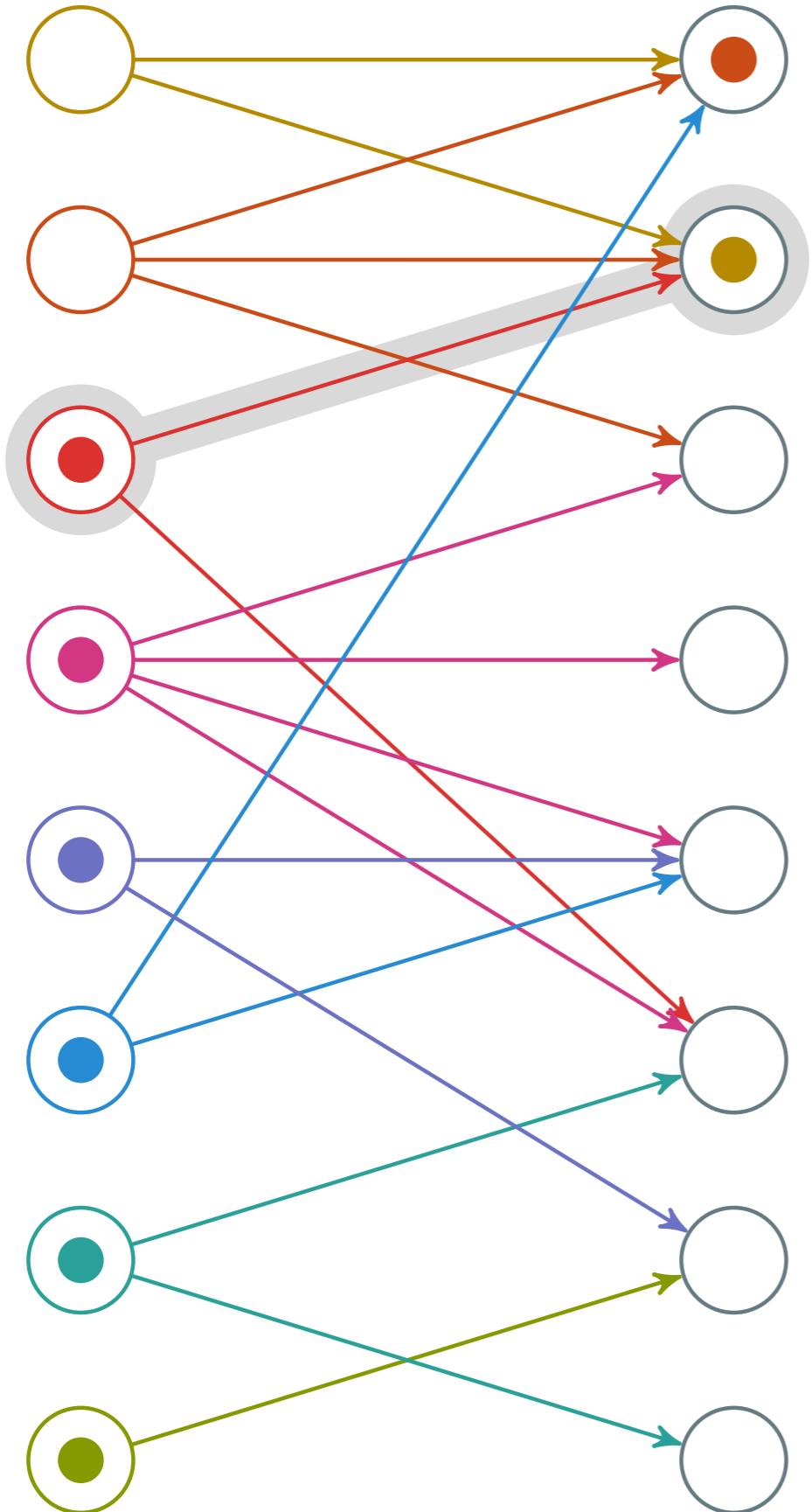


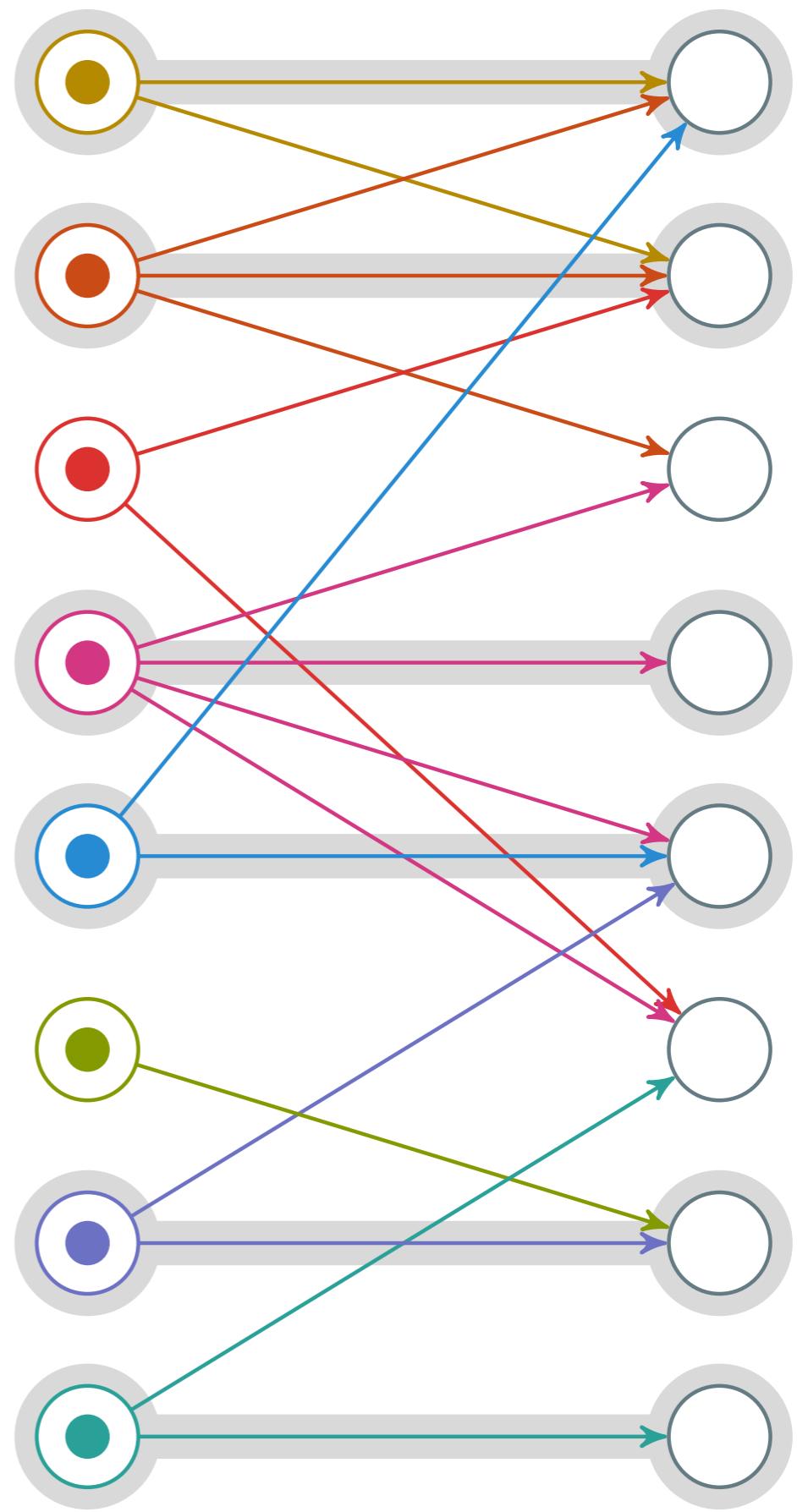
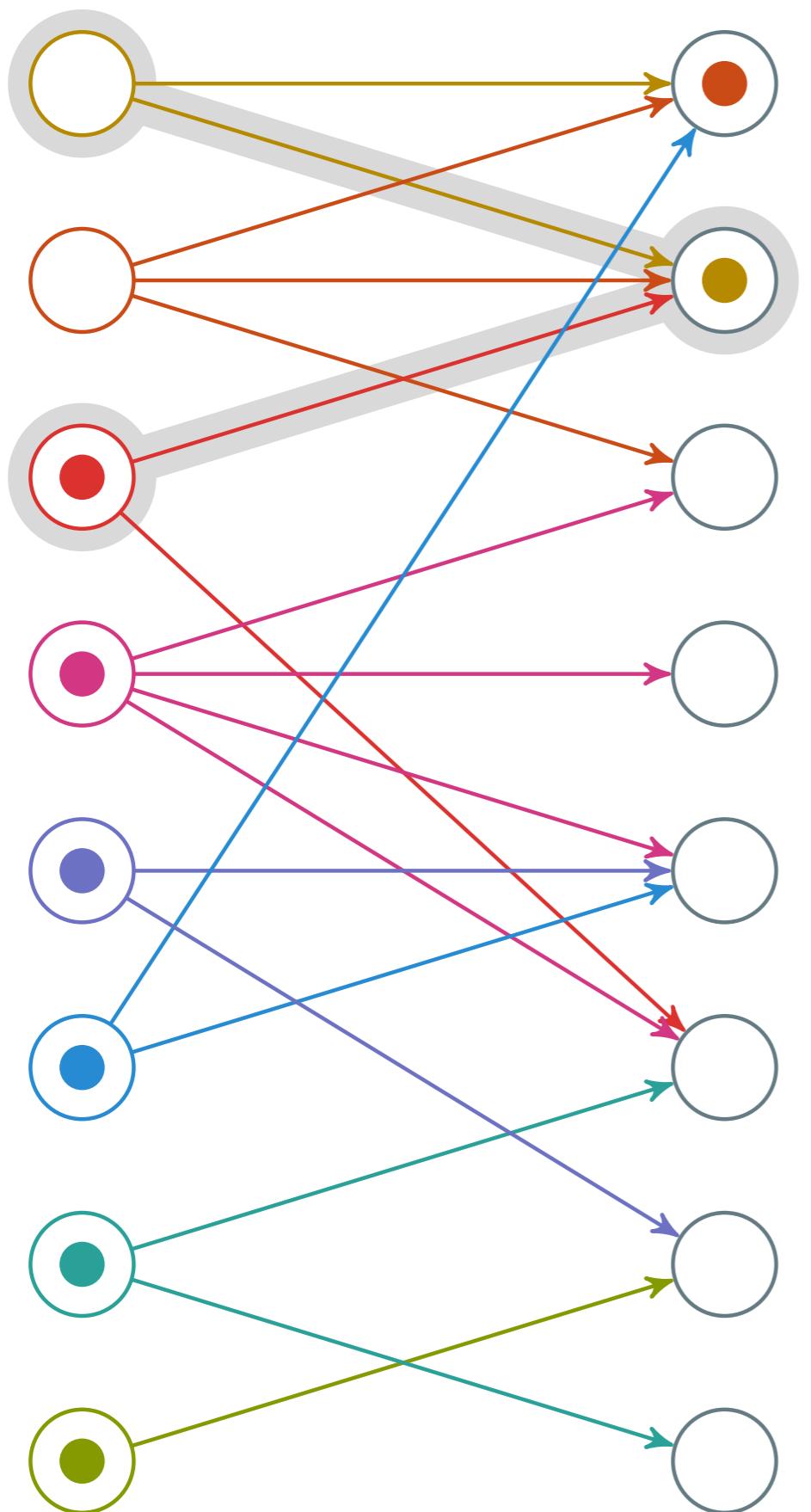


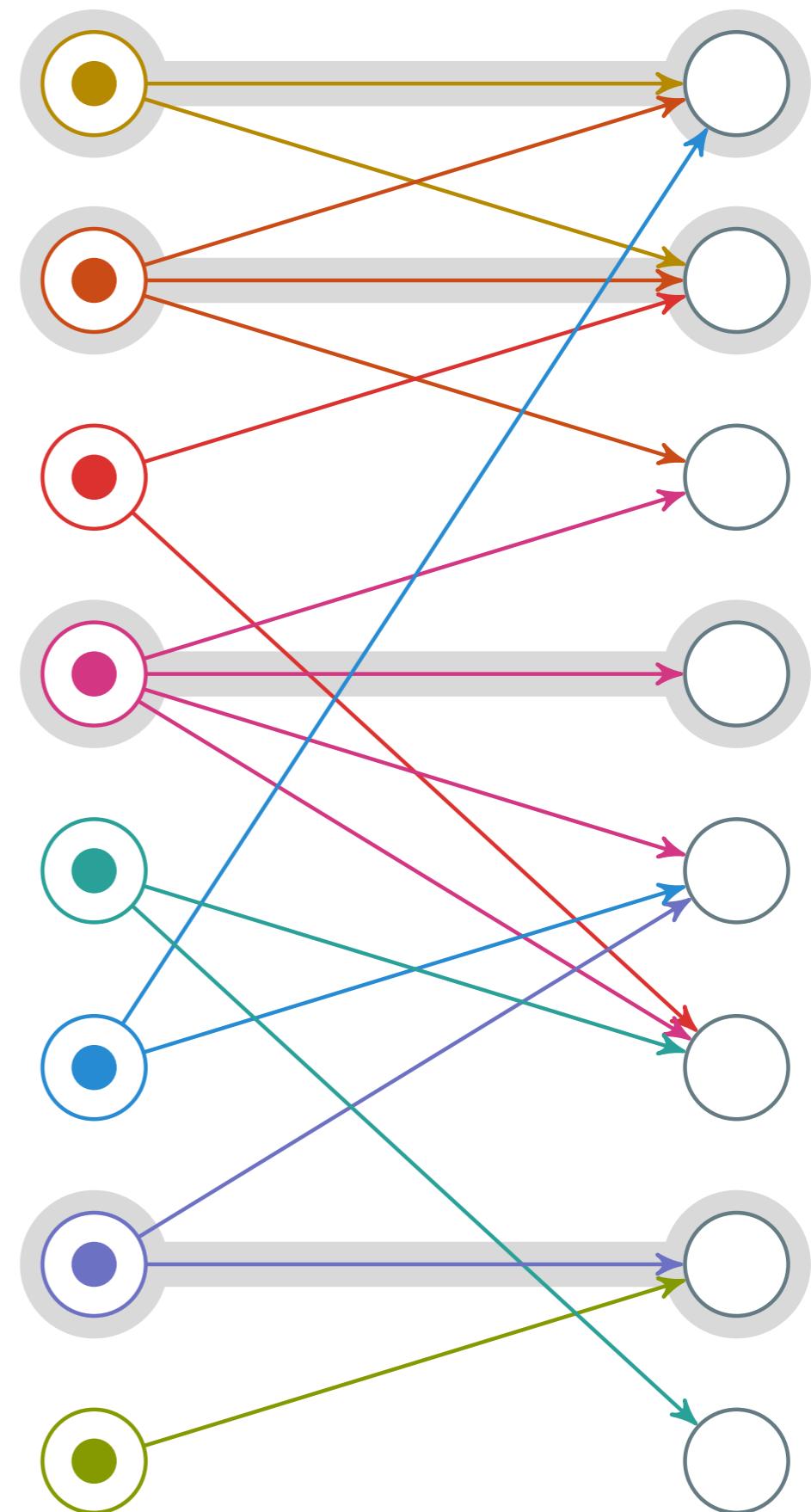
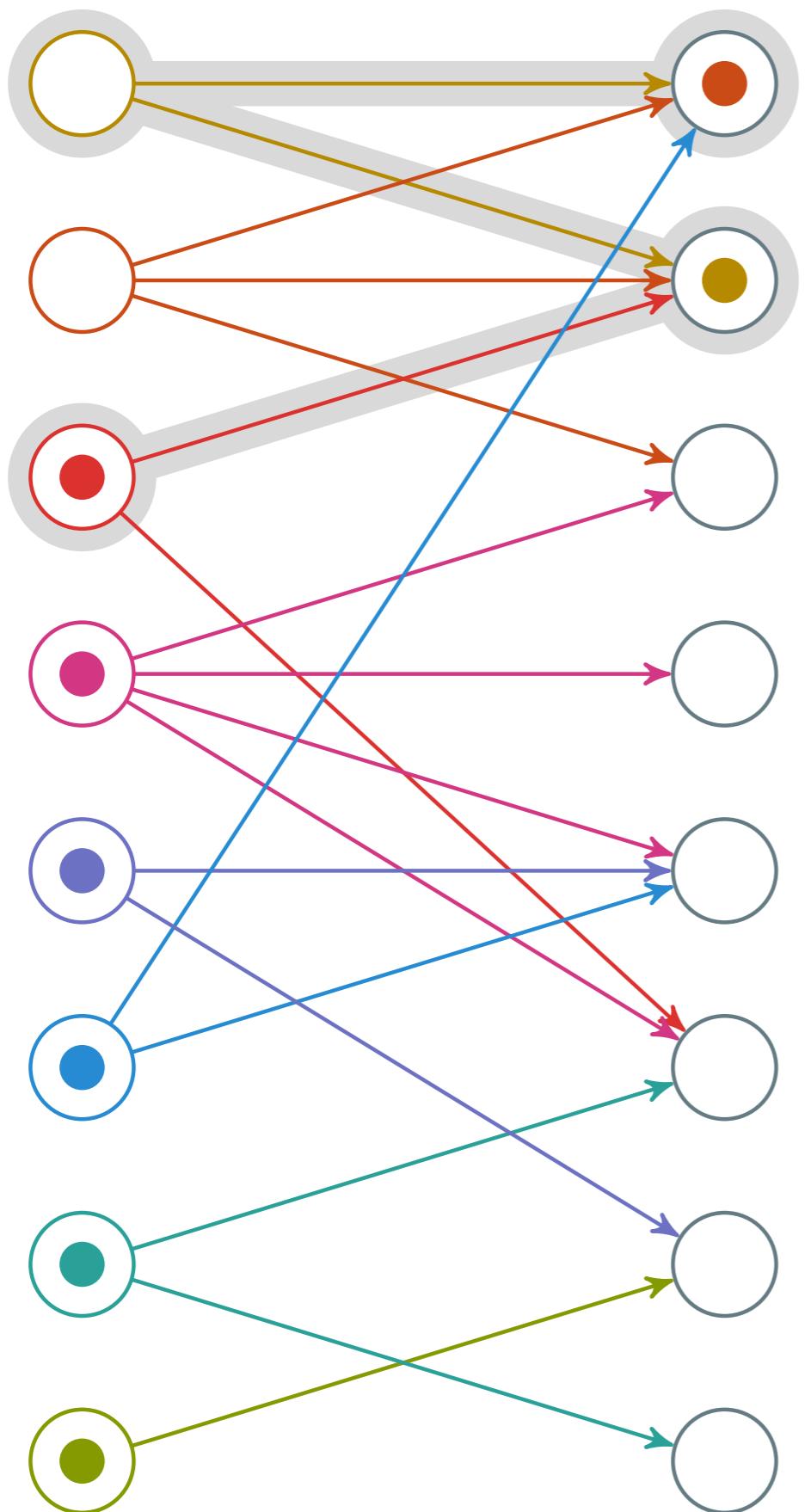


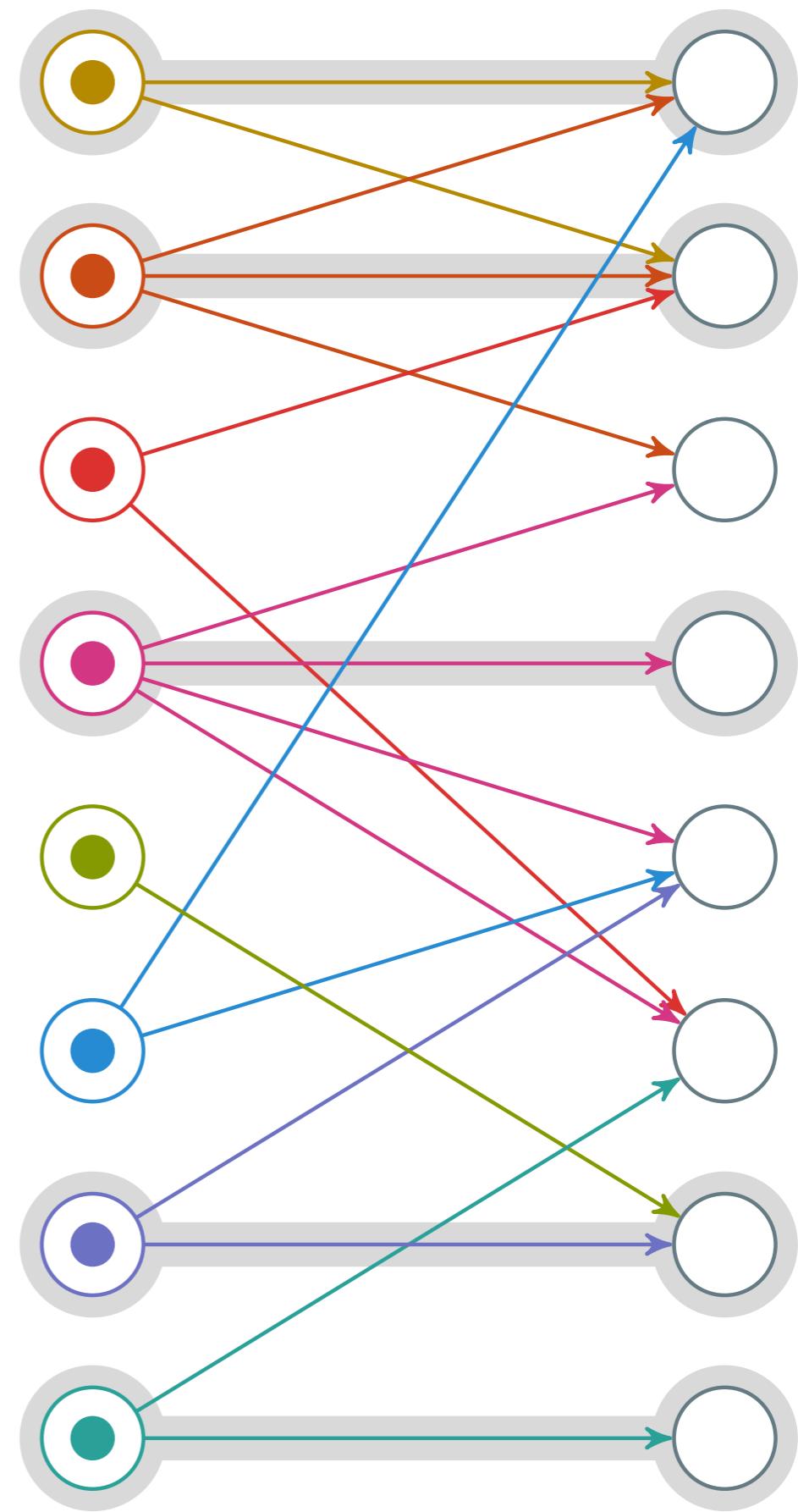
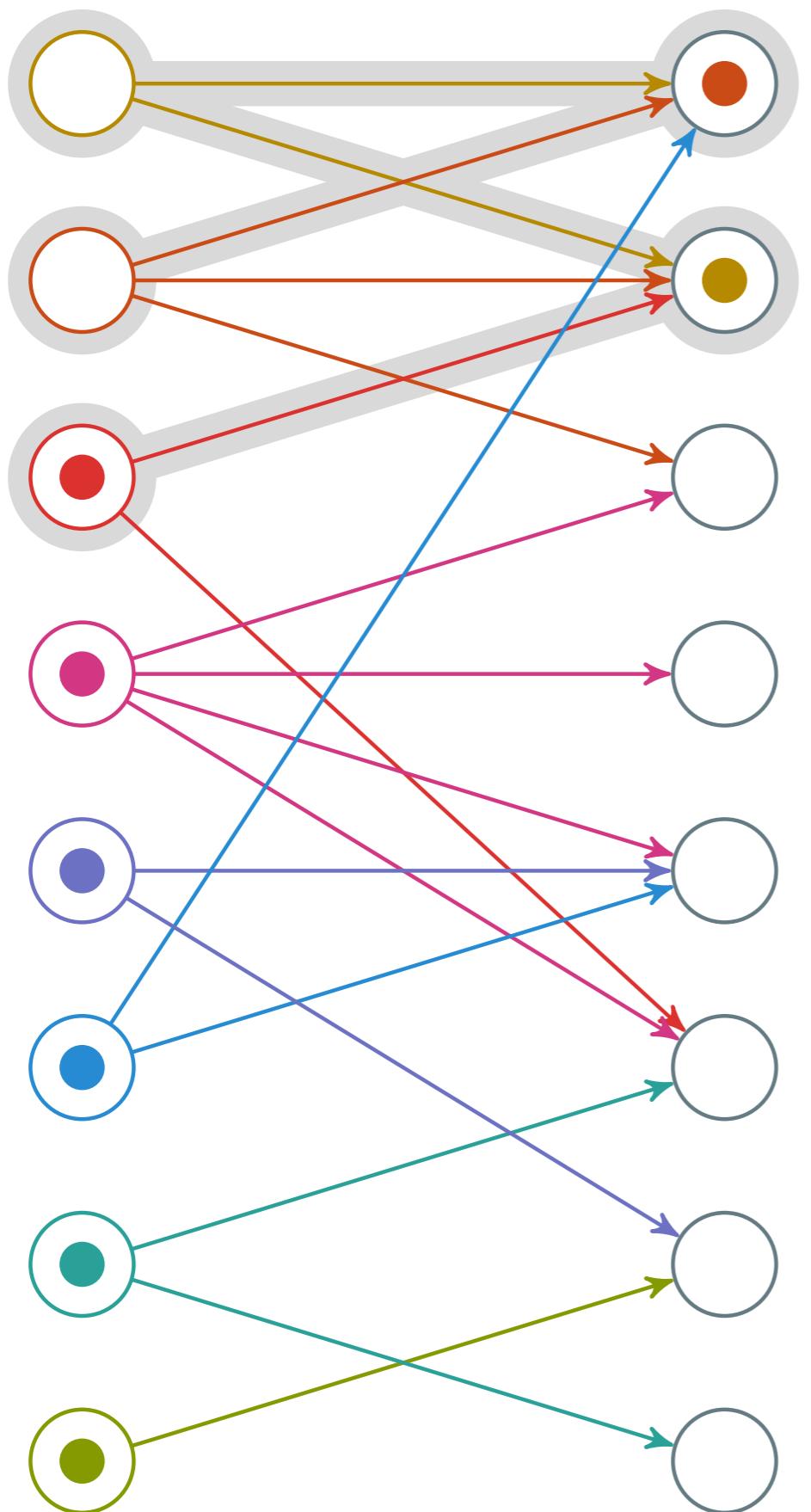


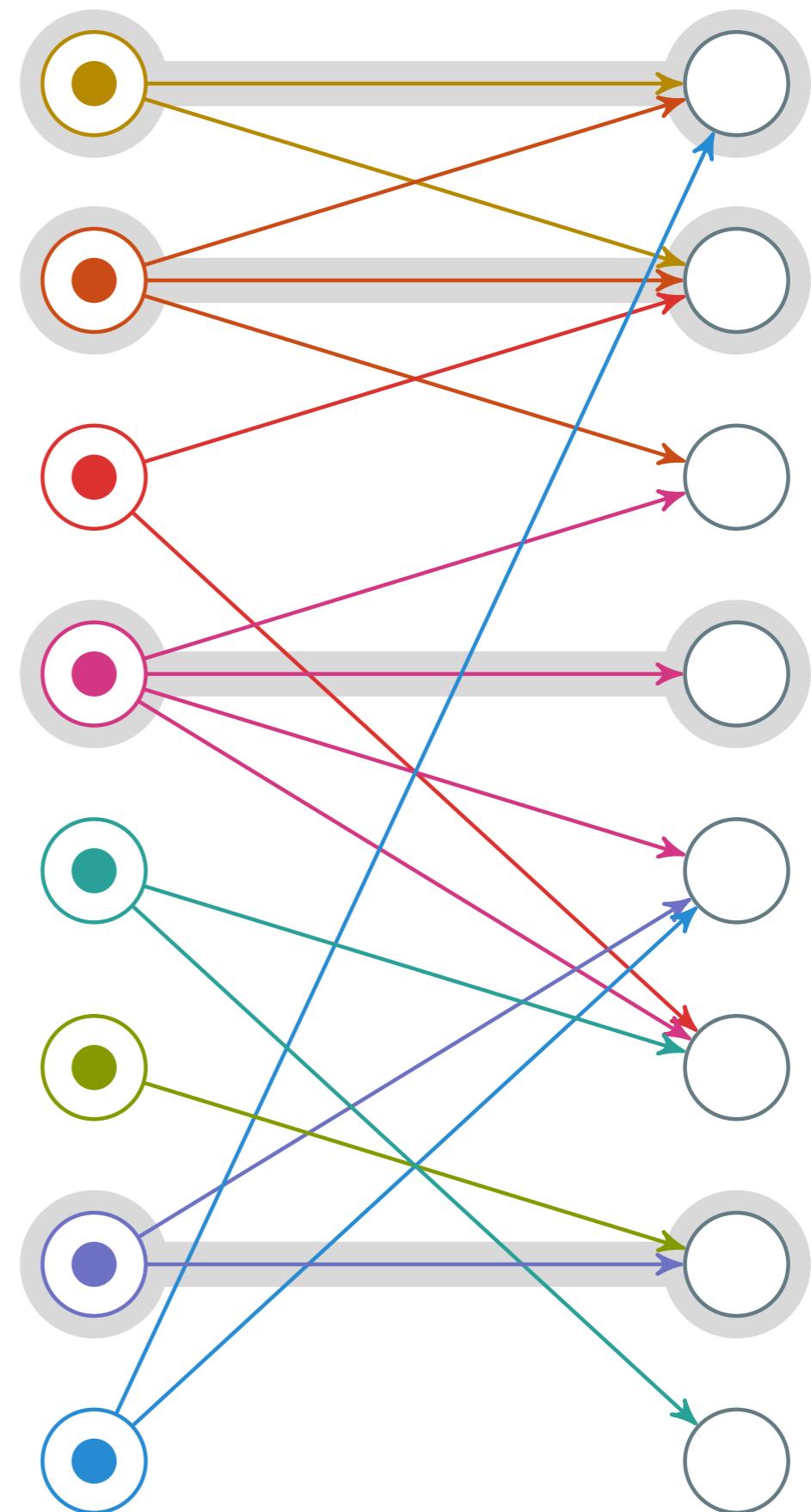
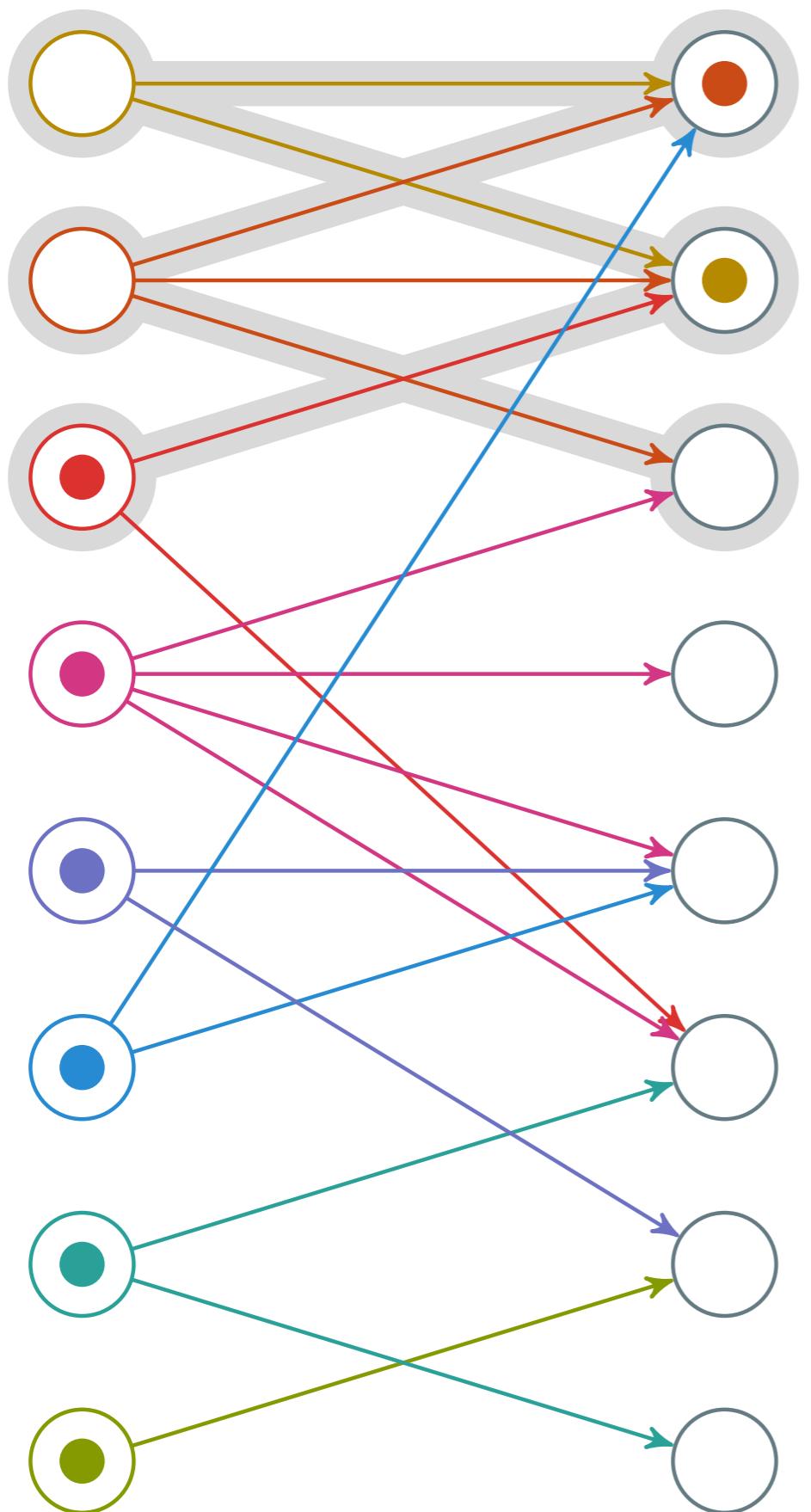


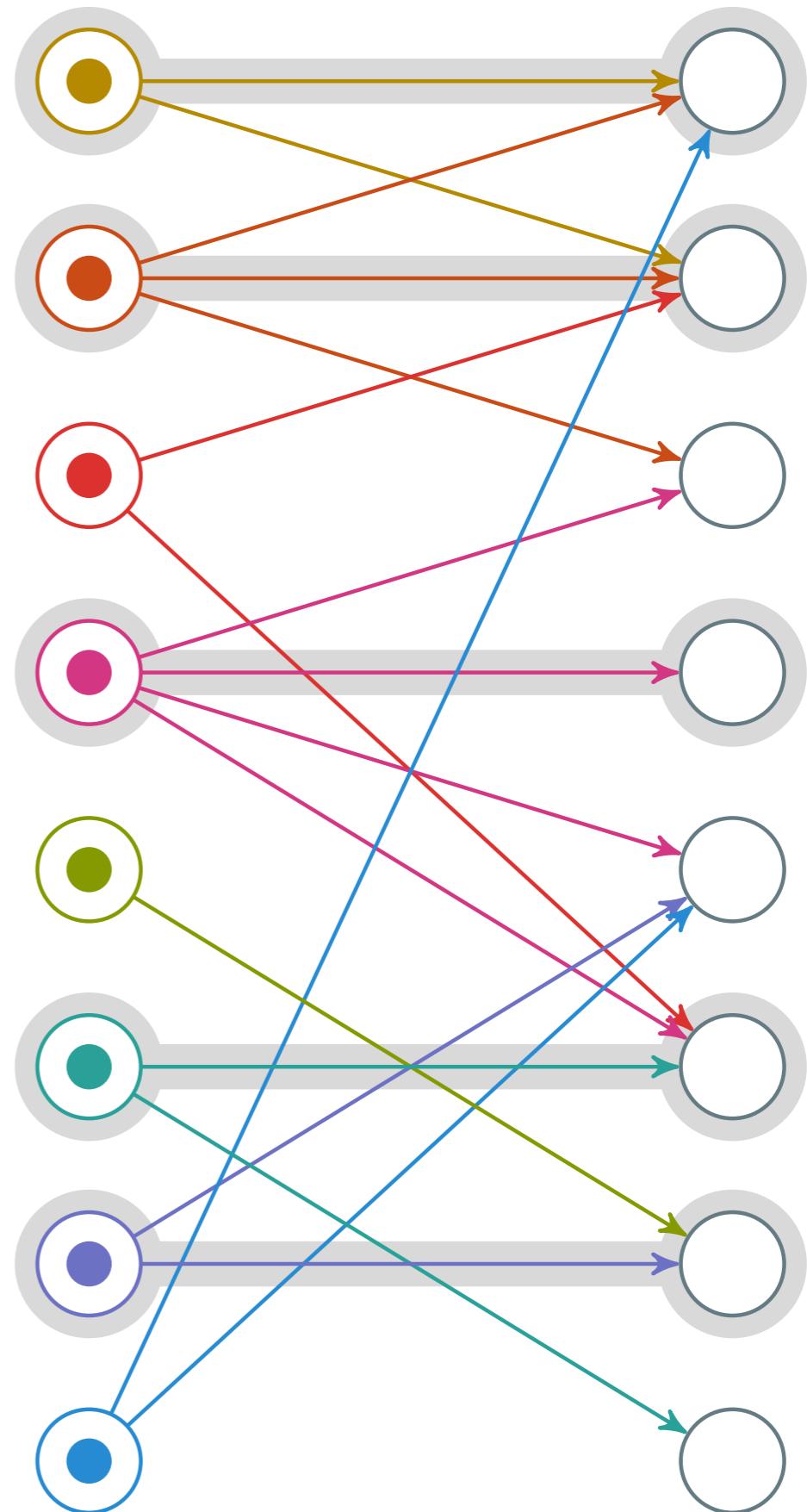
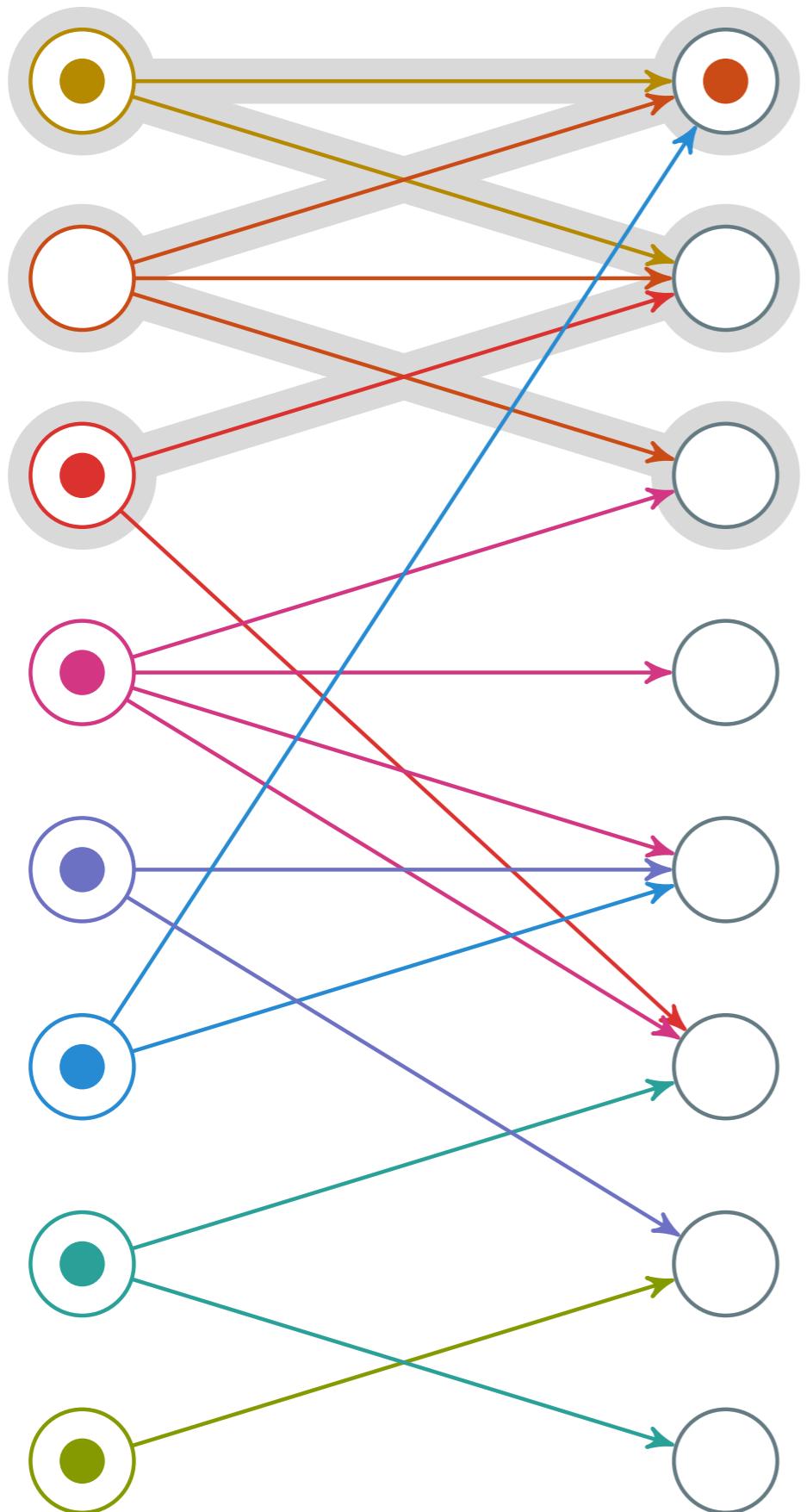


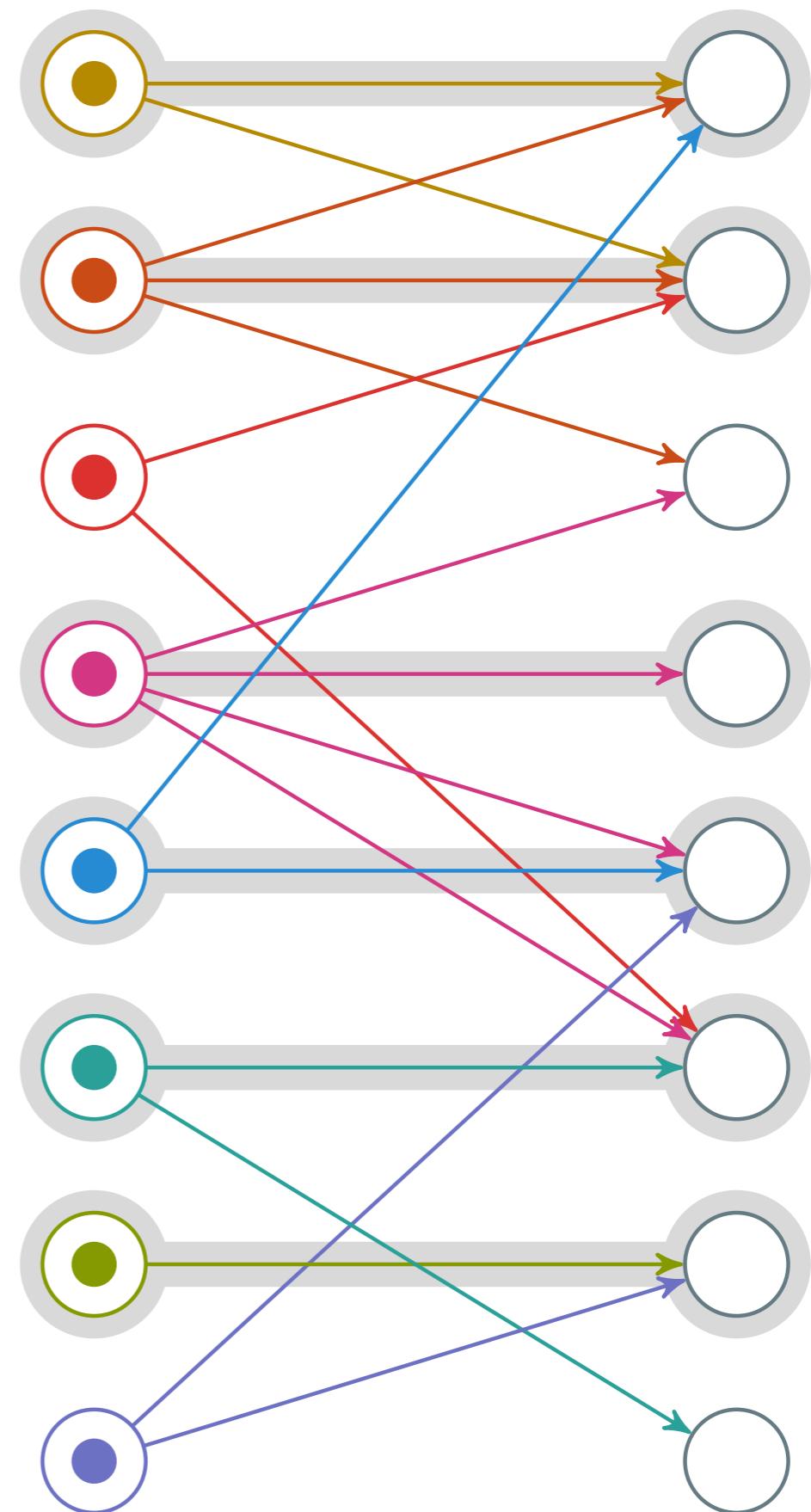
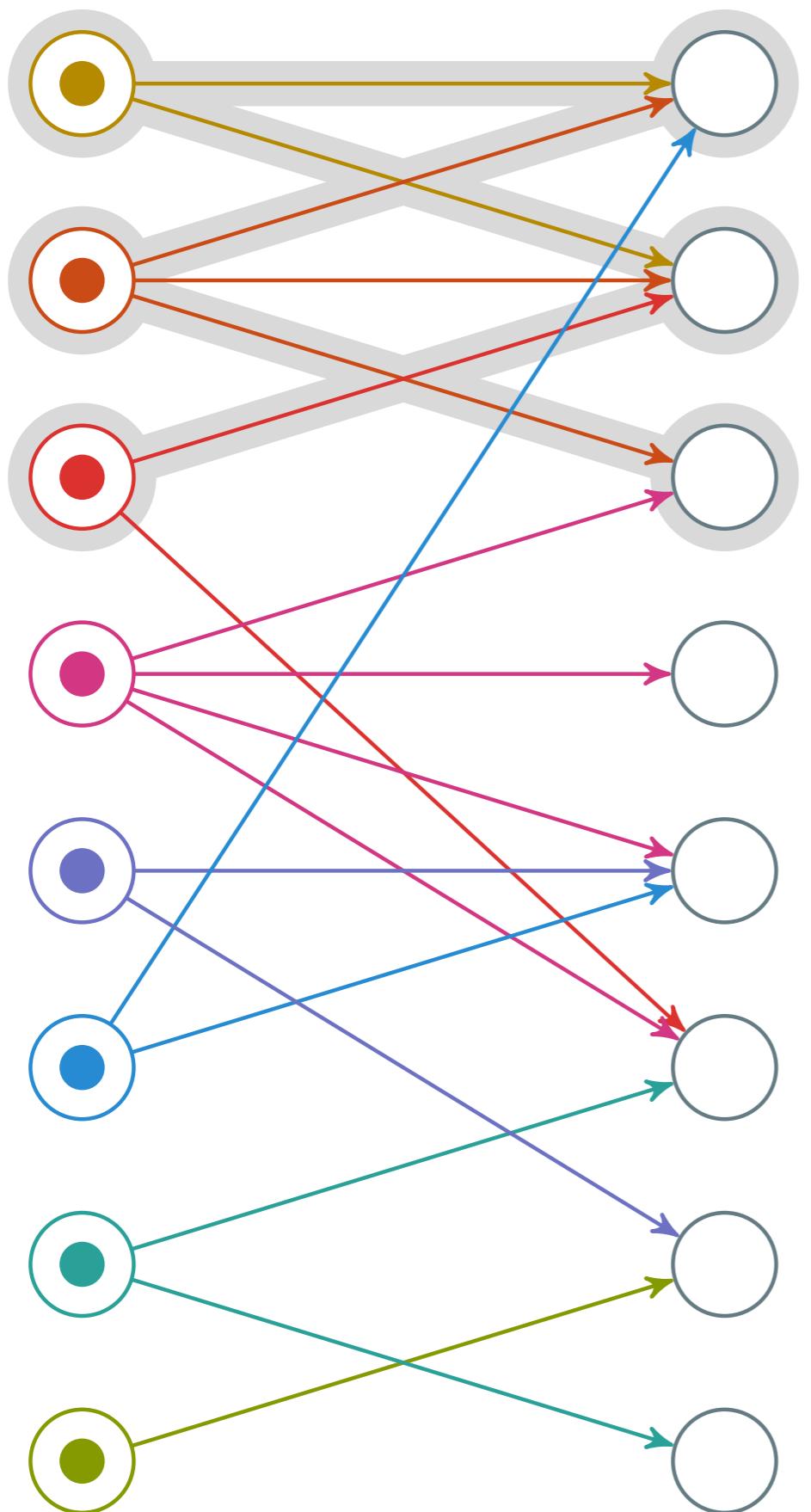


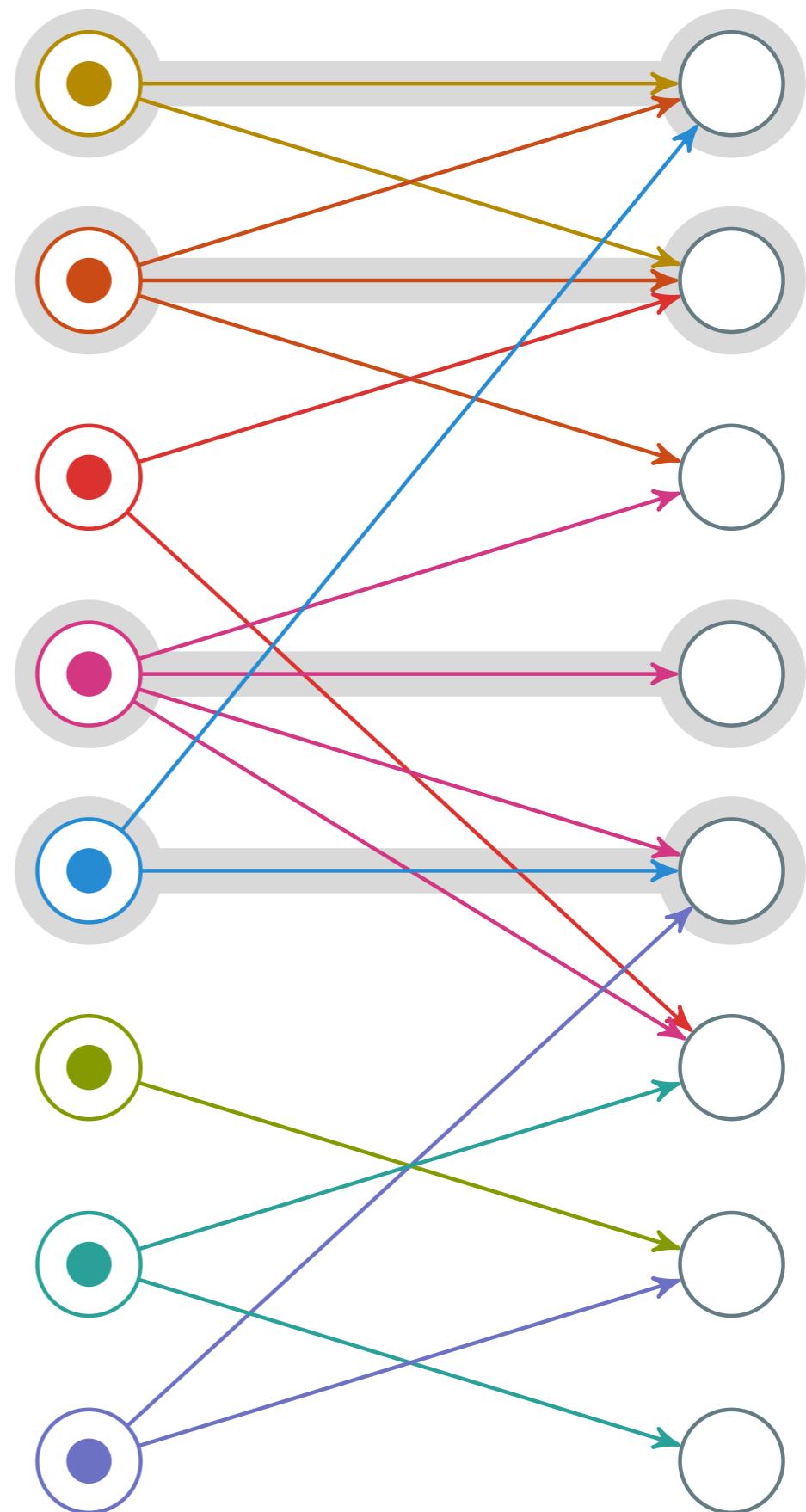
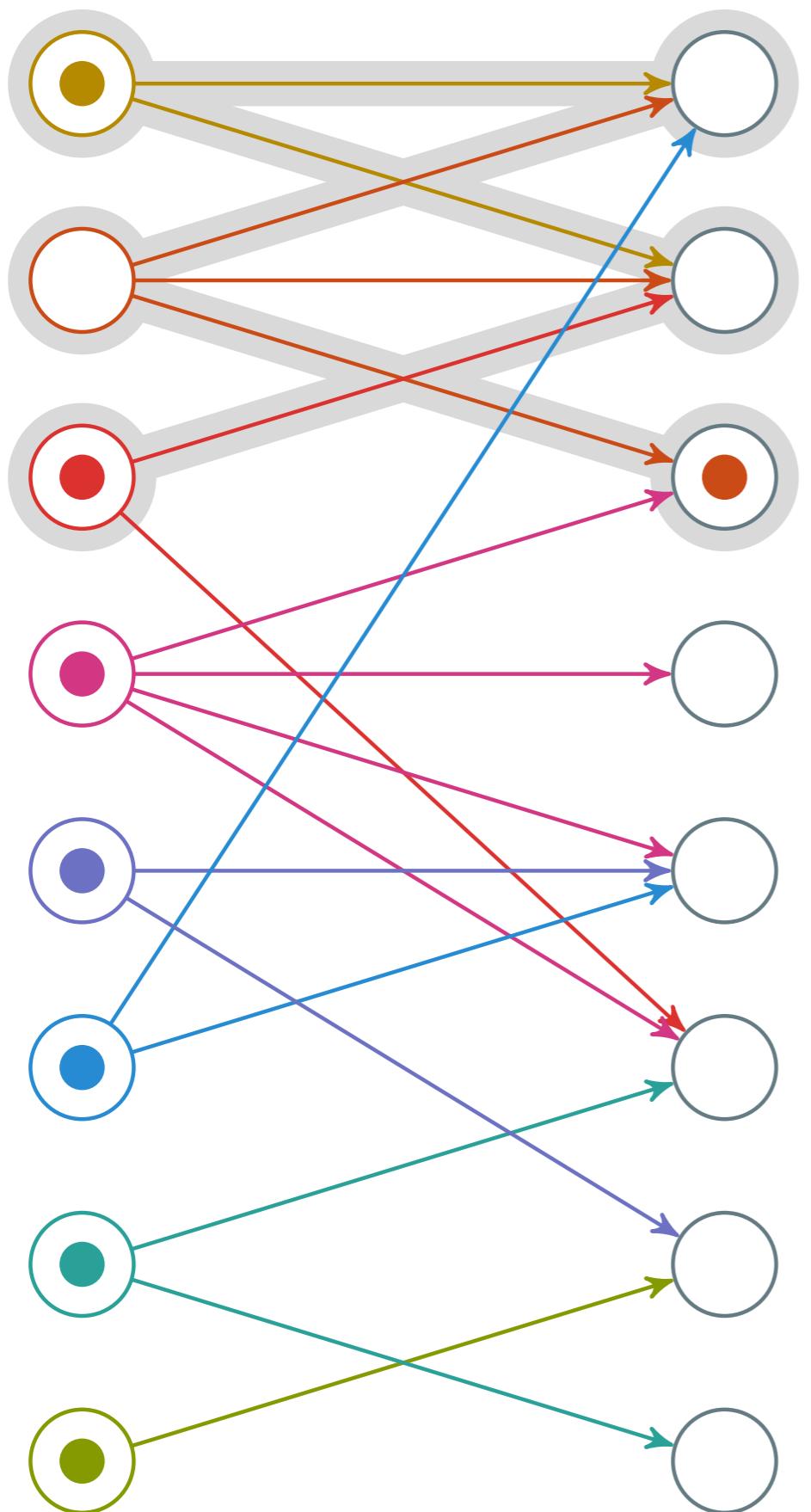


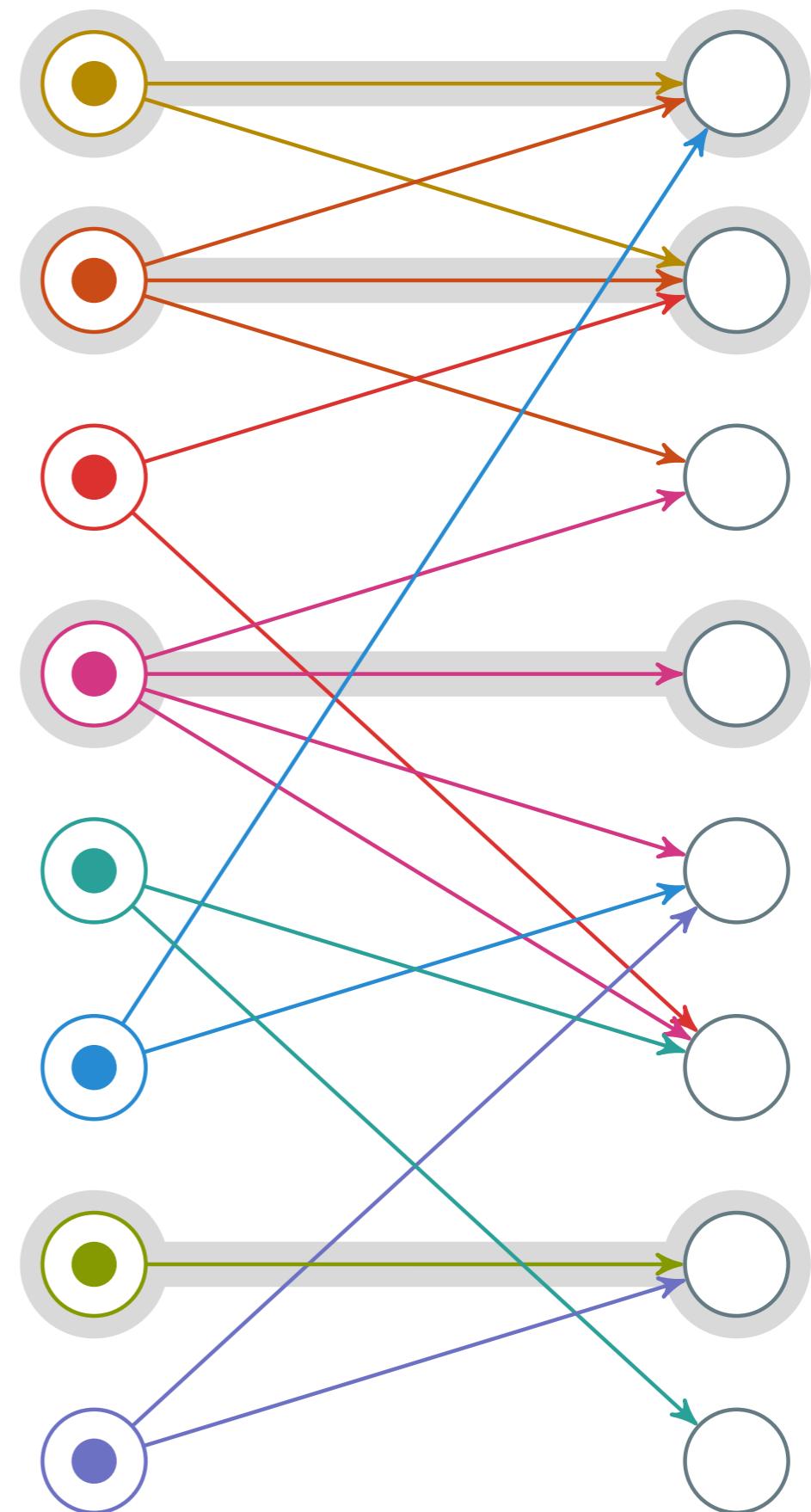
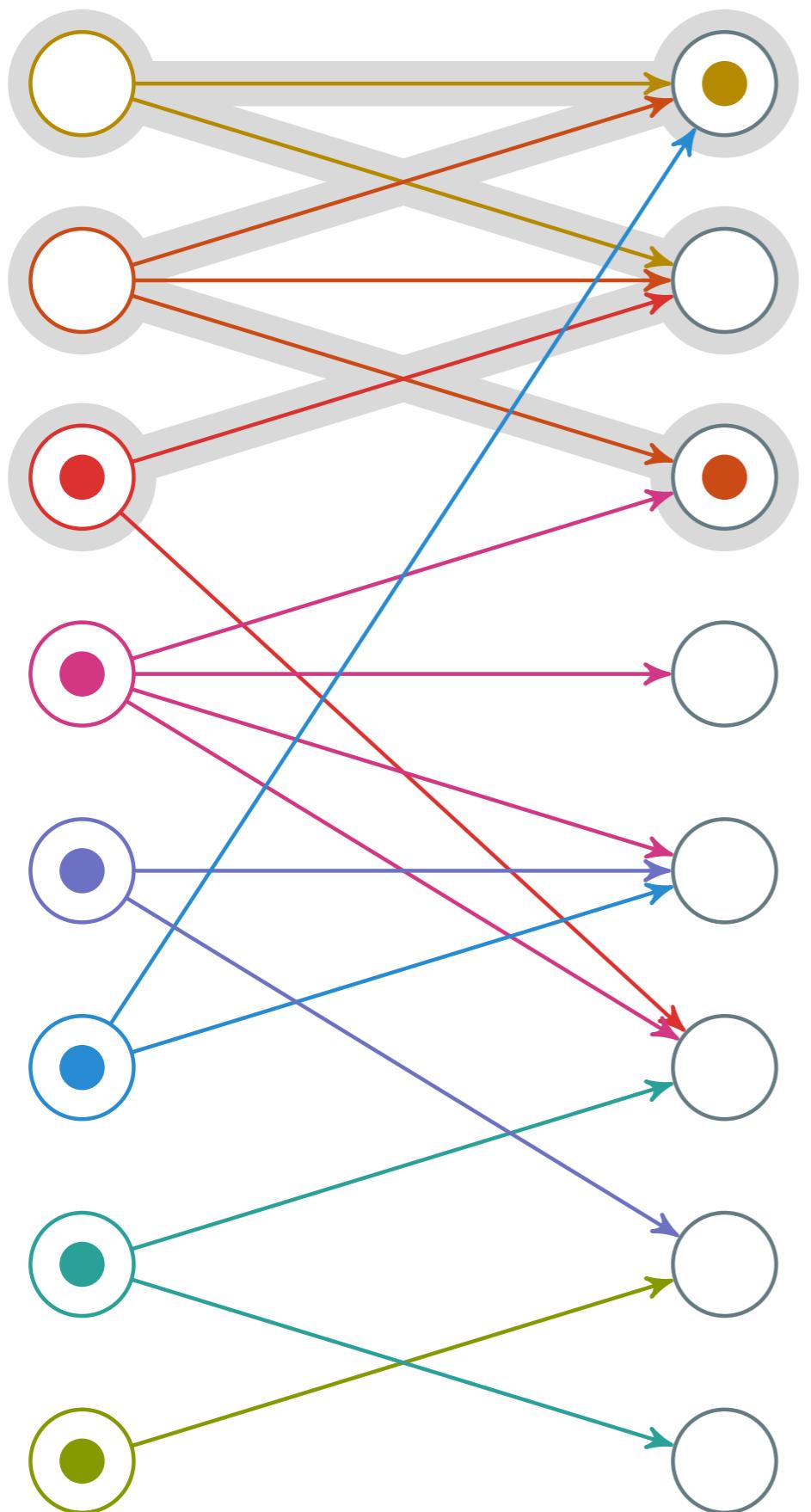


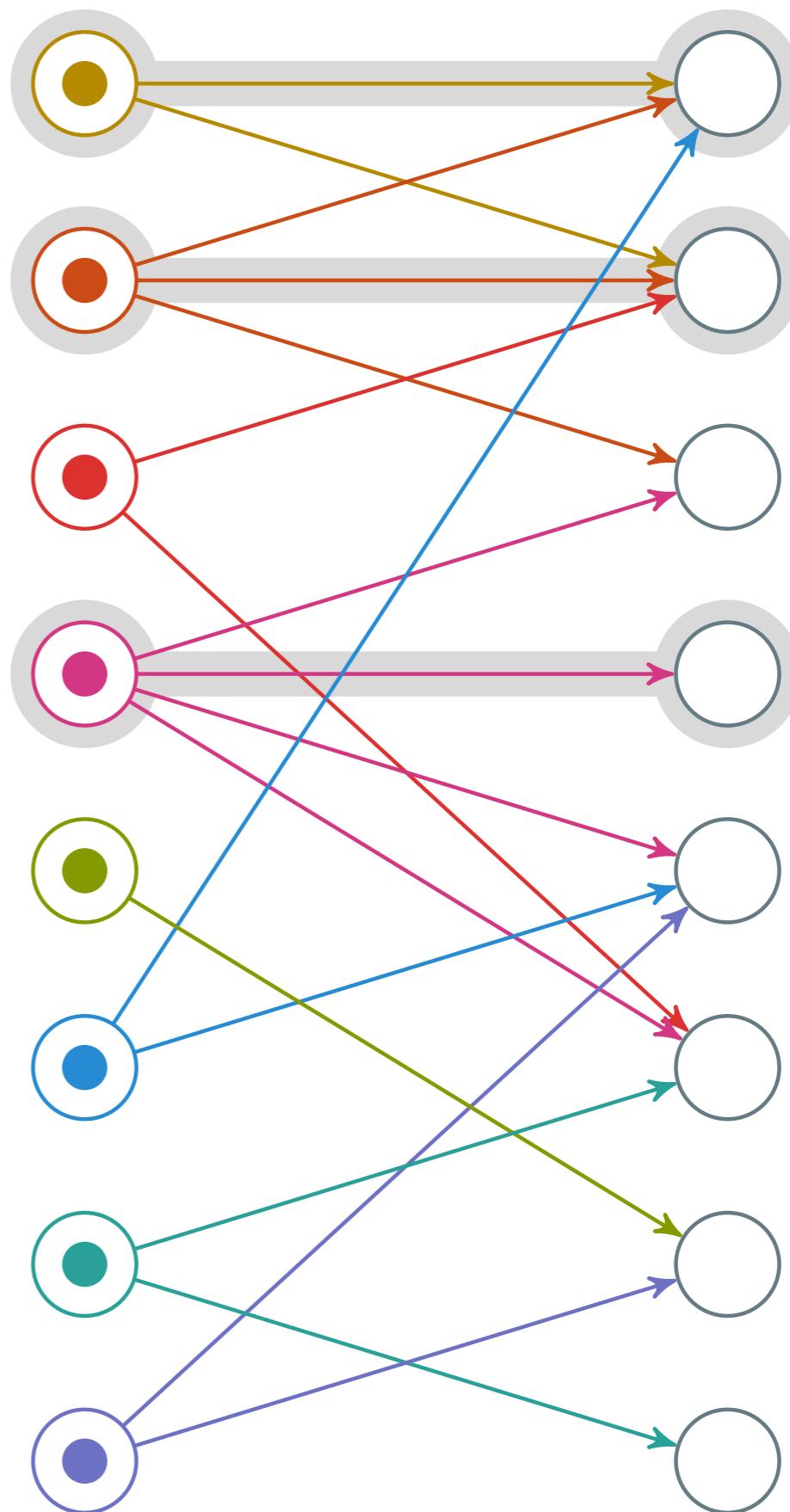
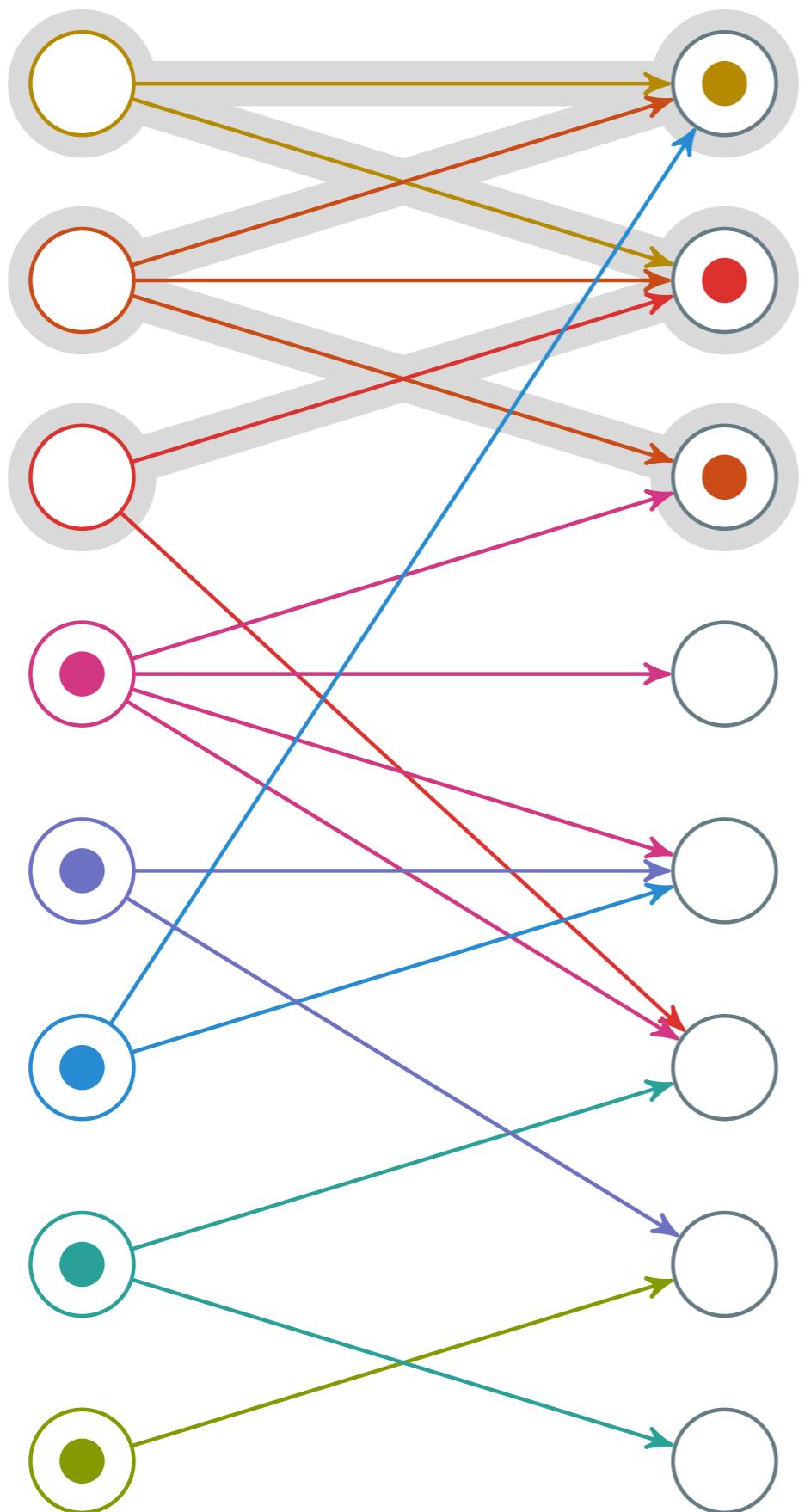


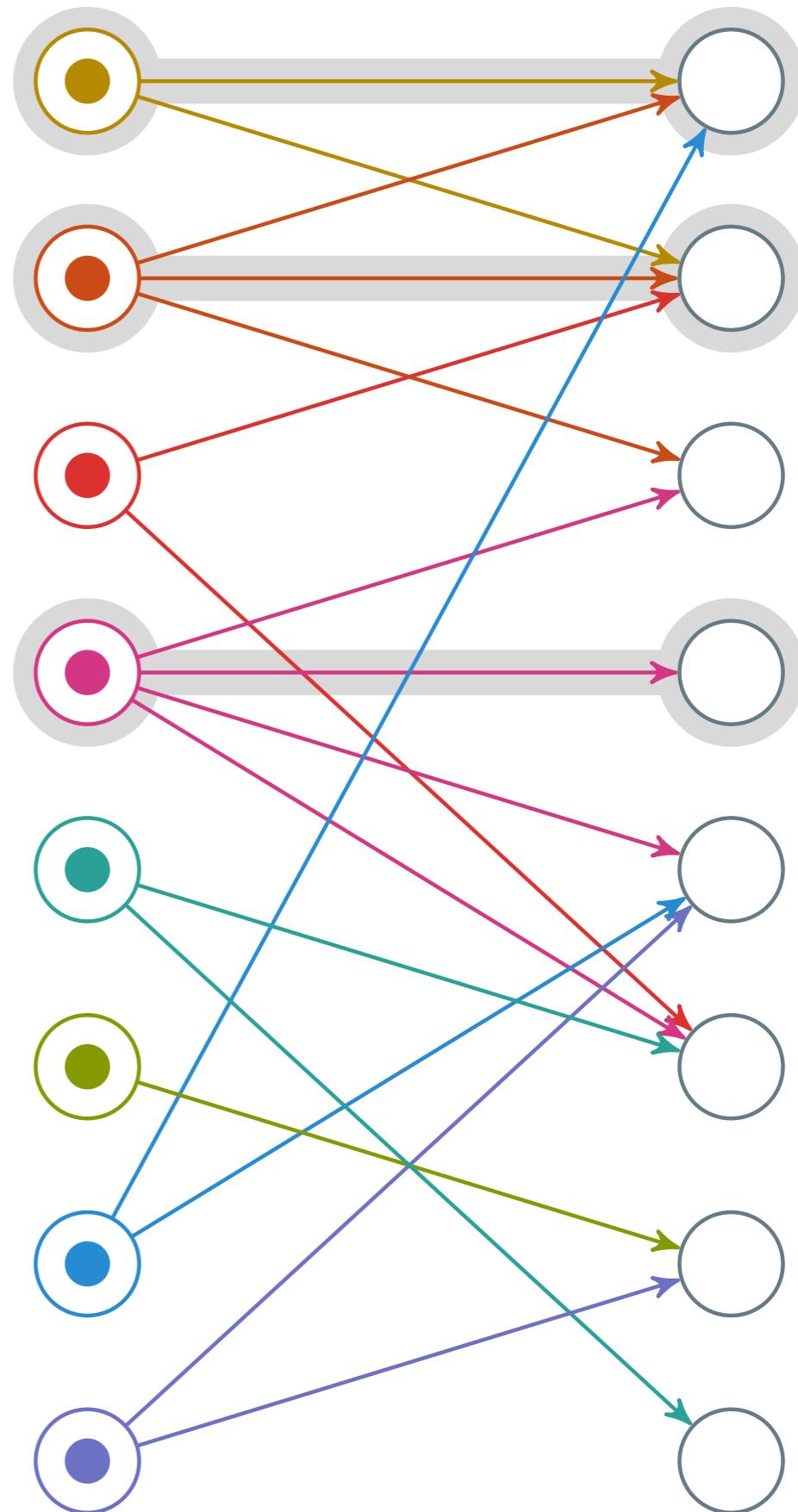
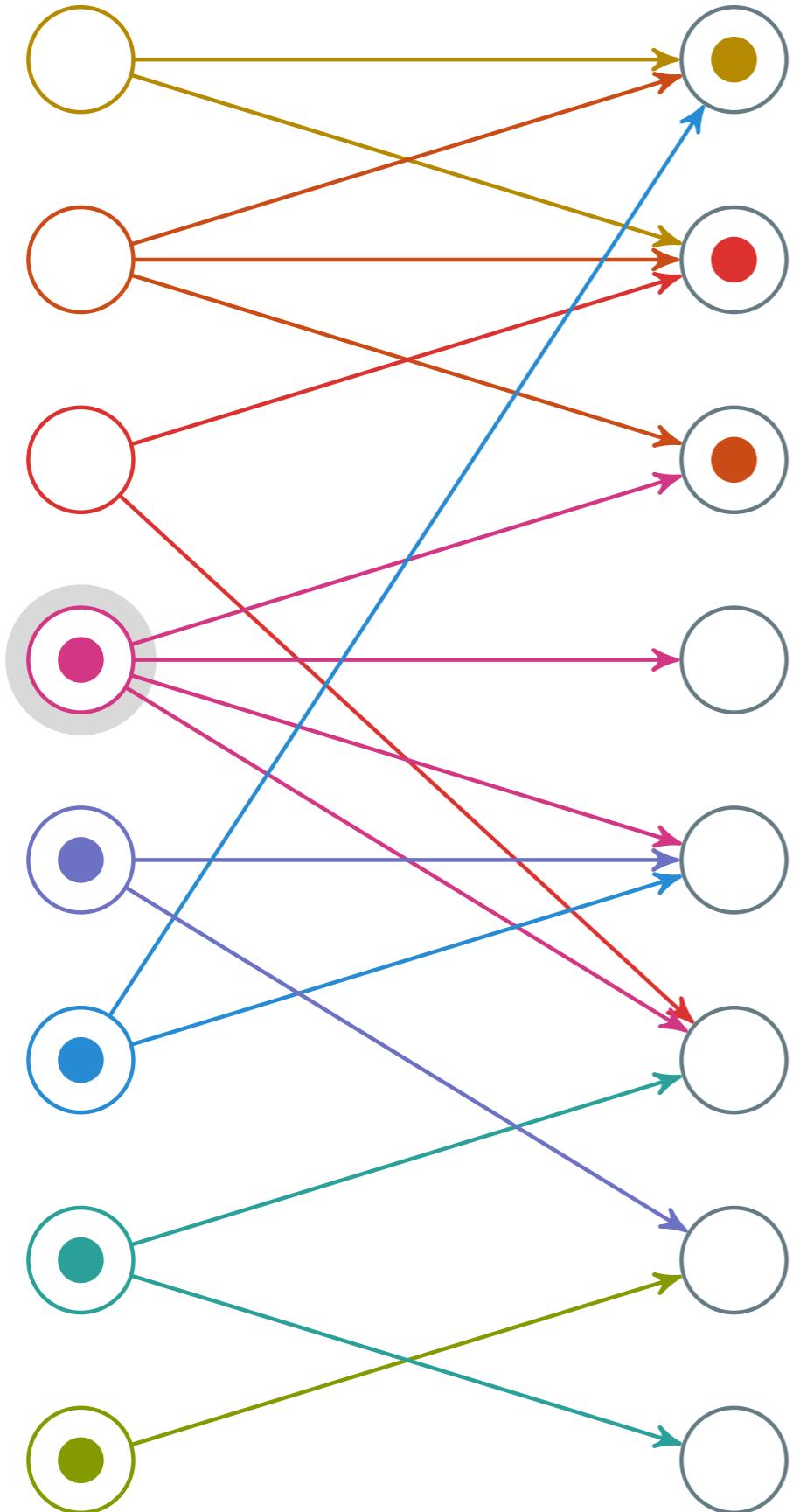


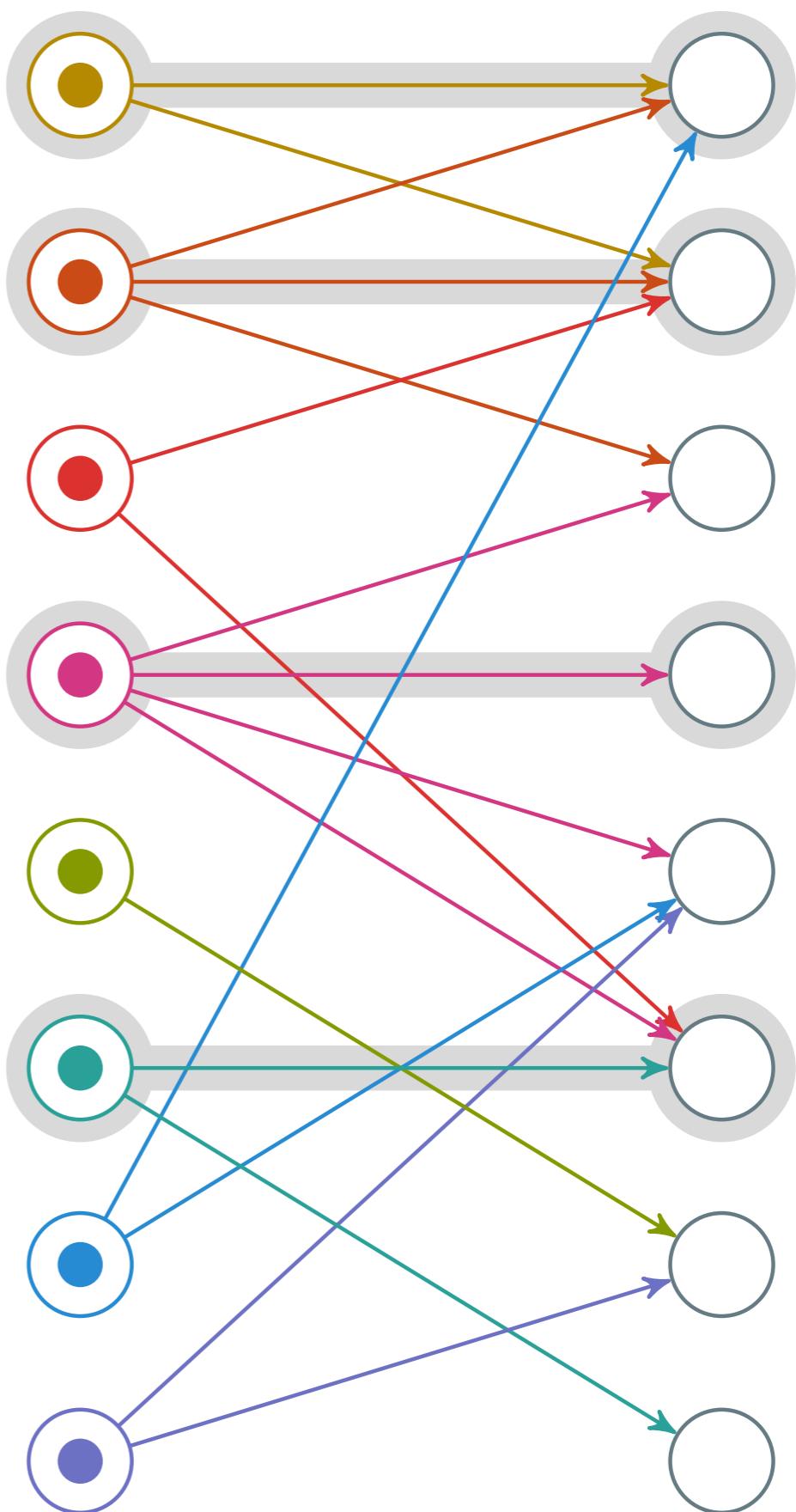
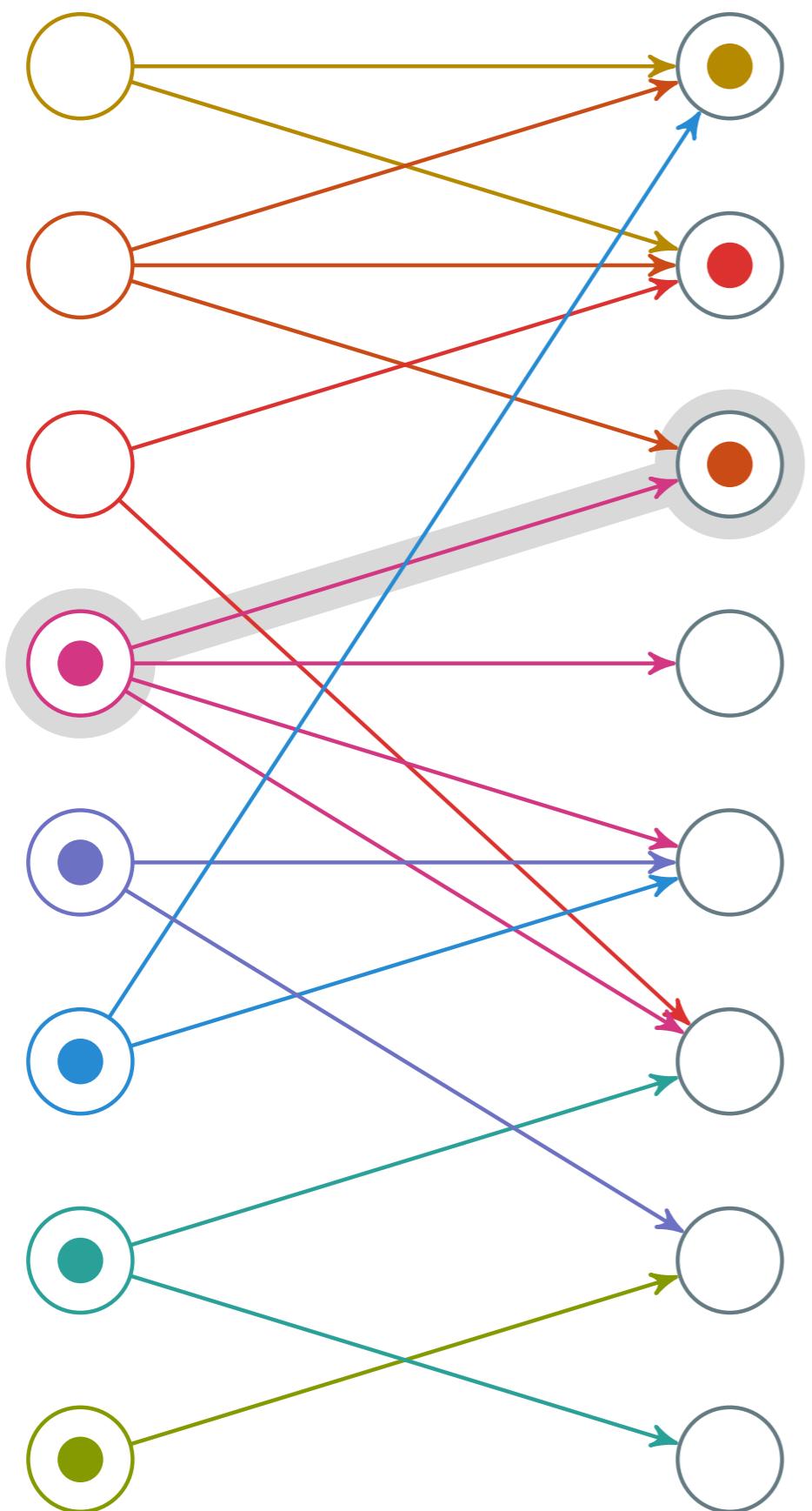


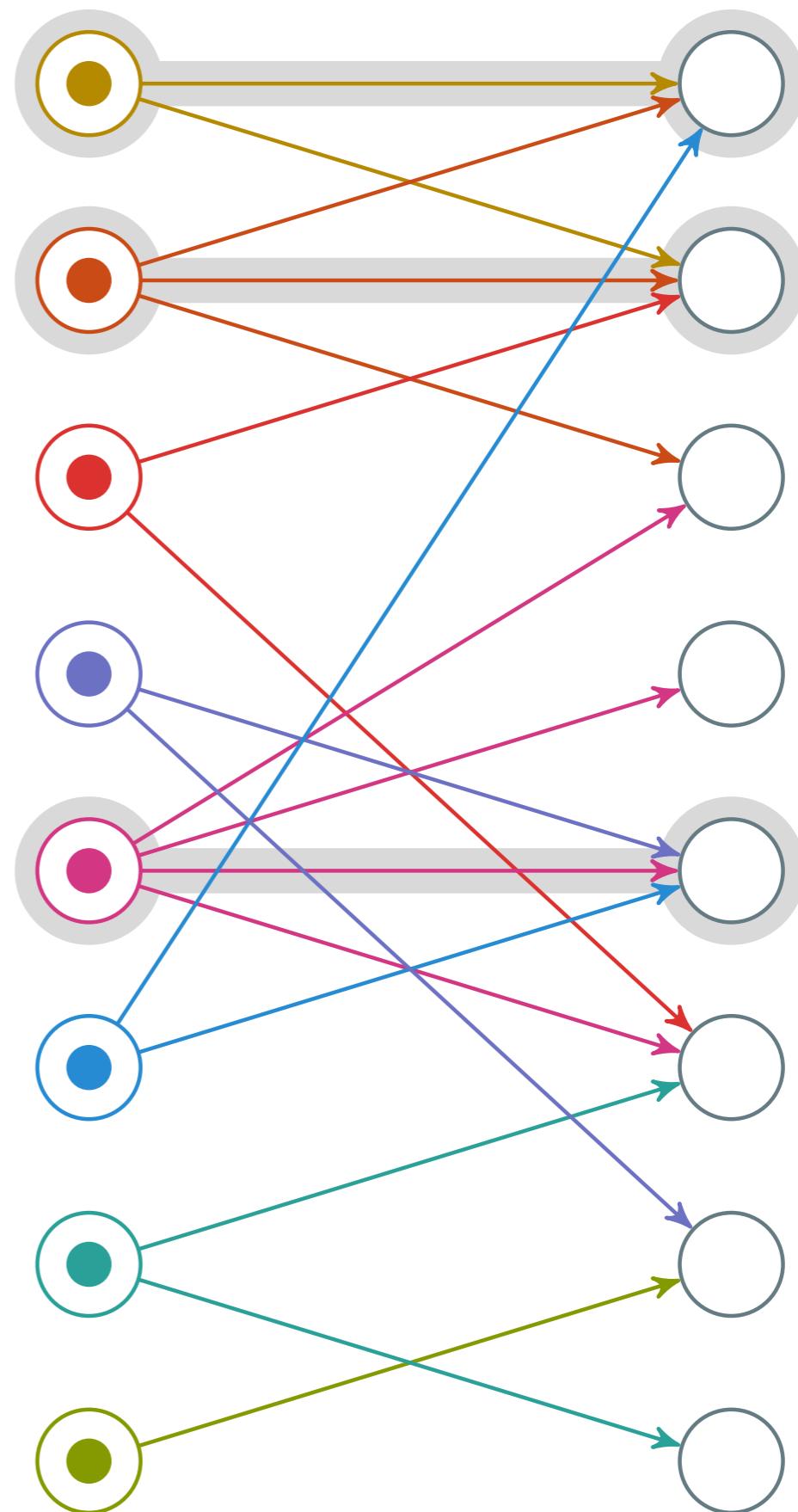
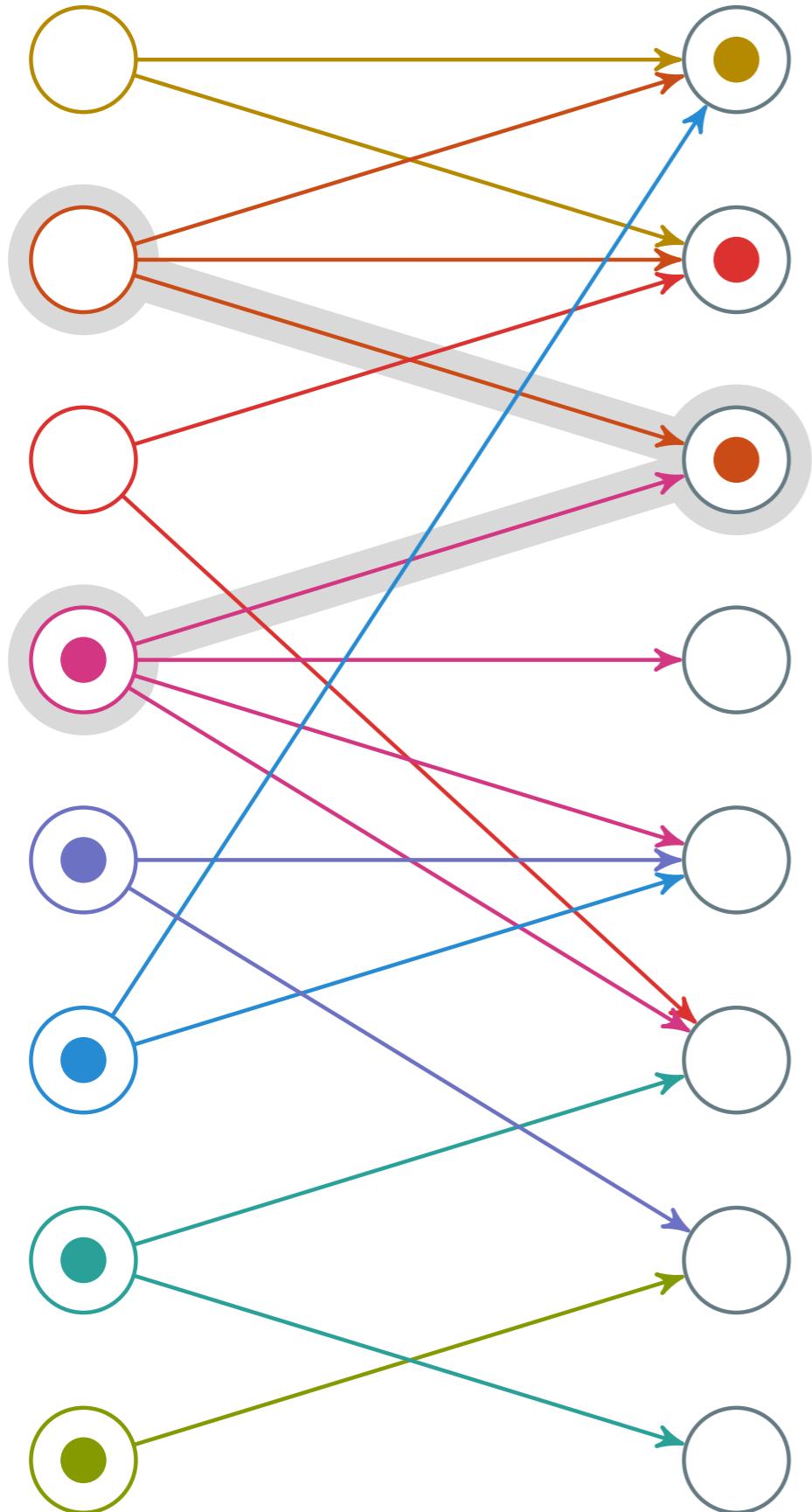


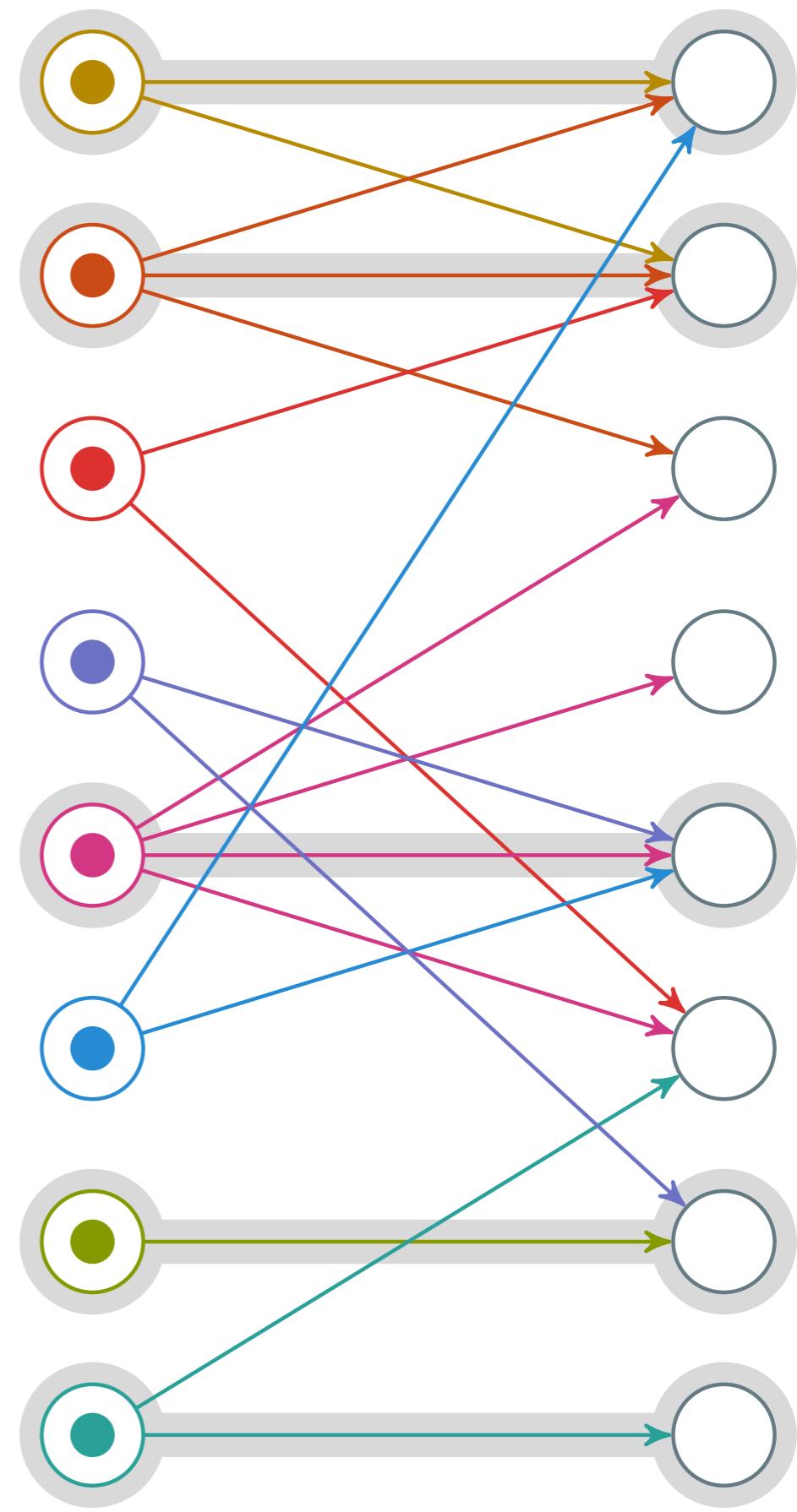
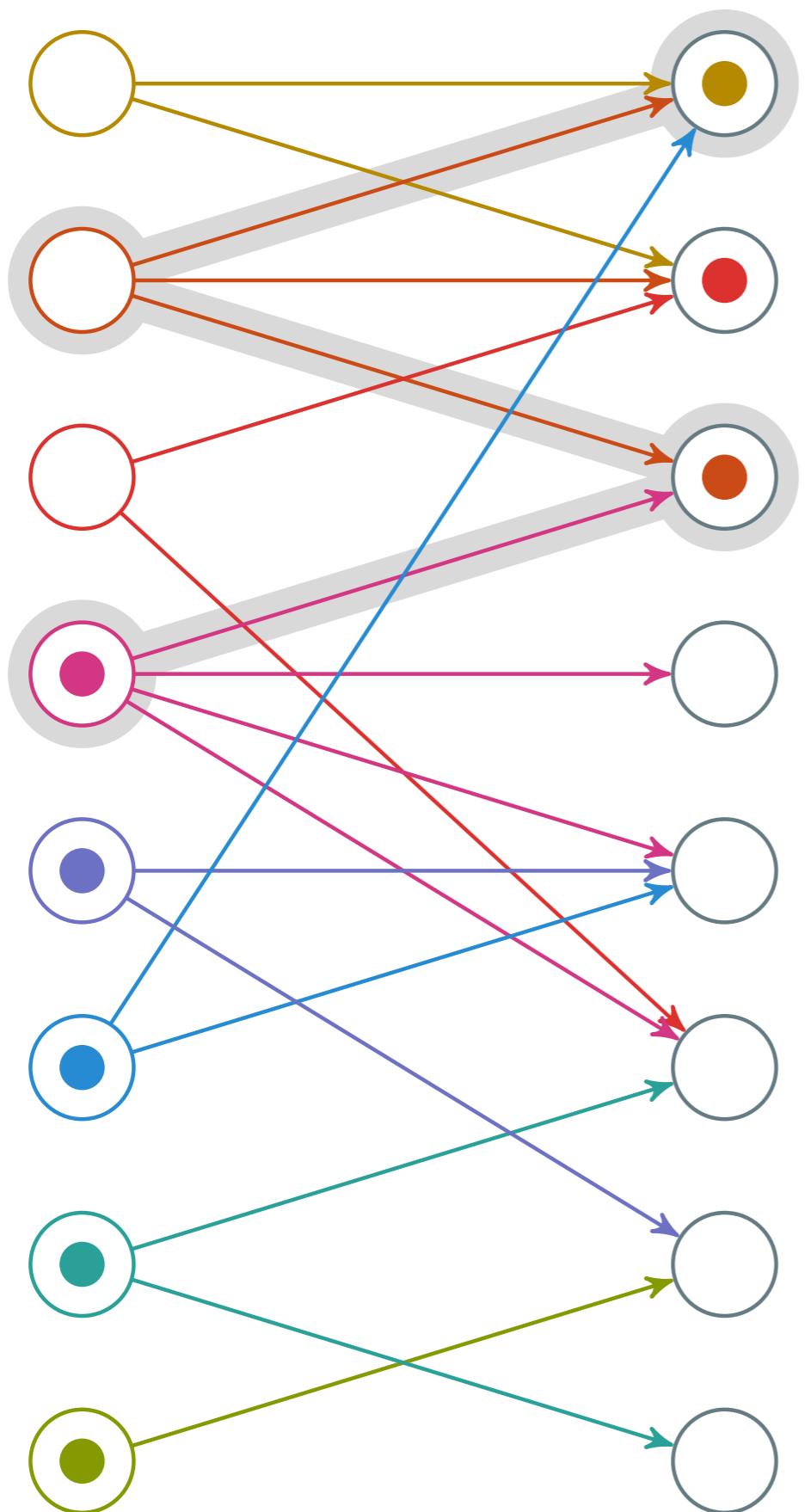


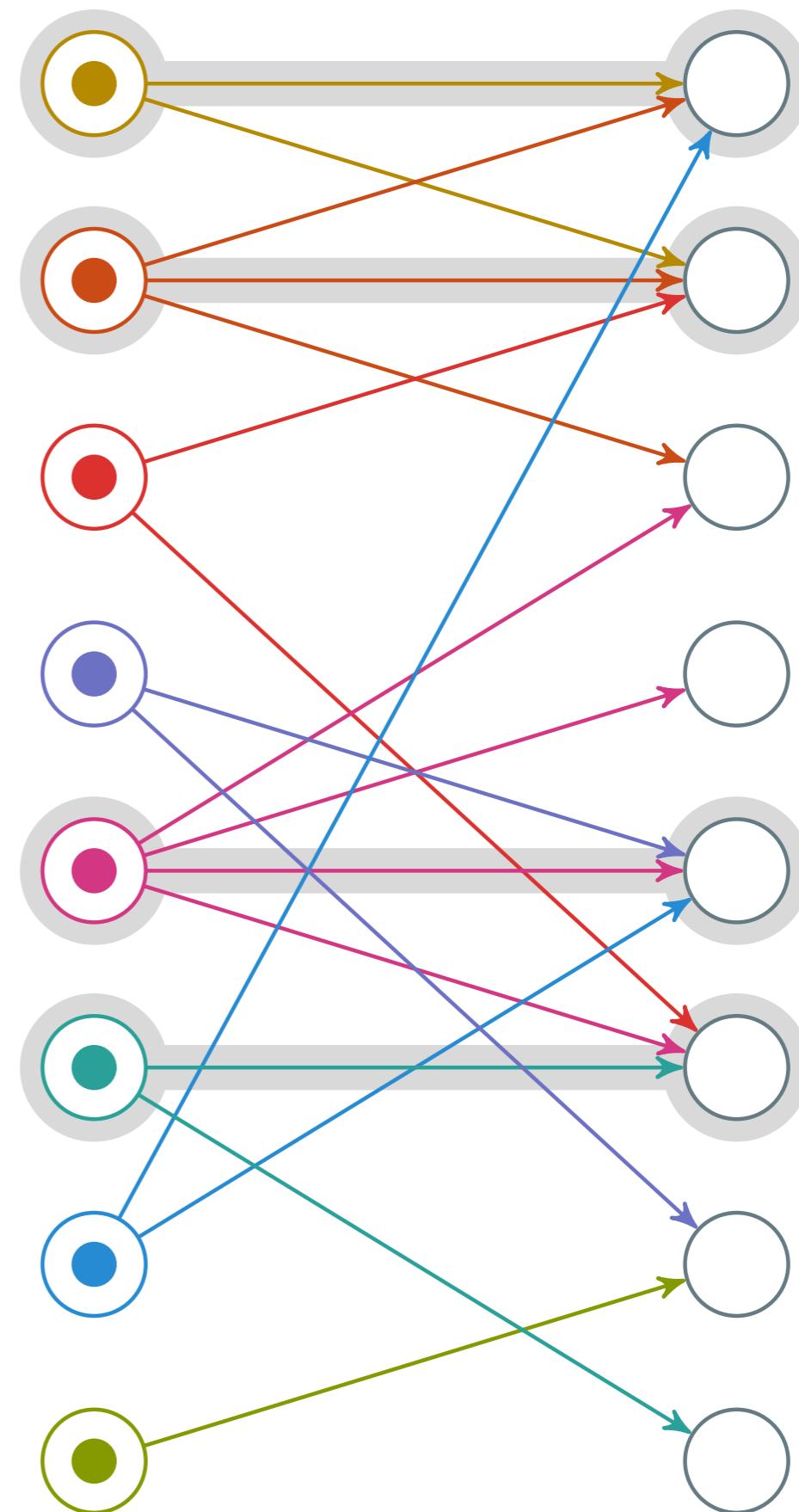
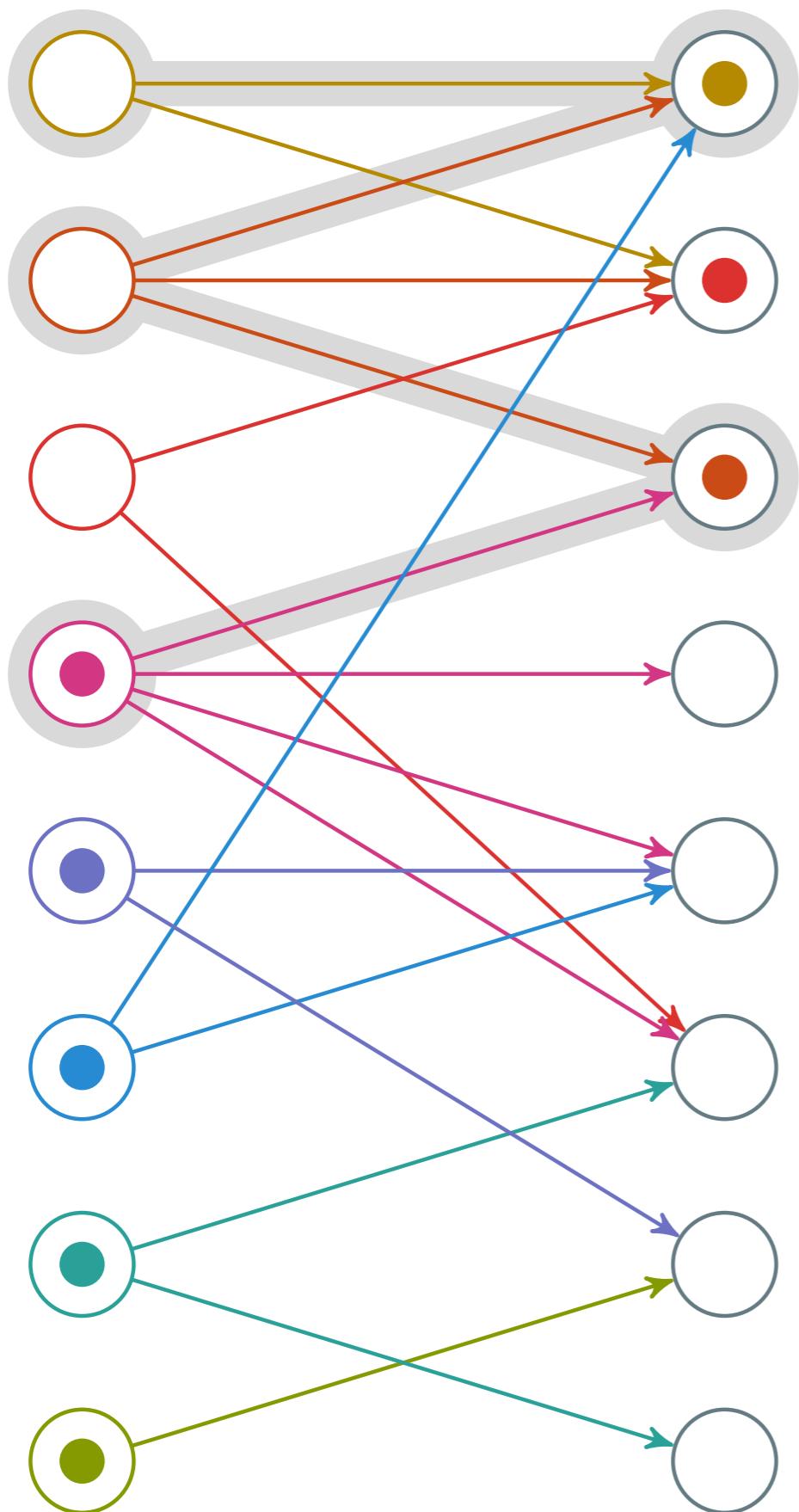


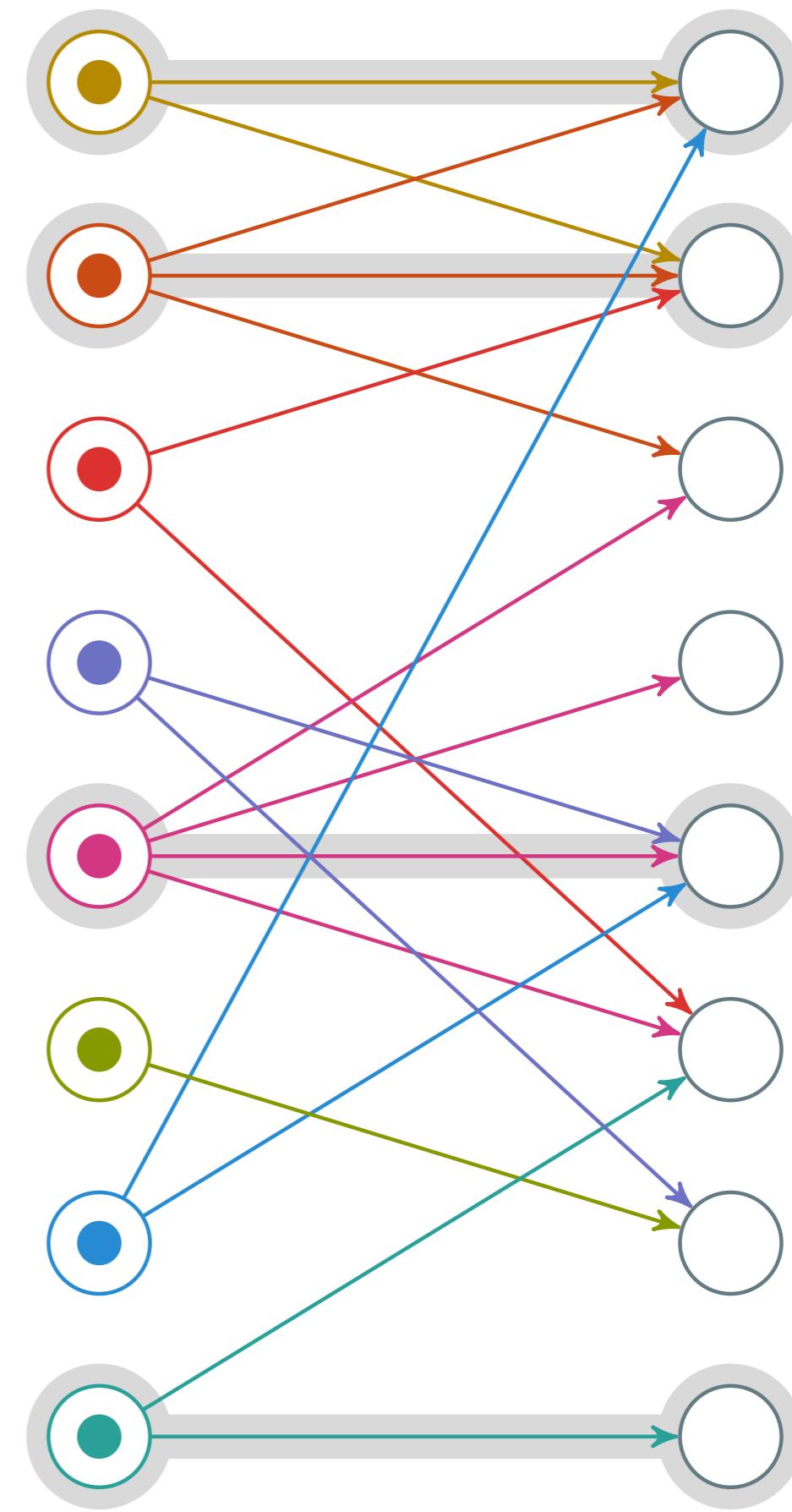
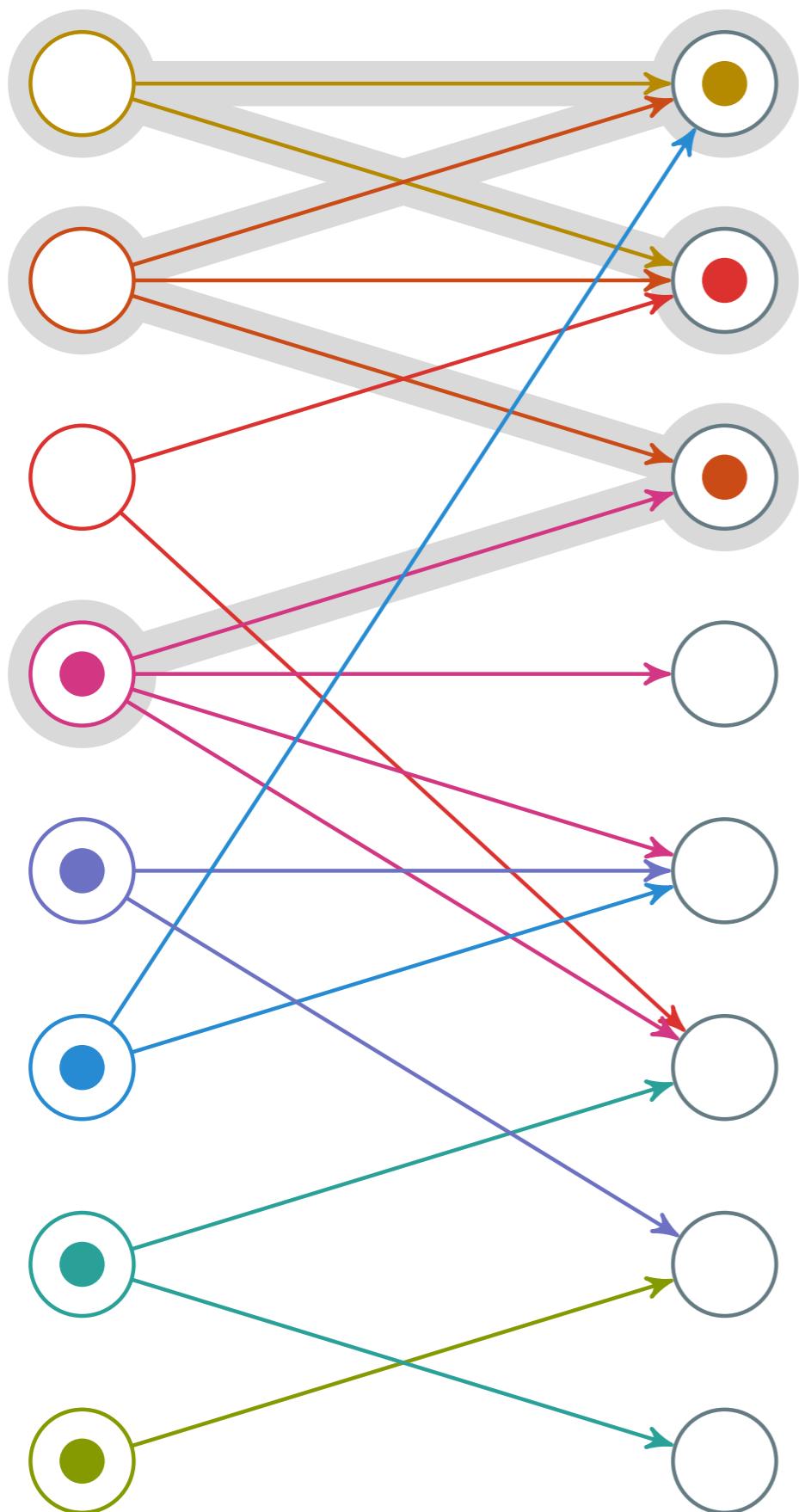


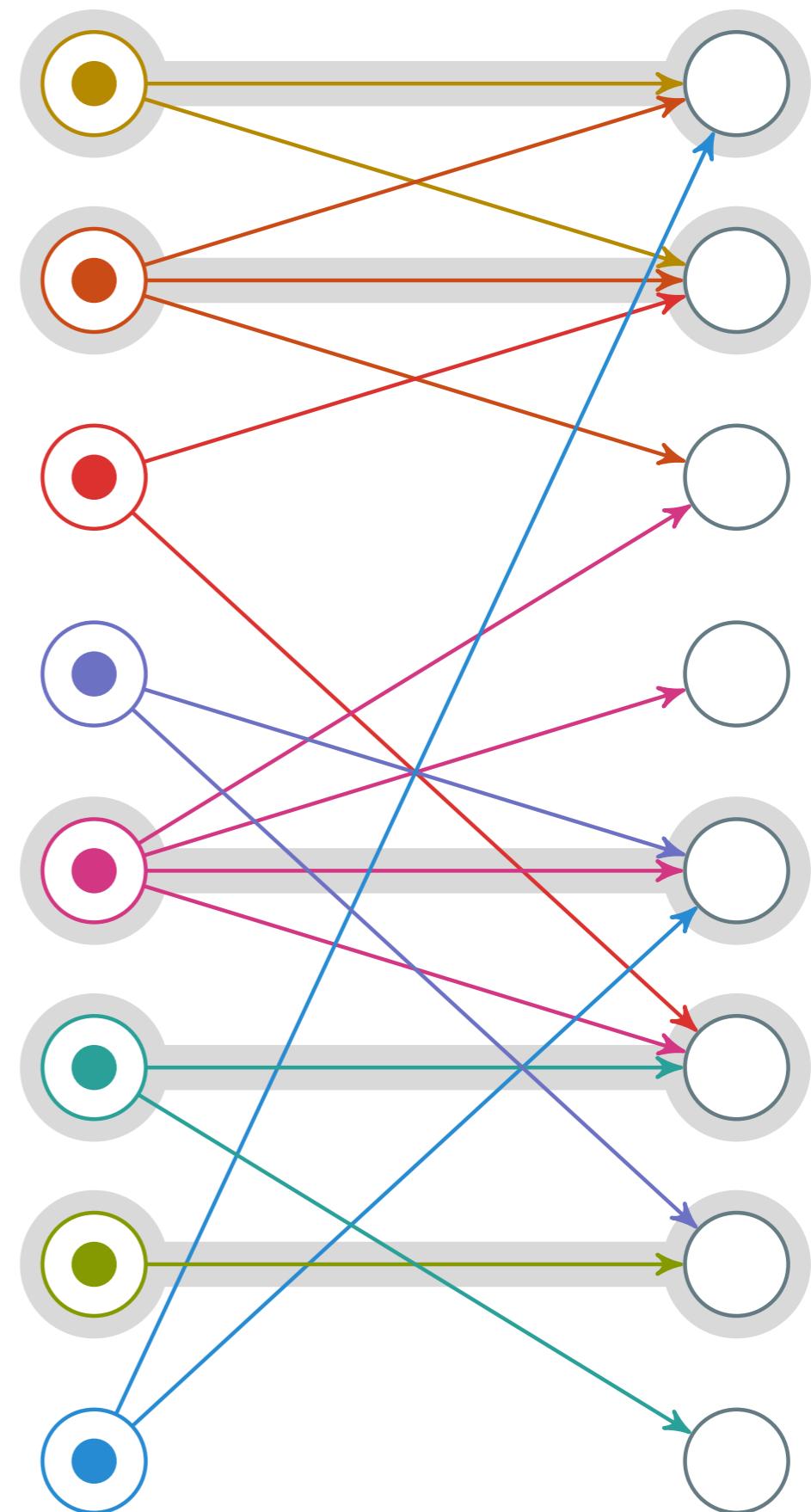
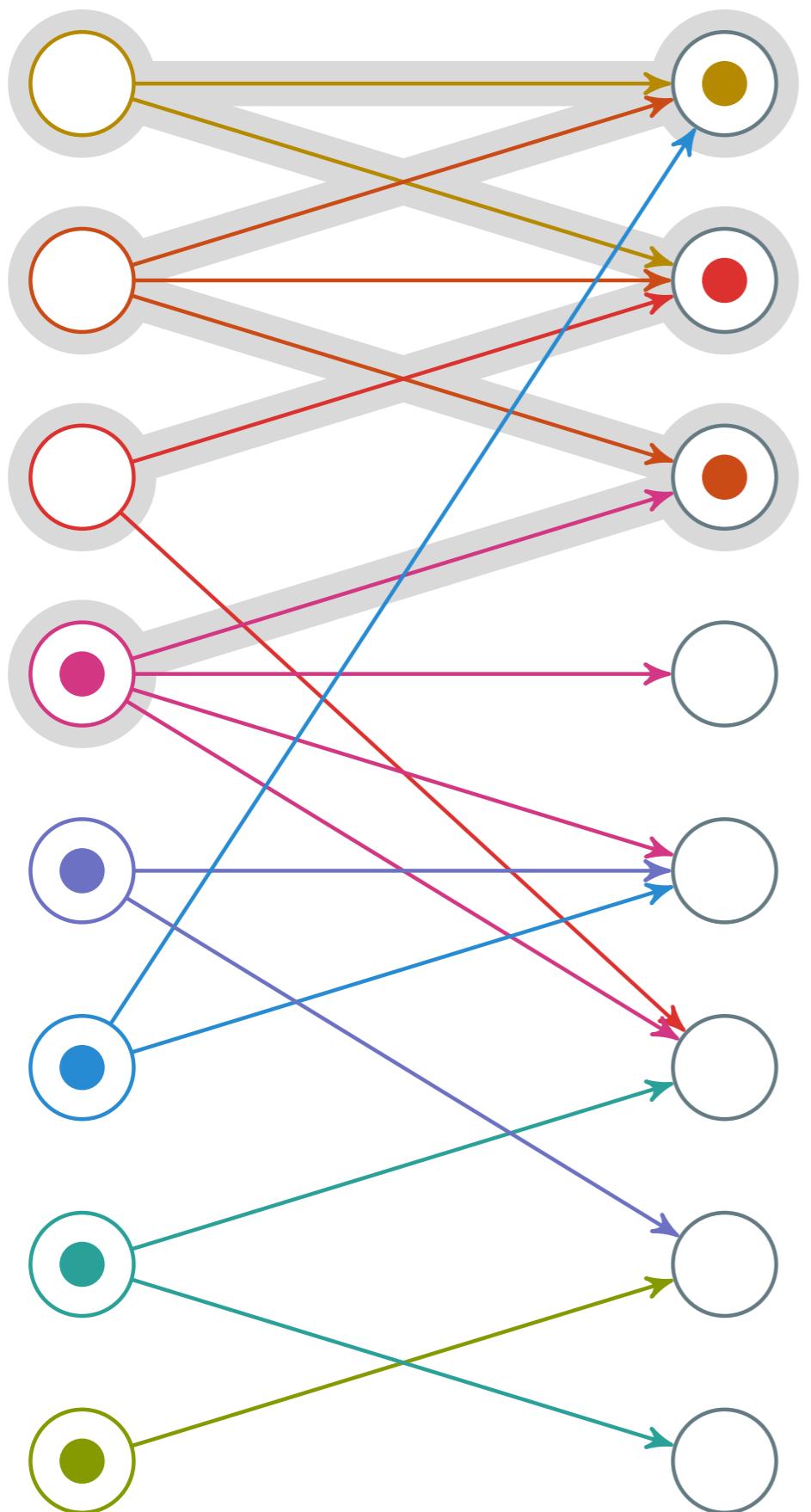


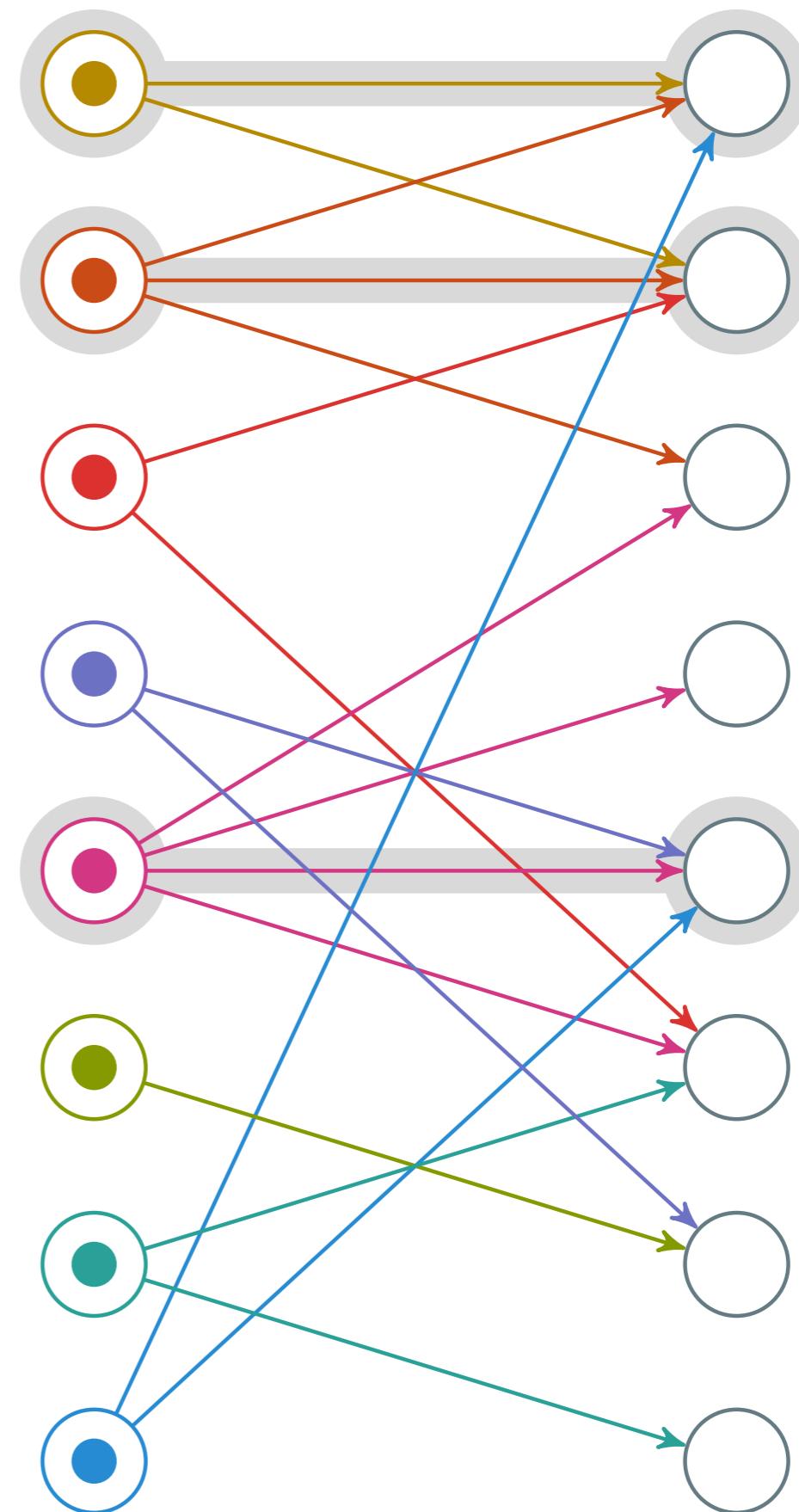
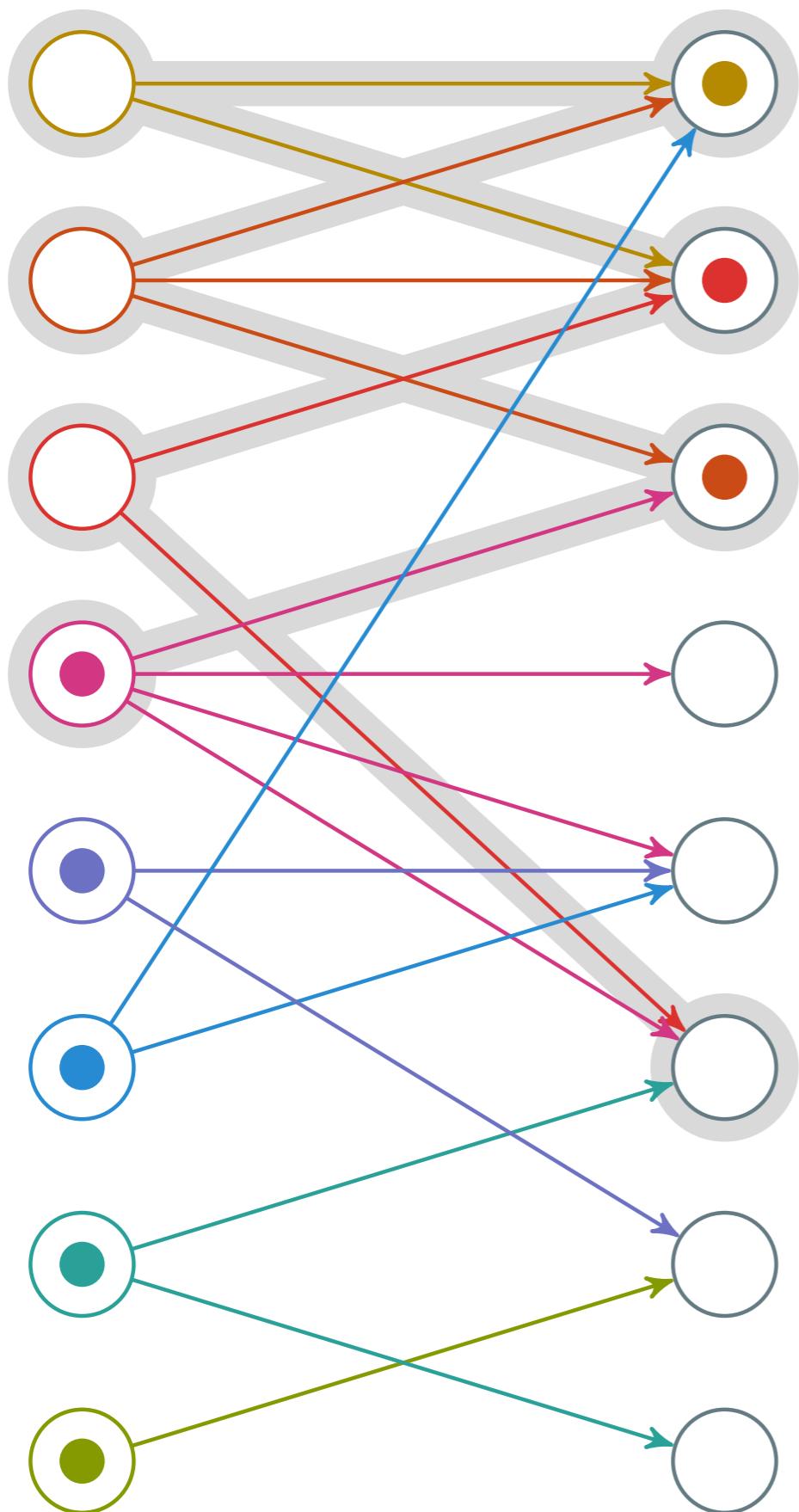


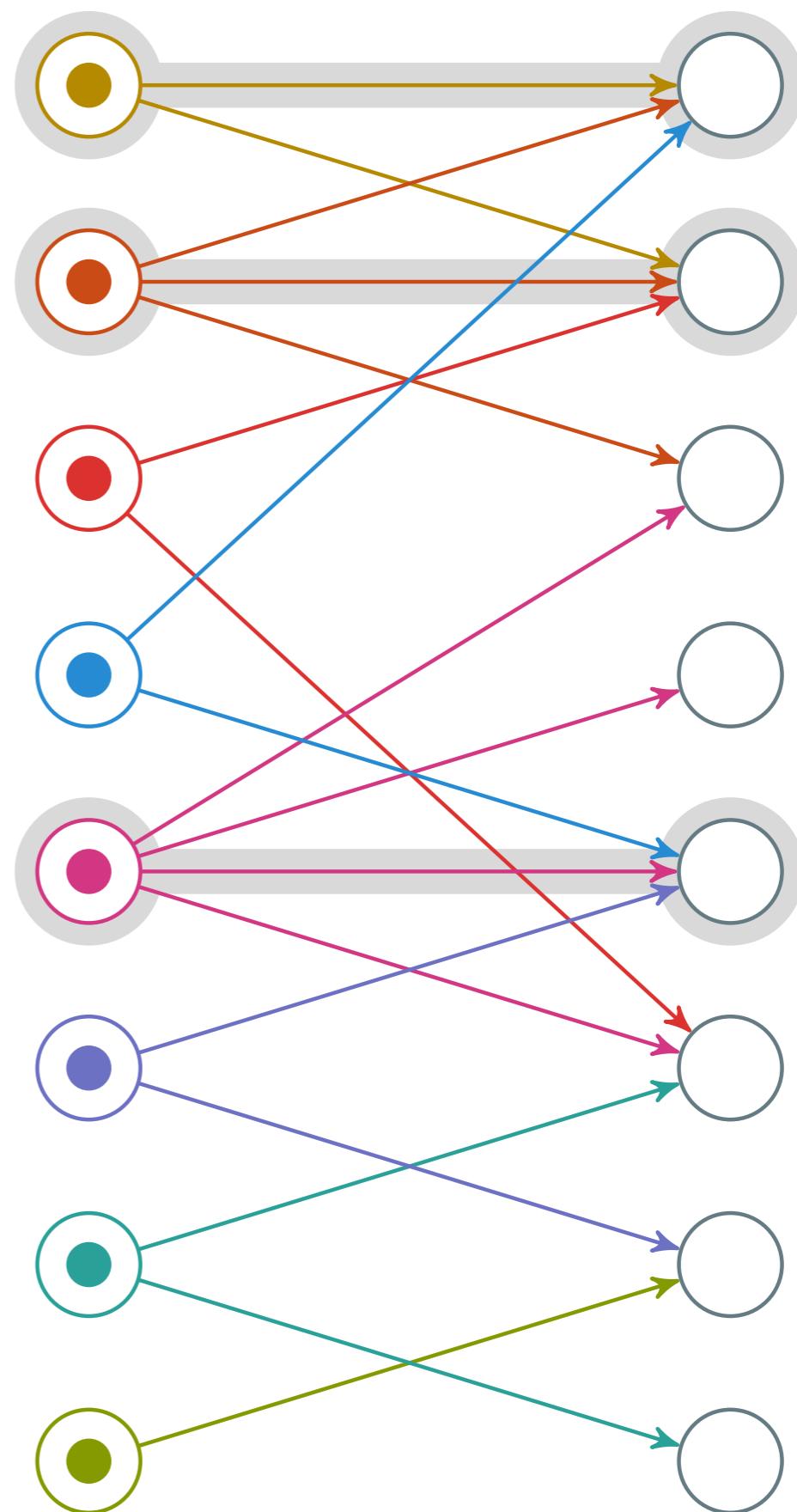
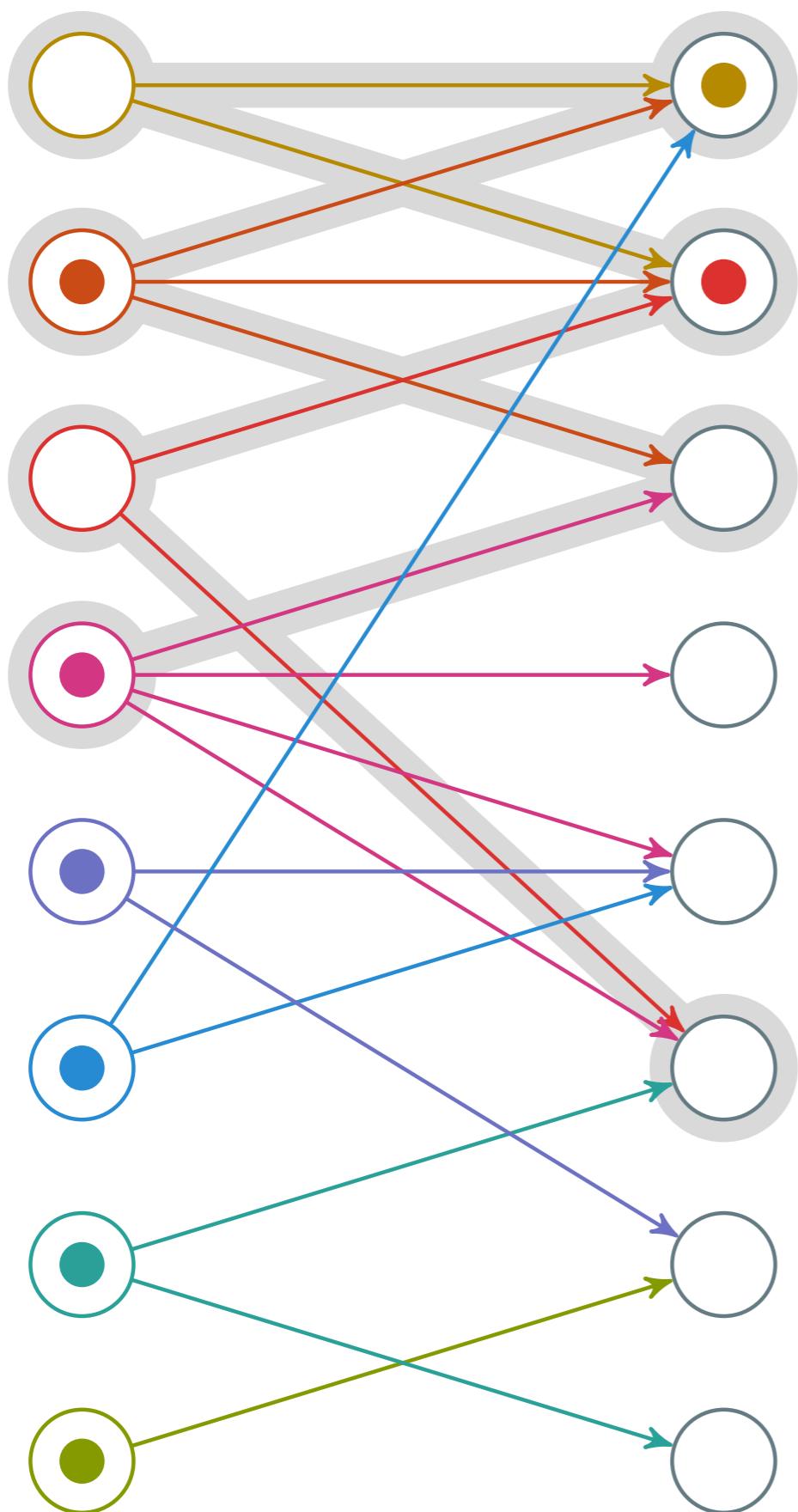


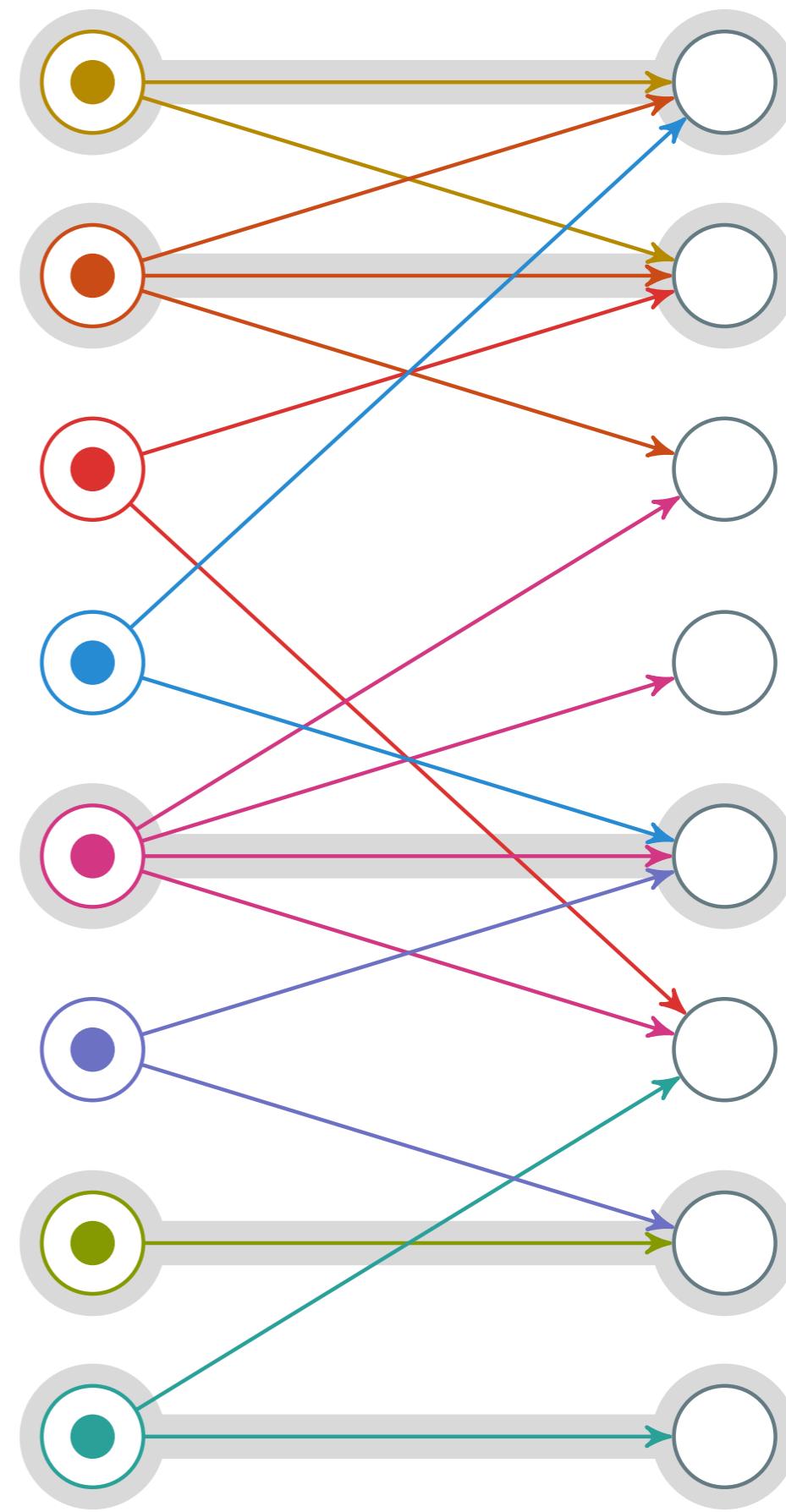
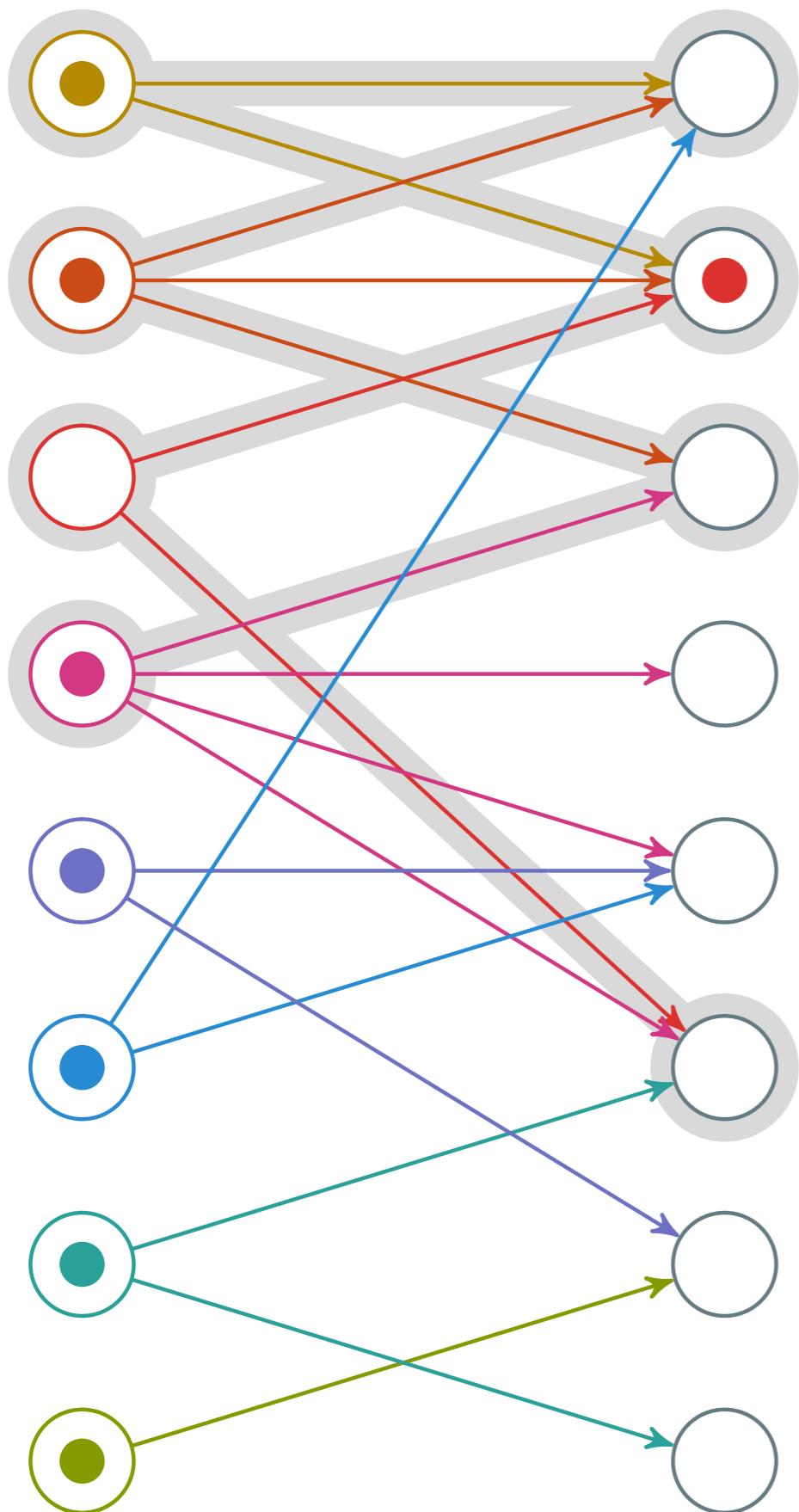


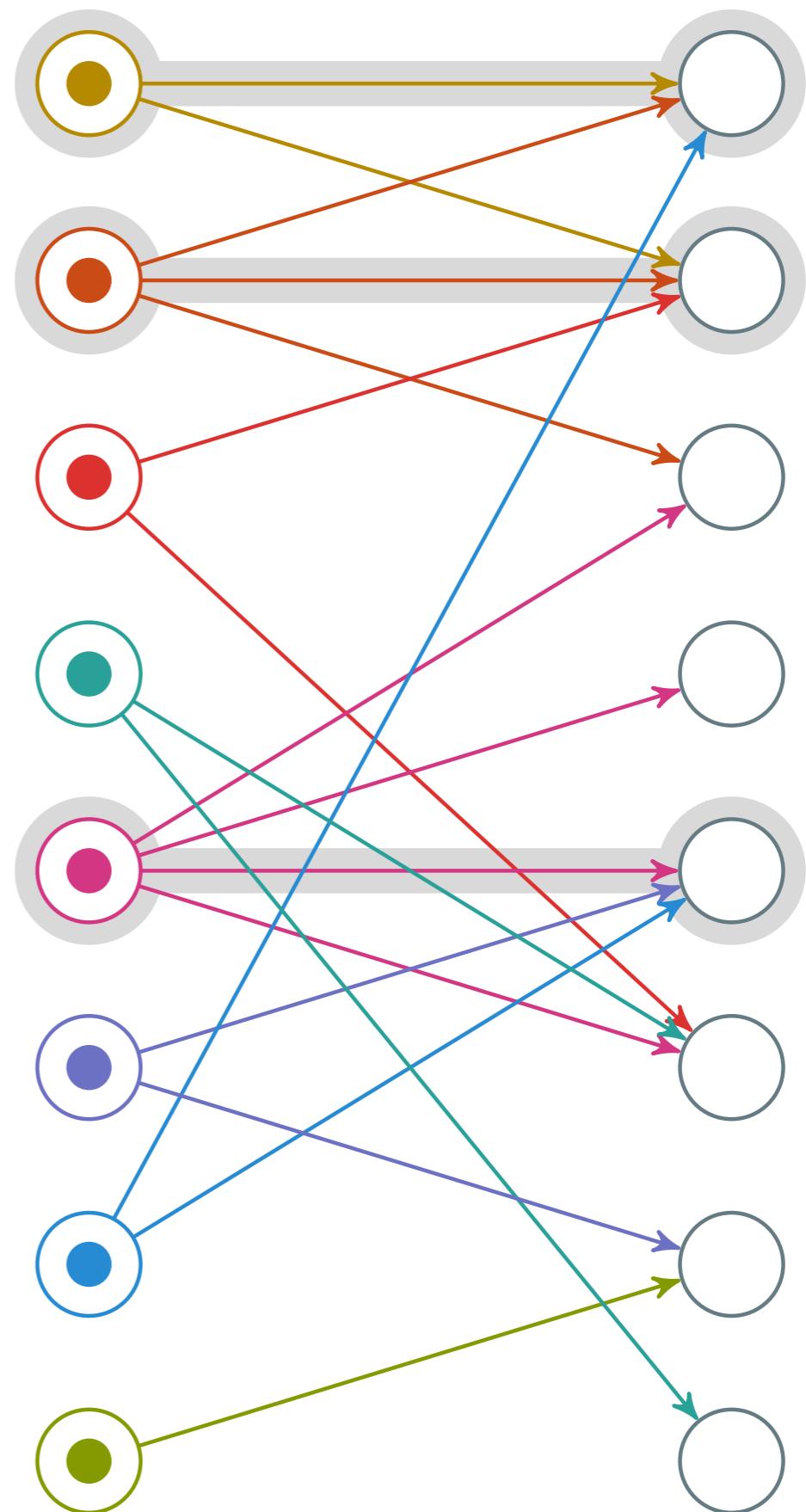
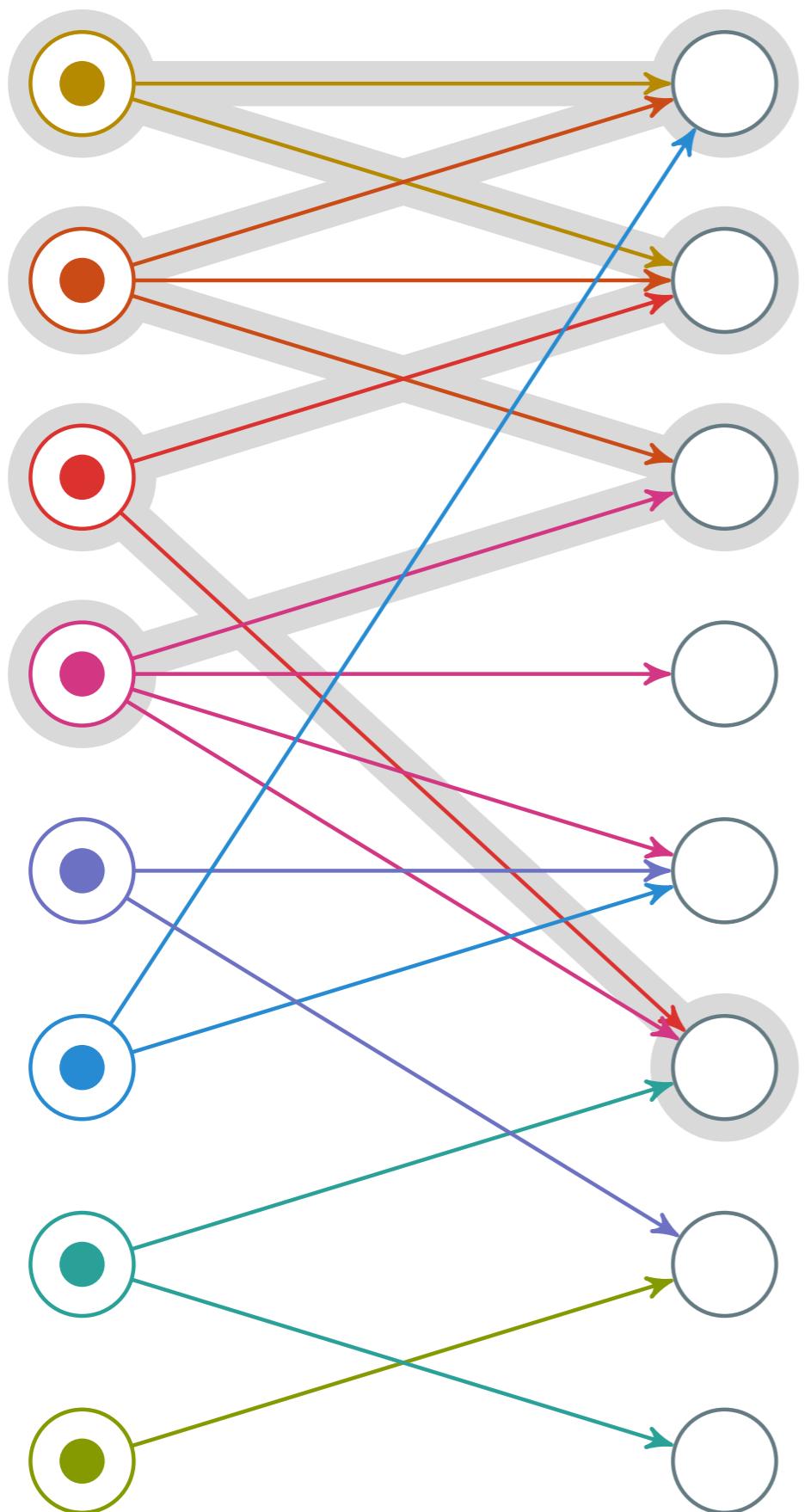


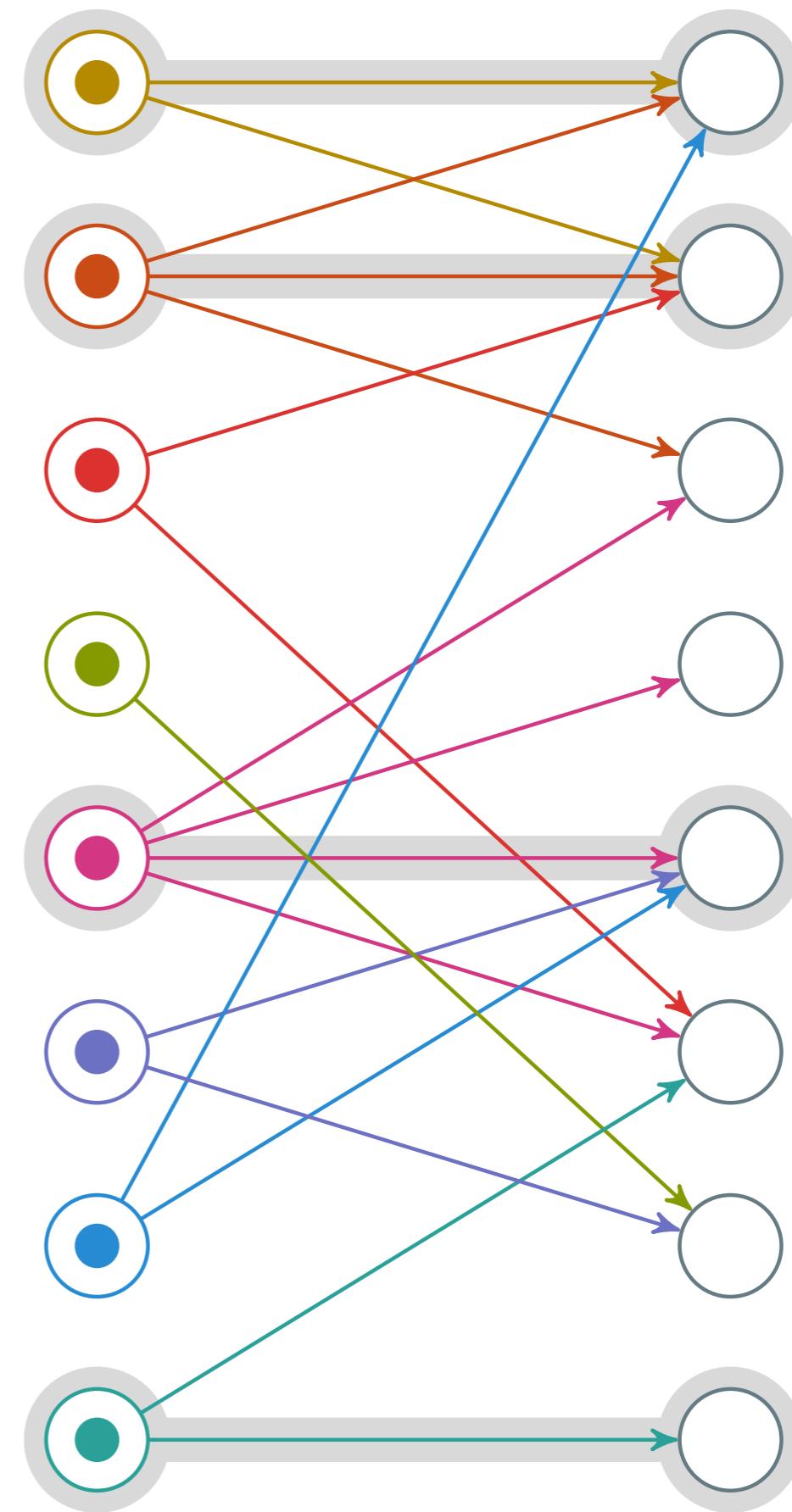
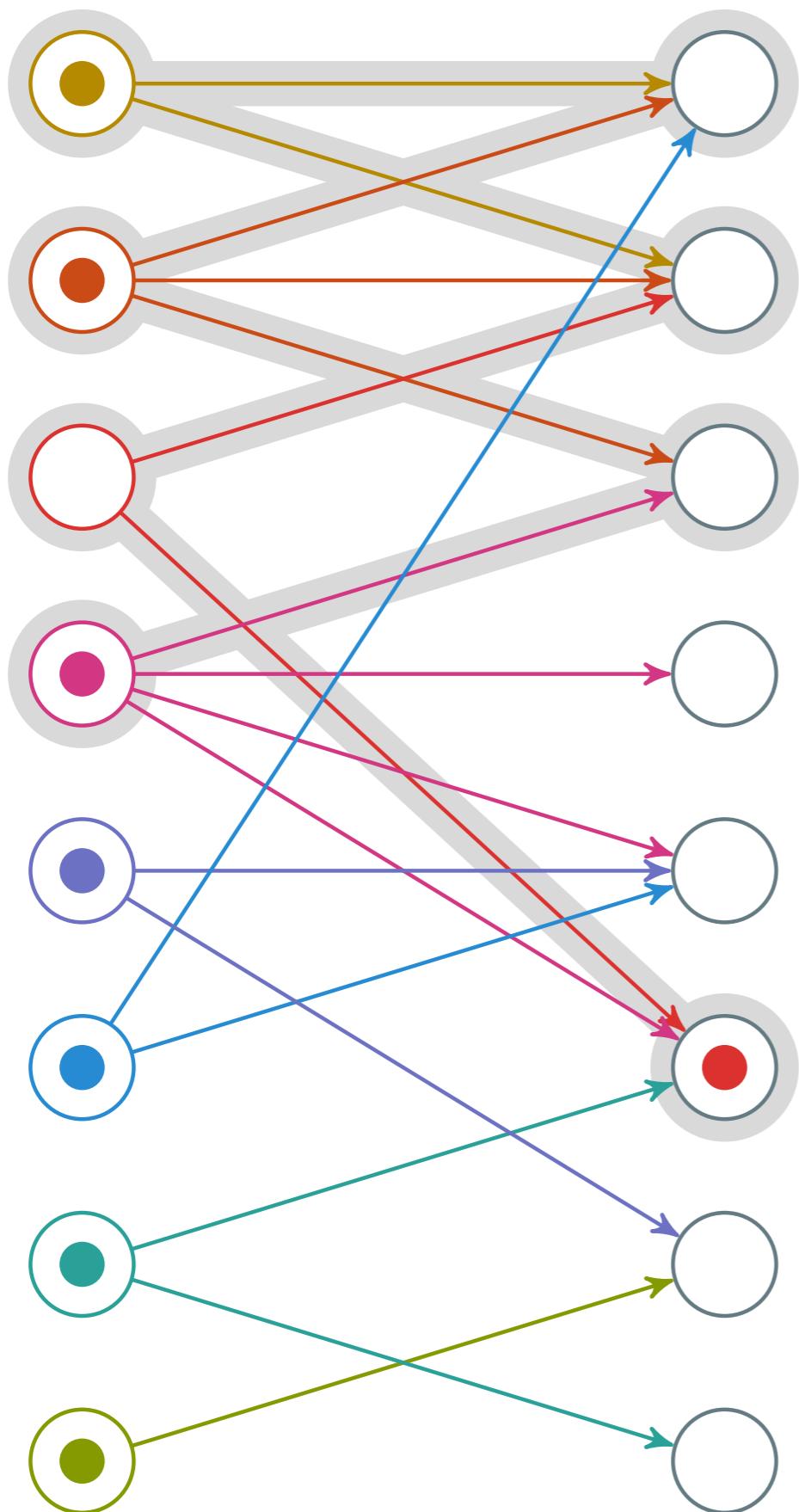


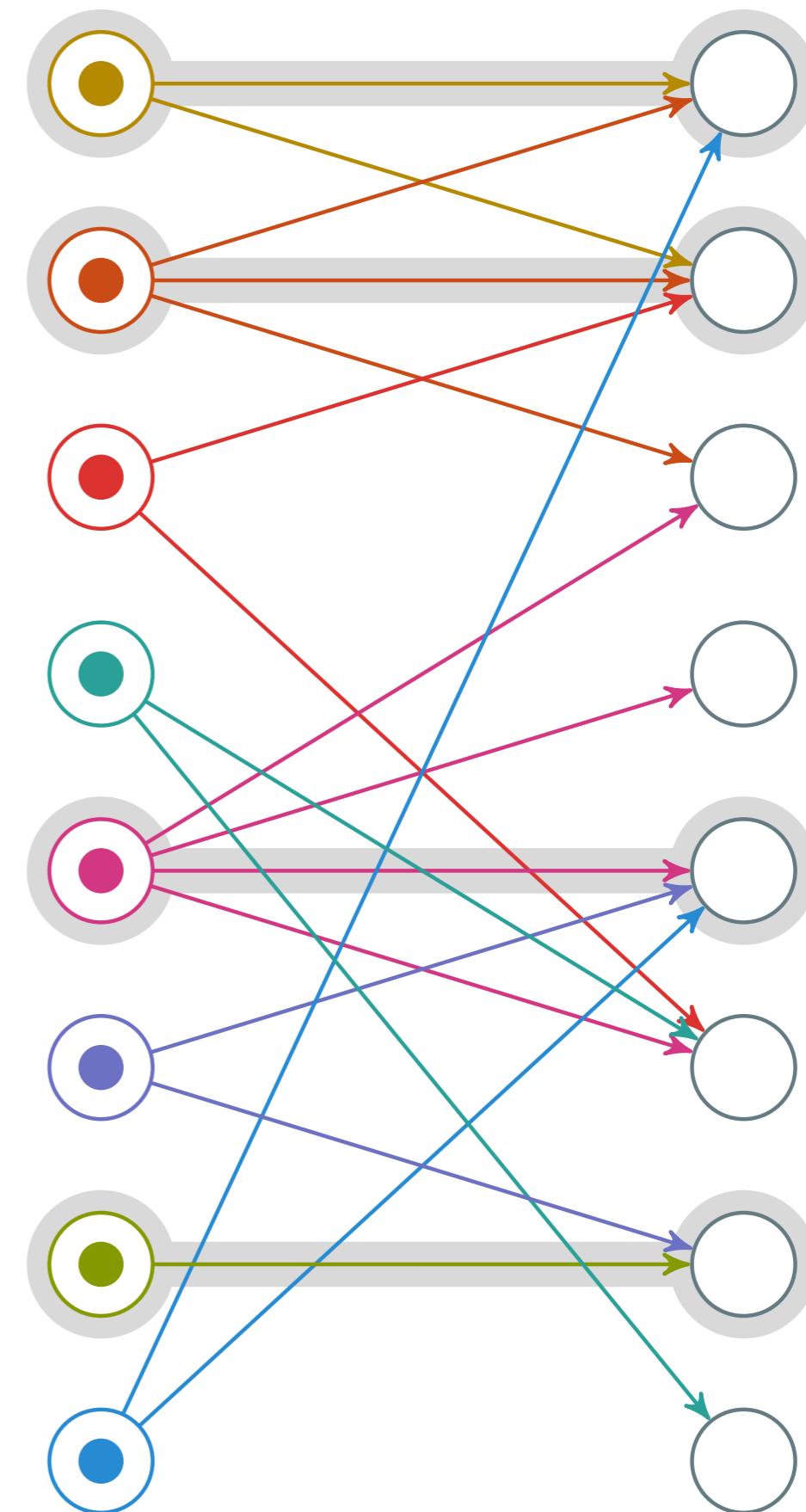
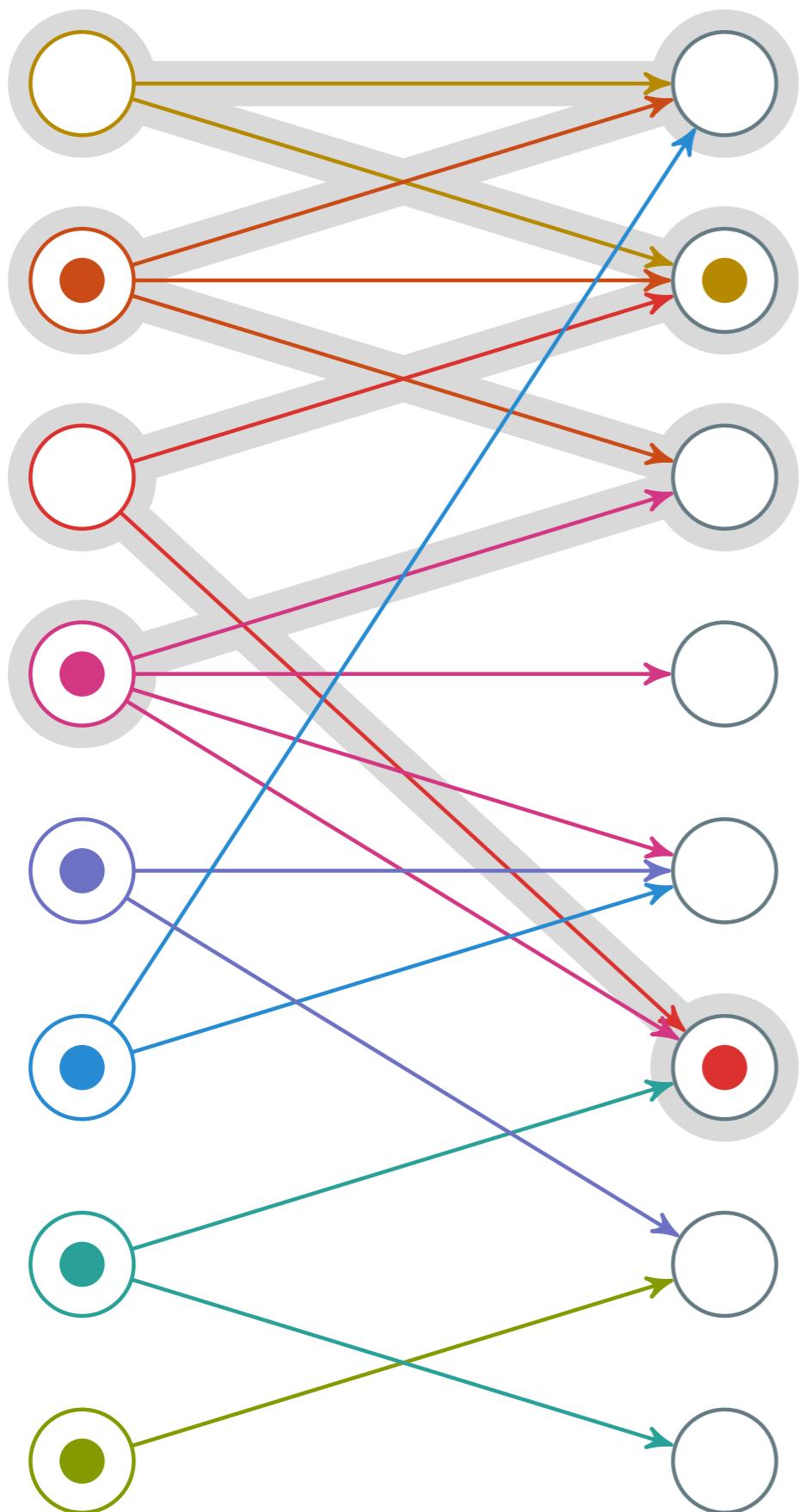


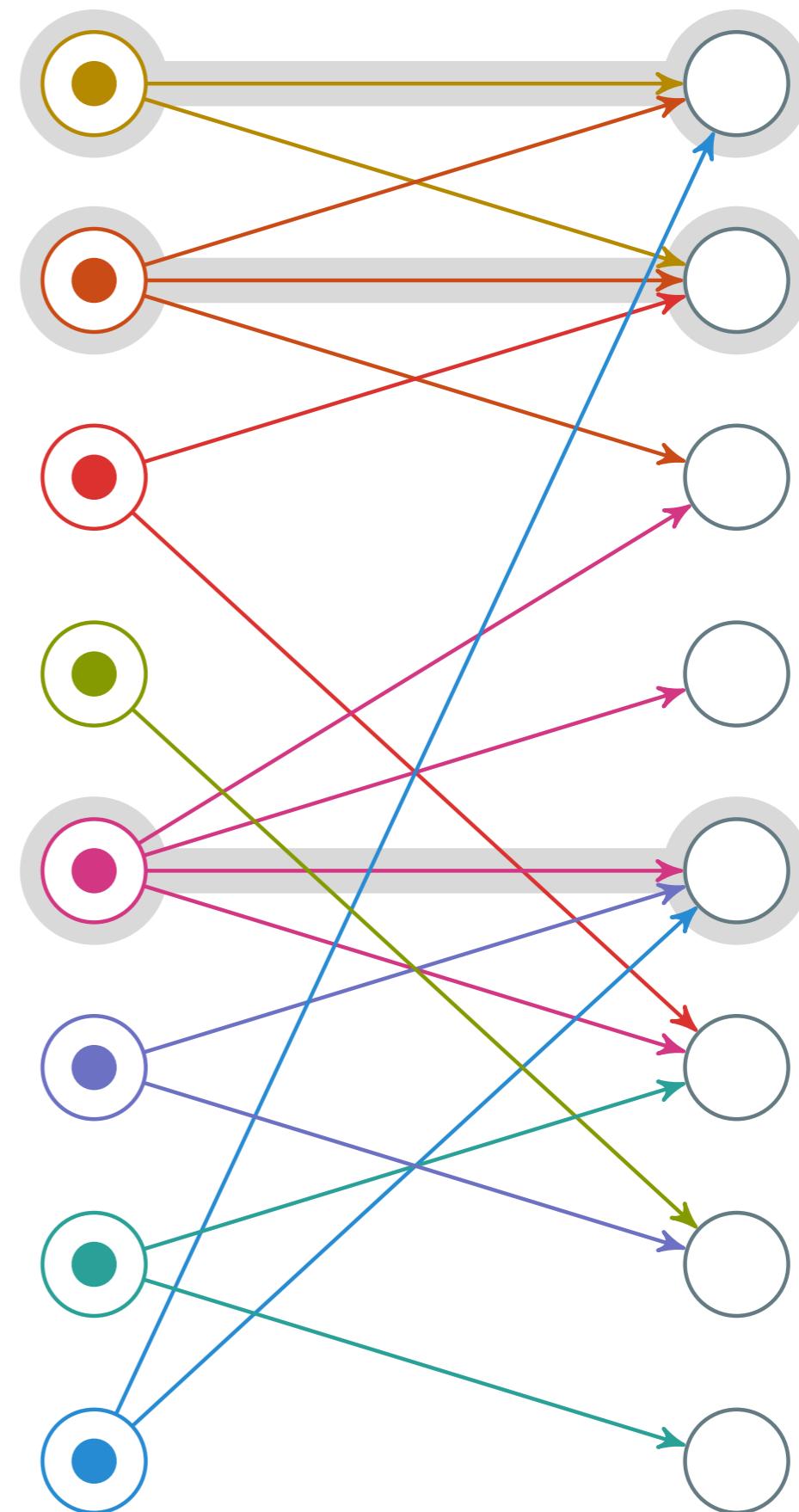
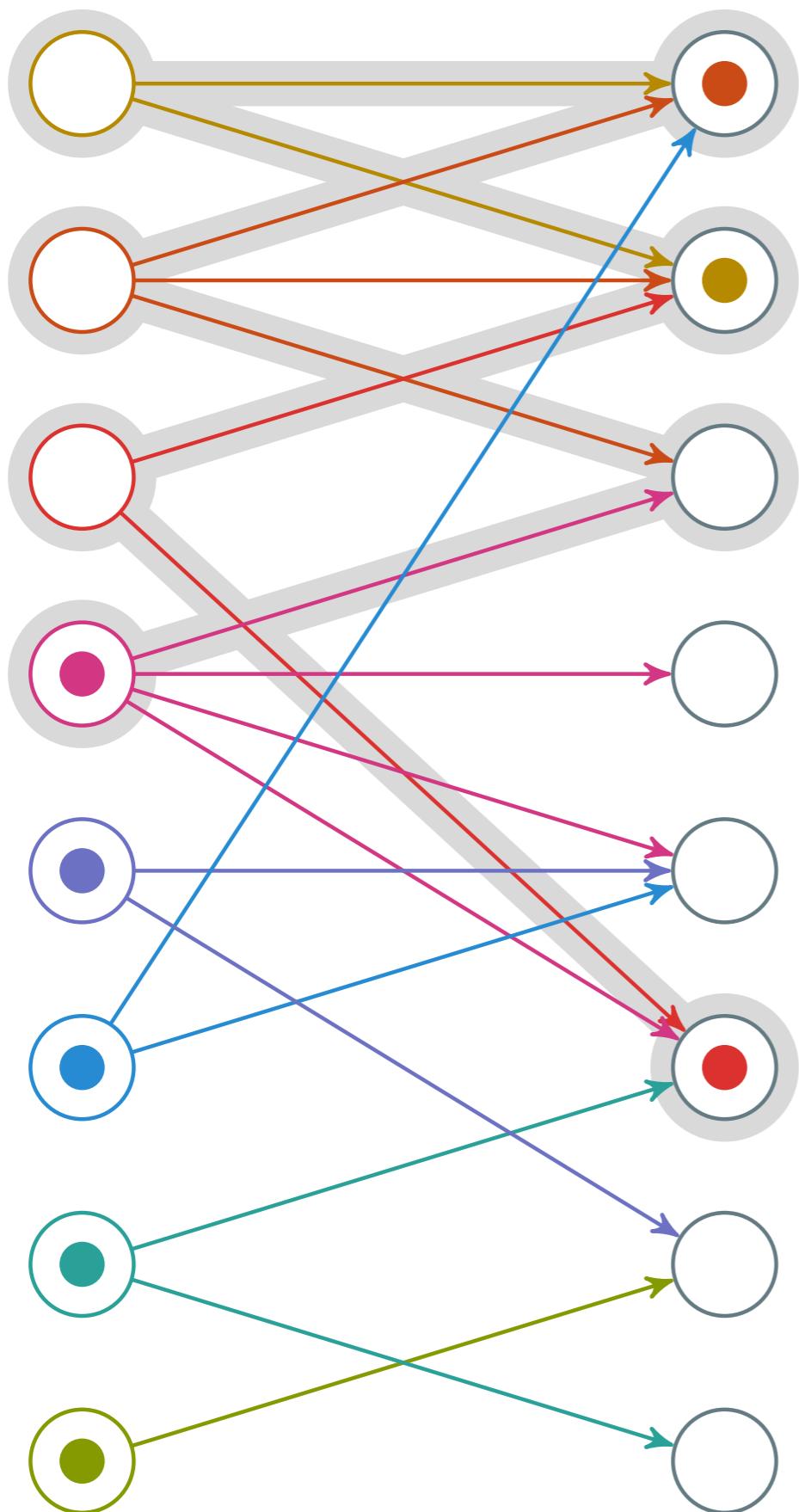


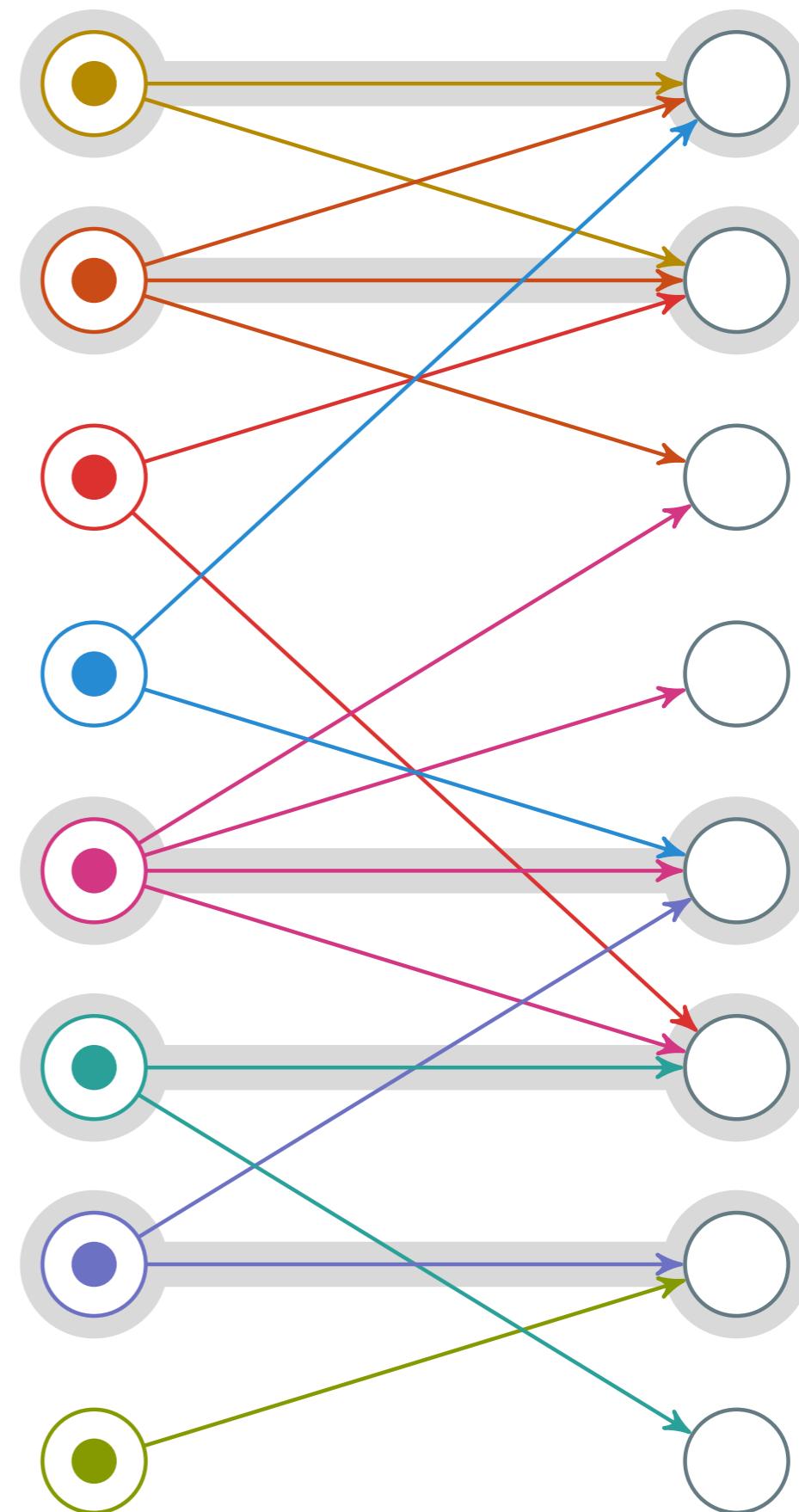
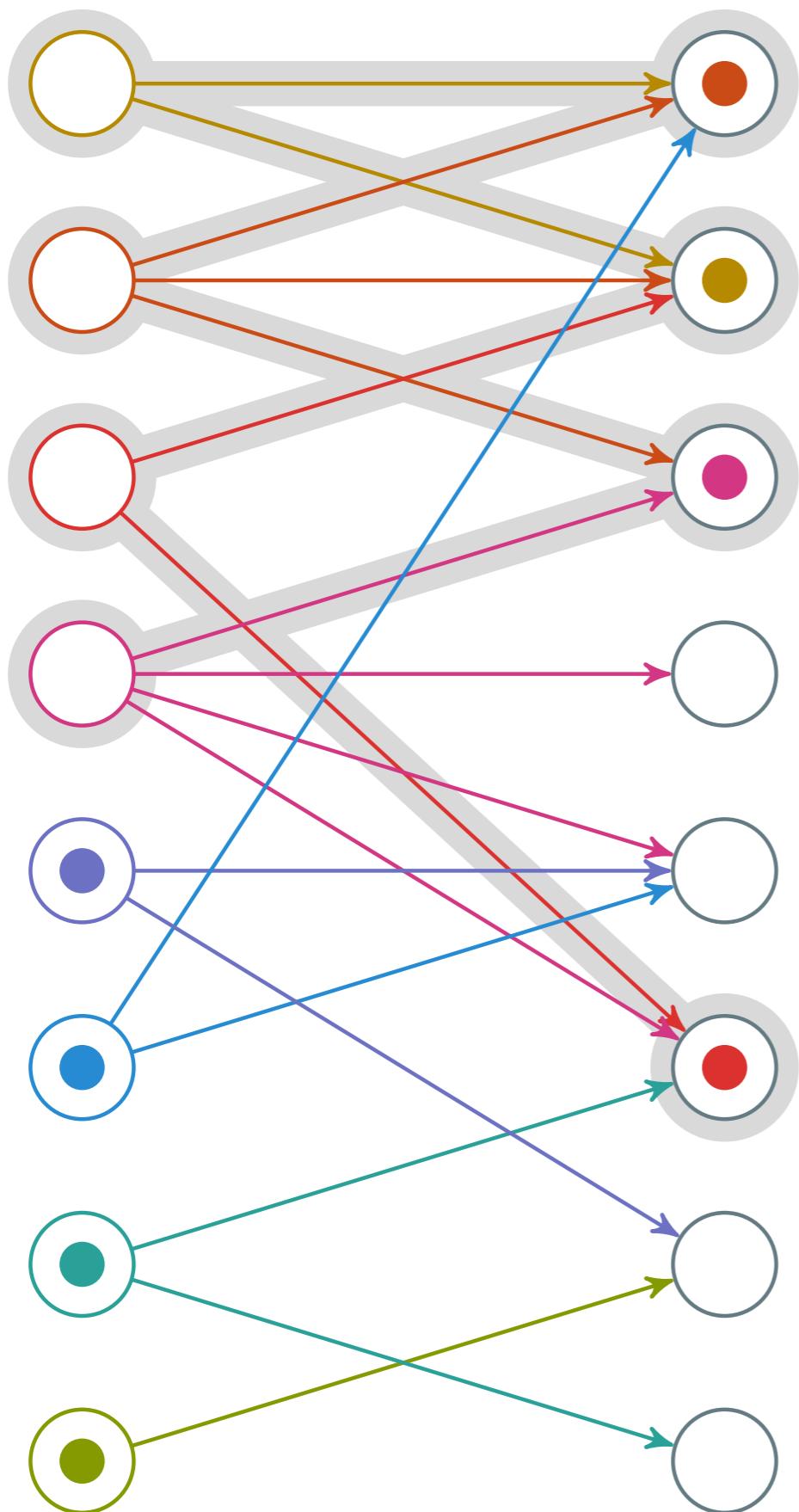


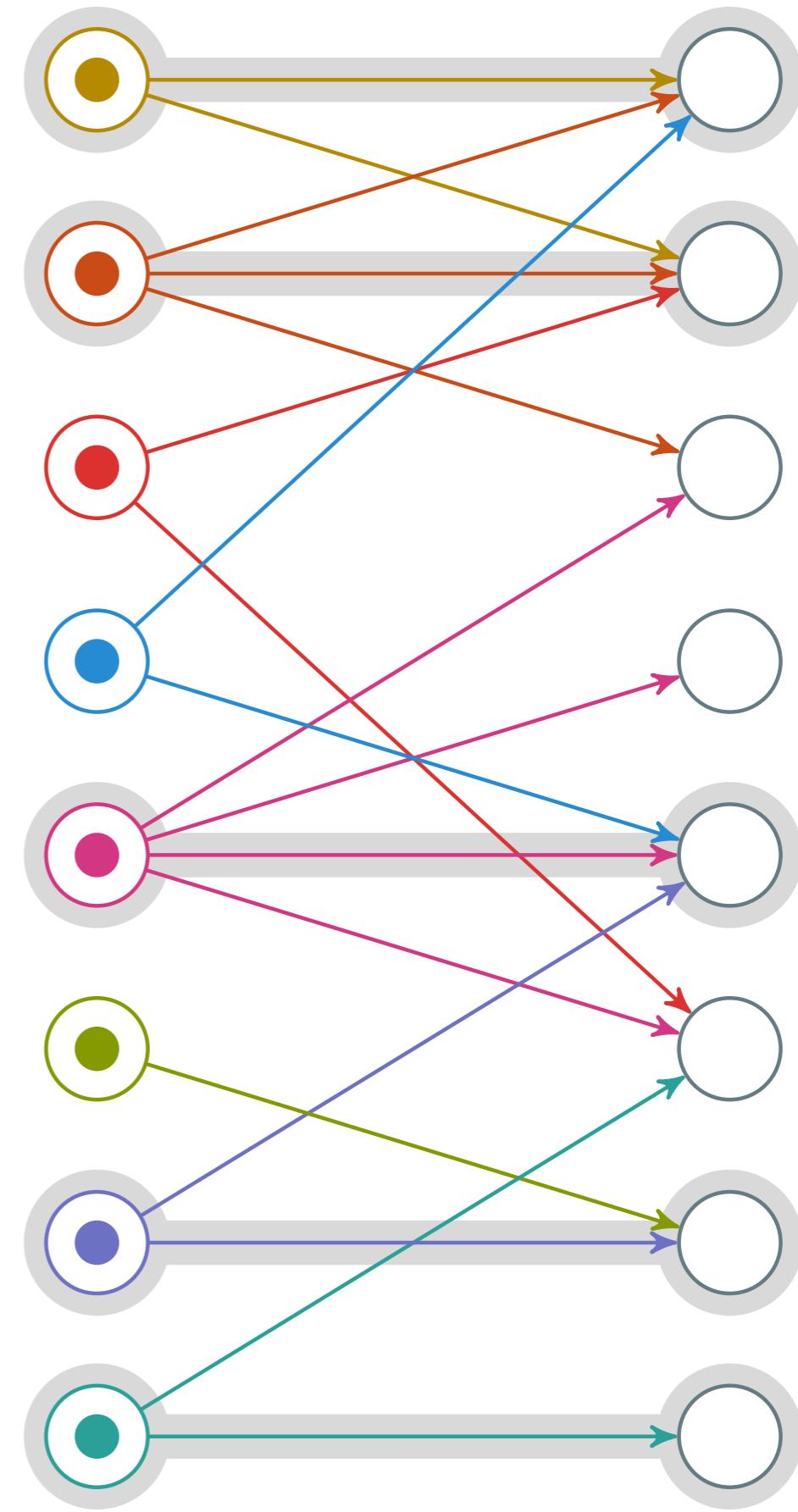
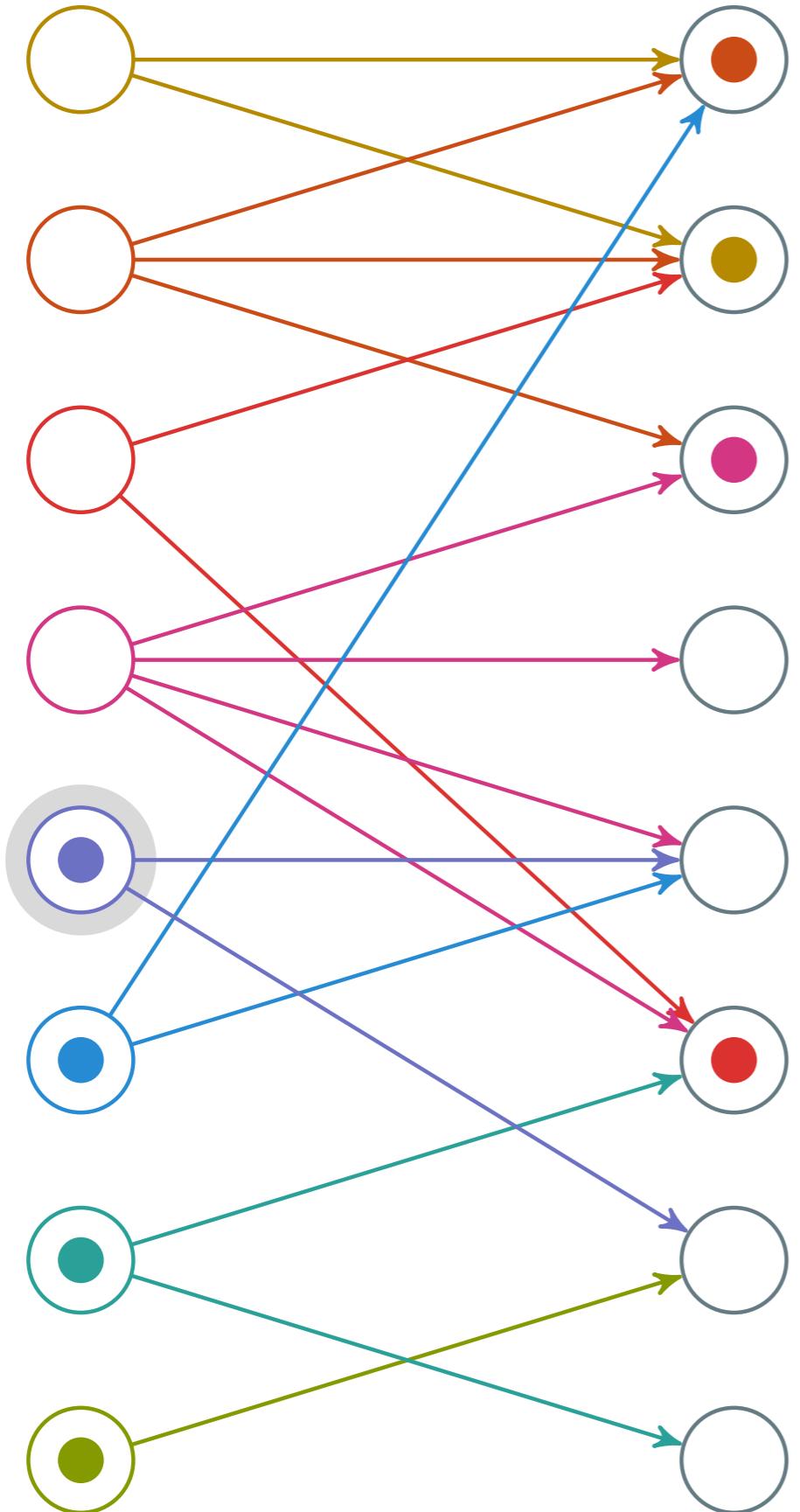


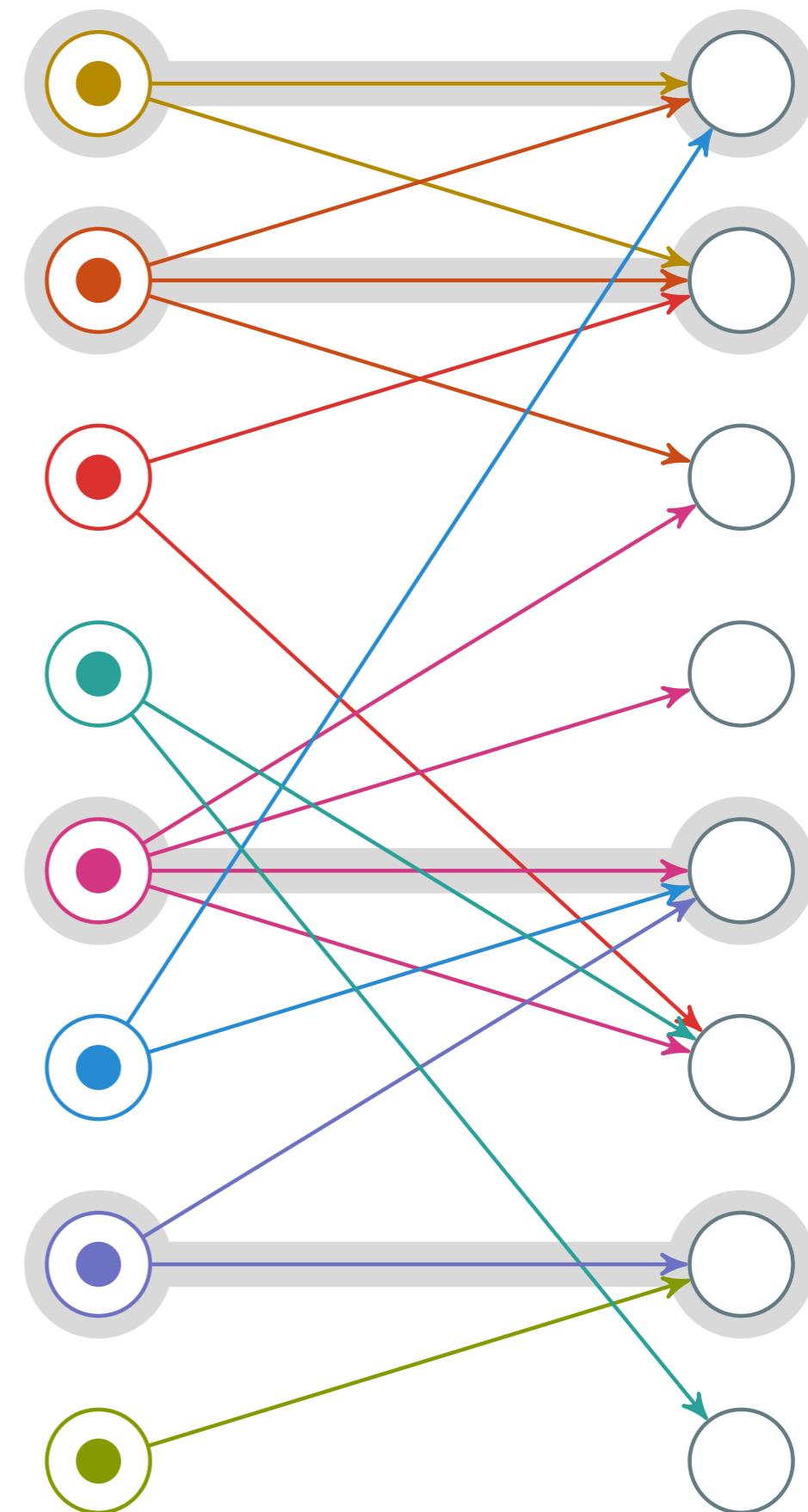
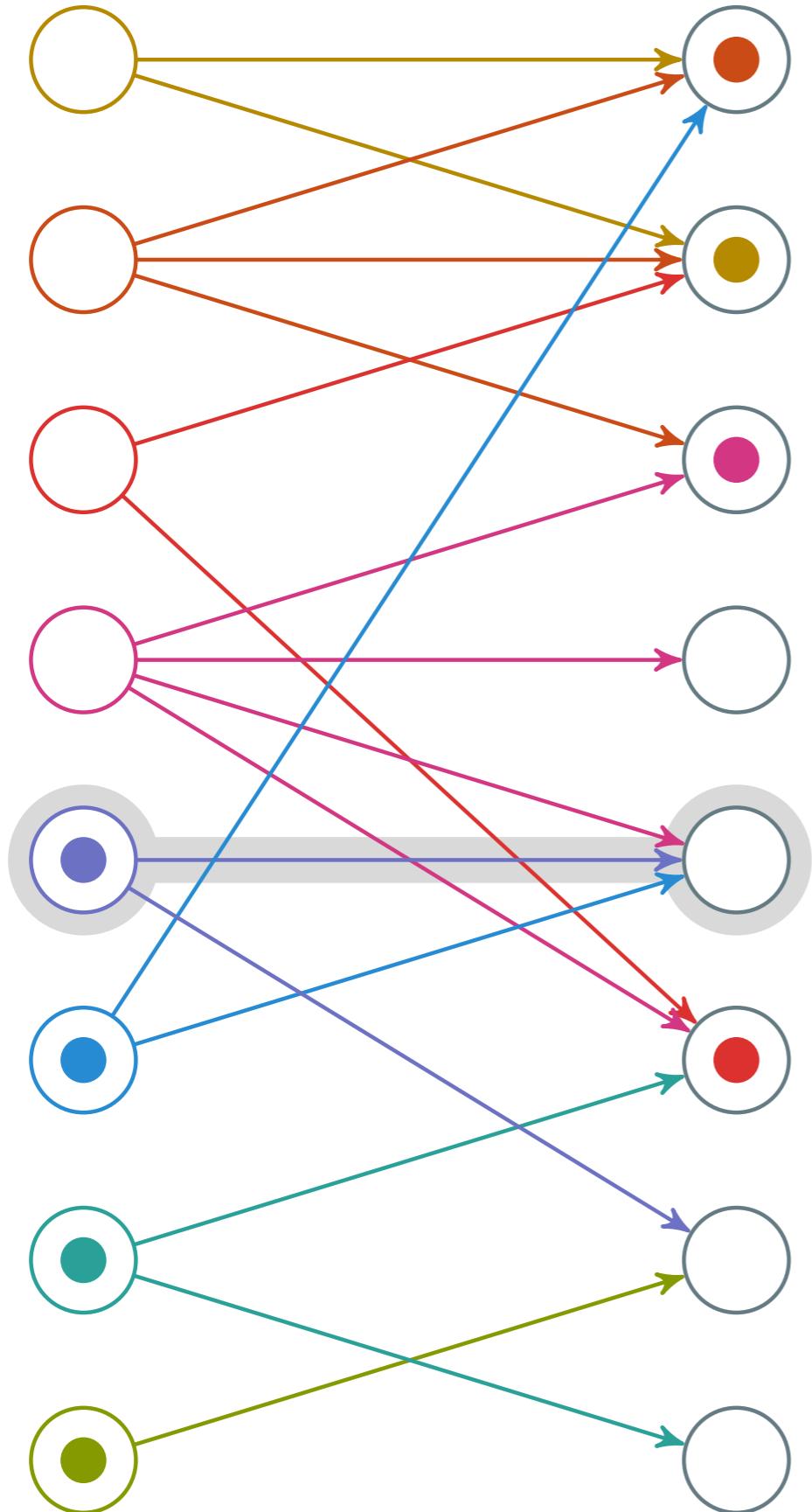


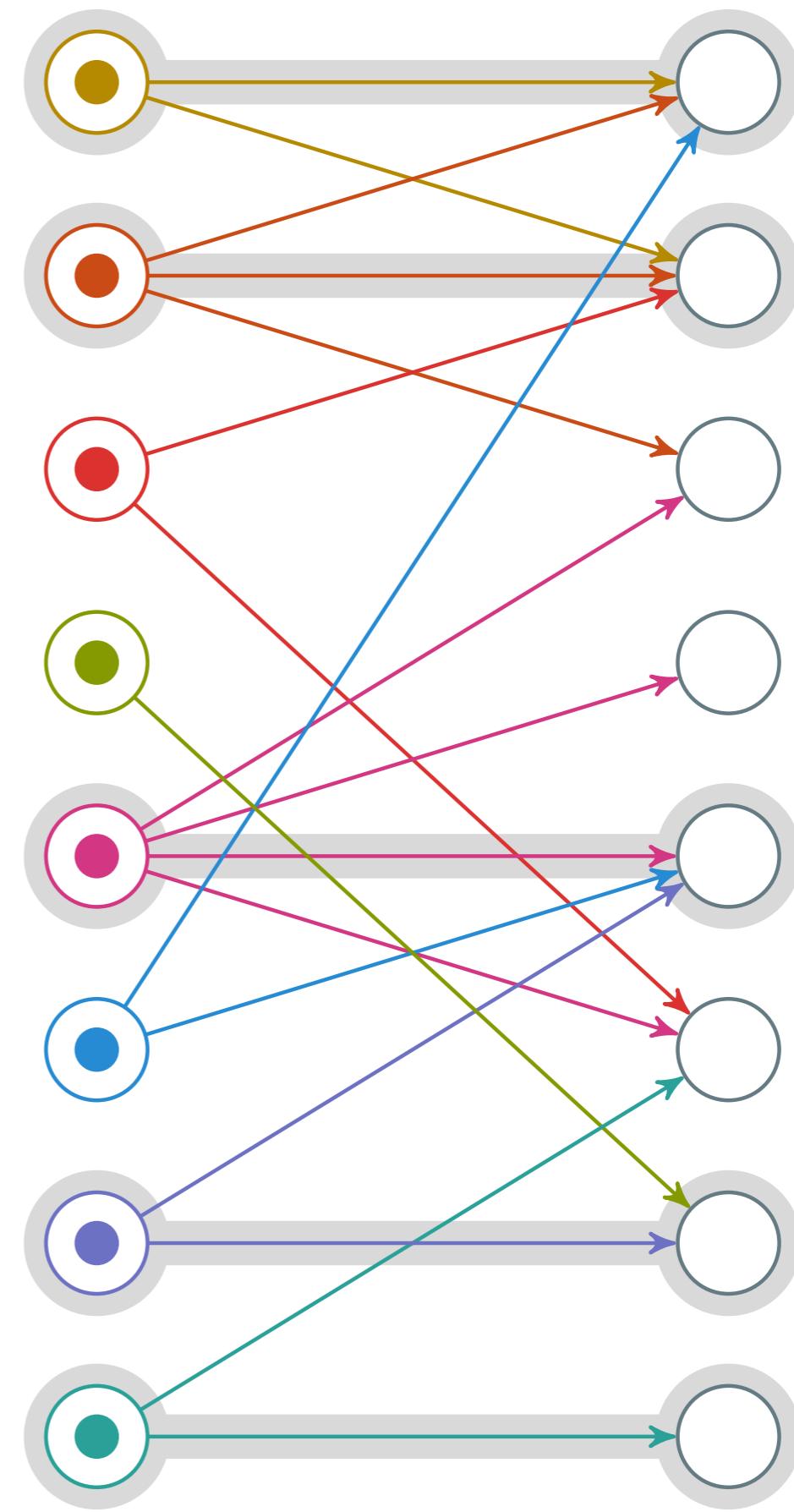
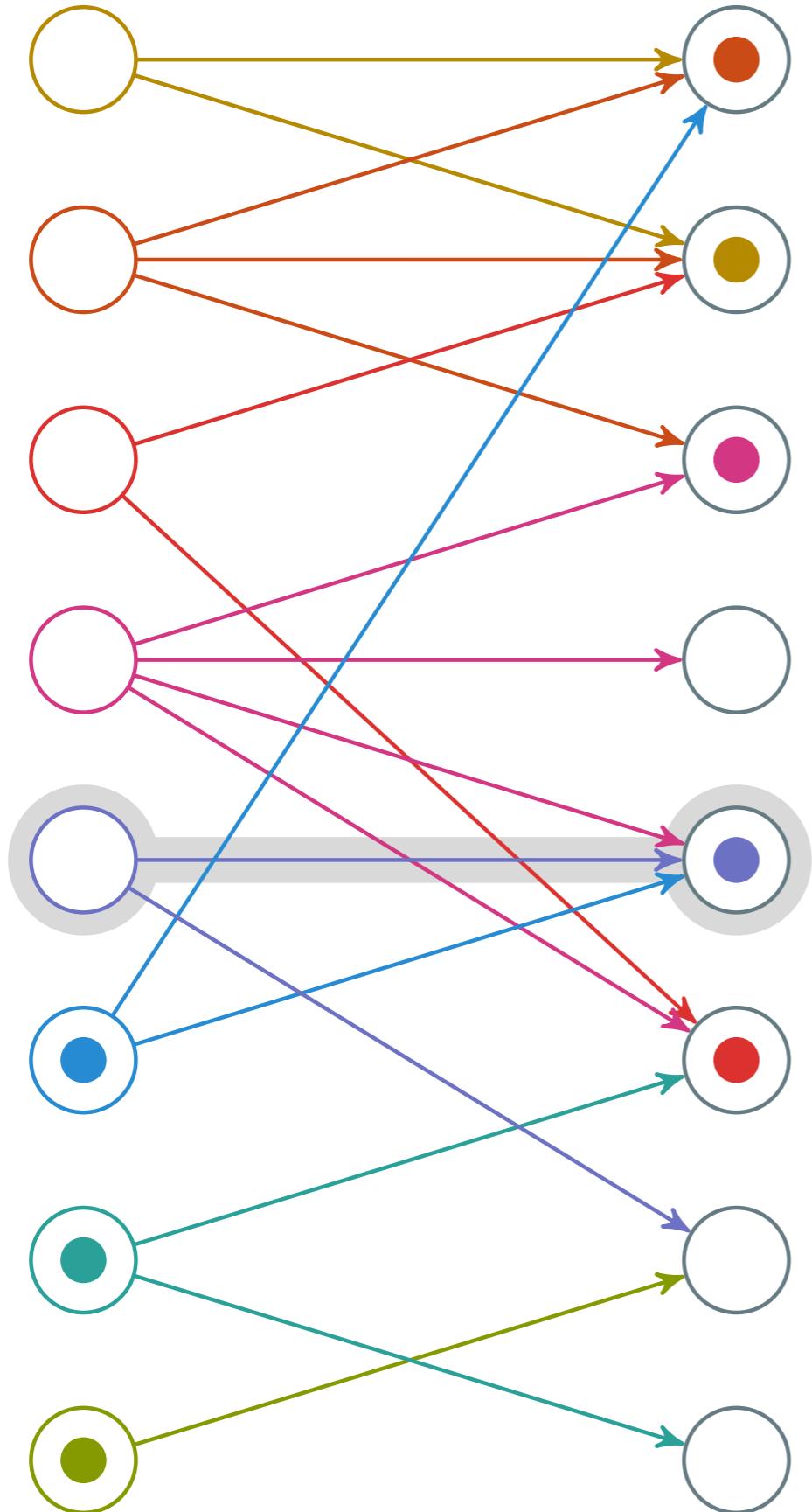


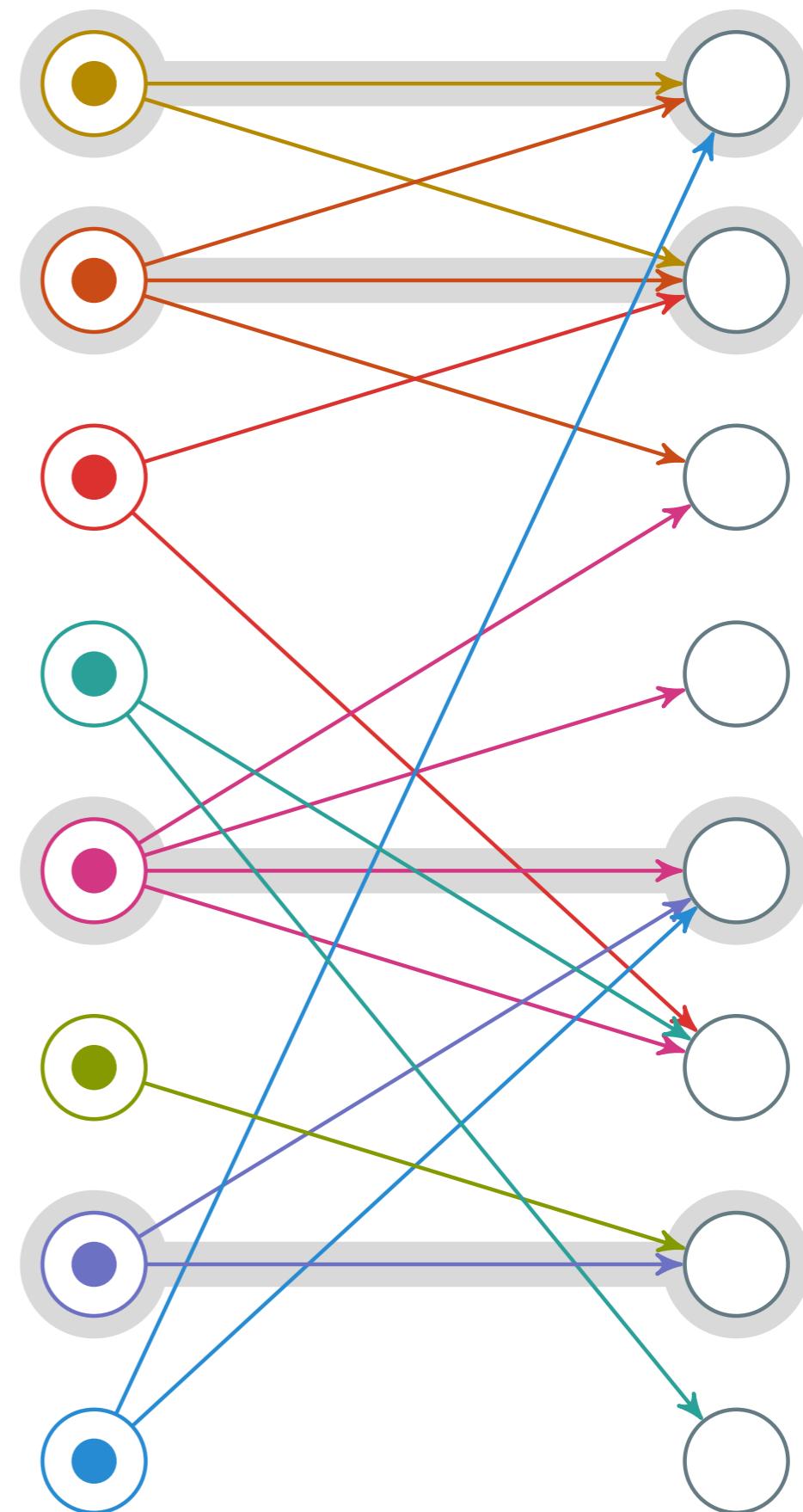
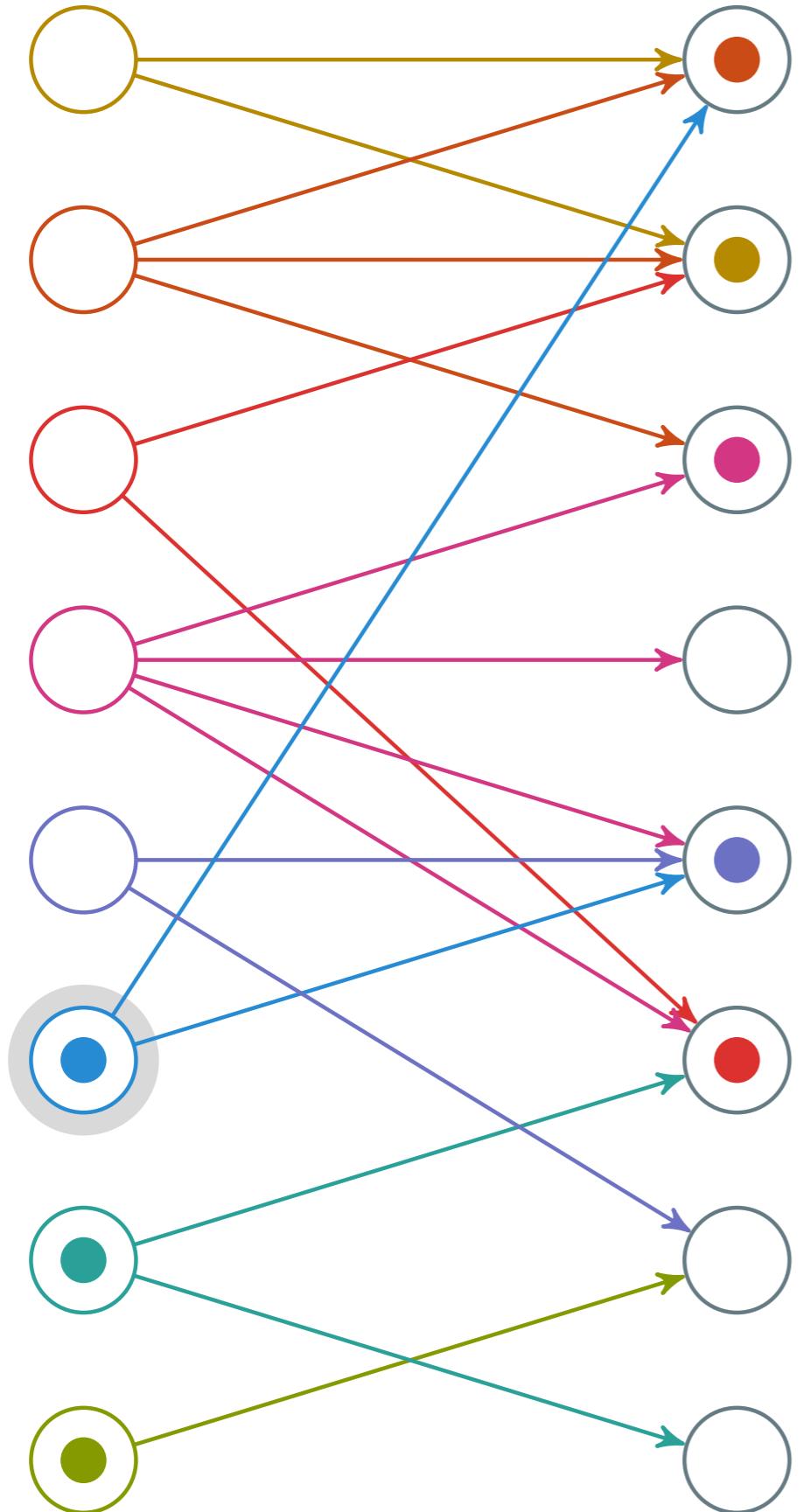


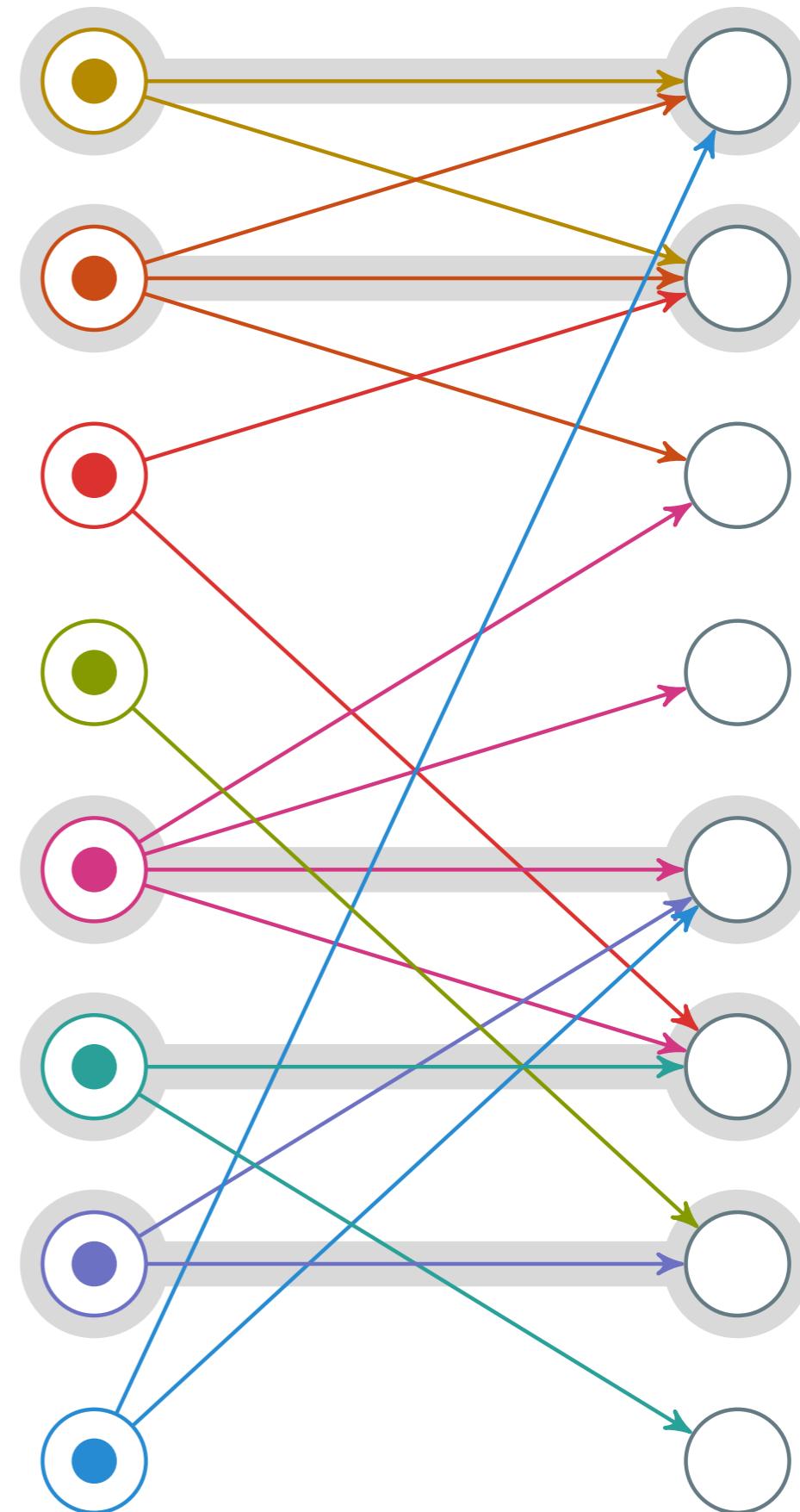
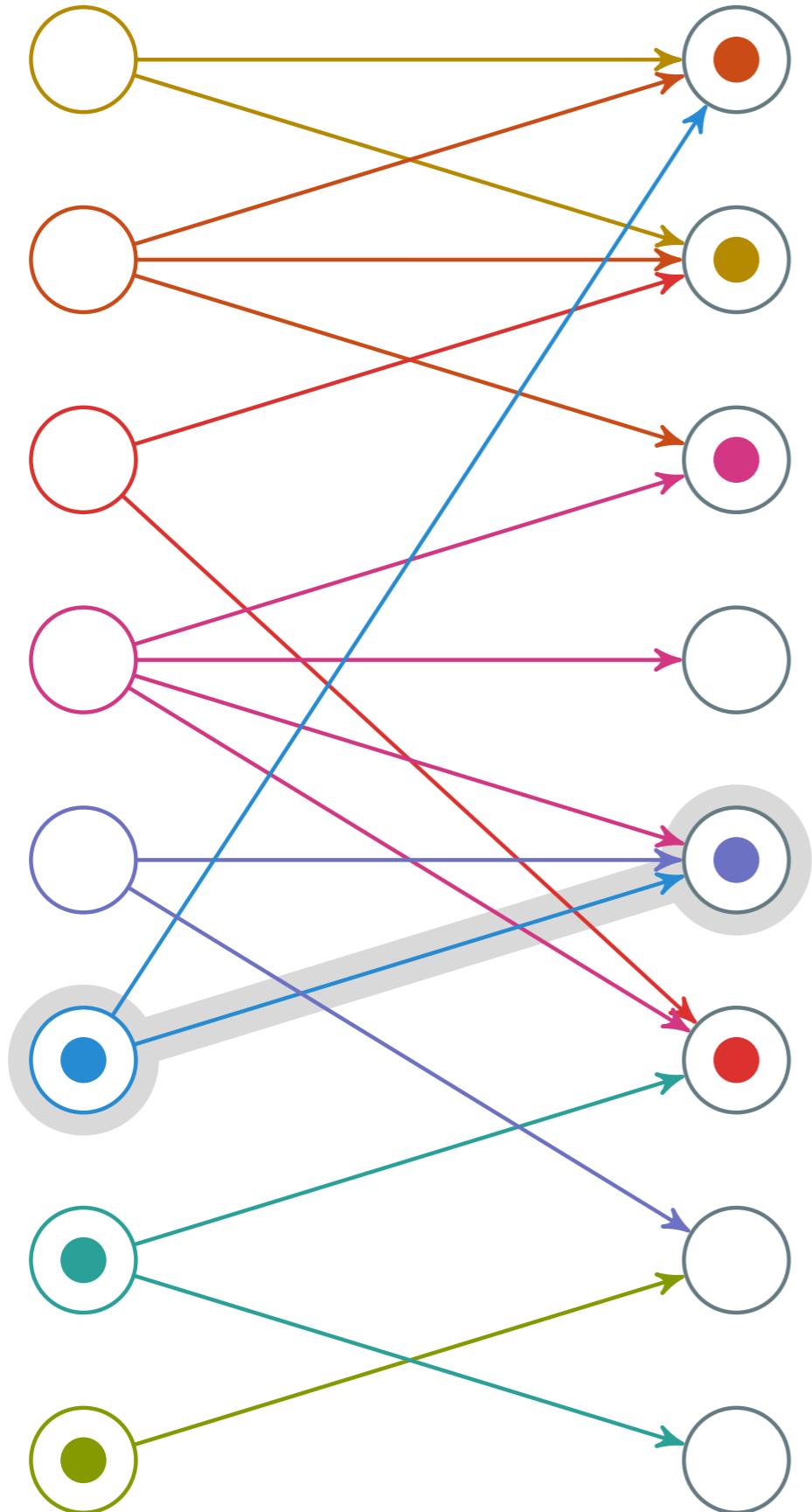


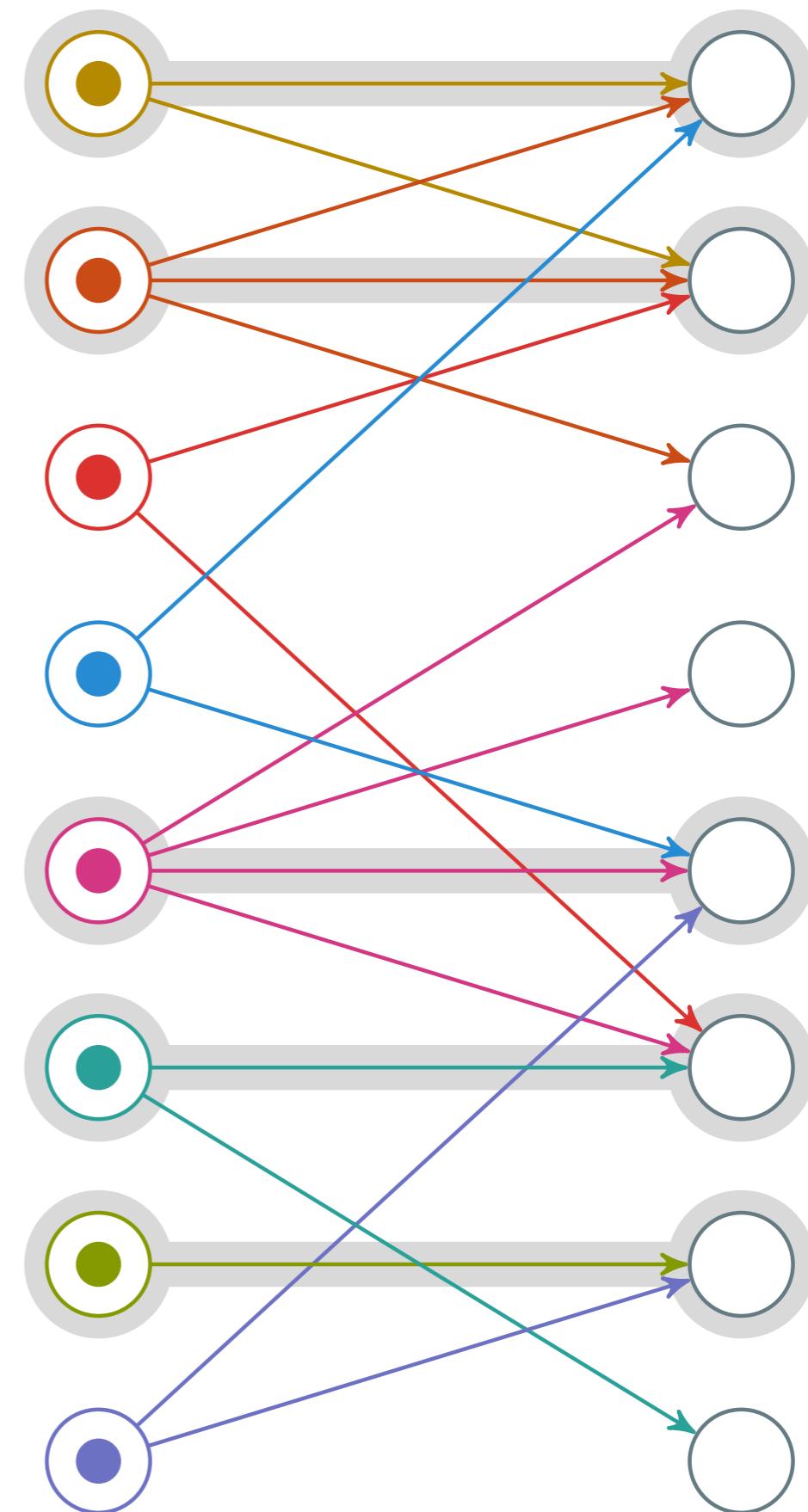
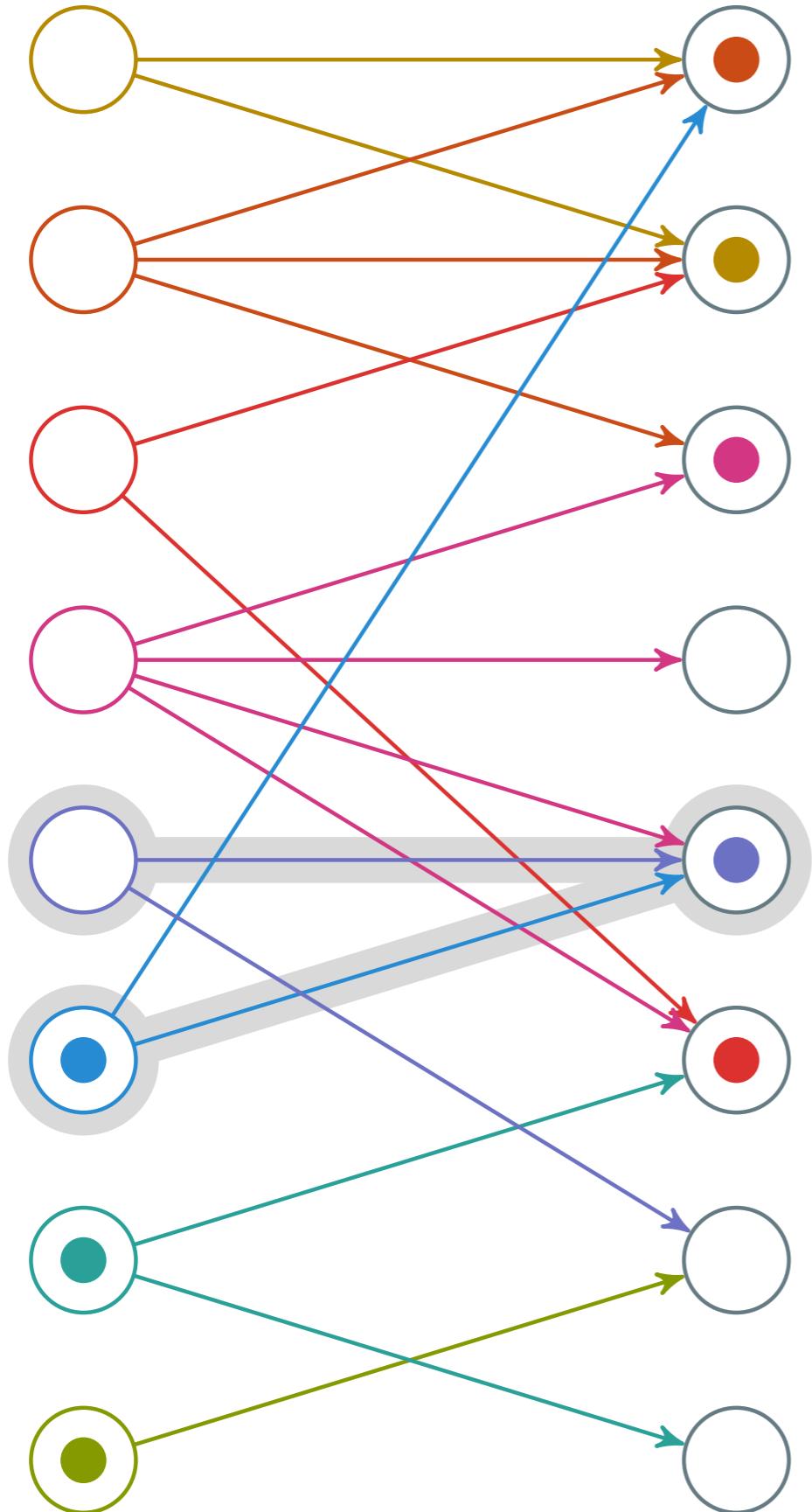


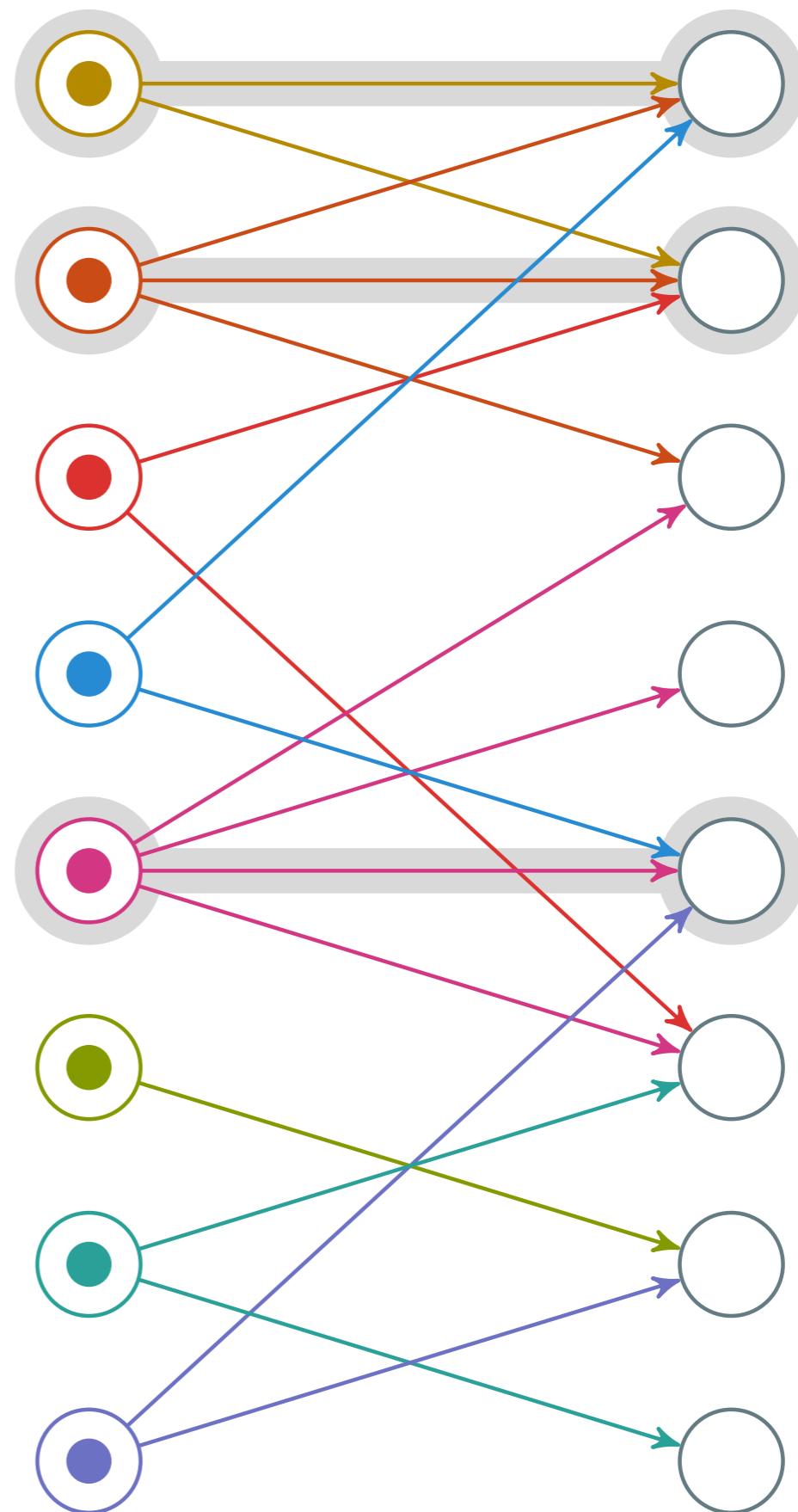
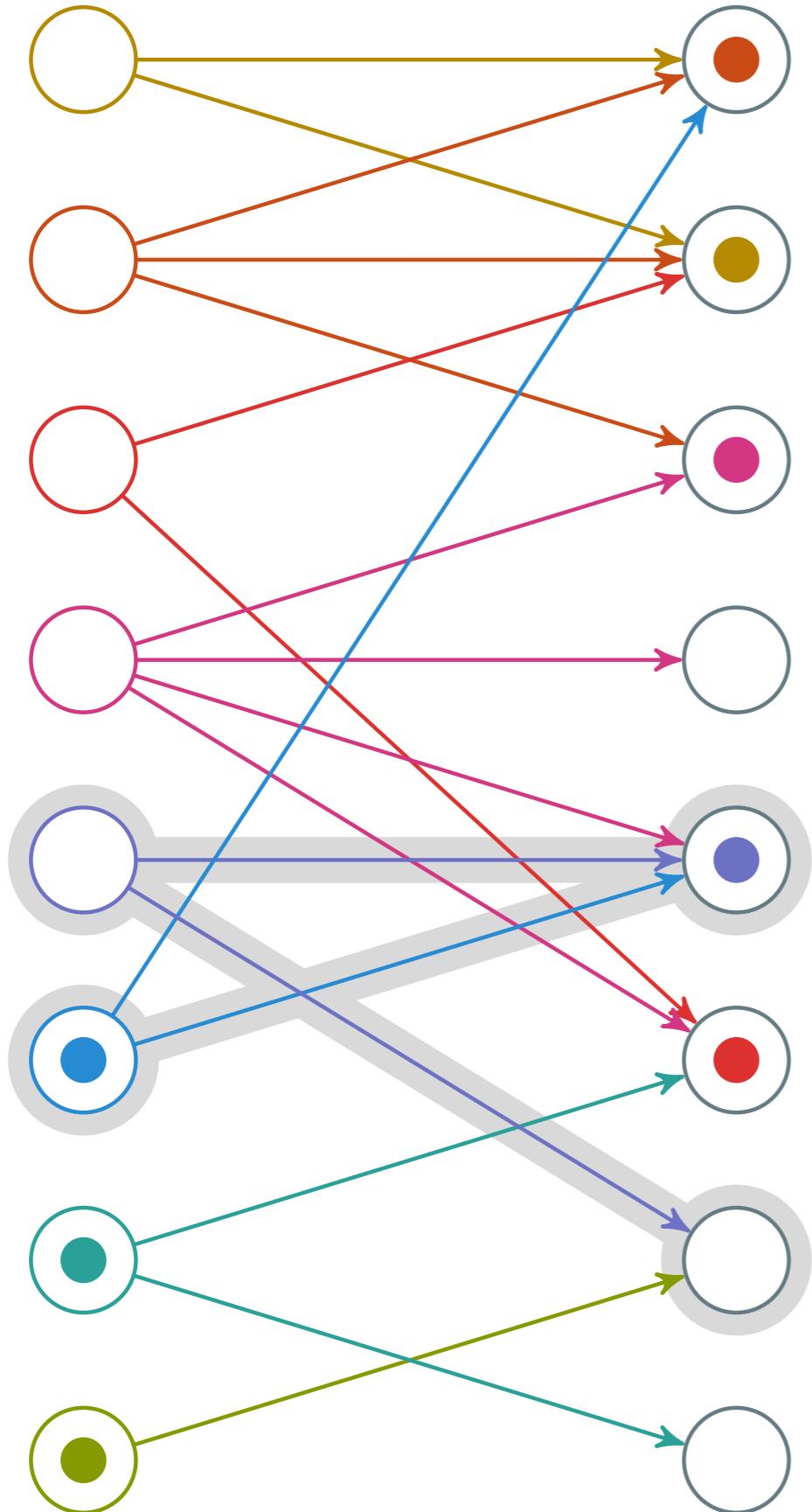


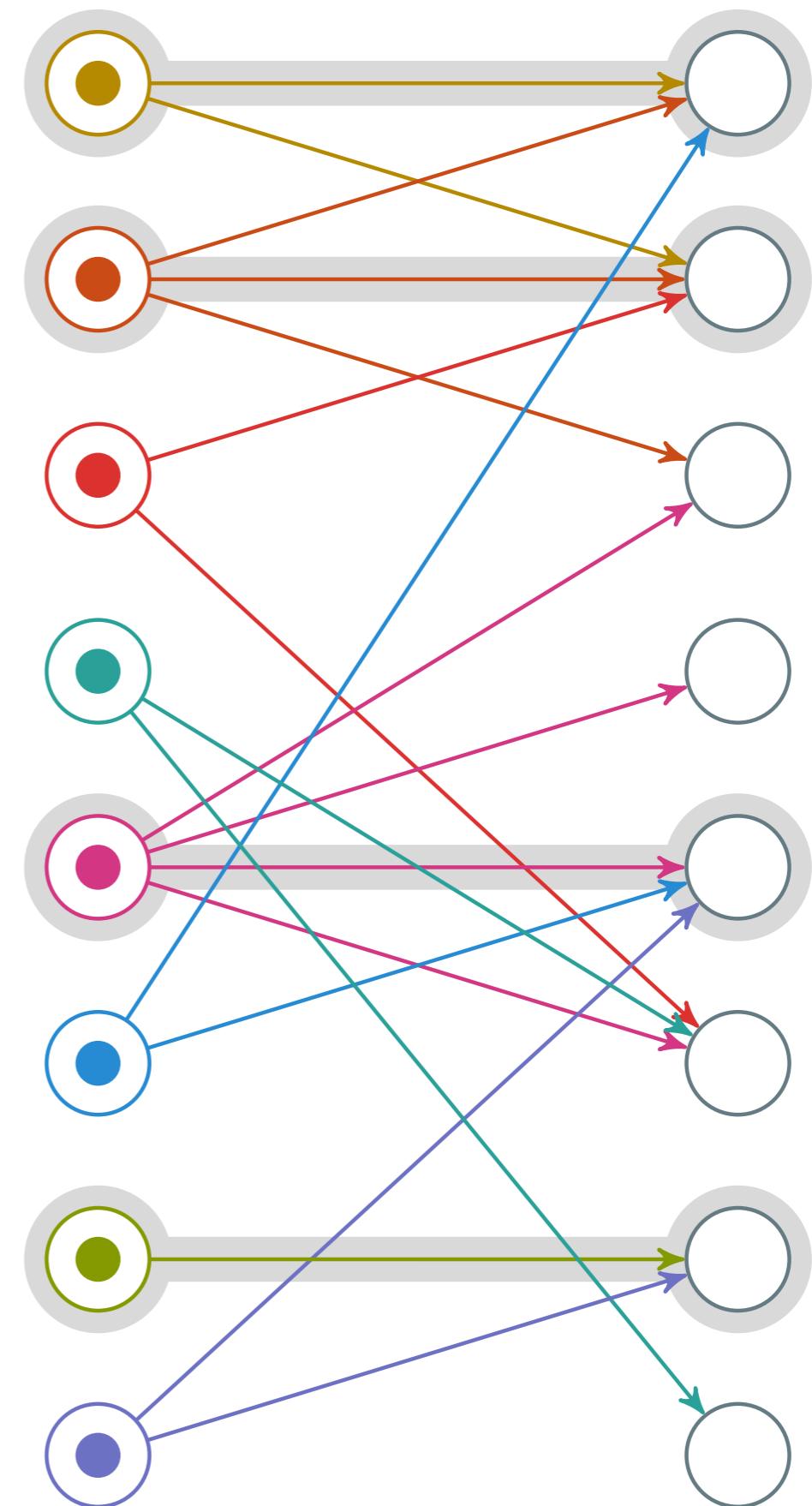
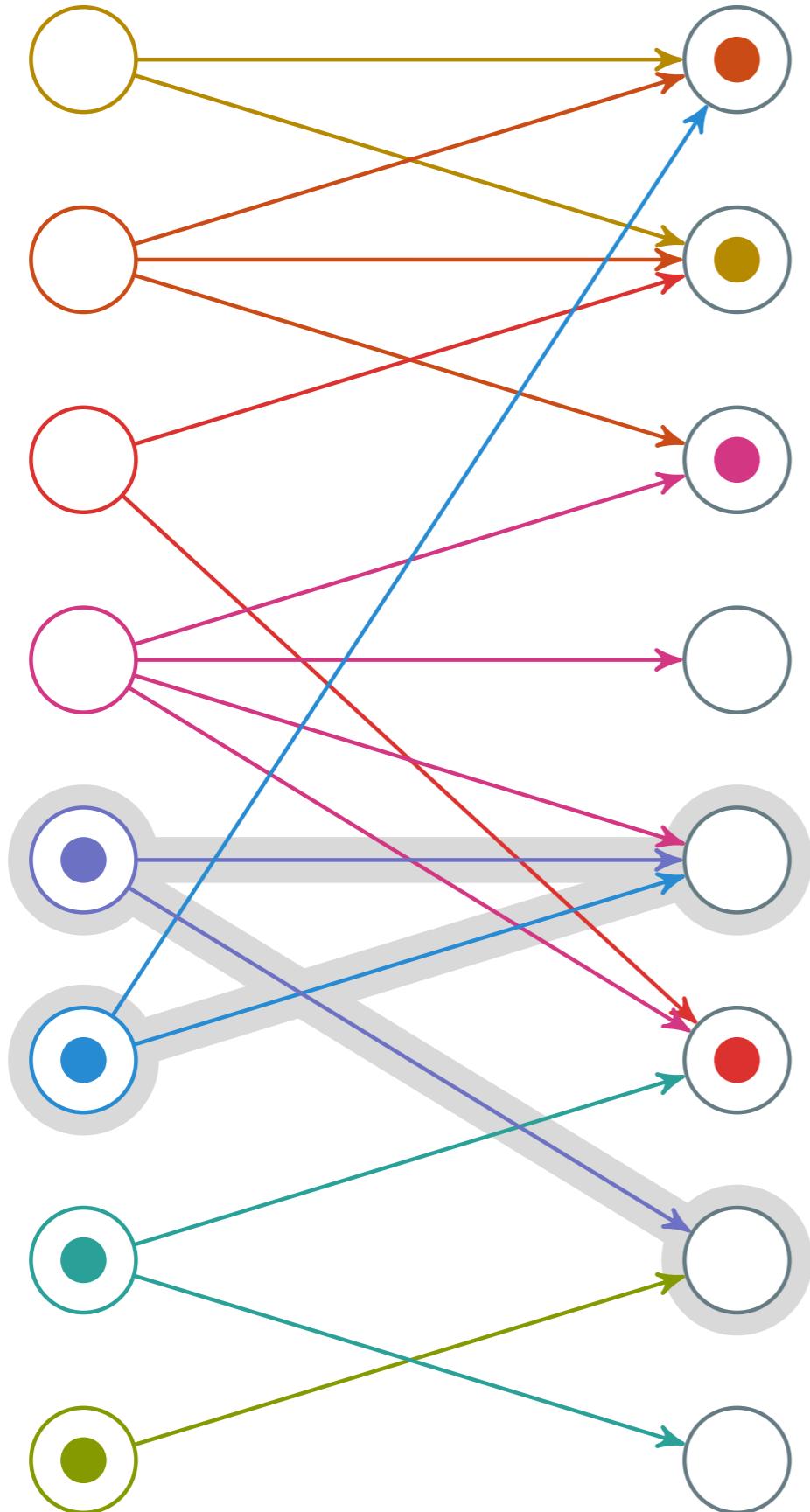


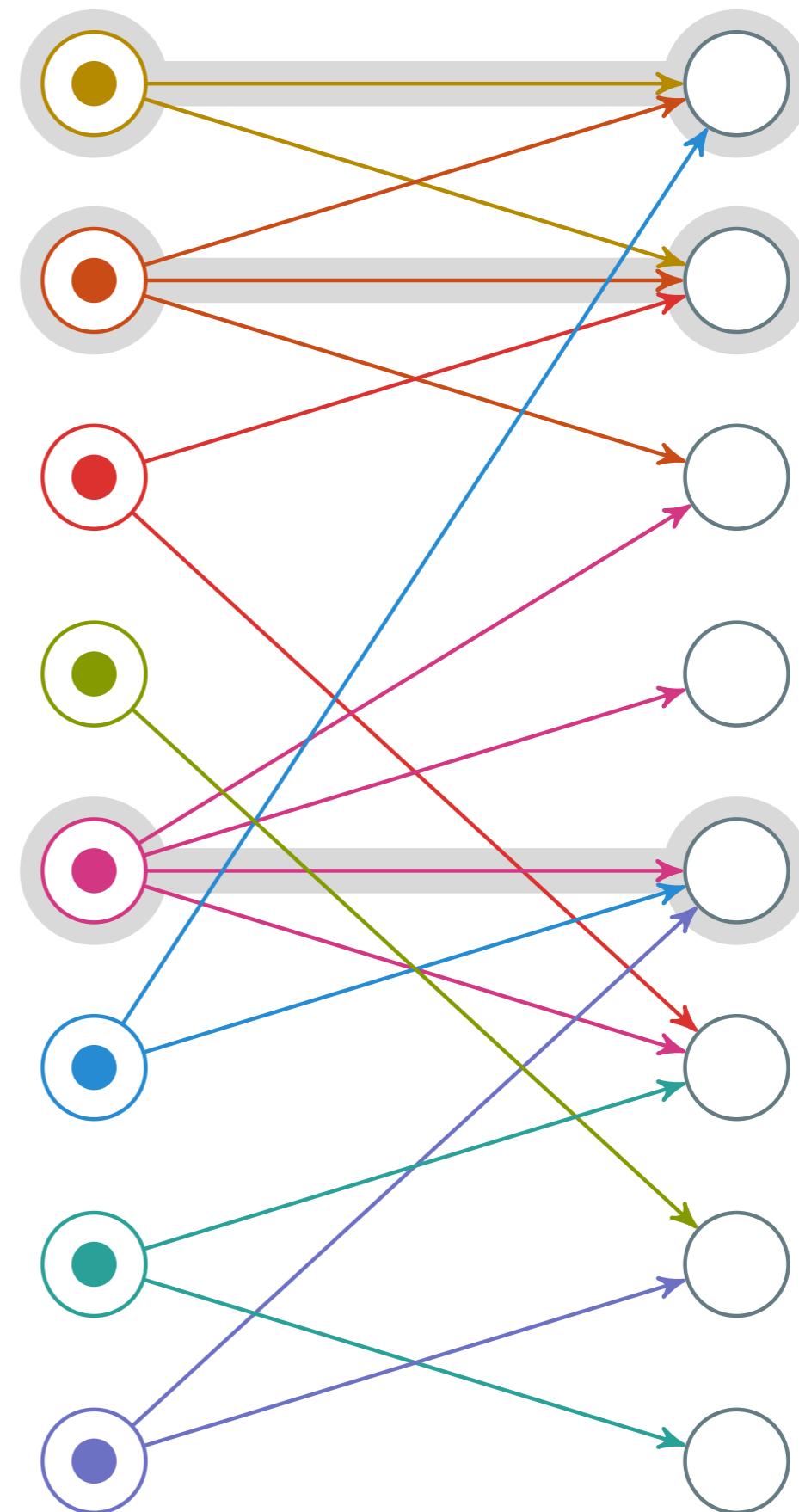
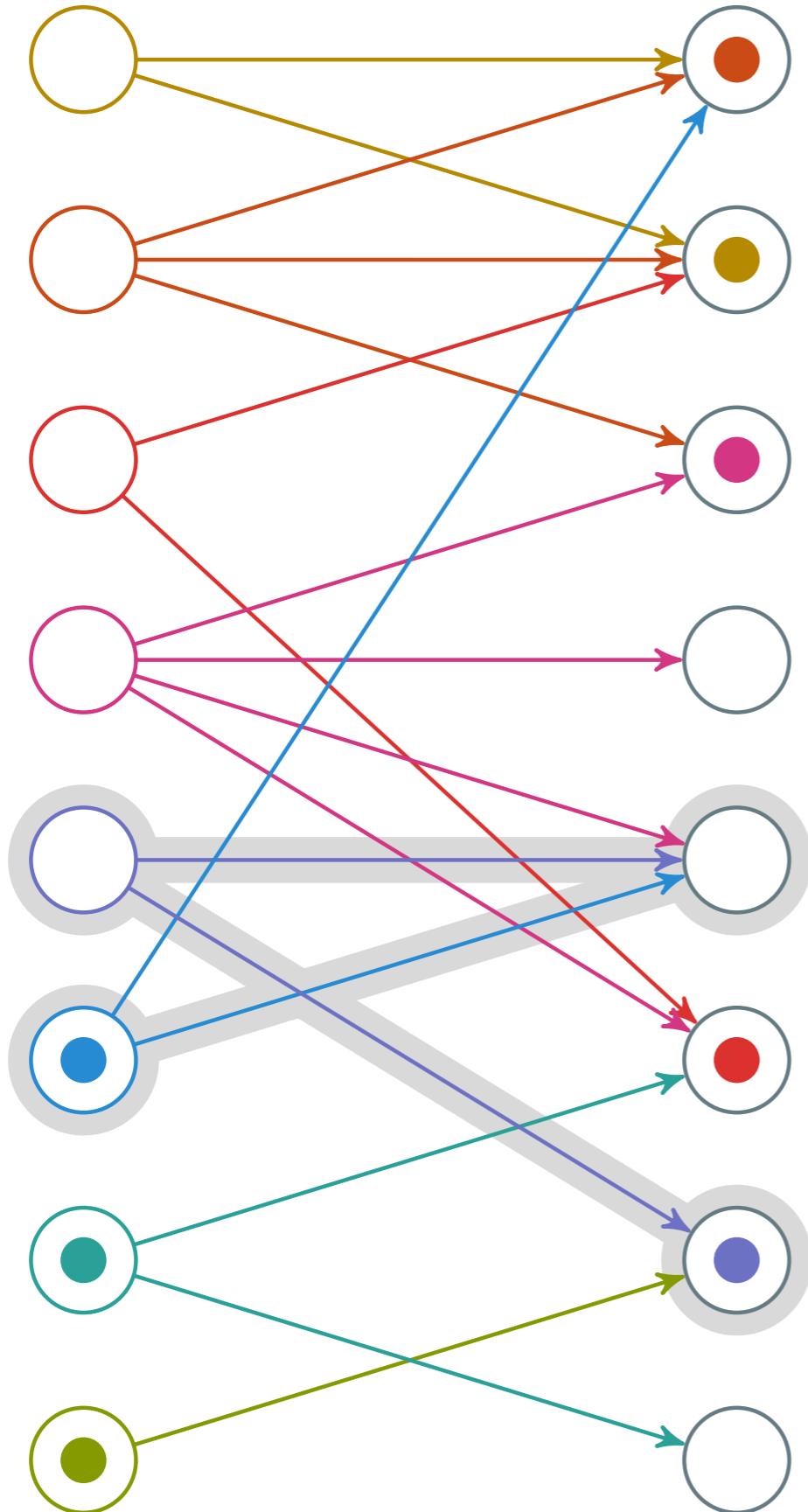


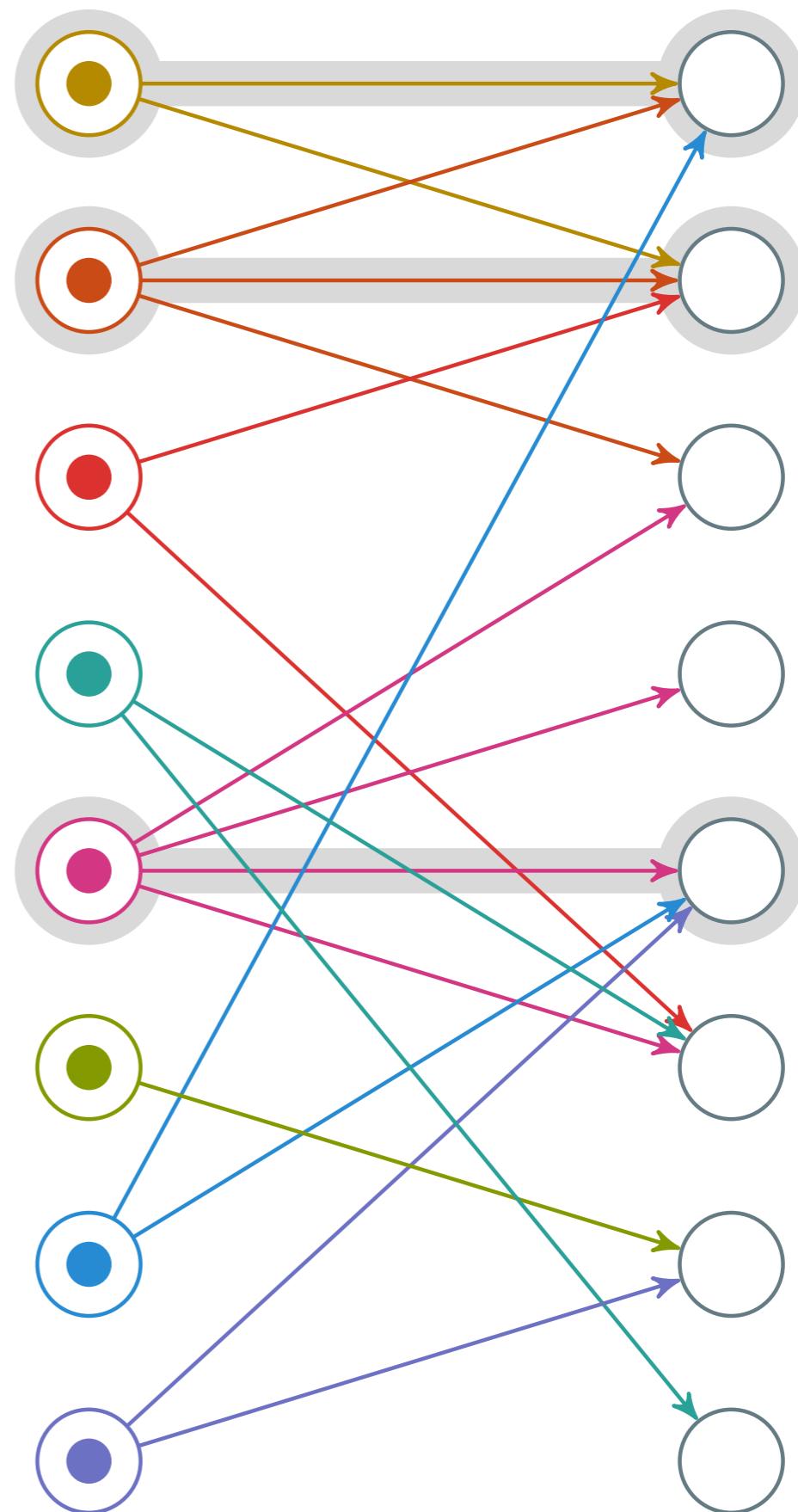
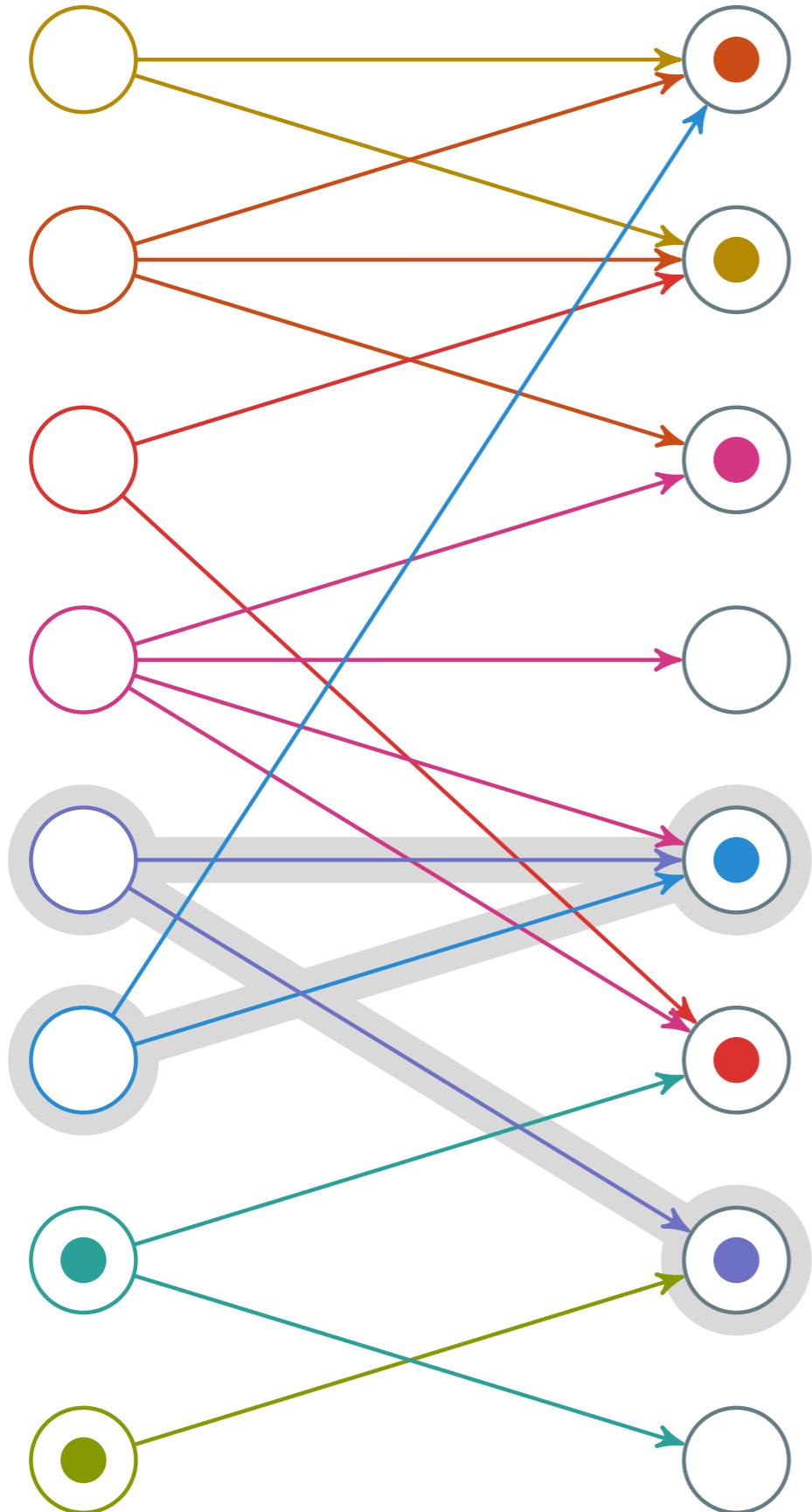


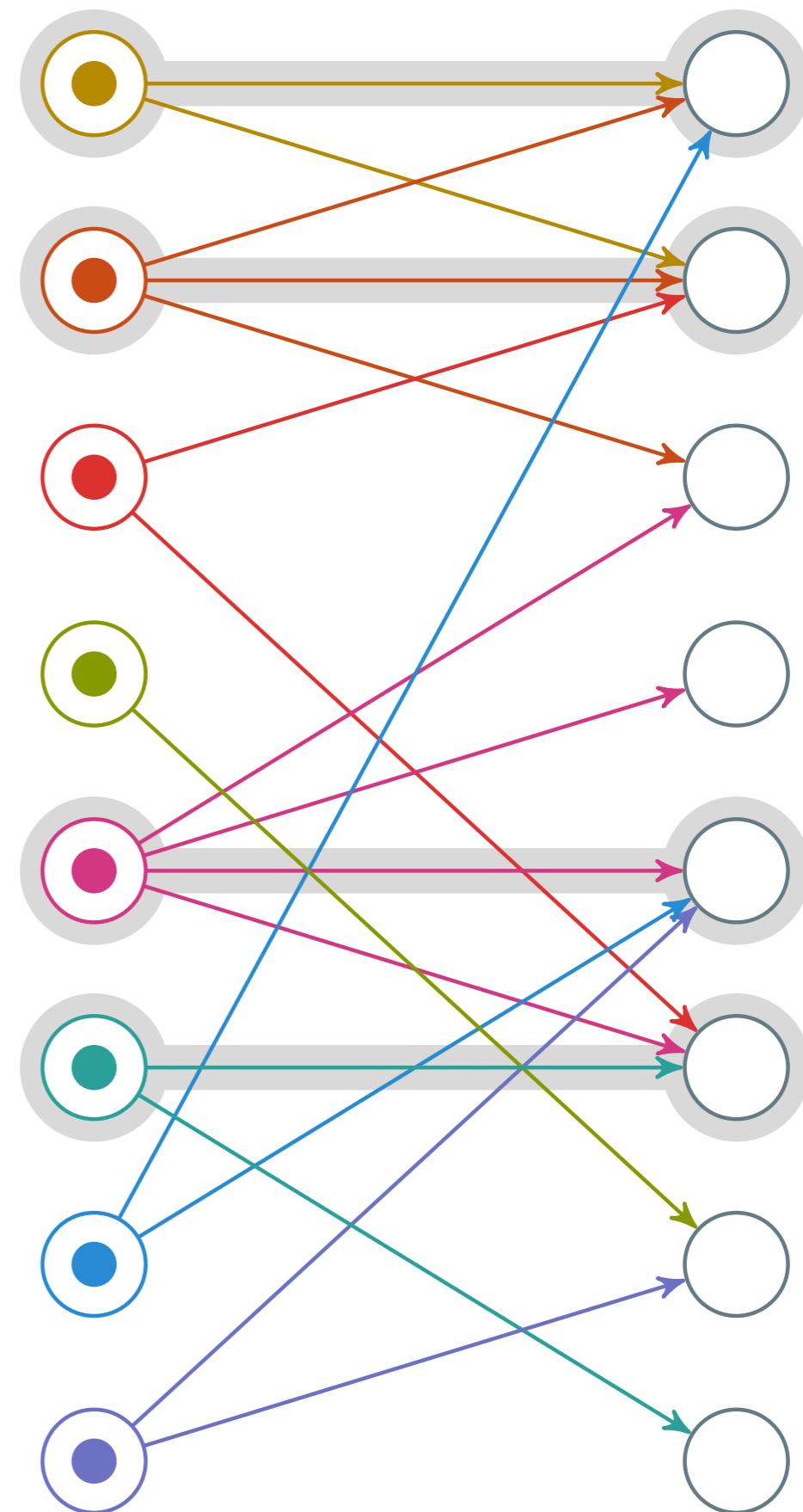
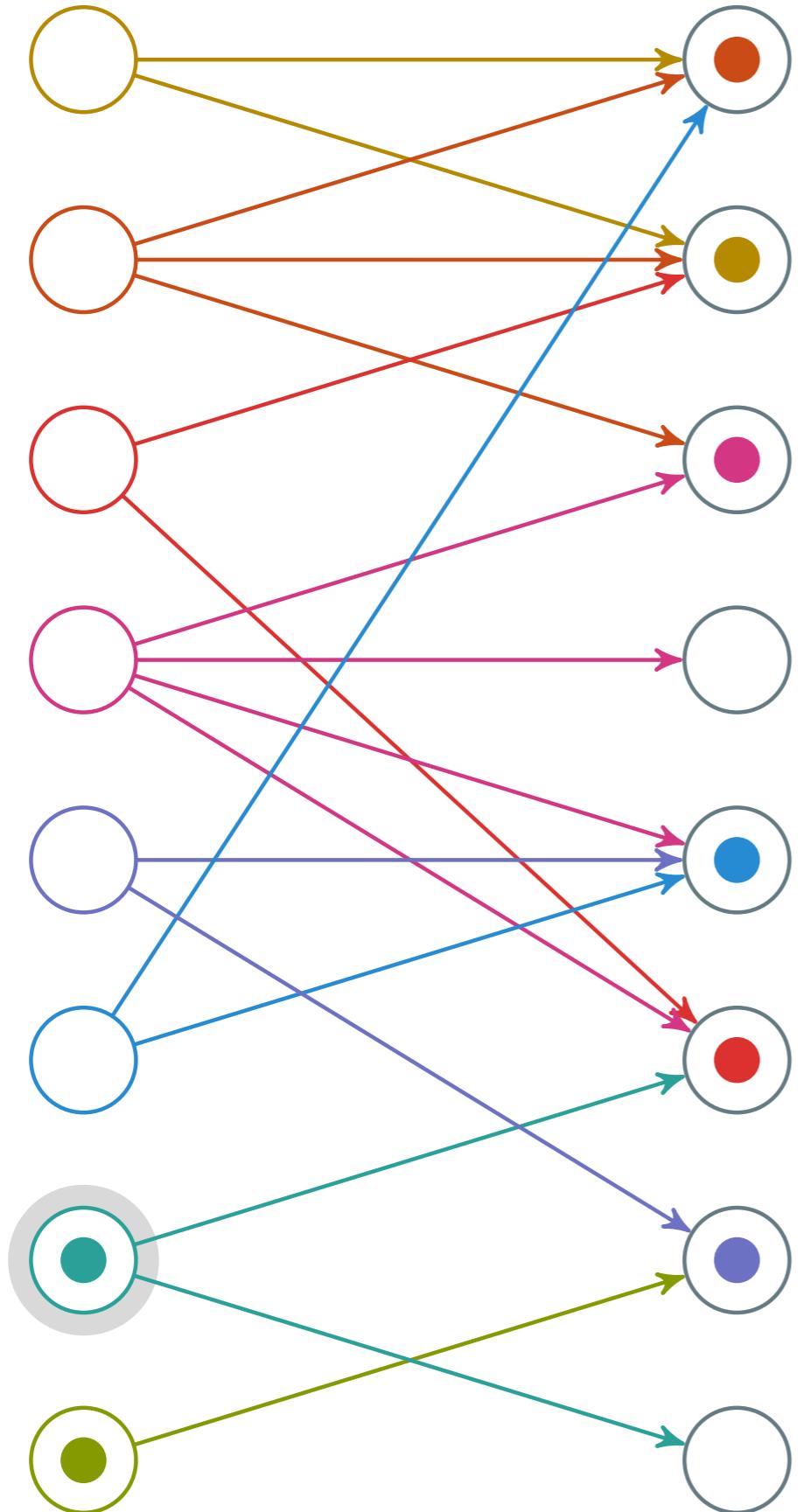


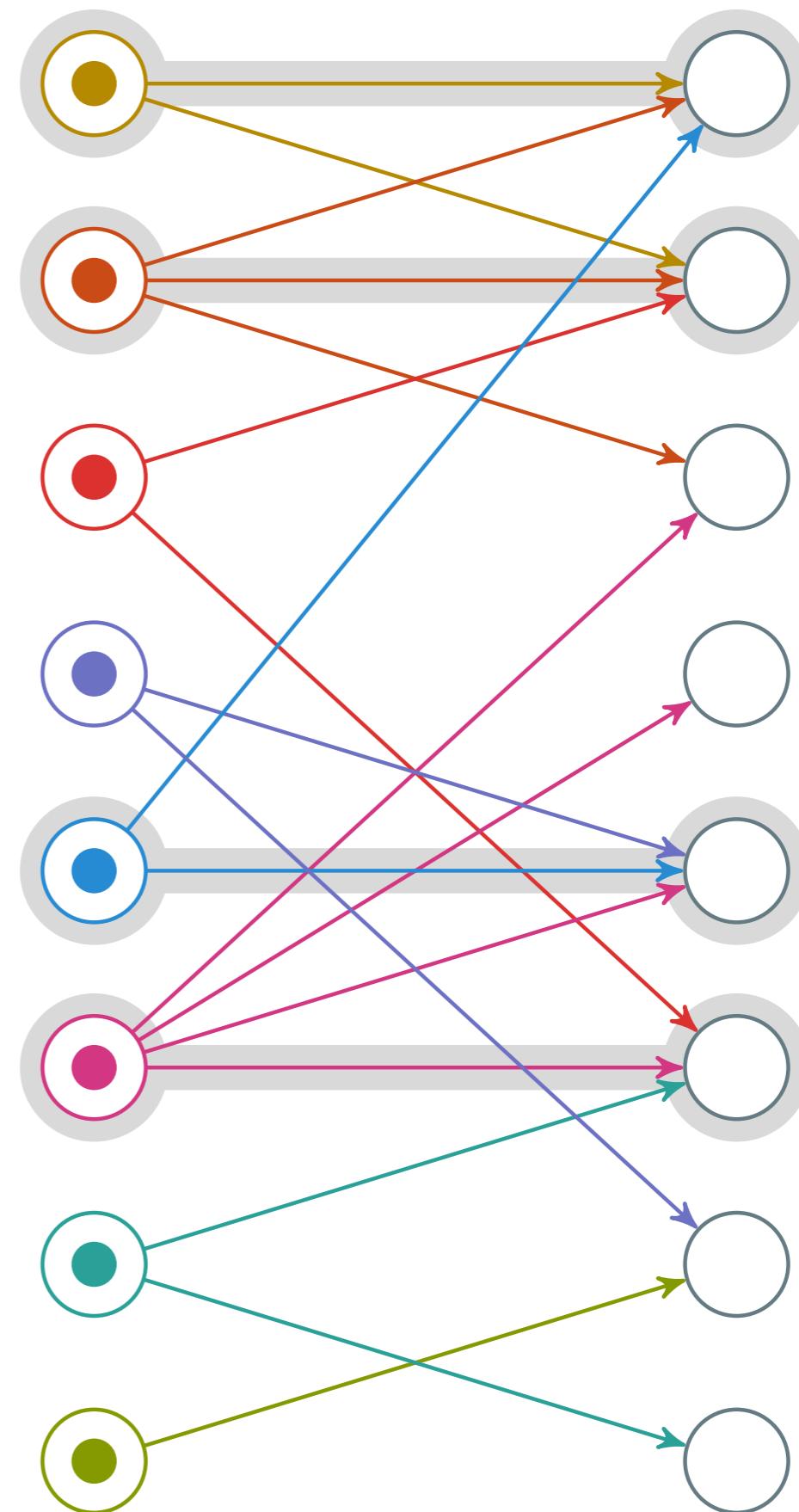
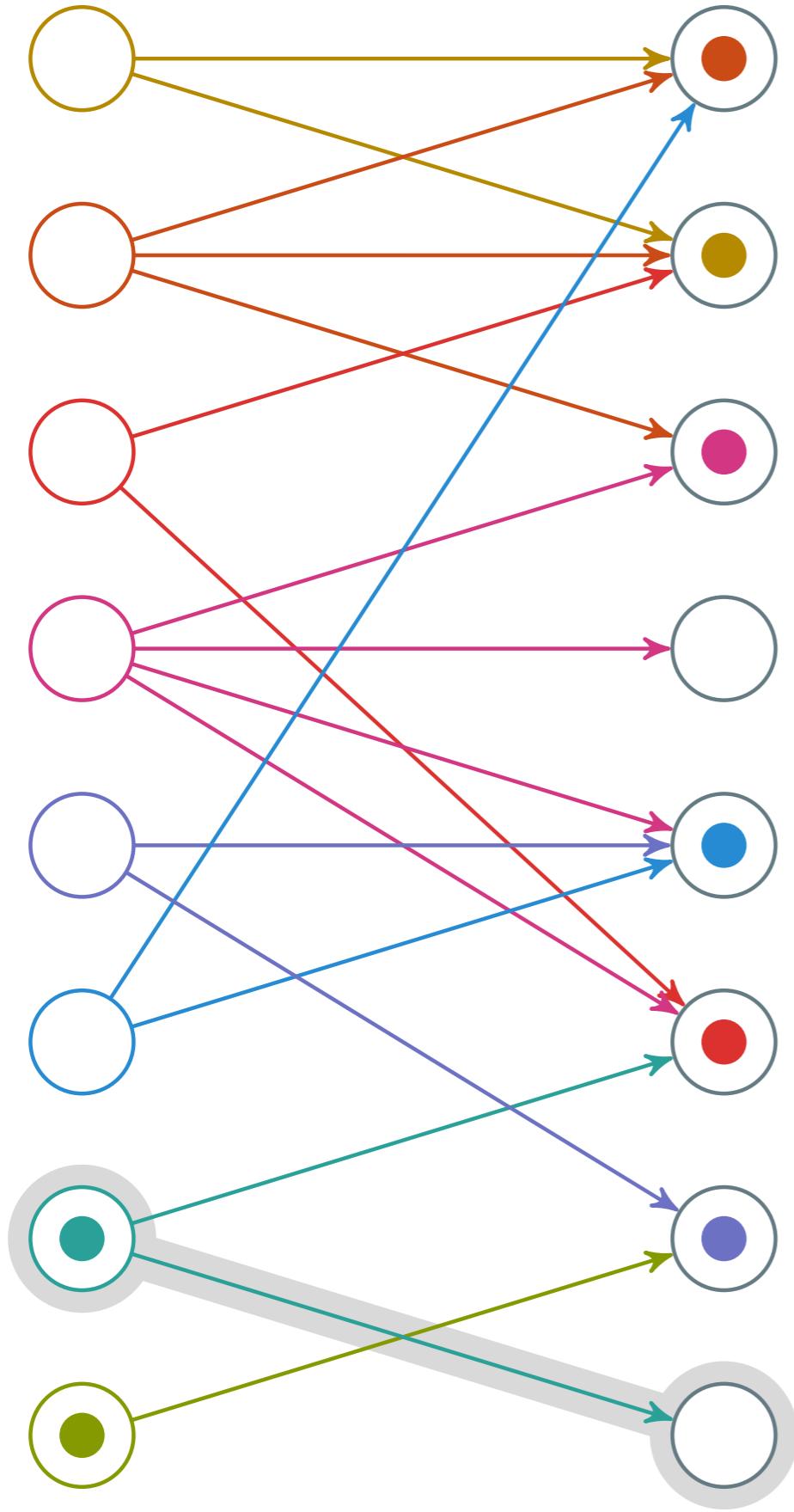


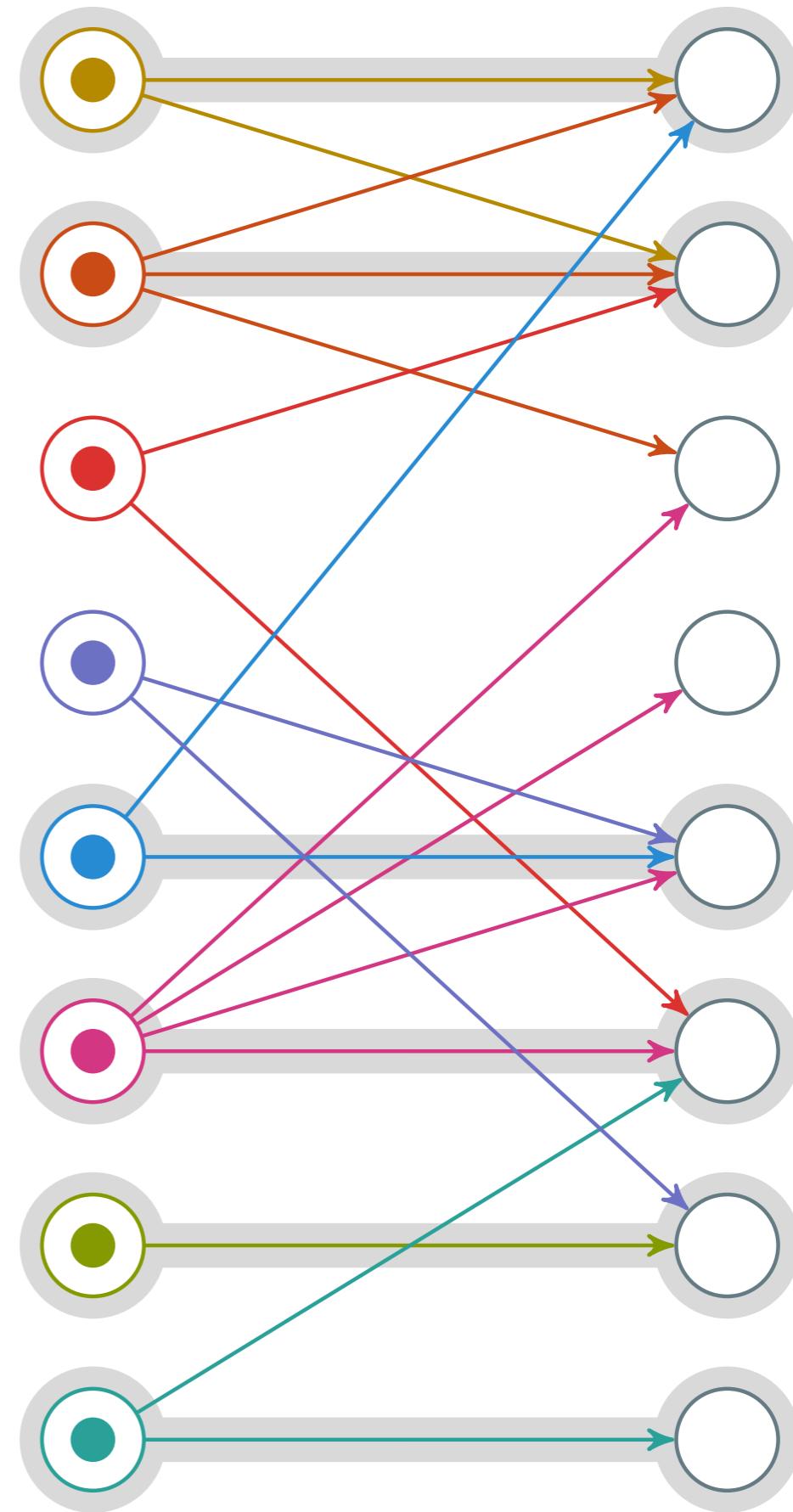
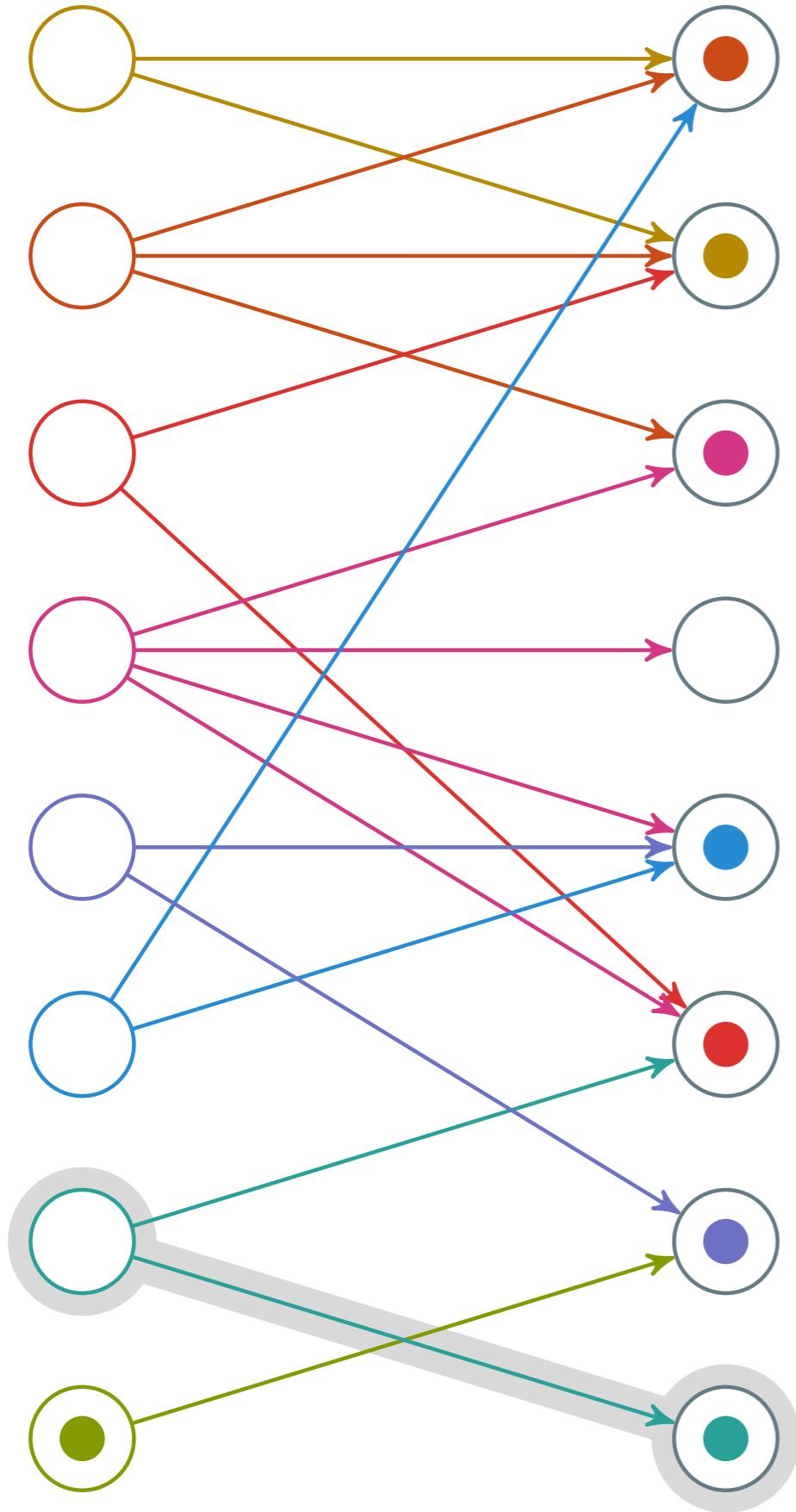


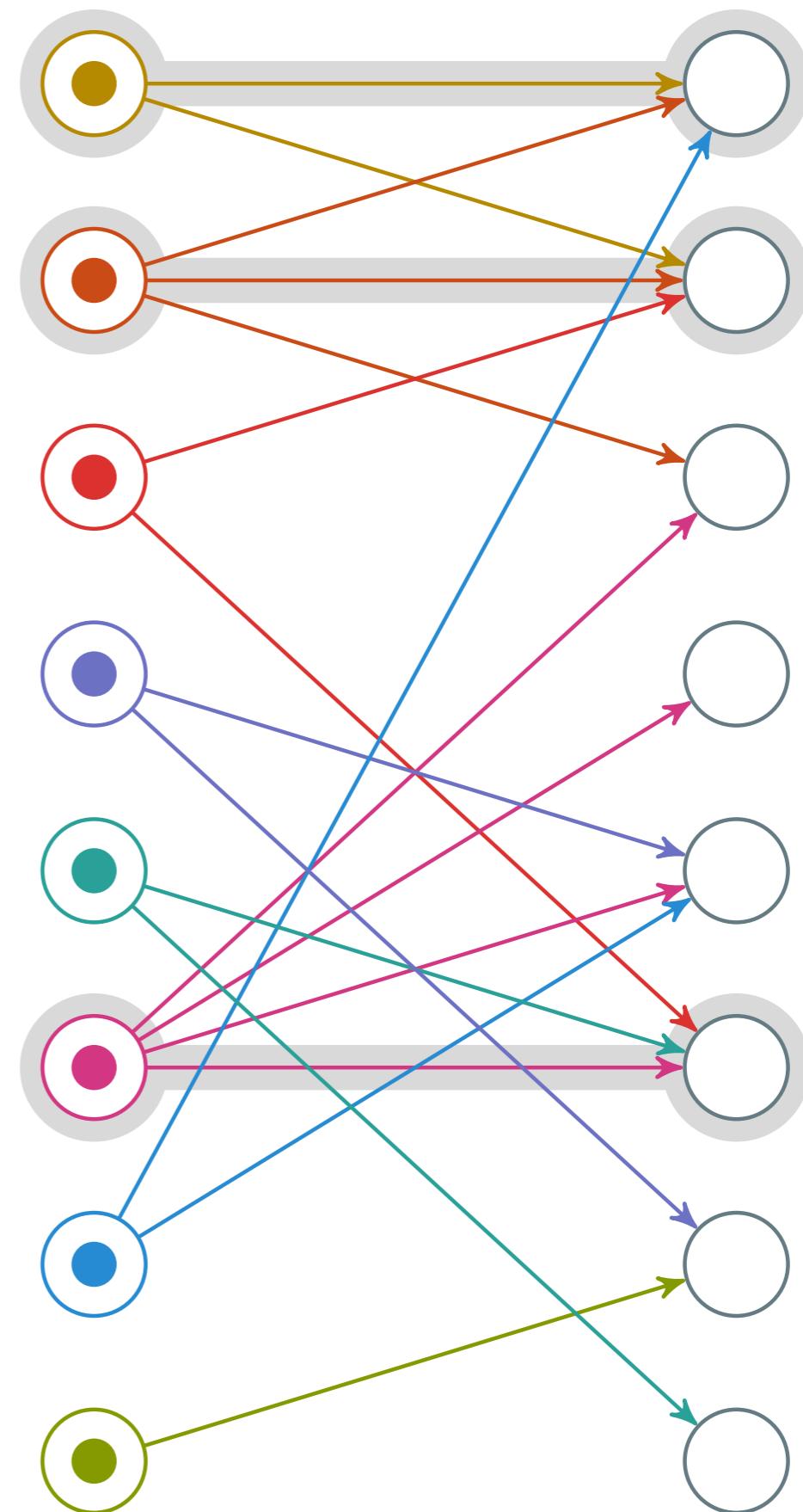
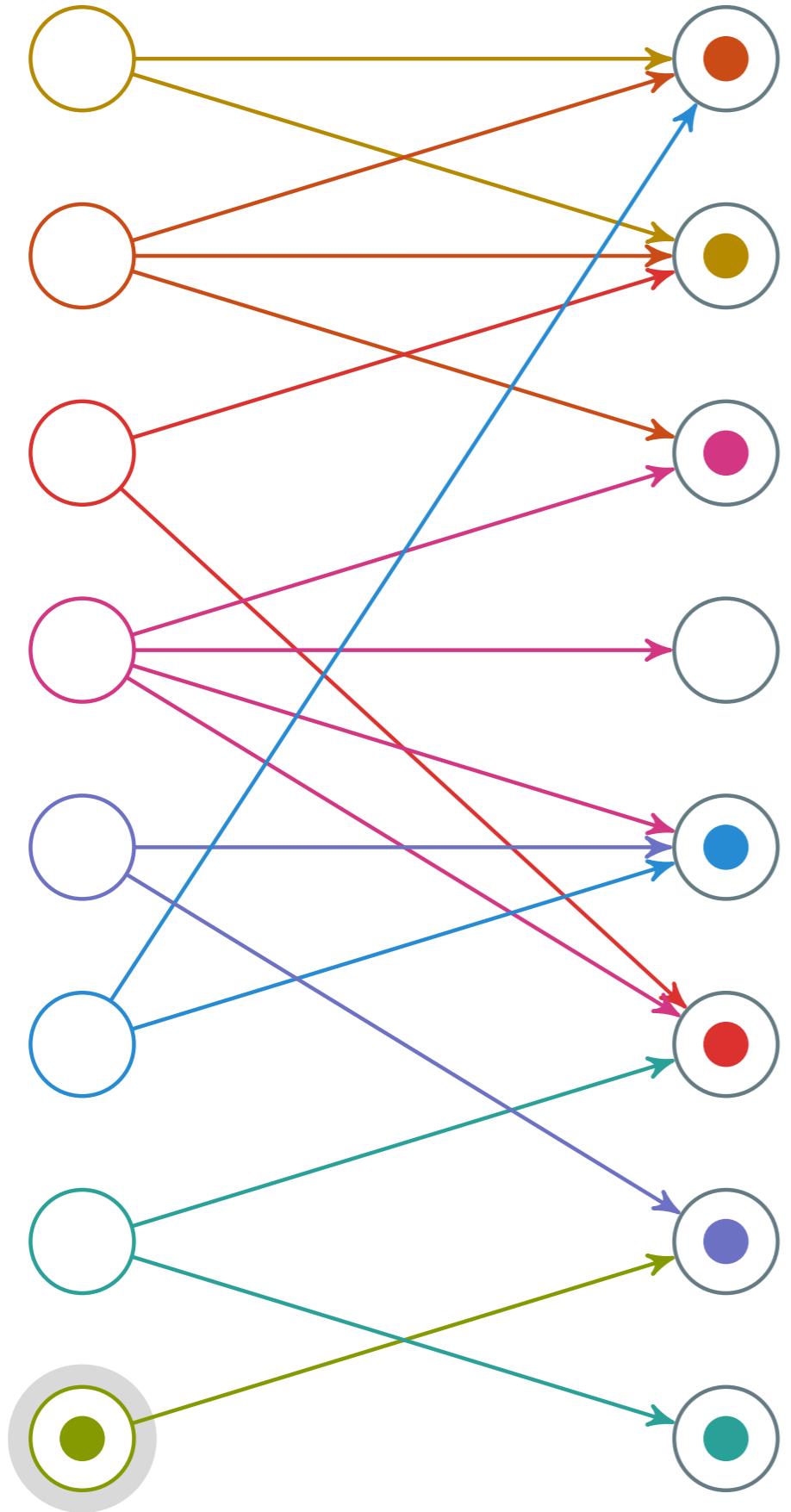


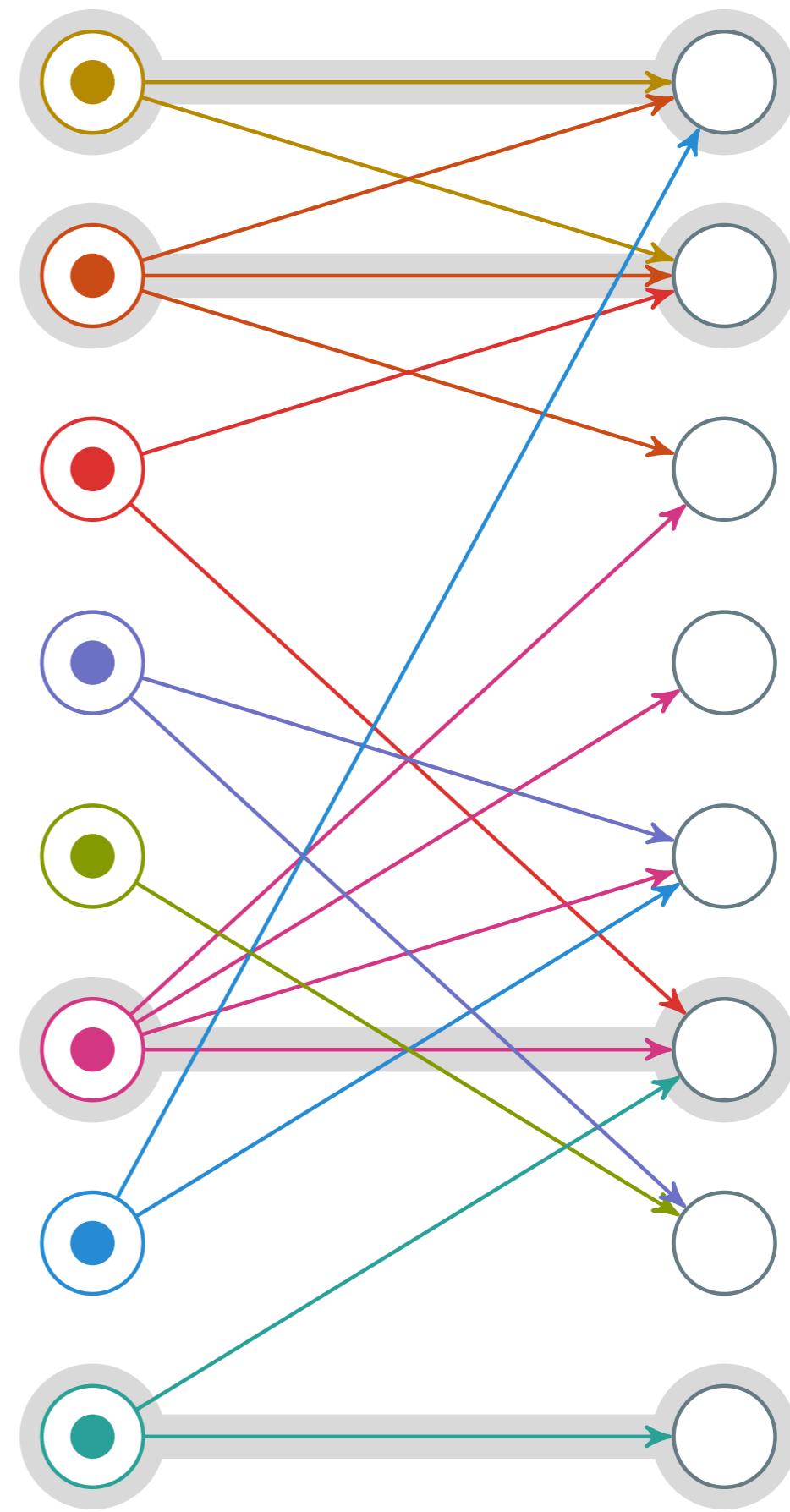
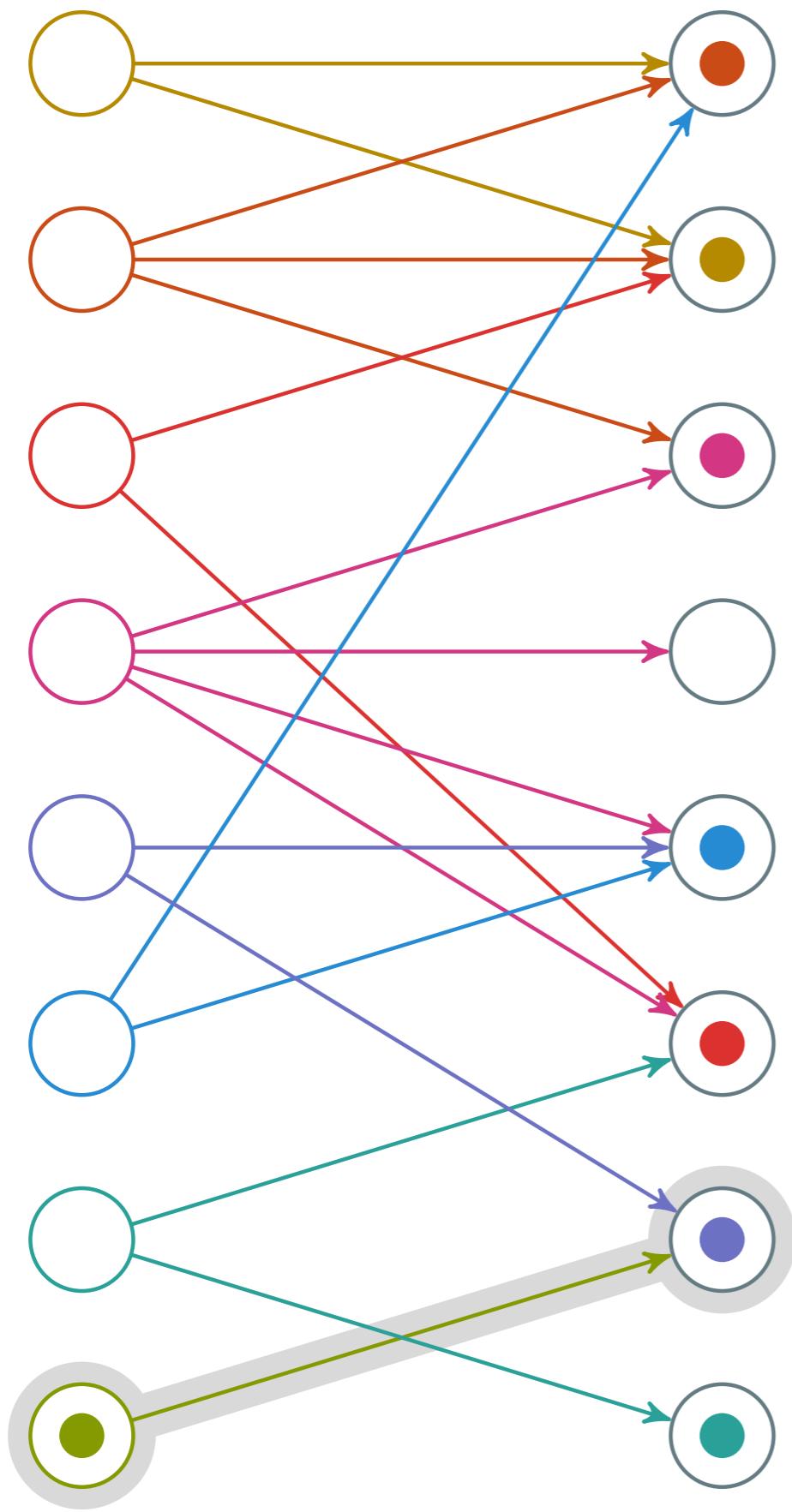


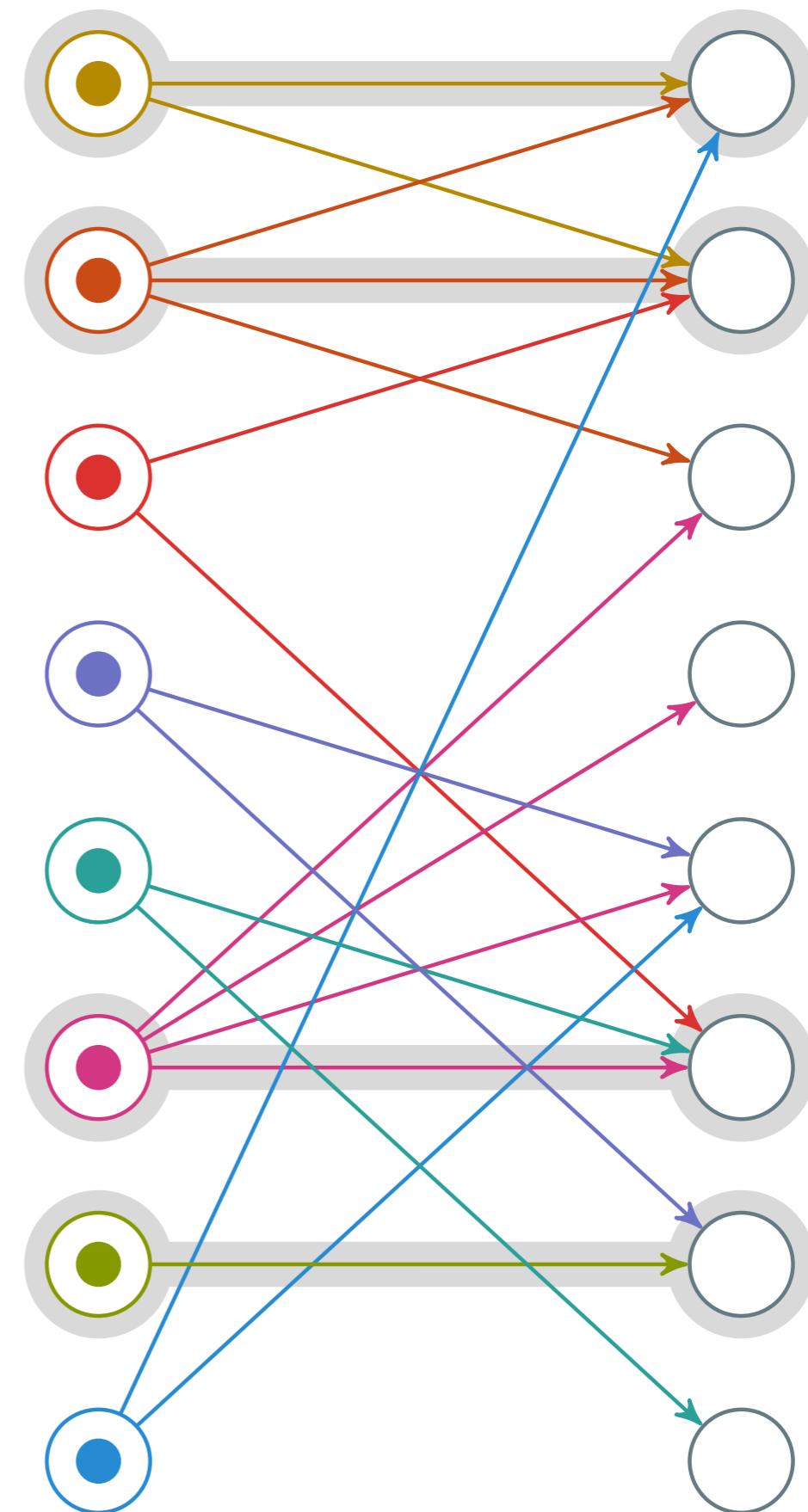
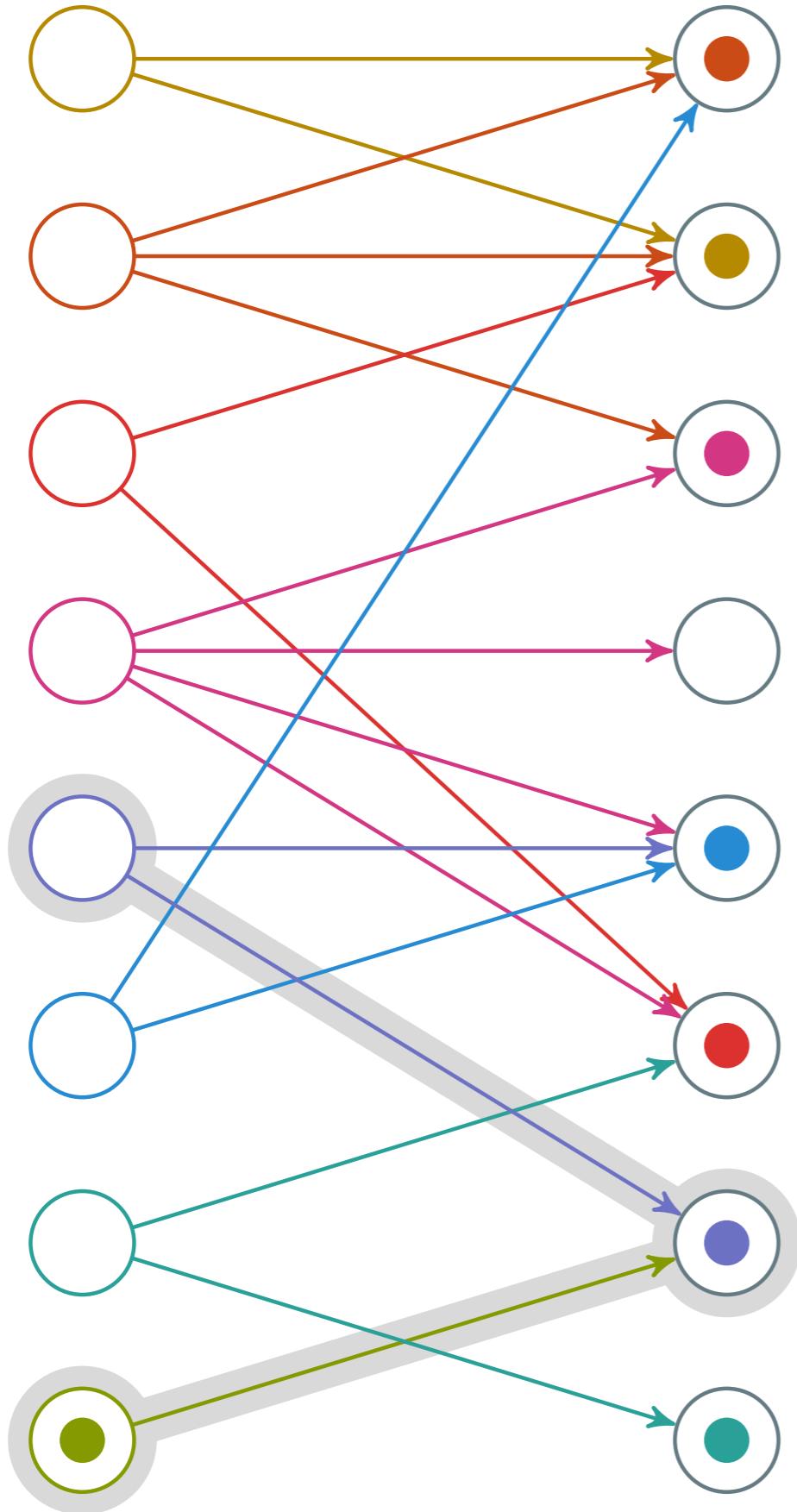


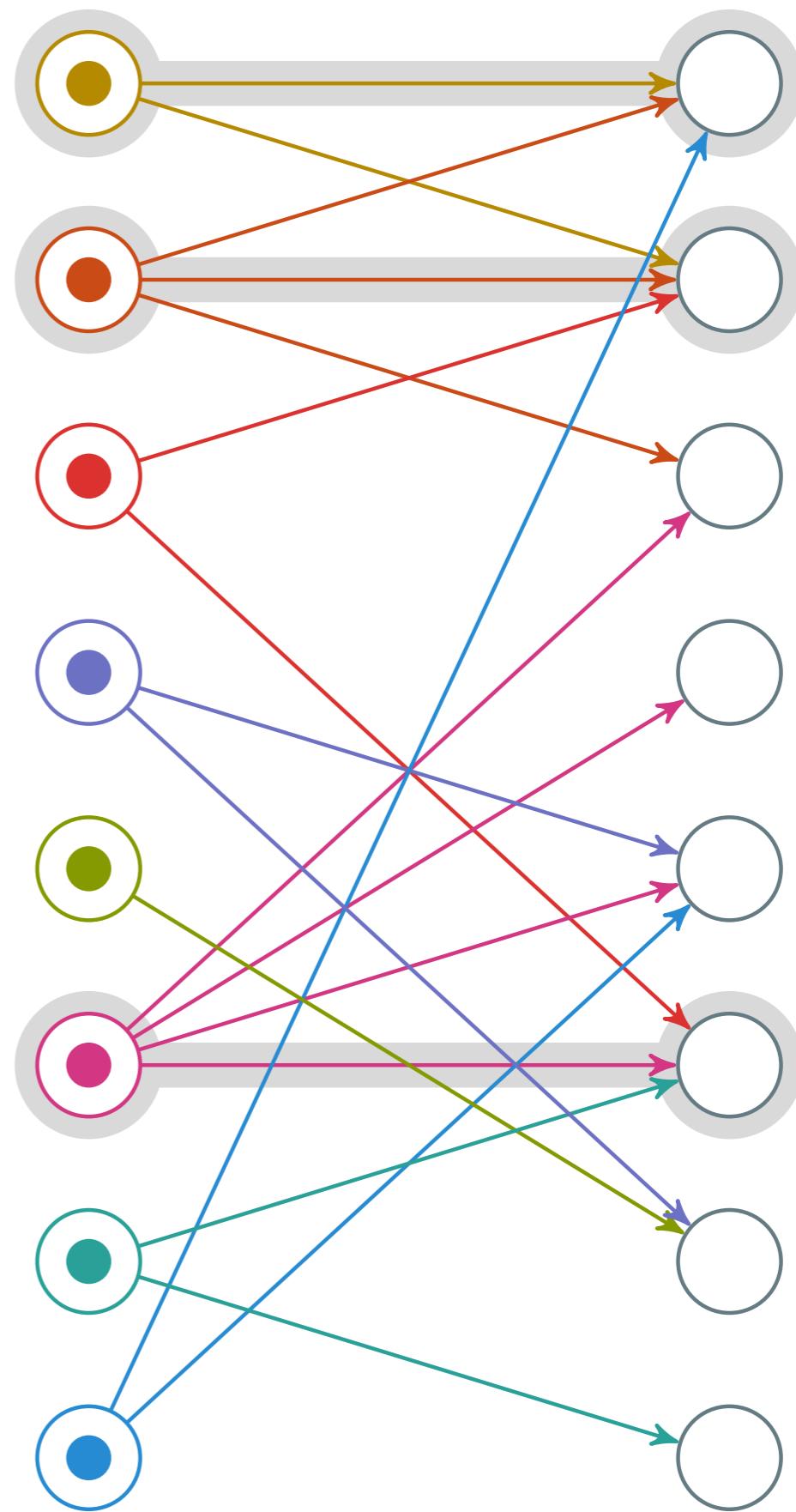
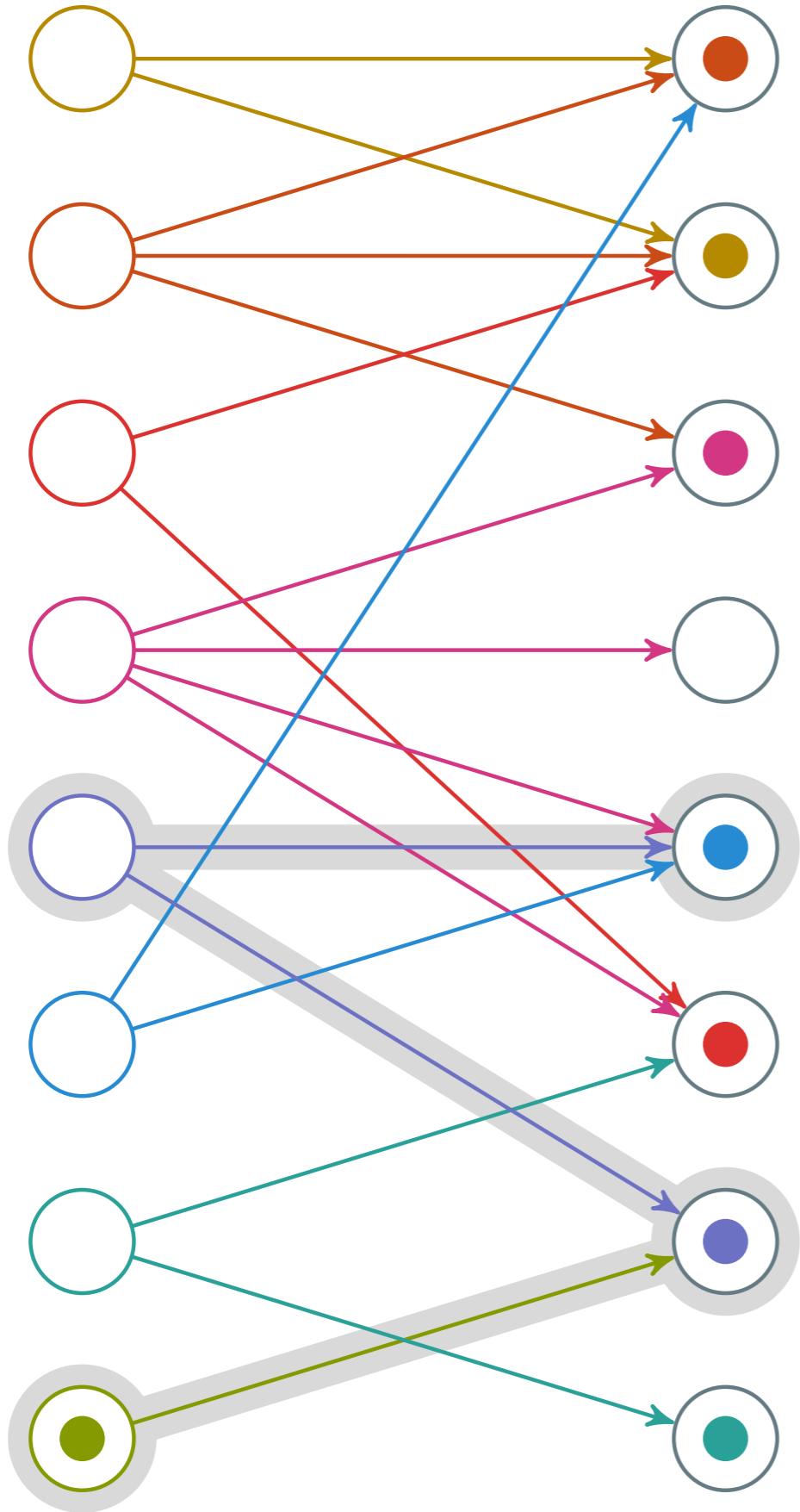


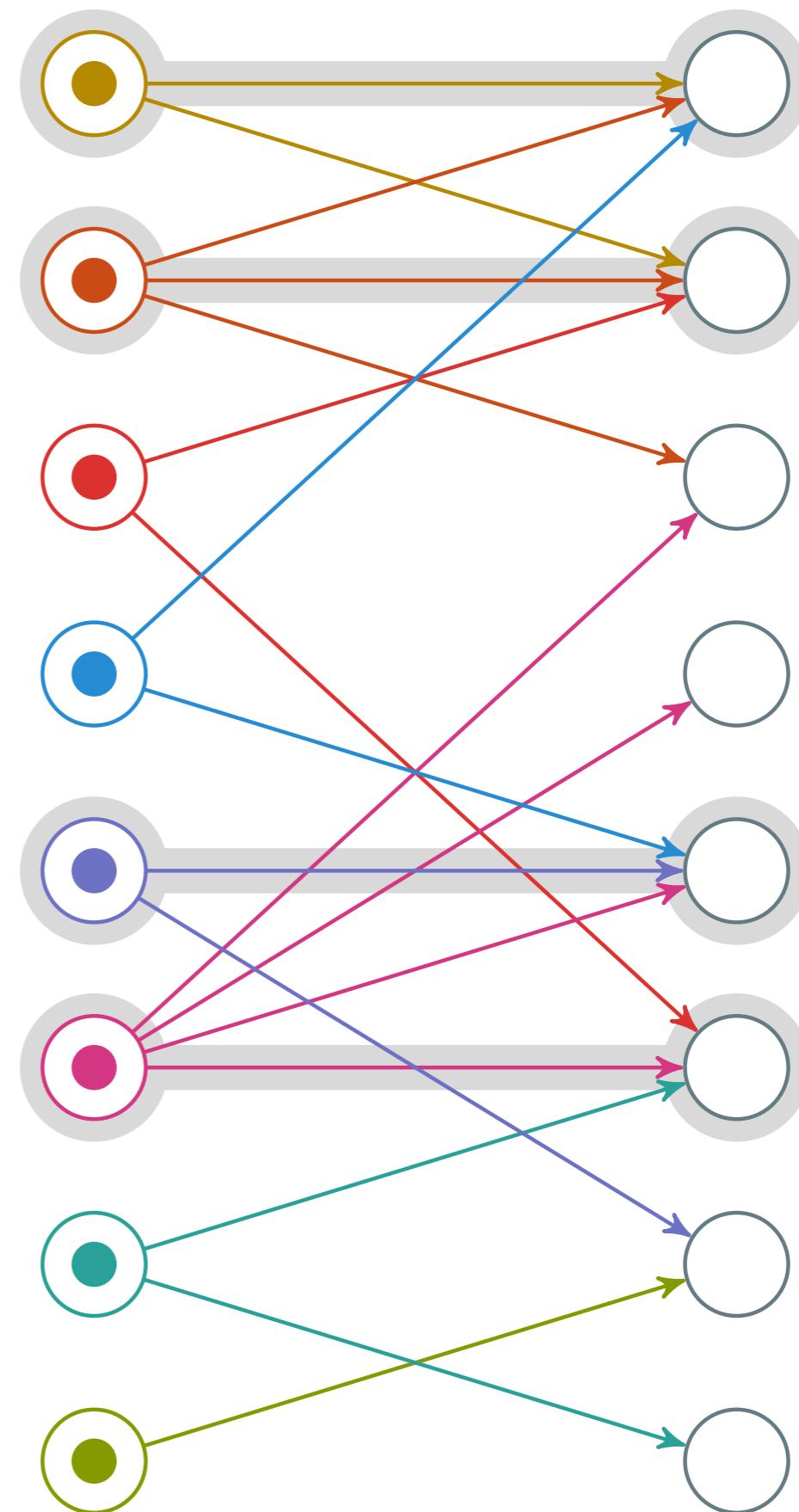
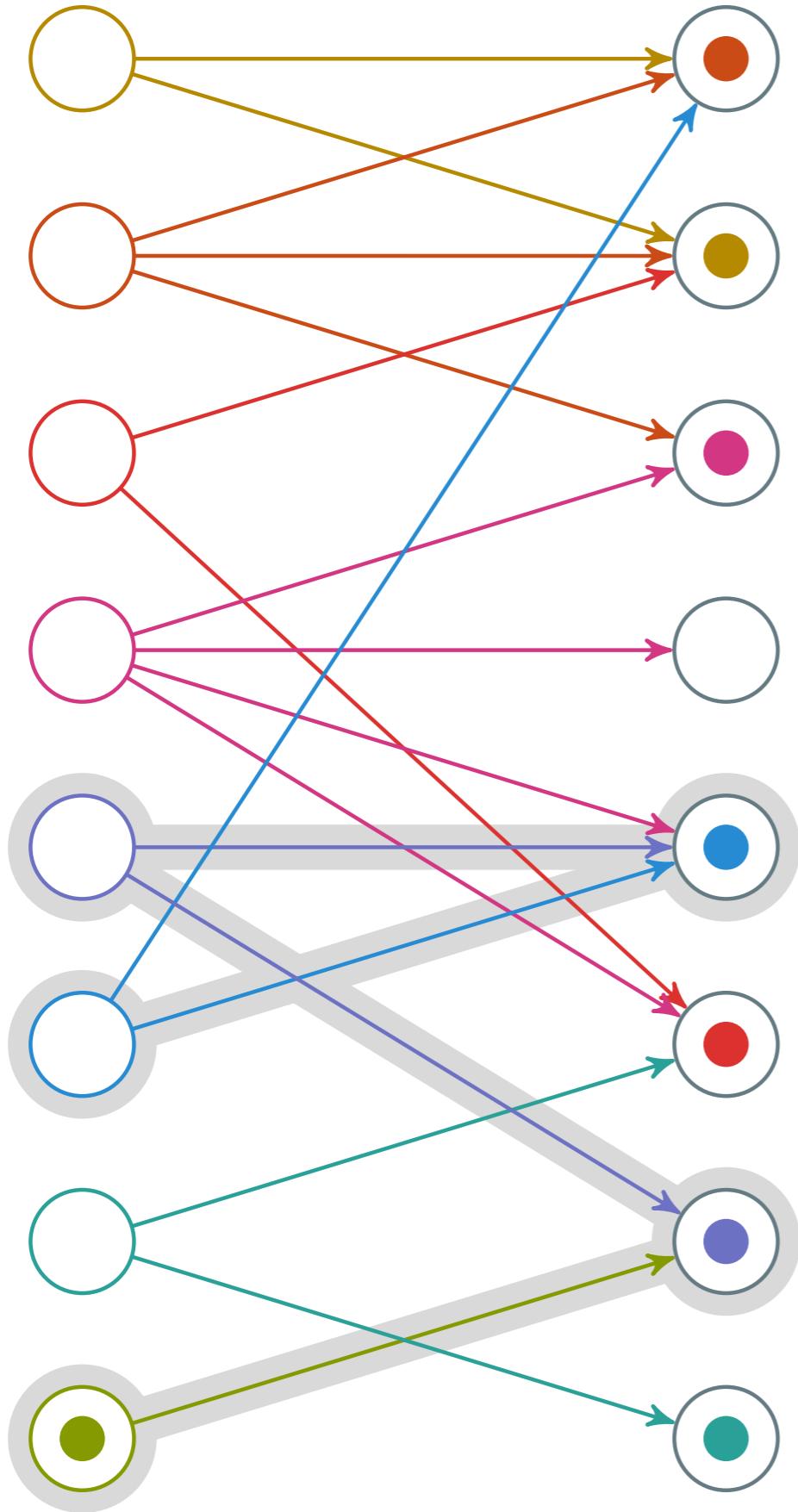


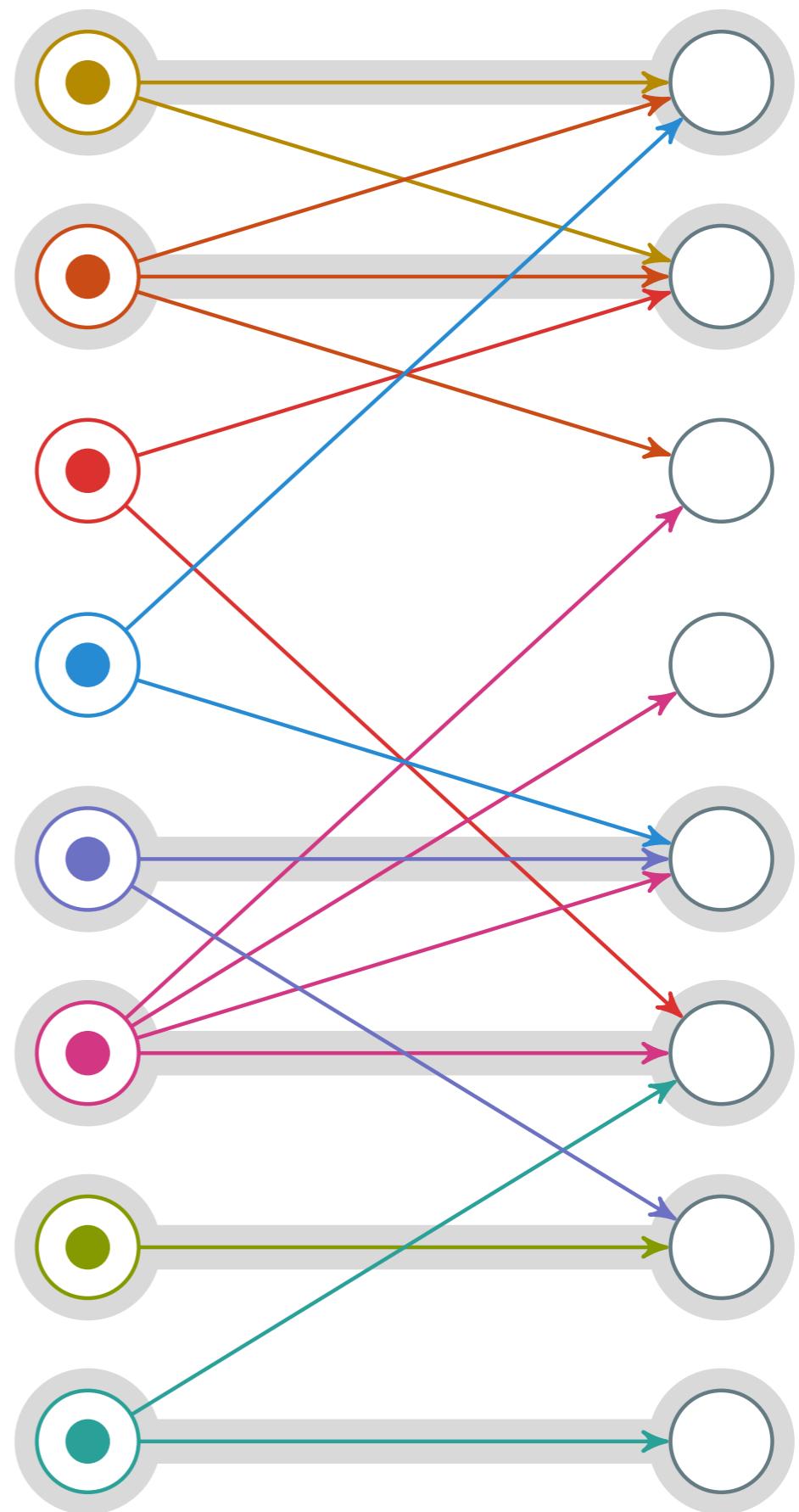
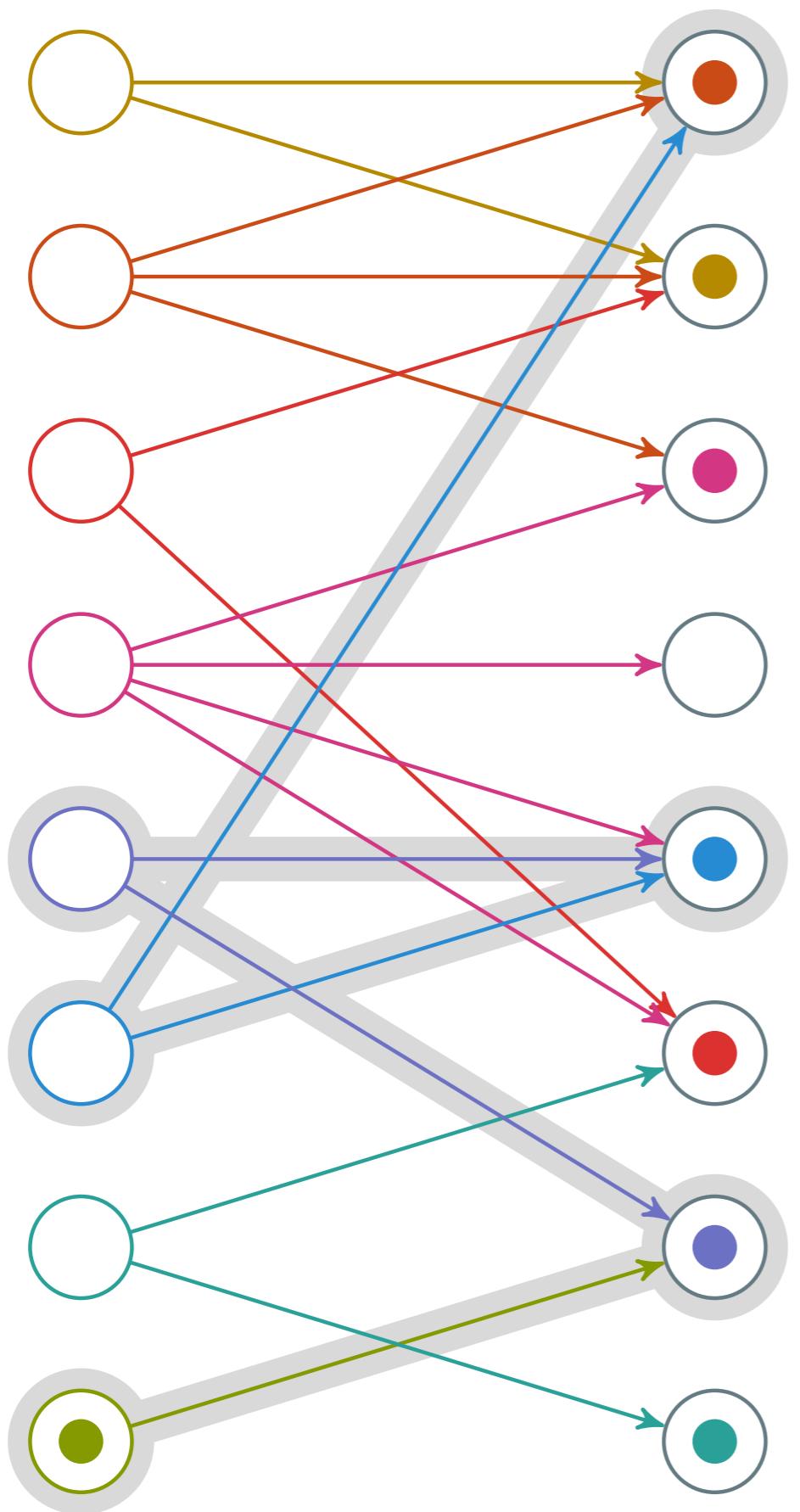


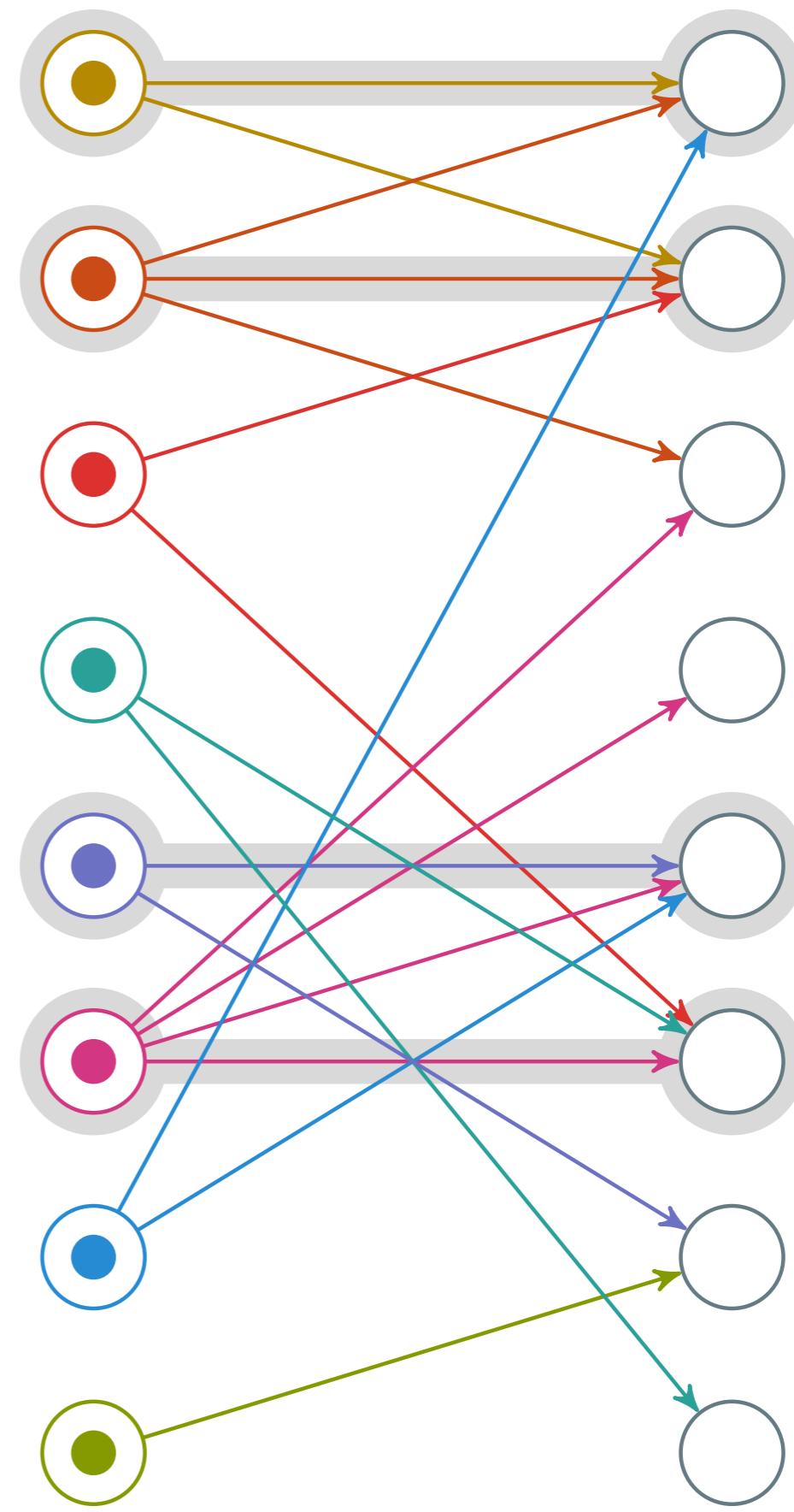
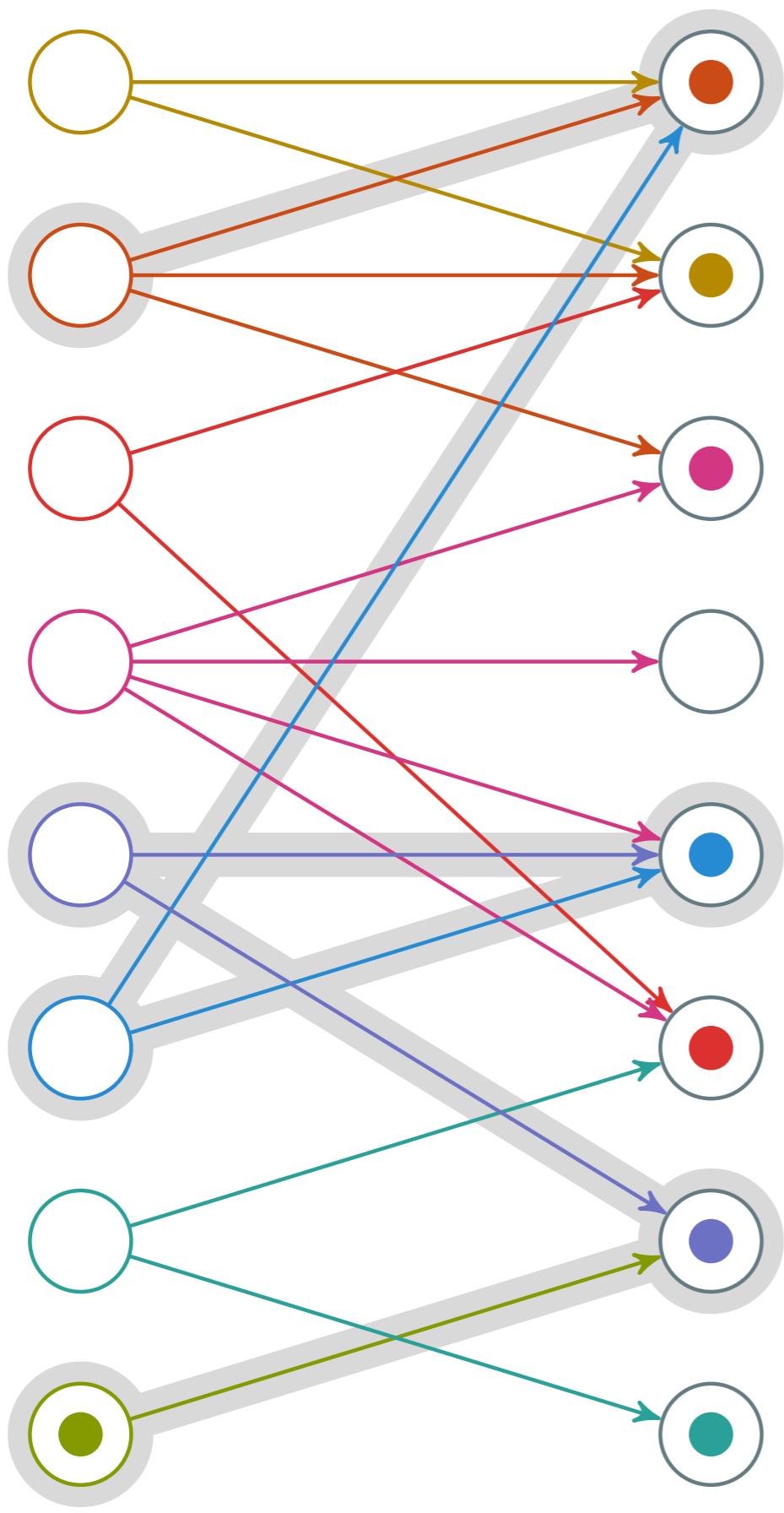


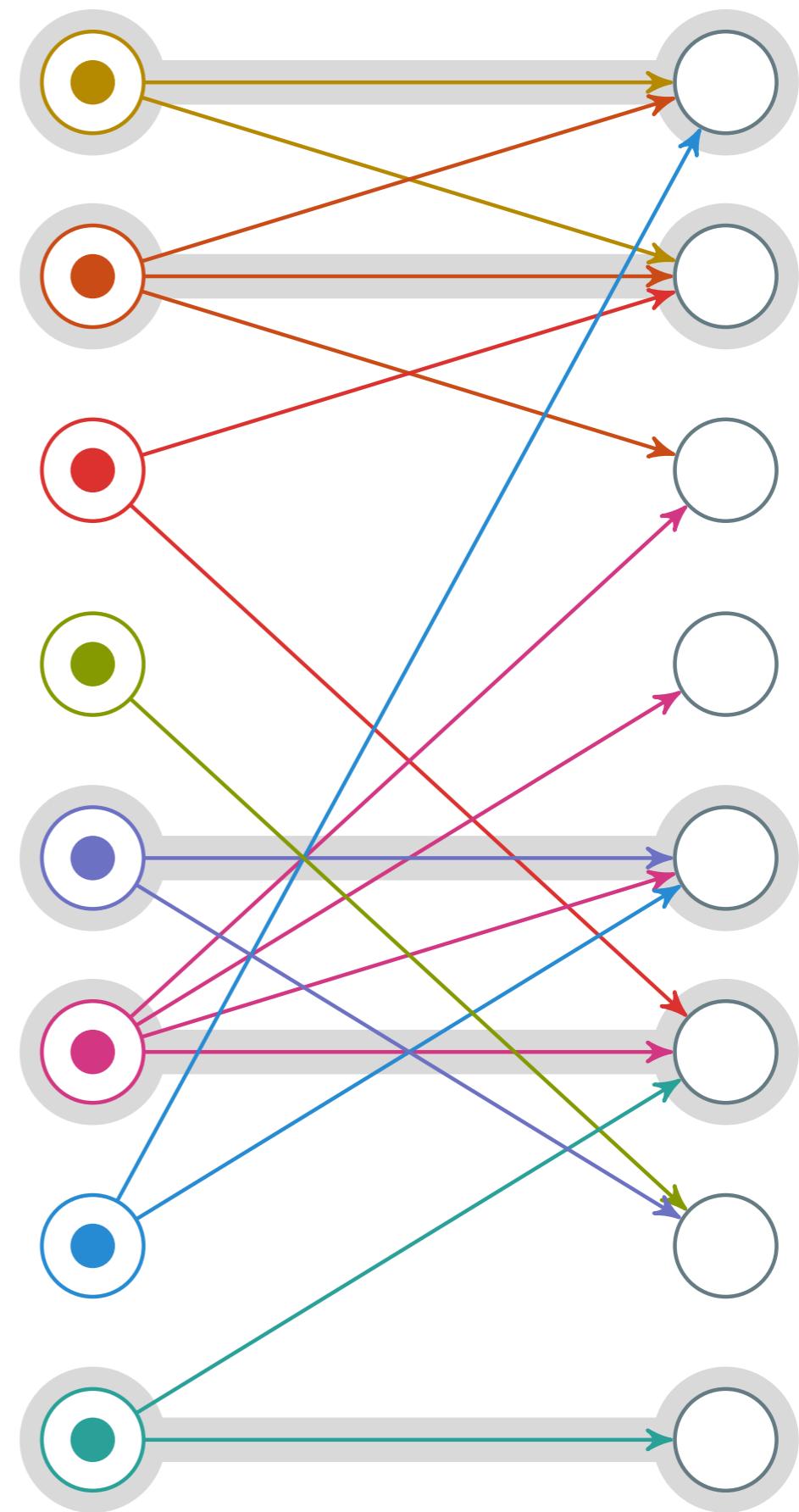
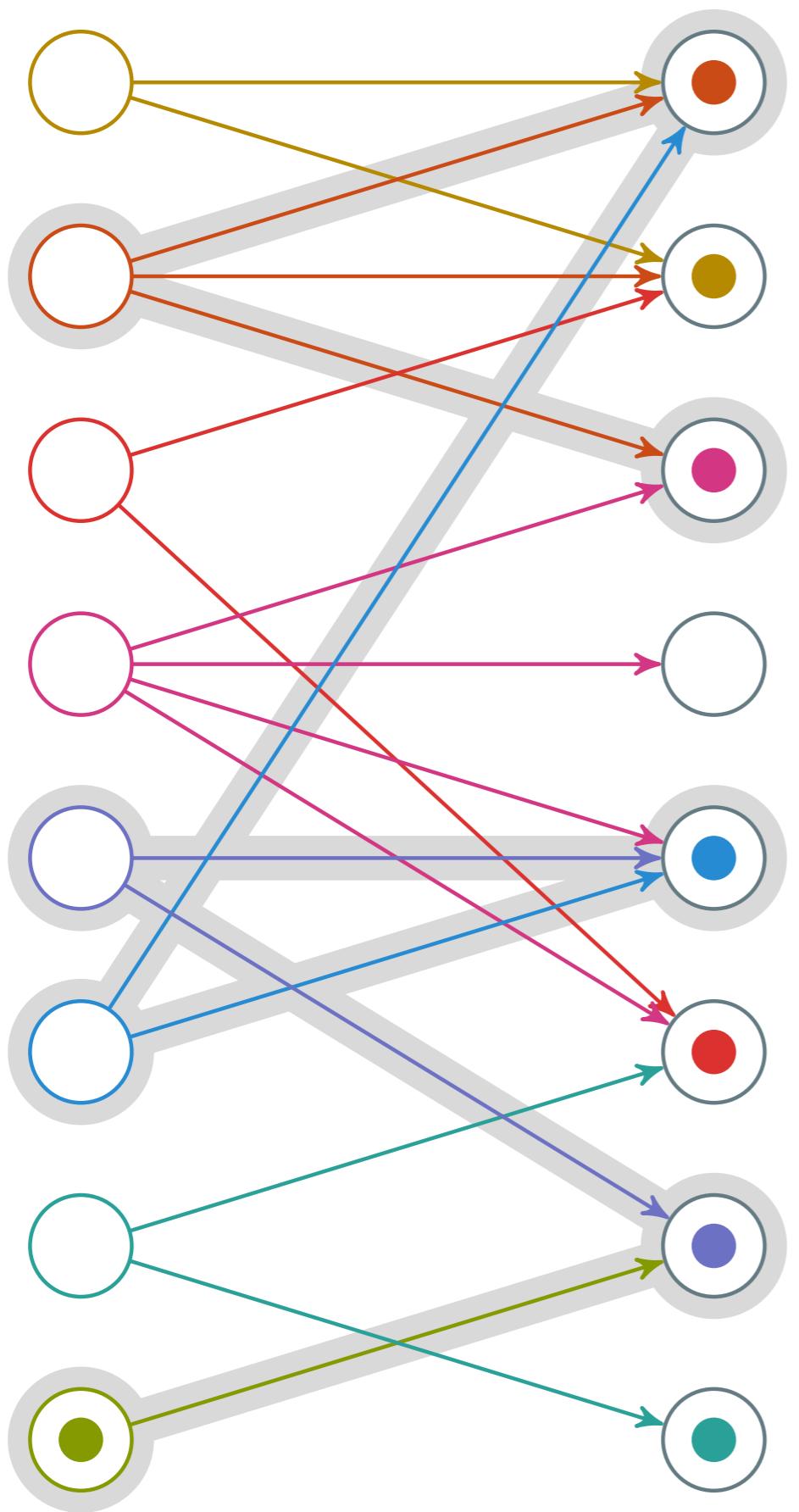


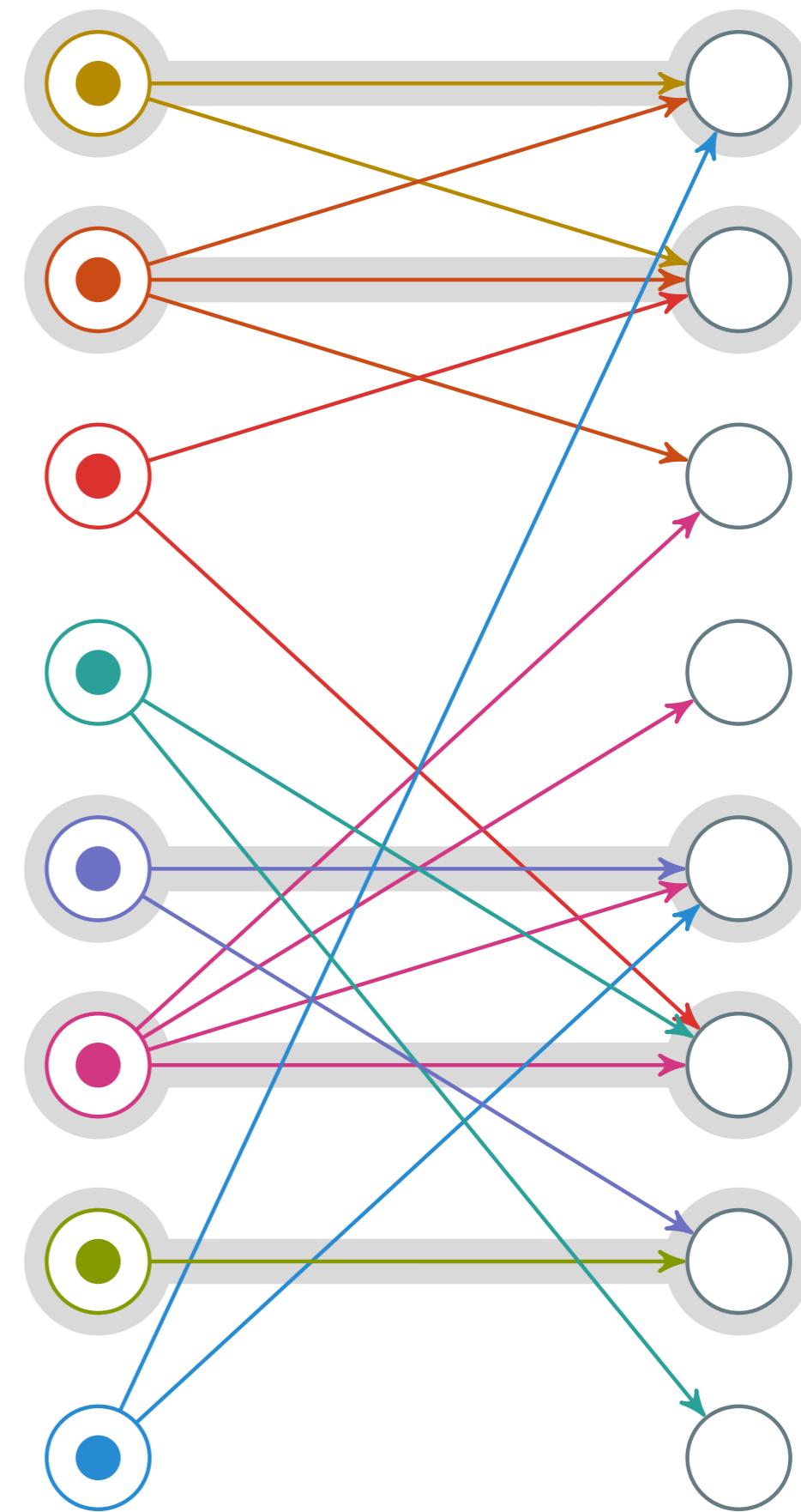
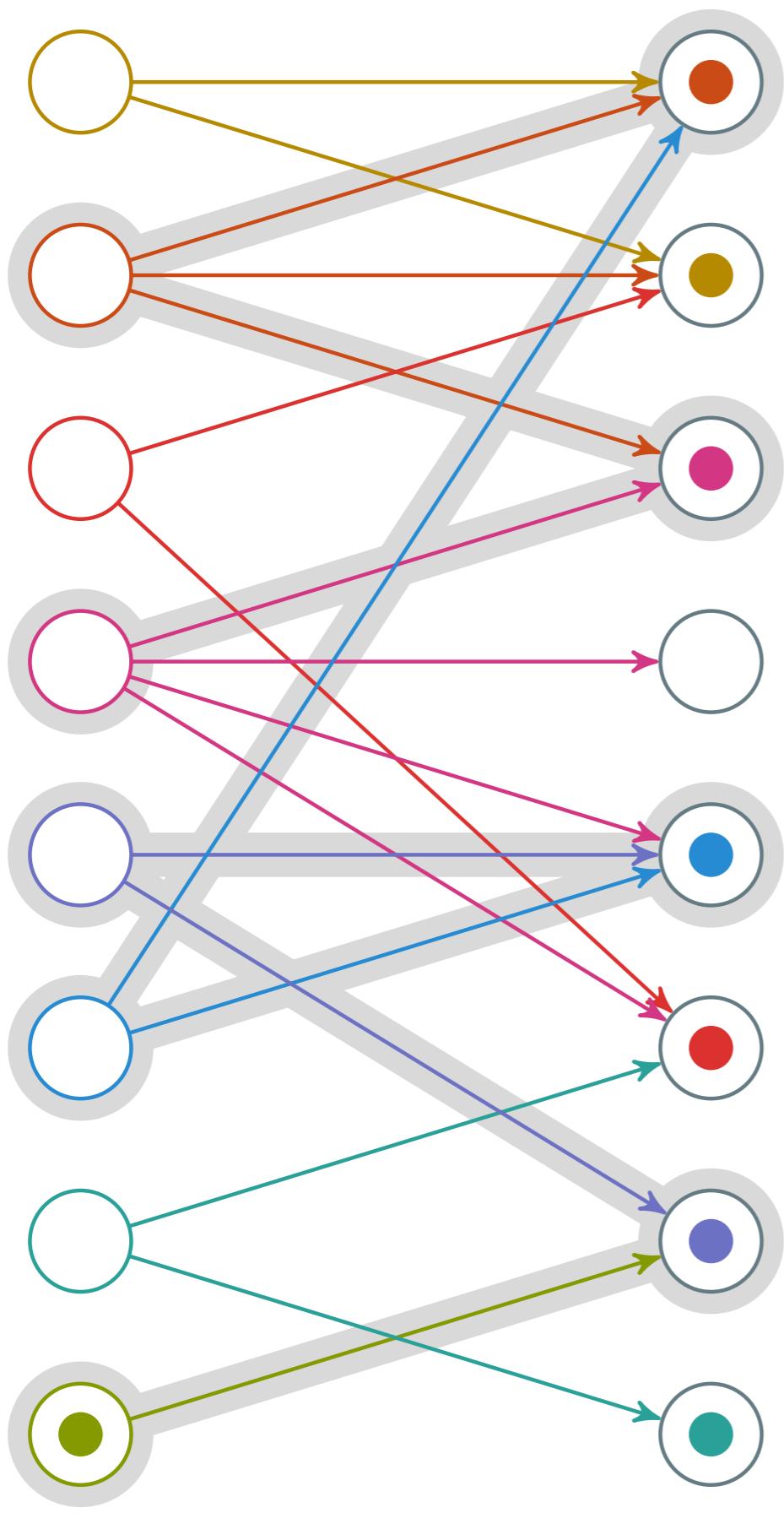


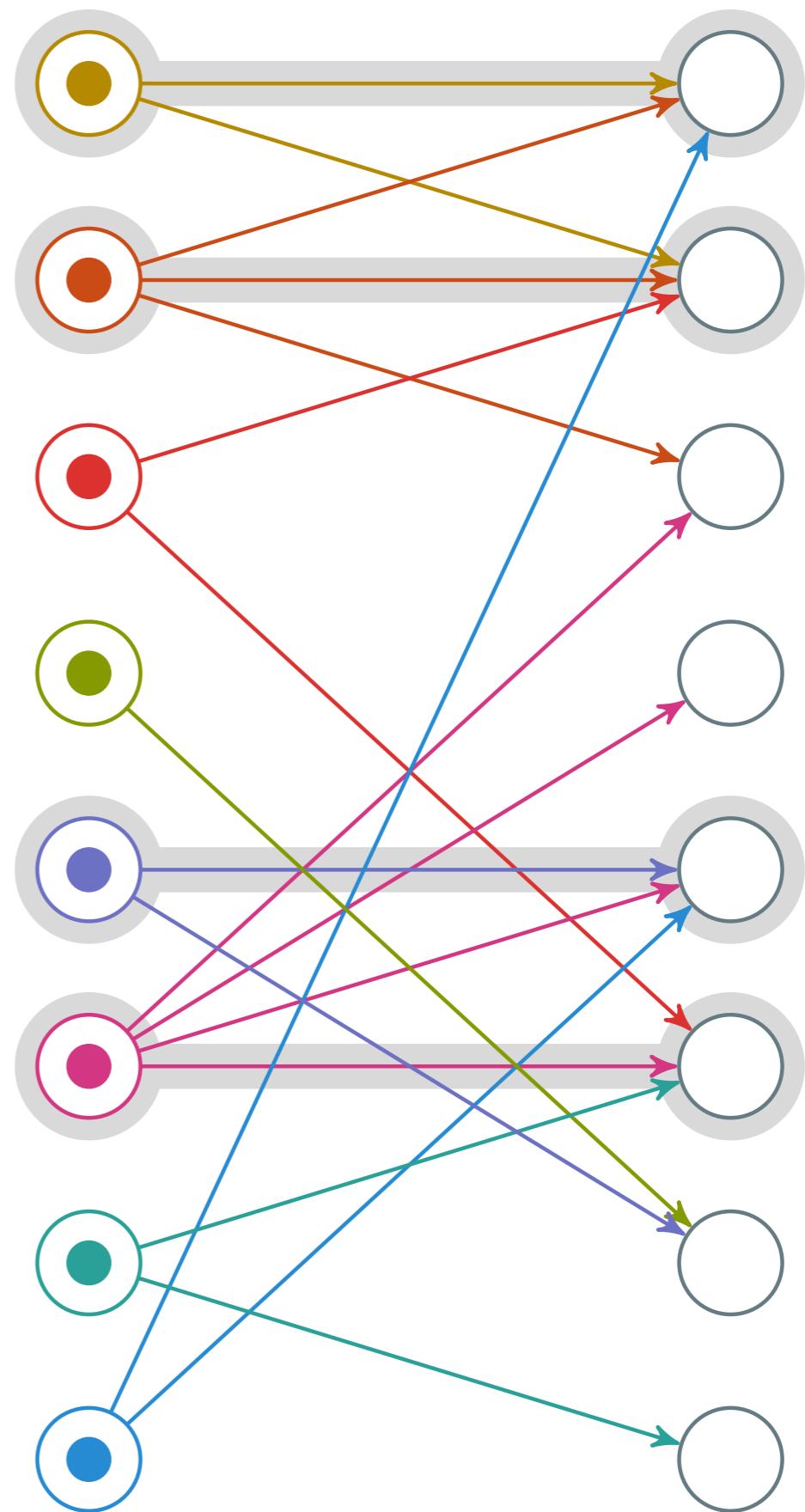
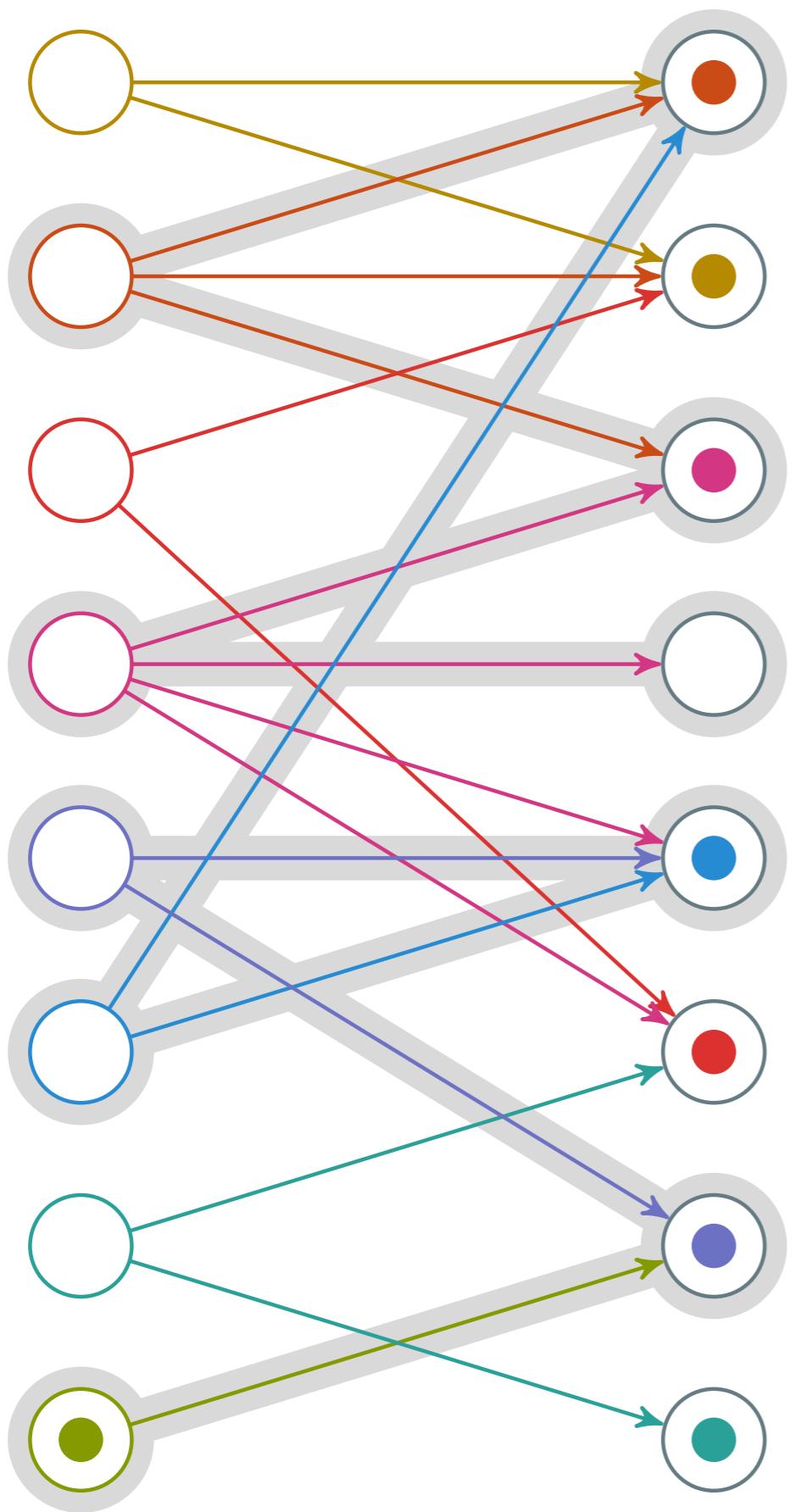


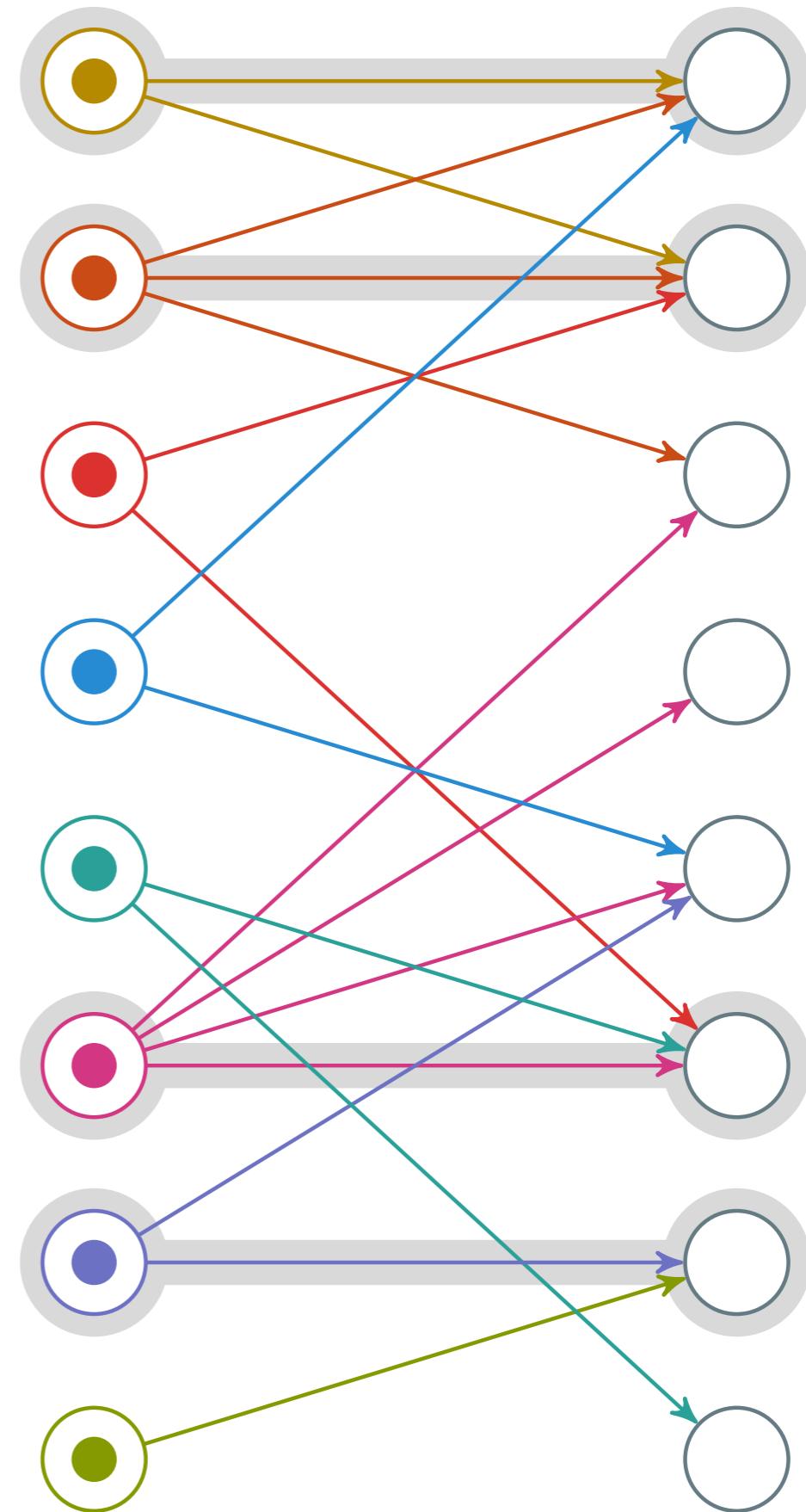
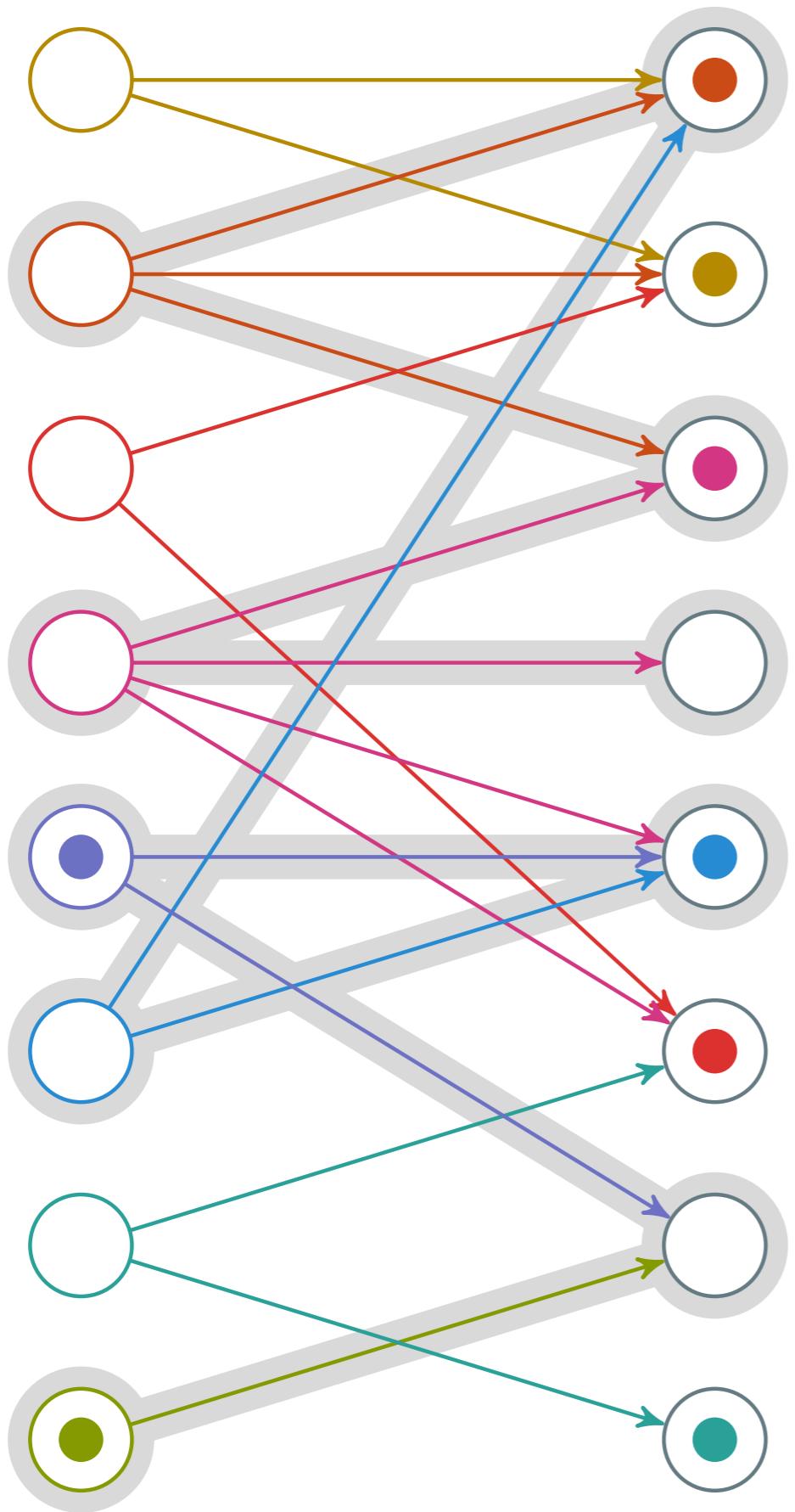


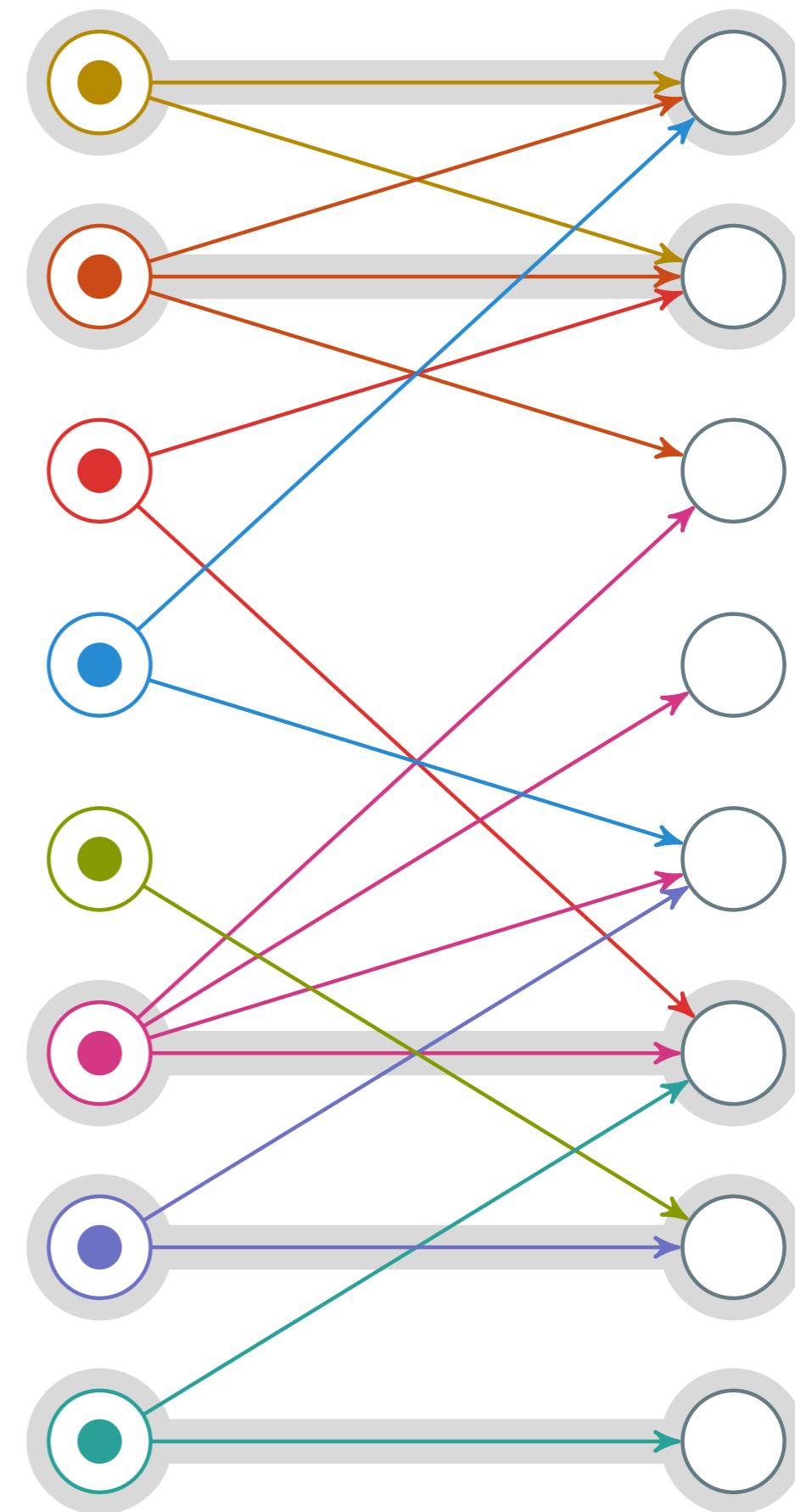
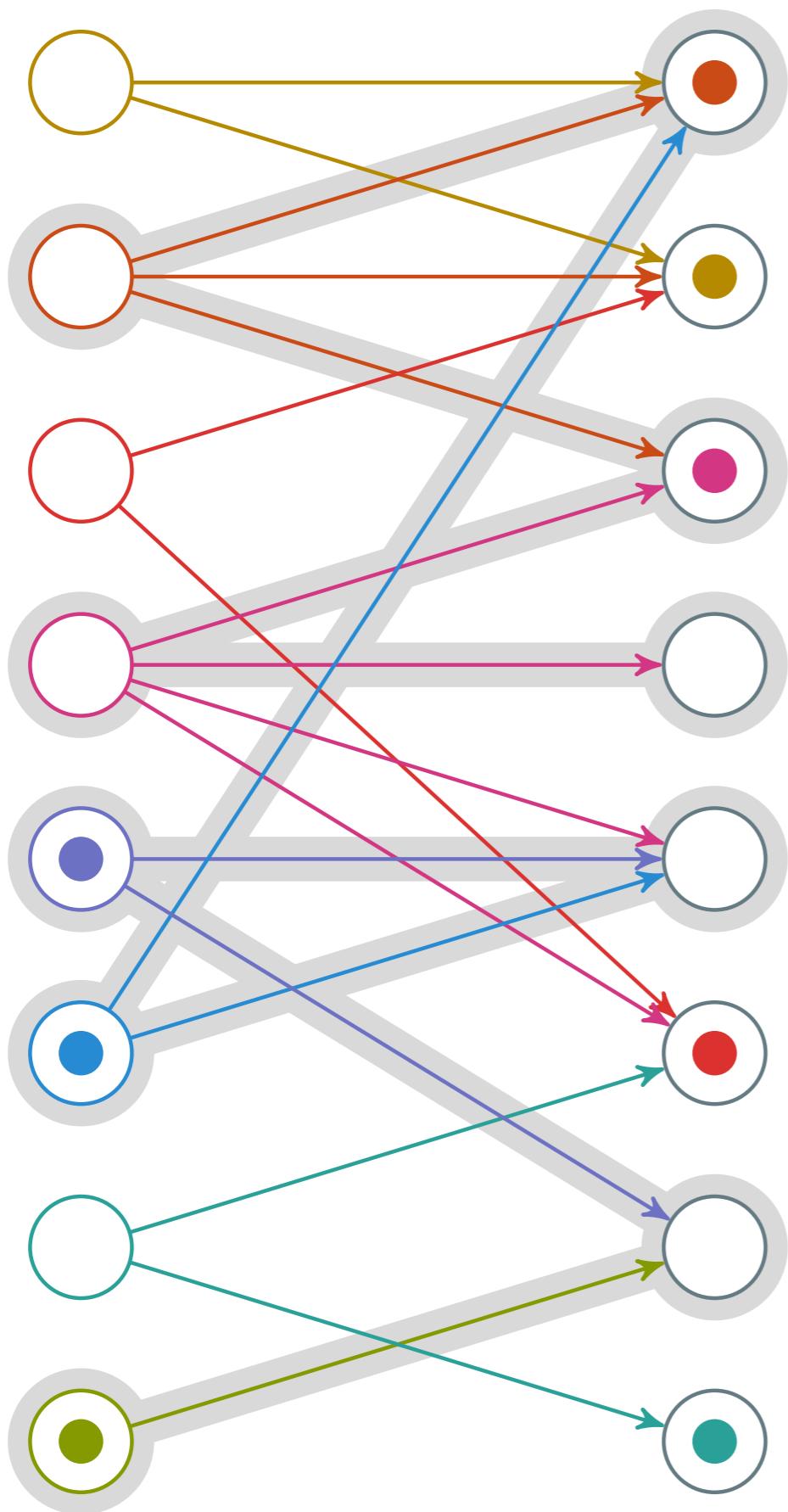


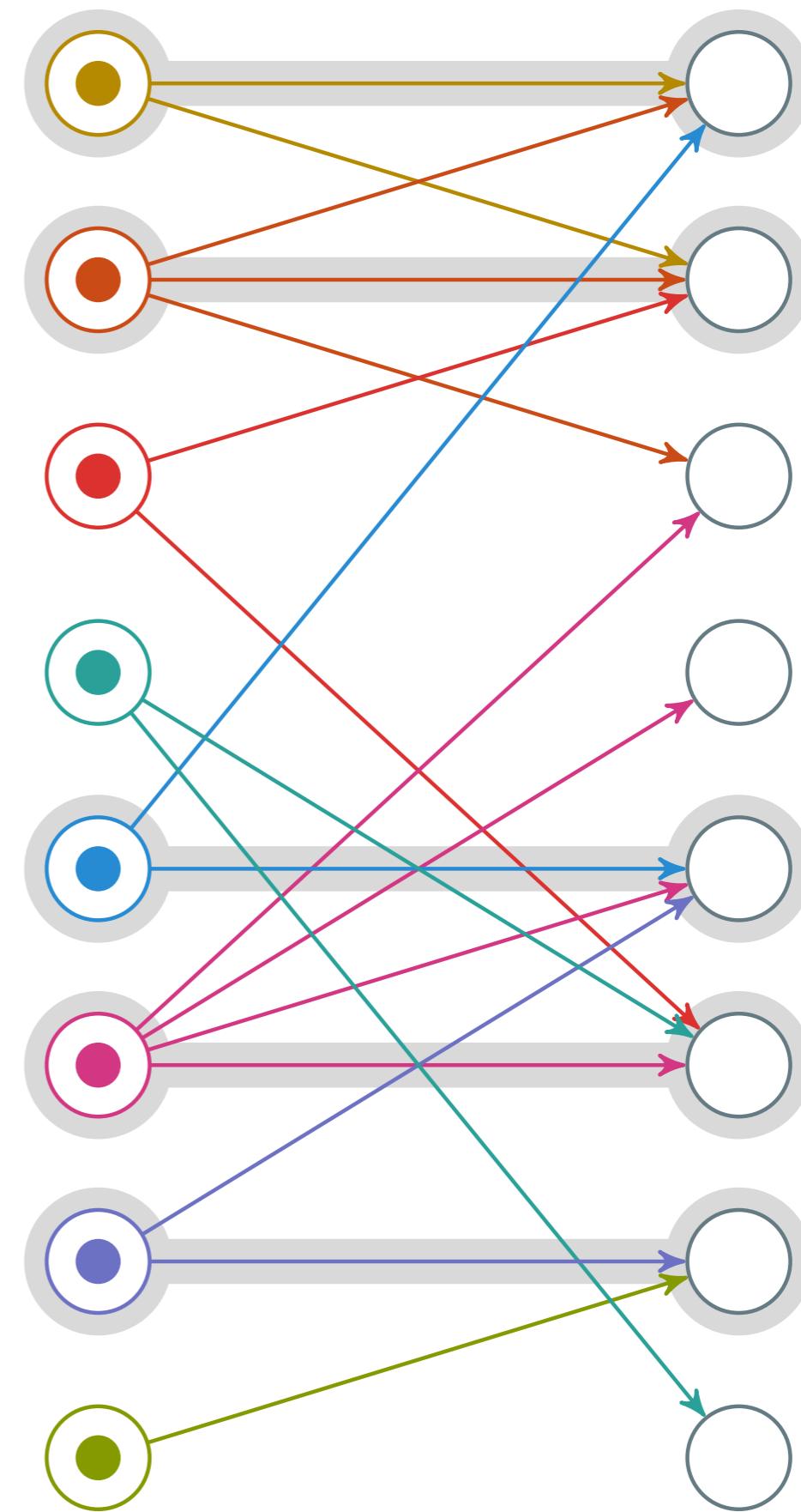
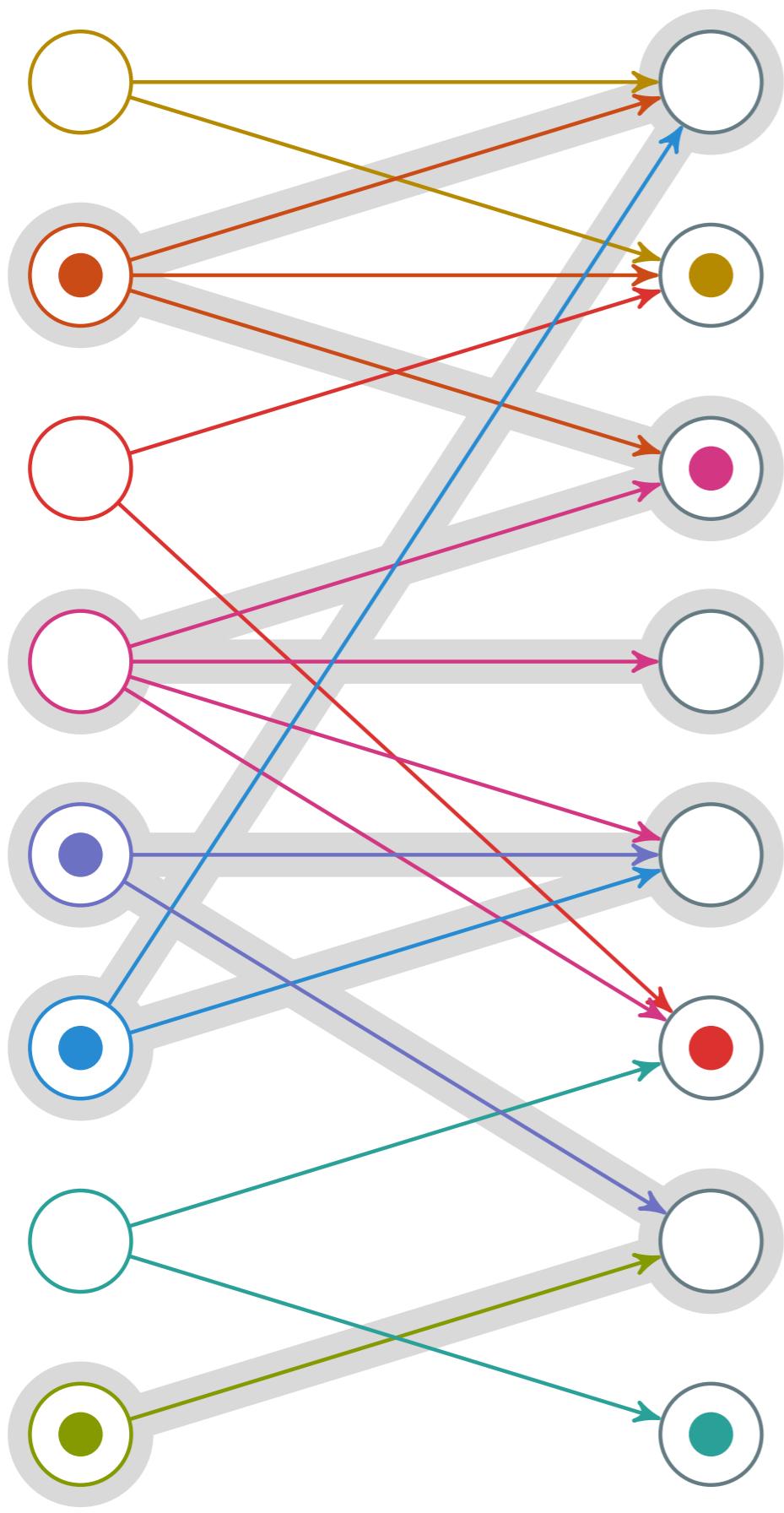


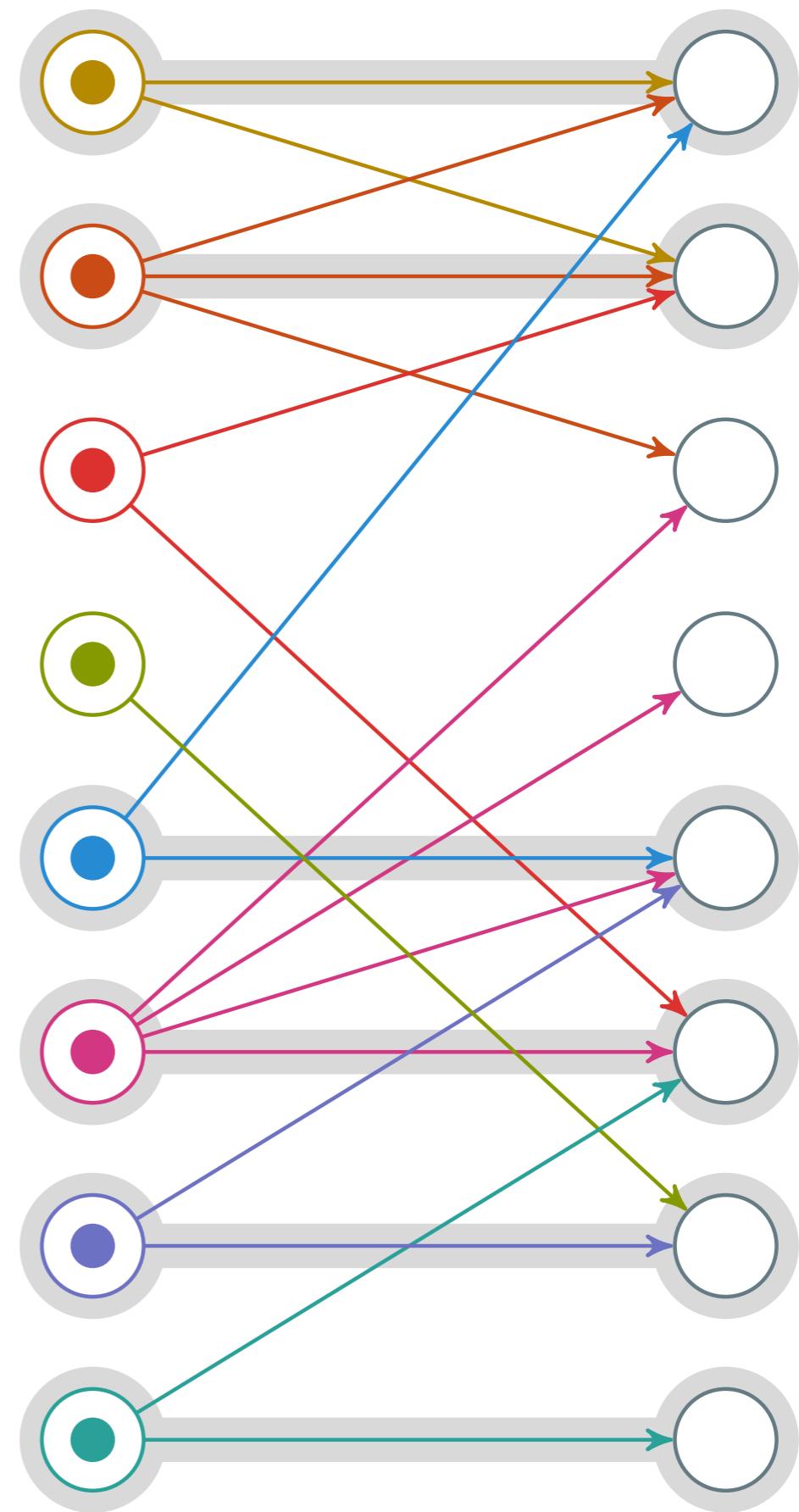
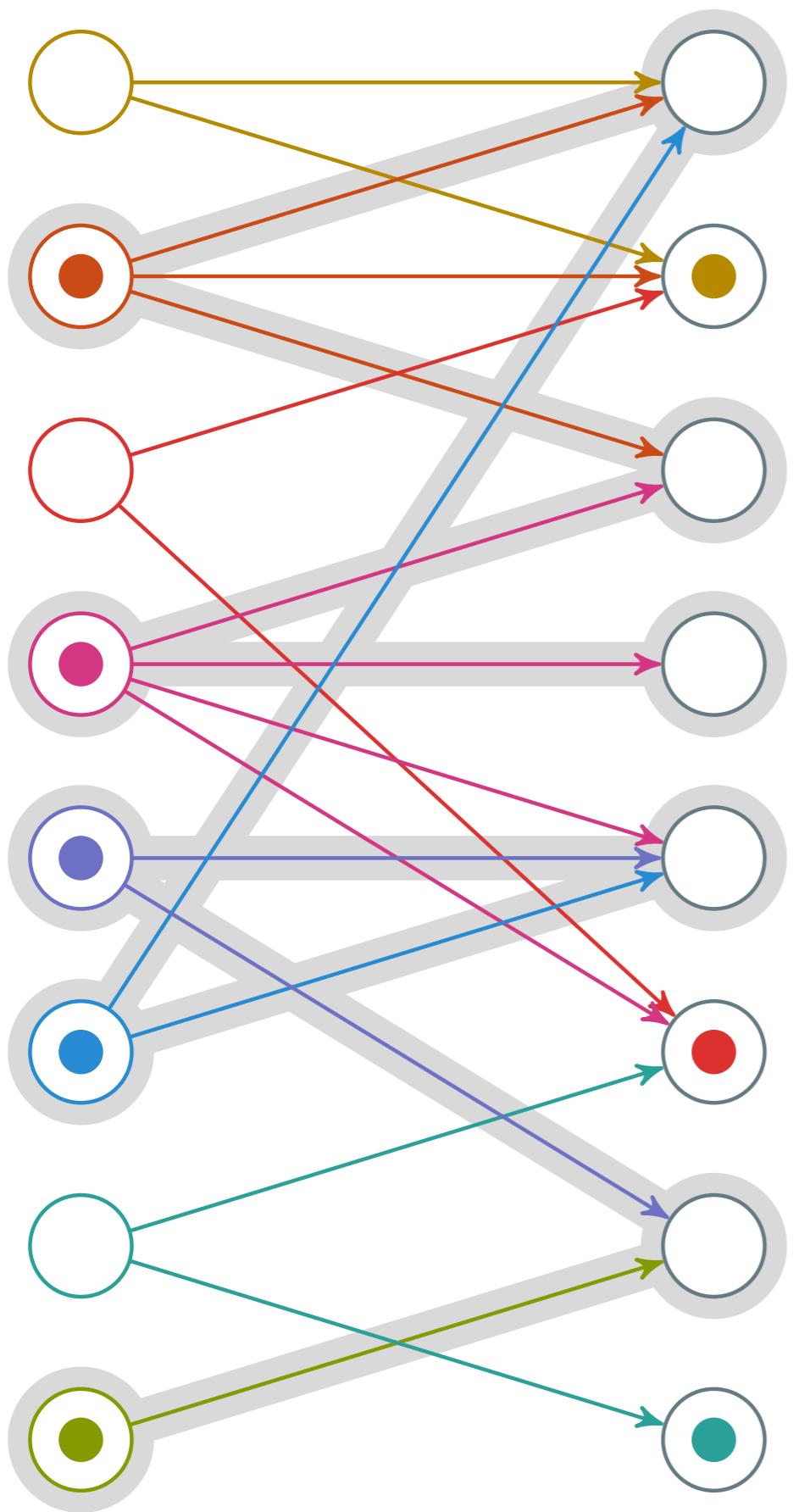


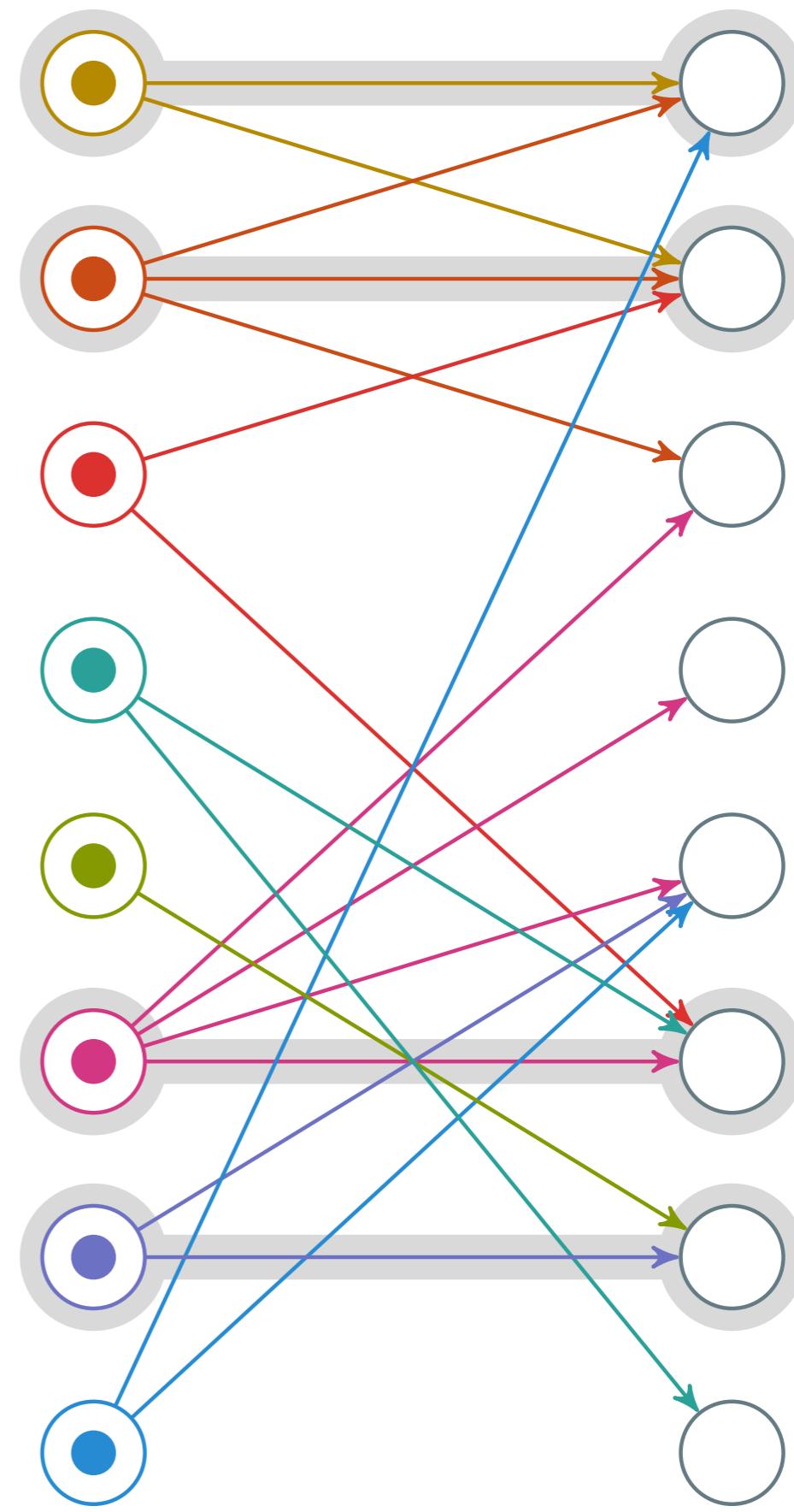
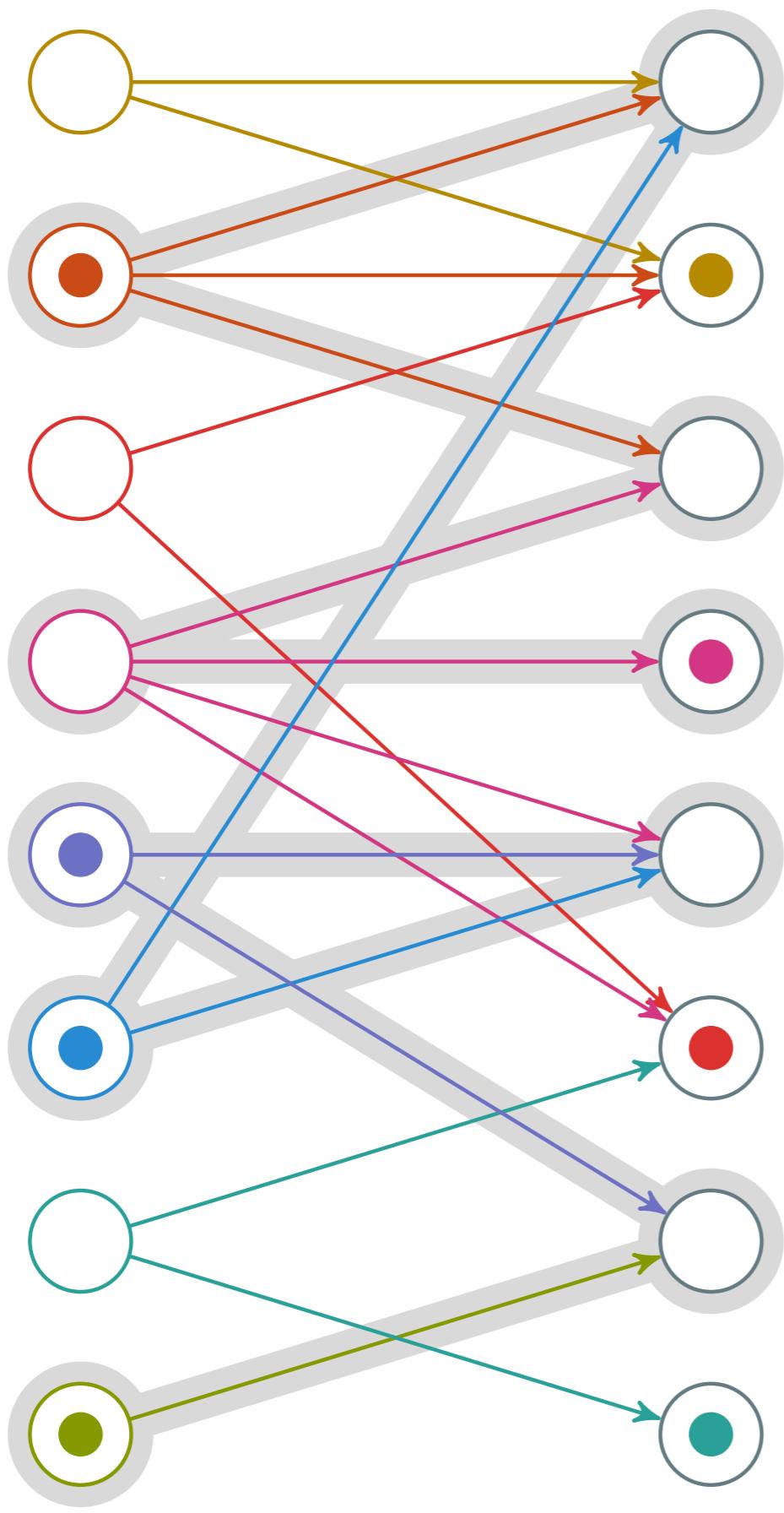


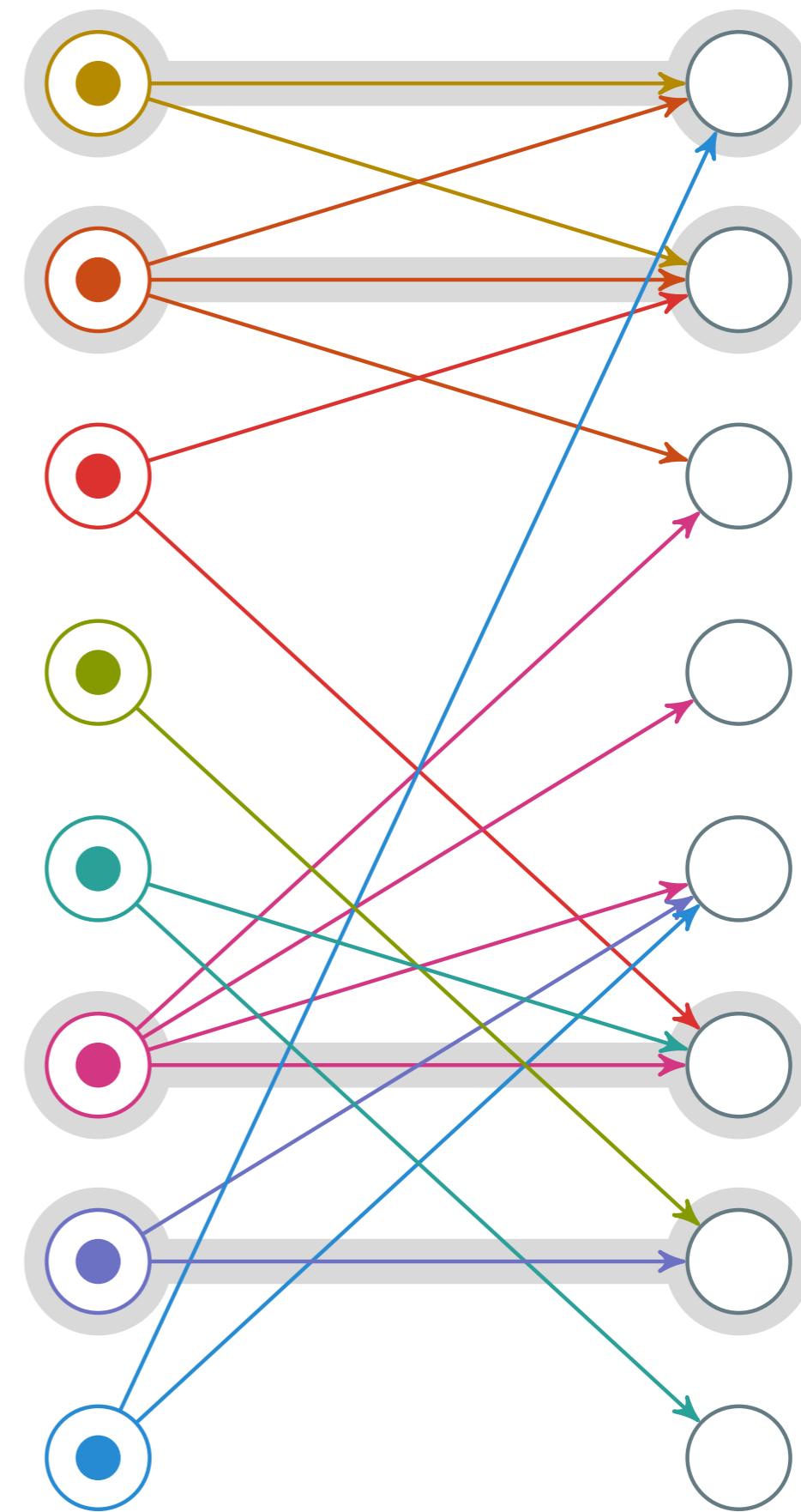
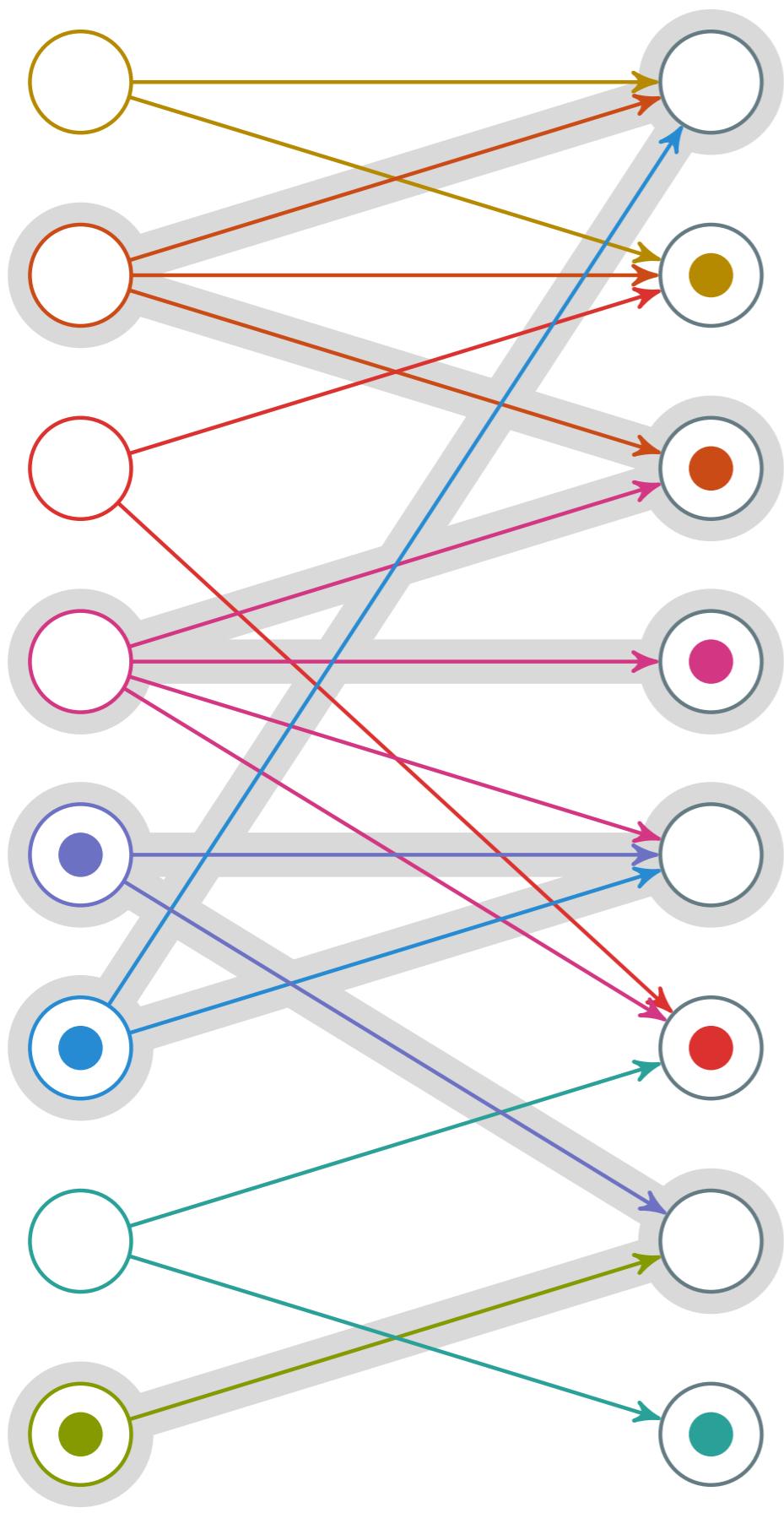


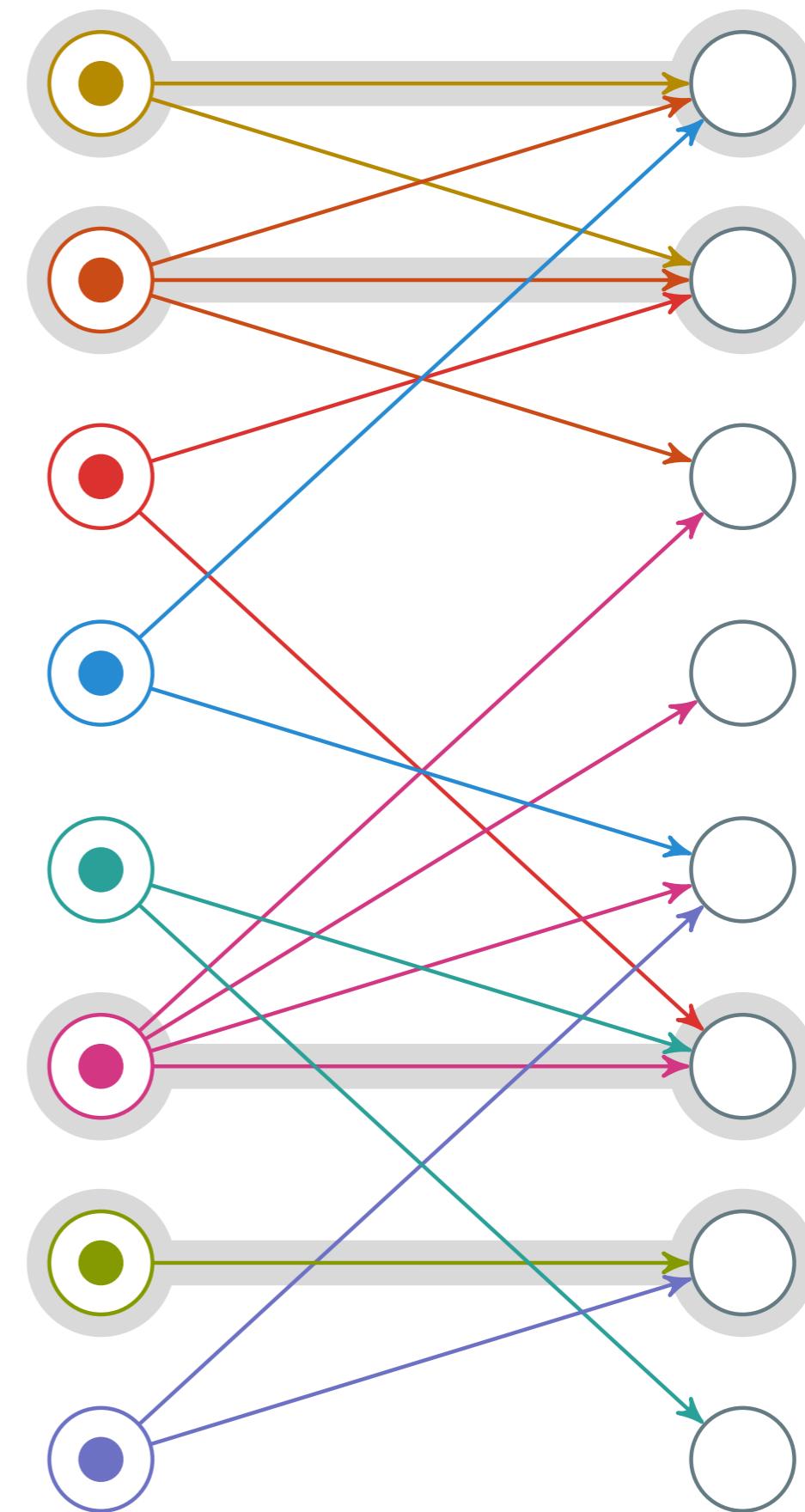
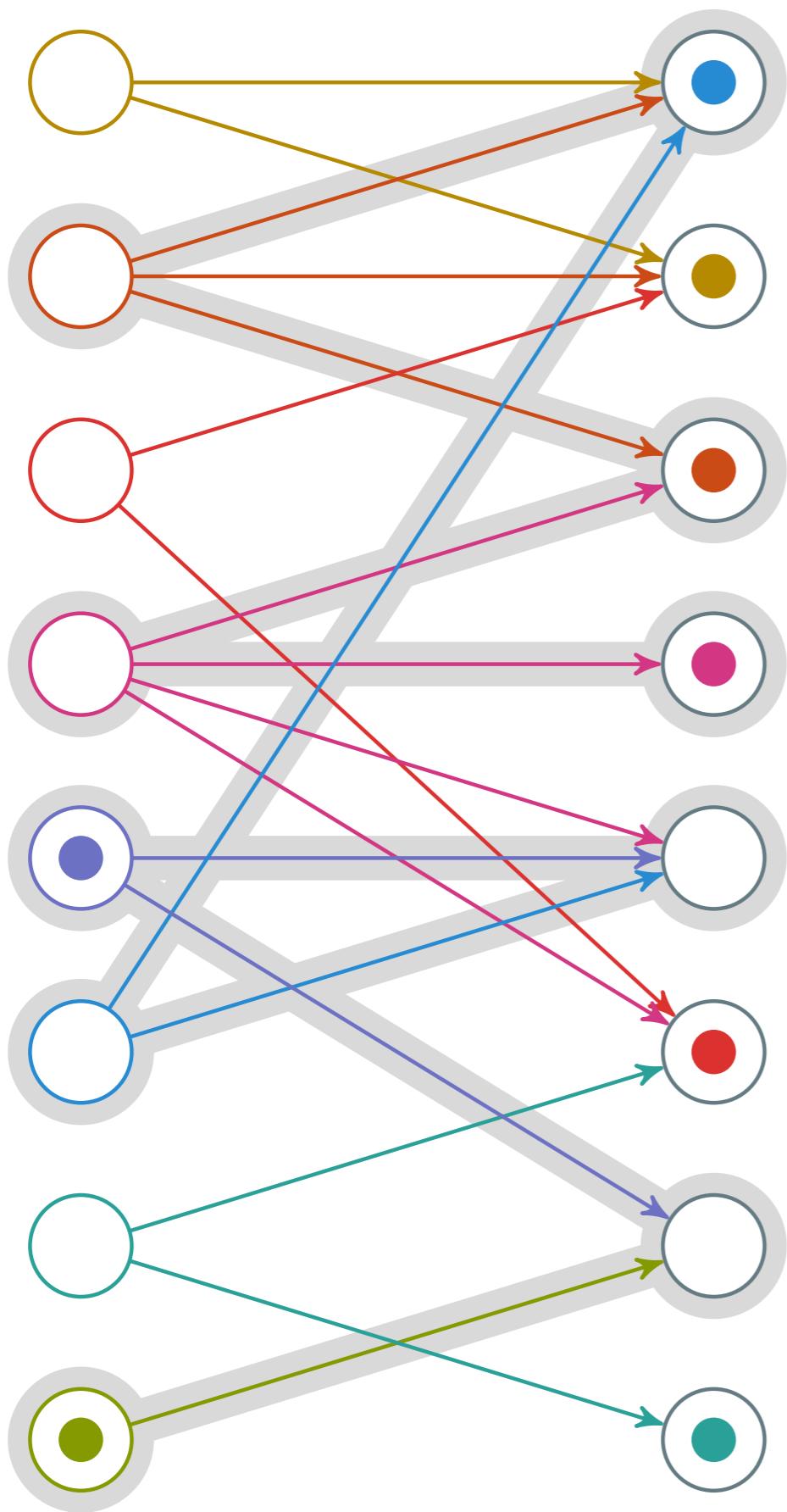


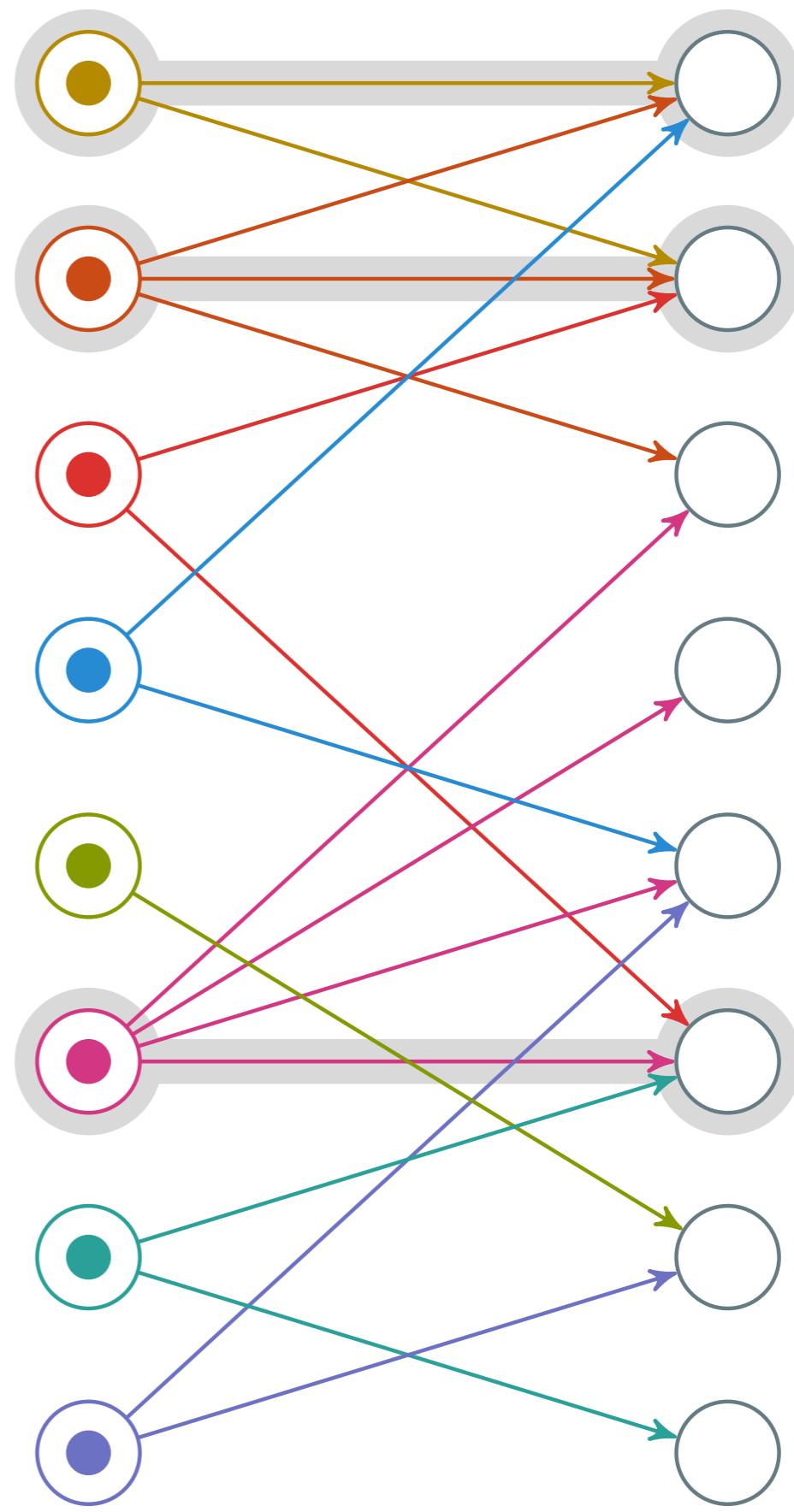
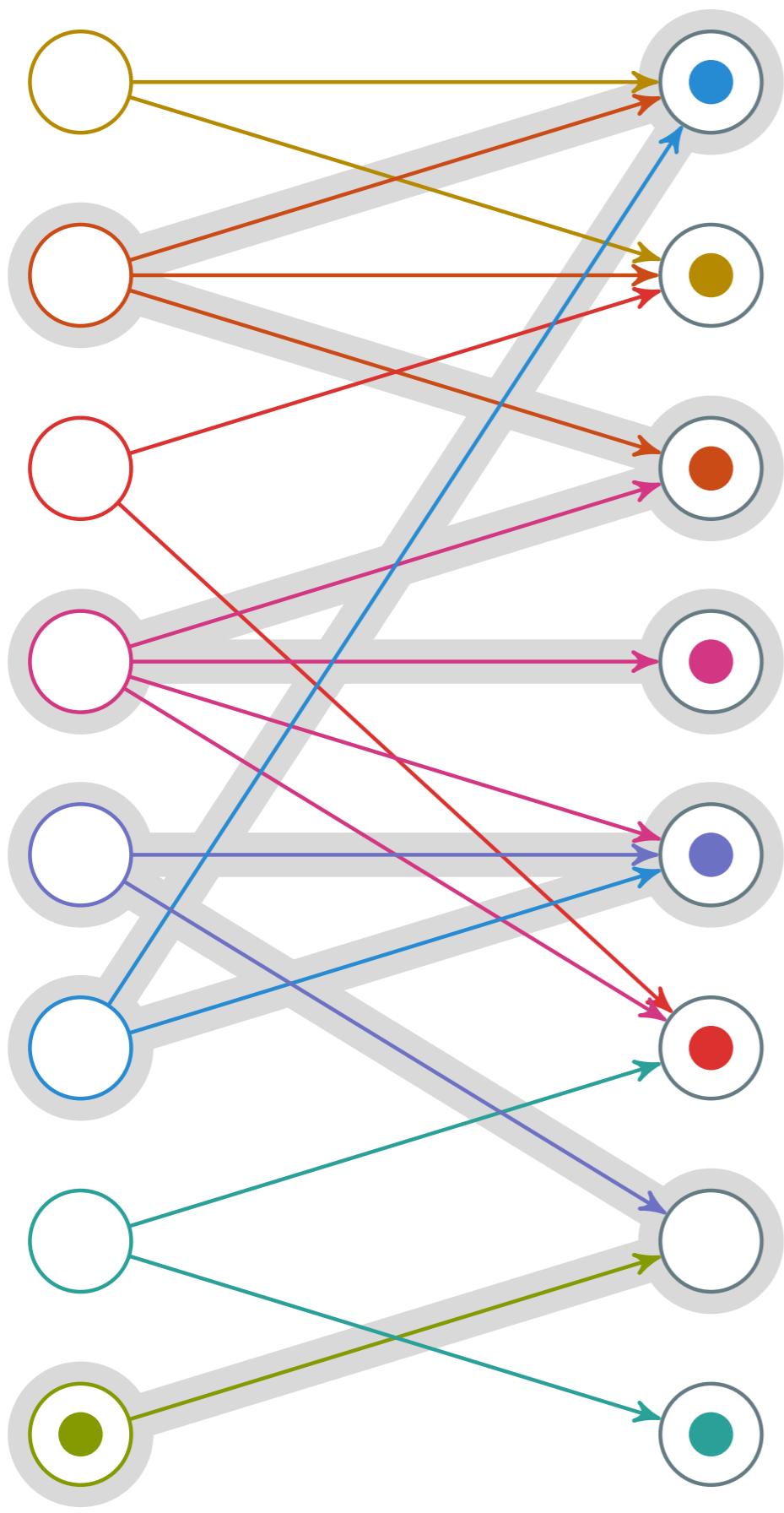


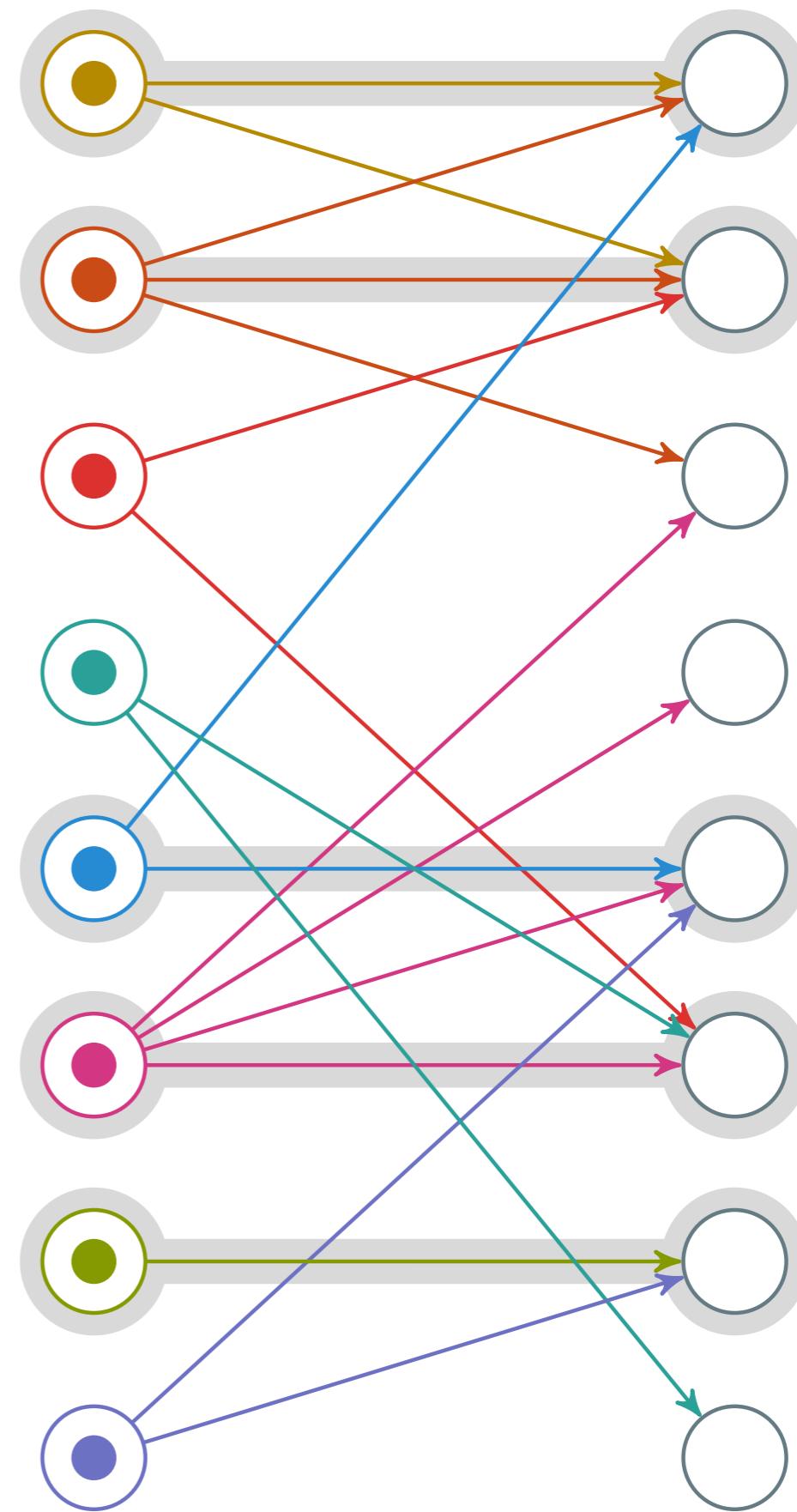
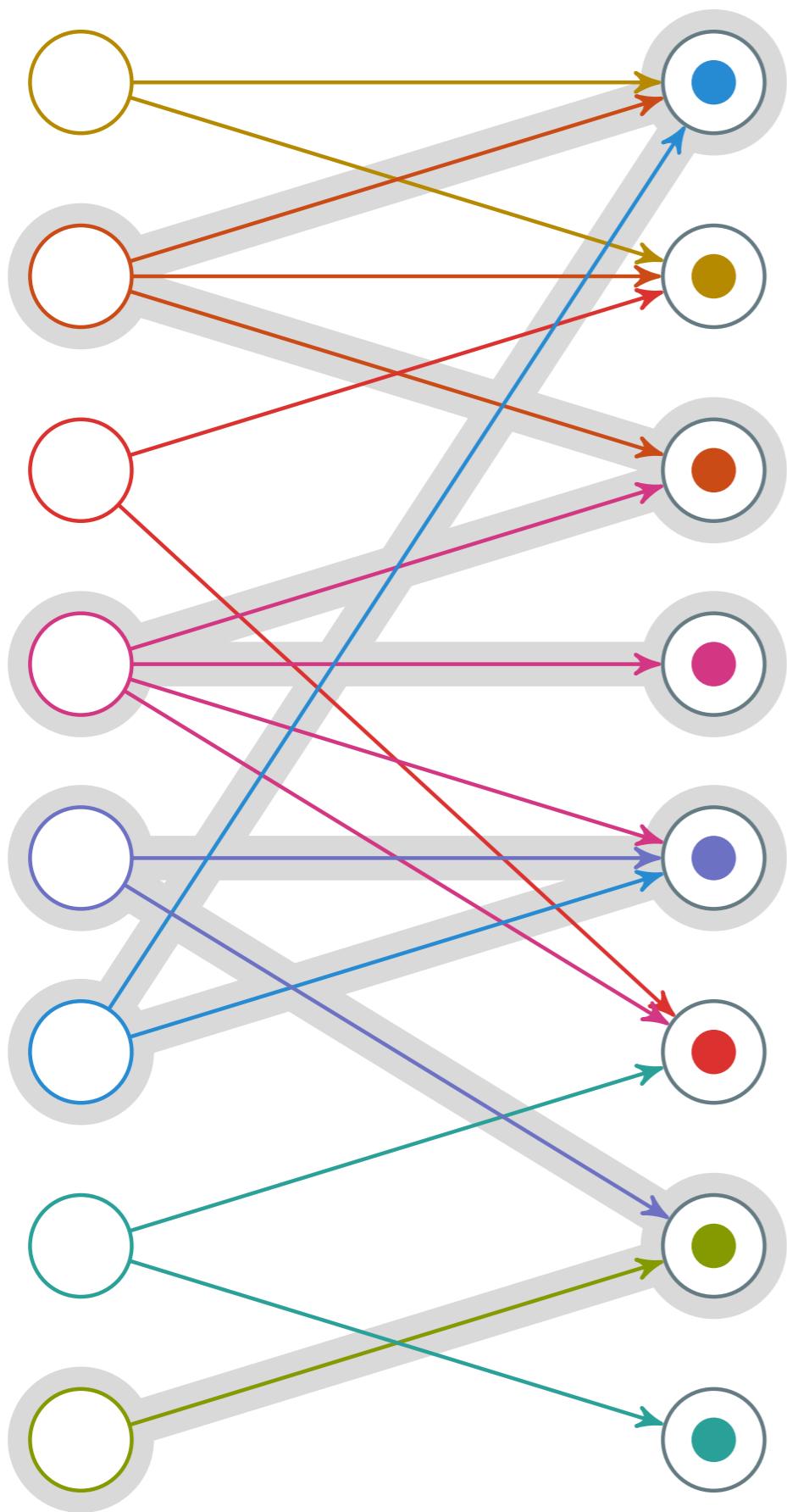


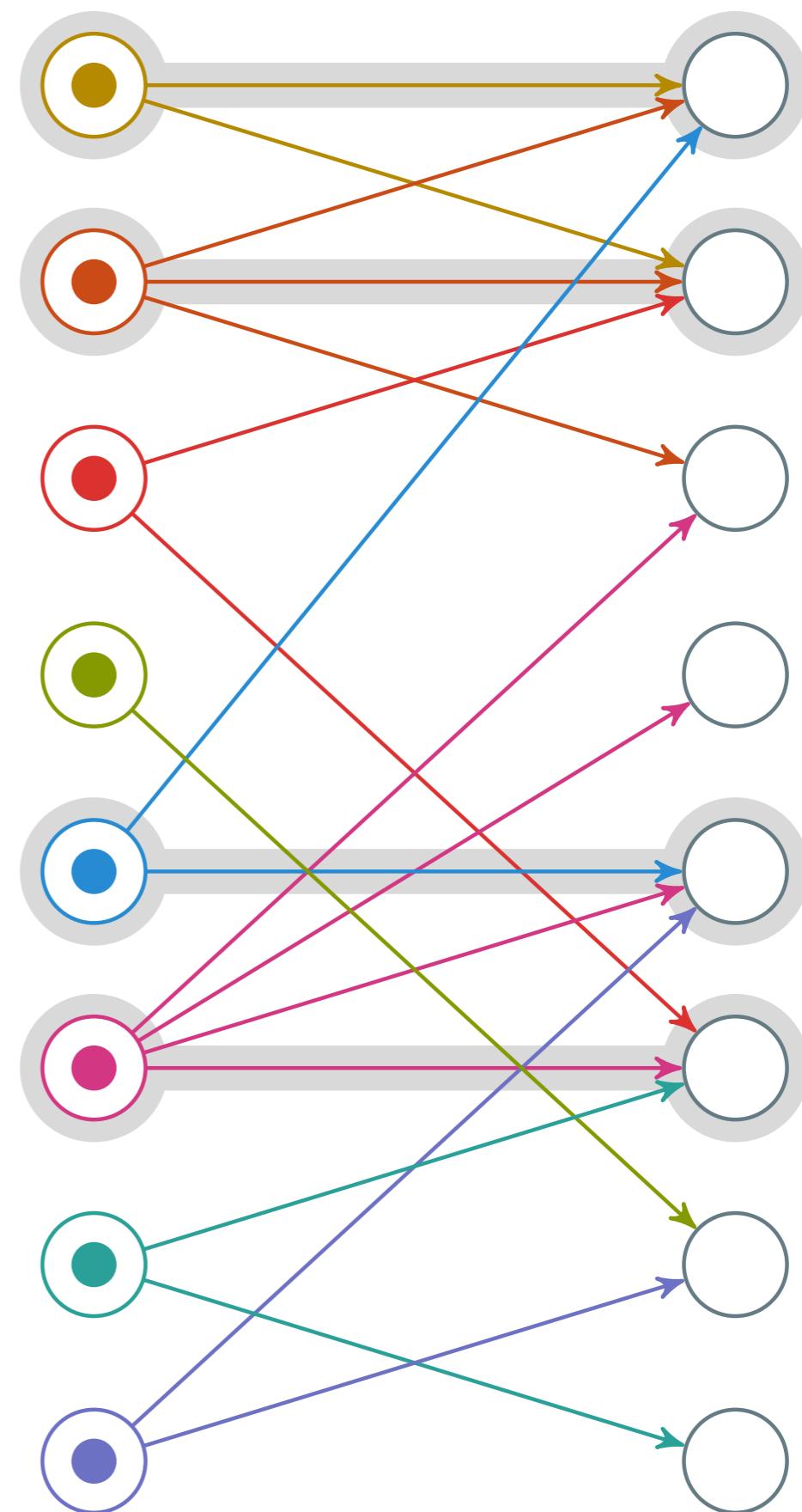
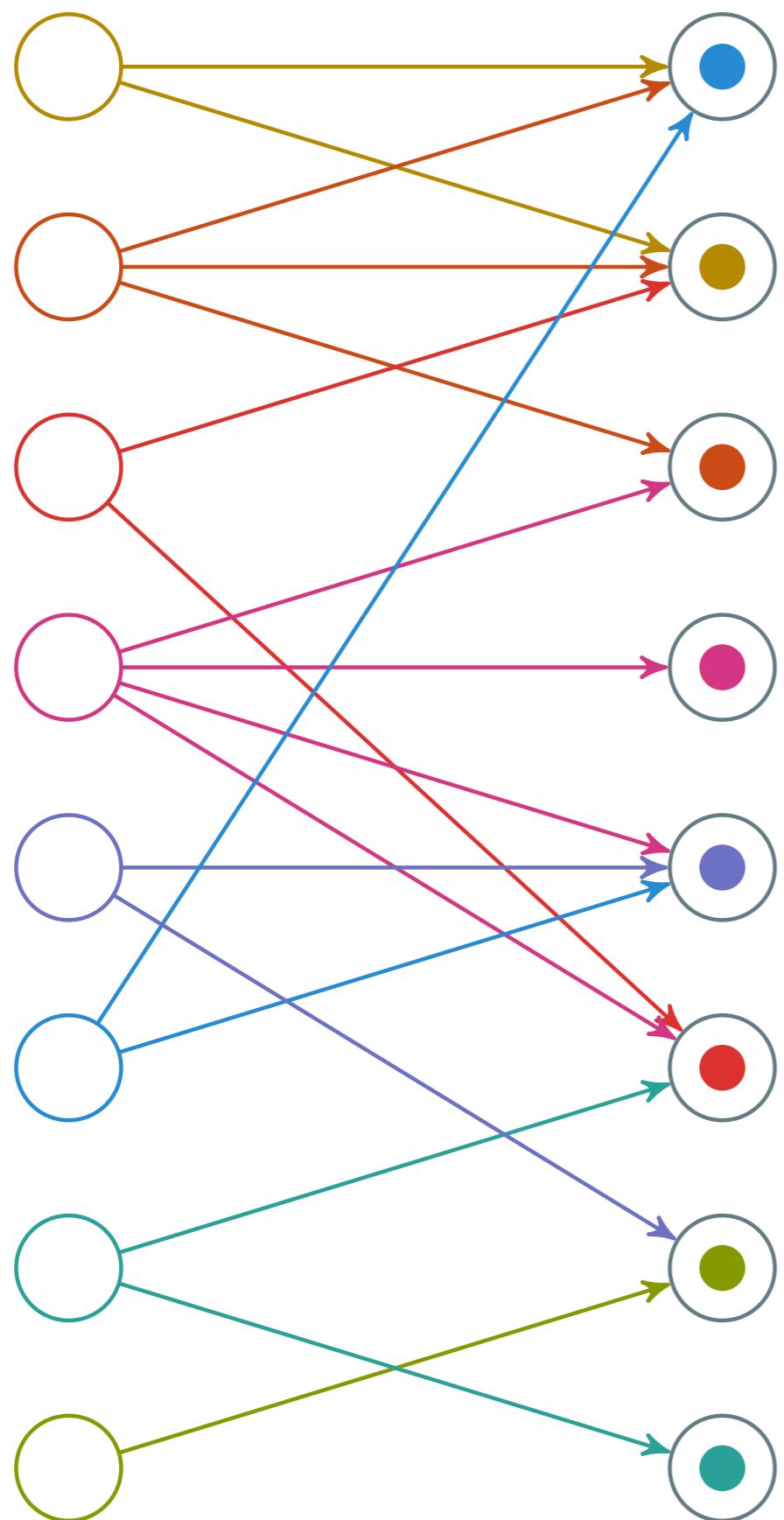












Ford-Fulkerson



Brute force



0

100 %

$$n = 8$$

Ford-Fulkerson



Brute force



0

100 %

$$n = 8$$

Ford-Fulkerson



Brute force



0

100 %

$$n = 8$$

Ford-Fulkerson



Brute force



0

100 %

$$n = 8$$

Her trengte Ford-Fulkerson 70 operasjoner, mens det var 40320 permutasjoner å teste.

Ford-Fulkerson



Brute force



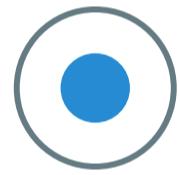
0

100 %

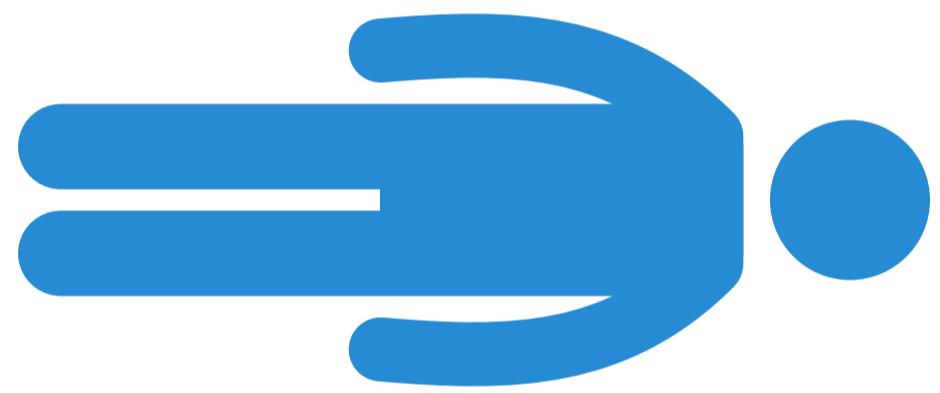
$$n = 8$$

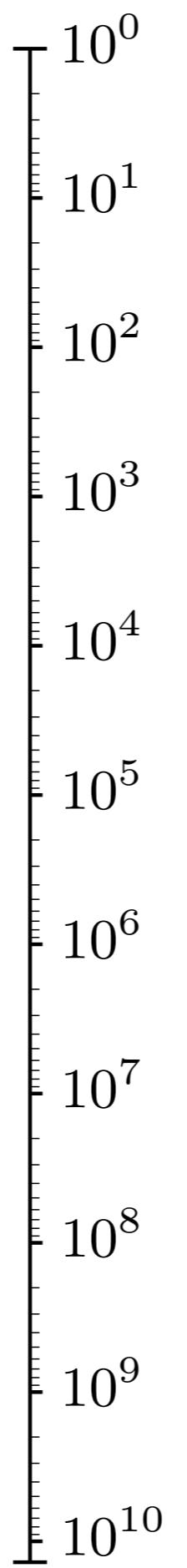
Med andre ord: En dobling av problemet gir ca. dobbel kjøretid for F-F, mens brute force ser ut til å stoppe helt opp.

Og dette var for et leke-eksempel...



135





Rakettlanding

10^0

10^1

10^2

10^3

10^4

10^5

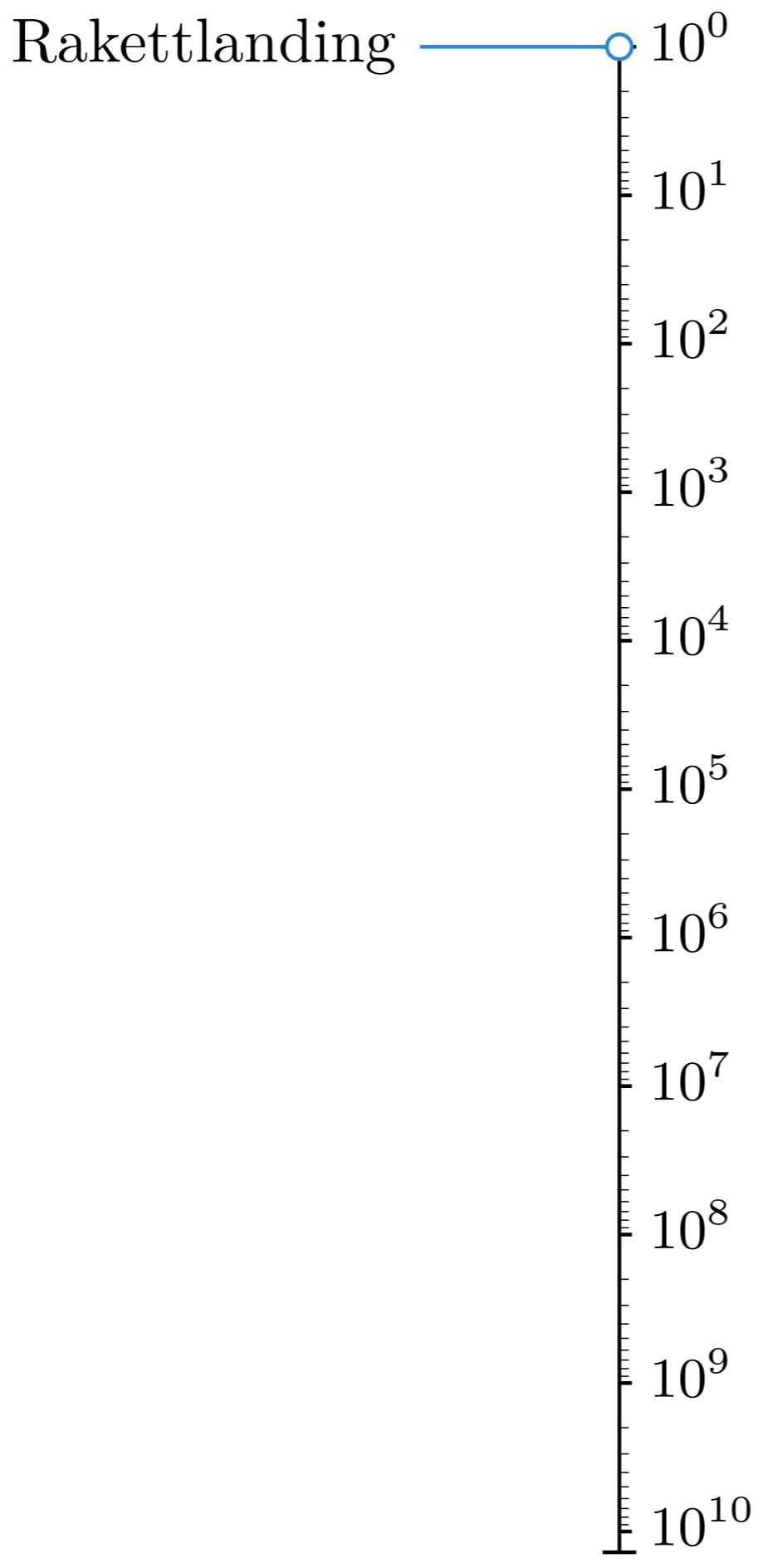
10^6

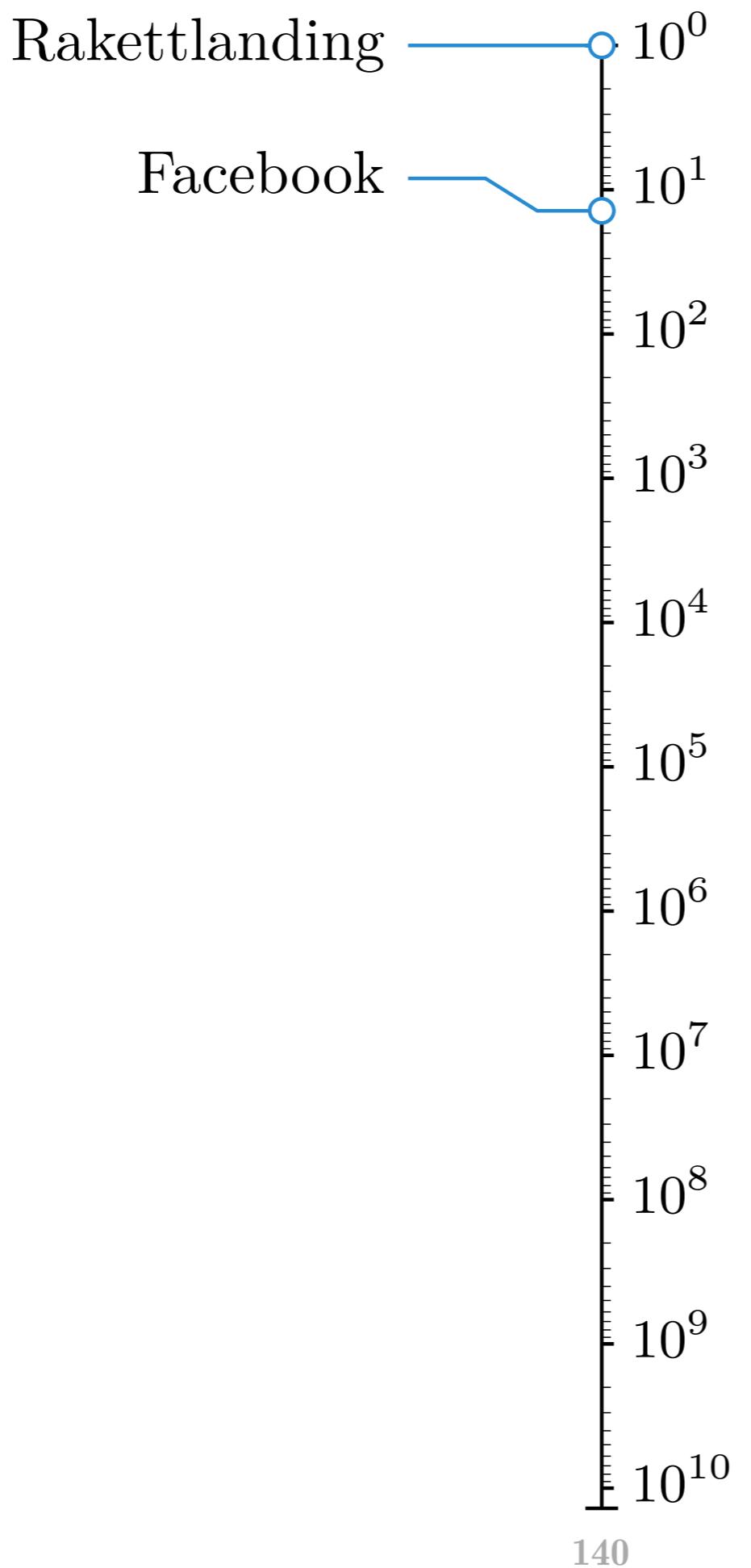
10^7

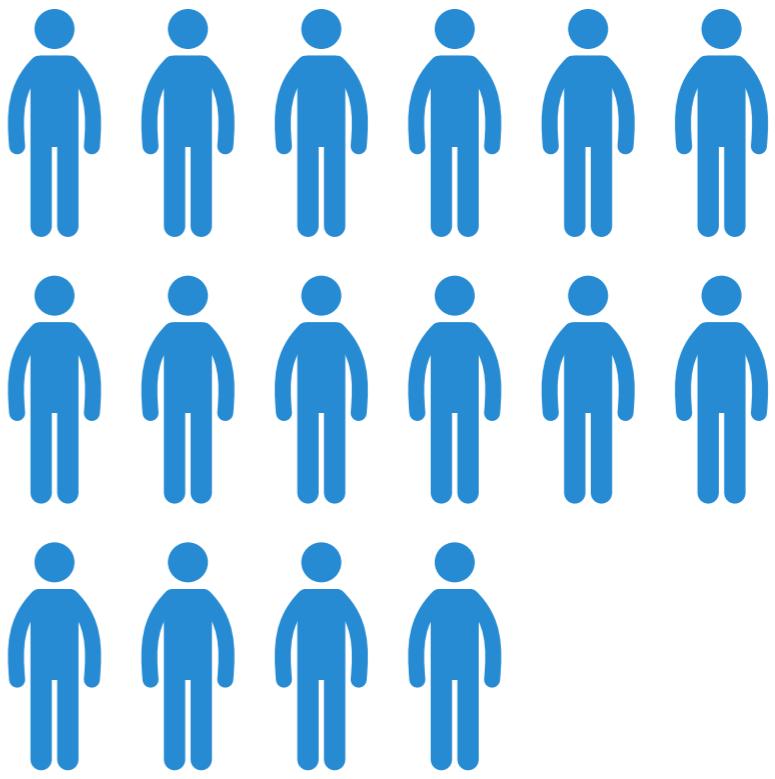
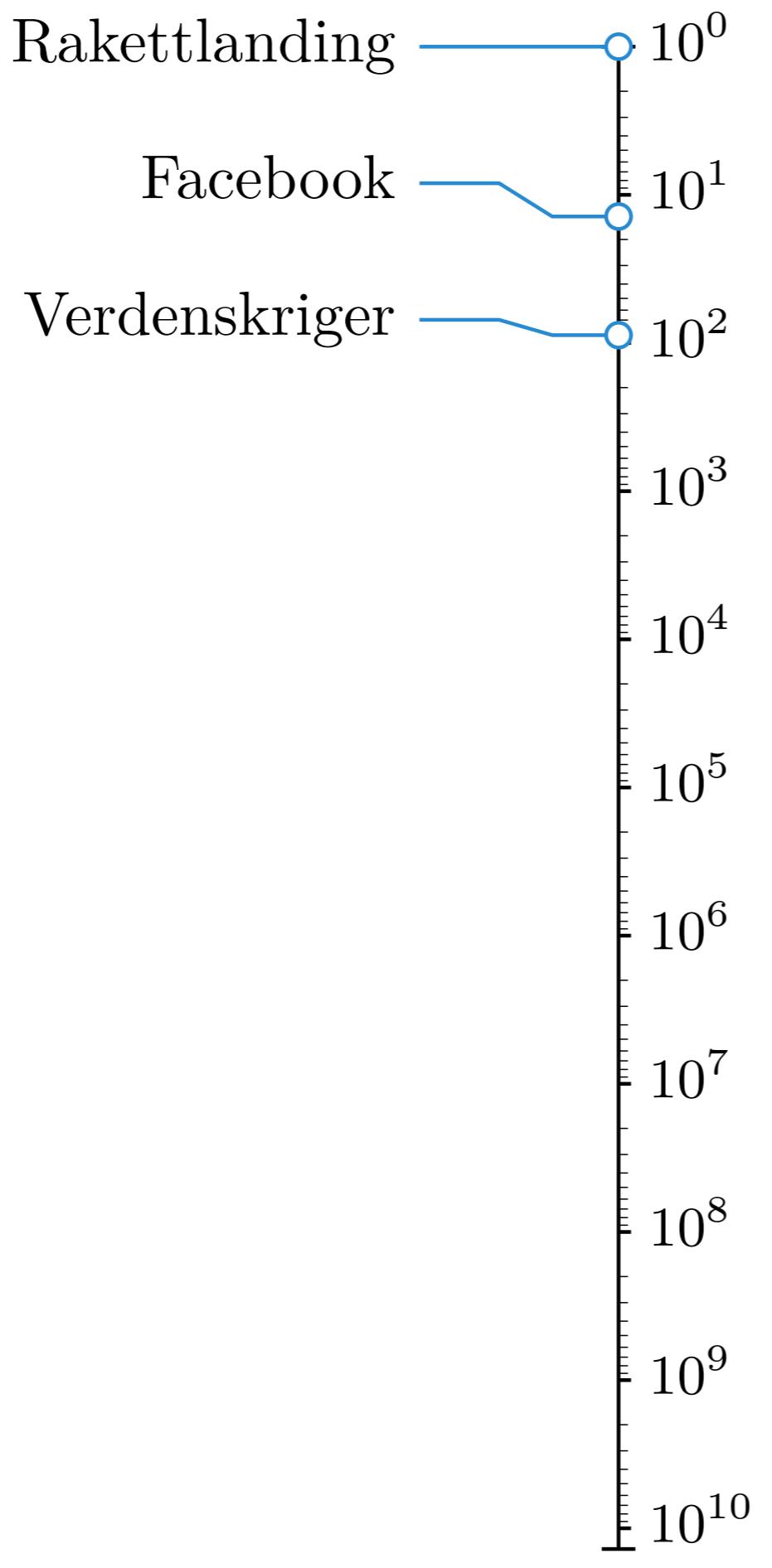
10^8

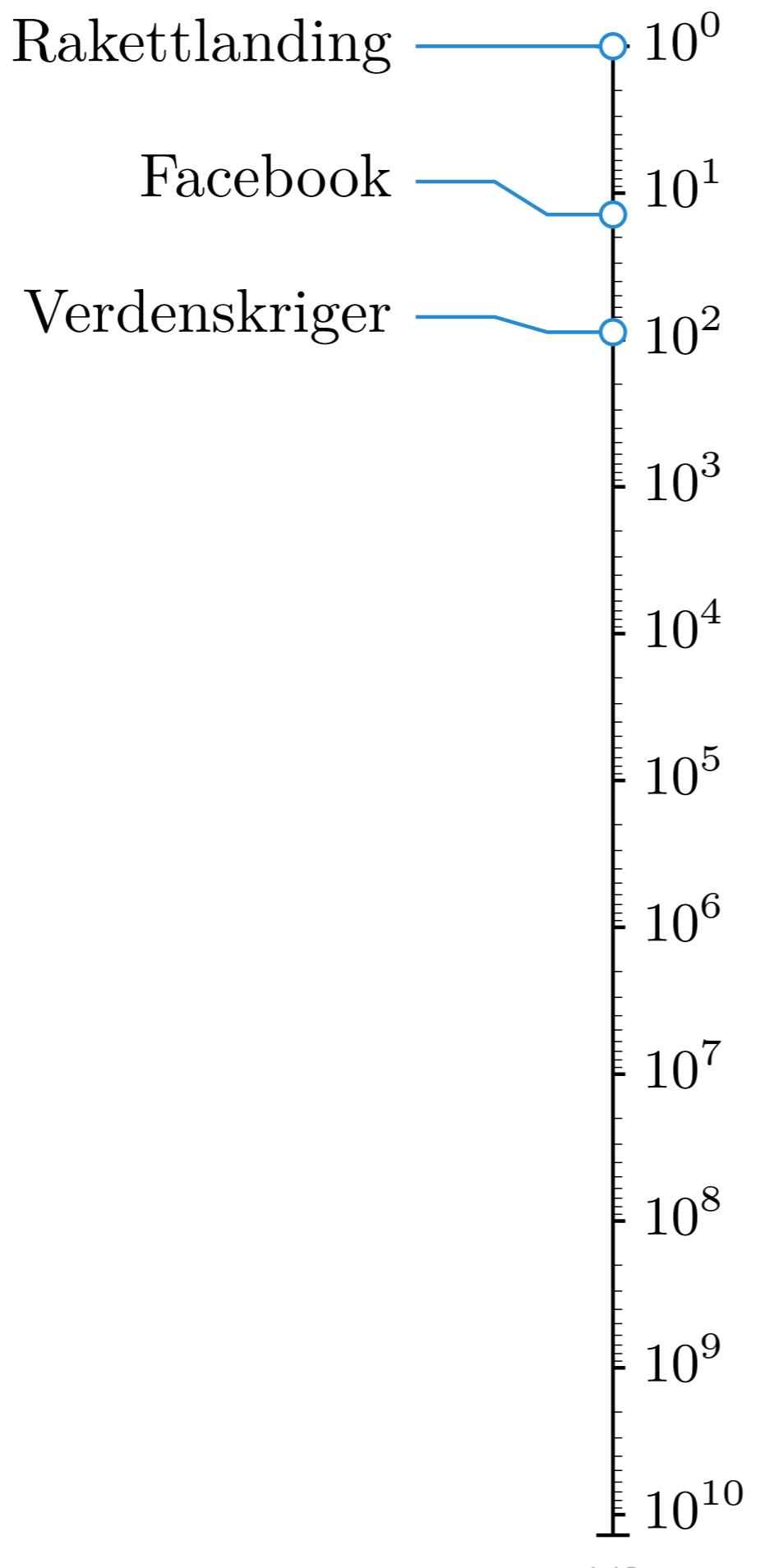
10^9

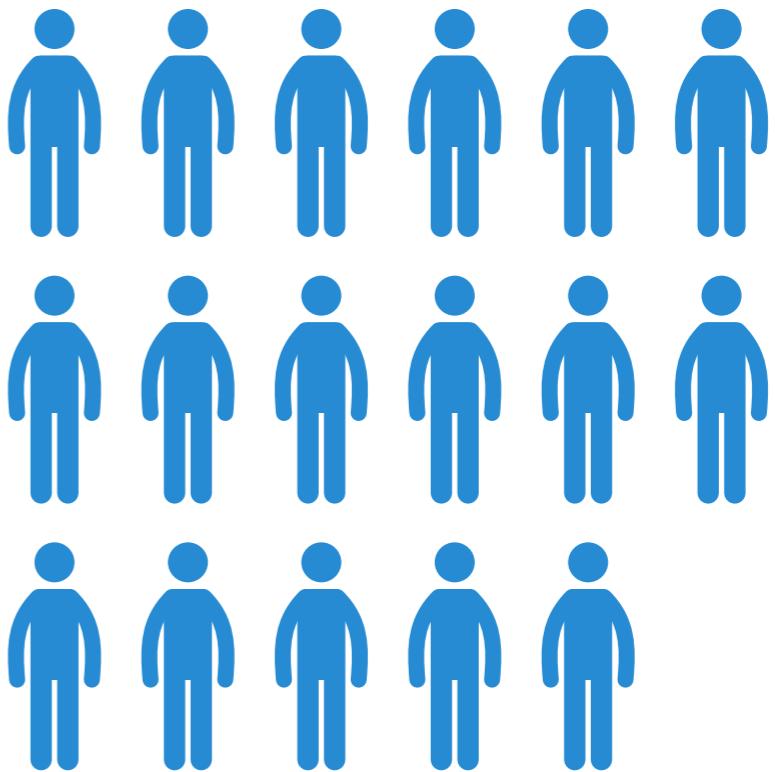
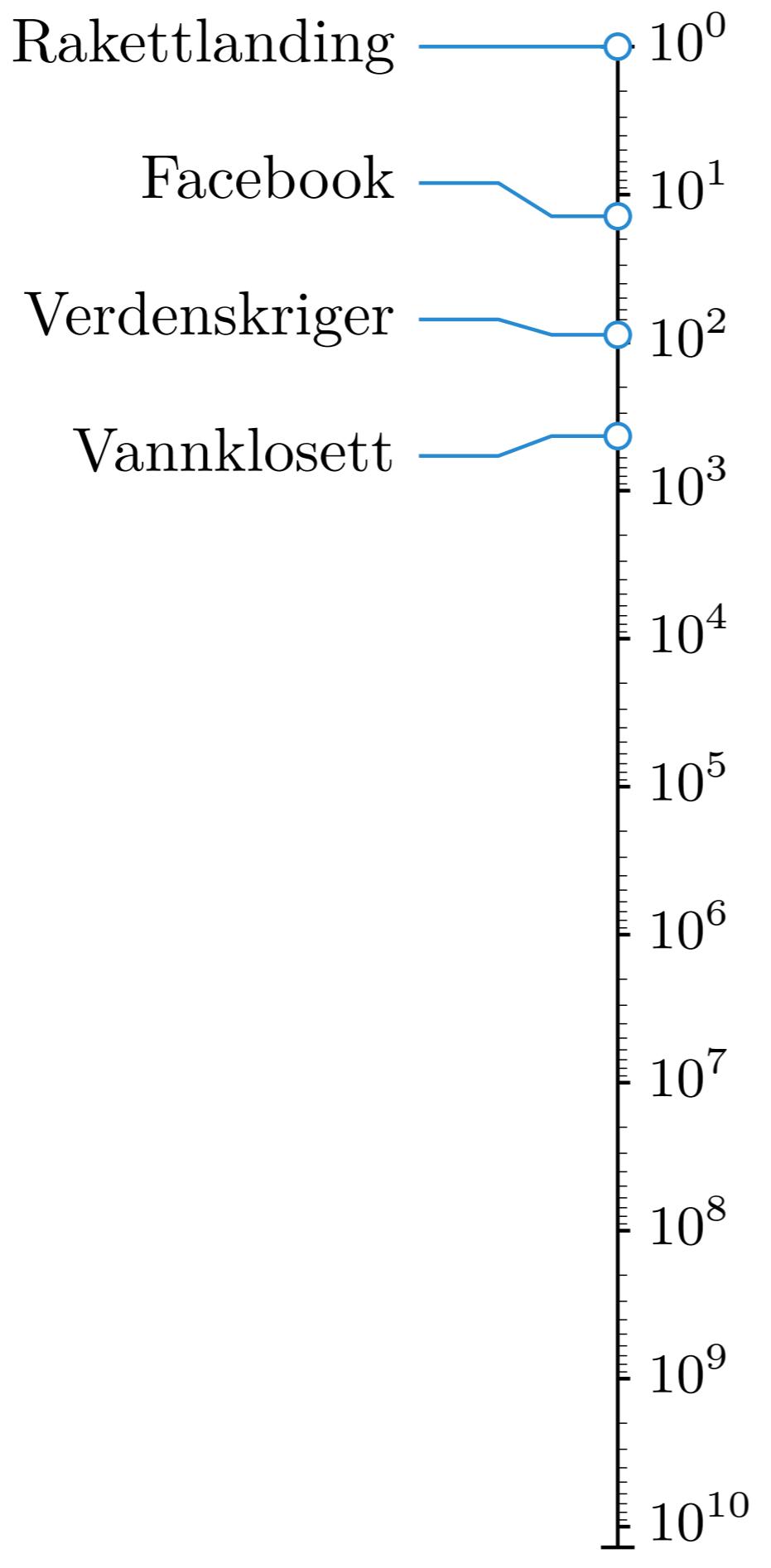
10^{10}

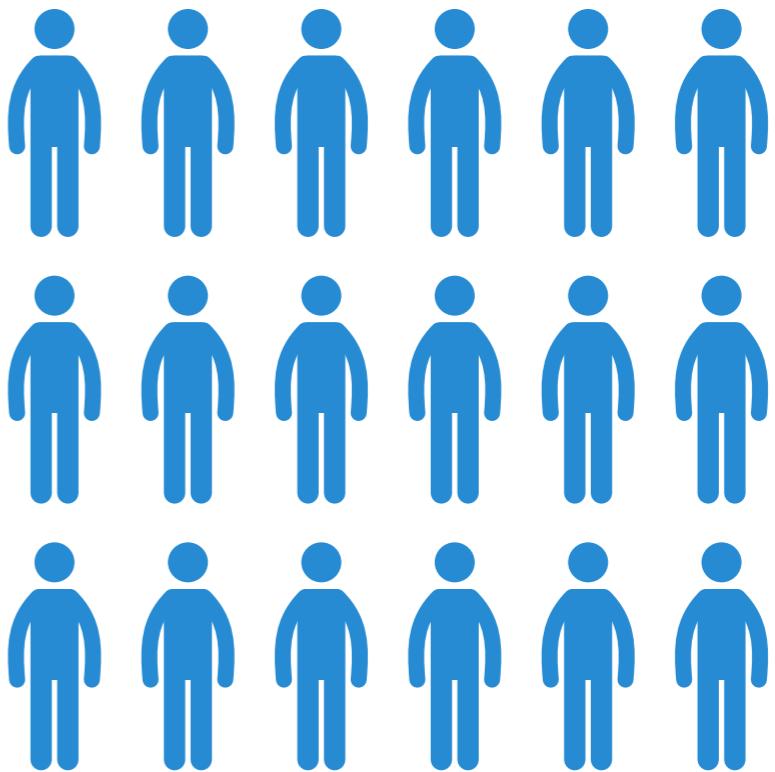
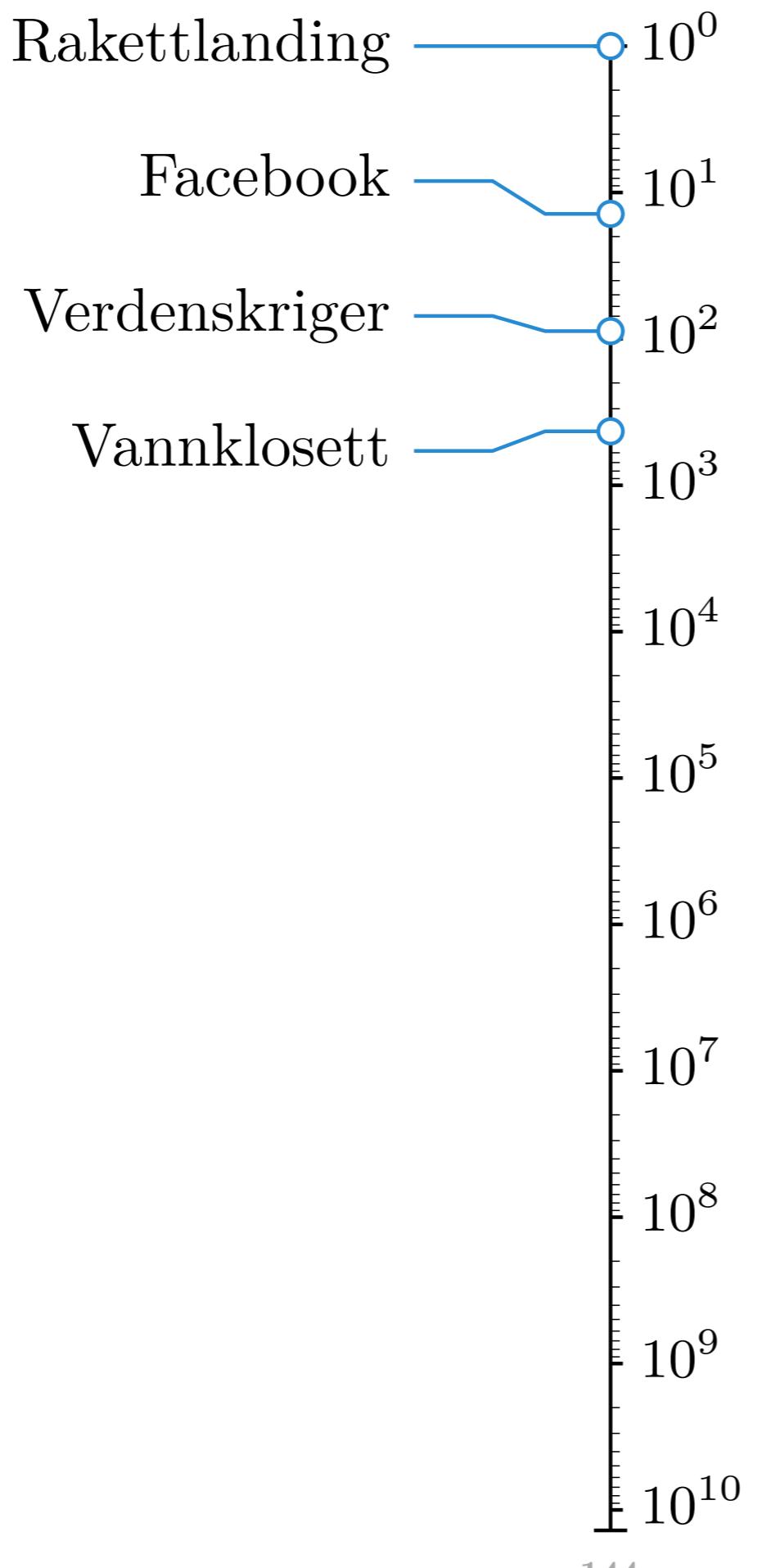


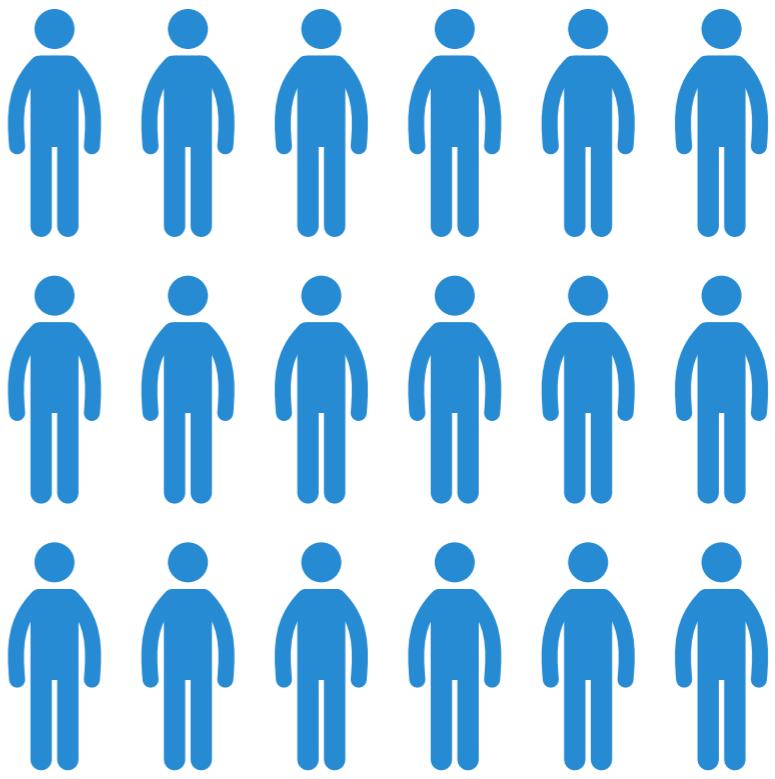
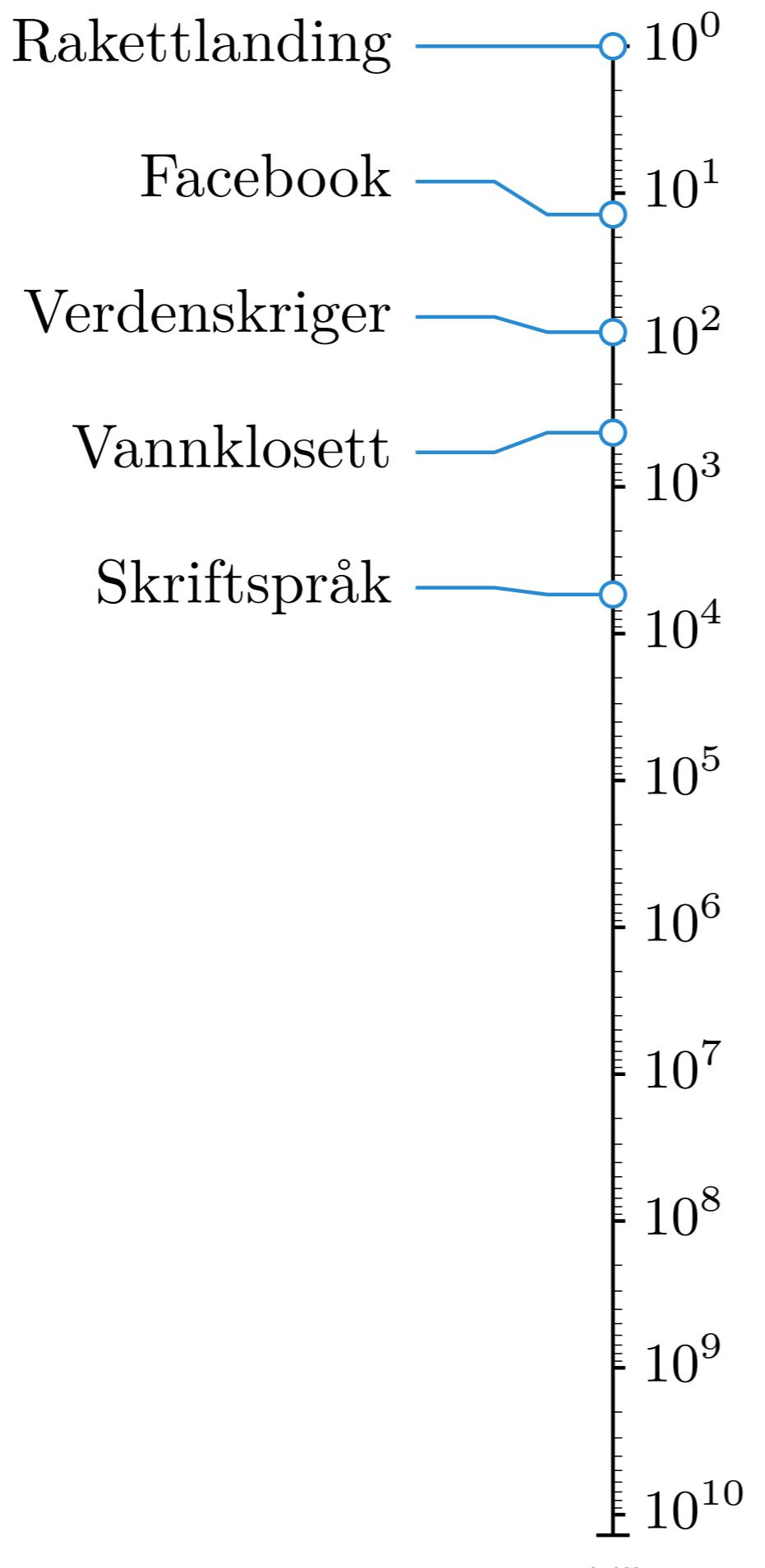


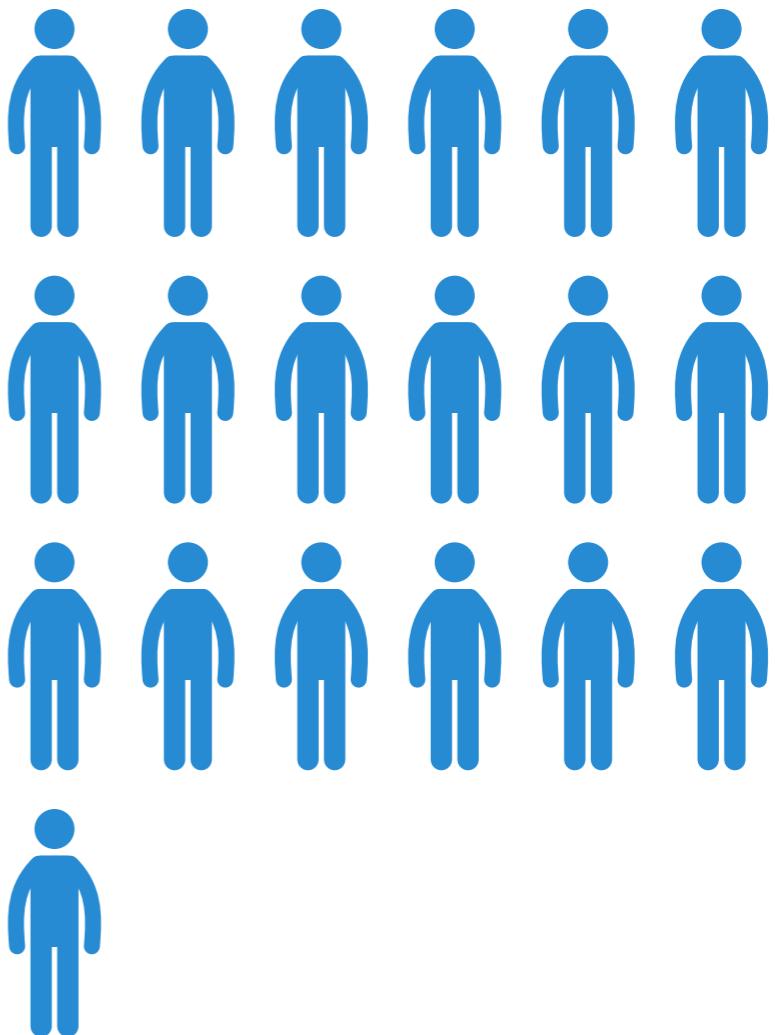
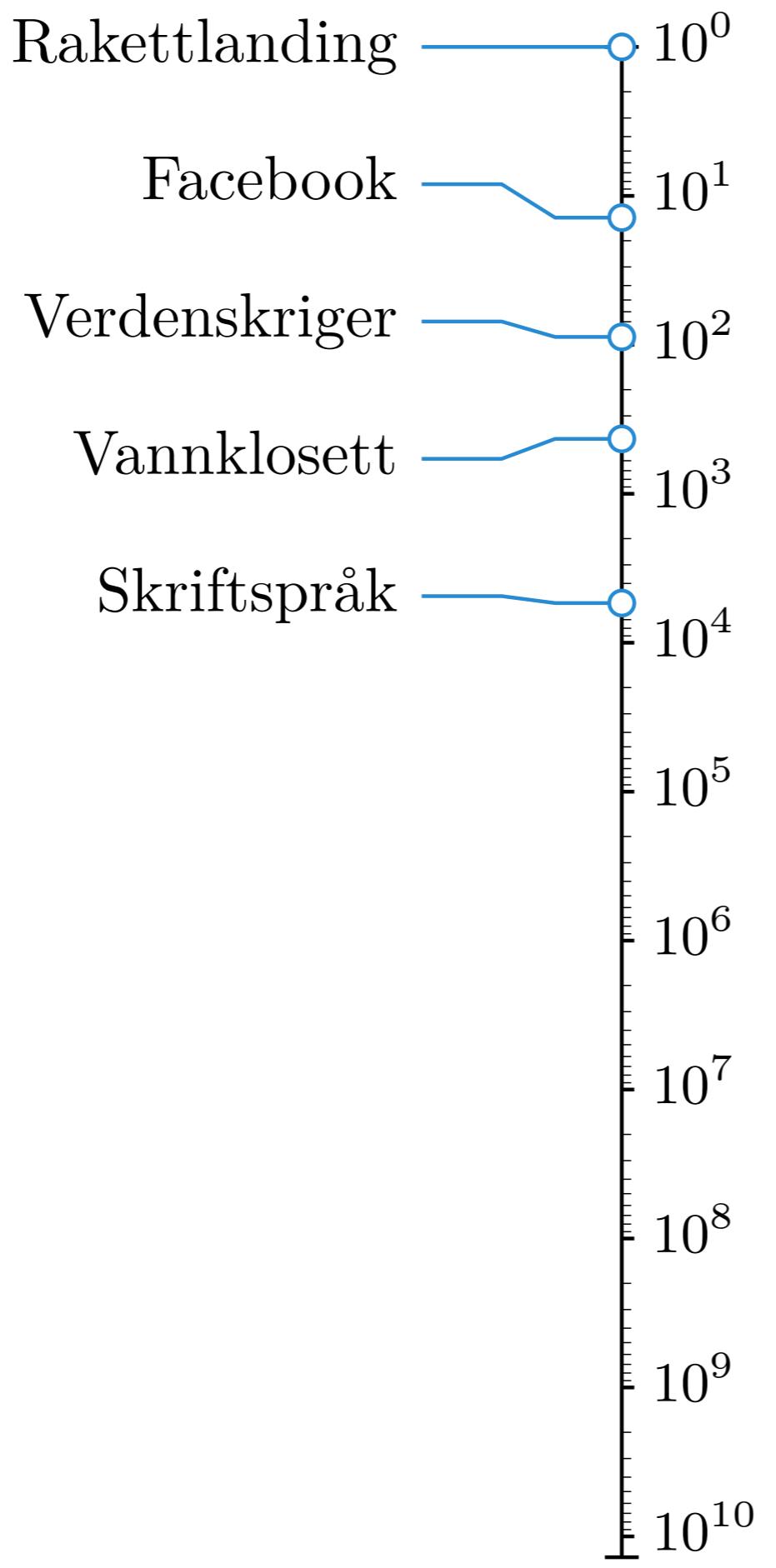


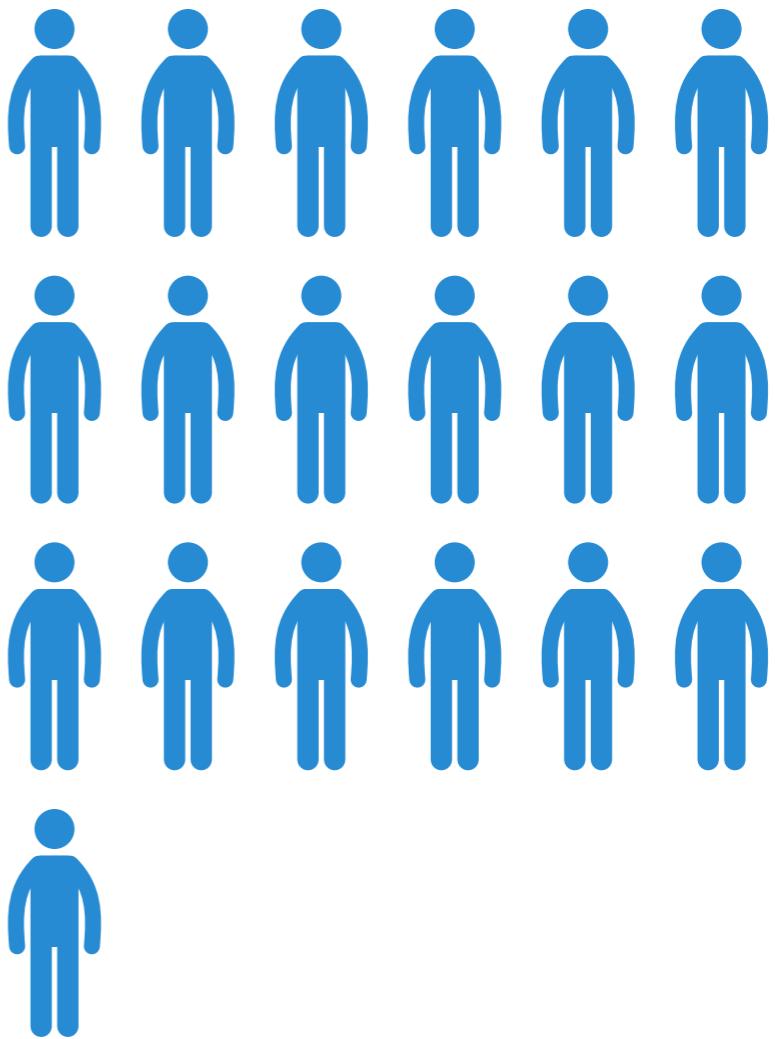
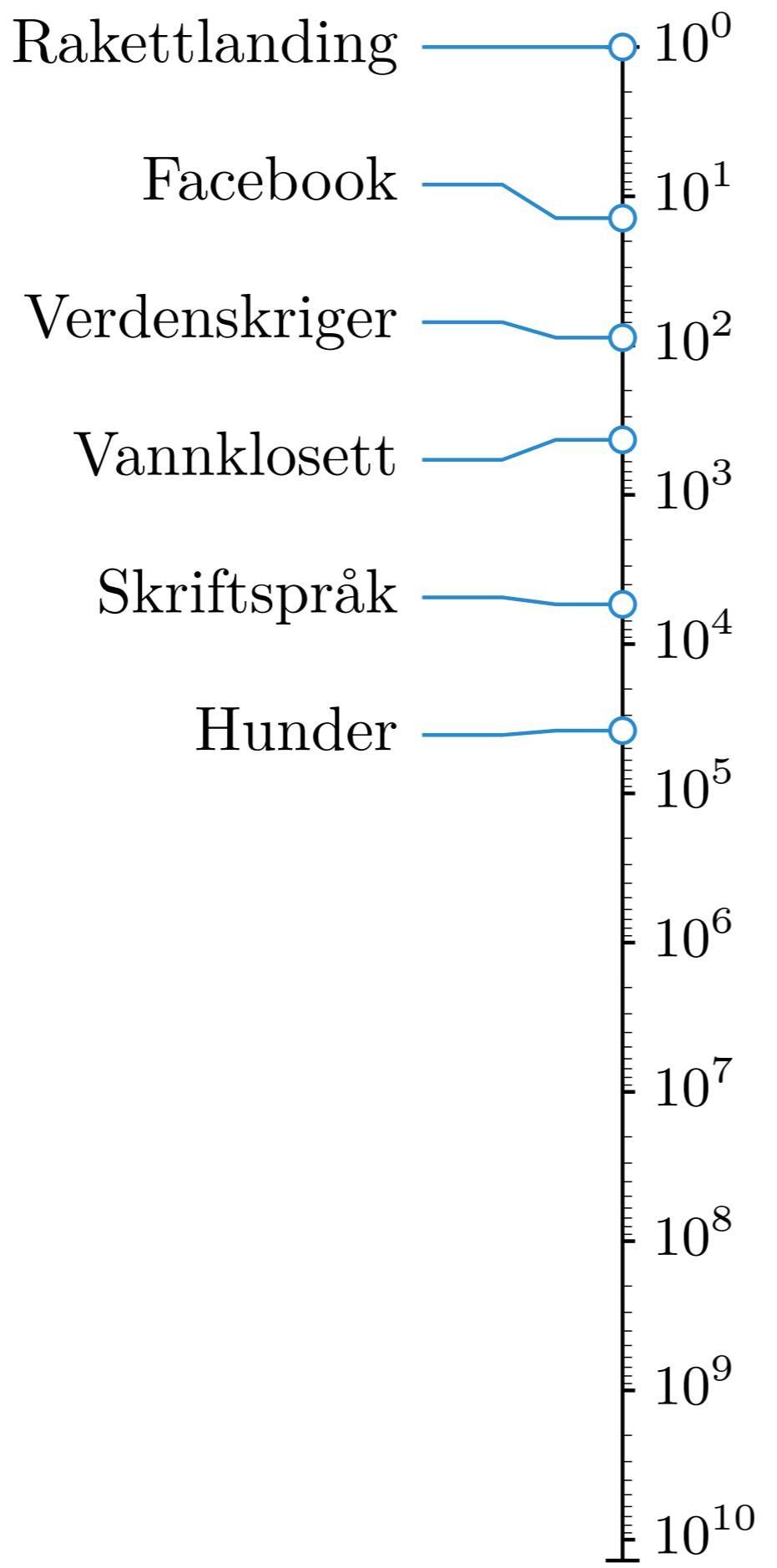


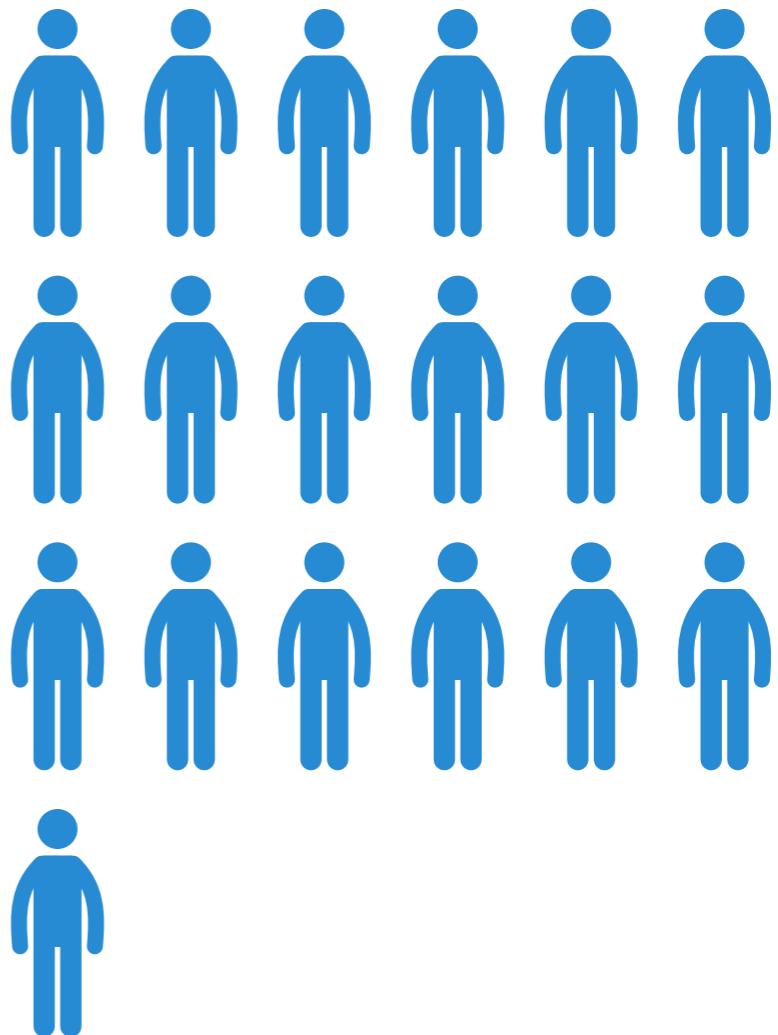
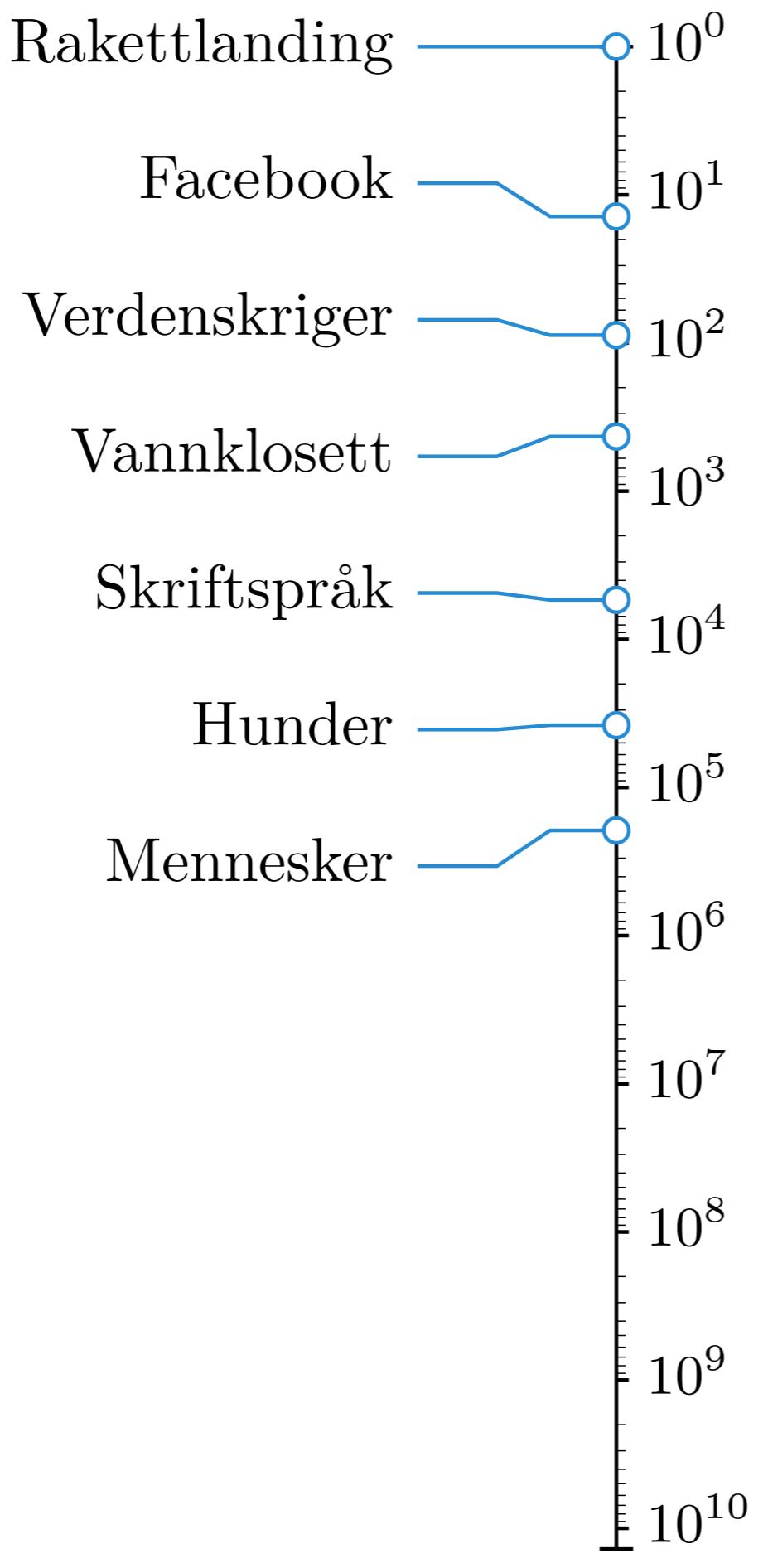


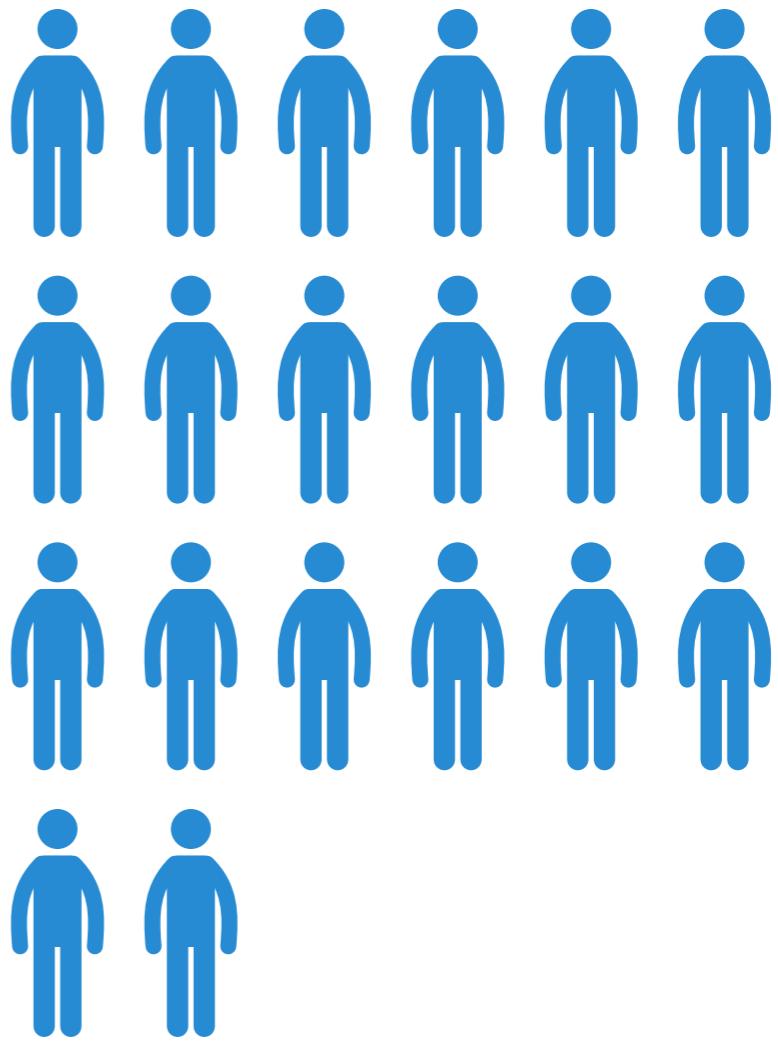
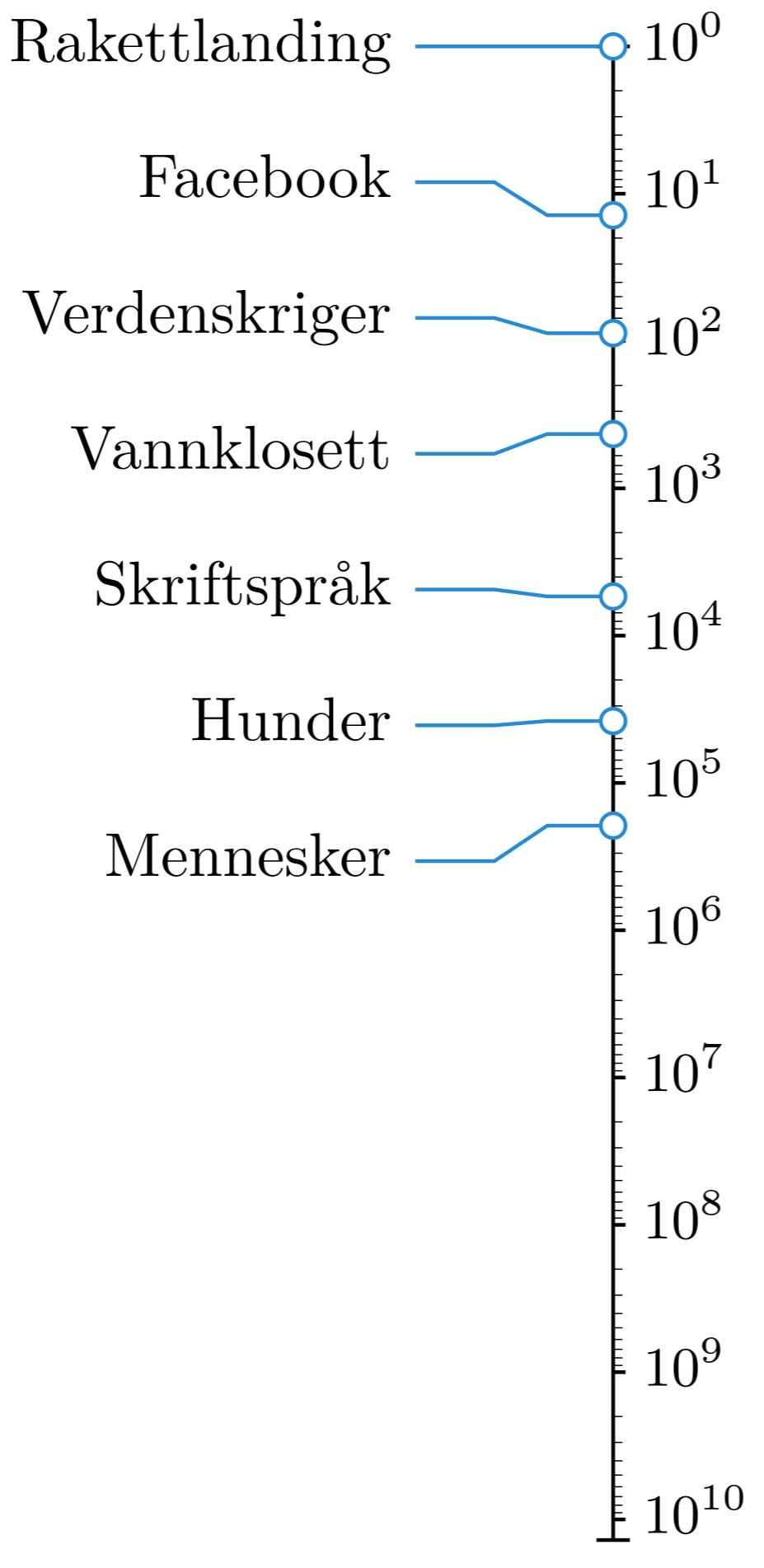


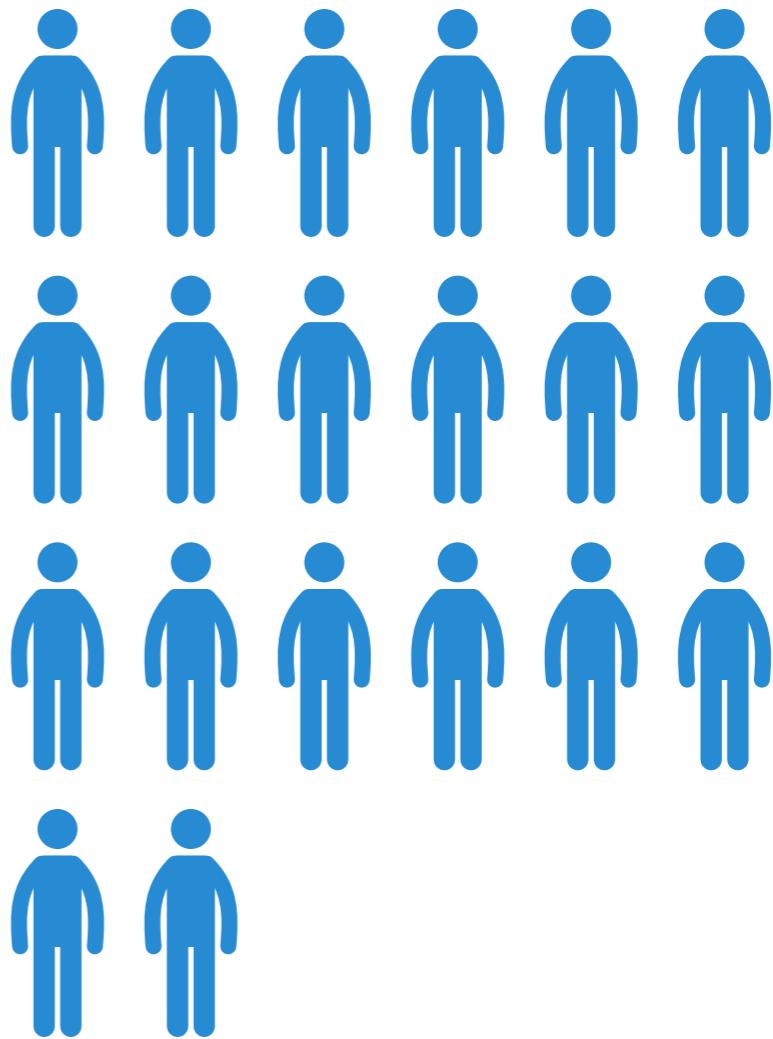
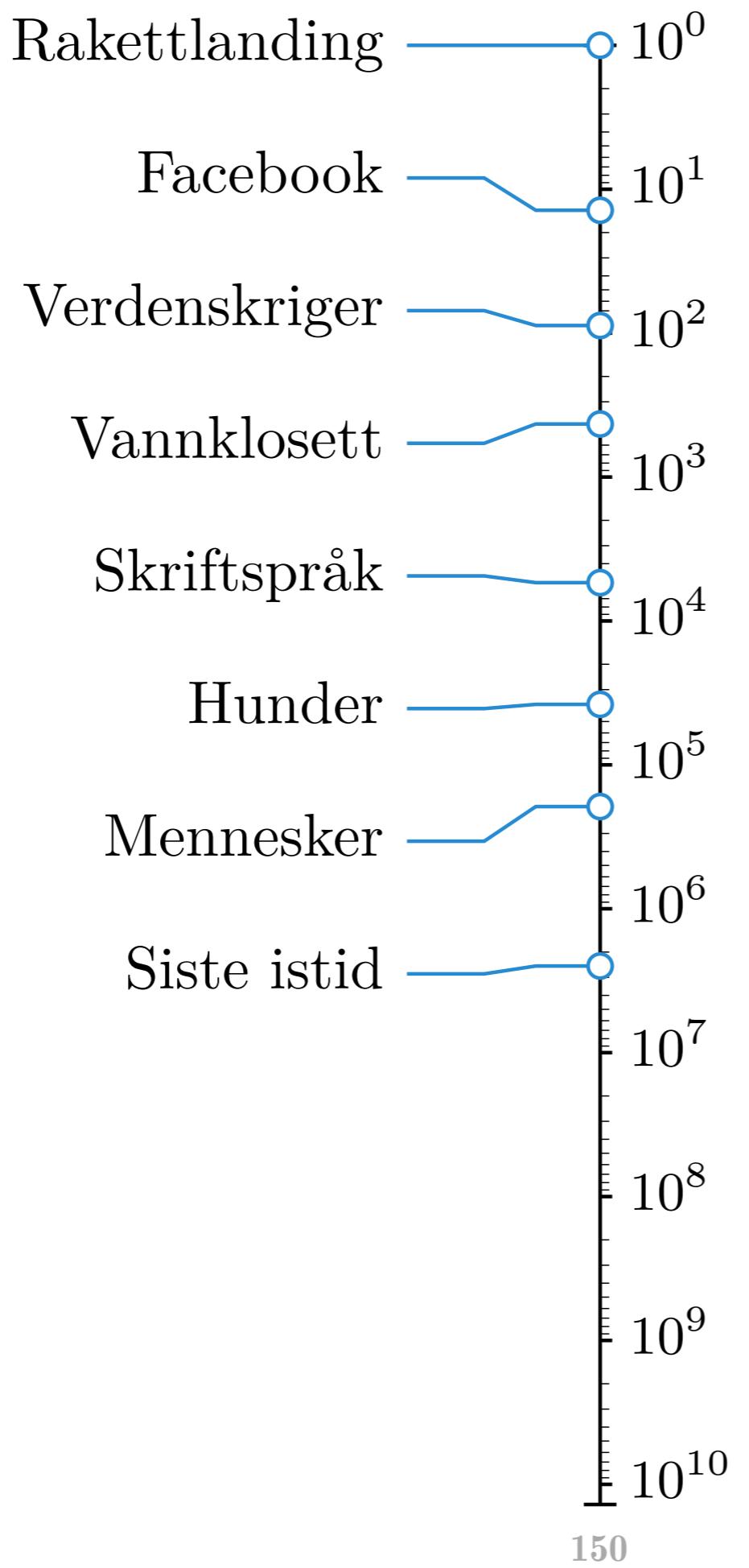


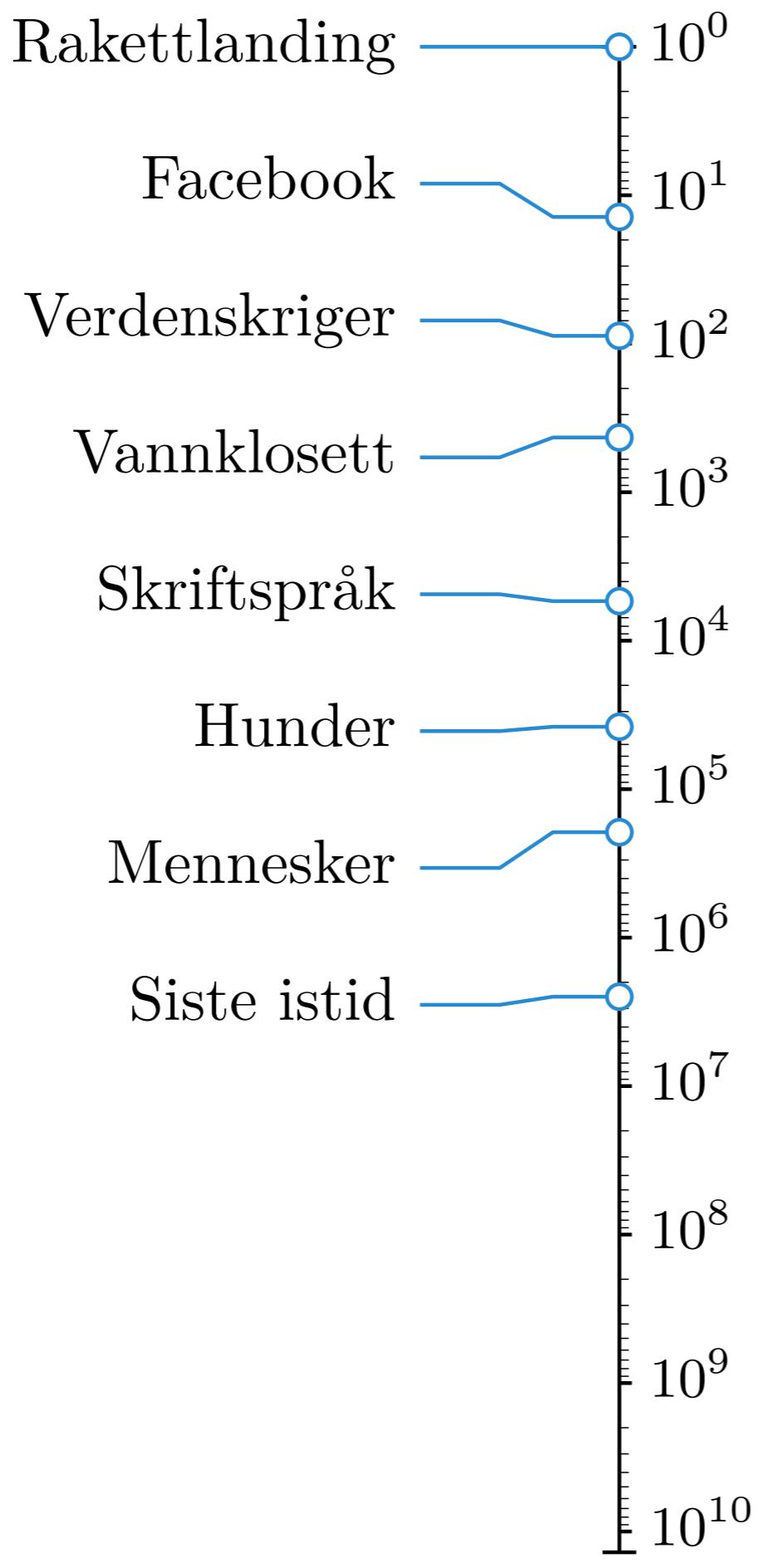


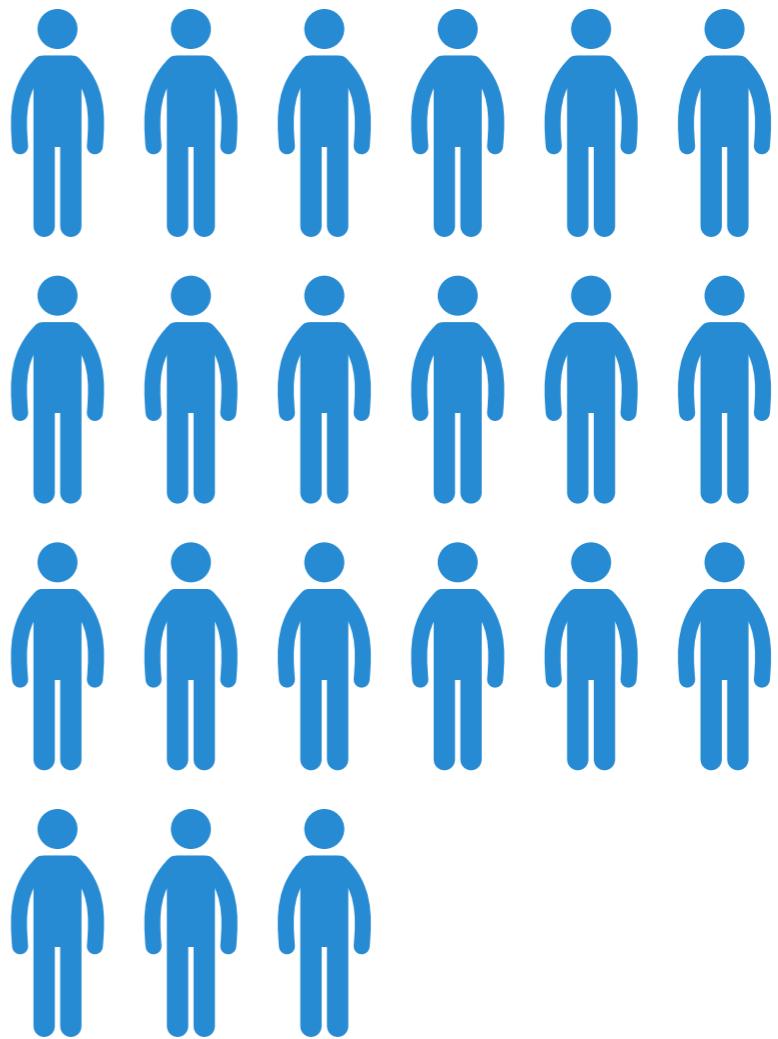
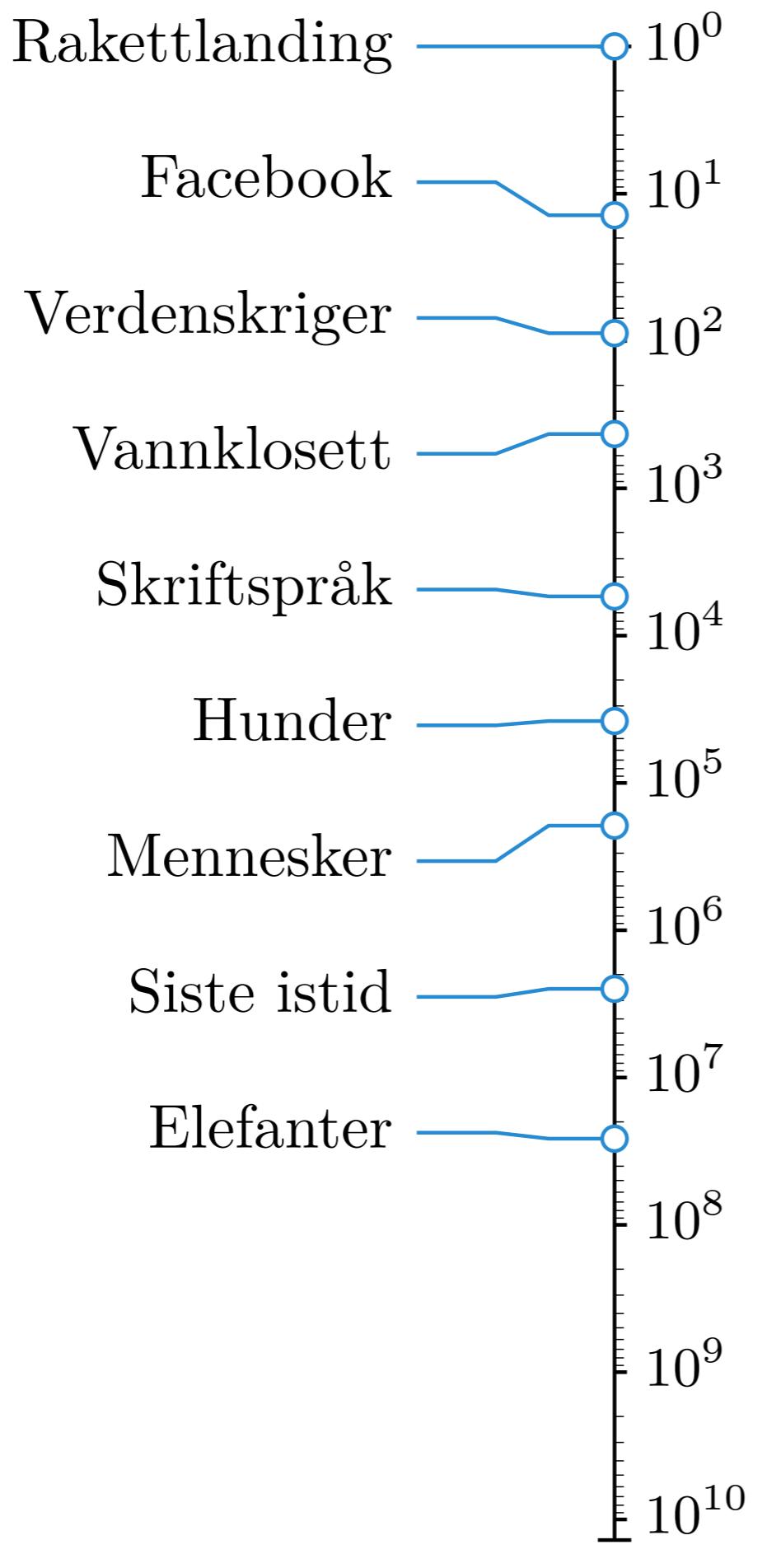


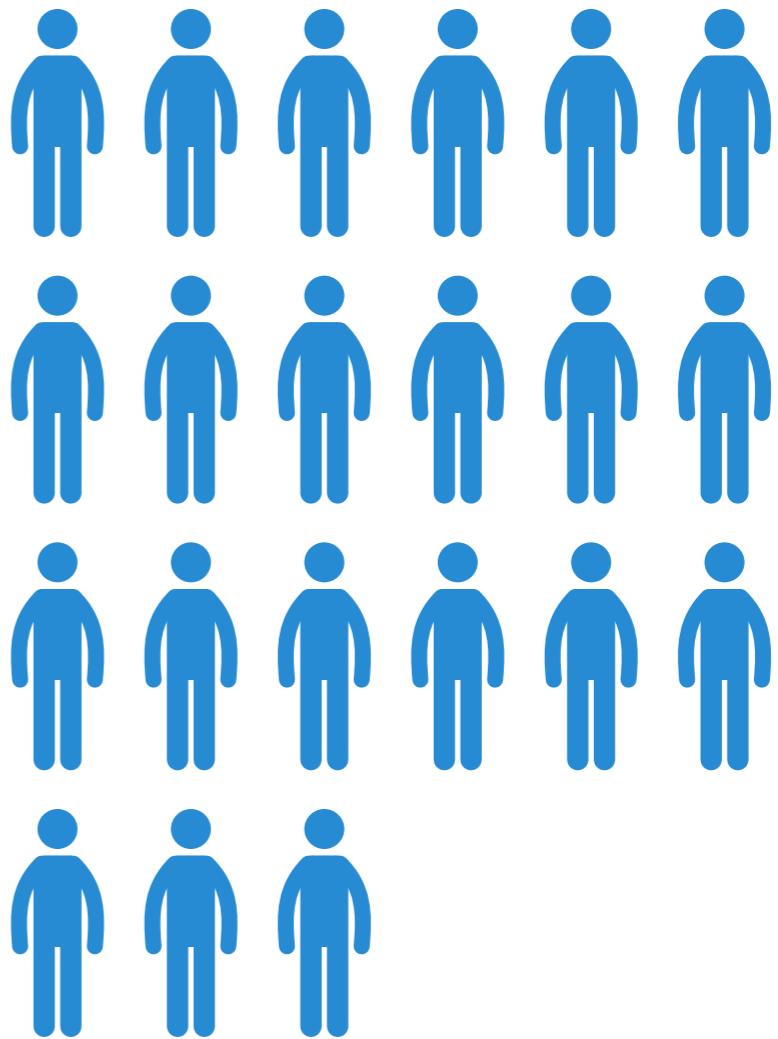
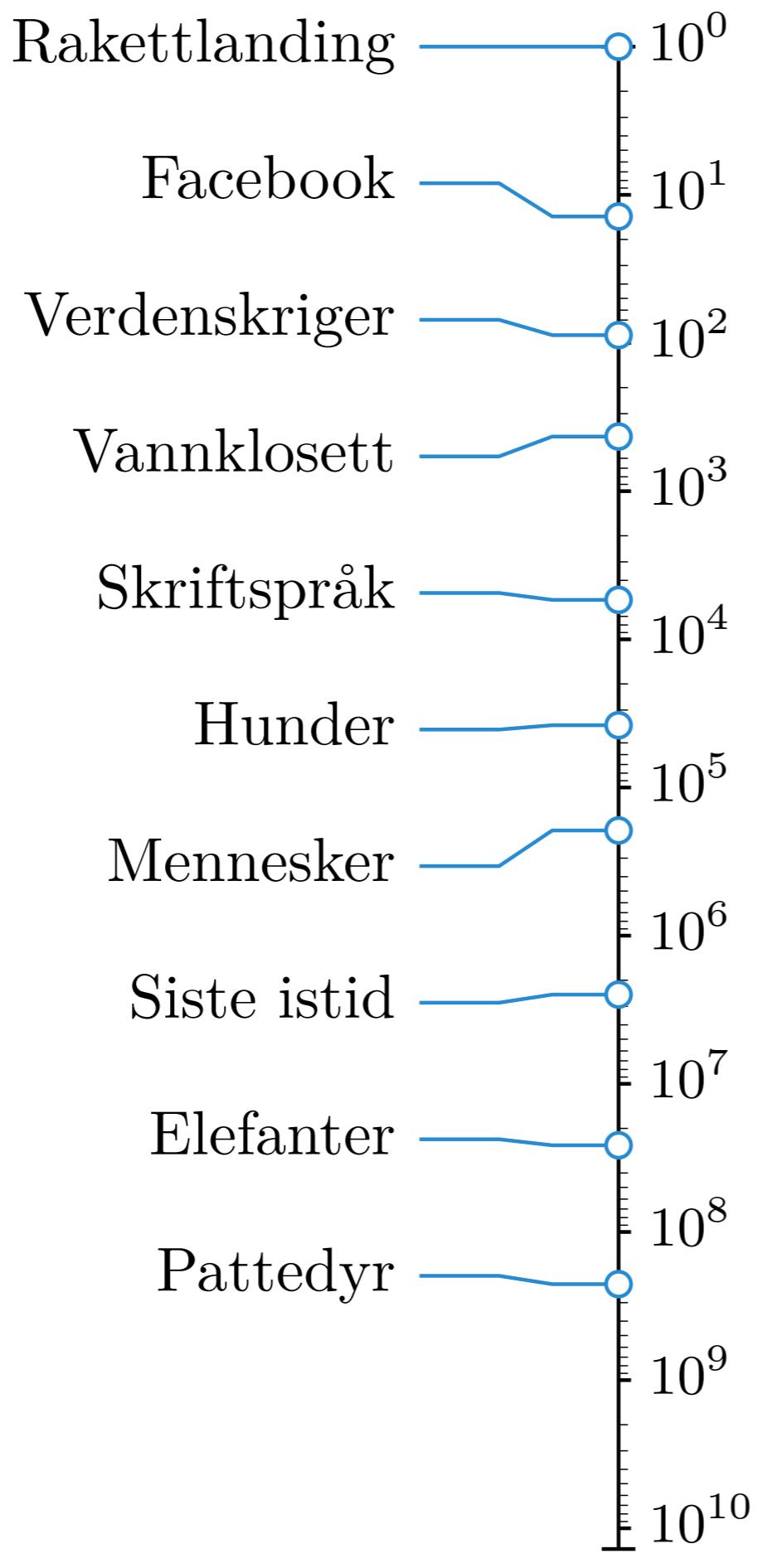


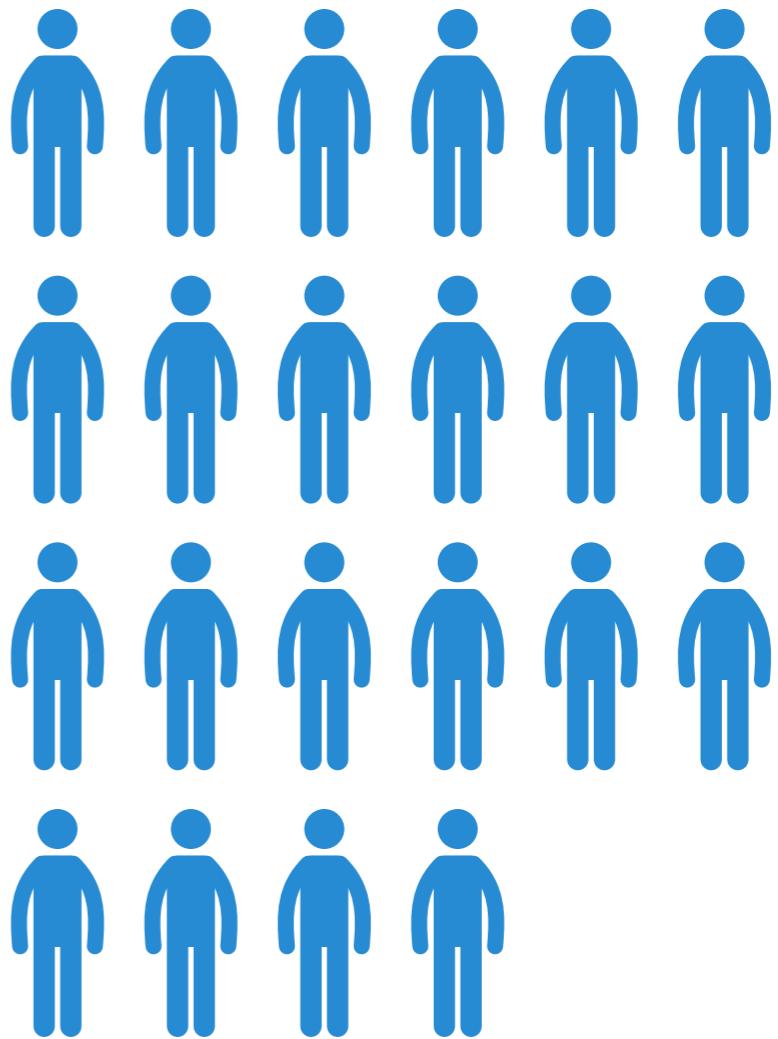
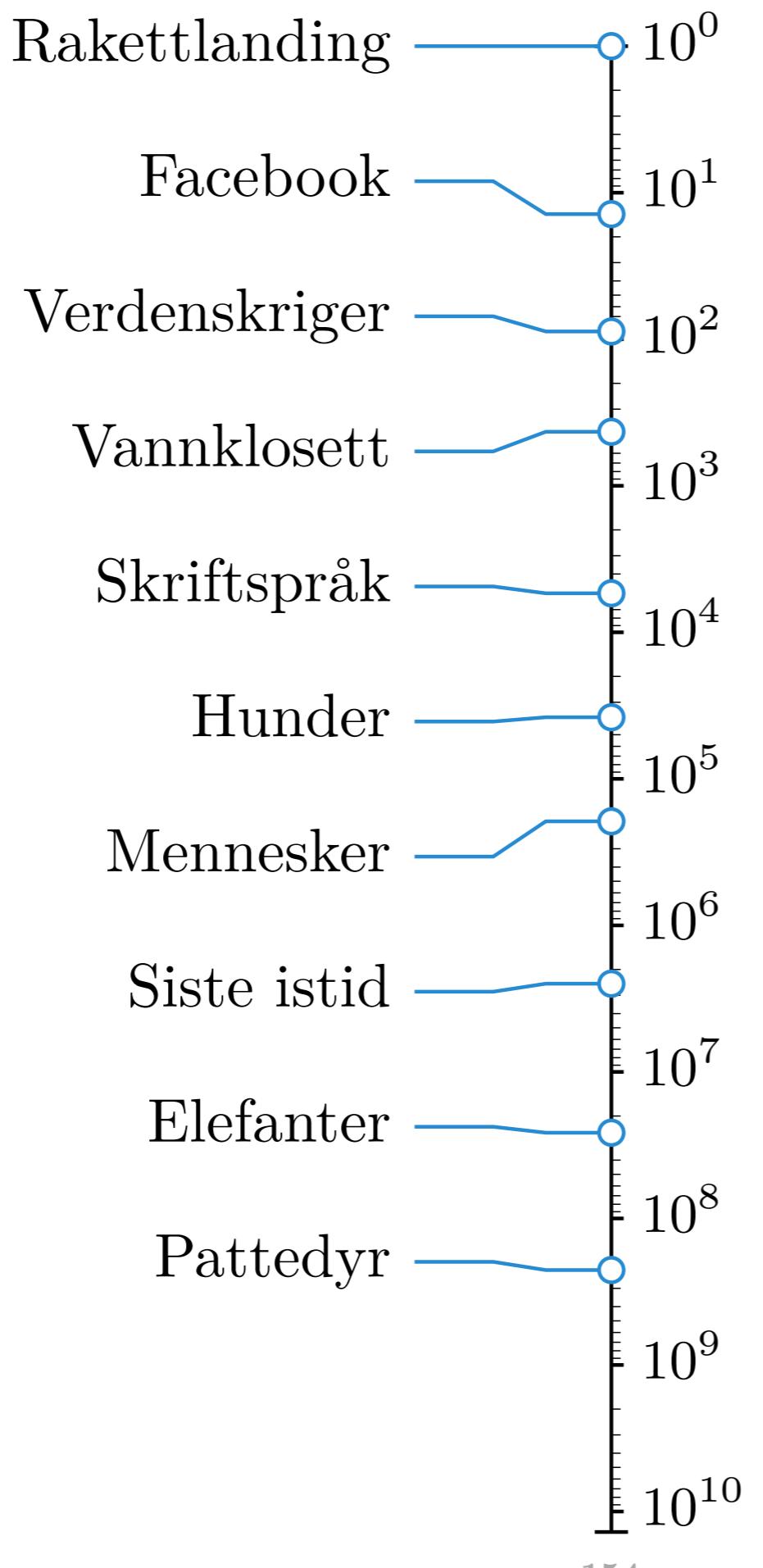


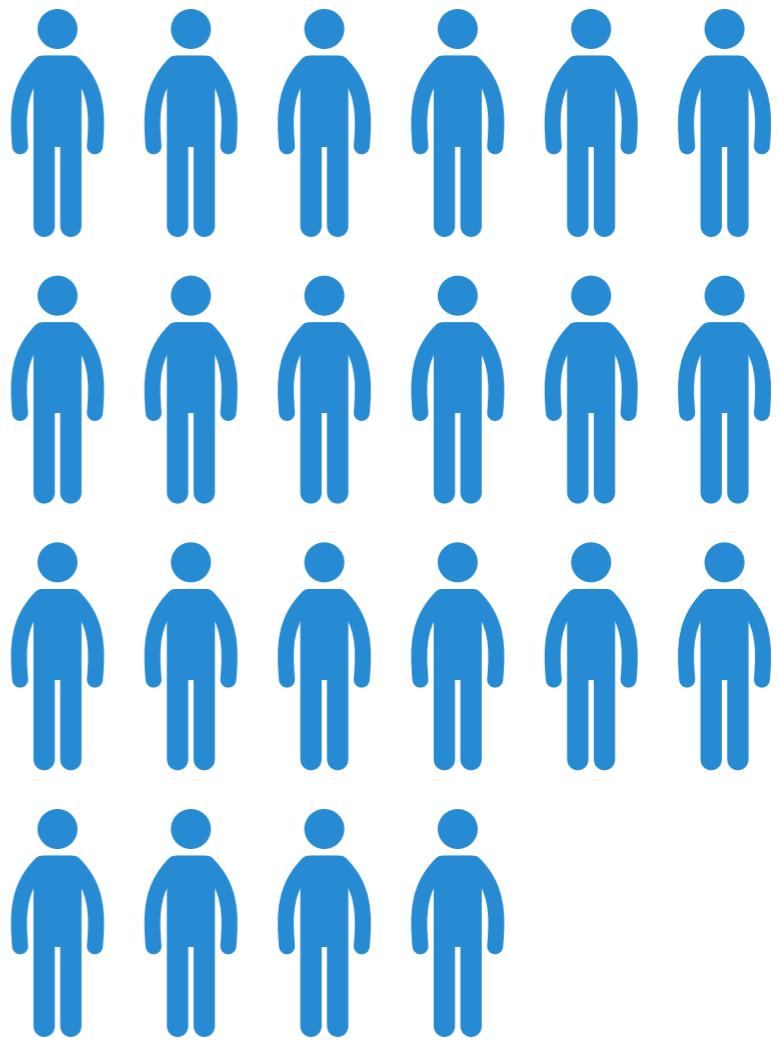
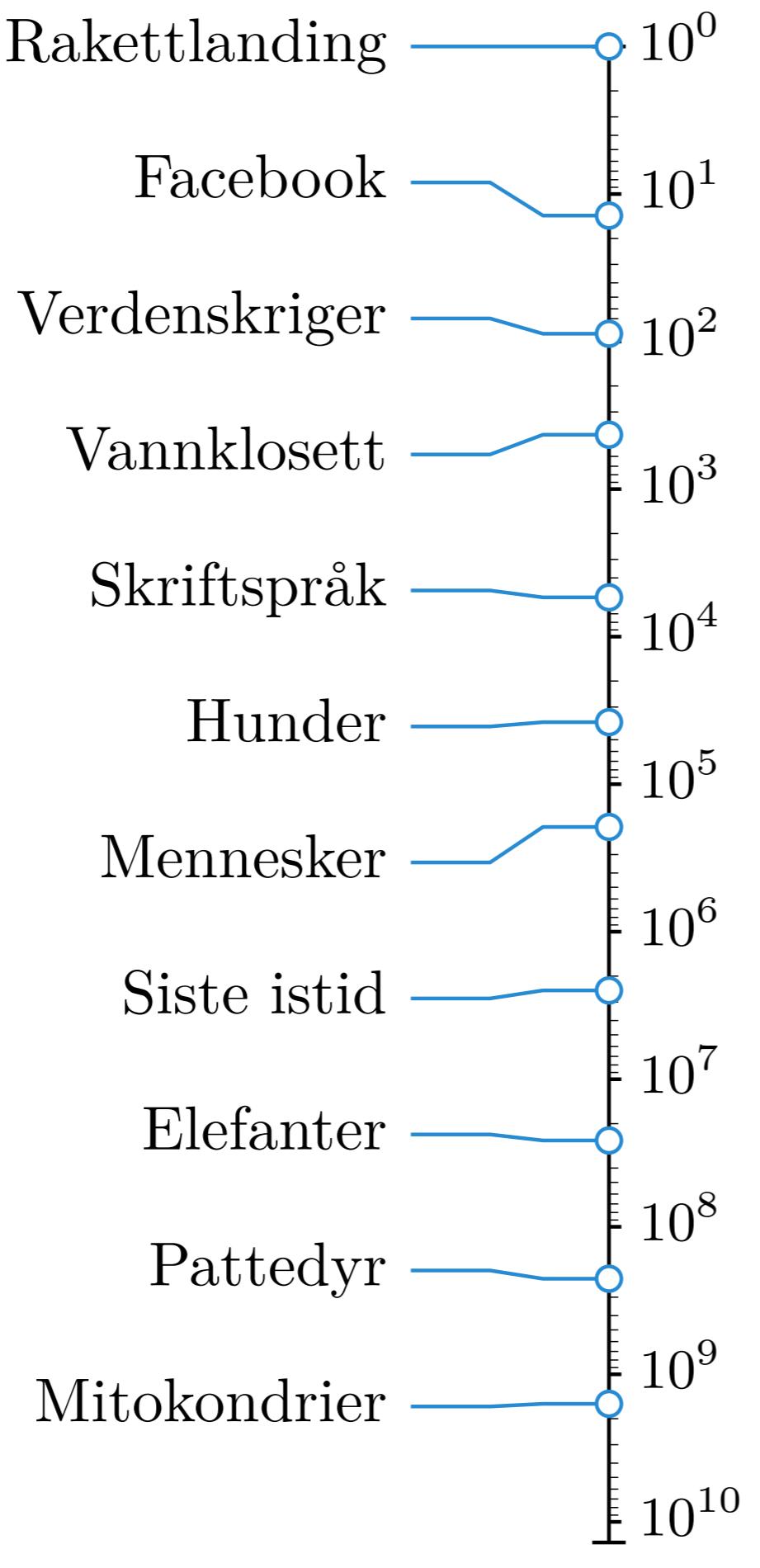


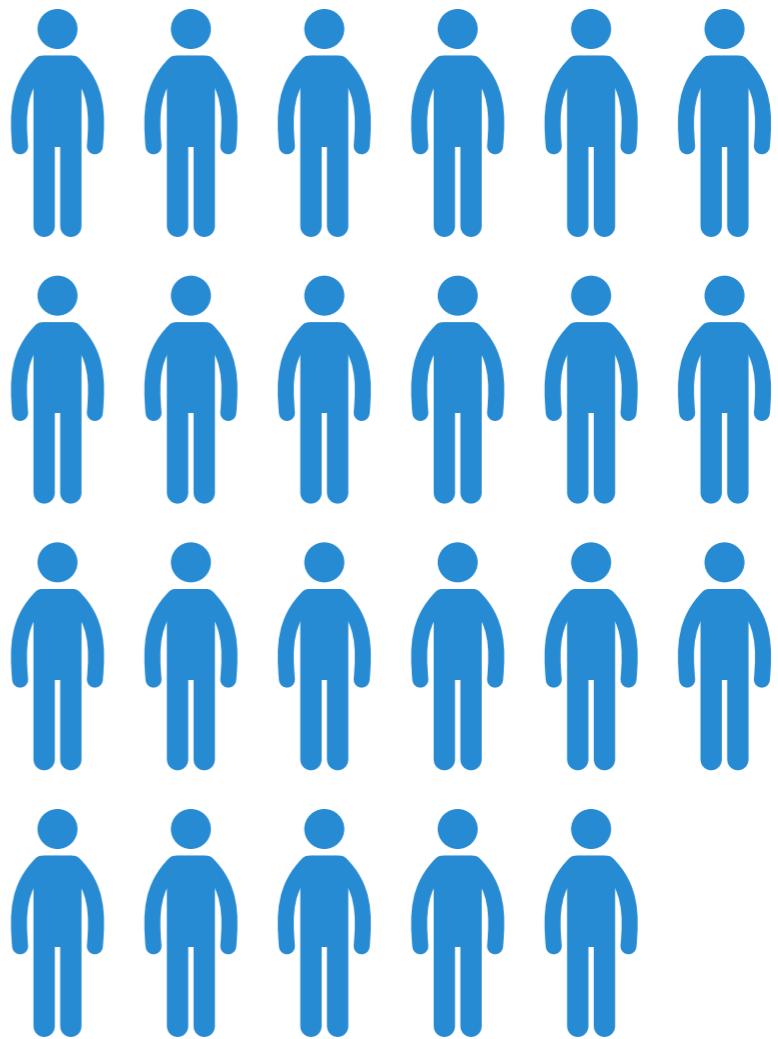
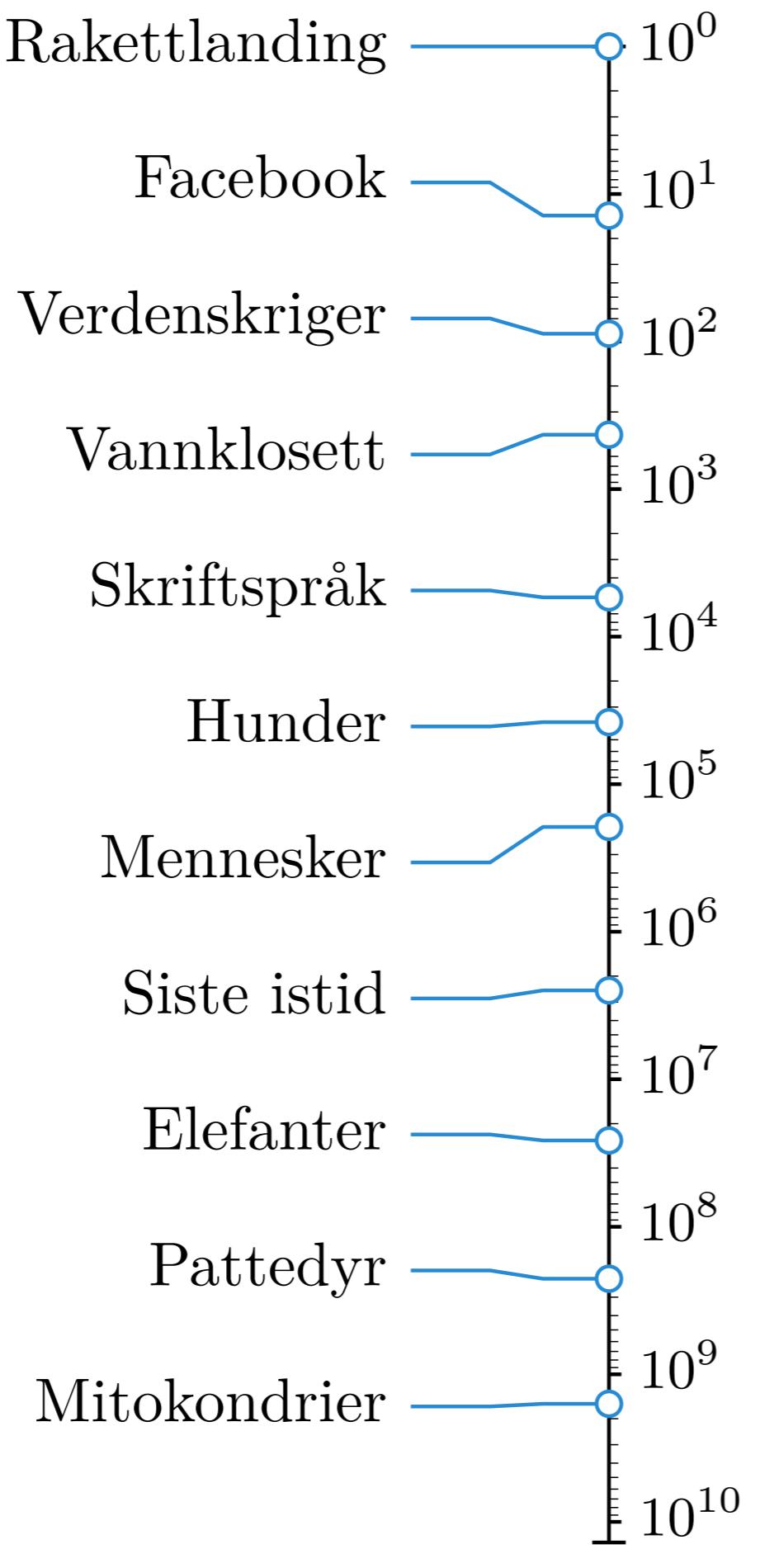


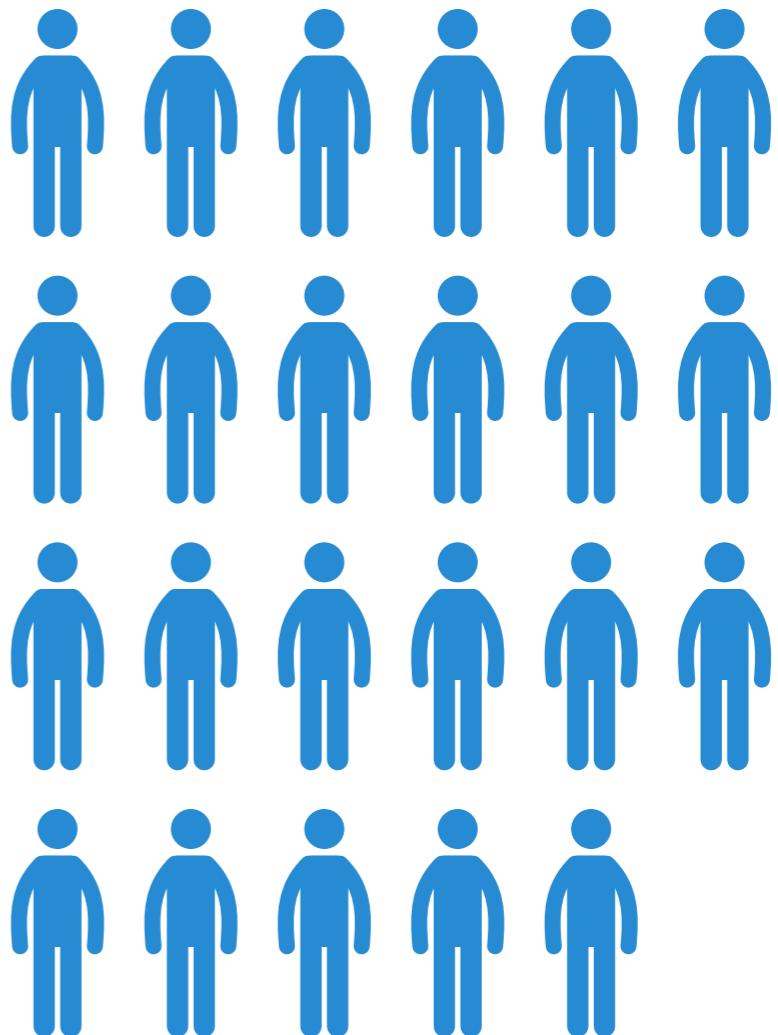
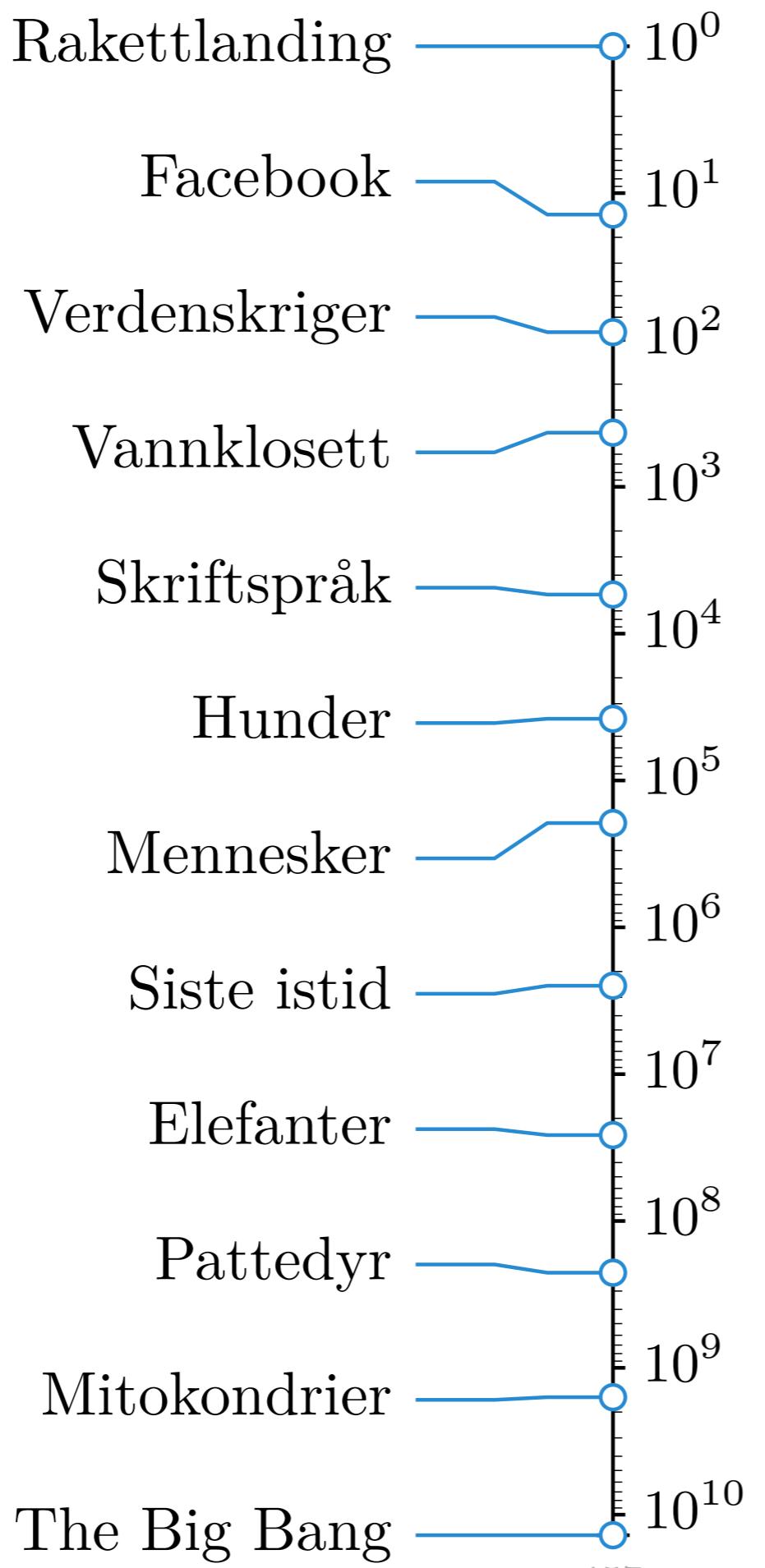


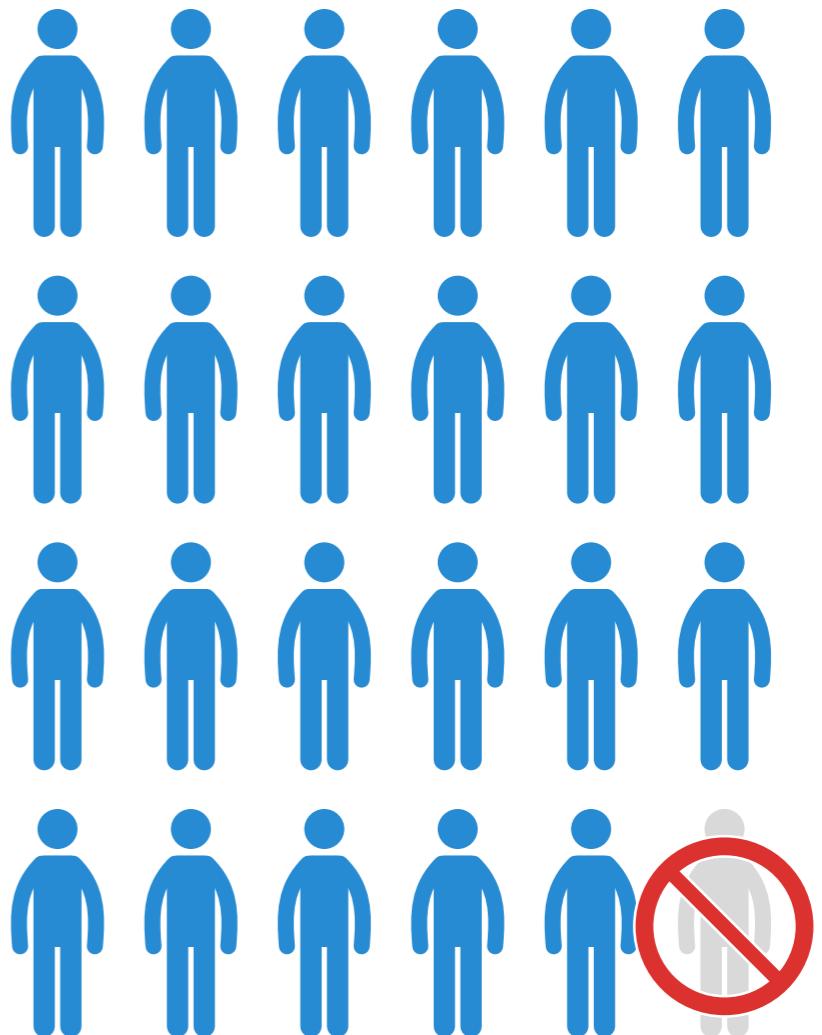
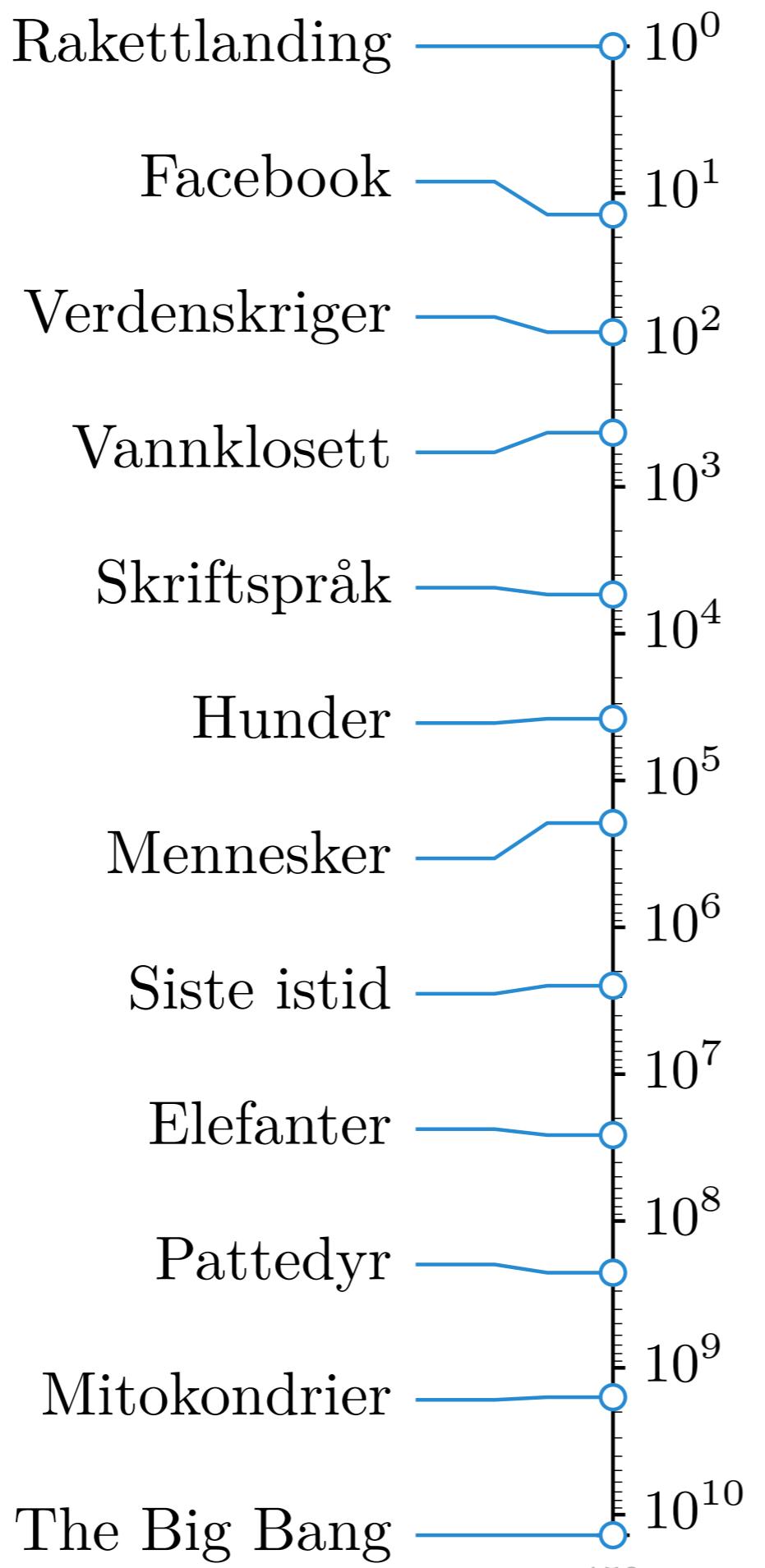




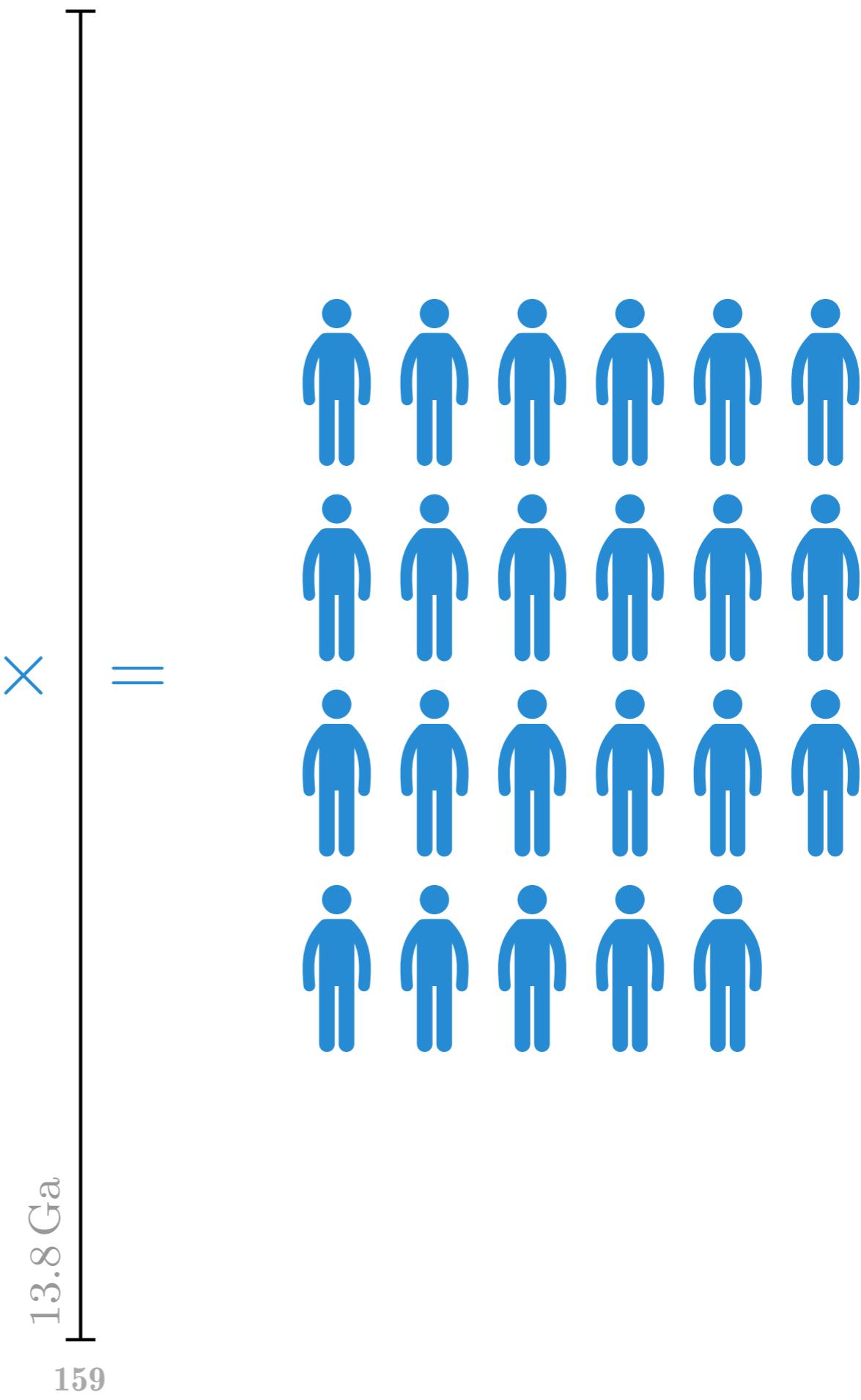




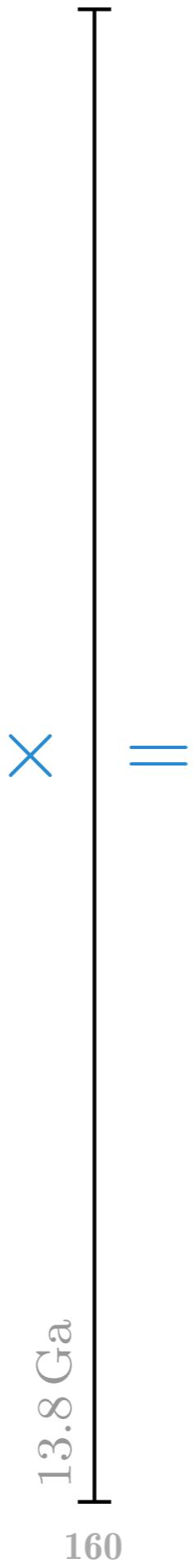




1



1



1

13.8 Ga

161

X

=



1 000 000 000 000 000 000 000 000

X

=

13.8 Ga

162

1 000 000 000 000 000 000 000 000 000

13.8 Ga

163

X

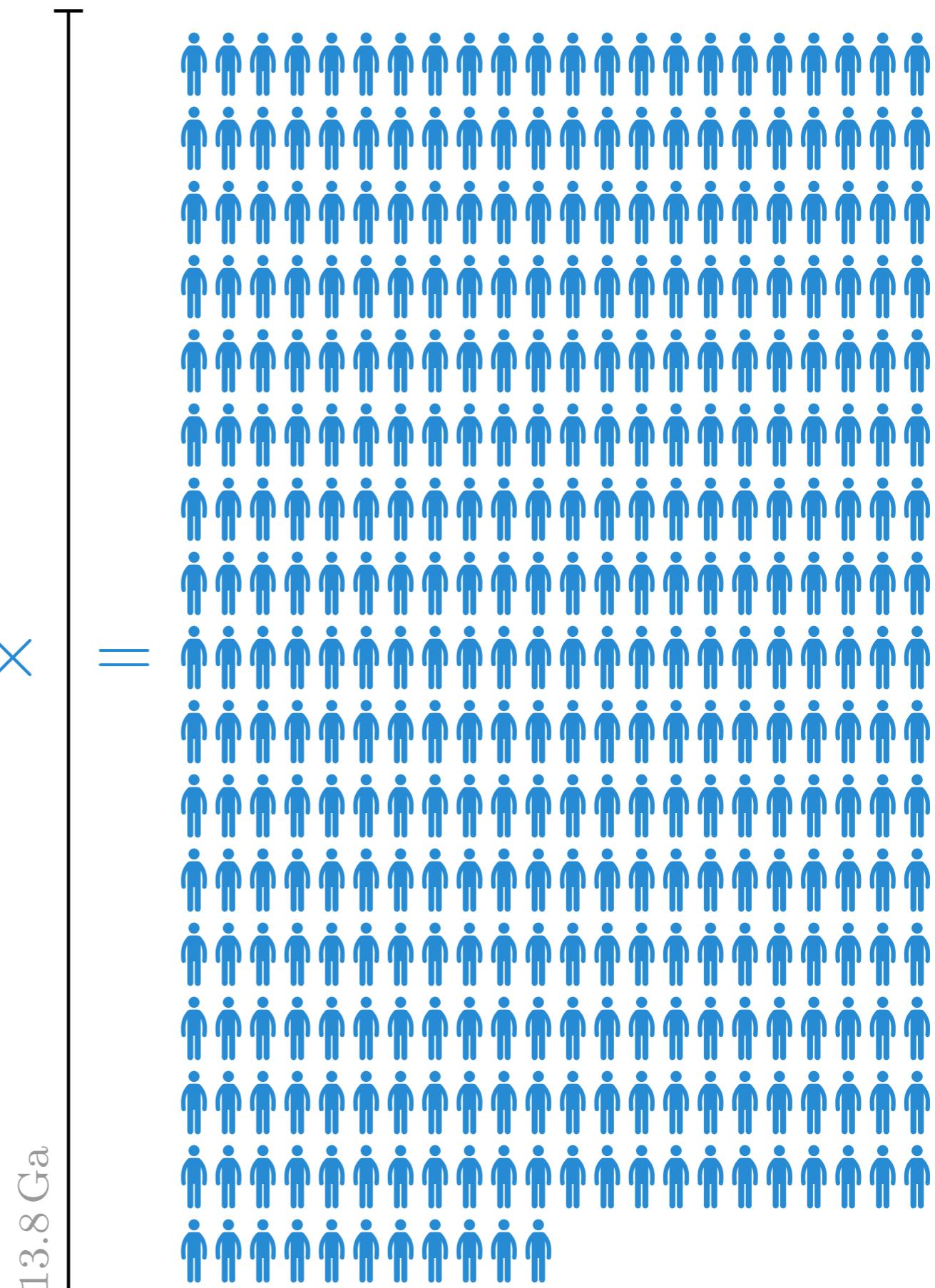
=

13.8 Ga

164

13.8 Ga

165

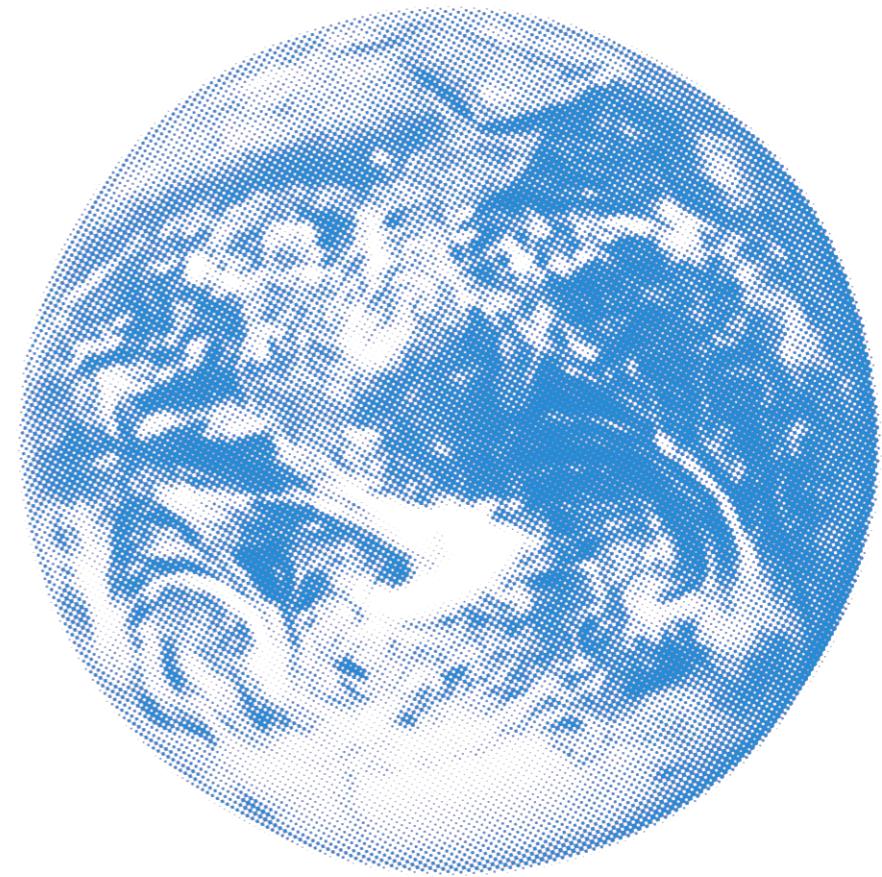


$$\frac{1}{2}h =$$

Med en smartere algoritme

$$\frac{1}{2}h$$

=

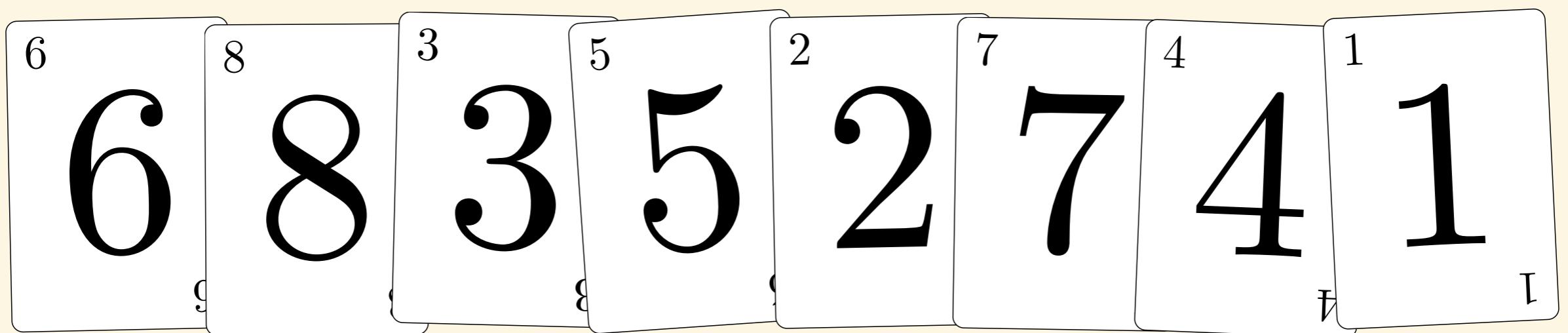


Her er matchingen utført i hvert land for seg (og mange av tallene er estimerer). Algoritmen som brukes er Chandran-Hochbaum.

Med en smartere algoritme

- Mange viktige problemer krever algoritmiske løsninger
- Forskjellen på gode og dårlige løsninger er i kosmisk skala
- Vi er interessert i «de store linjene»

- Matching: Forelesning 12
- Men: Sortering er en god modell for algoritmisk tenking!

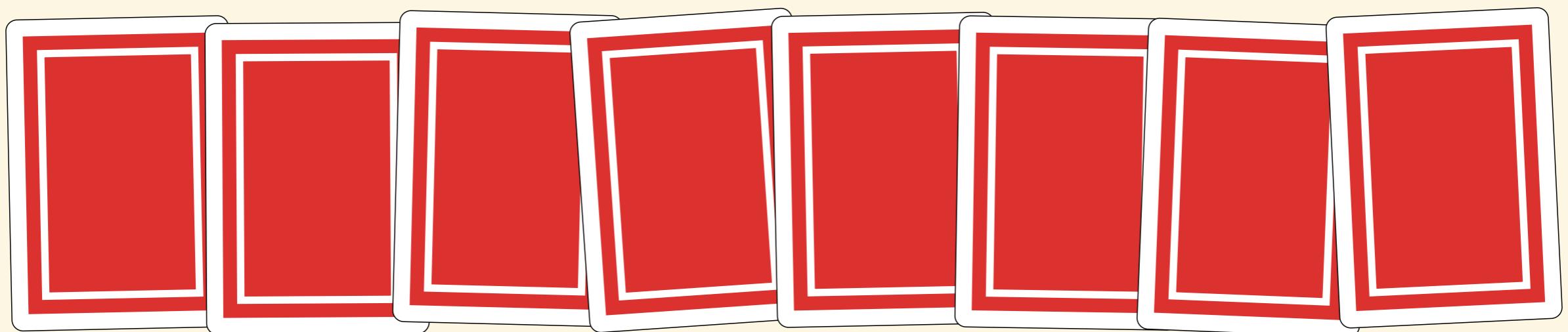


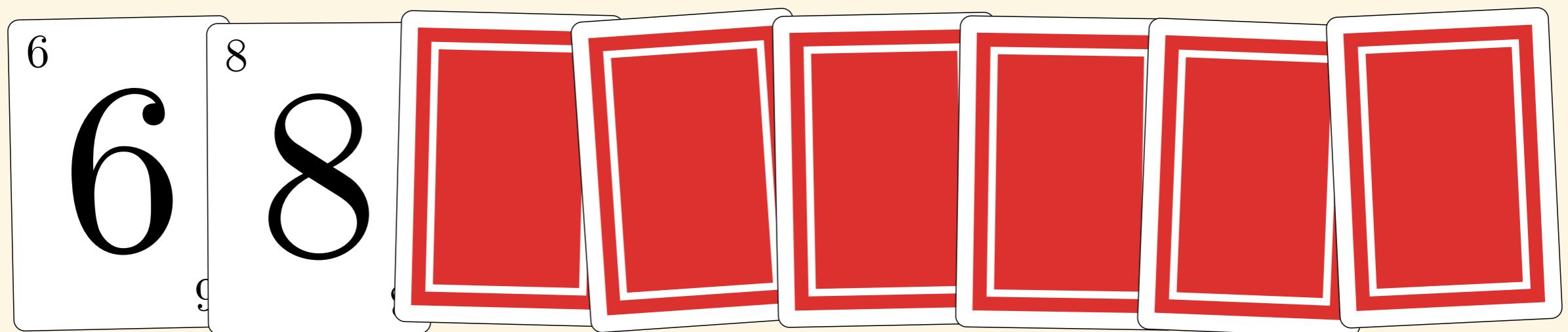
Mulig fremgangsmåte: Vi begynner med å se på bare de første kortene, og så sorterer vi oss gradvis mot høyre.

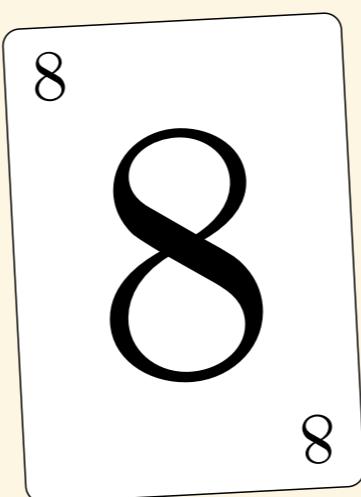
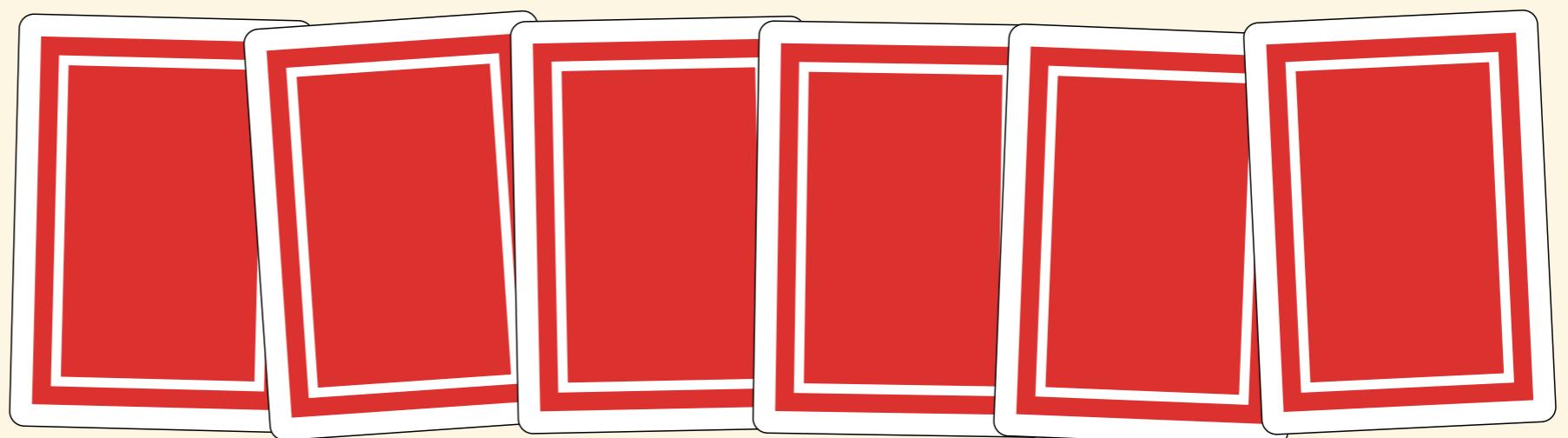
Vi utnytter struktur i problemet – nemlig det at det er mulig å sortere litt etter litt. Dette ignoreres helt i en brute-force-løsning.

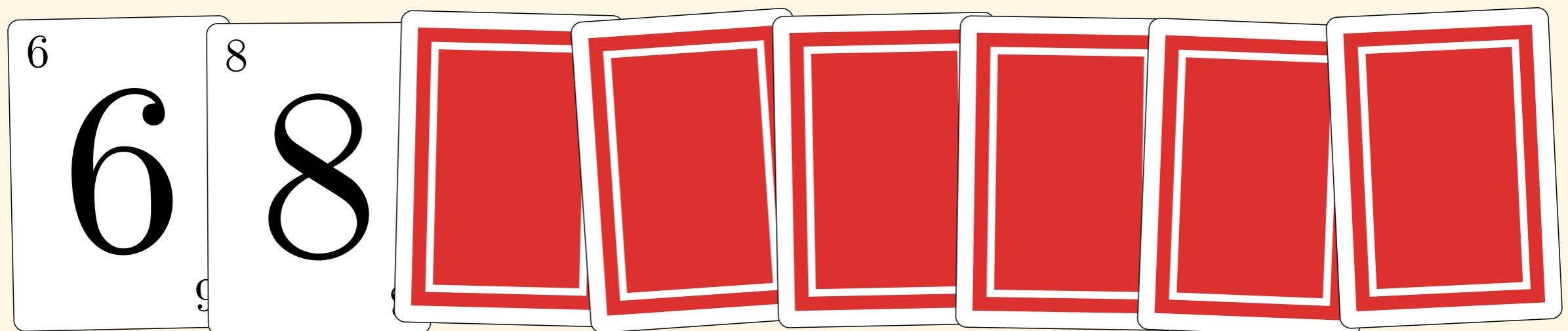
Vi ser er hele tiden bare interessert i å sette inn neste kort.

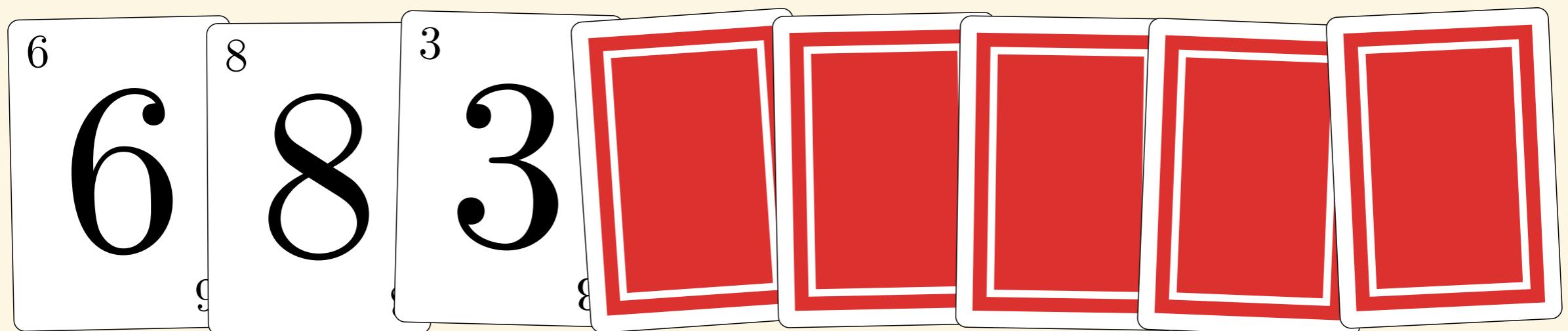
Vi bygger altså trinnvis på forrige del-løsning, uten å begynne fra scratch hele tiden, som i en brute force-løsning.

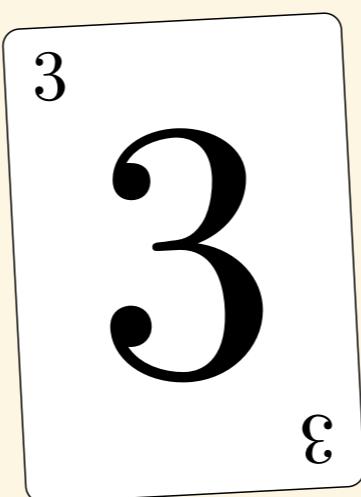
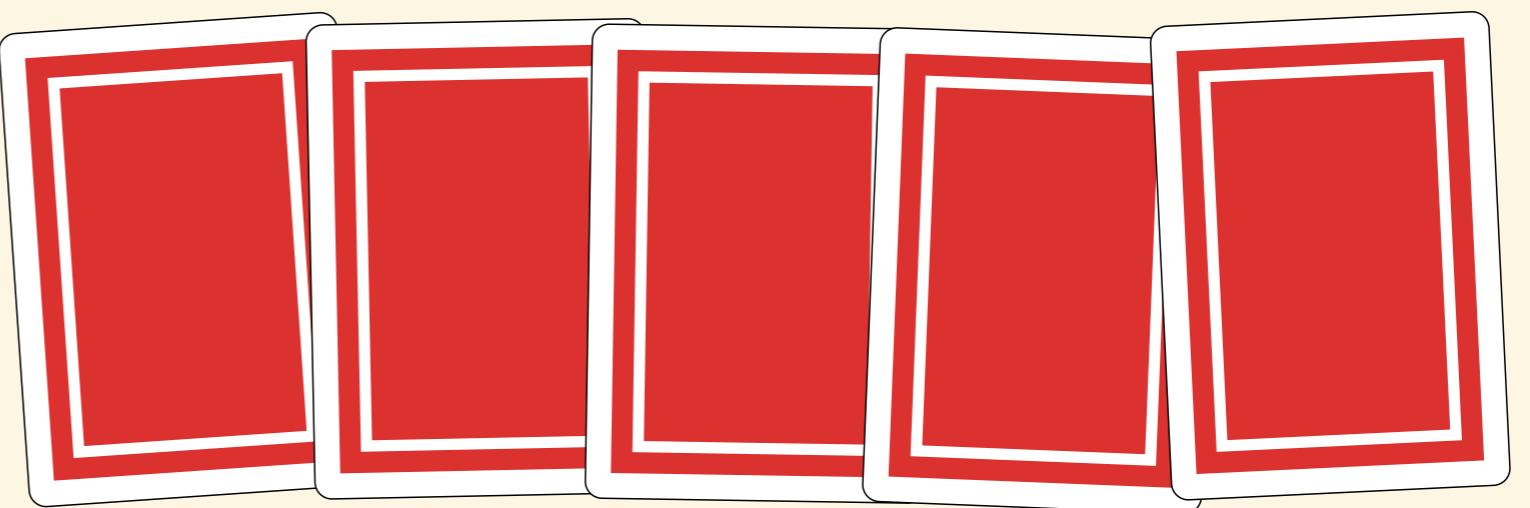
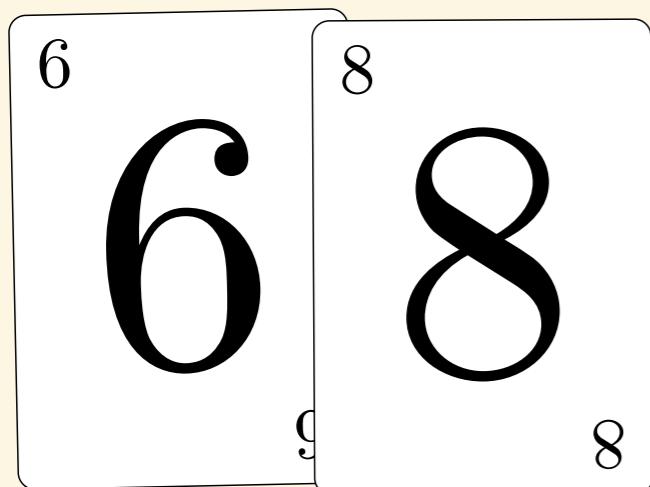


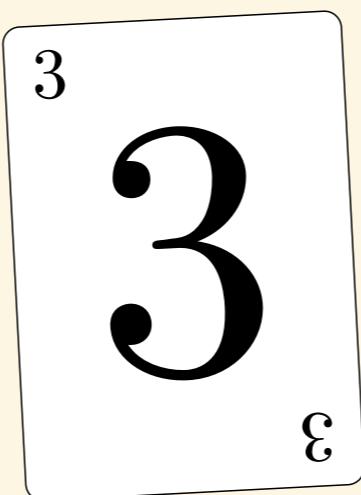
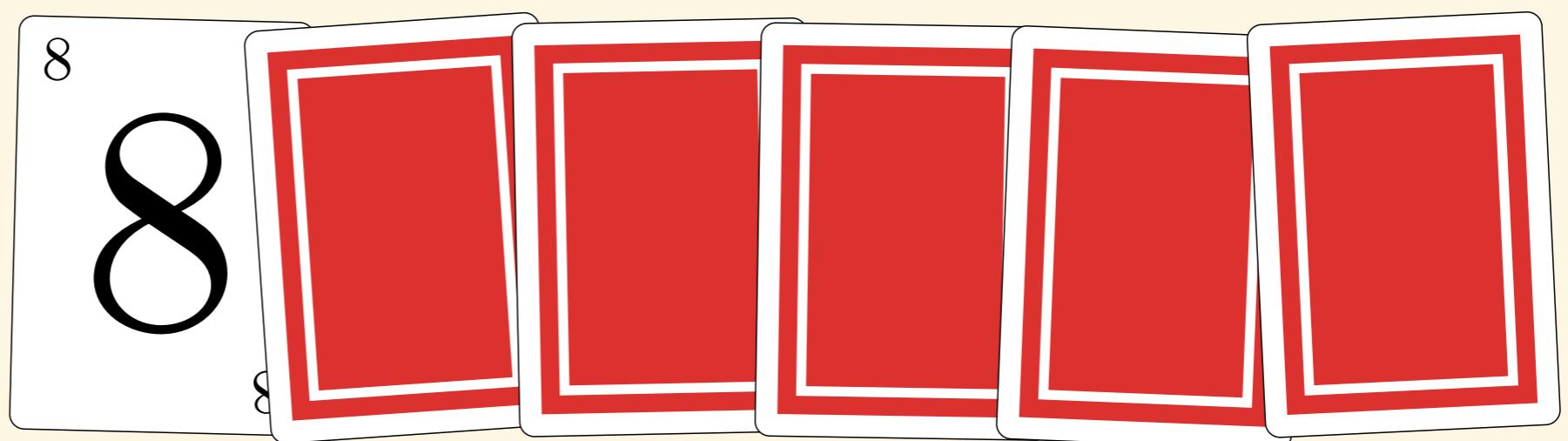


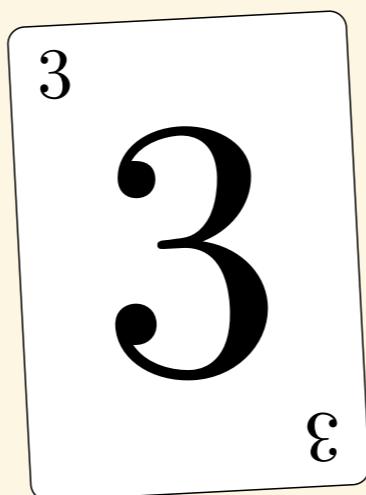
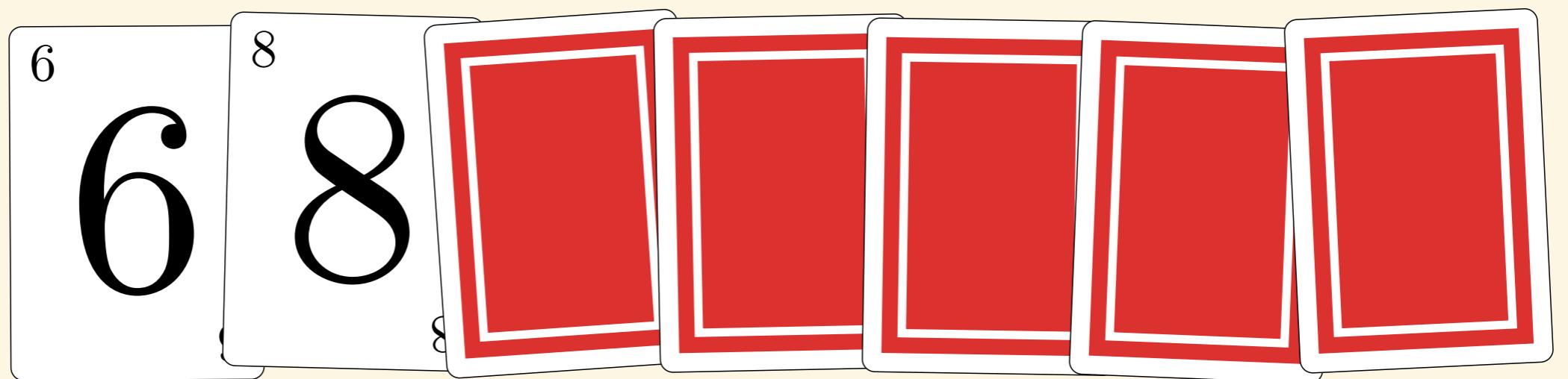


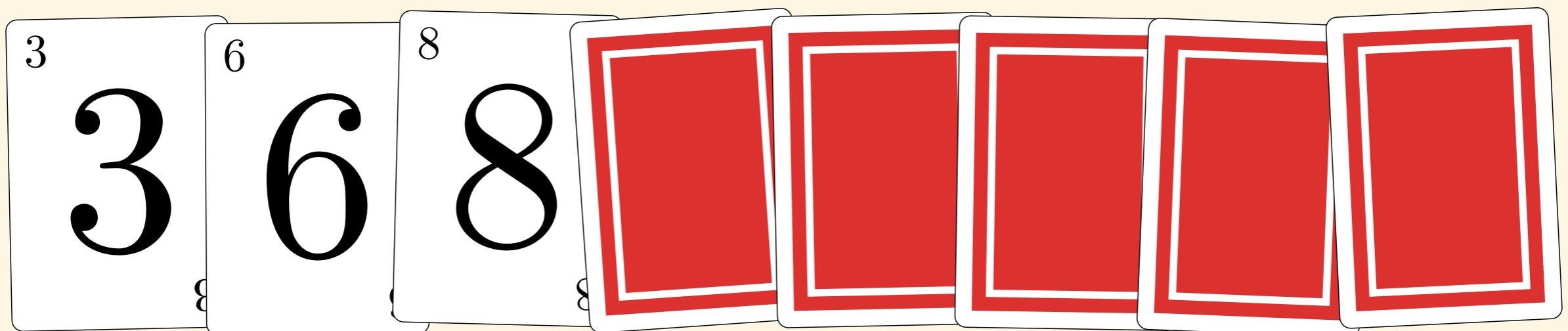


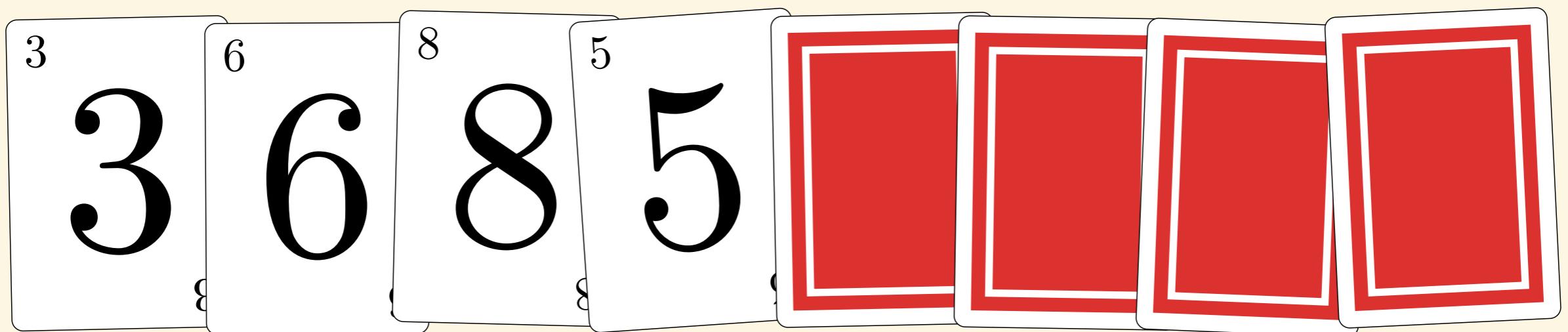


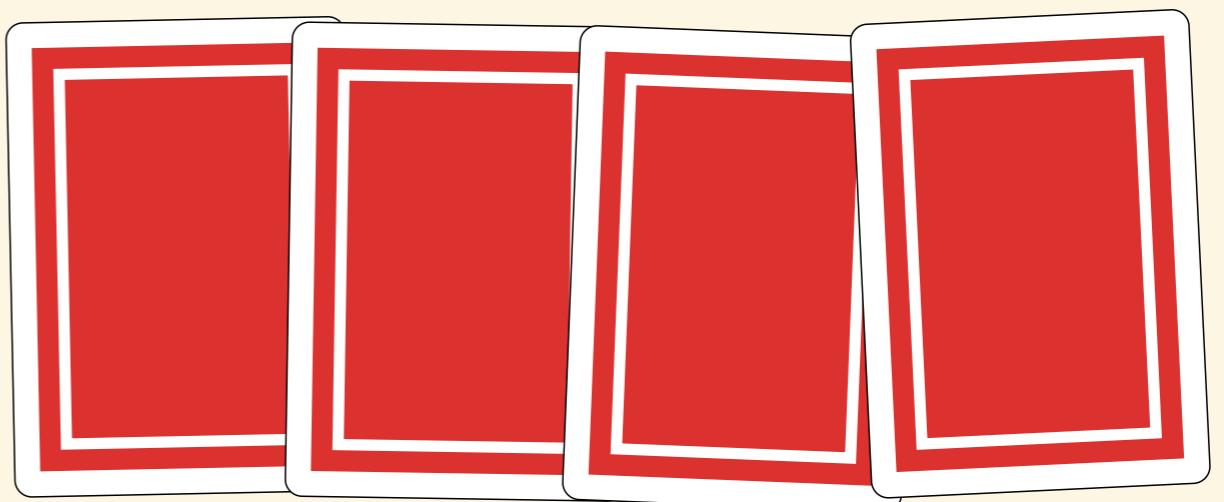
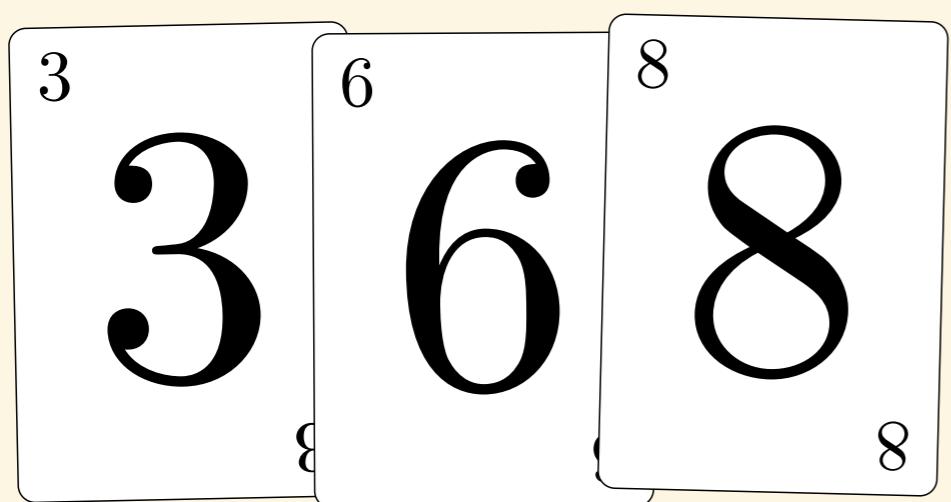


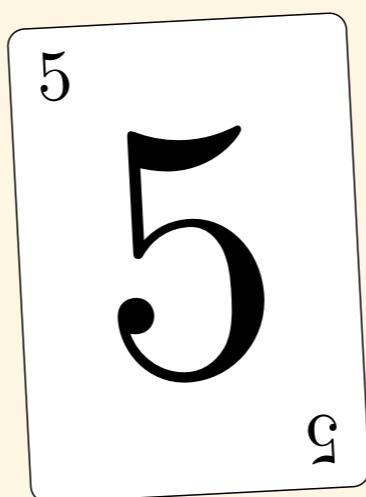
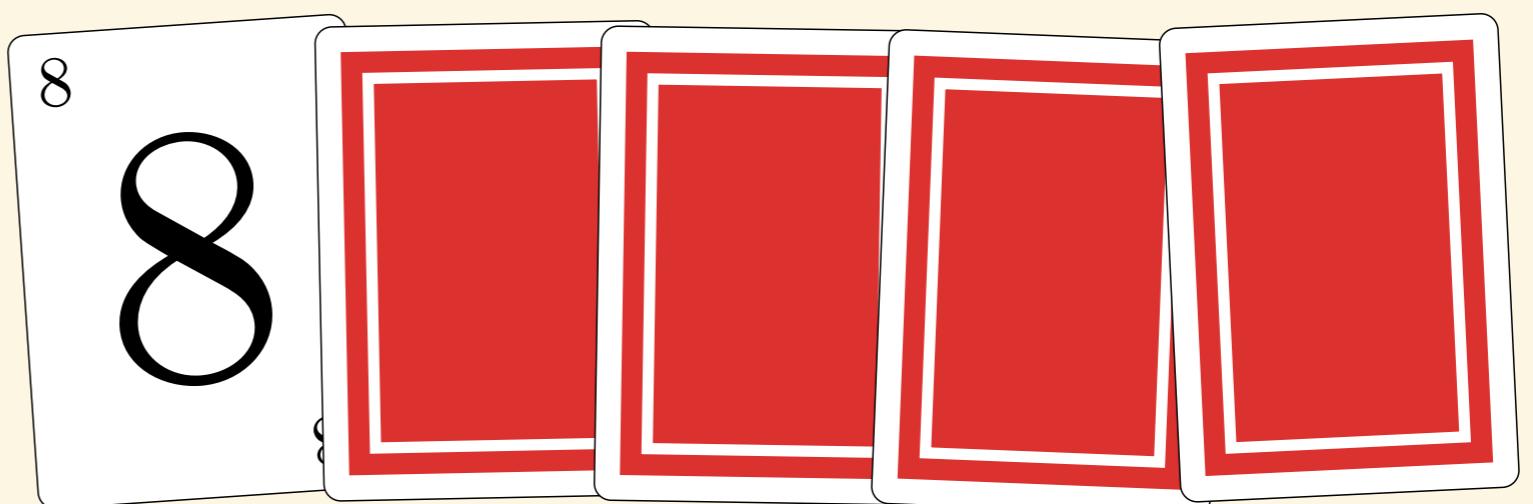
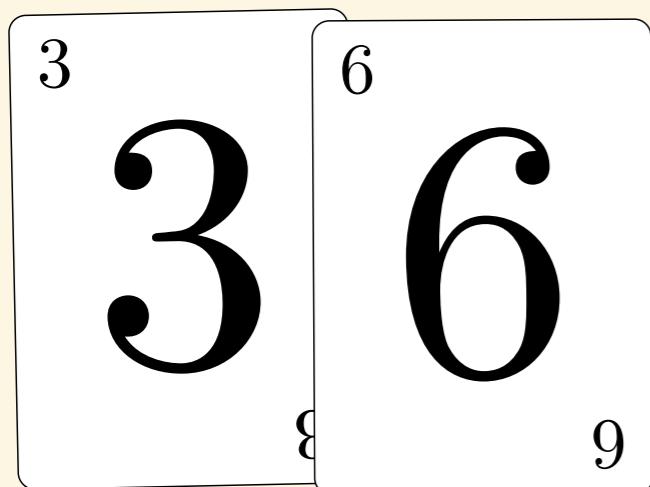


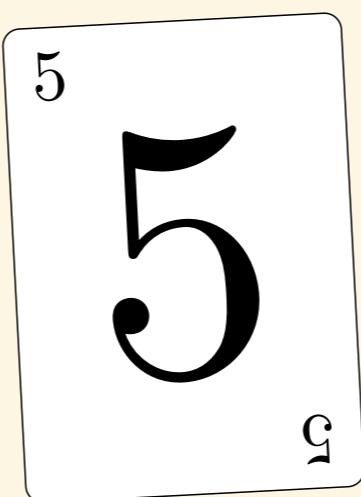
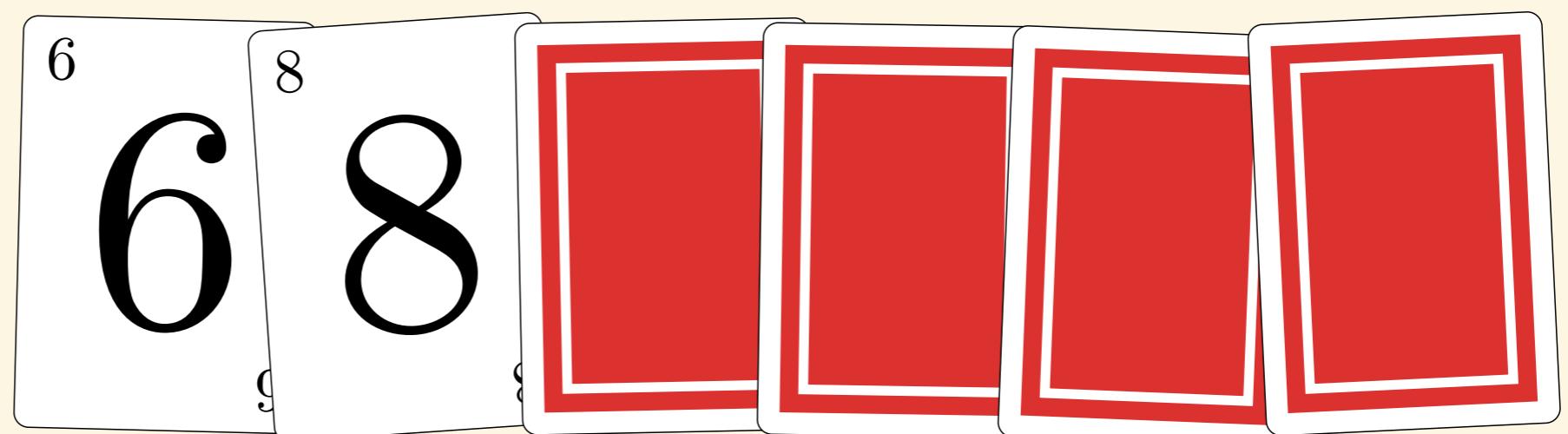
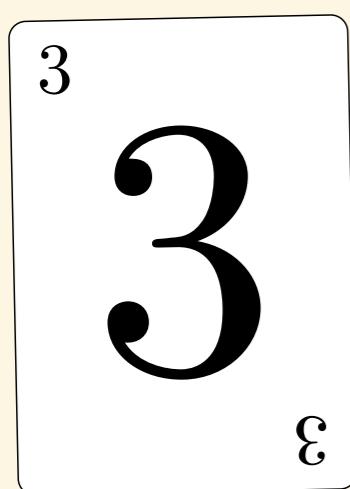


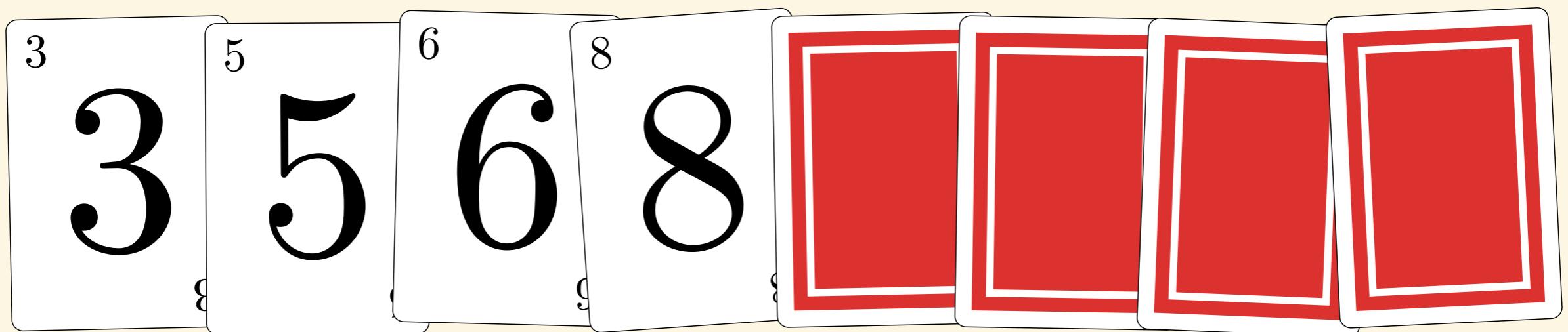


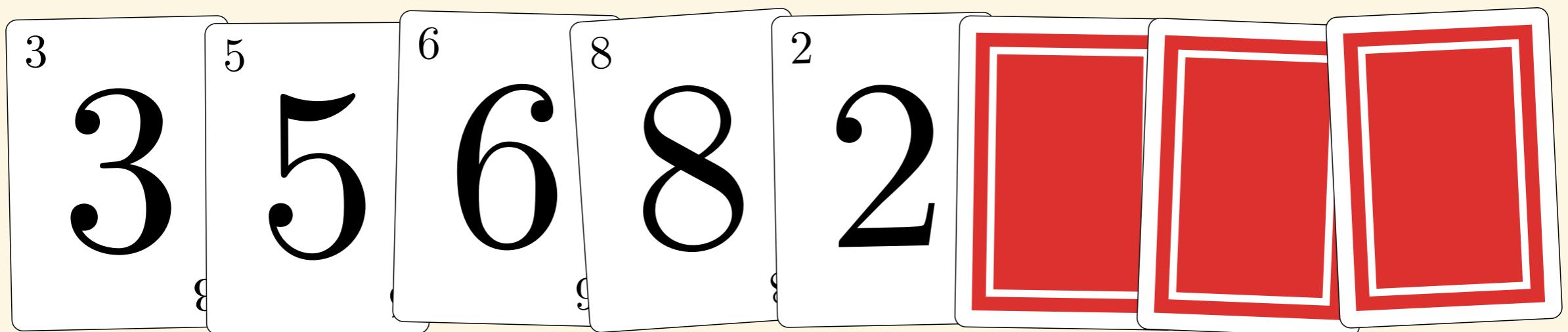


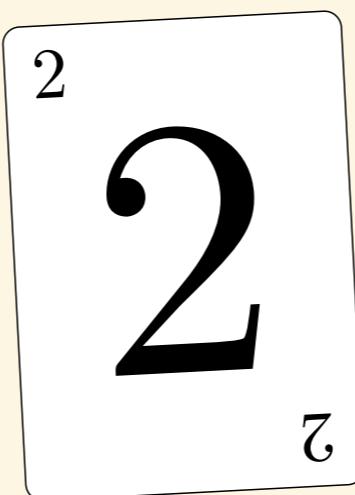
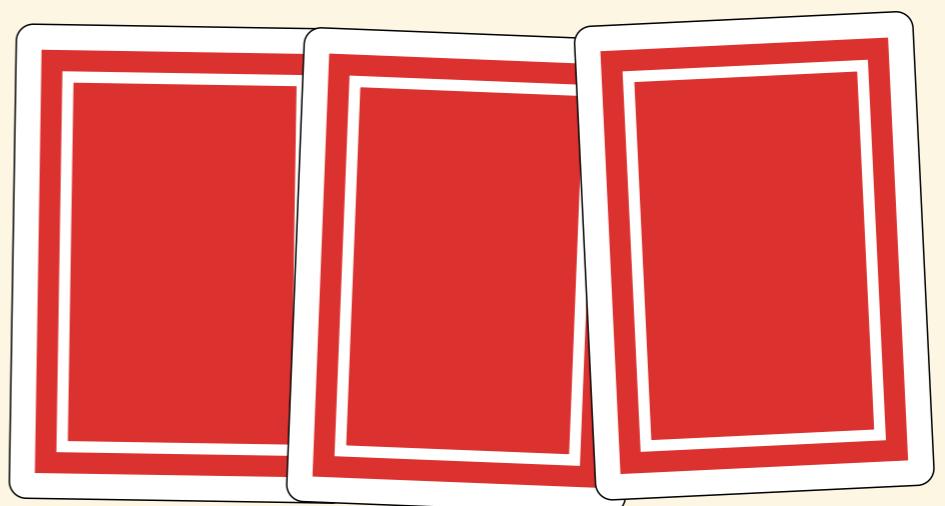
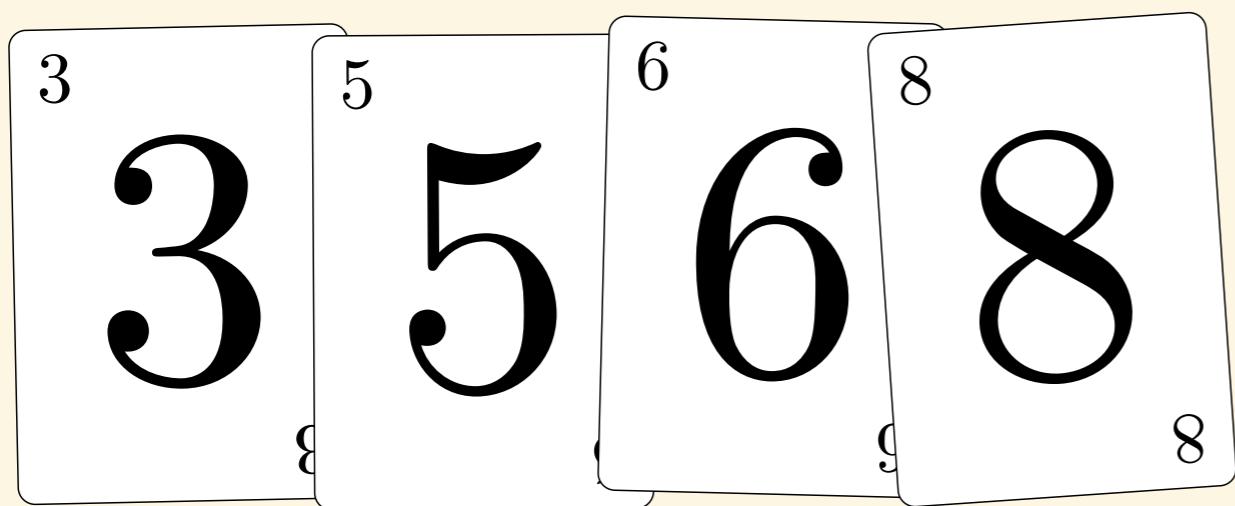


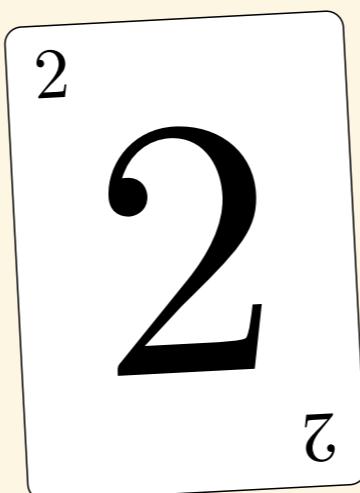
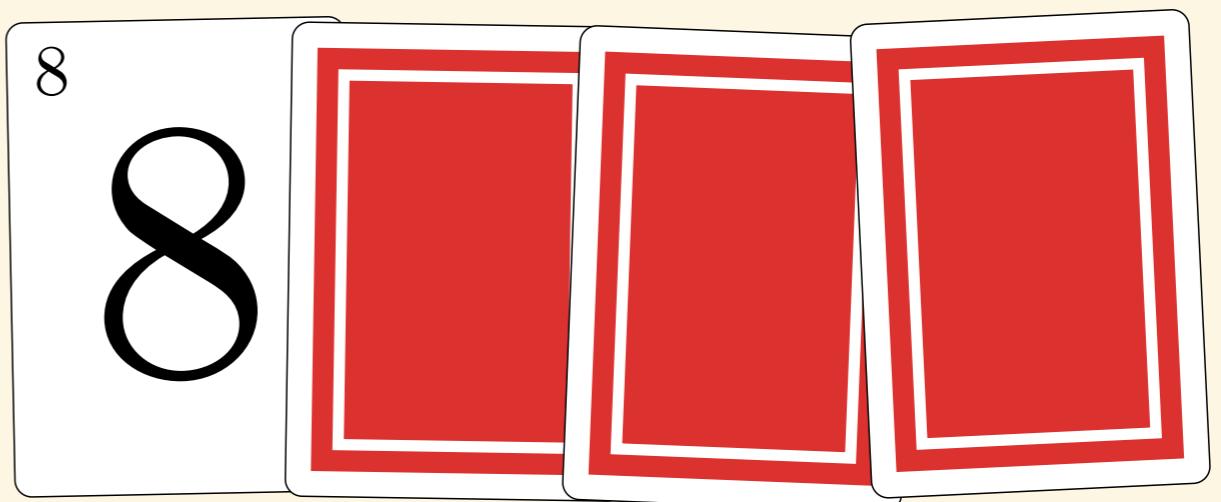
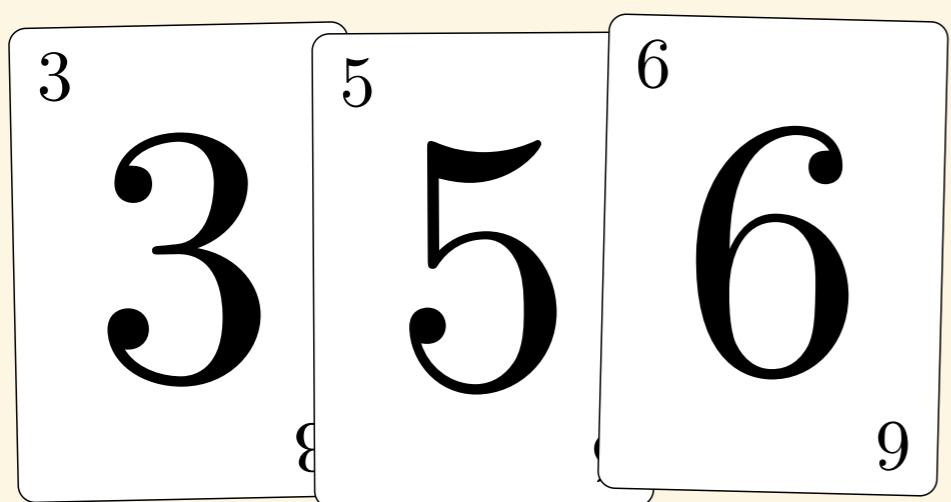


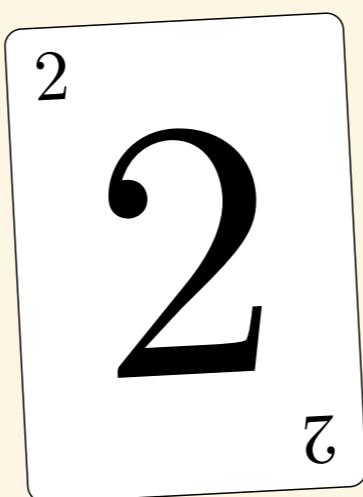
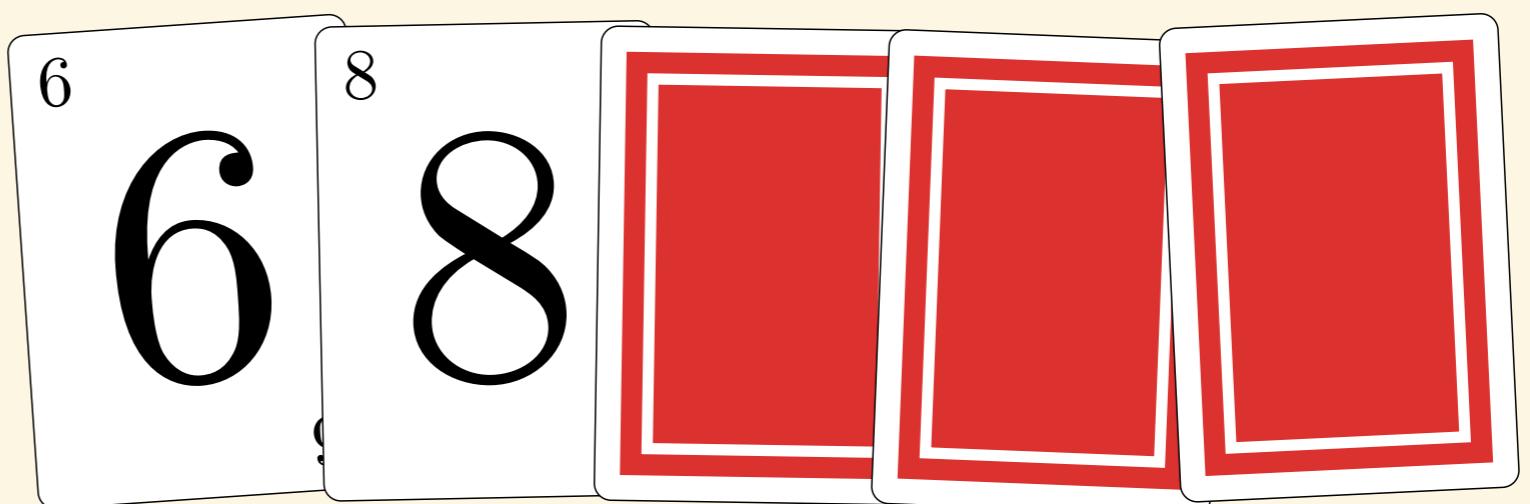
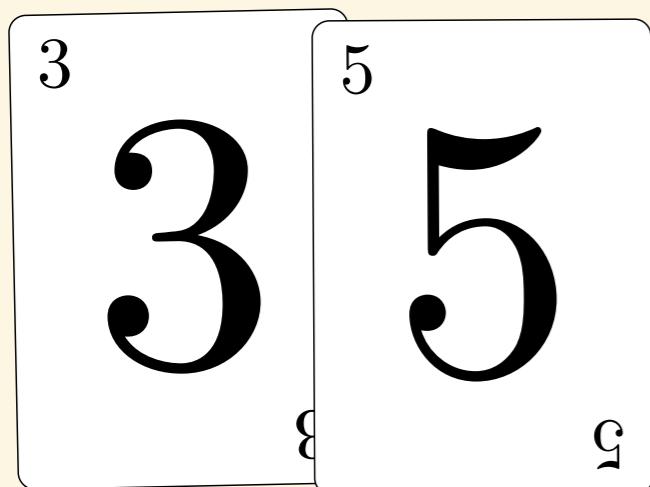


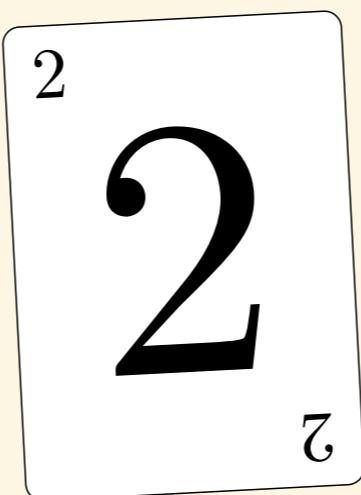
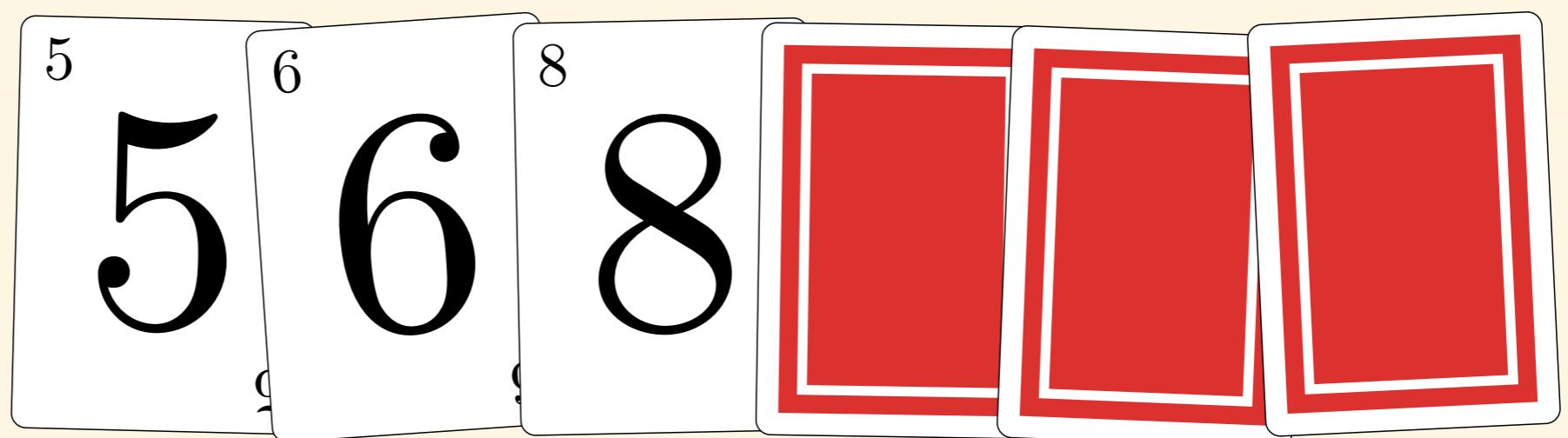
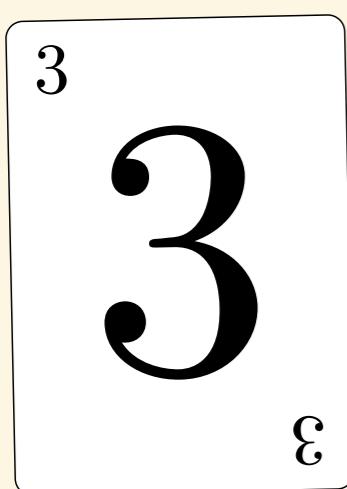


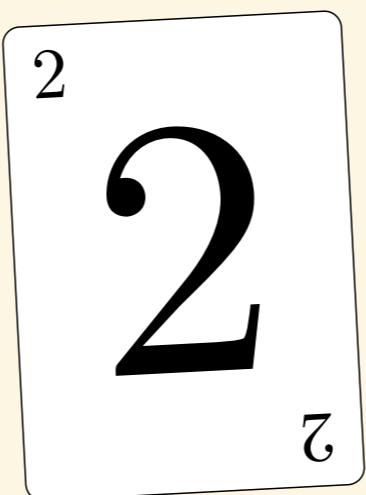
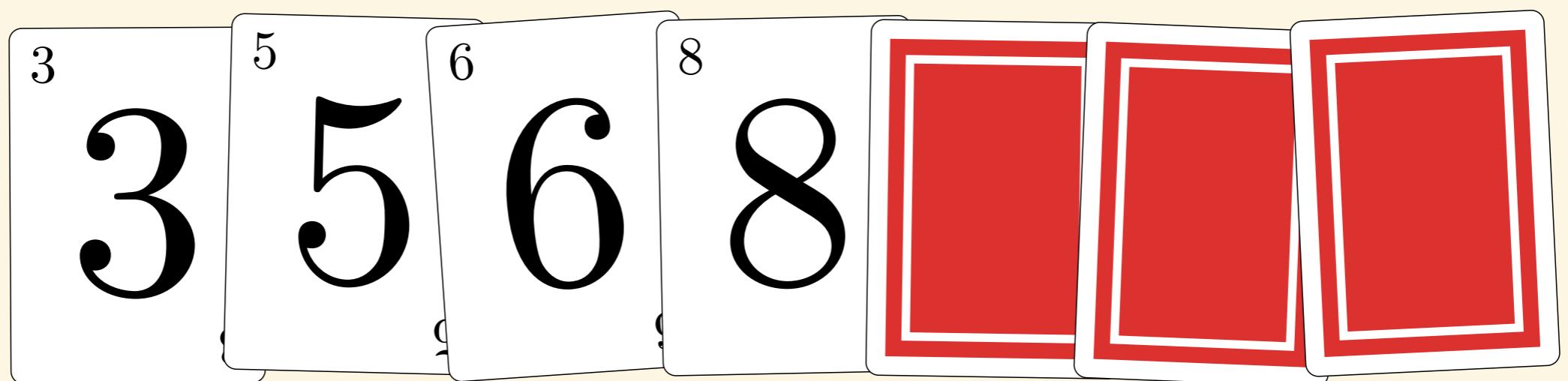


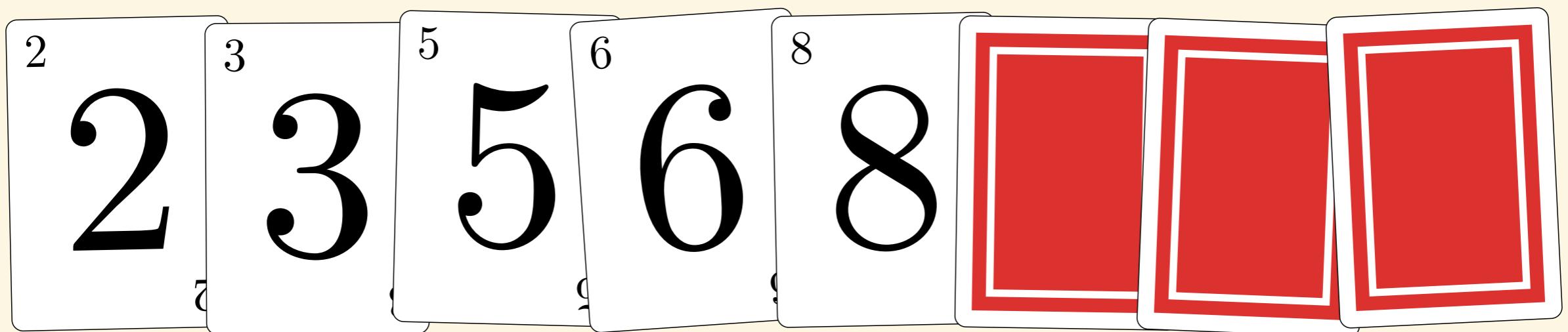


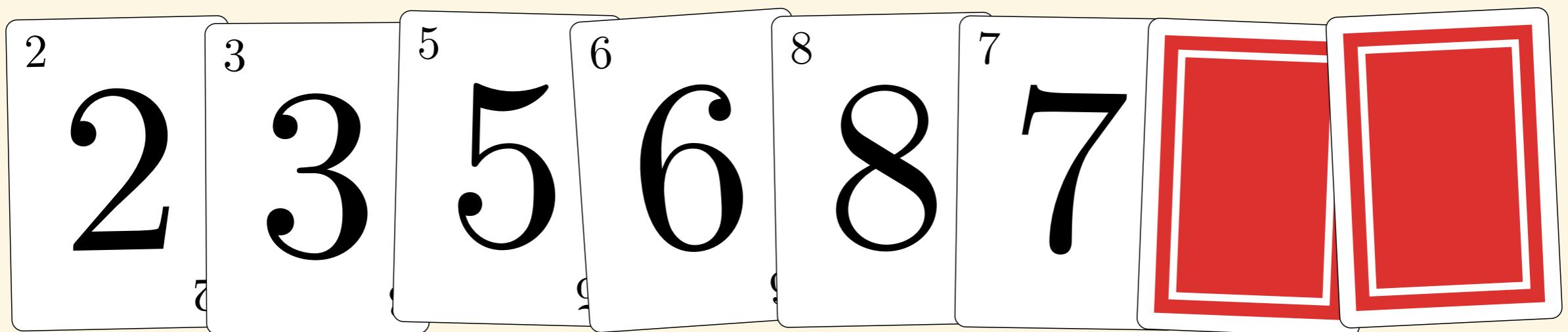


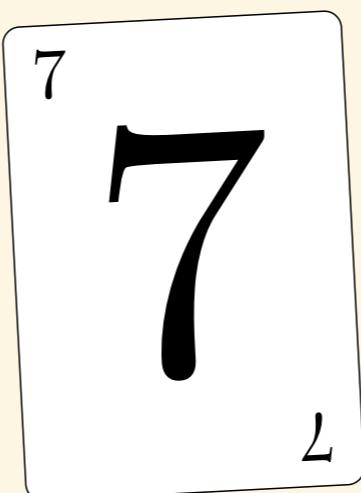
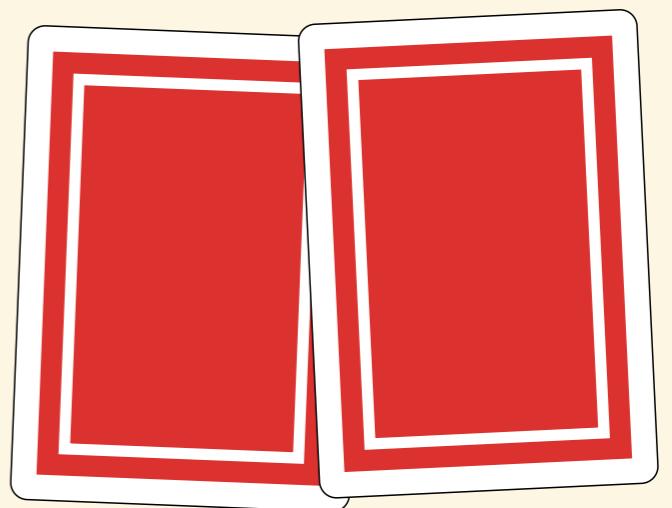
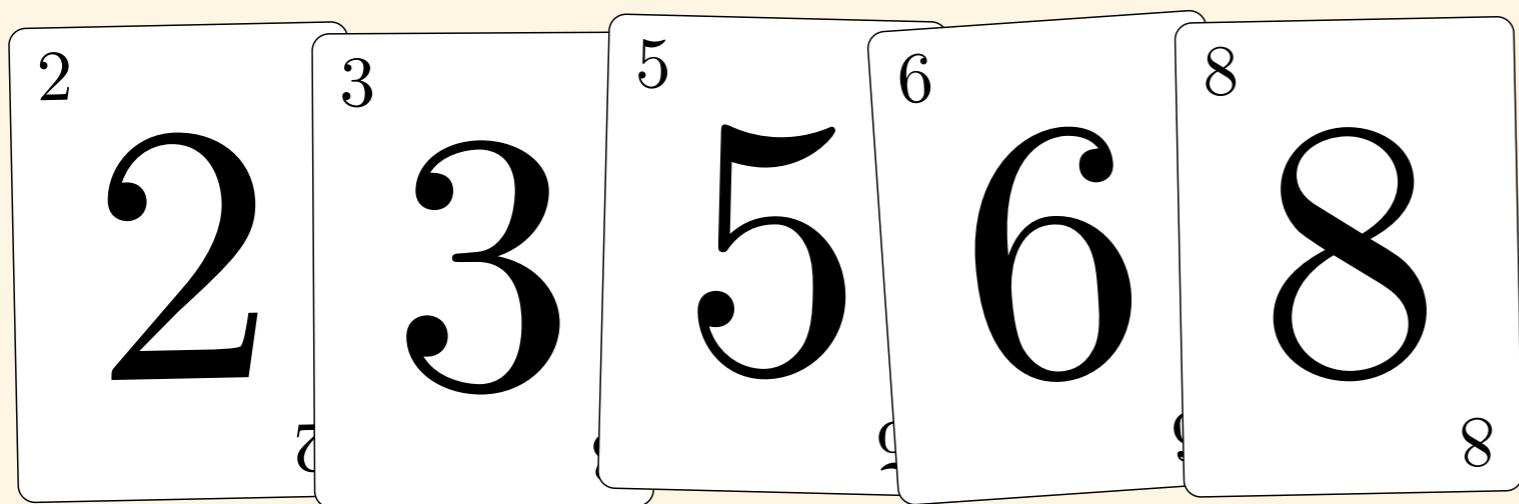


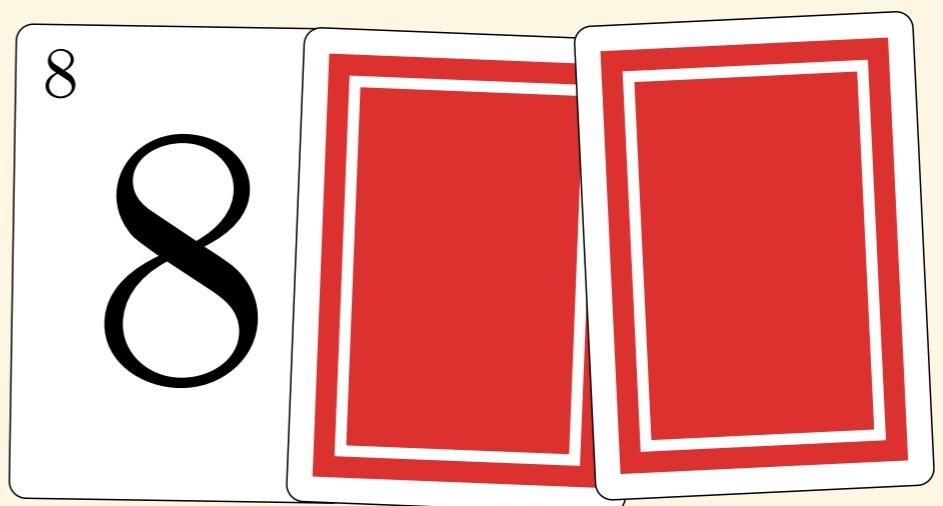
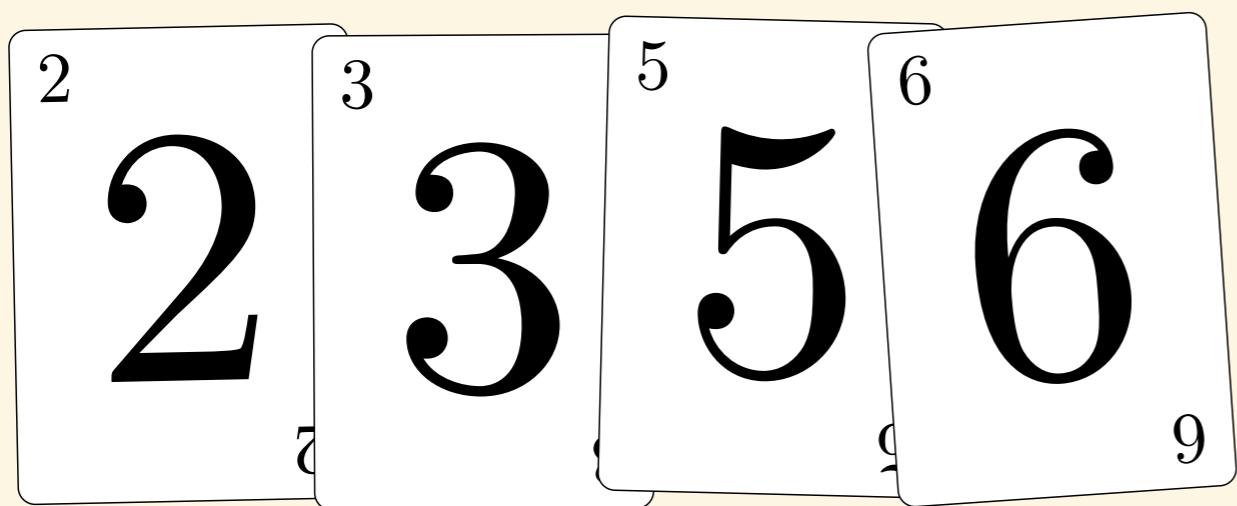


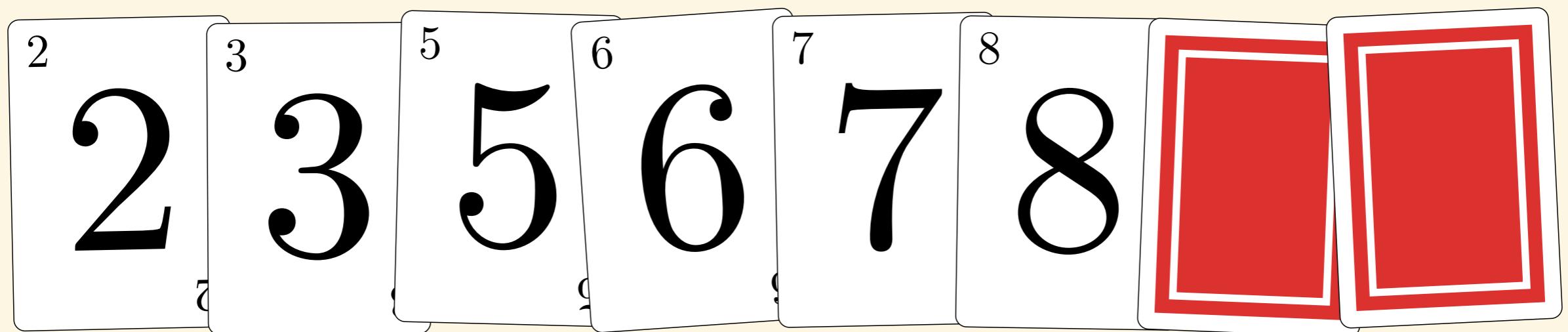


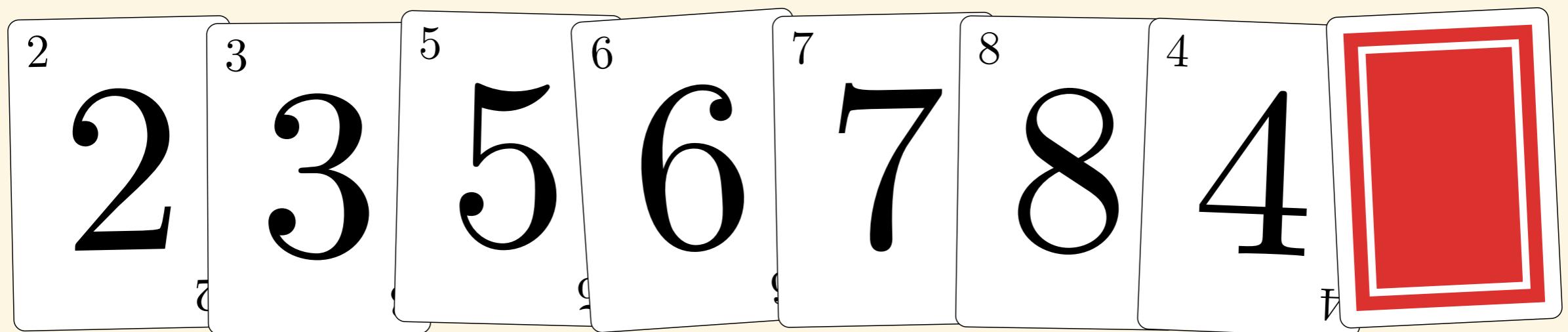


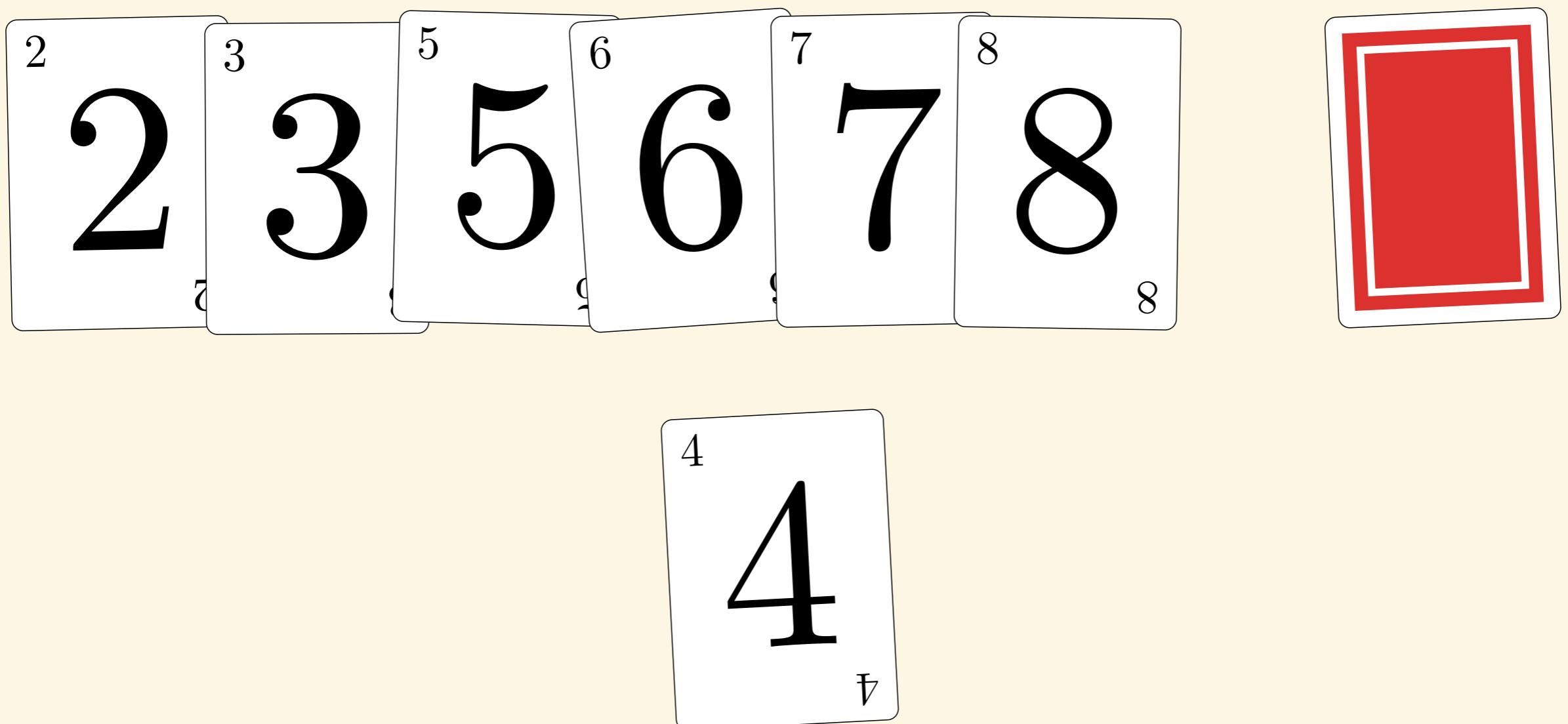


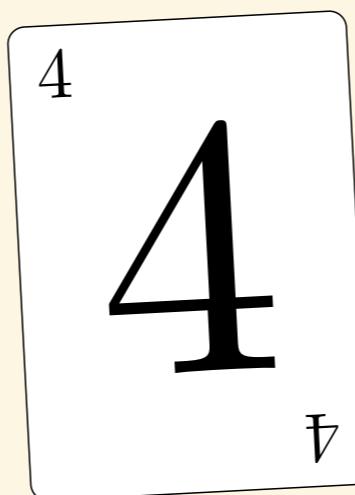
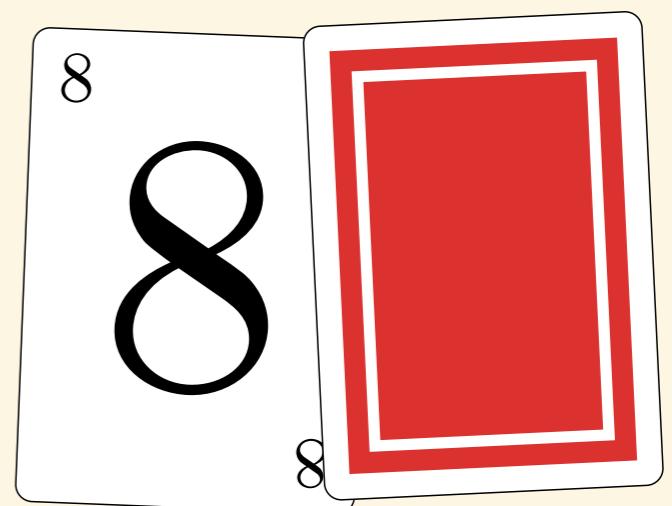
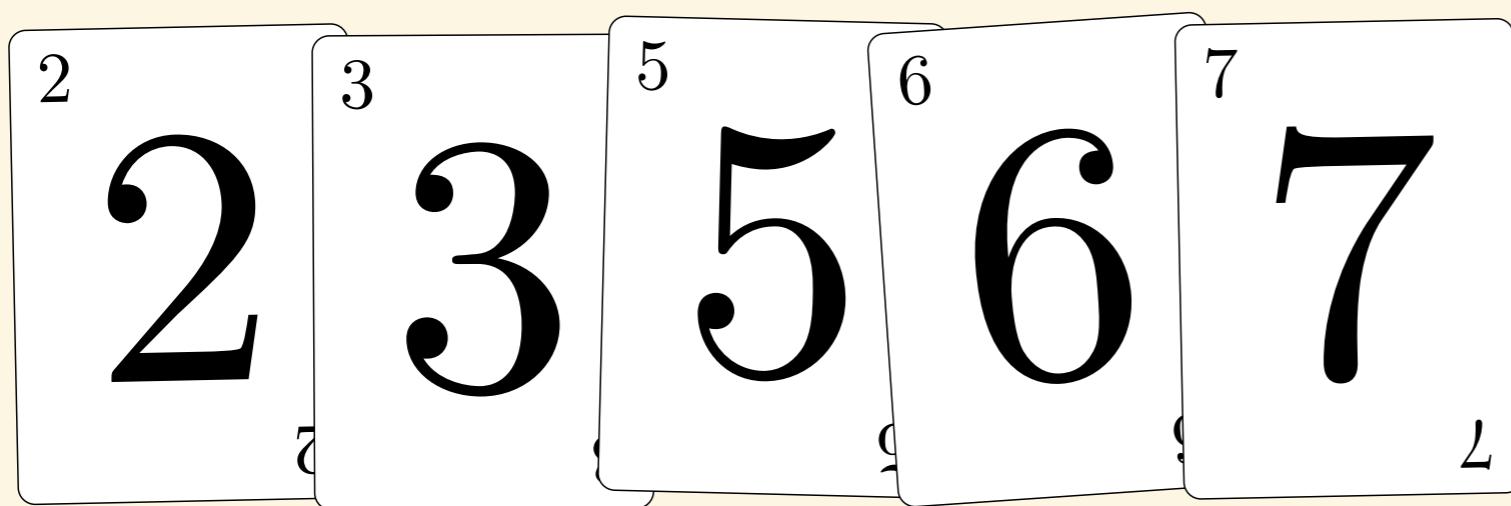


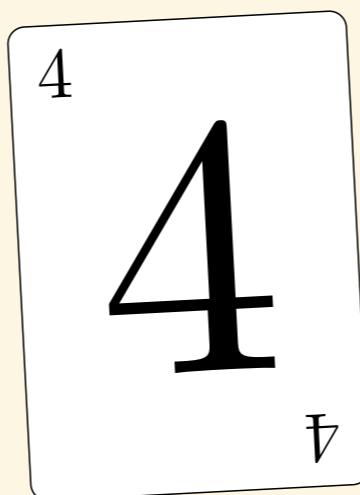
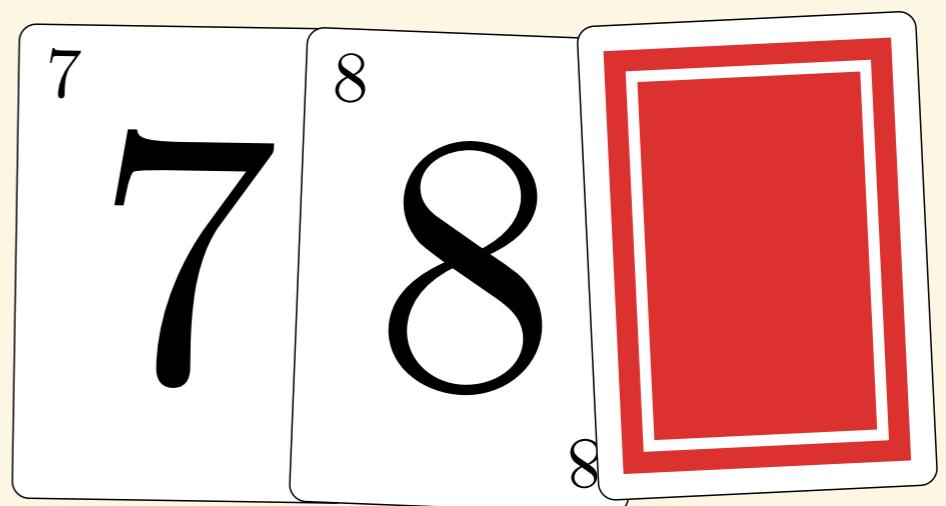
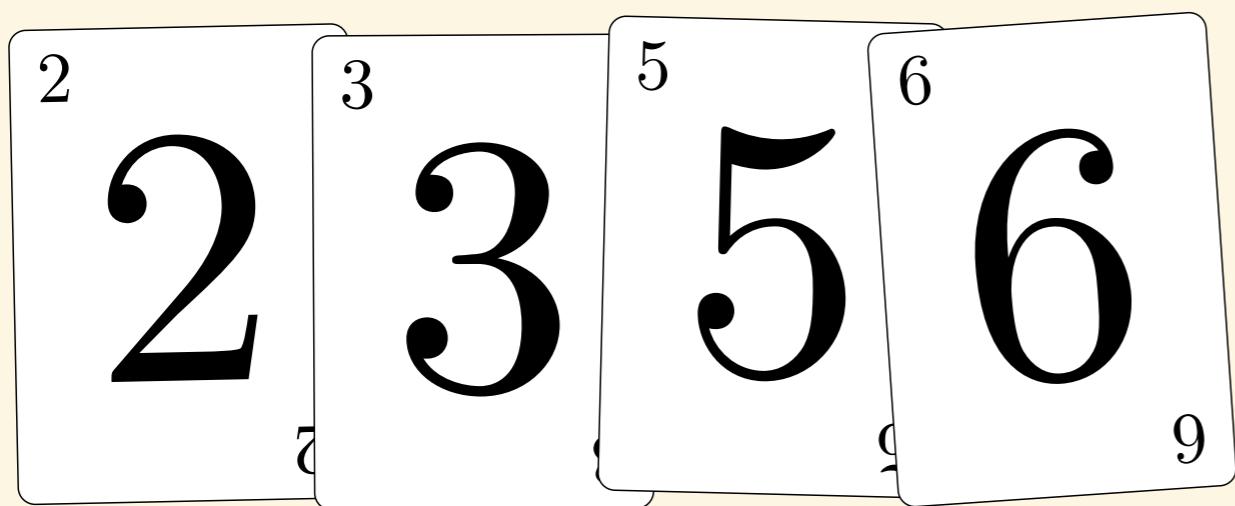


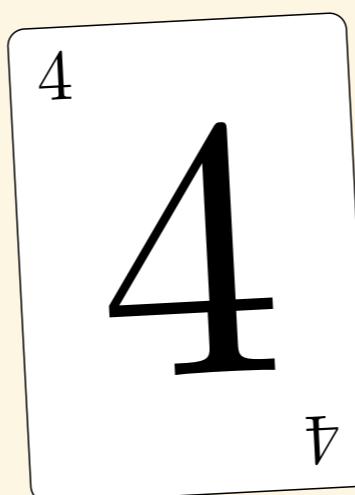
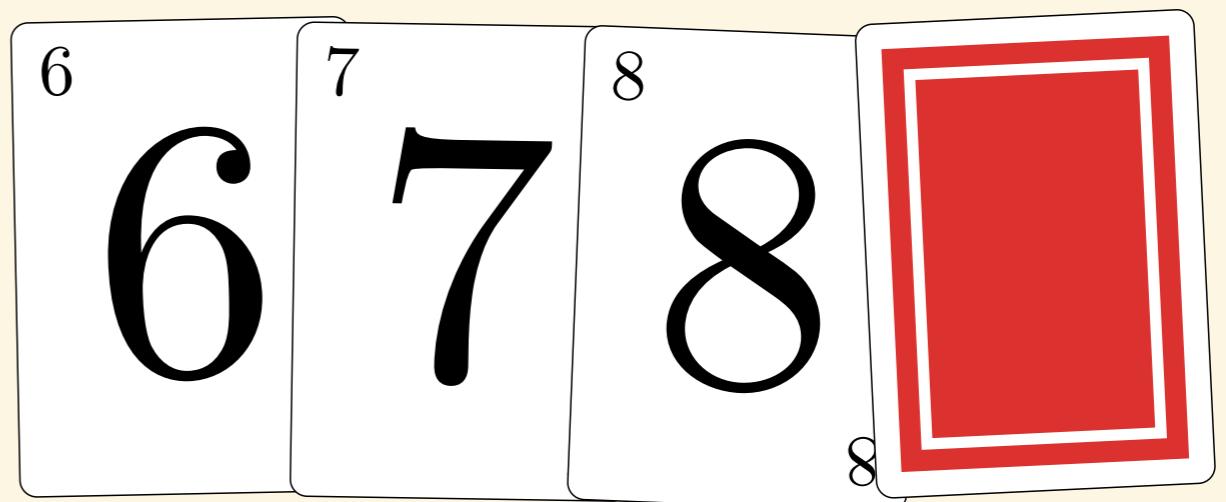
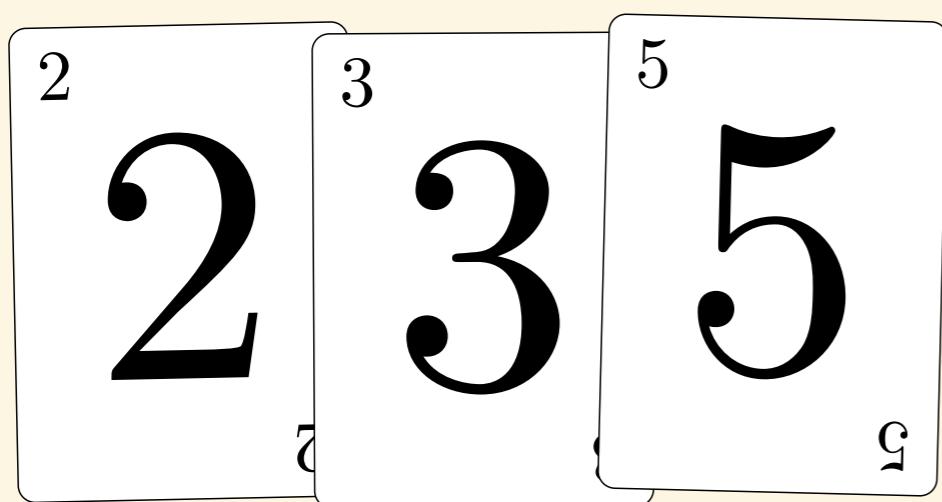


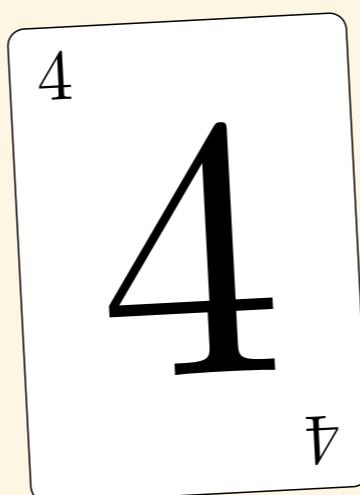
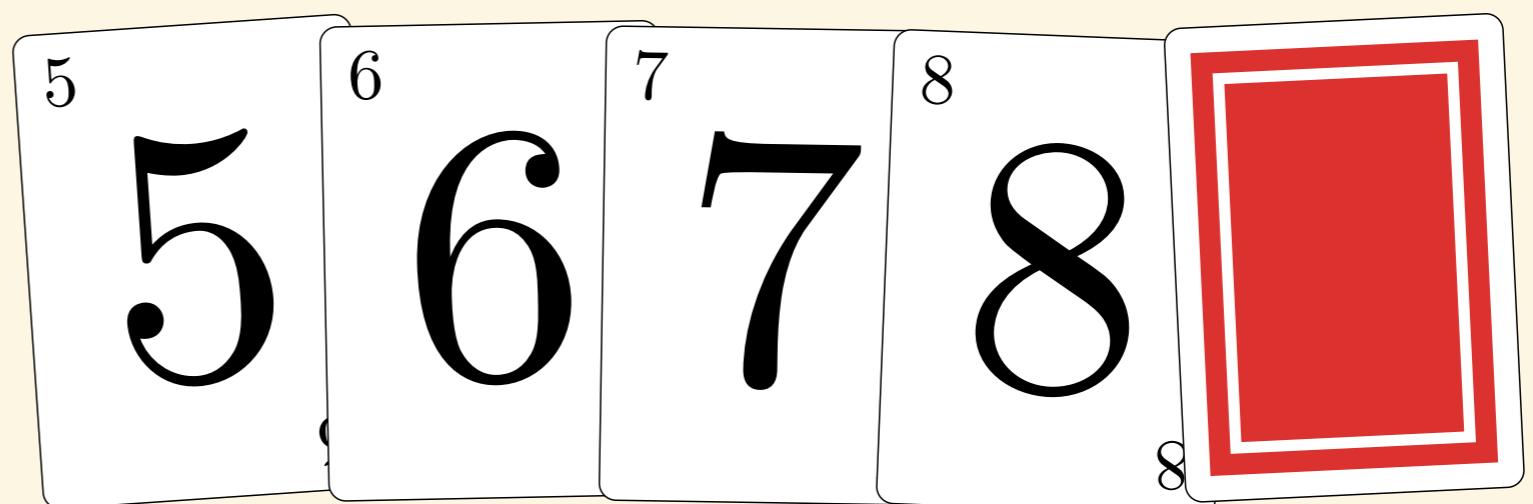
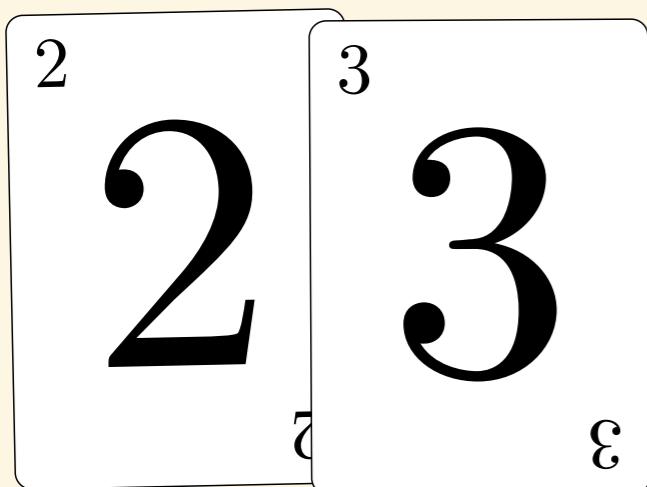


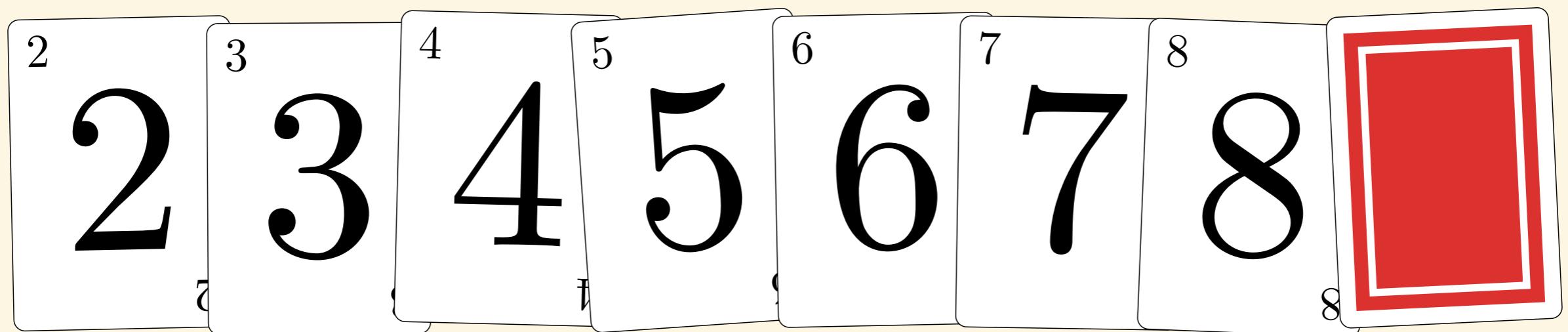


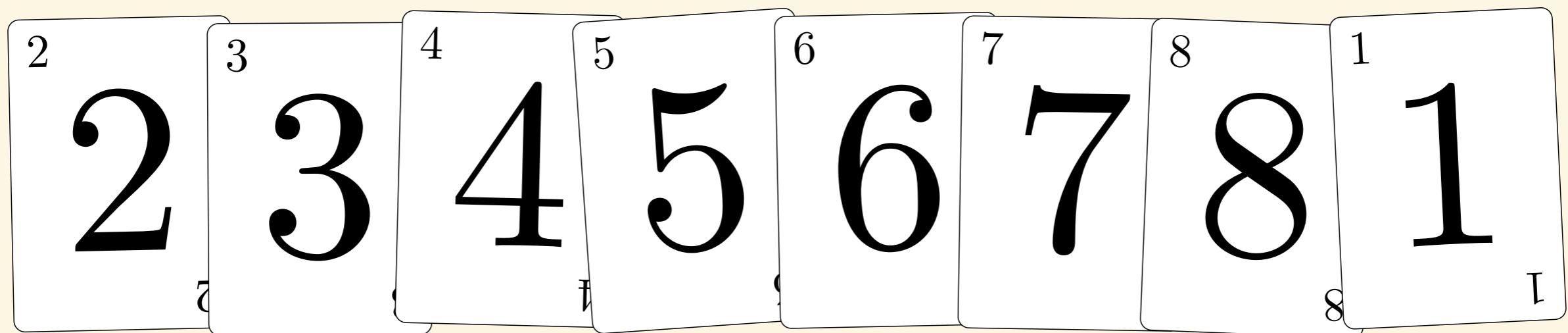


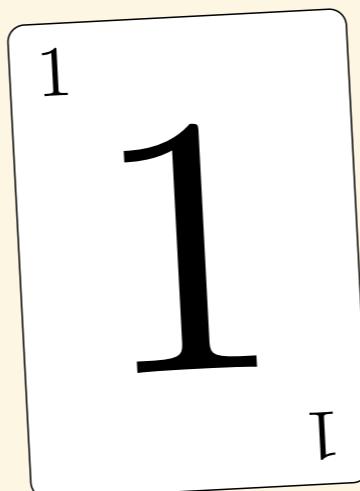
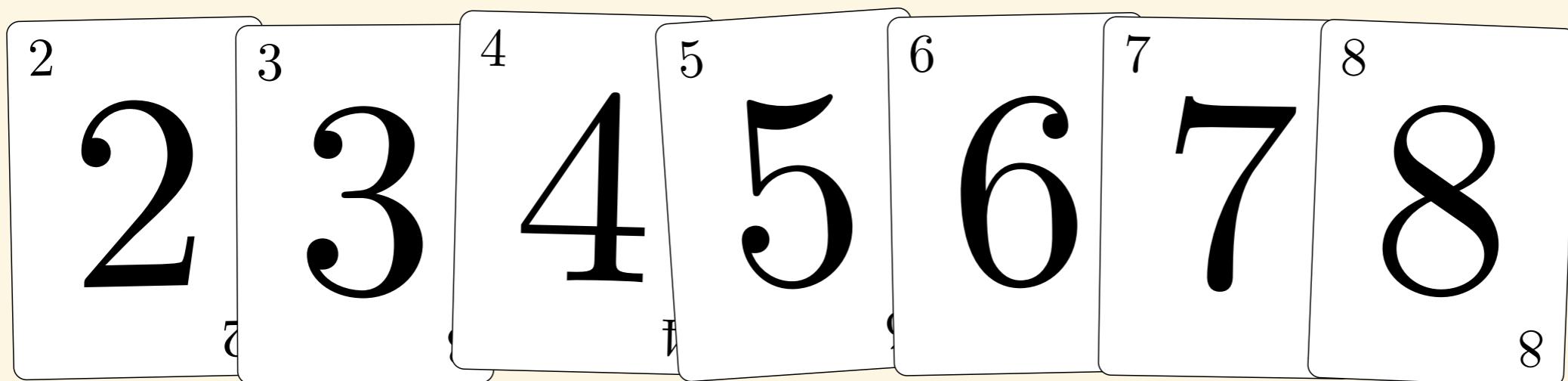


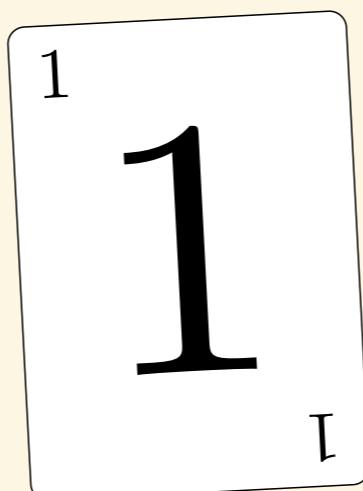
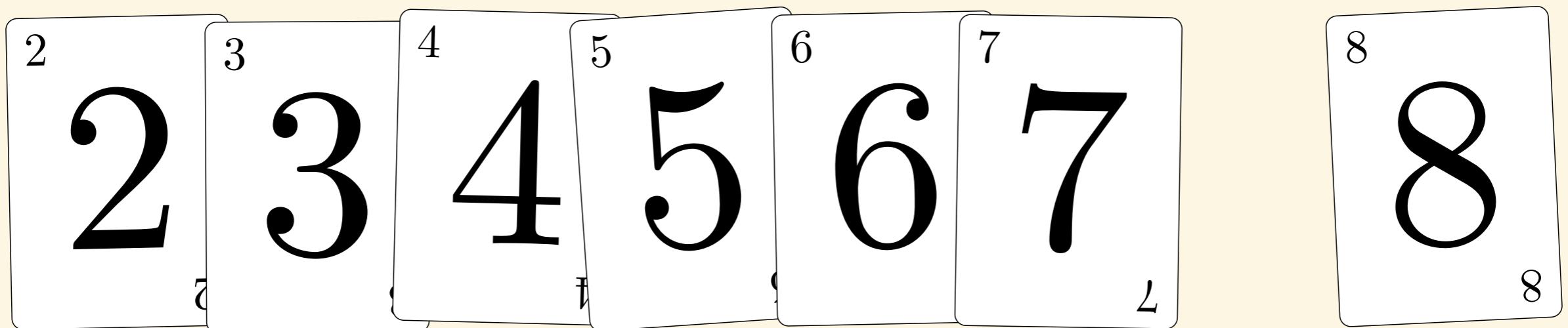


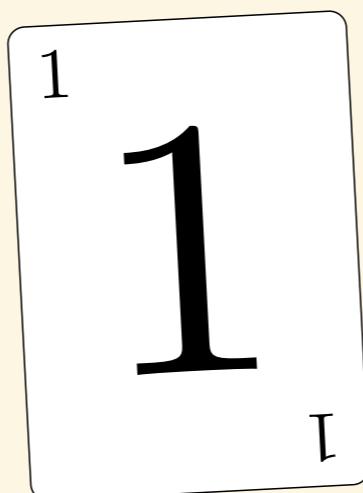
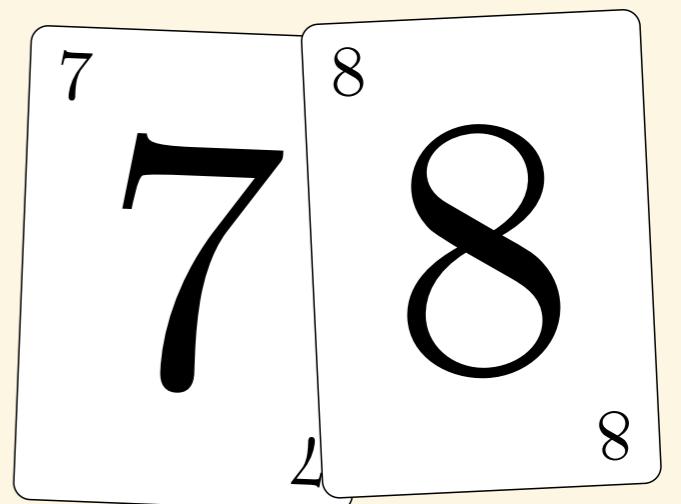
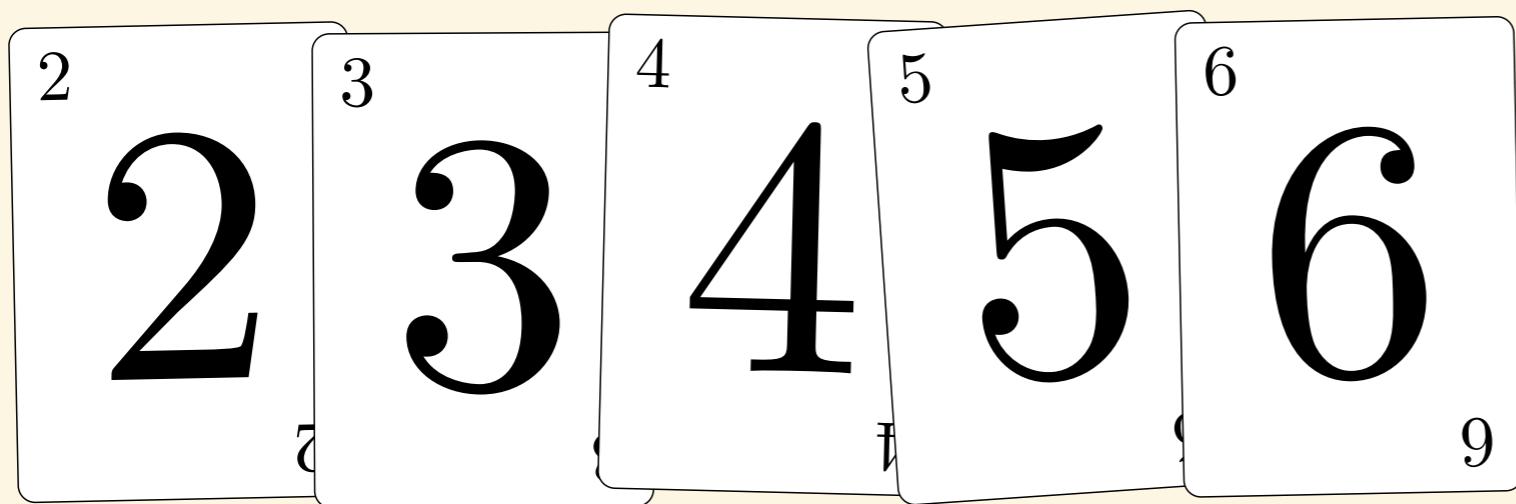


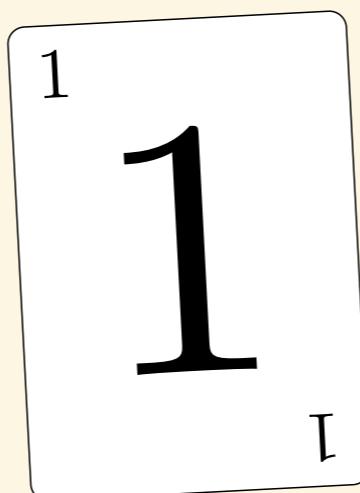
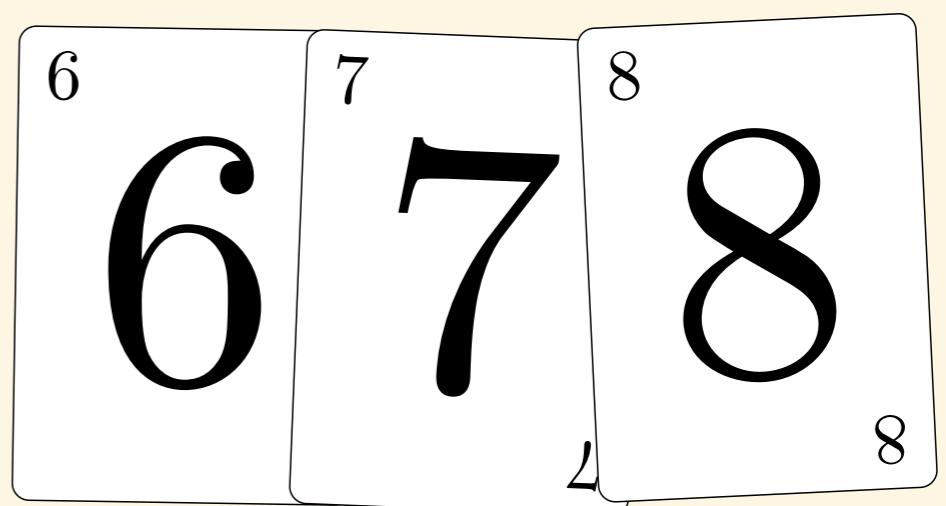
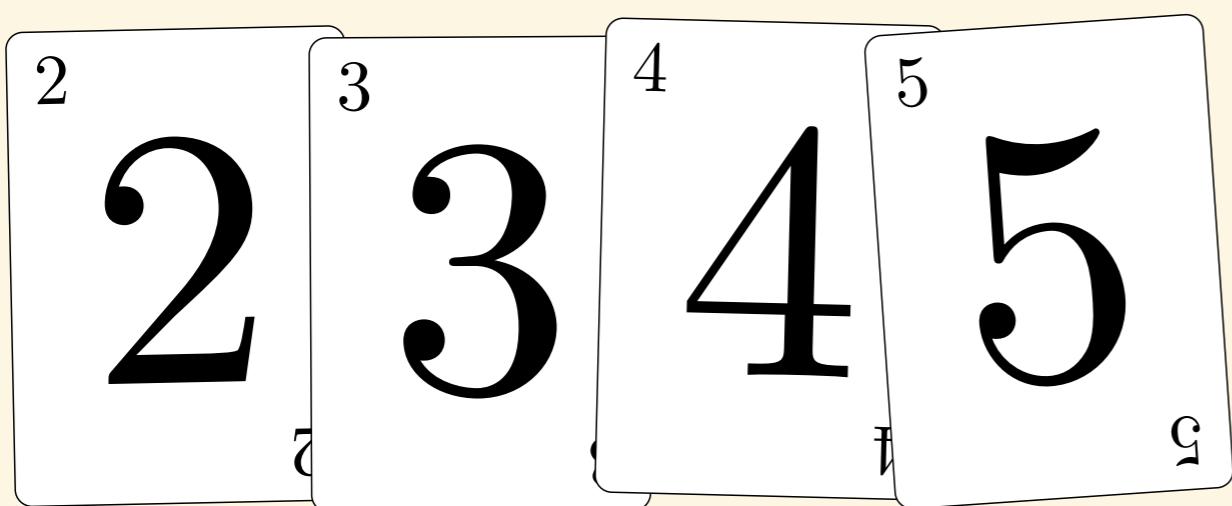


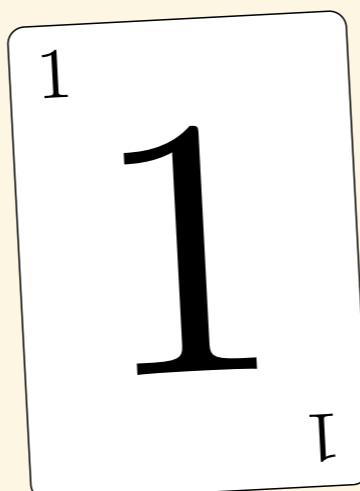
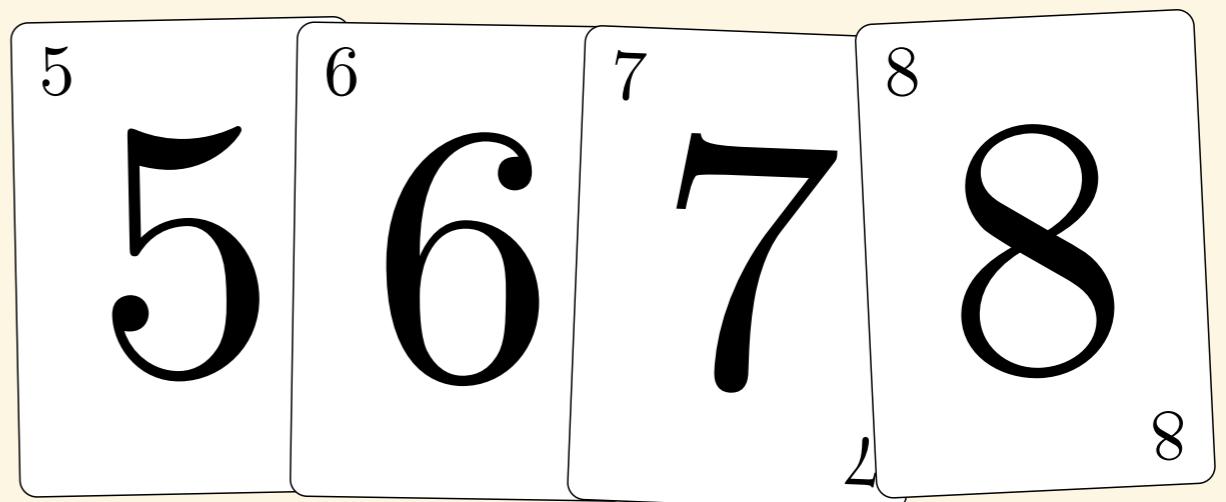
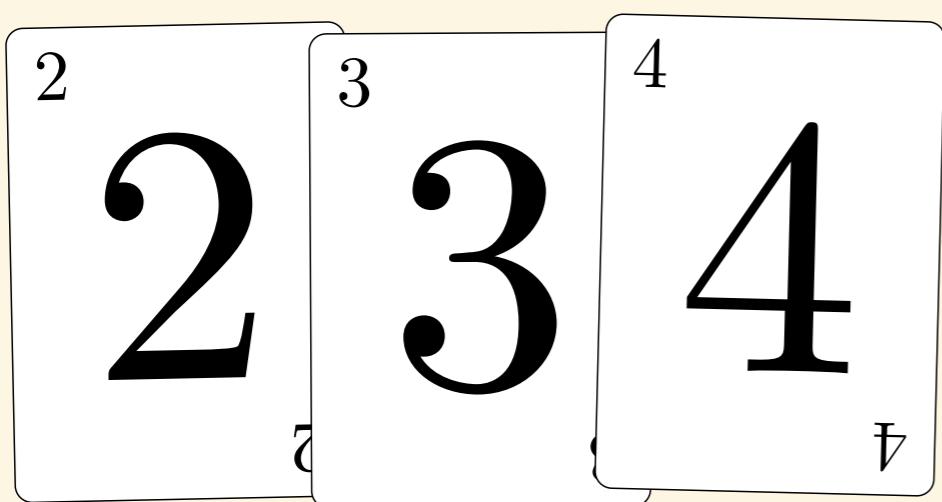


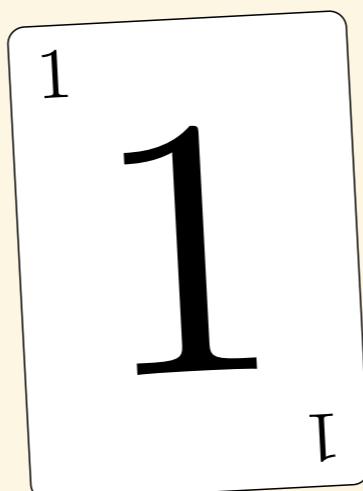
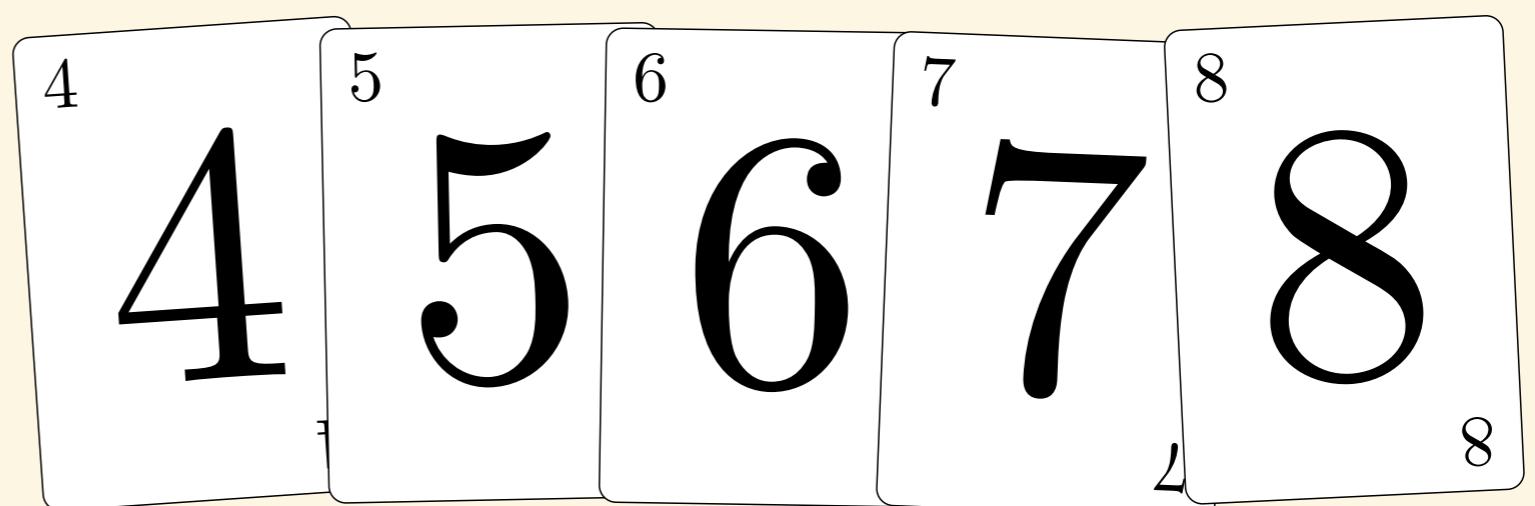
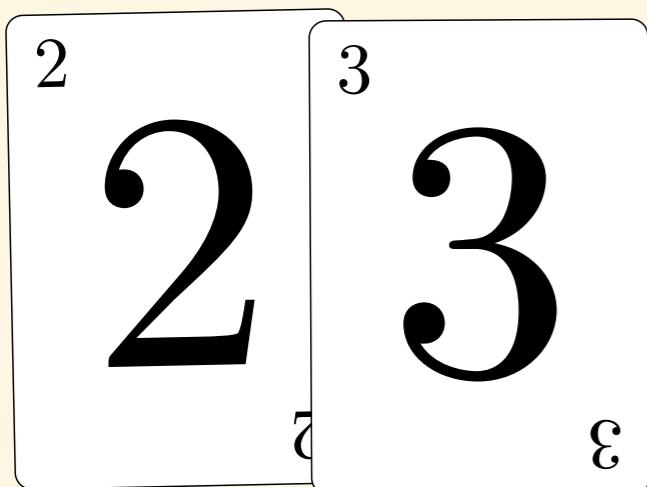


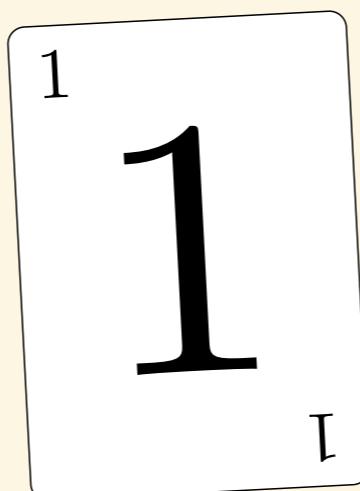
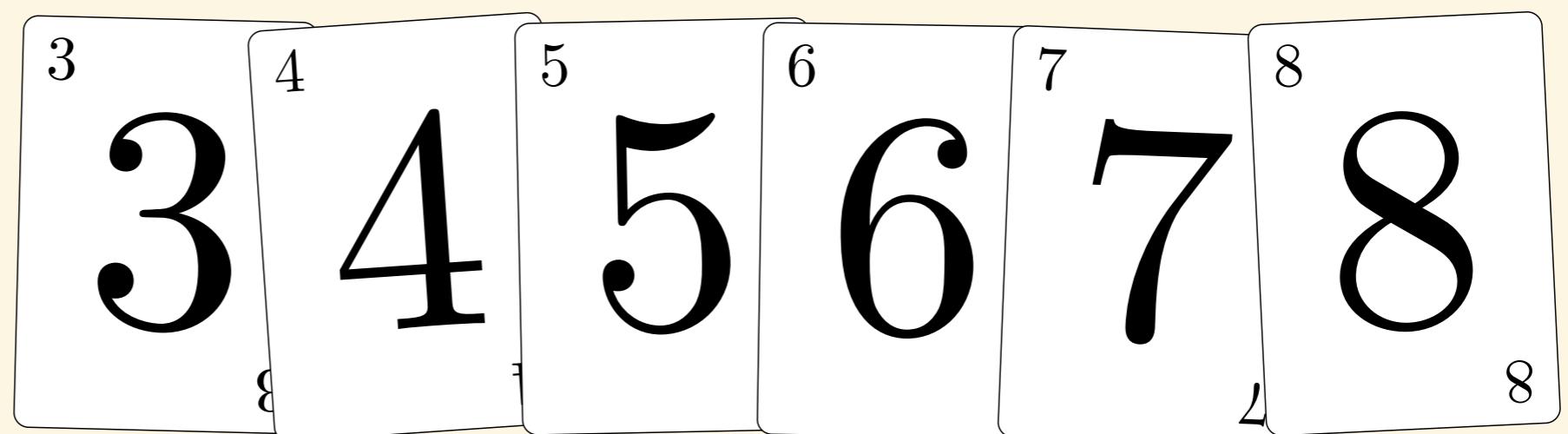
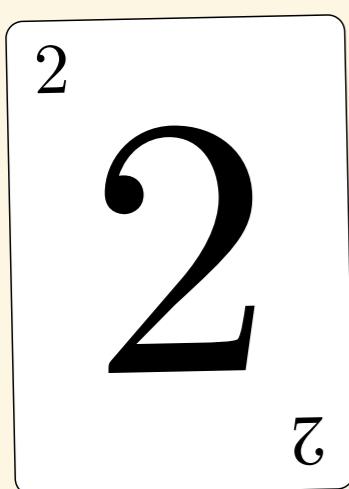


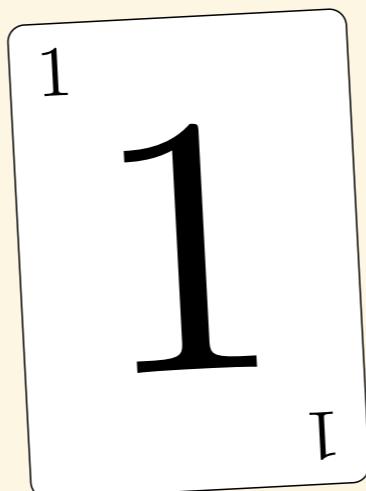
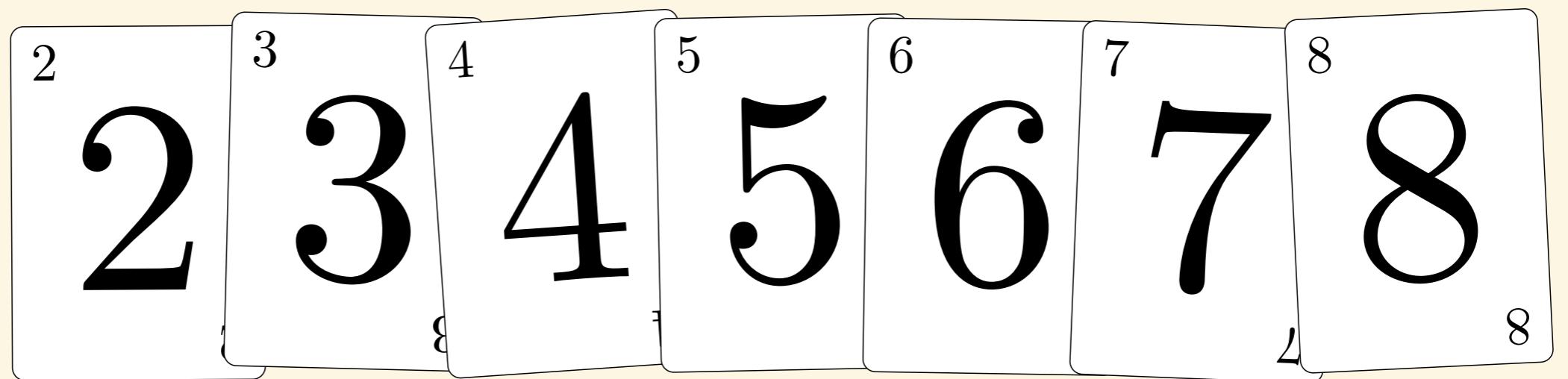


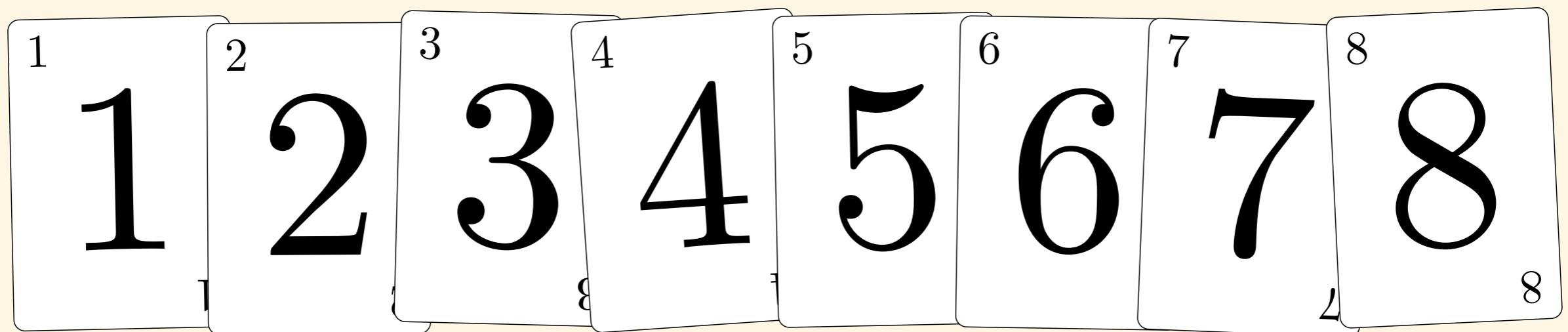












- Er dette bedre enn brute force?
Hvordan kan vi vite eller påstå det?
- Hvordan kom vi frem til denne algoritmen? Hva er det underliggende prinsippet?

Disse to tingene skal vi svare på i de neste to delene av forelesningen.



I'M
WAITING 2:5

Asymptotisk notasjon

Vi vil finne kjøretid
Men ignorere detaljer

Abstrakt maskin

- Kun enkle instruksjoner, som aritmetikk, flytting av data og programkontroll
- Disse tar konstant tid
- Vi kan håndtere heltall og flyttall
- Antar vanligvis at heltallene er maks $c \lg n$, for en eller annen c større eller lik 1

Verdiområdet her er ment å være akkurat stort nok til at vi får plass til tabellindekser (eller pekere) som kan adressere hele inputen vår.

Hva er n ?

- Problem: Relasjon mellom input og output
- Instans: Én bestemt input
- Problemstørrelse, n :
Lagringsplass som trengs for en instans
- Kan variere hvordan vi måler størrelse

- Kjøretid er en funksjon av problemstørrelsen; større problemer krever mer tid ... men hvor mye?
- Eksempel: Hvis vi teller operasjoner, og hver operasjon tar ett mikrosekund, hva rekker vi på et år?

$\lg n$	logaritmisk	
\sqrt{n}	—	1×10^{31}
n	lineær	3×10^{15}
$n \lg n$	linearitmisk	7×10^{13}
n^2	kvadratisk	6×10^7
n^3	kubisk	1×10^5
2^n	eksponentiell	51
$n!$	faktoriell	17

Alt på formen n^k kaller vi *polynomisk*
 (Ta en kikk på Problem 1-1)

Det store tallet vi hadde tidligere
hadde en eksponent på 85

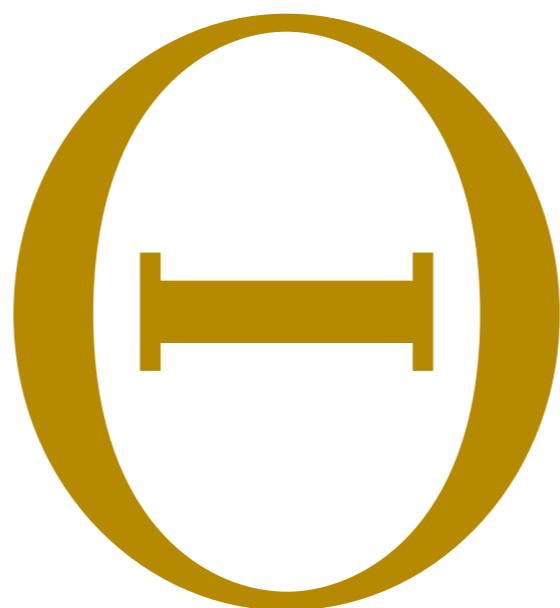
$\lg n$	logaritmisk	$2 \times 10^{949\,978\,419\,116\,565}$
\sqrt{n}	—	1×10^{31}
n	lineær	3×10^{15}
$n \lg n$	linearitmisk	7×10^{13}
n^2	kvadratisk	6×10^7
n^3	kubisk	1×10^5
2^n	eksponentiell	51
$n!$	faktoriell	17

Tallene her er altså hvor stor n
kan bli før det tar mer enn ett år
å bli ferdig, dersom vi bruker f(n)
mikrosekunder, der f(n) er
funksjonen til venstre.

Alt på formen n^k kaller vi *polynomisk*
(Ta en kikk på Problem 1-1)

- Vi er interessert i hvor fort kjøretiden vokser
- Vi er kun interessert i en veldig grov «størrelsesorden»
- Asymptotisk notasjon: Dropp konstanter og lavere ordens ledd

asymptotisk notasjon



Theta

asymptotisk notasjon

42 =

asymptotisk notasjon

$$42 = \Theta(1)$$

asymptotisk notasjon

$$42 = \Theta(1)$$

$$2n + 5 =$$

asymptotisk notasjon

$$42 = \Theta(1)$$

$$2n + 5 = \Theta(n)$$

asymptotisk notasjon

$$42 = \Theta(1)$$

$$2n + 5 = \Theta(n)$$

$$4n^2 + 100n =$$

asymptotisk notasjon

$$42 = \Theta(1)$$

$$2n + 5 = \Theta(n)$$

$$4n^2 + 100n = \Theta(n^2)$$

asymptotisk notasjon

$$42 = \Theta(1)$$

$$2n + 5 = \Theta(n)$$

$$4n^2 + 100n = \Theta(n^2)$$

$$an^3 + bn^2 =$$

asymptotisk notasjon

$$42 = \Theta(1)$$

$$2n + 5 = \Theta(n)$$

$$4n^2 + 100n = \Theta(n^2)$$

$$an^3 + bn^2 = \Theta(n^3)$$

asymptotisk notasjon

$$42 = \Theta(1)$$

$$2n + 5 = \Theta(n)$$

$$4n^2 + 100n = \Theta(n^2)$$

$$an^3 + bn^2 = \Theta(n^3)$$

$$n^k + n^{k-1} =$$

asymptotisk notasjon

$$42 = \Theta(1)$$

$$2n + 5 = \Theta(n)$$

$$4n^2 + 100n = \Theta(n^2)$$

$$an^3 + bn^2 = \Theta(n^3)$$

$$n^k + n^{k-1} = \Theta(n^k)$$

asymptotisk notasjon

$$42 = \Theta(1)$$

$$2n + 5 = \Theta(n)$$

$$4n^2 + 100n = \Theta(n^2)$$

$$an^3 + bn^2 = \Theta(n^3)$$

$$n^k + n^{k-1} = \Theta(n^k)$$

$$n + \lg n =$$

asymptotisk notasjon

$$42 = \Theta(1)$$

$$2n + 5 = \Theta(n)$$

$$4n^2 + 100n = \Theta(n^2)$$

$$an^3 + bn^2 = \Theta(n^3)$$

$$n^k + n^{k-1} = \Theta(n^k)$$

$$n + \lg n = \Theta(n)$$

asymptotisk notasjon

$$42 = \Theta(1)$$

$$2n + 5 = \Theta(n)$$

$$4n^2 + 100n = \Theta(n^2)$$

$$an^3 + bn^2 = \Theta(n^3)$$

$$n^k + n^{k-1} = \Theta(n^k)$$

$$n + \lg n = \Theta(n)$$

$$n^2 + n \lg n =$$

asymptotisk notasjon

$$42 = \Theta(1)$$

$$2n + 5 = \Theta(n)$$

$$4n^2 + 100n = \Theta(n^2)$$

$$an^3 + bn^2 = \Theta(n^3)$$

$$n^k + n^{k-1} = \Theta(n^k)$$

$$n + \lg n = \Theta(n)$$

$$n^2 + n \lg n = \Theta(n^2)$$

asymptotisk notasjon

$$42 = \Theta(1)$$

$$2n + 5 = \Theta(n)$$

$$4n^2 + 100n = \Theta(n^2)$$

$$an^3 + bn^2 = \Theta(n^3)$$

$$n^k + n^{k-1} = \Theta(n^k)$$

$$n + \lg n = \Theta(n)$$

$$n^2 + n \lg n = \Theta(n^2)$$

$$2^n + n^k =$$

$$42 = \Theta(1)$$

$$2n + 5 = \Theta(n)$$

$$4n^2 + 100n = \Theta(n^2)$$

$$an^3 + bn^2 = \Theta(n^3)$$

$$n^k + n^{k-1} = \Theta(n^k)$$

$$n + \lg n = \Theta(n)$$

$$n^2 + n \lg n = \Theta(n^2)$$

$$2^n + n^k = \Theta(2^n)$$

$$42 = \Theta(1)$$

$$2n + 5 = \Theta(n)$$

$$4n^2 + 100n = \Theta(n^2)$$

$$an^3 + bn^2 = \Theta(n^3)$$

$$n^k + n^{k-1} = \Theta(n^k)$$

$$n + \lg n = \Theta(n)$$

$$n^2 + n \lg n = \Theta(n^2)$$

$$2^n + n^k = \Theta(2^n)$$

$$n! + 2^n =$$

$$42 = \Theta(1)$$

$$2n + 5 = \Theta(n)$$

$$4n^2 + 100n = \Theta(n^2)$$

$$an^3 + bn^2 = \Theta(n^3)$$

$$n^k + n^{k-1} = \Theta(n^k)$$

$$n + \lg n = \Theta(n)$$

$$n^2 + n \lg n = \Theta(n^2)$$

$$2^n + n^k = \Theta(2^n)$$

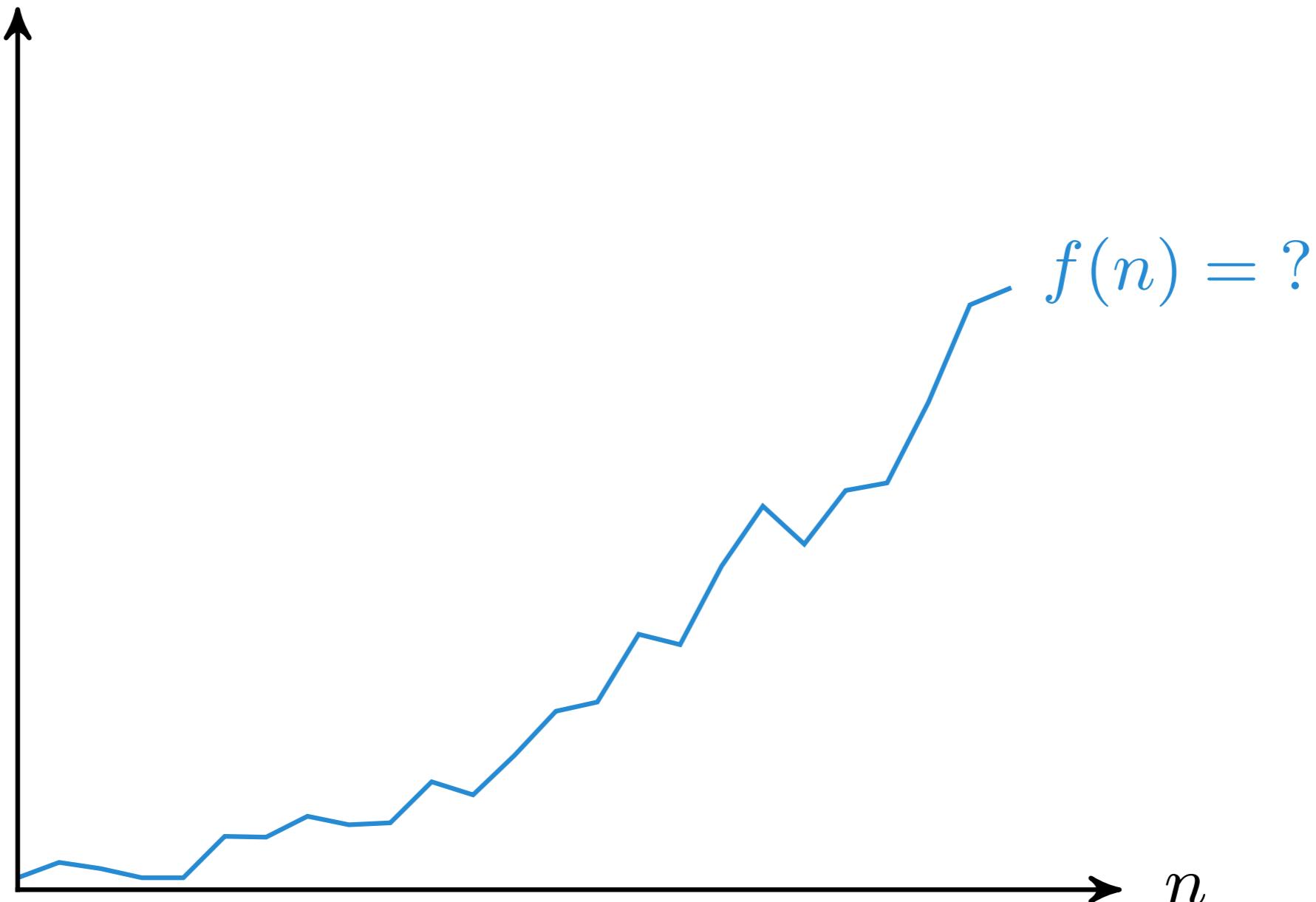
$$n! + 2^n = \Theta(n!)$$

Om vi ignorerer konstantfaktorer, så er vi bare interessert i om en funksjon garantert «går forbi» en annen, når problemet blir stort nok. Den har da høyere vekstrate eller «orden».

Vi velger det enkleste eksemplet vi kan som «representant» for en klasse med funksjoner som vokser like fort, og bruker det i den asymptotiske notasjonen.

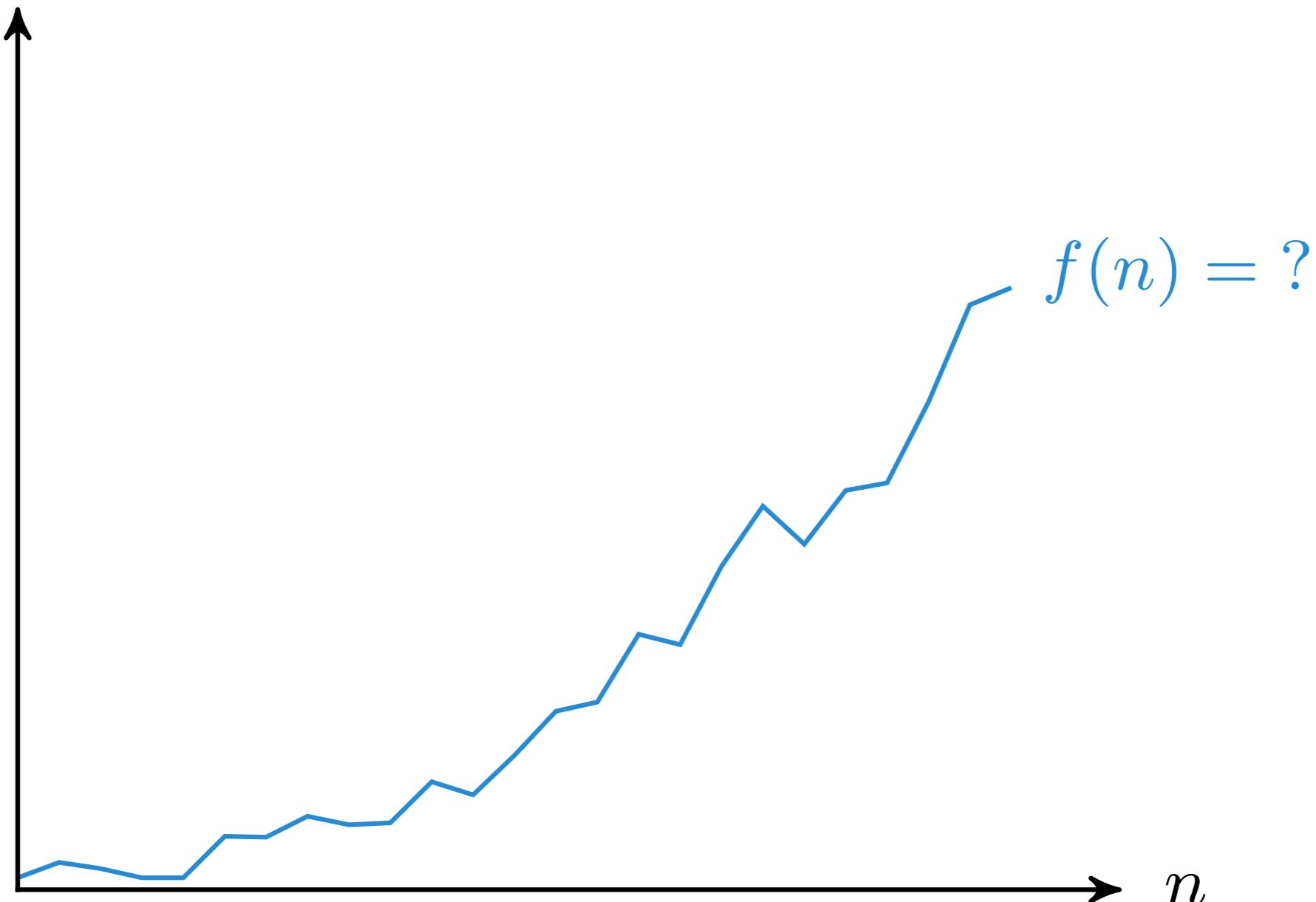
Merk: Vi kunne ha brukt uttrykket til venstre også inne i theta-notasjonen – det ville ha betydd akkurat det samme, men bare vært mer komplisert.

$f(n)$

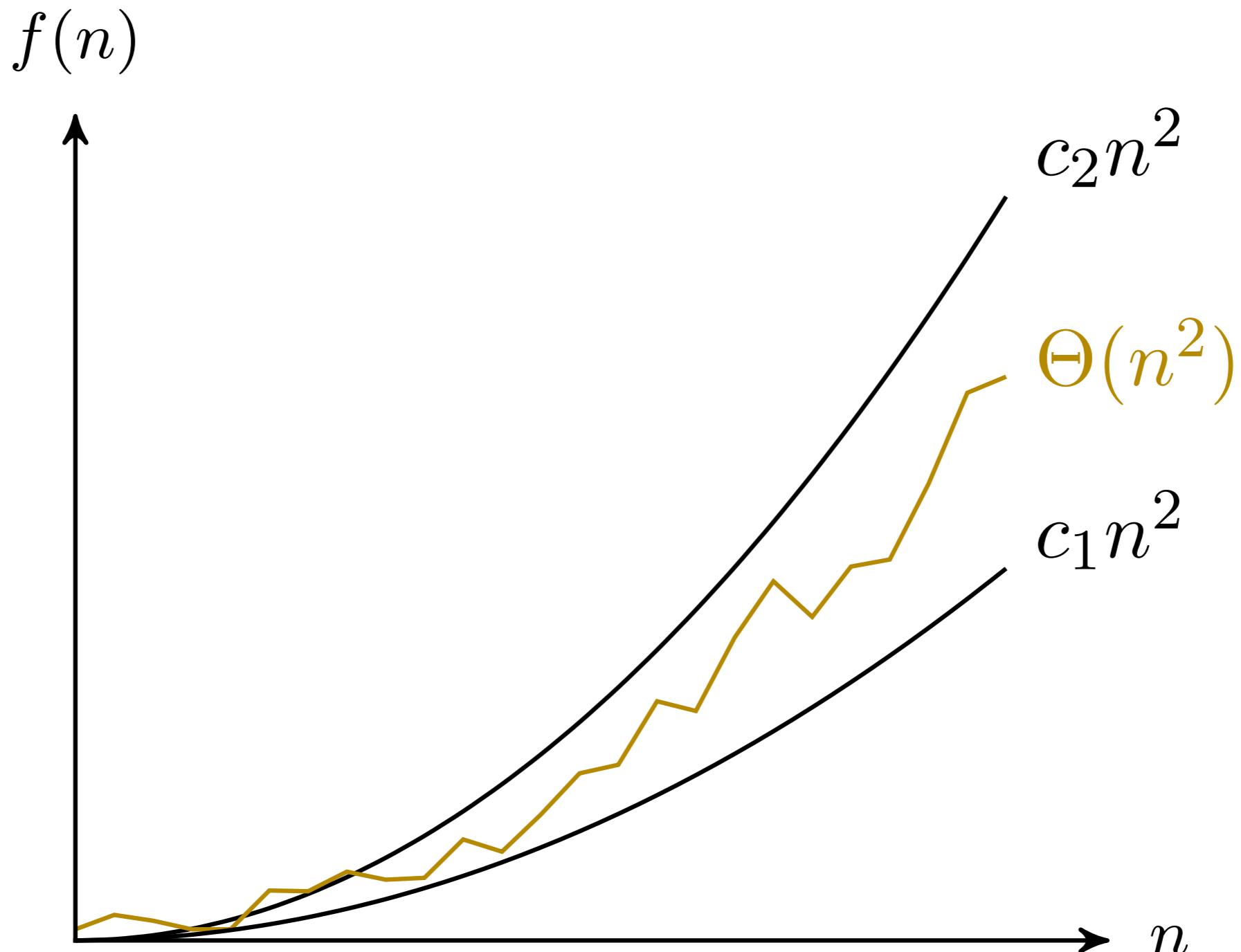


Nøyaktig kjøretid kan være hårete, varierende eller udefinert

$f(n)$

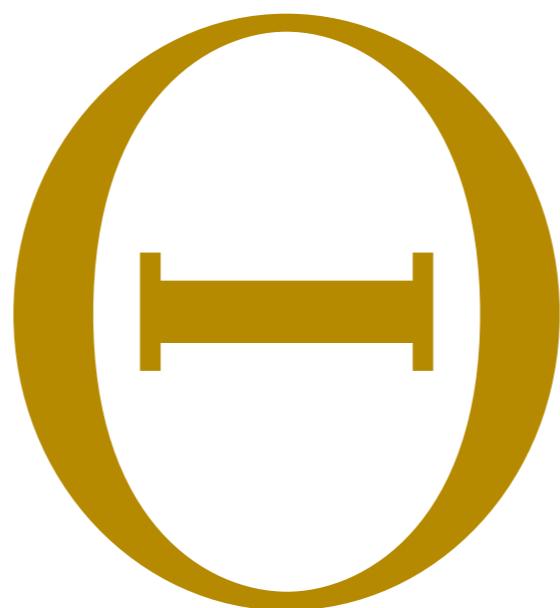


Vi ønsker å «skissere formen» til funksjonen

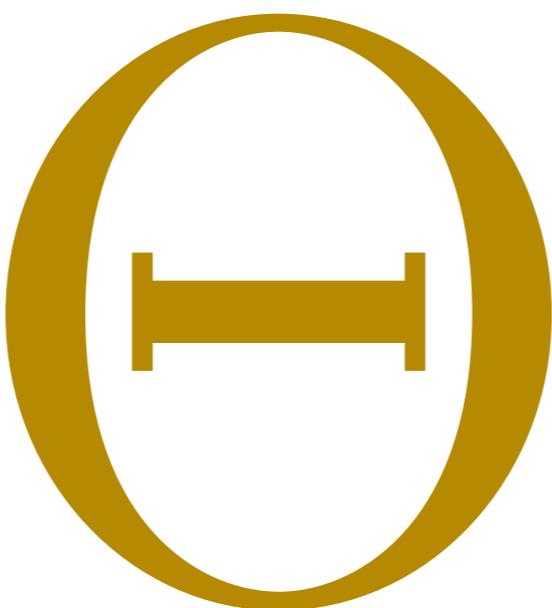
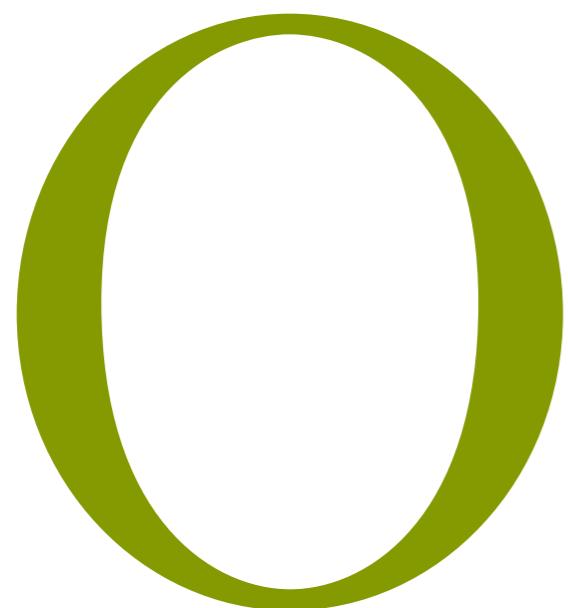


Ligger mellom to skaleringer av samme kurve, for store n

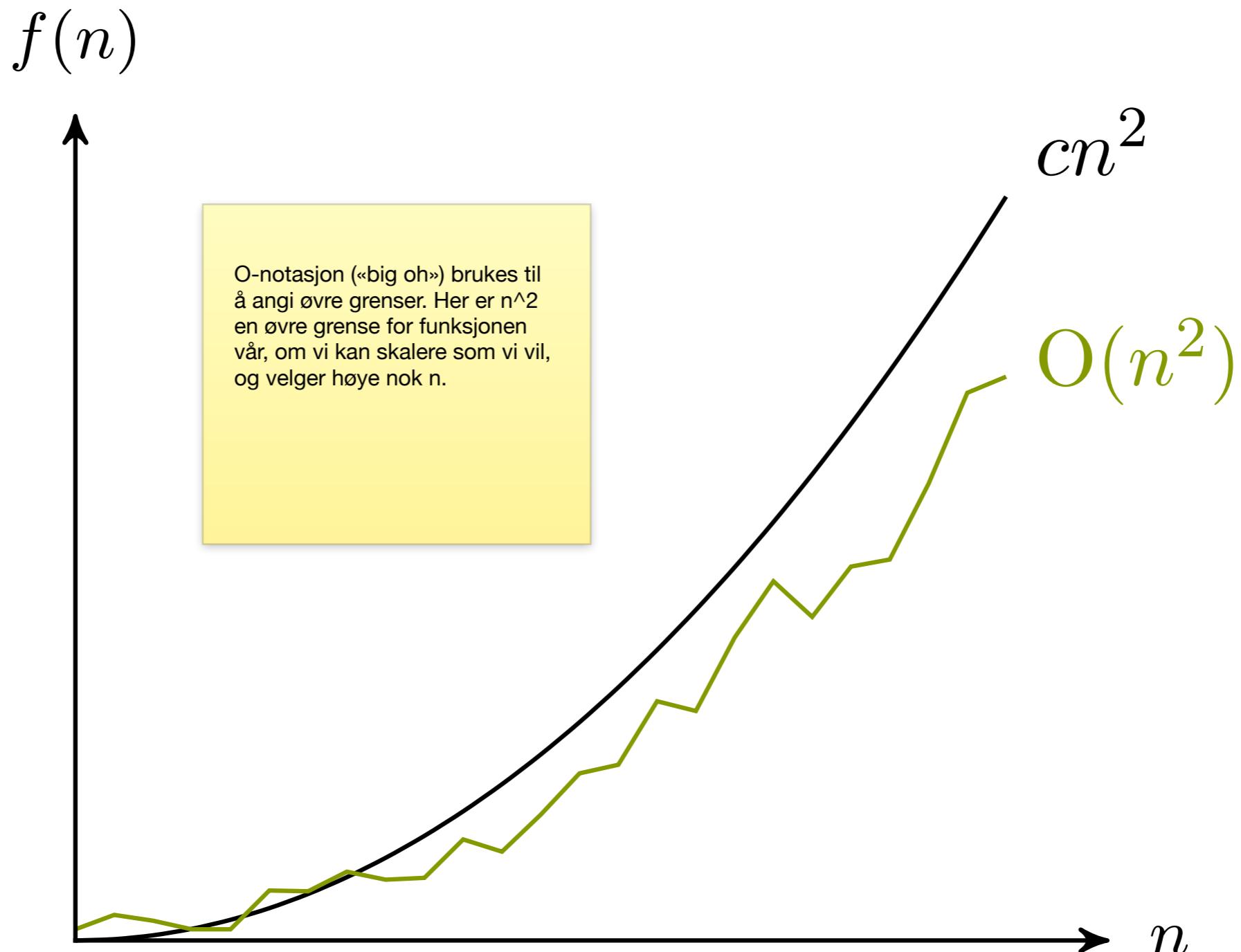
asymptotisk notasjon



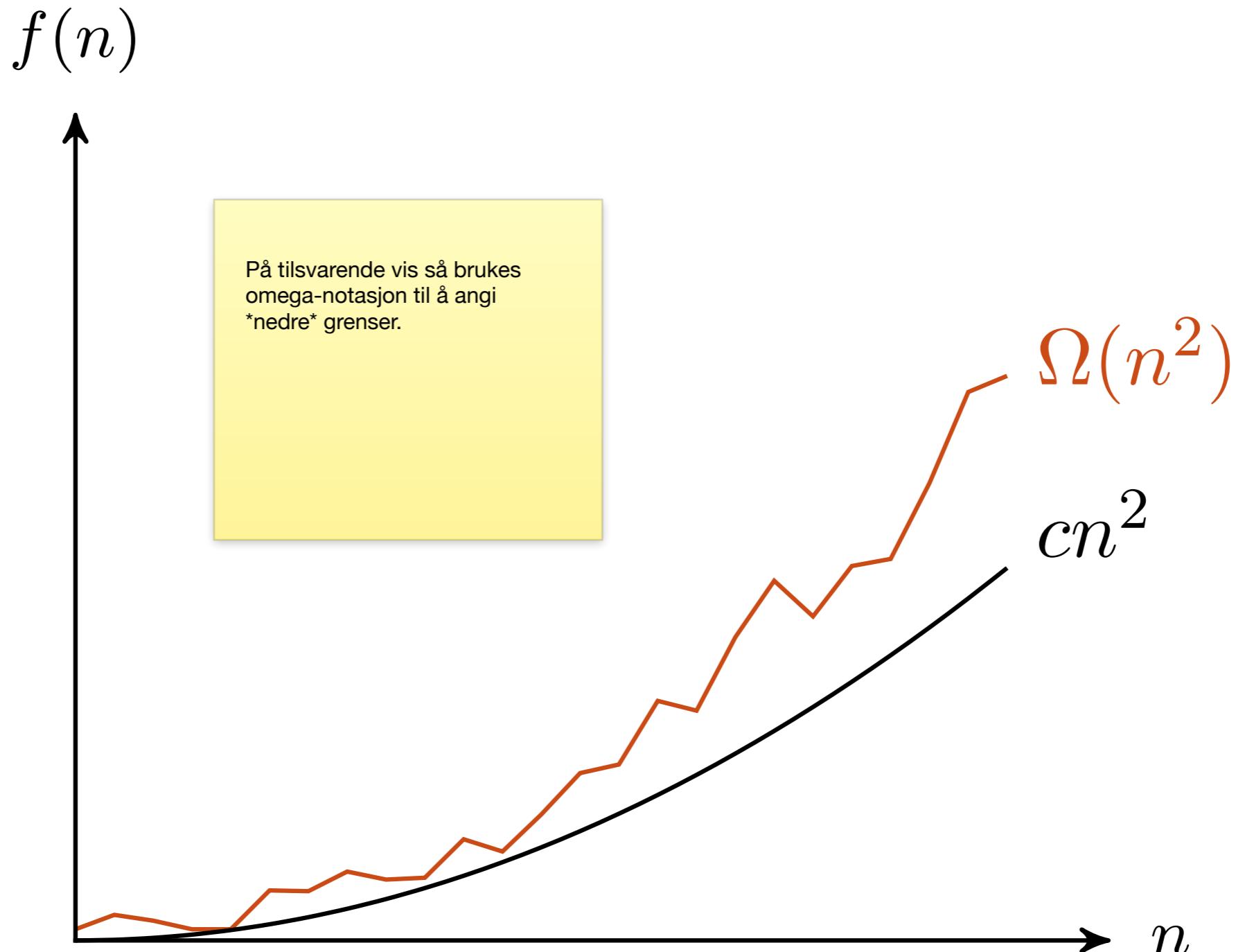
Theta



O, theta og omega



Ligger under skalert kurve, for store n



Ligger over skalert kurve, for store n

Her gir $g(n)$ en asymptotisk øvre grense for $f(n)$.

$$f(n) = O(g(n))$$

$$f(n) = O(g(n))$$

Hvorfor ikke $f \in O(g)$?

Slik som det defineres i boka, så produserer de asymptotiske operatorene *mengder* med funksjoner.

$$f(n) = O(g(n))$$

Hvorfor ikke $f \in O(g)$?

Historisk ... og praktisk. $n^2 + O(n)$, etc.

zengemass wird

zwi schen 0

$$\tau(n) = n \cdot \sum_{k=1}^n \frac{1}{k} - r,$$

für den Werth n nicht erreicht. Da nun nach der Summenformel (1a) des vor. Abschnitts

$$\sum_{k=1}^n \frac{1}{k} = E + \log n + \frac{1}{2n} - \dots$$

$$\tau(n) = n \log n + O(n),$$

durch das Zeichen $O(n)$ eine Grösse ausdrücken, auf die die Ordnung von n nicht übergeht. In Bezug auf n die Ordnung von n in sich sie wirklich Glieder von der Ordnung n in sich bei dem bisherigen Schlussverfahren dargestellt. Dies näher untersuchen, betrachten wir zweitens $S(n)$, welche die Summe aller Theiler Zahl n bestimmt. Für die entsprechende Funktion

$$\sigma(n) = \sum_{k=1}^n S(k)$$

2, 3) des 11. Abschnittes den anderen Ausdruck

$$\sigma(n) = \sum_{k=1}^n k \cdot \left[\frac{n}{k} \right],$$

mit Hilfe der allgemeinen Formel (8) des

26

Fra «Die analytische Zahlentheorie» av P. G. H. Bachmann (1894).
<https://archive.org/details/dieanalytische00bachgoog>

praktis

Riemann vermutete ferner, ohne dies zu beweisen, daß die Richtigkeit aufführen zu können, daß die Zetafunktion Nullstellen dem Streich hat.

für die Richtigkeit aufführen zu können, daß die Zetafunktion Nullstellen dem Streich hat.

I. Es gibt unendlich viele Nullstellen von $\xi(s)$ sechs Eigenschaften besitzt:

0 $\leq s \leq 1$, die natürlich symmetrisch zur reellen Achse $s = \frac{1}{2}$ verteilt liegen.

II. Wenn für $T > 0$ unter $N(T)$ die Anzahl

stellen (mehrfache mehrfach gezählt) verstanden werden, Ordinate zwischen 0 (exkl.) und T (inkl.) liegt,

(1)
$$N(T) = \frac{1}{2\pi} T \log T - \frac{1 + \log(2\pi)}{2\pi} T + O(\log T)$$

Hierbei verstehe ich unter der Bezeichnung $O(1)$ T von einer gewissen Stelle an unterhalb einer festen Grenze ξ alle Werte an, wenn $g(x)$ eine für alle reellen x definierte und positive Funktion ist, die für alle $x > \xi$ gewisse Werte an annimmt, wobei $f(x) = O(g(x))$ bedeutet, daß

$$\limsup_{x \rightarrow \infty} \frac{f(x)}{g(x)}$$

endlich ist, d. h., daß es zwei Zahlen ξ und A gibt, so daß für alle $x \geq \xi$ gilt

$$f(x) < Ag(x)$$

Dieselbe ist für jedes T endlich, da es sich um ein komplexes Gebiet handelt.

praktis

Fra «Handbuch der Lehre von der Verteilung der Primzahlen» av E. Landau (1909)
<https://archive.org/details/handbuchderlehre01landuoft>

Opprinnelig: Notasjonen representerte en anonym, ukjent representant for en klasse funksjoner

Nå: Notasjonen representerer selve klassen ... men brukes fortsatt på den opprinnelige måten; «abuse of notation»

$$f(n) = o(g(n))$$

$$f(n) = \omega(g(n))$$

$$f(n) = o(g(n))$$

$$f(n) = \omega(g(n))$$

Som før, men for *alle* $c > 0$

Disse er strengere varianter av de store operatorene.

ω >

Ω ≥

Θ =

0 ≤

o <

Klasser av input

Best, verst og forventet

the Good the Bad and the Average



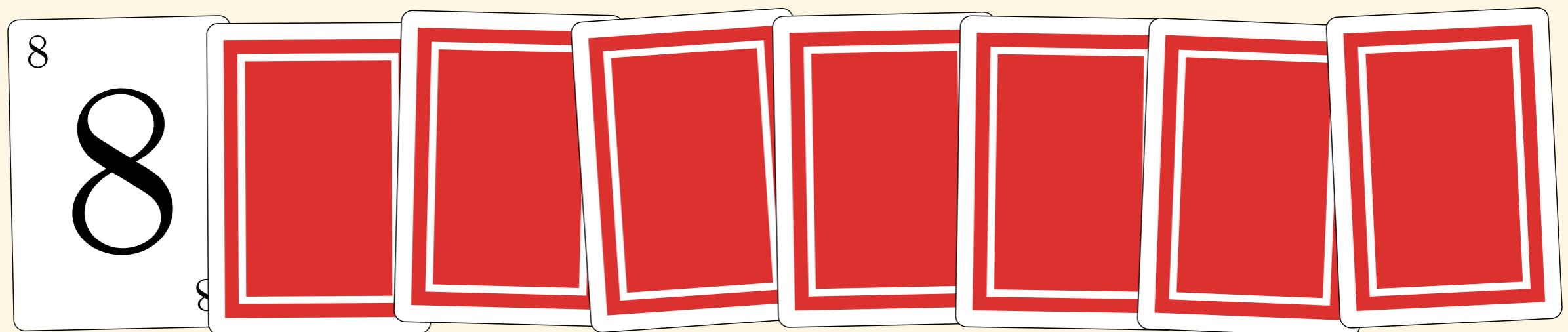
- **Kjøretid:** Funksjon av problemstørrelse
- **Best-case:**
Beste mulige kjøretid for en gitt størrelse
- **Worst-case:** Verste mulige
- **Average-case:**
Forventet, gitt en sannsynlighetsfordeling
- Bruker vanligvis worst-case

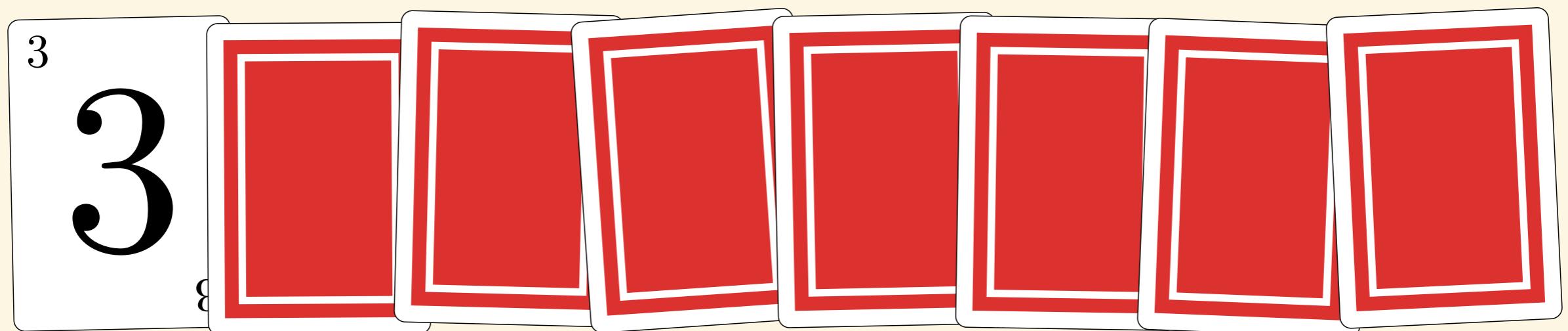
Om vi ikke har noen eksplisitt sannsynlighetsfordeling, antar vi bare at alle inputs er like sannsynlige.

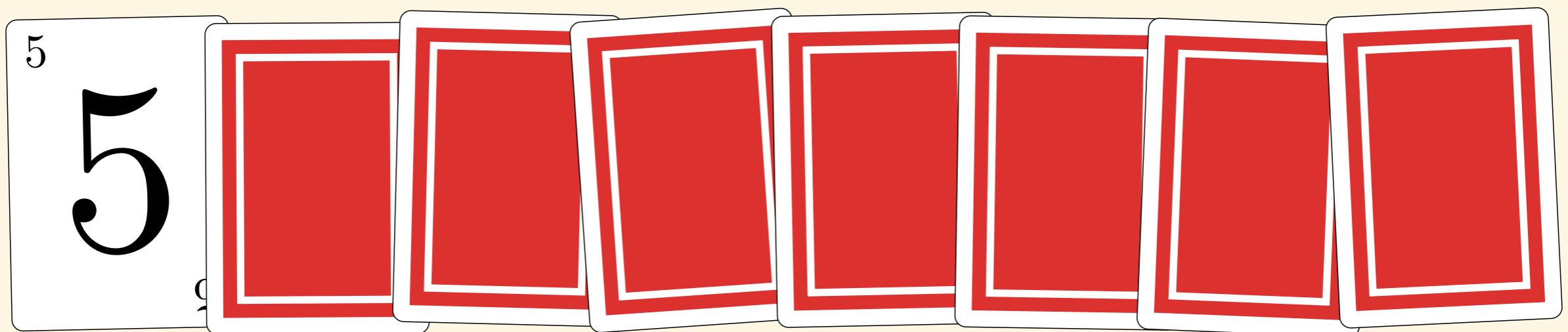
Så ...

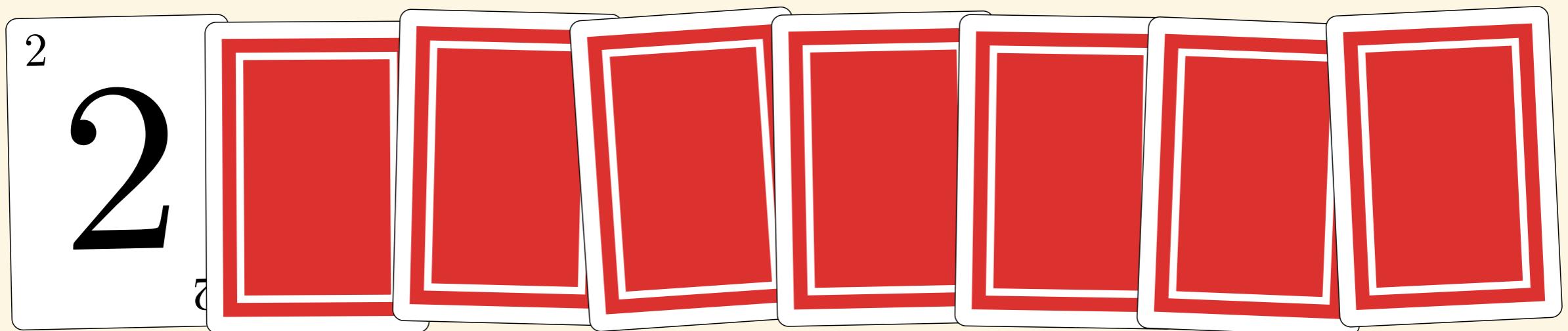
Hva kan vi nå si om sorteringen vår? Er den bedre enn brute force?

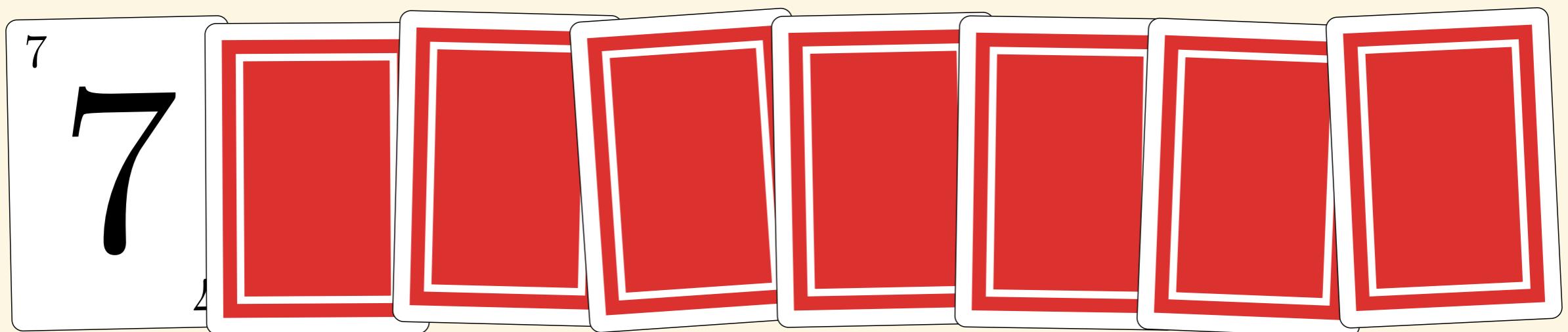
Først, brute force ...

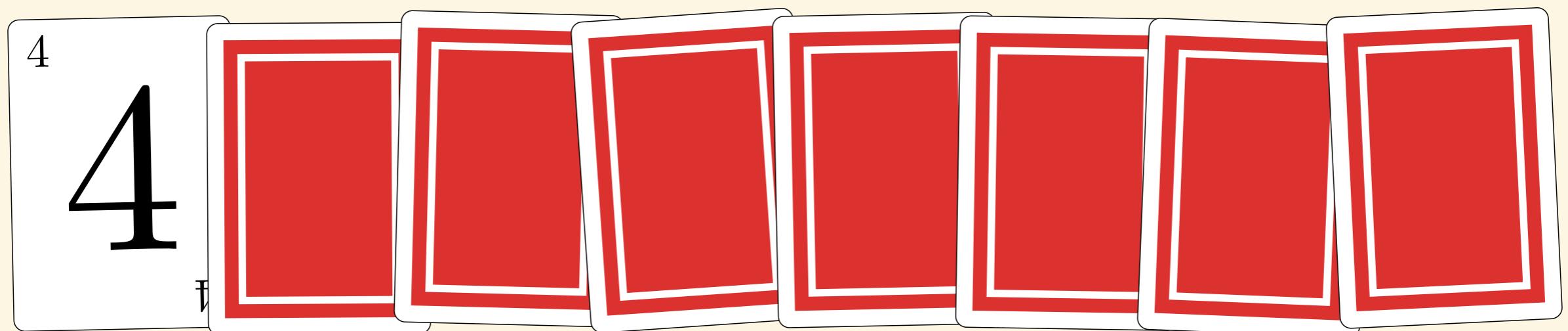


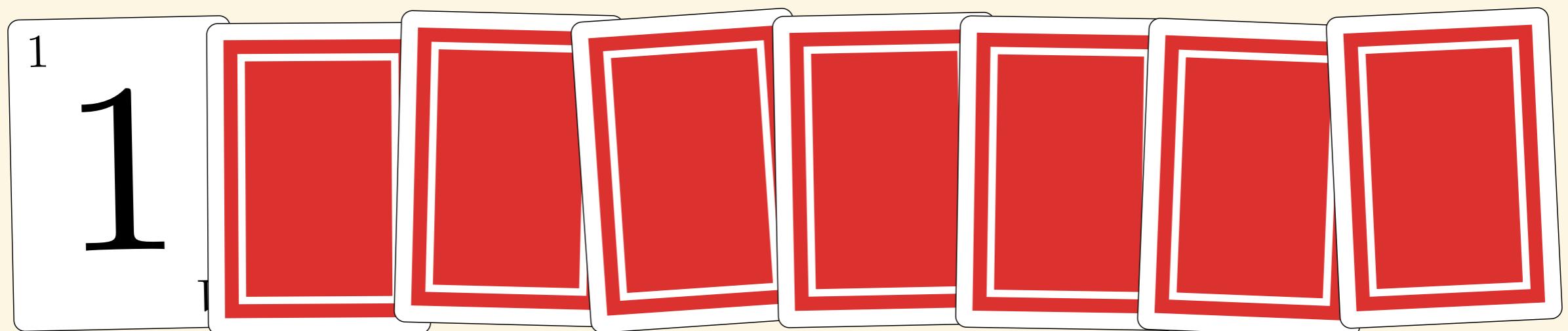


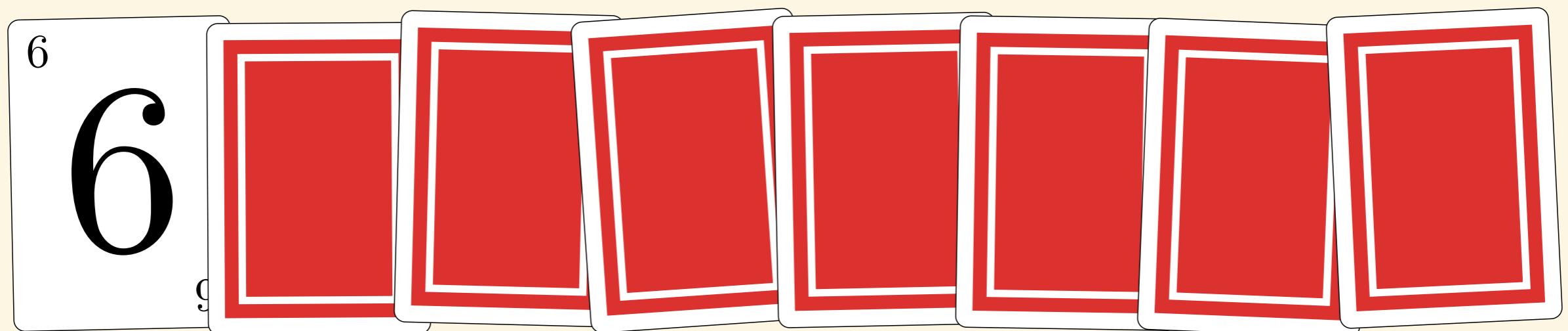


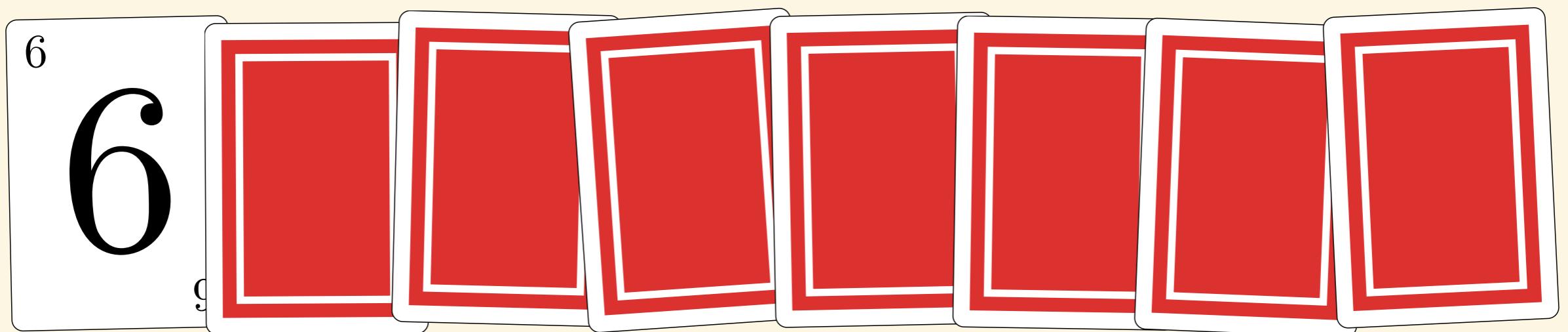






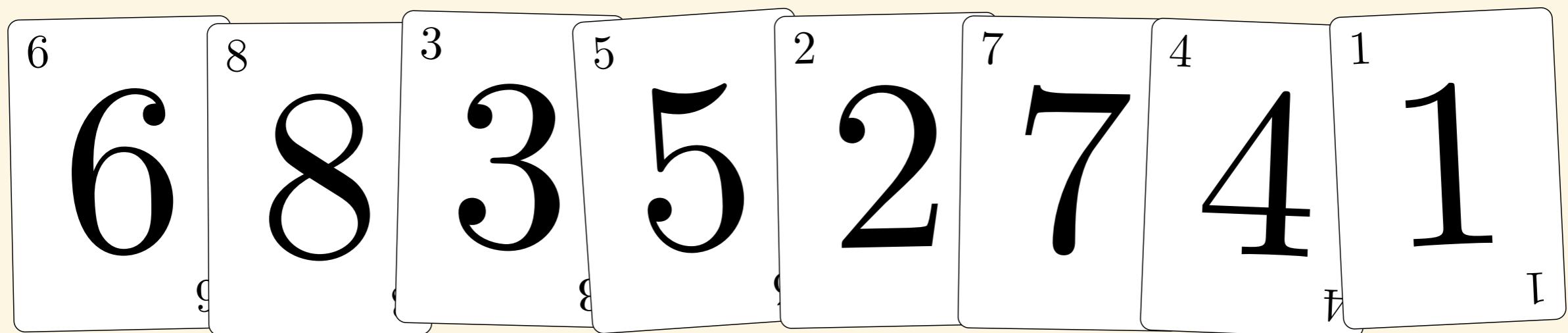






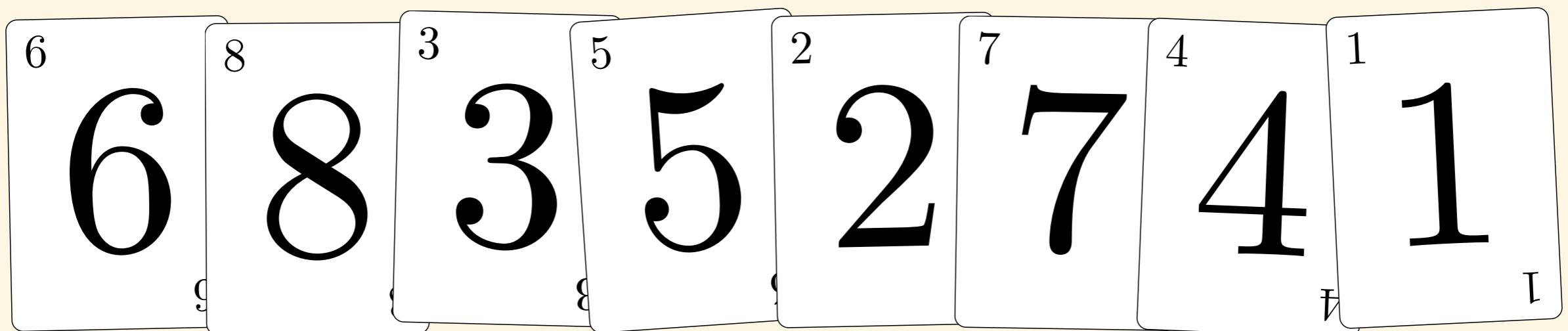
8

Vi har 8 muligheter for første posisjon. For hvert kort vi velger der, har vi igjen 7 muligheter til neste posisjon, etc.



8 7 6 5 4 3 2 1

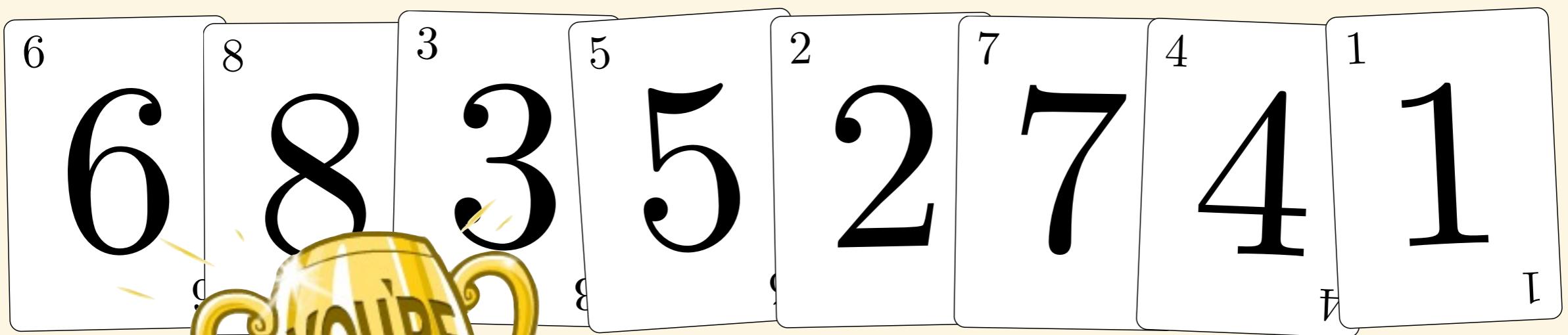
Mulighetene for hver posisjon er uavhengige av hverandre, så det totale antall muligheter blir produktet. Alt dette må prøves.



$$8 \times 7 \times 6 \times 5 \times 4 \times 3 \times 2 \times 1$$

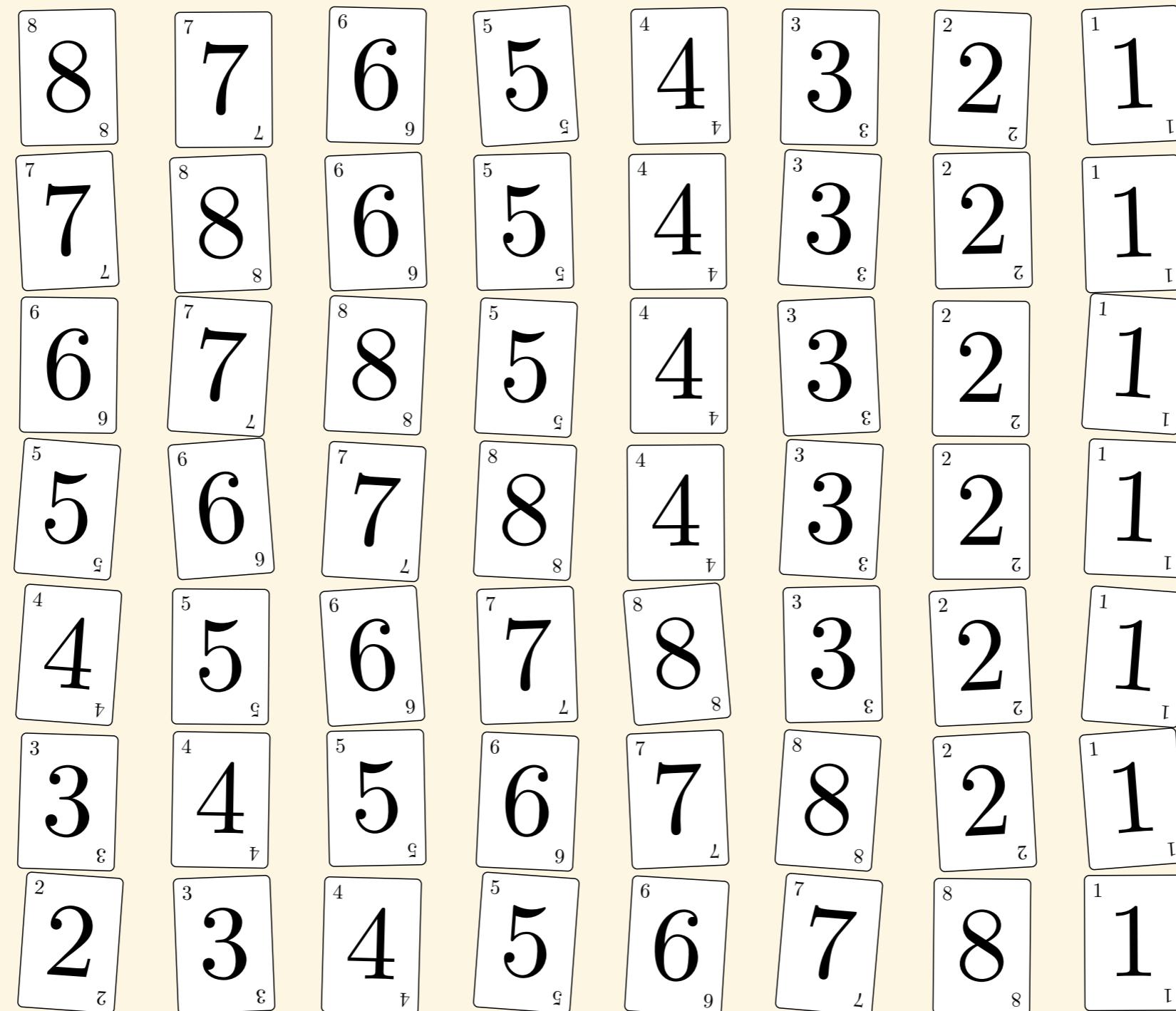
Og dette er jo definisjonen på $n!$
(''n fakultet'', på engelsk ''n factorial'').

Og den er noe av det verste vi
kan komme borti ...

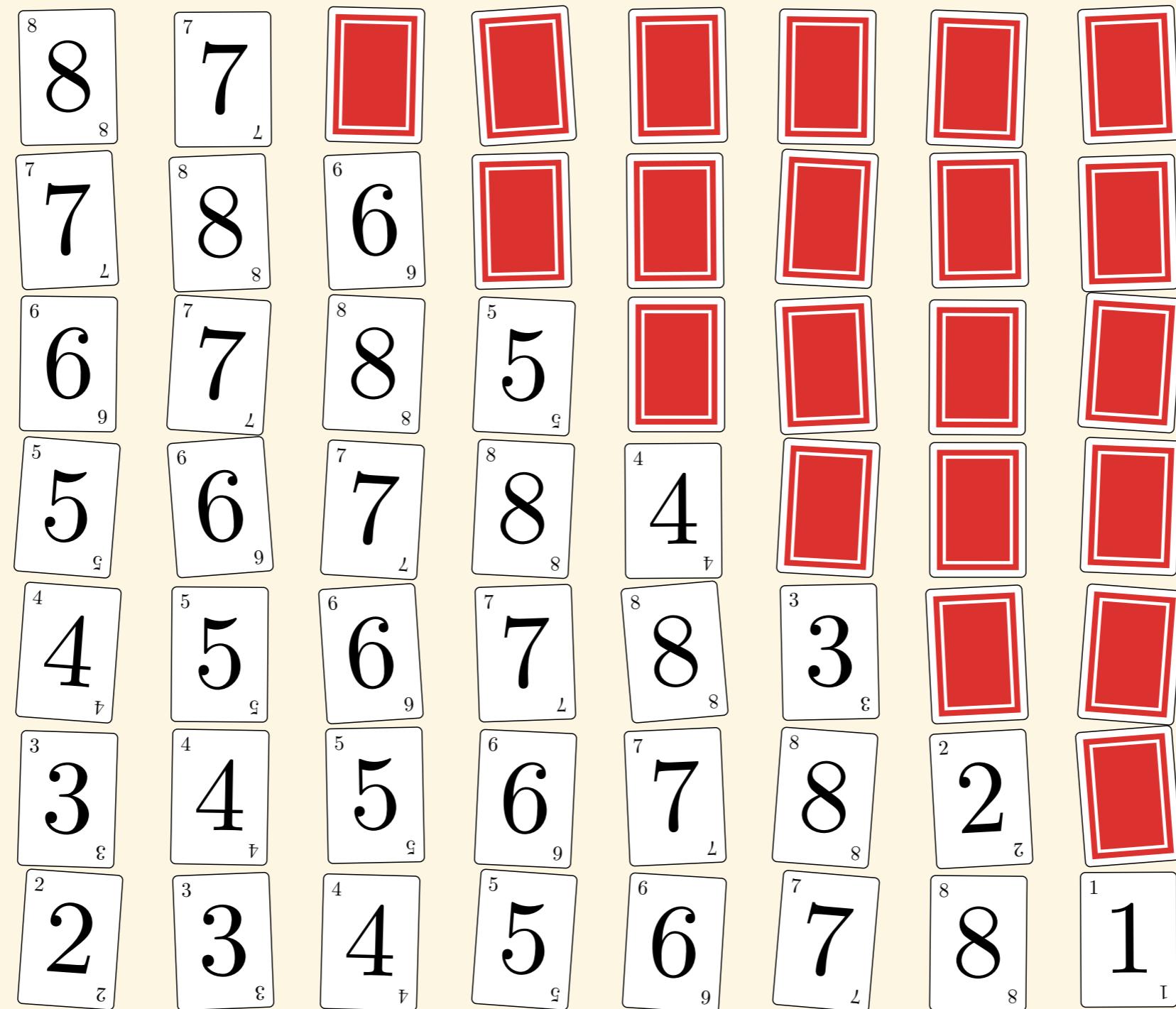


$n!$

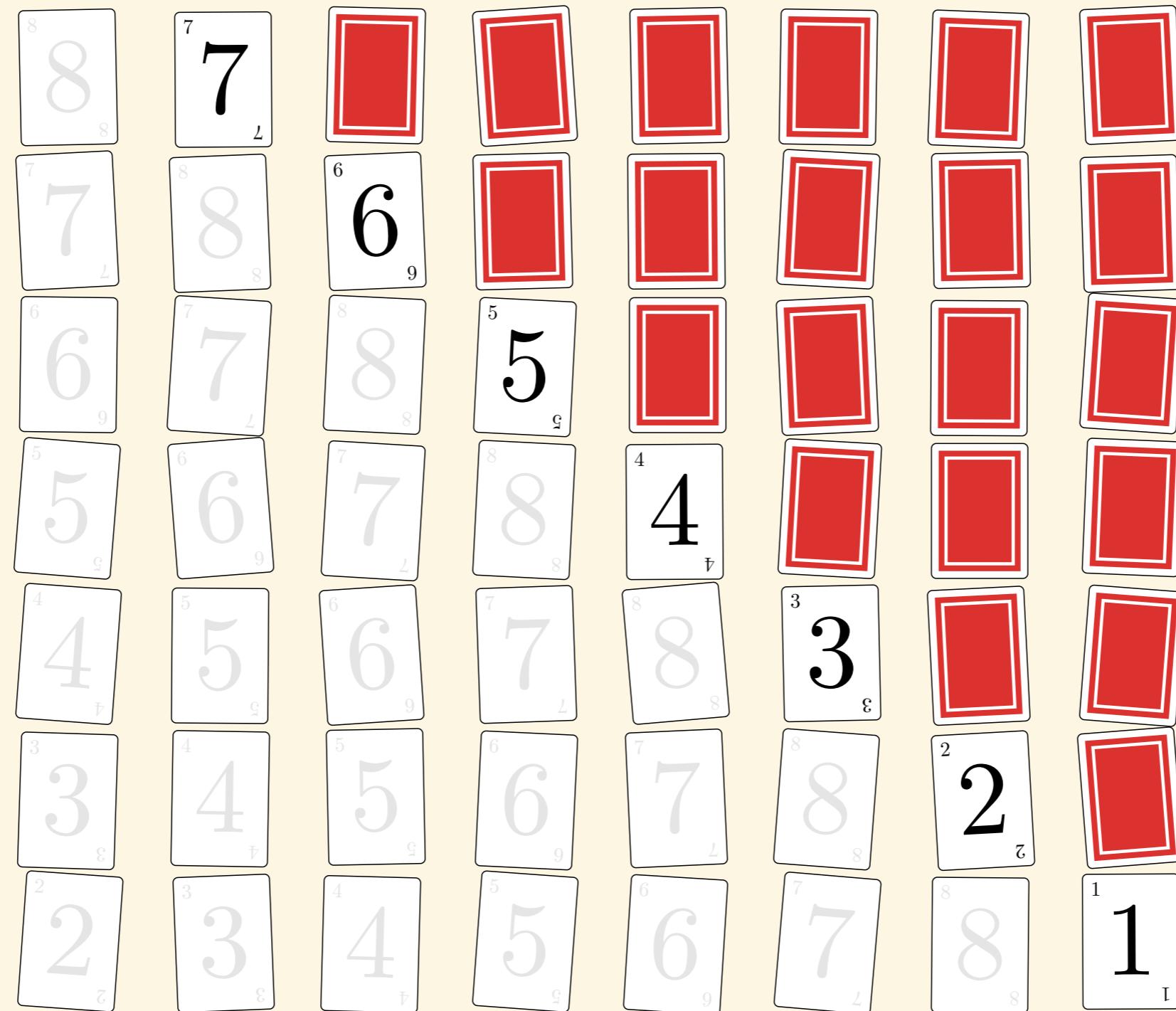
Så, sortering ved innsetting ...



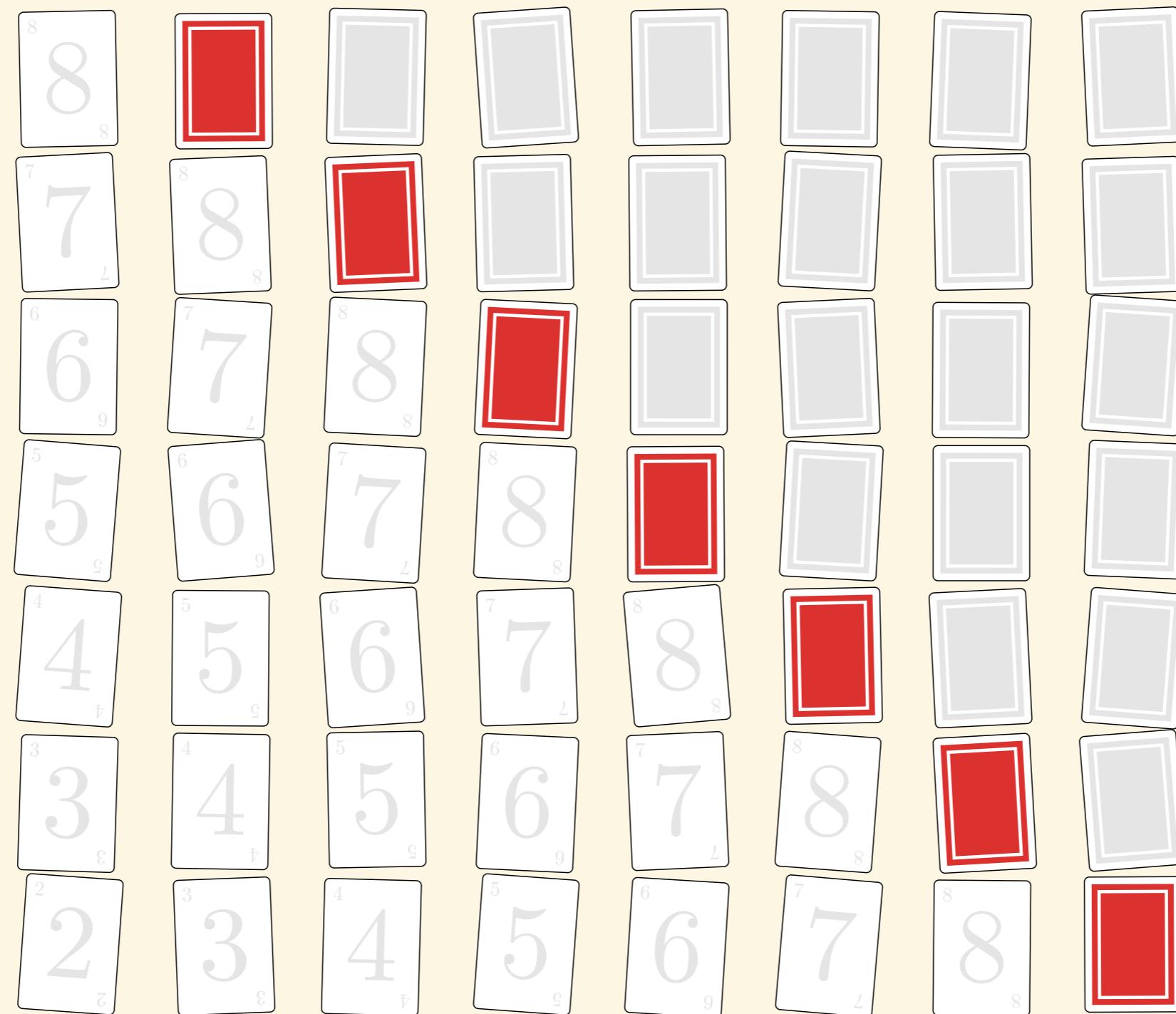
Verste tilfelle: $A = \langle n, \dots, 1 \rangle$. Hver rad er starten på en iterasjon



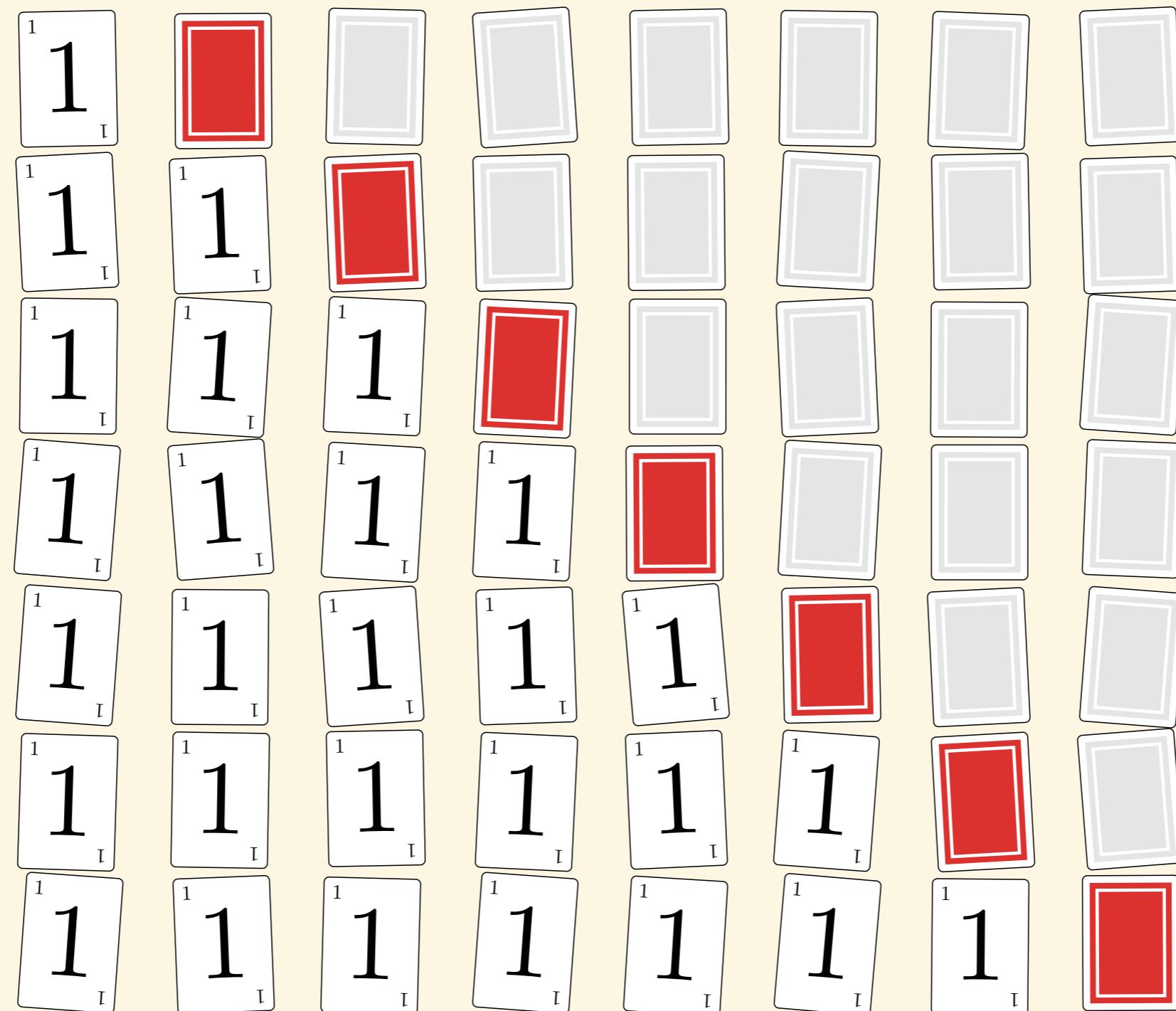
Verste tilfelle: $A = \langle n, \dots, 1 \rangle$. Hver rad er starten på en iterasjon



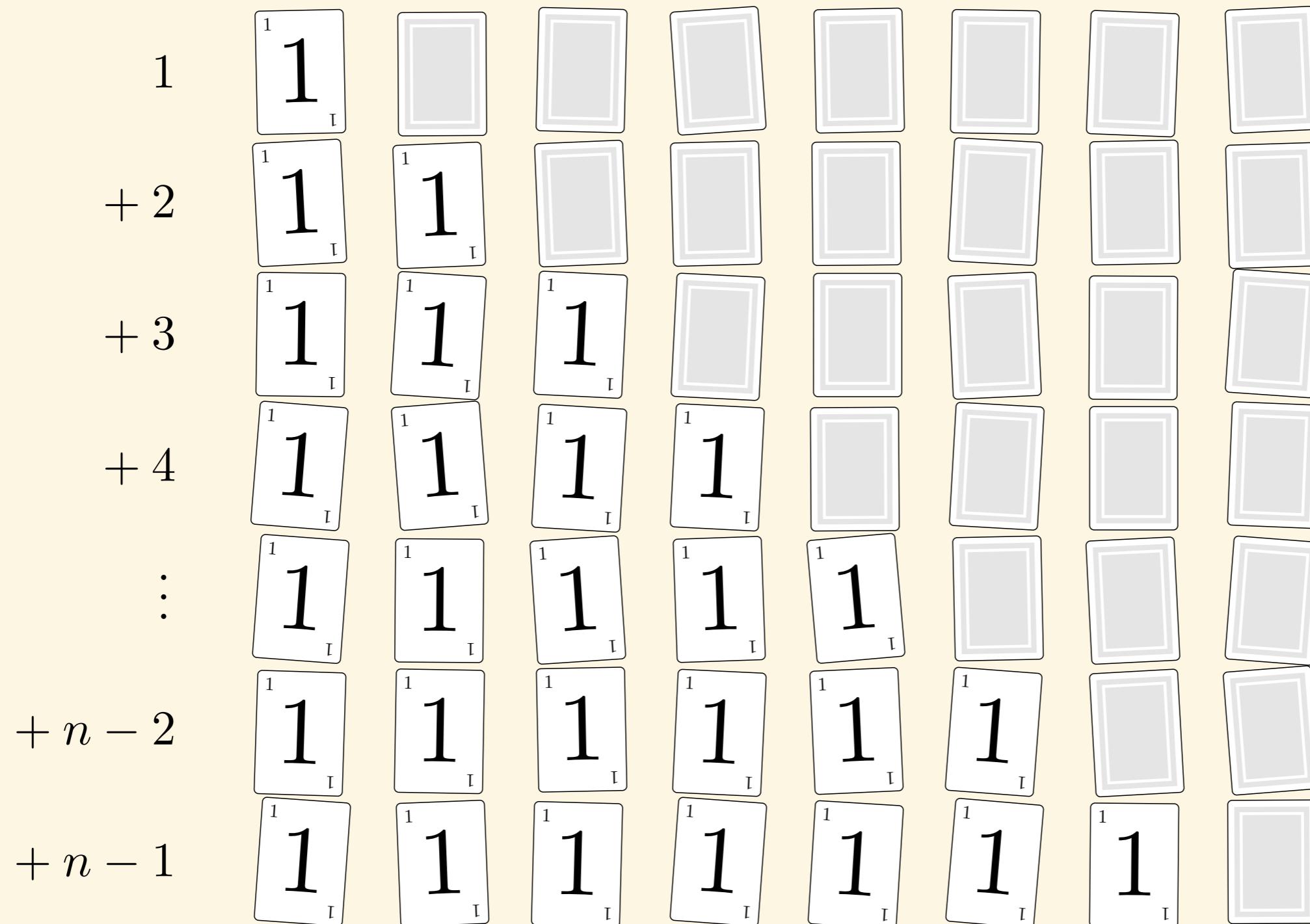
$A[j]$ må flyttes forbi $A[1..j-1]$ andre, $j = 2 \dots n$



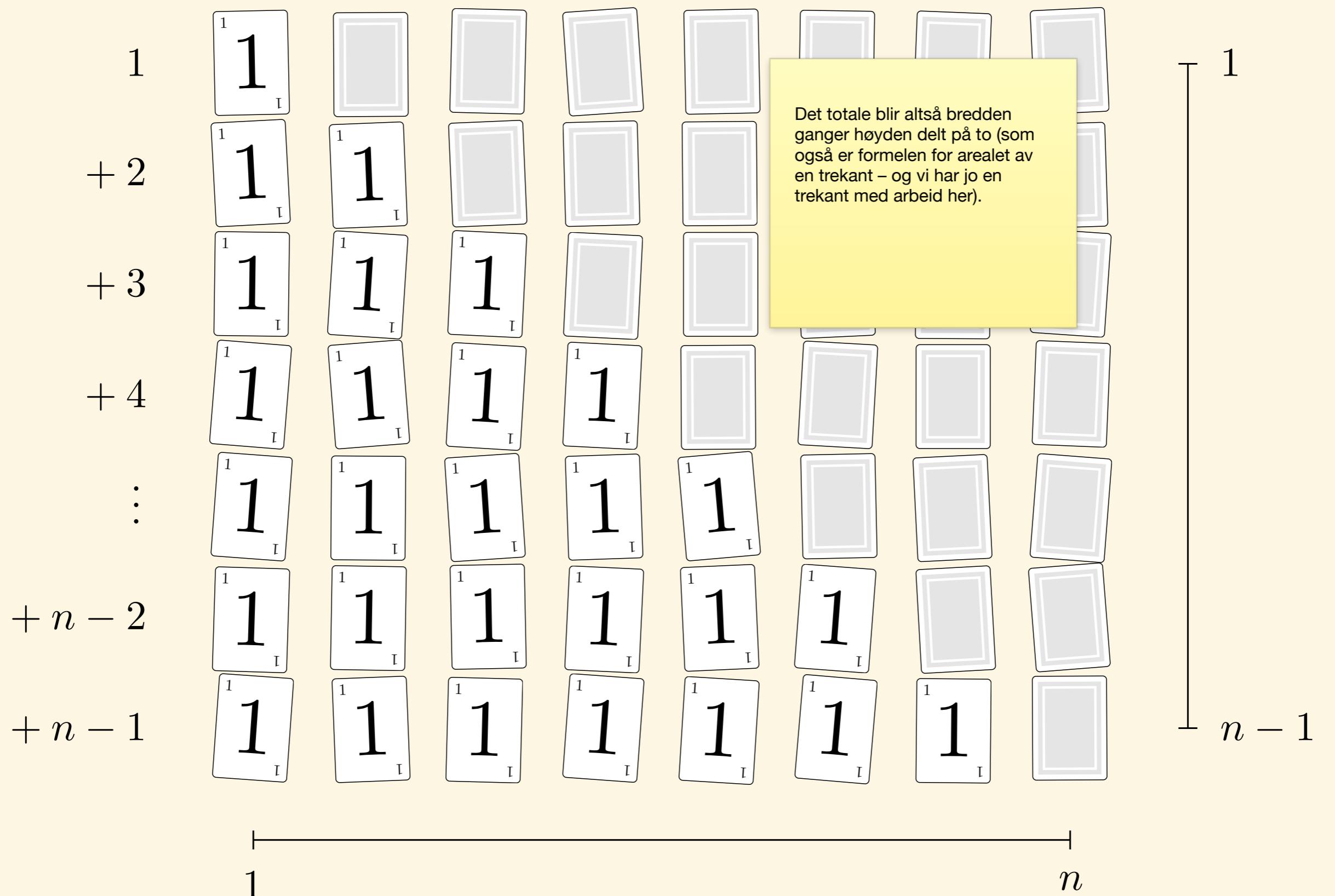
$A[j]$ må flyttes forbi $A[1..j-1]$ andre, $j = 2 \dots n$



Hver flytting er 1 operasjon. Hvor mange totalt?



Hver flytting er 1 operasjon. Hvor mange totalt?



$$\sum_{i=1}^{n-1} i =$$

I iterasjon nr i gjør vi i flytte-operasjoner

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

Halvparten av rektanglet $n \cdot (n - 1)$, «på skrå»

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

$\frac{1}{2}n^2 - \frac{1}{2}n$ uten konstantfaktorer og lavere-ordens ledd er n^2

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

INSERTION-SORT har altså kvadratisk kjøretid



Om det ikke var klart: Vi har gått fra noe kosmisk dårlig til noe ... sånn passe dårlig. Som jo er en enorm forbedring! (Vi skal se på bedre sorteringsalgoritmer senere.)

Noe å tenke på:
Vi har så langt antatt worst-case.
Hva blir best-case for disse to algoritmene?
Hva blir average-case?

Men ... hvordan kom vi frem til denne algoritmen? Kan vi trekke noen lærdom her som vi kan bruke på vanskeligere problemer?

3:5

Dekomponering



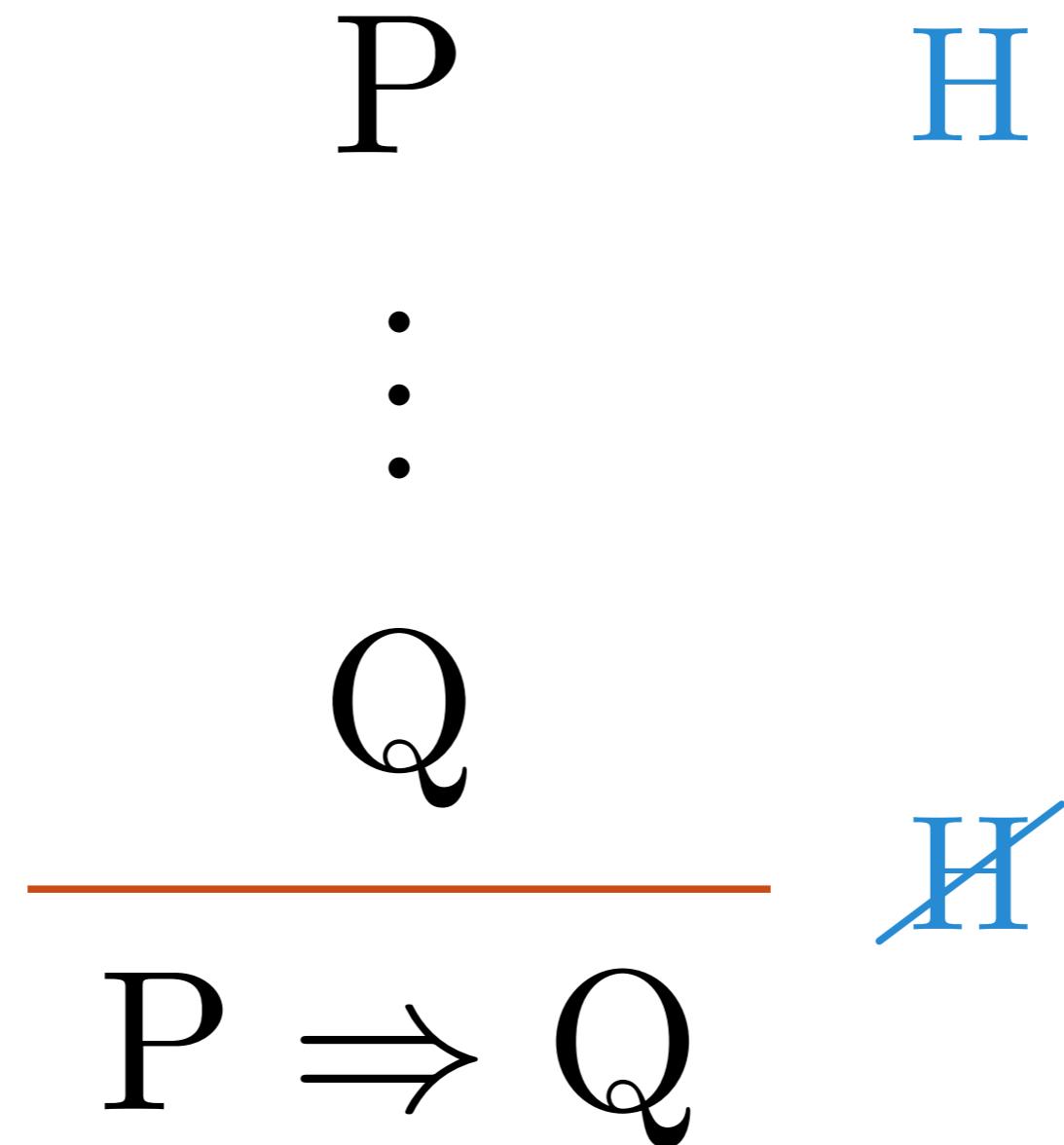
Induksjon

$$\frac{P(a)}{\forall x P(x)}$$

\forall -introduksjon: Vis P for vilkårlig a

$$\frac{\text{vilkårlig} \quad \downarrow}{\overline{P(a)}} \quad \forall x P(x)}$$

\forall -introduksjon: Vis P for vilkårlig a



\Rightarrow -introduksjon: Midlertidig anta P og vis deretter Q

$$P \Rightarrow Q, P$$

$$Q$$

\Rightarrow -eliminasjon: Modus ponens

$$\frac{\text{vilkårlig}}{\forall x P(x)}$$
$$\frac{P \quad \vdots \quad Q}{P \Rightarrow Q} H$$
$$\frac{P \Rightarrow Q, \quad P}{Q}$$

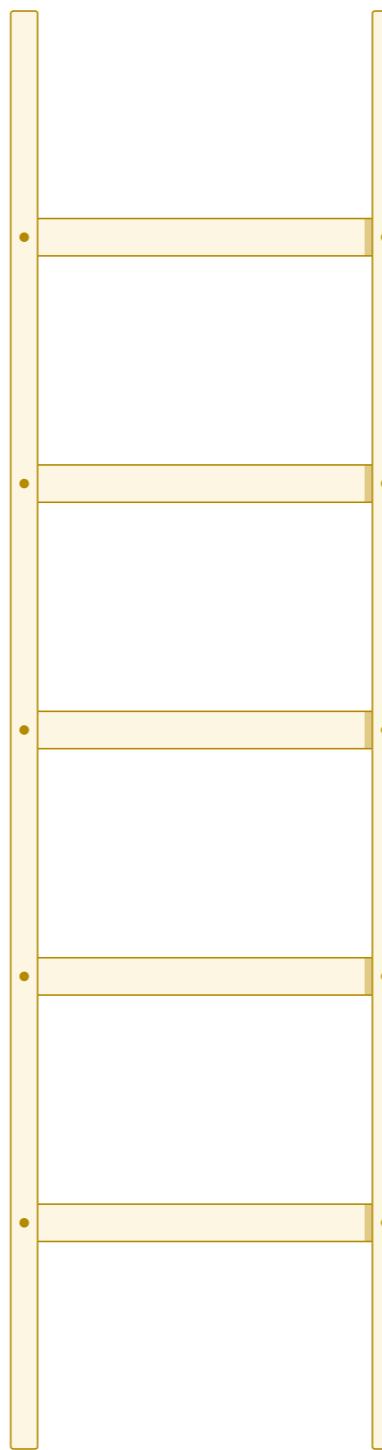
Induksjon kombinerer disse

$$\frac{\frac{\frac{P(a)}{\forall x P(x)}}{\text{vilkårlig}} \quad \frac{\frac{P \quad H}{\vdots} \quad \frac{Q}{P \Rightarrow Q}}{\cancel{H}}}{P \Rightarrow Q, \quad P} \quad \frac{}{Q}$$

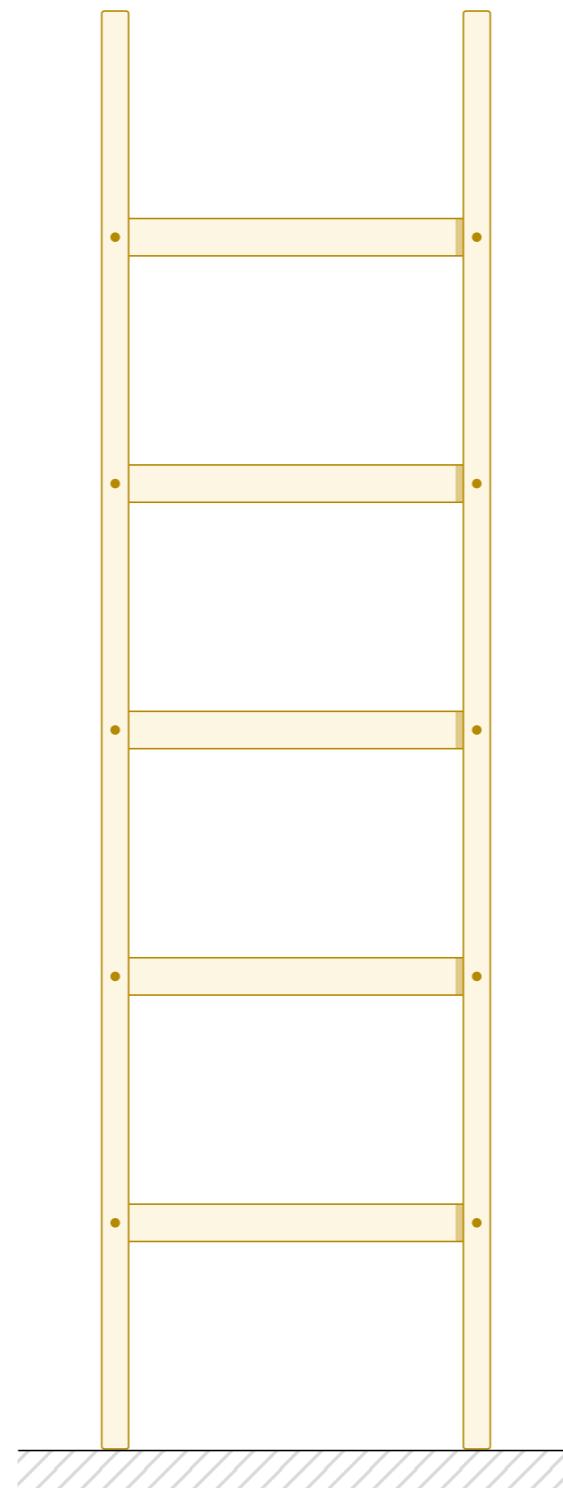
Vi introduserer og eliminerer mange implikasjoner

Generell induksjon kan foregå i et nettverk av utsagn . . .

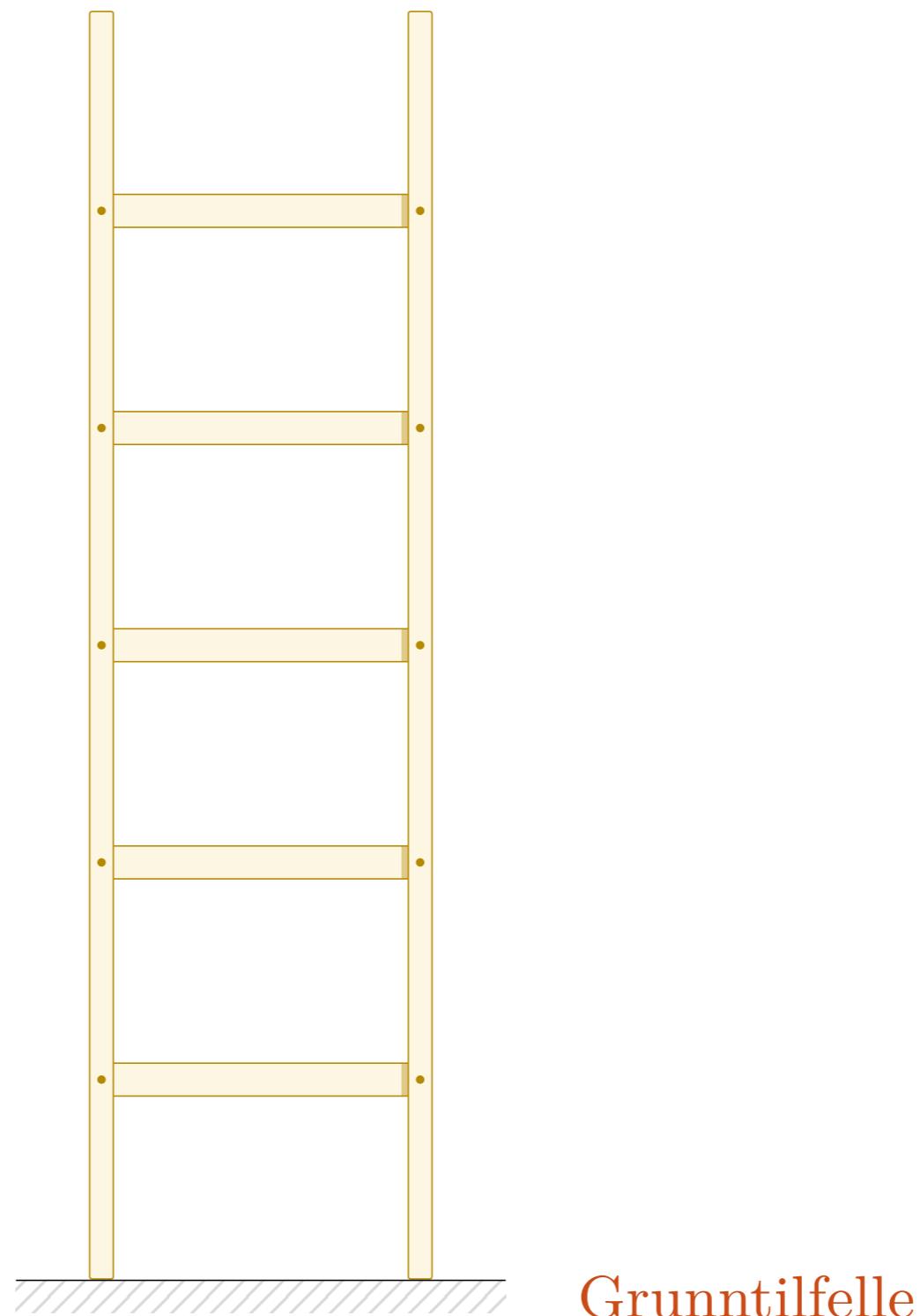
... men vi kan alltid ordne dem i en serie med trinn



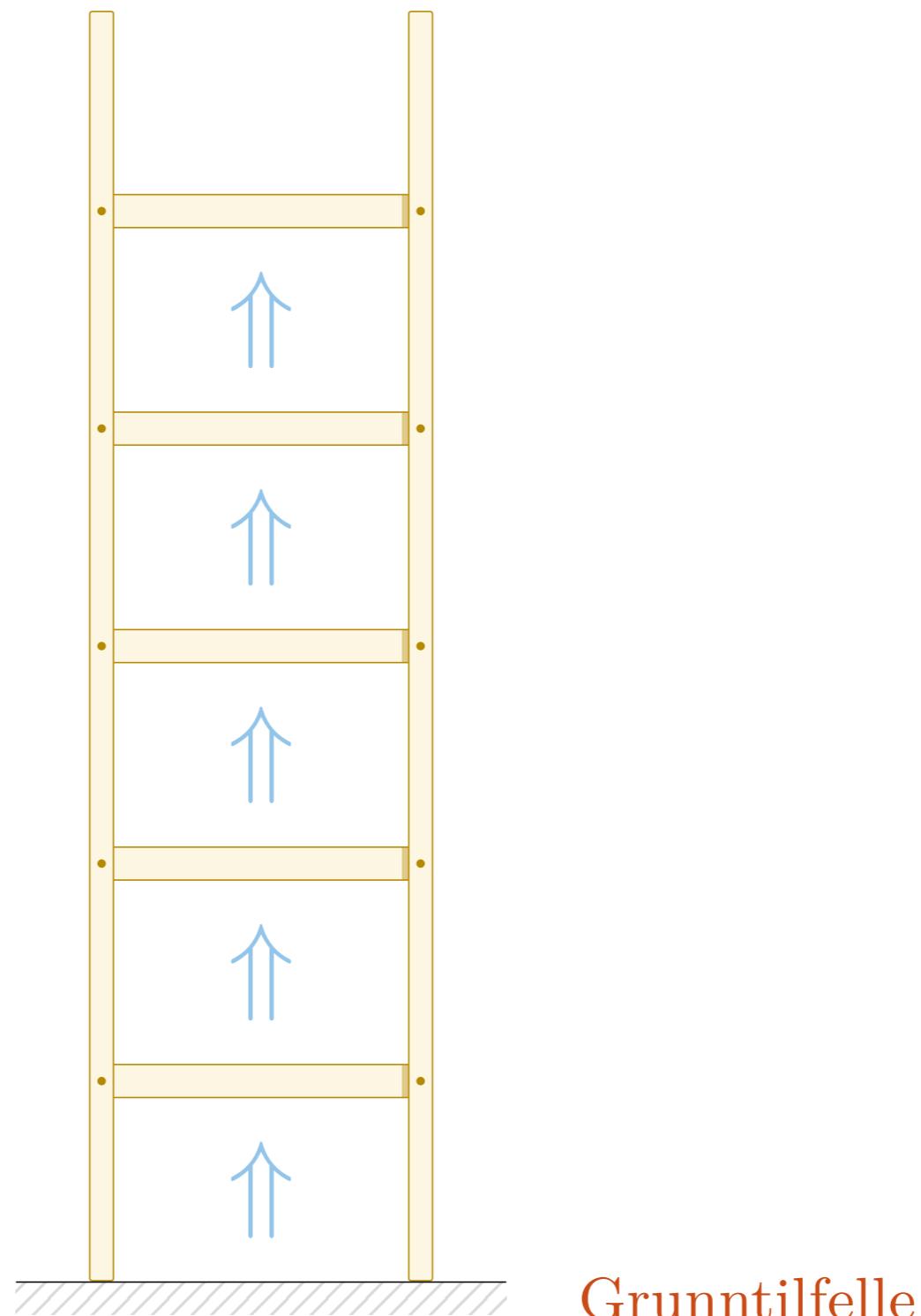
... men vi kan alltid ordne dem i en serie med trinn



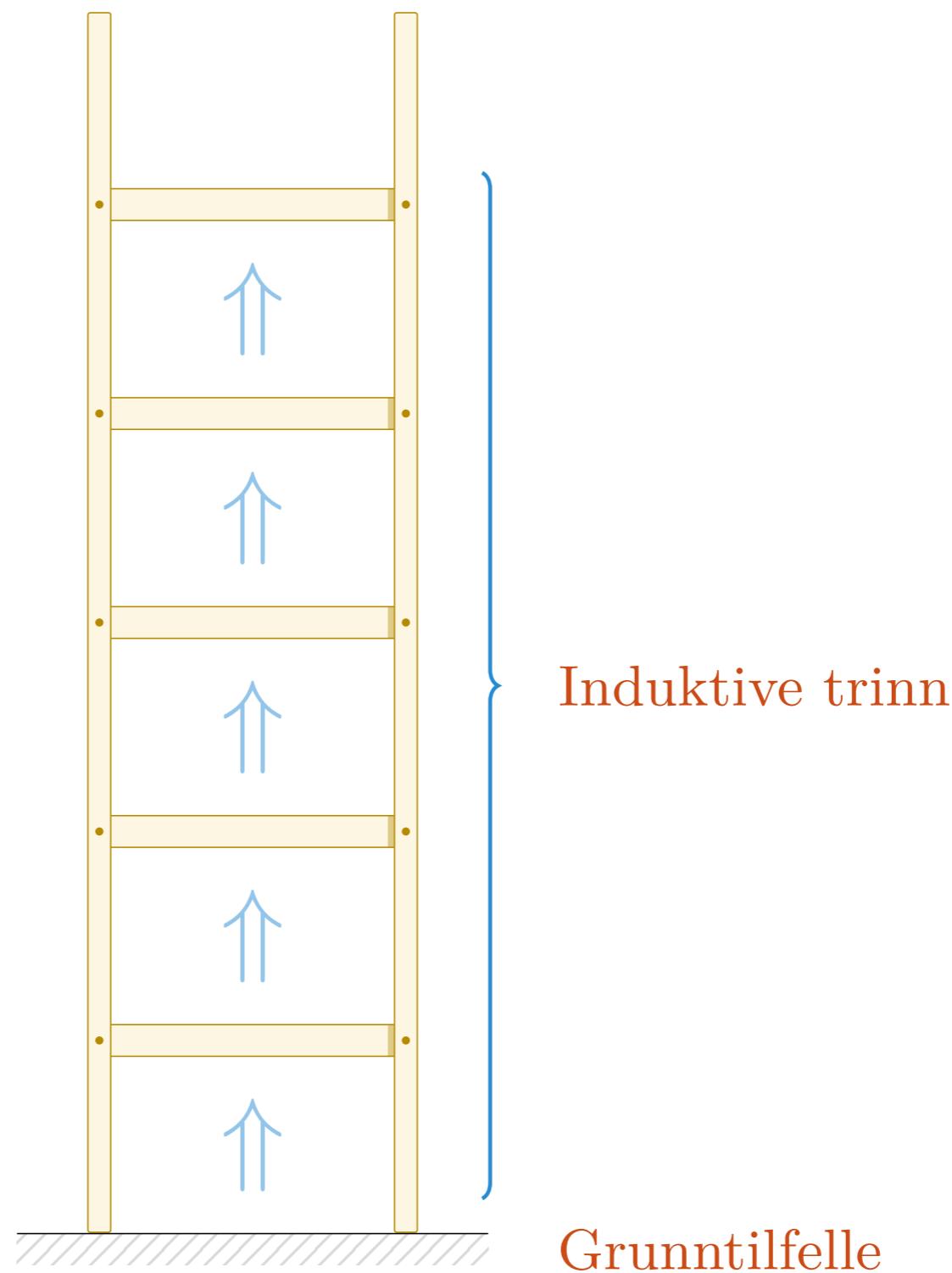
Ett eller flere tilfeller baserer seg ikke på noen andre



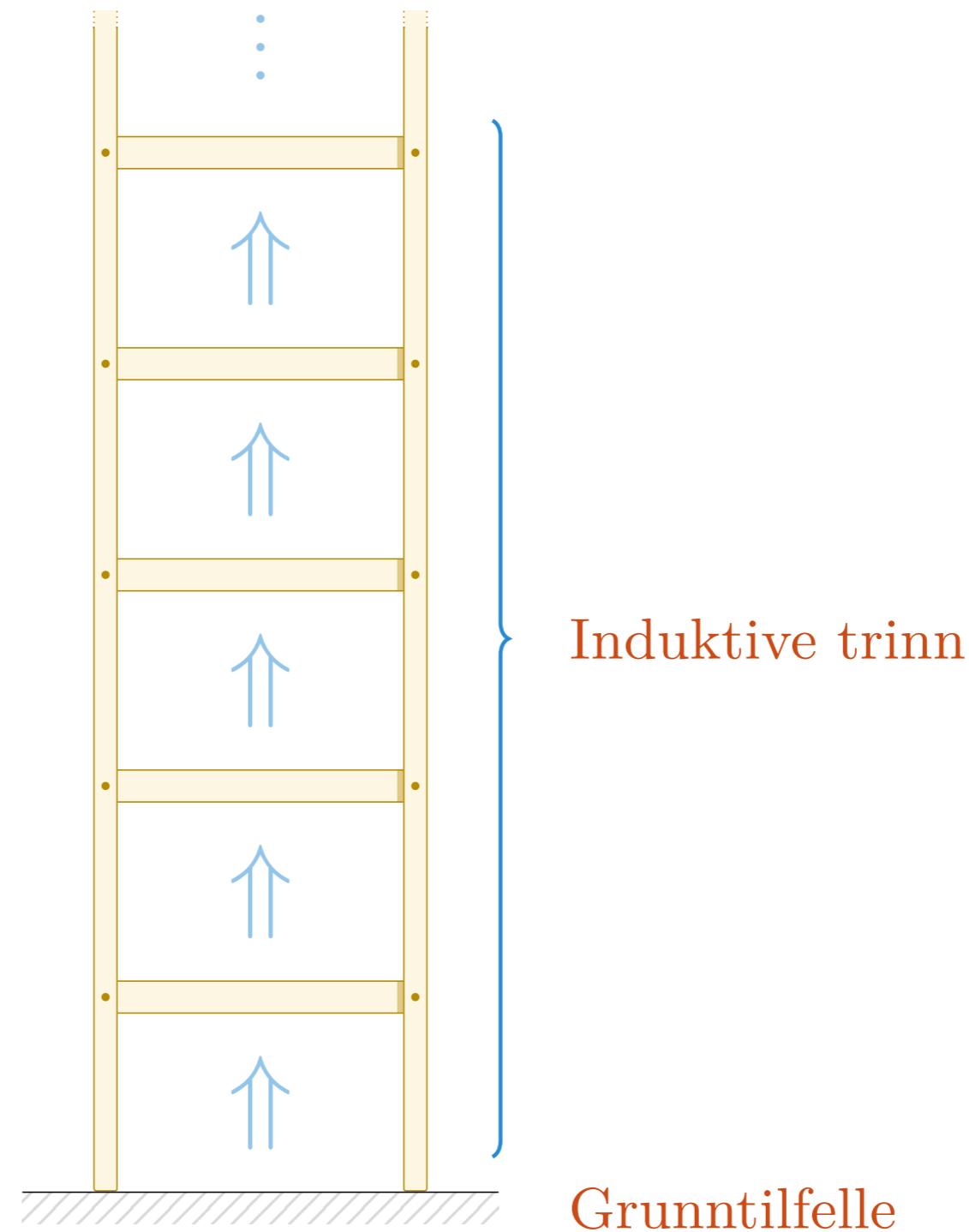
Ett eller flere tilfeller baserer seg ikke på noen andre

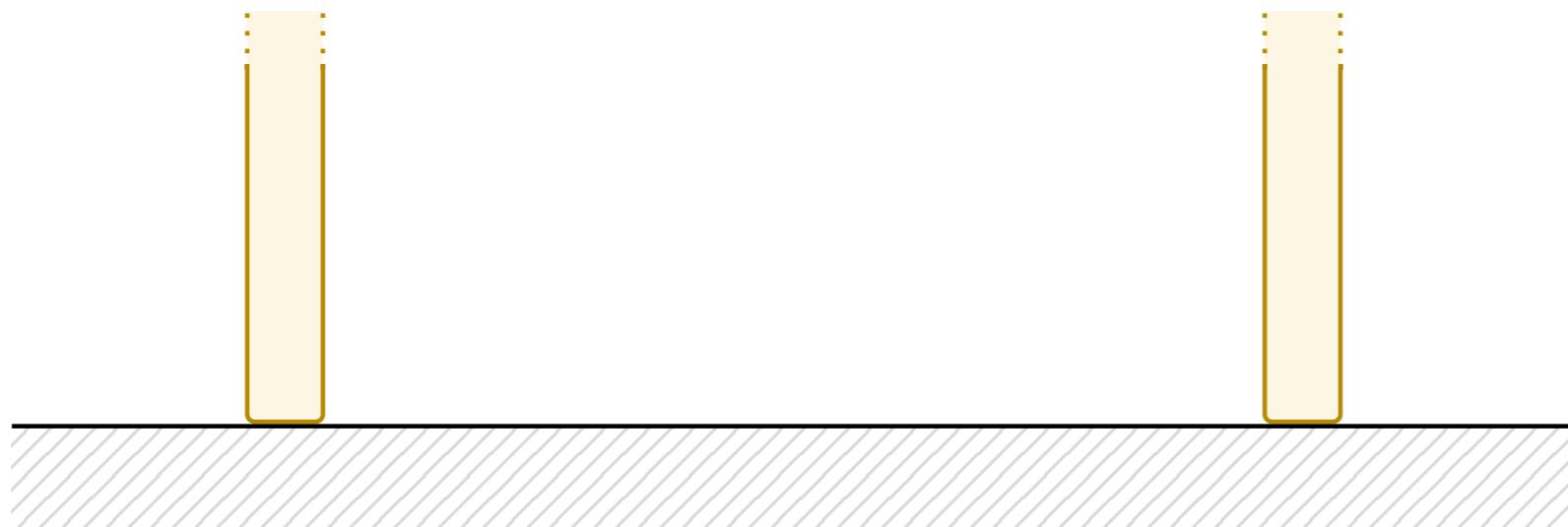


De andre er induktive: De følger av tidligere trinn

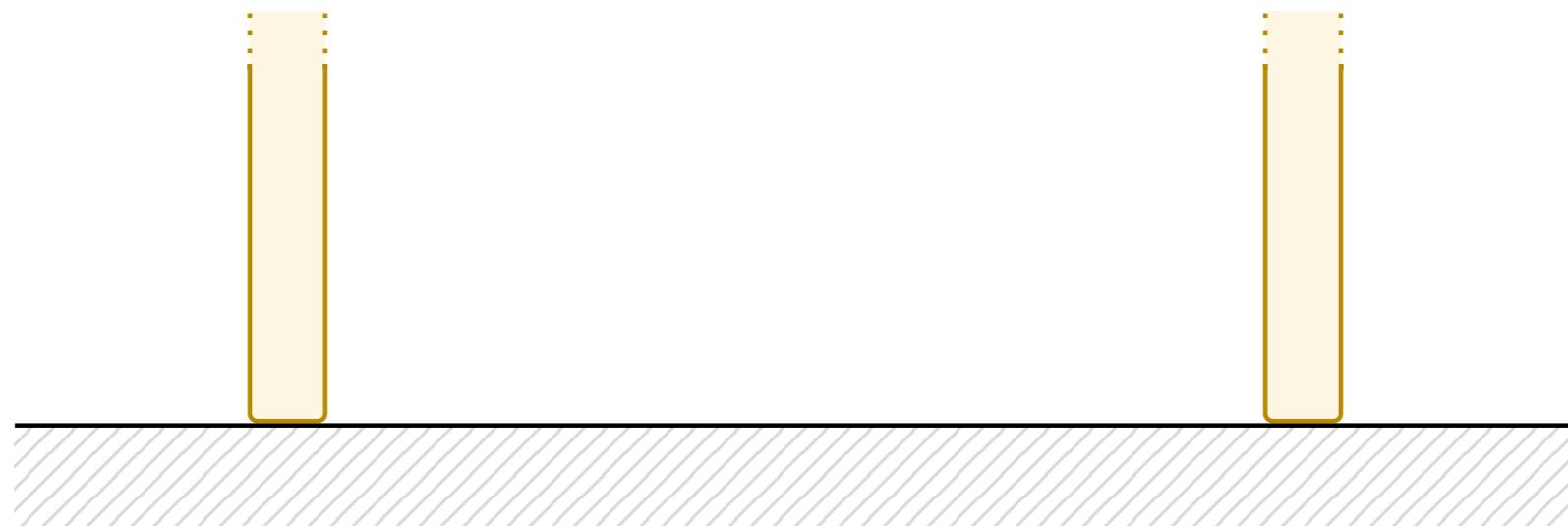


De andre er induktive: De følger av tidligere trinn

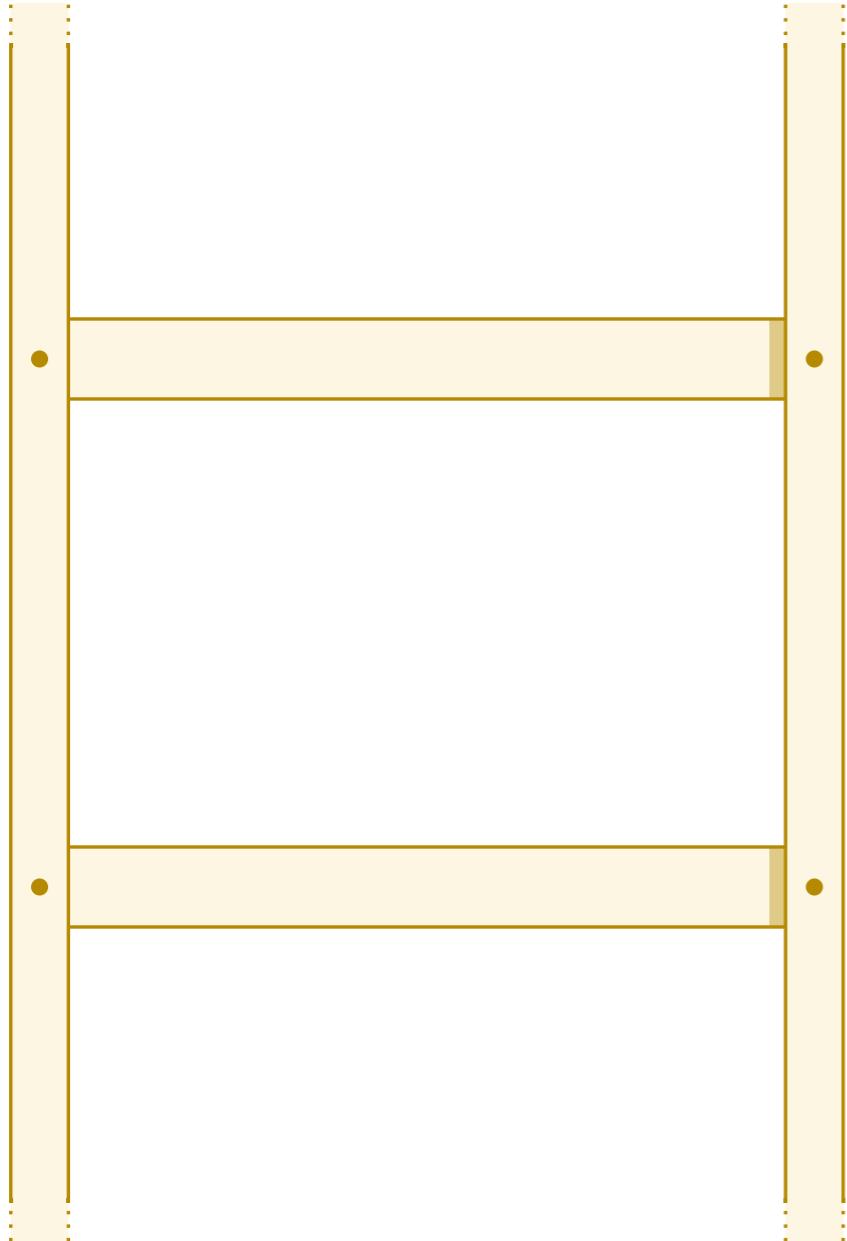




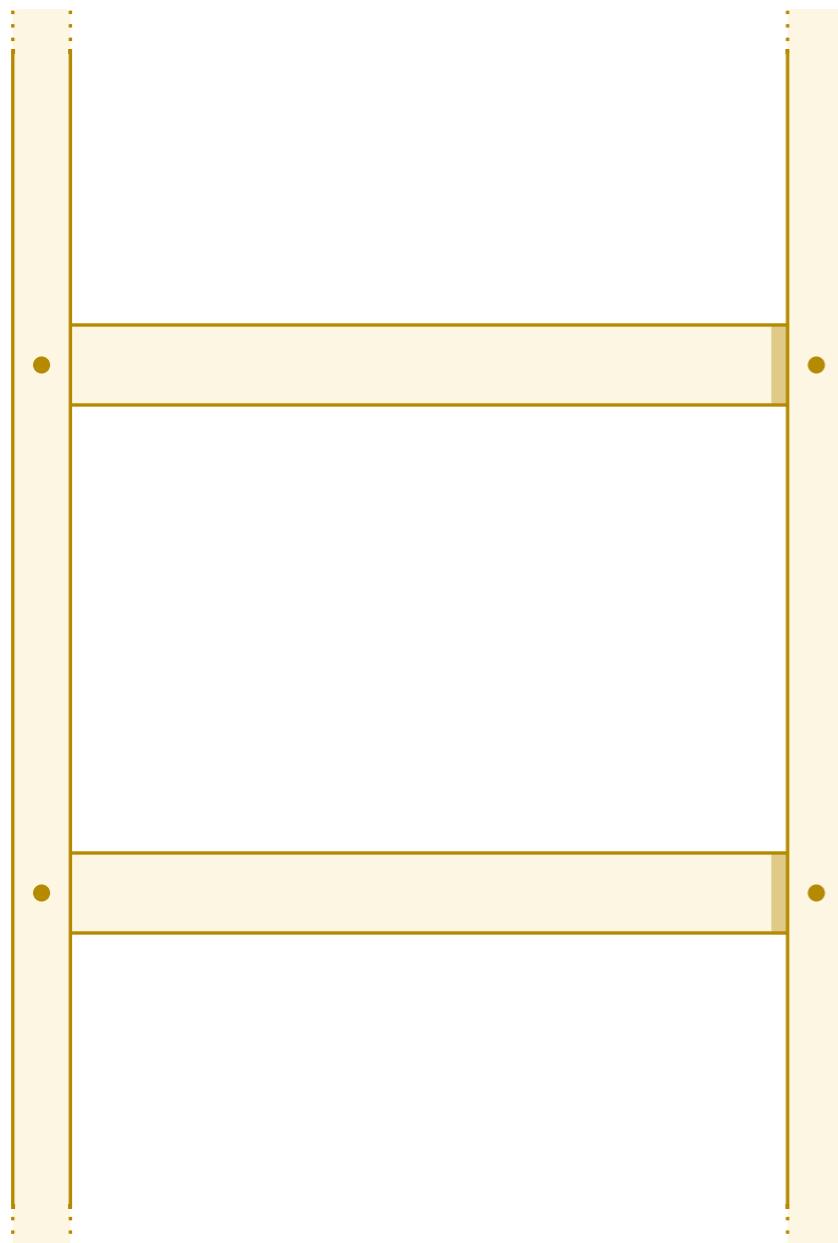
Grunntilfellet (evt. grunntilfellene) viser vi for seg



(Det er ofte svært enkelt)



Vil beskrive alle induktive trinn

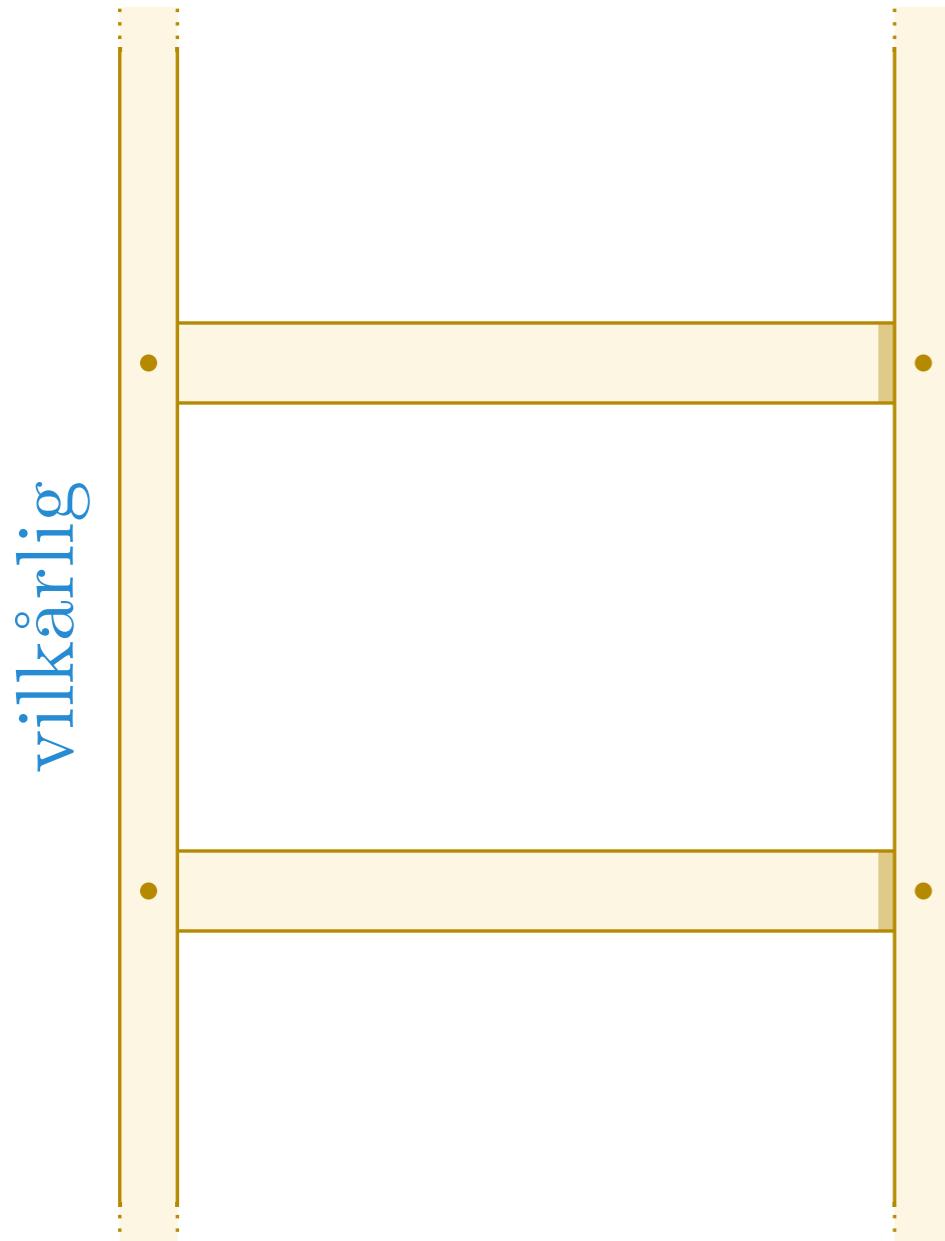


Vil beskrive alle induktive trinn

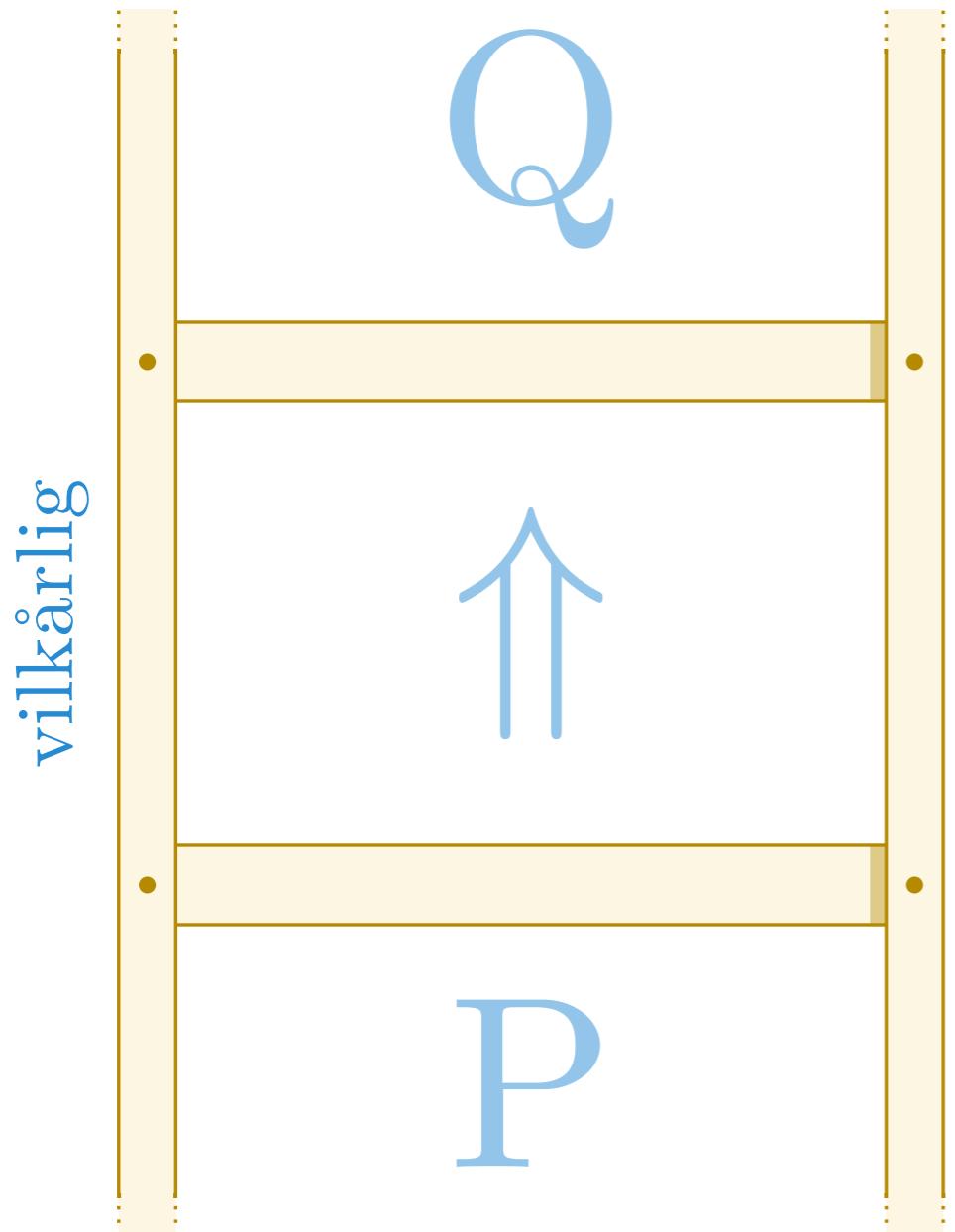
Så vi ser på et vilkårlig et

$$\frac{\text{vilkårlig}}{P(a)}$$

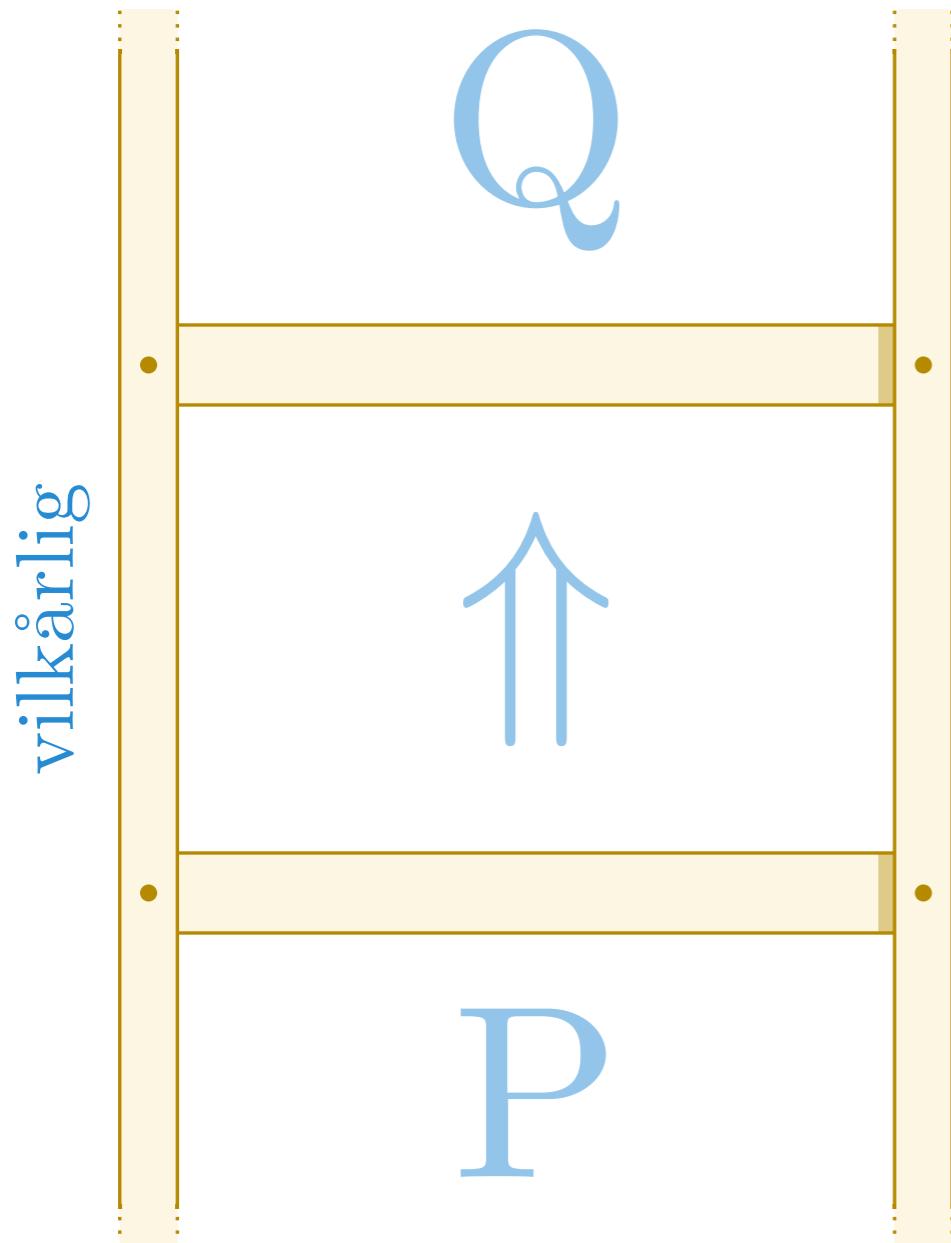
$$\forall x P(x)$$



$$\frac{\text{vilkårlig}}{\forall x P(x)}$$



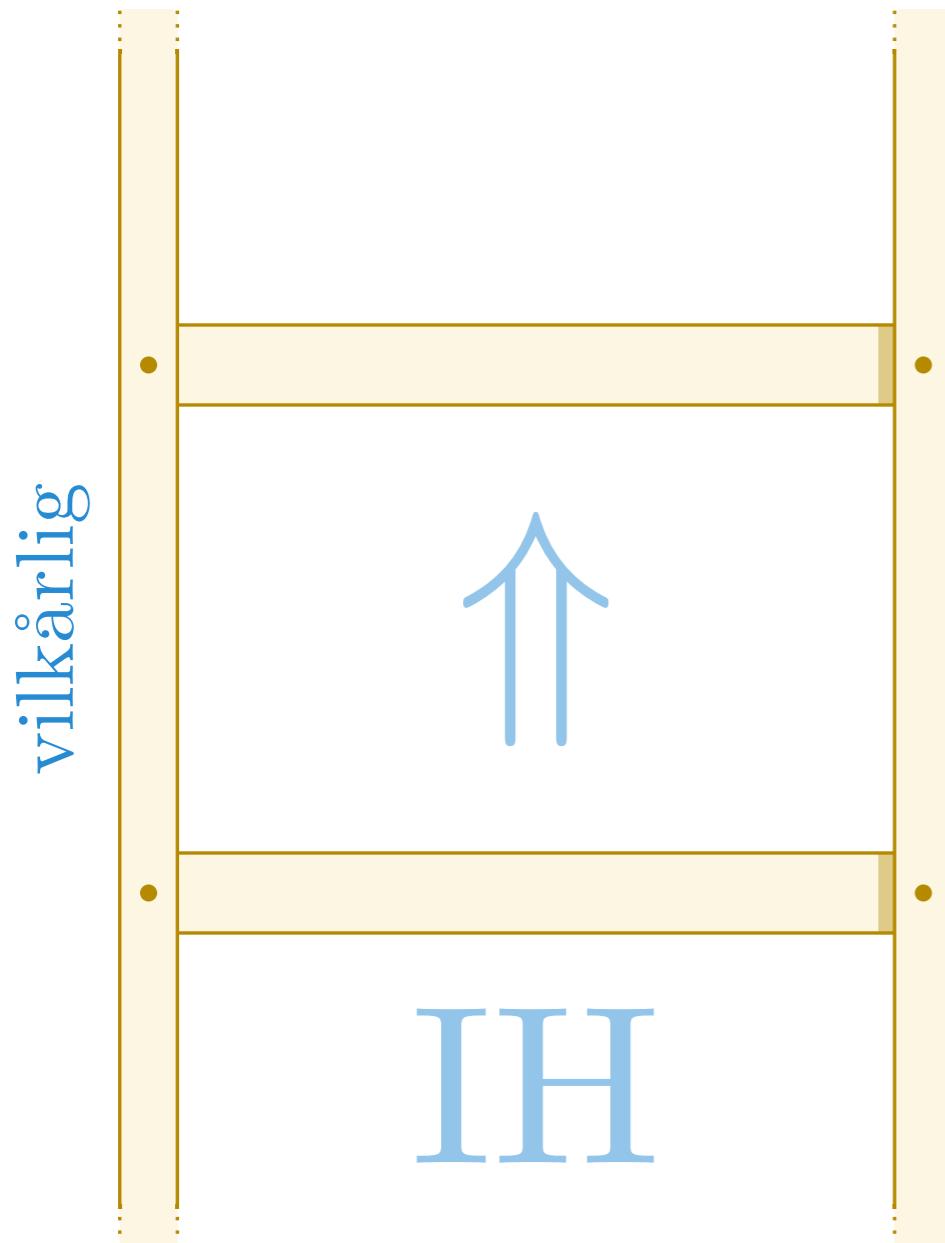
Vil vise implikasjon



Vil vise implikasjon

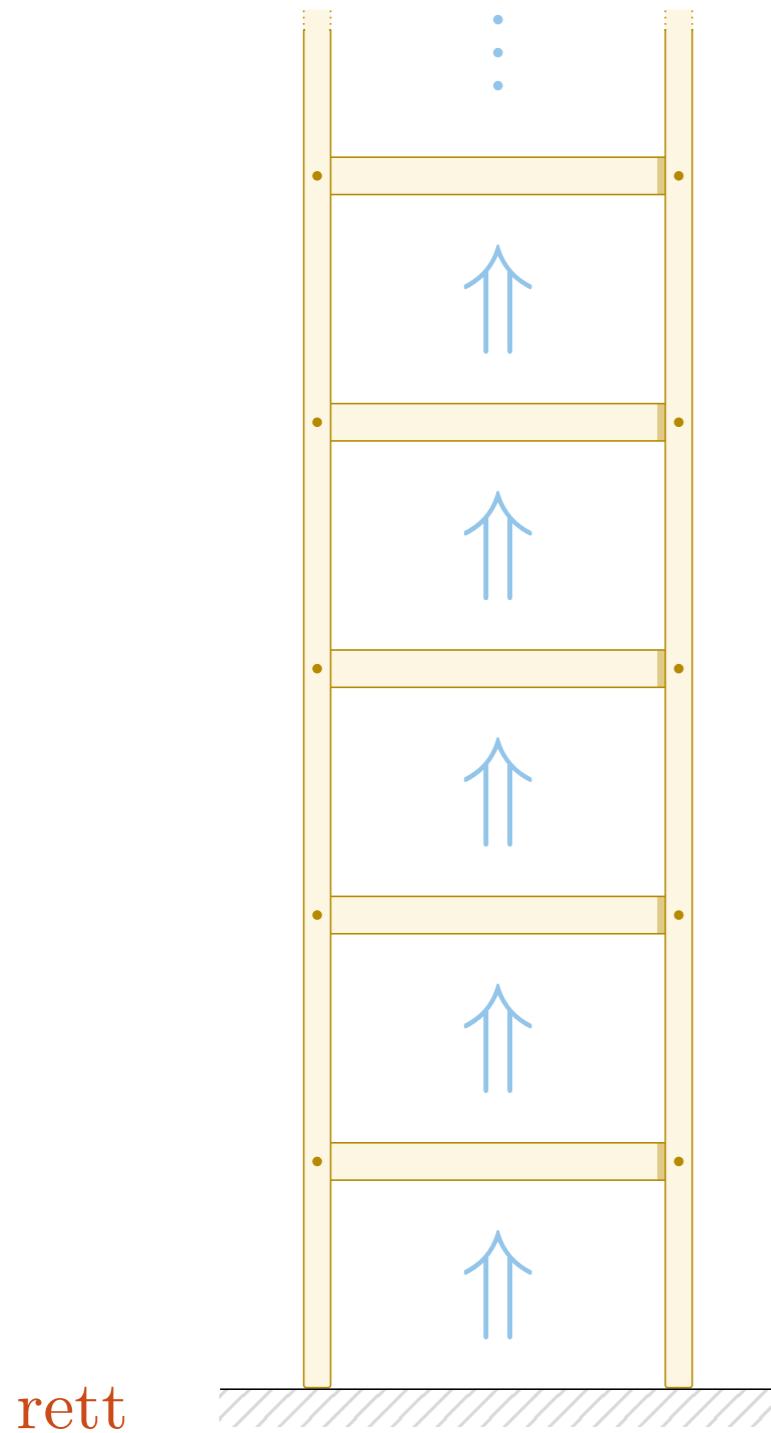
$$\frac{P \Rightarrow Q}{\text{H}}$$

Anta forrige trinn

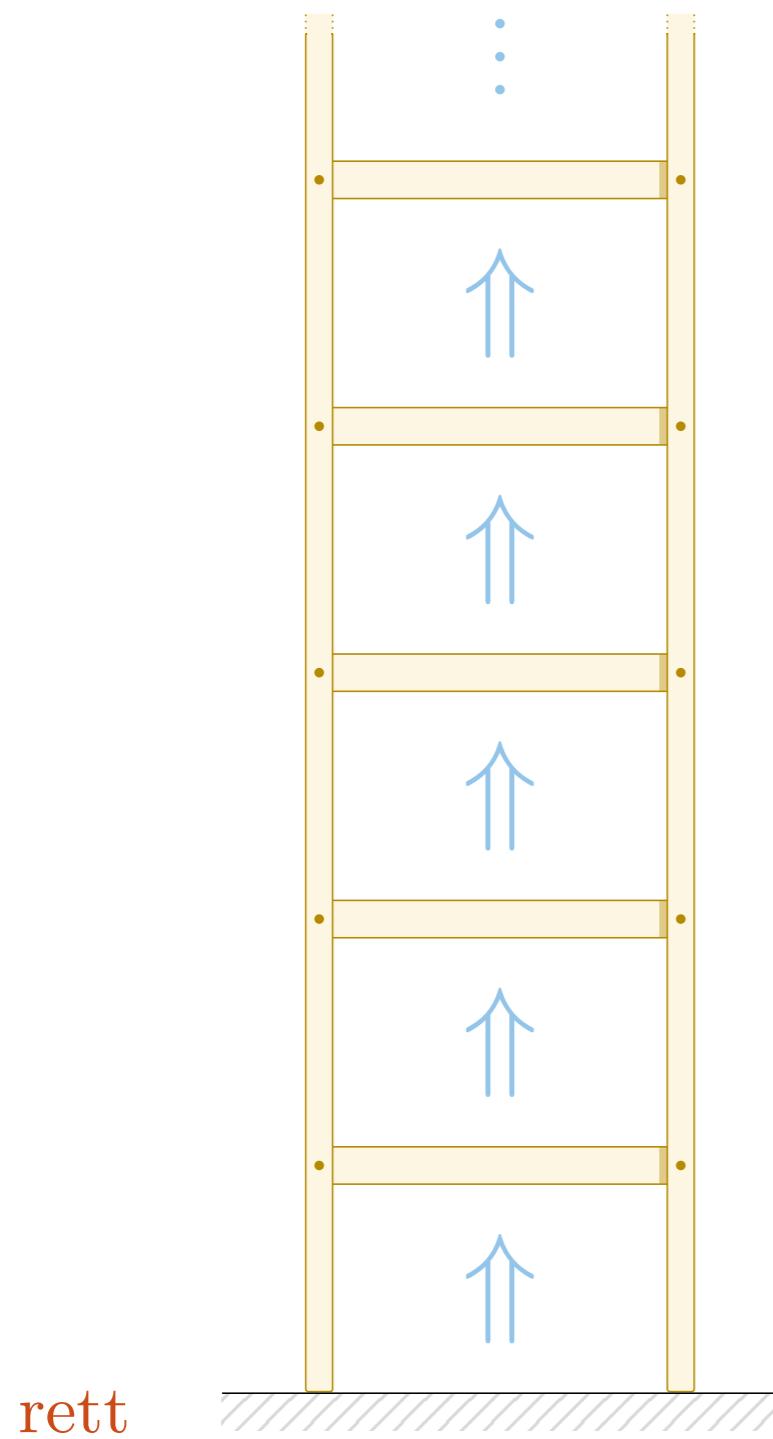


$$\frac{P \Rightarrow Q}{\begin{array}{c} P \\ \vdots \\ Q \\ \hline H \end{array}}$$

P = Induksjonshypotesen



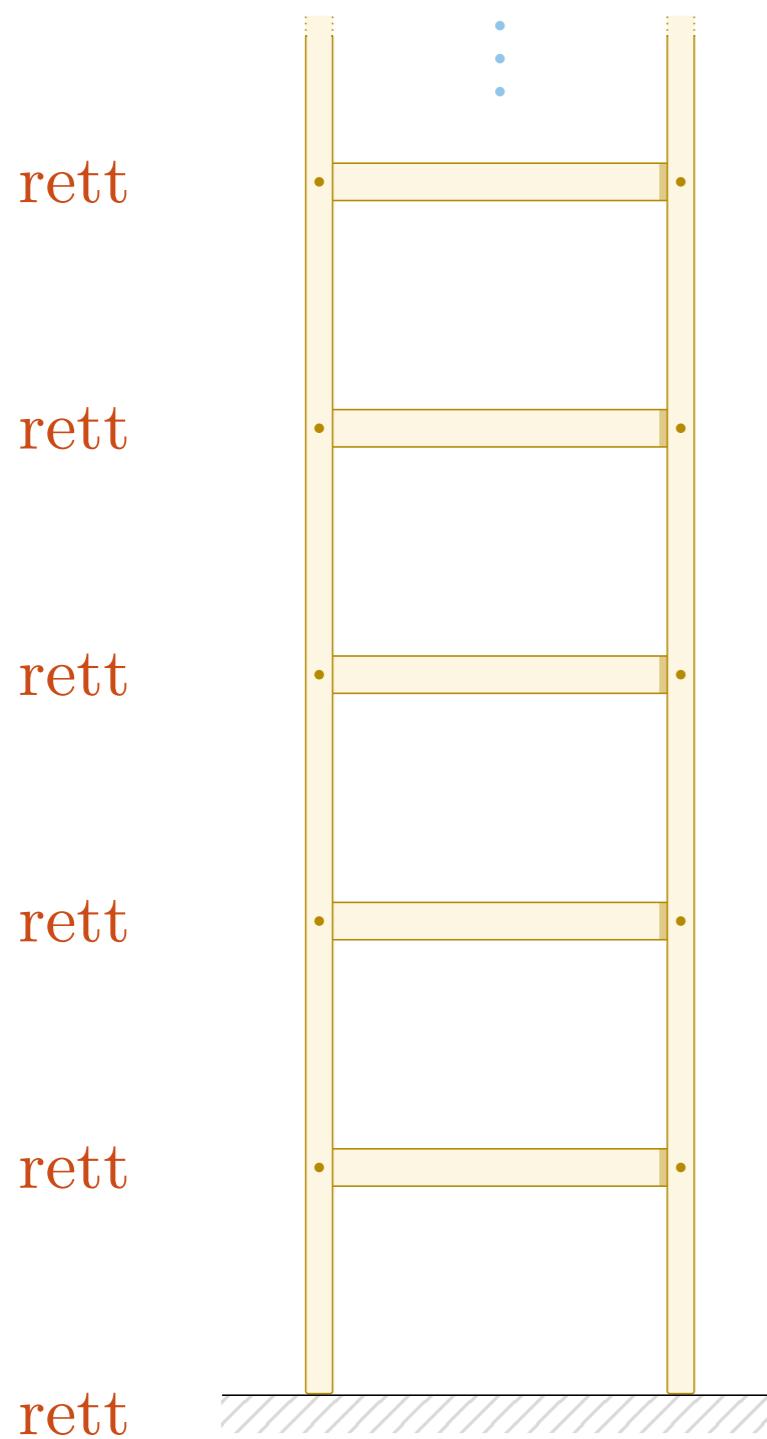
Vi vet at grunntilfellet stemmer



Vi vet at grunntilfellet stemmer

$$\frac{P \Rightarrow Q, P}{Q}$$

Dette «smitter» til alle de andre

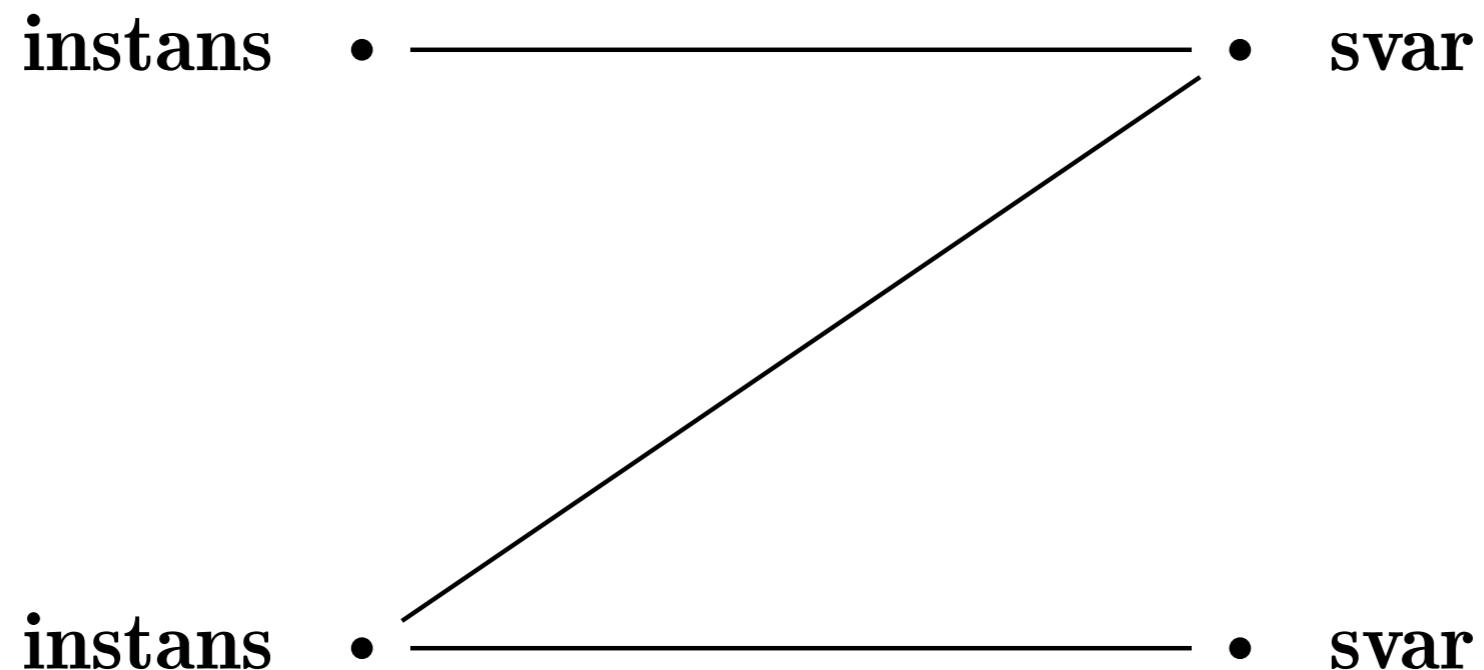


Vi vet at grunntilfellet stemmer

Dette «smitter» til alle de andre

$$\frac{P \Rightarrow Q, P}{Q}$$

Problem



Et problem er en relasjon mellom instanser og riktige svar

Problem

instans • —————→ • **svar**

instans • —————→ • **svar**

En algoritme finner ett riktig svar for hver instans

Problem

instans • —————→ • **svar**

instans • —————→ • **svar**

Algoritmen må løse alle instanser. Vi ser på en vilkårlig instans

Problem

instans • ----- → • **svar**

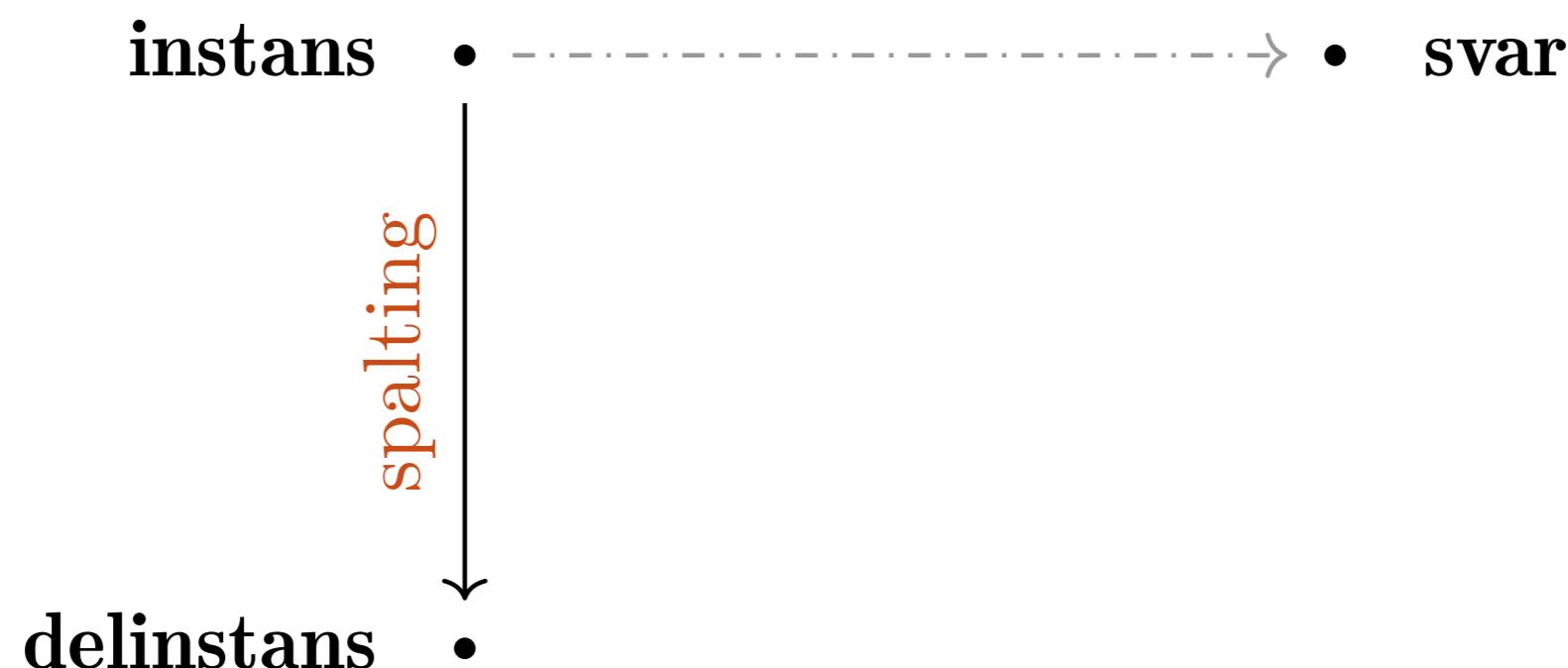
Vi vet ikke ennå hvordan vi løser instansen

Problem

instans • ----- → • **svar**

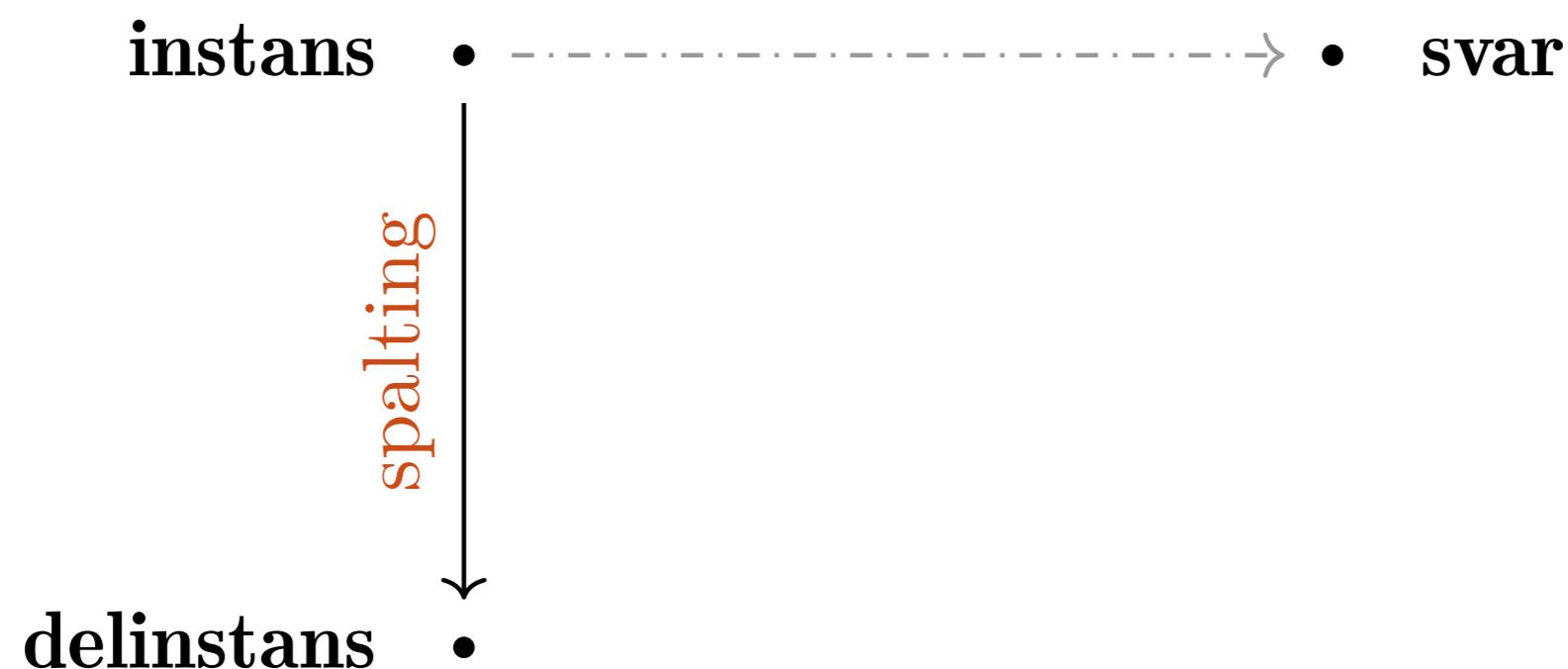
Vi spalter instansen i én eller flere delinstanser

Problem



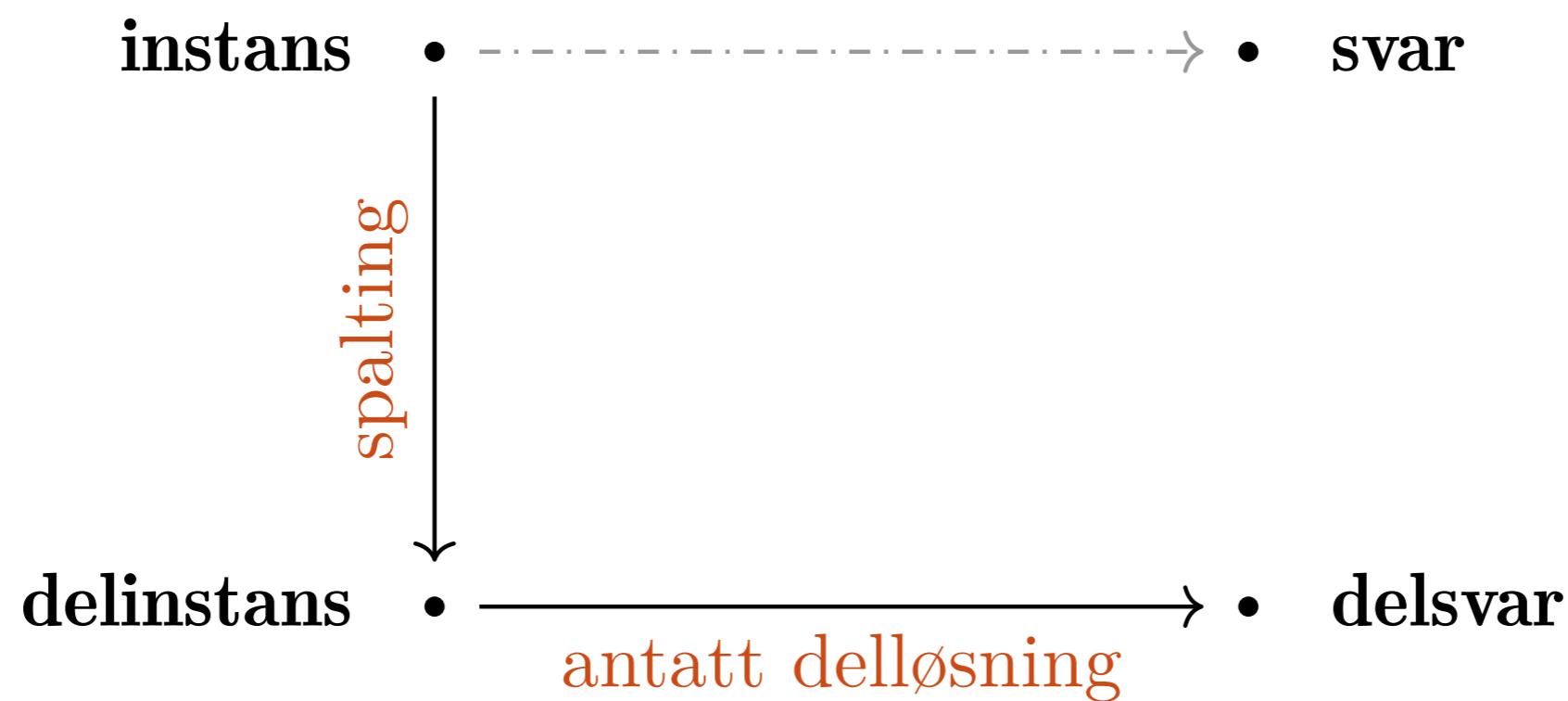
Vi spalter instansen i én eller flere delinstanser

Problem



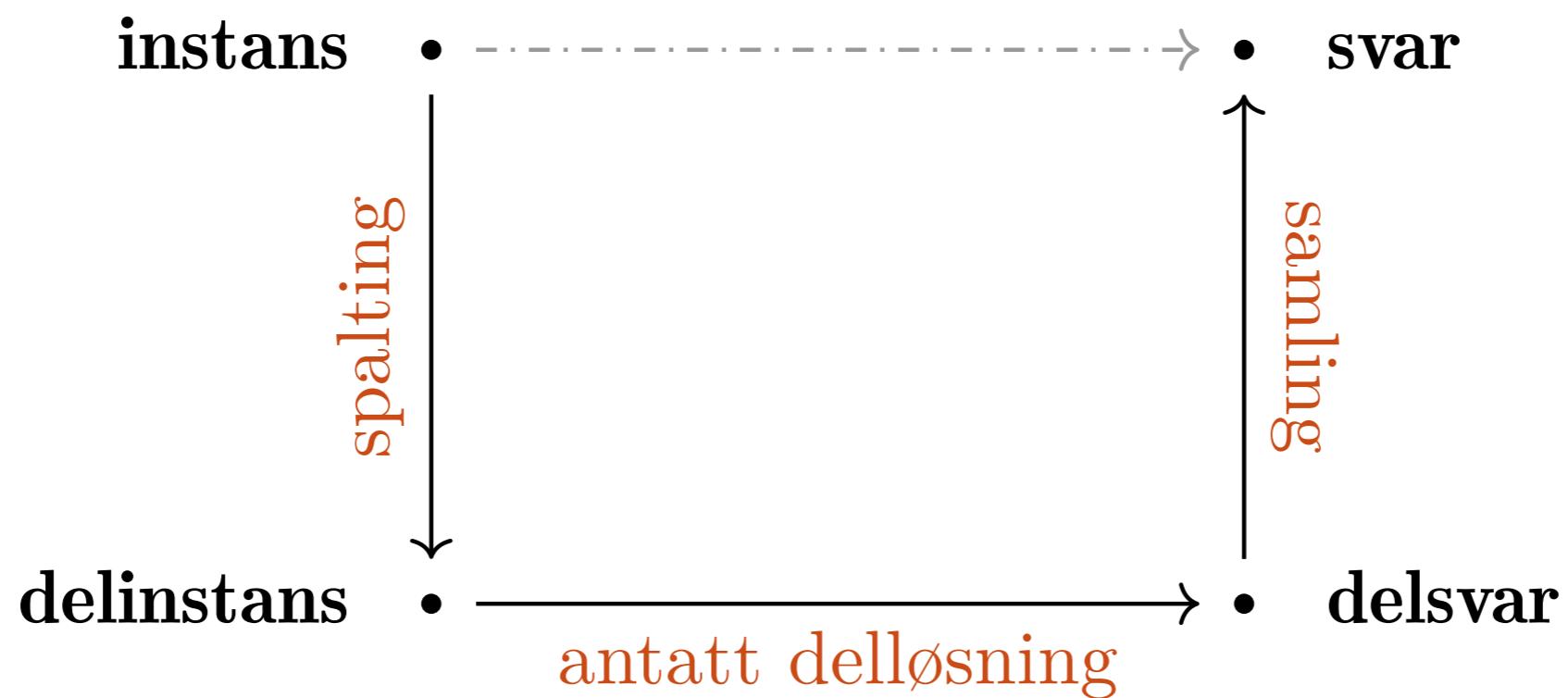
Vi antar at vi kan løse delinstansene

Problem



Vi antar at vi kan løse delinstansene

Problem



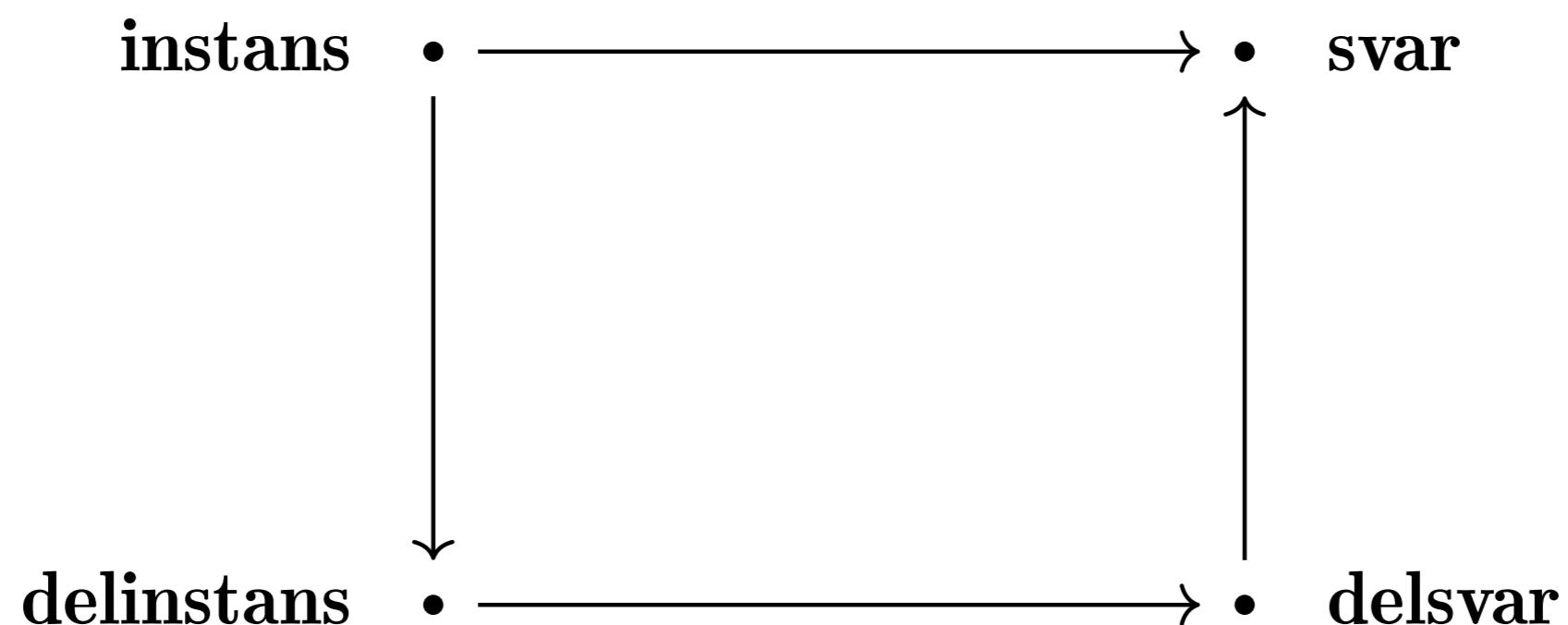
Vi samler så delsvar til et endelig svar

Problem



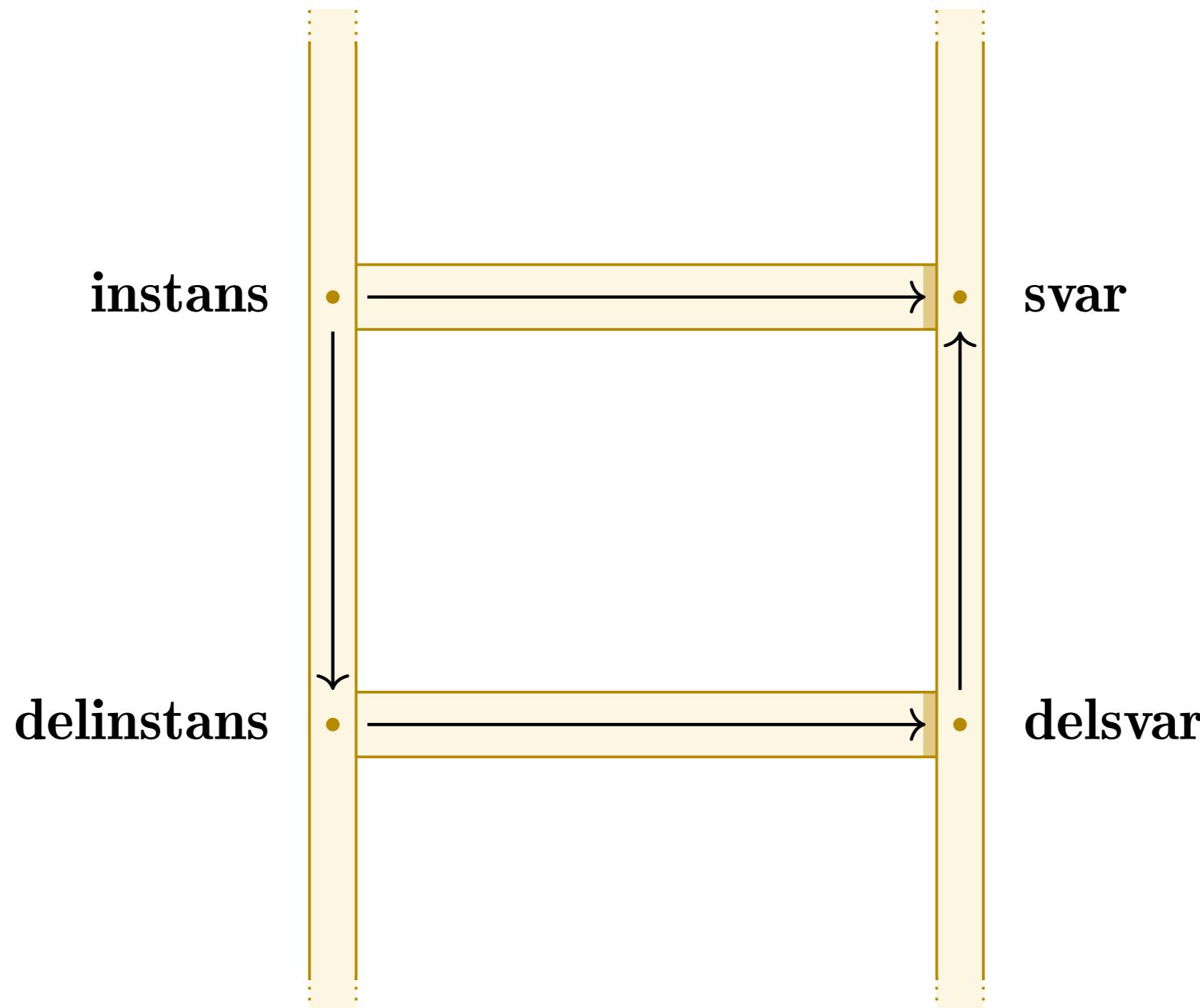
Om vi setter sammen disse får vi en løsning!

Problem



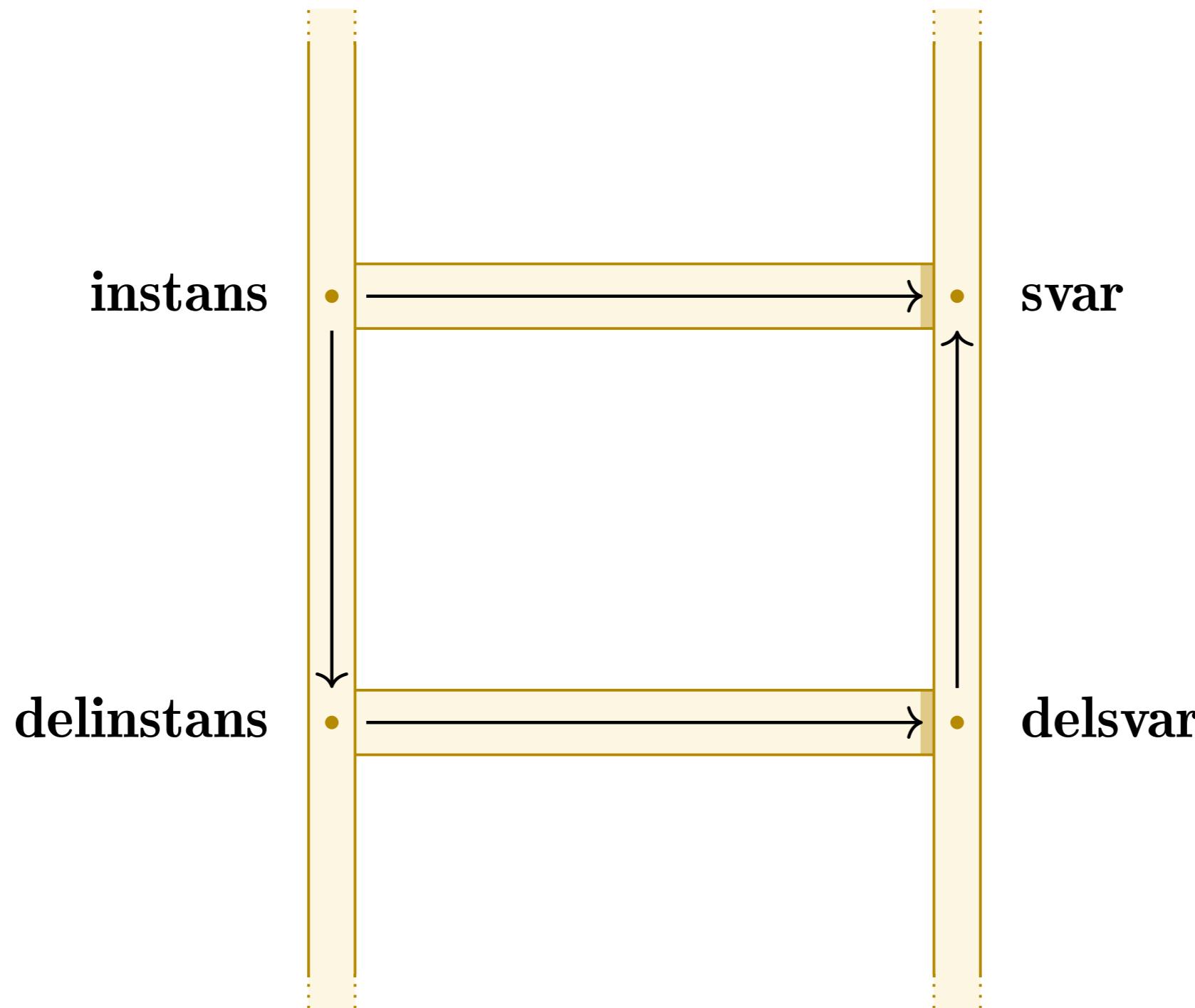
Hvorfor fungerer dette?

Problem

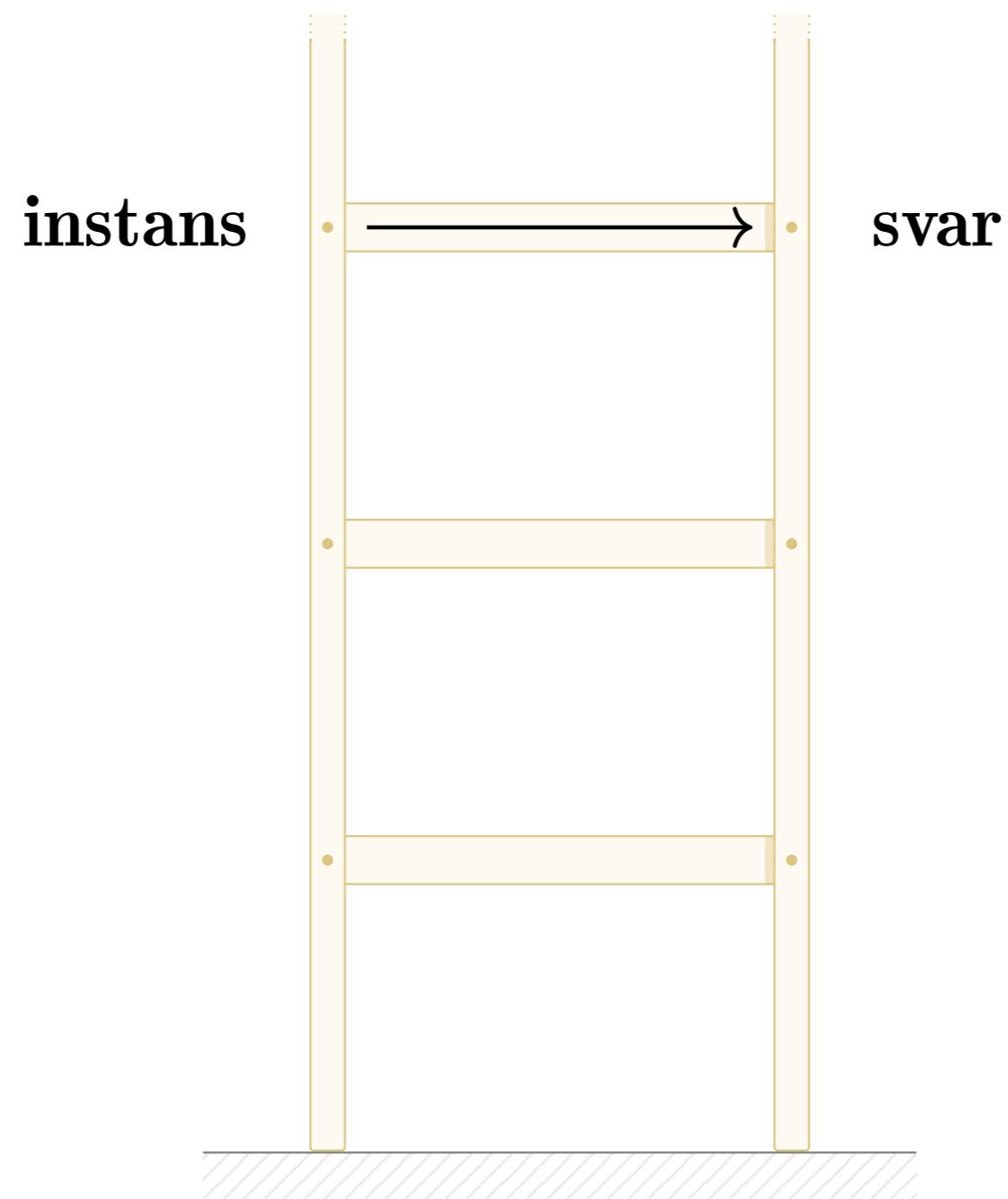


Det er samme prinsipp som før!

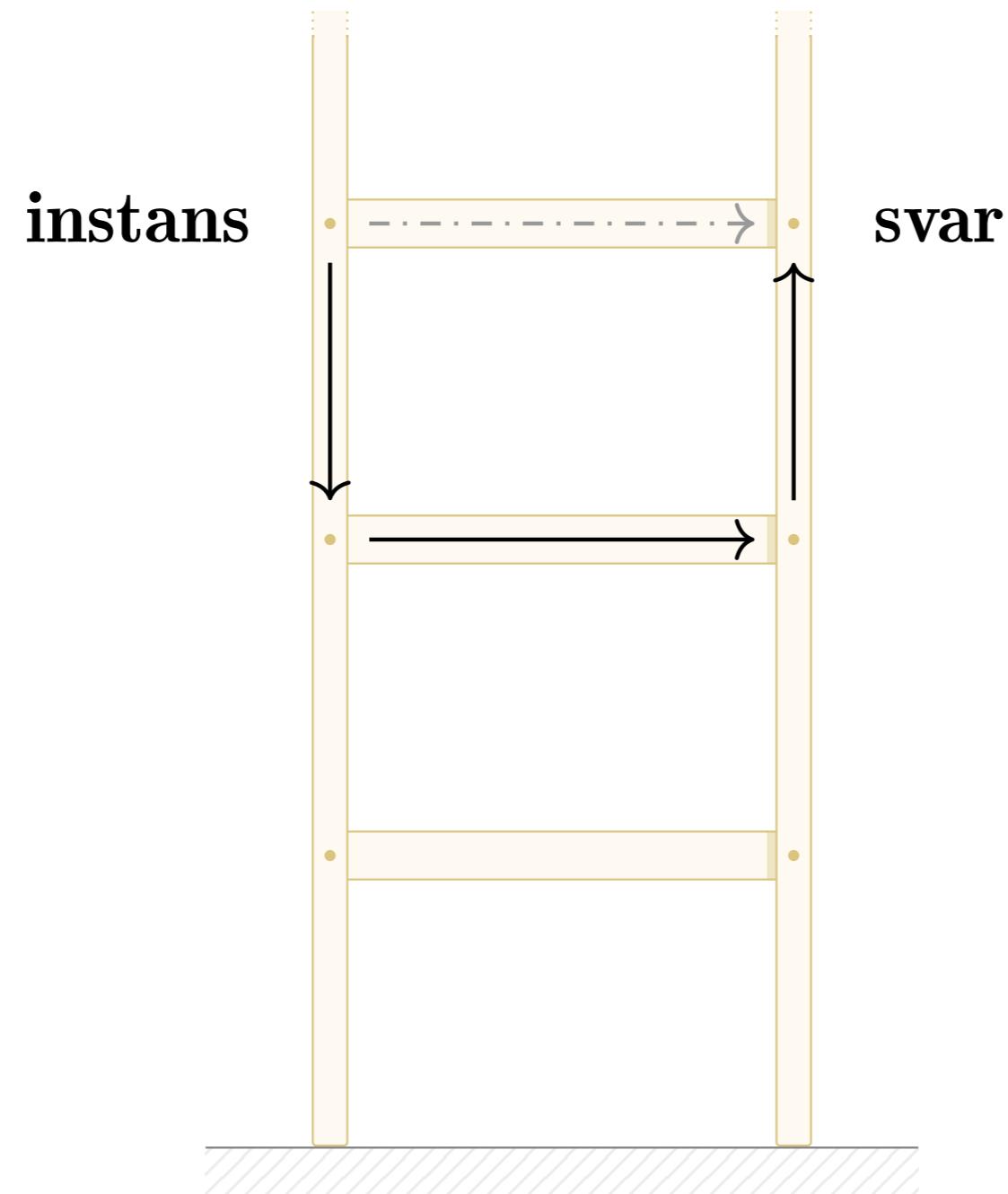
Problem



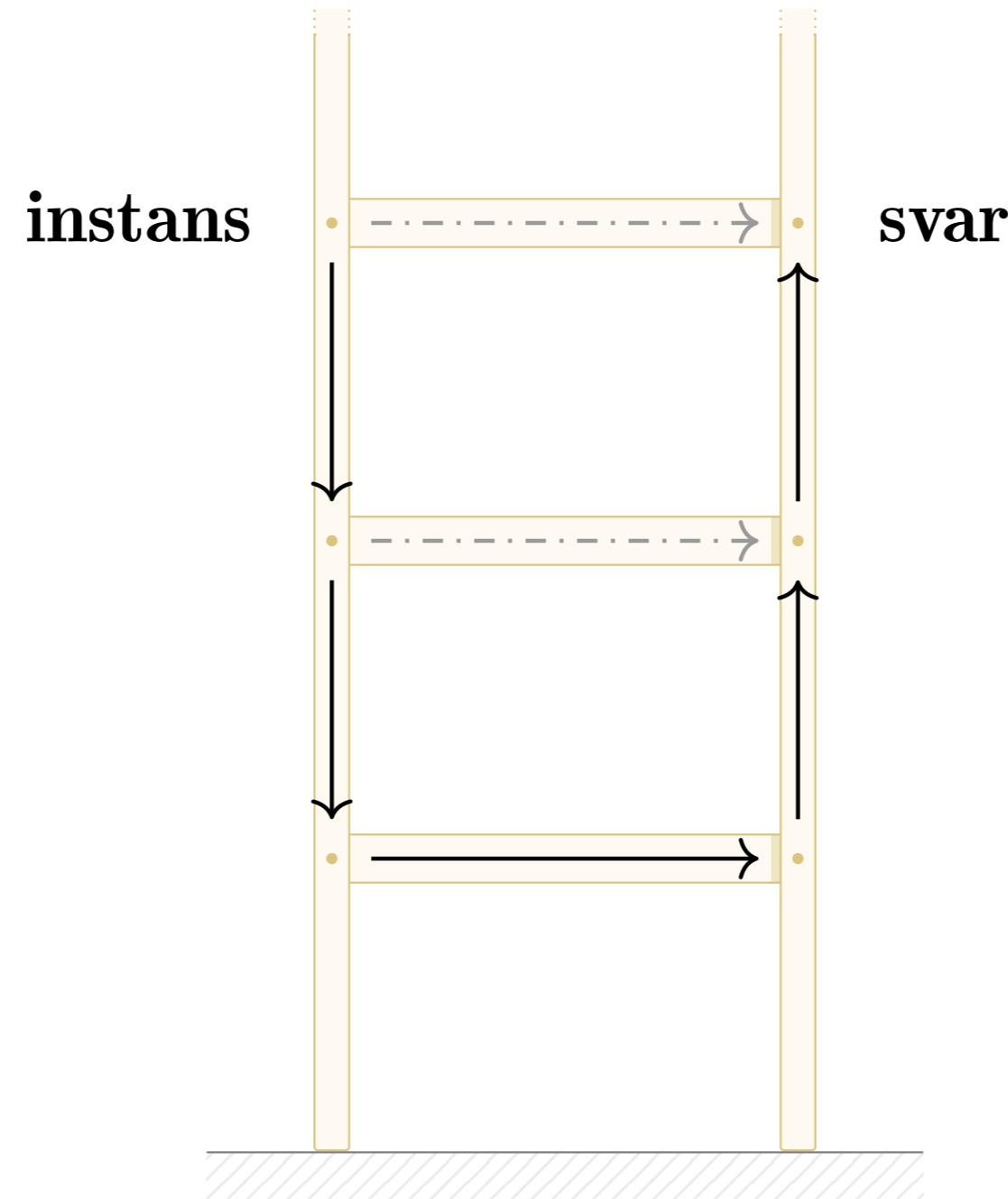
Vis grunntilfelle. Anta delløsning. Vis korrekt spalting/samling



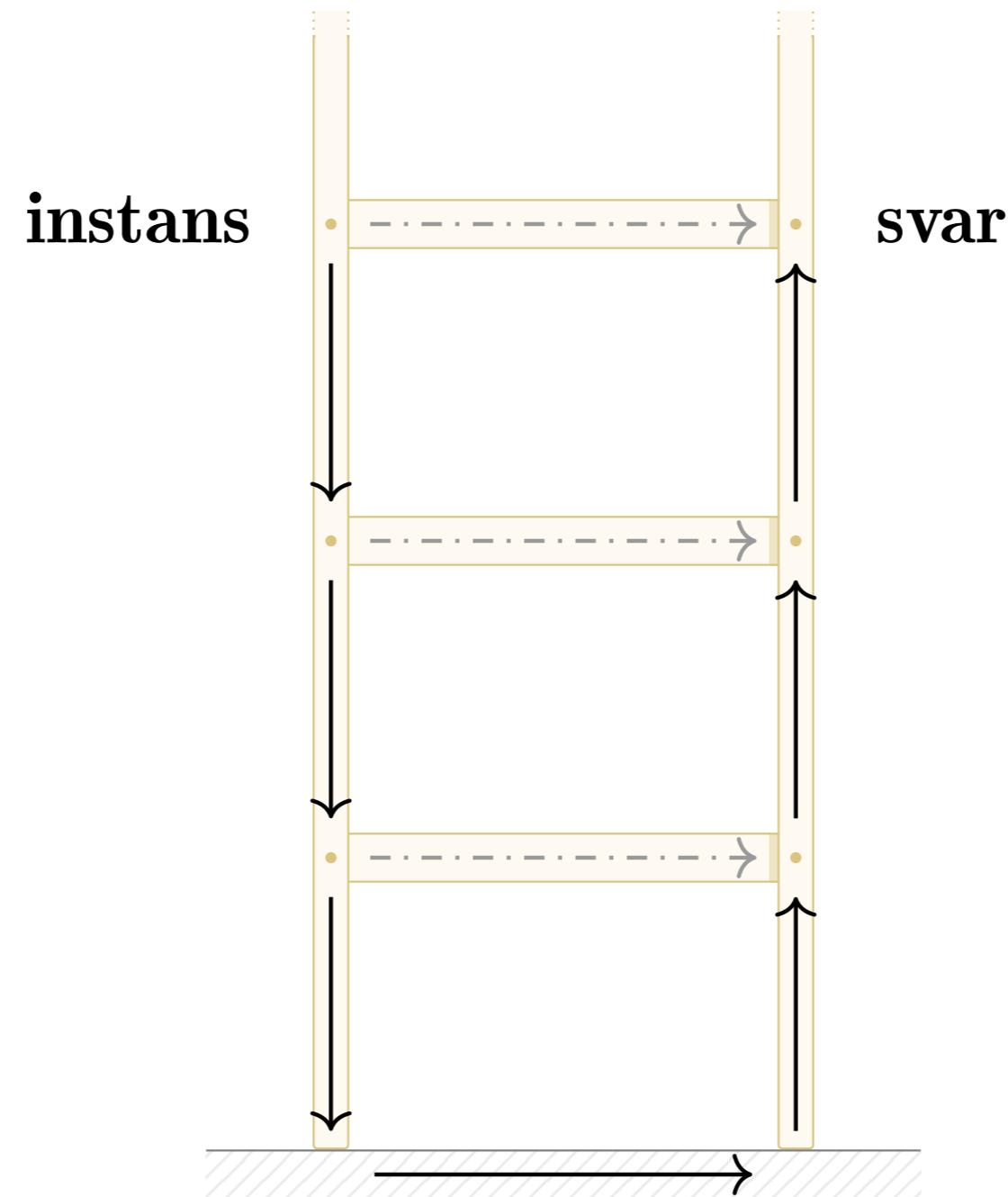
Poenget er at det samme skjer på hvert nivå



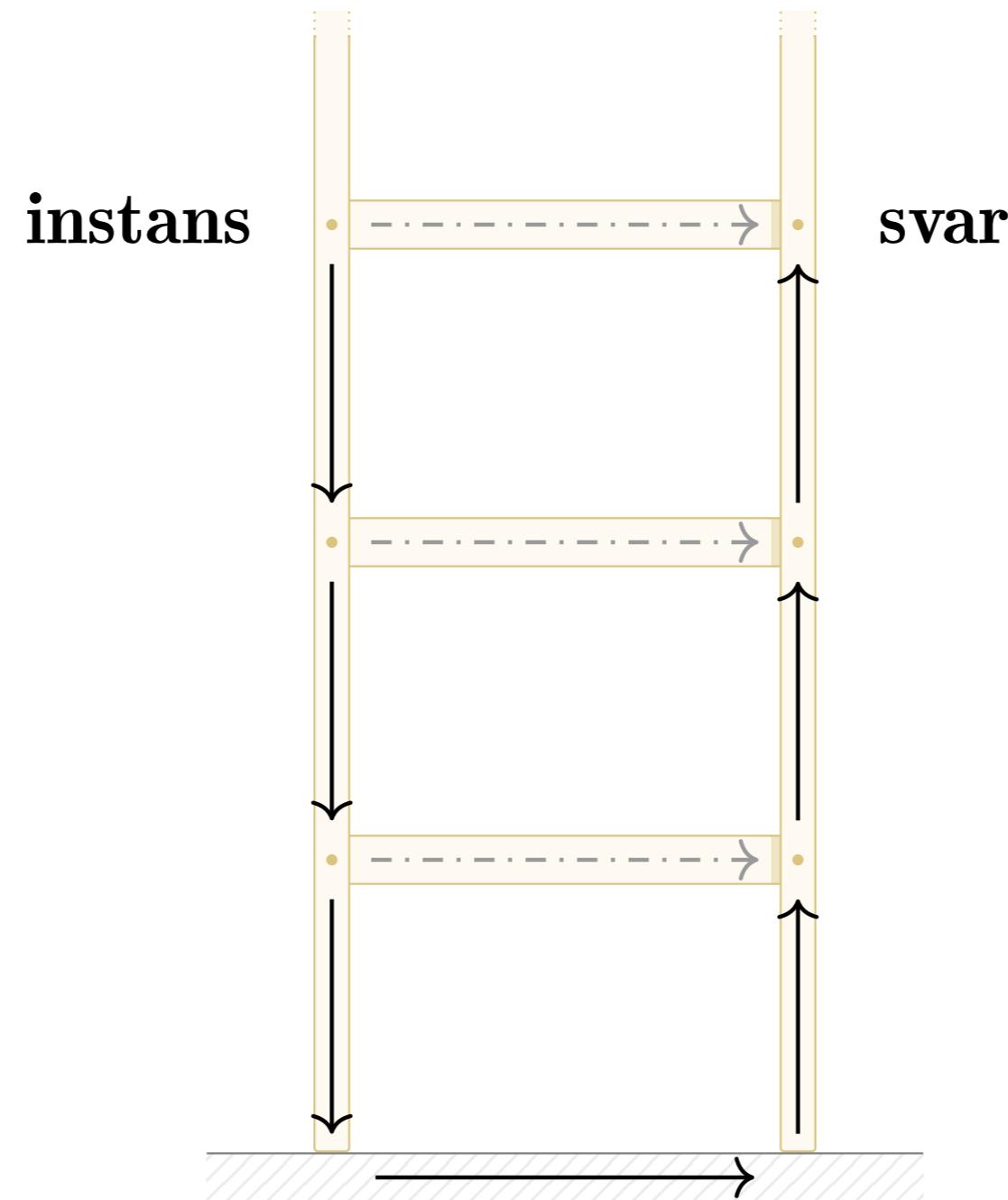
Poenget er at det samme skjer på hvert nivå



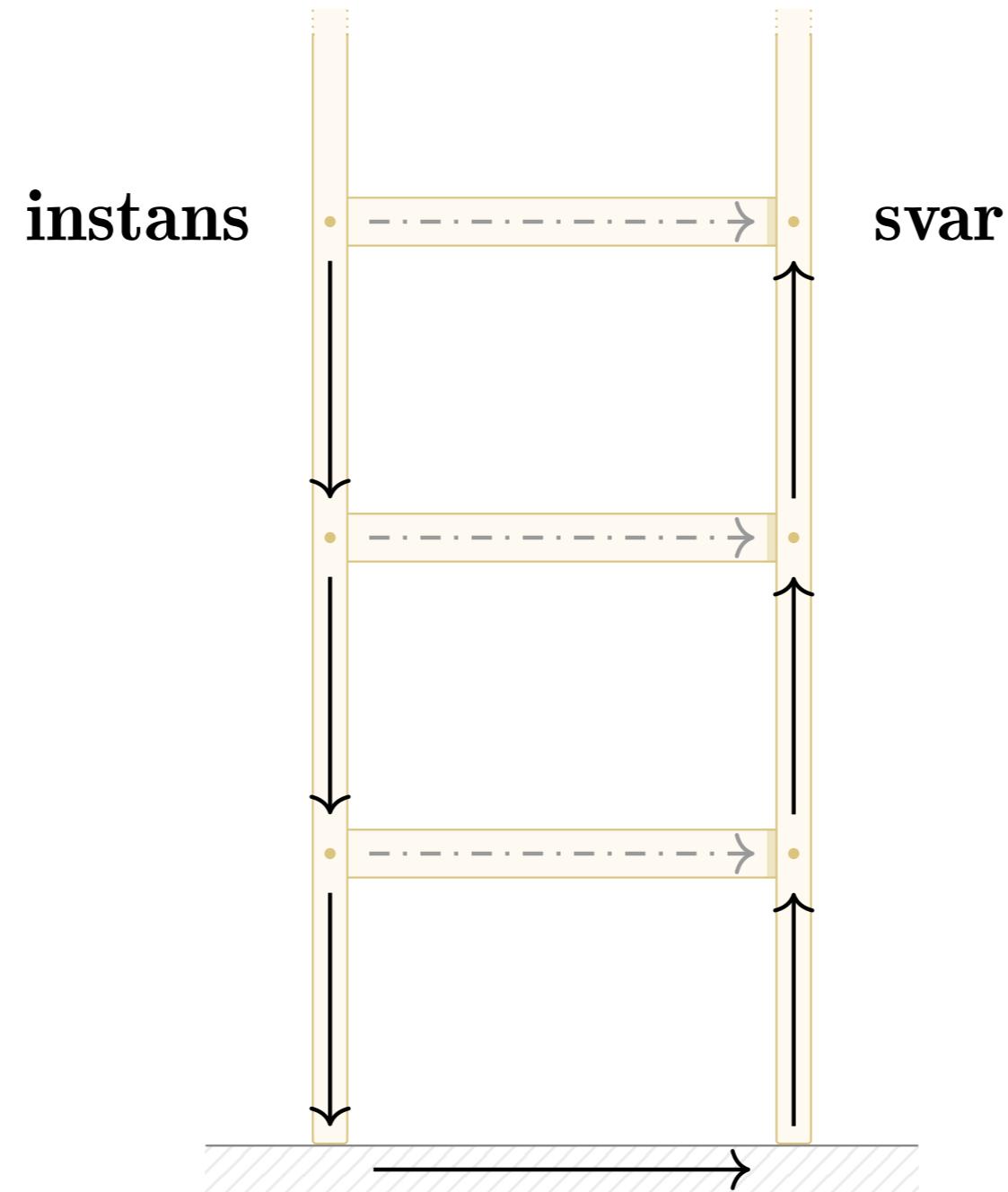
Poenget er at det samme skjer på hvert nivå



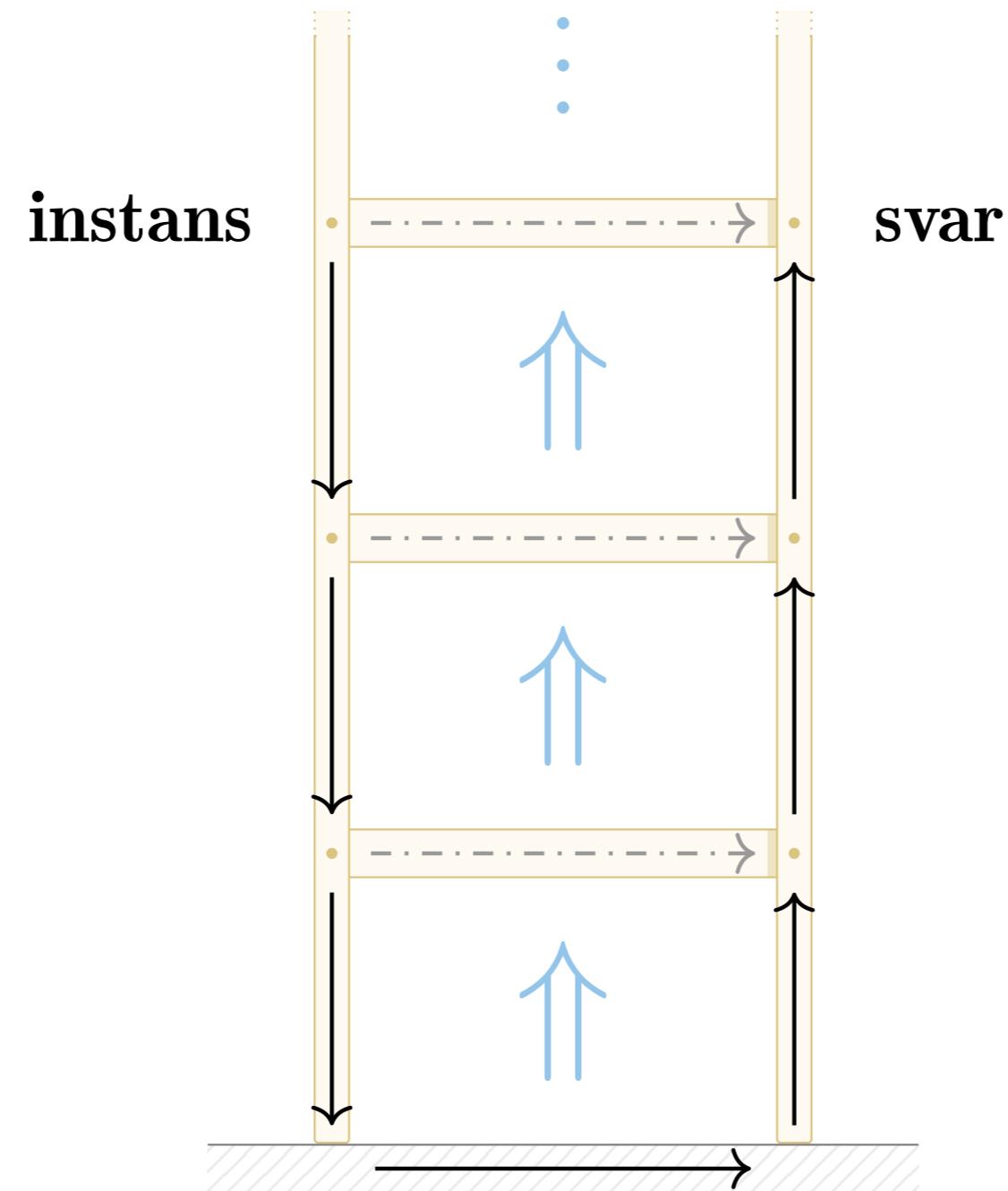
Poenget er at det samme skjer på hvert nivå



Om grunntilfelle, spalting og samling er rett...

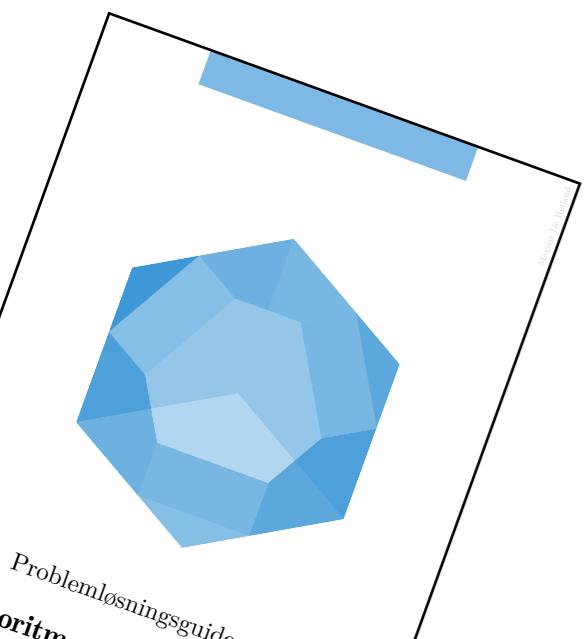


... så er alle svarene riktige



... så er alle svarene riktige

Generell strategi

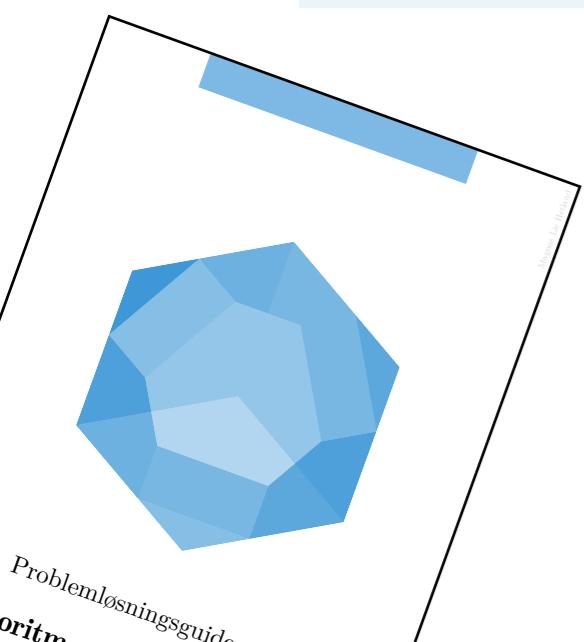


Tolkning

T

Definér problemet eller problemene du står overfor.
Klargjør hva din oppgave er: Hva skal du gjøre med
problemene?

Vanligvis: Hva er relasjonen mellom input og output?

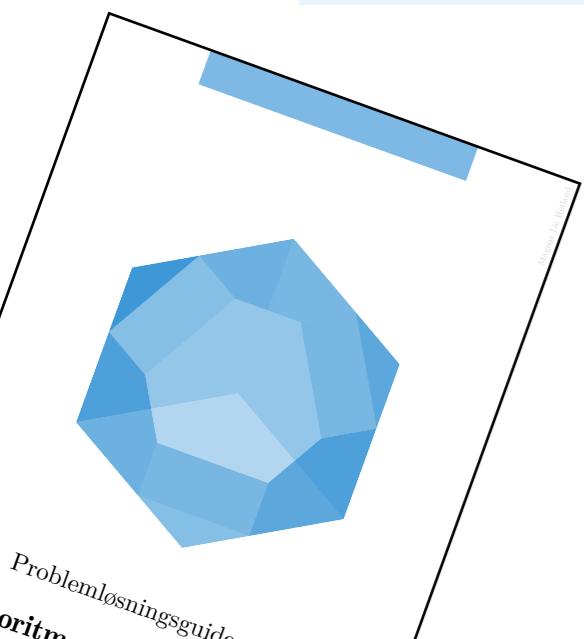


Analyse

A

Plukk problemet fra hverandre og plassér det i en større kontekst. List opp alt du har av relevant kunnskap og relevante verktøy.

Vanligvis: Del en vilkårlig instans i delinstanser.

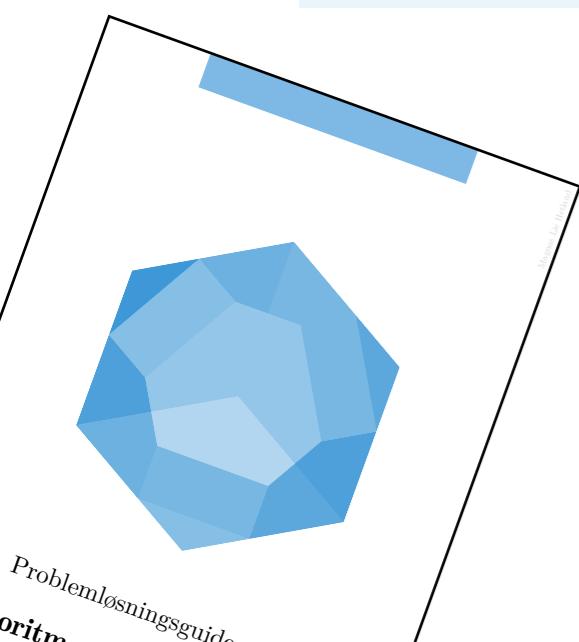


Syntese

S

Koble sammen bitene og fyll inn det som mangler av transformasjoner, mindre beregningstrinn og eventuelle korrekthetsbevis.

Vanligvis: Bygg løsning av hypotetiske delløsninger.



Tolkning

T

Hva er relasjonen mellom input og output?

Analyse

A

Del en vilkårlig instans i delinstanser.

Syntese

S

Bygg løsning av hypotetiske delløsninger.

Tolkning

T

Hva er relasjonen mellom input og output?

Analyse

A

Del en vilkårlig instans i delinstanser.

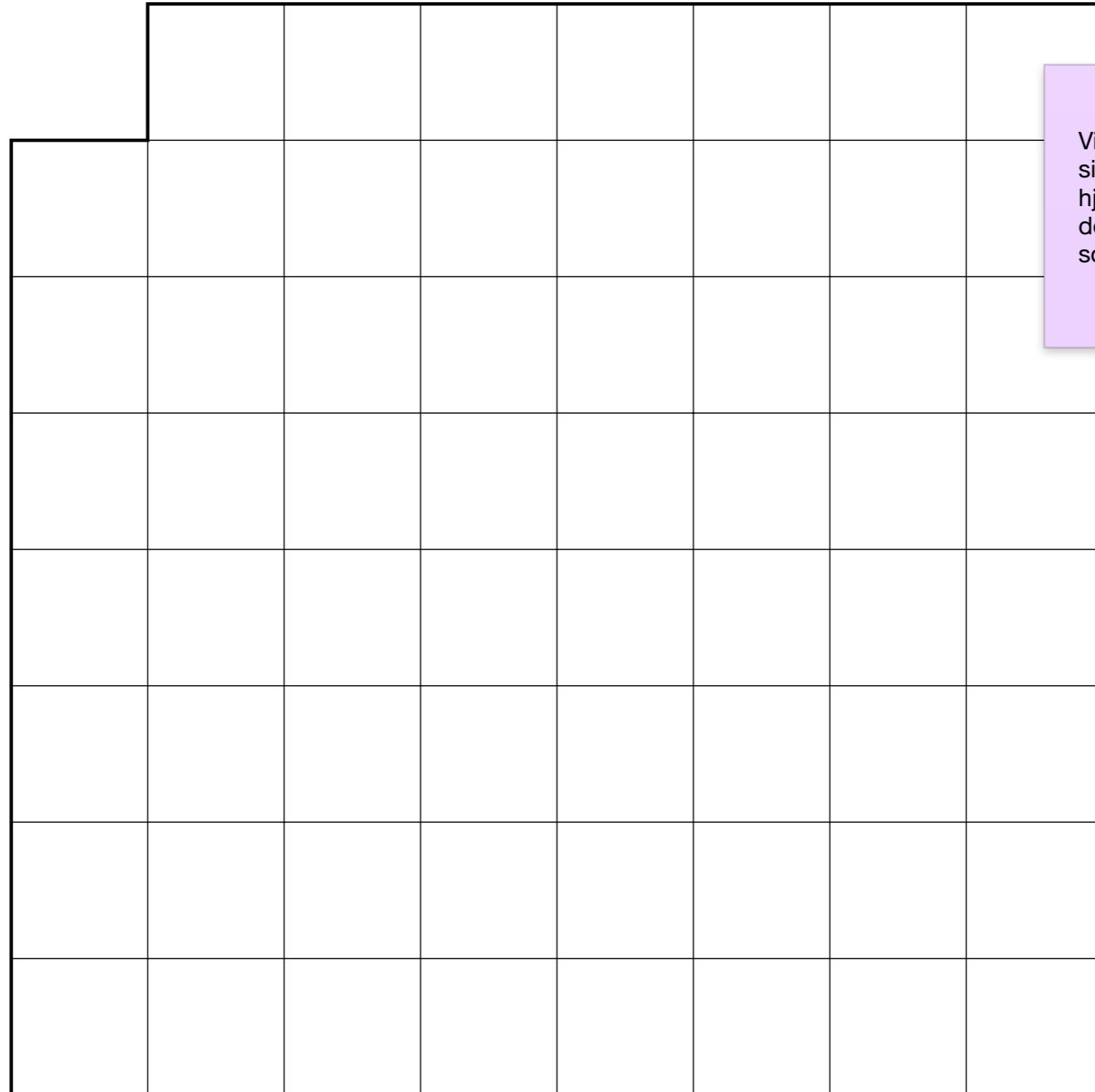
Syntese

S

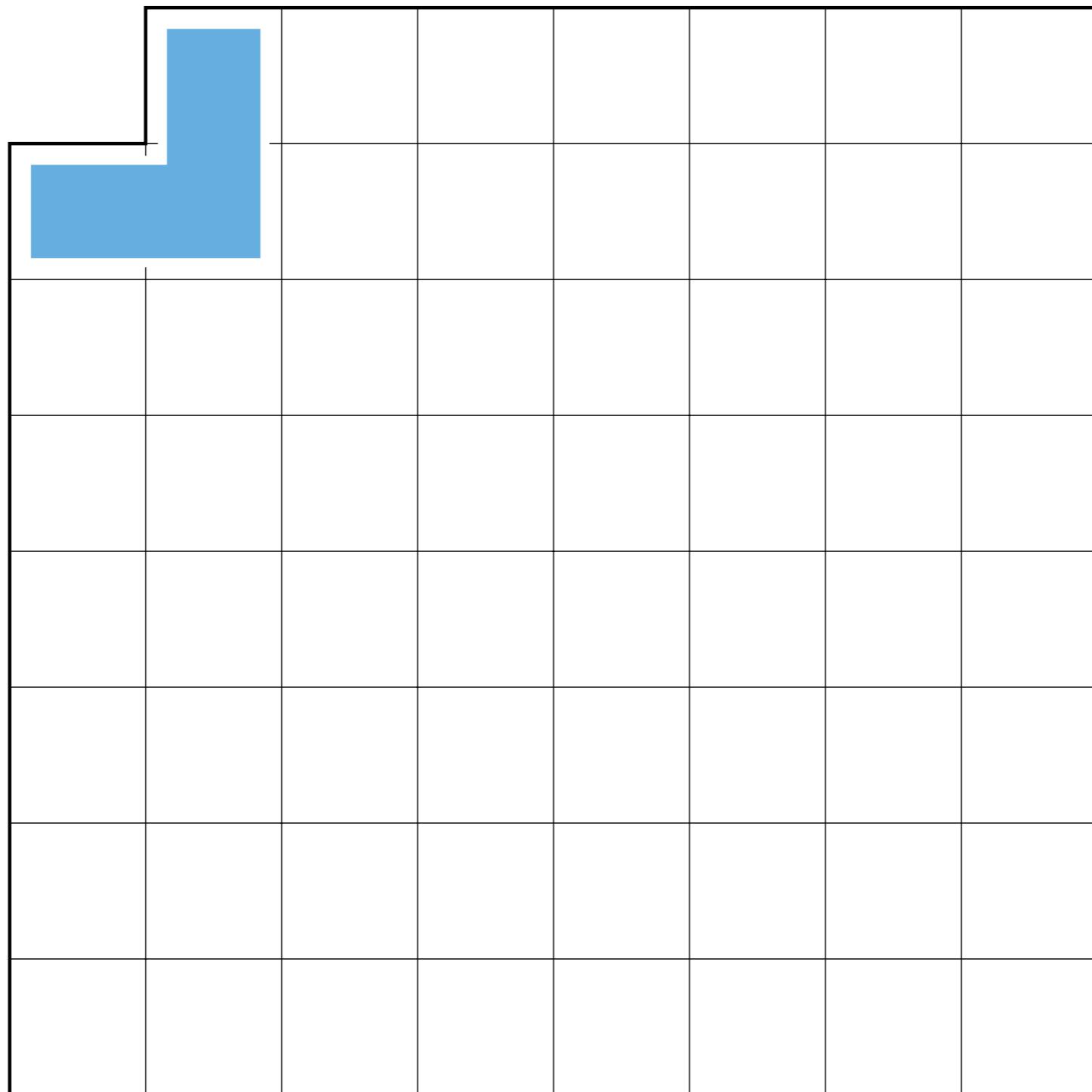
Bygg løsning av hypotetiske delløsninger.

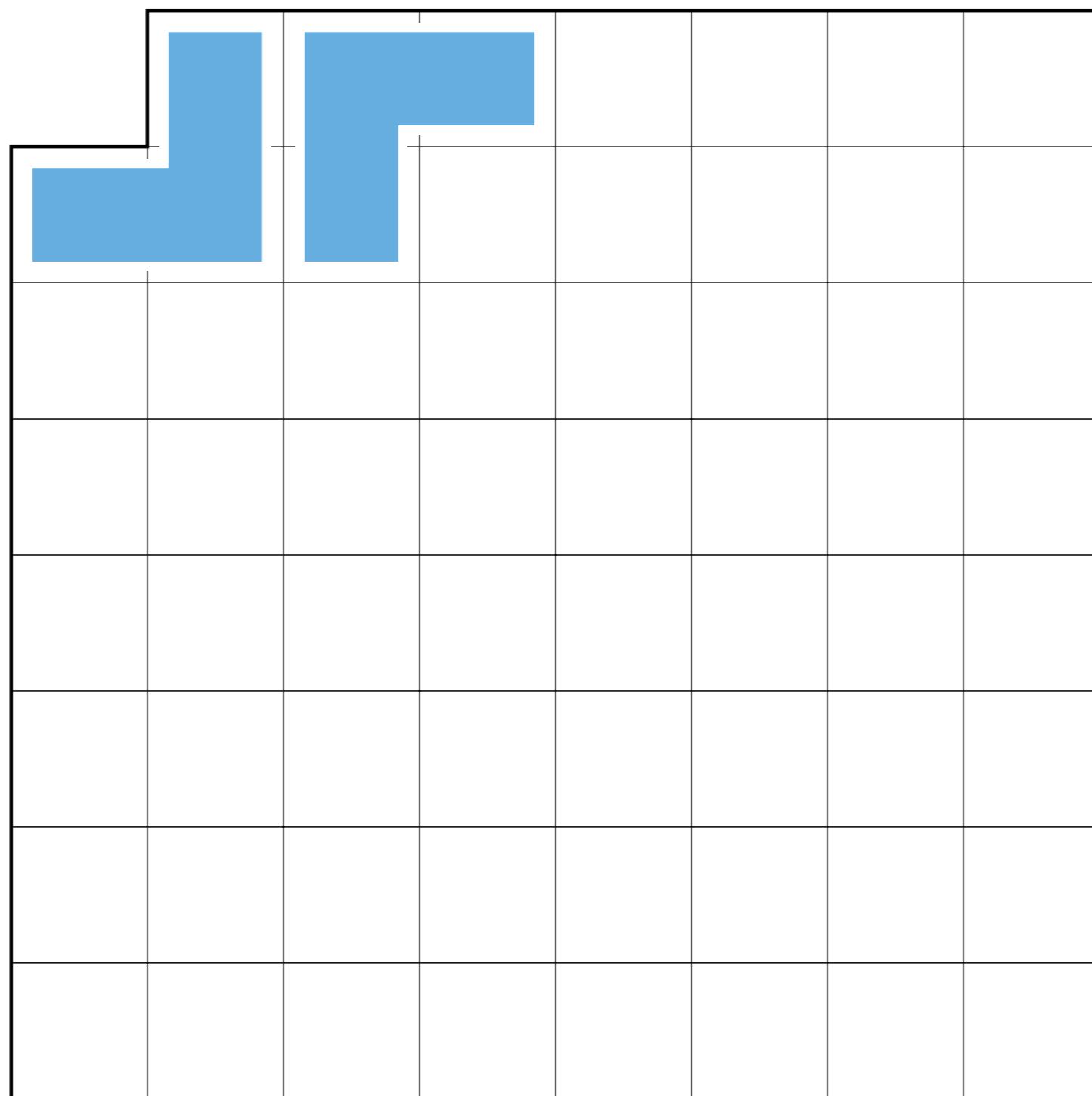
(Dette er ikke hele historien; mer i forelesning 7)

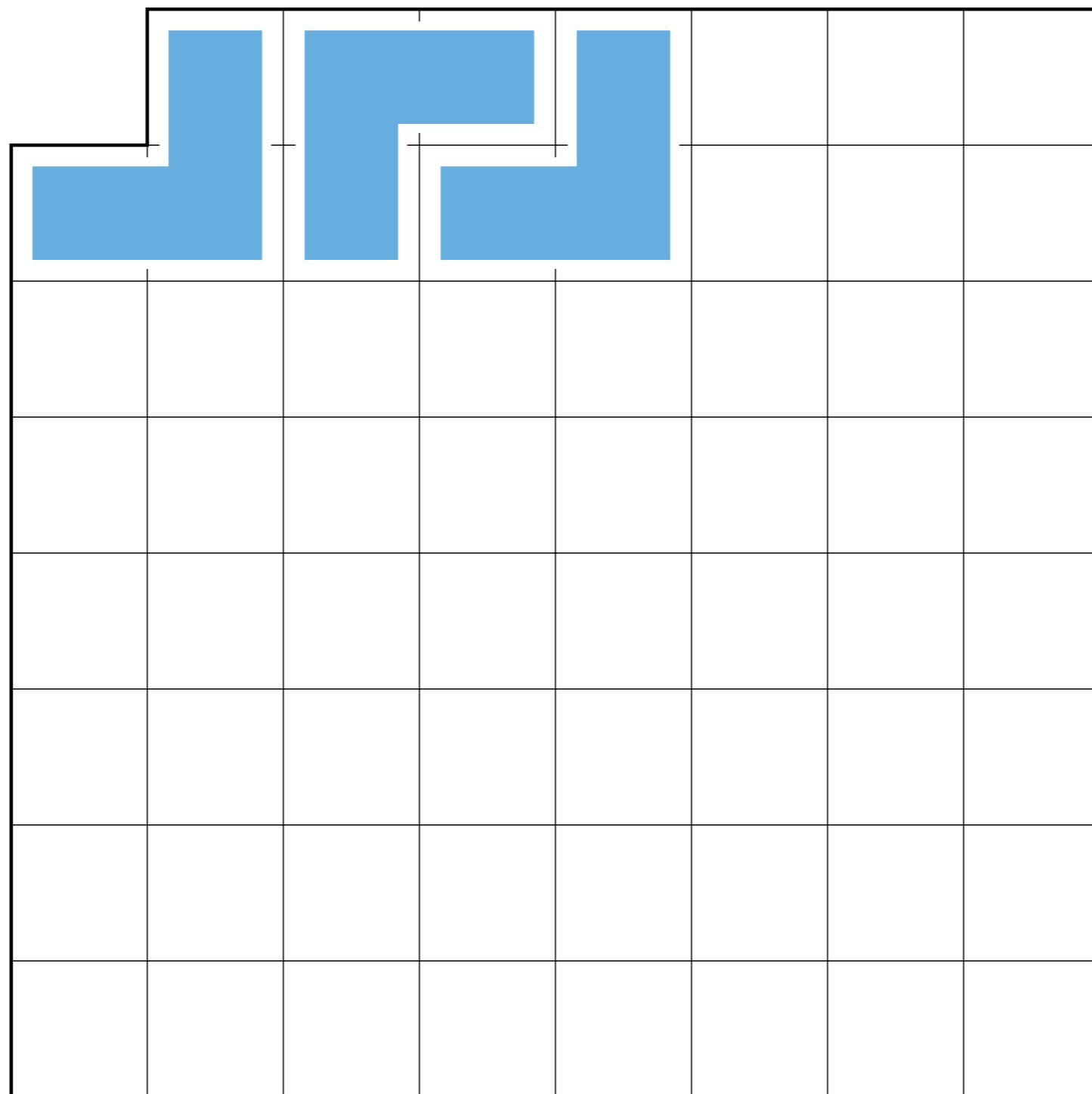
L-problemet

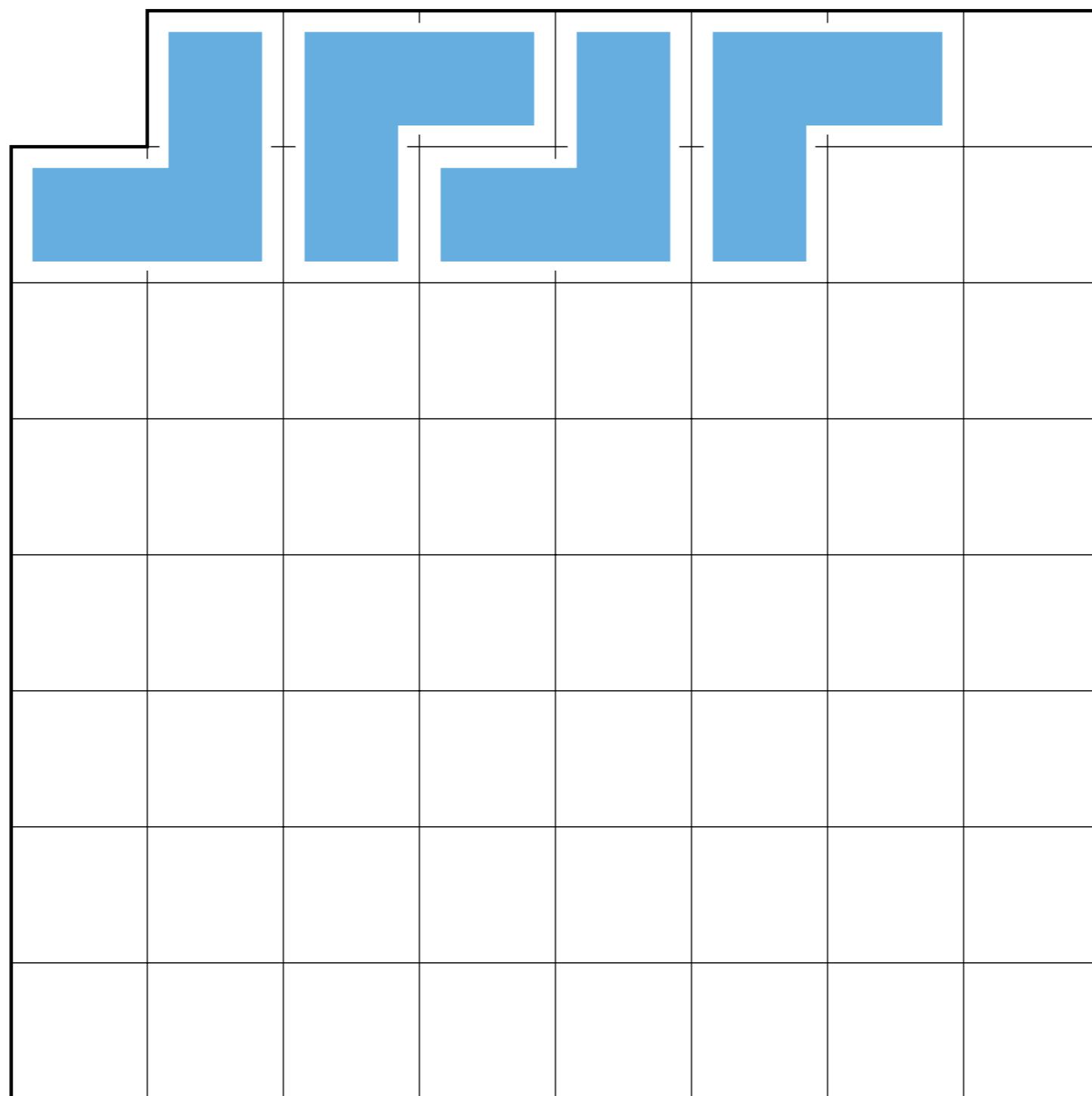


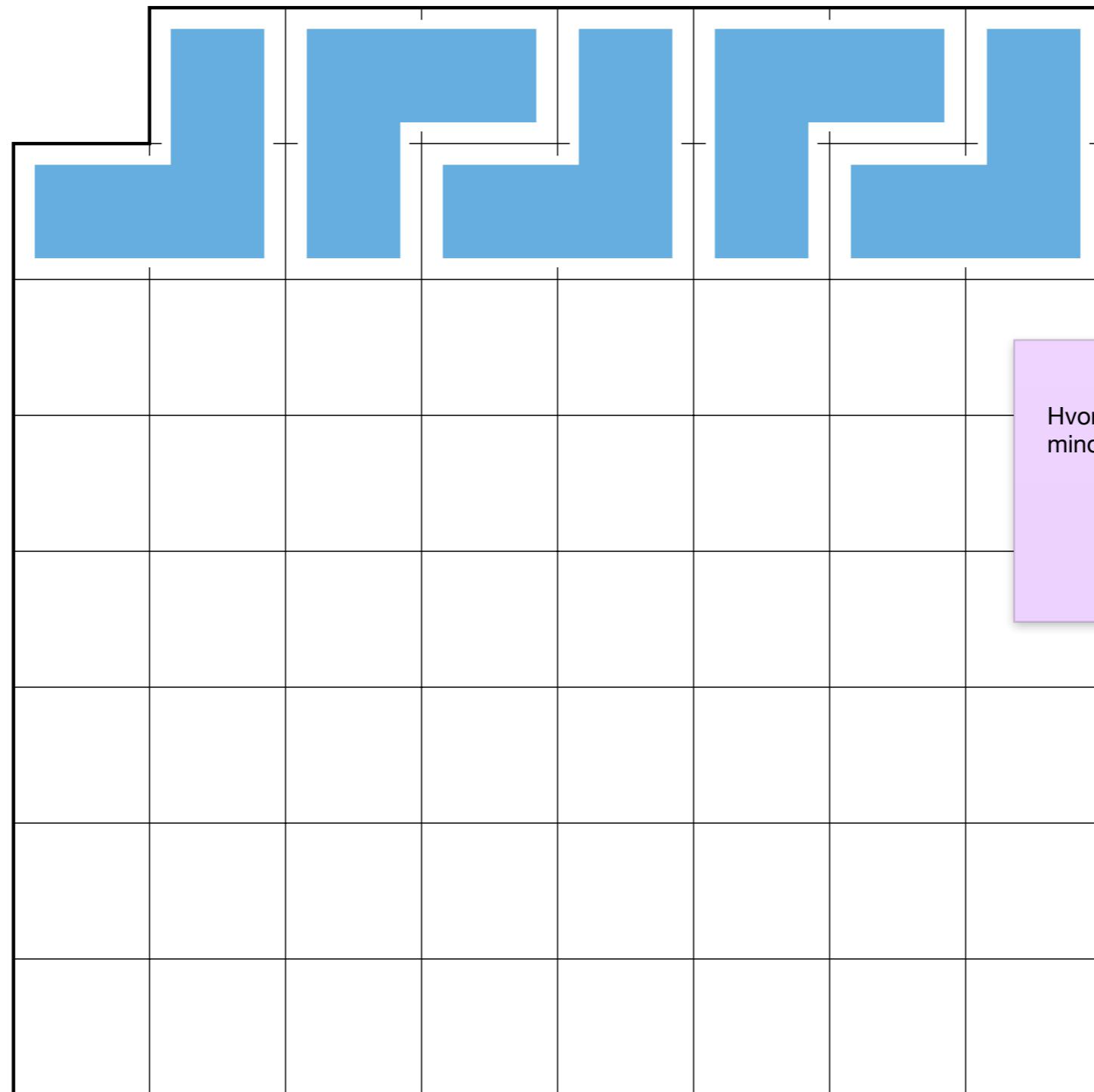
Vi har et kvadratisk rutenett der sidene er toerpotenser, og der ett hjørne mangler. Vi ønsker å dekke dette «brettet» med L-formede brikker som består av 3 ruter.











Tolkning

T

Hva er relasjonen mellom input og output?

Analyse

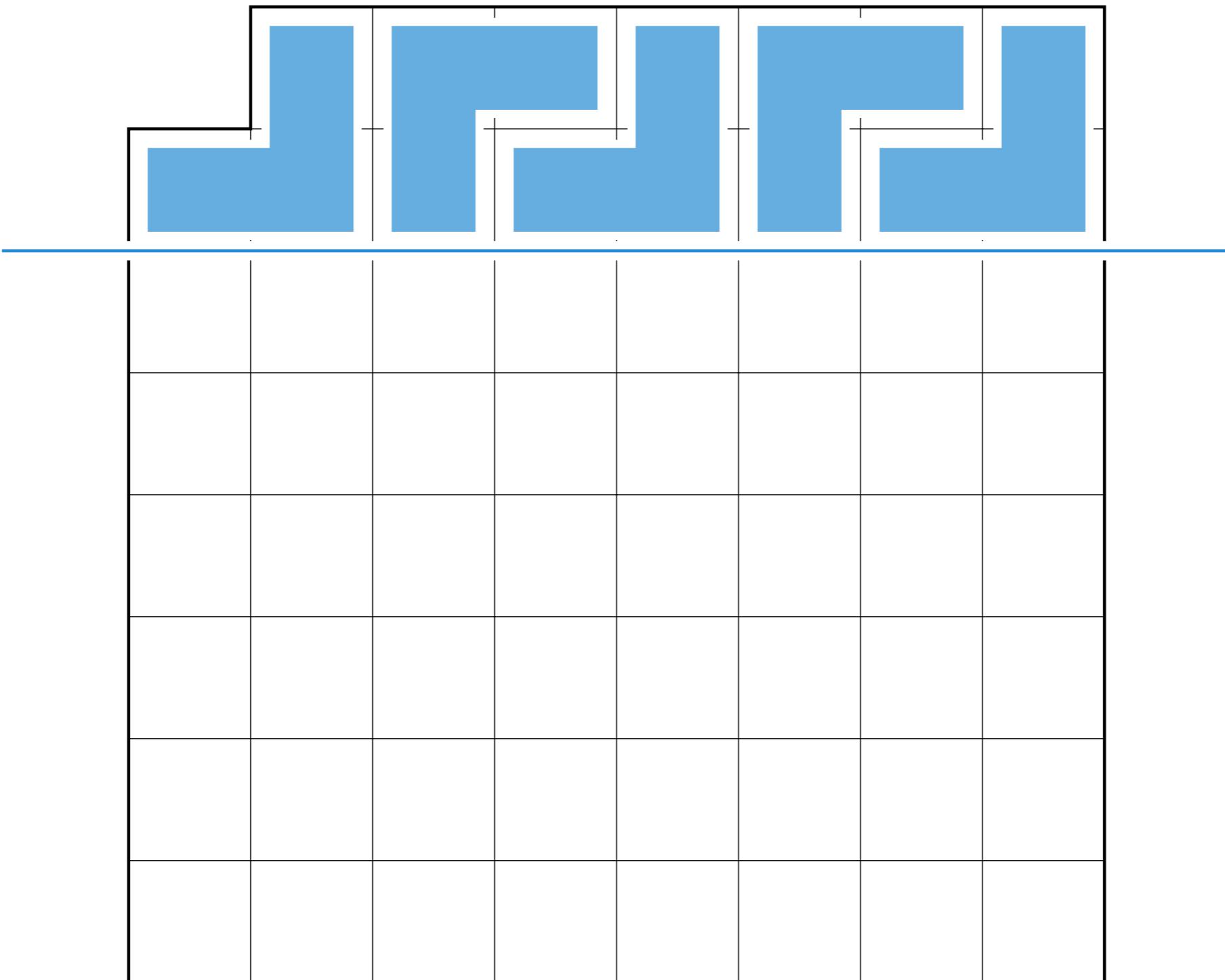
A

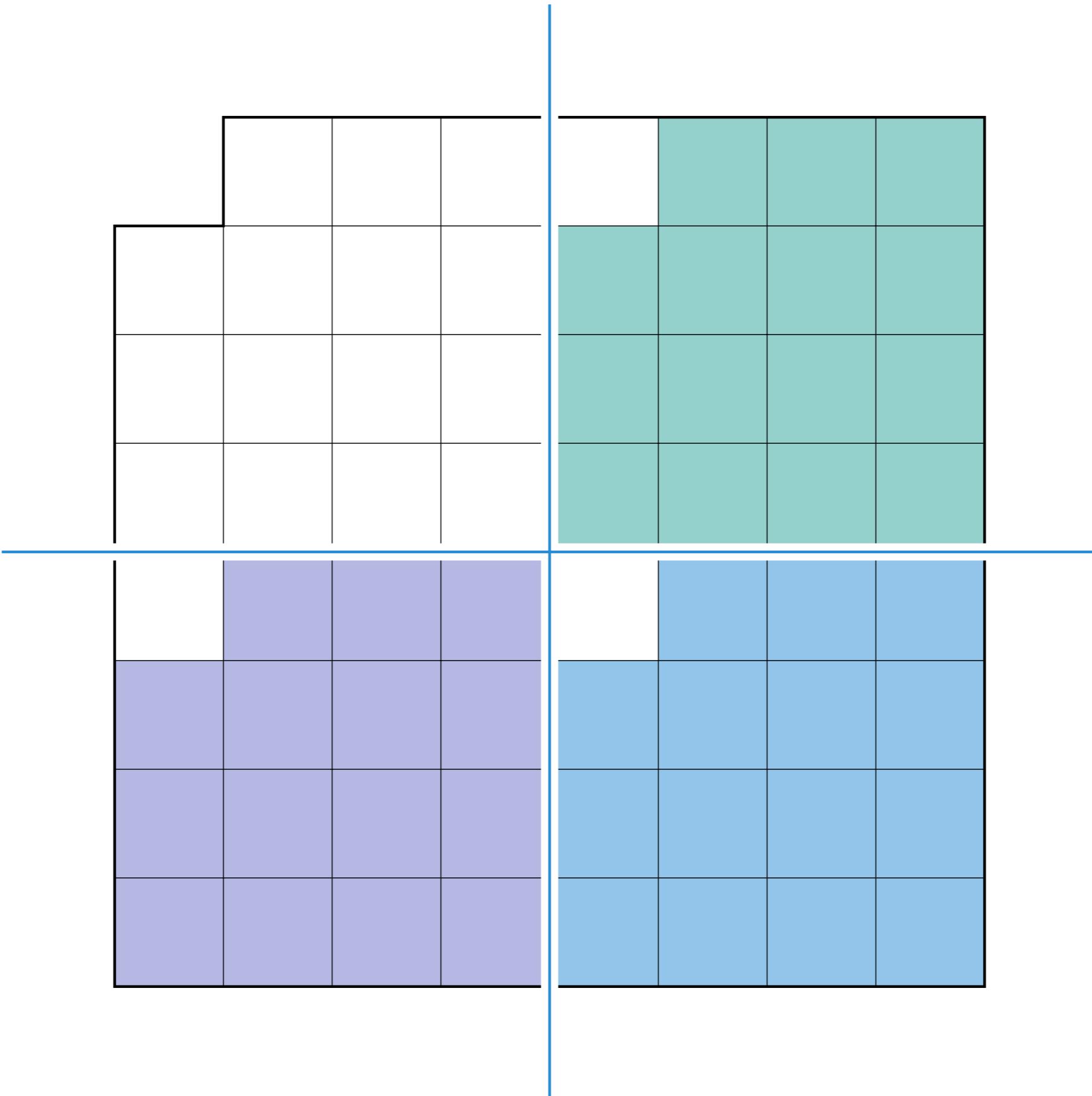
Del en vilkårlig instans i delinstanser.

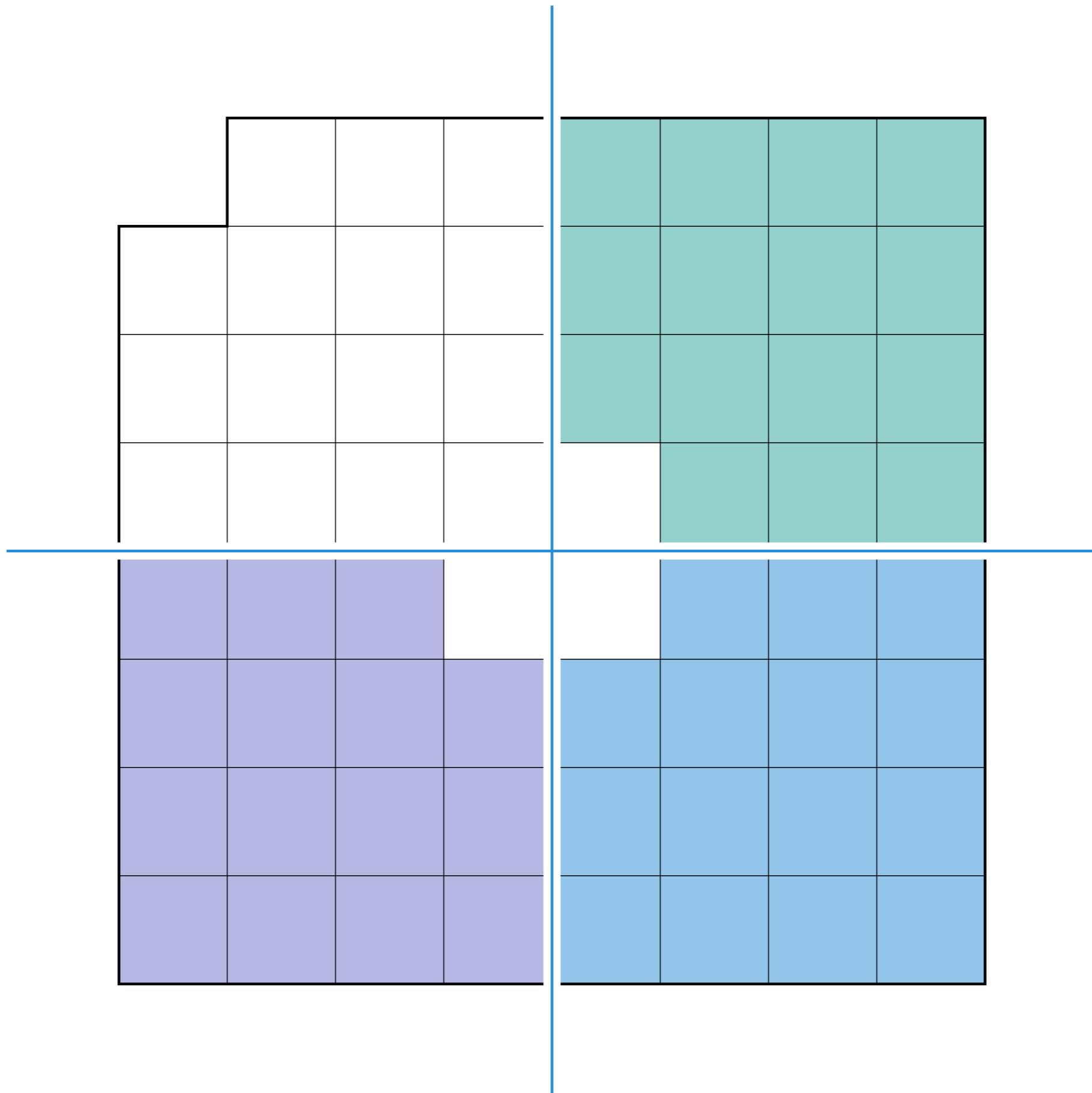
Syntese

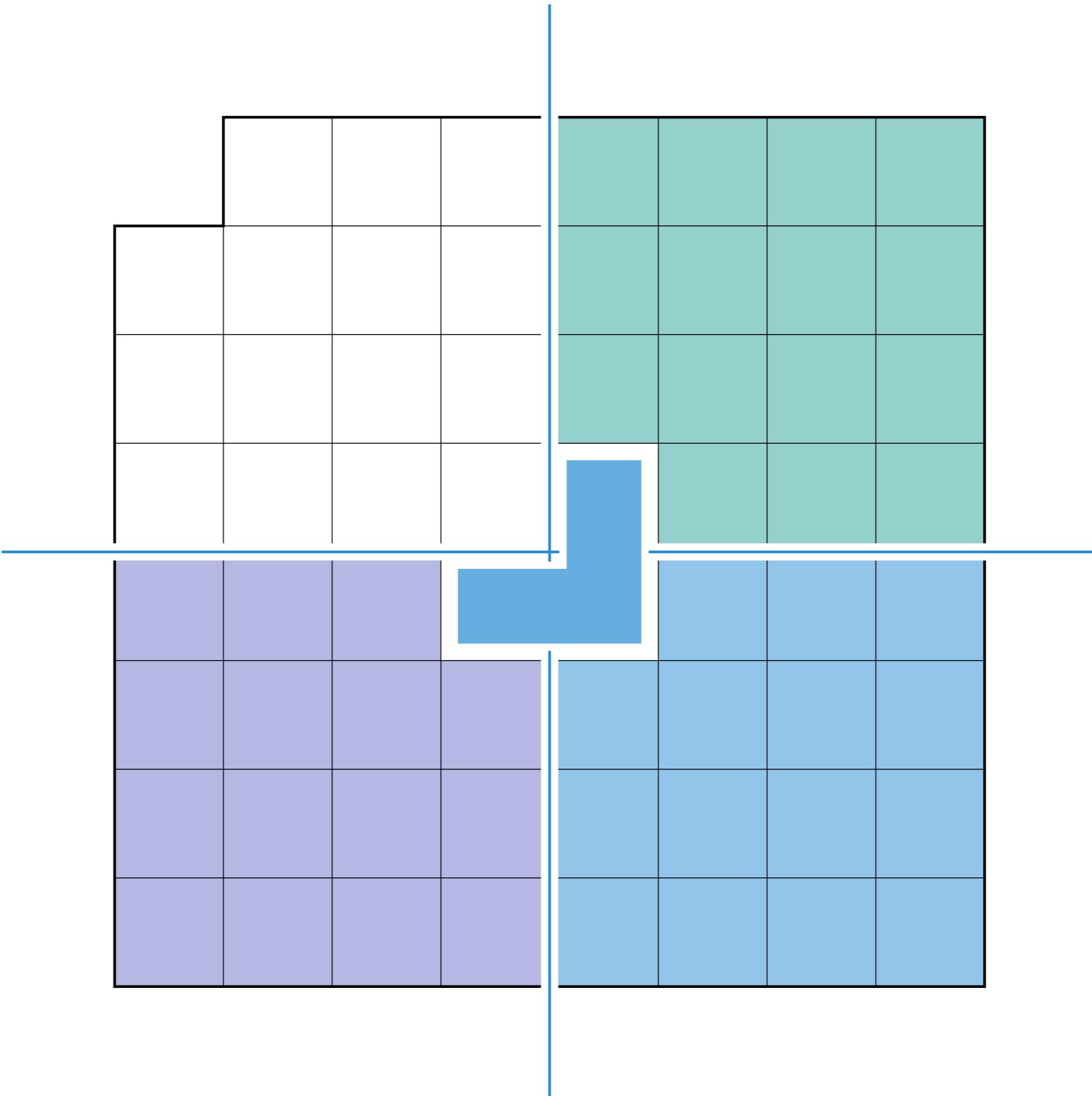
S

Bygg løsning av hypotetiske delløsninger.









COVER(A)

Dekk «nesten-kvadrat» A med L-brikker

COVER(A)
1 place middle L

Manglende hjørne i samme retning som A

$\text{COVER}(A)$

- 1 place middle L
- 2 if A is 2×2

Grunntilfelle: Vi har løst denne delinstansen!

```
COVER(A)
1 place middle L
2 if A is 2 × 2
3      return
```

Grunntilfelle: Vi har løst denne delinstansen!

```
COVER(A)
1 place middle L
2 if A is  $2 \times 2$ 
3   return
4 for each quadrant Q
```

Spalt i fire delinstanser: Mindre kvadrater med manglende hjørne

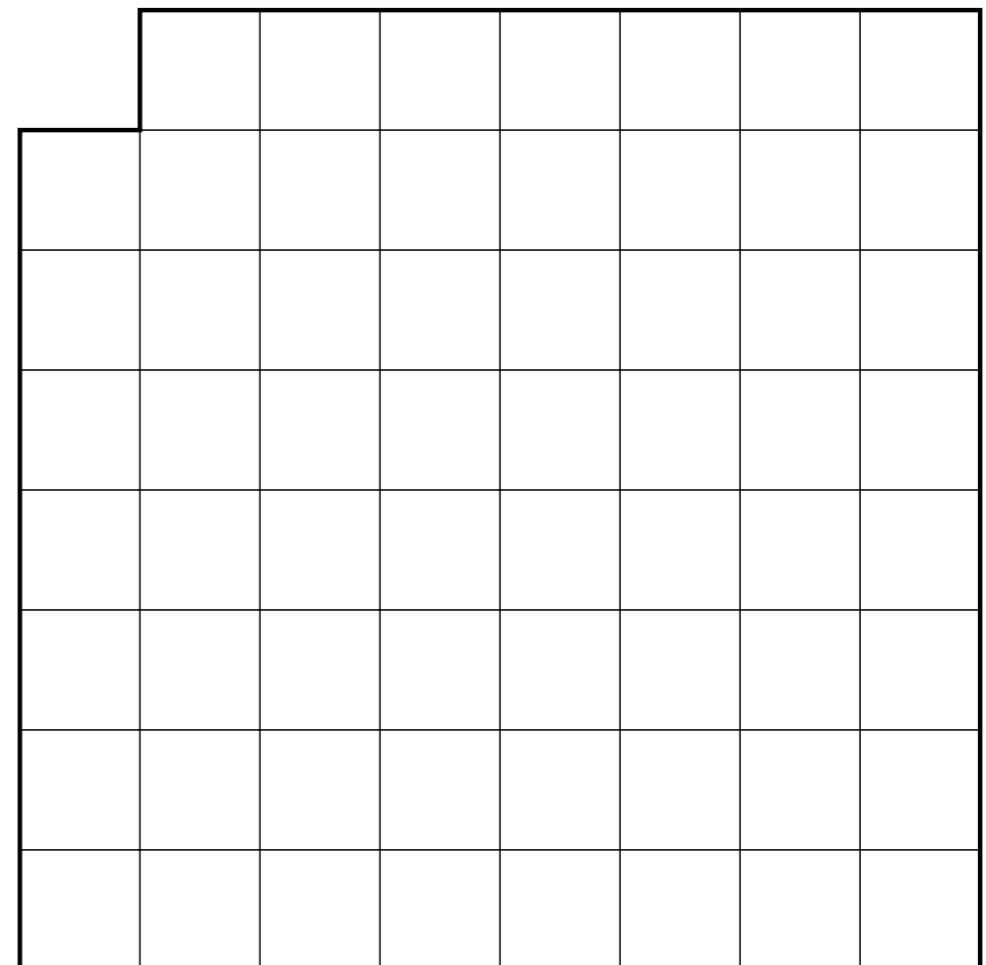
```
COVER(A)
1 place middle L
2 if A is  $2 \times 2$ 
3   return
4 for each quadrant Q
5   COVER(Q)
```

Løs disse fire. Nå vet vi jo hvordan!

$\text{COVER}(A)$

```
1 place middle L
2 if A is  $2 \times 2$ 
3   return
4 for each quadrant Q
5    $\text{COVER}(Q)$ 
```

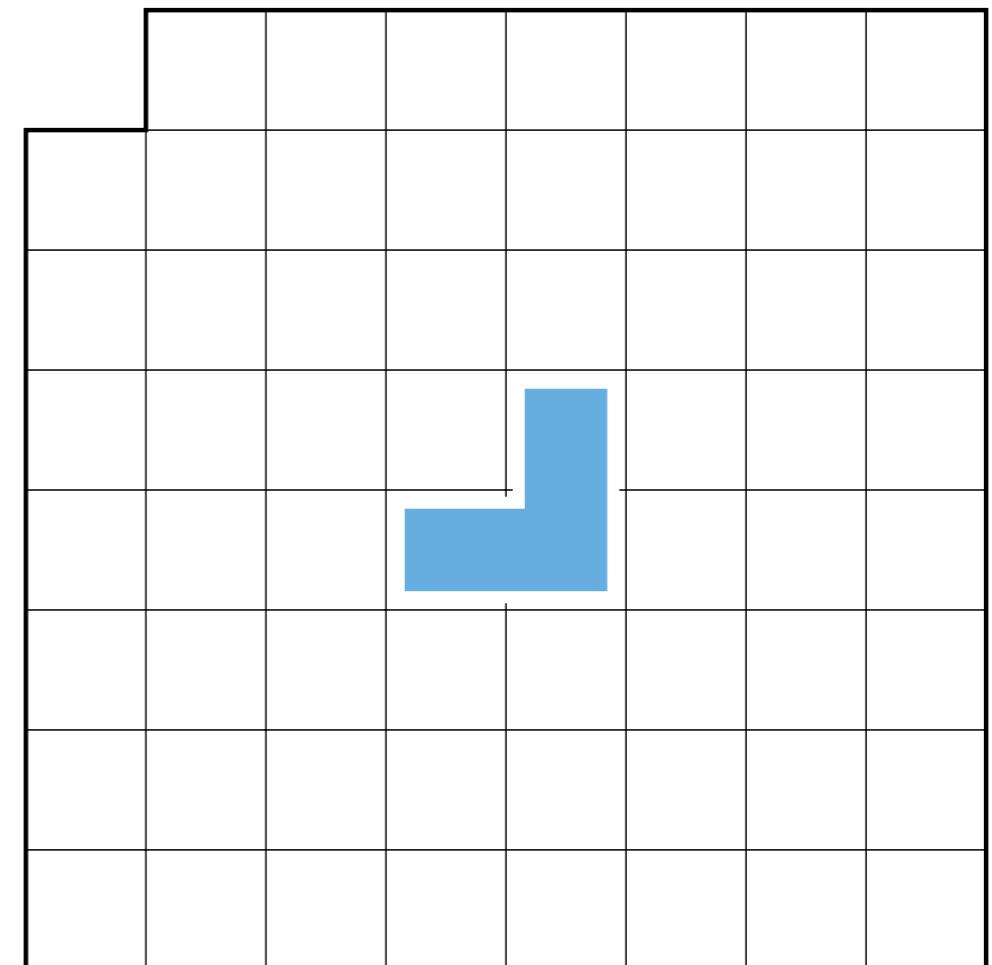
$A, Q = \blacksquare, \square$



$\text{COVER}(A)$

```
1 place middle L
2 if A is  $2 \times 2$ 
3   return
4 for each quadrant Q
5   COVER(Q)
```

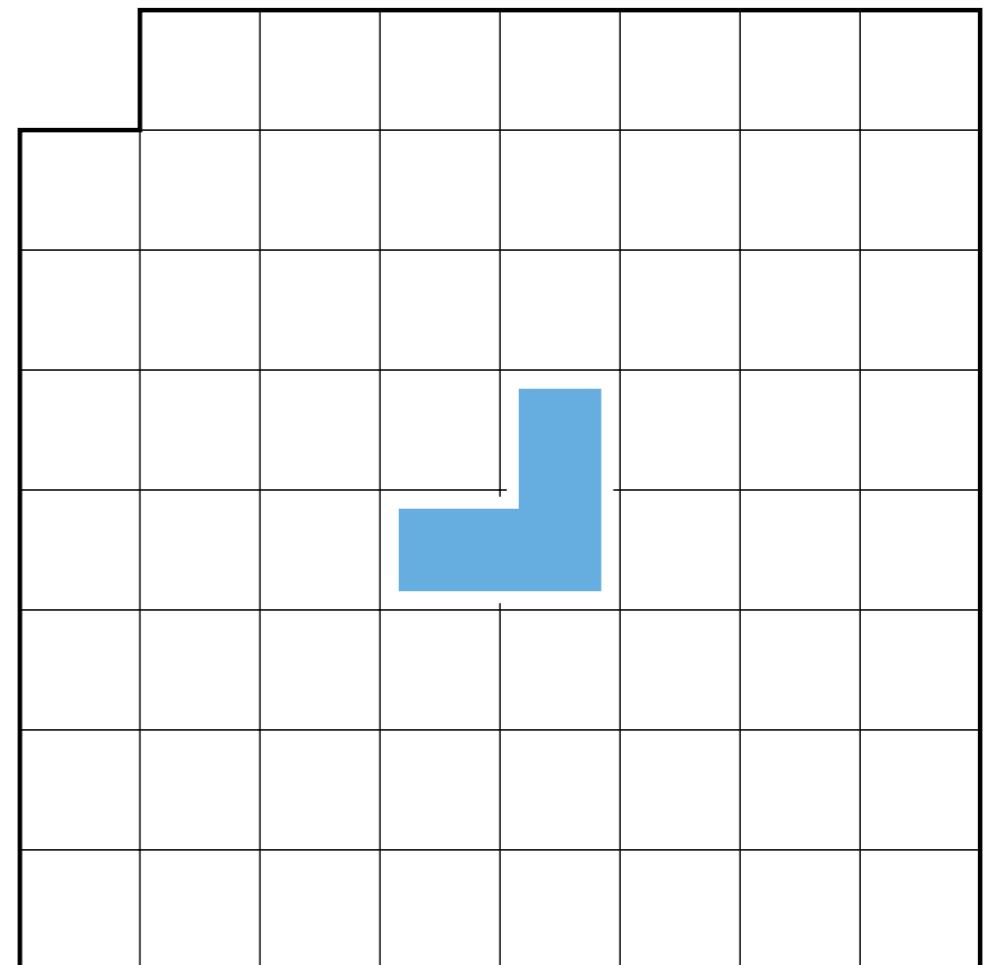
$A, Q = \blacksquare, \square$



$\text{COVER}(A)$

```
1 place middle L
2 if A is  $2 \times 2$ 
3   return
4 for each quadrant Q
5    $\text{COVER}(Q)$ 
```

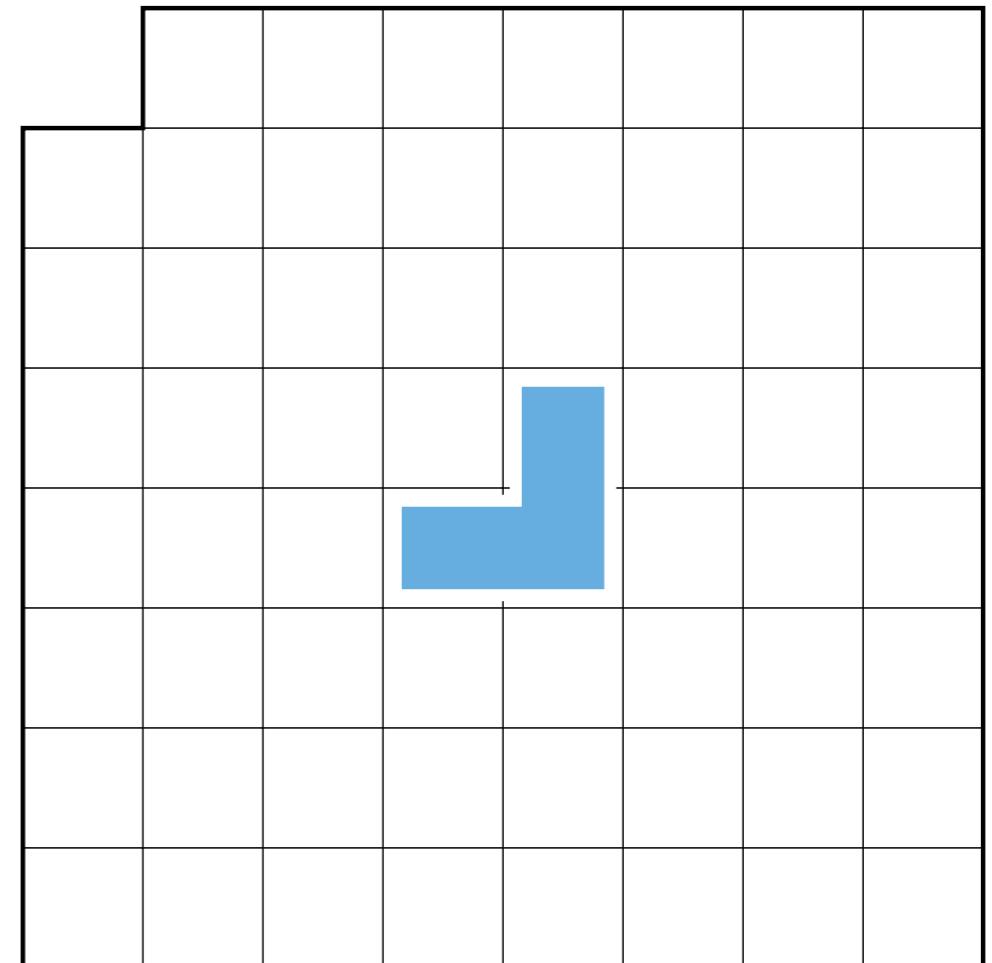
$A, Q = \blacksquare, \square$



$\text{COVER}(A)$

```
1 place middle L
2 if A is  $2 \times 2$ 
3   return
4 for each quadrant Q
5   COVER(Q)
```

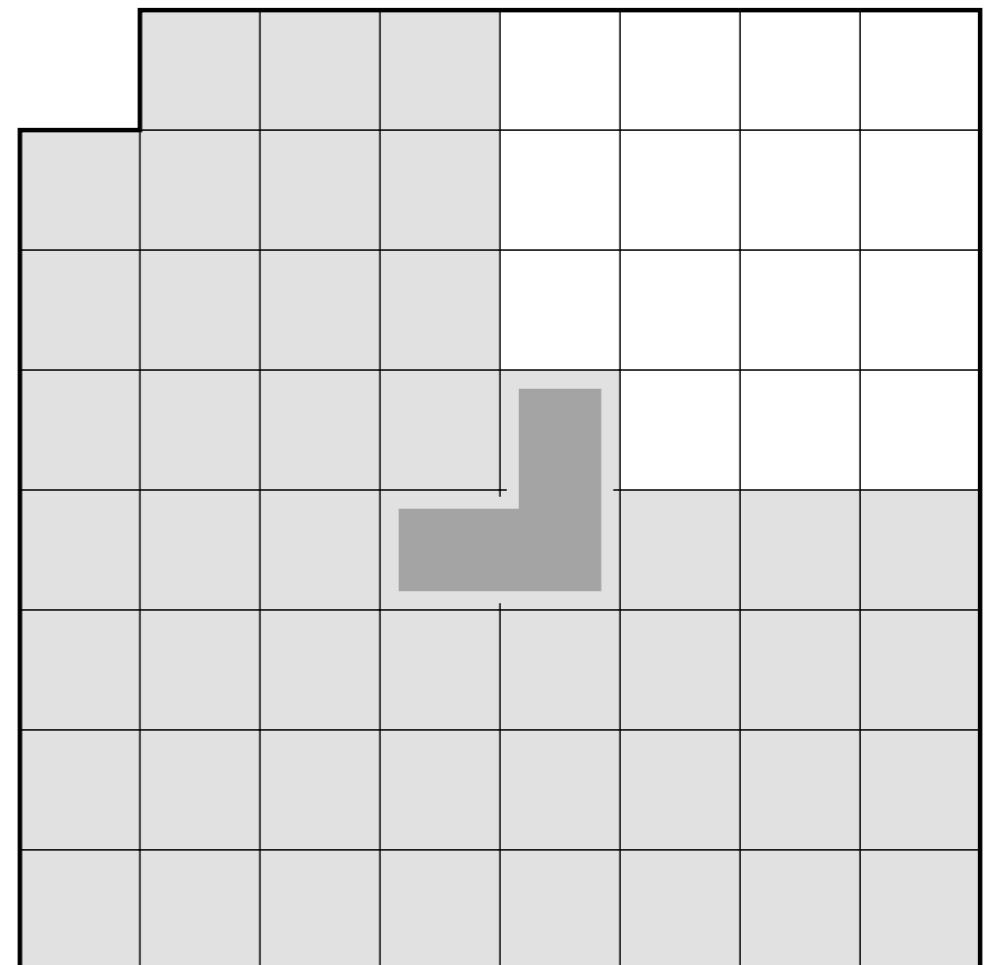
$A, Q = \blacksquare, \square$



$\text{COVER}(A)$

```
1 place middle L
2 if A is  $2 \times 2$ 
3   return
4 for each quadrant Q
5   COVER(Q)
```

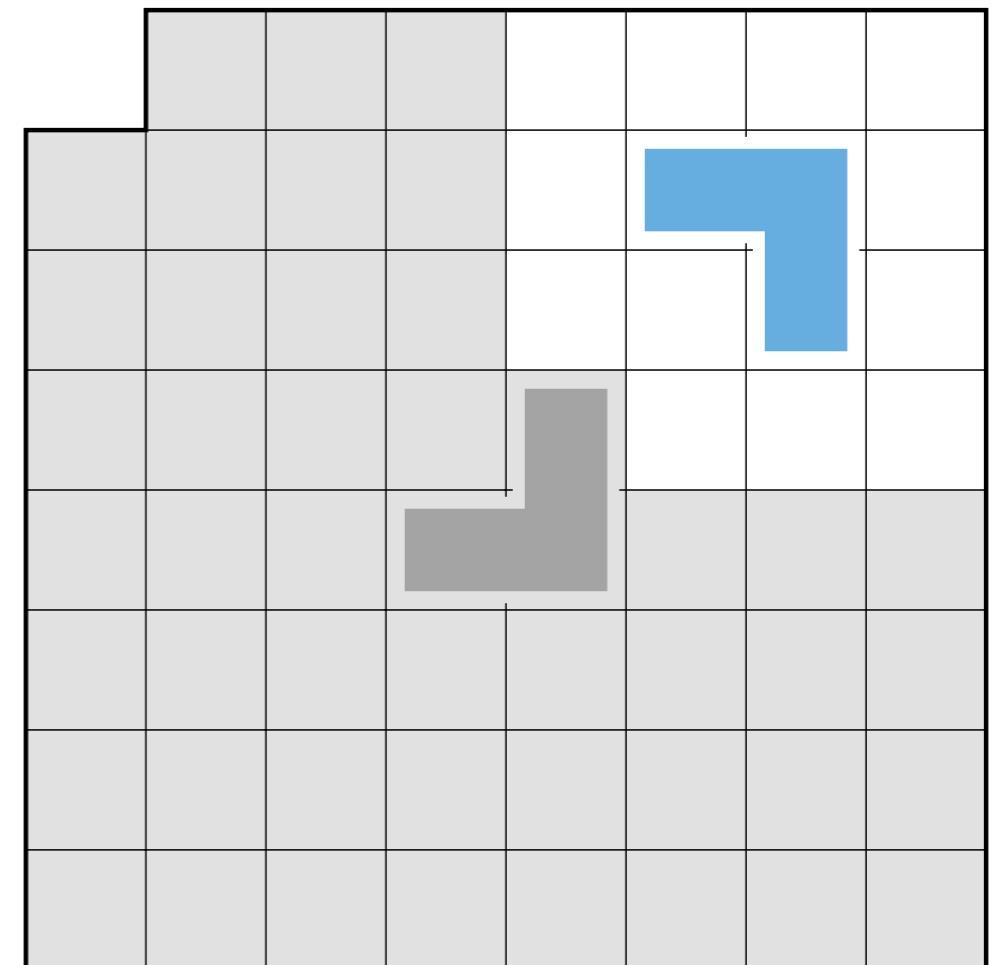
$A, Q = \boxed{\text{■}}, \boxed{\text{□}} \rightarrow \boxed{\text{■}}, \boxed{\text{□}}$



$\text{COVER}(A)$

```
1 place middle L
2 if A is  $2 \times 2$ 
3   return
4 for each quadrant Q
5   COVER(Q)
```

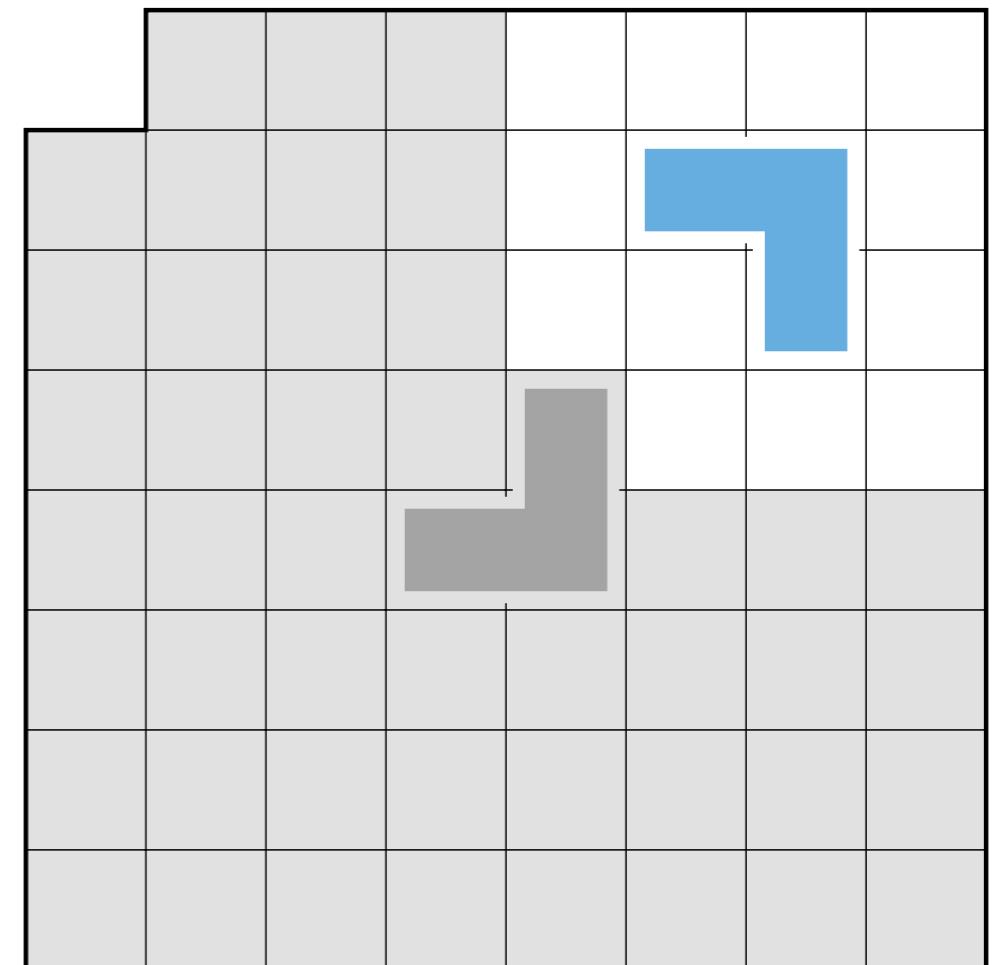
$A, Q = \boxed{}, \boxed{} \rightarrow \boxed{}, \boxed{}$



$\text{COVER}(A)$

```
1 place middle L
2 if A is  $2 \times 2$ 
3   return
4 for each quadrant Q
5   COVER(Q)
```

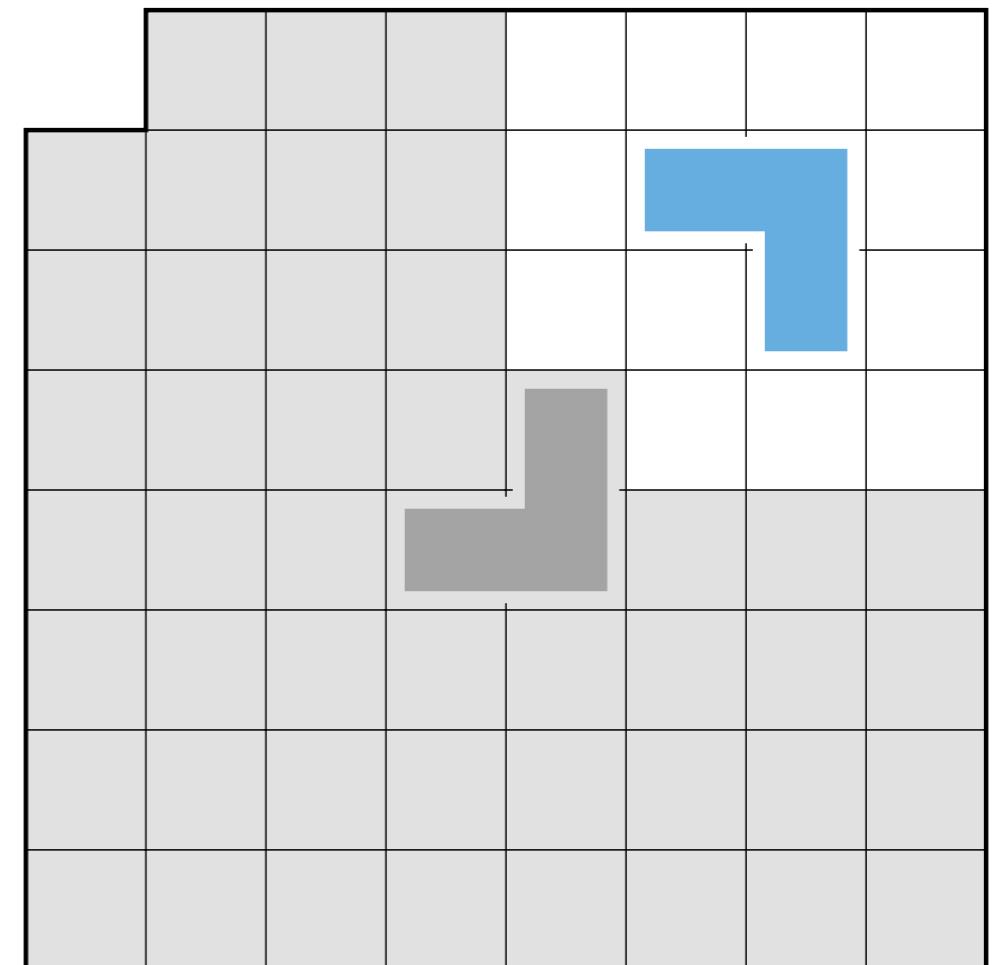
$A, Q = \boxed{}, \boxed{} \rightarrow \boxed{}, \boxed{}$



$\text{COVER}(A)$

```
1 place middle L
2 if A is  $2 \times 2$ 
3   return
4 for each quadrant Q
5   COVER(Q)
```

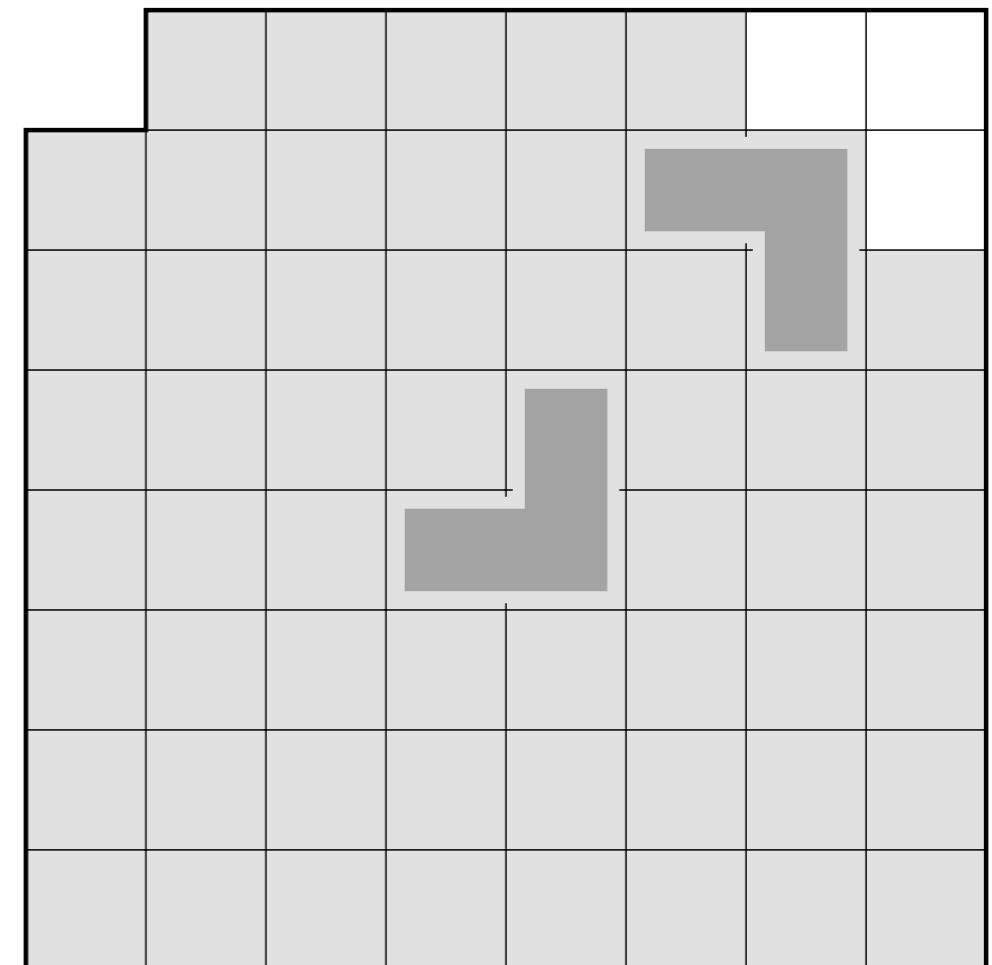
$A, Q = \boxed{}, \boxed{} \rightarrow \boxed{}, \boxed{}$



$\text{COVER}(A)$

```
1 place middle L
2 if A is  $2 \times 2$ 
3   return
4 for each quadrant Q
5   COVER(Q)
```

$A, Q = \boxed{\textcolor{gray}{\square}}, \boxed{\square} \rightarrow \boxed{\square}, \boxed{\textcolor{gray}{\square}} \rightarrow \boxed{\square}, \boxed{\square}$



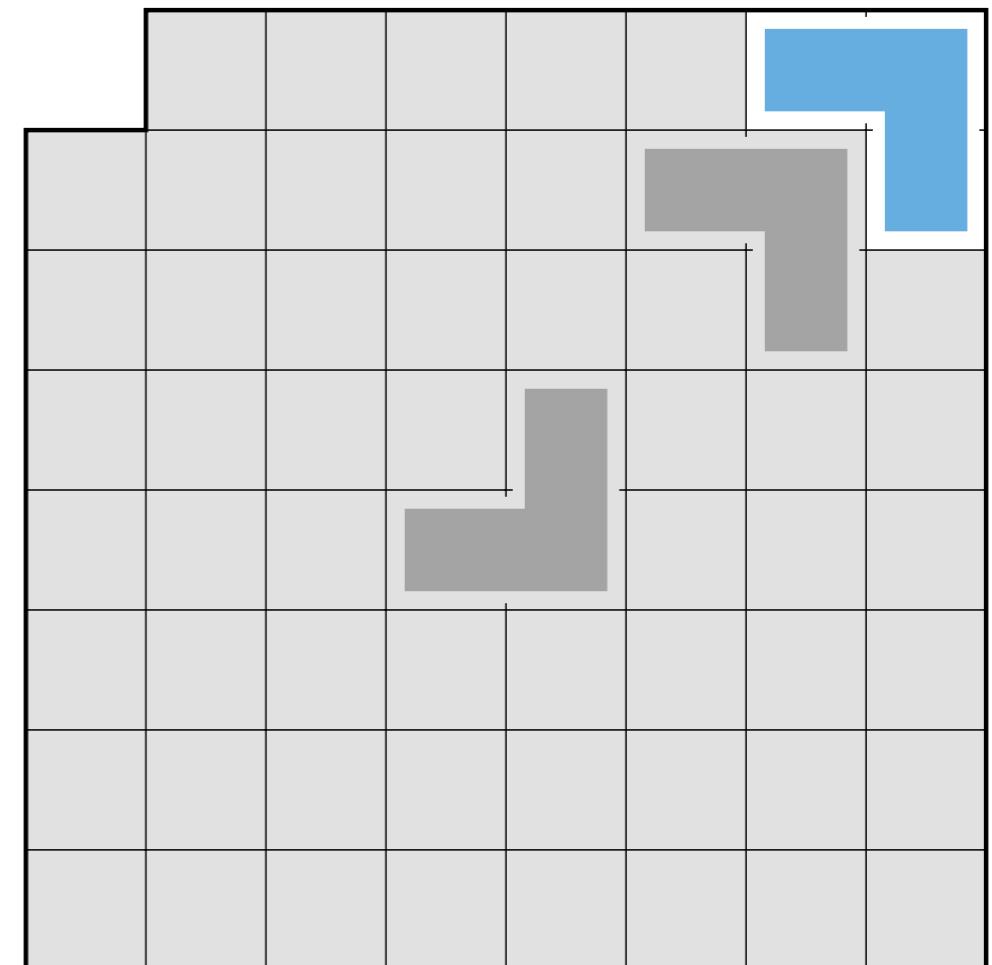
$\text{COVER}(A)$

```

1 place middle L
2 if A is  $2 \times 2$ 
3   return
4 for each quadrant Q
5   COVER(Q)

```

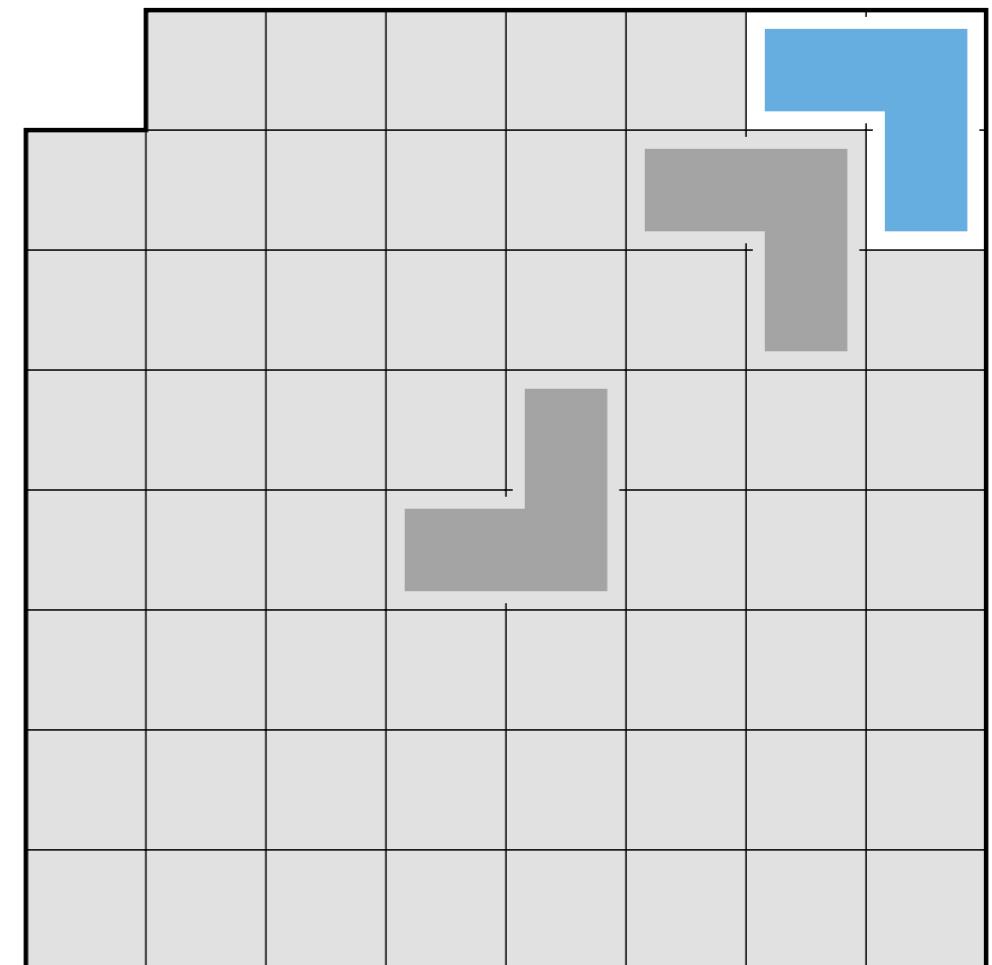
$A, Q = \boxed{}, \boxed{} \rightarrow \boxed{}, \boxed{} \rightarrow \boxed{}, \boxed{}$



COVER(A)

```
1 place middle L
2 if A is  $2 \times 2$ 
3   return
4 for each quadrant Q
5   COVER(Q)
```

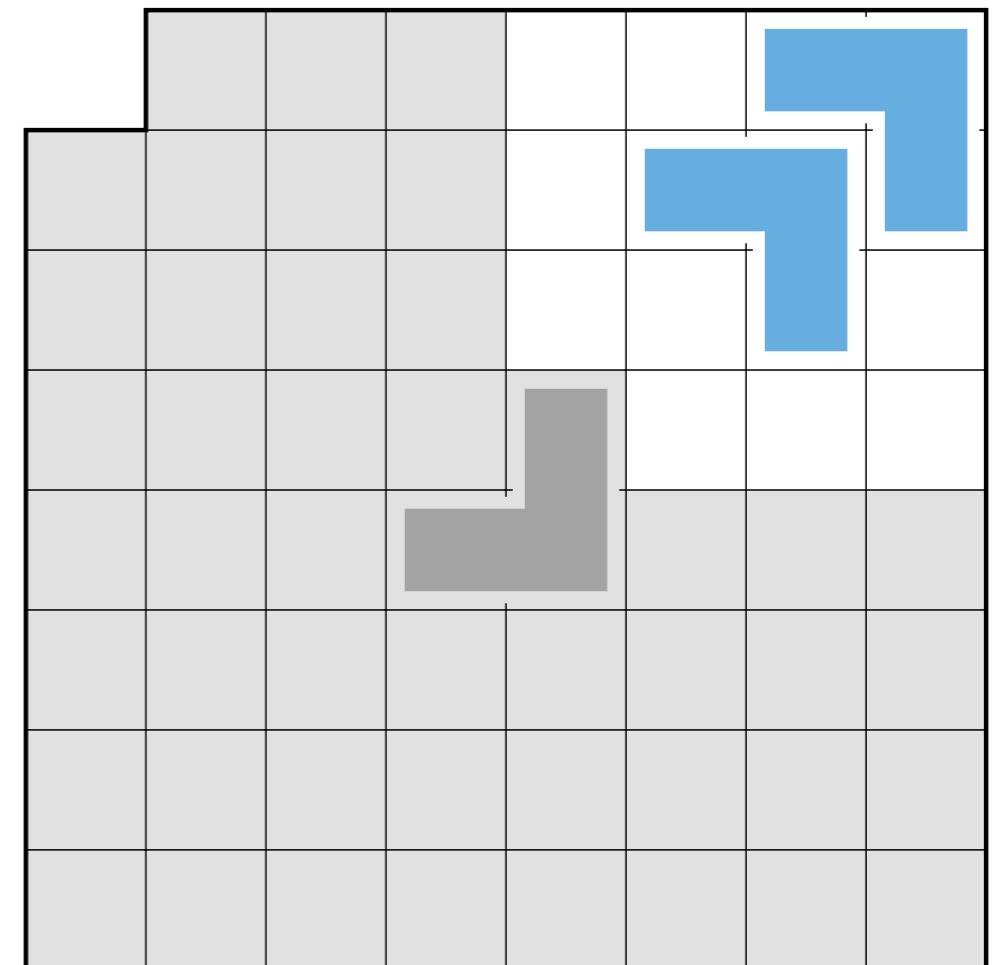
A, Q =  \rightarrow  \rightarrow 



$\text{COVER}(A)$

```
1 place middle L
2 if A is  $2 \times 2$ 
3   return
4 for each quadrant Q
5   COVER(Q)
```

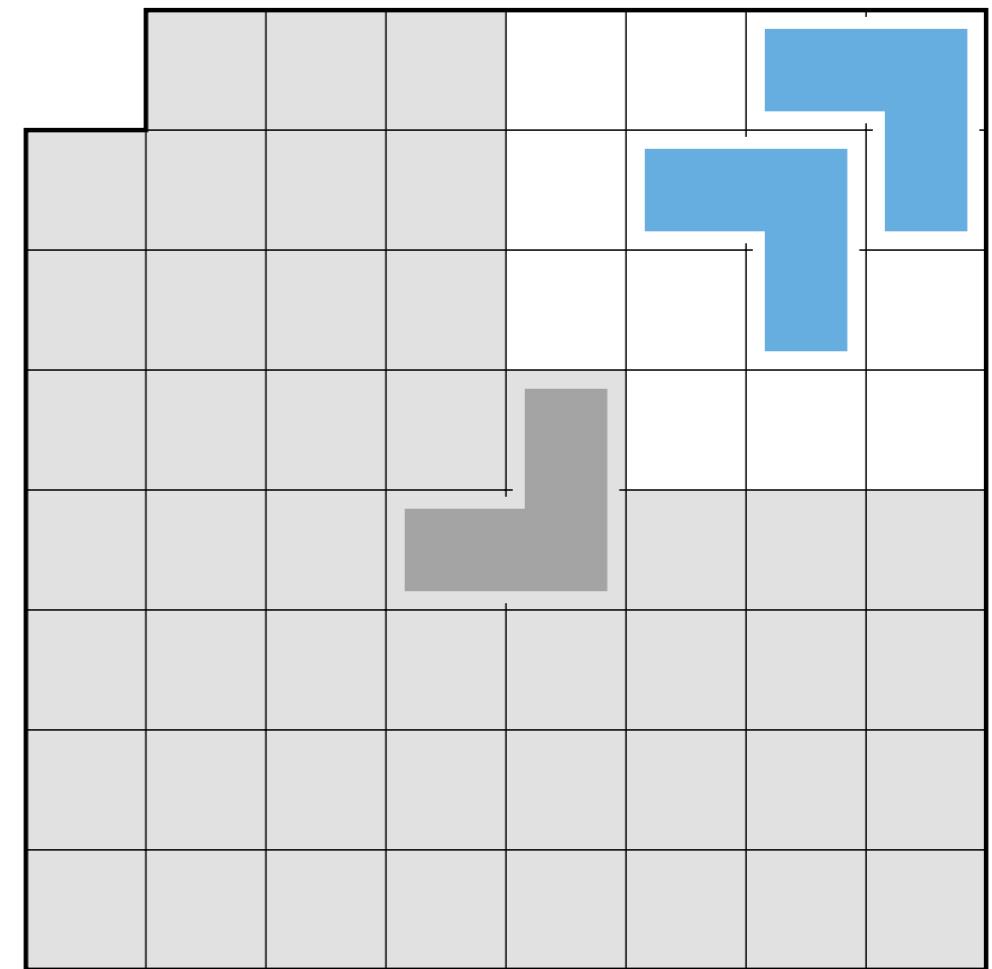
$A, Q = \blacksquare, \square \rightarrow \blacksquare, \square$



$\text{COVER}(A)$

```
1 place middle L
2 if A is  $2 \times 2$ 
3   return
4 for each quadrant Q
5   COVER(Q)
```

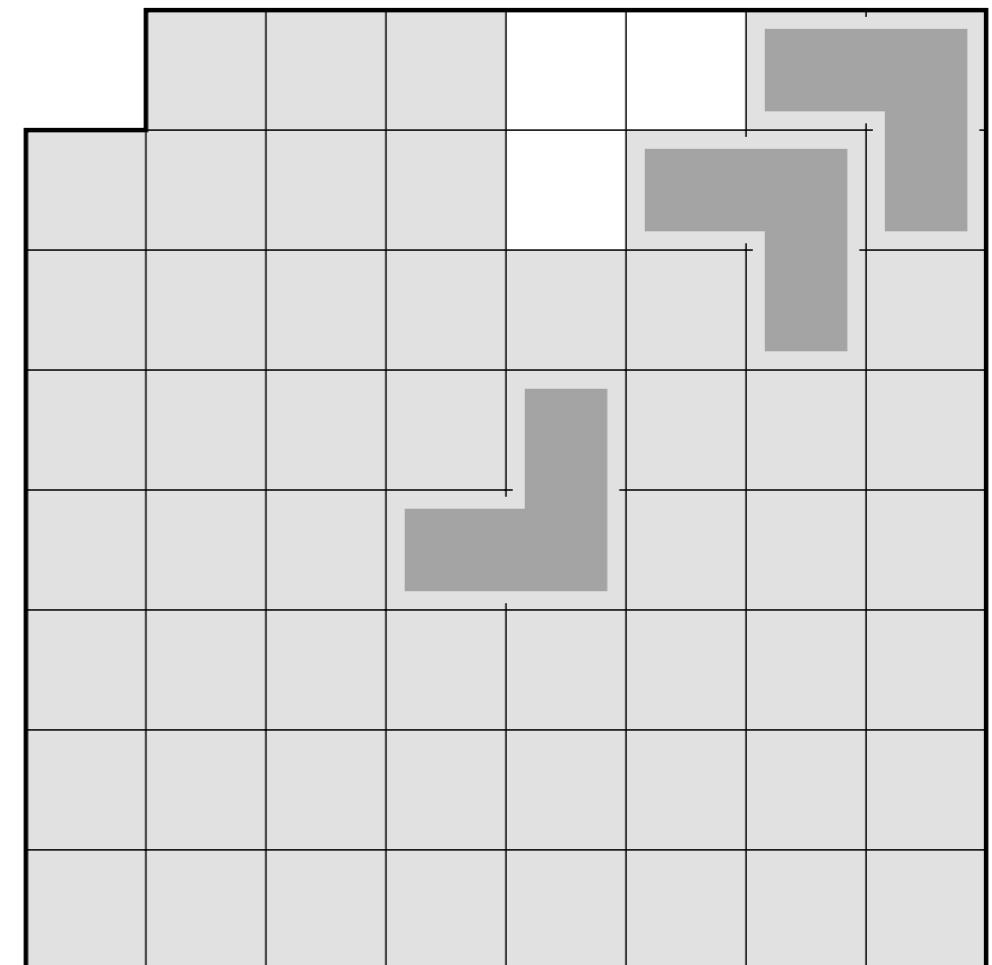
$A, Q = \boxed{}, \boxed{} \rightarrow \boxed{}, \boxed{}$



$\text{COVER}(A)$

```
1 place middle L
2 if A is  $2 \times 2$ 
3   return
4 for each quadrant Q
5   COVER(Q)
```

$A, Q = \boxed{}, \boxed{} \rightarrow \boxed{}, \boxed{} \rightarrow \boxed{}, \boxed{}$



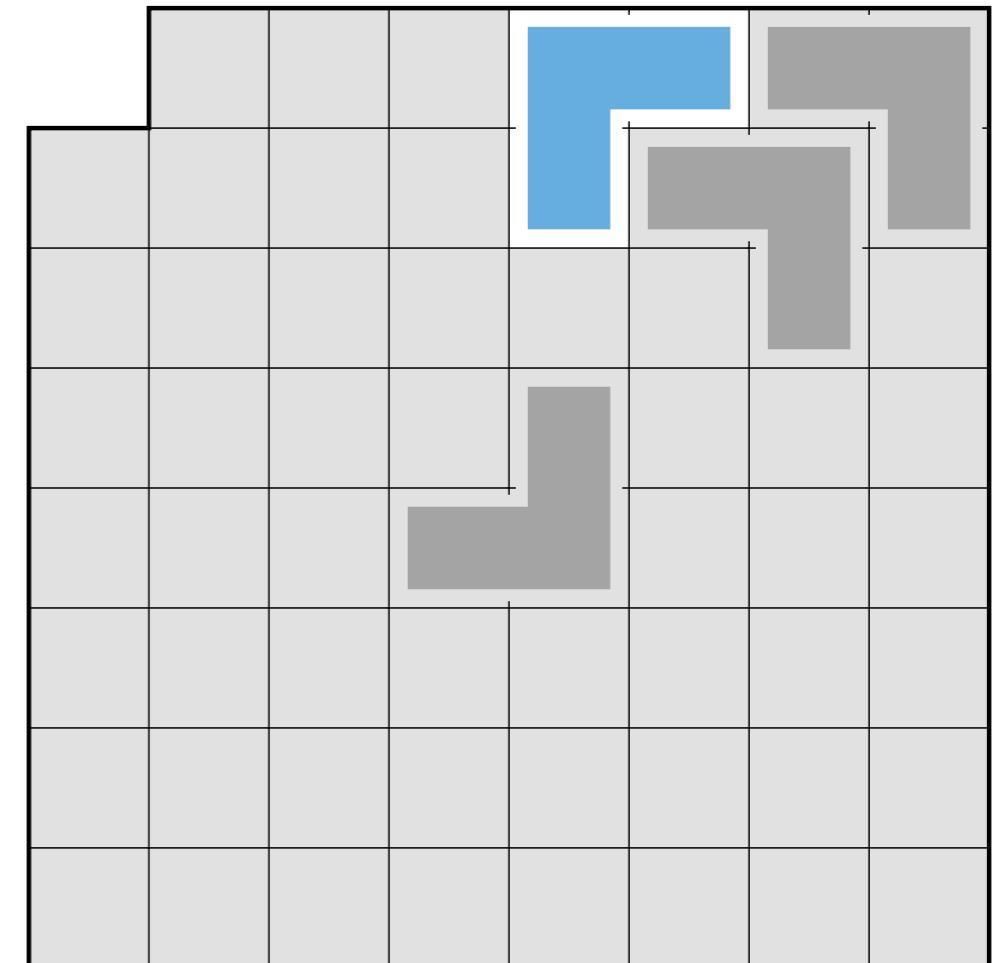
$\text{COVER}(A)$

```

1 place middle L
2 if A is  $2 \times 2$ 
3   return
4 for each quadrant Q
5   COVER(Q)

```

$A, Q = \boxed{}, \boxed{} \rightarrow \boxed{}, \boxed{} \rightarrow \boxed{}, \boxed{}$



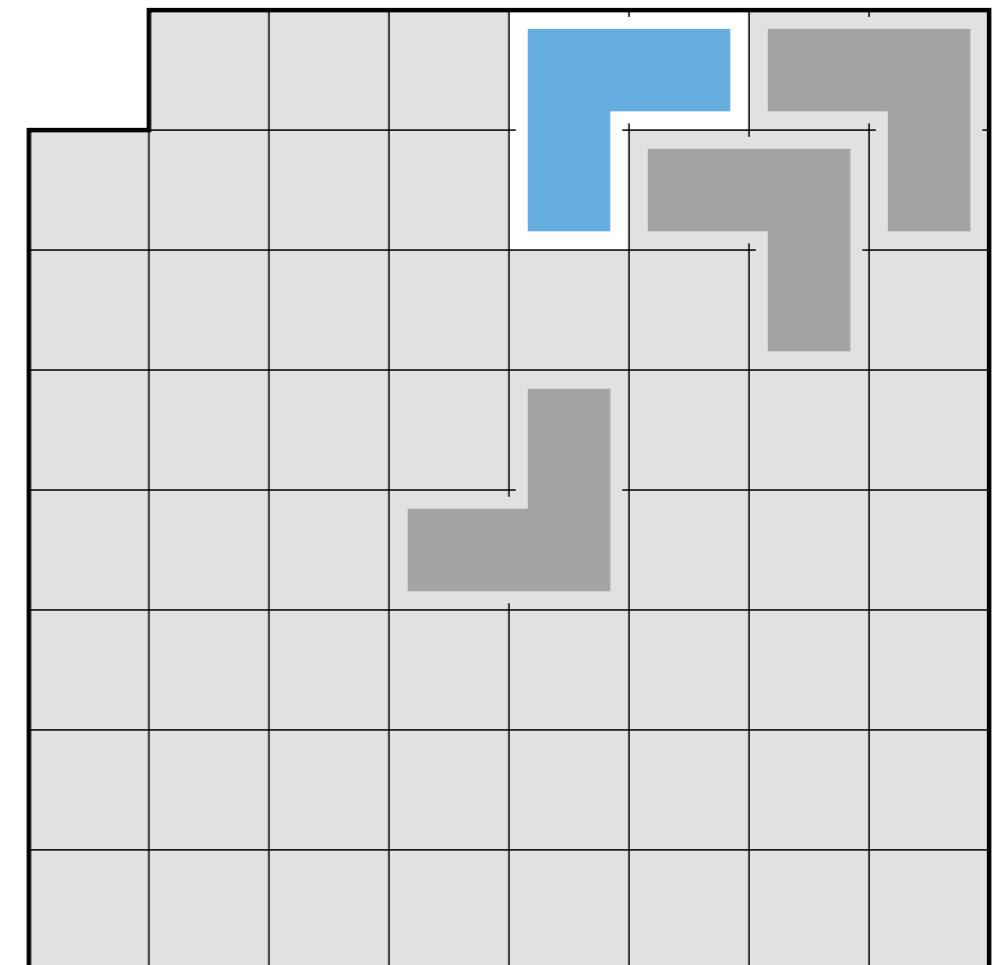
$\text{COVER}(A)$

```

1 place middle L
2 if A is  $2 \times 2$ 
3   return
4 for each quadrant Q
5   COVER(Q)

```

$A, Q = \boxed{}, \boxed{} \rightarrow \boxed{}, \boxed{} \rightarrow \boxed{}, \boxed{}$

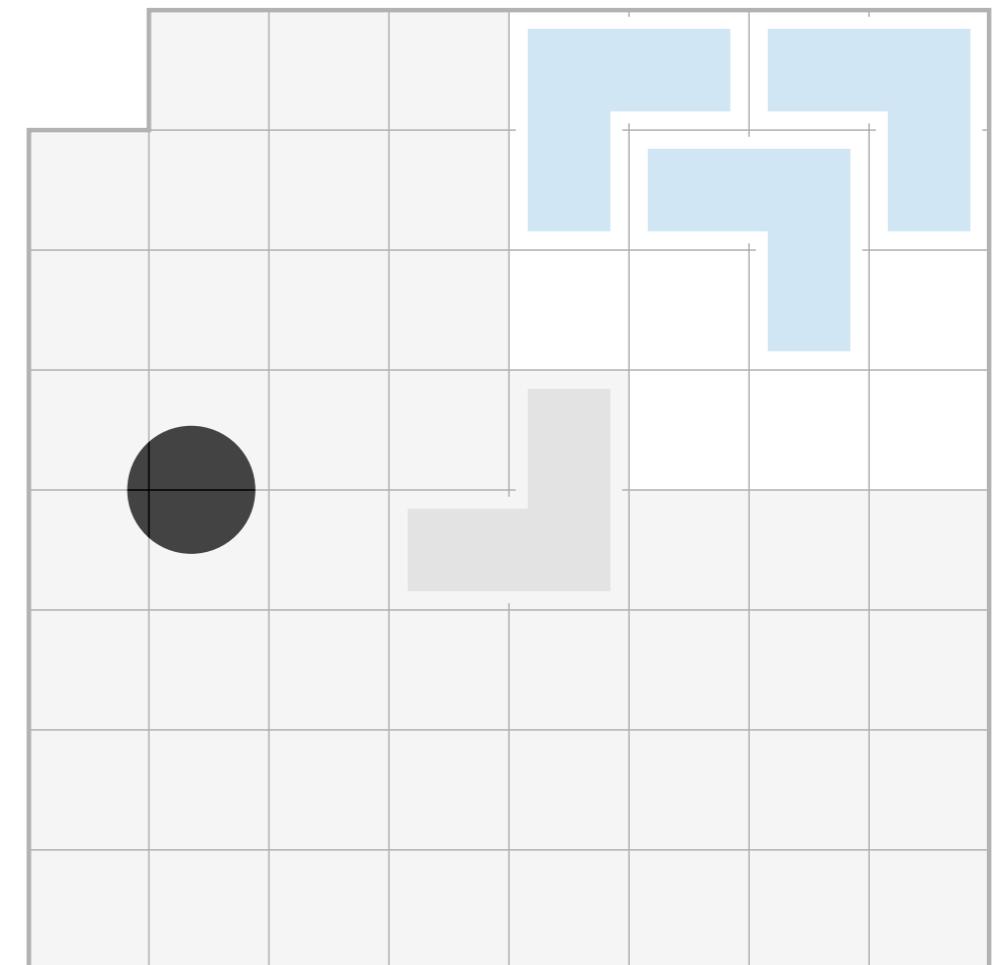


$\text{COVER}(A)$

```

1 place middle L
2 if A is  $2 \times 2$ 
3   return
4 for each quadrant Q
5    $\text{COVER}(Q)$ 

```

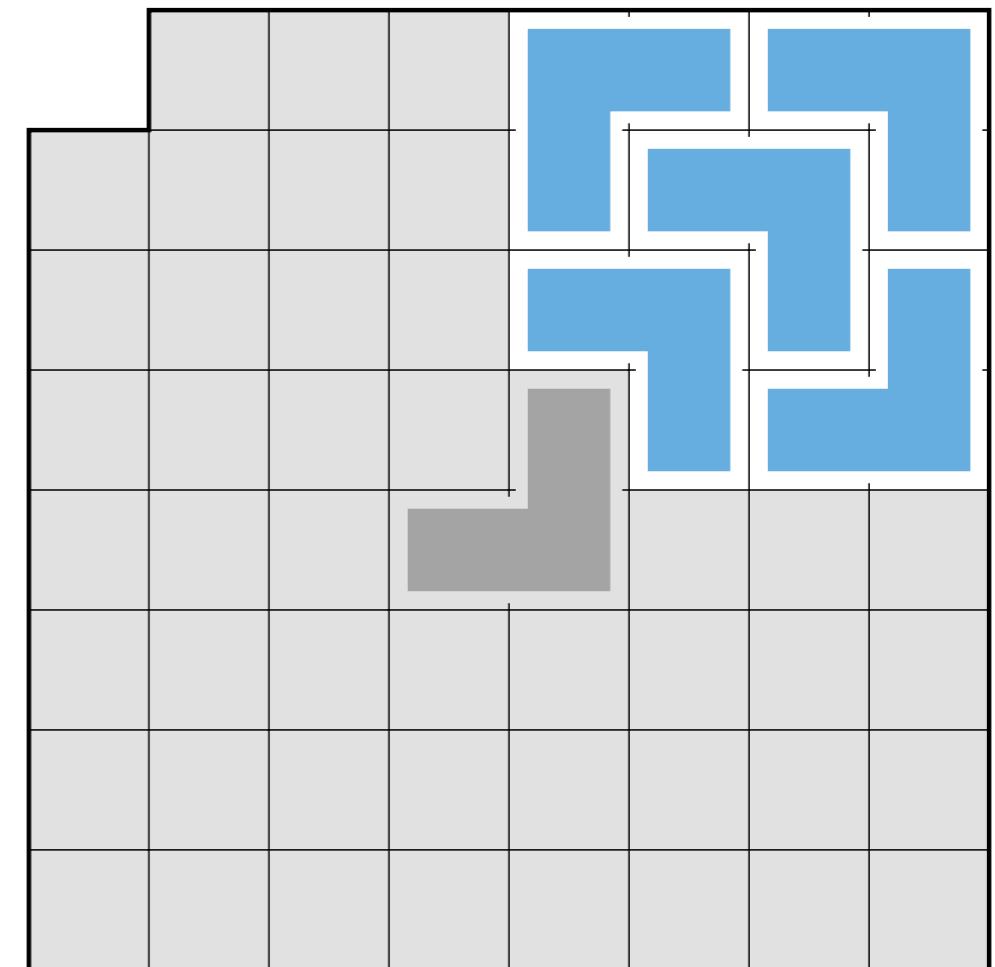


$A, Q = \square, \square \rightarrow \square, \square$

COVER(A)

```
1 place middle L
2 if A is  $2 \times 2$ 
3   return
4 for each quadrant Q
5   COVER(Q)
```

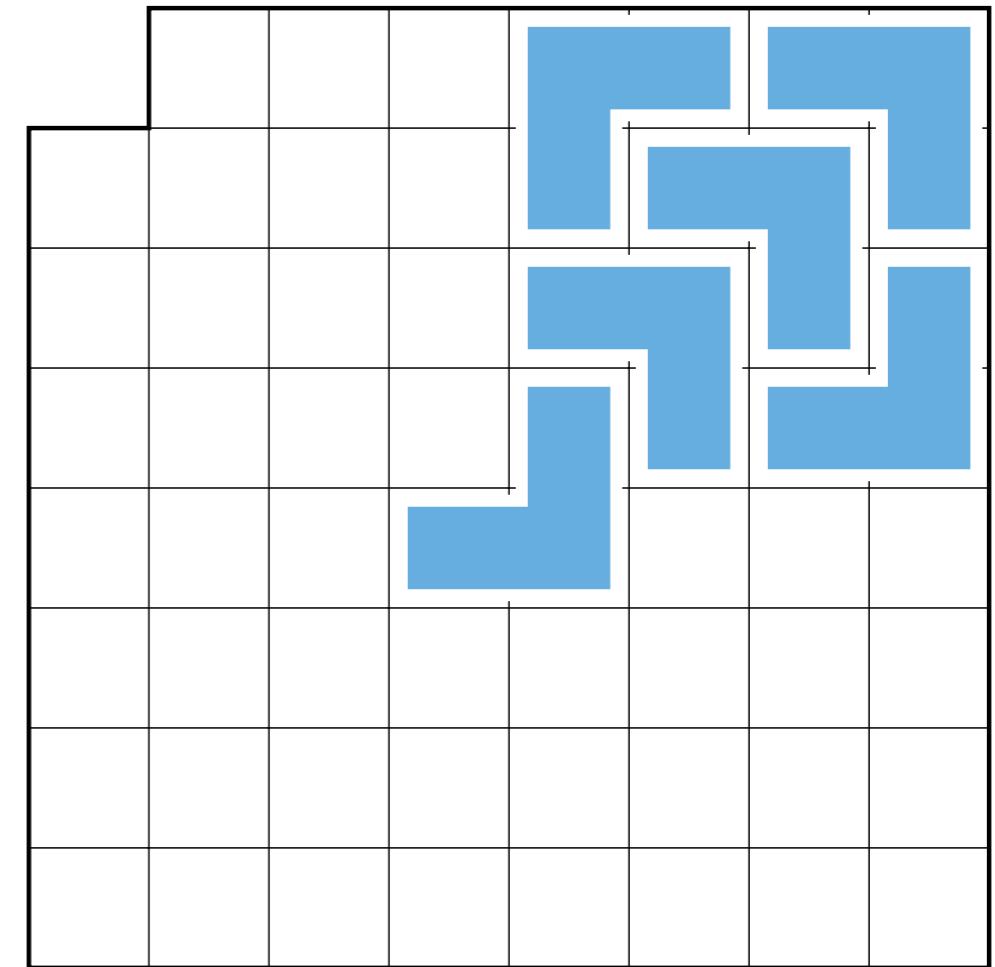
$A, Q = \boxed{}, \boxed{} \rightarrow \boxed{}, \boxed{}$



COVER(A)

```
1 place middle L
2 if A is  $2 \times 2$ 
3   return
4 for each quadrant Q
5   COVER(Q)
```

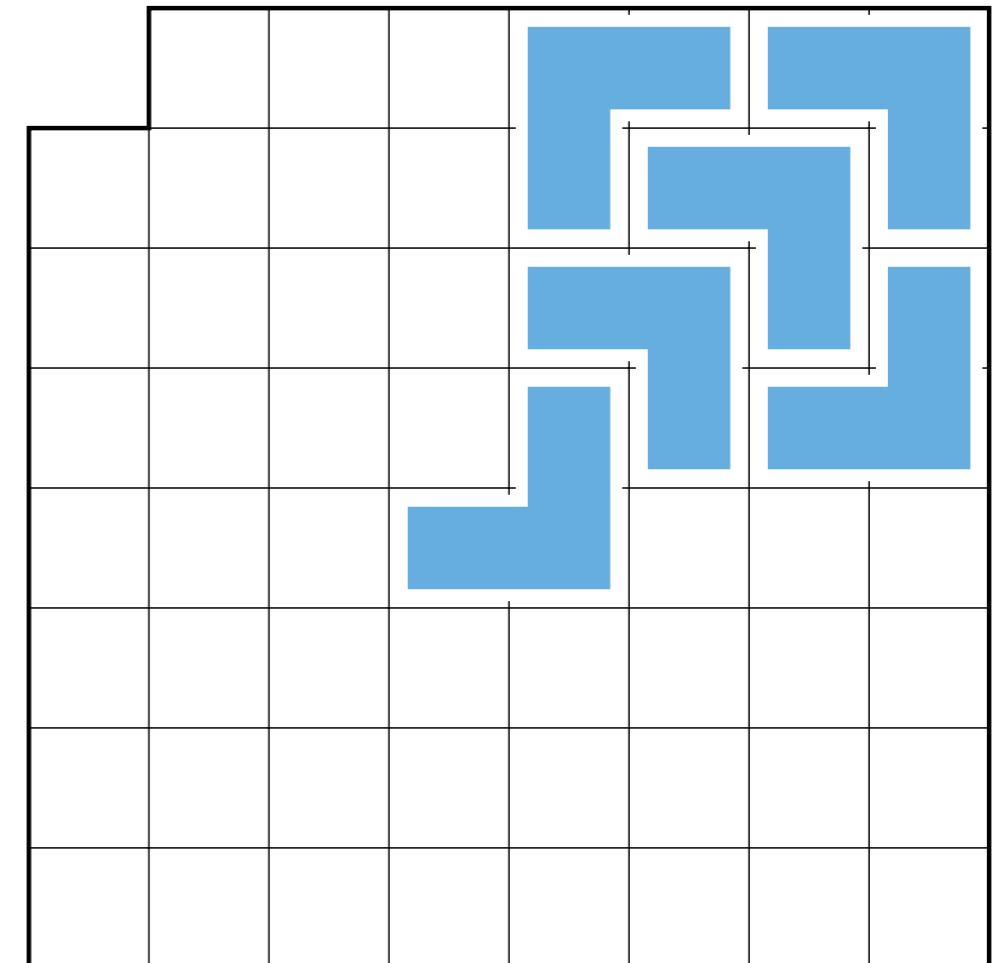
A, Q = , 



$\text{COVER}(A)$

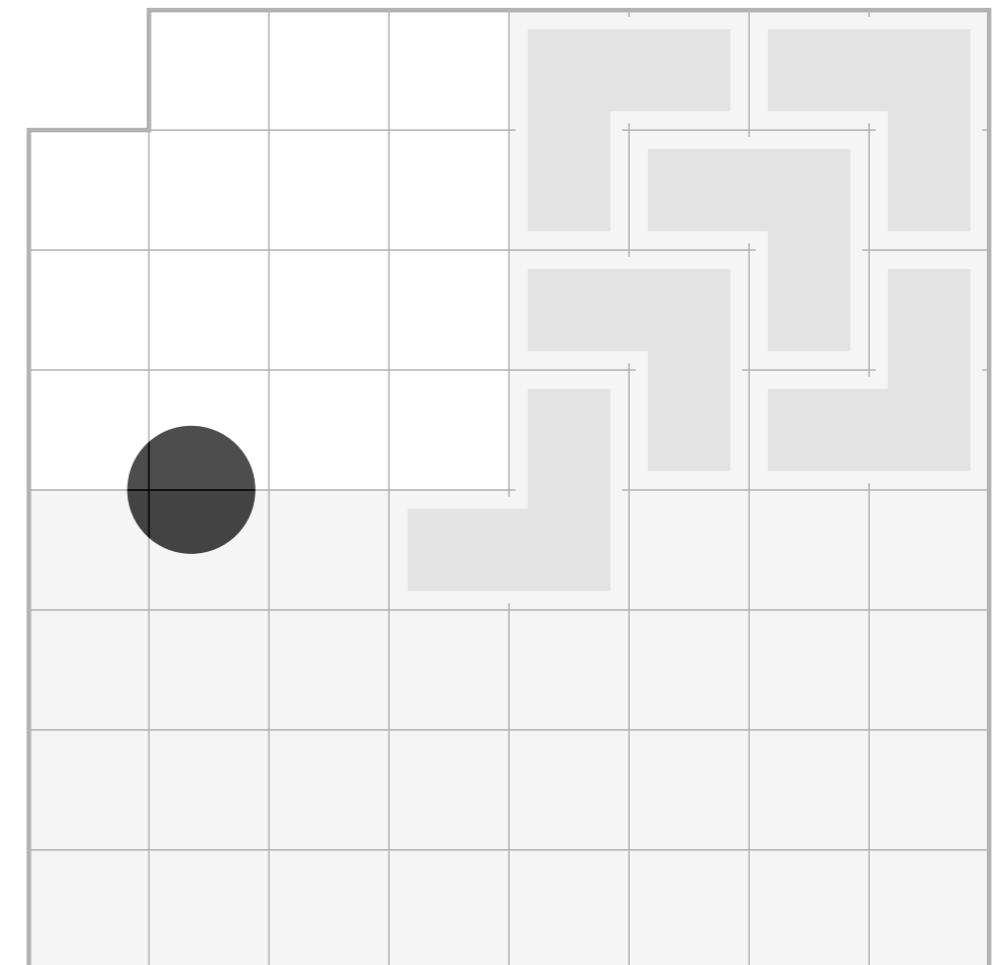
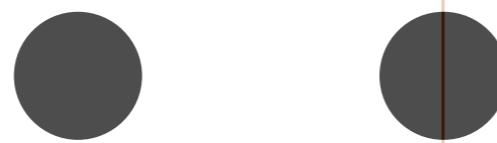
```
1 place middle L
2 if A is  $2 \times 2$ 
3   return
4 for each quadrant Q
5   COVER(Q)
```

$A, Q = \blacksquare, \square$



COVER(A)

```
1 place middle L
2 if A is  $2 \times 2$ 
3   return
4 for each quadrant Q
5   COVER(Q)
```

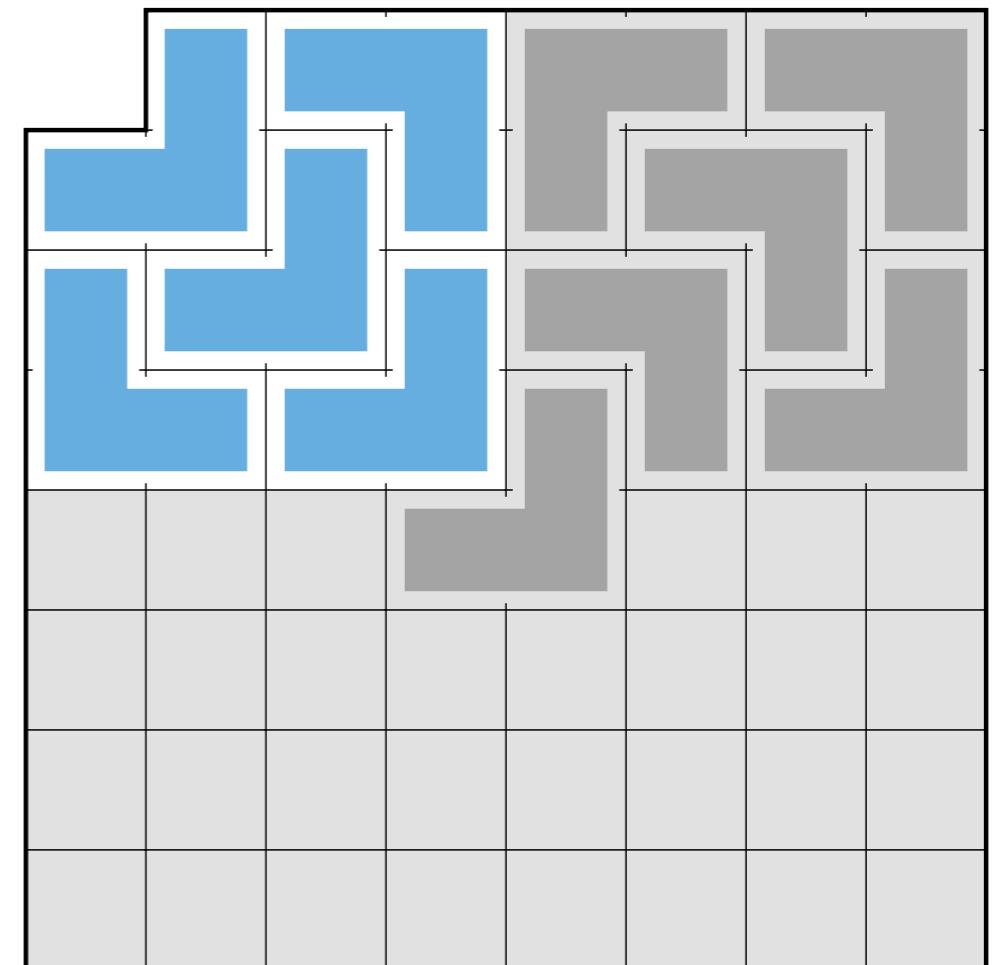


$A, Q = \boxed{}, \boxed{} \rightarrow \boxed{}, \boxed{}$

COVER(A)

```
1 place middle L
2 if A is  $2 \times 2$ 
3   return
4 for each quadrant Q
5   COVER(Q)
```

A, Q = , \rightarrow ,



COVER(A)

1 place middle L

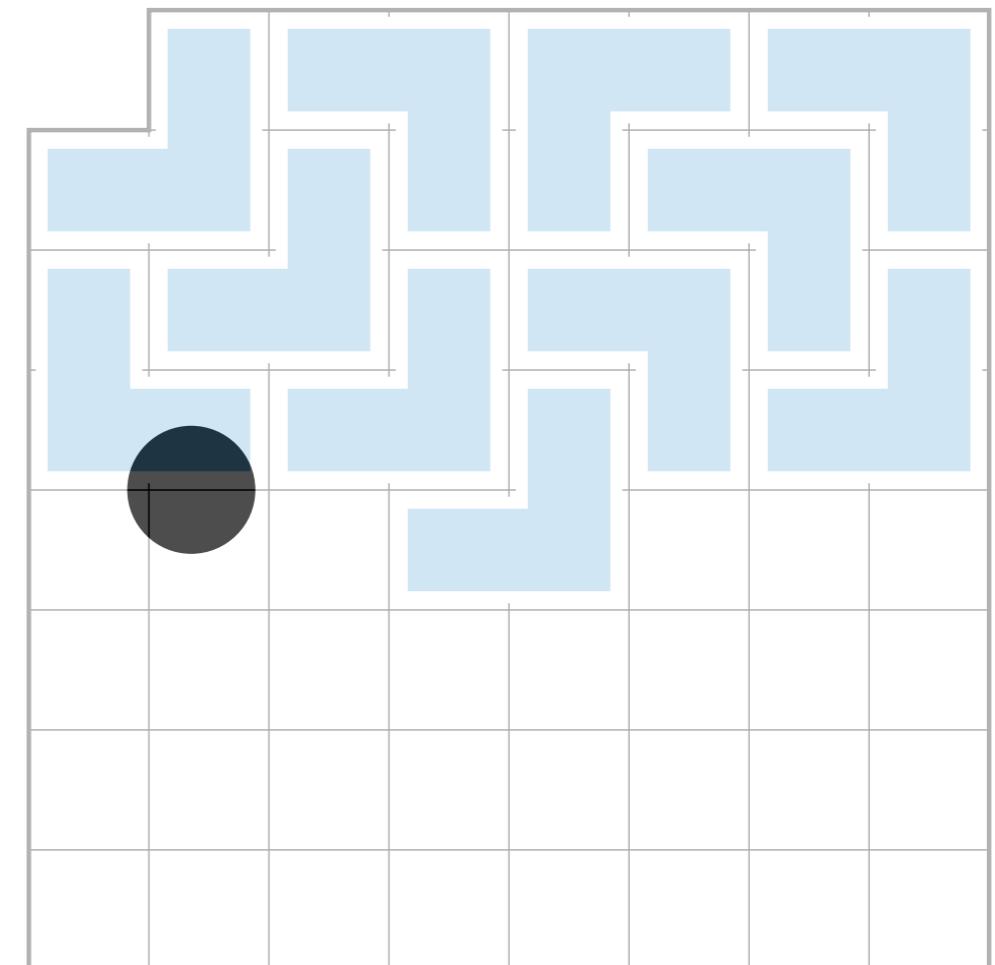
2 if A is 2×2

 return

4 for each quadrant Q

 COVER(Q)

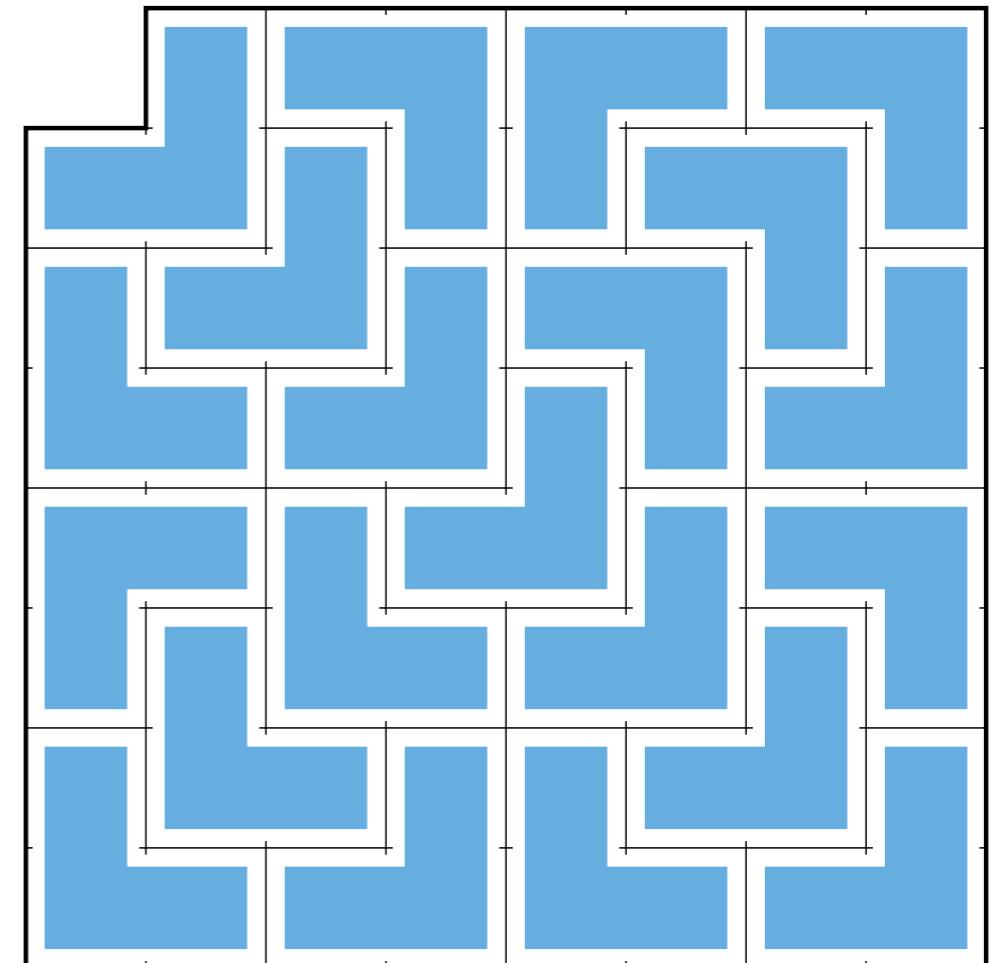
A, Q = , 



COVER(A)

```
1 place middle L
2 if A is  $2 \times 2$ 
3   return
4 for each quadrant Q
5   COVER(Q)
```

A, Q = , 



Kjerneprinsipp

- Bryt ned problemet så det kan løses trinn for trinn
- Fokusér på ett (representativt) trinn

Rekursiv dekomp.

Induksjon over delproblemer

En rekursiv prosedyre kaller seg selv.

Evt.: Den er definert vha. seg selv.

Evt.: Den bruker seg selv som subroutine.

Om du er litt rusten på rekursjon:

Lurt å børste støv av kunnskapene!

Rekursjon ... og induksjon

- Del opp i mindre problemer
- Induktivt premiss: Anta at du kan løse de mindre problemene
- Induksjonstrinn: Konstruer fullstendig løsning ut fra del-løsningene

Induktivt premiss kalles ofte «induksjonshypotese». Ordet «premiss» passer godt til algoritmedesign, siden det også kan bety «betingelse» – og vi beskriver her betingelser for at trinnet vårt skal bli korrekt.

Vi må også sørge for at ting terminerer – at ting blir rett når vi kommer til grunntilfellet (base case) i rekursjonen/ induksjonen.

Løkkeinvarianter

Iterativ dekomponering

Induksjon: Iterativ utgave

- Invariant: Egenskap som ikke endres
- Initialisering: Inv. er sann ved start
- Vedlikehold i hver iterasjon
 - Induktivt premiss: Antatt sann først
 - Induksjonstrinn: Vist sann etterpå
- Terminering: Vis at løkka stopper

Nok en gang ...

- Anta at du kan løse mindre problemer
- Bruk dette til å lage en løsning
- Invariant: «Det har gått bra så langt»
- Dette kan vi anta, og «dra med oss»

Dette er selvfølgelig ikke den eneste varianten vi kan velge – men den enkleste til å begynne med.

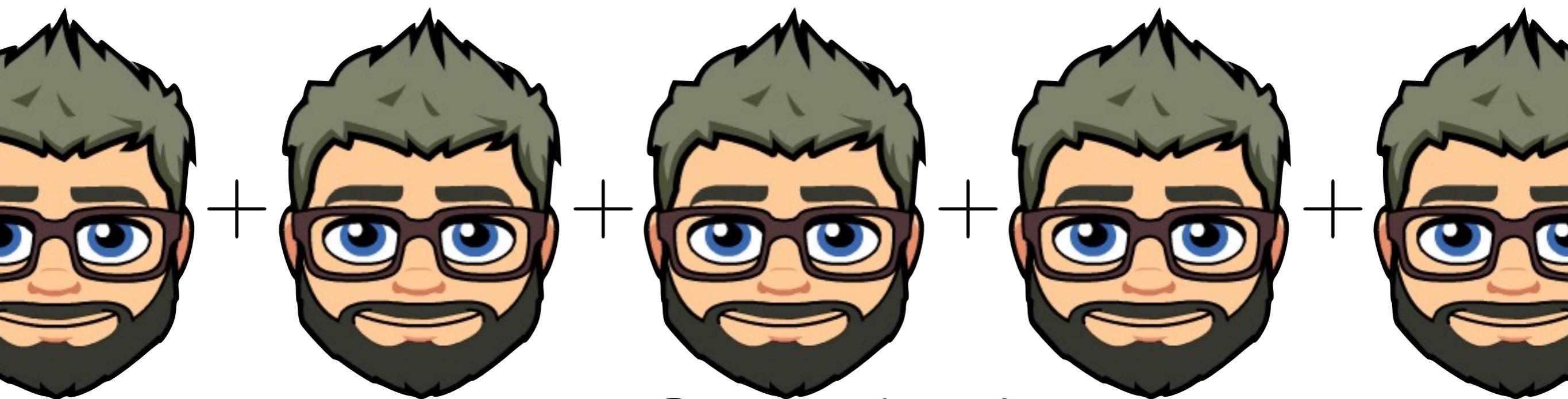


I begge tilfeller

- Anta at du kan løse mindre instanser
- Bruk dette til å finne en løsning

Vi kan tenke på det induktive premiss som forarbeide. Arbeidet frem til ett trinn blir forarbeide for det neste!

**La oss bruke dekomponering på et par
eksempelproblemer ...**



4.5

Eksempel: Sum

Tolkning

T

Hva er relasjonen mellom input og output?

Analyse

A

Del en vilkårlig instans i delinstanser.

Syntese

S

Bygg løsning av hypotetiske delløsninger.

Sum Rekursiv

Dekomponering

- Vi vil summere elementene i en tabell
- Rekursjon: Summér alle unntatt siste
- Grunntilfelle: Tom sum er null
- Induktivt premiss: Summen er rett
- Induksjonstrinn: Legg til siste element

SUM(A, i)

Summen av A[1 .. i]

SUM(A, i)
1 if $i < 1$

Grunntilfelle

```
SUM(A, i)
1  if i < 1
2      return 0
```

Summen av en tom sekvens

```
SUM(A, i)
1  if i < 1
2      return 0
3  tmp = SUM(A, i - 1)
```

Induksjonshypotese: $\text{SUM}(A, i - 1)$ er summen av $A[1 \dots i - 1]$

```
SUM(A, i)
1  if i < 1
2      return 0
3  tmp = SUM(A, i - 1)
4  return tmp + A[i]
```

Induktivt trinn: Sørg for at $\text{SUM}(A, i)$ er summen av $A[1..i]$

rekursiv sum

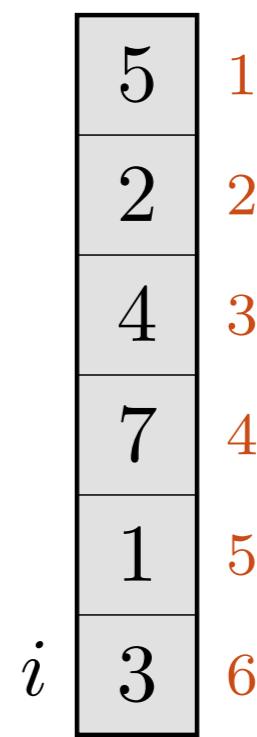
```
SUM(A, i)
1  if  $i < 1$ 
2      return 0
3  tmp = SUM(A,  $i - 1$ )
4  return tmp + A[i]
```

$tmp = -$

5	1
2	2
4	3
7	4
1	5
3	6

rekursiv sum

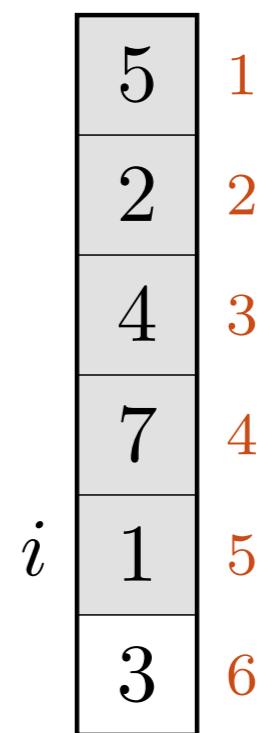
```
SUM(A, i)
1  if i < 1
2      return 0
3  tmp = SUM(A, i - 1)
4  return tmp + A[i]
```



$tmp = -$

rekursiv sum

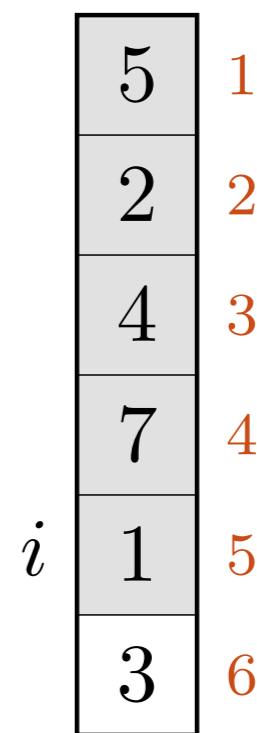
```
SUM(A, i)
1  if  $i < 1$ 
2      return 0
3  tmp = SUM(A,  $i - 1$ )
4  return tmp + A[i]
```



$tmp = \dots$

```
SUM(A, i)
1  if i < 1
2      return 0
3  tmp = SUM(A, i - 1)
4  return tmp + A[i]
```

tmp = ->-



```
SUM(A, i)
1  if  $i < 1$ 
2      return 0
3  tmp = SUM(A,  $i - 1$ )
4  return tmp + A[i]
```

5	1
2	2
4	3
7	4
1	5
3	6

$tmp = \dots \rightarrow \dots \rightarrow \dots$

rekursiv sum

```
SUM(A, i)
1  if i < 1
2      return 0
3  tmp = SUM(A, i - 1)
4  return tmp + A[i]
```

5	1
2	2
4	3
i	4
1	5
3	6

tmp = ->->-

```
SUM(A, i)
1  if  $i < 1$ 
2      return 0
3  tmp = SUM(A,  $i - 1$ )
4  return tmp + A[i]
```

5	1
2	2
4	3
7	4
1	5
3	6

$tmp = \dots \rightarrow \dots \rightarrow \dots \rightarrow \dots$

```
SUM(A, i)
1  if i < 1
2      return 0
3  tmp = SUM(A, i - 1)
4  return tmp + A[i]
```

5	1
2	2
4	3
7	4
1	5
3	6

tmp = ->->->->-

```
SUM(A, i)
1  if  $i < 1$ 
2      return 0
3  tmp = SUM(A,  $i - 1$ )
4  return tmp + A[i]
```

i	5	1
	2	2
	4	3
	7	4
	1	5
	3	6

$tmp = \dots + A[1] + A[2] + A[3] + A[4] + A[5]$

```
SUM(A, i)
1  if i < 1
2      return 0
3  tmp = SUM(A, i - 1)
4  return tmp + A[i]
```

i	5	1
	2	2
	4	3
	7	4
	1	5
	3	6

$tmp = \dots + A[1] + A[2] + A[3] + A[4] + A[5]$

```
SUM(A, i)
1  if  $i < 1$ 
2      return 0
3  tmp = SUM(A,  $i - 1$ )
4  return tmp + A[i]
```

i	5	1
	2	2
	4	3
	7	4
	1	5
	3	6

$tmp = \dots + A[1] + A[2] + A[3] + A[4] + A[5]$

```
SUM(A, i)
1  if  $i < 1$ 
2      return 0
3   $tmp = \text{SUM}(A, i - 1)$ 
4  return  $tmp + A[i]$ 
```

i	5	1
	2	2
	4	3
	7	4
	1	5
	3	6

$tmp = - \rangle - \rangle - \rangle - \rangle - \rangle -$

SUM(A, i)

```
1 if  $i < 1$ 
2     return 0
3 tmp = SUM(A, i - 1)
4 return tmp + A[i]
```

5	1
2	2
4	3
7	4
1	5
3	6

$tmp = \dots + A[0] + A[1] + A[2] + A[3] + A[4] + A[5]$

```
SUM(A, i)
1  if i < 1
2      return 0
3  tmp = SUM(A, i - 1)
4  return tmp + A[i]
```

5	1
2	2
4	3
7	4
1	5
3	6

$tmp = \dots + A[0] + A[1] + A[2] + A[3] + A[4] + A[5] + A[6] + A[7] + A[8]$

SUM(A, i)

```
1 if  $i < 1$ 
2     return 0
3  $tmp = \text{SUM}(A, i - 1)$ 
4 return  $tmp + A[i]$ 
```

→ 0

$tmp = - \rangle - \rangle - \rangle - \rangle - \rangle - \rangle -$

5	1
2	2
4	3
7	4
1	5
3	6

```
SUM(A, i)
1  if  $i < 1$ 
2      return 0
3   $tmp = \text{SUM}(A, i - 1)$ 
4  return  $tmp + A[i]$ 
```

i	5	1
	2	2
	4	3
	7	4
	1	5
	3	6

$tmp = \dots \rightarrow 0$

```
SUM(A, i)
1  if  $i < 1$ 
2      return 0
3  tmp = SUM(A,  $i - 1$ )
4  return tmp + A[i]
```

→ 5

$tmp = \dots \rightarrow \dots \rightarrow \dots \rightarrow \dots \rightarrow 0$

i	5	1
	2	2
	4	3
	7	4
	1	5
	3	6

rekursiv sum

```
SUM(A, i)
1  if i < 1
2      return 0
3  tmp = SUM(A, i - 1)
4  return tmp + A[i]
```

i	5	1
	2	2
	4	3
	7	4
	1	5
	3	6

$tmp = \dots \rightarrow \dots \rightarrow \dots \rightarrow 5$

SUM(A, i)

```
1 if  $i < 1$ 
2     return 0
3  $tmp = \text{SUM}(A, i - 1)$ 
4 return  $tmp + A[i]$ 
```

→ 7

$tmp = \dots \rightarrow \dots \rightarrow \dots \rightarrow 5$

i	5	1
	2	2
	4	3
	7	4
	1	5
	3	6

```
SUM(A, i)
1  if  $i < 1$ 
2      return 0
3   $tmp = \text{SUM}(A, i - 1)$ 
4  return  $tmp + A[i]$ 
```

5	1
2	2
4	3
7	4
1	5
3	6

$tmp = \dots \rightarrow \dots \rightarrow \dots \rightarrow 7$

SUM(A, i)

```
1 if  $i < 1$ 
2     return 0
3 tmp = SUM(A,  $i - 1$ )
4 return tmp + A[i]
```

→ 11

tmp = → → → → 7

5	1
2	2
4	3
7	4
1	5
3	6

```
SUM(A, i)
1  if  $i < 1$ 
2      return 0
3   $tmp = \text{SUM}(A, i - 1)$ 
4  return  $tmp + A[i]$ 
```

$tmp = \dots \rightarrow \dots \rightarrow 11$

5	1
2	2
4	3
i	4
7	5
1	5
3	6

SUM(A, i)

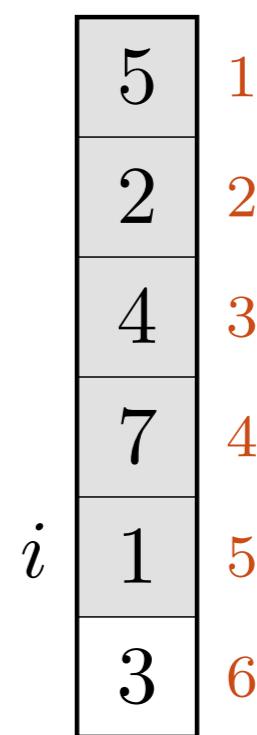
```
1 if  $i < 1$ 
2     return 0
3 tmp = SUM(A,  $i - 1$ )
4 return tmp + A[i]
```

→ 18

tmp = → → 11

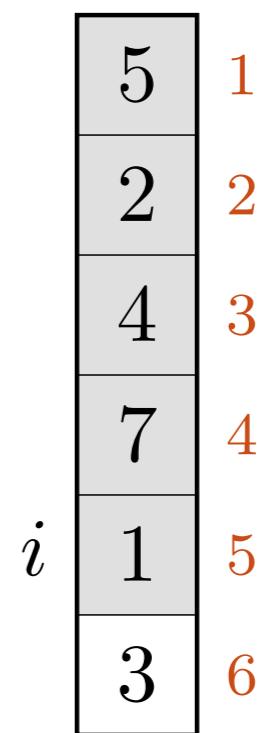
5	1
2	2
4	3
i	4
1	5
3	6

```
SUM(A, i)
1  if i < 1
2      return 0
3  tmp = SUM(A, i - 1)
4  return tmp + A[i]
```



$tmp = \dots \rightarrow 18$

```
SUM(A, i)
1  if i < 1
2      return 0
3  tmp = SUM(A, i - 1)
4  return tmp + A[i]
→ 19
```



$tmp = \rightarrow 18$

```
SUM(A, i)
1  if i < 1
2      return 0
3  tmp = SUM(A, i - 1)
4  return tmp + A[i]
```

5	1
2	2
4	3
7	4
1	5
3	6

i

$tmp = 19$

```
SUM(A, i)
1  if  $i < 1$ 
2      return 0
3   $tmp = \text{SUM}(A, i - 1)$ 
4  return  $tmp + A[i]$ 
→ 22
```

$tmp = 19$

5	1
2	2
4	3
7	4
1	5
3	6

Sum

Iterativ

Dekomponering

- Invariant: Vi har summert rett så langt
- Initialisering: Tom sum er null
- Vedlikehold:
 - Induktivt premiss: Summen er rett før iterasjonen
 - Induksjonstrinn: Legg til neste element
- Terminering: Til slutt har vi summert alle

iterativ sum

SUM(A)

Vi vil summere elementene i A

SUM(A)
1 $res = 0$

Initialisering: Sum så langt er 0

```
SUM(A)
1   res = 0
2   for j = 1 to A.length
```

Induksjonshypotese: *res* er summen av $A[1..j - 1]$

```
SUM(A)
1   res = 0
2   for j = 1 to A.length
3       res = res + A[j]
```

Induktivt trinn: Sørg for at res er summen av $A[1..j]$

```
SUM(A)
1  res = 0
2  for j = 1 to A.length
3      res = res + A[j]
4  return res
```

Terminering: $j = n$, så res er summen av $A[1..n]$

iterativ sum

SUM(A)

```
1  res = 0
2  for j = 1 to A.length
3      res = res + A[j]
4  return res
```

res = -

5	1
2	2
4	3
7	4
1	5
3	6

iterativ sum

SUM(A)

```
1  res = 0
2  for j = 1 to A.length
3      res = res + A[j]
4  return res
```

res = 0

5	1
2	2
4	3
7	4
1	5
3	6

iterativ sum

```
SUM(A)
1  res = 0
2  for j = 1 to A.length
3      res = res + A[j]
4  return res
```

res = 0

<i>j</i>	5	1
	2	2
	4	3
	7	4
	1	5
	3	6

iterativ sum

SUM(A)

```
1  res = 0
2  for j = 1 to A.length
3      res = res + A[j]
4  return res
```

res = 5

j	5	1
	2	2
	4	3
	7	4
	1	5
	3	6

iterativ sum

```
SUM(A)
1  res = 0
2  for j = 1 to A.length
3      res = res + A[j]
4  return res
```

res = 5

j	5	1
	2	2
	4	3
	7	4
	1	5
	3	6

iterativ sum

SUM(A)

```
1  res = 0
2  for j = 1 to A.length
3      res = res + A[j]
4  return res
```

res = 7

j	5	1
	2	2
	4	3
	7	4
	1	5
	3	6

iterativ sum

```
SUM(A)
1  res = 0
2  for j = 1 to A.length
3      res = res + A[j]
4  return res
```

res = 7

5	1
2	2
4	3
7	4
1	5
3	6

iterativ sum

SUM(A)

```
1  res = 0
2  for j = 1 to A.length
3      res = res + A[j]
4  return res
```

res = 11

5	1
2	2
4	3
7	4
1	5
3	6

iterativ sum

```
SUM(A)
1  res = 0
2  for j = 1 to A.length
3      res = res + A[j]
4  return res
```

res = 11

5	1
2	2
4	3
7	4
1	5
3	6

iterativ sum

SUM(A)

```
1  res = 0
2  for j = 1 to A.length
3      res = res + A[j]
4  return res
```

res = 18

5	1
2	2
4	3
7	4
1	5
3	6

iterativ sum

SUM(A)

```
1  res = 0
2  for j = 1 to A.length
3      res = res + A[j]
4  return res
```

res = 18

5	1
2	2
4	3
7	4
1	5
3	6

j

iterativ sum

SUM(A)

```
1  res = 0
2  for j = 1 to A.length
3      res = res + A[j]
4  return res
```

res = 19

5	1
2	2
4	3
7	4
1	5
3	6

j

iterativ sum

```
SUM(A)
1  res = 0
2  for j = 1 to A.length
3      res = res + A[j]
4  return res
```

res = 19

5	1
2	2
4	3
7	4
1	5
3	6

j

iterativ sum

SUM(A)

```
1  res = 0
2  for j = 1 to A.length
3      res = res + A[j]
4  return res
```

res = 22

5	1
2	2
4	3
7	4
1	5
3	6

iterativ sum

SUM(A)

```
1  res = 0
2  for j = 1 to A.length
3      res = res + A[j]
4  return res
```

→ 22

res = 22

5	1
2	2
4	3
7	4
1	5
3	6

Rek. vs. Iter.

Et spørsmål om perspektiv

Rekursjon og iterasjon er i all hovedsak ekvivalente ting. Her har vi en sammenligning av hvordan de to variantene oppfører seg; for begge to er det induktive premisset at den grå biten er summert allerede. I den rekursive varianten gjør vi det rekursivt før vi legger til det siste elementet. I den iterative varianten har vi allerede gjort det iterativt når vi skal legge til det siste elementet. Men ... det er jo nesten samme sak, da.

```
SUM(A, i)
1 if  $i < 1$ 
2     return 0
3  $tmp = \text{SUM}(A, i - 1)$ 
4 return  $tmp + A[i]$ 
```

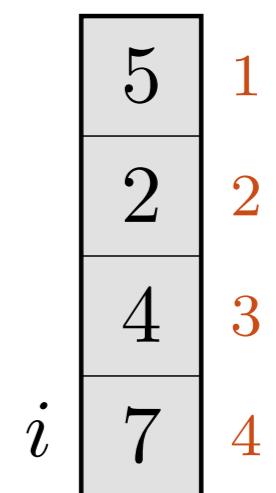
```
SUM(A)
1  $res = 0$ 
2 for  $j = 1$  to  $A.length$ 
3      $res = res + A[j]$ 
4 return  $res$ 
```

5	1
2	2
4	3
7	4

```
SUM(A, i)
1  if  $i < 1$ 
2      return 0
3  tmp = SUM(A,  $i - 1$ )
4  return tmp + A[i]
```

```
SUM(A)
1  res = 0
2  for  $j = 1$  to A.length
3      res = res + A[j]
4  return res
```

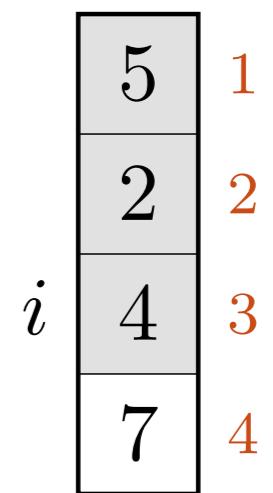
tmp = -



```
SUM(A, i)
1  if  $i < 1$ 
2      return 0
3  tmp = SUM(A,  $i - 1$ )
4  return tmp + A[i]
```

```
SUM(A)
1  res = 0
2  for  $j = 1$  to A.length
3      res = res + A[j]
4  return res
```

tmp = ->-



```
SUM(A, i)
1 if  $i < 1$ 
2     return 0
3  $tmp = \text{SUM}(A, i - 1)$ 
4 return  $tmp + A[i]$ 
```

```
SUM(A)
1  $res = 0$ 
2 for  $j = 1$  to  $A.length$ 
3      $res = res + A[j]$ 
4 return  $res$ 
```

$tmp = - \rightarrow -$

5	1
2	2
4	3
7	4

i

```

SUM(A, i)
1 if  $i < 1$ 
2     return 0
3  $tmp = \text{SUM}(A, i - 1)$ 
4 return  $tmp + A[i]$ 

```

```

SUM(A)
1  $res = 0$ 
2 for  $j = 1$  to  $A.length$ 
3      $res = res + A[j]$ 
4 return  $res$ 

```

$tmp = - \rightarrow - \rightarrow -$

i	5	1
	2	2
	4	3
	7	4

```
SUM(A, i)
1 if  $i < 1$ 
2     return 0
3  $tmp = \text{SUM}(A, i - 1)$ 
4 return  $tmp + A[i]$ 
```

```
SUM(A)
1  $res = 0$ 
2 for  $j = 1$  to  $A.length$ 
3      $res = res + A[j]$ 
4 return  $res$ 
```

$tmp = - \rightarrow - \rightarrow -$

5	1
2	2
4	3
7	4

```

SUM(A, i)
1 if  $i < 1$ 
2     return 0
3  $tmp = \text{SUM}(A, i - 1)$ 
4 return  $tmp + A[i]$ 

```

```

SUM(A)
1  $res = 0$ 
2 for  $j = 1$  to  $A.length$ 
3      $res = res + A[j]$ 
4 return  $res$ 

```

$tmp = - \rightarrow - \rightarrow - \rightarrow -$

i	5	1
	2	2
	4	3
	7	4

```

SUM(A, i)
1 if  $i < 1$ 
2     return 0
3  $tmp = \text{SUM}(A, i - 1)$ 
4 return  $tmp + A[i]$ 

```

```

SUM(A)
1  $res = 0$ 
2 for  $j = 1$  to  $A.length$ 
3      $res = res + A[j]$ 
4 return  $res$ 

```

$tmp = - \rightarrow - \rightarrow - \rightarrow -$

i	5	1
	2	2
	4	3
	7	4

```
SUM(A, i)
1  if  $i < 1$ 
2      return 0
3  tmp = SUM(A,  $i - 1$ )
4  return tmp + A[i]
```

```
SUM(A)
1  res = 0
2  for  $j = 1$  to A.length
3      res = res + A[j]
4  return res
```

tmp = ->->->->-

5	1
2	2
4	3
7	4

```

SUM(A, i)
1 if  $i < 1$ 
2   return 0
3  $tmp = \text{SUM}(A, i - 1)$ 
4 return  $tmp + A[i]$ 

```

```

SUM(A)
1  $res = 0$ 
2 for  $j = 1$  to  $A.length$ 
3    $res = res + A[j]$ 
4 return  $res$ 

```

$tmp = - \rightarrow - \rightarrow - \rightarrow -$

$res = 0$

5	1
2	2
4	3
7	4

```
SUM(A, i)
1 if  $i < 1$ 
2     return 0
3  $tmp = \text{SUM}(A, i - 1)$ 
4 return  $tmp + A[i]$ 
```

→ 0

$tmp = - \rightarrow - \rightarrow - \rightarrow -$

```
SUM(A)
1  $res = 0$ 
2 for  $j = 1$  to  $A.length$ 
3      $res = res + A[j]$ 
4 return  $res$ 
```

$res = 0$

j	5	1
	2	2
	4	3
	7	4

```

SUM(A, i)
1 if  $i < 1$ 
2   return 0
3    $tmp = \text{SUM}(A, i - 1)$ 
4   return  $tmp + A[i]$ 

```

$tmp = \dots \rightarrow 0$

```

SUM(A)
1  $res = 0$ 
2 for  $j = 1$  to  $A.length$ 
3    $res = res + A[j]$ 
4 return  $res$ 

```

$res = 5$

i, j	5	1
	2	2
	4	3
	7	4

```
SUM(A, i)
1 if  $i < 1$ 
2     return 0
3  $tmp = \text{SUM}(A, i - 1)$ 
4 return  $tmp + A[i]$ 
```

→ 5

$tmp = \dots \rightarrow 0$

```
SUM(A)
1  $res = 0$ 
2 for  $j = 1$  to  $A.length$ 
3      $res = res + A[j]$ 
4 return  $res$ 
```

$res = 5$

i	5	1
j	2	2
	4	3
	7	4

```
SUM(A, i)
1 if  $i < 1$ 
2     return 0
3  $tmp = \text{SUM}(A, i - 1)$ 
4 return  $tmp + A[i]$ 
```

$tmp = \dots \rightarrow \dots \rightarrow 5$

```
SUM(A)
1  $res = 0$ 
2 for  $j = 1$  to  $A.length$ 
3      $res = res + A[j]$ 
4 return  $res$ 
```

$res = 7$

i, j	5	1
	2	2
	4	3
	7	4

```

SUM(A, i)
1 if  $i < 1$ 
2     return 0
3  $tmp = \text{SUM}(A, i - 1)$ 
4 return  $tmp + A[i]$ 

```

$\rightarrow 7$

$tmp = \dots \rightarrow 5$

```

SUM(A)
1  $res = 0$ 
2 for  $j = 1$  to  $A.length$ 
3      $res = res + A[j]$ 
4 return  $res$ 

```

$res = 7$

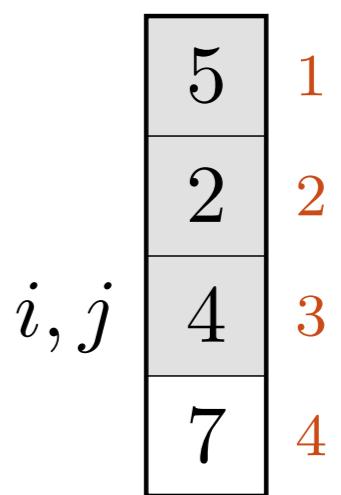
5	1
2	2
4	3
7	4

```
SUM(A, i)
1 if  $i < 1$ 
2     return 0
3  $tmp = \text{SUM}(A, i - 1)$ 
4 return  $tmp + A[i]$ 
```

$tmp = \dots \rightarrow 7$

```
SUM(A)
1  $res = 0$ 
2 for  $j = 1$  to  $A.length$ 
3      $res = res + A[j]$ 
4 return  $res$ 
```

$res = 11$



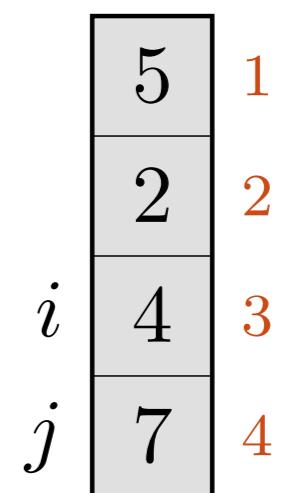
```
SUM(A, i)
1 if  $i < 1$ 
2     return 0
3  $tmp = \text{SUM}(A, i - 1)$ 
4 return  $tmp + A[i]$ 
```

→ 11

$tmp = - \rightarrow 7$

```
SUM(A)
1  $res = 0$ 
2 for  $j = 1$  to  $A.length$ 
3      $res = res + A[j]$ 
4 return  $res$ 
```

$res = 11$

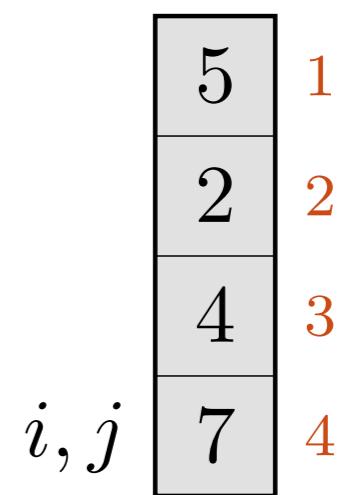


```
SUM(A, i)
1  if  $i < 1$ 
2      return 0
3  tmp = SUM(A,  $i - 1$ )
4  return tmp + A[i]
```

$tmp = 11$

```
SUM(A)
1  res = 0
2  for  $j = 1$  to A.length
3      res = res + A[j]
4  return res
```

$res = 18$



```
SUM(A, i)
1 if  $i < 1$ 
2     return 0
3  $tmp = \text{SUM}(A, i - 1)$ 
4 return  $tmp + A[i]$ 
```

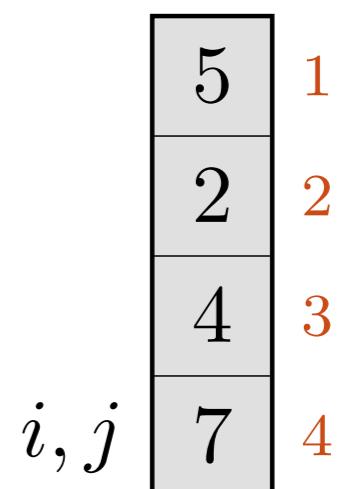
→ 18

$tmp = 11$

```
SUM(A)
1  $res = 0$ 
2 for  $j = 1$  to  $A.length$ 
3      $res = res + A[j]$ 
4 return  $res$ 
```

→ 18

$res = 18$



OK, endelig litt ordentlig algoritmedesign! (Selv om vi jo allerede har designet denne algoritmen, egentlig.)

5:5

Eksempel: Insertion-sort



Denne publikasjonen, fra 1959, introduserer Shellsort, en forbedring av Insertion Sort.

Tolkning

T

Hva er relasjonen mellom input og output?

Analyse

A

Del en vilkårlig instans i delinstanser.

Syntese

S

Bygg løsning av hypotetiske delløsninger.

**Dekomponeringen er omtrent den samme;
vi bare bytter ut sum med sortering. I
stedet for å legge til neste element, så setter
vi inn neste element på rett plass.**

Dere kan få utallige algoritmer ut av akkurat samme idé: Anta at du har gjort noe rett for de $n-1$ første elementene, og så bygg deg videre til n ved å gjøre noe med det siste elementet.

Løsningen blir da rett, enten du bruker rekursjon eller iterasjon.

Rek. Dekomp.

- Vi vil sortere elementene i en tabell
- Rekursjon: Sortér alle unntatt det siste
- Induktivt premiss:
Anta at dette blir rett
- Induksjonstrinn: Sett inn siste element

Iter. Dekomp.

- Invariant: Vi har sortert rett så langt
- Initialisering: Tom sekvens er sortert
- Vedlikehold:
 - Induktivt premiss:
Sorteringen er rett før iterasjonen
 - Induksjonstrinn: Sett inn neste element
- Terminering: Til slutt har vi sortert alle

INSERTION-SORT(A)

Sortér A

```
INSERTION-SORT(A)
1  for  $j = 2$  to  $A.length$ 
```

Induksjonshypotese: $A[1..j]$ er sortert

```
INSERTION-SORT(A)
1  for  $j = 2$  to  $A.length$ 
2       $key = A[j]$ 
```

Induksjonstrinn: Sett inn $A[j]$ på rett plass

```
INSERTION-SORT(A)
1  for  $j = 2$  to  $A.length$ 
2       $key = A[j]$ 
3       $i = j - 1$ 
```

Vi går nedover fra forrige element

INSERTION-SORT(A)

```
1  for  $j = 2$  to  $A.length$ 
2       $key = A[j]$ 
3       $i = j - 1$ 
4      while  $i > 0$  and  $A[i] > key$ 
```

Fortsatt elementer større enn $A[j]$?

```
INSERTION-SORT(A)
1  for  $j = 2$  to A.length
2      key = A[j]
3      i = j - 1
4      while  $i > 0$  and A[i] > key
5          A[i + 1] = A[i]
```

Da gjør vi plass ett hakk lenger ned

INSERTION-SORT(A)

```
1  for  $j = 2$  to  $A.length$ 
2       $key = A[j]$ 
3       $i = j - 1$ 
4      while  $i > 0$  and  $A[i] > key$ 
5           $A[i + 1] = A[i]$ 
6           $i = i - 1$ 
```

```
INSERTION-SORT(A)
1  for  $j = 2$  to A.length
2      key = A[j]
3      i = j - 1
4      while  $i > 0$  and A[i] > key
5          A[i + 1] = A[i]
6          i = i - 1
7      A[i + 1] = key
```

Større elementer er nå til høyre

INSERTION-SORT(A)

```
1  for  $j = 2$  to  $A.length$ 
2       $key = A[j]$ 
3       $i = j - 1$ 
4      while  $i > 0$  and  $A[i] > key$ 
5           $A[i + 1] = A[i]$ 
6           $i = i - 1$ 
7       $A[i + 1] = key$ 
```

$i, j, key = -, -, -$

Merk at vi gjør flere ting på én gang, her (noe vi ikke egentlig hadde trengt). Samtidig som vi leter etter rett plass for $A[j]$, så flytter vi unna elementene som står i veien.

Hvert element som står i veien kopieres et hakk opp. Det første av disse overskriver $A[j]$, det neste overskriver $A[j-1]$, etc.

5	1
2	2
4	3
4	4
6	5
1	6
3	

INSERTION-SORT(A)

```
1  for  $j = 2$  to A.length
2      key = A[j]
3      i = j - 1
4      while  $i > 0$  and A[i] > key
5          A[i + 1] = A[i]
6          i = i - 1
7      A[i + 1] = key
```

$i, j, key = -, 2, -$

j	5	1
	2	2
	4	3
	6	4
	1	5
	3	6

INSERTION-SORT(A)

```
1  for  $j = 2$  to A.length
2      key = A[j]
3      i = j - 1
4      while  $i > 0$  and A[i] > key
5          A[i + 1] = A[i]
6          i = i - 1
7      A[i + 1] = key
```

$i, j, key = -, 2, 2$

j	5	1
	2	2
	4	3
	6	4
	1	5
	3	6

INSERTION-SORT(A)

```
1  for  $j = 2$  to A.length
2      key = A[j]
3      i = j - 1
4      while  $i > 0$  and A[i] > key
5          A[i + 1] = A[i]
6          i = i - 1
7      A[i + 1] = key
```

$i, j, key = 1, 2, 2$

i	5	1
j	2	2
	4	3
	6	4
	1	5
	3	6

INSERTION-SORT(A)

```
1  for  $j = 2$  to  $A.length$ 
2       $key = A[j]$ 
3       $i = j - 1$ 
4      while  $i > 0$  and  $A[i] > key$ 
5           $A[i + 1] = A[i]$ 
6           $i = i - 1$ 
7       $A[i + 1] = key$ 
```

$i, j, key = 1, 2, 2$

i	5	1
j	2	2
	4	3
	6	4
	1	5
	3	6

INSERTION-SORT(A)

```
1  for  $j = 2$  to  $A.length$ 
2       $key = A[j]$ 
3       $i = j - 1$ 
4      while  $i > 0$  and  $A[i] > key$ 
5           $A[i + 1] = A[i]$ 
6           $i = i - 1$ 
7       $A[i + 1] = key$ 
```

$i, j, key = 1, 2, 2$

i	5	1
j	5	2
	4	3
	6	4
	1	5
	3	6

INSERTION-SORT(A)

```
1  for  $j = 2$  to A.length
2      key = A[j]
3      i = j - 1
4      while  $i > 0$  and A[i] > key
5          A[i + 1] = A[i]
6          i = i - 1
7      A[i + 1] = key
```

$i, j, key = 0, 2, 2$

j	5	1
	5	2
	4	3
	6	4
	1	5
	3	6

INSERTION-SORT(A)

```
1  for  $j = 2$  to A.length
2      key = A[j]
3      i = j - 1
4      while  $i > 0$  and A[i] > key
5          A[i + 1] = A[i]
6          i = i - 1
7      A[i + 1] = key
```

$i, j, key = 0, 2, 2$

j	5	1
	5	2
	4	3
	6	4
	1	5
	3	6

INSERTION-SORT(A)

```
1  for  $j = 2$  to  $A.length$ 
2       $key = A[j]$ 
3       $i = j - 1$ 
4      while  $i > 0$  and  $A[i] > key$ 
5           $A[i + 1] = A[i]$ 
6           $i = i - 1$ 
7       $A[i + 1] = key$ 
```

492

$i, j, key = 0, 2, 2$

j	2	1
	5	2
	4	3
	6	4
	1	5
	3	6

INSERTION-SORT(A)

```
1  for  $j = 2$  to A.length
2      key = A[j]
3      i = j - 1
4      while  $i > 0$  and A[i] > key
5          A[i + 1] = A[i]
6          i = i - 1
7      A[i + 1] = key
```

$i, j, key = 0, 3, 2$

2	1
5	2
4	3
6	4
1	5
3	6

j

INSERTION-SORT(A)

```
1  for  $j = 2$  to A.length  
2      key = A[j]  
3      i = j - 1  
4      while  $i > 0$  and A[i] > key  
5          A[i + 1] = A[i]  
6          i = i - 1  
7      A[i + 1] = key
```

$i, j, key = 0, 3, 4$

2	1
5	2
4	3
6	4
1	5
3	6

j

INSERTION-SORT(A)

```
1  for  $j = 2$  to A.length
2      key = A[j]
3      i = j - 1
4      while  $i > 0$  and A[i] > key
5          A[i + 1] = A[i]
6          i = i - 1
7      A[i + 1] = key
```

$i, j, key = 2, 3, 4$

	2	1
i	5	2
j	4	3
	6	4
	1	5
	3	6

INSERTION-SORT(A)

```
1  for  $j = 2$  to A.length
2      key = A[j]
3      i = j - 1
4      while  $i > 0$  and A[i] > key
5          A[i + 1] = A[i]
6          i = i - 1
7      A[i + 1] = key
```

$i, j, key = 2, 3, 4$

	2	1
i	5	2
j	4	3
	6	4
	1	5
	3	6

INSERTION-SORT(A)

```
1  for  $j = 2$  to A.length
2      key = A[j]
3      i = j - 1
4      while  $i > 0$  and A[i] > key
5          A[i + 1] = A[i]
6          i = i - 1
7      A[i + 1] = key
```

$i, j, key = 2, 3, 4$

	2	1
i	5	2
j	5	3
	6	4
	1	5
	3	6

INSERTION-SORT(A)

```
1  for  $j = 2$  to A.length
2      key = A[j]
3      i = j - 1
4      while  $i > 0$  and A[i] > key
5          A[i + 1] = A[i]
6          i = i - 1
7      A[i + 1] = key
```

$i, j, key = 1, 3, 4$

i	2	1
	5	2
j	5	3
	6	4
	1	5
	3	6

INSERTION-SORT(A)

```
1  for  $j = 2$  to  $A.length$ 
2       $key = A[j]$ 
3       $i = j - 1$ 
4      while  $i > 0$  and  $A[i] > key$ 
5           $A[i + 1] = A[i]$ 
6           $i = i - 1$ 
7       $A[i + 1] = key$ 
```

$i, j, key = 1, 3, 4$

i	2	1
	5	2
j	5	3
	6	4
	1	5
	3	6

INSERTION-SORT(A)

```
1  for  $j = 2$  to  $A.length$ 
2       $key = A[j]$ 
3       $i = j - 1$ 
4      while  $i > 0$  and  $A[i] > key$ 
5           $A[i + 1] = A[i]$ 
6           $i = i - 1$ 
7       $A[i + 1] = key$ 
```

$i, j, key = 1, 3, 4$

i	2	1
	4	2
j	5	3
	6	4
	1	5
	3	6

INSERTION-SORT(A)

```
1  for  $j = 2$  to A.length
2      key = A[j]
3      i = j - 1
4      while  $i > 0$  and A[i] > key
5          A[i + 1] = A[i]
6          i = i - 1
7      A[i + 1] = key
```

$i, j, key = 1, 4, 4$

i	2	1
	4	2
	5	3
j	6	4
	1	5
	3	6

INSERTION-SORT(A)

```
1  for  $j = 2$  to A.length
2      key = A[j]
3      i = j - 1
4      while  $i > 0$  and A[i] > key
5          A[i + 1] = A[i]
6          i = i - 1
7      A[i + 1] = key
```

$i, j, key = 1, 4, 6$

i	2	1
	4	2
	5	3
j	6	4
	1	5
	3	6

INSERTION-SORT(A)

```
1  for  $j = 2$  to A.length
2      key = A[j]
3      i = j - 1
4      while  $i > 0$  and A[i] > key
5          A[i + 1] = A[i]
6          i = i - 1
7      A[i + 1] = key
```

$i, j, key = 3, 4, 6$

2	1
4	2
i	5
j	6
1	4
3	5
3	6

INSERTION-SORT(A)

```
1  for  $j = 2$  to A.length
2      key = A[j]
3      i = j - 1
4      while  $i > 0$  and A[i] > key
5          A[i + 1] = A[i]
6          i = i - 1
7      A[i + 1] = key
```

$i, j, key = 3, 4, 6$

2	1
4	2
i	5
j	6
1	4
3	5
3	6

INSERTION-SORT(A)

```
1  for  $j = 2$  to A.length
2      key = A[j]
3      i = j - 1
4      while  $i > 0$  and A[i] > key
5          A[i + 1] = A[i]
6          i = i - 1
7      A[i + 1] = key
```

$i, j, key = 3, 4, 6$

2	1
4	2
i	3
j	4
6	5
1	5
3	6

INSERTION-SORT(A)

```
1  for  $j = 2$  to A.length
2      key = A[j]
3      i = j - 1
4      while  $i > 0$  and A[i] > key
5          A[i + 1] = A[i]
6          i = i - 1
7      A[i + 1] = key
```

$i, j, key = 3, 5, 6$

2	1
4	2
i	5
6	4
j	1
3	5
	6

INSERTION-SORT(A)

```
1  for  $j = 2$  to A.length
2      key = A[j]
3      i = j - 1
4      while  $i > 0$  and A[i] > key
5          A[i + 1] = A[i]
6          i = i - 1
7      A[i + 1] = key
```

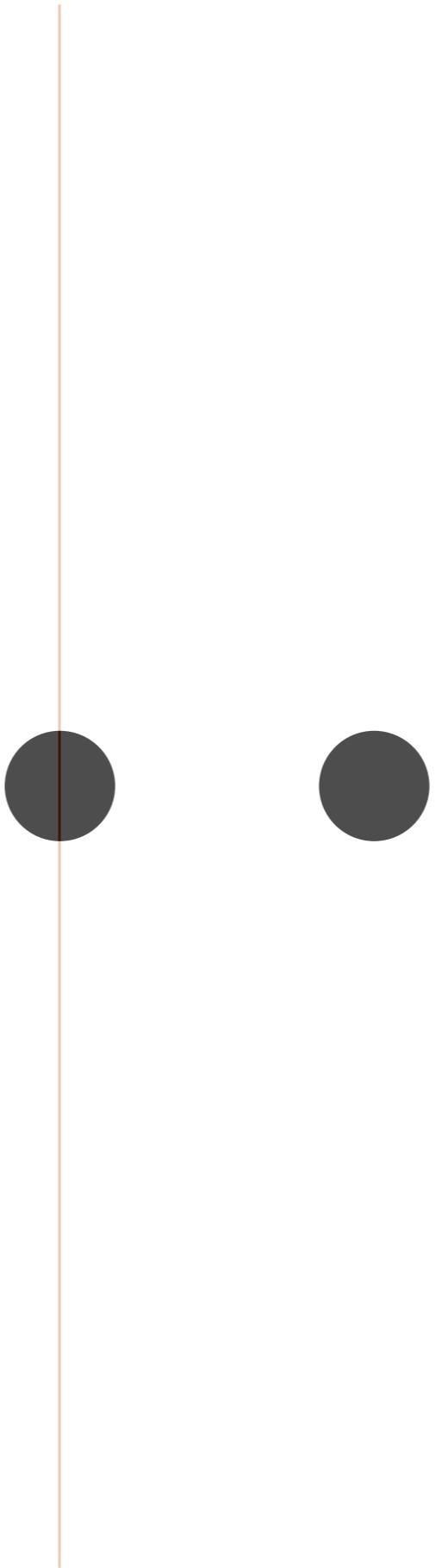
$i, j, key = 3, 5, 1$

2	1
4	2
i	5
6	4
j	1
	5
	3

INSERTION-SORT(A)

```
1  for  $j = 2$  to A.length
2      key = A[j]
3      i = j - 1
4      while  $i > 0$  and A[i] > key
5          A[i + 1] = A[i]
6          i = i - 1
7      A[i + 1] = key
```

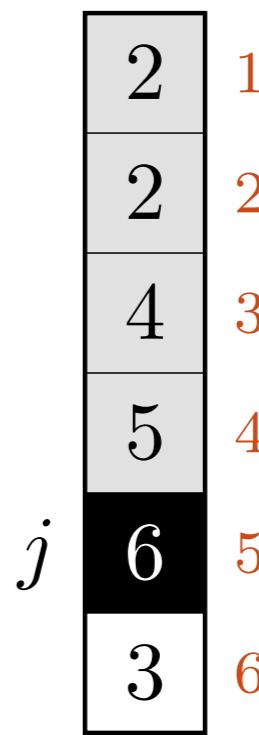
$i, j, key = 4, 5, 1$



2	1
4	2
5	3
6	4
1	5
3	6

INSERTION-SORT(A)

```
1  for  $j = 2$  to A.length
2      key = A[j]
3      i = j - 1
4      while  $i > 0$  and A[i] > key
5          A[i + 1] = A[i]
6          i = i - 1
7      A[i + 1] = key
```



$i, j, key = 0, 5, 1$

INSERTION-SORT(A)

```
1  for  $j = 2$  to A.length
2      key = A[j]
3      i = j - 1
4      while  $i > 0$  and A[i] > key
5          A[i + 1] = A[i]
6          i = i - 1
7      A[i + 1] = key
```

$i, j, key = 0, 5, 1$

1	1
2	2
4	3
5	4
6	5
3	6

j

INSERTION-SORT(A)

```
1  for  $j = 2$  to A.length
2      key = A[j]
3      i = j - 1
4      while  $i > 0$  and A[i] > key
5          A[i + 1] = A[i]
6          i = i - 1
7      A[i + 1] = key
```

$i, j, key = 0, 6, 1$

1	1
2	2
4	3
5	4
6	5
j	3
	6

INSERTION-SORT(A)

```
1  for  $j = 2$  to A.length
2      key = A[j]
3      i = j - 1
4      while  $i > 0$  and A[i] > key
5          A[i + 1] = A[i]
6          i = i - 1
7      A[i + 1] = key
```

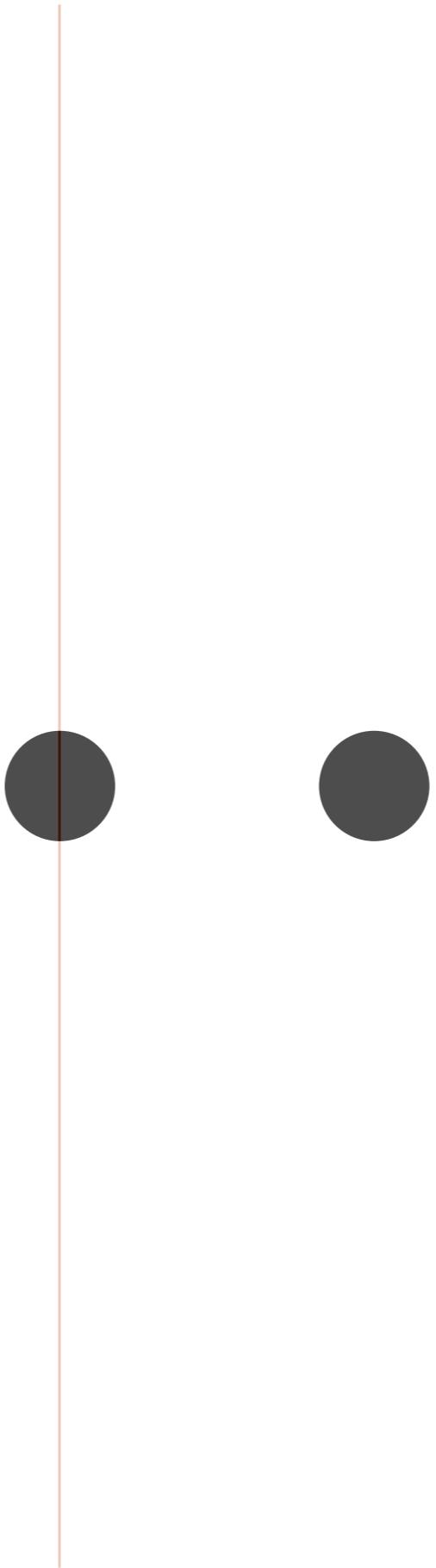
$i, j, key = 0, 6, 3$

1	1
2	2
4	3
5	4
6	5
j	3
	6

INSERTION-SORT(A)

```
1  for  $j = 2$  to A.length
2      key = A[j]
3      i = j - 1
4      while  $i > 0$  and A[i] > key
5          A[i + 1] = A[i]
6          i = i - 1
7      A[i + 1] = key
```

$i, j, key = 5, 6, 3$

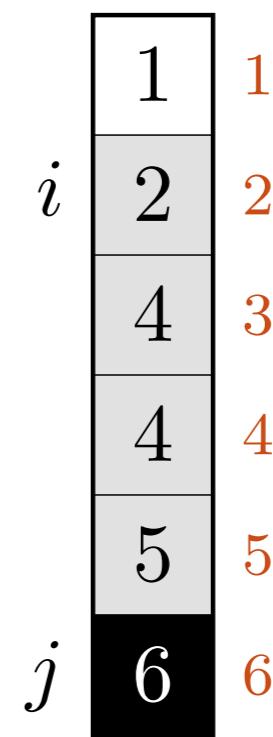


1	1
2	2
4	3
5	4
6	5
j	3

INSERTION-SORT(A)

```
1  for  $j = 2$  to A.length
2      key = A[j]
3      i = j - 1
4      while  $i > 0$  and A[i] > key
5          A[i + 1] = A[i]
6          i = i - 1
7      A[i + 1] = key
```

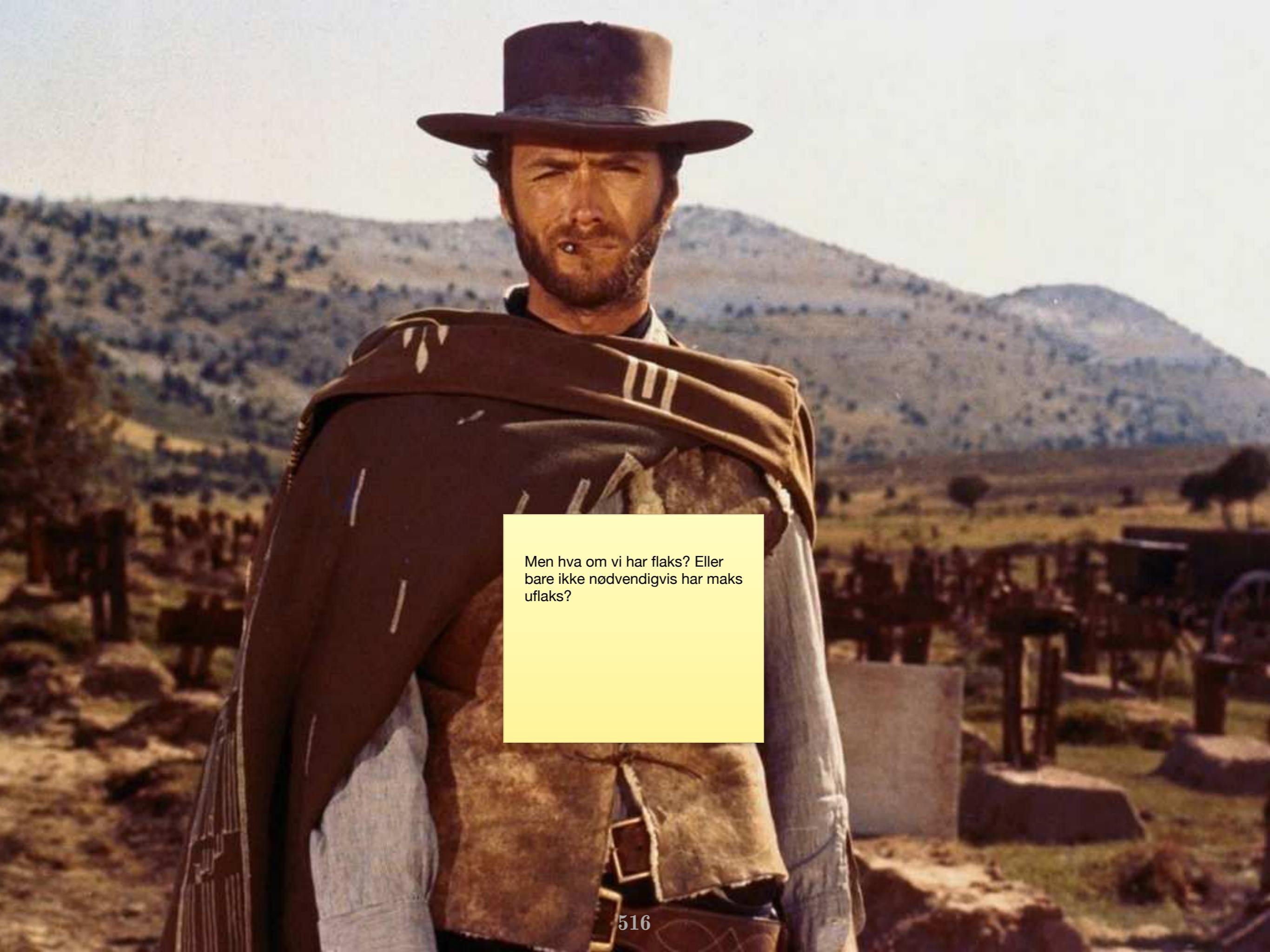
$i, j, key = 2, 6, 3$



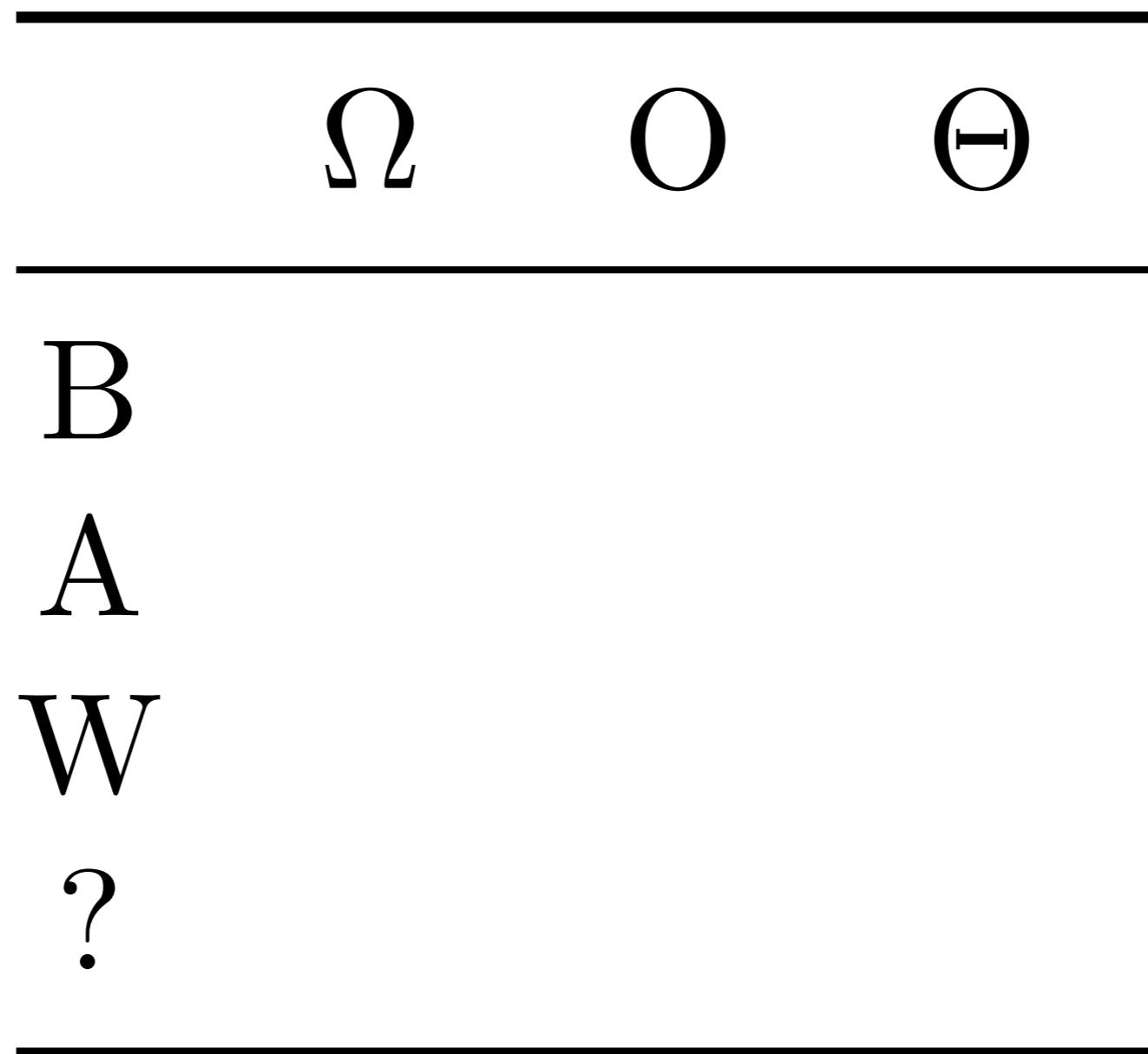
INSERTION-SORT(A)

```
1  for  $j = 2$  to A.length
2      key = A[j]
3      i = j - 1
4      while  $i > 0$  and A[i] > key
5          A[i + 1] = A[i]
6          i = i - 1
7      A[i + 1] = key
```

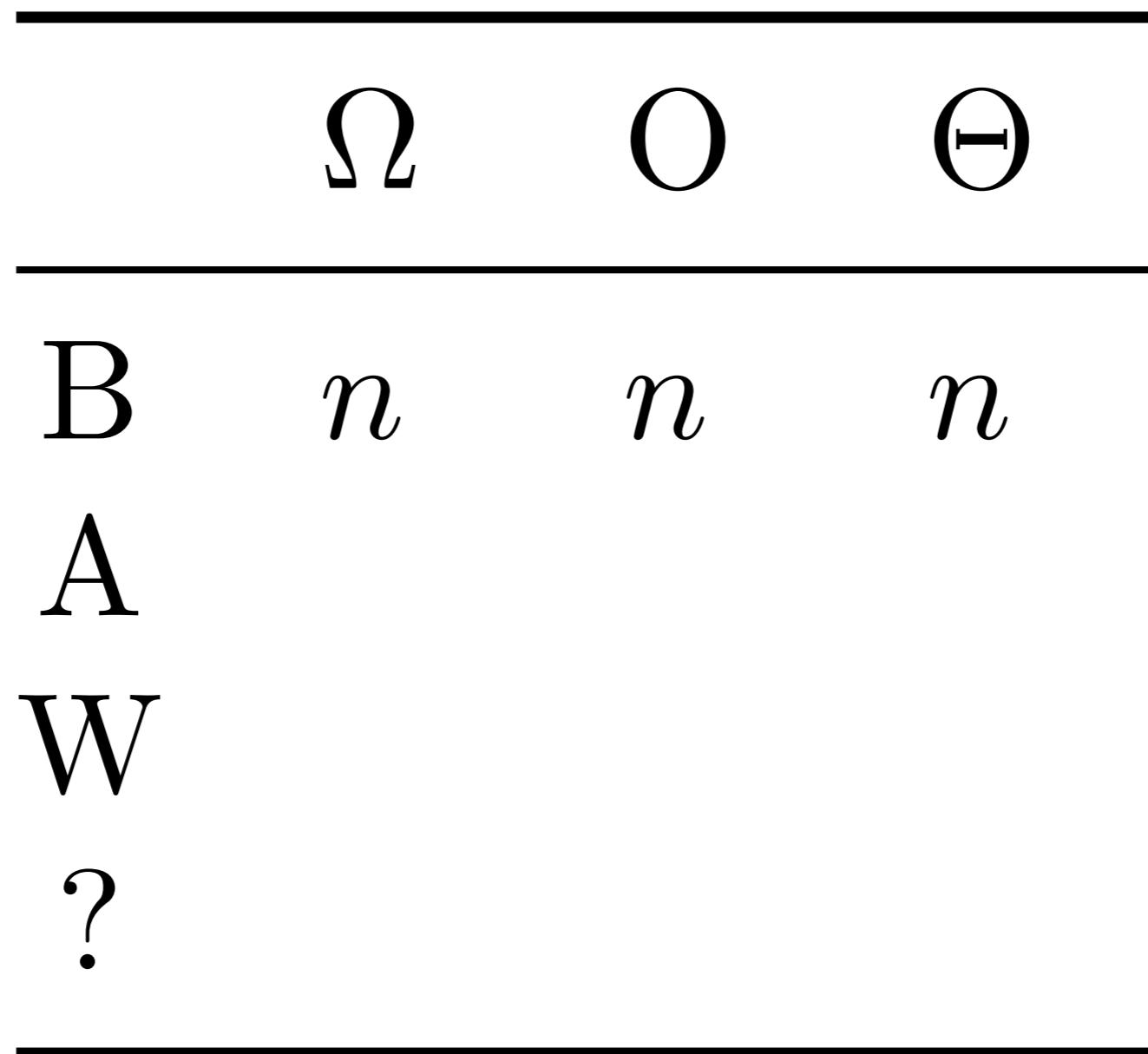
1	1
2	2
3	3
4	4
5	5
6	6



Men hva om vi har flaks? Eller
bare ikke nødvendigvis har maks
uflaks?



Best-, average- & worst-case og helt generelt



Best-, average- & worst-case og helt generelt

	Ω	O	Θ
B	n	n	n
A	n^2	n^2	n^2
W	n^2	n^2	n^2
?			

Best-, average- & worst-case og helt generelt

	Ω	O	Θ
B	n	n	n
A	n^2	n^2	n^2
W	n^2	n^2	n^2
?	n	n^2	

Best-, average- & worst-case og helt generelt

	Ω	O	Θ
B	n	n	n
A	n^2	n^2	n^2
W	n^2	n^2	n^2
?	n	n^2	—

Kan ikke bruke Θ på INSERTION-SORT for ukjent input

- Vet vi om vi har best- worst- eller average-case, kan alle tre asymptotiske notasjoner brukes!
- Vet vi det ikke, må vi velge en notasjon som favner alle muligheter

Hovedbudskap (igjen):

- Brute force er ofte helt ubruklig
- Dekomponer problemet i stedet:
 - Anta at du kan løse mindre instanser
 - Bruk dette til å finne en løsning

1. Hva og hvorfor?
2. Asymptotisk notation
3. Dekomponering
4. Eksempler
5. Eksempel: Insertion-sort



Bonusmateriale

$$n! > 2^n$$

Et lite induksjonseksempel

Vil vise

$$n! > 2^n$$

P(*n*)

Vi vil vise dette for alle heltall $n \geq 4$

Vil vise

$$n! > 2^n$$

P(*n*)

Ekvivalent: Vi vil vise dette for et vilkårlig heltall $n \geq 4$

Vil vise

$$n! > 2^n$$

P(*n*)

Grunntilfelle

$$4! > 2^4$$

P(4)

$$4! = 24 > 15 = 2^4$$

Vil vise

$$n! > 2^n$$

P(*n*)

Grunntilfelle

$$4! > 2^4$$

P(4)

Vi starter her

Vil vise

$$n! > 2^n$$

P(*n*)

Grunntilfelle

$$4! > 2^4$$

P(4)

Vi vil så vise $P(4) \implies P(5) \implies P(6) \dots$

Vil vise

$$n! > 2^n$$

P(*n*)

Grunntilfelle

$$4! > 2^4$$

P(4)

Husk: n er vilkårlig, så det holder med $P(n - 1) \implies P(n)$

Vil vise

$$n! > 2^n$$

P(*n*)

Grunntilfelle

$$4! > 2^4$$

P(4)

Generelt: For å vise $A \implies B$, anta A og utled B

Vil vise

$$n! > 2^n$$

P(*n*)

Grunntilfelle

$$4! > 2^4$$

P(4)

Induksjonshypotese

$$(n - 1)! > 2^{n-1}$$

P(*n* - 1)

Vi antar altså P(*n* - 1) og vil utlede P(*n*)

Vil vise

$$n! > 2^n$$

 $P(n)$

Grunntilfelle

$$4! > 2^4$$

 $P(4)$

Induksjonshypotese

$$(n - 1)! > 2^{n-1}$$

 $P(n - 1)$

Induksjonstrinn

$$\implies n! > 2^n$$

 $P(n)$ Vi antar altså $P(n - 1)$ og vil utlede $P(n)$

Vil vise

$$n! > 2^n$$

P(*n*)

Grunntilfelle

$$4! > 2^4$$

P(4)

Induksjonshypotese

$$(n - 1)! > 2^{n-1}$$

P(*n* - 1)

Induksjonstrinn

$$\implies n! > 2^n$$

P(*n*)For å gjøre det, bryter vi ned $n!$ og 2^n rekursivt

Vil vise

$$n! > 2^n$$

 $P(n)$

Grunntilfelle

$$4! > 2^4$$

 $P(4)$

Induksjonshypotese

$$(n - 1)! > 2^{n-1}$$

 $P(n - 1)$

Induksjonstrinn

$$\implies n! > 2^n$$

 $P(n)$

Rekursjon

$$n! = n \cdot (n - 1)!$$

$$2^n = 2 \cdot 2^{n-1}$$

For å gjøre det, bryter vi ned $n!$ og 2^n rekursivt

Vil vise

$$n! > 2^n$$

 $P(n)$

Grunntilfelle

$$4! > 2^4$$

 $P(4)$

Induksjonshypotese

$$(n - 1)! > 2^{n-1}$$

 $P(n - 1)$

Induksjonstrinn

$$\implies n! > 2^n$$

 $P(n)$

Rekursjon

$$n! = n \cdot (n - 1)!$$

$$2^n = 2 \cdot 2^{n-1}$$

Vi antar $(n - 1)! > 2^{n-1}$ og vet $n > 2$, så $P(n - 1) \implies P(n)$

Vil vise

$$n! > 2^n$$

 $P(n)$

Grunntilfelle

$$4! > 2^4$$

 $P(4)$

Induksjonshypotese

$$(n - 1)! > 2^{n-1}$$

 $P(n - 1)$

Induksjonstrinn

$$\implies n! > 2^n$$

 $P(n)$

Rekursjon

$$n! = n \cdot (n - 1)!$$

$$2^n = 2 \cdot 2^{n-1}$$

Vi vet nå at $P(n)$ for $n = 4$ og $P(n - 1) \implies P(n)$ for $n > 4$

Vil vise

$$n! > 2^n$$

 $P(n)$

Grunntilfelle

$$4! > 2^4$$

 $P(4)$

Induksjonshypotese

$$(n - 1)! > 2^{n-1}$$

 $P(n - 1)$

Induksjonstrinn

$$\implies n! > 2^n$$

 $P(n)$

Rekursjon

$$n! = n \cdot (n - 1)!$$

$$2^n = 2 \cdot 2^{n-1}$$

Altså har vi vist at $n! > 2^n$ for alle $n \geq 4$

Vil vise

$$n! > 2^n$$

 $P(n)$

Grunntilfelle

$$4! > 2^4$$

 $P(4)$

Induksjonshypotese

$$(n - 1)! > 2^{n-1}$$

 $P(n - 1)$

Induksjonstrinn

$$\implies n! > 2^n$$

 $P(n)$

Rekursjon

$$n! = n \cdot (n - 1)!$$

$$2^n = 2 \cdot 2^{n-1}$$

Den rekursive strukturen til $n!$ og 2^n var sentral!

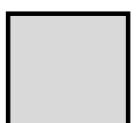
Noen vanlige dekomponeringer

- (i) Splitt av siste element
- (ii) Løs rekursivt for resten
- (iii) Sett inn siste element

?

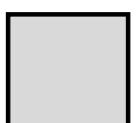
spalting

?



løsning

!



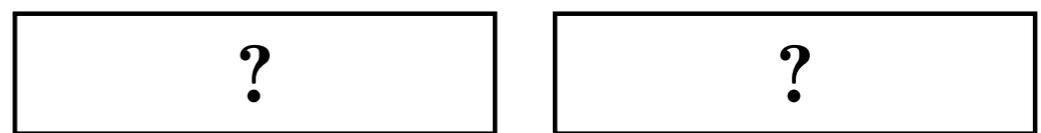
samling

!

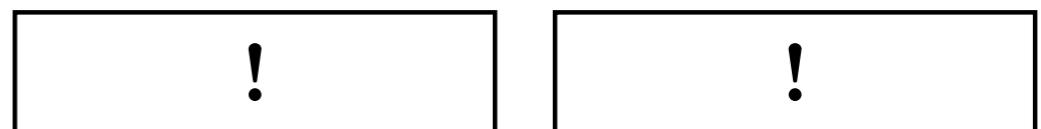
- (i) Del f.eks. på midten
- (ii) Løs halvdeler rekursivt
- (iii) Flett dem sammen



spalting



løsning



samling



- (i) Del f.eks. på midten
(ii) Løs én halvdel rekursivt

?

spalting

?



løsning

!

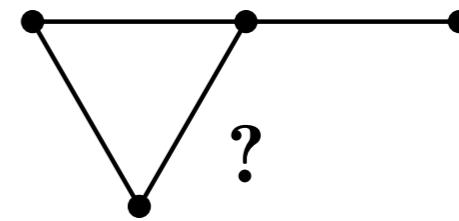


samling

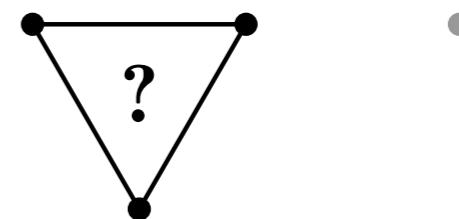
!



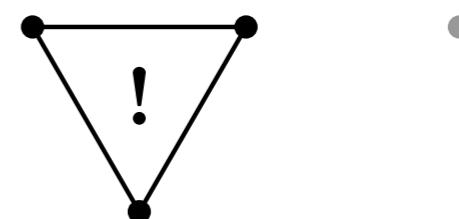
- (i) Splitt av siste node
- (ii) Løs rekursivt for resten
- (iii) Sett inn siste node



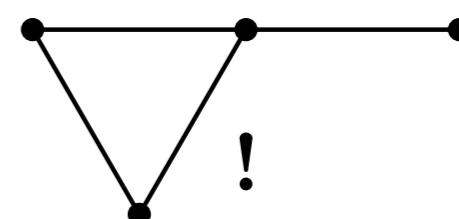
spalting



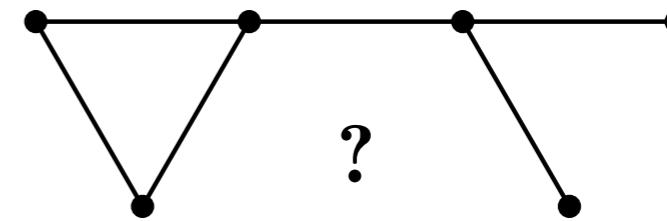
løsning



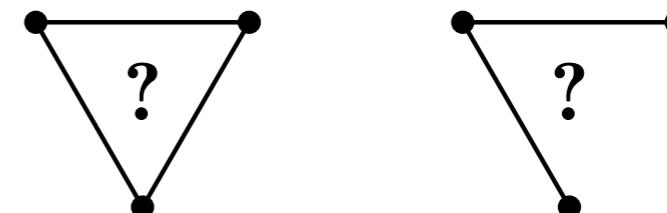
samling



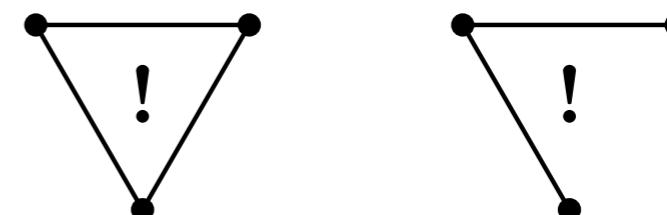
- (i) Fjern en kant
(ii) Løs hver del rekursivt
(iii) Sett inn kanten



spalting



løsning



samling

