

Forelesning 1

Problemer og algoritmer



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

1:5

Hva og hvorfor?

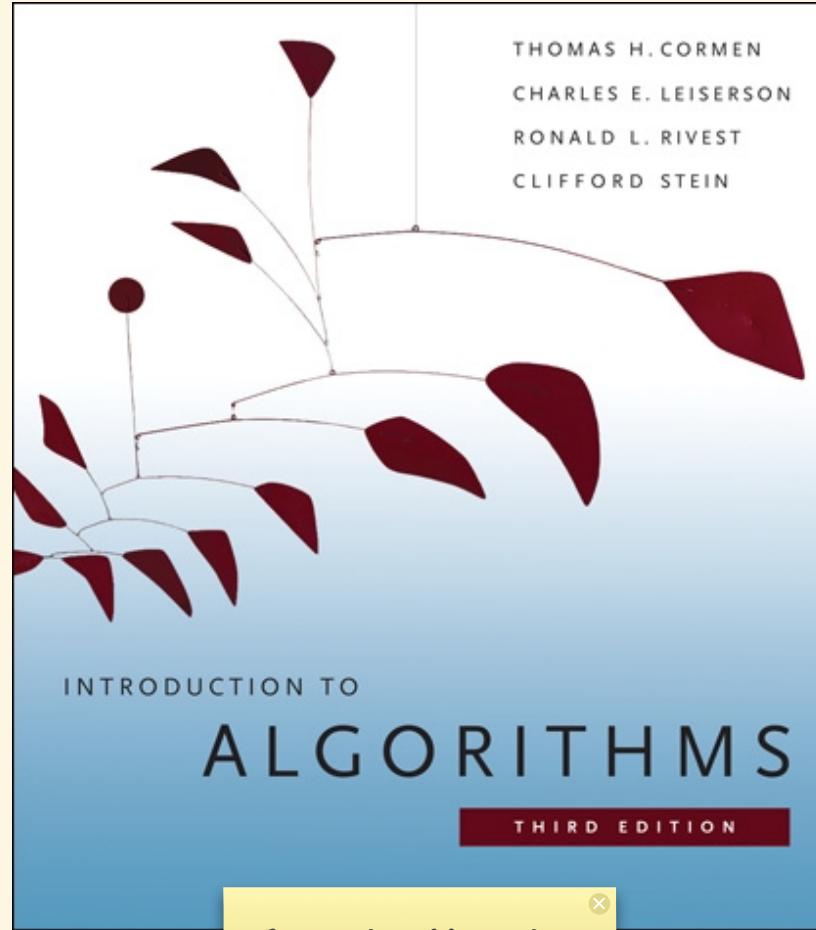


Om faget

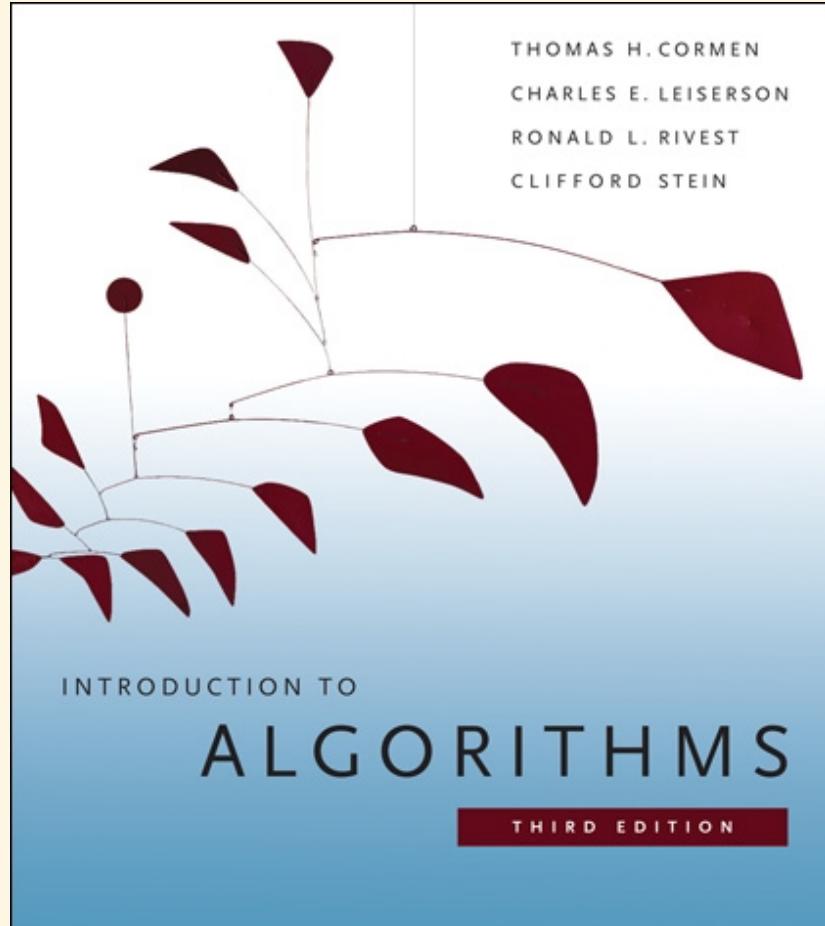
En kort orientering

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
RONALD L. RIVEST
CLIFFORD STEIN

INTRODUCTION TO

ALGORITHMS

THIRD EDITION



Det står også mye viktig
i dette heftet!

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

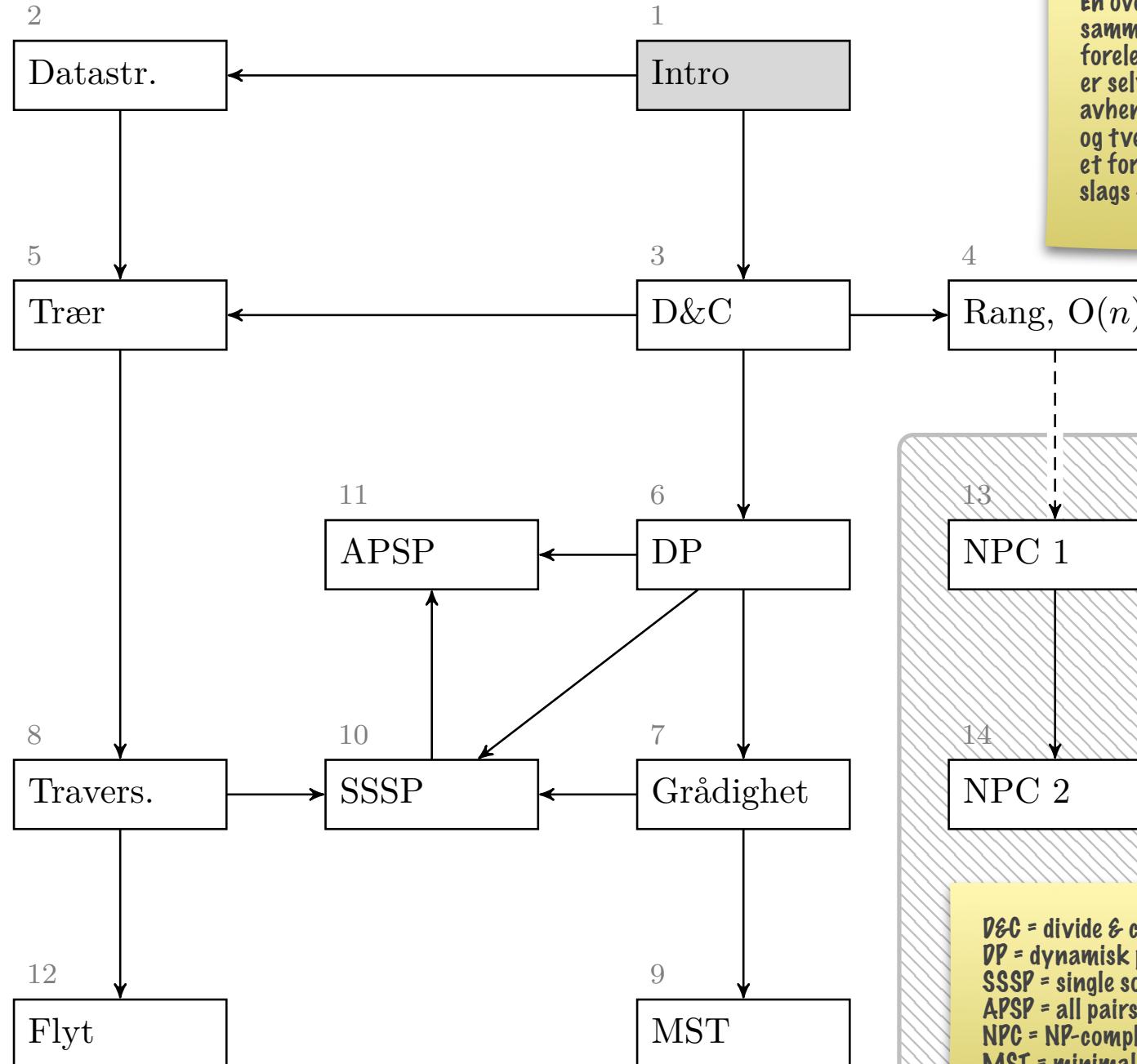
Her kan du komme i
kontakt med oss, og her
legger vi ut beskjeder.

piazza.com/ntnu.no/fall2017/tdt4120

Mer om øvinger her også.

Det er bare å begynne på øving 1!





En oversikt over noen sammenhenger mellom forelesningstemaene. Det er selvfølgelig avhengigheter på kryss og tvers – dette er bare et forsøk på å skissere et slags «kart».

D&C = divide & conquer
 DP = dynamisk programmering
 SSSP = single source shortest path
 APSP = all pairs shortest path
 NPC = NP-complete
 MST = minimal spanning tree

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?

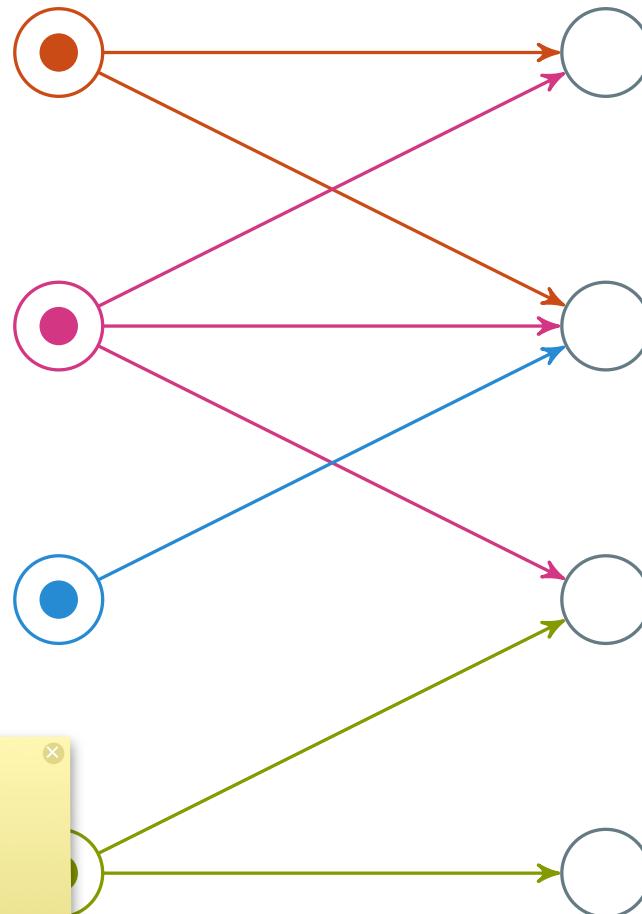
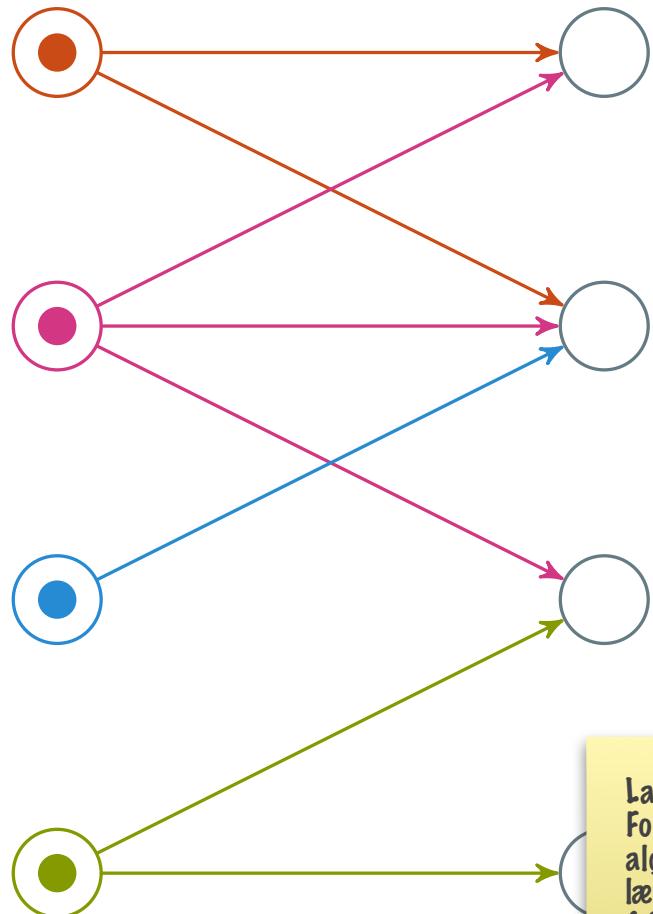
Dette kalles bogo-sort,
og er noen særlig god
løsning.

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.

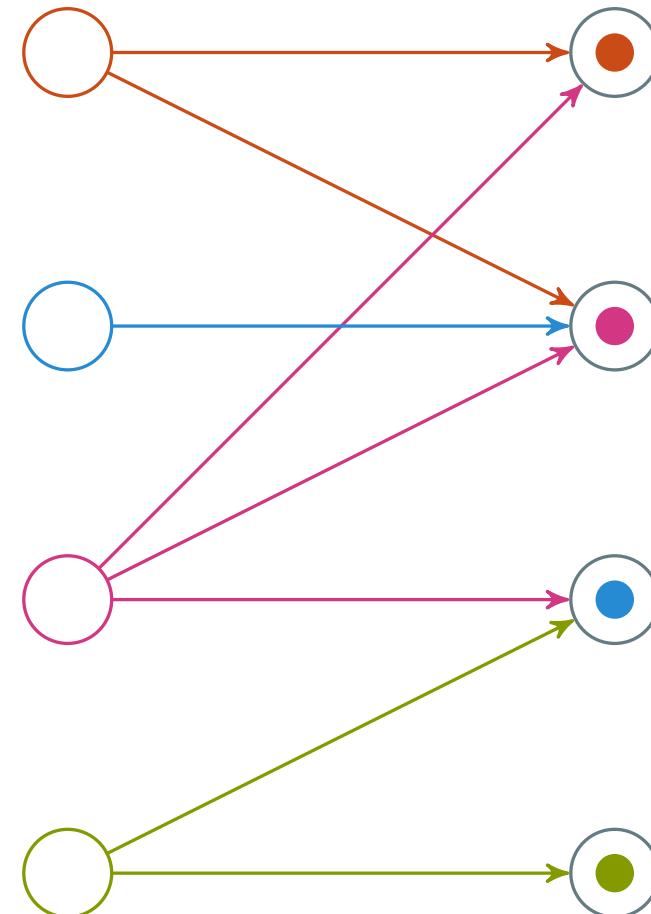
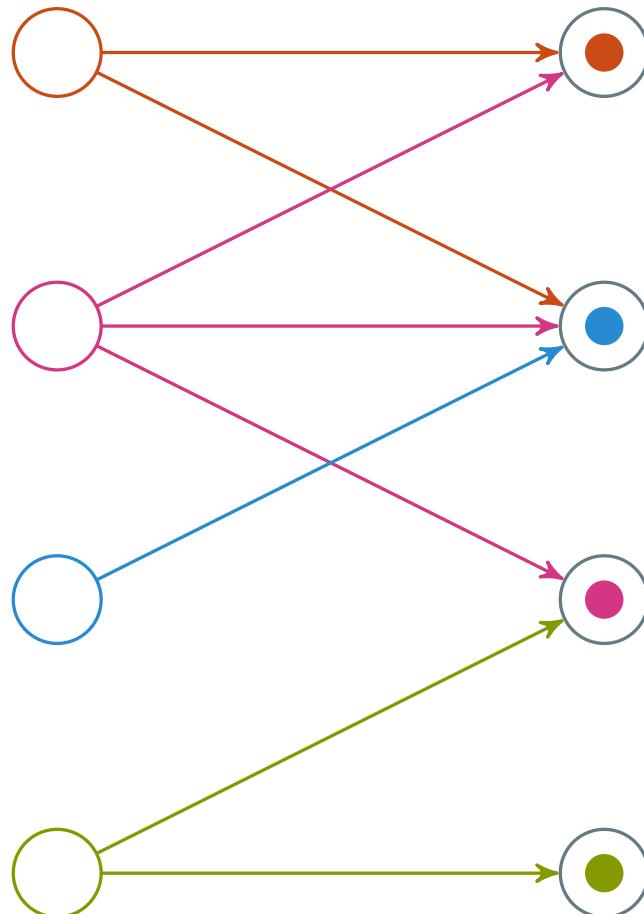
Her er det kanskje mindre opplagt hvordan
vi skal gå frem - og om det er greit å prøve
alle muligheter.





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.

motivasjon → matching



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



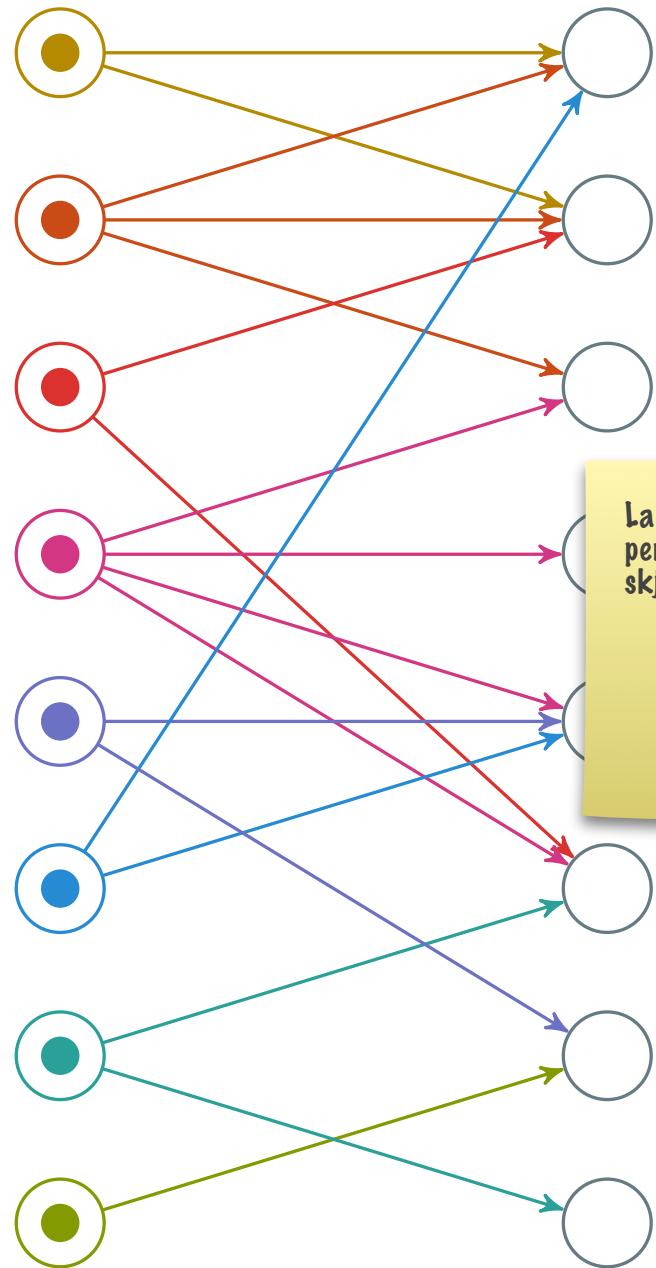
0

100 %

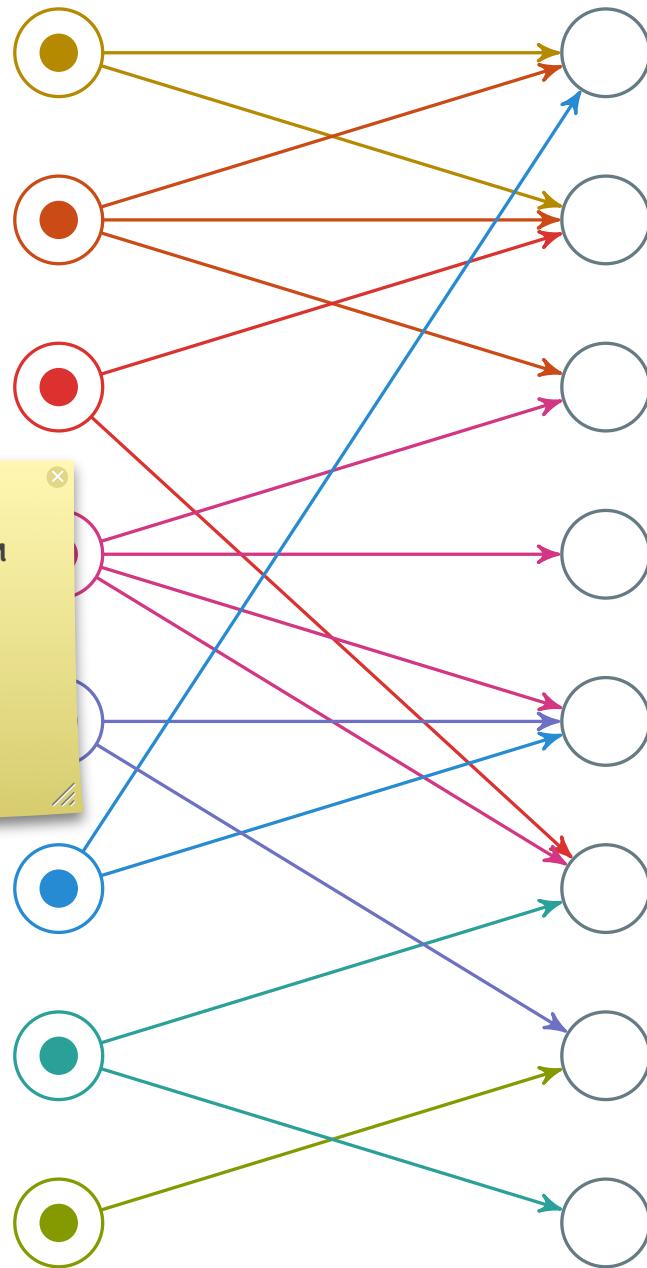
$$n = 4$$

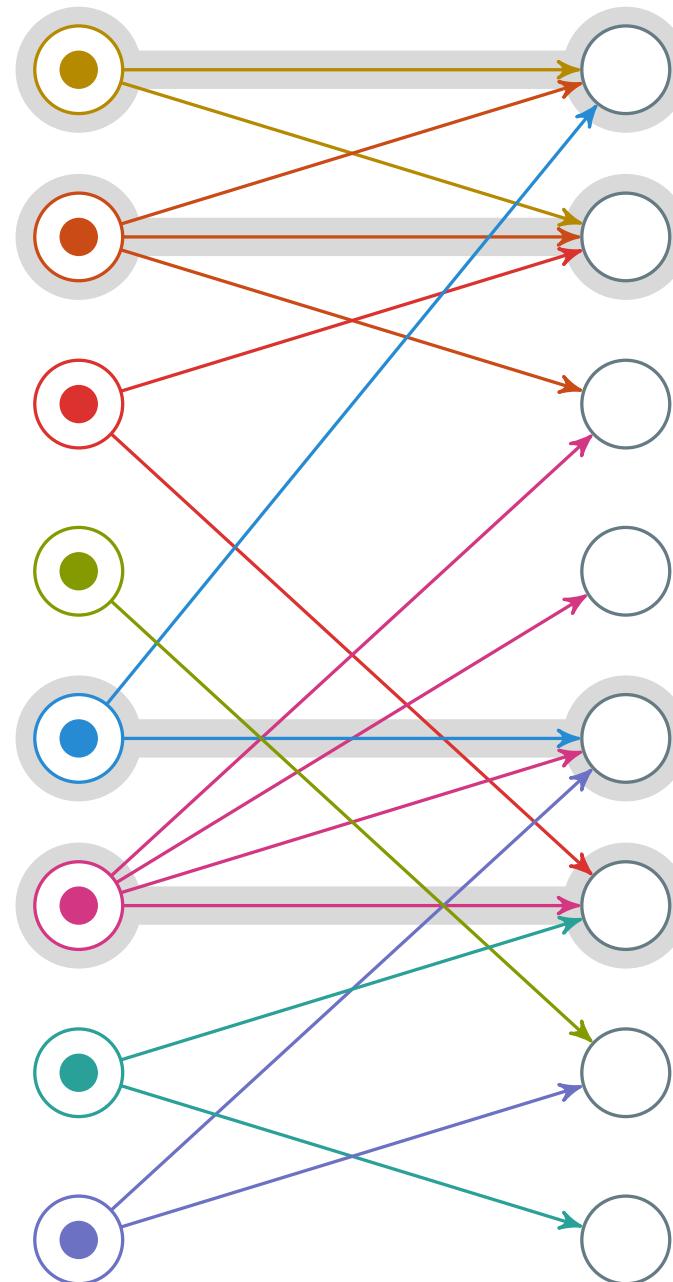
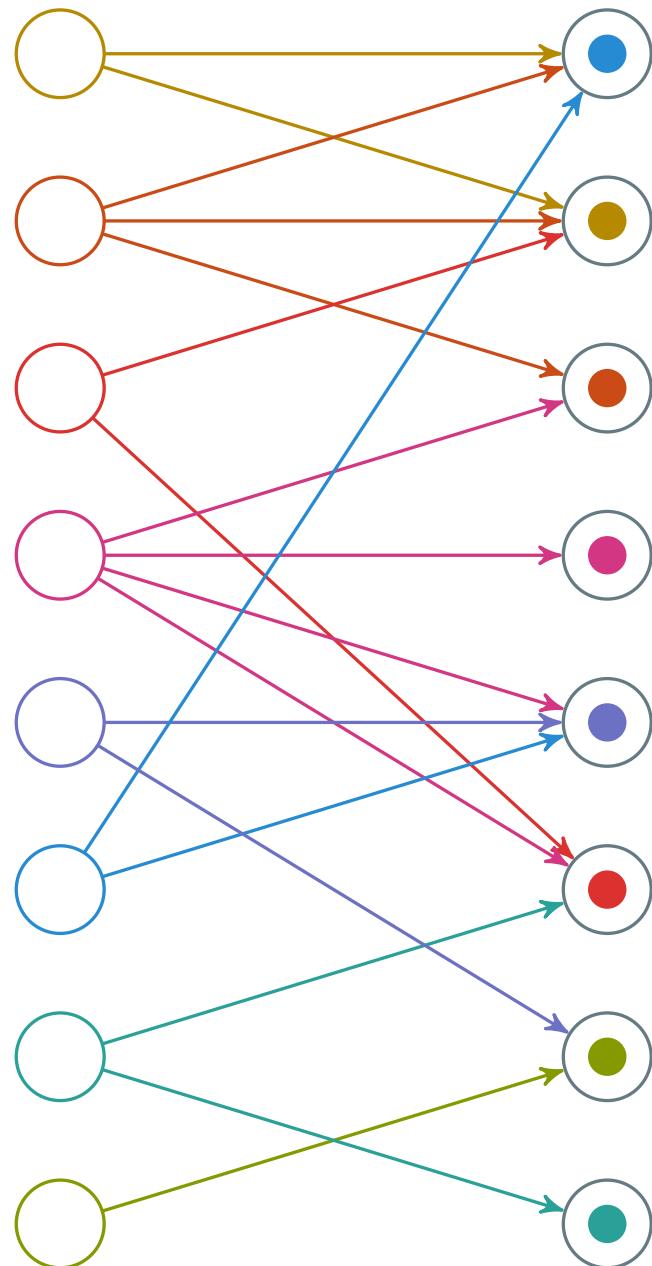
Ford-Fulkerson er litt komplisert - kanskje en enkel brute-force-løsning er like grei, da? Eller?

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



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



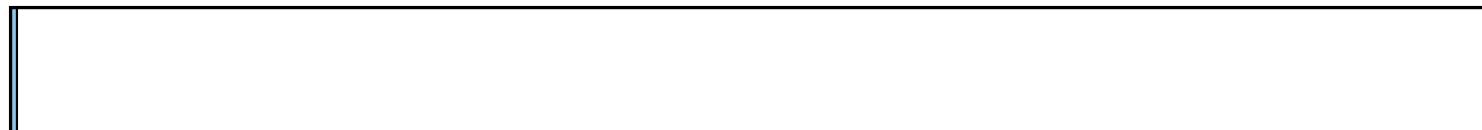


Her trengte Ford-Fulkerson 70 operasjoner, mens det var 40320 permutasjoner å teste.

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...

Reelle tall for USA

- 121 678 på venteliste (2016)
- 17 107 donorer (2014)
- Hvor mange mulige løsninger blir det?
- Med ett mikrosekund per løsning ...
hvor lang tid tar det å teste alle?

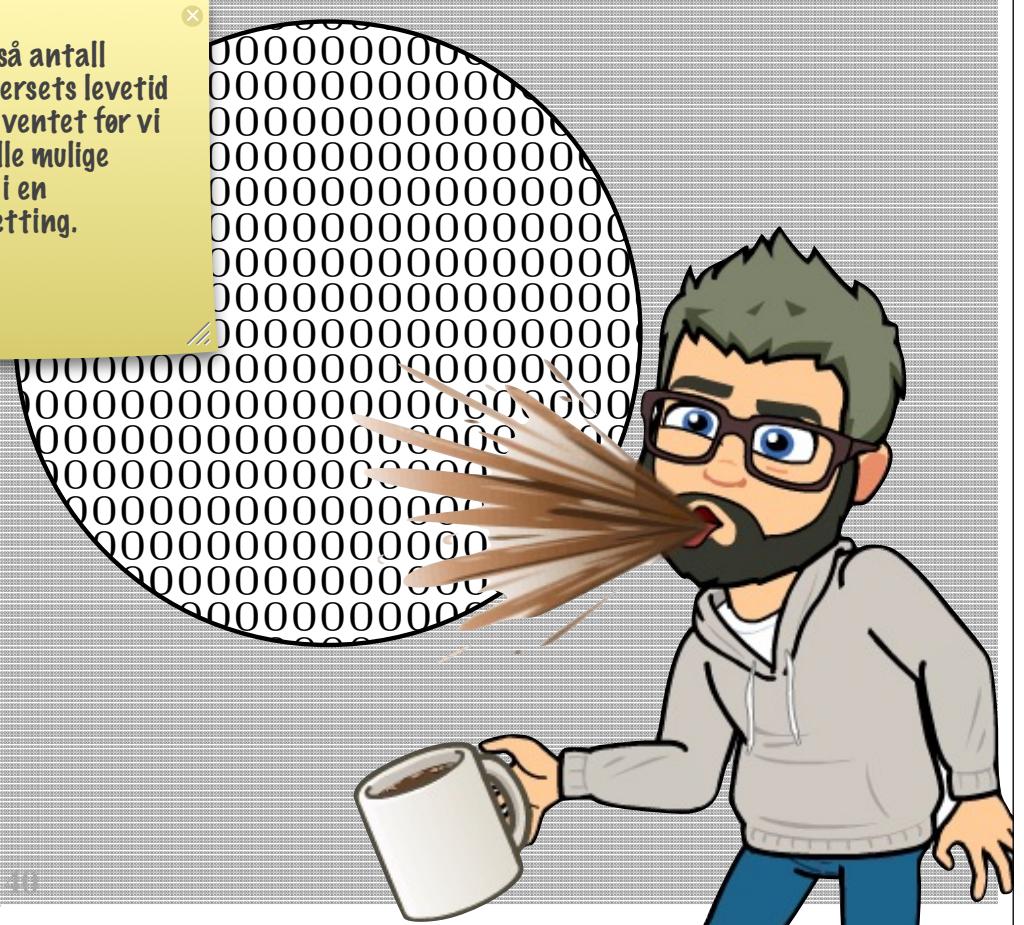
$$\frac{121\,678!}{(121\,678 - 17\,107)!} \mu\text{s} = 1.4 \times 10^{86\,444} \mu\text{s}$$

$$= 4.3 \times 10^{86\,430} \text{ a}$$

$$= 4.3 \times 10^{85\,430} T_U$$

Tid fra The Big Bang til Heat Death

Dette er altså antall ganger universets levetid vi måtte ha ventet før vi fikk prøvd alle mulige matchinger i en realistisk setting.



$$\frac{121\,678!}{(121\,678 - 17\,107)!} \mu s = 1.4 \times 10^{86\,444} \mu s$$

$$= 4.3 \times 10^{86\,430} a$$

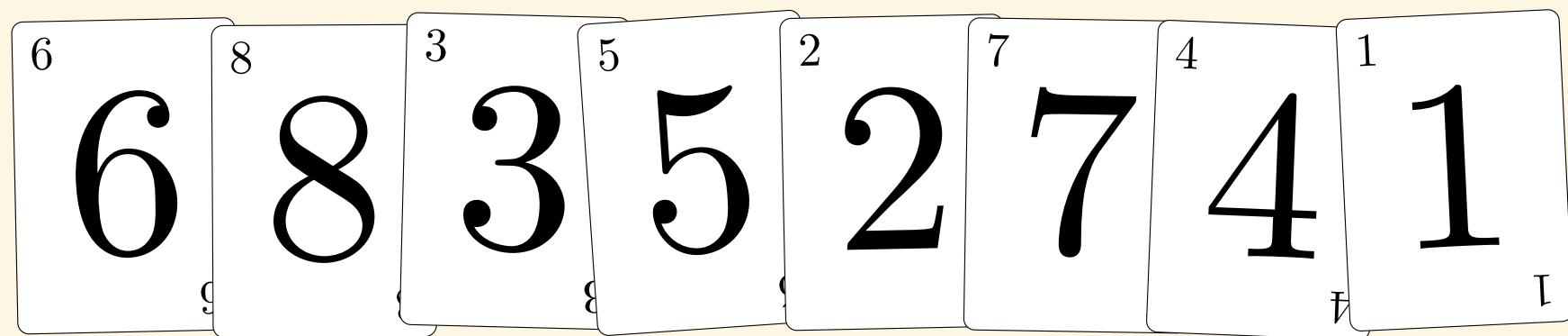
$$= 4.3 \times 10^{85\,430} T_U$$

I dette tilfellet spiller det liten rolle om vi bruker mikrosekunder eller universlevealder som enhet.
Resultatet er uansett bare «ca. uendelig». Vi ønsker å fokusere på de *viktige* forskjellene – mellom effektivt og totalt ubruklig, heller enn mellom mer eller mindre ubruklig. Det er derfor vi ser litt stort på det når vi regner kjøretid.

- 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!

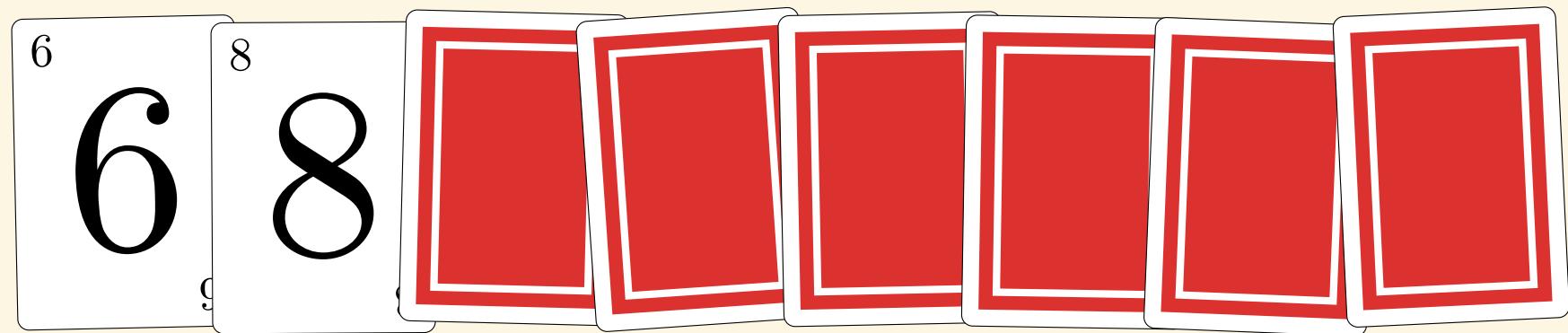
Mulig fremgangsmåte: Vi begynner med å se på bare de første kortene, og så sorterer vi oss gradvis mot høyre.

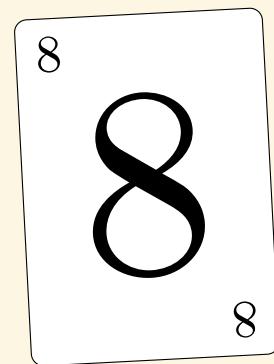
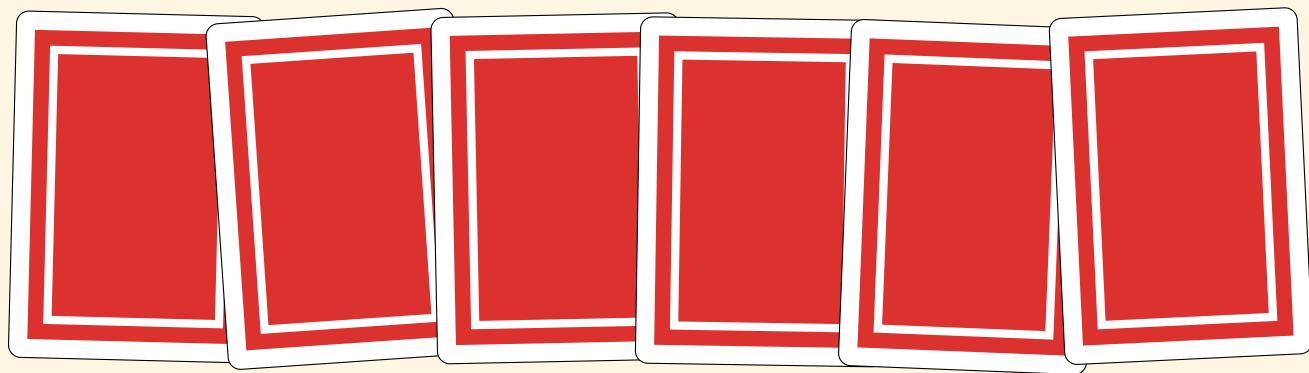
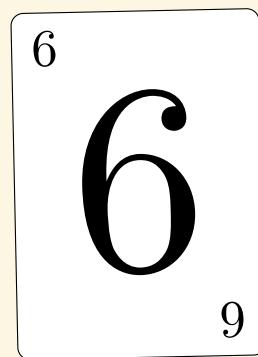


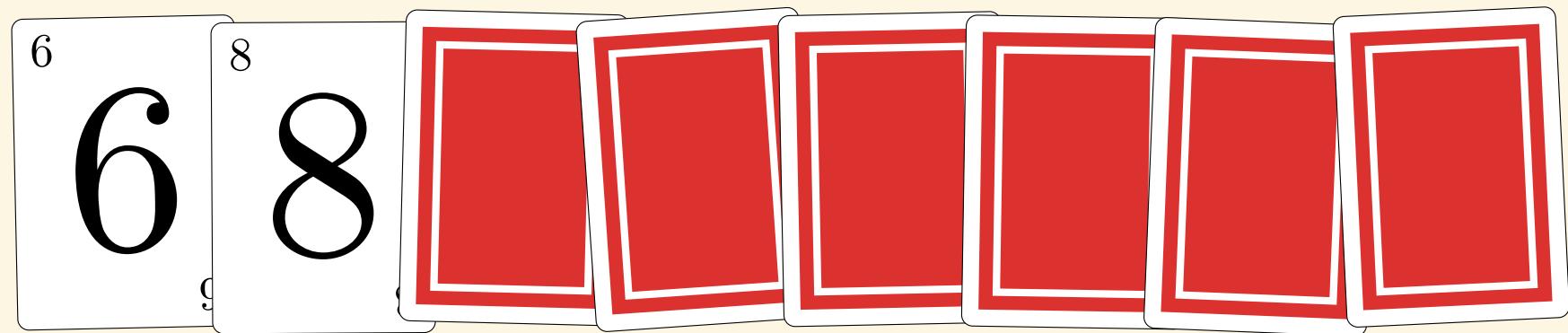
Vi utnytter struktur i problemet - nemlig det at det er mulig å sortere litt etter litt. Dette ignoreres helt i en brute-force-løsning.

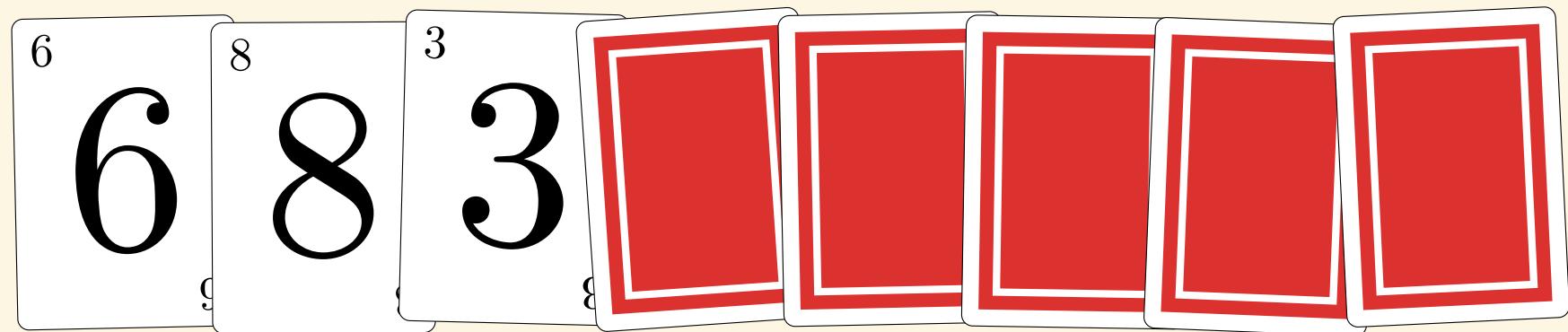
Vi ser er hele tiden bare interessert i å sette inn neste kort.

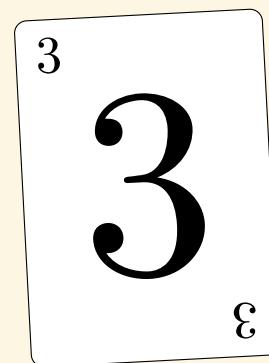
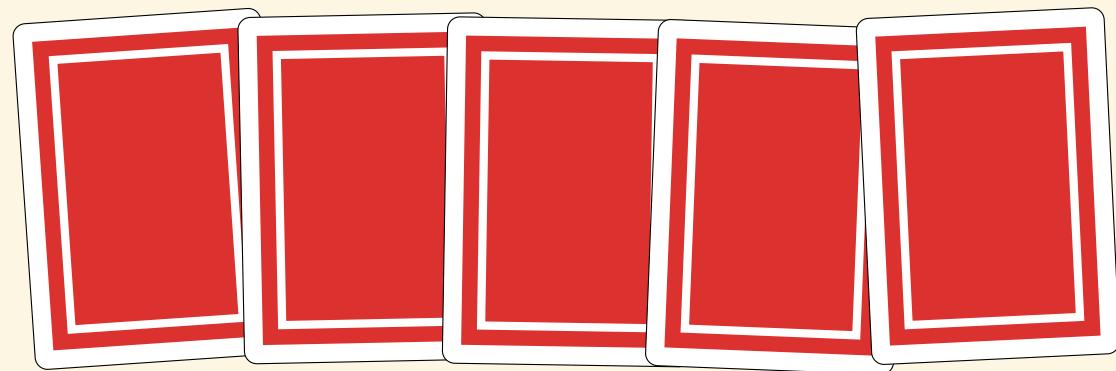
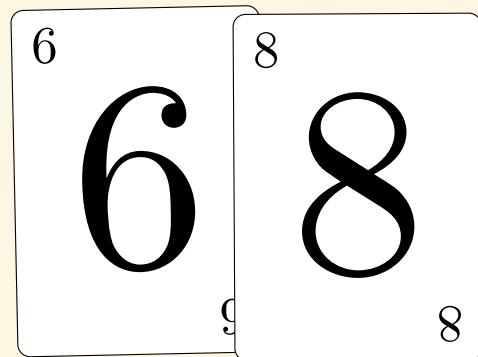
Vi bygger altså trinnvis på forrige del-løsning, uten å begynne fra scratch hele tiden, som i en brute force-løsning.

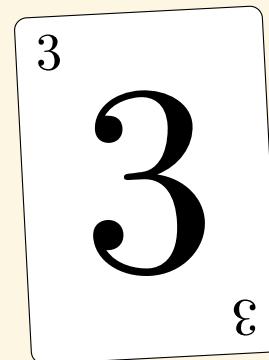
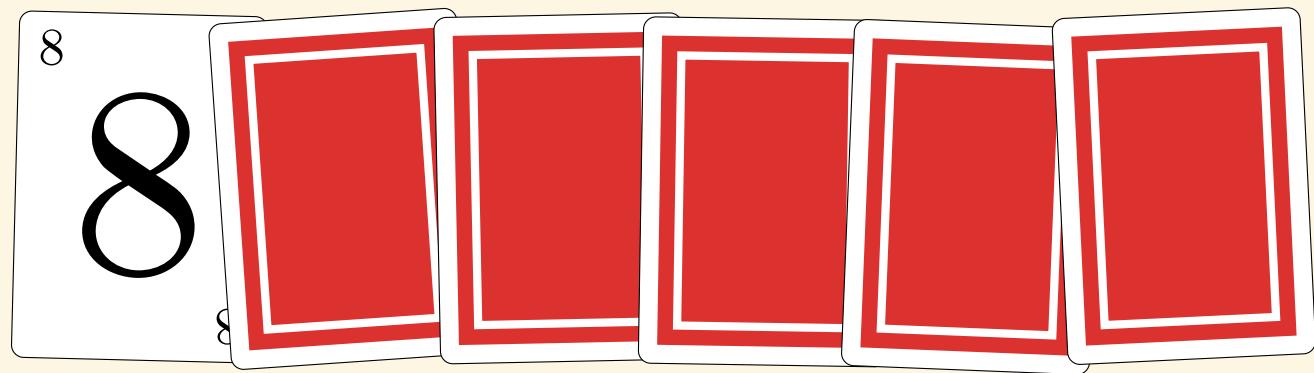
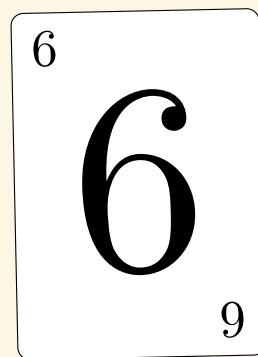


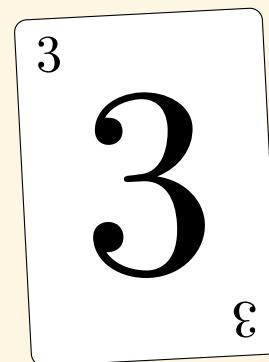
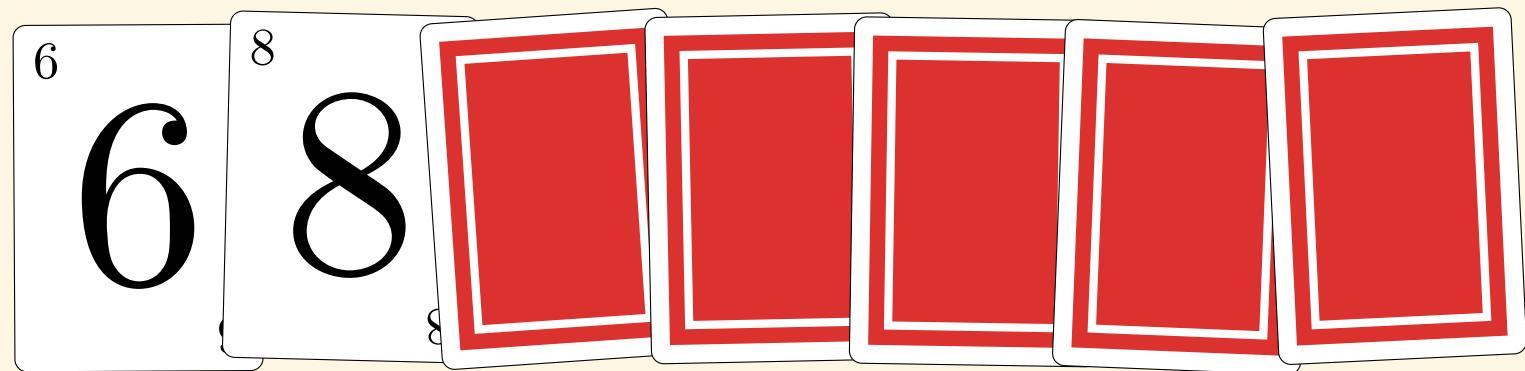


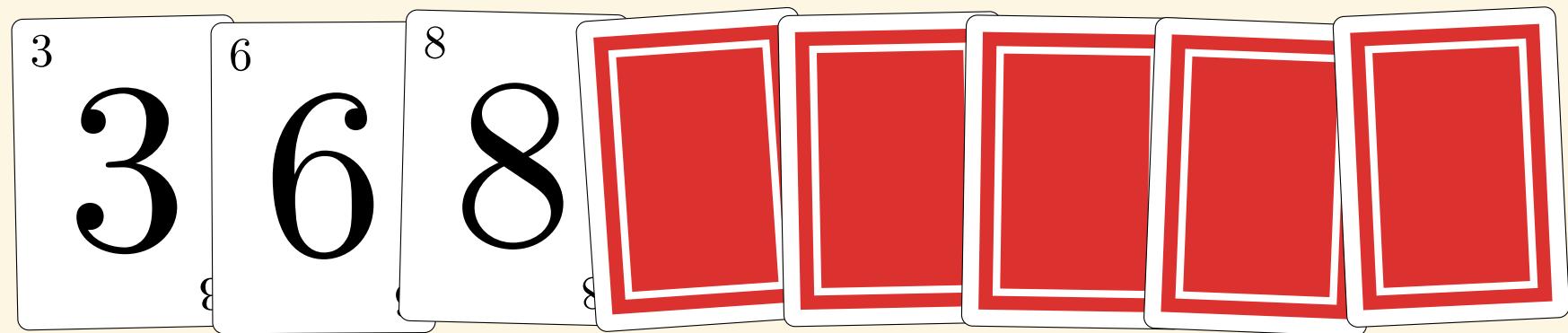


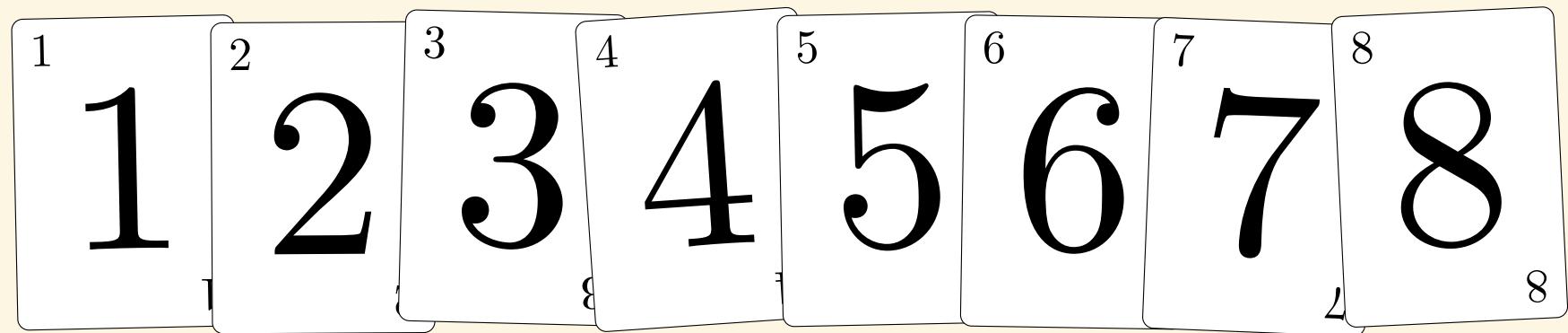












- Er dette bedre enn brute force?
Hvordan kan vi vite eller påstå det?
- Hvordan kom vi frem til denne
algoritmen? Hva er det underliggende
prinsippet?

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 enn 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?

asymptotisk notasjon

Det store tallet vi hadde tidligere hadde en eksponent på 85 430...

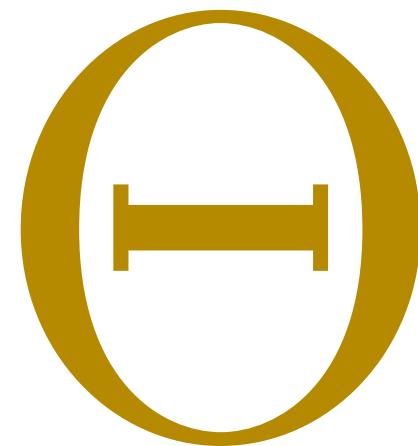
$\lg n$	logaritmisk	$2 \times 10^{949\,978\,419\,116\,565}$
\sqrt{n}	—	1×10^{31}
n	lineær	3×10^{15}
$n \lg n$	linearitmisk	7×10^{13}
n^2	kvadratisk	6×10^7
n^3	kubisk	1×10^5
2^n	eksponentiell	51
$n!$	faktoriell	17

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

$$42 = \Theta(1)$$

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

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

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

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

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

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

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

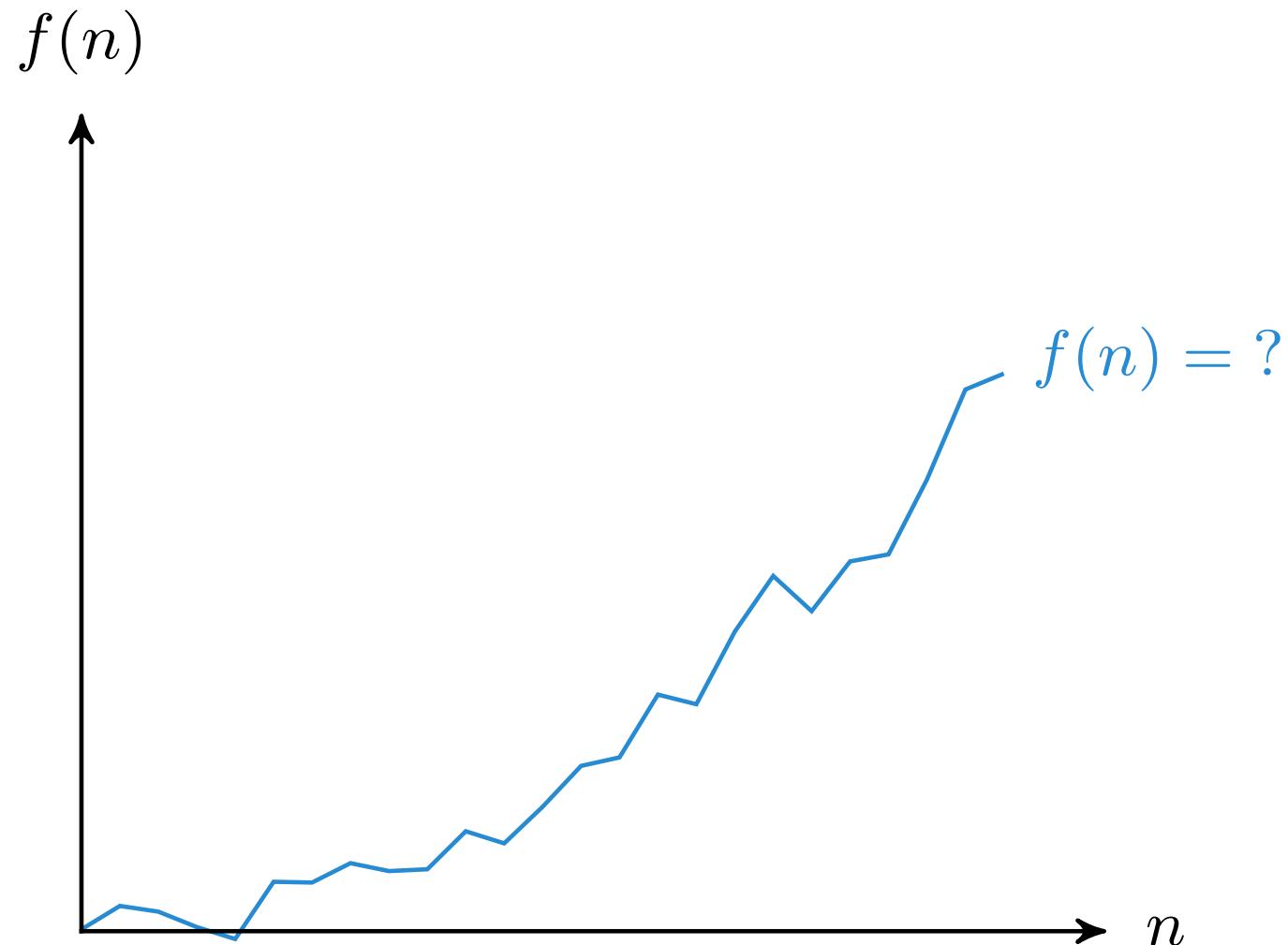
$$n! + 2^n = \Theta(n!)$$

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.

asymptotisk notasjon



Nøyaktig kjøretid kan være hårete, varierende eller udefinert

asymptotisk notasjon

$f(n)$



$f(n) = ?$

n

Vi ønsker å «skissere formen» til funksjonen

asymptotisk notasjon

$f(n)$



n

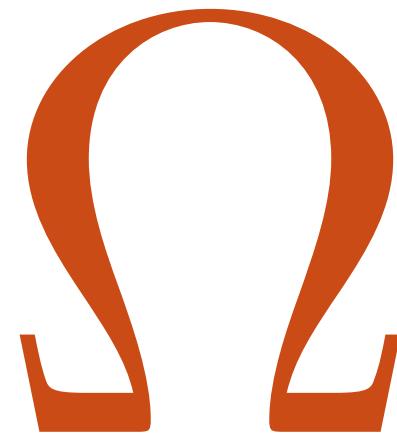
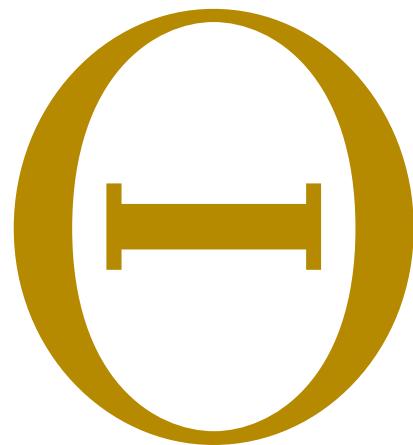
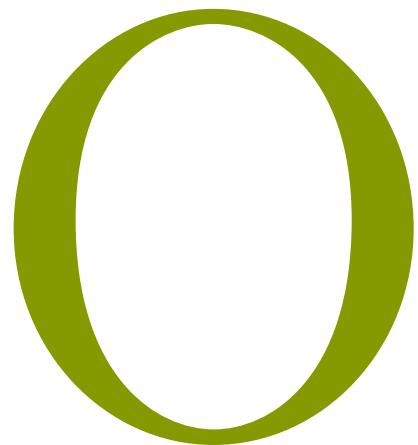
$c_2 n^2$

$\Theta(n^2)$

$c_1 n^2$

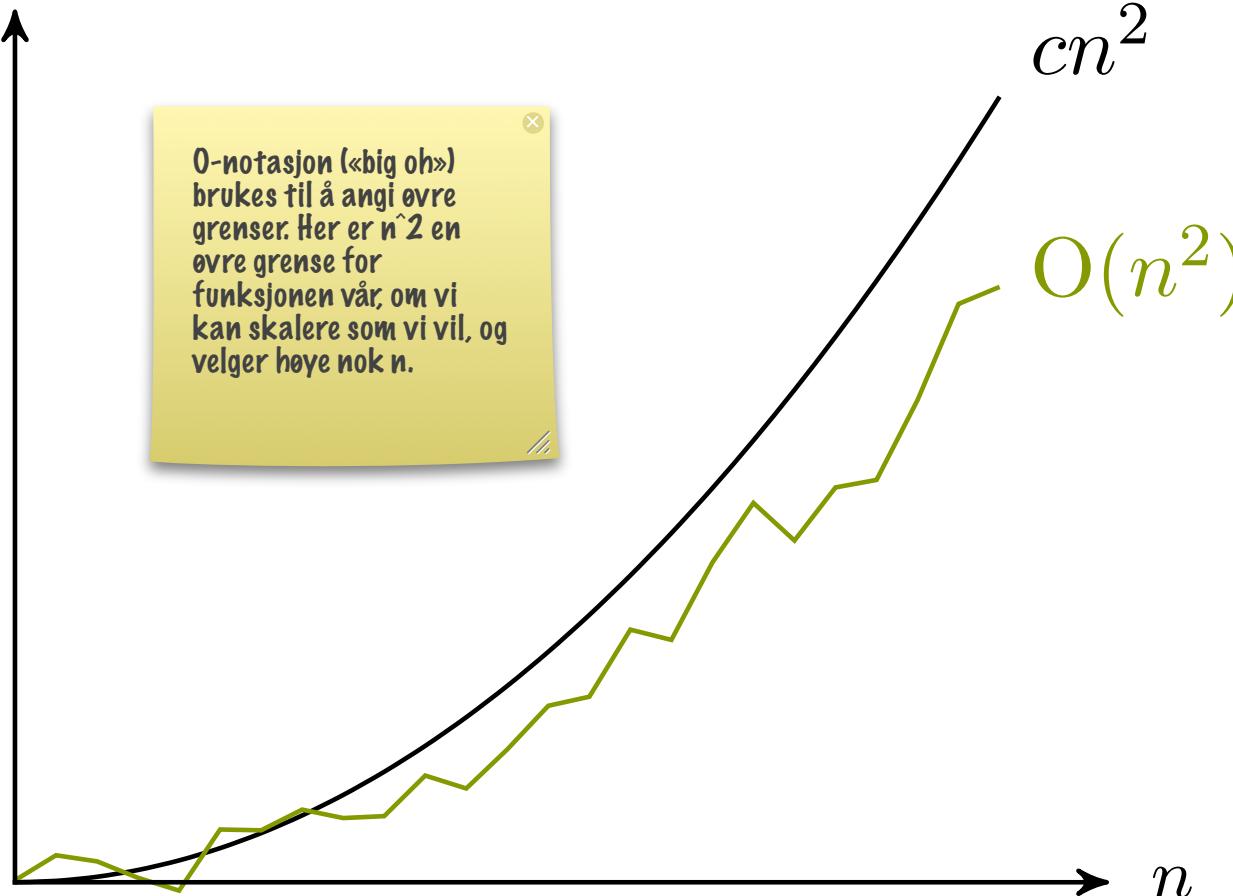
Ligger mellom to skaleringer av samme kurve, for store n

asymptotisk notasjon



O, theta og omega

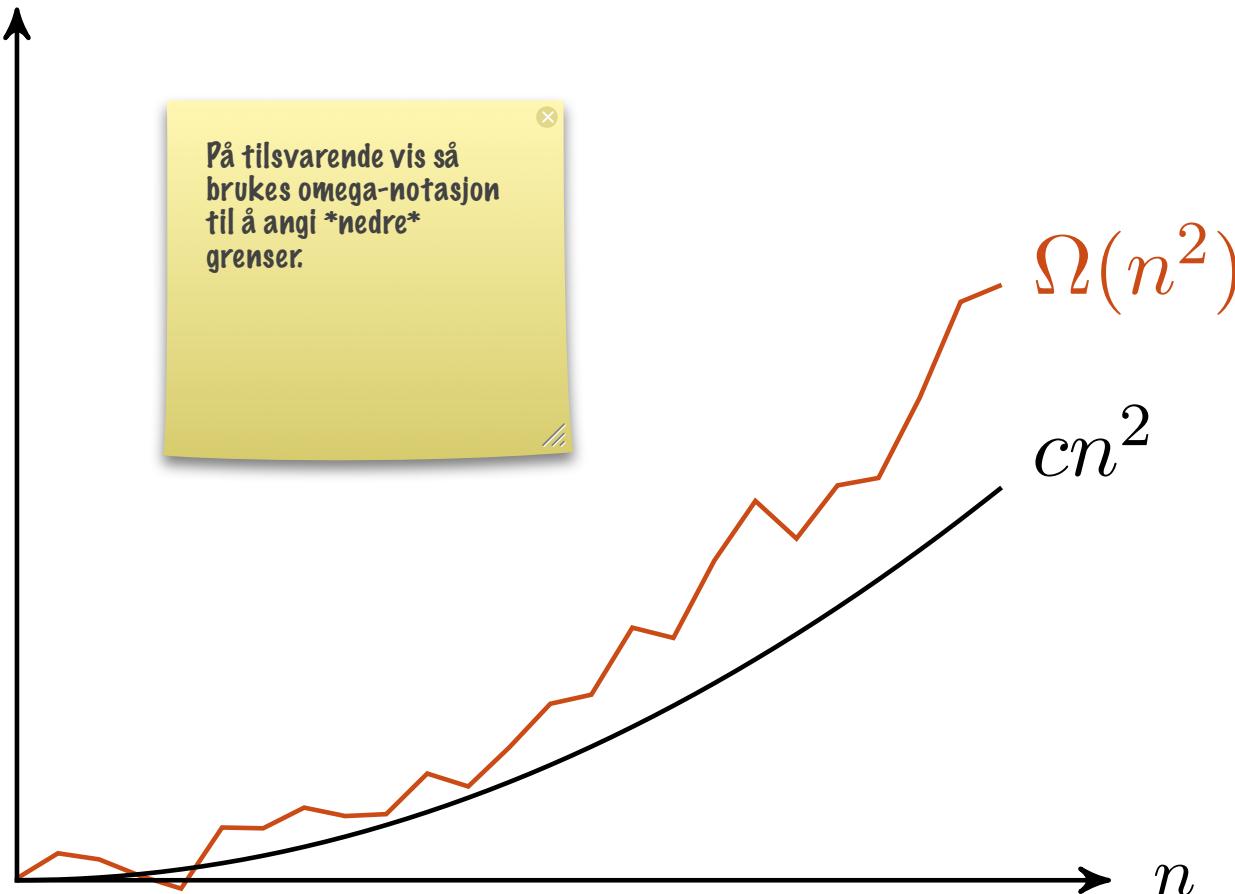
$f(n)$



Ligger under skalert kurve, for store n

asymptotisk notasjon

$f(n)$



Ligger over skalert kurve, for store n

asymptotisk notasjon

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

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

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.

Se også <https://piazza.com/class/isltatih5pc4up?cid=21>

sie auch setzen darf

$$\tau(n) = \sum_{k=1}^n \left[\frac{n}{k} \right].$$

Nun ist aber $\left[\frac{n}{k} \right] = \frac{n}{k} - \varrho$, wo ϱ einen Werth zwischen 0 und 1 bedeutet, und demgemäß wird

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

r sicher den Werth n nicht erreicht. Da nun nach der Euler'schen Summenformel (1a) des vor. Abschnitts

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

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

wir durch das Zeichen $O(n)$ eine Grösse ausdrücken, Ordnung in Bezug auf n die Ordnung von n nicht übersteht; ob sie wirklich Glieder von der Ordnung n in sich bleibt bei dem bisherigen Schlussverfahren dahingestellt. Vor wir dies näher untersuchen, betrachten wir zweitens die Funktion $S(n)$, welche die Summe aller Theiler ganzen Zahl n bestimmt. Für die entsprechende

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

ir in Nr. 2, 3) des 11. Abschnittes den anderen Ausdruck

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

erner mit Hilfe der allgemeinen Analytischen Zahlentheorie.

Fra «Die analytische Zahlentheorie» av P. G. H. Bachmann (1894).
<https://archive.org/details/dieanalytischez00bachgoog>

sonst aber nur verschieden ist, so daß alle von Nullstellen dem Streifen $0 \leq \sigma \leq 1$ angehören

Riemann vermutete ferner, ohne mehr als heuristische Gründe für die Richtigkeit aufführen zu können, daß die Zetafunktion folgende sechs Eigenschaften besitzt:

I. Es gibt unendlich viele Nullstellen von $\zeta(s)$ im Streifen $0 \leq \sigma \leq 1$, die natürlich symmetrisch zur reellen Achse und auch nach der Funktionalgleichung symmetrisch zur Geraden $\sigma = \frac{1}{2}$ verteilt liegen.

II. Wenn für $T > 0$ unter $N(T)$ die Anzahl¹⁾ der Nullstellen (mehrfache mehrfach gezählt) verstanden wird, deren Ordinate zwischen 0 (exkl.) und T (inkl.) liegt, so ist

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

Hierbei verstehe ich unter der Bezeichnung $O(\log T)$ eine Funktion von T , deren Quotient durch $\log T$ absolut genommen für alle T von einer gewissen Stelle an unterhalb einer festen Schranke liegt. Ich verstehe allgemein, wenn $g(x)$ eine für alle reellen x von einem gewissen Werte an definierte und positive Funktion von x ist und $f(x)$ eine von einem gewissen reellen x an definierte reelle oder komplexe Funktion von x , unter der Schreibweise

$$f(x) = O(g(x))$$

(sprich: O von $g(x)$), daß

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

endlich ist, d. h., daß es zwei Zahlen ξ und A gibt, für welche bei allen $x \geq \xi$

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

ik

praktisch

0 ≤

225

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

da es sich wegen der Beschränkung auf die ersten Ebene handelt.

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

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

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

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

Som før, men for *alle* $c > 0$

Disse er strengere
varianter av de store
operatorene.

ω >

Ω ≥

Θ =

O ≤

o <

Klasser av input

Best, verst og forventet

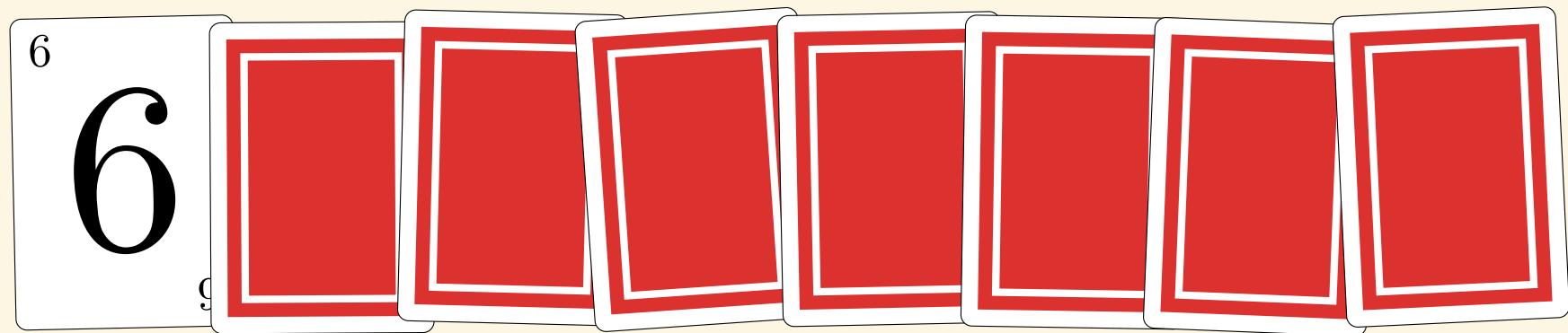
- **Kjøretid:** Funksjon av problemstørrelse
- **Best-case:**
Beste mulige kjøretid for en gitt størrelse
- **Worst-case:** Verste mulige
- **Average-case:**
Forventet, gitt en sannsynlighetsfordeling
- Bruker vanligvis worst-case

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 ...

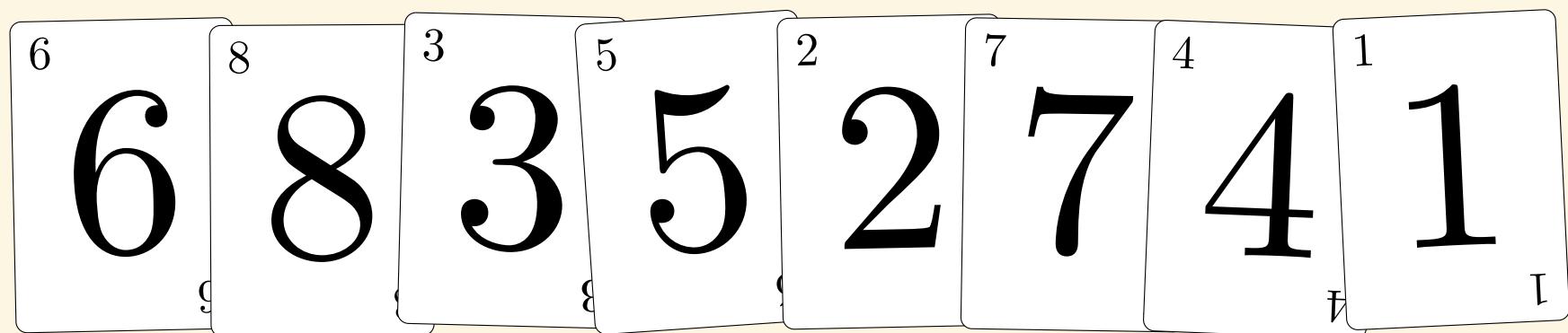


8

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

asymptotisk notasjon › eksempel › brute force

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

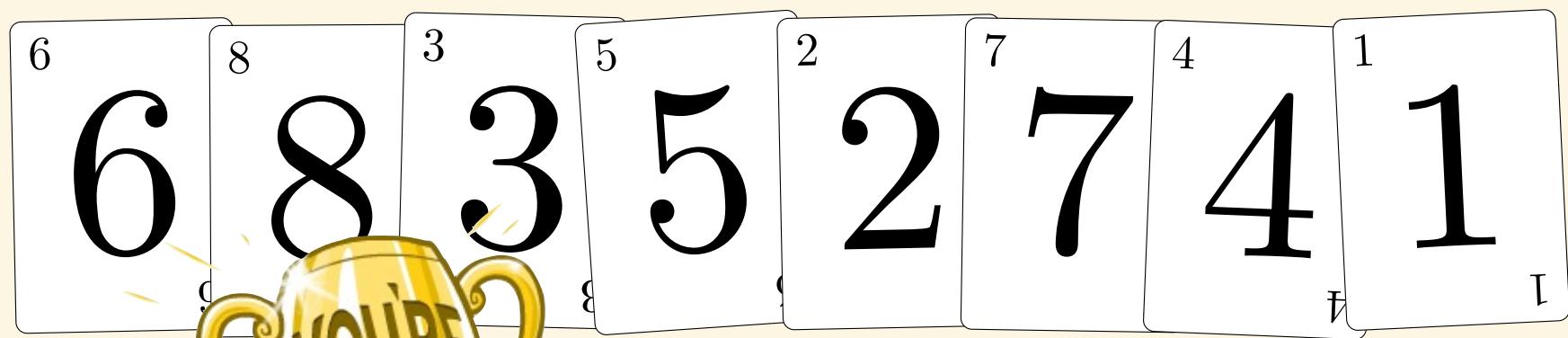


$$8 \times 7 \times 6 \times 5 \times 4 \times 3 \times 2 \times 1$$

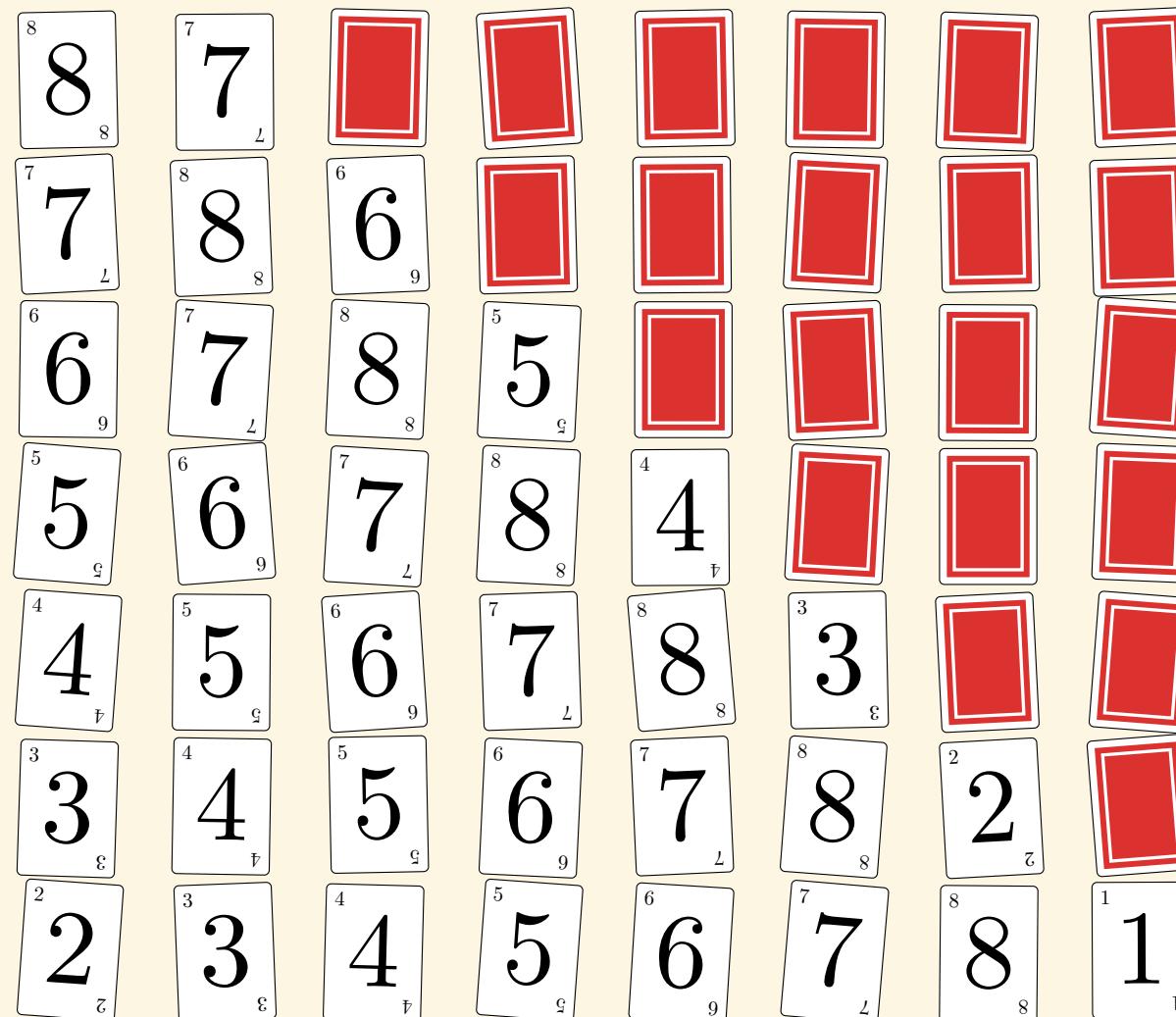
asymptotisk notasjon › eksempel › brute force

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

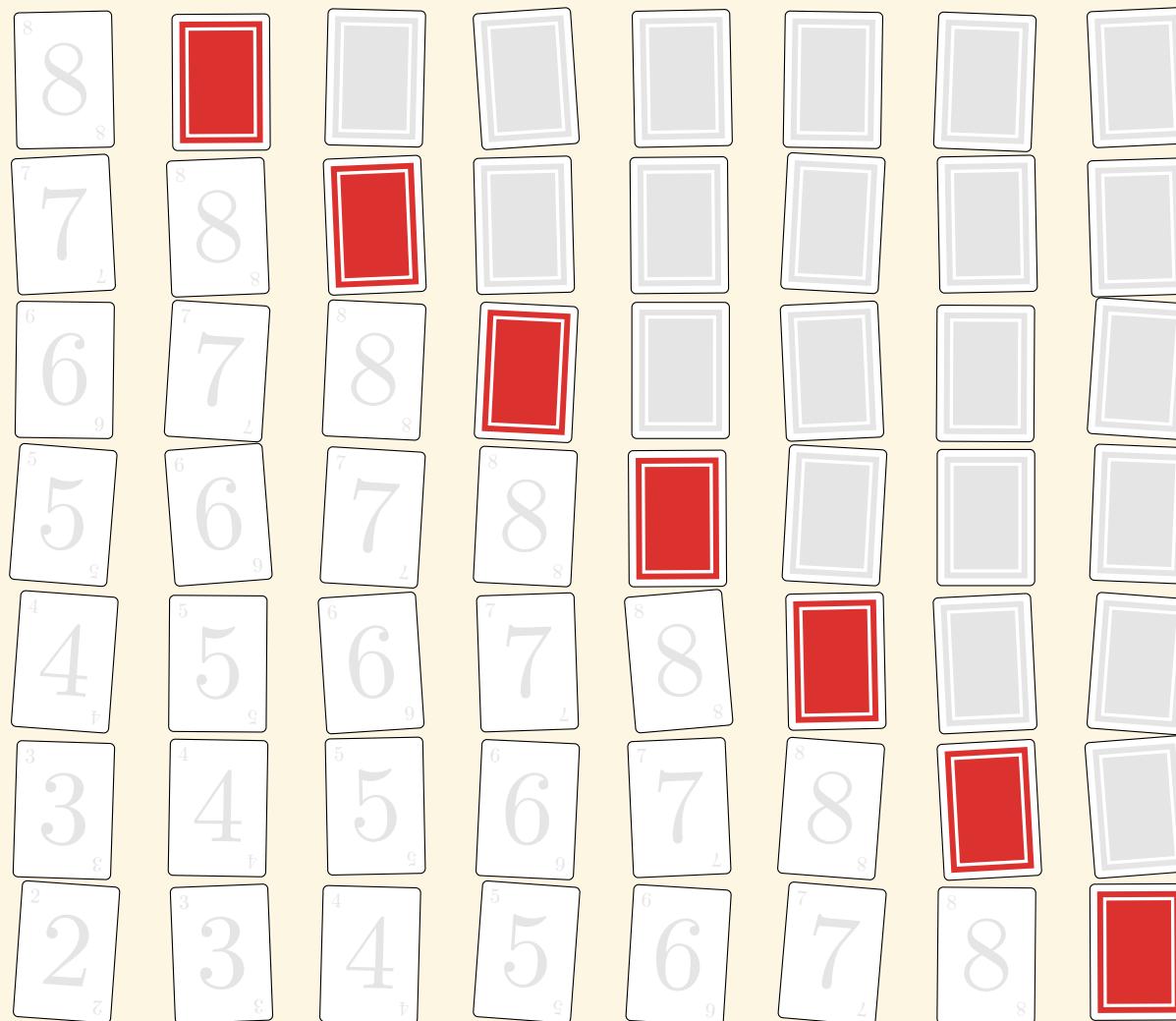
Og den er noe av det
verste vi kan komme
borti ...



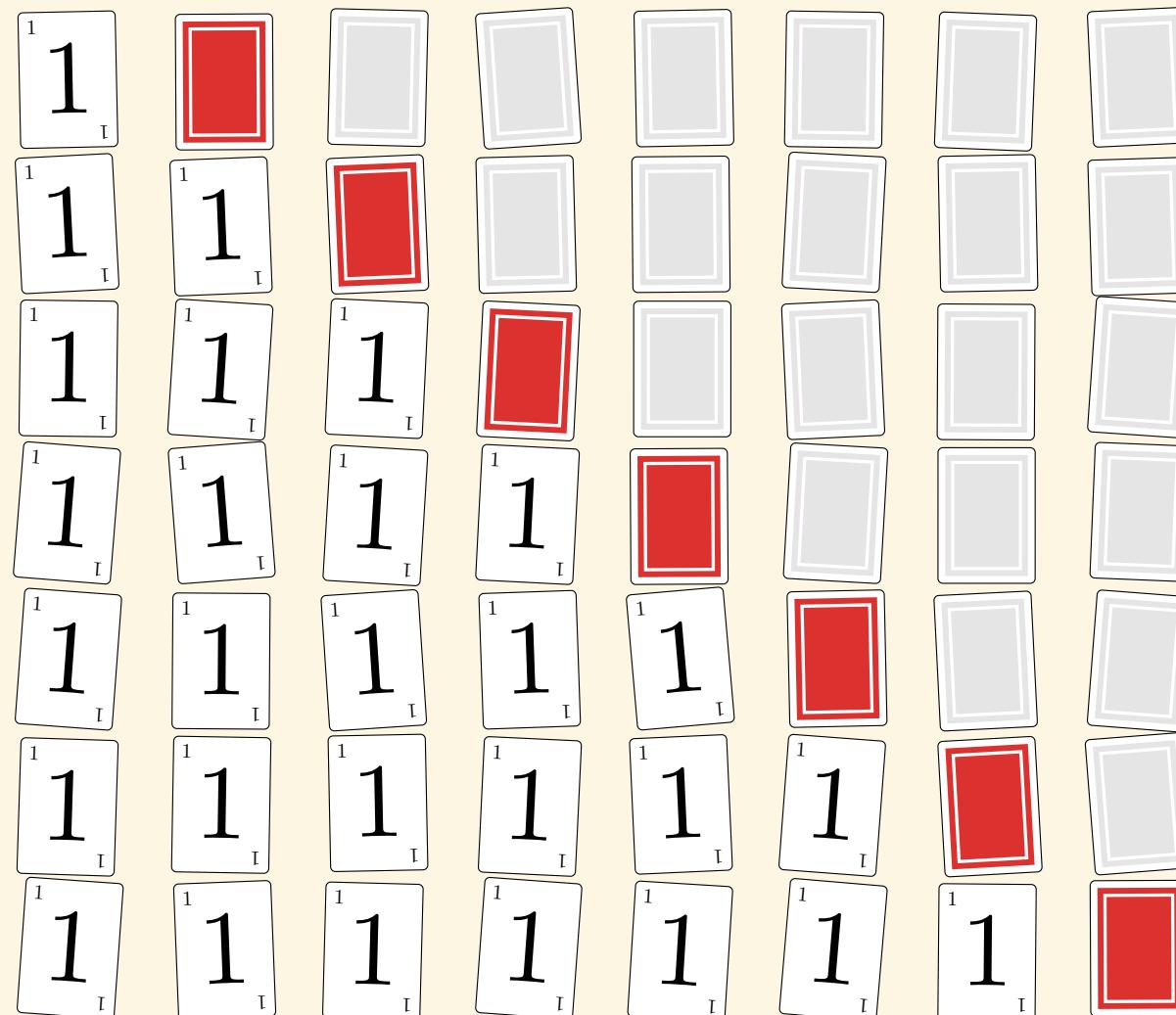
Så, sortering ved innsetting ...



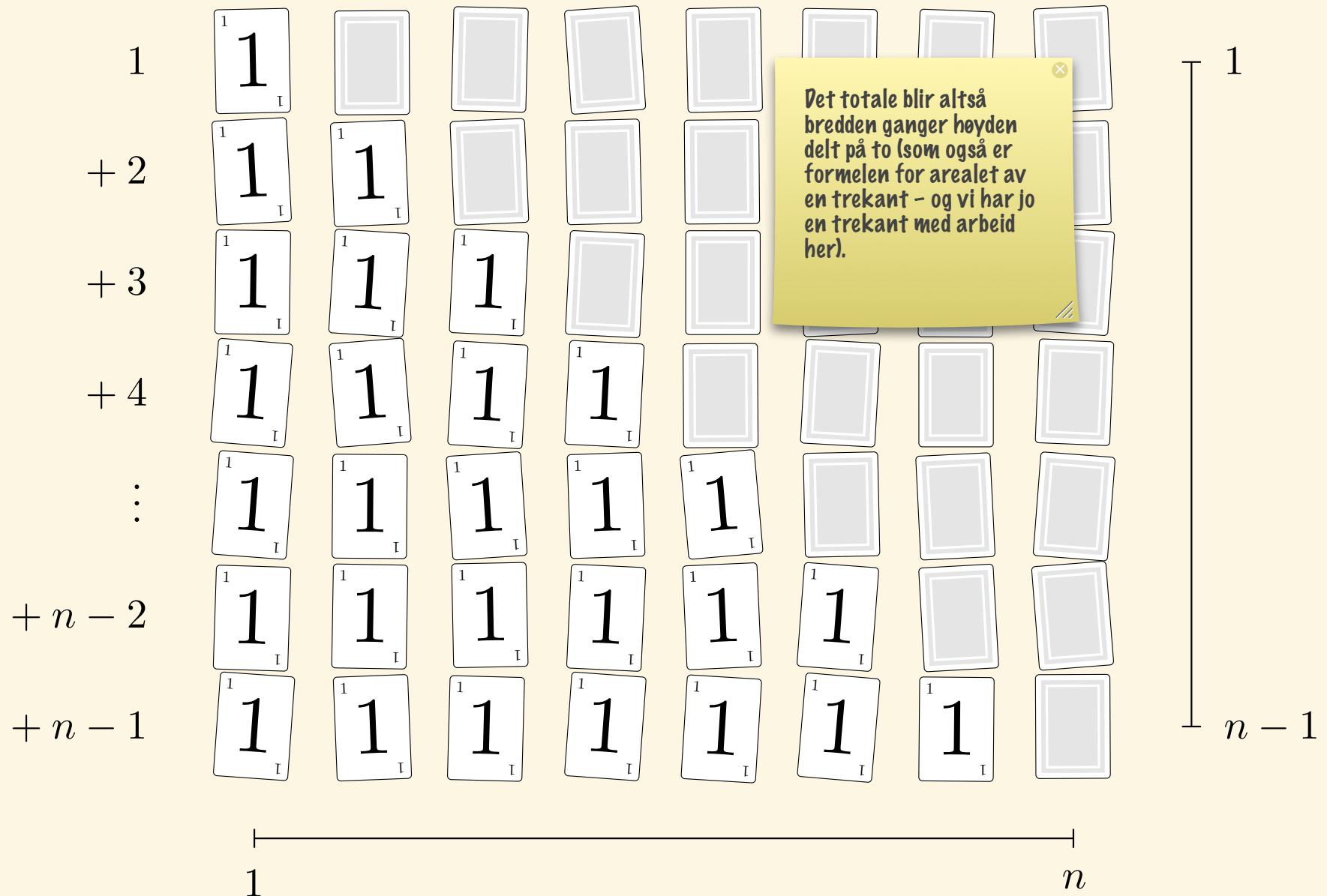
Verste tilfelle: $A = \langle n, \dots, 1 \rangle$. Hver rad er starten på en iterasjon



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



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



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

- En invariant er en egenskap som ikke endres under kjøring
- Initialisering: Vis at invarianten er sann før start
- Vedlikehold
 - Induktivt premiss: Anta at den er sann før en iterasjon
 - Induksjonstrinn: Vis at den er 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»

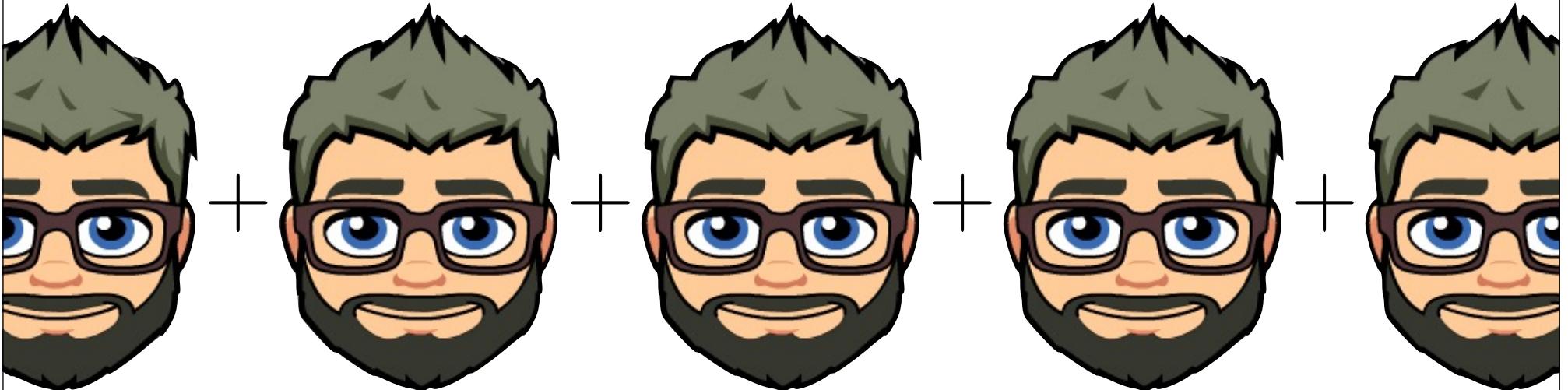
Dette er selvfølgelig ikke
den eneste varianten vi
kan velge – men den
enkleste til å begynne
med.



I begge tilfeller

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

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



4:5

Eksempel: Sum

Sum Rekursiv

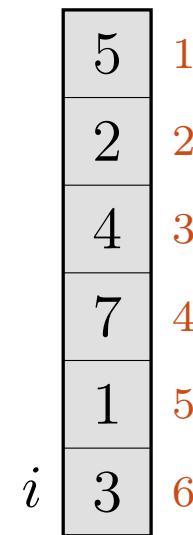
Dekomponering

- Vi vil summere elementene i en tabell
- Rekursjon: Summér alle unntatt det siste
- Induktivt premiss: Anta at dette blir rett
- Induksjonstrinn: Legg til siste element

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

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

$tmp = -$



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

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

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

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

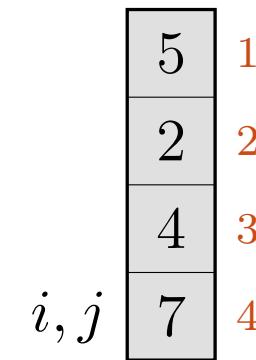
→ 18

$tmp = 11$

```
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



**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.**

Vere 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)

```
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       $A[i + 1] = key$ 
```

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       $A[i + 1] = key$ 
```

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
6	4
1	5
3	6

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

```
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

**Vi har jo egentlig allerede sett at kjøretiden
er kvadratisk – men vi kan se litt mer
detaljert på kostnaden for hver bit av
algoritmen.**

INSERTION-SORT(A)

1	for $j = 2$ to $A.length$	n
2	$key = A[j]$	$n - 1$
3	$i = j - 1$	$n - 1$
4	while $i > 0$ and $A[i] > key$	$\sum_{j=2}^n t_j$
5	$A[i + 1] = A[i]$	$\sum_{j=2}^n (t_j - 1)$
6	$i = i - 1$	$\sum_{j=2}^n (t_j - 1)$
7	$A[i + 1] = key$	$n - 1$

$$T(n) = \Theta(n^2)$$

	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-case: Alt er allerede på plass, så vi trenger ikke flytte noen ting. Vi trenger bare å gjennom tabellen én gang, og se at alt er der det skal.

Average-case: Intuitivt kan vi tenke oss at vi må flytte hvert element halvparten av veien det må gå i worst-case. Det gir oss halvparten av worst-case, og konstantfaktorer bryr vi oss ikke om.

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:

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

