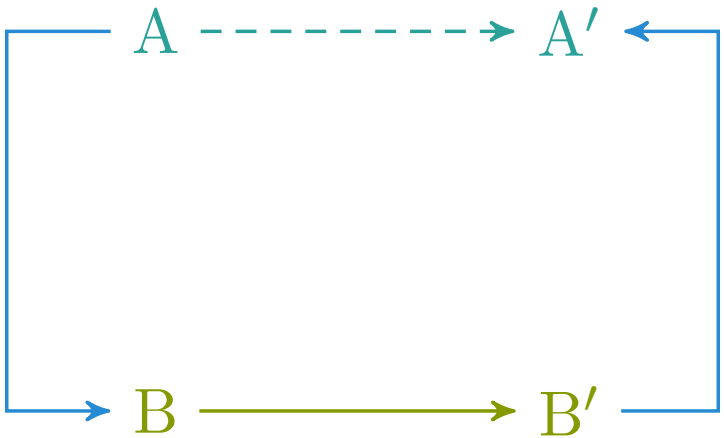
# Forelesning 4

## Rangering i lineær tid

# Strukturell induksjon

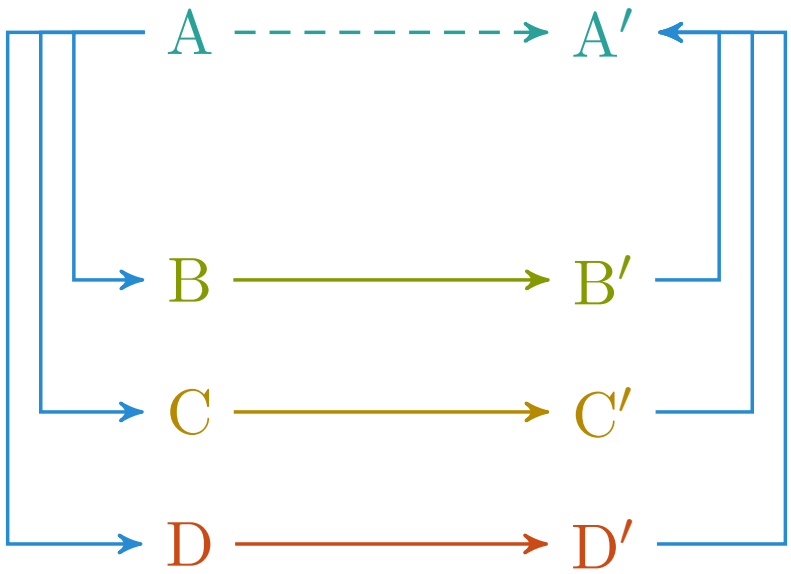
- Veldig likt det vi har gjort så langt
- I stedet for at induksjonshypotesen gjelder lavere heltall ...
- ... så gjelder den «komponenter»
- En komponent kan være så mangt, så lenge «del-av»-relasjonen tilsvarer et endelig tre

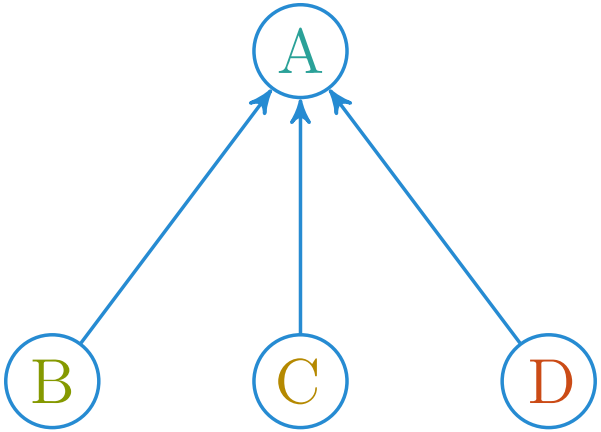
# Dekomponering og reduksjon



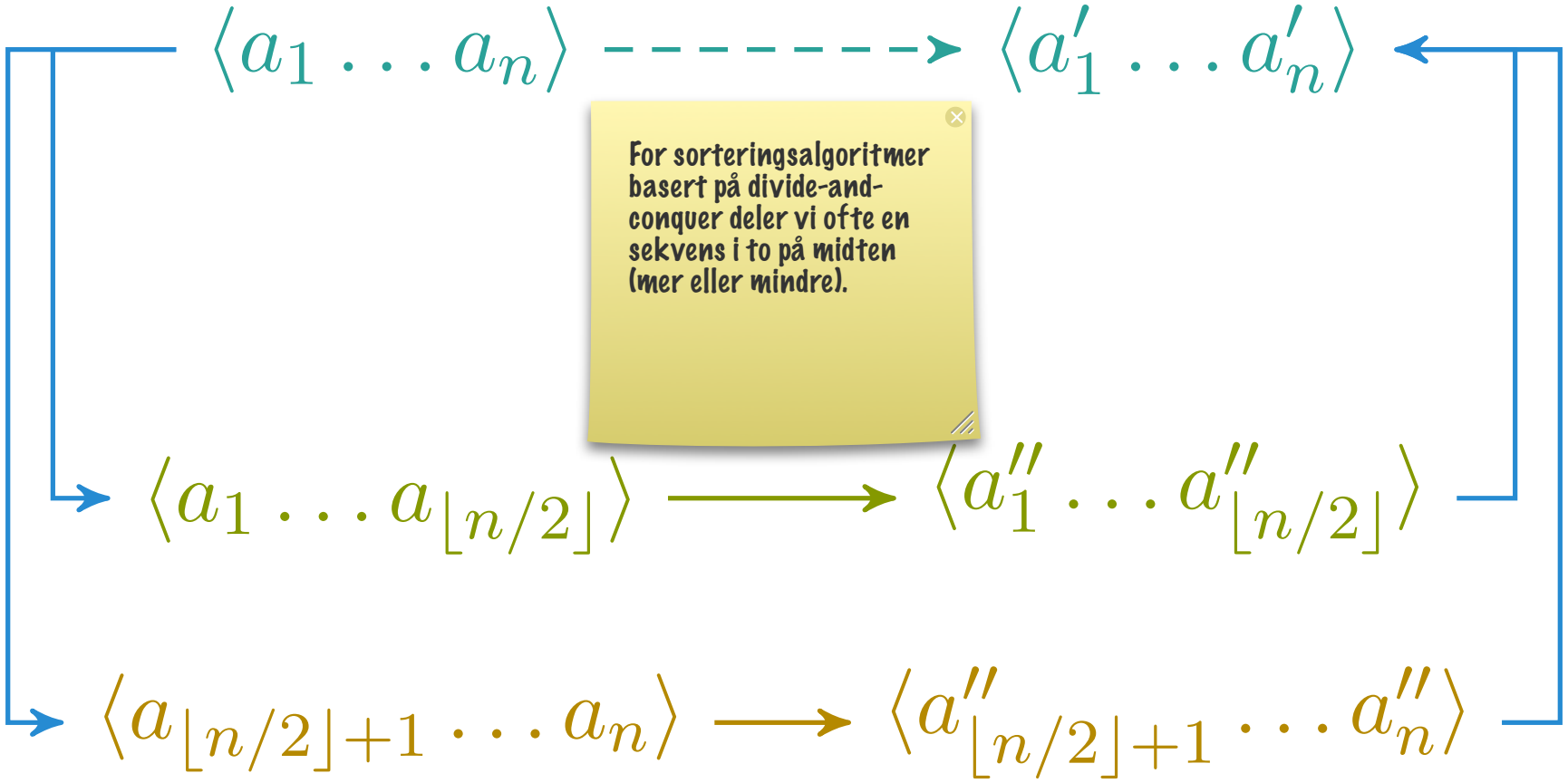

Vi kan transformere A-input til B-input, løse B, og så transformere B-output til A-output. Vi har da en løsning på A, og løser egentlig A *ved hjelp av *B*.


En «dekomponering» der vi bare har én «komponent» kalles gjerne en *reduksjon*. Man kan også redusere til helt andre problemer – og altså transformere input/output for ett problem til input/output for et helt annet.




Vi kan også dekomponere i flere problemer, naturligvis.


$\langle a_1 \ldots a_n \rangle$
$\langle a'_1 \ldots a'_n \rangle$
For sorteringsalgoritmer basert på divide-and-conquer deler vi ofte en sekvens i to på midten (mer eller mindre).
$\langle a_1 \ldots a_{\lfloor n/2 \rfloor} \rangle$
$\langle a''_1 \ldots a''_{\lfloor n/2 \rfloor} \rangle$
$\langle a_{\lfloor n/2 \rfloor+1} \ldots a_n \rangle$
$\langle a''_{\lfloor n/2 \rfloor+1} \ldots a''_n \rangle$

$$\sum_{i=1}^{n} x_i = \sum_{i=1}^{\lfloor n/2 \rfloor} x_i + \sum_{i=\lfloor n/2 \rfloor + 1}^{n} x_i$$

Vi kan sette opp dekomponering som en prosedyre som avgjør om vi kan finne rett svar for en gitt propblemstørrelse - eller for et gitt delproblem.

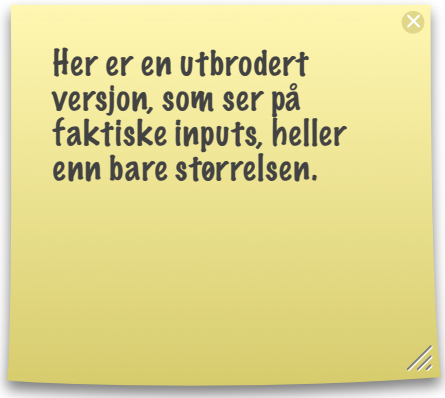
```
CORRECT(n)
1  if n > 1
2        m = ⌊n/2⌋
3        c1 = CORRECT(m)
4        c2 = CORRECT(n − m)
5        return c1 and c2
6  else return TRUE
```

Vi prøver naturligvis å sette opp denne så den alltid returnerer TRUE, men *poenget* er å se på hvilken struktur prosedyren har.

Det vil si: Hvilke rekursive kall gjør den? Det er dem vi baserer oss på når vi tenker induktivt (eller rekursivt), og strukturen gjenspeiles i den faktiske algoritmen.

```
CORRECT(n)
1  if n > 1
2        m = ⌊n/2⌋
3        c1 = CORRECT(m)
4        c2 = CORRECT(n − m)
5        return c1 and c2
6  else return TRUE
```

```
CORRECT(A, p, r)
1  if p < r
2        q = ⌊(p + r)/2⌋
3        c1 = CORRECT(A, p, q)
4        c2 = CORRECT(A, q + 1, r)
5        return c1 and c2
6  else return TRUE
```



```
CORRECT(A, p, r)
1  if p < r
2        q = ⌊(p + r)/2⌋
3        c1 = CORRECT(A, p, q)
4        c2 = CORRECT(A, q + 1, r)
5        return c1 and c2
6  else return TRUE
```

```
SUM(A, p, r)
1  if p < r
2        q = ⌊(p + r)/2⌋
3        x = SUM(A, p, q)
4        y = SUM(A, q + 1, r)
5        return x + y
6  else return A[p]
```

Her har vi fylt inn det induktive trinnet – dvs. selve innholdet i den rekursive funksjonen. Ellers er algoritmen helt likt med den generiske prosedyren som uttrykker dekomponeringen.

```
SUM(A, p, r)
1   if p < r
2         q = ⌊(p + r)/2⌋
3         x = SUM(A, p, q)
4         y = SUM(A, q + 1, r)
5         return x + y
6   else return A[p]
```

```
TIME(A, p, r)
1   if p < r
2         q = ⌊(p + r)/2⌋
3         t_1 = TIME(A, p, q)
4         t_2 = TIME(A, q + 1, r)
5         return t_1 + t_2 + 1
6   else return 1
```

```
TIME(A, p, r)
1   if p < r
2         q = ⌊(p + r)/2⌋
3         t_1 = TIME(A, p, q)
4         t_2 = TIME(A, q + 1, r)
5         return t_1 + t_2 + 1
6   else return 1
```

```
TIME(n)
1   if n > 1
2         m = ⌊n/2⌋
3         t_1 = TIME(m)
4         t_2 = TIME(n − m)
5         return t_1 + t_2 + 1
6   else return 1
```

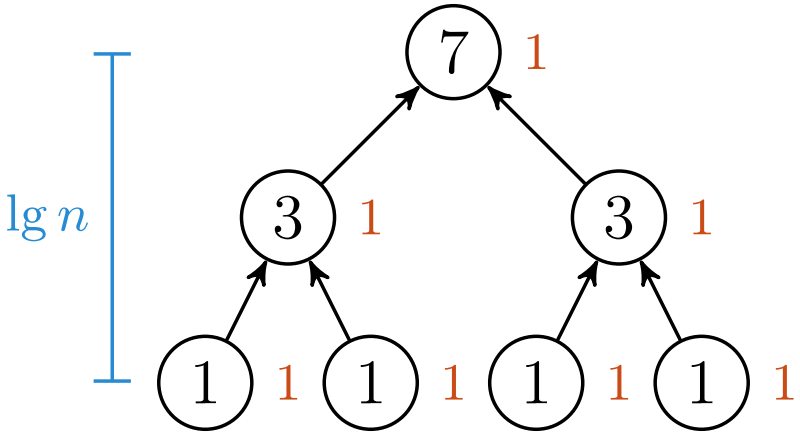
$$T(n) = 2 \cdot T(n/2) + 1$$

$$T(1) = 1$$

$T(n) = 2 \cdot T(n/2) + 1$

$T(1) = 1$

```
TIME(n)
1   if n > 1
2         m = ⌊n/2⌋
3         t1 = TIME(m)
4         t2 = TIME(n − m)
5         return t1 + t2 + 1
6   else return 1
```




$T(n) = 1 + 2 + 4 + \cdots + n$

$T(n) = 2n - 1$

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

Vi kan sette opp en rekursiv versjon av Insertion Sort, for å få mer innsikt i ideene bak rekursiv dekomponering.

# Insertion Sort

## SCIENTIFIC AND BUSINESS APPLICATIONS

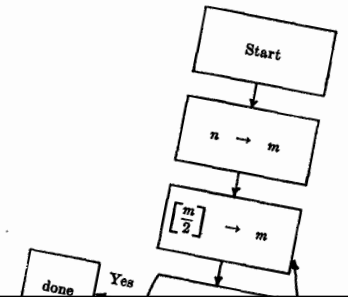
### A High-Speed Sorting Procedure

D. L. SHELL, *General Electric Company, Cincinnati, Ohio*

There are a number of methods that have been used for sorting purposes in various machine programs from time to time. Most of these methods are reviewed by Harold Seward [1] in his thesis. One tacit assumption runs through his entire discussion of internal sorting procedures, namely, that the internal memory is relatively small. In other words, the number of items to be sorted is so large that they cannot possibly all fit into the memory at one time.

The methods of internal sorting which he discusses are sorting by:

1) Finding the smallest.
2) Interchanging pairs.
3) Sifting.
4) Partial sort.
5) Merging pairs.
) Floating decimal sort.
he first four




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

# … recursive edition!

```
CORRECT(n)
1  if n > 1
2        c = CORRECT(n − 1)
3        return c
4  else return TRUE
```

Denne dekomponeringen er inkrementell, heller enn en typisk splitt-og-hersk-dekomponering. I stedet for å dele i to halvdeler, splitter vi bare av ett element.

Denne dekomponeringen er mindre balansert, og for sortering kan det gi dårligere kjøretid.

(Men i andre tilfeller kan dette være akkurat det vi vil ha.)

```
CORRECT(n)
1   if n > 1
2         c = CORRECT(n − 1)
3         return c
4   else return TRUE
```

```
CORRECT(A, j)
1   if j > 1
2         c = CORRECT(A, j − 1)
3         return c
4   else return TRUE
```

Algoritmen har igjen samme struktur som dekomponerings-prosedyren vår – bortsett fra at vi har konkretisert hva som skjer i det induktive trinnet.

```
CORRECT(A, j)
1   if j > 1
2         c = CORRECT(A, j − 1)
3         return c
4   else return TRUE
```

```
REC-INS-SORT(A, j)
1   if j > 1
2         REC-INS-SORT(A, j − 1)
3         key = A[j]
4         i = j − 1
5         while i > 0 and A[i] > key
6               A[i + 1] = A[i]
7               i = i − 1
8         A[i + 1] = key
```

Vi har har bare «stjålet» innsettings-fremgangsmåten fra Insertion-Sort.

```
REC-INS-SORT(A, j)
1   if j > 1
2         REC-INS-SORT(A, j − 1)
3         key = A[j]
4         i = j − 1
5         while i > 0 and A[i] > key
6               A[i + 1] = A[i]
7               i = i − 1
8         A[i + 1] = key
```

```
TIME(A, j)
1   if j > 1
2         t = TIME(A, j − 1)
3         return t + j
4   else return 1
```

For finne kjøretiden, setter vi opp en prosedyre for å regne ut denne. Den vil også ha samme struktur som algoritmen, men i stedet for å *utføre* det induktive trinnet, så angir den bare hvor mye *arbeid* trinnet tar – altså j operasjoner (eller deromkring).

```
TIME(A, j)
1  if j > 1
2        t = TIME(A, j − 1)
3        return t + j
4  else return 1
```

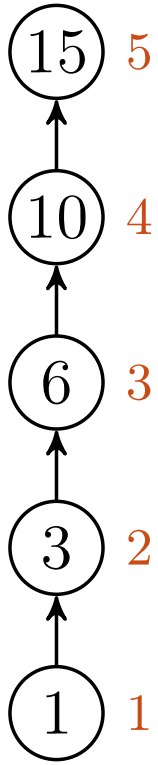
Vi kan forenkle kjøretids-prosedyren til å basere seg kun på problemstørrelsen.

```
TIME(n)
1  if n > 1
2        t = TIME(n − 1)
3        return t + n
4  else return 1
```

$$T(n) = T(n-1) + n$$
$$T(1) = 1$$

$$T(n) = T(n-1) + n$$
$$T(1) = 1$$

```
TIME(n)
1   if n > 1
2         t = TIME(n − 1)
3         return t + n
4   else return 1
```




$$T(n) = 1 + 2 + 3 + \cdots + n$$
$$T(n) = n(n+1)/2$$

$$T(n) = \mathbf{O}(n^2)$$

```
REC-INS-SORT(A, j)
1   if j > 1
2         REC-INS-SORT(A, j − 1)
3         key = A[j]
4         i = j − 1
5         while i > 0 and A[i] > key
6               A[i + 1] = A[i]
7               i = i − 1
8         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
```

Den rekursive algoritmen er naturligvis ekvivalent med den iterative – selv om den rekursive nok har noe overhead.

«[Von Neumann's] manuscript, written in ink, is 23 pages long; the first page still shows traces of the penciled phrase "TOP SECRET," which was subsequently erased. (In 1945, work on computers was classified, due to its connections with military problems.)»

Donald Knuth, «Von Neumann's First Computer Program»
http://dl.acm.org/citation.cfm?doid=356580.356581

# Merge Sort

Some rekursiv sum

# Samme dekomponering

## Klassisk D&C

```
CORRECT(A, p, r)
1   if p < r
2         q = ⌊(p + r)/2⌋
3         c1 = CORRECT(A, p, q)
4         c2 = CORRECT(A, q + 1, r)
5         return c1 and c2
6   else return TRUE
```

```
MERGE-SORT(A, p, r)
1   if p < r
2         q = ⌊(p + r)/2⌋
3         MERGE-SORT(A, p, q)
4         MERGE-SORT(A, q + 1, r)
5         MERGE(A, p, q, r)
```

```
MERGE-SORT(A, p, r)
1   if p < r
2         q = ⌊(p + r)/2⌋
3         MERGE-SORT(A, p, q)
4         MERGE-SORT(A, q + 1, r)
5         MERGE(A, p, q, r)
```

```
TIME(A, p, r)
1   if p < r
2         q = ⌊(p + r)/2⌋
3         t_1 = TIME(A, p, q)
4         t_2 = TIME(A, q + 1, r)
5         return t_1 + t_2 + n
6   else return 1
```

```
TIME(A, p, r)
1  if p < r
2        q = ⌊(p + r)/2⌋
3        t_1 = TIME(A, p, q)
4        t_2 = TIME(A, q + 1, r)
5        return t_1 + t_2 + n
6  else return 1
```

```
TIME(n)
1  if n > 1
2        m = ⌊n/2⌋
3        t_1 = TIME(m)
4        t_2 = TIME(n − m)
5        return t_1 + t_2 + n
6  else return 1
```

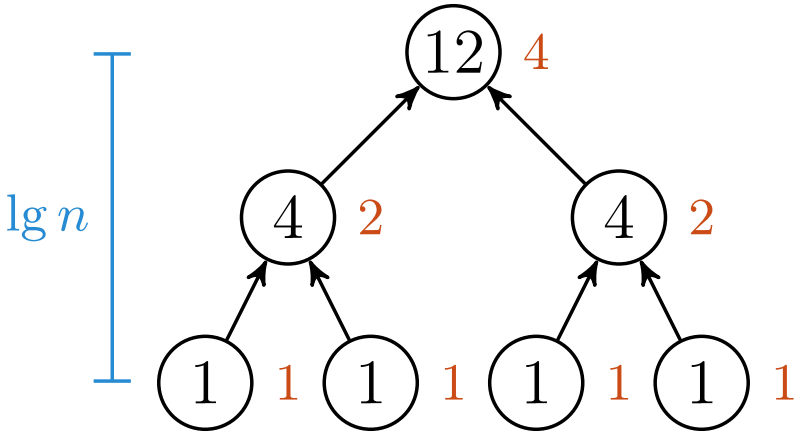
$$T(n) = 2 \cdot T(n/2) + n$$

$$T(1) = 1$$

$T(n) = 2 \cdot T(n/2) + n$

$T(1) = 1$

```
TIME(n)
1   if n > 1
2         m = ⌊n/2⌋
3         t1 = TIME(m)
4         t2 = TIME(n − m)
5         return t1 + t2 + n
6   else return 1
```



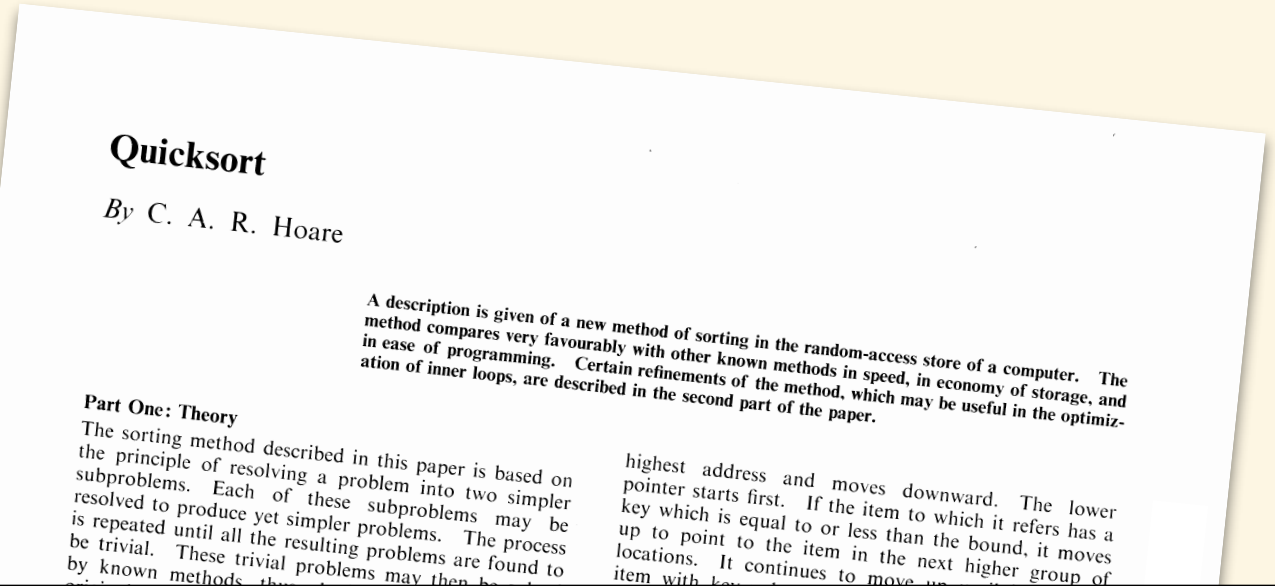

$T(n) = n + \cdots + n$

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

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

# Quicksort

Fra 1962



## Quicksort

*By* C. A. R. Hoare

**A description is given of a new method of sorting in the random-access store of a computer. The method compares very favourably with other known methods in speed, in economy of storage, and in ease of programming. Certain refinements of the method, which may be useful in the optimization of inner loops, are described in the second part of the paper.**

**Part One: Theory**

The sorting method described in this paper is based on the principle of resolving a problem into two simpler subproblems. Each of these subproblems may be resolved to produce yet simpler problems. The process is repeated until all the resulting problems are found to be trivial. These trivial problems may then be by known methods

highest address and moves downward. The lower pointer starts first. If the item to which it refers has a key which is equal to or less than the bound, it moves up to point to the item in the next higher group of locations. It continues to move up

# Samme dekomponering

## … igjen!

```
CORRECT(A, p, r)
1  if p < r
2        q = ⌊(p + r)/2⌋
3        c1 = CORRECT(A, p, q)
4        c2 = CORRECT(A, q + 1, r)
5        return c1 and c2
6  else return TRUE
```

```
CORRECT(A, p, r)
1  if p < r
2        q = PARTITION(A, p, r)
3        c1 = CORRECT(A, p, q − 1)
4        c2 = CORRECT(A, q + 1, r)
5        return c1 and c2
6  else return TRUE
```

```
CORRECT(A, p, r)
1   if p < r
2         q = PARTITION(A, p, r)
3         c_1 = CORRECT(A, p, q − 1)
4         c_2 = CORRECT(A, q + 1, r)
5         return c_1 and c_2
6   else return TRUE
```

```
QUICKSORT(A, p, r)
1   if p < r
2         q = PARTITION(A, p, r)
3         QUICKSORT(A, p, q − 1)
4         QUICKSORT(A, q + 1, r)
```

```
PARTITION(A, p, r)
1   x = A[r]
2   i = p − 1
3   for j = p to r − 1
4         if A[j] ≤ x
5               i = i + 1
6               exchange A[i] with A[j]
7   exchange A[i + 1] with A[r]
8   return i + 1
```

## BC

```
TIME(n)
1   if n > 1
2         m = ⌊n/2⌋
3         t_1 = TIME(m − 1)
4         t_2 = TIME(n − m)
5         return t_1 + t_2 + n
6   else return 1
```

Ca. som MERGE-SORT

## WC

```
TIME(n)
1   if n > 1
2         t = TIME(n − 1)
3         return t + n
4   else return 1
```

Som REC-INS-SORT

# Randomized-Quicksort

## Velg pivot tilfeldig

```
RANDOMIZED-PARTITION(A, p, r)
1  i = RANDOM(p, r)
2  exchange A[r] and A[i]
3  return PARTITION(A, p, r)
```

```
RANDOMIZED-QUICKSORT(A, p, r)
1  if p < r
2        q = RANDOMIZED-PARTITION(A, p, r)
3        RANDOMIZED-QUICKSORT(A, p, q − 1)
4        RANDOMIZED-QUICKSORT(A, q + 1, r)
```

# Intuisjon for AC

## Få nivåer blir skikkelig ille

Detaljert utregning i 7.4

- Noen partisjoner blir balanserte
- Noen blir mer ubalanserte
- Jevnt fordelt hvor ille de er
- Om f.eks. annenhvert nivå er helt ubalansert vil det bare ca. doble dybden
- Prosentandel ubalansert: Konstantfaktor

$$T(n) = \mathrm{O}(n^2)$$

$$T_{\mathrm{A}}(n) = \Theta(n \lg n)$$

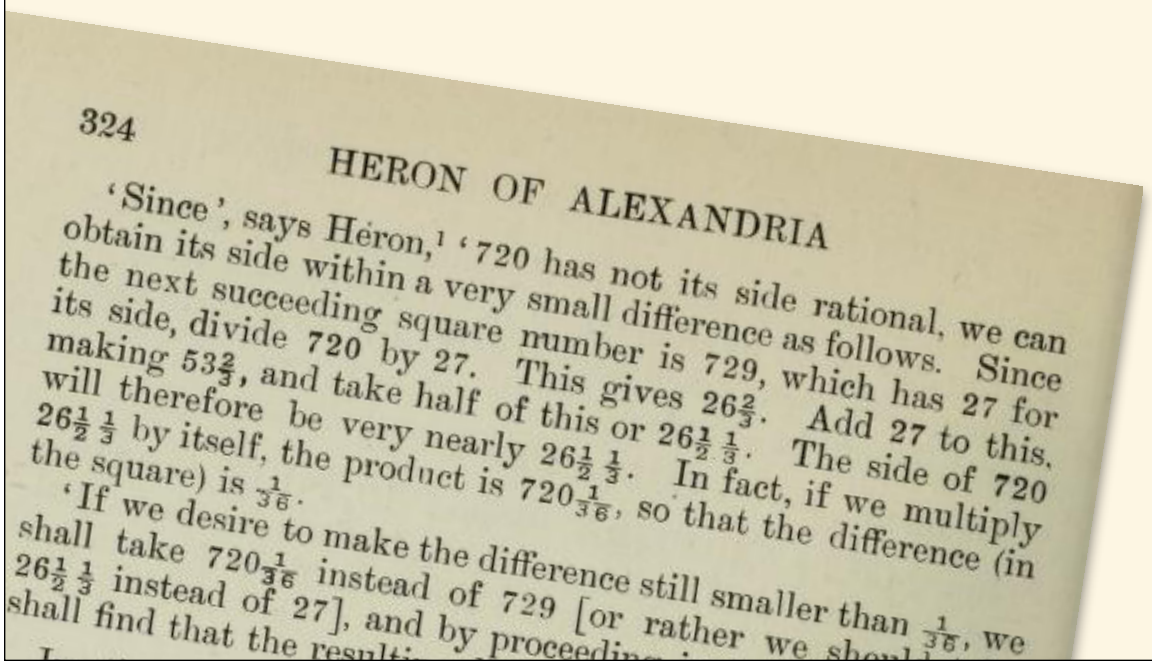
$$T(n) = \Omega(n \lg n)$$

$$T_{\mathrm{W}}(n) = \Theta(n^2)$$

$$T_{\mathrm{A}}(n) = \Theta(n \lg n)$$

$$T_{\mathrm{B}}(n) = \Theta(n \lg n)$$

# Binærsøk



324 HERON OF ALEXANDRIA

'Since', says Heron,[1] '720 has not its side rational, we can obtain its side within a very small difference as follows. Since the next succeeding square number is 729, which has 27 for its side, divide 720 by 27. This gives $26\frac{2}{3}$. Add 27 to this, making $53\frac{2}{3}$, and take half of this or $26\frac{1}{2}\frac{1}{3}$. The side of 720 will therefore be very nearly $26\frac{1}{2}\frac{1}{3}$. In fact, if we multiply $26\frac{1}{2}\frac{1}{3}$ by itself, the product is $720\frac{1}{36}$, so that the difference (in the square) is $\frac{1}{36}$.

'If we desire to make the difference still smaller than $\frac{1}{36}$, we shall take $720\frac{1}{36}$ instead of 729 [or rather we should take $26\frac{1}{2}\frac{1}{3}$ instead of 27], and by proceeding [illegible] shall find that the result

Sir Thomas Little Heath sin beskrivelse av Herons metode for å approksimere kvadratrøtter ved hjelp av en nær slektning av binærsøk – en metode som antagelig ble brukt allerede av babylonerne.

(Heath, A History of Greek Mathematics, vol. 2, 1921; Fowler & Robson, Square Root Approximations in Old Babylonian Mathematics: YBC 7289 in Context, 1998)

# «Tenk på et atom»

## ... av $10^{80}$ mulige

**Trenger $\lg(10^{80}) = 268$ ja-nei-spørsmål**

# Samme dekomponering

## Men bruker bare halvparten

```
CORRECT(n)
1  if n > 1
2        m = ⌊n/2⌋
3        return CORRECT(m)
4  else return TRUE
```

**Litt unøyaktig (±1) ... men kan bruke sterk induksjon**

```
CORRECT(n)
1  if n > 1
2        m = ⌊n/2⌋
3        return CORRECT(m)
4  else return TRUE
```

```
CORR(A, p, r, v)
1  c = TRUE
2  if p < r
3        q = ⌊(p + r)/2⌋
4        if v ≤ A[q]
5              c = CORR(A, p, q, v)
6        else c = CORR(A, q + 1, r, v)
7  return c
```

```
CORR(A, p, r, v)
1  c = TRUE
2  if p < r
3        q = ⌊(p + r)/2⌋
4        if v ≤ A[q]
5              c = CORR(A, p, q, v)
6        else c = CORR(A, q + 1, r, v)
7  return c
```

```
BISECT(A, p, r, v)
1  i = p
2  if p < r
3        q = ⌊(p + r)/2⌋
4        if v ≤ A[q]
5              i = BISECT(A, p, q, v)
6        else i = BISECT(A, q + 1, r, v)
7  return i
```

```
BISECT(A, p, r, v)
1  i = p
2  if p < r
3        q = ⌊(p + r)/2⌋
4        if v ≤ A[q]
5              i = BISECT(A, p, q, v)
6        else i = BISECT(A, q + 1, r, v)
7  return i
```

```
TIME(A, p, r, v)
1  t = 0
2  if p < r
3        q = ⌊(p + r)/2⌋
4        if v ≤ A[q]
5              t = TIME(A, p, q, v)
6        else t = TIME(A, q + 1, r, v)
7  return t + 1
```

```
TIME(A, p, r, v)
1  t = 0
2  if p < r
3        q = ⌊(p + r)/2⌋
4        if v ≤ A[q]
5              t = TIME(A, p, q, v)
6        else t = TIME(A, q + 1, r, v)
7  return t + 1
```

```
TIME(n)
1  if n > 1
2        m = ⌊n/2⌋
3        t = TIME(m)
4        return t + 1
5  else return 1
```

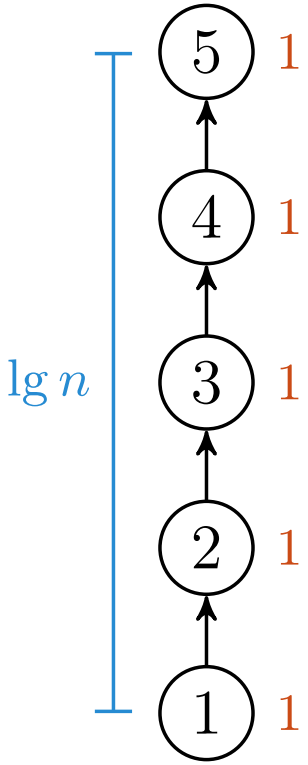
$$T(n) = T(n/2) + 1$$

$$T(1) = 1$$

$T(n) = T(n/2) + 1$

$T(1) = 1$

```
TIME(n)
1   if n > 1
2         m = ⌊n/2⌋
3         t = TIME(m)
4         return t + 1
5   else return 1
```




$T(n) = 1 + \cdots + 1$

$T(n) = \lg n + 1$

$$T_{\mathrm{W}}(n) = \Theta(\lg n)$$

$$T_{\mathrm{A}}(n) = \Theta(\lg n)$$

$$T_{\mathrm{B}}(n) = \Theta(1)$$

Konstant best-case bare hvis man sjekker om midt-elementet er det man søker etter. Ellers også logaritmisk best-case (som i min implementasjon)

```
BISECT(A, p, r, v)
1  i = p
2  if p < r
3        q = ⌊(p + r)/2⌋
4        if v ≤ A[q]
5              i = BISECT(A, p, q, v)
6        else i = BISECT(A, q + 1, r, v)
7  return i
```

```
BISECT(A, p, r, v)
1  while p < r
2        q = ⌊(p + r)/2⌋
3        if v ≤ A[q]
4              r = q
5        else p = q + 1
6  return p
```

# Max. Subarray

- Tabell med positive og negative tall
- Vil finne intervall med størst sum
- Dekomponering (vanlig D&C):
  - Finn beste i hhv høyre og venstre halvdel
  - Finn beste som krysser midten
  - Velg den beste av disse

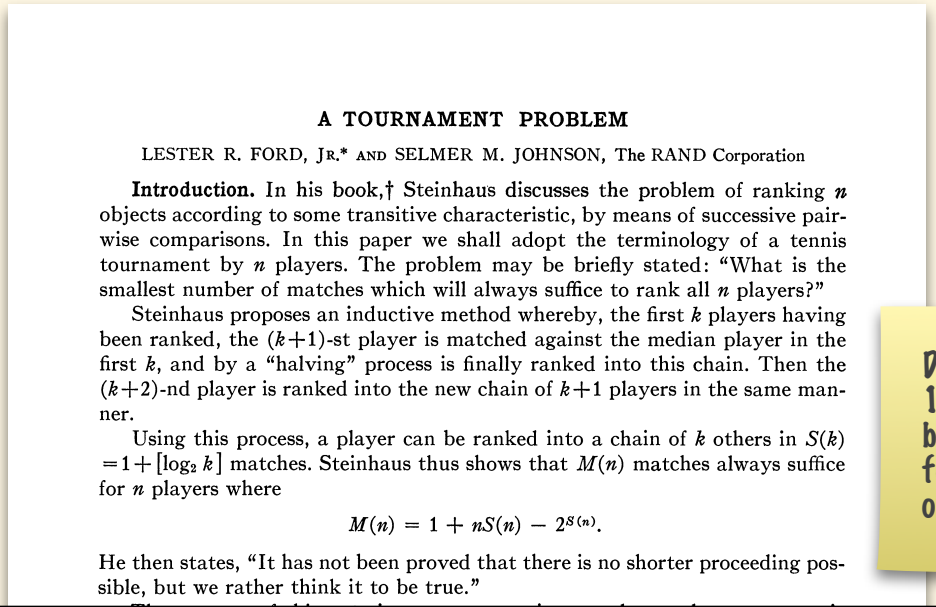
$$T(n) = 2T(n/2) + n$$

$$= \Theta(n \log n)$$

# Kan gjøres bedre!

## ... med dynamisk programmering

# Sorteringsgrensen



A TOURNAMENT PROBLEM

LESTER R. FORD, JR.* AND SELMER M. JOHNSON, The RAND Corporation

**Introduction.** In his book,† Steinhaus discusses the problem of ranking $n$ objects according to some transitive characteristic, by means of successive pairwise comparisons. In this paper we shall adopt the terminology of a tennis tournament by $n$ players. The problem may be briefly stated: "What is the smallest number of matches which will always suffice to rank all $n$ players?"

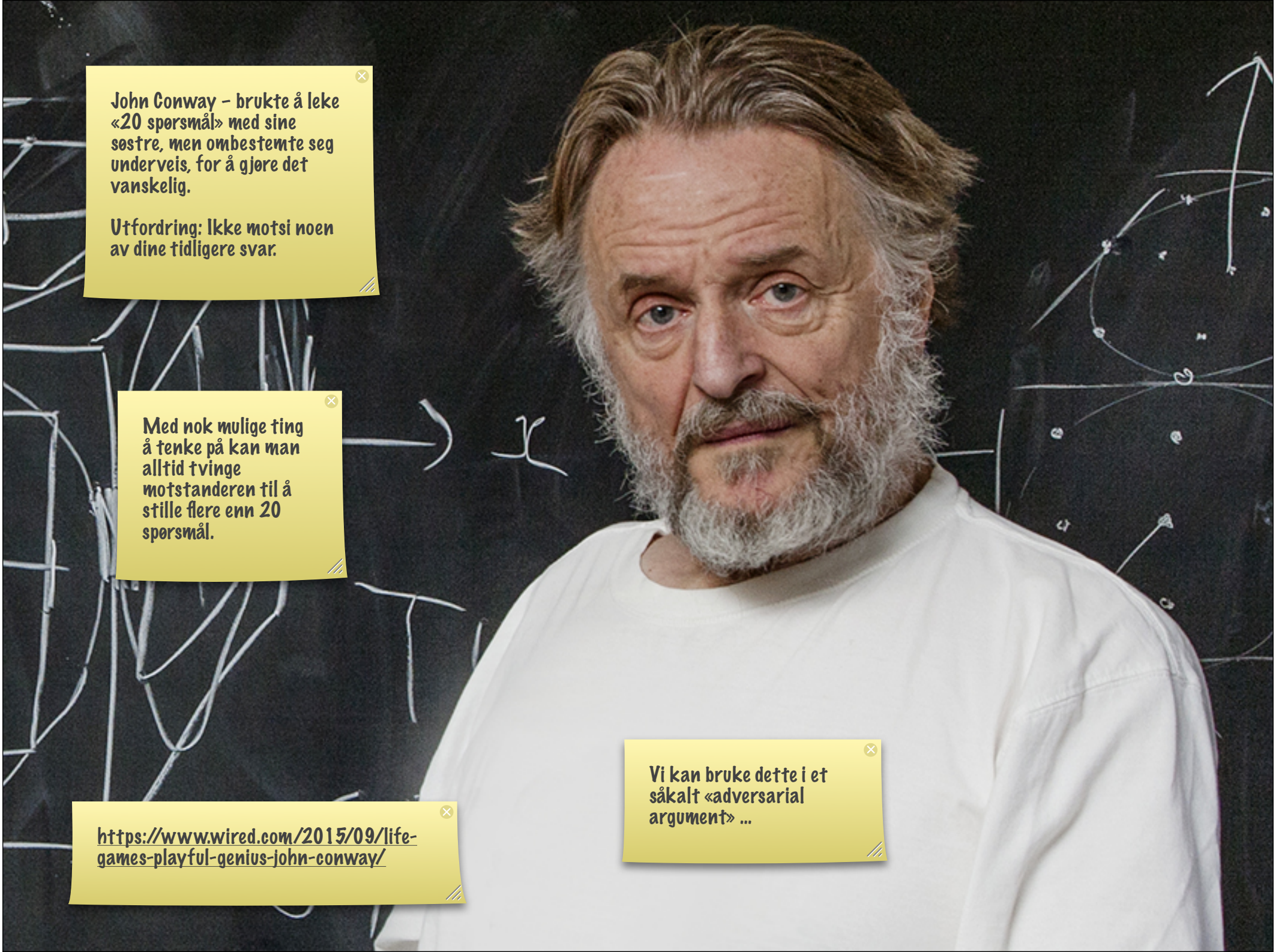
Steinhaus proposes an inductive method whereby, the first $k$ players having been ranked, the $(k+1)$-st player is matched against the median player in the first $k$, and by a "halving" process is finally ranked into this chain. Then the $(k+2)$-nd player is ranked into the new chain of $k+1$ players in the same manner.

Using this process, a player can be ranked into a chain of $k$ others in $S(k) = 1 + [\log_2 k]$ matches. Steinhaus thus shows that $M(n)$ matches always suffice for $n$ players where

$$M(n) = 1 + nS(n) - 2^{S(n)}.$$

He then states, "It has not been proved that there is no shorter proceeding possible, but we rather think it to be true."

Denne artikkelen, fra 1959, beskriver beslutningstre-modellen for analyse av sortering og rangering.


John Conway – brukte å leke «20 spørsmål» med sine søstre, men ombestemte seg underveis, for å gjøre det vanskelig.
Utfordring: Ikke motsi noen av dine tidligere svar.
Med nok mulige ting å tenke på kan man alltid tvinge motstanderen til å stille flere enn 20 spørsmål.
https://www.wired.com/2015/09/life-games-playful-genius-john-conway/
Vi kan bruke dette i et såkalt «adversarial argument» ...

# «Tenk på en permutasjon»

## ... av n! mulige

Trenger maks lg(n!) ja-nei-spørsmål

Worst-case: Tenk deg at en kjiping kjenner algoritmen din og velger input for deg.

# «Tenk på en permutasjon»

## … av n! mulige

**Trenger lg(n!) ja-nei-spørsmål**

$$\ln(n!) = n \ln n - n + \mathrm{O}(\ln n)$$

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

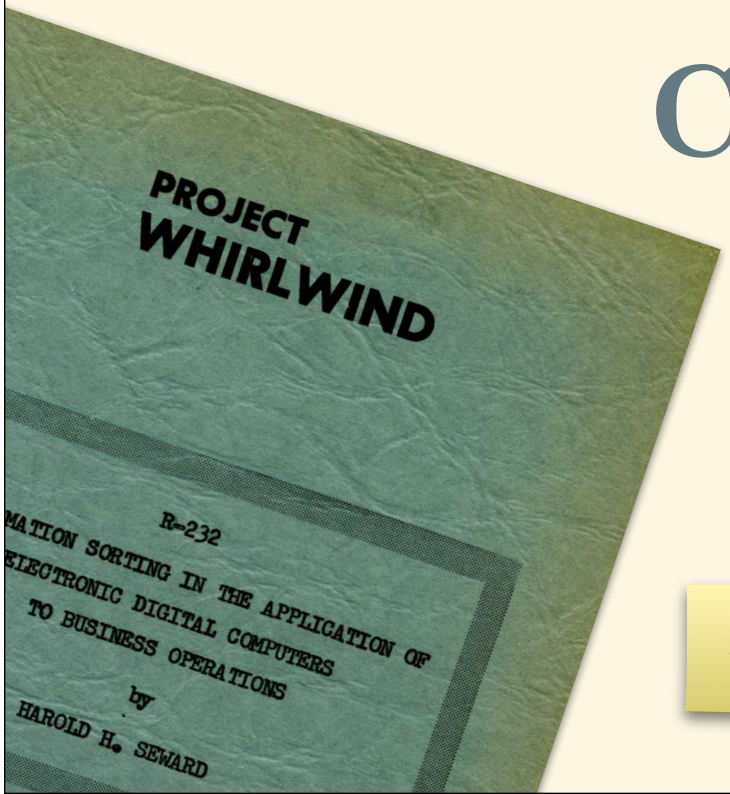
$$T_{\mathrm{B}}(n) = \mathbf{O}(\infty)$$

$$T_{\mathrm{B}}(n) = \mathbf{\Theta}(\,?\,)$$

$$T_{\mathrm{B}}(n) = \mathbf{\Omega}(n)$$

# Tellesortering

## Counting Sort




Fra 1954

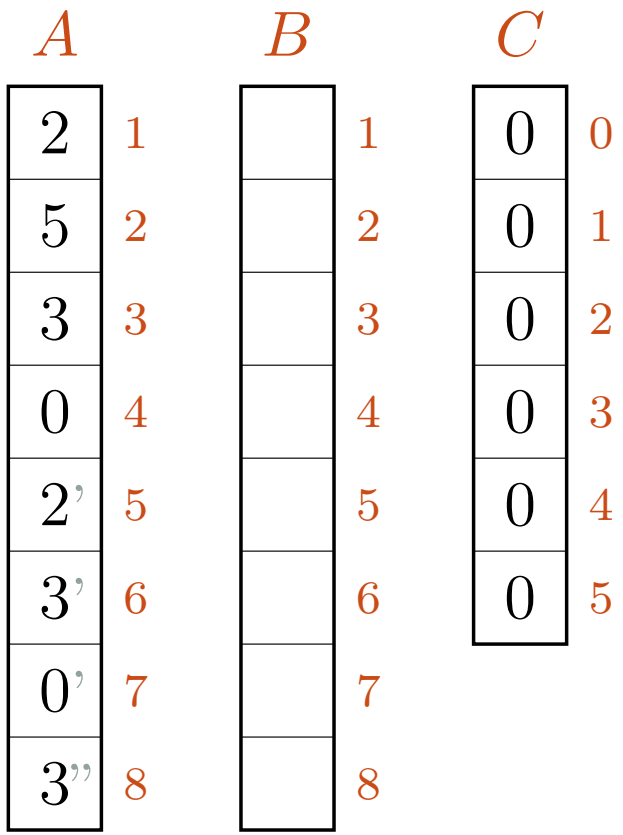
# Bryt grensen

## ... vha. ekstra antagelser

Vi antar at elementene består av ett siffer! (Eller at de er verdier i et begrenset verdiområde 0...k-1)

```
COUNTING-SORT(A, B, k)
 1  let C[0..k] be a new array
 2  for i = 0 to k
 3        C[i] = 0
 4  for j = 1 to A.length
 5        C[A[j]] = C[A[j]] + 1
 6  for i = 1 to k
 7        C[i] = C[i] + C[i − 1]
 8  for j = A.length downto 1
 9        B[C[A[j]]] = A[j]
10        C[A[j]] = C[A[j]] − 1
```

Tell opp hvor mange forekomster vi har av hver verdi i A. Tellingene havner i C.



```
COUNTING-SORT(A, B, k)
 1  let C[0..k] be a new array
 2  for i = 0 to k
 3        C[i] = 0
 4  for j = 1 to A.length
 5        C[A[j]] = C[A[j]] + 1
 6  for i = 1 to k
 7        C[i] = C[i] + C[i − 1]
 8  for j = A.length downto 1
 9        B[C[A[j]]] = A[j]
10        C[A[j]] = C[A[j]] − 1
```

Gjør C *kumulativ*: Dvs., element nummer x gjøres om til summen av de x første elementene.

| | $A$ | | $B$ | | $C$ | |
|---|---|---|---|---|---|---|
| 1 | 2 | 1 | | 1 | 2 | 0 |
| 2 | 5 | 2 | | 2 | 0 | 1 |
| 3 | 3 | 3 | | 3 | 2 | 2 |
| 4 | 0 | 4 | | 4 | 3 | 3 |
| 5 | 2’ | 5 | | 5 | 0 | 4 |
| 6 | 3’ | 6 | | 6 | 1 | 5 |
| 7 | 0’ | 7 | | 7 | | |
| 8 | 3’’ | 8 | | 8 | | |

```
COUNTING-SORT(A, B, k)
 1  let C[0..k] be a new array
 2  for i = 0 to k
 3      C[i] = 0
 4  for j = 1 to A.length
 5      C[A[j]] = C[A[j]] + 1
 6  for i = 1 to k
 7      C[i] = C[i] + C[i − 1]
 8  for j = A.length downto 1
 9      B[C[A[j]]] = A[j]
10      C[A[j]] = C[A[j]] − 1
```

Hvert element i C angir nå posisjonen til den *siste* forekomsten av den tilhørende verdien. For eksempel er C[2] = 4, som betyr at det siste totallet skal være på posisjon 4.

Vi går gjennom A og setter inn forekomstene, én etter én, basert på verdiene i C. Hver gang vi setter inn en forekomst, dekrementerer vi den tilhørende verdien i C.

| | $A$ | | $B$ | | $C$ | |
|---|---|---|---|---|---|---|
| 1 | 2 | 1 | | 1 | 2 | 0 |
| 2 | 5 | 2 | | 2 | 2 | 1 |
| 3 | 3 | 3 | | 3 | 4 | 2 |
| 4 | 0 | 4 | | 4 | 7 | 3 |
| 5 | 2’ | 5 | | 5 | 7 | 4 |
| 6 | 3’ | 6 | | 6 | 8 | 5 |
| 7 | 0’ | 7 | | 7 | | |
| 8 | 3’’ | 8 | | 8 | | |

Det betyr at like verdier vil settes inn fra siste til første forekomst, så for å ikke bytte om på dem (dvs., for å holde algoritmen *stabil*) går vi gjennom A baklengs.

```
COUNTING-SORT(A, B, k)
 1  let C[0..k] be a new array
 2  for i = 0 to k
 3      C[i] = 0
 4  for j = 1 to A.length
 5      C[A[j]] = C[A[j]] + 1
 6  for i = 1 to k
 7      C[i] = C[i] + C[i − 1]
 8  for j = A.length downto 1
 9      B[C[A[j]]] = A[j]
10      C[A[j]] = C[A[j]] − 1
```

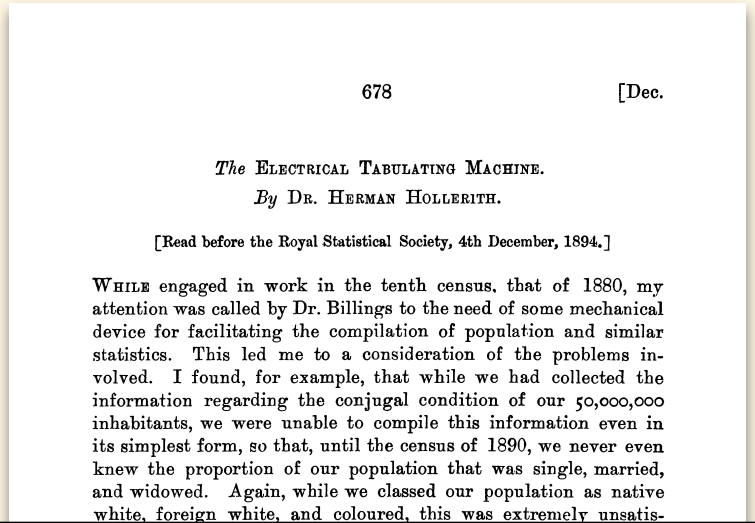
| $A$ | | $B$ | | $C$ | |
|---|---|---|---|---|---|
| 2 | 1 | 0 | 1 | 0 | 0 |
| 5 | 2 | 0’ | 2 | 2 | 1 |
| 3 | 3 | 2 | 3 | 2 | 2 |
| 0 | 4 | 2’ | 4 | 4 | 3 |
| 2’ | 5 | 3 | 5 | 7 | 4 |
| 3’ | 6 | 3’ | 6 | 7 | 5 |
| 0’ | 7 | 3’’ | 7 | | |
| 3’’ | 8 | 5 | 8 | | |

Merk: Like tall har ikke byttet rekkefølge seg imellom! (Jfr. apostrofene ved siden av.) Det er en generell egenskap ved tellesortering; vi sier at den er en *stabil* sorteringsalgoritme.

$$T(n) = \Theta(n + k)$$

# Radikssortering

# Radix Sort



678 [Dec.

*The* Electrical Tabulating Machine.
*By* Dr. Herman Hollerith.

[Read before the Royal Statistical Society, 4th December, 1894.]

While engaged in work in the tenth census, that of 1880, my attention was called by Dr. Billings to the need of some mechanical device for facilitating the compilation of population and similar statistics. This led me to a consideration of the problems involved. I found, for example, that while we had collected the information regarding the conjugal condition of our 50,000,000 inhabitants, we were unable to compile this information even in its simplest form, so that, until the census of 1890, we never even knew the proportion of our population that was single, married, and widowed. Again, while we classed our population as native white, foreign white, and coloured, this was extremely unsatis-

# Utvid verdiområdet

## ... uten lineær tidsøkning

```
RADIX-SORT(A, d)
1  for i = 1 to d
2          sort* A by digit d
```

*Må være stabil†

†Må ikke bytte om like verdier

(Bruk f.eks. COUNTING-SORT)

$$T(n) = \Theta(d \cdot (n + k))$$

Kjøretiden er altså avhengig av antall verdier (n), antall mulige verdier for hvert siffer (k) og antall siffer (d).

# Båssortering

# Bucket Sort

**Sorting by Address Calculation***

E. J. Isaac and R. C. Singleton

*Stanford Research Institute, Menlo Park, California*

*General Description of Method*

Sorting in a random access memory is essentially a process of associating the address of the location in which each item is to be placed with the identifying key of the item. The fewer times the items have to be moved from one location to another, the more efficient the sorting process.

The association of memory address to key in the sorted results can be looked upon as a functional relation; for convenience this will be called the "address function." Figure 1 shows a typical plot of key against memory address. The function is discontinuous and is similar to a cumulative histogram.

If this function were known in advance for a particular batch of data and if

Fra 1955/56

# Bryt grensen

## ... denne gang for AC

Ingen garantier, men antar at inputs er uniformt fordelte tilfeldige tall i intervallet [0,1).

```
BUCKET-SORT(A)
1  n = A.length
2  create B[0..n − 1]
3  for i = 1 to n
4        make B[i] an empty list*
5  for i = 1 to n
6        add A[i] to B[⌊nA[i]⌋]
7  for i = 0 to n − 1
8        sort list B[i]
9  concatenate B[0] ... B[n − 1]
```

*Eller en dynamisk tabell

$$T_{\mathrm{W}}(n) = \Theta(n^2)$$

$$T_{\mathrm{A}}(n) = \Theta(n)$$

$$T_{\mathrm{B}}(n) = \Theta(n)$$