IN2010: Forelesning 8 – Sortering II

Ragnhild Kobro Runde

Sorteringsalgoritmer i IN2010

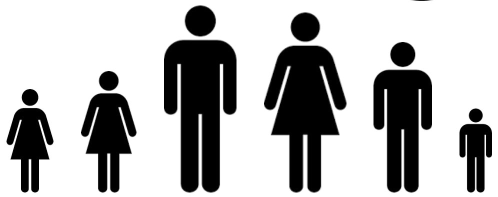
- Boblesortering (bubble-sort)
- Selection-sort
- **Instikksortering** (insertion-sort)
- Heapsortering

- Flettesortering (merge-sort)
- **Quicksort**

- Bøttesortering (bucket-sort)
- **Radix-sortering**

UNIVERSITETET I OSLO

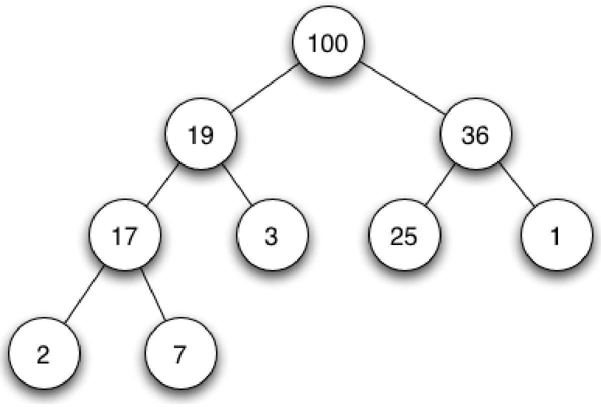
Selection-sort



```
Algorithm SelectionSort(A):
  for i ← 1 to n-1 do
      s ← i
      for j ← i+1 to n do
          if A[j] < A[s] then
              s ← j
      swap A[i] and A[s]
  return A
```

1	2	3	4	5	6	7	8	9	10
4	7	4	5	3	2	1	8	1	3

Hvem skal ut?

Boblesortering	Selection Sort
Innstikksortering	Heapsortering 

Go to www.menti.com and use the code 51 89 59

Sorteringsalgoritmer i IN2010

- Boblesortering (bubble-sort)
- Selection-sort
- **Instikksortering** (insertion-sort)
- Heapsortering

- Flettesortering (merge-sort)
- **Quicksort**

- Bøttesortering (bucket-sort)
- **Radix-sortering**

Flettesortering

https://pixabay.com/en/people-group-crowd-line-silhouette-312122/

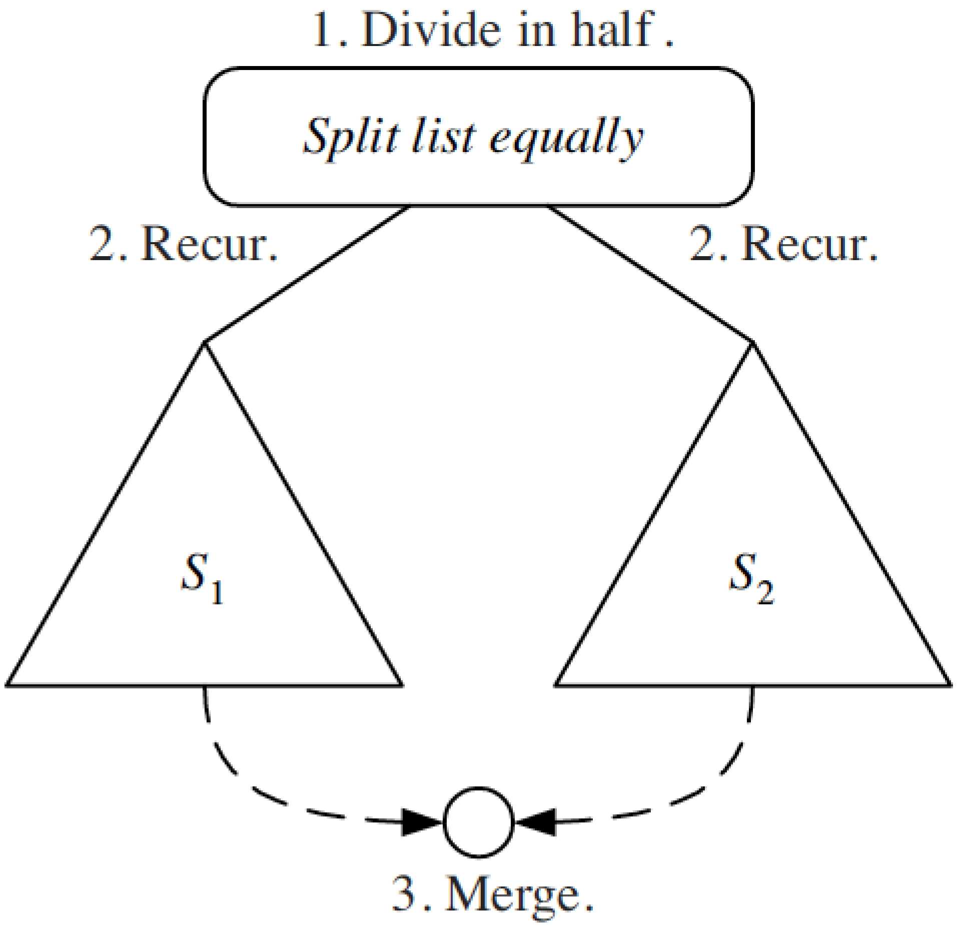
Generell ide:

- Har: To sorterte sekvenser A_1 og A_2
- Ønsker: En stor sortert sekvens A med alle elementene fra A_1 og A_2
- Metode: Leser det minste elementet fra A_1 og A_2 og legger i A. Fortsetter slik til alt ligger i A.

UNIVERSITETET I OSLO

Flettesortering: Splitt og hersk

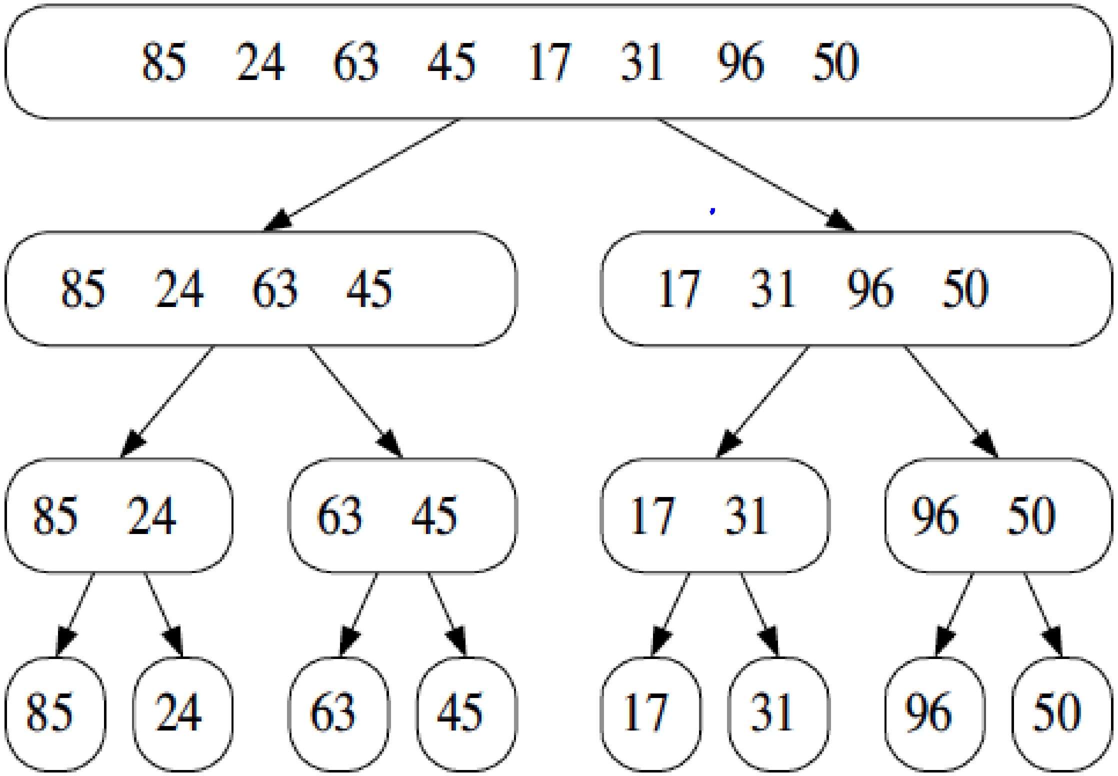
```
Algorithm MergeSort(A):
  if (A.size() > 1)
        A₁ ← A[1...n/2]
        A₂ ← A[n/2+1...n]
        MergeSort(A₁)
        MergeSort(A₂)
        A ← merge(A₁,A₂)
  return A
```



Hva blir kjøretiden (i O-notasjon)?

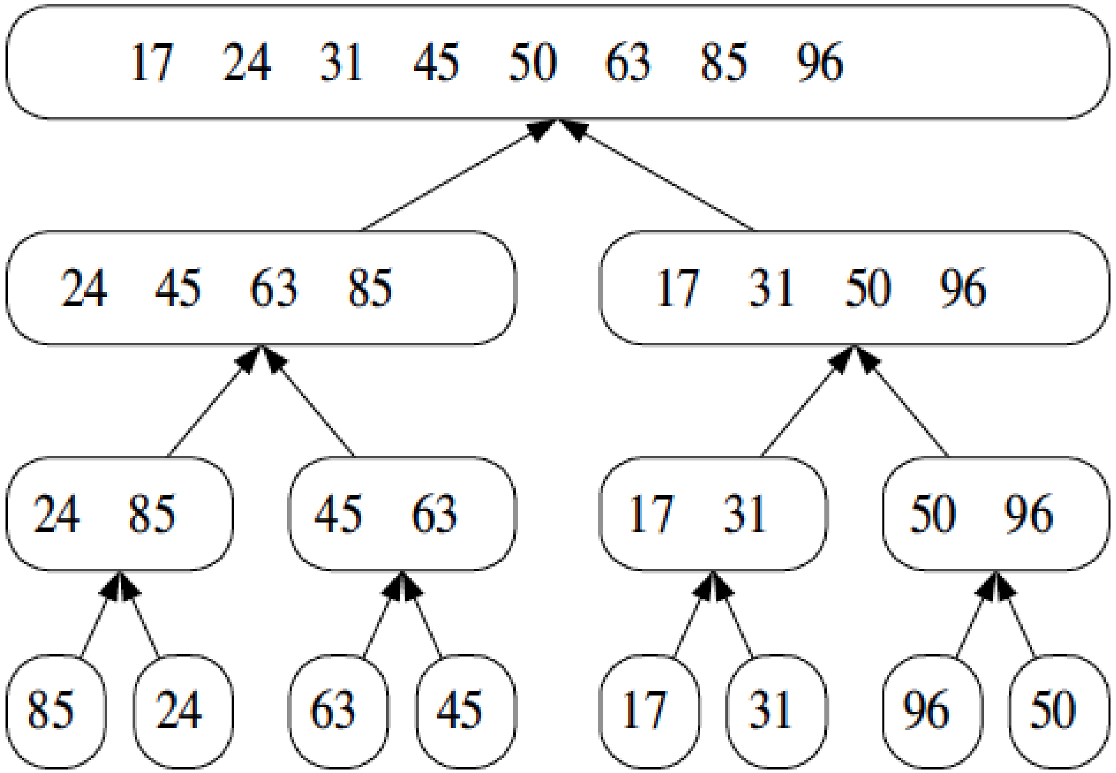
Flettesortering: eksempel

Fase 1: Splitt



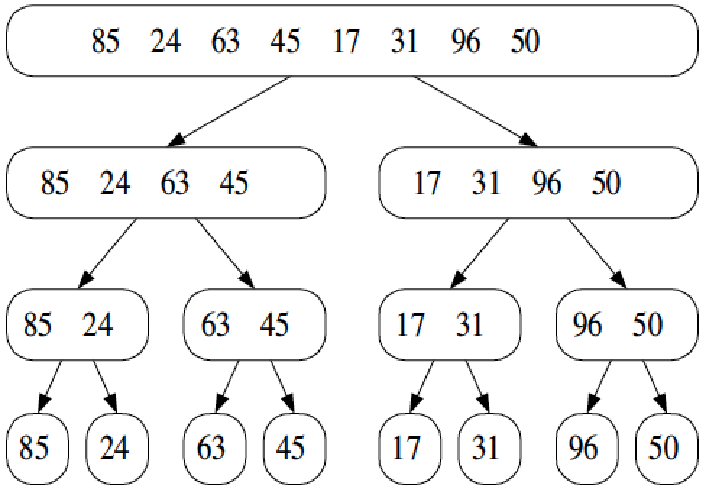
Flettesortering: eksempel

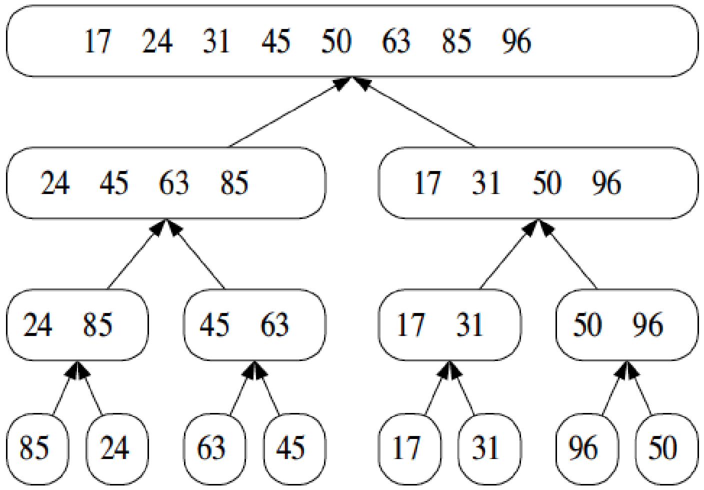
Fase 2: Hersk



UNIVERSITETET
I OSLO

Analyse av flettesortering





Quicksort: generell ide

1. Finn et «middels stort» element fra mengden som skal sorteres. Dette kalles pivot-elementet.
2. Del resten av elementene i to:
    1. De som er mindre enn pivot-elementet
    2. De som er større enn pivot-elementet
3. Sorter disse mengdene hver for seg (ved hjelp av quicksort).
4. Returner sorteringen av de «små» elementene, etterfulgt av pivot-elementet, etterfulgt av sorteringen av de «store» elementene.

1	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**	**10**

UNIVERSITETET I OSLO

inPlaceQuickSort

Arrayen som skal sorteres

Fra-indeks

Til-indeks

```
Algorithm inPlaceQuickSort(S,a,b):
  if a ≥ b then return   // 0 or 1 element
  l ← inPlacePartition(S,a,b)
  inPlaceQuickSort(S,a,l-1)
  inPlaceQuickSort(S,l+1,b)
```

1	2	3	4	5	6	7	8	9	10

inPlacePartition

UNIVERSITETET I OSLO

```
Algorithm inPlacePartition(S,a,b):
    let r be the index of the pivot
    swap S[r] and S[b]
    p ← S[b]
    l ← a
    r ← b-1
    while l ≤ r do
        while l ≤ r and S[l] ≤ p do
            l ← l+1
        while r ≥ l and S[r] ≥ p do
            r ← r-1
        if l < r then
            swap S[l] and S[r]
    swap s[l] and s[b]
    return l
```

Quicksort: Hvordan velge pivot?

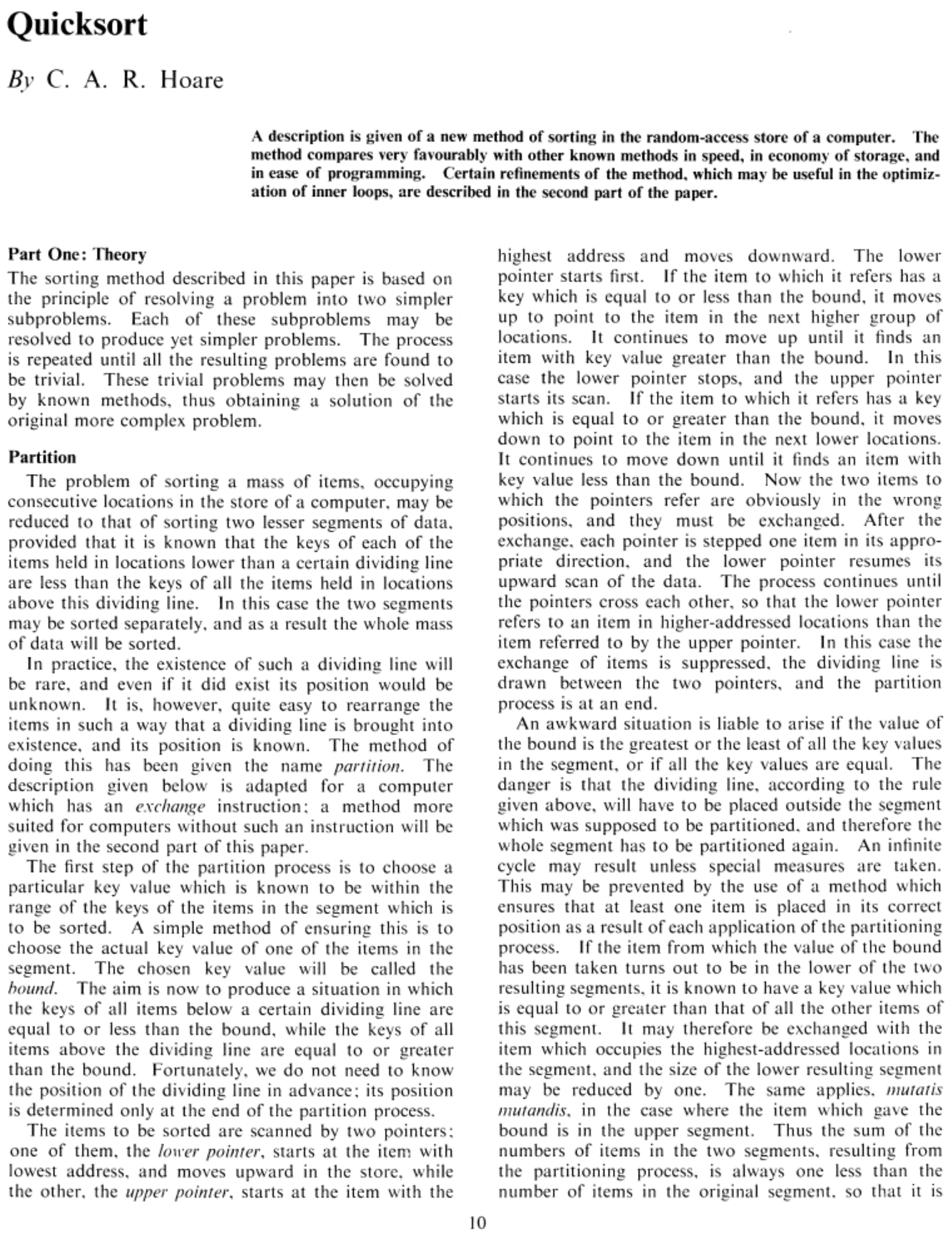
Ideelt: Et pivot-element som deler mengden i to like store halvdeler.

Mulige valg:

- Det første/siste elementet i arrayen.
- Et tilfeldig element i arrayen.
- Midten-av-tre partisjonering: Ser på det første, midterste og siste elementet i arrayen og velger det mellomste av disse som pivot.

Analyse av quicksort

UNIVERSITETET I OSLO



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 solved by known methods, thus obtaining a solution of the original more complex problem.

Partition

The problem of sorting a mass of items, occupying consecutive locations in the store of a computer, may be reduced to that of sorting two lesser segments of data, provided that it is known that the keys of each of the items held in locations lower than a certain dividing line are less than the keys of all the items held in locations above this dividing line. In this case the two segments may be sorted separately, and as a result the whole mass of data will be sorted.

In practice, the existence of such a dividing line will be rare, and even if it did exist its position would be unknown. It is, however, quite easy to rearrange the items in such a way that a dividing line is brought into existence, and its position is known. The method of doing this has been given the name *partition*. The description given below is adapted for a computer which has an *exchange* instruction; a method more suited for computers without such an instruction will be given in the second part of this paper.

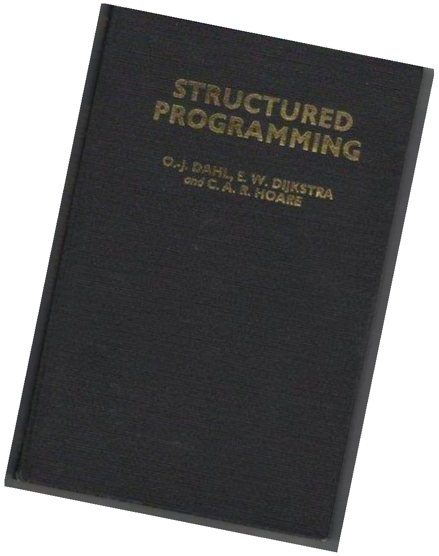
The first step of the partition process is to choose a particular key value which is known to be within the range of the keys of the items in the segment which is to be sorted. A simple method of ensuring this is to choose the actual key value of one of the items in the segment. The chosen key value will be called the *bound*. The aim is now to produce a situation in which the keys of all items below a certain dividing line are equal to or less than the bound, while the keys of all items above the dividing line are equal to or greater than the bound. Fortunately, we do not need to know the position of the dividing line in advance; its position is determined only at the end of the partition process.

The items to be sorted are scanned by two pointers; one of them, the *lower pointer*, starts at the item with lowest address, and moves upward in the store, while the other, the *upper pointer*, starts at the item with the 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 until it finds an item with key value greater than the bound. In this case the lower pointer stops, and the upper pointer starts its scan. If the item to which it refers has a key which is equal to or greater than the bound, it moves down to point to the item in the next lower locations. It continues to move down until it finds an item with key value less than the bound. Now the two items to which the pointers refer are obviously in the wrong positions, and they must be exchanged. After the exchange, each pointer is stepped one item in its appropriate direction, and the lower pointer resumes its upward scan of the data. The process continues until the pointers cross each other, so that the lower pointer refers to an item in higher-addressed locations than the item referred to by the upper pointer. In this case the exchange of items is suppressed, the dividing line is drawn between the two pointers, and the partition process is at an end.

An awkward situation is liable to arise if the value of the bound is the greatest or the least of all the key values in the segment, or if all the key values are equal. The danger is that the dividing line, according to the rule given above, will have to be placed outside the segment which was supposed to be partitioned, and therefore the whole segment has to be partitioned again. An infinite cycle may result unless special measures are taken. This may be prevented by the use of a method which ensures that at least one item is placed in its correct position as a result of each application of the partitioning process. If the item from which the value of the bound has been taken turns out to be in the lower of the two resulting segments, it is known to have a key value which is equal to or greater than that of all the other items of this segment. It may therefore be exchanged with the item which occupies the highest-addressed locations in the segment, and the size of the lower resulting segment may be reduced by one. The same applies, *mutatis mutandis*, in the case where the item which gave the bound is in the upper segment. Thus the sum of the numbers of items in the two segments, resulting from the partitioning process, is always one less than the number of items in the original segment, so that it is

10

https://amturing.acm.org/photo/hoare_4622167.cfm



The Computer Journal, 5(1), pp. 10-16, 1962.

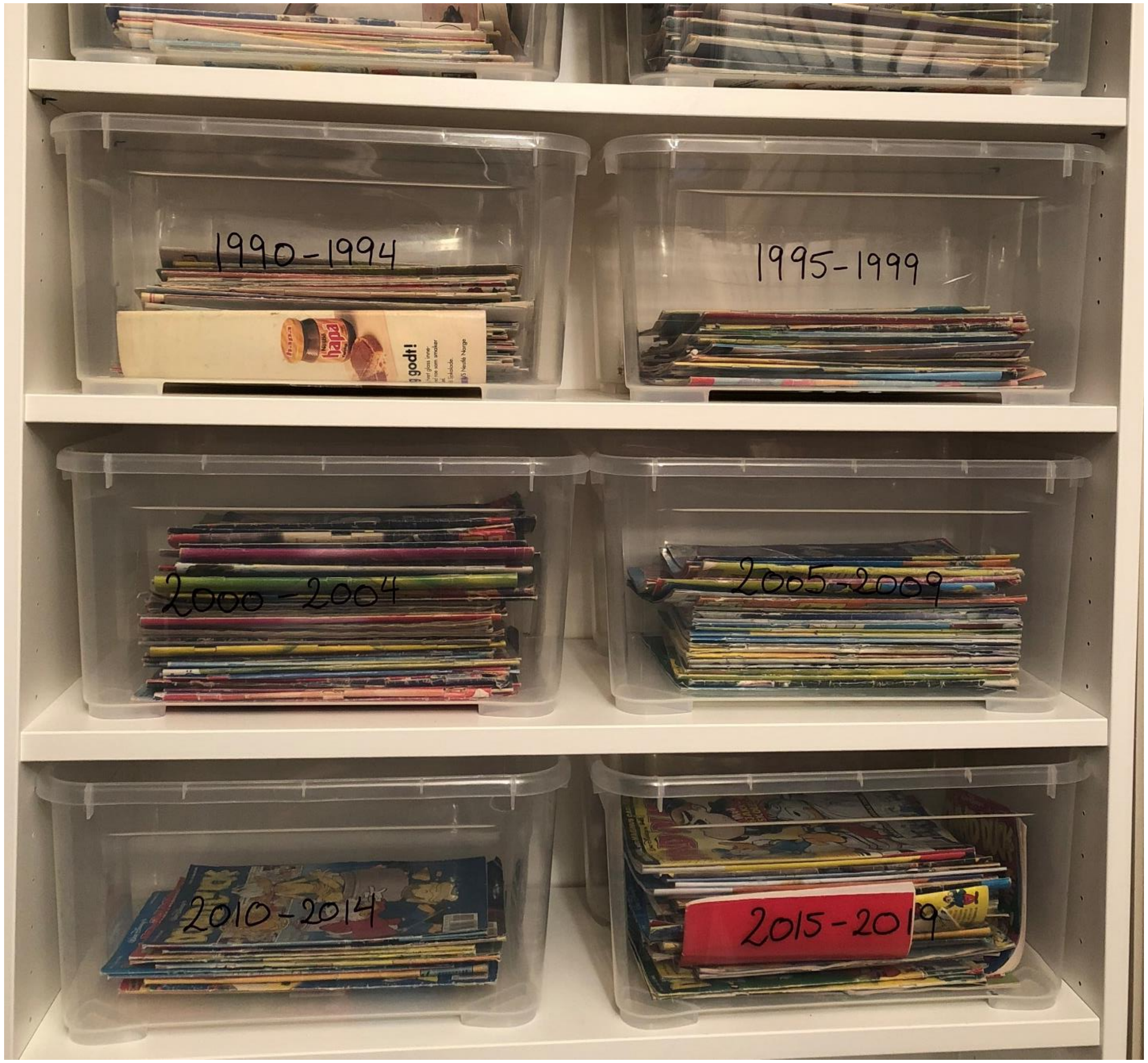
Sorteringsalgoritmer i IN2010

- Boblesortering (bubble-sort)
- Selection-sort
- **Instikksortering** (insertion-sort)
- Heapsortering

- Flettesortering (merge-sort)
- **Quicksort**

- Bøttesortering (bucket-sort)
- **Radix-sortering**

Bøttesortering



Bøttesortering

En **verdibasert** sorteringsmetode.

Arrayen som skal sorteres, nøklene er heltall fra 0 til N-1

```
Algorithm bucketSort(S):
  let B be an array of N lists, intitially empty
  for each item x in S do
    let k be the key of x
    remove x from S and insert it at the end of B[k]
  for i ← 0 to N-1 do
    for each item x in B[i] do
      remove x from B[i] and insert it at the end of S
```

Hva blir kjøretiden (i O-notasjon)?

Radix-sortering

Generalisering av bøttesortering.

Ide: Sorterer først etter det *minst* signifikante sifferet, deretter det nestminste osv.

Eksempel: 10 heltall i intervallet 0 til 999.

0	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**
0	1	512	343	64	125	216	27	8	729

Litt om oblig 3

Neste forelesning: 26. oktober (Stein Michael Storleer)

HASHING