Java-Implementasjon av Merge-Sort

- Interface SortObject

```java
public interface SortObject {
  //sorter sekvens S i ikke-synkende
  rekkefølge vha comparator c
  public void sort (Sequence S, Comparator c);
}
```


Java-Implementasjon av Merge-Sort

```java
public class ListMergeSort implements SortObject {
  public void sort(Sequence S, Comparator c) {
    int n = S.size();
    if (n < 2) return; // 0 eller 1 element er
   // allerede sortert .
    // splitt=divide
    Sequence S1 = (Sequence)S.newContainer();
    // sett inn første halvdel av S i S1
    for (int i=1; i <= (n+1)/2; i++) {
     S1.insertLast(S.remove(S.first()));
    }
    Sequence S2 = (Sequence)S.newContainer();
    // sett inn andre halvdel av S i S2
    for (int i=1; i <= n/2; i++) {
      S2.insertLast(S.remove(S.first()));
    }
    sort(S1,c); // 2 rekursive kall
    sort(S2,c);
    merge(S1,S2,c,S); // hersk=conquer
  }
```


Java-Implementasjon av Merge-Sort

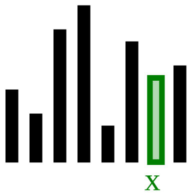
```java
public void merge(Sequence S1, Sequence S2,
    Comparator c, Sequence S) {
  while(!S1.isEmpty() && !S2.isEmpty()) {
    if(c.isLessThanOrEqualTo(S1.first().element(),
      S2.first().element())) {
      // S1's 1ste elt <= S2's 1ste elt
      S.insertLast(S1.remove(S1.first()));
    }
    else { // S2's 1ste elt er minst
      S.insertLast(S2.remove(S2.first()));
    }
  }

  if(S1.isEmpty()) {
    while(!S2.isEmpty()) {
      S.insertLast(S2.remove(S2.first()));
    }
  }
  if(S2.isEmpty()) {
    while(!S1.isEmpty()) {
      S.insertLast(S1.remove(S1.first()));
    }
  }
}
```

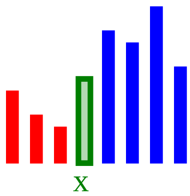

Quicksort

Ide bak Quick Sort

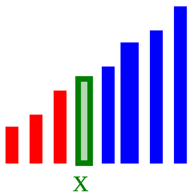
1. Velg pivot
velg et enkelt element



2.Splitt
flytt om på elementene
til Lav og Høy og sett
x på korrekt plass E



3. 2 Rekursive kall
sorter Lav/Høy rekursivt



In-Place Quick-Sort

- **Divide steget**: La siste element være pivot. Vi bruker 2 pekere: venstrepeker *v*, og høyrepeker *h*.

85	24	63	45	17	31	96	**50**
v						*h*	

- Utfør ombyttning (swap) når *v* peker på element større enn pivot-element og *h* peker på element mindre enn pivot-element.

85	24	63	45	17	31	96	**50**
v					*h*		

31	24	63	45	17	85	96	**50**
v					*h*		

In Place Quick Sort)

31	24	63	45	17	85	96	**50**
		v		*h*			

31	24	17	45	63	85	96	**50**
		v		*h*			

31	24	17	45	63	85	96	**50**
			h	*v*			

- Tilsist: Ombyttning med pivot

31	24	17	45	**50**	85	96	63
			h	*v*			

In Place Quick Sort kode

```
public class ArrayQuickSort implements SortObject {

  public void sort(Sequence S, Comparator c){
     quicksort(S, C, 0, S.size()-1);
  }

  private void quicksort (Sequence S, Comparator c,
                          int leftBound,
                          int rightBound) {
          //sorter fra leftBound til RightBound

      if (S.size() < 2) return; //sekvens med 0 eller
                   // 1 element er sortert
      if (leftBound >= rightBound) return; //avslutt
                                   //rekursjon
      // velg siste element som pivot

      Object pivot = S.atRank(rightBound).element();
    // her er de to pekerne
      int venstrepeker = leftBound; // går oppover

      int høyrepeker = rightBound - 1; //går nedover
```

In Place Quick Sort kode

```
// ytre løkke
while (venstrepeker <= høyrepeker) {

  //øk venstrepeker inntil den peker på et
element større enn pivot, eller pekerne krysser

   while ((venstrepeker <= høyrepeker) &&
              (c.isLessThanOrEqualTo
              (S.atRank(venstrepeker).element(),pivot))
     venstrepeker++;

  //senk høyrepeker inntil den peker på et
element mindre enn pivot, eller pekerne krysser

   while (høyrepeker >= venstrepeker) &&
              (c.isGreaterThanOrEqualTo
              (S.atRank(høyrepeker).element(),pivot))
     høyrepeker--;

  //om pekerne ikke krysser, så bytt om

   if (venstrepeker < høyrepeker)
     S.swap(S.atRank(venstrepeker),S.atRank(høyrepeker));

} // ytre løkke fortsetter inntil pekerne /////
//krysser hverandre.
```


In Place Quick Sort kode

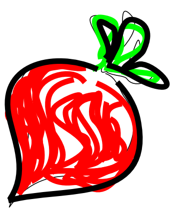
• //Sett så pivot på rett plass

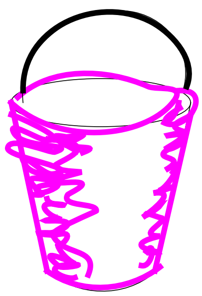
```
S.swap(S.atRank(venstrepeker),S.atRank(rightBound));

    // 2 rekursive kall
    quicksort (S, c, leftBound, venstrepeker-1);
    quickSort (S, c, venstrepeker+1, rightBound);
  }
}
```


Mer om Sortering

- Begrenset input: kun heltall < Maxverdi
- radix sortering - ingen sammenlikninger!
- bøtte sortering - ingen sammenlikninger!
- in-place sortering
- hvor raskt kan vi sortere generellt?





Radix sortering brukes f.eks. på desimaltall:

1. sorter disse siffre | 2. sorter disse siffre | 3. Sorter disse.

0	3	2
2	2	4
0	1	6
0	1	5
0	3	1
1	6	9
1	2	3
2	5	2

0	3	1
0	3	2
2	5	2
1	2	3
2	2	4
0	1	5
0	1	6
1	6	9

0	1	5
0	1	6
1	2	3
2	2	4
0	3	1
0	3	2
2	5	2
1	6	9

0	1	5
0	1	6
0	3	1
0	3	2
1	2	3
1	6	9
2	2	4
2	5	2

NB: Trenger STABIL sortering!.

Voila!

Radix Sortering Kjøretid

Anta heltall med radix r og max b siffer.

for $k = 0$ **to** b - 1
sorter array på en *stabil* måte,
og se kun på bit k

Anta stabil sortering kan utføres i tid O(n). Da blir totaltid

$$O(bn)$$

Stabil sortering? Bruk Bøtte-sortering

Bøtte-sortering

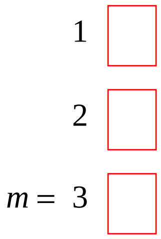
- n tall
- Hvert tall $\in$ {1, 2, 3, ... m}
- Stabil
- Tid: O ($n + m$)

For eksempel, $m = 3$ og input-tabell er:

2	1	3	1	2

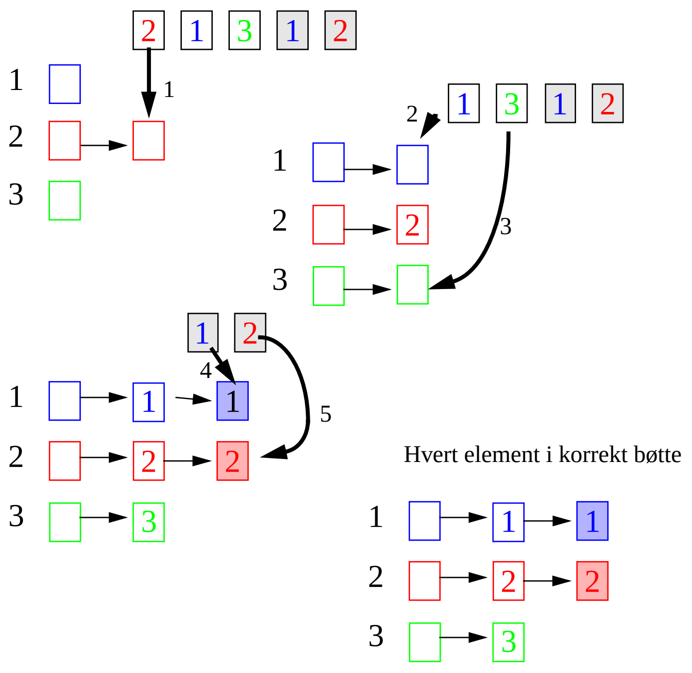
(NB To forskjellige “2”ere og “1”ere)

Vi trenger 3 bøtter



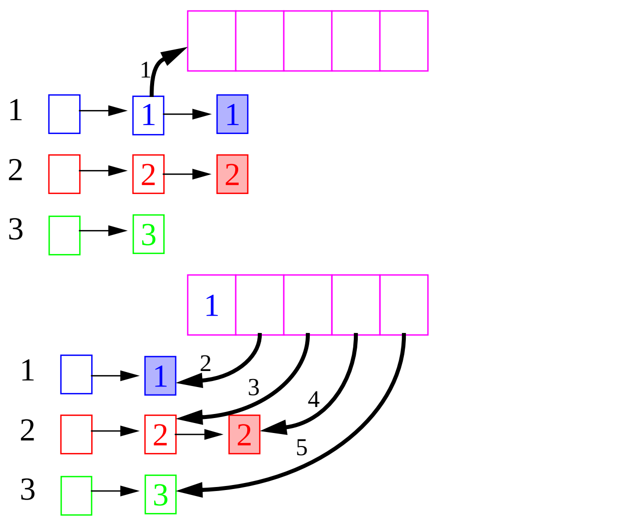
Bøttesortering

Hvert element legges i sin bøtte



Bøtte-sortering

Flytt så elementer fra bøttene tilbake til tabellen:

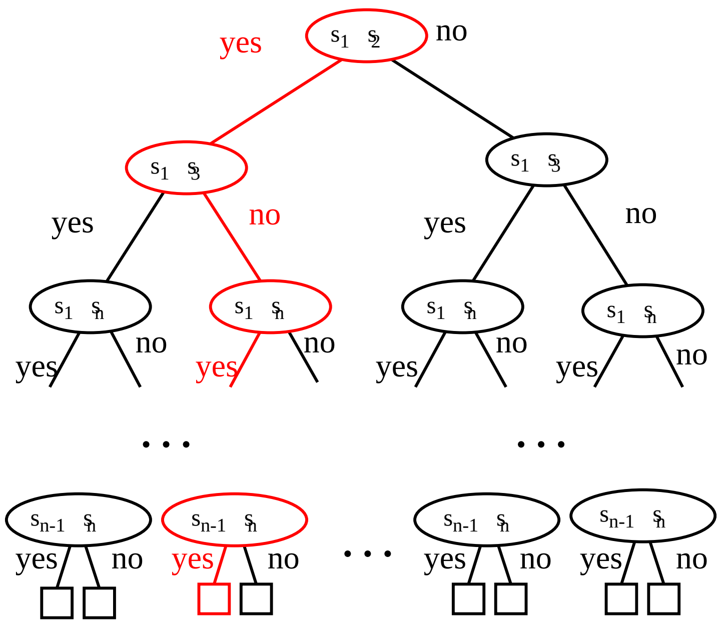


NB, sorteringen var stabil!:

1	1	2	2	3

Decision-tre for sammenlikningsbasert sortering.

- intern node: sammenlikning
- extern node: permutasjon
- utførelse av en algoritme : rot-til-løv sti



Hvor raskt kan vi sortere?

- **Proposisjon**: Kjøretid for sammenliknings-basert-sortering av n elementer er $\Omega(n \log n)$, dvs det finnes konstant k >0 s.a. kjøretiden er > k n log n.
- **Argument for dette**:
- Kjøretiden må være lik eller større enn dybden på decision treet T.
- Hver intern-node i T representerer en sammenlikning som bestemmer rekkefølge på to element.
- Hver ekstern-node i T representerer en permutasjon av input.
- Siden det finnes n! permutasjoner av n element må T ha n! løv, dvs T må ha høyde log(n!)
- Siden *n! har minst n/2 termer større enn n/2, har vi*:
  $\log(n!) \quad (n/2) \log(n/2)$
- Total Kjøretid: $\Omega(n \log n)$.