# Sortering Del 2

# Plan

- Kvikksortering (Quick sort)
- Flettesortering (Merge sort)
- Analysere algoritmene

# Visualisering

- Videoer i «Emnets mediefiler»
- Animasjoner
  - cs.usfca.edu/~galles/visualization/ComparisonSort.html

# Neste gang

- Shellsortering
- Radixsortering (Radix sort)
- Oppgaver

# Sortering hittil

- Vi har sett hittil:
  - utvalgssortering
  - sortering ved innsetting
- Begge disse algoritmene bruker nøstede løkker (en ytre løkke og en indre løkke)
- I verste fall, om vi har n element så vil antall sammenligninger være $O(n^2)$.
- De er ikke effektive når n er stor.
- Vi skal se på noen algoritmer som er mer effektive.

# Kvikksortering (Quick Sort)

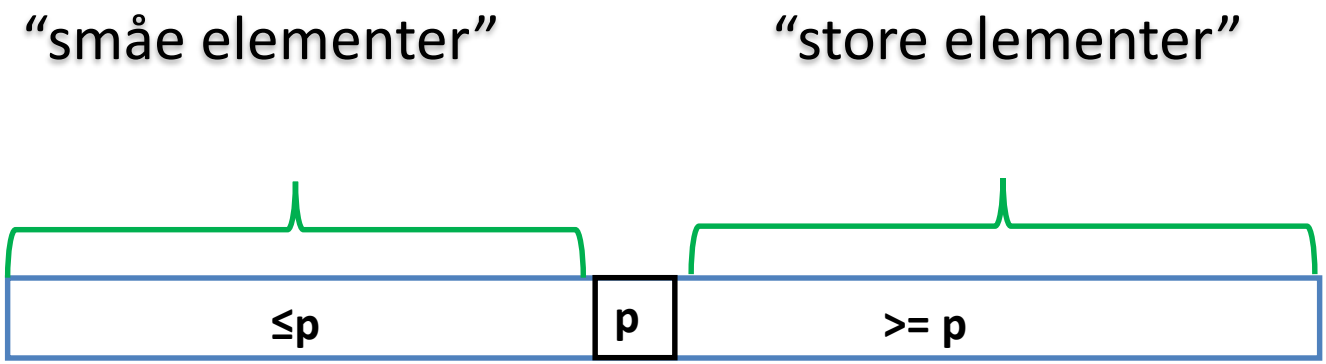
- *Kvikksortering* ordner en liste av verdier ved å splitte/ partisjonere listen med hensyn på et element kalt pivot eller splittelement. Vi kaller elementet  p for pivot.

- Vi deler listen i to, en del som inneholder de småe elementene (<= p) og en del som inneholder de store (>= p)
  (Mange bøker vil plassere alle som er lik pivotelemntet enten til venstre eller høyre.)

- Algoritmen fungerer på kjeda struturer, men vanligvis tenker vi at vi har elementene i en tabell

# Kvikksortering

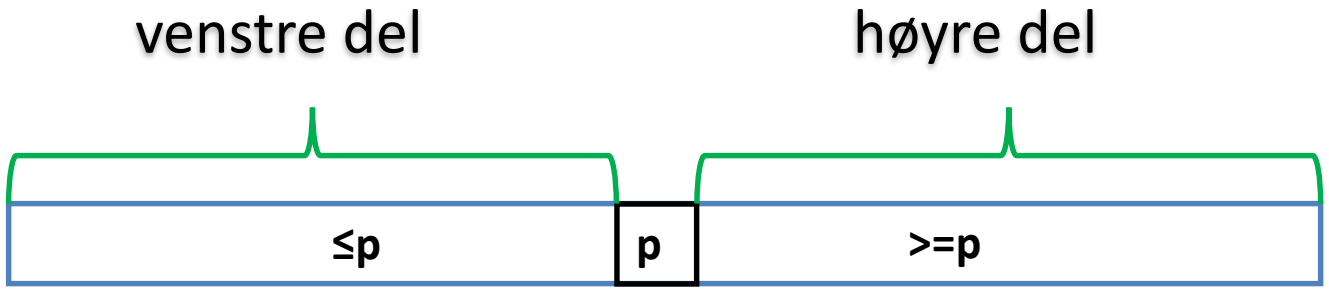
- Etter oppdeling ser det slik ut



# Kvikksortering

Etter oppdeling, anvender vi kvikksort rekursivt på begge partisjonene.

- Kvikksorter venstre del
- Kvikksorter høyre del

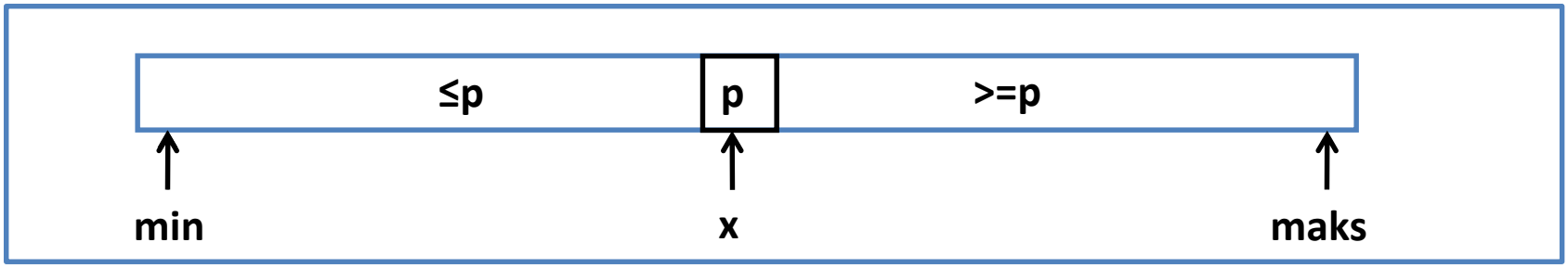


# Kvikksortering

- Kvikksortering blir mest effektivt hvis det etter hver splitting fører til to like (eller nesten like) store partisjoner. Finnes ulike strategier
  - Den første. Gir dårlig resultat om tabell er sortert / omvendt sortert
  - Den midterste
  - En tilfeldig (bruke Random-klassen i Java). Om man skal basere valget på en verdi er denne den sikreste
  - Medianen av et odde antall elementer. Vår bok finner medianen av 3 elementer

- Vi bruker to metoder:
  - `kvikkSort`
    - utfører den rekursive algoritmen
  - `finnPartisjon`
    - omorganiserer elementene i to partisjoner.

# Kvikksortering




```
Algoritme kvikkSort(tabell, min, maks)
  // basistilfelle om min >= maks som betyr ingen eller et element
  // i begge tilfeller vil da delen være sortert og vi gjør ingenting
  hvis (min < maks)  // mer enn to elementer
        Velg et pivotelement, p fra tabell[min…maks]
        Partisjoner elementene i tabell[min…maks]  mhp.  p  slik at:
             tabell[min]…tabell[x-1] < = p
             tabell[x] = p
             tabell[x+1]…tabell[maks] > p

        kvikkSort(tabell, min, x-1)
        kvikkSort(tabell, x+1, maks)
```

# Basistilfelle

- Det er vanlig å si at vi har basistille når det er null eller ett element igjen (slik som på forrige lysark)
- Men, kan gjøre implementasjonen litt mer effektiv ved å slutte rekursjonen tidligere på for eksempel 20 elementer.
  - Problemet er at det blir mye overhead når det er få elementer igjen
  - Kan gå over til en enklere metode (for eksempel sortering ved innsetting som boken gjør)
  - Blir enda mer effektivt om man bruker sortering ved innsetting på hele tabellen til slutt

# finnPartisjon

min – minste indeks, maks – største indeks
venstre – indeks som går fra min mot høyere indeks (fra venstre mot høyre)
hoyre – indeks som går fra maks mot lavere indeks (fra høyre mot venstre)

velg pivot (som medianen av første elementet, elementet i midten og siste elementet) og bytt om med element i nest siste posisjon posisjon min

sett venstre til min + 1 og hoyre til maks - 2
så lenge (venstre < hoyre ){

flytt venstre til et element > pivot
flytt hoyre til et element < pivot

hvis venstre < hoyre
bytt elementene i posisjonene venstre og hoyre
}
bytt pivot med elementet i posisjon venstre

# Illustrasjon av partisjonering

- Gitt følgende liste (fylles inn på forelesing, svar på neste lysark)

  90 65 7 55 120 110 8

# Illustrasjon av partisjonering

- Gitt følgende liste

  90 65 7 55 120 110 8
- Sorterer første, median og siste element (markert som over)

  8 65 7 55 120 110 90
- Vi velger 55 (midtelementet) som pivot (splittelement). Bytter om med nest siste og får

  8 65 7 110 120 55 90
- Leter fra venstre etter et stort element. Finner 65
- Leter fra høyre etter et lite element. Finner 7
- Bytter om 65 og 7 og får

  8 7 65 110 120 55 90

# Illustrasjon av partisjonering

- 8 7 65 110 120 55 90
- Leter fra venstre etter et stort element. Finner 65.
- Leter fra høyre etter et lite elment. Finner 7
- Når har høyre passer venstre og vi er ferdige
- Bytter pivot (55) og 65 og får
- 8 7 55 110 120 65 90

Ser at alle til venstre for pivot er mindre eller lik pivot og at alle til høyre er større eller lik pivot

# Kvikksortering (1)

```java
public static final int MIN_SIZE = 5;

public static <T extends Comparable<? super T>> void quickSort(T[] array, int n) {
    kvikksorter(array, 0, n - 1);
    // Om vi brukar basis som ett eller null element, kan vi ta bort denne
    sorteringVedInnsetting(array, n);
}
```

# Kvikksortering (2)

```
// Grovsorterer
public static <T extends Comparable<? super T>> void kvikksorter(T[] a,
        int forste, int siste) {
    if (siste - forste + 1 >= MIN_SIZE) { // forste < siste, minst to elemnt
        int p = partition(a, forste, siste);
        kvikksorter(a, forste, p - 1);
        kvikksorter(a, p + 1, siste);
    }  // else basis, gjer ingenting
}
```

Merk: No er basis at det er mindre enn MIN_ZIZE element igjen i delen som skal sorterast. Det betyr at tabellen er berre nesten sortert når denne metoden er ferdig.

Derfor har vi kallet til sortering ved innsetting etter at Kvikksorering er ferdig på forrige lysark.

# Overlasting av metodenavn

- Ofte skal hele tabellen sorteres. For å gjøre bruk av metoden enklere, kan vi også ha en variant av algoritmen som bare tar tabellen som skal sorteres som parameter.

```
public static <T extends Comparable<T> void kvikkSort (T[] data){
      // sorterer hele tabellen
        kvikkSort(data, 0, data.length – 1);
}//metode
```

```
   /* kan diskuteres om metoden under nå bør være private */
   private static <T extends Comparable<T>
              void kvikkSort (T[] data, int min, int maks){
   //… samme innhold som metoden forrige lysark
   }
```

- kall

```
   SorterTabell.kvikkSort(tabell);
```

# Eksamen

- Forventer ikke at dere skriver kode partition-metoden
- Tilstrekkelig å kunne forklare hva metoden gjør. Gjerne med tilhørende figur

# finnPartisjon

```
Vises i Eclipse
```

# Kvikksortering

Kortversjonen felles for både tabell og kjedet struktur

Kvikksort(S)

S er en liste av elementer

Hvis antall elementer i S er 0 eller 1 // basis

    så returner.

Velg et element fra S som pivot.

Splitt opp S-{pivot} i to atskilte grupper.

    $S_1 = \{s \in S - \{pivot\} \mid s \leq pivot\}$ og

    $S_2 = \{s \in S - \{pivot\} \mid s >= pivot\}$

Returner Kvikksort($S_1$) + {pivot} + Kvikksort($S_2$).

# Analyse Kvikksortering

- Beste fall (som også blir gjennomsnitt)
  - Fylles inn på forelesing. Hovedpunkter neste ark

# Analyse av kvikksortering

- Beste fall (som også blir gjennomsnitt)
  - Hvert element i tabellen blir sammenlignet, dvs. O(n).
  - Første gang splittes hele tabellen.
  - Hvis hver deltabell blir splittet i like (eller nesten like) deler får vi $\approx \log_2 n$ splittinger.
  - På hvert nivå har vi sammenligning av O(n) elementer
  - Dette gir at antall sammenligninger i beste tilfelle er $O(n\log_2 n)$
  - Det kan vises at antall sammenligninger i gjennomsnitt blir $O(n\log_2 n)$.
    - Krever en god del statistikk for å vise dette. Trenger ikke vises til eksamen.

# Analyse Kvikksortering

- Verste fall

# Analyse av kvikksortering

- I verste tilfelle (skjev splitting hele tiden)
- Når tallene er slik at vi etter hver splitting får en en del med ett element og en del med resten.

[ 12 47 33 13 57 48 86 92]
[12] 13 [47 33 57 48 86 92] venstre har ett element (12)

Dette krever n/2 antall splittinger.

Dvs. totalt antall sammenligninger $O(n^2)$

Kan få tilsvarende situasjon på høyre side

# Flettesortering (Merge sort)

- *Rekursiv Flettesortering* ordner en liste av verdier ved gjentatt halvering av listen inntil hver delliste har kun ett element for deretter å sette sammen dellistene.

- Mer presist:
  - dele listen i to like eller nesten like store deler.
  - rekursivt sortere hver del
    - Basistilfelle når en del inneholder 1 element
      Kan ha med 0 element også. I så fall fungerer algoritmen også på en tabell av lengde 0.
  - flette de to sorterte delene til en sortert del.

# Generell flettealgoritme

Har to sorterte tabeller tabB og tabB.

**Ny tabell sortert tabell tabC som satt sammen av tabA og tabB**

tabA: 1 3 6

tabB: 2 4 5 8 9

tabC: fylles inn på forelesing

# Svar forrige lysark

**tabA: 1 3 6 7**
**tabB: 2 4 5 8 9**

**tabC: 1 2 3 4 5 6 7 8 9**

# Generell flettealgoritme

a , b og c settes til første indeks i tabA, tabB og tabC h.h.v

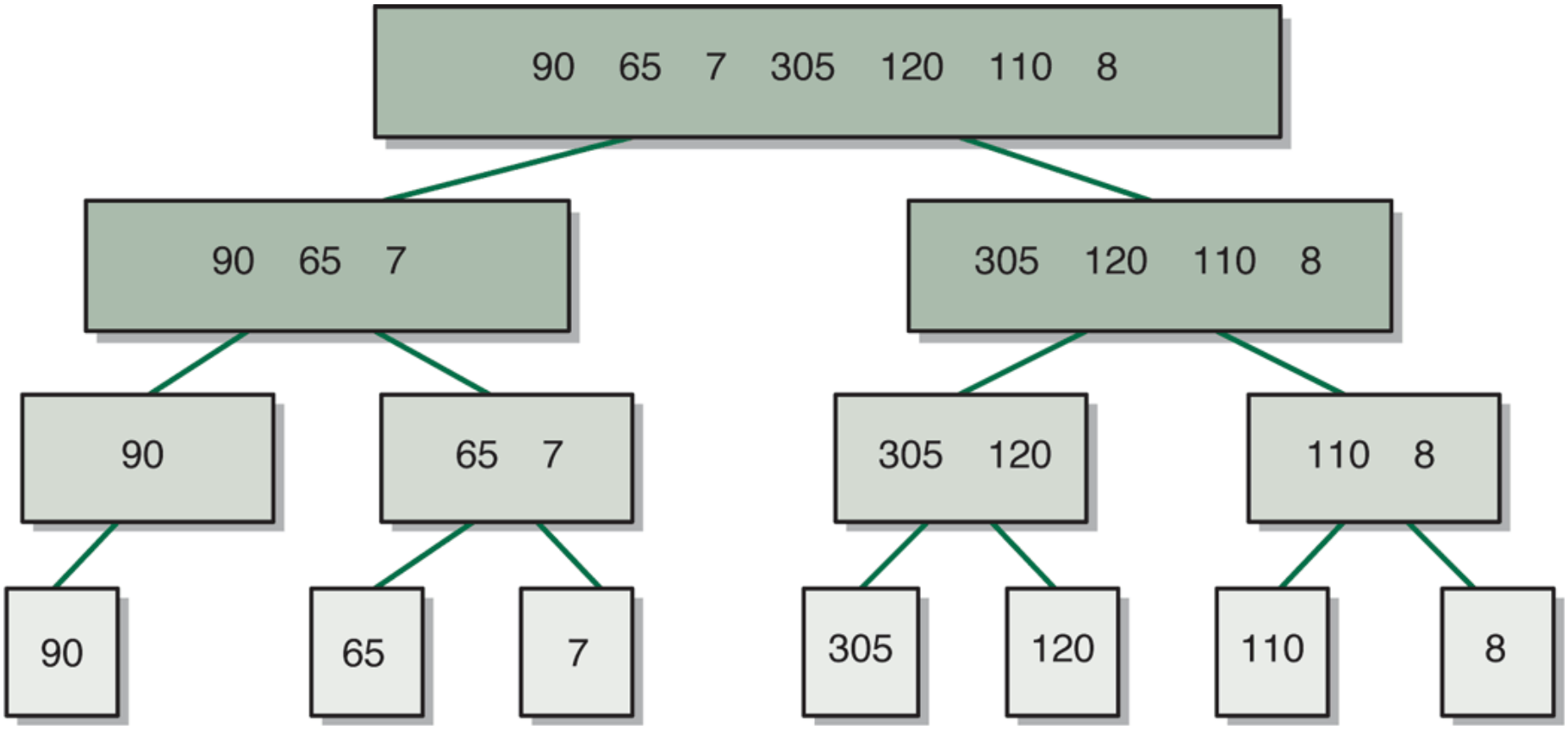
**så lenge** (flere elementer igjen i begge tabellene){

**hvis** (tabA[a] <= tabB[b]) {
tabC[c] = tabA[a]
a++
}
**ellers{**
tabC[c] = tabB[b]
b++
}
c++
**gjenta**

// Enten vil tabA eller tabB være tom, så bare en av setningene nedenfor gjør noe
Kopier resten fra tabA til tabC
Kopier resten fra tabB til tabC

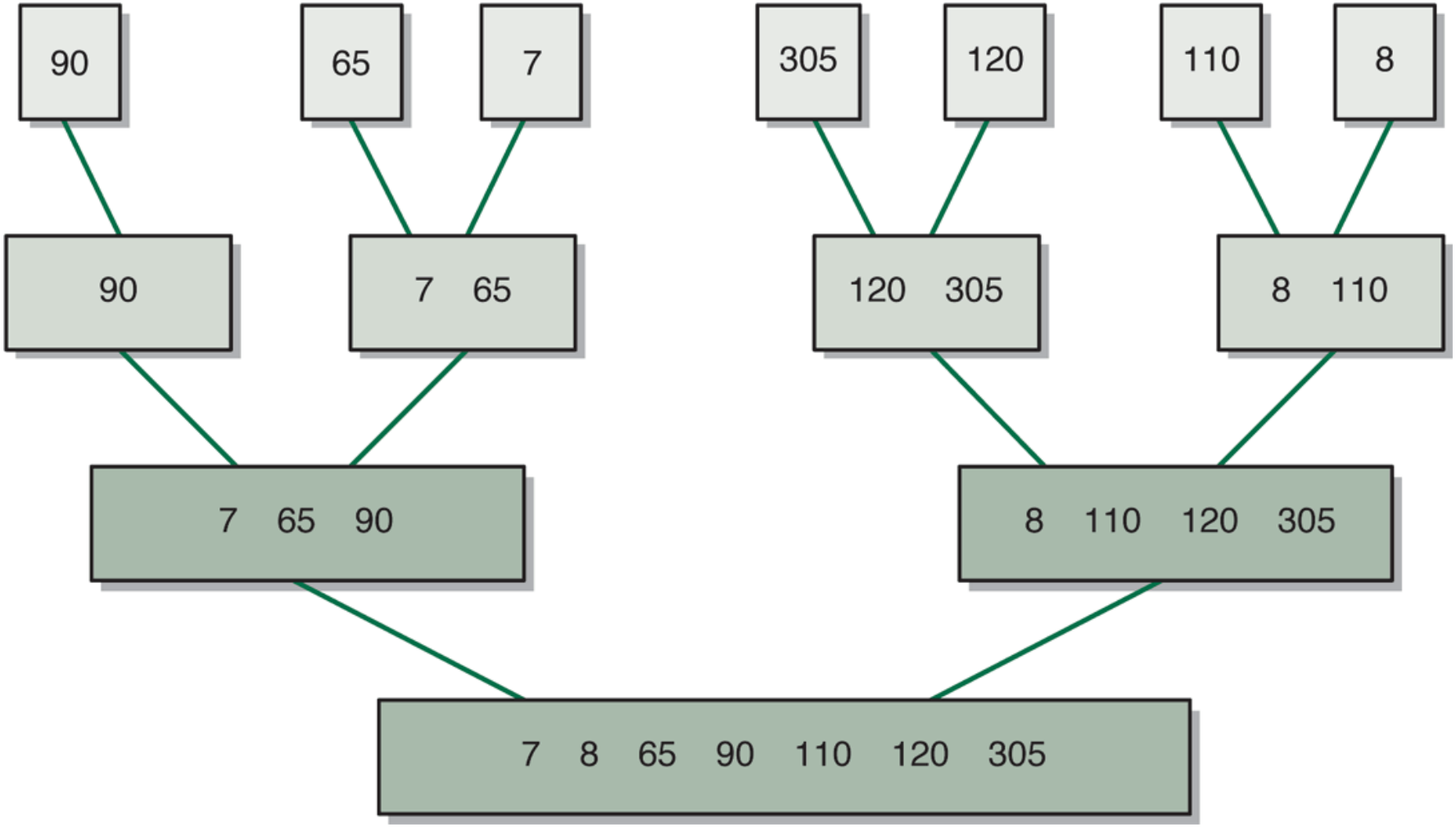
# Oppdeling ved flettesortering

## Rekkefølgen er ikke vist



# Fletting av større og større deler

## Rekkefølgen er ikke vist



# Flettesortering (1)

```java
// Flettesortering
public static <T extends Comparable<? super T>> void flettesortering(T[] a, int n) {
    flettesortering(a, 0, n - 1);
}

public static <T extends Comparable<? super T>> void flettesortering(T[] a,
        int first, int last) {
    // The cast is safe because the new array contains null entries
    @SuppressWarnings("unchecked")
    T[] tempArray = (T[]) new Comparable<?>[a.length]; // unchecked cast
    flettesortering(a, tempArray, first, last);
}
```

# Flettesortering (2)

```
private static <T extends Comparable<? super T>> void flettesortering(T[] a,
        T[] tempTab, int forste, int siste) {
    if (forste >= siste) { // basis: 0/ ett element
        // gjer ingenting
    } else {
        int midten = (forste + siste) / 2;
        flettesortering(a, tempTab, forste, midten);
        flettesortering(a, tempTab, midten + 1, siste);
        flette(a, tempTab, forste, midten, siste);
    }
}
```

# Fletting (1)

```
private static <T extends Comparable<T>>
 void flette(T[] tabell, int forste, int midten,
                                          int siste){

 /* Fletter to sorterte deltabeller,
    tabell[forste,midten] og tabell[midten+1,siste]
    til en sortert tabell
     Forkrav: forste <= midten <= siste
     Deltabellene tabell[forste … midten] og
    tabell[midten+1 … siste] er hver sorterte i ikke-
    avtagende rekkefølge.

     ResultatTabell[forste … siste] er sortert.

     Implementasjon : Denne metoden fletter to
      deltabeller til en hjelpetabell og kopierer
      resultatet til den originale tabellen.
 */
 …
```

# Fletting (2)

```
int storrelse = siste – forste + 1;
T[] hjelpeTabell = (T[])(new Comparable[storrelse]);

//Initierer lokale indekser

// start og slutt på venstre deltabell
int forsteV = forste;
int sisteV  = midten;

// start og slutt på høyre deltabell
int forsteV = midten+1;
int sisteH  = siste;
```

# Fletting (3)

```
/* Så lenge begge deltabellene ikke er tomme,
   kopier det minste elementet til hjelpetabellen.*/

 int indeks = 0;

 while((forsteV <= sisteV) && (forsteH <= sisteH)){
    if(tabell[forsteV].compareTo(tabell[forsteH])<= 0){
        hjelpeTabell[indeks] = tabell[forsteV];
        forsteV++;
    } else {
        hjelpeTabell[indeks] = tabell[forsteH];
        forsteH++;
    }
    indeks++;
 }// while
```

# Fletting (4)

```
// kopiere resten av venstre del (kan være tom)
 while(forsteV <= sisteV){
    hjelpeTabell[indeks] = tabell[forsteV];
    forsteV++;
    indeks++;
 }//while

// kopiere resten av høyre del (kan være tom)
while(forsteH <= sisteH){
    hjelpeTabell[indeks] = tabell[forsteH];
    forsteH++;
    indeks++;
 }//while
```

# Fletting (5)

```
//Kopier resultatet tilbake til den originale tabellen
  int h = 0;
  for(indeks = forste; indeks <= siste; indeks++){
       tabell[indeks] = hjelpeTabell[h];
       h++;
  }
 }//flette */
```

# Analyse av flettesortering, verste fall

- Hvor mange delinger får vi?
  - Anta n er en potens av 2: $n = 2^k$

# Analyse av flettesortering, verste fall

- Hvor mange delinger får vi?
  - Anta n er en potens av 2: $n = 2^k$
  - Antall rekursjonsnivåer: $\lfloor \log_2 n \rfloor + 1$
    (sml. analyse av binærsøk der vi så på antall halveringer)

- På hvert nivå sammenligner vi hvert element i deltabellene, dvs. krever O(n).

- Til sammen gir dette $O(n \log_2 n)$ mhp. sammenligninger i verste tilfelle.

# Analyse av flettesortering, beste fall

- Hvor mange delinger får vi?
  - Anta n er en potens av 2: $n = 2^k$

# Analyse av flettesortering, beste fall

- Hvor mange delinger får vi?
  - Anta n er en potens av 2: $n = 2^k$

- Antall nivå blir det samme $\log_2 n$
- Når det gjelder flettingen, kan alle i en del være mindre enn alle i den andre delen. Da sparer vi ca halvparten av sammenligningene i forhold til verste fall
- Men fremdels O(n)

# Analyse av flettesortering

| Antall deler. | Nivå. |
|---|---|
| 1 | 0 |
| 2 | 1 |
| 4 | 2 |
| … | … |
| n | $\lfloor \log_2 n \rfloor$ |

Ser på verste og beste tilfelle i fletteprosessen:
(kommer tilbake til i en kladdeoppgave)

Verste tilfelle: n–1 sml. :Eks: [1 4 6] [2 5 7]  krever 5 sml.

Beste tilfelle: n/2 sml. Eks. [1 2 3][4 5 6] krever 3 sml.

# Prosessen

- Metoden **flette**, fletter sammen **to sorterte deler** som ligger ved siden av hverandre i tabellen til en sortert del. Basistilfellet for **fletteSort** er at delen som skal sorteres er en eller null lang. En slik del er sortert.

- Vi får rekursive kall helt til delene er en eller null lang. Når vi avslutter kallet, blir venstre del og høyre del flettet sammen. Vi har vist oppdeling og fletting.

# Flettesortering – Oppdeling og fletting

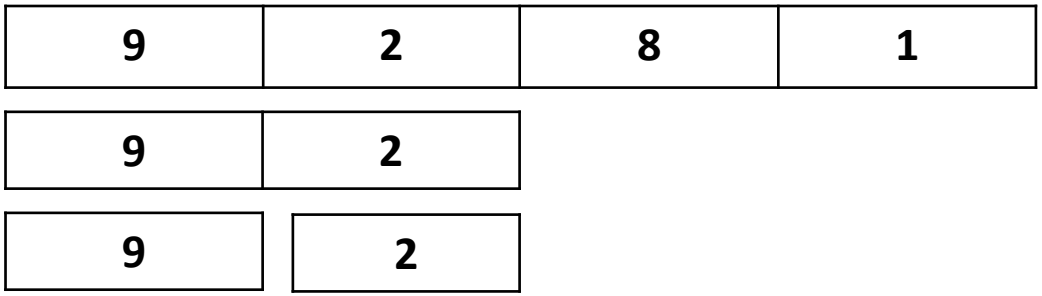
| 9 | 2 | 8 | 1 |
|---|---|---|---|

# Flettesortering – Oppdeling og fletting

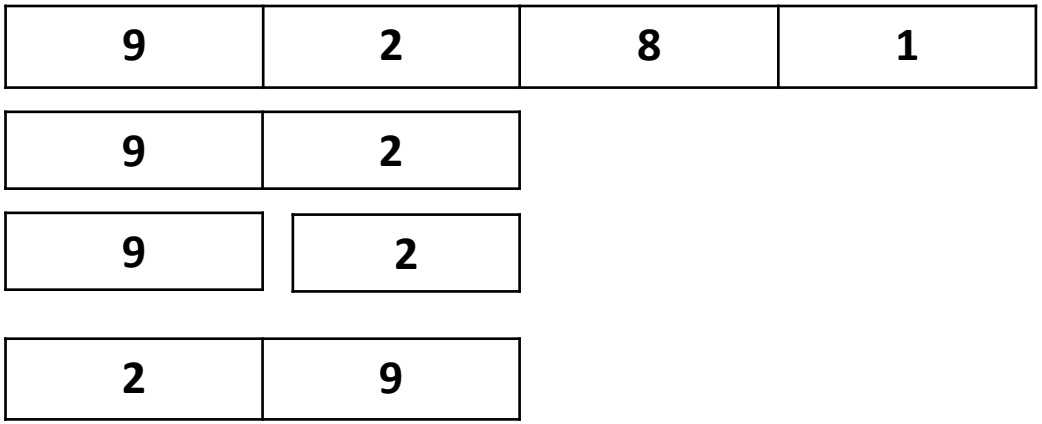
| 9 | 2 | 8 | 1 |
|---|---|---|---|

| 9 | 2 |
|---|---|

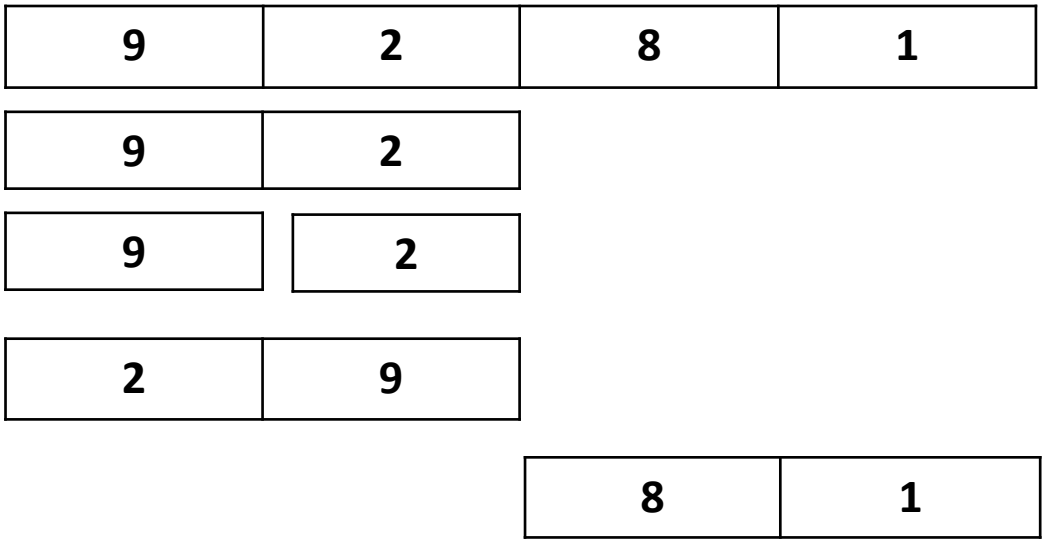
# Flettesortering – Oppdeling og fletting



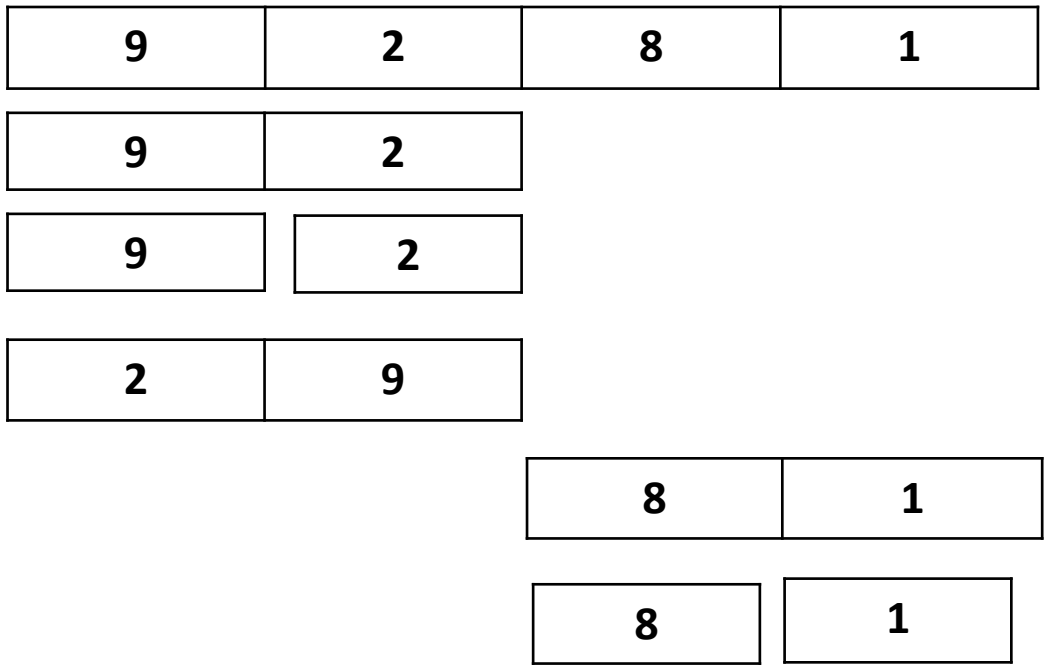
# Flettesortering – Oppdeling og fletting



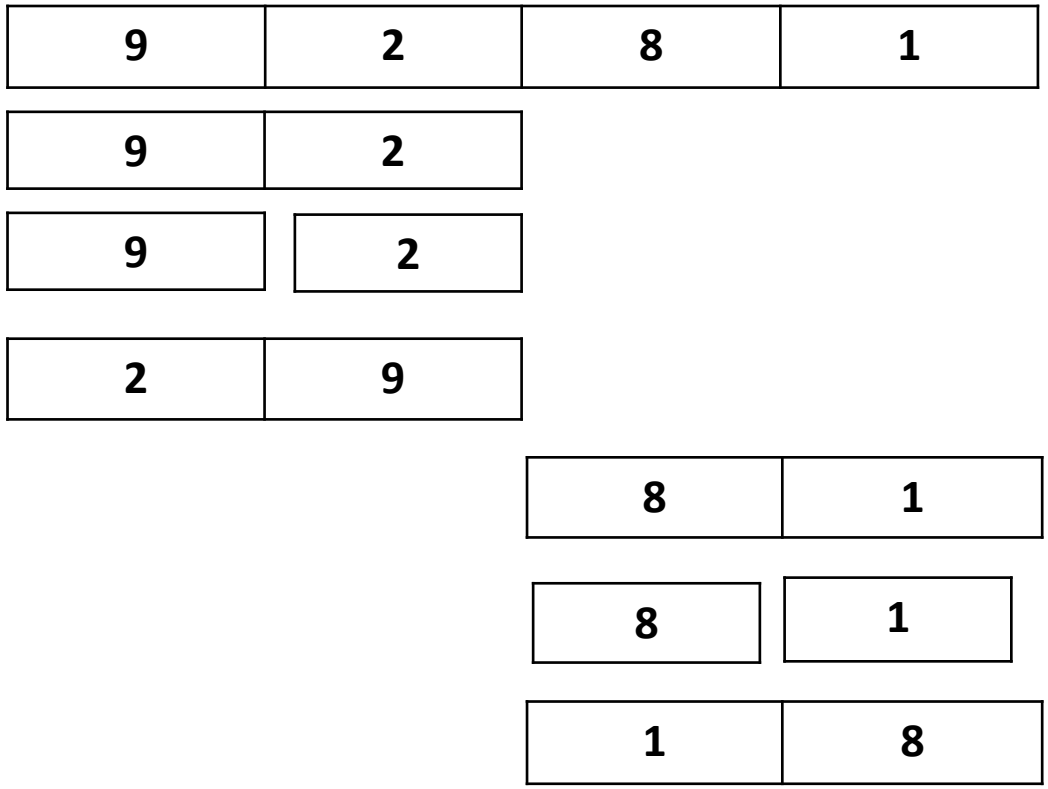
# Flettesortering – Oppdeling og fletting



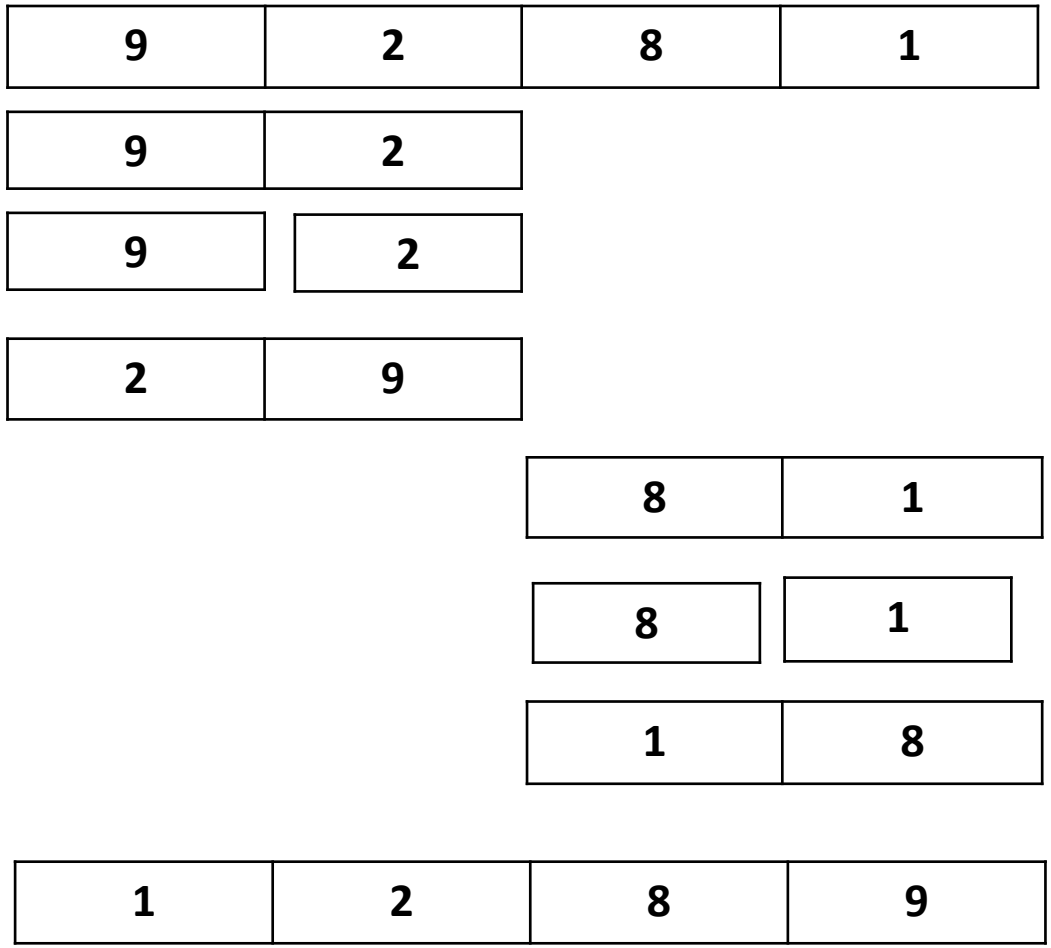
# Flettesortering – Oppdeling og fletting



# Flettesortering – Oppdeling og fletting

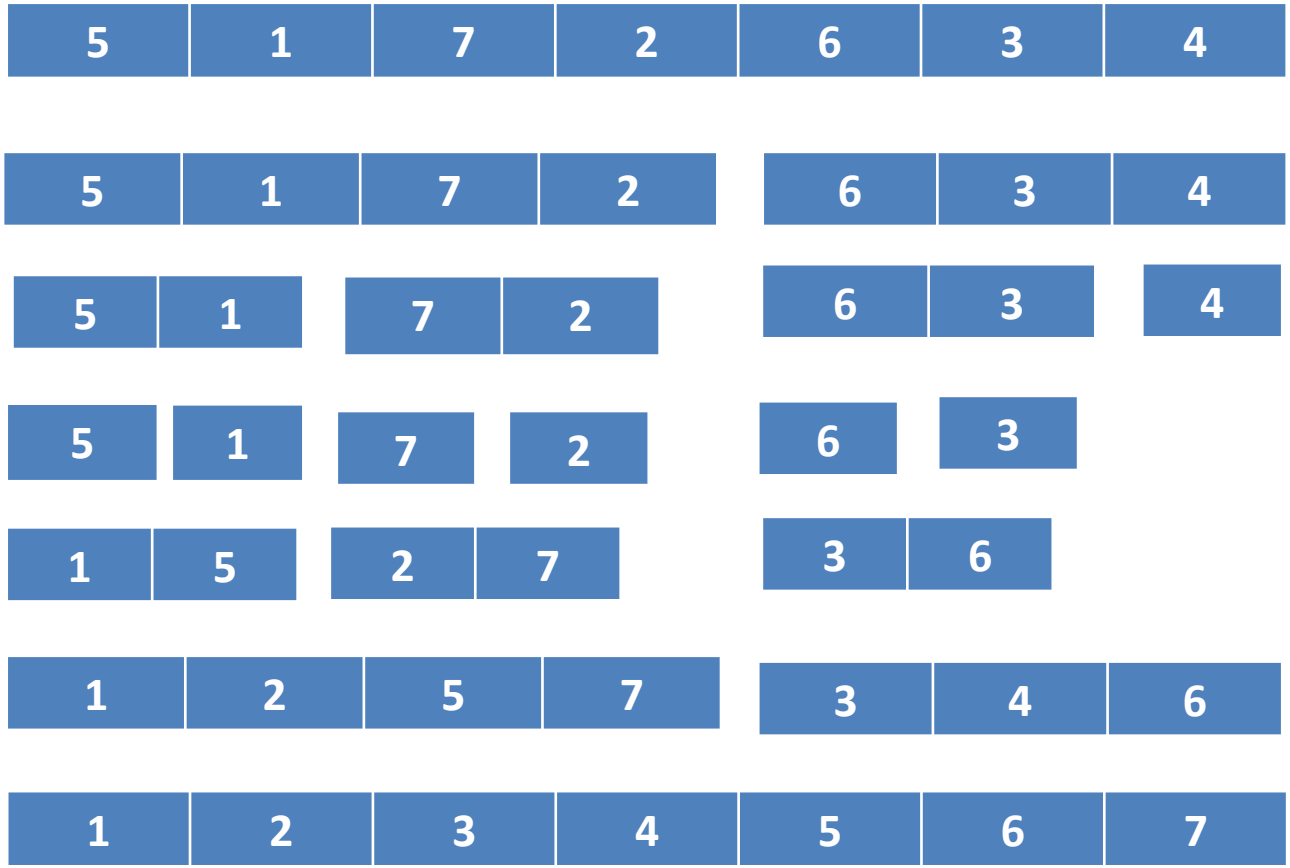


# Flettesortering – Oppdeling og fletting



# Flettesortering – Oppgave

Vis oppdeling og fletting – Rekkefølgen gitt ved animasjon



# Rekursive metoder

- Dele opp
  - Kvikksortering – arbeidskrevende
  - Flettesortering – bare å beregne midten
- Rekursive kall
- Kombinere løsninger av delproblem
  - Kvikksortering – ingenting siden pivotelementet er på rett plass
  - Flettesortering - arbeidskrevende