<u>Begreper å kunne:</u>

Stabilitet

- Ofte sorterer vi på nøkler
    - F.eks kan man sortere et person-objekt etter navn.
- En sorteringsalgoritme kalles stabli dersom
    - for alle elementer x, y med samme nøkkel k(person x, person y, nøkkel navn)
    - Hvis “x” forekom før “y” før sorteringsprosessen
    - så forekommer “x” før “y” etter sortering
- Om en sorteringsalgoritme er stabil eller ikke, avhenges av implementasjonen.

In-place

- En algoritme er in-place dersom den ikke bruker ekstra datastrukturer
- Å mellomlagre resultater i en annen datastruktur er ikke in-place
- I konteksten av sorteringsalgoritmer betyr det at
    - Algoritmen bruke O(1) minne
- Insertion-, Selection- og Bubblesort er alle “In-place” algoritmer da vi operer på kun en array og ikke noe mer enn det.
- Heapsort er unntaket da de ikke er in-place

<u>Merge sort</u>

Ideen bak merge sort er å splitte arrayet i to ca. like store deler. Splitter dem i halvparten da de vil være tilnærmet like i størrelse

- Sortere de to mindre arrayene
- Deretter flette (eller “merge”) de to sorterte arrayene sammen

Litt mer presis forklaring:

1. La “n” angi størrelsen på arrayet A
2. Hvis “n <= 1”, returner A. # For å ta hensyn til tilfeller hvor vi har en tom liste
3. La “i - (n/2)”
4. Splitt arrayet i to deler A[0,,,i-1] og A[i,,,,n-1]
5. Anvend merge sort rekursivt på A[0,,,i-1] og A[i,,,n-1]
6. Flett sammen A[0,,,i-1] og A[i,,,n-1] sortert

Algoritme ved merge

- Input: To sorterte arrayer A(1) og A(2) og et array A, der |A1| + |A2| = |A| = n
- Output: Et sortert array “A” med elementene fra A1 og A2

Procedure Merge(A1,A2,A):

```
i <- 0
j <- 0
while i < |A1| and j < |A2| do
    if A1[i] < A2[j] then
        A[i +j] <- A1[i]
        i <- i + 1
    else
        A[i +j] <- A2[j]
        j <- j + 1
    end
end

while i < |A1| do
    A[i + j] <- A1[i]
    i <- i + 1
end

while j < |A2| do
    A[i + j] <- A2[j]
    j <- j + 1
end

return A
```

Algoritme ved mergesort

Input: Et array “A” med “n” elementer

Output: Et sortert array med de samme “n” elementene

Procedure MergeSort(A)

```
if n <= 1 then
    return A
i <- (n/2)
A1 <- MergeSort(A[0,,,i-1])
A2 <- MergeSort(A[i,,,n-1])
```

```
    return Merge(A1,A2,A)
```

Kjøretidskompleksitet for Merge er O(n)

Kjøretidskompleksitet for MergeSort er O(n*log(n))

<u>Quick sort</u>

Ideen bak quicksort er å velge et element så

- samle alt som er mindre enn elementet til venstre for det
- samle alt som er større enn elementet til høyre for det
- Dette gjøres rekursivt

Litt mer presis forklaring:

1. Vi velger en (0 <= i < n) som kalles pivot-elementet ("i" er det elementet)
    a. Søk fra venstre mot høyre etter et element som er større enn A[i]
    b. Søk fra høyre mot venstre etter et element som er mindre enn A[i]
    c. Bytt plass på disse og søk etter nye som kan byttes
    d. Avslutt når høyre og venstre søkene krysser
2. Fortsett rekursivt på alle som er henholdsvis til venstre og høyre for pivot

Valg av pivotelement:

- Strategier ved å velge pivot elementet kan være følgende:
    - Velg tilfeldig(vil gi oss rask kjøretid)
    - Velg den medianverdien av A[0], A[n//2] og A[n-1]
- Å velge første eller siste som pivot
    - gir verste tilfelle for arrayer som allerede er sortert
- Vi antar en funksjon ChoosePivot
    - som velger pivot etter strategien nevnt over

Algoritme: Partition

Input: Et array "A" med "n" elementer, "low" og "high" er indekser

Output: Flytter elementer som er henholdsvis mindre og større til venstre og høyre enn en gitt index som returneres

```
Procedure Partition(A,low,high)
    p <- ChoosePivot(A,low,high)
    A[p],A[high] <- A[high], A[p]
    pivot <- A[high]
    left <- low
```

```
    right <- high - 1
    while left <= right do
        while left <= right and A[left] <= pivot do
            left <- left + 1
        end
        while right >= left and A[right] >= pivot do
            right <- right - 1
        end
        if left < right then

    end
    A[left], A[high] <- A[high], A[left]
    return left
```

Algoritme: Quicksort

Input: Et array “A” med “n” elementer, “low” og “high” er indekser

Output: Et sortert array med de samme “n” elementene

```
Procedure Quicksort(A,low,high)
    if low >= high then
        return A
    p <- Partition(A,low,high)
    Quicksort(A,low, p-1)
    Quicksort(A,p+1,high)
    return A
```

Kjøretidsanalysen for Partition er O(high-low)

Kjøretidsanalysen for Quicksort er O(n*log(n)) i beste tilfellet, i verste tilfellet er det O(n^2)

<u>Bucket sort</u>

For å kunne implementere denne algoritmen, må vi få mer informasjon om de ulike verdiene som skal sorteres fordi ideen bak "Bucketsort" er å lage "N" bøtter

- hvor hver bøtte svarer til en kategori eller sort
- og kategoriene er ordnet

Elementene vi skal sortere har en kategori, som vi kaller nøkkelen

I bucket sort plasserer vi hvert element i riktig bøtte, altså elementene plasseres i henhold til deres nøkkel, som vil tilsvare kategorien

Så løper vi gjennom hver bøtte

- plasserer elementene tilbake i arrayet

Merk at man noen ganger ønsker å sortere bøttene hver for seg

Algoritme: Bucket sort

Input: Et array "A" med "n" elementer

Output: Et array med de samme "n" elementene sortert etter nøkler

```
Procedure BucketSort(A)
        B <- [] // La B være et array med "N" tomme lister
        for i <- 0 to n-1 do
                la "k" være nøkkelen assosiert med A[i]
                Legg til A[i] på slutten av listen B[k]
        end
        j <- 0
        for k <- 0 to N - 1 do
                for hver x i listen B[k] do
                        A[j] <- x
                        j <- j + 1
                end
        end
```

Gir en viss ide ved Bucket sort

Kjøretidsanalysen for Bucketsort er O(n+k)

<u>Radix sort</u>

Radix sort er nært beslektet bucket sort

Eksempel med å sortere med kort som vi så i forrige, er en type radix sort

En svært god intuisjon for radix sort er følgende

- suksessiv anvendelse av bucket sort

Radix sort kan brukes for data som kan ordnes leksikografisk

Leksikografiske ordninger

Leksikografiske ordninger er en generalisering av alfabetisk rekkefølge

- Vi kan ordne ord etter bokstavene (som i seg selv er ordnet)
    - Der første bokstav prioriterer andre, som prioriteres over tredje, osv
- Vi kan ordne tall på samme måte
    - Der første siffer prioriteres over andre, som prioriteres over tredje, osv
- Mer generelt kan vi tenke oss tupler med symboler (a1, a2, … , ad)
- Vi sier at (a1,a2, …., ad) < (b1, b2, ….. , bd) hvis
    - a1 < b1 eller
    - aa = b1 og (a2, …., ad) < (b2, ….,bd)
- · Fra korteksempelet har vi for eksempel at 7♣ < J♣
    - · fordi ♣ = ♣, men 7 < J

1814, 232, 2888, 31, 1455, 2242, 4345, 1470, 515, 3632

1814, 0232, 2888, 0031, 1455, 2242, 4345, 1470, 0515, 3632

[1470] [0031] [0232, 2242, 3632] [] [1814] [1455, 4345, 0515] [] [] [2888] []

1470, 0031, 0232, 2242, 3632, 1814, 1455, 4345, 0515, 2888

[] [1814, 0515] [] [0031, 0232, 3632] [2242, 4345] [1455] [] [1470] [2888] []

1814, 0515, 0031, 0232, 3632, 2242, 4345, 1455, 1470, 2888

[0031] [] [0232, 2242] [4345] [1455, 1470] [0515] [3632] [] [1814, 2888] []

0031, 0232, 2242, 4345, 1455, 1470, 0515, 3632, 1814, 2888

[0031, 0232, 0515] [1455, 1470, 1814] [2242, 2888] [3632] [4345] [] [] [] [] []

0031, 0232, 0515, 1455, 1470, 1814, 2242, 2888, 3632, 4345

Algoritme: Radix sort

Input: Et array "A" med "n" positive heltall

Output: Et sortert array med de samme "n" positive heltallene

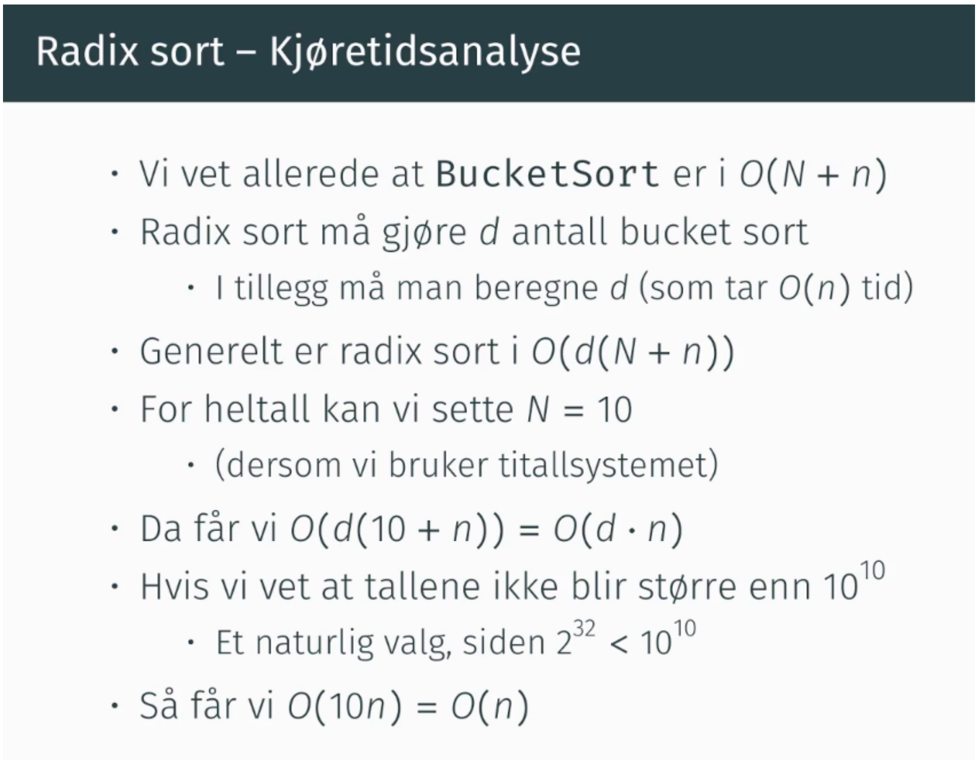
Procedure RadixSort(A)

    d <- antall siffer i det største tallet

    for i <- d down to 0 do

        A <- BucketSort(A) etter de "i"te siffere end

return A


Radix sort – Kjøretidsanalyse
Vi vet allerede at BucketSort er i $O(N + n)$
Radix sort må gjøre d antall bucket sort
I tillegg må man beregne d (som tar $O(n)$ tid)
Generelt er radix sort i $O(d(N + n))$
For heltall kan vi sette $N = 10$
(dersom vi bruker titallsystemet)
Da får vi $O(d(10 + n)) = O(d \cdot n)$
Hvis vi vet at tallene ikke blir større enn 10^{10}
Et naturlig valg, siden $2^{32} < 10^{10}$
Så får vi $O(10n) = O(n)$