Programmering og Problemløsning, 2019

Rekursion og Sortering

Martin Elsman

Datalogisk Institut
Københavns Universitet
DIKU

10. oktober, 2019

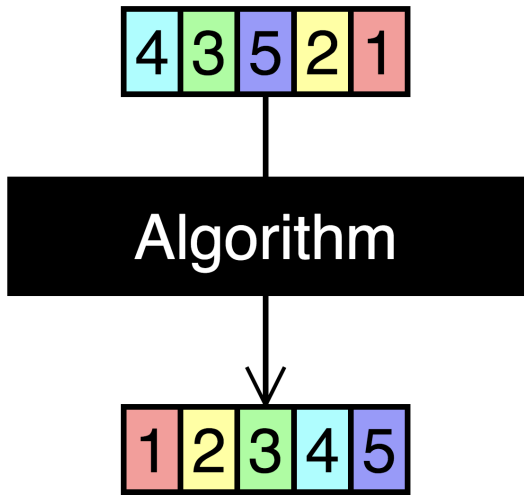
Rekursion

En metode for hvilken en løsning til et problem findes ved at løse mindre instanser af det samme problem.

- Rekursion kan anvendes til at løse en lang række forskellige problemstillinger.
- I dag vil vi se på brug af rekursion i forbindelse med sortering.

Sorteringsalgoritmer

- Insertion sort
- Bubble Sort
- Selection sort
- Mergesort
- Quicksort



Insertion Sort

- Gennemløb en liste et element af gangen.
- For hvert element, indsæt det på rette plads i resultatlisten.

En implementation i F#:

```
let rec insert xs y =
  if List.isEmpty xs then [y]
  else let x = List.head xs
       in if y < x then y :: xs
          else x :: insert (List.tail xs) y

let isort xs = List.fold (fun acc x -> insert acc x) [] xs

let xs = [7;55;34;23;5;42;32;34;8]
do printf "%A\n" (isort xs)
```

Bemærk:

- Nøgleordet **`rec`** er nødvendigt før en funktion kan henvise til sig selv...
- `List.fold` benyttes til gennemløb og opbygning af ny sorteret liste.

Analyse af Insertion Sort

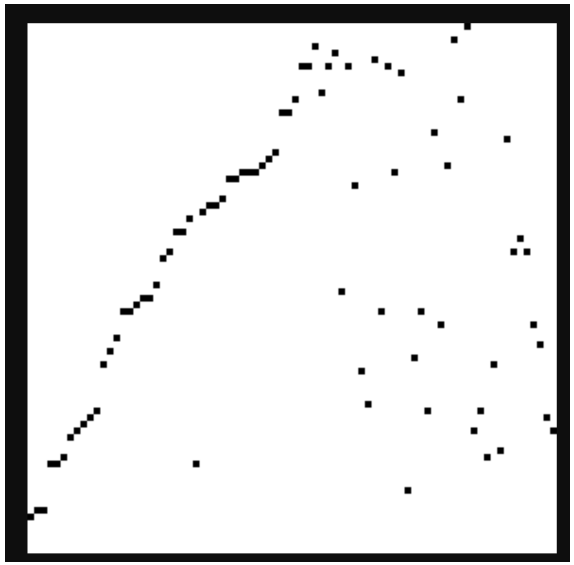
Funktionen `insert` køres N gange og for hver kørsel gennemløbes (i gennemsnit) en fjerdedel af listen ($N/4$ elementer).

Summary:

Best time:	$O(N)$
Worst time:	$O(N^2)$
Average time:	$O(N^2)$

I bedste tilfælde er listen omvendt sorteret hvorved `insert` altid kører i konstant tid...

I animationen insættes elementerne bagfra...



(animation)

Kørsel:

```
bash-3.2$ fsharpc --nologo isort.fs && mono isort.exe
[5; 7; 8; 23; 32; 34; 34; 42; 55]
```

Bubble Sort

- Listen `xs` gennemløbes $N =$ `List.length xs` gange.
- For hvert gennemløb, ombyt sidestillede elementer der er forkert ordnet (bubble).

En implementation i F#:

```
let rec bubble (xs:int list) =
  if List.isEmpty xs then []    // x::y::ys
  else let x = List.head xs     //  => y::bubble(x::ys) (y<x)
       let ys = List.tail xs
       in if List.isEmpty ys then [x]
          else let y = List.head ys
               in if x < y then x :: bubble ys
                  else y :: bubble (x::List.tail ys)

let bsort xs =
  List.fold (fun acc _ -> bubble acc) xs xs
```

Bemærk:

- `List.fold` benyttes til at foretage N kald af `bubble`.

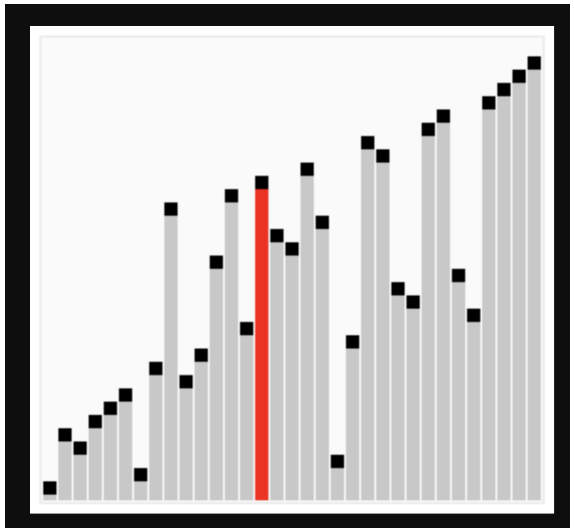
Analyse af Bubble Sort

Funktionen `bubble` køres N gange og for hver kørsel gennemløbes hele listen (N elementer).

Summary:

Best time:	$O(N^2)$
Worst time:	$O(N^2)$
Average time:	$O(N^2)$

I værste tilfælde skal det sidste element flyttes helt i front...



(animation)

Selection Sort

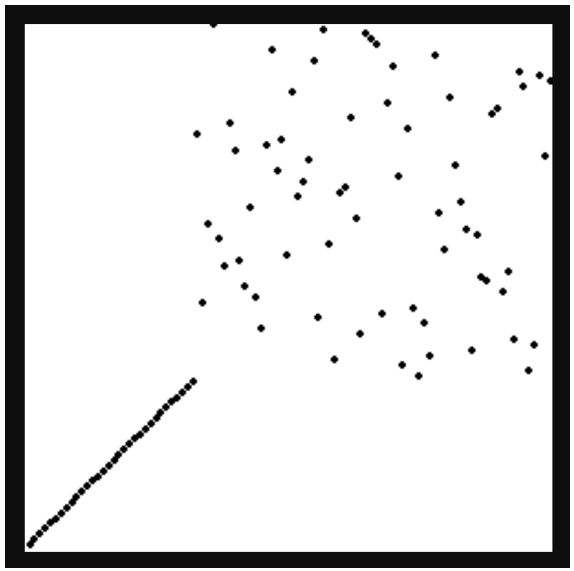
- Udtræk det mindste element i listen.
- Gentag processen rekursivt indtil der ikke længere er elementer i listen.

En implementation i F#:

```
let rec select (xs:int list) (m,ys) =
  if List.isEmpty xs then (m,ys)
  else let x = List.head xs
       let xs = List.tail xs
       in if x < m then
            if m <> System.Int32.MaxValue then
              select xs (x,m::ys)
            else select xs (x,ys)
          else select xs (m,x::ys)
let rec ssort xs =
  if List.isEmpty xs then xs
  else let (m,xs) = select xs (System.Int32.MaxValue,[])
       in m :: ssort xs
```

Analyse af Selection Sort

Funktionen `select` køres N gange og for hver kørsel gennemløbes listen (i gennemsnit $N/2$ elementer).



(animation)

Summary:

Best time:	$O(N^2)$
Worst time:	$O(N^2)$
Average time:	$O(N^2)$

Mergesort — divide-and-conquer (del-og-hersk!)

John von Neumann, 1945

- Opdel listen i to lige store dele.
- Sortér (rekursivt) hver liste.
- Flet (merge) de to resultater.

En implementation i F#:

```
let rec msort xs =
  let sz = List.length xs
  if sz < 2 then xs
  else let n = sz / 2
       let ys = xs.[0..n-1]
       let zs = xs.[n..sz-1]
       in merge (msort ys) (msort zs)
```

Bemærk:

- Mergesort benytter sig af slice-syntaksen (e.g., `xs.[0..n-1]`) for at udtrække dele af en liste.
- Mergesort benytter sig af utility-funktionen `merge` (next slide).

Utility-funktionen `merge`

```
let rec merge xs ys =
  if List.isEmpty xs then ys
  else if List.isEmpty ys then xs
  else let x = List.head xs
       let y = List.head ys
       let xs = List.tail xs
       let ys = List.tail ys
       in if x < y then x :: merge xs (y::ys)
          else y :: merge (x::xs) ys
```

Bemærk:

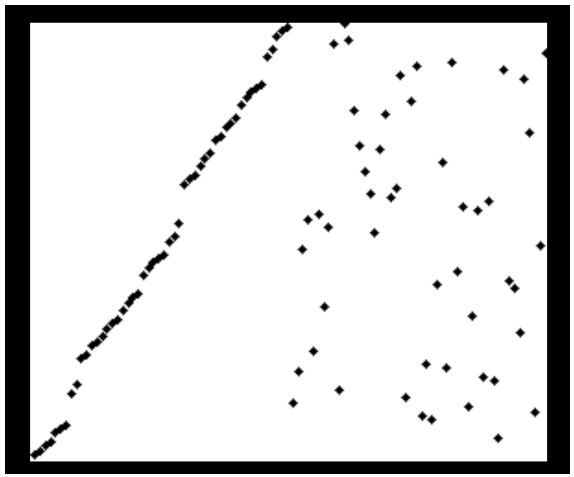
- Funktionen `merge` fletter to sorterede lister sammen således at resultatet er sorteret.

Analyse af Mergesort

Kald-træet for `msort` er $\log(N)$ dybt og `merge` kaldes i hver knude. Det viser sig at Best time = Worst time = Average time = $O(N\log(N))$.

Summary:

Best time:	$O(N\log(N))$
Worst time:	$O(N\log(N))$
Average time:	$O(N\log(N))$



(animation)

Quicksort — divide-and-conquer (del-og-hersk!)

Tony Hoare, 1959

- Vælg et element x i listen (pivot).
- Del listen i tre dele, dem mindre en x, dem lig med x og dem større end x.
- Sortér de to lister indeholdende henholdsvis små og store elementer.
- Sammensæt de tre lister.

En implementation i F#:

```
let rec qsort xs =
  if List.isEmpty xs then xs
  else let pivot = xs.[0]
       let (xs, es, ys) = partition pivot xs
       in qsort xs @ es @ qsort ys
```

Bemærk:

- Vi vælger det første element i listen som pivot; en bedre løsning er at vælge et tilfældigt element.
- Quicksort benytter sig af utility-funktionen `partition` (next slide).

Utility-funktionen `partition`

```
let partition y xs =
  List.foldBack (fun x (xs,es,ys) ->
                   if x < y then (x::xs,es,ys)
                   else if x > y then (xs,es,x::ys)
                   else (xs,x::es,ys)) xs ([],[],[])
```

Bemærk:

- Funktionen `partition` partitionerer listen (ved brug af `List.foldBack`) i de elementer der er henholdsvis mindre end y, lig med y og større end y.

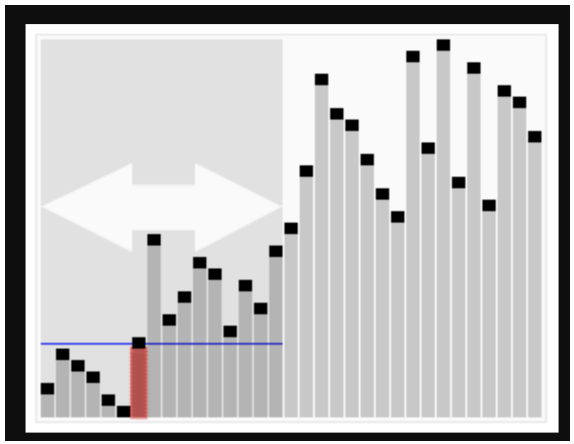
Analyse af Quicksort

Kald-træet for `qsort` er $\log(N)$ dybt (average) og `partition` kaldes i hver knude. Det viser sig at Worst time = $O(N^2) >$ Average time $O(N\log(N)) >$ Best time = $O(N)$.

Summary:

Best time:	$O(N)$	*I hvilket tilfælde?*
Worst time:	$O(N^2)$	*I hvilket tilfælde?*
Average time:	$O(N\log(N))$	

- Ved at vælge et "random" pivot kan risikoen for worst-time opførsel minimeres.
- Quicksort kan (relativt) let implementeres for arrays med in-place opdateringer og meget lidt ekstra pladsforbrug.
- Ved brug af forskellige "tweaks" kan quicksort optimeres til at køre ca. tre gange hurtigere end konkurenterne mergesort og heapsort (ikke vist her).



(animation)