ALGORITMER

Formålet med denne note er at give en introduktion til:

Algoritmer
- Specifikation, f.eks. v.h.a. pseudokode
- Analyse
  - Køretid, v.h.a. asymptotiske notation
  - Korrekthed, f.eks. v.h.a. invarianter

Dette emne er et uddrag af kurset
DM578 – Algoritmer og Datastrukturer
som ligger på 2. semester (d.v.s. i foråret 2024)
og undervises af Rolf Fagerberg

En algoritme er en „opskrift" på, hvordan man løser et givet problem.

Mere formelt (Fra Computer Science: An Overview Brookshear, Brylow):

> An algorithm is an ordered set of unambiguous, executable steps that define a terminating process.

Til at beskrive algoritmer er det nyttigt at bruge pseudokode:
„Høj-niveau sprog", hvor man bruger keywords, som i programmerings-sprog, men også almindelig tekst.

Eks

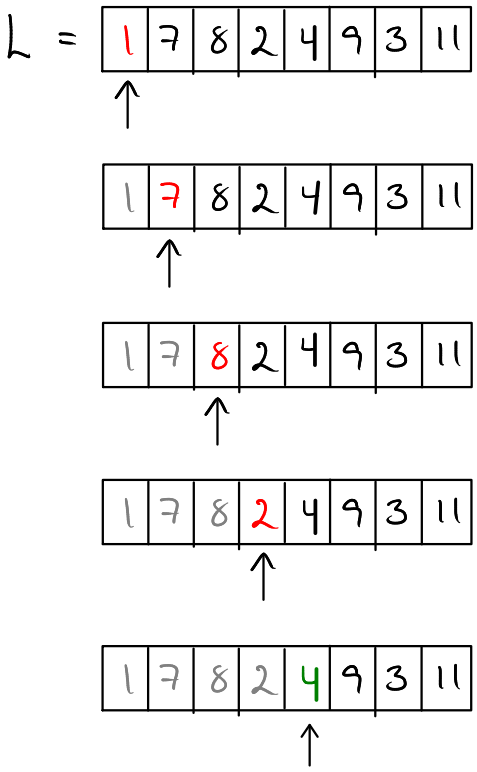
Søg efter et element x i en liste L

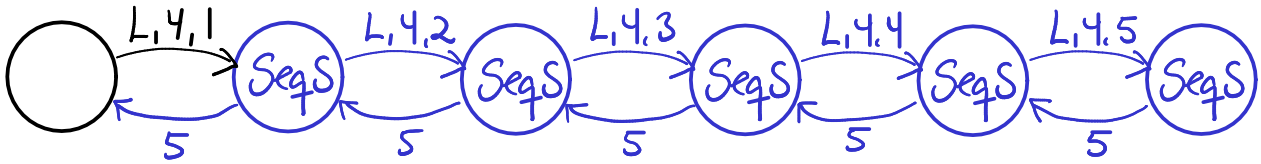
Kan gøres v.h.a. sekventiel søgning (også kaldet lineær søgning):

```
SequentialSearch(L, x, i)
Hvis i ≤ |L|
    Hvis L[i] = x
        Returner i
    Ellers
        Returner SequentialSearch(L, x, i+1)
Ellers
    Returner "Ikke fundet"
```

Er dette en algoritme?
Ordered, unambiguous, executable, terminating?

Eks: $L = [1, 7, 8, 2, 4, 9, 3, 11]$, $x = 4$





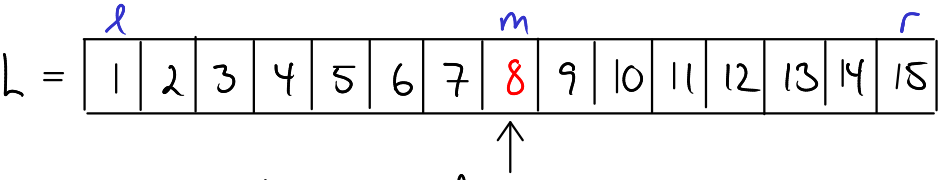
x og samtlige elementer til venstre for x checkes.
Hvis x ikke findes i L, checkes alle elementer i L.

Bemærk, at den samlede køretid er proportional med #check.
Derfor siger vi, at køretiden er $O(n)$.

O-notationen giver en øvre grænse for „størrelses-ordenen" af køretiden.

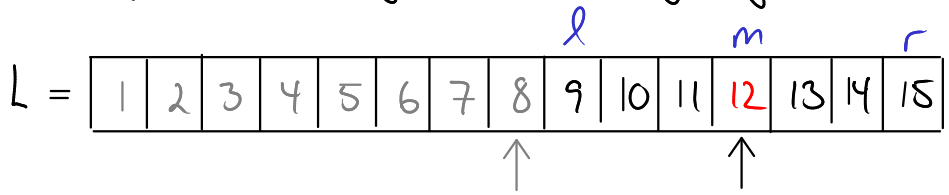
Hvis L er sorteret, kan det gøres mere effektivt v.h.a. binær søgning:

Eks: L = | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 | 12 | 13 | 14 | 15 |, X = 10

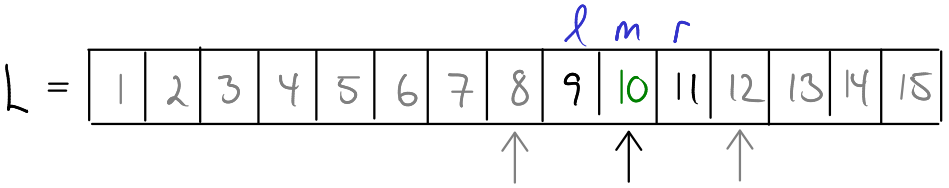


Check midterste element

X > 8, så vi fortsætter søgningen til højre for 8:

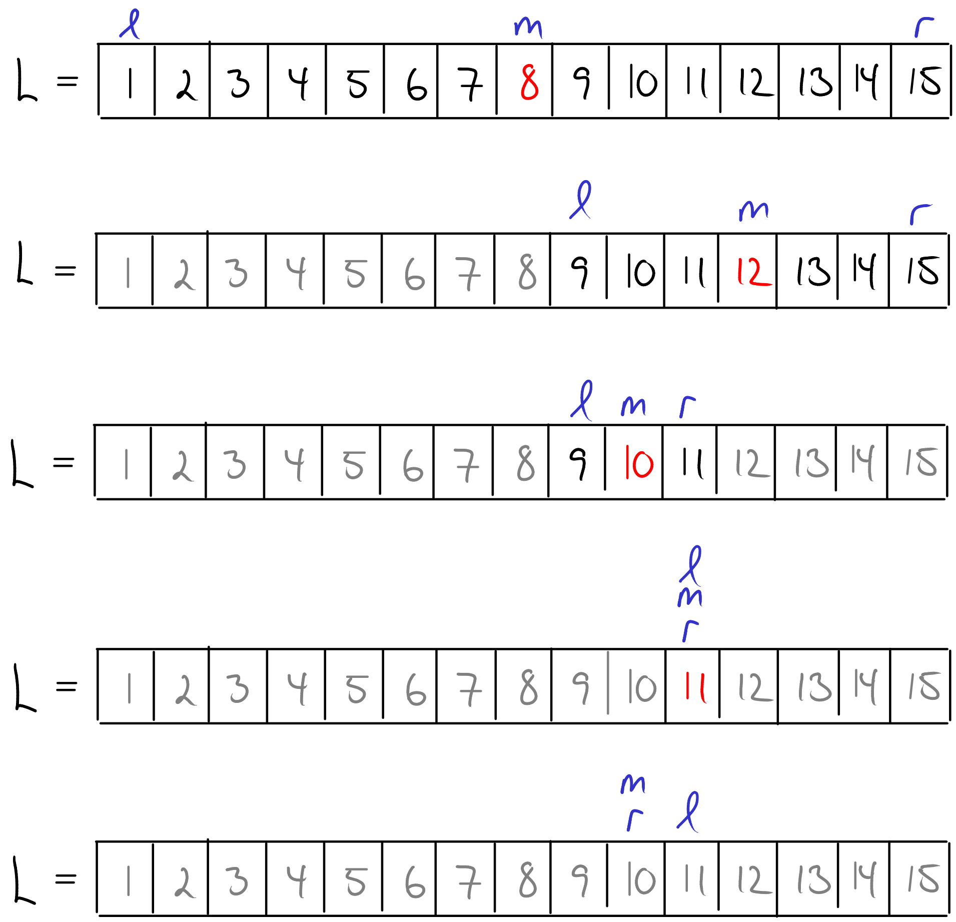


X < 12



```
BinarySearch (L, x, l, r)
Hvis l ≤ r
    m := ⌊(l+r)/2⌋
    Hvis L[m] = x
        Returner m
    Ellers
        Hvis x < L[m]
            Returner BinarySearch(L, x, l, m-1)
        Ellers
            Returner BinarySearch(L, x, m+1, r)
Ellers
    Returner "Ikke fundet"
```

Eks: L = | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 | 12 | 13 | 14 | 15 | , $x = 10{,}5$



$r < l \Rightarrow$ „Ikke fundet"

I eksemplerne ovenfor checkede vi h.h.v. 3 og 4 tal ud af 15. Bemærk, at 4 er worst-case.

Hvor mange tal kommer vi i værste tilfælde til at checke, når $|L| = n$?

Efter første check er der $\leq n/2$ elementer tilbage:

Efter andet check er der $\leq n/4$ elementer tilbage:

...

Efter i'te check er der $\leq n/2^i$ elementer tilbage.

Det sidste check foretages senest, når der er 1 element tilbage.

$$\frac{n}{2^i} < 1 \iff n < 2^i \iff \log_2 n < i$$

D.v.s. max # check: $\lfloor \log_2 n \rfloor + 1$

Dermed er køretiden $O(\log n)$.

Bestemmelse af køretid

Vælg en karakteristisk operation, op., sådan at algoritmens køretid er proportional med den samlede tid brugt på op.
I ovenstående eks. var op. sammenligning af elementer.

SequentialSearch laver højst n sammenligninger. Derfor er dens køretid $O(n)$. Det ville den også være, hvis algoritmen lavede $n/2$ eller $3n$ sammenligninger. Løst sagt betyder $O(n)$ „højst proportional med n" eller „vokser højst så hurtigt som n".

Definition:

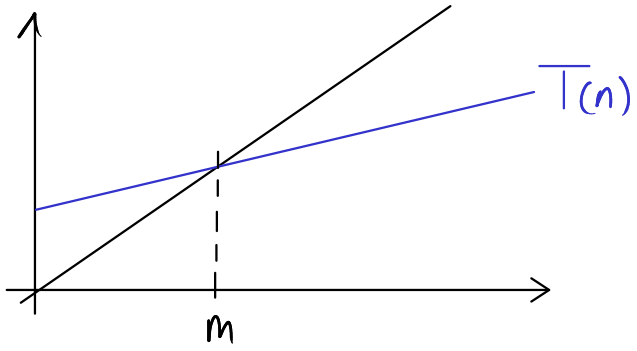
$T(n) \in O(f(n))$, hvis der findes $k, m \in \mathbb{Z}$, så $T(n) \leq k \cdot f(n)$, for alle $n \geq m$.

Eller:

$$T(n) \in O(f(n)) \Updownarrow \exists k, m \in \mathbb{Z} : \forall n \in \mathbb{Z}, n \geq m : T(n) \leq k \cdot f(n)$$

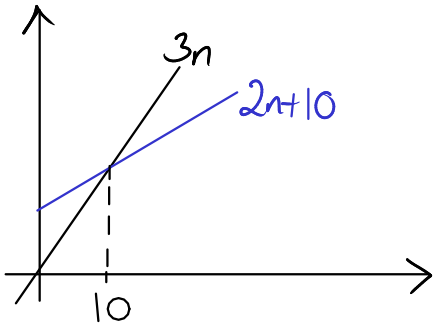
D.v.s. $f(n) \in O(T(n))$, hvis $\frac{f(n)}{T(n)}$ er (begrænset af) et tal; d.v.s. vokser ikke, når n vokser.
Også selvom ovenstående kun gælder for store n-værdier.

D.v.s. $T(n) \in O(n)$, hvis grafen for $T(n)$ fra et vist punkt ligger under en ret linie gennem $(0,0)$:



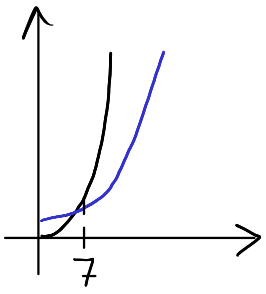
Eks: $2n+10 \in O(n)$:

$2n+10 \leq 3n$, for $n \geq 10$



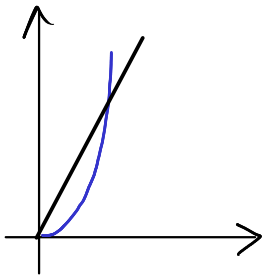
Eks: $n^2/2 + 3n + 1 \in O(n^2)$:

$n^2/2 + 3n + 1 \leq n^2$, for $n \geq 7$



Eks: $n^2 \notin O(n)$:

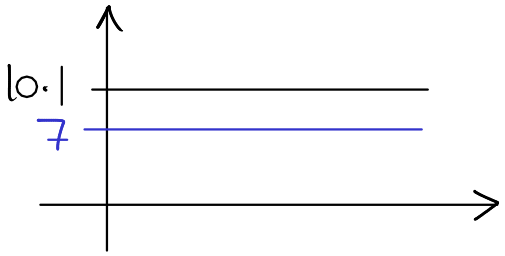
$n^2 \leq k \cdot n \iff n \leq k$



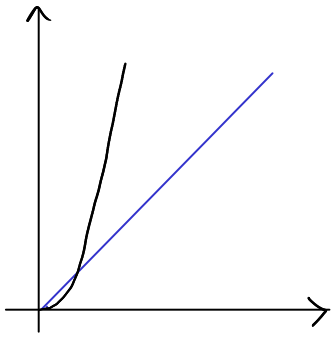
Eks: $\lfloor \log_2 n \rfloor + 1 \in O(\log n)$:

$\lfloor \log_2 n \rfloor + 1 \leq 2 \log_2 n$, for $n \geq 2$

Eks: $7 \in O(1)$



Eks: $n \in O(n^2)$



Bemærk:

$\log_a(n) \in O(\log_b(n))$, hvis $a, b \in O(1)$

fordi:

$$\log_a(n) = \frac{\log_b(n)}{\log_b(a)}, \text{ og}$$

$a, b \in O(1) \Rightarrow \log_b(a) \in O(1)$

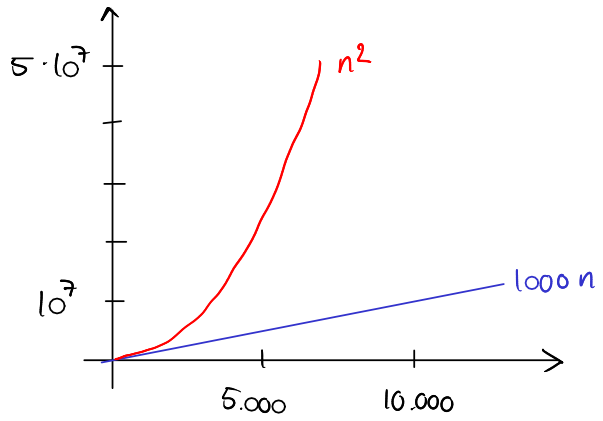
Derfor skriver vi blot $O(\log n)$ i st. for $O(\log_2 n)$

Eks: $a = 2$, $b = 10$.

$$\log_2(n) = \frac{\log_{10}(n)}{\log_{10}(2)} \approx \frac{\log_{10}(n)}{0{,}3} \approx 3{,}3 \cdot \log_{10}(n)$$

Det, der betyder noget, er det mest betydende led, d.v.s. det der vokser hurtigst.
Man må altid fjerne mindre betydende led samt konstanter, der ganges på det mest betydende led.

Eks:



Hvis n er lille, bekymrer vi os ikke så meget om køretiden, men vælger blot den simpleste algoritme.
Hvis n er stor, er n^2 meget større end 1000 n.

Eks: #op. i sek: 10^9

#op	n=100		n=10^6		n=10^9	
	#op.	tid	#op	tid	#op.	tid
$\lceil \log_2 n \rceil$	7	7 ns	20	20 ns	30	30 ns
n	100	100 ns	10^6	1 ms	10^9	1 s
n^2	10.000	10 µs	10^{12}	17 min	10^{18}	32 år

$10^9 \approx 30.000.000 \cdot 30$

D.v.s. om køretiden er $\frac{1}{2} \cdot n$ eller $10 \cdot n$ er relativt uvigtigt. Det, der betyder noget, er, om den er proportional med n eller log n.

Overstående tabel angav, hvor lang tid det tager at løse et problem af en given størrelse v.h.a. en algoritme med en given køretid.
Nu vender vi det rundt: Givet en bestemt køretid, hvor stor en instans kan vi løse inden for den givne tid?
Vi antager stadig, at der kan udføres 10^9 op. i sek.
Problem-størrelserne nedenfor er angivet med et betydende ciffer.

#op. for input af str. n	1 ms	1 s	1 min	1 døgn	1 år
$\log_2 n$	$10^{300.000}$				
n	10^6	10^9		$9 \cdot 10^{13}$	
$n \log_2 n$	$6 \cdot 10^4$			$2 \cdot 10^{12}$	
n^2	10^3			$9 \cdot 10^6$	
n^3	10^2	10^3		$4 \cdot 10^4$	
2^n	20	30			55

Eksempler på køretider:

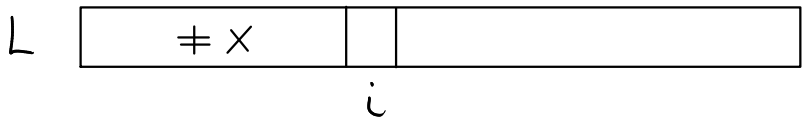
1	konstant
$\log(n)$	logaritmisk
n	lineær
$n \log(n)$	
n^2	kvadratisk
n^3	polynomiel
n^{10}	polynomiel
2^n	eksponentiel
10^n	eksponentiel

Korrekthed

Virker algoritmen, som du skal?
Til at bevise korrekthuden af en iterativ (eller rekursiv) algoritme kan man bruge en invariant.

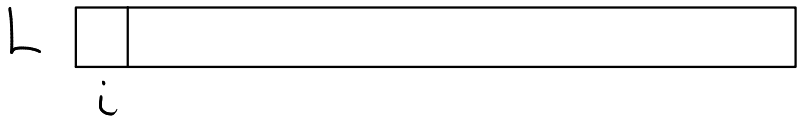
SequentialSearch(L, x, i)
I
Hvis $i \leq |L|$
  Hvis $L[i] = x$
    Returner i
  Ellers
    Returner SequentialSearch(L, x, i+1)
Ellers
  Returner „Ikke fundet"

Invariant I: $x \neq L[j]$, for $1 \leq j \leq i-1$



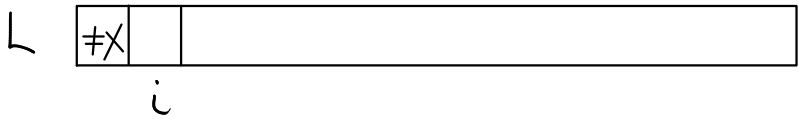
I er opfyldt, hver gang vi kommer til *I*,
Dette kan bevises v.h.a. induktion.

Første gang vi kommer til *I*:

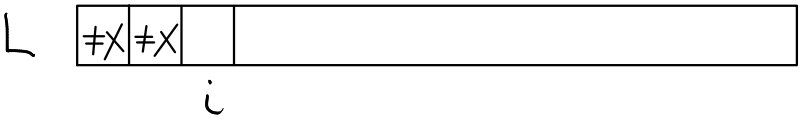


I siger, at der ikke er nogen elementer til venstre for plads i, som er lig med x.
Der er ingen elementer til venstre for plads i, så udsagnet er trivielt opfyldt.
Dette udgør basis-skridtet.

Derefter checker vi, om L[i] ≠ x.
Hvis det er tilfældet, inkrementeres i.
Nu har vi følgende situation:



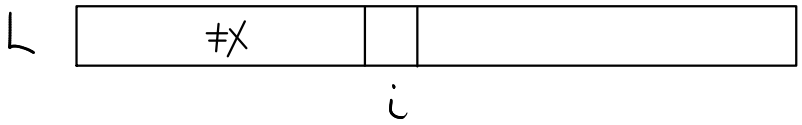
D.v.s. næste gang, vi kommer til *I*, er I igen opfyldt.
Hvis vi igen finder, at L[i] ≠ x, inkrementeres i endnu en gang. Nu har vi:



D.v.s. næste gang, vi kommer til *I*, er I igen opfyldt.
Og sådan kan vi fortsætte...

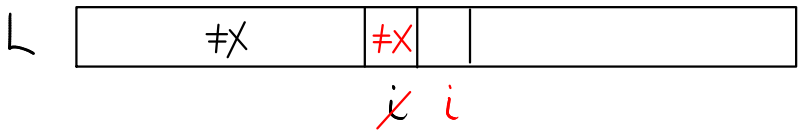
Generelt:

Antag, at I er opfyldt, når vi kommer til $*I*$:

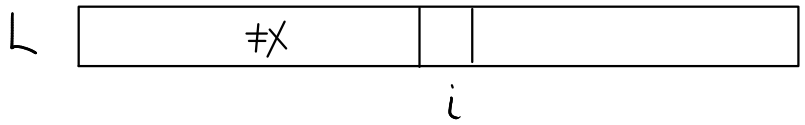


Dette er induktionsantagelsen.

Bemærk, at i kun bliver inkrementeret, hvis $L[i] \neq X$:



D.v.s. næste gang vi kommer til $*I*$, gælder I igen:



Dette udgør induktionsskridtet.

D.v.s. i induktionsskridtet viser vi:

Hvis I gjaldt, sidst vi var ved I,
så gælder den også denne gang.

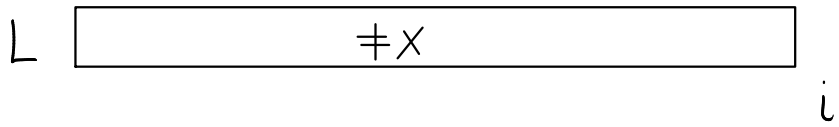
Afslutning:

Når vi afslutter sidste rekursive kald, skyldes det en af to ting:

- $i = |L|+1$

  I dette tilfælde returneres „Ikke fundet"

  Da $i > n$, følger det af I, at x ikke findes i L.



  D.v.s. korrekt output i dette tilfælde.

- $L[i] = X$

  I dette tilfælde returneres i, og det er indeks til en plads i L, hvor man kan finde x.

  D.v.s. også korrekt output i dette tilfælde.

BinarySearch (L, x, ℓ, r)

I

Hvis $\ell \leq r$

$m := \lfloor \frac{\ell + r}{2} \rfloor$

Hvis $L[m] = x$

Returner m

Ellers

Hvis $x < L[m]$

Returner BinarySearch$(L, x, \ell, m-1)$

Ellers

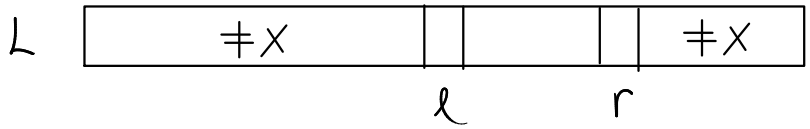
Returner BinarySearch$(L, x, m+1, r)$

Ellers

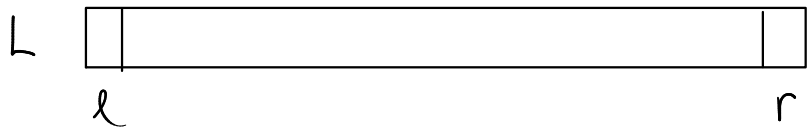
Returner „Ikke fundet"

Invariant I:

Hvis x findes i L, findes den i $L[\ell..r]$:



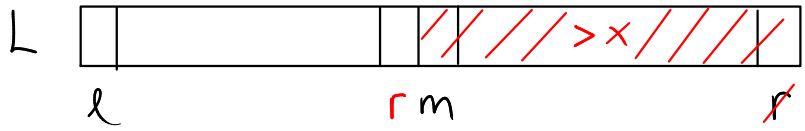
Første gang vi kommer til *I*, er det trivielt sandt:



Derefter checker vi, om $x \neq L[m]$.
Hvis det er tilfældet, er der to muligheder:

- $x < L[m]$

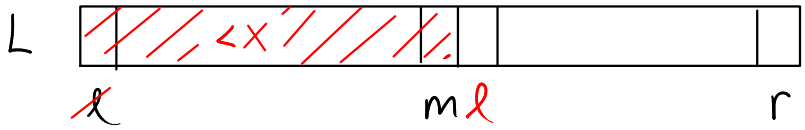
  I dette tilfælde opdateres r:



  Herefter er I stadig opfyldt

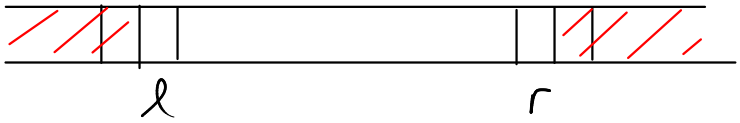
- $x > L[m]$

  I dette tilfælde opdateres l:

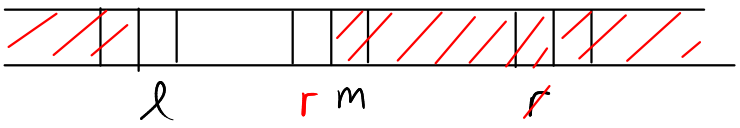


  Herefter er I stadig opfyldt

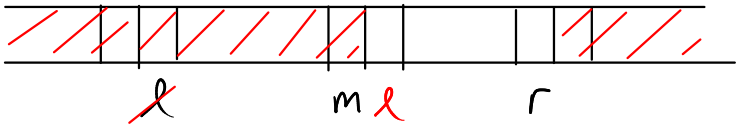
Generelt:
Lige inden vi checker $l \leq r$, gælder I:



Efter gennemløb af løkken gælder I stadig:



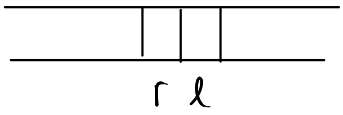
eller



Når sidste rekursive kald afsluttes, skyldes det en af to ting:

- $l > r$

  I dette tilfælde returneres „Ikke fundet"
  Da $l > r$, er $L[l..r]$ tomt:

  Dermed følger det af I, at x ikke findes i L.
  D.v.s. korrekt output i dette tilfælde.

- $L[m] = x$

  I dette tilfælde returneres i, og det er indeks til en plads i L, hvor man kan finde x.
  D.v.s. også i dette tilfælde korrekt output.

De to algoritmer er rekursive, d.v.s. kaldv sig selv.
De kunne også skrives som iterative algoritme, hvor det meste af arbejdet foregår i en løkke:

SequentialSearch It (L, x)
i := 1
Sålænge i ≤ |L| og L[i] ≠ x
  i++
Hvis i ≤ |L|
  Returner i
Ellers
  Returner „Ikke fundet"

BinarySearch It (L, x)
l := 1, r := |L|, m := $\lfloor \frac{l+r}{2} \rfloor$
Sålænge l ≤ r og L[m] ≠ x
  Hvis x < L[m]
    r := m - 1
  Ellers
    l := m + 1
Hvis l ≤ r
  Returner m
Ellers
  Returner „Ikke fundet"

Korrekthed kan stadig bevises v.h.a. en invariant (som da kaldes en løkke-invariant):

SequentialSearch It (L, x)

```
i := 1
Sålænge *I* i ≤ |L| og L[i] ≠ x
    i++
Hvis i ≤ |L|
    Returner i
Ellers
    Returner „Ikke fundet"
```

BinarySearch It (L, x)

```
l := 1, r := |L|, m := ⌊(l+r)/2⌋
Sålænge *I* l ≤ r og L[m] ≠ x
    Hvis x < L[m]
        r := m-1
    Ellers
        l := m+1
Hvis l ≤ r
    Returner m
Ellers
    Returner „Ikke fundet"
```