

ALGORITMER

Formålet med denne note er at give en introduktion til:

Algoritmer

- Specifikation f.eks. v.h.a. pseudokode
- Analyse
 - Kørtid v.h.a. asymptotisk notation
 - Korrekthed f.eks. v.h.a. invarianter

Dette emne er et uddrag af kurset

DM578: Algoritmer og Datastrukturer

som ligger på 2. semester og undervises af Rolf Fagerberg.

En **algoritme** er en **opskrift** på, hvordan et givet problem kan løses.

Mer formelt (fra Computer Science: An Overview af Brooksheer, Brylaw):

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

Til at beskrive algoritmer kan det være nyttigt at bruge **pseudokode**:

„Høj-niveau sprog“, hvor man bruger **keywords** som i programmeringssprog, men også almindelig **tekst**.

Eks

Søg efter et element, x , i en liste, L

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

SequentialSearch (L, x, i)

Hvis $i \leq |L|$

Hvis $L[i] = x$

Returner i

Ellers

Returner SequentialSearch($L, x, i+1$)

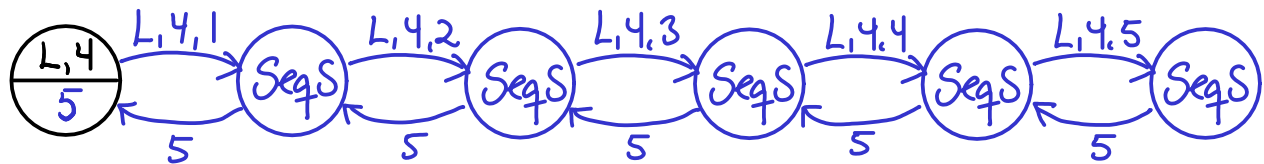
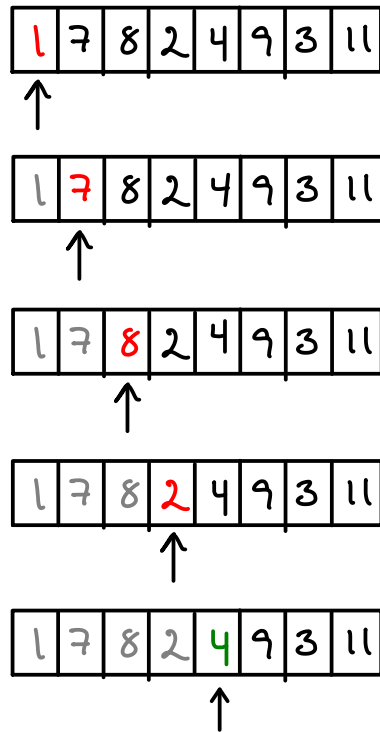
Ellers

Returner „Ikke fundet“

Er dette en algoritme?

Ordered, unambiguous, executable, terminating?

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



Hvis x findes på plads i i L , sammenlignes x med samtlige elementer på plads $1, 2, \dots, i$ i L .
Hvis x ikke findes i L , sammenlignes x med samtlige elementer i L .

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

Med O -notation giver man en øvre grænse for „størrelsesordenen“ af køretiden.

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

Eks :

$L = [1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15]$, $x = 10$

1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
---	---	---	---	---	---	---	---	---	----	----	----	----	----	----

↑
Sammenlign midterste element med x
 $x > 8$, så vi fortsætter søgningen til højre for 8:

1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
---	---	---	---	---	---	---	---	---	----	----	----	----	----	----

↑
↑
 $x < 12$

1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
---	---	---	---	---	---	---	---	---	----	----	----	----	----	----

↑
↑
↑
 $x = 10$

BinarySearch (L, x, l, r)

Hvis $l \leq r$

$$m := \left\lfloor \frac{l+r}{2} \right\rfloor$$

Hvis $L[m] = x$

Returner m

Ellers

Hvis $x < L[m]$

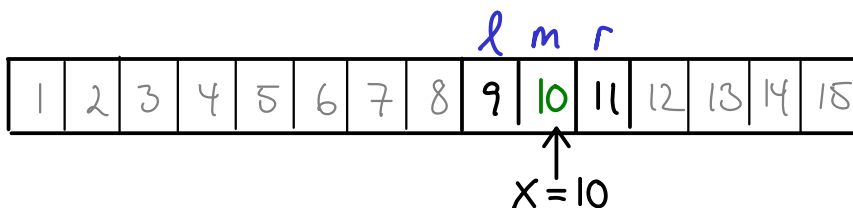
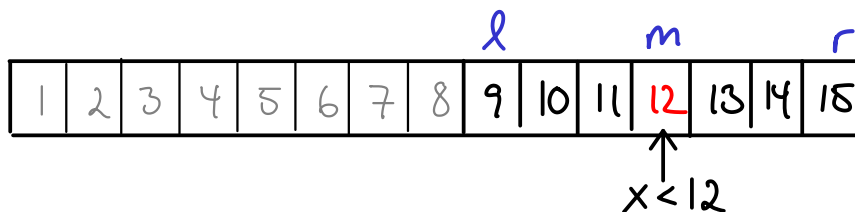
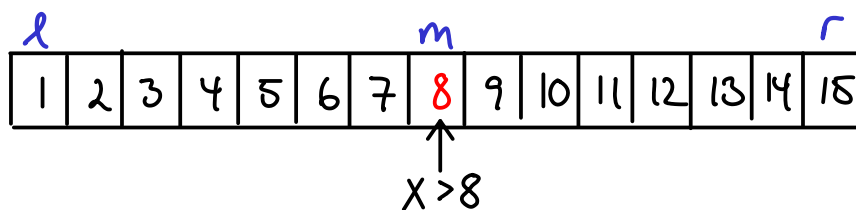
Returner BinarySearch($L, x, l, m-1$)

Ellers

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

Ellers

Returner „Ikke fundet“



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

$L = [1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15]$
 l m r

$L = [1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15]$
 l m r

$L = [1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15]$
 l m r

$L = [1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15]$
 l
 m
 r

$L = [1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15]$
 m
 r l

$r < l \rightarrow$ „ikke fundet“

I eksemplerne overfor sammenlignede vi x med
h.h.v. 3 og 4 tal ud af 15.

Bemærk, at 4 sammenligninger er worst-case.

Mer generelt:

Hvor mange tal sammenlignes med x i worst-case, når $|L| = n$?

Efter første sammenligning er der $\leq n/2$ tal tilbage:



Efter anden sammenligning er der $\leq n/4$ tal tilbage:



$\vdots$

Efter i 'te sammenligning er der $\leq n/2^i$ tal tilbage.

Den sidste sammenligning foretages senest, når der er 1 element tilbage.

At komme under 1 element kræver $\leq i$ sml., hvor $n/2^i < 1$.

$$\frac{n}{2^i} < 1 \quad \Leftrightarrow$$

$$n < 2^i \quad \Leftrightarrow$$

$$\log_2 n < i \quad \Leftrightarrow$$

$$\lfloor \log_2 n \rfloor \leq i-1 \quad \Leftrightarrow$$

$$\lfloor \log_2 n \rfloor + 1 \leq i$$

O.v.s. binder søgning efter x i en liste med n tal bruger aldrig mere end $\lfloor \log_2 n \rfloor + 1$ sammenligninger. Dermed er køretiden $O(\log n)$.

Bestemmelse af køretid

Vælg en **karakteristisk operation**, sådan at algoritmens samlede køretid er proportional med den samlede tid brugt på denne operation.

I overstående eksempler var den karakteristiske operation **sammenligning af to tal**.

Lineær søgning laver højst n sammenligninger.

Derfor er dens køretid, som nævnt tidligere, $O(n)$.

Det ville den også være, hvis algoritmen lavede $n/2$ eller $3n$ sammenligninger i worst-case.

Løst sagt betyder $O(n)$ „**højst proportional med n** “ eller „**vokser ikke hurtigere end n** “.

Definition

$$T(n) \in O(f(n)) \quad \Leftrightarrow$$

$$\exists k, n_0 : \forall n \geq n_0 : T(n) \leq k \cdot f(n)$$

$$\Leftrightarrow \frac{T(n)}{f(n)} \leq k$$

Asymptotisk notation

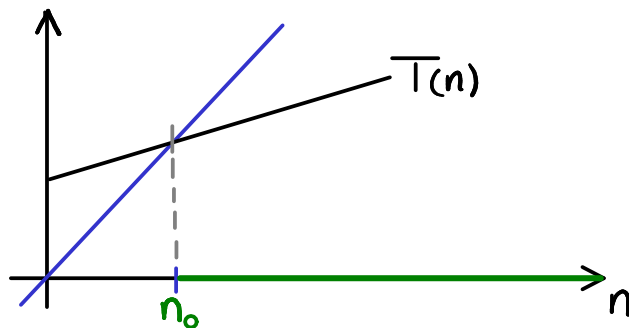
O	(Store) O	Øvre grænse
Θ	Theta	Øvre og nedre grænse
Ω	(Store) Omega	Nedre grænse
o	Lille o	Skarp øvre grænse
ω	Lille omega	Skarp nedre grænse

I dette kursus bruger vi kun O -notation.

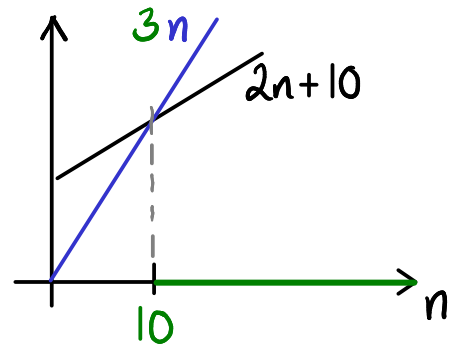
Eks: $f(n) = n$

$$T(n) \in O(n) \Leftrightarrow \exists k, n_0 : \forall n \geq n_0 : T(n) \leq k \cdot n \quad (\text{per def.})$$

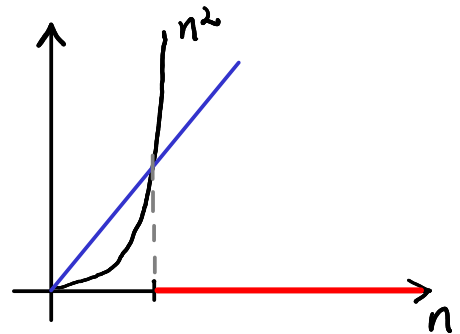
D.v.s. $T(n) \in O(n)$, hvis grafen for $T(n)$ fra et vist punkt ligger under en ret linje gennem $(0,0)$:



Eks: $2n+10 \in O(n)$, da
 $2n+10 \leq 3n$, for $n \geq 10$
 $\uparrow$ $\uparrow$
 k n_0



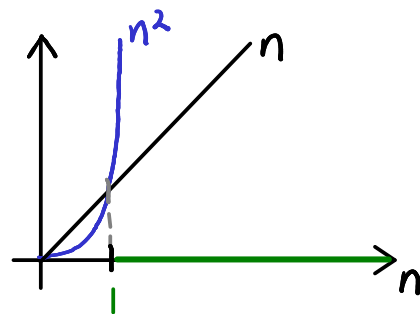
Eks: $n^2 \notin O(n)$, da
 $n^2 \leq k \cdot n \Leftrightarrow n \leq k$
 For ethvert valg af
 k og n_0 findes der
 et $n \geq n_0$, så $n > k$



D.v.s.

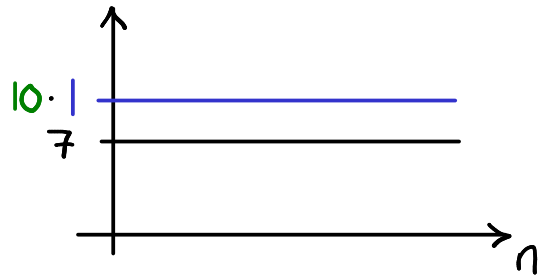
$$\nexists k, n_0 : \forall n \geq n_0 : n^2 \leq k \cdot n$$

Eks: $n \in O(n^2)$, da
 $n \leq n^2$, for $n \geq 1$



Eks: $7 \in O(1)$, da

$7 \leq 10 \cdot 1$, for alle n



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

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

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 stedet for $O(\log_2 n)$.

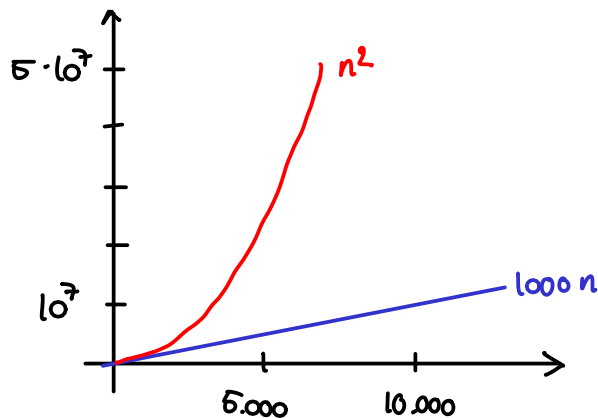
Eks: $a=2$, $b=10$.

$$\log_2(n) = \frac{\log_{10}(n)}{\log_{10}(2)} \approx \frac{\log_{10}(n)}{0.3} \approx 3.3 \cdot \log_{10}(n)$$

Det, der betyder noget, er det **mest betydnende led**, d.v.s. det, der **vokser hurtigst**.

Man må altid fjerne mindre betydnende led samt konstanter, der ganges på det mest betydnende led.

Eks:



Hvis n er lille, bekymrer vi os typisk ikke så meget om køretiden, og vælger evt. blot den simpleste algoritme.

Hvis n er stor, er n^2 meget større end $1000n$.

Eks: 10^9 operationer i sekundet

#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 μ s	10^{12}	17 min	10^{18}	32 år

*: $10^9 > 30.000.000 \cdot 30$

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

Overstående tabel angiver, **hvor lang tid** det tager at løse et problem af en given størrelse.

Nu vender vi det rundt:

Hvor stor en instans kan vi løse v.h.a. en algoritme med en given køretid inden for en given tid?

Vi antager stadig, at der kan udføres 10^9 operationer i sekundet.

Problem-størrelserne, som er fyldt ind i tabellen allerede, er angivet med et betydende ciffer bortset fra størrelserne i nederste række, som er angivet med to betydende cifre.

#op. for input af str. n	1 ms	1 s	1 min	1 dag	1 år
$\log_2 n$	$10^{301.030}$				
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	{	polynomiel
$\log(n)$	logaritmisk		
n	lineær		
$n \cdot \log(n)$			
n^2	kvadratisk		
n^3		{	eksponentiel
n^{10}			
2^n			
10^n			

Korrekthed

Virker algoritmer, som den skal?

Til at bevise korrekthed 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

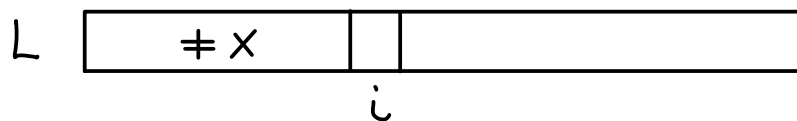
Ellers

Returner $\text{SequentialSearch}(L, x, i+1)$

Ellers

Returner „Ikke fundet“

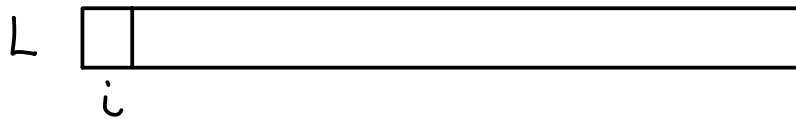
Invariant I: $x \neq L[j]$, for $1 \leq j \leq i-1$



I er opfyldt, hver gang vi kommer til ***I***.

Dette kan bevises v.h.a. **induktion**:

Første gang vi kommer til ***I***:



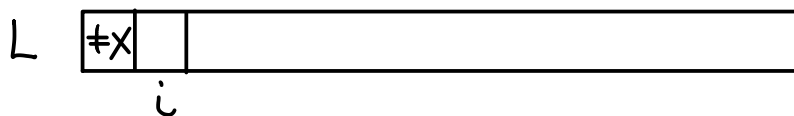
Invarianten siger, at der ikke er nogen elementer før plads i , som er $=x$.

Der er ingen elementer før plads i (da $i=1$), så udsagnet er trivielt opfyldt.

Dette udgør basistilfældet.

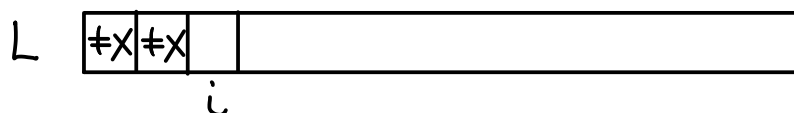
Derefter sammenligner vi x med $L[i]$.

Hvis $x \neq L[i]$, inkrementeres i :



D.v.s. **andet gang** vi kommer til ***I***, er I også opfyldt.

Hvis vi igen finder, at $x \neq L[i]$, inkrementeres i endnu en gang:

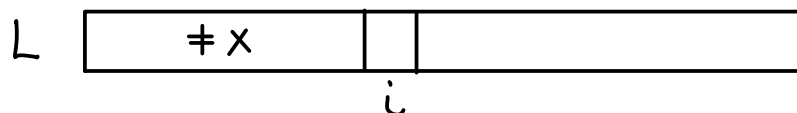


D.v.s. **tredje gang** vi kommer til ***I***, er I også opfyldt.

Og sådan kan vi fortsætte...

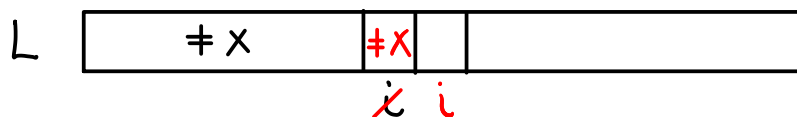
Generelt:

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

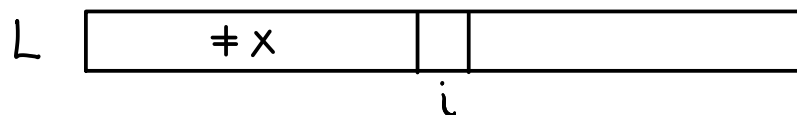


Dette er induktionsantagelsen

Bemærk, at i kan inkrementeres, hvis $x \neq L[i]$:



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



Dette udgør induktionsskridtet.

D.v.s. i induktionsskridtet viser vi:

Hvis I gælder *sidst*, vi var ved $*I*$,
gælder den også *denne gang*.

Afslutning:

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

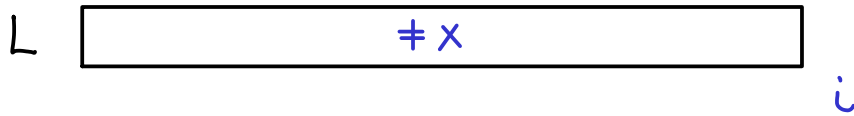
$L[i] = x$:

I dette tilfælde returneres i , og det er korrekt, da x findes på plads i i L .

$i = |L| + 1$:

I dette tilfælde returneres „Ikke fundet“.

Da $i > |L|$, følger det af invarianten, at x ikke findes i L :



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

Et korrekthedsbevis v.h.a. en invariant består af følgende dele:

- Opskriv I .

I skal opfylde to ting:

- I skal være opfyldt, hver gang i er ved starten af et kald, d.v.s. I skal være en invariant.
- I , anvendt ved algoritmens afslutning, skal kunne bruges til at vise, at algoritmens output er korrekt.

- Initialization (= Basistilfælde):

Bevis, at I gælder første gang

- Maintenance (= Induktionsskridt):

Bevis, at hvis I var opfyldt sidste gang, er den det også denne gang.

- Termination

Brug det faktum, at invarianten er opfyldt ved algoritmens afslutning, til at bevise, at output er korrekt.

BinarySearch (L, x, l, r)

I

Hvis $l \leq r$

$$m := \left\lfloor \frac{l+r}{2} \right\rfloor$$

Hvis $L[m] = x$

Returner m

Ellers

Hvis $x < L[m]$

Returner $\text{BinarySearch}(L, x, l, m-1)$

Ellers

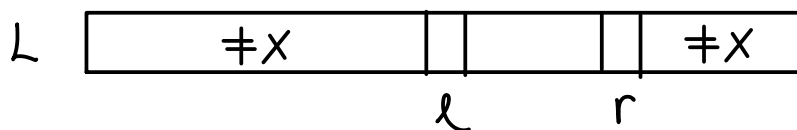
Returner $\text{BinarySearch}(L, x, m+1, r)$

Ellers

Returner „Ikke fundet“

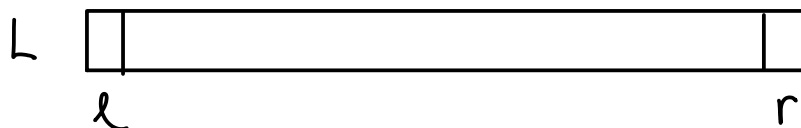
Invariant I:

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



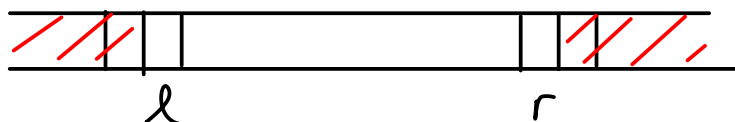
Initialization:

Første gang vi kommer til ~~I~~ , er I trivielt sand:



Maintenance:

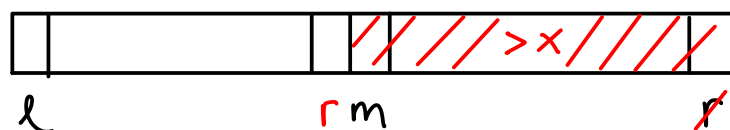
Antag, at I gjaldt ved starten af forrige kald:



Siden da er der sket en af to ting afhængigt af, om $x < L[m]$ eller $x > L[m]$ i forrige kald (der gjaldt ikke, at $x = L[m]$, for så var vi ikke fortsat til nuværende kald).

$x < L[m]$:

I dette tilfælde opdateres r :



Herefter er I stadig opfyldt

$x > L[m]$:

I dette tilfælde opdateres l :



Herefter er I stadig opfyldt

Termination:

Når sidste kald afsluttes, skyldes det er af to ting:

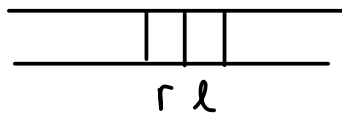
$L[m] = x$:

I dette tilfælde returneres m , og det er korrekt, da x findes på plads m i L .

$l > r$:

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

De to algoritmer er **rekursive**, d.v.s. kalder sig selv.

De kunne også skrives som **iterative** algoritmer, hvor det meste af arbejdet foregår i en **løkke**:

SequentialSearchIt (L, x)

$i := 1$

Så længe $i \leq |L|$ og $L[i] \neq x$
 $i++$

Hvis $i \leq |L|$

Returner i

Ellers

Returner „Ikke fundet“

BinarySearchIt (L, x)

$l := 1, r := |L|, m := \lfloor \frac{l+r}{2} \rfloor$

Så længe $l \leq r$ og $L[m] \neq x$

Hvis $x < L[m]$

$r := m - 1$

Ellers

$l := m + 1$

Hvis $l \leq r$

Returner m

Ellers

Returner „Ikke fundet“

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

SequentialSearchIt (L, x)

$i := 1$

Så længe ***I*** $i \leq |L|$ og $L[i] \neq x$
 $i++$

Hvis $i \leq |L|$

Returner i

Ellers

Returner „Ikke fundet“

BinarySearchIt (L, x)

$l := 1, r := |L|, m := \lfloor \frac{l+r}{2} \rfloor$

Så længe ***I*** $l \leq r$ og $L[m] \neq x$

Hvis $x < L[m]$

$r := m - 1$

Ellers

$l := m + 1$

Hvis $l \leq r$

Returner m

Ellers

Returner „Ikke fundet“