F05 - Sortering och sökning

5DV149 Datastrukturer och algoritmer
Kapitel 15

Niclas Börlin
niclas.borlin@cs.umu.se

2024-02-01 Tor

Innehåll

- Sortering
  - Varför ska man sortera?
  - Sortering kontra sorterad datatyp
  - Stabilitet
  - Grundprinciper för sortering
  - Genomgång av några sorteringsalgoritmer
- Linjär och binär sökning
- Läsanvisningar:
  - s. 59, Kap 15.

Sortering

Varför sortering?

- För att snabba upp andra algoritmer!
  - Sökning går fortare
- Nödvändigt för stora datamängder

Sortering kontra Sorterad datatyp

- Sortering förändrar ordningen mellan objekten i en struktur efter en sorteringsordning (fallande, ökande)
  - Gränsytan är oförändrad
  - Exempel: Vi sorterar en Lista
- Sorterad datatyp:
  - De strukturförändrande operationerna i gränsytan (`Insert`, `Remove`) upprätthåller en sorteringsordning mellan de lagrade elementen
  - Exempel: Gränsytan för Sorterad lista behöver modifieras jämfört med Lista

Saker att beakta

- Absolut komplexitet:
  - Totala komplexiteten i alla implementationssteg
    - Tabell (konstruerad som) Lista (implementerad som) dubbellänkad lista
- Passar en viss typ av sortering för en viss typ av konstruktion?
  - Fält-baserad lista
  - Länkad lista
- Ska man
  1. sortera och sedan söka, eller
  2. behålla listan osorterad och göra linjär sökning?
- Sortering är ofta $O(n \log n)$
  - Sökning i sorterad mängd är ofta $O(\log n)$
- Linjärsökning är $O(n)$

Stabilitet

- Den inbördes relationen mellan två objekt med samma sorteringsnyckel bibehålls vid sortering:
  - Lista sorterad efter efternamn:
    (Alm, Lars), (Bok, Bo), (Ek, Eva), (Gran, Anna), (Löv, Eva)
  - Lista stabilt omsorterad efter andra elementvärdet:
    (Gran, Anna), (Bok, Bo), (Ek, Eva), (Löv, Eva), (Alm, Lars)
- Ett till exempel på stabil sortering (index för att förtydliga):

Före:	1	0_1	5	2	3_1	0_2	4	3_2	7
Efter:	0_1	0_2	1	2	3_1	3_2	4	5	7

- Alla sorteringsalgoritmer går inte att göra stabila
- Mer om detta senare

Grundprinciper

- Urvalssortering
  - Välj ut det in-element som är på tur att sättas in
  - Sätt in det först/sist
- Insticksortering
  - Välj ett godtyckligt in-element och sätt in det på rätt plats
- Utbytessortering
  - Byt plats på objekt som ligger fel inbördes
- Samsortering
  - Sammanslagning av redan sorterade strukturer
- Nyckelsortering:
  - Kräver mer information/kunskap om nyckelmängden
  - T.ex. sortera 200 tentor i kodordning där koden har max 3 siffror

Inplace

- En del sorteringsalgoritmer finns i två versioner
  - En version som bygger en "ny" sorterad kopia av indatat
    - Behöver typiskt $O(n)$ minne
    - Oftast enklare algoritm
  - En version som jobbar direkt i indatat, oftast ett fält
    - Kallas för in-place-versioner
    - Behöver endast $O(1)$ minne
    - Ibland komplexa algoritmer

Sorteringsalgoritmer

- Idag:
  - Urvalssortering (*Selection Sort*)
  - Instickssortering (*Insertion Sort*)
  - Bubbelsortering (*Bubble Sort*)
  - Mergesort
  - Quicksort
  - Facksortering (*Bucket sort*)
- Senare:
  - Heapsort
  - (Radix Exchange Sort)

Blank

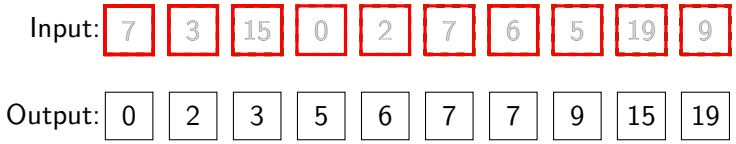
Blank

Selection sort — urvalssortering

Selection sort av lista

- Algoritmen i grova drag:
  - Välj ut det bästa elementet i in-listan
  - Stoppa in elementet sist i ut-listan

Selection sort — exempel

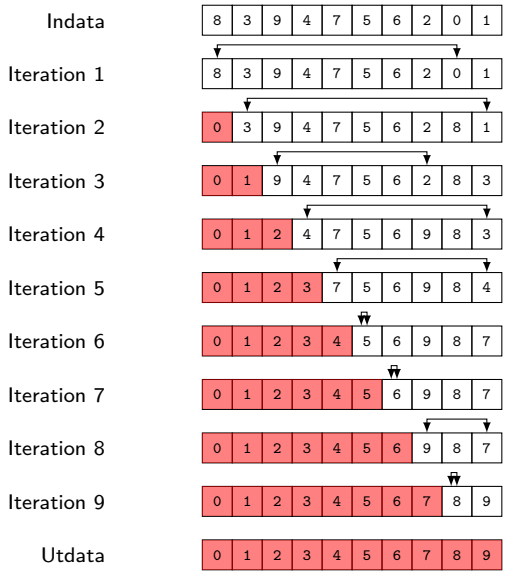


- Komplexitet:
  - $O(n^2)$

Inplace Selection sort av fält

- Jobba i fältet
- Betrakta fältet som två delar:
  - en sorterad del i början
  - en osorterad del i slutet
  - Den sorterade delen växer med ett element för varje varv av huvudalgoritmen
- Algoritmen i grova drag:
  - Initialt är alla element osorterade
  - Leta upp det bästa elementet bland de osorterade elementet
  - Byt plats på det bästa och det första osorterade elementet
  - Öka den sorterade delen av fältet med ett element

Inplace Selection sort — exempel



- Stabil?
  - Nej!

Selection sort — algoritm

- Algoritm:

```
Algorithm selection_sort(a: Array, n: Int)
  // i indicates first unsorted element in a

  for i ← 0 to n - 2 do

    // find the smallest value among the unsorted
    ix ← i
    // start with the ith element as the smallest
    for j ← i + 1 to n - 1 do
      if a[j] < a[ix] then
        ix ← j

    // swap the ith element for the smallest
    t ← a[j]
    a[j] ← a[ix]
    a[ix] ← t


  return a
```


Blank

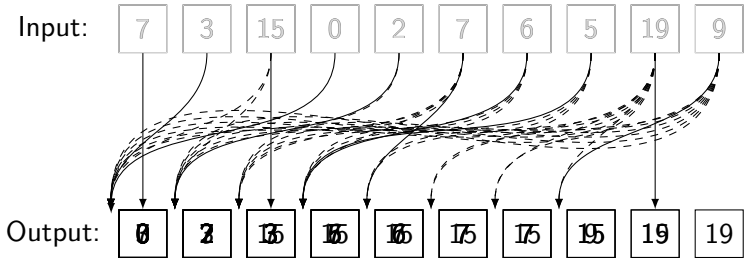
Blank

Insertion sort — insickssortering

Insertion sort av lista

- Algoritmen i grova drag:
  - Plocka ut första elementet från in-listan
  - Leta upp platsen i ut-listan där elementet ska skjutas in

Insertion sort av lista — exempel

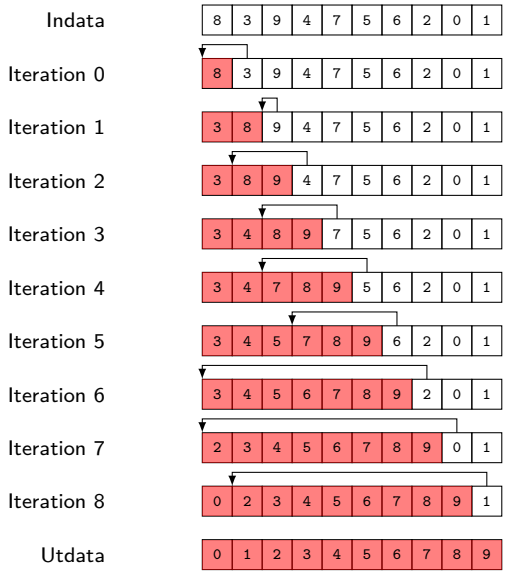


- Komplexitet:
  - $O(n^2)$

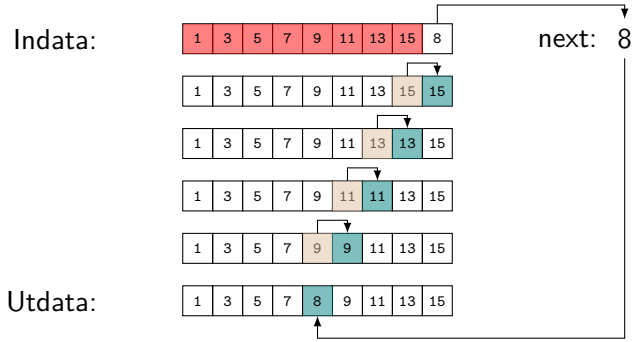
Inplace Insertion sort av fält

- Jobba i fältet
- Betrakta fältet som två delar:
  - en sorterad del i början
  - en osorterad del i slutet
  - Den sorterade delen växer med ett element för varje varv av huvudalgoritmen
- Algoritmen i grova drag:
  - Börja med ett element (ett element är sorterat)
  - Ta det första osorterade elementet och sortera in på rätt plats bland de sorterade elementen

Insertion sort — exempel



Insertion sort — Sidospår: insättning



Insertion sort — algoritm

- Algoritm:

```
Algorithm insertion_sort(a: Array, n: Int)
  // i indicates first unsorted element in a

  for i ← 1 to n - 1 do

    // new value to insert in sorted part of a
    next ← a[i]

    // start with last sorted element
    j ← i - 1

    // as long as new element is smaller and
    // we're inside the array
    while j >= 0 and next < a[j] do

      // shift element right
      a[j + 1] ← a[j]

      // continue to the left
      j ← j - 1

    // insert new value in its sorted place
    a[j+1] ← next

  return a
```


Blank

Bubble sort — bubbelsortering

Bubble Sort

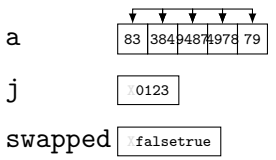
- Algoritmen i grova drag:
  - Upprepa följande tills ingen förändring sker:
    - Jämför alla elementen ett par i taget
      - Börja med element 0 och 1, därefter 1 och 2, osv
    - Om elementen är i fel ordning, byt plats på dem

Bubble Sort — algoritm

- Algoritm:

```
Algorithm bubble_sort(a: Array, n: Int)
  do
    // so far no swap has taken place
    swapped ← false

    // for each adjacent pair in a...
    for j ← 0 to n - 2 do

      // if the elements are in the wrong order...
      if a[j] > a[j + 1] then

        // ...swap the elements
        tmp ← a[j]
        a[j] ← a[j + 1]
        a[j + 1] ← tmp

        // remember that a swap has taken place
        swapped ← true

  while swapped = true

  return a
```


Bubble Sort exempel



Bubble Sort — stabilitet, komplexitet

- Algoritm:

```
Algorithm bubble_sort(a: Array, n: Int)
  do
    // so far no swap has taken place
    swapped ← false

    // for each adjacent pair in a...
    for j ← 0 to n - 2 do

      // if the elements are in the wrong order...
      if a[j] > a[j + 1] then

        // ...swap the elements
        tmp ← a[j]
        a[j] ← a[j + 1]
        a[j + 1] ← tmp

        // remember that a swap has taken place
        swapped ← true

  while swapped = true

  return a
```

- Stabil sortering?
  - Ja!
- Tidskomplexitet för sortering av fält?
  - Bästa-falls?
    - $O(n)$ (inga utbyten)
  - Värsta-falls?
    - $O(n^2)$
- Går att få samma komplexitet för sortering av lista då vi alltid refererar till ett element och dess efterföljare

Divide-and-Conquer

- Rekursiv algoritmprincip:
  - Grundidén är att dela upp problemet i mindre och mindre problem
  - Lös problemen för basfallet
  - Slå ihop till en totallösning
- *Mergesort* och *Quicksort* är av denna typ
- Komplexitet: $O(n \log n)$

Blank

Blank

Merge sort — samsortering

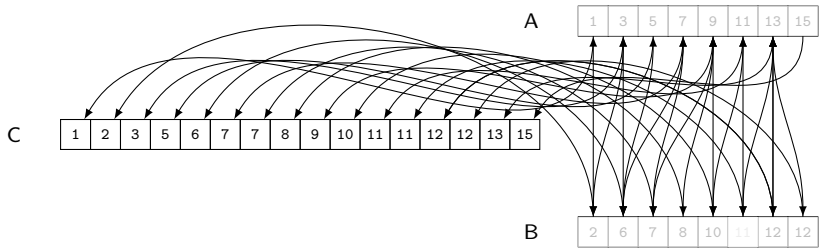
Merge Sort

- Algoritmen i grova drag
  - Om sekvensen har ett element
    - Returnera sekvensen (den är redan sorterad)
  - annars
    - Dela sekvensen i två ungefär lika stora delsekvenser
    - Sortera delsekvenserna rekursivt
    - Slå samman delsekvenserna (*Merge*)
    - Returnera den sammanslagna sekvensen

Merge

- *Merge Sort* använder en delalgoritm — *Merge*
- Algoritm för att slå samman två redan sorterade sekvenser:
  - Så länge bägge sekvenserna har element:
    - Jämför första (=minsta) elementet i vardera sekvensen
    - Flytta det minsta av de två elementen till utsekvensen
  - Flytta över alla element som finns kvar i sekvenserna

Merge — exempel



Merge

- Algoritm för *Merge*:

```
Algorithm merge(A, B: Array, na, nb: Int)
    C ← create_array(na + nb)

    ia ← 0 // Where to read from in A
    ib ← 0 // Where to read from in B
    ic ← 0 // Where to write to in C

    // While there are elements in both A and B...
    while ia < na and ib < nb do
        if A[ia] <= B[ib] then    // Smallest in A...
            C[ic] ← A[ia]         // ...copy from A
            ia ← ia + 1           // ...advance in A
        else                      // Smallest in B...
            C[ic] ← B[ib]         // ...copy from B
            ib ← ib + 1           // ...advance in B

        ic ← ic + 1               // Advance in C

    // While there are elements in A...
    while ia < na do
        C[ic] ← A[ia]             // ...copy from A
        ia ← ia + 1               // ...advance in A and C
        ic ← ic + 1

    // While there are elements in B...
    while ib < nb do
        C[ic] ← B[ib]             // ...copy from B
        ib ← ib + 1               // ...advance in B and C
        ic ← ic + 1

    return C
```

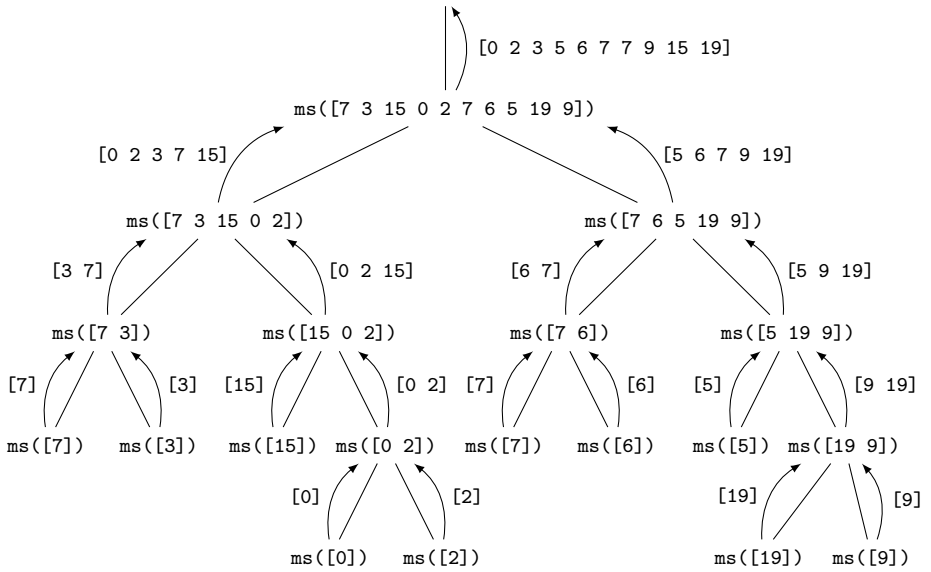

Merge Sort — algoritm

- Algoritm för *Merge Sort*:

```
Algorithm merge_sort(a: Array, n: Int)
  if n < 2 then
    // Already sorted
    return a

  // Split a in two parts
  (left, right) ← split(a, n/2)

  // Lengths of left and right parts, respectively
  nl ← floor(n/2)
  nr ← n - nl

  // Sort left half recursively
  left ← merge_sort(left, nl)

  // Sort right half recursively
  right ← merge_sort(right, nr)

  // Merge sorted arrays
  a ← merge(left, right, nl, nr)

  return a
```


1

Merge sort, anropsträd



Merge sort, stabil?

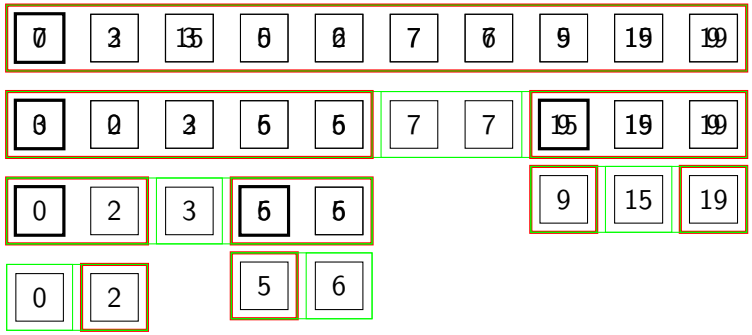
- Är *Merge sort* stabil?
  - Ja, om *Merge sort* lägger den första halvan i `left`
    - Lika element i `A` sorteras före `B` av *Merge*

Quicksort

Quicksort

- Algoritm:
  - Välj ut ett pivåelement
  - Dela upp listan i tre delar: Less, Equal, Greater
  - Sortera Less och Greater rekursivt
  - Slå ihop Less+Equal+Greater

Quicksort — Exempel

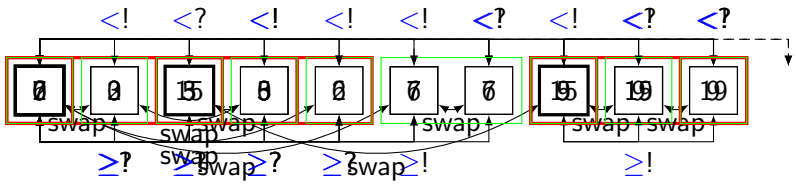


Inplace Quicksort

- Algoritm:
  - Välj ett pivåelement (PE)
  - Traversera parallellt från båda hållen i S:
    - Gå framåt från början av S tills man hittar ett element som är $\geq$ PE
    - Gå bakåt från slutet av S tills man hittar ett element som är $<$ PE
    - Byt plats på dessa två element.

    Upprepa till traverseringarna möts.
  - Stoppa in PE på rätt plats.
  - Rekursivt anrop för Less och Greater.

Inplace Quicksort — exempel



- Stabil?
  - Nej!

Quicksort — val av pivåelement

- Komplexiteten är $O(n \log n)$ i bästa fallet
- Valet av pivåelement är kritiskt:
  - Vill ha ett pivåelement som har ett mitten-värde
  - Vid sned fördelning får man i praktiken insticks-/urvalssortering med $O(n^2)$
- Alternativ för att få en enkel tilldelning:
  - Välj första/sista, slumpmässigt
  - Medel/median mellan några stycken
    - Median mellan första/mitten/sista
  - Det största av de två första som skiljer sig åt

Hur snabbt kan man sortera?

- Det snabbaste vi kan sortera med jämförelsebaserad algoritmer är $O(n \log n)$
- Den enda information vi använder är resultatet av en jämförelse av två nyckelvärden
  - $a < b$?
- Nyckelsortering använder mer information om nyckeltypen
  - Kan göras snabbare än $O(n \log n)$

Hur sortera snabbare än $O(n \log n)$?

- Vi måste veta mer om nycklarna vi vill sortera efter:
  - Det måste gå att avbilda nycklarna på heltal V
  - Vi behöver känna till ett minsta och största värde
    - V_{min} .. V_{max} för V
- Komplexiteten blir $O(n + m)$ där n är antalet element och m är $V_{\text{max}} - V_{\text{min}}$

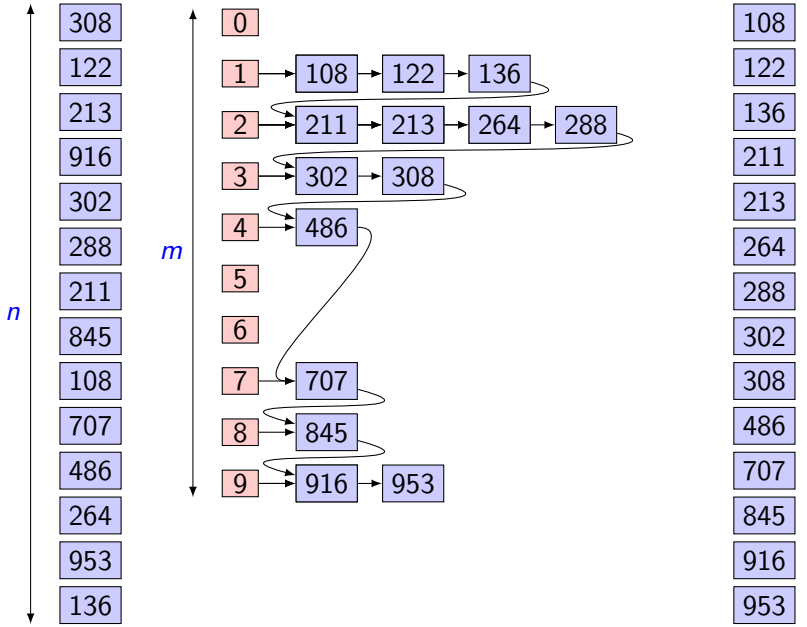
Facksortering (*Bucket sort*)

- Notera! På Wikipedia kallas denna algoritm *Pidgeonhole sort*

1. Skapa ett fack för varje nyckelvärde i intervallet $V_{\min}$ .. $V_{\max}$
2. Gå igenom sekvensen S
   - Lägg elementen i det fack dess nyckelvärde motsvarar
   - Komplexitet $O(n)$
3. Länka samman facken till en sekvens i ordning $V_{\min}$ .. $V_{\max}$
   - Komplexitet $O(m)$

- Kan göras stabil

Facksortering, nyckelvärdet är $v/100$



Sammanfattning

Algoritm	Tidskomplexitet bästa	värsta	Stabil?	Minnesbehov
Insertion Sort	$O(n^2)$	$O(n^2)$	Ja	$O(1)$
Selection Sort	$O(n)$	$O(n^2)$	Nej[1]	$O(1)$
Bubble Sort	$O(n)$	$O(n^2)$	Ja	$O(1)$
Merge Sort	$O(n \log n)$	$O(n \log n)$	Ja	$O(n)$
Quicksort	$O(n \log n)$	$O(n^2)$	Nej	$O(\log n)$[2]
Heapsort	$O(n \log n)$	$O(n \log n)$	Nej	$O(1)$
Bucket sort	$O(n + m)$	$O(n + m)$	Ja	$O(m)$

- Fundera på:
  - Hur hanterar algoritmen en redan sorterad lista?
  - Hur hanterar algoritmen en motsatt sorterad lista?

[1] Ja om $O(n)$ extra minne.
[2] $O(n)$ i värsta-fallet

Länkar

- Wikipedia — Sorting algorithms
  - `https://en.wikipedia.org/wiki/Sorting_algorithm`
- Animering av sorteringsalgorithmer
  - `http://www.sorting-algorithms.com`
- Dansat *merge sort*
  - `https://www.youtube.com/watch?v=XaqR3G_NVoo`

Sökning

Vad betyder "lika med"?

- Vid sökning och sortering definierar man ofta en extern likhetsfunktion (*match function*) som avgör om två elementvärden *a* och *b* är lika
- Det går att låta funktionen ta två argument och returnera `True` of argumenten anses lika, annars `False`
- Om man i stället definierar en jämförelsefunktion (*compare function*) och begär att den ska returnera ett heltal

<0	om *a* kommer före *b* i sorteringsordningen
0	om *a* och *b* är lika enligt sorteringsordningen
>0	om *a* kommer efter *b* i sorteringsordningen

  så blir algoritmerna ännu flexiblare på sorterade data
- En jämförelsefunktion gör det möjligt att använda samma söknings- och sorteringsalgoritmer på olika data

Jämförelsefunktioner, exempel

```
Algorithm Compare-int(a, b: Int)
// Input: Two integers a and b.
// Output: An integer
//         < 0 if and only if a < b
//         = 0 if and only if a = b, and
//         > 0 if and only if a > b
return a - b
```

```
Algorithm Compare-strings(a, b: String)
// Input: Two strings a and b.
// Output: An integer
//         < 0 if and only if a comes before b
//         = 0 if and only if a is equal to b
//         > 0 if and only if a comes after b
if Isless-than(a,b) then
  return -1
else if Isequal(a,b) then
  return 0
else
  return +1
```


Jämförelsefunktioner, exempel

```
Algorithm Compare-record(a, b: Record)
// Input: Two records a and b with fields lastname, firstname,
//        and age
// Output: An integer <0, =0, or >0 to indicate whether a is
//         considered to be before, equal to, or after b,
//         respectively. The ordering is decided by last name,
//         then first name.  In case of a tie, a younger age is
//         given precedence.
cmp ← Compare-strings(a.lastname, b.lastname)
if cmp = 0 then
  cmp ← Compare-strings(a.firstname, b.firstname)
if cmp = 0 then
  cmp ← Compare-int(a.age, b.age)
return cmp
```

Linjär sökning

- Starta från början och sök tills elementet hittats eller sekvensen är slut
- Komplexitet:
  - Om elementet finns: I medel, gå igenom halva listan $O(n)$
  - Om elementet saknas: Gå igenom hela listan $O(n)$
- Om listan är sorterad:
  - Om elementet finns: I medel, gå igenom halva listan $O(n)$
  - Om elementet saknas: I medel, gå igenom halva lista $O(n)$

Algoritm linjär sökning, osorterad lista (jämför `seek`)

```
Algorithm Linsearch(l: List, v: Value, Value-isequal: Function)
// Input: An unsorted list, a search value, and a equality
//        function. The Value-isequal function should accept
//        two element values and return True if the values
//        are considered equal.
// Output: (True, pos), where pos is the position for the first
//         match, or (False, None) if no match is found.
p ← First(l)
while not Isend(p, l)) do
  if Value-isequal(v, Inspect(p, l)) then
    return (True, p)
  p ← Next(p, l)
return (False, None)
```


Algoritm linjär sökning, sorterad lista

```
Algorithm Linsearch-sorted(l: List, v: Value, Compare: Function)
// Input: A sorted list, a search value, and a compare function.
// Output: (True, pos), where pos is the position for the first
//         match, or (False, None) if no match is found.
p ← First(l)
while not Isend(l) do
  // Compare
  c ← Compare(v, Inspect(p, l))
  if c = 0 then
    // Found it
    return (True, p)
  else if c > 0 then
    // Past where it could be, give up
    return (False, None)
  else
    // Still before where it could be, continue
    p ← Next(p, l)
return (False, None)
```


Binär sökning

- Om sekvensen har index (t.ex. i ett fält) kan man söka binärt
- Successiv halvering av sökintervallet
- Sök efter elementet med värde v:
  - Jämför med elementet $x[m]$ närmast mitten av intervallet
    - Om likhet — klart!
    - Om v kommer före $x[m]$ i sorteringsordningen, fortsätt sökningen rekursivt i delintervallet till vänster om m
    - Om v kommer efter $x[m]$ i sorteringsordningen, fortsätt sökningen rekursivt i delintervallet till höger om m
- Vi får värsta-falls och medelkomplexitet $O(\log n)$

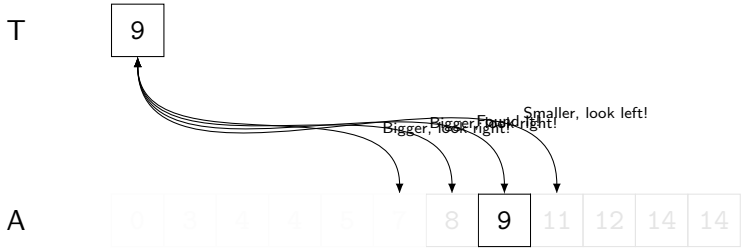
Algoritm iterativ binär sökning

```
Algorithm Binsearch-iter(a: Array, v: Value, Compare: Function)
left ← Low(a)
right ← High(a)
while left <= right do
  // Check in the middle
  mid ← (left + right)/2
  // Compare
  c ← Compare(v, Inspect-value(a, mid))
  if c = 0 then
    // Found it
    return (True, mid)
  else if c < 0 then
    // Look left
    right ← mid - 1
  else
    // Look right
    left ← mid + 1
return (False, None)
```


Algoritm rekursiv binär sökning

```
Algorithm Binsearch-rec(a: Array, v: Value, Compare: Function,
                        left, right: Index)
if right < left then
  return (False, None)
// Check in the middle
mid ← (left + right)/2
// Compare
c ← Compare(v, Inspect-value(a, mid))
if c = 0 then
  // Found it
  return (True, mid)
else if c < 0 then
  // Look left
  return Binsearch-rec(a, v, Compare, left, mid-1)
else
  // Look right
  return Binsearch-rec(a, v, Compare, mid+1, right)

Algorithm binsearch-main(a: Array, v: Value, Compare: Function)
// Call the recursive function to do the work.
return Binsearch-rec(a, v, Compare, Low(a), High(a))
```


Binär sökning, exempel



Exempel, sorterat fält

- Sök efter elementvärdet 13:
  - Linjär sökning: 8 jämförelser innan träff.

1	2	4	4	6	7	9	13	14	19
?	?	?	?	?	?	?	!		

  - Binär sökning: 2 jämförelser innan träff.

1	2	4	4	6	7	9	13	14	19
[				?					]
					[		!		]

- Sök efter elementvärdet 10:
  - Linjär sökning: 8 jämförelser innan man ger upp.

1	2	4	4	6	7	9	13	14	19
?	?	?	?	?	?	?	?		

  - Binär sökning: 4 jämförelser innan man ger upp.

1	2	4	4	6	7	9	13	14	19
[				?					]
					[		?		]
					[?	]			
						[?]			