# EDAA45 Programmering, grundkurs

## Läsvecka 11: Sökning, sortering

Björn Regnell

Datavetenskap, LTH

Lp1-2, HT 2016

# 11 Sökning, sortering

- Veckans labb: `survey2`
- Repetition: Vad är en algoritm?
- Jämföra strängar
- Jämförelse Scala och Java
- Linjärsökning
- Binärsökning
- Sortering

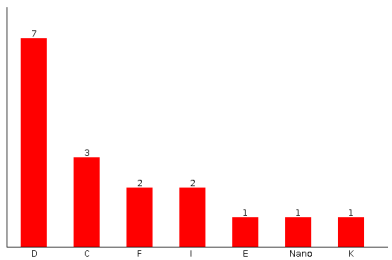
# Veckans labb: survey2

# Veckans labb: survey2

Nya version 2 av labben är enklare och kräver ej att du implementerar parsning av `args`. Välj själv vilken du gör.

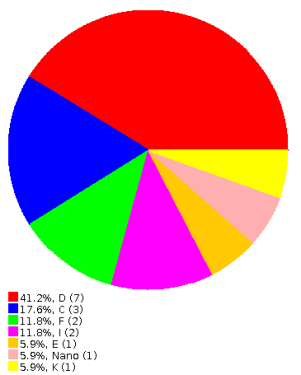
**Förberedelse:**

- Studera givna koden: workspace/w10_survey2
- Fyll i denna enkät: `https://goo.gl/forms/hC6JK2UQXVpbGECc2`




**Grunduppgift:**

- Implementera en klass `Table` för hantering av strängmatriser med rubrikrad från kolumnsepararade textfiler.
- Öva på att kombinera matriser, sortering, registrering för att räkna statistik.
- Använda inbyggda sorteringsfunktioner: `sortBy` och `sortWith`
- Implementera din egen sortering ”från scratch”.




**Extrauppgift:**

- Implementera stapeldiagram.

# Repetition: Vad är en algoritm?

# Repetition: Vad är en algoritm?

En algoritm är en stegvis beskrivning av hur man löser ett problem.
Exempel: SWAP, MIN, Registrering, Sökning, Sortering

# Repetition: Vad är en algoritm?

En algoritm är en stegvis beskrivning av hur man löser ett problem.
Exempel: SWAP, MIN, Registrering, Sökning, Sortering

Problemlösningsprocessens olika steg (inte nödvändigtvis i denna ordning):

- Dela upp problemet i enklare delproblem och sätt samman.
- Finns redan färdig lösning på (del)problem?
- Formulera (del)**problemet** och ange tydligt indata och utdata:
  exempel MIN: indata: sekvens av heltal; utdata: minsta talet
- Kom på en **lösningsidé**: (kan vara mycket klurigt och svårt)
  exempel MIN: iterera över talen och håll reda på ”minst hittills”
- Formulera en **stegvis beskrivning** som löser problemet:
  exempel: pseudo-kod med sekvens av instruktioner
- Implementera en **körbar lösning** i ”riktig” kod:
  exempel: en Scala-metod i en klass eller i ett singelobjekt
- Har algoritmen acceptabel komplexitet i förhållande till tids- och minneskrav?

# Repetition: Vad är en algoritm?

En algoritm är en stegvis beskrivning av hur man löser ett problem.
Exempel: SWAP, MIN, Registrering, Sökning, Sortering

Problemlösningsprocessens olika steg (inte nödvändigtvis i denna ordning):

- Dela upp problemet i enklare delproblem och sätt samman.
- Finns redan färdig lösning på (del)problem?
- Formulera (del)**problemet** och ange tydligt indata och utdata:
  exempel MIN: indata: sekvens av heltal; utdata: minsta talet
- Kom på en **lösningsidé**: (kan vara mycket klurigt och svårt)
  exempel MIN: iterera över talen och håll reda på ”minst hittills”
- Formulera en **stegvis beskrivning** som löser problemet:
  exempel: pseudo-kod med sekvens av instruktioner
- Implementera en **körbar lösning** i ”riktig” kod:
  exempel: en Scala-metod i en klass eller i ett singelobjekt
- Har algoritmen acceptabel komplexitet i förhållande till tids- och minneskrav?

Det krävs ofta **kreativitiet** i stegen ovan – även i att **känna igen** problemet!
Simpelt exempel: Du stöter på problemet MAX och ser likheten med MIN.

# Repetition: Vad är en algoritm?

En algoritm är en stegvis beskrivning av hur man löser ett problem.
Exempel: SWAP, MIN, Registrering, Sökning, Sortering

Problemlösningsprocessens olika steg (inte nödvändigtvis i denna ordning):

- Dela upp problemet i enklare delproblem och sätt samman.
- Finns redan färdig lösning på (del)problem?
- Formulera (del)**problemet** och ange tydligt indata och utdata:
  exempel MIN: indata: sekvens av heltal; utdata: minsta talet
- Kom på en **lösningsidé**: (kan vara mycket klurigt och svårt)
  exempel MIN: iterera över talen och håll reda på ”minst hittills”
- Formulera en **stegvis beskrivning** som löser problemet:
  exempel: pseudo-kod med sekvens av instruktioner
- Implementera en **körbar lösning** i ”riktig” kod:
  exempel: en Scala-metod i en klass eller i ett singelobjekt
- Har algoritmen acceptabel komplexitet i förhållande till tids- och minneskrav?

Det krävs ofta **kreativitiet** i stegen ovan – även i att **känna igen** problemet!
Simpelt exempel: Du stöter på problemet MAX och ser likheten med MIN.

**Övning**: Diskutera hur du löser detta problem i relation till stegen ovan:

*Att räkna antalet förekomster av olika unika ord i en textsträng.*

# Jämföra strängar

# Att jämföra strängar lexikografiskt

Teckenstandard UTF-16: Alla stora bokstäver är ”mindre” än alla små:

```
scala> Array("hej","Hej","gurka").sorted
```

# Att jämföra strängar lexikografiskt

Teckenstandard UTF-16: Alla stora bokstäver är ”mindre” än alla små:

```
scala> Array("hej","Hej","gurka").sorted
```

```
res0: Array[String] = Array(Hej, gurka, hej)
```

# Att jämföra strängar lexikografiskt

Teckenstandard UTF-16: Alla stora bokstäver är ”mindre” än alla små:

```
scala> Array("hej","Hej","gurka").sorted
```

```
res0: Array[String] = Array(Hej, gurka, hej)
```

- Antag att vi vill lösa detta problem ”från scratch”:
  **att sortera en sekvens med strängar**
- För att göra detta behöver vi lösa dessa delproblemen:
  - **att jämföra strängar**
  - **sökning i sekvenser**
  - **SWAP** (om på-plats-sortering i förändringsbar sekvens)
- Vad betyder det att två strängar är ”lika”?
- Vad betyder det att en sträng är ”mindre” än en annan?

# Att jämföra strängar lexikografiskt

Teckenstandard UTF-16: Alla stora bokstäver är ”mindre” än alla små:

```
scala> Array("hej","Hej","gurka").sorted
```

```
res0: Array[String] = Array(Hej, gurka, hej)
```

- Antag att vi vill lösa detta problem ”från scratch”:
  **att sortera en sekvens med strängar**
- För att göra detta behöver vi lösa dessa delproblemen:
  - **att jämföra strängar**
  - **sökning i sekvenser**
  - **SWAP** (om på-plats-sortering i förändringsbar sekvens)
- Vad betyder det att två strängar är ”lika”?
- Vad betyder det att en sträng är ”mindre” än en annan?

Vi använder här strängjämförelse, sökning och sortering för att illustrera typiska **imperativa algoritmer**. **Normalt** använder man **färdiga lösningar** på dessa problem!

# Jämföra strängar: likhet

Antag att vi inte kan göra `s1 == s2` utan bara kan jämföra strängar tecken för tecken, t.ex. så här: `s1(i) == s2(i)`. Antag också att vi inte har tillgång till annat än metoderna `length` och `apply` på strängar, samt **`while`** och variabler av grundtyp. **Lös problemet att *avgöra om två strängar är lika*.**

# Jämföra strängar: likhet

Antag att vi inte kan göra `s1 == s2` utan bara kan jämföra strängar tecken för tecken, t.ex. så här: `s1(i) == s2(i)`. Antag också att vi inte har tillgång till annat än metoderna `length` och `apply` på strängar, samt **`while`** och variabler av grundtyp. **Lös problemet att *avgöra om två strängar är lika*.**

- Indata: två strängar
- Utdata: **`true`** om lika annars **`false`**

1. Klura ut din lösningsidé
2. Formulera algoritmen i pseudokod
3. Implementera algoritmen i Scala:
   ```
   def isEqual(s1: String, s2: String): Boolean = ???
   ```

# Algoritmexempel: stränglikhet, pseudokod

```
def isEqual(s1: String, s2: String): Boolean = {
  if (/* lika längder */) {
    var foundDiff = false
    var i = /* första index */
    while (!foundDiff && /* i inom indexgräns */) {
      if (/* tecken på plats i är olika */) foundDiff = true
      else i = /* nästa index */
    }
    !foundDiff
  } else false
}
```

# Algoritmexempel: stränglikhet, pseudokod

```
def isEqual(s1: String, s2: String): Boolean = {
  if (/* lika längder */) {
    var foundDiff = false
    var i = /* första index */
    while (!foundDiff && /* i inom indexgräns */) {
      if (/* tecken på plats i är olika */) foundDiff = true
      else i = /* nästa index */
    }
    !foundDiff
  } else false
}
```

Detta är en variant av s.k. **linjärsökning** där vi söker från början i en sekvens till vi hittar det vi söker efter (här söker vi efter tecken som skiljer sig åt).

# Algoritmexempel: stränglikhet, pseudokod

```
def isEqual(s1: String, s2: String): Boolean = {
  if (/* lika längder */) {
    var foundDiff = false
    var i = /* första index */
    while (!foundDiff && /* i inom indexgräns */) {
      if (/* tecken på plats i är olika */) foundDiff = true
      else i = /* nästa index */
    }
    !foundDiff
  } else false
}
```

Detta är en variant av s.k. **linjärsökning** där vi söker från början i en sekvens till vi hittar det vi söker efter (här söker vi efter tecken som skiljer sig åt).

Hur ser implementationen i exekverbar Scala ut?

# Algoritmexempel: stränglikhet, implementation

```
def isEqual(s1: String, s2: String): Boolean = {
  if (s1.length == s2.length) {
    var foundDiff = false
    var i = 0
    while (!foundDiff && i < s1.length) {
      if (s1(i) != s2(i)) foundDiff = true
      else i += 1
    }
    !foundDiff
  } else false
}
```

# Algoritmexempel: stränglikhet, implementation

```
def isEqual(s1: String, s2: String): Boolean = {
  if (s1.length == s2.length) {
    var foundDiff = false
    var i = 0
    while (!foundDiff && i < s1.length) {
      if (s1(i) != s2(i)) foundDiff = true
      else i += 1
    }
    !foundDiff
  } else false
}
```

Fördjupning: jämför ovan med implementationen av `String.equals` här:
hg.openjdk.java.net/jdk8u/jdk8u60/jdk/file/935758609767/src/share/classes/java
och använd `timed` nedan och jämför prestanda med `isEqual` ovan.

Obs! Mät många gånger så att JVM:en optimerar din kod.

```
def timed[T](block: => T): (Double, T) = {
  val (t, res) = (System.nanoTime, block)
  ((System.nanoTime - t) / 1e9, res)
}
```

# Algoritmexempel: stränglikhet, prestanda

```
1  scala> val enMiljon = 1000000
2
3  scala> val s = Array.fill(enMiljon)('x').mkString
4
5  scala> val t = s.updated(enMiljon - 1, 'y')
6
7  scala> timed { s == t }
8  res42: (Double, Boolean) = (3.76459E-4,false)
9
10 scala> timed { isEqual(s,t) }
11 res43: (Double, Boolean) = (3.31597E-4,false)
```

Ovan är kört efter ”uppvärmning” på i7-4790K CPU @ 4.00GHz
Skillnaden inom mätfelmarginalen!

# Jämföra strängar: ”mindre än”

Med `s1 < s2` menar vi att strängen s1 ska sorteras före strängen s2 enligt hur de enskilda tecknen är ordnade med uttrycket `s1(i) < s2(i)`.
Antag också att vi inte har tillgång till annat än metoderna `length` och `apply` på strängar, samt `while` och variabler av grundtyp, samt `math.min`
**Lös problemet att *avgöra om en sträng är ”mindre” än en annan*.**

- Indata: två strängar, s1, s2
- Utdata: `true` om s1 ska sorteras före s2 annars `false`

1. Klura ut din lösningsidé
2. Formulera algoritmen i pseudokod
3. Implementera algoritmen i Scala:
   ```
   def isLessThan(s1: String, s2: String): Boolean = ???
   ```

# Jämföra strängar: ”mindre än”

Pseudokod:

```
def isLessThan(s1: String, s2: String): Boolean = {

  val minLength = /* minimum av längderna på s1 och s2 */

  def firstDiff(s1: String, s2: String): Int =
    /* index för första skillnaden (om de börjar lika: minLength) */

  val diffIndex = firstDiff(s1, s2)
  if (diffIndex == minLength) /* s1 är kortare än s2 */
  else /* tecknet s1(diffIndex) är mindre än tecknet s2(diffIndex) */
}
```

# Jämföra strängar: ”mindre än”

```
def isLessThan(s1: String, s2: String): Boolean = {

  val minLength = math.min(s1.length, s2.length)

  def firstDiff(s1: String, s2: String): Int = {
    var foundDiff = false
    var i = 0
    while (!foundDiff && i < minLength)
      if (s1(i) != s2(i)) foundDiff = true
      else i += 1
    i
  }

  val diffIndex = firstDiff(s1, s2)
  if (diffIndex == minLength) s1.length < s2.length
  else s1(diffIndex) < s2(diffIndex)
}
```

# Jämföra strängar i Java

- I Java kan man **inte** jämföra strängar med operatorerna <, <=, >, och >=
- Dessutom ger operatorerna == och != *inte* innehålls(o)likhet utan **referens(o)likhet** :(
- Istället **måste** man i Java använda metoderna `equals` och `compareTo`
  Dessa fungerar även i Scala eftersom strängar i Scala och Java är av samma typ, nämligen `java.lang.String`.

# Jämföra strängar i Java

- I Java kan man **inte** jämföra strängar med operatorerna <, <=, >, och >=
- Dessutom ger operatorerna == och != *inte* innehålls(o)likhet utan **referens(o)likhet** :(
- Istället **måste** man i Java använda metoderna `equals` och `compareTo` Dessa fungerar även i Scala eftersom strängar i Scala och Java är av samma typ, nämligen `java.lang.String`.
- `s1.compareTo(s2)` ger ett heltal som är:
  - `0` om s1 och s2 har samma innehåll
  - **negativt** om s1 < s2 i lexikografisk mening, alltså s1 ska sorteras **före**
  - **positivt** om s1 > s2 i lexikografisk mening, alltså s1 ska sorteras **efter**

# Jämföra strängar i Java

- I Java kan man **inte** jämföra strängar med operatorerna <, <=, >, och >=
- Dessutom ger operatorerna == och != *inte* innehålls(o)likhet utan **referens(o)likhet** :(
- Istället **måste** man i Java använda metoderna `equals` och `compareTo`
  Dessa fungerar även i Scala eftersom strängar i Scala och Java är av samma typ, nämligen `java.lang.String`.
- `s1.compareTo(s2)` ger ett heltal som är:
  - `0` om s1 och s2 har samma innehåll
  - **negativt** om s1 < s2 i lexikografisk mening, alltså s1 ska sorteras **före**
  - **positivt** om s1 > s2 i lexikografisk mening, alltså s1 ska sorteras **efter**
- Undersök följande:

```
1  scala> new String("hej") eq new String("hej") // motsvarar == i Java
2  scala> "hej".equals("hej")                    // samma som == i Scala
3  scala> "hej".compareTo("hej")
4  scala> "hej".compareTo("HEJ")         // alla stora är 'före' alla små
5  scala> "HEJ".compareTo("hej")
```

docs.oracle.com/javase/8/docs/api/java/lang/String.html#compareTo-java.lang.String-

# Jämföra strängar i Java: exempel

Vad skriver detta Java-program ut?

```
public class StringEqTest {
    public static void main(String[] args){
        boolean eqTest1 =
          (new String("hej")) == (new String("hej"));
        boolean eqTest2 =
          (new String("hej")).equals(new String("hej"));
        int eqTest3 =
          (new String("hej")).compareTo(new String("hej"));
        System.out.println(eqTest1);
        System.out.println(eqTest2);
        System.out.println(eqTest3);
    }
}
```

# Jämföra strängar i Java: exempel

Vad skriver detta Java-program ut?

```
public class StringEqTest {
    public static void main(String[] args){
        boolean eqTest1 =
          (new String("hej")) == (new String("hej"));
        boolean eqTest2 =
          (new String("hej")).equals(new String("hej"));
        int eqTest3 =
          (new String("hej")).compareTo(new String("hej"));
        System.out.println(eqTest1);
        System.out.println(eqTest2);
        System.out.println(eqTest3);
    }
}
```

```
1 $ javac StringEqTest.java
2 $ java StringEqTest
3 false
4 true
5 0
```

# Jämförelse Scala och Java

# Grundläggande likheter och skillnader

- **Sökning** återkommer i många skepnader:
  i en datastruktur, vilken det än må vara, vill man ofta kunna
  **hitta ett element med en viss egenskap**.

# Grundläggande likheter och skillnader

- **Sökning** återkommer i många skepnader:
  i en datastruktur, vilken det än må vara, vill man ofta kunna
  **hitta ett element med en viss egenskap**.
  Några färdiga linjärsökningar i Scalas standardbibliotek:

```
1  scala> Vector("gurka","tomat","broccoli").indexOf("tomat")
2  res0: Int = 1
3
4  scala> Vector("gurka","tomat","broccoli").indexWhere(_.contains("o"))
5  res1: Int = 1
6
7  scala> Vector("gurka","tomat","broccoli").find(_.contains("o"))
8  res2: Option[String] = Some(tomat)
```

# Grundläggande likheter och skillnader

- **Sökning** återkommer i många skepnader:
  i en datastruktur, vilken det än må vara, vill man ofta kunna
  **hitta ett element med en viss egenskap**.
  Några färdiga linjärsökningar i Scalas standardbibliotek:

```
1  scala> Vector("gurka","tomat","broccoli").indexOf("tomat")
2  res0: Int = 1
3
4  scala> Vector("gurka","tomat","broccoli").indexWhere(_.contains("o"))
5  res1: Int = 1
6
7  scala> Vector("gurka","tomat","broccoli").find(_.contains("o"))
8  res2: Option[String] = Some(tomat)
```

- Sökning efter ett visst index i en sekvens:
  - Indata: en sekvens och ett **sökkriterium**
  - Utdata: index för första eftersökta element, annars -1

# Grundläggande likheter och skillnader

- **Sökning** återkommer i många skepnader:
  i en datastruktur, vilken det än må vara, vill man ofta kunna
  **hitta ett element med en viss egenskap**.
  Några färdiga linjärsökningar i Scalas standardbibliotek:

```
1  scala> Vector("gurka","tomat","broccoli").indexOf("tomat")
2  res0: Int = 1
3
4  scala> Vector("gurka","tomat","broccoli").indexWhere(_.contains("o"))
5  res1: Int = 1
6
7  scala> Vector("gurka","tomat","broccoli").find(_.contains("o"))
8  res2: Option[String] = Some(tomat)
```

- Sökning efter ett visst index i en sekvens:
  - Indata: en sekvens och ett **sökkriterium**
  - Utdata: index för första eftersökta element, annars -1
- Två typiska varianter av sökning i en sekvens:
  - Linjärsökning: börja från början och sök tills ett eftersökt element är funnet
  - Binärsökning: antag sorterad sekvensen; börja i mitten, välj rätt halva ...

# Linjärsökning

# Linjärsökning: hitta index för elementet 42

Skriv pseudokod för:

```
def indexOf42(xs: Vector[Int]): Int = ???
```

# Linjärsökning: hitta index för elementet 42

Skriv pseudokod för:

`def indexOf42(xs: Vector[Int]): Int = ???`

```
def indexOf42(xs: Vector[Int]): Int = {
  var i = /* index för första elementet */
  var found = false
  while (!found && /* index inom indexgräns */) {
    if (/* element på plats i är 42 */) found = true
    else i = /* nästa index */
  }
  if (/* hittat */) i else -1
}
```

# Linjärsökning: hitta index för elementet 42

Implementera:

```
def indexOf42(xs: Vector[Int]): Int = ???
```

# Linjärsökning: hitta index för elementet 42

Implementera:

```
def indexOf42(xs: Vector[Int]): Int = ???
```

```
def indexOf42(xs: Vector[Int]): Int = {
  var i = 0
  var found = false
  while (!found && i < xs.length) {
    if (xs(i) == 42) found = true
    else i += 1
  }
  if (found) i else -1
}
```

# Linjärsökning: hitta index för elementet x

Implementera:

```
def indexOf(xs: Vector[Int], x: Int): Int = ???
```

# Linjärsökning: hitta index för elementet x

Implementera:

```
def indexOf(xs: Vector[Int], x: Int): Int = ???
```

```
def indexOf(xs: Vector[Int], x: Int): Int = {
  var i = 0
  var found = false
  while (!found && i < xs.length) {
    if (xs(i) == x) found = true
    else i += 1
  }
  if (found) i else -1
}
```

# Linjärsökning: hitta index för elementet p(x)

Implementera:

```
def indexWhere(xs: Vector[Int], p: Int => Boolean): Int = ???
```

# Linjärsökning: hitta index för elementet p(x)

Implementera:

```
def indexWhere(xs: Vector[Int], p: Int => Boolean): Int = ???
```

```
def indexWhere(xs: Vector[Int], p: Int => Boolean): Int = {
  var i = 0
  var found = false
  while (!found && i < xs.length) {
    if (p(xs(i))) found = true
    else i += 1
  }
  if (found) i else -1
}
```

# Linjärsökning: generalisera till godtycklig typ

Implementera:

```
def indexWhere[T](xs: Vector[T], p: T => Boolean): Int = ???
```

# Linjärsökning: generalisera till godtycklig typ

Implementera:

```
def indexWhere[T](xs: Vector[T], p: T => Boolean): Int = ???
```

```
def indexWhere[T](xs: Vector[T], p: T => Boolean): Int = {
  var i = 0
  var found = false
  while (!found && i < xs.length) {
    if (p(xs(i))) found = true
    else i += 1
  }
  if (found) i else -1
}
```

# Linjärsökning: generalisera till godtycklig typ

Typinferensen fungerar bättre om stegad funktion:
`def indexWhere[T](xs: Vector[T])(p: T => Boolean): Int`

```
def indexWhere[T](xs: Vector[T])(p: T => Boolean): Int = {
  var i = 0
  var found = false
  while (!found && i < xs.length) {
    if (p(xs(i))) found = true
    else i += 1
  }
  if (found) i else -1
}
```

# Linjärsökning: generalisera till godtycklig sekvens

Implementera:

```
def indexWhere[T](xs: Seq[T])(p: T => Boolean): Int = ???
```

# Linjärsökning: generalisera till godtycklig sekvens

Implementera:

```
def indexWhere[T](xs: Seq[T])(p: T => Boolean): Int = ???
```

```
def indexWhere[T](xs: Seq[T])(p: T => Boolean): Int = {
  var i = 0
  var found = false
  while (!found && i < xs.length) {
    if (p(xs(i))) found = true
    else i += 1
  }
  if (found) i else -1
}
```

# Linjärsökning: generalisera till godtycklig sekvens

Implementera:

```
def find[T](xs: Seq[T])(p: T => Boolean): Option[T] = ???
```

# Linjärsökning: generalisera till godtycklig sekvens

Implementera:

```
def find[T](xs: Seq[T])(p: T => Boolean): Option[T] = ???
```

```
def find[T](xs: Seq[T])(p: T => Boolean): Option[T] = {
  var i = 0
  var found = false
  while (!found && i < xs.length) {
    if (p(xs(i))) found = true
    else i += 1
  }
  if (found) Some(xs(i)) else None
}
```

# Binärsökning

# Binärsökning: snabbare men kräver sorterad sekvens

```
scala> val enMiljon = 1000000

scala> val xs = Array.tabulate(enMiljon)(i => i + 1)   // sorterad

scala> xs(enMiljon - 1)
res0: Int = 1000000

scala> timed { xs.indexOf(enMiljon) }        // linjärsökning
res42: (Double, Int) = (0.016622994,999999)

scala> import scala.collection.Searching._  // ger tillgång till metoden search
import scala.collection.Searching._

scala> timed { xs.search(enMiljon) }        // binärsökning
res54: (Double, collection.Searching.SearchResult) = (2.45691E-4,Found(999999))
```

# Binärsökning: snabbare men kräver sorterad sekvens

```
1  scala> val enMiljon = 1000000
2
3  scala> val xs = Array.tabulate(enMiljon)(i => i + 1)   // sorterad
4
5  scala> xs(enMiljon - 1)
6  res0: Int = 1000000
7
8  scala> timed { xs.indexOf(enMiljon) }        // linjärsökning
9  res42: (Double, Int) = (0.016622994,999999)
10
11 scala> import scala.collection.Searching._  // ger tillgång till metoden search
12 import scala.collection.Searching._
13
14 scala> timed { xs.search(enMiljon) }        // binärsökning
15 res54: (Double, collection.Searching.SearchResult) = (2.45691E-4,Found(999999))
```

Citat från scaladoc för `search`:
”The sequence **should be sorted** before calling; otherwise, the results are **undefined.**”

# Binärsökning: lösningsidé

**Problemlösningsidé**: Om sekvensen är sorterad kan vi utnyttja detta för en mer effektiv sökning, genom att jämföra med mittersta värdet och se om det vi söker finns före eller efter detta, och upprepa med ”halverad” sekvens tills funnet.

# Binärsökning: lösningsidé

**Problemlösningsidé**: Om sekvensen är sorterad kan vi utnyttja detta för en mer effektiv sökning, genom att jämföra med mittersta värdet och se om det vi söker finns före eller efter detta, och upprepa med ”halverad” sekvens tills funnet.

- **Indata**: sorterad sekvens av heltal och det eftersökta elementet
- **Utdata**: `Found(index)` för det eftersökta elementet
  om saknas: `InsertionPoint(index)`

# Binärsökning: lösningsidé

**Problemlösningsidé**: Om sekvensen är sorterad kan vi utnyttja detta för en mer effektiv sökning, genom att jämföra med mittersta värdet och se om det vi söker finns före eller efter detta, och upprepa med ”halverad” sekvens tills funnet.

- **Indata**: sorterad sekvens av heltal och det eftersökta elementet
- **Utdata**: `Found(index)` för det eftersökta elementet
  om saknas: `InsertionPoint(index)`

```
sealed trait SearchResult {
  def insertionPoint: Int
}

case class Found(foundIndex: Int) extends SearchResult {
  override def insertionPoint = foundIndex
}

case class InsertionPoint(insertionPoint: Int) extends SearchResult
```

# Binärsökning: pseudokod, iterativ lösning

## **Pseudo-kod**: iterativ lösning

```
def binarySearch(xs: Vector[Int])(elem: Int): SearchResult = {
  var found = false
  var (low, high) = (/* lägsta index */, /* högsta index */)
  var mid = /* något startvärde */
  while (!found && /* finns fler element kvar */) {
    mid = /* mittpunkten i intervallet (low, high) */
    if (xs(mid) == elem) found = true
    else if (xs(mid) < elem) /* flytta intervallets undre gräns */
    else /* flytta intervallets övre gräns" */
  }
  if (found) Found(mid)
  else InsertionPoint(low)
}
```

# Binärsökning: implementation, iterativ lösning

## **Implementation**: iterativ lösning

```
def binarySearch(xs: Vector[Int])(elem: Int): SearchResult = {
  var found = false
  var (low, high) = (0, xs.length - 1)
  var mid = -1
  while (!found && low <= high) {
    mid = (low + high) / 2
    if (xs(mid) == elem) found = true
    else if (xs(mid) < elem) low = mid + 1
    else high = mid - 1
  }
  if (found) Found(mid)
  else InsertionPoint(low)
}
```

# Binärsökning: instrumentering av iterativ lösning

```
def waitForEnter: Unit = scala.io.StdIn.readLine("")
def show(msg: String): Unit = {println(msg); waitForEnter}

def binarySearch(xs: Vector[Int])(elem: Int): SearchResult = {
  var found = false                              ; show(s"found = $found")
  var (low, high) = (0, xs.length - 1)           ; show(s"(low, high) = ($low, $high)")
  var mid = -1                                   ; show(s"mid = $mid")
  while (!found && low <= high) {                ; show(s"while ${!found && low <= high}")
    mid = (low + high) / 2                       ; show(s"mid = $mid")
    if (xs(mid) == elem) {found = true           ; show(s"found = $found")}
    else if (xs(mid) < elem) {low = mid + 1      ; show(s"low = $low")}
    else {high = mid - 1                         ; show(s"high = $high")}
  }
  if (found) Found(mid)
  else InsertionPoint(low)
}
```

```
1  scala> binarySearch(Vector(0,1,2,3,42,5))(42)
2  found = false
3  (low, high) = (0, 5)
4  mid = -1
5  while true
6  mid = 2
7  low = 3
8  while true
9  mid = 4
10 found = true
11 res0: collection.Searching.SearchResult = Found(4)
```

# Binärsökning: rekursiv lösning

## **Fördjupning**: rekursiv lösning

```
def binarySearch(xs: Vector[Int])(elem: Int): SearchResult = {
  def loop(low: Int, high: Int): SearchResult =
    if (low > high) InsertionPoint(low)
    else (low + high) / 2 match {
      case mid if xs(mid) == elem => Found(mid)
      case mid if xs(mid) < elem  => loop(mid + 1, high)
      case mid                    => loop(low, mid - 1)
    }

  loop(0, xs.length - 1)
}
```

# Binärsökning: generisk rekursiv lösning

**Fördjupning**: iterativ generisk lösning med implicit ordning

```
def binarySearch[T](xs: Seq[T])(elem: T)(implicit ord: Ordering[T]): SearchResult = {
  import ord._
  def loop(low: Int, high: Int): SearchResult =
    if (low > high) InsertionPoint(low)
    else (low + high) / 2 match {
      case mid if xs(mid) == elem => Found(mid)
      case mid if xs(mid) < elem  => loop(mid + 1, high)
      case mid                    => loop(low, mid - 1)
    }

  loop(0, xs.length - 1)
}
```

För den intresserade:

Se fördjupningsuppgifter om implicita ordningar i veckans övning.

# Tidskomplexitet, sökning

**Fördjupning:**

Algoritmteoretisk analys av sökalgoritmerna ger:

- Linjärsökning: tiden är proportionell mot $n$, skrivs: $O(n)$
- Binärsökning: tiden är proportionell mot $\log_2 n$, skrivs: $O(\log n)$

# Tidskomplexitet, sökning

**Fördjupning:**

Algoritmteoretisk analys av sökalgoritmerna ger:

- Linjärsökning: tiden är proportionell mot $n$, skrivs: $O(n)$
- Binärsökning: tiden är proportionell mot $\log_2 n$, skrivs: $O(\log n)$

Empirisk analys: Vi har en vektor med 1000 element. Vi har mätt tiden för att söka upp ett element många gånger och funnit att det tar ungefär 1 $\mu$s både med linjärsökning och binärsökning.

Hur lång tid tar det om vi har fler element i vektorn?

| | 1000 | 10 000 | 100000 | 1 000 000 | 10 000 000 |
|---|---|---|---|---|---|
| linjärsökning | 1 | 10 | 100 | 1000 | 10 000 |
| binärsökning | 1 | 1.33 | 1.67 | 2.00 | 2.33 |

Kurserna:

”**Utvärdering av programvarusystem**”, obl. för D1, studerar detta **empiriskt**

”**Algoritmer, datastrukturer och komplexitet**”, obl. för D2, studerar detta **analytiskt**

# Sortering

# Sorteringsproblemet

**Problem**: Vi har en osorterad sekvens med heltal. Vi vill ordna denna osorterade sekvens i en sorterad sekvens från minst till störst.

# Sorteringsproblemet

**Problem**: Vi har en osorterad sekvens med heltal. Vi vill ordna denna osorterade sekvens i en sorterad sekvens från minst till störst.

En *generalisering* av problement:

Vi har många element av godtycklig typ och en **ordningsrelation** som säger vad vi menar med att ett element är *mindre än* eller *större än eller lika med* ett annat element.

Vi vill lösa problemet att ordna elementen i sekvens så att för varje element på plats $i$ så är efterföljande element på plats $i + 1$ större eller lika med elementet på plats $i$.

# Två enkla sporteringsalgoritmer: Insättningssortering & Urvalssortering

- Insättningssortering **lösningsidé**: Ta ett element i taget från den osorterade listan och **sätt in** det på **rätt plats** i den sorterade listan och upprepa till det inte finns fler osorterade element.

# Två enkla sporteringsalgoritmer: Insättningssortering & Urvalssortering

- Insättningssortering **lösningsidé**: Ta ett element i taget från den osorterade listan och **sätt in** det på **rätt plats** i den sorterade listan och upprepa till det inte finns fler osorterade element.
- Urvalsssortering **lösningsidé**: **Välj ut** det minsta kvarvarande elementet i den osorterade listan och placera det **sist** i den sorterade listan och upprepa till det inte finns fler osorterade element.

# Sortera till ny vektor med insättningssortering: pseudo-kod

Det är nog lättare att förstå **insertion sort** om man sorterar till en ny vektor. Vi ska sedan se hur man sorterar ”på plats” (eng. *in place*) i en array.

**Indata**: en osorterad vektor med heltal
**Utdata**: en ny, sorterad vektor med heltal

```
def insertionSort(xs: Vector[Int]): Vector[Int] = {
  val sorted = /* tom ArrayBuffer */
  for (/* alla element i xs */) {
     /* linjärsök rätt position i sorted */
     /* sätt in element på rätt plats i sorted */
  }
  sorted.toVector
}
```

# Sortera till ny vektor med insättningssortering: implementation i Scala

```
def insertionSort(xs: Vector[Int]): Vector[Int] = {
  val sorted = scala.collection.mutable.ArrayBuffer.empty[Int]
  for (elem <- xs) {
     // linjärsök rätt position i sorted:
     var pos = 0
     while (pos < sorted.length && sorted(pos) < elem) {
       pos += 1
     }
     // sätt in element på rätt plats i sorted:
     sorted.insert(pos, elem)
  }
  sorted.toVector
}
```

# Sortera till ny vektor med insättningssortering: implementation i Java med foreach-sats

```
import java.util.ArrayList;

public class JSort {
    public static ArrayList<Integer> insertionSort(ArrayList<Integer> xs) {
        ArrayList<Integer> sorted = new ArrayList<Integer>();
        for (int elem : xs) {
            // linjärsök rätt position i sorted:
            int pos = 0;
            while (pos < sorted.size() && sorted.get(pos) < elem) {
                pos++;
            }
            // sätt in element på rätt plats i sorted:
            sorted.add(pos, elem);
        }
        return sorted;
    }
}
```

stackoverflow.com/questions/85190/how-does-the-java-for-each-loop-work

Javasamlingar måste ”wrappa” primitiva **int** i klassen `Integer` (mer om detta senare)

# Sortera till ny vektor med urvalssortering: pseudo-kod

Det är nog lättare att förstå **selection sort** om man sorterar till en ny vektor. Vi ska sedan se hur man sorterar ”på plats” (eng. *in place*) i en array.

**Indata**: en osorterad vektor med heltal
**Utdata**: en sorterad vektor med heltal

```
def selectionSort(xs: Vector[Int]): Vector[Int] = {
  val unsorted = xs.toBuffer
  val sorted = scala.collection.mutable.ArrayBuffer.empty[Int]
  while (/* unsorted inte är tom */) {
    var indexOfMin = /* index för minsta element i unsorted */
    /* flytta elementet unsorted(indexOfMin) till sist i sorted */
  }
}
```

# Sortera till ny vektor med urvalssortering: implementation i Scala

```
def selectionSort(xs: Vector[Int]): Vector[Int] = {
  val unsorted = xs.toBuffer
  val sorted = scala.collection.mutable.ArrayBuffer.empty[Int]
  while (unsorted.nonEmpty) {
    var indexOfMin = 0
    // index för minsta element i unsorted:
    for (i <- 1 until unsorted.length) {
      if (unsorted(i) < unsorted(indexOfMin)) indexOfMin = i
    }
    val elem = unsorted.remove(indexOfMin)  // ta bort ur unsorted
    sorted.append(elem)  // lägg sist i sekvensen med sorterade
  }
  sorted.toVector
}
```

# Sortera till ny vektor med urvalssortering: implementation i Scala

```
def selectionSort(xs: Vector[Int]): Vector[Int] = {
  val unsorted = xs.toBuffer
  val sorted = scala.collection.mutable.ArrayBuffer.empty[Int]
  while (unsorted.nonEmpty) {
    var indexOfMin = 0
    // index för minsta element i unsorted:
    for (i <- 1 until unsorted.length) {
      if (unsorted(i) < unsorted(indexOfMin)) indexOfMin = i
    }
    val elem = unsorted.remove(indexOfMin)  // ta bort ur unsorted
    sorted.append(elem)  // lägg sist i sekvensen med sorterade
  }
  sorted.toVector
}
```

- Funkar tom sekvens?
- Funkar en sekvens med ett element?
- Funkar det för osorterad sekvens med (minst) två element?
- Vad händer om sekvensen är sorterad?

# Sortera till ny vektor med urvalssortering: implementation i Java

```
public static ArrayList<Integer> selectionSort(ArrayList<Integer> unsorted) {
    ArrayList<Integer> sorted = new ArrayList<Integer>();
    while (unsorted.size() > 0) {
        int indexOfMin = 0;
        // index för minsta element i unsorted:
        for (int i = 1; i < unsorted.size(); i++) {
            if (unsorted.get(i) < unsorted.get(indexOfMin)) {
                indexOfMin = i;
            }
        }
        int elem = unsorted.remove(indexOfMin);  // ta bort ur unsorted
        sorted.add(elem);  // lägg sist i sekvensen med sorterade
    }
    return sorted;
}
```

OBS! Ovan algoritm ”**förstör**” innehållet i inparametern!
Hur förhindra det?

# Sortera till ny vektor med urvalssortering: implementation i Java

```
public static ArrayList<Integer> selectionSort(ArrayList<Integer> xs) {
    ArrayList<Integer> unsorted = new ArrayList<Integer>(xs); //ref copy
    ArrayList<Integer> sorted = new ArrayList<Integer>();
    while (unsorted.size() > 0) {
        int indexOfMin = 0;
        // index för minsta element i unsorted:
        for (int i = 1; i < unsorted.size(); i++) {
            if (unsorted.get(i) < unsorted.get(indexOfMin)) {
                indexOfMin = i;
            }
        }
        int x = unsorted.remove(indexOfMin);  // ta bort ur unsorted
        sorted.add(x);  // lägg sist i sekvensen med sorterade
    }
    return sorted;
}
```

# Urvalssortering på plats: pseudo-kod

**Indata:** en array med heltal
**Utdata:** samma array, men nu sorterad

```
def selectionSortInPlace(xs: Array[Int]): Unit = {
  def minIndex(fromIndex: Int): Int = {
    /* index för minsta element från fromIndex  */
  }

  for (i <- /* från första till NÄST sista index */) {
    /* byt plats mellan xs[i] och xs[minIndex(i)] */
  }
}
```

# Urvalssortering på plats: pseudo-kod

**Indata:** en array med heltal
**Utdata:** samma array, men nu sorterad

```
def selectionSortInPlace(xs: Array[Int]): Unit = {
  def minIndex(fromIndex: Int): Int = {
    /* index för minsta element från fromIndex  */
  }

  for (i <- /* från första till NÄST sista index */) {
    /* byt plats mellan xs[i] och xs[minIndex(i)] */
  }
}
```

Se animering här: Urvalssortering på Wikipedia

# Urvalssortering på plats: implementation i Scala

**Indata:** en array med heltal
**Utdata:** samma array, men nu sorterad

```
1  def selectionSortInPlace(xs: Array[Int]): Unit = {
2    def minIndex(fromIndex: Int): Int = {
3      var result = fromIndex
4      for (i <- fromIndex + 1 until xs.length) {
5        if (xs(i) < xs(result)) result = i
6      }
7      result
8    }
9
10   def swap(i: Int, j: Int): Unit = {
11     val temp = xs(i)
12     xs(i) = xs(j)
13     xs(j) = temp
14   }
15
16   for (i <- 0 until xs.length - 1) { // till NÄST sista
17     swap(i, minIndex(i))
18   }
19 }
```

# Selection sort, in place, Java

Om man ”slår ihop” del-lösningarna minIndex och swap så kan man skriva detta lite kortare och utnyttja att variablen min nedan kan användas i stället för temp.
OBS! Det är viktigare att koden är lättläst än att den är kort och koden optimeras åt oss av kompilatorn och/eller JVM så extra variabler och funktionsanrop är sällan problem.

```
1  public static void selectionSortInPlace(int[] xs) {
2      for (int i = 0; i < xs.length - 1; i++) {
3          int min = Integer.MAX_VALUE;
4          int minIndex = -1;
5          for (int k = i; k < xs.length; k++) {
6              if (xs[k] < min) {
7                  min = xs[k];
8                  minIndex = k;
9              }
10         }
11         xs[minIndex] = xs[i];
12         xs[i] = min;
13     }
14 }
```

# Selection sort, in place, Java

Om man ”slår ihop” del-lösningarna minIndex och swap så kan man skriva detta lite kortare och utnyttja att variablen min nedan kan användas i stället för temp.
OBS! Det är viktigare att koden är lättläst än att den är kort och koden optimeras åt oss av kompilatorn och/eller JVM så extra variabler och funktionsanrop är sällan problem.

```
1  public static void selectionSortInPlace(int[] xs) {
2      for (int i = 0; i < xs.length - 1; i++) {
3          int min = Integer.MAX_VALUE;
4          int minIndex = -1;
5          for (int k = i; k < xs.length; k++) {
6              if (xs[k] < min) {
7                  min = xs[k];
8                  minIndex = k;
9              }
10         }
11         xs[minIndex] = xs[i];
12         xs[i] = min;
13     }
14 }
```

Det finns ett specialfall som kommer krascha denna implementation. Vilket?

# Selection sort, in place, Java

Om man ”slår ihop” del-lösningarna minIndex och swap så kan man skriva detta lite kortare och utnyttja att variablen min nedan kan användas i stället för temp.
OBS! Det är viktigare att koden är lättläst än att den är kort och koden optimeras åt oss av kompilatorn och/eller JVM så extra variabler och funktionsanrop är sällan problem.

```
1  public static void selectionSortInPlace(int[] xs) {
2      for (int i = 0; i < xs.length - 1; i++) {
3          int min = Integer.MAX_VALUE;
4          int minIndex = -1;
5          for (int k = i; k < xs.length; k++) {
6              if (xs[k] < min) {
7                  min = xs[k];
8                  minIndex = k;
9              }
10         }
11         xs[minIndex] = xs[i];
12         xs[i] = min;
13     }
14 }
```

Det finns ett specialfall som kommer krascha denna implementation. Vilket?

```
new int[]{Integer.MAX_VALUE, Integer.MAX_VALUE}
```

# Insättningssortering på plats – pseudo-kod

**Indata:** en array med heltal
**Utdata:** samma array, men nu sorterad

```
def insertionSortInPlace(xs: Array[Int]): Unit = {
  for (i <- 1 until xs.length) {  //från ANDRA till sista
    var j = i
    while (j > 0 && xs(j - 1) > xs(j)) {
      /* byt plats på xs(j) och xs(j - 1) */
      j -= 1;  // stega bakåt
    }
  }
}
```

# Insättningssortering på plats – pseudo-kod

**Indata:** en array med heltal
**Utdata:** samma array, men nu sorterad

```
def insertionSortInPlace(xs: Array[Int]): Unit = {
  for (i <- 1 until xs.length) {  //från ANDRA till sista
    var j = i
    while (j > 0 && xs(j - 1) > xs(j)) {
      /* byt plats på xs(j) och xs(j - 1) */
      j -= 1;  // stega bakåt
    }
  }
}
```

Se animering här: Insättningssortering på wikipedia
Gå igenom alla specialfall och kolla så att detta fungerar!

# Insertion sort, in place, Scala

```
def insertionSortInPlaceSwap(xs: Array[Int]): Unit = {
  def swap(i: Int, j: Int): Unit = {
    val temp = xs(i)
    xs(i) = xs(j)
    xs(j) = temp
  }
  for (i <- 1 until xs.length) {  //från ANDRA till sista
    var j = i
    while (j > 0 && xs(j - 1) > xs(j)) {
      swap(j, j - 1)
      j -= 1;  // stega bakåt
    }
  }
}
```

# Insertion sort, in place, with swap, Java

Vi kan tyvärr inte ha lokala funktioner i Java.

```
private void swap(int[] xs, int a, int b) {
    int temp = xs[a];
    xs[a] = xs[b];
    xs[b] = temp;
}

public void insertionSortInPlaceSwap(int[] xs) {
    for (int i = 1; i < xs.length; i++) {
        int j = i;
        while (j > 0 && xs[j - 1] > xs[j]) {
            swap(xs, j, j - 1);
            j = j - 1;
        }
    }
}
```

# Insertion sort, in place, kortare implementation

Kortare! Men inte mer lättläst?

```
public void insertionSortInPlace(int[] xs) {
    for (int i = 1; i < xs.length; i++) {
        int current = xs[i];
        int j = i;
        while (j > 0 && xs[j - 1] > current) {
            xs[j] = xs[j - 1];
            j--;
        }
        xs[j] = current;
    }
}
```

# Läs mer om insättnings- och urvalssortering

## Insertion sort

- Wikipedia: svenska och engelska: Insertion sort
- AlgoRythmics Insert-sort with Romanian folk dance

## Selection sort

- Wikipedia: svenska och engelska: Selection sort
- AlgoRythmics Select-sort with Gypsy folk dance

# Det finns många olika sorteringsalgoritmer

- Visualisering av 15 olika sorteringsalgoritmer på 6 min
- Olika sorteringsalgoritmer har olika komplexitet:
  i bästa fall, i värsta fall, i medeltal, för nästan sorterad.
  Olika sorteringsalgoritmers egenskaper enl. wikipedia
- Olika sorteringsalgoritmer lämpar sig olika väl för parallellisering på många kärnor.

# Tidskomplexitet, sortering, medeltal

Urvalssortering, insättningssortering: $O(n^2)$
”Bra” metoder, tex Quicksort, Timsort: $O(n \log n)$

Vi har en vektor med 1000 element. Vi har mätt tiden för att sortera elementen många gånger och funnit att det tar ungefär 1 ms både med urvalssortering (eller någon annan ”dålig” metod) och en ”bra” metod. Hur lång tid tar det om vi har fler element i vektorn?

| | 1,000 | 10,000 | 100,000 | 1,000,000 | 10,000,000 |
|---|---|---|---|---|---|
| dålig | 1 | 100 | $10^4$ | $10^6$ | $10^8$ |
| bra | 1 | 13.3 | 167 | 2000 | 23000 |

# Bogo sort

```
def bogoSort(xs: Vector[Int]) = {
  var result = xs
  while(result != result.sorted) {
    result = scala.util.Random.shuffle(result)
  }
  result
}
```

När blir denna färdig?

# Bogo sort

```
def bogoSort(xs: Vector[Int]) = {
  var result = xs
  while(result != result.sorted) {
    result = scala.util.Random.shuffle(result)
  }
  result
}
```

När blir denna färdig?

`https://en.wikipedia.org/wiki/Bogosort`

Antal jämförelser i medeltal vid många mätningar: $O(n \cdot n!)$

# Sortera samlingar med godtyckligt ordningspredikat

```
def sortWith(xs: Vector[Int])(lt: (Int, Int) => Boolean ): Vector[Int] = {
  val sorted = scala.collection.mutable.ArrayBuffer.empty[Int]
  for (elem <- xs) {  // insertion sort using lt as "less than"
     var pos = 0
     while (pos < sorted.length && lt(sorted(pos), elem)) {
       pos += 1
     }
     sorted.insert(pos, elem)
  }
  sorted.toVector
}
```

# Sortera samlingar med godtyckligt ordningspredikat

```
def sortWith(xs: Vector[Int])(lt: (Int, Int) => Boolean ): Vector[Int] = {
  val sorted = scala.collection.mutable.ArrayBuffer.empty[Int]
  for (elem <- xs) {  // insertion sort using lt as "less than"
     var pos = 0
     while (pos < sorted.length && lt(sorted(pos), elem)) {
       pos += 1
     }
     sorted.insert(pos, elem)
  }
  sorted.toVector
}
```

```
1  scala> val xs = Vector(1,2,1,2,12,42,1)
2  xs: scala.collection.immutable.Vector[Int] = Vector(1, 2, 1, 2, 12, 42, 1)
3
4  scala> sortWith(xs)(_ < _)
5  res0: Vector[Int] = Vector(1, 1, 1, 2, 2, 12, 42)
6
7  scala> sortWith(xs)(_ > _)
8  res1: Vector[Int] = Vector(42, 12, 2, 2, 1, 1, 1)
```

# Fördjupning: Sortera samlingar av godtycklig typ

```
def sortWith[T](xs: Vector[T])(lt: (T, T) => Boolean ): Vector[T] = {
  val sorted = scala.collection.mutable.ArrayBuffer.empty[T]
  for (elem <- xs) {
     var pos = 0
     while (pos < sorted.length && lt(sorted(pos), elem)) {
       pos += 1
     }
     sorted.insert(pos, elem)
  }
  sorted.toVector
}
```

# Fördjupning: Sortera samlingar av godtycklig typ

```
def sortWith[T](xs: Vector[T])(lt: (T, T) => Boolean ): Vector[T] = {
  val sorted = scala.collection.mutable.ArrayBuffer.empty[T]
  for (elem <- xs) {
     var pos = 0
     while (pos < sorted.length && lt(sorted(pos), elem)) {
       pos += 1
     }
     sorted.insert(pos, elem)
  }
  sorted.toVector
}
```

```
1  scala> case class Gurka(namn: String, vikt: Int)
2
3  scala> val xs = Vector(Gurka("a", 100), Gurka("b", 50), Gurka("c", 100))
4
5  scala> sortWith(xs)(_.vikt > _.vikt)
6  res0: Vector[Gurka] = Vector(Gurka(c,100), Gurka(a,100), Gurka(b,50))
7
8  scala> sortWith(xs)(_.vikt >= _.vikt)
9  res1: Vector[Gurka] = Vector(Gurka(a,100), Gurka(c,100), Gurka(b,50))
```

# Vad menas med att en sorteringsalgoritm är stabil?

En sorteringsalgoritm är **stabil** om **ordningen** mellan element som anses **lika** enligt sorteringsordningsrelationen **bevaras**.

Fördjupning: en.wikipedia.org/wiki/Sorting_algorithm#Stability

# Sortera samlingar med inbyggda sortBy och sortWith

```
1  scala> case class Gurka(namn: String, vikt: Int)
2  defined class Gurka
3
4  scala> val xs = Vector(Gurka("a", 100), Gurka("b", 50), Gurka("c", 100))
5
6  scala> xs.sortBy(_.vikt)
7  res0: scala.collection.immutable.Vector[Gurka] =
8          Vector(Gurka(b,50), Gurka(a,100), Gurka(c,100))
9
10 scala> xs sortWith (_.vikt > _.vikt)
11 res1: scala.collection.immutable.Vector[Gurka] =
12         Vector(Gurka(a,100), Gurka(c,100), Gurka(b,50))
```

# Fördjupning: sortera samlingar implicit ordning

```
scala> case class Gurka(namn: String, vikt: Int)
defined class Gurka

scala> val xs = Vector(Gurka("a", 100), Gurka("b", 50), Gurka("c", 100))

scala> xs.sorted
<console>:15: error: No implicit Ordering defined for Gurka.
       xs.sorted
          ^

scala> implicit object minGurkOrdning extends Ordering[Gurka] {
         def compare(x: Gurka, y: Gurka): Int =
           if (x == y) 0
           else if (x.vikt < y.vikt) -1
           else 1
       }

scala> xs.sorted
res0: scala.collection.immutable.Vector[Gurka] =
       Vector(Gurka(b,50), Gurka(a,100), Gurka(c,100))
```

# Vad kommer (inte) på tentan?

Detta **kan** komma på tentan:

- Använda färdiga söknings- och sorteringsfunktioner på samlingar av specifik typ
- Implementera egen linjärsökning i samlingar av specifik typ
- Implementera egen binärsökning i samlingar av specifik typ
- Implementera egen sortering till ny samling av specifik typ (du får själv välja algoritm, lämpligen insättnings- eller urvalssortering)

Detta kommer **inte** på tentan:

- Implementera generisk sökning
- Implementera generisk sortering
- Implementera sortering ”på plats”
- Bogo sort