Vecka 7. Sekvenser och enumerationer

Programmering, grundkurs (pgk)

Björn Regnell

Datavetenskap, LTH, Lunds universitet
`https://cs.lth.se/pgk`

EDAA45, Lp1-2, HT2024

Kompilerad den 29 september 2024

7 Sekvenser och enumerationer

- Vad är en sekvens?
- Vad är en sekvensalgoritm?
- Använda färdiga sekvenssamlingsmetoder
- Repeterade parametrar
- Enumerationer
- Registrering
- Skapa lösningar på sekvensproblem från grunden
- Implementera insert, remove, append
- Exempel: Polygon
- Jämföra strängar
- Sökning och sortering
- Uppgifter denna vecka
- Kontrollskrivning

Vad är en sekvens?

ORDNINGEN SPELAR ROLL

Vad är en sekvens?

- En sekvens är en **följd av element** som
  - har **ordningsnummer** (t.ex. numrerade från noll)
  - är av en viss **typ** (t.ex. heltal).

Vad är en sekvens?

- En sekvens är en **följd av element** som
  - har **ordningsnummer** (t.ex. numrerade från noll)
  - är av en viss **typ** (t.ex. heltal).
- En sekvens kan innehålla flera element som är lika.
- En sekvens kan vara **tom** och har då längden noll.
- Exempel på en icke-tom sekvens med dubbletter:

```
scala> val xs = Vector(42, 0, 42, -9, 0, 5)
xs: scala.collection.immutable.Vector[Int] =
  Vector(42, 0, 42, -9, 0, 5)
```

Vad är en sekvens?

- En sekvens är en **följd av element** som
  - har **ordningsnummer** (t.ex. numrerade från noll)
  - är av en viss **typ** (t.ex. heltal).
- En sekvens kan innehålla flera element som är lika.
- En sekvens kan vara **tom** och har då längden noll.
- Exempel på en icke-tom sekvens med dubbletter:

```
scala> val xs = Vector(42, 0, 42, -9, 0, 5)
xs: scala.collection.immutable.Vector[Int] =
  Vector(42, 0, 42, -9, 0, 5)
```

- **Indexering** ger ett element via dess ordningsnummer:

```
scala> xs(2)
res0: Int = 42

scala> xs.apply(2)
res1: Int = 42
```

Exempel: En sträng är en sekvens av tecken

```
scala> "haj po daj"
```

Längd? Vad ligger på första platsen? Elementtyp? Dubbletter?

Exempel: En sträng är en sekvens av tecken

```
scala> "haj po daj"
```

Längd? Vad ligger på första platsen? Elementtyp? Dubbletter?

```
scala> "haj po daj".length
res1: Int = 10

scala> "haj po daj".apply(0)
res2: Char = h

scala> "haj po daj"(0)
res3: Char = h

scala> "haj po daj".distinct
res4: String = haj pod
```

Iterera över element i en sekvens

- Att **iterera** (eng. *iterate*), ä.k. traversera (eng. *traverse*), innebär att **gå igenom** och behandla element i en samling.
- Exempel på iterering med `foreach`, `map`, **`for`**:

```
scala> val xs = Vector(1,2,3)
val xs: Vector[Int] = Vector(1, 2, 3)

scala> xs.foreach(x => println(x + 1))
2
3
4

scala> xs.map(_ + 1)
val res0: Vector[Int] = Vector(2, 3, 4)

scala> for x <- xs yield x - 1
val res1: Vector[Int] = Vector(0, 1, 2)
```

Lägg till i början och i slutet av en sekvens

- Med metoderna `+:` och `:+` kan du skapa en ny sekvens med nya element tillagda i början resp. i slutet.
- Minnesregel: ”**Colon on the collection side**”

```
scala> val xs = Vector(1,2,3)
scala> 42 +: xs          // ger ny Vector(42, 1, 2, 3)
scala> xs :+ 42          // ger ny Vector(1, 2, 3, 42)
```

Lägg till i början och i slutet av en sekvens

- Med metoderna `+:` och `:+` kan du skapa en ny sekvens med nya element tillagda i början resp. i slutet.
- Minnesregel: ”**Colon on the collection side**”

```
scala> val xs = Vector(1,2,3)
scala> 42 +: xs           // ger ny Vector(42, 1, 2, 3)
scala> xs :+ 42           // ger ny Vector(1, 2, 3, 42)
```

- Semantik: operatornotation med operatorer som **slutar med kolon** är **högerassociativa**
- Anropet `42 +: xs` skrivs av kompilatorn om till `xs.+:(42)`

```
1  scala> xs.+:(42)
2  res4: scala.collection.immutable.Vector[Int] = Vector(42, 1, 2, 3)
3
```

Lägg till i början och i slutet av en sekvens

- Med metoderna `+:` och `:+` kan du skapa en ny sekvens med nya element tillagda i början resp. i slutet.
- Minnesregel: ”**Colon on the collection side**”

```
scala> val xs = Vector(1,2,3)
scala> 42 +: xs          // ger ny Vector(42, 1, 2, 3)
scala> xs :+ 42          // ger ny Vector(1, 2, 3, 42)
```

- Semantik: operatornotation med operatorer som **slutar med kolon** är **högerassociativa**
- Anropet `42 +: xs` skrivs av kompilatorn om till `xs.+:(42)`

```
1  scala> xs.+:(42)
2  res4: scala.collection.immutable.Vector[Int] = Vector(42, 1, 2, 3)
3
```

- Konkatenering (sammanfogning) av sekvenser: `xs ++ ys`

Egenskaper hos några sekvenssamlingar i Scala

- `Vector`
  - **Oföränderlig**. Snabb på att skapa kopior med små förändringar.
  - Allsidig prestanda: **bra till det mesta**.
- `List`
  - **Oföränderlig**. Snabb på att skapa kopior med små förändringar.
  - Snabb indexering & uppdatering **i början**.
  - Smidig & snabb vid **rekursiva** algoritmer.
  - Långsam vid upprepad **indexering** på godtyckliga ställen.
- `ArrayBuffer`
  - **Föränderlig**: **snabb indexering & uppdatering**.
  - Kan **ändra storlek** efter allokering. Snabb att indexera överallt.
- `ListBuffer`
  - **Föränderlig**: snabb indexering & uppdatering **i början**.
  - Snabb om du bygger upp sekvens genom många tillägg i början.
- `Array` eller `scala.collection.mutable.ArraySeq`
  - **Föränderlig**: **snabb indexering & uppdatering**.
  - Kan **ej ändra storlek**; storlek ges vid allokering.
  - Har särställning i JVM: ger snabb allokering och access.

Vilken sekvenssamling ska jag välja?

- Välj `Vector` om ...
  a) du vill ha oföränderlighet: `val xs = Vector[Int](1,2,3)`
  b) du behöver föränderlighet (notera **var**):
     `var xs = Vector.empty[Int]`
  c) du ännu inte vet vilken sekvenssamling som är bäst; du kan alltid ändra efter att du mätt prestanda och kollat flaskhalsar vid upprepade körningar.
- Välj `List` om ...
  du har en **rekursiv** sekvensalgoritm och/eller **mestadels jobbar i början**.
- Välj `ArrayBuffer` om ...
  det behövs av prestandaskäl och du **inte** vet storlek vid allokering:
  `val xs = scala.collection.mutable.ArrayBuffer.empty[Int]`
- Välj `ListBuffer` om ...
  det behövs av prestandaskäl och du bara behöver lägga till i början:
  `val xs = scala.collection.mutable.ListBuffer.empty[Int]`
- Välj `Array` eller `ArraySeq` om ...
  det verkligen behövs av prestandaskäl och du **vet** storlek vid allokering:
  `val xs = Array.fill(initSize)(initValue)`

Några konstigheter med Array

- **Referenslikhet** (och inte innehållslikhet):

```
scala> Vector(1,2,3) == Vector(1,2,3) //innehållslikhet
val res0: Boolean = true

scala> Array(1,2,3) == Array(1,2,3) // referenslikhet
val res1: Boolean = false  // aaargh!!
```

  Notera: Metoden == mellan två `ArraySeq` ger **innehållslikhet**.
- Special-syntax för allokering **utan** explicit initialisering:
  `val xs = new Array[String](1000)  // 1000 null-referenser`
- Fungerar inte lika bra med generiska typer:

```
scala> def box[T](x: T) = Vector[T](x)  //funkar fint

scala> def abox[T](x: T) = Array[T](x)
  error: No ClassTag available for T
```

Oföränderlig eller förändringsbar?

- **Oföränderlig**: Kan ej ändra elementreferenserna, men effektiv på att skapa kopia som är (delvis) förändrad **Vector** eller **List**
- **Förändringsbar**: kan ändra elementreferenserna
  - Kan **ej ändra storlek** efter allokering: **Array** eller **ArraySeq**: indexera och uppdatera varsomhelst
  - Kan även ändra storlek efter allokering: **ArrayBuffer** eller **ListBuffer**
- **Ofta funkar oföränderlig sekvenssamling utmärkt**, men om man **efter prestandamätning** upptäcker en flaskhals kan man ändra från **Vector** till t.ex. **ArrayBuffer**.

Vad är en sekvensalgoritm?

Vad är en sekvensalgoritm?

- En algoritm är en stegvis beskrivning av lösningen på ett problem.
- En **sekvensalgoritm** är en algoritm där **element i sekvens** utgör en viktig del av **problembeskrivningen** och/eller **lösningen**.
- Exempelproblem: sortera en sekvens av personer efter deras ålder.

Vad är en sekvensalgoritm?

- En algoritm är en stegvis beskrivning av lösningen på ett problem.
- En **sekvensalgoritm** är en algoritm där **element i sekvens** utgör en viktig del av **problembeskrivningen** och/eller **lösningen**.
- Exempelproblem: sortera en sekvens av personer efter deras ålder.
- **Sju** ofta återkommande programmeringsproblem som löses med en sekvensalgoritm:
  - **Kopiering** av alla element i en sekvens till en **ny** sekvens
  - **Uppdatering** av sekvensen: ta bort, lägga till, ändra **enskilda** element
  - **Transformering**: applicera en **funktion** på **alla** element
  - **Filtrering**: urval av vissa element som uppfyller ett **villkor**
  - **Sökning** efter ett element som uppfyller ett **sökkriterium**
  - **Sortering** enligt någon **ordning**
  - **Registrering** kategorisera eller **räkna element** med vissa egenskaper

KUT FSSR

ORDNINGEN
SPELAR
ROLL
KUT FSSR

Använda färdiga sekvenssamlingsmetoder

Använda färdiga sekvenssamlingsmetoder

- Ofta kan man implementera sekvensalgoritmer genom anrop av en eller flera **färdiga** metoder.
- Dessa färdiga metoder är **optimerade och vältestade** och är att föredra om möjligt.
- Studera Scalas api-dokumentation och kursens quickref för att se vad man kan göra med färdiga metoder.
`https://docs.scala-lang.org/overviews/collections-2.13/introduction.html`
- Det är **lärorikt** att ”**uppfinna hjulet**” och implementera några sekvensalgoritmer **själv** för bättre förståelse, även om de redan finns färdiga i Scalas samlingsbibliotek.

Några användbara samlingsmetoder vid implementation av sekvensalgoritmer

`xs.map(f)`	transformering, motsv. `for x <- xs yield f(x)`
`xs.map(x => x)`	kopiering, motsv. `for x <- xs yield x`
`xs.filter(p)`	filtrering, ta med x om p(x)
`xs.filterNot(p)`	filtrering, ta med x om !p(x)
`xs.distinct`	filtrering, ta bort dubbletter
`xs.take(n)`	ny sekvens med de första n elements, resten skippade
`xs.drop(n)`	ny sekvens där de första n elements är skippade
`xs.takeWhile(p)`	filtrera, ta med i början så länge p(x)
`xs.dropWhile(p)`	filtrera, skippa i början så länge p(x)
`xs.find(p)`	sök framifrån efter första element x där p(x) är sant
`xs.indexOf(x)`	sök framifrån efter index för element som är samma som x
`xs.lastIndexOf(x)`	sök bakifrån efter index för element som är samma som x
`xs.sorted`	sortera med inbyggd (implicit given) ordning
`xs.sorted.reverse`	sortera i omvänd ordning
`xs.sortBy(f)`	sortera i ordning enligt f(x)
`xs.sortWith(lt)`	sortera enligt ”less than”-funktionen `lt: (A, A) => Boolean`
`xs.count(p)`	räkna antalet element där p(x) är sant

Lär dig fler smidiga metoder i quickref

Uppdaterad sekvens med kraftfulla metoden `patch`

Metoden `patch` kan användas så:

`xs.patch(fromPos, ys, nbrReplaced)`

för att skapa en **ny** sekvens där **ett** eller **flera** element i xs är...

- utbytta (eng. *replaced*)
- borttagna (eng. *removed*)
- tillagda (eng. *inserted*)

.. med nya element ur `ys`

```
1  scala> val xs = Vector(1,2,3)
2
3  scala> xs.patch(2, Vector(-1), 1)     // replaced one elem
4  res0: scala.collection.immutable.Vector[Int] = Vector(1, 2, -1)
5
6  scala> xs.patch(1, Vector(42), 0)     // inserted one elem
7  res11: scala.collection.immutable.Vector[Int] = Vector(1, 42, 2, 3)
8
9  scala> xs.patch(0, Vector(), 2)       // removed two elems
10 res2: scala.collection.immutable.Vector[Int] = Vector(3)
```

Använda for-uttryck för filtrering med hjälp av gard

I ett for-uttryck kan man ha en **gard** (eng. *guard*) i form av ett booleskt uttryck efter nyckelordet **`if`**. Då kommer uttrycket efter **`yield`** bara göras om gard-uttrycket är sant.

Syntaxen är så här: (parenteser behövs ej runt gard-uttrycket)

```
for x <- xs if uttryck1 yield uttryck2
```

Använda for-uttryck för filtrering med hjälp av gard

I ett for-uttryck kan man ha en **gard** (eng. *guard*) i form av ett booleskt uttryck efter nyckelordet **if**. Då kommer uttrycket efter **yield** bara göras om gard-uttrycket är sant.

Syntaxen är så här: (parenteser behövs ej runt gard-uttrycket)

```
for x <- xs if uttryck1 yield uttryck2
```

Exempel:

```
scala> val udda = for x <- 1 to 6 if x % 2 == 1 yield x
```

Använda for-uttryck för filtrering med hjälp av gard

I ett for-uttryck kan man ha en **gard** (eng. *guard*) i form av ett booleskt uttryck efter nyckelordet **`if`**. Då kommer uttrycket efter **`yield`** bara göras om gard-uttrycket är sant.

Syntaxen är så här: (parenteser behövs ej runt gard-uttrycket)

```
for x <- xs if uttryck1 yield uttryck2
```

Exempel:

```
scala> val udda = for x <- 1 to 6 if x % 2 == 1 yield x
```

udda blir `Vector(1, 3, 5)`

Använda samlingsmetoden `filter` för filtrering

Alla samlingar i `scala.collection` har metoden `filter`. Den har ett predikat som parameter `p: T => Boolean` och ger en ny samling med de element för vilka predikatet är sant.

```
xs.filter(p)
```

Använda samlingsmetoden `filter` för filtrering

Alla samlingar i `scala.collection` har metoden `filter`. Den har ett predikat som parameter `p: T => Boolean` och ger en ny samling med de element för vilka predikatet är sant.

```
xs.filter(p)
```

Exempel: Antag att `xs` är `(1 to 6).toVector`

```
xs.filter(_ % 2 == 1)
```

Använda samlingsmetoden `filter` för filtrering

Alla samlingar i `scala.collection` har metoden `filter`. Den har ett predikat som parameter `p: T => Boolean` och ger en ny samling med de element för vilka predikatet är sant.

```
xs.filter(p)
```

Exempel: Antag att `xs` är `(1 to 6).toVector`

```
xs.filter(_ % 2 == 1)
```

uttryckets resultat blir `Vector(1, 3, 5)`, vilket motsvarar:

```
for x <- xs if x % 2 == 1 yield x
```

Använda samlingsmetoden `filter` för filtrering

Alla samlingar i `scala.collection` har metoden `filter`. Den har ett predikat som parameter `p: T => Boolean` och ger en ny samling med de element för vilka predikatet är sant.

```
xs.filter(p)
```

Exempel: Antag att `xs` är `(1 to 6).toVector`

```
xs.filter(_ % 2 == 1)
```

uttryckets resultat blir `Vector(1, 3, 5)`, vilket motsvarar:

```
for x <- xs if x % 2 == 1 yield x
```

I själva verket skriver Scala-kompilatorn om for-uttryck med gard till anrop av metoden `filter` före kodgenerering sker.

Vanliga sekvensproblem som funktionshuvuden

Indata och utdata för några vanliga sekvensproblem:

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

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

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

def sort(xs: Vector[Int]): Vector[Int] = ???

def freq(xs: Vector[Int]): Vector[(Int, Int)] = ???  // (heltal, frekvens)
```

Övning: Hur implementera dessa med `for`-uttryck och/eller färdiga samlingsmetoder?

Tips: För `sort`&`freq` se `sorted`, `distinct`, `count` i quickref

Implementation av sekvensproblem med `for`-uttryck och/eller färdiga samlingsmetoder

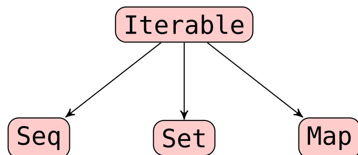
```scala
def copy(xs: Vector[Int]): Vector[Int] = for x <- xs yield x

def filter(xs: Vector[Int], p: Int => Boolean): Vector[Int] =
  for x <- xs if p(x) yield x

def findIndices(xs: Vector[Int], p: Int => Boolean): Vector[Int] =
  (for i <- xs.indices if p(xs(i)) yield i).toVector

def sort(xs: Vector[Int]): Vector[Int] = xs.sorted // mer om sortering sen

def freq(xs: Vector[Int]): Vector[(Int, Int)] = // mer om registrering snart
  for x <- xs.distinct yield x -> xs.count(_ == x)
```

Övning: Hur implementera dessa med `map` och `filter` och/eller andra färdiga samlingsmetoder?

Implementation av sekvensproblem med map, filter

```scala
def copy(xs: Vector[Int]): Vector[Int] = xs.map(x => x)

def filter(xs: Vector[Int], p: Int => Boolean): Vector[Int] = xs.filter(p)

def findIndices(xs: Vector[Int], p: Int => Boolean): Vector[Int] =
  xs.indices.filter(i => p(xs(i))).toVector

def sort(xs: Vector[Int]): Vector[Int] = xs.sorted // mer om sortering sen

def freq(xs: Vector[Int]): Vector[(Int, Int)] = // mer om registrering snart
  xs.distinct.map(x => x -> xs.count(_ == x))
```

Hierarki av samlingstyper i `scala.collection` v2.13



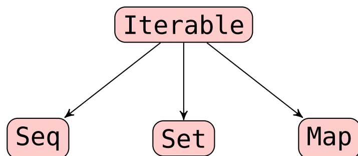
`Iterable` har metoder som är implementerade med hjälp av:

```scala
def foreach[U](f: Elem => U): Unit
def iterator: Iterator[A]
```

- `Seq`: ordnade i sekvens
- `Set`: unika element
- `Map`: par av (nyckel, värde)

Samlingen **`Vector`** är en `Seq` som är en `Iterable`.

Hierarki av samlingstyper i `scala.collection` v2.13



`Iterable` har metoder som är implementerade med hjälp av:

```
def foreach[U](f: Elem => U): Unit
def iterator: Iterator[A]
```

`Seq`: ordnade i sekvens

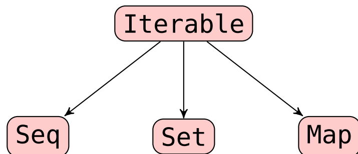
`Set`: unika element

`Map`: par av (nyckel, värde)

Samlingen **`Vector`** är en `Seq` som är en `Iterable`.

De konkreta samlingarna är uppdelade i dessa paket:
`scala.collection.immutable` där flera är **automatiskt** importerade
`scala.collection.mutable` som **måste importeras** explicit

Hierarki av samlingstyper i `scala.collection` v2.13



`Iterable` har metoder som är implementerade med hjälp av:

```
def foreach[U](f: Elem => U): Unit
def iterator: Iterator[A]
```

`Seq`: ordnade i sekvens

`Set`: unika element

`Map`: par av (nyckel, värde)

Samlingen **`Vector`** är en `Seq` som är en `Iterable`.

De konkreta samlingarna är uppdelade i dessa paket:
`scala.collection.immutable` där flera är **automatiskt** importerade
`scala.collection.mutable` som **måste importeras** explicit

(undantag: primitiva `scala.Array`)

Lämna det öppet: använd `Seq`

Typen **`collection.immutable.Seq`** är supertyp till alla sekvenssamlingar i `collection.immutable`.

Lämna det öppet: använd Seq

Typen **`collection.immutable.Seq`** är supertyp till alla sekvenssamlingar i `collection.immutable`. Exempel: kopiering av sekvens:

- Kopiering av **specifik** heltalssekvens:

```
def copyIntVector(xs: Vector[Int]): Vector[Int] = for x <- xs yield x
```

- Kopiering som fungerar för alla oföränderliga heltalssekvenser:

```
def copyIntSeq(xs: Seq[Int]): Seq[Int] = for x <- xs yield x
```

Lämna det öppet: använd Seq

Typen **`collection.immutable.Seq`** är supertyp till alla sekvenssamlingar i `collection.immutable`. Exempel: kopiering av sekvens:

- Kopiering av **specifik** heltalssekvens:

```
def copyIntVector(xs: Vector[Int]): Vector[Int] = for x <- xs yield x
```

- Kopiering som fungerar för alla oföränderliga heltalssekvenser:

```
def copyIntSeq(xs: Seq[Int]): Seq[Int] = for x <- xs yield x
```

```
1  scala> val xs = Vector(1,2,3)
2  xs: Vector[Int] = Vector(1, 2, 3)
3
4  scala> val ys = copyIntVector(xs)
5  ys: Vector[Int] = Vector(1, 2, 3)
6
7  scala> val zs = copyIntSeq(xs)
8  val zs: Seq[Int] = Vector(1, 2, 3)
```

Implementation med generiska funktioner

Genom att generalisera funktionshuvudena blir våra lösningar användbara för **alla** sekvenser av typen `Seq[T]`, där den obundna **typparametern** T vid anrop kan bindas till godtycklig typ. (Mer om typparametrar senare.)

```scala
def copy[T](xs: Seq[T]): Seq[T] = xs.map(x => x)

def filter[T](xs: Seq[T], p: T => Boolean): Seq[T] = xs.filter(p)

def findIndices[T](xs: Seq[T], p: T => Boolean): Seq[Int] =
  xs.indices.filter(i => p(xs(i))).toVector

def sort[T: Ordering](xs: Seq[T]): Seq[T] = xs.sorted // mer om Ordering sen

def freq[T](xs: Seq[T]): Seq[(T, Int)] =
  xs.distinct.map((_, xs.count(_ == x)))
```

Implementation med generiska funktioner

Genom att generalisera funktionshuvudena blir våra lösningar användbara för **alla** sekvenser av typen `Seq[T]`, där den obundna **typparametern** T vid anrop kan bindas till godtycklig typ. (Mer om typparametrar senare.)

```
def copy[T](xs: Seq[T]): Seq[T] = xs.map(x => x)

def filter[T](xs: Seq[T], p: T => Boolean): Seq[T] = xs.filter(p)

def findIndices[T](xs: Seq[T], p: T => Boolean): Seq[Int] =
  xs.indices.filter(i => p(xs(i))).toVector

def sort[T: Ordering](xs: Seq[T]): Seq[T] = xs.sorted // mer om Ordering sen

def freq[T](xs: Seq[T]): Seq[(T, Int)] =
  xs.distinct.map((_, xs.count(_ == x)))
```

Standardbibliotekets metoder försöker ordna så att det blir samma konkreta typ in som ut, men ibland väljs annan lämplig konkret samling, t.ex. kan en `Array` bli en `ArrayBuffer`.

Använda Java-samlingar i Scala med `CollectionConverters`

Med hjälp av **import** `scala.jdk.CollectionConverters.*`
får man smidig **interoperabilitet** med Java och dess standardbibliotek, speciellt metoderna **asJava** och **asScala**:

```
1  scala> import scala.jdk.CollectionConverters.*
2
3  scala> Vector(1,2,3).asJava
4  res0: java.util.List[Int] = [1, 2, 3]
5
6  scala> val xs = new java.util.ArrayList[String]()
7  xs: java.util.ArrayList[String] = []
8
9  scala> xs.add("hej")
10 res1: Boolean = true
11
12 scala> xs.asScala
13 res2: scala.collection.mutable.Buffer[String] = Buffer(hej)
```

Läs mer här: https://docs.scala-lang.org/overviews/collections-2.13/conversions-between-java-and-scala-collections.html

Fördjupning: Skapa generisk Array

- I JVM bytekod går det tyvärr **inte** att skapa en primitiv generisk array.
- Maskinkoden måste istället skapa en array av den mest generella referenstypen `Object` och sedan **typtesta och typkonvertera under körtid**.
  Se t.ex. Java-implementationen av `ArrayList`:
  http://developer.classpath.org/doc/java/util/ArrayList-source.html

- Men det ***går*** att skapa en generisk array i Scala (men inte i Java). Då behövs en `reflect.ClassTag` som möjliggör typinformation vid körtid för arrayer.

```
scala> def fyll[T](n: Int, x: T): Array[T] = Array.fill(n)(x)
-- Error:
1 |def fyll[T](n: Int, x: T): Array[T] = Array.fill(n)(x)
  |                                                      ^
  |  No ClassTag available for T

scala> def fyll[T: reflect.ClassTag](n: Int, x: T): Array[T] = Array.fill(n)(x)

scala> fyll(42, "hej")
res2: Array[String] = Array(hej, hej, hej, hej, hej, hej, hej, hej, hej, hej, hej, h
```

- Kompilatorn skapar då maskinkod som automatiskt gör typkonverteringarna.

Repeterade parametrar

Repeterade parametrar blir sekvens

Med en asterisk efter parametertypen kan antalet argument variera:

```
def sumSizes(xs: String*): Int = xs.map(_.length).sum
```

```
scala> sumSizes("Zaphod")
res0: Int = 6

scala> sumSizes("Zaphod","Beeblebrox")
res1: Int = 16

scala> sumSizes("Zaphod","Beeblebrox","Ford","Prefect")
res3: Int = 27

scala> sumSizes()
res4: Int = 0
```

Repeterade parametrar (eng. *repeated parameters*) blir en sekvens av typen `Seq` och som mer specifikt är en `ArraySeq`

Sekvenssamling som argument till repeterade parametrar

```
def sumSizes(xs: String*): Int = xs.map(_.size).sum

val veg = Vector("gurka","tomat")
```

Om du *redan har* en sekvenssamling så kan du applicera den på en funktion som har repeterade parametrar med hjälp av en asterisk *

Den ska skrivas direkt **efter** den sekvenssamling, som du vill att kompilatorn ska tolka som en sekvens av argument, så här:

```
scala> sumSizes(veg*)
res5: Int = 10
```

Enumerationer

Enumerationer har en ordning

En uppräkning av färger i en kortlek med **enum**:

```
enum Suit:
  case Spade, Heart, Club, Diamond
```

Användbara metoder för att hantera elementens **ordningen**:

```
scala> Suit.Spade.ordinal      // från element till heltal
val res0: Int = 0

scala> Suit.Club.ordinal
val res1: Int = 2

scala> Suit.fromOrdinal(3)    // från heltal till element
val res2: Suit = Diamond

scala> Suit.values            // alla element i ordning
val res3: Array[Suit] = Array(Spade, Heart, Club, Diamond)

scala> Suit.valueOf("Spade")  // från sträng till element
val res4: Suit = Spade
```

Enumerationer kan ha parametrar och medlemmar

En **enum** kan ha parametrar. Använd **val** för extern synlighet:

```
enum Color(val consoleColor: String):
  case Black extends Color(Console.BLUE) //Blå färg syns på svart bakgrund
  case Red   extends Color(Console.RED)
```

I **enum**-kroppen kan du ha medlemmar, tex metoder:

```
enum Suit(val color: Color):
  def show(isConsoleColor: Boolean = true): String =
    if isConsoleColor then color.consoleColor + toString + Console.RESET
    else toString

  case Spade   extends Suit(Color.Black)
  case Heart   extends Suit(Color.Red)
  case Club    extends Suit(Color.Black)
  case Diamond extends Suit(Color.Red)
```

```
scala> println(Suit.Club.show(isConsoleColor = false))
Club
```

Enum kan bli fullfjädrade case-klasser

Vill du kunna göra mönster-matching på enum-värden så behövs parametrar på alternativen för att det ska bli motsvarande case-klasser:

```
enum Veg:
  def taste: String
  case Tomato(taste: String)
  case Banana(taste: String)
```

Ovan expanderas automatiskt av kompilatorn till motsvarande detta:

```
sealed trait Veg:
  def taste: String
object Veg:
  case class Tomato(taste: String) extends Veg
  case class Banana(taste: String) extends Veg
```

Enum och mönster-matchning

Med parametrar på varje fall och en abstrakt medlem för varje attribut...

```
enum Veg:
  def taste: String
  case Tomato(taste: String)
  case Banana(taste: String)
```

...så gör den automatiska expansionen till case-klasser att detta fungerar fint:

```
scala> val v = Veg.Tomato("nice")
val v: Veg = Tomato(nice)             // notera typen : Veg

scala> v.taste  // funkar eftersom Veg har en taste
val res0: String = najs

scala> val dontLikeBananas = v match:
           case Veg.Tomato(t) => t
           case Veg.Banana(_) => "always bad!"
```

Den abstrakta medlemmen `def taste: String` behövs för att attributet ska synas via referenser som är av den mindre specifika typen Veg. (Mer om abstrakta medlemmar i veckan om arv.)

Fördelar med **enum** jämfört med uppräkning med heltal

Varför inte bara så här?

```scala
val (Spade, Heart, Club, Diamond) = (0, 1, 2, 3)
```

Alla element har samma specifika typ enligt **enum**-deklarationen:

```
1 scala> Suit.Heart                   // alla element är av typen Suit
2 val res5: Suit = Heart
```

- Detta är säkrare jämfört med att bara använda heltalsvärden: kompilatorn kan hjälpa dig att skilja på element av olika typ och ge felmeddelande om du använder fel typ oavsiktligt.
- Ej tillåtna värden kan inte representeras (jmf alla möjliga heltal, där bara några är relevanta).

Detta får du prova på veckans labb: först använda heltal sedan **enum**.

Registrering

Registrering

- **Registrering** innefattar algoritmer för att kategorisera eller räkna antalet förekomster av element med vissa specifika egenskaper.
- Exempel:

  Utfallsfrekvens vid kast med en tärning 1000 gånger:

utfall		antal
1	$\rightarrow$	178
2	$\rightarrow$	187
3	$\rightarrow$	167
4	$\rightarrow$	148
5	$\rightarrow$	155
6	$\rightarrow$	165

Registrering av tärningskast i `Array`

Vi låter plats 0 representera antalet ettor, plats 1 representerar antalet tvåor etc. **Övning**: implementera `???`

```
scala> def rollDice(): Int = ???  //dra slumptal 1-6

scala> val reg = ??? //skapa heltalsarray med 6 platser
reg: Array[Int] = Array(0, 0, 0, 0, 0, 0)

scala> for k <- 1 to 1000 do
         ??? //kasta tärning, räkna ut rätt index
         ??? //registrera

scala> for i <- 1 to 6 do println(s"$i: ${reg(i - 1)}")
1: 178
2: 187
3: 167
4: 148
5: 155
6: 165
```

Registrering av tärningskast i Array

Lösning:

```
scala> def rollDice() = scala.util.Random.nextInt(6) + 1

scala> val reg = new Array[Int](6)
reg: Array[Int] = Array(0, 0, 0, 0, 0, 0)

scala> for k <- 1 to 1000 do
         val i = rollDice() - 1
         reg(i) = reg(i) + 1    // eller: reg(i) += 1

scala> for i <- 1 to 6 do println(s"$i: ${reg(i - 1)}")
1: 178
2: 187
3: 167
4: 148
5: 155
6: 165
```

Skapa lösningar på sekvensproblem från grunden

Skapa lösningar på sekvensproblem från grunden

- Normalt använder man färdiga samlingsmetoder
- Det finns ofta en färdig metod som gör det man vill
- Annars kan man ofta göra det man vill genom att kombinera flera färdiga samlingsmetoder

Skapa lösningar på sekvensproblem från grunden

- Normalt använder man färdiga samlingsmetoder
- Det finns ofta en färdig metod som gör det man vill
- Annars kan man ofta göra det man vill genom att kombinera flera färdiga samlingsmetoder

- Vi ska nu i lärosyfte implementera några egna varianter av uppdatering från grunden.

För problem av typen KUTFSSR ingår det i kursen att kunna 1) lösa dessa med färdiga samlingsmetoder, och 2) implementera egna lösningar med hjälp av sekvens, alternativ, repetition, abstraktion (**SARA**).

Skapa ny sekvenssamling eller ändra på plats?

Två olika principer vid sekvensalgoritmkonstruktion:

- Skapa **ny sekvens** utan att förändra insekvensen
- Ändra **på plats** (eng. *in-place*) i **förändringsbar** sekvens

Skapa ny sekvenssamling eller ändra på plats?

Två olika principer vid sekvensalgoritmkonstruktion:

- Skapa **ny sekvens** utan att förändra insekvensen
- Ändra **på plats** (eng. *in-place*) i **förändringsbar** sekvens

Välja mellan att skapa ny sekvens eller ändra på plats?

- Ofta är det **lättast att skapa ny samling** och kopiera över elementen efter eventuella förändringar medan man loopar.
- Om man har mycket stora samlingar kan man behöva ändra på plats för att spara tid/minne.

Algoritm: SEQ-COPY

Pseudokod för algoritmen SEQ-COPY som kopierar en sekvens, här en Array med heltal:

Indata : Heltalsarray *xs*
Utdata: En ny heltalsarray som är en kopia av *xs*.

result $\leftarrow$ en ny array med plats för *xs.length* element
$i \leftarrow 0$
while $i < xs.length$ **do**
    $result(i) \leftarrow xs(i)$
    $i \leftarrow i + 1$
end
result

Implementation av SEQ-COPY med while

```
object seqCopy:

  def arrayCopy(xs: Array[Int]): Array[Int] =
    val result = new Array[Int](xs.length)
    var i = 0
    while i < xs.length do
      result(i) = xs(i)
      i += 1
    result

  def test: String =
    val xs = Array(1,2,3,4,42)
    val ys = arrayCopy(xs)
    if xs sameElements ys then "OK!" else "ERROR!"

  def main(args: Array[String]): Unit = println(test)
```

Implementera insert, remove, append

Typ-alias för att abstrahera typnamn

Med hjälp av nyckelordet **type** kan man deklarera ett **typ-alias** för att ge ett **alternativt** namn till en viss typ. Exempel:

```
1  scala> type Pt = (Int, Int)            // typalias
2  scala> type Pts = Vector[Pt]           // nästlad typalias
3
4  scala> def distToOrigo(pt: Pt): Double = math.hypot(pt._1, pt._2)
5
6  scala> val xs: Pts = Vector((1,1), (2,2), (3,4))
7  val xs: Pts = Vector((1,1), (2,2), (3,4))
8
9  scala> xs.head
10 val res0: Pt = (1,1)
11
12 scala> xs.map(distToOrigo)
13 val res1: Vector[Double] = Vector(1.4142135623730951, 2.8284271247461903, 5.0)
```

Typ-alias kan vara bra när:

- man har en lång och krånglig typ och vill använda ett kortare namn,
- man vill kunna lätt byta implementation senare
  (t.ex. om man vill använda en case-klass i stället för en tupel).

Exempel: SEQ-INSERT/REMOVE-COPY

Nu ska vi ”uppfinna hjulet” och som träning implementera **insättning** och **borttagning** till en **ny** sekvens utan användning av sekvenssamlingsmetoder (förutom `length` och `apply`):

```
object PointSeqUtils:
  type Pt = (Int, Int)  // a type alias to make the code more concise

  def primitiveInsertCopy(pts: Array[Pt], pos: Int, pt: Pt): Array[Pt] = ???

  def primitiveRemoveCopy(pts: Array[Pt], pos: Int): Array[Pt] = ???
```

Pseudo-kod för SEQ-INSERT-COPY

Indata : `pts: Array[Pt], pt: Pt, pos: Int`

Utdata: En kopia av *pts* men där *pt* är infogat på plats *pos*

---

result ← en ny `Array[Pt]` med plats för *pts.length* + 1 element
for *i* ← 0 **to** *pos* − 1 **do**
| *result*(*i*) ← *pts*(*i*)
end
result(*pos*) ← *pt*
for *i* ← *pos* + 1 **to** *xs.length* **do**
| *result*(*i*) ← *pts*(*i* − 1)
end
result

Pseudo-kod för SEQ-INSERT-COPY

Indata : `pts: Array[Pt], pt: Pt, pos: Int`

Utdata: En kopia av *pts* men där *pt* är infogat på plats *pos*

---

result $\leftarrow$ en ny `Array[Pt]` med plats för *pts.length* $+ 1$ element
for $i \leftarrow 0$ **to** $pos - 1$ **do**
| *result*$(i) \leftarrow$ *pts*(i)
end
result$(pos) \leftarrow pt$
for $i \leftarrow pos + 1$ **to** *xs.length* **do**
| *result*$(i) \leftarrow$ *pts*$(i - 1)$
end
result

Övning: Skriv pseudo-kod för SEQ-REMOVE-COPY

Insättning/borttagning i kopia av primitiv Array

```
object PointSeqUtils:
  type Pt = (Int, Int)  // a type alias to make the code more concise

  def primitiveInsertCopy(pts: Array[Pt], pos: Int, pt: Pt): Array[Pt] =
    val result = new Array[Pt](pts.length + 1)  // initialized with null
    for i <- 0 until pos do result(i) = pts(i)
    result(pos) = pt
    for i <- pos + 1 to pts.length do result(i) = pts(i - 1)
    result

  def primitiveRemoveCopy(pts: Array[Pt], pos: Int): Array[Pt] =
    if pts.length > 0 then
      val result = new Array[Pt](pts.length - 1) // initialized with null
      for i <- 0 until pos do result(i) = pts(i)
      for i <- pos + 1 until pts.length do result(i - 1) = pts(i)
      result
    else Array.empty

  // ovan metoder implementerade med hjälp av den kraftfulla metoden patch:

  def insertCopy(pts: Array[Pt], pos: Int, pt: Pt) = pts.patch(pos, Array(pt), 0)

  def removeCopy(pts: Array[Pt], pos: Int) = pts.patch(pos, Array.empty[Pt], 1)
```

Insättning/borttagning i kopia av primitiv Array

```
object PointSeqUtils:
  type Pt = (Int, Int)  // a type alias to make the code more concise

  def primitiveInsertCopy(pts: Array[Pt], pos: Int, pt: Pt): Array[Pt] =
    val result = new Array[Pt](pts.length + 1)  // initialized with null
    for i <- 0 until pos do result(i) = pts(i)
    result(pos) = pt
    for i <- pos + 1 to pts.length do result(i) = pts(i - 1)
    result

  def primitiveRemoveCopy(pts: Array[Pt], pos: Int): Array[Pt] =
    if pts.length > 0 then
      val result = new Array[Pt](pts.length - 1) // initialized with null
      for i <- 0 until pos do result(i) = pts(i)
      for i <- pos + 1 until pts.length do result(i - 1) = pts(i)
      result
    else Array.empty

  // ovan metoder implementerade med hjälp av den kraftfulla metoden patch:

  def insertCopy(pts: Array[Pt], pos: Int, pt: Pt) = pts.patch(pos, Array(pt), 0)

  def removeCopy(pts: Array[Pt], pos: Int) = pts.patch(pos, Array.empty[Pt], 1)
```

Man gör **mycket lätt fel** på gränser/specialfall: +-1, to/until, tom sekvens etc.

Exempel: Polygon

Exempel: PolygonWindow

- En polygon kan representeras som en punktsekvens, där varje punkt är ett heltalspar.
- `PolygonWindow` nedan är ett fönster som kan rita en polygon.

```
1  class PolygonWindow(width: Int, height: Int):
2    val w = new introprog.PixelWindow(width, height, title = "PolygonWindow")
3
4    def draw(pts: Seq[(Int, Int)]): Unit =
5      if pts.size > 0 then
6        for i <- 1 until pts.size do
7          w.line(pts(i - 1)._1, pts(i - 1)._2, pts(i)._1, pts(i)._2)
8        val last = pts.length - 1
9        w.line(pts(last)._1, pts(last)._2, pts(0)._1, pts(0)._2)
```

Exempel: PolygonWindow

- En polygon kan representeras som en punktsekvens, där varje punkt är ett heltalspar.
- `PolygonWindow` nedan är ett fönster som kan rita en polygon.

```
class PolygonWindow(width: Int, height: Int):
  val w = new introprog.PixelWindow(width, height, title = "PolygonWindow")

  def draw(pts: Seq[(Int, Int)]): Unit =
    if pts.size > 0 then
      for i <- 1 until pts.size do
        w.line(pts(i - 1)._1, pts(i - 1)._2, pts(i)._1, pts(i)._2)
      val last = pts.length - 1
      w.line(pts(last)._1, pts(last)._2, pts(0)._1, pts(0)._2)
```

```
object PolygonTest:
  val star =  Array((100,180), (150,100), (180,180), (90,130), (200, 130))
  val pw = new PolygonWindow(400,400)
  def main(args: Array[String]): Unit = pw.draw(star.toSeq)
```

Implementera Polygon

- En polygon kan representeras som en sekvens av punkter.
- Vi vill kunna lägga till punkter, samt ta bort punkter.
- En polygon kan implementeras på många olika sätt:

Implementera Polygon

- En polygon kan representeras som en sekvens av punkter.
- Vi vill kunna lägga till punkter, samt ta bort punkter.
- En polygon kan implementeras på många olika sätt:
  - **Förändringsbar** (eng. *mutable*)
    - Med punkterna i en `Array`
    - Med punkterna i en `ArrayBuffer`
    - Med punkterna i en `ListBuffer`
    - Med punkterna i en `Vector`
    - Med punkterna i en `List`
  - **Oföränderlig** (eng. *immutable*)
    - Som en case-klass med en oföränderlig `Vector` som returnerar nytt objekt vid uppdatering. Vi kan låta datastrukturen vara **publik** eftersom allt är oföränderligt.
    - Som en ”vanlig” klass med någon lämplig **privat** datastruktur där vi **inte** möjliggör förändring av efter initialisering och där vi returnerar nytt objekt vid uppdatering.

Implementera Polygon

- En polygon kan representeras som en sekvens av punkter.
- Vi vill kunna lägga till punkter, samt ta bort punkter.
- En polygon kan implementeras på många olika sätt:
  - **Förändringsbar** (eng. *mutable*)
    - Med punkterna i en `Array`
    - Med punkterna i en `ArrayBuffer`
    - Med punkterna i en `ListBuffer`
    - Med punkterna i en `Vector`
    - Med punkterna i en `List`
  - **Oföränderlig** (eng. *immutable*)
    - Som en case-klass med en oföränderlig `Vector` som returnerar nytt objekt vid uppdatering. Vi kan låta datastrukturen vara **publik** eftersom allt är oföränderligt.
    - Som en ”vanlig” klass med någon lämplig **privat** datastruktur där vi **inte** möjliggör förändring av efter initialisering och där vi returnerar nytt objekt vid uppdatering.

Val av implementation **beror på** sammanhang & användning!

Exempel: PolygonArray, ändring på plats

```
1  class PolygonArray(val maxSize: Int):
2    type Pt = (Int, Int)
3    private val points = new Array[Pt](maxSize)  // initialized with null
4    private var n = 0
5    def size = n
6
7    def draw(w: PolygonWindow): Unit = w.draw(points.take(n).toSeq)
8
9    def append(pts: Pt*): Unit =
10     for i <- pts.indices do points(n + i) = pts(i)
11     n += pts.length
12
13   def insert(pos: Int, pt: Pt): Unit = // exercise: change pt to varargs pts
14     for i <- n until pos by -1 do points(i) = points(i - 1)
15     points(pos) = pt
16     n += 1
17
18   def remove(pos: Int): Unit = // exercise: change pos to fromPos, replaced
19     for i <- pos until n do points(i) = points(i + 1)
20     n -= 1
21
22   override def toString = points.mkString("PolygonArray(",",",")")
```

Exempel: PolygonVector, variabel referens till oföränderlig datastruktur

```
 1 class PolygonVector:
 2   type Pt = (Int, Int)
 3   private var points = Vector.empty[Pt]  // note var declaration to allow mutation
 4   def size = points.size
 5
 6   def draw(w: PolygonWindow): Unit = w.draw(points.take(size))
 7
 8   def append(pts: Pt*): Unit =
 9     points ++= pts.toVector
10
11   def insert(pos: Int, pt: Pt): Unit = // exercise: change pt to varargs pts
12     points = points.patch(pos, Vector(pt), 0)
13
14   def remove(pos: Int): Unit = // exercise: change pos to fromPos, replaced
15     points = points.patch(pos, Vector(), 1)
16
17   override def toString = points.mkString("PrimitivePolygon(",",",")")
```

Exempel: Polygon som oföränderlig case class

```
1   object Polygon:
2     type Pt = (Int, Int)
3     type Pts = Vector[Pt]
4     def apply(pts: Pt*) = new Polygon(pts.toVector)
5
6   case class Polygon(points: Polygon.Pts):
7     import Polygon.Pt
8
9     def size = points.size // for convenience but not really necessary (why?)
10
11    def append(pts: Pt*): Polygon = copy(points ++ pts.toVector)
12
13    def insert(pos: Int, pts: Pt*): Polygon = copy(points.patch(pos, pts, 0))
14
15    def remove(pos: Int, replaced: Int = 1): Polygon =
16      copy(points.patch(pos, Seq(), replaced))
17
18    override def toString = points.mkString("Polygon(", "," ,")")
```

Jämföra strängar

Att jämföra strängar lexikografiskt

Teckenstandard UTF-8: Alla stora bokstäver är ”mindre” än alla små:

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

Att jämföra strängar lexikografiskt

Teckenstandard UTF-8: 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-8: 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-8: 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 */) then
    var foundDiff = false
    var i = /* första index */
    while !foundDiff && /* i inom indexgräns */ do
      if /* tecken på plats i är olika */ then foundDiff = true
      else i = /* nästa index */
    end while
    !foundDiff
  else false
end isEqual
```

Algoritmexempel: stränglikhet, pseudokod

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

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 */) then
    var foundDiff = false
    var i = /* första index */
    while !foundDiff && /* i inom indexgräns */ do
      if /* tecken på plats i är olika */ then foundDiff = true
      else i = /* nästa index */
    end while
    !foundDiff
  else false
end isEqual
```

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 then
    var foundDiff = false
    var i = 0
    while !foundDiff && i < s1.length do
      if s1(i) != s2(i) then foundDiff = true
      else i += 1
    end while
    !foundDiff
  else false
end isEqual
```

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 then /* 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 do
      if (s1(i) != s2(i)) foundDiff = true
      else i += 1
    end while
    i
  end firstDiff

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

Sökning och sortering

Sökning

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

Sökning

- **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:

```
scala> Vector("gurka","tomat","broccoli").indexOf("tomat")
res0: Int = 1

scala> Vector("gurka","tomat","broccoli").indexWhere(_.contains("o"))
res1: Int = 1

scala> Vector("gurka","tomat","broccoli").find(_.contains("o"))
res2: Option[String] = Some(tomat)
```

Sökning

- **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:

```
scala> Vector("gurka","tomat","broccoli").indexOf("tomat")
res0: Int = 1

scala> Vector("gurka","tomat","broccoli").indexWhere(_.contains("o"))
res1: Int = 1

scala> Vector("gurka","tomat","broccoli").find(_.contains("o"))
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

Sökning

- **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:

```
scala> Vector("gurka","tomat","broccoli").indexOf("tomat")
res0: Int = 1

scala> Vector("gurka","tomat","broccoli").indexWhere(_.contains("o"))
res1: Int = 1

scala> Vector("gurka","tomat","broccoli").find(_.contains("o"))
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: hitta index för elementet x

Implementera `indexOf`:

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

Utdata: index `i` där `xs(i) == x`
Om värde saknas. returnera `-1`

Linjärsökning: hitta index för elementet x

Implementera `indexOf`:

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

Utdata: index `i` där `xs(i) == x`
Om värde saknas. returnera `-1`

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

(Är du nyfiken på binärsökning, se kapitel 12: Valfri fördjupning.)

Sortering

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

Sortering

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.

- 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.

Det finns många olika sorteringsalgoritmer

- Visualisering av 15 olika sorteringsalgoritmer på 6 min:
  `https://www.youtube.com/watch?v=kPRA0W1kECg`
- Olika sorteringsalgoritmer har olika tids- & minneskomplexitet: i bästa fall, i värsta fall, i medeltal, för nästan sorterad, etc.
  `https://en.wikipedia.org/wiki/Sorting_algorithm`
- Olika sorteringsalgoritmer lämpar sig olika väl för parallellisering på många kärnor.

Bogo sort

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

När blir denna färdig?

Antal jämförelser i medeltal vid n element: $n \cdot n!$

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

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 */ do
     /* linjärsök rätt position i sorted */
     /* sätt in element på rätt plats i sorted */
  end for
  sorted.toVector
```

Sortera till ny vektor med insättningssortering: implementation

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

Sorter till ny samling 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 do  // insertion sort using lt as "less than"
     var pos = 0
     while pos < sorted.length && lt(sorted(pos), elem) do
       pos += 1
     end while
     sorted.insert(pos, elem)
  end for
  sorted.toVector
end sortWith
```

Sorter till ny samling 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 do  // insertion sort using lt as "less than"
     var pos = 0
     while pos < sorted.length && lt(sorted(pos), elem) do
       pos += 1
     end while
     sorted.insert(pos, elem)
  end for
  sorted.toVector
end sortWith
```

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

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 do  //från ANDRA till sista
    var j = i
    while j > 0 && xs(j - 1) > xs(j) do
      /* 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 do  //från ANDRA till sista
    var j = i
    while j > 0 && xs(j - 1) > xs(j) do
      /* 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!

Insättningssortering på plats – implementation

```
def insertionSortInPlaceSwap(xs: Array[Int]): Unit =
  def swap(i: Int, j: Int): Unit =
    val temp = xs(i)
    xs(i) = xs(j)
    xs(j) = temp
  end swap

  for i <- 1 until xs.length do  //från ANDRA till sista
    var j = i
    while j > 0 && xs(j - 1) > xs(j) do
      swap(j, j - 1)
      j -= 1;  // stega bakåt
    end while
  end for
end insertionSortInPlaceSwap
```

Uppgifter denna vecka

Denna veckas övning: `sequences`

- Kunna läsa och skriva pseudokod för sekvensalgoritmer och implementera sekvensalgoritmer enligt pseudokod.
- Kunna implementera sekvensalgoritmer, både genom kopiering till ny sekvens och genom förändring på plats i befintlig sekvens.
- Kunna använda inbyggda metoder för uppdatering av, linjärsökning i, och sortering av sekvenssamlingar.
- Kunna beskriva skillnaden i användningen av föränderliga och oföränderliga sekvenser, speciellt vid uppdatering.
- Förstå hur sorteringsordningen är definierad för strängar.
- Kunna sortera sekvenssamlingar innehållande objekt av grundtyper med hjälp av inbyggda och egendefinierade sorteringsordningar med metoderna `sorted`, `sortBy` och `sortWith`.
- Kunna implementera linjärsökning enligt olika sökkriterier.
- Kunna beskriva egenskaperna hos sekvenssamlingarna `Vector`, `List`, `Array`, `ArrayBuffer` och `ListBuffer`.
- Förstå bieffekter av uppdatering av delade referenser till föränderliga element.
- Kunna använda funktioner med repeterade parametrar.
- Känna till hur man implementerar funktioner med repeterade parametrar.
- Kunna implementera heltalsregistrering i en heltalsarray.

Denna veckas laboration: `shuffle`

- Kunna skapa och använda sekvenssamlingar.
- Kunna implementera sekvensalgoritmen SHUFFLE som modifierar innehållet i en array på plats.
- Kunna registrera antalet förekomster av olika värden i en sekvens.

Kontrollskrivning

Kontrollskrivning

Ta med **legitimation** och **snabbreferens**, blyertspenna, suddgummi, **röd** penna till rättningen, förtäring. Ingen mobil. Jackor och väskor vid väggen.

- **Diagnostisk**: vad har du lärt dig hittills?
- **Kamraträttad**: träna på att läsa och bedöma kod
- **Obligatorisk**: sjukdom **måste** meddelas i förväg
- Kan ge **samarbetsbonus om man har visat samarbetskontrakt**

  Max 5p i samarbetsbonus på **första ordinarie tentamen**, som adderas i slutet av kursen till din tentamenspoäng (max 100) och baseras på **medelvärdet** av resultaten på kontrollskrivningen i din samarbetsgrupp (se kap ”Anvisningar” i kompendiet).
- Detta är ingen ”riktig” tenta och du ska inte anmäla dig i LTH:s tentaanmälningssystem.

OBS! Du ska gå på kontrollskrivningen **även** om du inte har alla labbar godkända. Om du har mer än en icke godkänd labb efter dig så kontakta `mailto:bjorn.regnell@cs.lth.se`

Kontrollskriving – upplägg

Pågår i 5 timmar, se Timeedit för tid och plats.

- **Moment 0**: Genomgång, legitimationskontroll, utdelning
- **Moment 1**: ca 2 h 15 min: Du löser uppgifterna individuellt
- **Moment 2**: ca 1 h 15 min: Parvis kamratbedömning
- **Moment 3**: ca 30 min: Inspektera din egen skrivning
- **Moment 4**: Insamling, avslutning

- KOM I GOD TID: sitt redo vid anvisad plats en kvart **före** hel timme
- Läs **noga** igenom instruktionerna på gammal kontrollskrivning här: `https://cs.lth.se/pgk/examination/`

Plugga själv OCH i din samarbetsgrupp

- Träffas och prata i din samarbetsgrupp om hur ni bäst pluggar **individuellt** och **tillsammans** inför kontrollskrivningen för att maximera lärandet i gruppen!
- Repetera teorin i läsperiod 1.
- Repetera lösningar till övningarna och labbarna.
- Träna på att koda med papper och penna!
- Använd tidigare års kontrollskrivningar: `http://cs.lth.se/pgk/examination/`

Lycka till på kontrollskrivningen!