More Functions on Lists
Sublists and element access:

- `xs.length`: The number of elements of `xs`.
- `xs.last`: The list’s last element, exception if `xs` is empty.
- `xs.init`: A list consisting of all elements of `xs` except the last one, exception if `xs` is empty.
- `xs take n`: A list consisting of the first `n` elements of `xs`, or `xs` itself if it is shorter than `n`.
- `xs drop n`: The rest of the collection after taking `n` elements.
- `xs(n)` (or, written out, `xs apply n`). The element of `xs` at index `n`. 
Creating new lists:

- `xs ++ ys` The list consisting of all elements of `xs` followed by all elements of `ys`.
- `xs.reverse` The list containing the elements of `xs` in reversed order.
- `xs updated (n, x)` The list containing the same elements as `xs`, except at index `n` where it contains `x`.

Finding elements:

- `xs indexOf x` The index of the first element in `xs` equal to `x`, or `-1` if `x` does not appear in `xs`.
- `xs contains x` same as `xs indexOf x >= 0`
Implementation of `last`

The complexity of `head` is (small) constant time.

What is the complexity of `last`?

To find out, let’s write a possible implementation of `last` as a stand-alone function.

```python
def last[T](xs: List[T]): T = xs match {
  case List() => throw new Error("last of empty list")
  case List(x) =>
  case y :: ys =>
}
```
Implementation of \texttt{last}

The complexity of \texttt{head} is (small) constant time.

What is the complexity of \texttt{last}?

To find out, let’s write a possible implementation of \texttt{last} as a stand-alone function.

\begin{verbatim}
def last[T](xs: List[T]): T = xs match {
  case List() => throw new Error("last of empty list")
  case List(x) => x
  case y :: ys =>
}
\end{verbatim}
Implementation of last

The complexity of head is (small) constant time.

What is the complexity of last?

To find out, let’s write a possible implementation of last as a stand-alone function.

```python
def last[T](xs: List[T]): T = xs match {
    case List() => throw new Error("last of empty list")
    case List(x) => x
    case y :: ys => last(ys)
}
```
The complexity of head is (small) constant time.

What is the complexity of last?

To find out, let’s write a possible implementation of last as a stand-alone function.

```python
def last[T](xs: List[T]): T = xs match {
    case List() => throw new Error("last of empty list")
    case List(x) => x
    case y :: ys => last(ys)
}
```

So, last takes steps proportional to the length of the list xs.
Implement \texttt{init} as an external function, analogous to \texttt{last}.

```python
def init[T](xs: List[T]): List[T] = xs match {
    case List() => throw new Error("init of empty list")
    case List(x) => ???
    case y :: ys => ???
}
```
Exercise

Implement `init` as an external function, analogous to `last`.

```python
def init[T](xs: List[T]): List[T] = xs match {
  case List() => throw new Error("init of empty list")
  case List(x) =>
    case y :: ys =>

```
How can concatenation be implemented?

Let’s try by writing a stand-alone function:

```python
def concat[T](xs: List[T], ys: List[T]) =
```
How can concatenation be implemented?

Let’s try by writing a stand-alone function:

```python
def concat[T](xs: List[T], ys: List[T]) = xs match {
  case List() =>
  case z :: zs =>
}```
Implementation of Concatenation

How can concatenation be implemented?

Let’s try by writing a stand-alone function:

```python
def concat[T](xs: List[T], ys: List[T]) = xs match {
  case List() => ys
  case z :: zs =>
```
Implementation of Concatenation

How can concatenation be implemented?

Let’s try by writing a stand-alone function:

```python
def concat[T](xs: List[T], ys: List[T]) = xs match {
case List() => ys
case z :: zs => z :: concat(zs, ys)
}
```
Implementation of Concatenation

How can concatenation be implemented?

Let’s try by writing a stand-alone function:

```python
def concat[T](xs: List[T], ys: List[T]) = xs match {
  case List() => ys
  case z :: zs => z :: concat(zs, ys)
}
```

What is the complexity of `concat`?
Implementation of reverse

How can reverse be implemented?

Let’s try by writing a stand-alone function:

```python
def reverse[T](xs: List[T]): List[T] = xs match {
  case List() =>
  case y :: ys =>
}
```
Implementation of reverse

How can reverse be implemented?

Let's try by writing a stand-alone function:

```python
def reverse[T](xs: List[T]): List[T] = xs match {
    case List() => List()
    case y :: ys =>
}
```
Implementation of reverse

How can reverse be implemented?

Let’s try by writing a stand-alone function:

```python
def reverse[T](xs: List[T]): List[T] = xs match {
  case List() => List()
  case y :: ys => reverse(ys) ++ List(y)
}
```
How can reverse be implemented?

Let’s try by writing a stand-alone function:

```python
def reverse[T](xs: List[T]): List[T] = xs match {
  case List() => List()
  case y :: ys => reverse(ys) ++ List(y)
}
```

What is the complexity of reverse?

*Can we do better?* (to be solved later).
Remove the n’th element of a list xs. If n is out of bounds, return xs itself.

```python
def removeAt[T](n: Int, xs: List[T]) = ???
```

Usage example:

```python
removeAt(1, List('a', 'b', 'c', 'd')) > List(a, c, d)
```
Flatten a list structure:

```python
def flatten(xs: List[Any]): List[Any] = ???

flatten(List(List(1, 1), 2, List(3, List(5, 8))))
  > res0: List[Any] = List(1, 1, 2, 3, 5, 8)
```