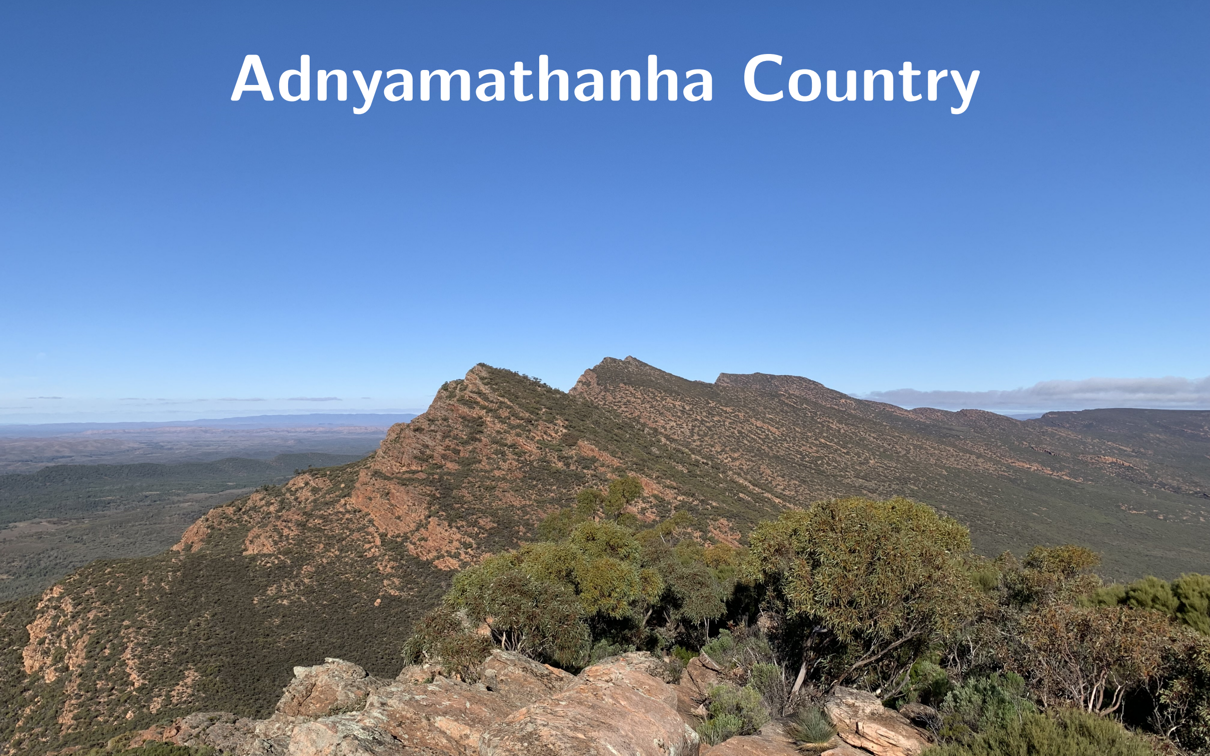
Adnyamathanha Country



We acknowledge, celebrate and pay our respects to the Ngunnawal and Ngambri people of the Canberra region and to all First Nations Australians on whose traditional lands we meet and work, and whose cultures are among the oldest continuing cultures in human history.

Learn more about Acknowledgement of Country here

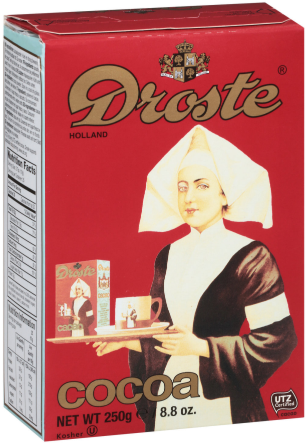
In the photo: Adnyamathanha Country in South Australia

Find out more about Canberra's Aboriginal history here

Recursion

Contents

The Droste effect



The Droste effect

More examples: here

Properties of recursion (informally)

The **size** of each instance

- may be the same
- may change

What about **termination**?

- The chain of instances may go on forever (**unbounded recursion**)
- The chain of instances may terminate

What is (often) desirable in programming?

Why is recursion important?

- Recursive programs can perform actions **repeatedly**
  - action may slightly each time
  - computers are great at repeated actions!
- Recursion is fundamental in **functional programming**
  - in functional programming there is no iteration
- Recursive programs are often **shorter** and **easier** to understand
- Recursion can be used to **traverse** (or navigate) complex data structures
  - lists, trees, . . .
- **Program verification** is easier (using induction)

Solving a problem with recursion (Thompson: §4.4, §4.5)

1. **Base case:** Identify the smallest version(s) of the problem
   - Solve the problem on a very small piece of data (like the empty list)
   - Write code that solves the problem **without recursion**
   - This avoids infinite repetition
2. **Step case:** Solve the problem using a smaller version of the problem
   - This involves a **recursive call** to a smaller version of the problem
   - Here "smaller" means closer to a base case
3. **Distinguish:** between the base and step cases using `case` or guards

Creating good recursive functions

A **base case** is a defining value for which the function is evaluated without recursively calling itself

In a good recursive function

- There must be a base case (or several base cases)
- Each recursive call must lead towards a base case

Multiplication via addition

(THOMPSON: §4.4, §4.5)

Multiplication (with positive integers) is repeated addition

$$\begin{aligned}
1 \times x &= x && && \textbf{base case}\\
2 \times x &= x + x &&= x + (1 \times x) &&\\
3 \times x &= x + x + x &&= x + (2 \times x) &&\\
4 \times x &= x + x + x + x &&= x + (3 \times x) &&\\
5 \times x &= x + x + x + x + x &&= x + (4 \times x) &&\\
&\vdots && &&\\
n \times x &= \underbrace{x + \cdots + x}_{n \text{ times}} &&= x + ((n-1) \times x) && \textbf{step case}
\end{aligned}$$

Multiplication via addition

(Thompson: §4.4, §4.5)

$$1 \times x = x$$ **base case**

$$2 \times x = x + x \qquad = x + (1 \times x)$$

$$\vdots$$

$$n \times x = x + \cdots + x = x + ((n-1) \times x)$$ **step case**

In Haskell

```
multiply :: Int -> Int -> Int
multiply n x = case n of
  1 -> x
  m -> x + multiply (m - 1) x
```

Remark This does not work for 0 or negative numbers. How can it be fixed?

More examples of recursion

(Thompson: §4.4, §4.5)

Compute the **factorial** of an integer

```
fact :: Int -> Int
fact n = case n of
  0 -> 1                                    -- base case
  x -> x * fact (x - 1)                     -- step case
```

Compute the **remainder** of an integer modulo another integer

```
fact :: Int -> Int -> Int
remainder n m = case (n - m) => 0 of
  True  -> remainder (n - m) m              -- base case
  False -> n                                -- step case
```

Remark Both functions do not work for negative inputs. How can we fix that?

More examples of recursion

(Thompson: §4.4, §4.5)

Take the first *n* letters from a word

```
myTake :: Int -> String -> String
myTake n word = case word of
  [] -> []
  x:xs
    | n <= 0      -> []
    | otherwise -> x : myTake (n-1) xs
```

Many more examples can be found on learnyouahaskell.com

Mutual recursion

(Thompson: §4.4, §4.5)

```haskell
isEven :: Integer -> Bool
isEven n
  | n == 0    = True
  | otherwise = isOdd (n-1)

isOdd :: Integer -> Bool
isOdd n
  | n == 0    = False
  | otherwise = isEven (n-1)
```

Remark These functions loop **forever** on negative inputs! How can they be fixed?

Summary

(Thompson: §4.4, §4.5)

Solving a problem with recursion

1. **Base case:** Identify the smallest version(s) of the problem
   - Solve the problem on a very small piece of data (like the empty list)
   - Write code that solves the problem **without recursion**
   - This avoids infinite repetition
2. **Step case:** Solve the problem using a smaller version of the problem
   - This involves a **recursive call** to a smaller version of the problem
   - Here "smaller" means closer to a base case
3. **Distinguish:** between the base and step cases using `case` or guards

Next weeks

Labs

- Week 4: Codeworld (useful for Assignment 1!)
- Week 5: Recursion and lists

Lectures

- Week 4: recursion with lists