# 1 Introduction

## 1.1 Basic Concepts in Computer Science

Computers are extremely complex, so we need different levels of abstraction. We can split computers into levels for simplicity, resulting in the following challenges:

- what services to provide at each level
- hot to implement them using lover-level services
- the interface between the current level and the bottom/top levels
- there should not be any access across multiple layers, allowing each layer to be altered separately

## 1.2 Goals of Programming

- to describe a computation so it can be done mechanically:
  - expressions compute values
  - commands cause effects
- to do so efficiently and correctly (complexity etc.)
- to allow easy modification as needs change
  - by structuring the code neatly
  - e.g. modules and classes to package various functions into easy-to-use abstractions

Programming in the small - writing code to do simple tasks, e.g. adding numbers or sorting lists
programming in the large - using multiple modules to solve a messy task, e.g. using Google Vision and Android Studio packages to create a mobile application

One example of how to organize modules is object-oriented programming.

### 1.3 Why OCaml?

Using OCaML - ignore underlying system and focus on core concepts

- Values are immutable
- functions can't have side effects

## 2 Recursion and Efficiency

### 2.1 Expression Evaluation

In order to evaluate expressions, we may reduce expressions until they become a value. We write $E \to E'$ to say that $E$ is reduced to $E'$. When we write code that deals with expressions only, it's termed as functional programming.

### 2.2 Recursion Requirements

To be useful:

- Must have terminating condition
- Must reach the terminating condition

### 2.3 Summing Integers

```
let rec nsum n =
  if n = 0 then
    0
  else
    n + nsum (n - 1)
```

The function call is as such:

$$\begin{aligned}\text{nsum } 3 &\Rightarrow 3 + (\text{nsum } 2)\\ &\Rightarrow 3 + (2 + (\text{nsum } 1))\\ &\Rightarrow 3 + (2 + (1 + \text{nsum } 0))\\ &\Rightarrow 3 + (2 + (1 + 0))\end{aligned}$$

The parentheses is a problem because the numbers are collected in a stack and are not evaluated until the innermost function is evaluated. This might lead to stack overflows for large `n`.

## 2.4 What is a Stack?

- A stack is a last-in-first-out data-structure
- We use a 'call stack' to keep track of the functions running at the moment
  - Each element includes the function call, as well as the variables in the expressions that haven't been evaluated (which can only be evaluated when the later recursive functions return a value)

**Add diagram to show!!**

## 2.5 Tail Recursion

- Idea is to evaluate the sum at each function call so there is no need to store any function calls with leftover variables to evaluate more expressions
- Only works when there is tail recursion optimisation in the compiler

```
let rec nsum n =
  if n = 0 then
    0
  else
    n + nsum (n - 1)
(*stack overflow for big values of n!*)

let rec summing n total =
  if n = 0 then
    total
  else
    summing (n - 1) (n + total)
(*no stack overflow!*)
```

However some languages, like Java, have compilers that can't recognise tail-recursive functions so either way would lead to a lot of memory stored in the stack.

This is the reason why iterative functions are recommended rather recursive functions for dynamic programming!

However, if memory is not a problem or the function has few recursive function calls then it is not worth the effort.

## 2.6 Complexity and Big O

### 2.6.1 Motivation

Compare code to see which is better.

- Faster code
- Lower memory

### 2.6.2 Calculating Complexities

There are a thousand and one resources online, the main method is with recurrence relations, and if the function has random variables, then use expectation values.

## 3 Lists

Things to note:

- all elements in a list have the same type
- lists are ordered

## 3.1 List Methods

```
(*concatenation*)
x @ [2; 10]
List.append x [2; 10]
(*reverse*)
List.rev [(1, "one"); (2, "two")]
```

Time Complexity: $\mathcal{O}(n)$
Space Complexity: $\mathcal{O}(n)$ (`append` stores a call stack and evaluates the expression only when the last function call returns a value)

### 3.2 Strings

In a few programming languages, strings simply are lists of characters. In OCaml they are a separate type, unrelated to lists, reflecting the fact that strings are an abstract concept in themselves.

#### 3.2.1 String Methods

```
String.length "abc"
"abc" ^ "def"  (* concatenate two strings *)
```

## 4 More Lists

### 4.1 List Utilities

```
let rec take i = function
  | [] -> []
  | x::xs ->
      if i > 0 then x :: take (i - 1) xs
      else []

let rec drop i = function
  | [] -> []
  | x::xs ->
      if i > 0 then drop (i-1) xs
      else x::xs
```

Function `take` is not iterative, but making it so would not improve its efficiency as the task requires copying up to i list elements, which must take $\mathcal{O}(n)$ space.

Function `drop` skips over $i$ list elements. This requires $\mathcal{O}(i)$ but constant space. It is iterative and much faster than `take`, because `drop`'s constant factor is smaller.

### 4.2 Linear Search

- Linear Scan: $\mathcal{O}(n)$
- Ordered Searching: $\mathcal{O}(\log n)$
- Indexed Lookup: $\mathcal{O}(1)$

### 4.3 Equality Tests

```
let rec member x = function
 | [] -> false
 | y::l ->
    if x = y then true
    else member x l
```

This function uses polymorphic equalities, i.e. `<`, `>`, `=` etc. work for multiple types of values. However, for more complex types, these compare operators will not work.

For simpler projects, where every use of these equalities is clear, it is okay to use them. But for larger codebases, it becomes dangerous.

### 4.4 Building and Unbuilding a List of Pairs

```
let rec zip xs ys =
  match xs, ys with
  | (x::xs, y::ys) -> (x, y) :: zip xs ys
  | _ -> []

let rec unzip = function
 | [] -> ([], [])
 | (x, y)::pairs ->
     let xs, ys = unzip pairs in
     (x::xs, y::ys)
```

The wildcard pattern, _, matches anything. Hence, we have to put it as the last pattern to test with, because patterns are tested in order of their definitions.

The iterative version of `unzip` would result in reversed lists.

```
let rec revUnzip = function
  | ([], xs, ys) -> (xs, ys)
  | ((x, y)::pairs, xs, ys) ->
     revUnzip (pairs, x::xs, y::ys)
```

We can write the unzip function in a cleaner way by having a helping function.

```
let conspair ((x, y), (xs, ys)) = (x::xs, y::ys)

let rec unzip = function
  | [] -> ([], [])
  | xy :: pairs -> conspair (xy, unzip pairs)
```

# 5 Sorting

## 5.1 Lower Bound

Height of binary tree of comparisons, $\mathcal{O}(\log(n!))$

## 5.2 Insertion Sort

```
let rec ins x = function
  | [] -> [x]
  | y::ys -> if x <= y then x :: y :: ys
             else y :: ins x ys

let rec insort = function
    | [] -> []
    | x::xs -> ins x (insort xs)
```

- Average time complexity: $\mathcal{O}(n^2)$ because average number of comparisons is $n/2$.
- Worst case time complexity:$\mathcal{O}(n^2)$

## 5.3 QuickSort

```
let rec quick = function
  | [] -> []
  | [x] -> [x]
  | a::bs ->
      let rec part l r = function
        | [] -> (quick l) @ (a :: quick r)
        | x::xs ->
            if (x <= a) then
              part (x::l) r xs
            else
              part l (x::r) xs
      in
      part [] [] bs
```

To remove the append, we can use:

```
let rec quik = function
  | ([], sorted) -> sorted
  | ([x], sorted) -> x::sorted
  | a::bs, sorted ->
     let rec part = function
       | l, r, [] -> quik (l, a :: quik (r, sorted))
       | l, r, x::xs ->
           if x <= a then
             part (x::l, r, xs)
           else
             part (l, x::r, xs)
     in
     part ([], [], bs)
```

- Average case: $C(n) = 2C(n/2) + n - 1 = \mathcal{O}(n \log(n))$
- Worst case: $C(n) = C(n-1) + C(1) + n - 1 = \mathcal{O}(n^2)$

With append: additional $n/2$ `cons` due to append, leading to a bigger constant.

## 5.4 Mergesort

```
let rec merge = function
  | [], ys -> ys
  | xs, [] -> xs
  | x::xs, y::ys ->
      if x <= y then
        x :: merge (xs, y::ys)
      else
        y :: merge (x::xs, ys)
```

```
let rec tmergesort = function
  | [] -> []
  | [x] -> [x]
  | xs ->
      let k = List.length xs / 2 in
      let l = tmergesort (take k xs) in
      let r = tmergesort (drop k xs) in
      merge (l, r)
```

- Average case = worst case = $\mathcal{O}(n \log n)$

# 6 Datatypes and Trees

## 6.1 Custom Datatypes

```ocaml
type vehicle = Bike
             | Motorbike
             | Car
             | Lorry
```

By defining a custom type, OCaml can type check for us

- Check if functions have exhaustive pattern matching
- Check for type typos

Internally, OCaml might use bits to store which type which is fast and efficient

```ocaml
let wheels = function
| Bike -> 2
| Motorbike -> 2
| Car -> 4
| Lorry -> 18
```

Each constructor can also have arguments.

```ocaml
type vehicle = Bike
             | Motorbike of int  (* engine size in CCs
                *)
             | Car       of bool (* true if a Reliant
                Robin *)
             | Lorry     of int  (* number of wheels *)
```

We can use these arguments for pattern matching.

```ocaml
let wheels = function
| Bike -> 2
| Motorbike _ -> 2
| Car robin -> if robin then 3 else 4
| Lorry w -> w
```

## 6.2 Error Handling

Exceptions are important as they allow us to find out which parts in the program have errors and caused the program to fail.

```
exception Failure
exception NoChange of int

let divide a b =
    if b = 0 then raise Failure
    else a / b

try
  print_endline "pre exception";
  raise (NoChange 1);
  print_endline "post exception";
with
  | NoChange _ ->
      print_endline "handled a NoChange exception"
```

Pattern matching works with exceptions too.

Note that handling an expression means evaluating another expression and returning its value instead.

One criticism of OCaml is that noting in a function declaration indicates which exceptions it might raise. One alternative is to return a value of datatype `option`.

```
let x = Some 1
let y = None
type 'a option = None | Some of 'a

let rec list_max = function
  | []    -> None
  | h::t -> begin
      match list_max t with
        | None    -> Some h
        | Some m -> Some (max h m)
      end
```

This way, every function returns either `None` or `Some` value.

### 6.2.1 Making Change with Exceptions

Because `try...with` can change the flow of execution, we can code backtracking with it.

```
exception Change
let rec change till amt =
match till, amt with
| _, 0          -> []
| [], _         -> raise Change
| c::till, amt -> if amt < 0 then raise Change
                  else try c :: change (c::till) (amt
                     - c)
                    with Change -> change till amt
```

change [5; 2] 6 ⇒ try 5::change [5; 2] 1
with Change -¿ change [2] 6
⇒ try 5::(try 5::change [5; 2] (-4)
with Change -¿ change [2] 1)
with Change -¿ change [2] 6
⇒ 5::(change [2] 1)
with Change -¿ change [2] 6
⇒ try 5::(try 2::change [2] (-1)
with Change -¿ change [] 1)
with Change -¿ change [2] 6
⇒ try 5::(change [] 1)
with Change -¿ change [2] 6
⇒ change [2] 6
⇒ try 2::change [2] 4
with Change -¿ change [] 6
⇒ try 2::(try 2::change [2] 2
with Change -¿ change [] 4)
with Change -¿ change [] 6
⇒ try 2::(try 2::(try 2::change [2] 0
with Change -¿ change [] 2)
with Change -¿ change [] 4)
with Change -¿ change [] 6
⇒ try 2::(try 2::[2]
with Change -¿ change [] 4)
with Change -¿ change [] 6
⇒ try 2::[2; 2]
with Change -¿ change [] 6
⇒ [2; 2; 2]