Foundations of Computer Science

OCaml

- **Expressions** only compute values.
- **Commands** only cause side effects.
- **Functional programming** separates expressions from side effects.

```
(* declarations *)
let variable_name = expression
let function_name arg1 arg2 = expression
let rec recursive_function arg1 = expression

(* type annotation *)
let variable_name: T = expression
let function_name (arg1: T1) (arg2: T2): T3 = expression

(* let expression *)
let name = expression1
  in expression2

(* types and exceptions *)
type 'a option =
  | None
  | Some of 'a

exception Fail
exception OutOfBounds of int

(* if-else expression *)
if boolean_condition1 then expression1
else if boolean_condition2 then expression2
else expression3

(* anonymous functions *)
fun arg -> expression

(* anonymous functions + match *)
function
  | value1 -> expression1
  | value2 -> expression2

(* match expression *)
match expression with
  | pattern1 -> expression1
  | pattern2 -> expression2

(* try-with expression *)
try
  raise ExceptionName
with
  | Fail -> expression1
  | Some x -> expression2

(* tuples *)
x, y, z = (x, y, z)
[x, y, z] = [(x, y, z)]
```

```
int + int
int - int
int * int
int / int

float +. float
float -. float
float *. float
float /. float

bool || bool
bool && bool
not bool

a :: [a] -> [a]
[a] @ [a] -> [a]

fst (a, b) -> a
snd (a, b) -> b

a = a
a > a
a < a
a >= a
a <= a
a <> a
```

Asymptotic Behaviour

Asymptotic complexity refers to how program cost grows with increasing inputs.

$$f(n) = O(g(n)) \text{ if there exists an } n_0 \text{ where } |f(n)| \leq c|g(n)| \text{ for all } n \geq n_0$$

Recurrence relation	Time complexity
$T(n+1) = T(n) + 1$	$O(n)$
$T(n+1) = T(n) + n$	$O(n^2)$
$T(n) = T(n/2) + 1$	$O(\log n)$
$T(n) = T(n/2) + n$	$O(n \log n)$

In a **tail recursive function** the recursive function call is the last step.

Sorting Algorithms

In a **comparison sort** we can only compare two items to see if they are bigger, smaller or equal.

- There are $n!$ permutations of n elements.
- Each comparison eliminates half of the permutation $2^{C(n)} = n!$
- The sort is at best $C(n) \geq \log n! \approx n \log n + 1.44n$

Algorithm | **Code**

Insertion Sort

`ins` inserts an item to a sorted list. — $O(1)$ best case, $O(n)$ average and worst

`insort` — $O(n)$ best case, $O(n^2)$ average and worst

```
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)
```

Quicksort

1. Choose a pivot a
2. Partition into two sublists: those $\leq a$ and $> a$
3. Recursively sort both sublists
4. Append the two lists together

Best and average case when sublists have equal lengths — $O(n \log n)$

Worst case — $O(n^2)$

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

Merge Sort

- Worst time complexity $O(n \log n)$
- Space complexity $O(n \log n)$

```
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
```

```
(* merge on the left panel *)

let rec mergesort = function
  | [] -> []
  | xs ->
    let k = (length xs) / 2
    in let l = take k xs
    in let r = drop k xs
    in merge (mergesort xs)
             (mergesort ys)
```

Data Structures

Polymorphic types have type parameters.

```
type 'a list =
  | Nil
  | Cons of 'a

type 'a tree =
  | Lf
  | Br 'a * 'a tree * 'a tree
```

Dictionaries attach values to keys.

Association Tree

- `lookup` is $O(n)$
- `update` is $O(1)$ and only shadows previous values.

```
type ('k, 'v) dict = ('k * 'v) list

let rec lookup a = function
  | [] -> raise Missing
  | (k, v) :: ps when x = a -> v
  | _ :: ps -> lookup a ps

let rec update l k v = (k, v) :: l
```

Binary Search Tree

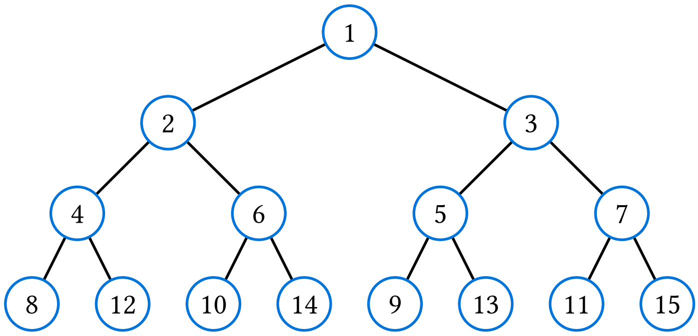
`lookup` and `update` are both $O(\log n)$ when balanced, $O(n)$ when unbalanced.

```
let rec lookup a = function
  | Lf -> raise Missing
  | Br (k, v) l r ->
    if a = k then v
    else if a < k then lookup a l
    else lookup a r

let rec update a b = function
  | Lf -> Br ((a, b), Lf, Lf)
  | Br ((k, v), l, r) ->
    if a = k then Br ((a, b), l, r)
    else if a < k then update a b l
    else update a b r
```

Functional arrays store values in a balanced tree, access time for each element is $O(\log n)$.

- To access node with index 12 (`1100`), read from right to left. Traverse left if `0`, right if `1`.
- Stop when only a 1 digit remains.



A **queue** is a FIFO structure.

$$\text{queue}([x_1; x_2; \ldots; x_n], [y_1; y_2; \ldots; y_m])$$

- `enq` add items to the front of the rear list.
- `deq` remove items from the front of the first list, if it is empty, reverse the rear and move it to front.

For a queue of length n

- n `enq` and n `deq` operations, cost $2n$
- 1 reverse list operation, cost n

So the **amortised time** per operation is $O(1)$

By calling `norm` in each `deq`, the list satisfies the property: if the front list is empty, the tail list is also empty.

```
type 'a queue =
  | Q of 'a list * 'a list

let norm = function
  | Q ([], xs) -> Q (List.rev tls, [])
  | q -> q
```

Tree Traversal Algorithms

The goal of tree traversal is to visit every node.

There's the usual pre-order, in-order and post-order.

- **Pre-order** is useful for copying a tree.
- **Post-order** is useful for destructing a tree.
- All three are special versions of **depth-first-search**.

Algorithm	Code

Breadth-first Traversal
• Uses a queue
• $1 + b + b^2 + \cdots + b^d = O(b^d)$ nodes to examine.
• Stores $O(b^d)$ nodes in memory, not suitable for infinite trees.
	`let rec breadth q =`
	`  if qnull q then []`
	`  else match qhd q with`
	`    | Lf -> breadth (deq q)`
	`    | Br (x, l, r) ->`
	`        x :: breadth (enq (enq (deq q) l) r)`

Depth-first Iterative Deepening 1. Search to depth 1 2. Discard previous search, search to depth 2 3. Repeat until found Takes $\frac{b}{b-1}$ the time of breadth-first search, but only $O(d)$ memory.	**Best-first Search** • Similar to breadth-first search • Uses a priority queue • Items are ranked using a heuristic to approximate distance of a node to the solution.

Lazy Lists

A function is not evaluated until the arguments are provided.

```
type 'a seq =
  | Nil
  | Cons of 'a * (unit -> 'a seq)
```

Create an infinite sequence starting from k

```
let rec from k =
  Cons (k, fun () -> from (k + 1))
```

```
let head = function
  | Cons (x, _) -> x

let tail = function
  | Cons (_, xf) -> xf ()
```

Procedural Programming

```
let createAccount =
  let amount = ref 0
  in fun dv ->
    amount := !amount + dv;
    !amount

let updateAmt = createAccount
updateAmt 0 (* 0 *)
updateAmt 5 (* 5 *)
updateAmt 5 (* 10 *)

while condition do
  commands
done
```

```
ref (* 'a -> 'a ref *)
(!) (* 'a ref -> 'a *)
(:=) (* 'a ref -> 'a -> () *)

let a = [|1; 2; 3|] : int array
a.(1) (* short for Array.get *)
a.(1) <- 123 (* short for Array.set *)

Array.get : 'a array -> int -> 'a
Array.set : 'a array -> int -> 'a -> ()
Array.make : int -> 'a -> 'a array
Array.init : int -> (int -> 'a) -> 'a array
```