Foundations of Computer Science

Tripos Revision Guide

CST Part IA — Michaelmas Term

Complete coverage of OCaml, Recursion & Efficiency, Lists, Sorting, Datatypes & Trees, Dictionaries, Higher-Order Functions, Lazy Lists, Queues & Search, and Procedural Programming

Topic Overview

Lecture	Key Content
1–2: Foundations & Efficiency	Abstraction, expression evaluation, recursion, tail-recursion, O-notation, recurrence relations
3–4: Lists	Primitives, append/reverse costs, take/drop, zip/unzip, making change
5: Sorting	Insertion sort, quicksort, mergesort, comparison lower bound
6: Datatypes & Trees	Variant types, exceptions, binary trees, traversal
7: Dictionaries	Association lists, binary search trees, functional arrays
8: Functions as Values	Currying, partial application, map/filter/fold, λ-calculus
9: Lazy Lists	Sequences, forcing, infinite lists, Newton–Raphson
10: Queues & Search	BFS/DFS, efficient functional queues, iterative deepening
11: Procedural Programming	References, `while`, closures over state, arrays

Tripos coverage: 1995–2025 • Full worked examples • Model answer templates

July 11, 2026

0 Contents

1 Foundations: Abstraction, Recursion & Efficiency

1.1 Levels of Abstraction

Computer science tames complexity through **levels of abstraction**: each level provides services to the one above while hiding how those services are implemented. This *abstraction barrier* means a lower level can be changed (e.g. a faster processor, a new data representation) without breaking code built on top of it. Classic cautionary tales are date/format crises: DEC's PDP-10 used a 12-bit date field good for only 11 years, and the Y2K bug arose from a 2-digit year field – both are failures to leave enough room in a low-level representation.

This course has two aims: teaching **programming** (using OCaml, mostly in the *functional* style) and teaching **fundamental principles** of algorithm design and efficiency analysis.

1.2 Expression Evaluation

OCaml computation is understood as **expression reduction**: $E_0 \to E_1 \to \cdots \to v$, where v is a value that cannot be reduced further. We write $E \to E'$ for "E reduces to E'" – mathematically $E = E'$, but reduction is directional.

1.3 Recursion vs. Iteration

Naive summation – builds up nested additions

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

Calling `nsum 3` produces `3 + (2 + (1 + (0)))`: none of the additions can happen until the base case is reached, so the computer must keep a stack of pending additions. For large `n` this risks **stack overflow**.

Iterative (tail-recursive) version, with accumulator

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

Here the addition happens immediately and is passed along in `total`. A recursive function whose computation does not nest is called **iterative** or **tail-recursive**. The general technique is to add an **accumulating argument**.

Recursion vs Iteration – Key Points

- Tail recursion is only efficient if the *compiler* detects and optimises it.
- The main benefit is **space** (constant stack use), not necessarily time.
- Do not contort code into tail-recursive form unless the gain is worth the loss of clarity – KISS (Keep It Simple, Stupid).
- Many algorithms (e.g. tree recursion) are naturally *not* tail recursive, and that is fine.

1.4 Big-O Notation

Definition

$f(n) = O(g(n))$ iff there exist constants $c > 0$ and n_0 such that $|f(n)| \leq c\,|g(n)|$ for all $n \geq n_0$. Informally: look at the most significant term and ignore constant factors.

Simple Facts (all frequently tripos-tested)

- $O(2g(n)) = O(g(n))$ – constant factors drop out.
- $O(\log_{10} n) = O(\ln n)$ – the base of a logarithm is irrelevant (it's a constant factor).
- $O(n^2 + 50n + 36) = O(n^2)$ – insignificant lower-order terms drop out.
- If $0 < c < d$ then $O(n^c) \subset O(n^d)$, $O(c^n) \subset O(d^n)$, and $O(\log n) \subset O(n^c)$ for any $c > 0$.

Common Complexity Classes

Class	Name
$O(1)$	constant
$O(\log n)$	logarithmic
$O(n)$	linear
$O(n \log n)$	quasi-linear
$O(n^2)$	quadratic
$O(n^3)$	cubic
$O(a^n)$	exponential (fixed a)

Growth-Rate Table (classic tripos-style data)

Complexity	1 second	1 minute	1 hour
n	1 000	60 000	3 600 000
$n \log n$	140	4 895	204 095
n^2	31	244	1 897
n^3	10	39	153
2^n	9	15	21

Polynomial-time algorithms (n^c) scale far better under increased resources than exponential ones – doubling compute time barely moves the needle for 2^n.

1.5 Recurrence Relations: The Standard Toolkit

To analyse a recursive function, read off a recurrence for its cost $T(n)$: constant-cost base case, and constant work plus recursive calls in the step case.

The Four Recurrences Tripos Loves

Recurrence	Complexity
$T(n+1) = T(n) + 1$	$O(n)$
$T(n+1) = T(n) + n$	$O(n^2)$
$T(n) = T(n/2) + 1$	$O(\log n)$
$T(n) = 2T(n/2) + n$	$O(n \log n)$

Worked Example: efficient exponentiation

```
let rec power x n =
  if n = 1 then x
  else if even n then power (x *. x) (n / 2)
  else x *. power (x *. x) (n / 2)
```

Justification: $x^1 = x$, $x^{2n} = (x^2)^n$, $x^{2n+1} = x \cdot (x^2)^n$. Recurrence: $T(n) = T(n/2) + 1 \Rightarrow O(\log n)$ time, versus $O(n)$ for the naive $x^{n+1} = x \cdot x^n$.

TRAP: Space vs Time

Space complexity can never exceed time complexity (it takes time to use space), but a function can be $O(n)$ time and $O(1)$ space (e.g. `summing`) or $O(n)$ time and $O(n)$ space (e.g. `nsum`, due to the pending-additions stack). Always state *both* when asked, and justify from the recursion structure, not just the final formula.

2 Lists

2.1 Core Primitives

A list is built from `[]` (nil) and `::` (cons, $O(1)$ time and space). Key primitives:

```
let null = function [] -> true | x :: l -> false
let hd (x :: l) = x
let tl (x :: l) = l
```

These are **polymorphic**: `null : 'a list -> bool`, `hd : 'a list -> 'a`, `tl : 'a list -> 'a list`.

2.2 Length: Naive vs Iterative

```
(* naive: O(n) time, O(n) space (not tail recursive) *)
let rec nlength = function [] -> 0 | x :: xs -> 1 + nlength xs

(* iterative: O(n) time, O(1) space *)
let rec addlen n = function [] -> n | x :: xs -> addlen (n + 1) xs
let length xs = addlen 0 xs
```

2.3 Append and the Cost of Reversal

```
let rec append xs ys =
  match xs, ys with
  | [], ys -> ys
  | x :: xs, ys -> x :: append xs ys
```

`append xs ys` (i.e. `xs @ ys`) costs $O(|xs|)$: it copies the *first* list only.

TRAP: Naive reversal is quadratic

```
let rec nrev = function [] -> [] | x :: xs -> (nrev xs) @ [x]
```

Each level does an $O(n)$ append, giving $O(n^2)$ overall – $n + (n-1) + \cdots + 1$. This is a classic "spot the inefficiency" tripos trap. Fix with an **accumulator**, the standard technique for eliminating append:

```
let rec rev_app xs ys =
  match xs, ys with
  | [], ys -> ys
  | x :: xs, ys -> rev_app xs (x :: ys)
let rev xs = rev_app xs []
```

This does exactly n conses: $O(n)$ time, tail-recursive so $O(1)$ auxiliary space.

2.4 take and drop

```
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
```

Both are $O(i)$ (they stop early). Used throughout the course to split a list in two, e.g. for mergesort.

2.5 zip / unzip

```
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)
```

`zip` truncates to the shorter list; `unzip` uses a local `let` to destructure the recursive result into a pair of lists.

2.6 Making Change: Greedy, All Solutions, and Backtracking

Greedy change (may fail even when a solution exists)

```
let rec change till amt =
  match till, amt with
  | _, 0 -> []
  | [], _ -> raise (Failure "no more coins!")
  | c :: till, amt ->
      if amt < c then change till amt
      else c :: change (c :: till) (amt - c)
```

Always uses the largest available coin – fails on cases like "6 using {5,2}" where a smaller coin must be chosen. Returning *all* possible solutions (a list of lists) avoids this at the cost of exploring the whole search space; a **backtracking** version using exceptions is often cleaner:

```
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
```

The recursive call is wrapped in a `try`; on failure it undoes the most recent coin choice and tries the next-smaller coin. This pattern – *try the greedy/recursive choice, catch and backtrack on failure* – recurs constantly in tripos questions about search and constraint satisfaction.

Tripos Pattern (1995 Paper 1 Q3): set operations on lists

Classic early-paper question, still instructive: given lists representing *sets* (no repeats), implement `intersect`, `subtract`, `union`, each preserving (a) no repeats, (b) $O(|\text{result}|)$ conses, (c) order taken from the first argument.

```
let rec intersect xs ys =
  match xs with
  | [] -> []
  | x :: xs -> if member x ys then x :: intersect xs ys
               else intersect xs ys

let rec subtract xs ys =
  match xs with
  | [] -> []
  | x :: xs -> if member x ys then subtract xs ys
               else x :: subtract xs ys

let union xs ys = xs @ subtract ys xs
```

Note `union` must not simply be `xs @ ys`, since that could duplicate elements common to both; using `subtract` on the second list removes duplicates while preserving condition (d) that `xs`'s elements come first, in order. The most general type is `union : 'a list -> 'a list -> 'a list` (requires only equality, from `member`).

3 Sorting Algorithms

3.1 The Comparison Lower Bound

Why $n \log n$ is optimal for comparison sorts

There are $n!$ permutations; each comparison can (at best) halve the remaining candidates, so $2^{C(n)} \geq n!$, giving

$$C(n) \geq \log(n!) \approx n \log n - 1.44n.$$

Mergesort achieves this bound; no comparison-based algorithm can do asymptotically better.

3.2 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)
```

$O(n^2)$ comparisons on average (each `ins` does $O(n)$ work into an already-sorted output). Simple, but too slow to be practical – valuable mainly as a stepping stone to mergesort/heapsort.

3.3 Quicksort

Divide and conquer with pivot

```
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
```

Choose pivot `a`; partition into $\leq a$ and $> a$; sort each half recursively; combine by appending. **Average case**: $T(n) = 2T(n/2) + n \Rightarrow O(n \log n)$. **Worst case** (already sorted / reverse sorted, unbalanced pivot): $T(n+1) = T(n) + n \Rightarrow O(n^2)$.

TRAP: Quicksort's worst case

Never claim quicksort is $O(n \log n)$ unconditionally – always qualify "on average" / "for random data" and mention the $O(n^2)$ worst case triggered by (nearly) sorted input with a poorly-chosen (e.g. first-element) pivot.

An **append-free** version avoids the `@` in the combine step by threading an accumulator `sorted`, the standard technique also used for reversal and tree traversal.

3.4 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)
```

`merge` does at most $m+n-1$ comparisons. Because `take`/`drop` always split evenly, $T(n) = 2T(n/2)+n$ holds *even in the worst case*, giving guaranteed $O(n \log n)$ – mergesort's key advantage over quicksort. In practice quicksort is often faster (better constants, no auxiliary merge step), so "which is best?" depends on the application and on whether worst-case guarantees matter.

Summary Table

Algorithm	Average	Worst case
Insertion sort	$O(n^2)$	$O(n^2)$
Quicksort	$O(n \log n)$	$O(n^2)$
Mergesort	$O(n \log n)$	$O(n \log n)$

Tripos Pattern (2025 P1 Q1): debugging a mergesort

Given a buggy mergesort (wrong `length`/`split` clause order, missing base cases matched *after* the catch-all `_` pattern, etc.), the examiner wants you to (i) spot that OCaml pattern clauses are tried *in order* so a catch-all `_` before `[]`/`[_]` makes those cases unreachable, (ii) check that `split` conses onto the correct accumulator (`h::l1`, not `l1::h`, which is a type error / wrong structure), and (iii) verify base cases return sorted trivially. This "spot five bugs" style question tests close reading of pattern-match *order* and structural correctness rather than new algorithmic content – always trace a tiny example (e.g. `[3;1;2]`) by hand to check your fixed version.

Tripos Pattern (2023 P1 Q1): frequency sort via run-length encoding

Given `sort` (a generic comparator-based sort) and functions `rle_encode`/`rle_decode` converting between a sorted list and (value, count) pairs, `freq_sort` sorts elements by *ascending frequency of occurrence*, tie-broken by value. The recipe: sort the input, RLE-encode it (now grouped by value with counts), sort the RLE list *by count* using the same generic `sort` with a comparator that projects out the count, then RLE-decode the result. This is a good example of combining several small, general functions (`sort`, `rle_encode`, `rle_decode`) rather than writing one monolithic function – a style the examiners reward.

4 Datatypes, Exceptions & Trees

4.1 Variant Types

```
type vehicle = Bike | Motorbike of int | Car of bool | Lorry of int
let wheels = function
  | Bike -> 2
  | Motorbike _ -> 2
  | Car robin -> if robin then 3 else 4
  | Lorry w -> w
```

Constructors with arguments (`Motorbike of int`) create *distinct* values – `Motorbike 450` is a `vehicle`, not an `int`. A single list can hold a mix of variant shapes, e.g. `[Bike; Car true; Motorbike 450]`, sidestepping OCaml's usual same-type-list restriction. Pattern matching on constructors is efficient and exhaustiveness-checked by the compiler – a major source of tripos marks is stating *why* datatypes beat encoding variants as raw integers or strings (type safety, compiler warnings for missing cases, no silent misspellings).

4.2 Exceptions

Exception Mechanics

- `exception Foo` / `exception Bar of int` declare new constructors of the built-in `exn` type.
- `raise E` abandons the current computation and jumps to the closest *dynamically enclosing* handler that matches.
- `try E1 with p1 -> E2 | p2 -> E3` evaluates `E1`; if it raises a matching exception, evaluates the corresponding branch instead.
- Handler matching is dynamic (found at run time), unlike identifier binding which is resolved statically.

Exceptions vs. `option`

`'a option = None | Some of 'a` is an alternative to exceptions for signalling failure. Trade-off: exceptions let error handling be written far from the point of failure and can carry arbitrary payload data, but nothing in a function's *type* indicates what it might raise (unlike Java's checked exceptions); `option` is explicit in the type but forces callers to pattern-match everywhere.

Tripos Pattern (2011 P1 Q2): reasoning about exceptions without using them

Given `cannot f x` (returns `true` iff evaluating `f x` raises a specific exception `Olive`), and functions built from it, a classic part asks you to re-implement an exception-using function *without* exceptions, typically by restructuring the recursive traversal so failure is detected structurally (e.g. checking membership before descending) rather than via control flow. Always justify equivalence by induction on the datatype (e.g. tree shape).

4.3 Binary Trees

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

let rec count = function
  | Lf -> 0
  | Br (v, t1, t2) -> 1 + count t1 + count t2

let rec depth = function
  | Lf -> 0
  | Br (v, t1, t2) -> 1 + max (depth t1) (depth t2)
```

Key invariant: $\text{count}(t) \leq 2^{\text{depth}(t)} - 1$, and $\text{leaves}(t) = \text{count}(t) + 1$ (provable by structural induction). A tree of depth 20 can hold up to $2^{20} - 1 \approx 10^6$ elements with only 20-step access paths – the core motivation for using trees over lists for large dictionaries.

4.4 Tree Traversal

Depth-first traversals (quadratic due to @)

```
let rec inorder = function
  | Lf -> []
  | Br (v, t1, t2) -> inorder t1 @ [v] @ inorder t2
```

`preorder` visits label-first (Polish notation); `inorder` visits label-between (gives a sorted list for a BST); `postorder` visits label-last (Reverse Polish). As written these are $O(n^2)$ worst case because of repeated `@`.

TRAP: quadratic traversal, and the accumulator fix

```
let rec inord = function
  | Lf, vs -> vs
  | Br (v, t1, t2), vs -> inord (t1, v :: inord (t2, vs))
```

Passing an accumulator `vs` (the "rest of the list so far") eliminates `@` entirely, giving $O(n)$: exactly the same trick used for list reversal and append-free quicksort. Expect a tripos part asking you to prove `inord(t, vs) = inorder(t) @ vs` and hence derive the linear-time bound.

5 Dictionaries: Association Lists, BSTs & Functional Arrays

5.1 The Dictionary Interface

An abstract dictionary supports `lookup` (retrieve), `update` (insert/replace), and (harder) `delete`, hiding its internal representation.

5.2 Association Lists – Simplest, Slowest

```
exception Missing
let rec lookup a = function
  | [] -> raise Missing
  | (x, y) :: pairs -> if a = x then y else lookup a pairs

let update (l, b, y) = (b, y) :: l
```

`update` is $O(1)$ (just cons) but never removes stale entries, so lookup is $O(n)$ and space grows unboundedly with the number of updates, not distinct keys. Works for *any* key type with equality (no ordering needed) – most general, least efficient.

5.3 Binary Search Trees

```
let rec lookup b = function
  | Br ((a, x), t1, t2) ->
      if b < a then lookup b t1
      else if a < b then lookup b t2
      else x
  | Lf -> raise (Missing b)

let rec update k v = function
  | Lf -> Br ((k, v), Lf, Lf)
  | Br ((a, x), t1, t2) ->
      if k < a then Br ((a, x), update k v t1, t2)
      else if a < k then Br ((a, x), t1, update k v t2)
      else Br ((a, v), t1, t2)
```

Both are $O(\log n)$ *if the tree is balanced* – $O(n)$ worst case (e.g. inserting sorted data builds a degenerate list-shaped tree, closely analogous to quicksort's worst case). `update` rebuilds only the nodes on the path to the target, *sharing* the untouched subtree: this is the essence of persistent functional data structures.

inorder gives sorted output

Running `inorder` on a BST yields the (key,value) pairs in sorted key order – the basis of *treesort* (build a BST then read it off inorder) and of merging two BSTs (inorder both, merge the sorted lists, rebuild).

Tripos Pattern (2012 P1 Q1): union / slice / drop-slice of BSTs

- `union t1 t2` (disjoint keys): insert every node of `t2` into `t1` via `update`, or symmetrically fold `Br` nodes together directly.
- `takeSlice t (x,y)`: keep only nodes with $x \leq$ key $\leq y$ – recurse, pruning subtrees that provably lie entirely outside the range (using the BST invariant to avoid visiting them).
- `dropSlice`: complement of `takeSlice`; hint says to reduce it to node deletion (find the node, splice out, e.g. by promoting the leftmost node of the right subtree).
- Part (d) asks why $t \neq \text{union}(\text{takeSlice}(t,x,y), \text{dropSlice}(t,x,y))$ in general: the two operations can rebalance/restructure the tree differently (e.g. different tree *shape* even though the same *key set* is represented) – a dictionary is a set of bindings, not a canonical tree shape.

5.4 Functional Arrays as Binary Trees

Arrays require only that keys be *integers*; in return, even worst-case access can be made $O(\log n)$ using Braun trees: element i is found by following the binary digits of i.

```
exception Subscript
let rec sub = function
  | Lf, _ -> raise Subscript
  | Br (v, t1, t2), k ->
      if k = 1 then v
      else if k mod 2 = 0 then sub (t1, k / 2)
      else sub (t2, k / 2)
```

`update` is analogous, replacing the target node (or extending a leaf into a branch to grow the array by one, raising `Subscript` if that would leave a gap). This structure is always balanced by construction, guaranteeing $O(\log n)$ even in the worst case – strictly better than BSTs, at the cost of needing integer keys.

Generality vs Efficiency Trade-off (a favourite essay-style question)

Structure	Requires on keys	Access time
Association list	equality only	$O(n)$
Binary search tree	total ordering	$O(\log n)$ avg, $O(n)$ worst
Functional array	integers	$O(\log n)$ worst case

Less generality (stronger assumptions on keys) buys better worst-case guarantees.

6 Functions as Values (Higher-Order Functions)

6.1 Anonymous Functions and Currying

```
fun n -> n * 2          (* anonymous function *)
let prefix a b = a ^ b  (* curried: short form of fun a -> fun b -> a ^ b *)
let promote = prefix "Senior "   (* partial application *)
```

A **curried** function of n arguments is really a chain of n one-argument functions, each returning the next. `let f x1 ... xn = E` is equivalent to `fun x1 -> ... -> fun xn -> E` (for non-recursive `f`). Currying enables **partial application**: fixing early arguments yields a new, useful function (e.g. fixing the comparator of a generic sort).

TRAP: functions cannot be compared for equality

OCaml functions are not comparable with `=` – there is no principled way to decide if two functions denote "the same" computation. Do not write code (or claim in an exam answer) that tests functions for equality.

6.2 The Core Functional Toolkit

```
let rec map f = function [] -> [] | x :: xs -> (f x) :: map f xs

let rec filter p = function
  | [] -> []
  | x :: xs -> if p x then x :: filter p xs else filter p xs

let rec exists p = function [] -> false | x :: xs -> (p x) || exists p xs
let rec all p = function [] -> true | x :: xs -> (p x) && all p xs
```

`exists`/`all` short-circuit thanks to lazy `||` / `&&`. Built from these: `member y xs = exists (fun x -> x=y) xs`; `inter xs ys = filter (fun x -> member x ys) xs`.

6.3 fold – the Universal List Consumer

fold : ('a -> 'b -> 'a) -> 'a -> 'b list -> 'a

```
let rec fold f acc = function
  | [] -> acc
  | x :: xs -> fold f (f acc x) xs
```

`fold` is tail-recursive (it *is* an accumulator pattern), $O(n)$ time, $O(1)$ auxiliary stack space (though it may build up large intermediate values depending on `f`). Almost any single-pass list computation – sum, length, reverse, map, filter – can be expressed via `fold`.

Tripos Pattern (2024 P1 Q1 & Q2): fold, statistics, primality

Q1 asks you to define `fold`, `map`, `filter` from scratch, then use them (with `float_of_int`, `sqrt`) to compute mean $\bar{x} = \frac{1}{n}\sum x_i$ and standard deviation $s = \sqrt{\frac{1}{n}\sum(x_i - \bar{x})^2}$ over exam marks after filtering out zero (unattempted) entries – the key insight is to *filter first*, since n (the count) changes once zeros are removed, so it must be recomputed, not assumed equal to the list length before filtering.
Q2 builds a primality test by trial division up to $\sqrt{n}$ (checking only p with $p^2 \leq n$ avoids needing a float square root), then `fold_range a b f acc` (a specialised integer fold summing `f` over $[b, a]$) and generic `fold`, and finally `all_primes a b` built by folding `is_prime` filtering over the range. This is a good template for "build a small function, then compose it with earlier parts" questions – always reuse your own earlier definitions rather than re-deriving from scratch.

6.4 Matrix Example: transpose and multiply

```
let rec transp = function
  | [] :: _ -> []
  | rows -> (map List.hd rows) :: (transp (map List.tl rows))

let rec dotprod xs ys =
  match xs, ys with
  | [], [] -> 0.0
  | x :: xs, y :: ys -> (x *. y) +. dotprod xs ys

let matprod arows brows =
  let cols = transp brows in
  map (fun row -> map (dotprod row) cols) arows
```

A textbook illustration of composing `map` at two nesting levels plus a curried helper (`dotprod row`, partially applied inside the inner `map`) instead of writing explicit nested loops.

7 Lazy Lists (Sequences)

7.1 Motivation and Type

Lazy lists represent **possibly infinite** data, computed **on demand**. Useful when (a) a computation may have infinitely many solutions but we only want a few, or (b) we want to model genuinely unbounded/infinite series.

The seq datatype

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

The tail is a *delayed* computation `unit -> 'a seq`, not a value. `fun () -> E` delays evaluation of `E`; calling it ("`xf ()`") **forces** that one step of evaluation. This uses OCaml's `unit` type (single value `()`) purely as a device to postpone work.

Infinite sequence & consuming it

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

let rec get n s =
  match n, s with
  | 0, _ -> []
  | n, Nil -> []
  | n, Cons (x, xf) -> x :: get (n - 1) (xf ())
```

`get n s` forces exactly the elements it needs (plus one extra, since the constructor for element n already contains element $n+1$'s delayed tail unevaluated – a subtlety worth mentioning if asked to trace evaluation precisely).

TRAP: force must stay inside a delay

```
let rec appendq xq yq =
  match xq with
  | Nil -> yq
  | Cons (x, xf) -> Cons (x, fun () -> appendq (xf ()) yq)
```

Note `xf ()` appears *inside* `fun () -> ...`: each force is wrapped in a delay, so `appendq` stays lazy. If `xq` is infinite, `yq` is simply never reached – concatenating an infinite sequence with anything is uninteresting. `interleave` fixes this by swapping the two sequences on each recursive step, so both contribute infinitely often. Filtering (`filterq`) necessarily contains an *unprotected* force (it must search until it finds a satisfying element), so filtering an infinite sequence with no matching element does not terminate – always flag this risk in an exam answer.

7.2 Numerical Application: Newton–Raphson

Square roots via an infinite sequence of approximations

```
let rec iterates f x = Cons (x, fun () -> iterates f (f x))
let next a x = (a /. x +. x) /. 2.0
let rec within eps = function
  | Cons (x, xf) ->
      match xf () with
      | Cons (y, yf) ->
          if abs_float (x -. y) <= eps then y
          else within eps (Cons (y, yf))
let root a = within 1e-6 (iterates (next a) 1.0)
```

`iterates f x` generates $x, f(x), f(f(x)), \ldots$; `within` scans until two consecutive terms are close enough. This composition of three small, general functions to solve a concrete numerical problem is a recurring tripos motif – expect to be asked to write an analogous pipeline (e.g. for a different fixed-point iteration).

Tripos Pattern: infinite trees / enumerations (2008 Q5, 2003 Q5 style)

A recurring family: define a *lazy binary tree* datatype (each subtree itself a delayed `unit -> lazy_tree`), then write a function producing a lazy *list* of all labels by interleaving the (lazy) traversals of the two children – you cannot use ordinary `@` since a subtree may be infinite. Similarly, "list all bit-lists of length 0,1,2,..." or "list all palindromes" questions want you to build an infinite lazy list by *diagonalising*: emit all length-k items before moving to length $k+1$, generated by `map`/`iterates`-style composition over a seed sequence, never by trying to compute "all lists" eagerly.

8 Queues, Stacks & Search Strategies

8.1 Breadth-First vs Depth-First

`preorder`/`inorder`/`postorder` are all **depth-first**: the left subtree is fully explored before the right. **Breadth-first** search instead explores all nodes at depth k before any at depth $k+1$ – useful when we want the *nearest* solution first, or when the tree may be infinite (depth-first can get stuck forever down an infinite left branch).

TRAP: naive BFS with append is disastrously slow

```
let rec nbreadth = function
  | [] -> []
  | Lf :: ts -> nbreadth ts
  | Br (v, t, u) :: ts -> v :: nbreadth (ts @ [t; u])
```

`ts @ [t; u]` copies the (potentially huge) list `ts` on *every* node visited, just to append two elements at the end – at depth d the pending list already has $O(2^d)$ entries. This is the textbook motivation for the efficient queue below.

8.2 Efficient Functional (Amortised $O(1)$) Queues

Two-list queue representation

Represent the queue $x_1x_2\cdots x_m\, y_n\cdots y_1$ as a pair `(front, rear)` $= ([x_1,\ldots,x_m],[y_1,\ldots,y_n])$: front stored in order, rear stored *reversed*. `enq` conses onto `rear` ($O(1)$); `deq`/`qhd` read from `front` ($O(1)$) *unless* front is empty, in which case `rear` is reversed into the new front ($O(n)$, but this only happens once per element over the queue's lifetime – **amortised** $O(1)$).

```
type 'a queue = Q of 'a list * 'a list
let norm = function Q ([], tls) -> Q (List.rev tls, []) | q -> q
let enq (Q (hds, tls)) x = norm (Q (hds, x :: tls))
exception Empty
let deq = function Q (x :: hds, tls) -> norm (Q (hds, tls)) | _ -> raise Empty
let qhd = function Q (x :: _, _) -> x | _ -> raise Empty
let qempty = Q ([], [])
let qnull = function Q ([], []) -> true | _ -> false
```

Amortised analysis: over n `enq`s and n `deq`s from empty, every element is consed once on entry and (at most) once during a reversal, so total work is $O(n)$: average $O(1)$ per operation, even though any *single* `deq` might cost $O(n)$ in the worst case (unsuitable for hard real-time deadlines).

Fast BFS using the queue

```
let rec breadth q =
  if qnull q then []
  else match qhd q with
    | Lf -> breadth (deq q)
    | Br (v, t, u) -> v :: breadth (enq (enq (deq q) t) u)
```

Measured speedup over `nbreadth`: roughly 200× on a depth-12 tree – data structure choice, not cleverness, is the whole story here.

8.3 Iterative Deepening

Trading time for space

BFS to depth d with branching factor b visits $1+b+\cdots+b^d = \frac{b^{d+1}-1}{b-1} = O(b^d)$ nodes, and must *store* $O(b^d)$ of them. **Iterative deepening** repeats depth-first search with depth bound $1,2,3,\ldots$, discarding each previous attempt: this only costs a constant factor $\frac{b}{b-1}$ extra *time* (since level $d+1$ alone has more nodes than all previous levels combined, for $b \geq 2$) but reduces **space to** $O(d)$ – exponential space saving for a constant time penalty.

8.4 Search Strategy Survey

Strategy	**Data structure**	**Trade-off**
Depth-first	stack	efficient, but incomplete on infinite trees
Breadth-first	queue	finds nearest solution, but $O(b^d)$ space
Iterative deepening	repeated DFS	breadth-first behaviour, $O(d)$ space
Best-first	priority queue	heuristic-guided, needs a ranking function

The data structure used to store pending nodes determines the search strategy – this one-line summary is a good way to open an essay-style answer on search.

9 Procedural (Imperative) Programming

9.1 References

Primitives

Syntax	Type	Effect
`ref E`	`'a -> 'a ref`	allocate a new cell holding `E`
`!p`	`'a ref -> 'a`	dereference: read current contents
`p := E`	`'a ref -> 'a -> unit`	update contents of `p`

`let p = ...` *itself* is never reassignable – only the *contents* of a reference cell can change. `!` is not a mathematical function since its result depends on mutable store.

TRAP: aliasing

```
let ps = [ref 77; p]
List.hd ps := 3   (* mutates the cell that List.hd ps points to *)
```

If two names refer to the *same* reference cell, mutating through either is visible via both – classic tripos questions (e.g. 2012 Q2) ask you to trace exactly which cells are shared after operations like `replicate n (ref 0)` (one shared cell, replicated n times!) versus `map ref [1;2;3;4]` (four distinct cells), then predict the outputs of subsequent `map (fun r -> r := ...)` calls. Draw the box-and-pointer diagram; do not just reason symbolically.

9.2 Commands and Sequencing

C_1; ...; C_n evaluates each C_i in order, discarding all but the last value, purely for side effects. A typical command has type `unit`.

9.3 Iteration: `while`

```
let length xs =
  let lp = ref xs in
  let np = ref 0 in
  let fin = ref false in
  while not !fin do
    match !lp with
    | [] -> fin := true
    | _ :: xs -> lp := xs; np := 1 + !np
  done;
  !np
```

`while B do C done` evaluates `B`; if true, runs `C` and repeats; terminates (returning `()`) once `B` is false. This is OCaml's one real looping construct (besides `for`, not covered).

9.4 Closures over Private State

Bank account: encapsulated mutable state

```
exception TooMuch of int
let makeAccount initBalance =
  let balance = ref initBalance in
  let withdraw amt =
    if amt > !balance then raise (TooMuch (amt - !balance))
    else begin balance := !balance - amt; !balance end
  in withdraw
```

Each call to `makeAccount` allocates a *fresh* `balance` cell, captured only by the returned `withdraw` closure – no other code can access it. This is the functional-language route to what object-oriented languages call encapsulated (private) instance state; two accounts created this way are completely independent.

9.5 Arrays

```
let aa = Array.init 5 (fun i -> i * 10)   (* [|0;10;20;30;40|] *)
Array.get aa 3      (* = 30 *)
Array.set aa 3 42   (* aa.(3) <- 42, same effect *)
```

Arrays give true $O(1)$ random access but are fixed-size (growing means allocate + copy) and unsafe in languages without bounds checking (OCaml *does* bounds-check, raising `Invalid_argument` – unlike C, where an out-of-bounds write is a classic buffer-overrun vulnerability).

Tripos Pattern (2012 P1 Q2c): predicting output of chained mutation

Given `rlist` built from shared and distinct references, and `slist = map (fn r => ref (!r)) rlist` (a *deep* copy – fresh cells with the same initial contents), successive `map` calls mutating `rlist` do *not* affect `slist` (different cells), while two `map`s over `rlist` itself compound their effects on the *same* cells. The examiners specifically test whether you understand that `ref (!r)` allocates new storage, whereas simply rebinding a name does not.

10 Past Paper Pattern Library

10.1 Topic Map: What Gets Asked, and How Often

FoCS Paper 1 has traditionally set two questions (Q1 and Q2), each with several sub-parts, drawing from across the course. Since roughly 2019 the questions have moved fully to **OCaml** syntax (pre-2019 papers use Standard ML / `fun`/`fn` syntax – the underlying ideas transfer directly, but rewrite the syntax when practising with old papers).

Topic	Typical treatment
Lists & HOFs	Define `map`/`filter`/`fold` (or a close variant) from scratch, then compose them to solve a bespoke problem (statistics, primality, run-length encoding, a small game).
Datatypes & trees	Define a new variant/recursive type for a bespoke structure (quadtrees, expression trees, game states), then write efficient search/update functions exploiting the type's invariant.
Exceptions	Either used as a tool within a bigger question (backtracking, signalling failure), or the direct subject (define, use, eliminate exceptions; prove equivalence).
References & imperative style	Trace aliasing/sharing carefully; write closures with private state; occasionally `while`-loop rewrites of earlier recursive functions.
Sorting	Either implement/debug a named algorithm, or use a generic `sort` as a building block for a higher-level task (frequency sort, checking sort correctness).
Lazy lists	Less frequent, but recurs as "define an infinite/lazy structure and consume it" (lazy trees, infinite enumerations).
Complexity	Almost always required *somewhere*: "state, with justification, the time/space complexity" of a function you just wrote. Never skip this.

Old (ML) vs New (OCaml) Syntax Cheat-Sheet, for practising pre-2019 papers

Standard ML	OCaml
`fun f x = ...`	`let rec f x = ...` (add `rec` if recursive)
`fn x => E`	`fun x -> E`
`val x = E`	`let x = E`
`[]`, `::`, `@`	same
`ref`, `!`, `:=`	same
list of pairs `(a,b)`	same tuple syntax
`raise Foo`, `Foo x`	same
comments `(* ... *)`	same

The concepts (pattern-matching, currying, references, exceptions, laziness) are identical; only surface syntax for anonymous functions and `let`/`val` differs meaningfully.

10.2 Worked Example: 2023 Paper 1 Q2 – Quadtrees

Efficient point search in a spatial data structure

```
type point = int * int
type qt = Empty | Quad of qt * qt * point * qt * qt
(* Quad(nw, ne, (x,y), sw, se): x splits nw/sw from ne/se; y splits sw/se from nw/ne *)

type range = int * int
type rel = LT | IN | GT
let compare_range n (lo, hi) =
  if n < lo then LT else if n > hi then GT else IN

let rec has_point (px, py) = function
  | Empty -> false
  | Quad (nw, ne, (x, y), sw, se) ->
      if px = x && py = y then true
      else match px < x, py < y with
        | true, true   -> has_point (px, py) sw   (* left, below split *)
```

```
        | true, false  -> has_point (px, py) nw
        | false, true  -> has_point (px, py) se
        | false, false -> has_point (px, py) ne
```

The key idea, worth stating explicitly for marks: at each node we can **discard three of the four quadrants** using only the comparison of the query point against (x, y), giving expected $O(\log n)$ search on a balanced quadtree (worst case $O(n)$, exactly analogous to a BST). The rectangular-region search (`has_point_in`) generalises this by pruning any quadrant whose (x, y)-derived bounding half-plane cannot intersect the query rectangle, rather than checking every point.

10.3 Worked Example: 2022 Paper 1 Q1 – `mapi` and a Wordle-style game

Combining a positional map with a lookup helper

```
let mapi f l =
  let rec go i = function
    | [] -> []
    | x :: xs -> f i x :: go (i + 1) xs
  in go 0 l

let rec lookfor y = function
  | [] -> None
  | (a, b) :: rest -> if b = y then Some a else lookfor y rest
```

`mapi` is `map` with an extra positional-index accumulator threaded through – the standard technique whenever a list function needs to know "where am I" as well as "what's here". Building `respond` (per-letter Green/Amber/Black feedback) from `mapi` and `lookfor` is then a direct composition: at position `i`, compare the guessed letter against the target's letter at `i` (Green), else search the rest of the target for that letter (Amber), else Black.

10.4 Worked Example: 2025 Paper 1 Q2 – Expression Trees and Polish Notation

Datatype design, evaluation, and stepwise reduction

```
type expr = Add of expr * expr | Mul of expr * expr | Number of int

let rec eval = function
  | Number n -> n
  | Add (e1, e2) -> eval e1 + eval e2
  | Mul (e1, e2) -> eval e1 * eval e2
(* eval : expr -> int *)

(* Polish-notation token stream, as a flat list *)
type token = N of int | Plus | Times
```

$(1 + 4) \times (10 + 2)$ is `Mul(Add(Number 1, Number 4), Add(Number 10, Number 2))`. The flat token type `token` is needed because Polish notation is a *list*, not a tree, so `reduce` must find an operator immediately followed by two number tokens and replace all three by their combined value – a single left-to-right scan, repeated (`reduce_all`) until only one number token remains. This tests the same skill as making change and backtracking: repeatedly transforming a list-shaped state until a fixed point is reached.

11 Model Answer Templates

Template 1: "Write brief notes on X, illustrated by Y"

1. **Define** the concept precisely (one or two sentences, using correct terminology: e.g. "a curried function returns a function as its result, so that `f x1 ... xn` is really `((f x1) x2) ... xn`").
2. **Give the general syntax/type** (e.g. `let f x y = E` vs. `fun x -> fun y -> E`).
3. **Illustrate with the requested example**, fully typed and commented.
4. **State the payoff**: why does this feature matter? (Partial application, code reuse, abstraction barrier, efficiency, safety.)

This four-step shape reliably scores well on the "brief notes" parts that open many tripos questions.

Template 2: "State, with justification, the time/space complexity"

1. Identify the **recursive structure**: how many recursive calls, on how much smaller an input, plus how much non-recursive work per call.
2. Write down the **recurrence relation** $T(n) = \ldots$, with a base case.
3. Solve it (usually by recognising it as one of the four standard forms) or state the closed form and verify by substitution/induction.
4. State **time and space separately**; note whether the function is tail-recursive (affects auxiliary stack space).
5. If there's a worst-case/average-case gap (quicksort, BSTs), state *both* and explain what triggers the worst case.

Template 3: "Write a function to ... [manipulate a recursive datatype]"

1. Write down the **type signature** first – this clarifies what the function must do and catches design errors early.
2. Pattern-match on the datatype's constructors; handle the **base case(s)** first.
3. In the recursive case, identify what to do **before**, **between**, and **after** the recursive call(s) (this is exactly the pre/in/postorder distinction, generalised).
4. If `@` or a similar costly operation appears in the naive version, consider an **accumulator** rewrite and mention the complexity improvement.
5. Trace your function on a tiny example by hand before finalising.

Template 4: "Rewrite this recursive function using `while`/references" (or vice versa)

1. Identify each piece of information the recursive version threads through its arguments/accumulators – each becomes a `ref`.
2. Identify the recursive function's base-case test – this becomes the *negation* of the `while` condition.
3. The recursive step's "do work, then recurse on smaller input" becomes "update the accumulator refs, then loop".
4. After the loop, read off the final answer from the relevant ref(s).
5. Comment on any trade-off: imperative code often loses the guarantee that all paths return a properly-typed value in one place, and makes reasoning about correctness (via induction) harder.

11 Consolidated Formula & Code Sheet

11.0 Complexity Cheat Sheet

Recurrence	Complexity
$T(n+1) = T(n) + 1$	$O(n)$
$T(n+1) = T(n) + n$	$O(n^2)$
$T(n) = T(n/2) + 1$	$O(\log n)$
$T(n) = 2T(n/2) + n$	$O(n \log n)$

Structure op	Cost
`::`, `ref`, `!`, `:=`	$O(1)$
`@` (append)	$O(\|\text{first list}\|)$
List `length`, naive `rev`	$O(n)$, $O(n^2)$ resp.
BST/functional array op	$O(\log n)$ balanced
Queue `enq`/`deq`	amortised $O(1)$

11.0 The Accumulator Pattern (used everywhere)

Naive recursion that nests non-tail calls (or misuses `@`) can almost always be made linear by threading an extra argument that accumulates the answer so far, then reversing/finalising once at the end. Seen in: `summing`, `rev_app`, append-free `quik`, `inord`/`preord`/`postord`, `fold`.

11.0 Core List Functions

```
let rec map f = function [] -> [] | x::xs -> f x :: map f xs
let rec filter p = function [] -> [] | x::xs -> if p x then x::filter p xs else filter p xs
let rec fold f acc = function [] -> acc | x::xs -> fold f (f acc x) xs
let rec append xs ys = match xs,ys with [],ys->ys | x::xs,ys-> x::append xs ys
let rec rev_app xs ys = match xs,ys with [],ys->ys | x::xs,ys-> rev_app xs (x::ys)
```

11.0 Sorting

```
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)
```

Quicksort $O(n \log n)$ avg / $O(n^2)$ worst; Mergesort $O(n \log n)$ always; Insertion sort $O(n^2)$.

11.0 Trees & Dictionaries

```
type 'a tree = Lf | Br of 'a * 'a tree * 'a tree
let rec lookup b = function
  | Br((a,x),t1,t2) -> if b<a then lookup b t1 else if a<b then lookup b t2 else x
  | Lf -> raise (Missing b)
```

11.0 Lazy Lists

```
type 'a seq = Nil | Cons of 'a * (unit -> 'a seq)
let rec from k = Cons (k, fun () -> from (k+1))
```

11.0 Queues

```
type 'a queue = Q of 'a list * 'a list  (* front, reversed rear *)
```

11 Tripos Exam Strategy & Common Pitfalls

11.0 Paper Structure & Technique

- Two long questions, each split into lettered sub-parts worth a handful of marks; sub-parts within a question typically build on each other – always reuse earlier parts rather than re-deriving.
- Write down the **type signature** of every function before its body: it is often worth marks on its own and catches design mistakes immediately.
- When a question says "or otherwise", using the suggested earlier function is usually the fastest correct route – don't reinvent it.
- If a question gives a **hint** (e.g. "first consider deleting a node from a BST"), that hint is almost always necessary scaffolding, not optional colour – follow it.
- For "state the complexity" parts: always show the recurrence, not just the final $O(\cdot)$ – partial credit is given for correct reasoning even with an arithmetic slip.

11.0 Top 10 FoCS-Specific Traps

Watch out for these

1. Claiming quicksort is $O(n \log n)$ without the "average case" qualifier.
2. Using `@` inside a recursive function without noticing the resulting quadratic blow-up (list reversal, tree traversal, naive BFS).
3. Forgetting that OCaml pattern-match clauses are tried *in order* – a wildcard `_` or unguarded variable pattern placed too early makes later clauses unreachable (classic in the 2025 mergesort bug-hunt style question).
4. Confusing `'a ref list` (a list of independently-mutable cells) with `'a list ref` (one cell holding a whole list) – know the difference and how aliasing behaves in each.
5. Assuming BST/functional-array operations are $O(\log n)$ *unconditionally* – BSTs need the tree to be balanced; state the worst case too.
6. Forgetting that functions cannot be compared with `=` in OCaml.
7. Writing a lazy-list function with a force *not* wrapped in a delay, silently making the whole thing eager (breaks laziness invariants, can loop forever on infinite input).
8. Recomputing `n` (a count/length) *before* filtering when the question requires it to be computed *after* filtering (2024 Q1 statistics trap).
9. Treating `exception` handling and `option` as interchangeable without discussing the trade-off (undeclared potential failures vs. forced caller-side checking) when a question asks you to compare them.
10. Skipping the "worked trace" – many questions implicitly reward tracing your function on a small example (e.g. the `change` backtracking trace); do this even when not explicitly asked, to catch your own bugs.

11.0 General Technique

- Manage time: with two questions in the time allotted, do not let one sub-part consume a disproportionate share – return to it if time allows.
- Comment your OCaml code briefly in the exam; markers reward clarity and it costs almost nothing.
- Prefer clear, direct recursive definitions over clever one-liners – the tripos rewards correctness and clarity ("free of needless complexity", as one paper literally states) over cleverness.
- When stuck on the "efficient" version of a function, write the naive version first for partial credit, then attempt the accumulator/queue/tree optimisation.

11 Final Exam Checklist

Before the exam, make sure you can...

- State and apply the definition of $O(\cdot)$, and simplify expressions like $O(n^2 + 50n + 36)$.
- Recognise and solve the four standard recurrence forms on sight.
- Write `nsum`/`summing`-style naive vs. accumulator pairs for at least: sum, length, reverse, tree traversal, quicksort combine step.
- Explain why naive list reversal and naive tree traversal are $O(n^2)$, and fix both with an accumulator.
- Write insertion sort, quicksort (with partition), and top-down mergesort from memory, with their average/worst-case complexities.
- Define a variant (tagged union) datatype and write pattern-matching functions over it, including one recursive datatype (binary tree or similar).
- Declare and raise/handle exceptions; convert between exception-based and `option`-based error handling.
- Implement `lookup`/`update` for an association list and a binary search tree, and state their complexities including worst case.
- Explain functional arrays as balanced binary trees and why they guarantee $O(\log n)$ worst case.
- Write curried functions, use partial application, and implement `map`, `filter`, `exists`, `all`, `fold` from scratch.
- Define the lazy-list (`seq`) type, write `from`/`get`/`interleave`, and explain the force/delay discipline.
- Implement the two-list amortised-$O(1)$ functional queue, and use it for a fast breadth-first search.
- Explain iterative deepening and why it trades a constant time factor for an exponential space saving.
- Write imperative OCaml using `ref`, `while`, and arrays; trace aliasing between references correctly; write a closure that encapsulates private mutable state.
- Reproduce the model-answer templates for "brief notes on X", "state the complexity", and "write a function on datatype Y" under time pressure.

Good luck – *practice tracing small examples by hand; it catches more bugs than staring at code.*