

UNIVERSITY OF BIRMINGHAM
School of Computer Science

First Year - BSc Artificial Intelligence and Computer Science
First Year - BSc Natural Sciences
First Year - BSc Computer Science
First Year - MEng Computer Science/Software Engineering

06 22754

Foundations of Computer Science
Summer Examinations 2012

Time allowed: 3 hr

[Answer ALL Questions]

[Use the Separate Answer Book for EACH Part]
Part A

[Use the Separate Answer Book for THIS Section]

IN THIS SECTION YOU ARE ALLOWED TO USE ALL THE FUNCTIONS IN THE APPENDIX

1. Write an OCaml function called \texttt{cd} which counts the number of elements which have consecutive duplicates. What is the asymptotic complexity of your function?

For example: \texttt{cd [1; 2; 3; 3; 4; 5; 5]} = 2
For example: \texttt{cd [1; 2; 2; 2; 3]} = 2

2. Using symbolic evaluation and showing all steps of computation, calculate

\texttt{map (fun n -> n+1) [1; 2]}

where \texttt{map} is defined as

\[
\text{let rec map f = function}
\]
\[
\quad | [ ] -> [ ] \\
\quad | x::xs -> (f x)::(map f xs)
\]

3. Write an OCaml function \texttt{sf} which orders the elements of a list according to their increasing frequency. What is the asymptotic complexity of your function?

For example: \texttt{sf [1; 2; 3; 4; 2; 3; 4; 4]} = [1; 2; 3; 4]
4. Consider this function, which "flips" a binary tree:

```ocaml
let rec flip = function
| Empty -> Empty
| Node (a, l, r) -> Node (a, flip r, flip l)
```

Using structural induction on trees show that flip is its own inverse, which means

\[
\text{flip (flip t)} = t
\]

for any tree \( t \).

5. (a) Explain the term “amortised complexity”.

(b) Give the 2-list implementation of queues ("Banker's queues").

(c) Explain why your implementation in (b) is superior to the naive implementation below:

```ocaml
Open List
type 'a queue = 'a list
let enq (x, xs) = xs @ [x]
let deq xs = (hd xs, tl xs)
```
Part B

[Use the Separate Answer Book for THIS Section]

6. (a) Give an inductive definition of a **Binary Tree** and of a **Quad Tree**. \[2\%\]

(b) Explain how a Quad Tree can be used to represent a square gray-scale image. Define appropriate primitive operators for Quad Trees and use them to create an efficient pseudocode algorithm that turns a gray-scale image upside down. \[6\%\]

(c) What additional conditions must be satisfied for a **Binary Tree** to be a valid **Binary Search Tree**? \[2\%\]

(d) Draw the **Binary Search Tree** that results from inserting the items \[5 \ 9 \ 7 \ 3 \ 1 \ 8 \ 4 \ 2\] in that order into an initially empty tree. Then draw the tree that results from removing item 5 from that tree using the standard **Binary Search Tree** delete algorithm. \[4\%\]

7. (a) Describe in words the general idea underlying the **Quicksort** algorithm for sorting an array of items. \[3\%\]

(b) What are the average and worst case time complexities of the **Quicksort** algorithm? Give a simple example of how the worst case can easily arise, and how it can be avoided. \[4\%\]

(c) What does it mean to say that a sorting algorithm is **stable**? Discuss the stability of the **Quicksort** algorithm and the relation of stability to the computational costs of the algorithm. \[3\%\]

(d) Suppose you had data about seven million customers to sort according to how much they had spent, but only needed to know the fifty who had spent the most. Explain how those details would affect your choice of sorting algorithm. \[3\%\]
8. (a) Explain what is meant by the terms hash table, hash collision, linear probing and secondary hashing. [3%]

(b) Comment on the advantages and disadvantages of using hash tables for data storage. [3%]

(c) Suppose four digit keys are to be stored in a hash table represented as an array of size 13. The hash function is simply the first digit. Why is that not a sensible choice of hash function? Draw the initially empty hash table and insert the following keys into it using linear probing: “3487”, “1346”, “4143”, “3571”, “2583” and “0937”. [3%]

(d) Suggest an improved hash function for the above type of data and explain why it is better. [2%]

9. (a) What does it mean to say a graph is planar? Suggest a practical application where that property is important. Draw the graphs $K_5$ and $K_{3,3}$. Why are these graphs important in the context of planarity? [4%]

(b) Outline the general idea of Dijkstra’s algorithm for finding the shortest path between two nodes in a weighted graph. Apply the algorithm to find the shortest path from A to F in the following weighted graph, showing the computations involved at each stage. [8%]
APPENDIX : THE LIST MODULE (SELECTED FUNCTIONS)

val length : 'a list -> int
Return the length (number of elements) of the given list.

val hd : 'a list -> 'a
Return the first element of the given list. Raise Failure "hd" if the list is empty.

val tl : 'a list -> 'a list
Return the given list without its first element. Raise Failure "tl" if the list is empty.

val nth : 'a list -> int -> 'a
Return the n-th element of the given list. The first element (head of the list) is at position 0. Raise Failure "nth" if the list is too short. Raise Invalid_argument"List.nth" if n is negative.

val rev : 'a list -> 'a list
List reversal.

val append : 'a list -> 'a list -> 'a list
Catenate two lists. Same function as the infix operator @. Not tail-recursive (length of the first argument). The @ operator is not tail-recursive either.

val rev_append : 'a list -> 'a list -> 'a list
List.rev_append l1 l2 reverses l1 and concatenates it to l2. This is equivalent to List.rev l1 @ l2, but rev_append is tail-recursive and more efficient.

val map : ('a -> 'b) -> 'a list -> 'b list
List.map f [a1; ...; an] applies function f to a1, ..., an, and builds the list [f a1; ...; f an] with the results returned by f. Not tail-recursive.

val rev_map : ('a -> 'b) -> 'a list -> 'b list
List.rev_map f l gives the same result as List.rev (List.map f l), but is tail-recursive and more efficient.

val fold_left : ('a -> 'b -> 'a) -> 'a -> 'a list -> 'a
List.fold_left f a [b1; ...; bn] is f (... (f (f a b1) b2) ...) bn.

val fold_right : ('a -> 'b -> 'b) -> 'a list -> 'b -> 'b
List.fold_right f [a1; ...; an] b is f a1 (f a2 (... (f an b) ...)). Not tail-recursive.

val for_all : ('a -> bool) -> 'a list -> bool
for_all p [a1; ...; an] checks if all elements of the list satisfy the predicate p. That is, it returns (p a1) && (p a2) && ... && (p an).

val exists : ('a -> bool) -> 'a list -> bool
exists p [a1; ...; an] checks if at least one element of the list satisfies the predicate p. That is, it returns (p a1) || (p a2) || ... || (p an).
val mem : 'a -> 'a list -> bool
mem a l is true if and only if a is equal to an element of l.

val find : ('a -> bool) -> 'a list -> 'a
find p l returns the first element of the list l that satisfies the predicate p. Raise
Not_found if there is no value that satisfies p in the list l.

val filter : ('a -> bool) -> 'a list -> 'a list
filter p l returns all the elements of the list l that satisfy the predicate p. The order of
the elements in the input list is preserved.

val find_all : ('a -> bool) -> 'a list -> 'a list
find_all is another name for List.filter.

val split : ('a * 'b) list -> 'a list * 'b list
Transform a list of pairs into a pair of lists: split [(a1,b1); ...; (an,bn)] is ([a1; ...; an],
[b1; ...; bn]). Not tail-recursive.

val combine : 'a list -> 'b list -> ('a * 'b) list
Transform a pair of lists into a list of pairs: combine [a1; ...; an] [b1; ...; bn] is [(a1,b1);
...; (an,bn)]. Raise Invalid_argument if the two lists have different lengths. Not tail-
recursive.

val sort : ('a -> 'a -> int) -> 'a list -> 'a list
Sort a list in increasing order according to a comparison function. The comparison
function must return 0 if its arguments compare as equal, a positive integer if the first
is greater, and a negative integer if the first is smaller (see Array.sort for a complete
specification). For example, compare is a suitable comparison function. The resulting
list is sorted in increasing order. List.sort is guaranteed to run in constant heap space
(in addition to the size of the result list) and logarithmic stack space. The current
implementation uses Merge Sort. It runs in constant heap space and logarithmic
stack space.

val stable_sort : ('a -> 'a -> int) -> 'a list -> 'a list
Same as List.sort, but the sorting algorithm is guaranteed to be stable (i.e. elements
that compare equal are kept in their original order). The current implementation uses
Merge Sort. It runs in constant heap space and logarithmic stack space.

val fast_sort : ('a -> 'a -> int) -> 'a list -> 'a list
Same as List.sort or List.stable_sort, whichever is faster on typical input.

val merge : ('a -> 'a -> int) -> 'a list -> 'a list -> 'a list
Merge two lists: Assuming that l1 and l2 are sorted according to the comparison
function cmp, merge cmp l1 l2 will return a sorted list containing all the elements of
l1 and l2. If several elements compare equal, the elements of l1 will be before the
elements of l2. Not tail-recursive (sum of the lengths of the arguments).