Foundations of Computer Science : Christmas Revision

Start your revision by going through the lecture notes and the supervision problems.

This sheet breaks down past exam questions by topic. Try to answer around one from each topic. For the “lists” topic do as many as you can. Take note of the questions you had trouble with, so we can go through them during the revision class. As you gain more experience restrict yourself to examination conditions.

For the revision class, the only requirement is to attempt the 2014-2016 past papers.

Language features

Type inference

- 2018p1q1 (c)
- 2014p1q2 (b)
- 2014p1q1 (a), (d), (e)
- 2009p1q1 (c)

Functions

- 2018p1q1 (b)
- 2016p1q1 (a)
- 2012p1q2 (a)
- 2004p1q6 (a)
- 2003p1q1 (a)
- 2001p1q5 (a (i))

Pattern matching:

- 2013p1q1 (a)

Exceptions

- 2011p1q2 (a), (b), (c)
- 2007p1q5
- 2001p1q5 (a (iii))

Data structures

Lists

- 2018p1q1 (a) Representing sets using lists
- 2016p1q1 (b) zarg (essentially foldr)

- 2014p1q2 (c) Compute cost difference to convert between two lists.
- 2014p1q2 (b) Run-length encoding on a list
- 2012p1q2 (a) replicate item function
- 2010p1q2 (a) foldl
- 2009p1q1 (a), (b) implement delFirst
- 2007P1Q6 (d) Replace the k-th instance of an item on a list
- 2004p1q1 map and foldr/foldl
- 2003p1q1 (b) foldr function and zipping, (c) Write a function that returns all elements except those at an index that is a multiple of three
- 2000p1q6 (c) check if a list is cyclic
- 2000p1q1 exf operation

Queues

- 2006p1q5 (a)
- 2014p1q2 (a)

Binary trees

- 2013p1q1 (b), (c)
- 2016p1q1 (c)
- 2008p1q5 (b), (c), (d)
- 2007p1q5 (a), (c)
- 2006p1q6 (b)
- 2005p1q6

Binary search trees

- 2012p1q1
- 2009p1q2 (c)

Data types

- 2013p1q1 (a)
- 2011p1q2 (c (i))
- 2007p1q6 (b), (c)

Sorting

- 2010p1q2 (b)
- 2009p1q2 (a)
- 2007p1q1
- 2005p1q5
- 2001p1q1 (c)
- 1998p1q1

Permutations

Exercise [Check if a list is a permutation of another]: Write an OCaml function which given two lists determines if one is a permutation of the other. (This is 2009p1q1(d))

Exercise [Generalised permutation]: 2009p1q1 (e)

Exercise [Generate all possible permutations]: 2006p1q5 (b)

Exercise [Lazy permutation generation]: Modify your code about to lazily generate all permutations of a list.

Exercise [Generate all possible derangements]: Modify your code from the previous exercise to generate all possible derangements. A derangement is a permutation such that none of its elements maps back to its original position.

Exercise [Find the next permutation]: (+) Write an OCaml function which given a permutation computes the lexicographically next permutation. For example, given $[2;3;1;4]$ it returns $[2;3;4;1]$.

Exercise [Generate all combinations]: Given n and k, write an OCaml function to generate all combinations. (Can you make it generate the combinations lazily?)

Parentheses

Exercise [Check if parentheses are balanced]: You are given a list containing `(` and `)`. Write a program in OCaml to check if a sequence of parentheses are balanced. A sequence of open and closed parentheses being balanced is defined recursively:

- The empty list is balanced.
- If `s` is balanced, then `(s)` is balanced.
- If `s` and `t` are balanced, then `st` are balanced. How efficient is your algorithm?

Exercise [Generate all balanced parentheses]: Write a program in OCaml to generate all balanced parentheses.

Exercise [Lazy generation]: Modify your code from the previous exercise to lazily generate all balanced parentheses.

Infinite structures

Infinite lists

- 2017p1q2: Infinite lists and enumeration
- 2015p1q2: (b), (c), (d) Enumerating
- 2010p1q1 (b), (c) interleave, map infinite lists, + iterates and iterates2

Infinite Binary trees

- 2004p1q5

Games

The lecture notes contain no mention of games, but these have appeared in past exam questions. If you want to gain a basic background, read pages 1 to 10 from Ferguson's book.

- 2018p1q2
- 2017p1q1
- 2011p1q1
- 2008p1q6

Imperative programming

Reference types

- 2012p1q2 (b), (c)
- 2007p1q6 (a)
- 2001p1q1 (a), b)

Mutability

- 2003p1q6 mutable BSTs

(Optional) BFS/DFS Exercise in Java

In this exercise, you will solve the Lights Out puzzle using BFS, DFS and iterative deepening DFS. You are provided with `Board.java` (Java class that represents the Lights out board) and `DfsSolution.java` (Java class that implements a DFS solution for Lights Out).

1. Argue why it is never beneficial to toggle the same light more than once.
2. Familiarise yourself with the DFS solution in `DfsSolution.java`. Explain how/why it works.
3. Run the DFS solution on the 4x4 board and on the 5x5 board and time it. Do these solutions use the minimum number of moves?
4. Code a BFS solution for the problem and compare the solutions obtained with those obtained by the DFS in terms of time, memory and moves needed.
5. Modify the DfsSolution code to become an iterative deepening DFS. (You should only need to modify a few lines in the explore function) Again, compare the solutions obtained with the standard DFS and BFS. What happens if you increase the board size?
6. (Optional) Read this tutorial and write an efficient non-brute-foce solution for the 5x5 case. How does it compare to your previous solutions?