CS2011 Tutorial Sheet: Algorithms

This is the first tutorial sheet of the second year and is to occupy one tutorial session. You may wish to use part of the session to discuss with your tutor arrangements for the second year.

Attempt the questions on this sheet **before your tutorial** writing answers in your LabBook.

This tutorial is introductory and covers the design of algorithms on lists. You may need to review elements of the two programming languages, C and SML.

Question 1.

Be prepared to explain to your tutor how linear lists (or ‘sequences’) of elements are handled in C and in SML. Your account should cover

1. In C: Arrays, pointer types (including the declaration of pointer types, the creation of pointers using ‘malloc’, referencing, dereferencing and assignment of pointers), recursive type definitions using pointer types, the representation of lists as linked lists and the definition of functions over lists using recursion and iteration;

2. In SML: tail recursion on lists, pattern matching and definition by cases.

Question 2.

We wish to calculate the intersection of two sets i.e. the set of elements common to both sets. Let us represent sets as linear lists (sequences) of elements without duplication. The naive algorithm for intersection is to take each item of the first list and compare it with each item in turn in the second list. If it is found then add it to the intersection.

Write a definition of this algorithm in SML using head-and-tail recursion on lists, or in C using linked lists.

Here is an alternative algorithm if the items in the lists are, say, integers. Step 1: Sort both lists into ascending order. Step 2: Compare the heads of the lists, if they are the same add to the intersection, if different then dispose of one item, and continue down the lists.

Write a definition of Step 2 of this algorithm in SML or in C, i.e. the intersection of lists of integers in ascending order. Is the result in ascending order?

We now consider the *performance* of these two algorithms to try to demonstrate that the second is indeed an improvement on the first. How do we measure the performance of an algorithm so as to compare algorithms for the same task? We will discuss this in detail in the lectures, but the idea is to count the number of operations performed as the size of the input varies.

In the case of the first algorithm, at worst, each search through a list exhausts the list (ie searches to the end). How many times does it access elements in the two lists in this case (in terms of the lengths of the two lists, which you may assume to be equal)?

The second algorithm has two phases and we **add** the two performances. Assume the lists to be sorted. How many accesses of the lists are required by the second phase in terms of the length of the lists (assume to be equal). We shall see later in the course that to sort a list of length N efficiently takes at worst approximately $N \log_2(N)$ comparisons. Is the second algorithm therefore an improvement?

Question 3.

An element at position i in a sequence s of integers is a *fixed point* if $s[i] = i$ (where $s[i]$ is the i-th item of s). Consider sequences of integers (which may be zero or negative) which are in *ascending order* and *without duplicates.*

Devise an algorithm (expressed in C or similar language, using arrays) that finds a fixed point in such sequences if there is one, and reports if not. Your algorithm should be efficient and based upon the following technique (called *binary search*): Choose an element midway along the sequence, is it a fixed point? If not, then determine on which side of the midway element a fixed point may occur. Explain why your algorithm finds a fixed point if there is one (i.e. give a correctness argument).

If you have time, you may like to try to assess the performance of this algorithm, as follows:

How many times does your algorithm access elements in a sequence of length N in the worst case? Hint: Consider lists of length $N = 2^M$. Is this an improvement on *linear search* which simply iterates element-by-element through the sequence from the beginning, testing each element to see if it is a fixed point?

Can one implement this algorithm efficiently using lists in SML? Explain your answer and whether a similar situation occurs for the algorithms in Question 2.