IMPERIAL COLLEGE OF SCIENCE, TECHNOLOGY AND MEDICINE

EXAMINATIONS 2023

BEng Honours Degree in Computing Part II
MEng Honours Degrees in Computing Part II
BEng Honours Degree in Mathematics and Computer Science Part II
MEng Honours Degree in Mathematics and Computer Science Part II
BEng Honours Degree in Mathematics and Computer Science Part III
MEng Honours Degree in Mathematics and Computer Science Part III
for Internal Students of the Imperial College of Science, Technology and Medicine

This paper is also taken for the relevant examinations for the Associateship of the City and Guilds of London Institute

PAPER COMP50001

ALGORITHM DESIGN AND ANALYSIS

Monday 15th May 2023, 10:00
Duration: 90 minutes

Answer ALL TWO questions

Paper contains 2 questions
Calculators not required

1 This question is about dynamic programming.

a The *edit distance* between two strings is calculated by the function *dist* and defined to be the minimum number of character inserts, deletions, and updates required to turn one string into the other. A *palindrome* is a string that is equal to its reversal.

A *nearest palindrome* of a given string is a palindrome whose edit distance to that string is minimal. For example, the nearest palindromes of `"abcbef"` include `"abcba"` as well as `"ebcbe"`, which both have an edit distance of 2.

i) Define *palindrome xs* to return *True* when *xs* is a palindrome, and *False* otherwise. State the complexity of your function.

ii) Briefly explain why the following properties hold for any string *xs* of length n:

A) all the characters in a nearest palindrome to *xs* must be from *xs*

B) the edit distance between *xs* and its nearest palindrome is at most $\lfloor n/2 \rfloor$

C) the length of a nearest palindrome to *xs* is bounded by $n + \lfloor n/2 \rfloor$

b i) Define *strings xs n* to produce all the strings of length n whose characters are drawn from *xs*. This need not be efficient but its complexity should be bounded by $O(m^n)$ where $m = \mathit{length}\ xs$.

ii) Using the *strings* function, define *palindromes xs* to return all the palindromes that can be formed from *xs*. *Hint: consider the properties of nearest palindromes and filter appropriate strings with the palindrome function.*

iii) Using *palindromes*, define *palindist xs* to calculate the edit distance between *xs* and its nearest palindromes. For example, *palindist* `"abXcYbZ"` $= \cancel{4}$. You may assume $\mathit{dist} :: \mathit{String} \to \mathit{String} \to \mathit{Int}$. This need not be efficient. 2

c i) Consider how *palindist* `"abcba"` relates to the result of applying *palindist* to the following strings: `"abcbaX"`, `"Xabcba"`, `"XabcbaX"`, `"XabcbaY"`. Using this relationship, define *palindist'*, a recursive version of *palindist*.

ii) Consider why strings make bad indices. Complete the definition of *palindist''*, which is a recursive version of *palindist'* that uses indices *i* and *j*:

$$\begin{aligned}
&\mathit{palindist''} :: \mathit{String} \to \mathit{Int} \\
&\mathit{palindist''}\ \mathit{xs} = \mathit{go}\ 0\ (\mathit{length}\ \mathit{xs} - 1)\ \textbf{where}\ \mathit{go} :: \mathit{Int} \to \mathit{Int} \to \mathit{Int} \\
&\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\quad \mathit{go}\ i\ j = \ldots
\end{aligned}$$

iii) Define *palindist'''*, an efficient version of *palindist''* that uses dynamic programming. You may use $\mathit{tabulate} :: \mathit{Ix}\ i \Rightarrow (i, i) \to (i \to a) \to \mathit{Array}\ i\ a$, which takes a range of indices and a function and creates an array by tabulating the function, and $\mathit{fromList} :: [a] \to \mathit{Array}\ \mathit{Int}\ a$, which returns an array whose elements are from a list.

The three parts carry, respectively, 25%, 30%, and 45% of the marks.

2 This question is about datatype representations.

a Given a natural number n, a *parent list* of the elements $0 \ldots n-1$ is defined to be a list $ps = [p_0, \ldots, p_{n-1}]$ where $0 \leqslant p_i < n$ for all p_i, and p_i is called the *parent* of i.

The *ancestors* of x is a list that includes x, the parent of x, the parent of the parent of x, and so on. The *origin* of x is defined as the ancestor of x whose parent is itself. The *family* of x is defined as all the elements who share an ancestor with x.

In a valid parent list any element x may be its own parent, but x cannot be the parent of any of its other ancestors. We will only consider valid parent lists.

i) Consider the case when $n = 10$, and the parent list is $[0,1,3,5,6,5,6,3,8,6]$. Draw a graph where the nodes are all the numbers $0 \ldots 9$ and an edge goes from a child to its parent. Write the origin and family of each element.

Given a valid parent list ps where $n = \mathit{length}\ ps$.

ii) Briefly explain why every element $x \in \{0, \ldots, n-1\}$ has an origin.

iii) Define a function *ancestors ps x*, which returns the ancestors of x. Use this to define *origin ps x*, which returns the origin of x.

You may assume the existence of function $(!) :: [a] \to \mathit{Int} \to a$, where $xs\ !\ i$ returns the ith element of xs in constant time.

iv) Given an element x, define *family ps x* to return a list of all the elements in the family of x. State the worst-case complexity of your function.

v) Given two elements x and y, define *adopt ps x y* to return the list ps modified so that if xo is the origin of x, and yo is the origin of y, then the origin xo or yo that has the biggest family will become the parent of the other origin. If the families are the same size, then xo becomes the parent of yo. This need not be efficient.

You may assume the existence of $\mathit{update} :: [a] \to \mathit{Int} \to a \to [a]$, where *update xs i x* returns the list xs modified so that $xs\ !\ i = x$ in constant time.

b The complexity of the *adopt* operation can be improved by changing the parent list datastructure to include more information.

i) Modify your definitions so that *adopt* is more efficient by avoiding the recalculation of *family*. You will have to change the type of the parent list to accomodate extra information.

Explain how to convert between the old & the new representations.

ii) Explain how your modified functions work and state their complexities in the worst case.

iii) Consider a parent list ps which is obtained by k arbitrary adopt operations on an intitial parent list where every element is its own parent.

State and explain the worst-case complexity of *adopt ps*.

The two parts carry, respectively, 60% and 40% of the marks.