Travelling Salesman Heuristics: Exercises in Haskell

Eerke Boiten

Technical Report No. 10-04
June 2004

Copyright © 2004 University of Kent at Canterbury
Published by the Computing Laboratory,
University of Kent, Canterbury, Kent, CT2 7NF, UK
1 Introduction

This document contains a collection of programming exercises in the functional programming language Haskell. The exercises are all concerned with the infamous Travelling Salesman Problem (TSP for short), both its exact solution and heuristic approximations. Full solutions to the exercises have not been included (these are available on request). However, the document contains both hints for students and comments for teachers (the latter in small italics). The next section gives the teaching context in which this material was used — in particular, it lists assumptions made about the students’ knowledge. Section 3 describes TSP informally. The following sections present some preliminary material: on recursion over lists (Section 4), on type “implementation” through type aliases (Section 5), and on computing the maximum of a list “under a function” (Section 6). The concept of a distance matrix is discussed in Section 7. After a short discussion of permutations, the exact, “brute force”, TSP algorithm is given in Section 8. The following sections consider a number of heuristic approximations to TSP. The final section contains samples of the exam questions that were asked about model solutions to these exercises.

2 History

The module CO312 Case Studies was introduced into the first year of the BSc Computer Science programme at the University of Kent in 1996. Its purpose was to reinforce teaching in other first year modules through the use of larger case studies. I was responsible for the “functional programming” component of the module from the start, to supplement the first year module on Functional Programming and Logic. Initially this was taught in Miranda; in recent years, Haskell was used. Textbooks by Simon Thompson were used throughout – since 1999, the second edition of “Haskell: The Craft of Functional Programming” [1]. This edition differs from the previous one and the Miranda textbook by starting with a combinator library for “pictures” before the traditional build up through topics such as recursion and list algorithms. A notation from this book used here is that “>” means “evaluates to”.

By the time the students started on their CO312 Haskell case studies, they would typically have covered:

- basic types Bool, Int and Char with operations;
- tuples; lists, list comprehensions, list library functions;
• list programs through pattern matching and recursion.

They would have seen library functions on lists, but often not the higher order ones, and without much awareness of parametric polymorphism; they would not have seen algebraic types, abstract types or type classes.

The module was assessed through 50% coursework and 50% exam. A series of seven lectures would build up to the coursework assessment, by offering preparatory exercises, to be solved individually or in a class. Correct and incorrect solutions to these exercises would be discussed in the following lecture. Students would then have a few weeks to do the final assessment. The exam would consist of a model solution to this assessment (also handed out well in advance) with a number of questions related to modifications or extensions of the code, e.g. based on changes to the problem or the suggested solution.

3 What is TSP?

Given a list of towns to be visited and a method of determining the distances between towns, the travelling salesman problem is to find a route that visits all required towns, and of those routes, the shortest one. The particular variant we are looking at here assumes those distances are represented in a matrix, and that we need to start at one of the towns and end up there again after visiting all the others.

This is a known hard problem – it is a so-called “NP-complete” one, which implies that there is no known algorithm that is guaranteed to always give the best possible solution in a reasonable amount of time. Furthermore, if anyone finds such an efficient algorithm for TSP, it can be modified to solve a large class of similarly difficult problems.

4 Recursion over lists

You will have seen the following three ways of writing programs over lists:

• list comprehensions, for example:

    \[ \text{[dist \  y \ x | \ y <- ys, y/=x]} \]

\(^1\)In technical terms, an algorithm whose running time is bounded by a polynomial in the size \( n \) of the input – here, the number of towns. The brute force TSP algorithm runs in time proportional to \( n! \), which is not bounded by any polynomial.
Only functions returning a list can be computed by using (only) a list comprehension.

- Higher order functions such as \texttt{map}, \texttt{filter}, \texttt{fold1} usually lead to short programs, but how clear these appear depends on experience.

- With explicit recursion, anything is possible. Depending on the structure of your data, you might use \textit{patterns} or \textit{guards} and conditionals to distinguish the various cases (for example, when to make a recursive call and when not to).

The exercises in this document assume only limited experience with functional programming, for this reason we look at explicit recursion rather than higher order functions, giving a quick rundown of the common patterns to use in recursions over lists.

The standard pattern has a case for \texttt{[]} and one for \texttt{(x:xs)} in terms of a recursive call using \texttt{xs}:

\[ f \ [ \ ] = 0 \]
\[ f \ (x:xs) = x + f \ xs \]

For functions which only make sense on non-empty lists, the standard pattern is:

\[ g \ [x] = x \]
\[ g \ (x:xs) = \text{max} \ (g \ xs) \ x \]

However, sometimes both the empty \textit{and} the singleton list need to be singled out.

\[ h \ [ \ ] = 0 \]
\[ h \ [x] = x + 1 \]
\[ h \ (x:xs) = 1 + h \ xs \]

The next few patterns involve looking at the first \textit{two} elements of the list. The important decision there is whether, in the recursive call, one or two elements are removed from the list.

An example of losing the first two elements is as follows. Note that this also implies that you need special cases for \textit{both} empty and singleton list (or alternatively: for lists of length 1 and 2).

\[ \text{odds} \ [ \ ] = [] \]
\[ \text{odds} \ [x] = [x] \]
\[ \text{odds} \ (x:y:xs) = x: \text{odds} \ xs \]
The list of differences between adjacent elements needs to preserve the second element as the next head of the list in the recursive call:

\[
\text{difs} \left[ \right] = \left[ \right] \\
\text{difs} \left[ x \right] = \left[ \right] \\
\text{difs} \left( x:y:xs \right) = (x-y)\cdot\text{difs} \left( y:xs \right)
\]

In the above examples, it is convenient that you can write \((x:y:xs)\) for \((x:(y:xs))\).

5 Type aliases

Thompson [1, section 5.1] says: “we can give names to types in Haskell, so that types are easier to read”. That is one aspect of their use – we might also look at it as giving an “implementation” of a more abstract type that we need in our program. For example, if we needed to discuss \textit{time}, we might say

\[
\text{type Time} = (\text{Int}, \text{Int}, \text{Int})
\]

intending to interpret the first number as hours, the second as minutes, and the third as seconds. Another example might be representing sets of numbers by lists, i.e.,

\[
\text{type SetInt} = [ \text{Int} ]
\]

This sort of thing can be done in a more general way with type parameters; Haskell also offers more advanced ways of “implementing abstract data types”. This mechanism will suffice for now, though.

If we implement more “abstract” values this way, we need to remember the relation between those values and the Haskell values, e.g., for sets,

\[
[\ ] \text{ represents } \emptyset \\
[1] \text{ represents } \{1\} \\
[3,1] \text{ represents } \{1,3\}
\]

This relation may be partially documented in definitions. The names of the functions and constants defined form part of the \textit{interface} to the type, e.g.,

\[
\text{empty} :: \text{SetInt} \\
\text{empty} = [\ ]
\]

\[
\text{union} :: \text{SetInt} \rightarrow \text{SetInt} \rightarrow \text{SetInt} \\
\text{union} \; xs \; ys = xs \; ++ \; ys
\]
noon :: Time
noon = (12,0,0)

hours, mins, secs :: Time -> Int
hours (h,m,s) = h
mins (h,m,s) = m
secs (h,m,s) = s

Also, we might define a straight type alias such as

type Town = Int

With this definition, Haskell will not find errors when you use a Town where
an Int is expected or vice versa. It serves mainly as documentation (adding
towns probably is not meaningful), and may make it easier to change your implementation of the newly defined type.

There are two main issues with this sort of implementation: junk and

confusion. (These are both technical terms!) Neither of these needs to be
avoided, but they need to be taken into account in programs.

5.1 Confusion

Confusion is when multiple Haskell values represent the same abstract value. For example, [1,1,3] and [1,3] also represent {1,3}. There is no confusion for Time unless one took (0,0,0) and (24,0,0) both to represent midnight.

For SetInt, we could reduce confusion by only considering lists without duplicates (some functions are defined in List.hs for this), or even sorted lists, but that complicates the code for the set operations. In general, for representing sets that way, we would need to be able to define equality and ordering on the elements.

The above remarks refer to type class constraints Ord and Eq. of course.

The consequence of having confusion in your data type is that == does not tell you much about equality of abstract values — it may return False for conceptually equal values.

For SetInt, equality might be defined by

equal :: SetInt -> SetInt -> Bool
equal xs ys = and [ elem x ys | x <- xs ] &&
          and [ elem y xs | y <- ys ]
(each of the and expressions represents an inclusion of sets).

With type classes this will show up as an instance of Eq.

5.2 Junk

Junk is when some Haskell values do not represent any abstract value. If
we do not insist on ordering and removal of duplicates, there is no junk for
SetInt. For Time, there is plenty:

\[
\text{nnn} :: \text{Time} \\
\text{nnn} = (0,9,99)
\]

is dubious already: 9 minutes and 99 seconds? I own a microwave oven
which treats this as confusion rather than as junk, and will happily heat for
10 minutes and 39 seconds if I enter '999'. However, it is even harder to
interpret

\[
\text{zntmohts} :: \text{Time} \\
\text{zntmohts} = (0,-12,-127)
\]

(14 minutes and 7 seconds before midnight!?), but Haskell will not produce
a type error for this.

The consequence of having junk in your data type is that you might need
to check for validity of inputs. If you're writing a function on such a type,
you need to make sure that you don't test it with values which are "junk".
One way of preventing that involves validation functions, e.g.:

\[
\text{validTime} :: \text{Time} \rightarrow \text{Bool} \\
\text{validTime} (h,m,s) \\
\quad = h>=0 \&\& m>=0 \&\& s>=0 \&\& h<24 \&\& m<60 \&\& s<60
\]

5.3 Two-dimensional arrays

A type which we will be using later is that of "matrices", or two-dimensional
arrays, containing numbers. You can also think of them as rectangular areas
in a spreadsheet. We will borrow mathematical notation for them, writing

\[
\begin{pmatrix}
1 & 2 & 6 \\
3 & 4 & 8
\end{pmatrix}
\]

for a 2-by-3 matrix containing, in the first row, numbers 1, 2 and 6; in the
second row, numbers 3, 4 and 8. (Alternatively, in the first column it has
numbers 1 and 3, etc.) Also,

\[
\begin{pmatrix}
1 \\
3 \\
17
\end{pmatrix}
\]

is a matrix with three rows, containing a single element each (i.e., it has one column). Thus, it is different from

\[
\begin{pmatrix}
1 & 3 & 17
\end{pmatrix}
\]

which has a single row and three columns.

These are a bit like Java int [][] [], and indeed we will implement them as [[Int]], but we will insist on them being “rectangular”, i.e., every row should have the same number of elements.

type Matrix = [ [ Int ] ]

We take Matrix as list of rows, i.e.

\[
[[1,2] , [3,4]] \text{ represents } \begin{pmatrix} 1 & 2 \\ 3 & 4 \end{pmatrix}
\]

\[
[[1,4,2]] \text{ represents } \begin{pmatrix} 1 & 4 & 2 \end{pmatrix}
\]

\[
[[1],[4],[2]] \text{ represents } \begin{pmatrix} 1 \\ 4 \\ 2 \end{pmatrix}
\]

There is junk for this representation of matrices: some concrete values do not represent abstract values, e.g.

\[
[[1] , [3,4]] \text{ represents } \begin{pmatrix} 1 \\ 3 & 4 \end{pmatrix}
\]

There is also (less obviously) confusion: for the matrix containing no values at all: how many rows and columns might it have?

\[
[[[]] = [] = [ [], [] ]
\]

\[
1 \times 0 = 0 \times 0 = 2 \times 0
\]

5.4 Indexing

For getting a value out of a matrix, indexing is the natural approach. Indices on a list xs run from 0 to (length xs)-1.

\[
[ [ 1, 2, 6 ], [ 3, 4, 8 ] ] !! 1 !! 2 = 8
\]

Note: Indexing is often the wrong method of dealing with lists.
Do not get tempted to use indexing for iterative/recursive/loop programs over lists (unlike for Java arrays). Recursion over lists is usually much better, less error prone. Compare the following two programs for merging sorted lists:

```haskell
merger :: [Int] -> [Int] -> [Int]
merger xs [] = xs
merger [] ys = ys
merger (x:xs) (y:ys)
  | x < y = x : merger xs (y:ys)
  | otherwise = y : merger (x:xs) ys

merge2 xs ys
  = merger 0 0
  where
    merger ix iy
      | ix == length xs && iy == length ys = []
      | ix == length xs = ys ++ iy : merger ix (iy+1)
      | iy == length ys = xs ++ iy : merger (ix+1) iy
      | (xs !! ix) < (ys !! iy) = xs ++ iy : merger (ix+1) iy
      | otherwise = ys ++ iy : merger ix (iy+1)
```

5.5 Exercises

1. Define a function

   ```haskell
   numRows :: Matrix -> Int
   ```

   giving the number of rows of a matrix. (Remember a matrix is a list of rows.)

2. Define a function

   ```haskell
   isEmpty :: Matrix -> Bool
   ```

   The following definition is not good enough:

   ```haskell
   isEmpty m = m == []
   ```

   Why? There is confusion.
3. Define a function

\[\text{numCols :: Matrix} \rightarrow \text{Int}\]

giving the number of columns of a matrix (still a list of rows, unfortunately). Ensure that

\[\text{numCols []}\]

does not give an error message.

\textit{Different answers to this will lead to different results for “junk” inputs, could explain that this is OK.}

4. Define a function

\[\text{rectangular :: Matrix} \rightarrow \text{Bool}\]

that checks whether all rows in the matrix have the same length.

You could do this by defining an auxiliary function \text{allequal} that checks whether all numbers in a list are equal.

5. (Advanced) What is wrong with the following program for matrix transposal (swapping rows and columns, e.g., \([[1,2],[3,4]]\) to \([[1,3],[2,4]]\))?

\[\text{tp :: Matrix} \rightarrow \text{Matrix}\]
\[\text{tp xss =}\]
\[| \text{isEmpty xss} = []\]
\[| \text{otherwise} = [ a:as \mid (a,as) <- \text{zip (head xss)} (\text{tp (tail xss))}]\]

\[\begin{array}{cc}
\text{xss} & \rightarrow \\
\text{xss} & \text{sx} \\
\end{array}\]

\textit{There is a function transpose in List.hs.}
6 Maximum under a function

These exercises have also been used as coursework rather than voluntary exercises. There are many instances of this problem in later exercises, so it is good for the students to be aware of ways of solving it both correctly and efficiently.

The function max computes the maximum of two numbers, and maximum gives the maximum of a list. They might have been defined as follows:

\[
\begin{align*}
\text{max} & \colon \text{Int} \to \text{Int} \to \text{Int} \\
\text{max a b} & \mid a \leq b = b \\
& \mid \text{otherwise} = a \\
\text{maximum} & \colon [\text{Int}] \to \text{Int} \\
\text{maximum [x]} & = x \\
\text{maximum (x:xs)} & = \text{max x (maximum xs)}
\end{align*}
\]

Note that the latter does not work for empty input, and is characterised by the fact that if

\[
\text{maximum [x1,x2,\ldots,xn]} \sim x_i
\]

then

\[
\text{elem x1 [x1,x2,\ldots,xn]} \sim \text{True} \\
x_i \geq x_1 \\
x_i \geq x_2 \\
\ldots \\
x_i \geq x_n
\]

If we have a function \( f \), and a list \( xs \), we could find the member of \( xs \) for which \( f \) is maximal by taking the \text{maximum} of all \( x \) in \( xs \) – but if we do this in the obvious way, we will have lost the information “which” \( x \) this maximum belonged to. Here, we work out a strategy to avoid this problem by keeping pairs of \( x \) and \( f x \) – a “tupling” strategy. It is all presented here as an exercise concerning a meaningless function \( f \), but the strategy can be reused often in the various TSP programs.

Given is a function \( f \) defined by

\[
\begin{align*}
f & \colon \text{Int} \to \text{Int} \\
f x & = 8*(x^{-2})-2*(x^{-3})+4*x-17
\end{align*}
\]
A more advanced version has the function \( f \) as a parameter, passed to all relevant functions.

The final aim is to write a relatively efficient\(^2\) function

\[
\text{maxf} :: [\text{Int}] \rightarrow \text{Int}
\]
such that if \( \text{maxf} \ [x_1, \ldots, x_n] = x_i \) then

\[
\text{elem} \ x_i \ [x_1, \ldots, x_n] \equiv \text{True}
\]
\[
f \ x_i \geq f \ x_1
\]
\[
f \ x_i \geq f \ x_2
\]
\[
\ldots
\]
\[
f \ x_i \geq f \ x_n
\]

This is a generalisation of the definition of maximum, where we do not compare the elements of the list but their images under a given function \( f \). It will be constructed bottom-up in the next few exercises.

6. Define a function

\[
\text{tupleWithf} :: [\text{Int}] \rightarrow [(\text{Int}, \text{Int})]
\]
such that

\[
\text{tupleWithf} \ [x_1, x_2, \ldots, x_n] = [(f \ x_1, x_1), (f \ x_2, x_2), \ldots, (f \ x_n, x_n)]
\]

*The particular order \((f \ x, x)\)* is chosen to give students the option of exploiting the instance of \textit{Ord} for tuples. A pitfall here is mixtures of patterns and recursion/comprehension which ignore the first element of the input.

7. Define a function

\[
\text{maxFirst} :: (\text{Int}, \text{Int}) \rightarrow (\text{Int}, \text{Int}) \rightarrow (\text{Int}, \text{Int})
\]

which returns, of its two input tuples, the one whose first component is the highest. For example,

\[
\text{maxFirst} \ (1, 2) (3, 4) \rightarrow (3, 4)
\]
\[
\text{maxFirst} \ (5, 3) (4, 6) \rightarrow (5, 3)
\]

For \( \text{maxFirst} \ (0, 1) (0, 2) \) it does not matter whether your program returns \((0, 1)\) or \((0, 2)\).

\(^2\)In the sense that it does not evaluate \( f \ x \) twice unless \( x \) occurs more than once in the input list.
8. Define a function

    maximumFirst :: [(Int,Int)] -> (Int,Int)

    which returns, from its list of input tuples, the tuple whose first component is highest. You may assume that the input list is non-empty.

    Pitfall: defining it for empty lists anyway, and then picking the wrong unit of \texttt{max} on \((\text{Int},\text{Int})\).

9. Using the functions defined or otherwise, define the function \texttt{maxf} (described earlier) which returns the element of its non-empty input list for which \(f\) is maximal.

    Most solutions which do not use tupling will end up computing \(f\) on average twice for all elements.

Questions 10 and 11 are two variations of this.

10. Using the functions defined above or otherwise, define the function \texttt{minf} which returns the element from its non-empty input list for which \(f\) is minimal.

    Two kinds of reuse possible: by analogy, or using reflection in the x-axis.

11. Using the functions defined above or otherwise, define the function

    \texttt{maxIndexf :: [Int] -> Int}

    such that if

    \texttt{maxIndexf xs \rightarrow i}

    for non-empty \(xs\), then

    \(0 \leq i < \text{length} \: xs\)

    \(f \: (xs!!i) \geq f \: (xs!!0)\)

    \(f \: (xs!!i) \geq f \: (xs!!1)\)

    \(\ldots\)

    \(f \: (xs!!i) \geq f \: (\text{last} \: xs)\)
i.e., it returns the index of the element of its input list for which f is maximal.

*A modified tupling strategy is more efficient but possibly less clear than looking up the index of \texttt{max f xs} in \texttt{xs} (\texttt{List.hs provides a number of ways, some involving Maybe}).*

7 Distance Matrices

For TSP, we need to represent distance information somehow. A natural way of doing this is in a *distance matrix*. A sample distance matrix is the following:

\begin{center}
\begin{tabular}{|l|c|c|c|c|}
\hline
 & 0 & 1 & 2 & 3 \\
\hline
Faversham = 0 & 0 & 7 & 12 & 10 \\
Whitstable = 1 & 7 & 0 & 5 & 7 \\
Herne Bay = 2 & 12 & 5 & 0 & 10 \\
Canterbury = 3 & 10 & 7 & 10 & 0 \\
\hline
\end{tabular}
\end{center}

This could be represented in Haskell by

```
[ [0,7,12,10], [7,0,5,7], [12,5,0,10], [10,7,10,0] ]
```

where we might store the information about which town names correspond to which indices separately.

*Haskell arrays would be an alternative if they had been taught to the students already. Maybe even with constant retrieval cost?*

Distance matrices have various properties. First, they are *square*: they have as many rows as columns, as we record all distances between a fixed set of towns. Second, they are likely to be *symmetric*: the distance from A to B is the same as the distance from B to A. As a consequence, most road atlases only present half of the distance matrix information in a triangular table\textsuperscript{3}. Third, the distance between a town and itself is always 0. These distances can be found on what is called the *main diagonal*, which consists of all positions \texttt{m!i!!i}. Finally, the distance of getting from A to B via C, i.e. the distance from A to C plus that from C to B, is never less than the

\textsuperscript{3}The AA road atlas for the UK has such a triangular table – as a consequence, it does not represent the fact that crossing the Severn near Bristol is a different distance going into Wales or going into England.
recorded distance from A to B “directly”, i.e., “a detour is never shorter”. This is represented by the “triangular property”:

\[ \forall i, j, k : d_{i\rightarrow j} \leq d_{i\rightarrow k} + d_{k\rightarrow j} \]

12. Define a function

\[ \text{square} :: \text{Matrix} \rightarrow \text{Bool} \]

which checks that a matrix is rectangular, and has equal numbers of rows and columns.

*As in Exercise 3, there is a risk of a run-time error for empty list input.*

13. Define a function

\[ \text{symmetric} :: \text{Matrix} \rightarrow \text{Bool} \]

which tests if the matrix is symmetric (you may use \text{transpose} from \text{List.hs}).

14. Define a function

\[ \text{all0} :: [\text{Int}] \rightarrow \text{Bool} \]

which checks whether all numbers in a list are 0.

*Some students would disagree with logicians on the answer for the empty list, as do many solutions using \text{nub} or recursion. This exercise is also likely to provide illustrations for gratuitous use of guards or conditionals (rather than \text{Boolean operators}) and phrases such as \text{== True}.*

15. Define a function

\[ \text{maindiag} :: \text{Matrix} \rightarrow [\text{Int}] \]
which returns the main diagonal of a square matrix. For example, the
main diagonal of
\[
\begin{pmatrix}
1 & 2 & 3 \\
4 & 5 & 6 \\
7 & 8 & 9
\end{pmatrix}
\]
is \([1,5,9]\).

Here, and the next exercise, is when indexing is useful; should probably give
bonus marks for correct recursive solutions. Non-square matrices are "junk"
here so the run-time errors they may cause are not an issue.

16. Define a function

\[
\text{triangle :: Matrix -> Bool}
\]

which checks a matrix for the triangular property.

8 Brute Force Travelling Salesman

The exact solution to the TSP can be obtained by generating all possible
permutations of the list of towns to be visited, and then selecting the one
which has the lowest cost. The cost is relative to a given distance matrix.
We call this a "brute force" solution because Hugs crashes, if we do this
naively, with a control stack overflow, at a list of about 8 towns (40,320
possible tours). A slightly less naive version (see Section 8.4) still takes too
long to compute any output for about 12 towns.

8.1 Permutations

The permutations of a list are all lists with all the same elements occurring
equally often (and no others). In other words, a permutation is any list you
can get by 0 or more times swapping elements in the list (you could imagine
that this does not make for an efficient algorithm though!).

The permutations of \([1,3]\) are \([[1,3],[3,1]]\). Those of \([1,2,3]\) are:
\([[1,2,3],[2,1,3],[2,3,1],[1,3,2],[3,1,2],[3,2,1]]\). More interestingly,
the permutations of \([1,1,3]\) are \([[1,1,3],[1,3,1],[3,1,1]]\). However,
as this is a set we may actually include some of these elements more
than once in the result, by not taking into consideration that 1 occurs twice.
(Note: \([]\) is not really Haskell, in real code we would have to use \([\ ]\)
instead.)

The approach we will take to defining a function to generate permuta-
tions
perms :: [Int] -> [[Int]]

is a “psychic bottom-up” solution: we’ll first define an auxiliary function
for no reason at all, and then it will turn out to be an essential piece of the
required function. (Though, as it happens, it will come in handy elsewhere,
too.)

The auxiliary function is

ins :: Int -> [Int] -> [[Int]]

such that ins y xs is the set (list) of all possible ways of inserting y into
xs, e.g.

ins 3 [2,4,5]
~>
[ [3,2,4,5], [2,3,4,5], [2,4,3,5], [2,4,5,3] ]

(or these same lists in some different order – duplicates, if any, need not be
removed, so ins a [b,c] will end up having three elements even when a, b
and c are all the same.)

We solve this by answering a list of questions. For different Haskell
problems, you might ask yourself similar questions in order to get started
on solutions.

- **What is the type of ins 2 []?**
  The type of any result of ins is [[Int]] so that must be the type here
  as well.
  This may have looked too obvious, but asking yourself this stops you
  from forgetting some square brackets later, hopefully.

- **How many elements has ins 2 []? Why?**
  In how many ways can you insert an element into an empty list? Only
  one: [2] ....

- **So what is ins x []?**
  So this is changing 2 to x. Probably the result is [x], but that’s not
  of type [[Int]]. So, the result must be [[x]] – the list of different
  ways of inserting x into an empty list is a singleton list containing the
  only way of doing it:

  ins x [] = [[x]]
• **Does ins 3 [4,5] relate to ins 3 [2,4,5] ? How?**
  Taking some “sensible” ordering of the various ways of inserting, we might have

  \[
  \text{ins 3 [4,5]} \rightarrow \quad [\text{[3,4,5]}, \quad [4,3,5], \quad [4,5,3]]
  \]
  \[
  \text{ins 3 [2,4,5]} \rightarrow [\text{[3,2,4,5]}, \quad [2,3,4,5],[2,4,3,5],[2,4,5,3]]
  \]

  (Nicely lined up!) So, going from \text{ins 3 [4,5]} to \text{ins 3 [2,4,5]}, we put the new value 2 in front of each result, and add one more list: the one we get by inserting 3 before 2.

• **So is there a general recursive relation?**
  To insert \(x\) into \(y:ys\), we either put \(x\) right before \(y:ys\), or we insert \(x\) into \(ys\), and put \(y\) in front of the result:

  \[
  \text{ins } x \ (y:ys) = (x:y:ys) : [\quad y:xs \mid xs \leftarrow \text{ins } x \ ys \ ]
  \]

  or, using ++ rather than : and \text{map} instead of the list comprehension

  \[
  \text{ins } x \ (y:ys) = [\quad [x,y]+ys \ ] \quad \text{++} \quad \text{map} \ (y:) \ (\text{ins } x \ ys)
  \]

  or, using an auxiliary function to do the “put \(y\) in front of each ...” bit

  \[
  \text{ins } x \ (y:ys) = (x:y:ys) : \text{consall } y \ (\text{ins } x \ ys)
  \]

  \[
  \text{consall } y \ [] = []
  \]

  \[
  \text{consall } y \ (xs:xs) = (y:xs) : \text{consall } y \ xss
  \]

  Having defined \text{ins}, we can now define

  \[
  \text{perms} : \int \rightarrow [[\int]]
  \]

  that gives all permutations of a list. We will not worry about removing duplicates or the ordering of results, as before.

• **What is the type of \text{perms} [] ?**
  Again, it must be [[\int]] like any other \text{perms} result.

• **How many elements does \text{perms} [] have? Why?**
  There’s only 1 way of (possibly) reordering an empty list ...

• **So what is \text{perms} [] ?**
  Not [] as that has the right type (as an empty list of lists), but too few elements (it has none).
perms [] = [[]]

- What is \texttt{perms [2,3]}?  
  That will be a list containing the elements \([2,3]\) and \([3,2]\), in some order (and possibly multiple times).

- Which elements of \texttt{perms [1,2,3]} relate to which elements of \texttt{perms [2,3]}?  
  \texttt{perms [1,2,3]} has six different elements: \([1,2,3]\), \([1,3,2]\), \([2,1,3]\), \([2,3,1]\), \([3,1,2]\), \([3,2,1]\), which may occur in any order. Three of these have 3 before 2, and three have 2 before 3. The latter are \([1,2,3]\), \([2,1,3]\), \([2,3,1]\).

- What is \texttt{ins 1 [2,3]}?  
  It is that same list \([1,2,3]\), \([2,1,3]\), \([2,3,1]\).

- In general, can you find a way to relate \texttt{perms (x:xs)} to \texttt{perms xs}, using \texttt{ins}? You need to get the types right – \texttt{ins} returns a list of lists.  
  \texttt{perms (x:xs) = concat [ins x ys | ys <- perms xs]}

  Without the \texttt{concat}, the right hand side is a \texttt{[[[Int]]]} rather than a \texttt{[[Int]]} as required. (Note that \texttt{concat (ins x ys) | ys <- perms xs} also corrects this type error, but gives a different result.)

  In words: once we know how to permute \texttt{xs}, we can permute a list with an additional element \texttt{x} by putting it in any possible place (using \texttt{ins}) in every possible permutation of the shorter list \texttt{xs}.

8.2 Cheap tours

A town is an integer (an index into a distance matrix), a tour is a list of towns, interpreted in a circular way: \([1,2,3]\) represents going from 1 to 2 to 3 to 1. As a consequence, there is no fundamental difference between the tours \([1,2,3,4]\) and \([3,4,1,2]\) ("confusion").

\texttt{type Town = Int  
type Tour = [Town]}

If the type \texttt{[Town]} is used below, it denotes a list of towns that should not be interpreted as a circular tour – but possibly even as a \texttt{set} of towns.

17. Define a function
cost :: Matrix -> Tour -> Int

such that

cost dist [a,b,c,d]
"> dist!!a!!b + dist!!b!!c + dist!!c!!d + dist!!d!!a

If you want to do this recursively, you might want to define an auxiliary function to do most, but not all, of the work. This is because cost dist [a,b,c,d] does not rely on cost dist [b,c,d]; the latter uses dist!!d!!b which is not included in the former!

There is no need to define cost for the empty tour.

18. Given a function cost as above, define a function

cheapest :: Matrix -> [Tour] -> Tour

which, given a list of tours (so a list of lists of numbers), select the (a) cheapest of them using the function cost and the given matrix.

(Use the ideas from Section 6!)

*Using sortBy is not a great idea as it does not implement tupling and as a consequence leads to recomputed cost. Students may not realise the positive consequences that lazy evaluation has on the efficiency of using sorting in place of minimum, anyway.*

### 8.3 Using the brute force program

With these ingredients, this is the “brute force” TSP program:

tsp :: Matrix -> [Int] -> Tour

\[
tsp m \mathrm{xs} = \mathrm{cheapest} \ m \ (\text{perms } \mathrm{xs})
\]

This is not the most efficient way of computing the exact solution, but it is a simple one. The more complicated “branch-and-bound” algorithm will do much better on average, though not necessarily in the worst case.

A Haskell module containing a definition of a sample 50x50 distance matrix called \( \text{dm} \) is at \texttt{www.cs.kent.ac.uk/people/staff/eab2/tsp/DM.hs}. The next few exercises assume you have downloaded this, and created a module \texttt{Brute.hs} containing the following lines:
module Brute
where
import DM

plus the definition of tsp as given above and all necessary auxiliary functions,
given above or developed in previous exercises.

19. What is \texttt{tsp dm [2,4,6,8,10]}?

20. Looking at \texttt{names :: [String]} in \texttt{Dm.hs}, what does the result to the
    previous question represent in terms of travel in the UK?

21. What is the highest value of \texttt{n} such that

\texttt{tsp dm [1..n]}

is computed by Hugs without error messages?

8.4 Small Improvements

This section lists two small improvements on the brute force TSP. It is not
essential to use them or understand them, but they may allow the exact
TSP solution to be computed for a slightly larger number of towns.

For the purpose of the TSP on a symmetric distance matrix, all permuta-
tions of 3 (or fewer) towns are equivalent: there is only one triangle
connecting the three towns, and it does not matter in which order it is
traversed. So instead of the function \texttt{perms}, we could use

\begin{verbatim}
perms' (x:xs)
  | (length xs) < 3 = [(x:xs)]
  | otherwise = concat [ ins x p | p <- perms' xs]
perms' [] = [[]]
\end{verbatim}

This reduces the number of possibilities to be considered by a factor of 6.

Also, tsp suffers from control stack overflows in Hugs: too many compari-
sons between costs are postponed by the lazy evaluation strategy. The
following variant of cheapest alleviates this problem. It is not necessary to
understand this code – just that \texttt{cheapest'} can be used in place of \texttt{cheapest}
(and \texttt{foldl1'} in place of \texttt{foldl1}) in this context.

\begin{verbatim}
cheapest' m xs = snd (foldl1' (<) [(cost m x, x) | x <- xs])
foldl1' f (x:xs) = foldl' f x xs
foldl' f a [] = a
foldl' f a (h:t) = (foldl' f $! f a h) t
\end{verbatim}
22. Repeat Exercise 21 with `cheapest' for `cheapest' and `perms' for `perms'.

9 Nearest Neighbour Heuristic

A **heuristic** algorithm for an optimisation problem is a method that gives a good (but not necessarily the best) solution (relatively) quickly. One obvious heuristic for TSP is to start somewhere, and then always to pick the nearest unvisited town.

23. Define a function

\[
\text{nearest} :: \text{Matrix} \to \text{Town} \to \text{[Town]} \to \text{Town}
\]

such that \( \text{nearest} \ \text{dm} \ x \ \text{ys} \) gives the town from \( \text{ys} \) which is nearest to the town \( x \) according to distance matrix \( \text{dm} \).

24. Look up the function `delete` in `List.hs`. What does it do?

25. Use these to define a function

\[
\text{nn} :: \text{Matrix} \to \text{[Town]} \to \text{Tour}
\]

such that \( \text{nn} \ \text{dm} \ x\text{s} \) returns a tour containing all towns from \( x\text{s} \), starting from the first town in \( x\text{s} \) and then picking the town nearest to the last town visited at every step.

It is probably useful to define an auxiliary function which has the same functionality but takes in the tour constructed so far as an extra argument, i.e.

\[
\text{nnLoop} :: \text{Matrix} \to \text{[Town]} \to \text{Tour} \to \text{Tour}
\]

*The accumulating argument (or at the very least: its last town) is necessary to determine the next town to visit. A direct recursive version is likely to construct the tour “inside-out” using the wrong selection criterion.*

*A somewhat analogous problem is the following. Assume we want to sum all numbers in a list, adding from left to right. The following function does not do that:*

\[
\text{sumall1} \ [x] = x \\
\text{sumall1} \ (x:xs) = x + \text{sumall1} \ xs
\]
because it doesn’t actually sum from left to right, but from right to left –
the first addition performed in sumall1 [3, 8, 9] is 8+9. A version using a
“result so far” can sum easily from left to right:

sumall2 xs = sumaux 0 xs
sumaux sumsofar [] = sumsofar
sumaux sumsofar (x:xs) = sumaux (sumsofar + x) xs

10 Combining tour segments

A second heuristic works by considering “segments” – partial tours each
containing some of the towns. The whole tour is obtained by starting with
each town in a single segment, and then gradually combining the segments
until a single segment is formed containing all the towns. The obvious
decision for combining segments is to pick those which are “close” to each
other. For example:

[0, 1, 2, 3, 4, 5, 6, 7, 8, 9]

[0, 1, 2, 4, 3]

[0, 1, 3, 4, 2]
A Segment is a list of Towns, representing a partial tour (in order) – not interpreted as a “circular” sequence. We will only consider non-empty Segments.

\[ \text{type Segment} = [\text{Town}] \]

A Segmentation is a list of Segments such that it contains all relevant Towns, each in exactly one of the segments. (In intermediate results some towns may be missing.)

\[ \text{type Segmentation} = [\text{Segment}] \]

The segmentation

\[ [ [1,2], [0], [3,4] ] \]

could be said to represent a collection of possible tours of the towns [0..4], namely those where where 1 is visited immediately before or after 2, and similarly for 3 and 4.

Some of the next few exercises are intended in the first place to get used to the Segmentation type, and have very simple answers.

26. Define a function

\[ \text{size :: Segmentation} \rightarrow \text{Int} \]

which reports the number of towns that occur in a segmentation. (E.g. the segmentations in the example all have size 5.)

27. Define a function

\[ \text{endpoints :: Segmentation} \rightarrow [\text{Town}] \]

which returns the list of all endpoints (i.e., first or last elements) of the given segmentation, without duplicates. For example,
endpoints [ [1,2,4,5], [3,6,7], [0], [8,9]]

=> [ 1, 5, 3, 7, 0, 8, 9 ]

(Note that 0 occurs only once.)

Allowing duplicates and then removing them is inefficient; segmentations do not contain duplicate towns.

28. Define a function

\[
\text{element} :: \text{Town} \rightarrow \text{Segmentation} \rightarrow \text{Bool}
\]

which reports whether the given town occurs anywhere in the segmentation.

\textit{Blind use of a library function is likely to lead to a type error.}

29. Define a function

\[
\text{splitOnTown} :: \text{Town} \rightarrow \text{Segmentation} \rightarrow (\text{Segment},\text{Segmentation})
\]

which splits the segmentation into the segment which has the town as an endpoint, and all the other segments. E.g.,

\[
\text{splitOnTown} 2 [ [1,2], [3], [0] ] \rightarrow ([1,2], [[3],[0]])
\]

You may assume that the town is an endpoint of one of the segments in the given segmentation.

(This is one example where an intermediate \text{Segmentation} does not contain all the relevant towns.)

30. Define a function

\[
\text{otherEnds} :: \text{Segmentation} \rightarrow \text{Town} \rightarrow [\text{Town}]
\]

such that \text{otherEnds} ss t returns all endpoints of ss, except for the endpoints of the segment that t is an endpoint of. For example,

\[
\text{otherEnds} [ [1,2,4,5], [3,6,7], [0], [8,9] ] 7
\rightarrow [ 1, 5, 0, 8, 9 ]
\]

31. Define a function
initial :: [Town] -> Segmentation

such that initial ts returns a Segmentation containing all of ts in
length ts separate segments.

32. Define a function

   complete :: Segmentation -> Bool

which reports whether the segmentation is complete, in the sense that
it puts all its elements in a single segment.

33. Define a function

   splitOnTowns :: (Town, Town) -> Segmentation
       -> (Segment, Segment, Segmentation)

such that

   splitOnTowns (x, y) ss

returns a triple: the segment of ss which has x as an endpoint, the
segment of ss which has y as an endpoint, and all the other segments
of ss. E.g.,

   splitOnTowns (2, 3) [ [1,2], [3], [0] ]
   ~> ([1,2], [3], [[0]])

You may assume that the towns are endpoints of two different segments
in the given segmentation.

34. Define a function

   merge :: (Town, Segment) -> (Town, Segment) -> Segment

which merges two segments. In a call

   merge (p1, s1) (p2, s2)
you should assume that \( p_1 \) is an endpoint of \( s_1 \), and \( p_2 \) of \( s_2 \). The result of this call should be a segment obtained by, if necessary, reversing one of \( s_1 \) and \( s_2 \), and concatenating them in such a way that \( p_1 \) is next to \( p_2 \) in the resulting segment.

The acceptable outcomes for \( \text{merge} (2, [1, 2]) \) \( (3, [3]) \) are \([1, 2, 3]\), also \([3, 2, 1]\), but not \([3, 1, 2]\); for \( \text{merge} (4, [1, 4]) \) \( (6, [8, 6]) \) they are \([1, 4, 6, 8]\) or \([8, 6, 4, 1]\), but not \([1, 4, 8, 6]\).

35. Define a function

\[ \text{join} :: (\text{Town}, \text{Town}) \to \text{Segmentation} \to \text{Segmentation} \]

such that for a call

\[ \text{join} (x, y) \, ss \]

(\( x \) and \( y \) may be assumed to be endpoints of different segments of \( ss \))
the result is a segmentation which is identical to \( ss \) except that the segments containing \( x \) and \( y \) have been merged.

So a possible result of \( \text{join} (3, 4) \, [ [0, 3] , [1, 2, 4] ] \) is
\([ [0, 3, 4, 2, 1] ] \), and \( \text{join} (0, 3) \, [ [0], [1, 2], [3], [4] ] \)
could give \([ [0, 3], [1, 2], [4] ] \) (see pictorial example).

A slightly more efficient overall solution can be obtained by not using the functions suggested, as they lead to repeated retrieval of the selected segments.

The functions defined so far could be used to implement any TSP heuristic that builds up the tour link by link. The determining decision is \textit{which} two towns to use for the next \textit{join}.
Our strategy is: “furthest town first”.

36. Define a function

\[ \text{howfar} :: \text{Town} \to \text{Matrix} \to \text{Segmentation} \to \text{Int} \]

which returns the sum of the distances of a given town to the endpoints of all other segments in the segmentation.

37. Define a function
furthest :: Segmentation -> Matrix -> Town

which returns the endpoint which has the largest how far value of all endpoints.

38. Define a function

\texttt{closest} :: Segmentation -> Matrix -> Town -> Town

such that closest ss dm t is an endpoint in ss, not in the same segment as t; and of all such endpoints, it is the one with the smallest distance (according to dm) to t.

39. Define a function

\texttt{nextJoin} :: Segmentation -> Matrix -> (Town,Town)

which returns the next two towns to be joined. These should be the furthest town in the segmentation, and the town closest to it.

40. Define a function

\texttt{tsp} :: Matrix -> [Town] -> Tour

which returns a tour constructed using the segment heuristic, by starting from an initial segmentation, repeatedly performing the join determined by \texttt{nextJoin}, and extracting the tour from the segmentation once it’s complete.

41. Compare the quality (cost) of tours generated using \texttt{tsp} to the exact solutions (tsp) and any other heuristic you have programmed for a representative number of (reasonably sized) inputs. Try to explain the results. (No more than 150 words.)

42. As the \texttt{nextJoin} function is the only “intelligence” of the \texttt{tsp} algorithm, turn it into a parameter of the algorithm so we can replace it by another. I.e., define a function

\texttt{tspH} :: Matrix -> [Town] -> JoinFn -> Tour

type JoinFn = Segmentation -> Matrix -> (Town,Town)
such that

\texttt{tsph m ts nextJoin}

gives the same result as \texttt{tspm m ts}.

43. Define a function

\texttt{myNextJoin :: JoinFn}

which can be used as a parameter to \texttt{tsph} to implement a different join selection strategy for the segment heuristic.

Explain the idea of the heuristic, present some relevant results, and contrast these with the exact solution and other heuristics. (No more than 200 words.)

11 TSP by extending tours

Adding one town to an existing tour:

![Diagram of a tour with an inserted town](image)

Given a tour, we insert a town into it in the place where it leads to the smallest increase in cost. Doing this for all remaining towns in sequence gives yet another TSP heuristic. Depending on the order in which the towns are inserted, this can be one of the better heuristics. The order in which the towns are added is determined in three different ways:

- by the order the towns are given initially;
• using remotest towns first;
• using remotest towns first (in a different way).

44. You will be able to use the functions ins and cheapest for this part.

Define a function

\[
\text{insTown} :: \text{Matrix} \rightarrow \text{Tour} \rightarrow \text{Town} \rightarrow \text{Tour}
\]

such that \(\text{insTown} \ dm \ ts \ t\) returns a tour obtained by inserting \(t\) somewhere in \(ts\) (without reordering \(ts\) itself), choosing of all the possibilities for doing so one which has minimal total distance according to the distance matrix \(dm\).

For example, \(\text{insTown} \ dm \ [1,2,3] \ 5\) will return one of \([5,1,2,3]\), \([1,5,2,3]\), \([1,2,5,3]\) or \([1,2,3,5]\) – which one depends on which of these four is cheapest according to \(dm\).

A more efficient solution does not reuse as suggested, but instead makes use of the fact that inserting a town involves removing one link, and replacing it by two others; the cost of these is the only thing that really needs to be computed. This is proportional to the size of the Tour rather than quadratic. Taking that even further, one might optimise the next question by memoising these link costs for the tour constructed so far.

45. Define a function

\[
\text{addTowns} :: \text{Matrix} \rightarrow \text{Tour} \rightarrow \text{[Town]} \rightarrow \text{Tour}
\]

such that \(\text{addTowns} \ m \ ts \ us\) adds all towns from \(us\) to the tour \(ts\), at each step adding the head of \(us\) using \(\text{insT}own\) until \(us\) is empty (in which case it returns \(ts\)). As usual, you may assume that \(us\) has no duplicates.

For this heuristic, the auxiliary “main loop” function with the tour so far as an accumulated argument is asked for explicitly although it is not strictly necessary; later variants need such auxiliary functions but it is left implicit there. See also the comment after Exercise 23.

46. Define a function (it has a short name to make your testing easier):

\[
\text{aT} :: \text{Matrix} \rightarrow \text{[Town]} \rightarrow \text{Tour}
\]
such that \( \text{at m ts} \) returns a tour constructed by repeated use of
\( \text{insTown} \); this can be achieved by calling \( \text{addTowns} \) with suitable argu-
ments.

For efficiency reasons, you may use the fact that all tours of length 3
(or less) are equivalent (see Section 8.4).

The next few exercises construct the tour by inserting the remotest
town first into the tour.

47. If we are not going to continuously insert the head of the list, it will
be useful to have a function which removes an element from a list.
Define a function

\[
\text{removeTown} :: \text{Town} \rightarrow [\text{Town}] \rightarrow [\text{Town}]
\]

such that \( \text{removeTown t ts} \) gives a list of towns containing all towns
in \( ts \) except for \( t \). You may assume that \( t \) occurs exactly once in \( ts \).

Recursive solutions often forget to include the segment just considered in the
final result.

48. The following heuristics are based on the idea of remoteness: the
remoteness of a town \( t \) with respect to a list of towns \( ts \) and a distance
matrix \( m \) is the sum of the distances (according to \( m \)) between \( t \) and
each town in \( ts \).
Define a function

\[
\text{remoteness} :: \text{Matrix} \rightarrow [\text{Town}] \rightarrow \text{Int}
\]

such that \( \text{remoteness m ts t} \) gives the remoteness of \( t \) with respect
to \( m \) and \( ts \). For example,

\[
\text{remoteness dm [3,4,5]} \rightarrow \text{dm!!3!!7 + dm!!4!!7 + dm!!5!!7}
\]

49. We will decide which town to add depending on its remoteness – in
particular, for version 1 we will pick the remotest town in each step.
Define a function

\[
\text{remotest} :: \text{Matrix} \rightarrow [\text{Town}] \rightarrow [\text{Town}] \rightarrow \text{Int}
\]
such that remotest \( m \) \( ts \) \( us \) returns the town \( u \) in \( us \) such that remoteness \( m \) \( ts \) \( u \) is the highest of all.

50. The solution to TSP is obtained by repeatedly adding the remotest town relative to the matrix and the tour constructed so far.

Define a function

\[
aRT1 : \text{Matrix} \to [\text{Town}] \to \text{Tour}
\]

such that \( aRT1 \ m \ ts \) returns a tour containing all towns of \( ts \), constructed as follows:

- initially, the tour contains a single town, which must be the remotest with respect to \( m \) and \( ts \);
- at every next step, the town from the remainder of the input is chosen which is remotest with respect to \( m \) and the tour constructed so far (rather than the input!), and added using \( \text{insTown} \).

Alternatively, we could consider the remoteness of towns with respect to the original input. Rather than by keeping the original input around in the program, this can be solved by sorting the input list, and then using the \( aT \) program.

51. Define a function

\[
\text{sortByRemoteness} : \text{Matrix} \to [\text{Town}] \to [\text{Town}]
\]

such that \( \text{sortByRemoteness} \ m \ ts \) contains all the towns of \( ts \), sorted by decreasing remoteness with respect to \( m \) and \( ts \). For example, if remoteness \( m \ [1,2,3] \ 1 \sim > 17 \), remoteness \( m \ [1,2,3] \ 2 \sim > 15 \), and remoteness \( m \ [1,2,3] \ 3 \sim > 19 \), then

\( \text{sortByRemoteness} \ m \ [1,2,3] \sim > [3,1,2] \).

52. Define a function

\[
aRT2 : \text{Matrix} \to [\text{Town}] \to \text{Tour}
\]

which works by first sorting the input towns by decreasing remoteness, and then applying \( aT \) to the result.
53. Compare the quality (in terms of cost and running time) of tours generated using aT, aRT1, and aRT2 with each other and the exact solutions (tsp) for a representative number of (reasonably sized) inputs. Explain and analyse the results. (No more than 200 words.)

12 TSP by combining tours

Rather than adding individual towns to existing tours, we may also choose to combine small tours into bigger ones – starting with tours which start and end in the same place, and ending up with one that contains all towns to be visited.

Combining two existing tours:

12.1 Some preliminary functions

The following general purpose function may be useful.

54. Define a function

```haskell
splits :: Tour -> [(Tour,Tour)]
```

such that `splits ts` returns a list containing all pairs `(ts1, ts2)` such that `ts1 ++ ts2 == ts`. For example, `splits [1,2,3]` should contain the pairs `([], [1,2,3]), ([1], [2,3]), ([1,2], [3]), ([1,2,3], [])` in some order.

As a `Tour` represents a circular tour, there is some confusion: `[1,2,3]` represents the same tour as `[2,3,1]` and `[3,1,2]`. 
55. Define a function

\[ \text{rots} :: \text{Tour} \rightarrow [\text{Tour}] \]

which lists all the tours you get from the input by “rotating” the list, i.e., starting in a different town but following the same tour.

56. Also, because the matrix is symmetric, there is no difference to the cost if we reverse the tour.

Define a function

\[ \text{revs} :: [\text{Tour}] \rightarrow [\text{Tour}] \]

which reverses each of the tours in the input list. (Function \texttt{reverse} is predefined.)

57. For a list of three towns, rotating and reversing together give all permutations – for longer lists, the function \texttt{perms} would return other permutations as well.

Define a function

\[ \text{rotrevs} :: \text{Tour} \rightarrow [\text{Tour}] \]

which returns the list of all tours that can be obtained by rotations and/or reversals of the input list. (For a list without duplicates of length \( n \), there should be \( 2n \) different ones.)

12.2 Merging tours

In this heuristic, we start with lots of small tours.

58. Define a function

\[ \text{singletons} :: [\text{Town}] \rightarrow [\text{Tour}] \]

such that \texttt{singletons ts} returns a \([\text{Tour}]\) with each town from \texttt{ts} in a separate tour.

We will repeatedly combine these until we have included all of the towns.
59. Define a function

```
complete :: [Tour] -> Bool
```

which reports whether the list of tours is complete, in the sense that it puts all its towns in a single tour.

For merging tours, there are two decisions to be taken: which two tours (from a list) to merge, and how to merge them.

60. For selecting two tours, we will take the tours in the list which are nearest to each other. We define the distance between two tours as the smallest distance between some town of one tour and some town of the other.

Define a function

```
distance :: Matrix -> Tour -> Tour -> Int
```

that returns for distance \( dm \) \( ts1 \) \( ts2 \) the distance according to \( dm \) between a town from \( ts1 \) and a town from \( ts2 \) that is minimal for all such pairs of towns.

61. Define a function

```
nearest2 :: Matrix -> [Tour] -> (Tour,Tour)
```

such that \( nearest2 \) \( dm \) \( tss \) gives a pair of tours \( ts1 \) and \( ts2 \) from \( tss \) for which their distance with respect to \( dm \) is minimal.

62. For the decision on how to merge two tours, imagine the problem of combining two paper loops. You would need to cut both in half, (so would need to decide at which point to cut each of them), and then stick them together at the cut, potentially reversing one of them.

Define a function

```
mergeTours :: Matrix -> Tour -> Tour -> Tour
```

which merges two tours in all possible ways, returning the cheapest of those.
A suggestion for `mergeTours dm ts1 ts2` is to list all possibilities by splitting `ts1` in an arbitrary point, then inserting into `ts1` at that point all rotations and reverses of `ts2`. Selecting the cheapest of those should be a familiar problem.

For example, for `merge dm [1,2] [3,4]`, the variations on `[3,4]` are just `[3,4]` and `[4,3]` which needs to be inserted at every possible point in `[1,2]`, leading to `[ [3,4,1,2], [4,3,1,2], [1,3,4,2], [1,4,3,2], [1,2,3,4], [1,2,4,3] ]`, of which a cheapest must be selected.

63. Define a function

```haskell
mergetsp:: Matrix -> [Town] -> Tour
```

which returns a tour constructed starting from an `singles` list of tours, repeatedly merging the nearest two tours, until it is complete.

64. Compare the quality (cost) of tours generated using `mergetsp` to the exact solutions (`tsp`) and other implemented heuristics for a representative number of (reasonably sized) inputs. Try to explain the results. (No more than 150 words.)

65. Discuss and define a reasonable alternative distance function, and use it in a variation on `mergetsp`.

66. Extend the comparison and analysis of Exercise 64 to also include your modification on `mergetsp`.

13 Exam Questions

Exam papers for CO312 consisted of a “case study” (a model solution to the year’s major assessment) which had been handed out well in advance, with questions such as the following.

1. Select a part of the code that, in your view, has been done in a clumsy or inefficient way, and present an alternative solution for it. Explain in which respect your solution improves the solution given above.

2. The required properties of a distance matrix could be characterised as follows:
isDM :: Matrix -> Bool
isDM m = square m && symmetric m &&
   all0 (maindiag m) && triangle m

For each of these four properties, explain:

- what does it mean for a matrix to fail this property, and
- how using a “distance matrix” which failed this property would
  impact on the various TSP algorithms. (Would the results still
  be correct or optimal? If not, could they be fixed easily? Would
  it still be OK to use perms’ instead of perms? etc.)

3. Distances between towns are represented by values of type Int. List all
   the changes that would need to be made to the code if we represented
   distances by values of the type Float instead.

4. Give a solution for the function nearest (Exercise 23) which is sig-
   nificantly different from the one presented in the case study. Discuss
   the relative merits of the two versions, addressing issues of clarity and
   efficiency.

5. The nearest neighbour heuristic (Exercise 25) could also be described
   as a program on segmentations. Rather than always picking the cheap-
   est link connecting any of the segments, it always connects the end of
   the first segment to the start of a segment which is nearest to it in the
   distance matrix.

   Give an alternative definition of nextJoin (Exercise 39) which does
   this. If convenient, you may assume that all of the segments in the
   Segmentation argument to nextJoin are singletons, except for the
   first one. Point out some of the reasons why this variant of “nearest
   neighbour” is less efficient in terms of running time than nn.

6. Consider a variant on the TSP where we only need to visit the towns
   given, but we no longer need to return to our original starting point.
   The answer to our problem would still be represented by a permutation
   of the input, but now interpreted as a true sequence rather than a cycle.

   How would this problem solved by a modification of the code given

   - for the “brute force” TSP;
   - for the heuristic(s)?
Would you expect your heuristic(s) now to result in solutions which are closer to the optimal solution than they were for the original TSP? Why?

7. Define a function

internal :: Segmentation -> Town -> Bool

which returns True if the town occurs in the segmentation, but is not an endpoint of the segments in the segmentation, and which returns False in all other cases.

8. A simpler heuristic than “furthest first” implemented in the tsp heuristic (Exercise 40) is “nearest first” which simply chooses for every next step the two endpoints of different segments which are nearest to each other. This can be done by only changing the nextJoin function (Exercise 39).

Define an alternative nextJoin function (same type as above) which encodes this strategy: it should return the pair of towns which, of all combinations of two endpoints of different segments, have the smallest distance between them according to the distance matrix.

9. Give a variant of the function distance (Exercise 60) which determines the distance between two tours not as the minimum, but as the average distance between pairs of towns from each of the tours. You may round down the average to an integer, i.e., use div rather than / for division.

10. If, in the function nearest2 (Exercise 61), you had accidentally swapped the two arguments in its definition, i.e., you had written

nearest2 :: Matrix -> [Tour] -> (Tour,Tour)
nearest2 tss dm

and left the rest unchanged, Hugs would not have reported a type error. Why not? How would you have discovered about the two arguments being swapped anyway?

11. Modify the mergeTsp function (Exercise 63) such that, rather than selecting the nearest tours at every step, it simply selects the first two elements of the list of tours to merge.
Keeping in mind that the newly merged tour is inserted at the front of the list of tours, explain how the resulting algorithm relates to the aT algorithm (Exercise 46).

Online materials

This document is available through the departmental publications web page at http://www.cs.kent.ac.uk/pubs/. A sample distance matrix is available at http://www.cs.kent.ac.uk/people/staff/eab2/tsp/DM.hs. A program to visualise tours computed using this distance matrix is at http://www.cs.kent.ac.uk/people/staff/eab2/tsp/mapper/.

Acknowledgements

I would like to thank the CO312 students from 2000/01 to 2003/04 for their feedback and ideas. In particular, Adam Sampson wrote the “TSP mapper” (see URL above) and also inspired the “segments” heuristic. Neil Renaud implemented in 2001 the “ remotest town first” strategy in Section 11 as his alternative heuristic – based on real field research, talking to a friend who had to solve TSP for his delivery job on a daily basis.

My colleague Claus Reinke suggested foldl1’ as the solution to the control stack overflow problem in brute force TSP; Olaf Chitil made some useful comments on this technical report.

References