CS106X — Handout #26

Spring 99-00 — April 30, 2000

Section Exercises 5: Recursion

Problem 1 :Recursion warm-up

A word is a palindrome if it reads the same forward and backwards. Write a recursive funciton which determines if an input string is a palindrome. The key recursive insight is that a palindrome is composed of a smaller palindrome plus a little bit more. For example, the palindrome "level" is composed of the smaller palindrome "eve", with an 'l' at the front and back.

Problem 2 : Recursive linked lists
For the remaining questions, use this type declaration for the nodes of the link lists:

```
typedef struct _Node {
   int value;
   struct _Node *next;
} Node;
```

a) Write a function that recursively computes the length of a linked list.

b) Write a recursive function that prints a linked list in reverse order.

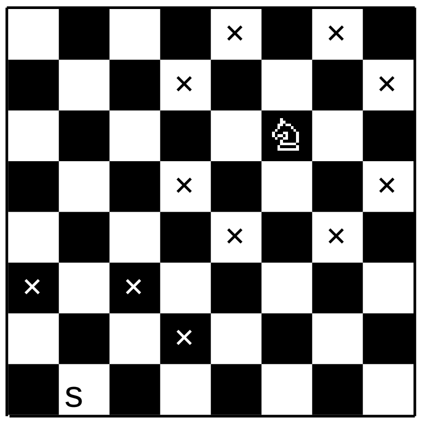
Problem 3 : Tougher recursive linked lists
a) Write a recursive function that given two lists will append the second list onto the first. For example, given the first list (1 4 6) and the second list (3 19 2), the function would destructively modify the first list to contain (1 4 6 3 19 2).

b) Write an iterative (ie. non-recursive) function to split a linked list of numbers into two linked lists that contain the odd and even numbers of the original list. For example, if the starting list was (4 2 5 3 14), then when Split(List, OddList, EvenList) ends, OddList is (5 3 1), and EvenList is (4 2 4). Don't make copies of the cells, just use the elements of the original list to make up the two new lists.

c) Rewrite the above Split function recursively. You'll probably find the recursive version simpler to write.

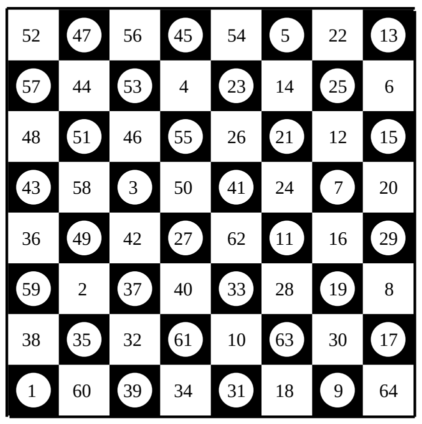
Problem 4: Knights Tour (Chapter 6, exercise 7, page 278)

In chess, a knight moves in an L-shaped pattern: two squares in one direction horizontally or vertically, and then one square at right angles to that motion. For example, the black knight in the following diagram can move to any of the eight squares marked with a black cross:



The mobility of the knight decreases toward the edges of the board, as illustrated by the position of the white knight, which can move only to the three squares marked by white crosses.

It turns out that a knight can visit all 64 squares on a chessboard without ever moving to the same square twice. A path for the knight that moves through all the squares without repeating a square is called a **knight's tour.** One such tour is shown in the following diagram, in which the numbers in the squares indicate the order in which they were visited:

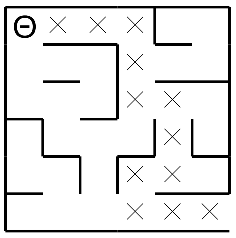


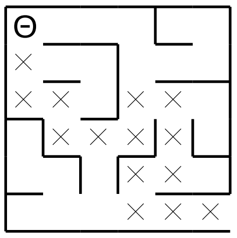
52	47	56	45	54	5	22	13
57	44	53	4	23	14	25	6
48	51	46	55	26	21	12	15
43	58	3	50	41	24	7	20
36	49	42	27	62	11	16	29
59	2	37	40	33	28	19	8
38	35	32	61	10	63	30	17
1	60	39	34	31	18	9	64

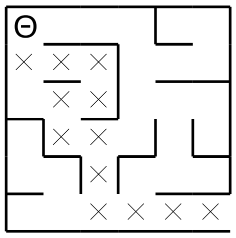
Write a program that uses backtracking recursion to find a knight's tour.

Problem 5: Shortest Path Through a Maze

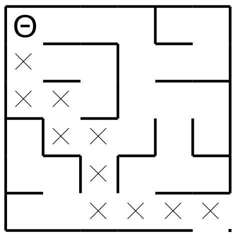
In many mazes, there are multiple paths. For example, the diagrams below show three solutions for the same maze:


length = 13


length = 15


length = 13

None of these solutions, however, is optimal. The shortest path through the maze has a path length of 11:



As a starting point, begin by considering the general maze solution covered in lecture and the textbook:

```
/*
  * Function: SolveMaze
  * Usage: if (SolveMaze(pt)) . . .
  * -------------------------------
  * This function attempts to generate a solution to the current
  * maze from point pt.  SolveMaze returns TRUE if the maze has
  * a solution and FALSE otherwise.  The implementation uses
  * recursion to solve the submazes that result from marking the
  * current square and moving one step along each open passage.
  */

 static bool SolveMaze(pointT pt)
 {
     directionT dir;

     if (OutsideMaze(pt)) return (TRUE);
     if (IsMarked(pt)) return (FALSE);
     MarkSquare(pt);
     for (dir = North; dir <= West; dir++) {
         if (!WallExists(pt, dir)) {
             if (SolveMaze(AdjacentPoint(pt, dir))) {
                 return (TRUE);
             }
         }
     }
     UnmarkSquare(pt);
     return (FALSE);
 }
```

Write a function

```
int ShortestPathLength(pointT pt);
```

that returns the length of the shortest path in the maze from the specified position to any exit. If there is no solution to the maze, `ShortestPathLength` should return the constant `NoSolution`, which is defined to have a value larger than the maximum permissible path length, as follows:

```
#define NoSolution 10000
```

Problem 6: Anagrams
Complete the recursive procedure `ListAnagrams,` which lists those permutations of a string which are either legal words or the concatenation of several **legal** words, where we define a **legal** word as any word in the Boggle dictionary of length **three** or more. For instance, the word **resin** has six legal anagrams:

resin
reins
risen
rinse
serin
siren

Only these six permutations of the word **resin** happen to be legal anagrams. Furthermore, there are over 30 legal anagrams of the word **mendicants**, three of which are:

antics mend
minced ants
scad tin **men**

Note that each of the three is the concatentaion of two or more words.

The code to generate all of the unique permutations is written, but you need to write the **recursive** predicate `IsLegalAnagram`, which returns `TRUE` if and only if a particular string is a legal word or the concatentaion of two or more legal words. A `WordIsLegal` predicate is also provided below.

```
static bool WordIsLegal(string wordToCheck)
{
    return (StringLength(wordToCheck) >= 3 &&
           IsWordInDictionary(wordToCheck));      // assume this takes care
of lookup in the
}                                          // dictionary OSPD2.. don't worry
about
                                           // dictionaries at all.

static void ListAnagrams(string current, int start, int end)
{
   int index;

   if (start == end) {
      if (IsLegalAnagram(current))
         printf("%s\n", current);
      return;
   }

   for (index = start; index < end; index++) {
      if (FindChar(current[index], current, index + 1) == -1) {
         SwapLetters(current, start, index);         // assume SwapLetters
is written
         ListAnagrams(current, start + 1, end);
         SwapLetters(current, start, index);
      }
   }
}
```