Informatics II, Spring 2024, Solution Exercise 2

Publication of exercise: February 26, 2024
Publication of solution: March 4, 2024
Exercise classes: March 4 - March 8, 2024

Learning Goal

- Understand how to define the subproblem of a recursive problem.
- Learn how to define the stopping conditions of a recursive program.

Task 1: Palindrome [Easy]

A **palindrome** is a word, number, phrase, or other sequence of symbols that reads the same backwards as forwards, such as 'madam' or 'racecar'.

Write an **recursive** algorithm to check whether a given string is a palindrome. The string can include numbers, letters (case-sensitive), and other symbols. For example, **'1a_b3cD45t54Dc3b_a1'** is a palindrome.

You can start by filling out the function below.

```c
/**
* Check the symbols in index i and index j
*/
int isPalindrome(char X[], int i, int j) {
    //Put your code here
}

int main(){
    char X[] = "1a_b3cD45t54Dc3b_a1";
    //Put your code here
}
```

```c
#include <stdio.h>
#include <stdlib.h>
#include <string.h>

/**
 * Check the symbols in index i and index j
*/
int isPalindrome(char X[], int i, int j) {
    if(j≤i) {
        return 1;
    }
    if(X[i] != X[j]) {
        return 0;
    }
    return isPalindrome(X, i+1, j-1);
}

int main() {
```

```
    char X[] = "1a_b3cD45t54Dc3b_a1";

    if (isPalindrome(X, 0, sizeof(X)-2) == 1) {
        printf("This␣is␣a␣Palindrome.");
    }
    else {
        printf("This␣is␣not␣a␣Palindrome.");
    }

    return 0;
}
```

Task 2: Eight Queen Puzzle [Medium]

The **Eight Queens Puzzle** is the problem of placing eight chess queens on an 8×8 chessboard so that no two queens threaten each other. Therefore, a solution must satisfy that no two queens share the same row, column, or diagonal. One possible solution is shown in Figure 1.

The eight queens puzzle is a special case of the more general **N Queens Problem** of placing N non-attacking queens on an $N \times N$ chessboard. In this task, you are required to calculate the number of all possible solutions for the N queens problem. Solutions exist for all natural numbers N with the exception of $N = 2$ and $N = 3$.

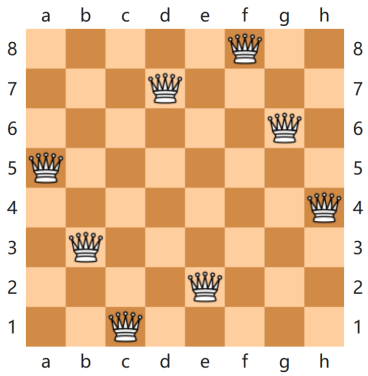

Figure 1: One solution for the eight queen puzzle

Given the number of queens N $(3 < N < 20)$, you need to print the number of all possible solutions. You can start by filling out the functions below.

```
int N = 8;
int count = 0;

/**
* Check if it's safe to place a new Queen in the given position
*/
int isSafe(int board[N][N], int row, int col) {
    //Put your code here
}
/**
* Try to place the N queens in the board
* and update the global variable 'count'
*/
void solveNQueens(int board[N][N], int row) {
    //Put your code here
}
```

```

int main() {
    int board[N][N];
    memset(board, 0, sizeof(board)); //initialize the board and fill with 0
    //Put your code here
}
```

```c
#include <stdio.h>
#include <stdlib.h>
#include <string.h>

int N = 8;
int count = 0;

void printBoard(int board[N][N]) {
    printf("Solution␣%d:\n", count);
    for (int i = 0; i < N; i++) {
        for (int j = 0; j < N; j++) {
            printf("%c␣", board[i][j] ? 'Q' : '.');
        }
        printf("\n");
    }
    printf("\n");
}

/**
 * Check if it's safe to place a new Queen in the given position
*/
int isSafe(int board[N][N], int row, int col) {
    // check column
    for (int i = 0; i < row; i++) {
        if (board[i][col]) {
            return 0;
        }
    }
    // check diag
    for (int i = row, j = col; i ≥0 && j ≥0; i--, j--) {
        if (board[i][j]) {
            return 0;
        }
    }
    // check inv-diag
    for (int i = row, j = col; i ≥0 && j < N; i--, j++) {
        if (board[i][j]) {
            return 0;
        }
    }
    return 1;
}


/**
 * Try to place N queens in the board
 * and update the global variable 'count'
*/
void solveNQueens(int board[N][N], int row) {
    if (row == N) {
        count++;
        // printBoard(board);
        return;
    }

    // try to place the queen col by col
    for (int col = 0; col < N; col++) {
```

```
        if (isSafe(board, row, col)==1) {
            board[row][col] = 1;
            // go to next row
            solveNQueens(board, row + 1);
            // reverse
            board[row][col] = 0;
        }
    }
}

int main() {
    int board[N][N];
    memset(board, 0, sizeof(board));

    // try to place the queen row by row
    solveNQueens(board, 0);

    printf("Total solutions: %d\n", count);

    return 0;
}
```

Task 3: Tower of Hanoi [Medium]

The **Tower of Hanoi** is a game consisting of three columns and a number of disks that can be placed on any of the columns. No two disks have the same size. At the beginning, all disks are stacked on the first column (the left one) in order of decreasing size (the smallest disk at the top). The goal of the game is to move all disks to the second column (the middle one). There are only two rules:

- In each step only the top-most disk from a column can be moved to the top of another column.
- A large disk may never be placed on top of a smaller disk.

Figure 2 illustrates the stacks after 5 moves in a game with 6 disks .

Figure 2: Tower of Hanoi with 6 disks

Your task is to display all steps in a game and count the number of steps you play. You can show them in formats like **'step 13: a ->b'** or **'step 13: T1 ->T2'**.

You can start by filling out the function below. You should do it in a recursive way .

```
int moveCount = 0;
/**
* Move n disks from source to destination via auxiliary
*/
void hanoi(int n, char src, char aux, char dst) {
    //Put your code here
}

int main() {
```

```
    //Put your code here
}
```

```c
#include <stdio.h>
#include <stdlib.h>

int moveCount = 0;

/**
 * Move disk n from src to dst via aux
*/
void hanoi(int n, char src, char aux, char dst) {
    if (n == 0) {
        return;
    }
    // move n-1 disks from source to auxiliary via destination
    hanoi(n - 1, src, dst, aux);

    // move the n-th disk from source to destination in 1 step
    printf("disk␣%d␣from␣%c␣to␣%c\n", n, src, dst);
    moveCount++;

    // move n-1 disks from auxiliary to destination via source
    hanoi(n - 1, aux, src, dst);
}

int main() {
    int n;

    printf("Enter␣the␣number␣of␣disks:␣");
    scanf("%d", &n);

    printf("\nHanoi␣Tower␣Steps:\n");
    hanoi(n, 'A', 'B', 'C');

    printf("\nTotal␣number␣of␣moves:␣%d\n", moveCount);

    return 0;
}
```

Task 4: Calculator [Hard]

In this task, you can implement your own calculator! Suppose you are given a numerical expression with only basic operators $(+, -, \times, \div)$ and parentheses ().

You can assume that:

- There is no blank inside the string.
- All numbers are integers, and negative integers are always enclosed in parentheses.

Please calculate the result of a numerical expression using a recursive function and print the result. Below are the example inputs and outputs.

Inputs & Outputs

```
(3+4)*5/6+7*8*(1+2-3)
Ans = 5.833333

(3+4)*5/6+7*8*(1+(-2)-1+1)
Ans = -50.166668
```

You can start by filling out the functions below.

```c
/**
* Find the position of the operator with the lowest
* priority between index h and index t
*/
int findOperator(char *expr, int h, int t) {
    //Put your code here
}
/**
* Calculate the result of the expression between index h and index t
*/
float calculator(char *expr, int h, int t) {
    //Put your code here
}

int main() {
    char expr[] = "(3+4)*5/6+7*8*(1+2-3)";
    //Put your code here
}
```

```c
#include <stdio.h>
#include <stdlib.h>
#include <string.h>

/**
 * Find the position of the operator with the lowest
 * priority between index h and index t
*/
int findOperator(char *expr, int h, int t) {
    int count = 0;
    // use count to make sure the current operator are not inside of a pair of '()'
    // because the ops inside '()' have higher priority

    // looking for '+' or '-'
    for (int i = t; i >=h; --i) {
        if (expr[i] == ')') {
            count++;
        } else if (expr[i] == '(') {
            count--;
        } else if ((expr[i] == '+' || expr[i] == '-') && count == 0) {
            return i;
        }
    }
    // looking for '*' or '/'
    for (int i = t; i >=h; --i) {
        if (expr[i] == ')') {
            count++;
        } else if (expr[i] == '(') {
            count--;
        } else if ((expr[i] == '*' || expr[i] == '/') && count == 0) {
            return i;
        }
    }
    // -1 means no operator
    return -1;
}

/**
 * Calculate the result of the expression between index h and index t
*/
float calculator(char *expr, int h, int t) {
    if (h > t) {
```

```
43          return 0.0;
44      }
45
46      // check if both sides have a pair of ()
47      // remove them because they don't make any sense
48      // !!!! Attension! You must check if the '(' and ')' are paired
49      if (expr[h] == '(' && expr[t] == ')') {
50          int flag = 1, count = 0;
51          // check the remaining part
52          for (int k=h+1; k≤t-1; k++) {
53              if (expr[k] == '(') count ++;
54              else if (expr[k] == ')') count --;
55
56              if (count < 0) {
57                  flag = 0;
58                  break;
59              }
60          }
61          if (flag == 1) {
62              h++;
63              t--;
64          }
65      }
66
67      int opIndex = findOperator(expr, h, t);
68      // printf("op idx %d", opIndex);
69
70      if (opIndex == -1 || (opIndex == h && expr[h] == '-')) {
71          // no operator or the given expr is a negative number (-xxx)
72          char num[20];
73          int numIndex = 0;
74          for (int i = h; i ≤t; ++i) {
75              num[numIndex++] = expr[i];
76          }
77          num[numIndex] = '\0';
78          // Hint: use atoi() to convert numerical string to number
79          return (float)(atoi(num));
80      }
81      else {
82          // do the math
83          char op = expr[opIndex];
84
85          // printf("\nLeft: ");
86          // for (int k=h; k≤opIndex-1; k++) {
87          // printf("%c", expr[k]);
88          // }
89          float left = calculator(expr, h, opIndex - 1);
90
91          // printf("\nRight: ");
92          // for (int k=opIndex+1; k≤t; k++) {
93          // printf("%c", expr[k]);
94          // }
95          float right = calculator(expr, opIndex + 1, t);
96
97          printf("\nOP: %c, Left=%f, Right=%f", op, left, right);
98
99          switch (op) {
100             case '+':
101                 return left + right;
102             case '-':
103                 return left - right;
104             case '*':
105                 return left * right;
106             case '/':
```

```
107                 return left / right;
108             default:
109                 printf("ERROR - Unknown Operator!\n");
110                 return 0;
111         }
112     }
113 }
114 
115 int main() {
116     char expr[100];
117     printf("Enter an expression: ");
118     fgets(expr, sizeof(expr), stdin);
119 
120     // Remove trailing newline character
121     if (expr[strlen(expr) - 1] == '\n') {
122         expr[strlen(expr) - 1] = '\0';
123     }
124 
125     float result = calculator(expr, 0, strlen(expr) - 1);
126 
127     printf("\nAns = %f\n", result);
128 
129     return 0;
130 }
```