National University of Singapore
School of Computing
CS1101S: Programming Methodology
Semester I, 2019/2020

S10
Searching and Sorting; Memoization

Problems:

1. Consider the following Source program:

```
function swap(A, i, j) {
    let temp = A[i];
    A[i] = A[j];
    A[j] = temp;
}

function reverse_array(A) {
    const len = array_length(A);
    const half_len = math_floor(len / 2);
    for (let i = 0; i < half_len; i = i + 1) {
        const j = len - 1 - i;
        swap(A, i, j);
    }
}

const arr = [1, 2, 3, 4, 5];
reverse_array(arr);
```

   Draw the diagram to show the environment during the evaluation of the program. Show all the frames that are created during the program evaluation, but do not draw empty frames. Show the final value of each binding.

2. The following function, `bubblesort_array`, is an implementation of the ***Bubble Sort*** algorithm to sort an array of numbers into ascending order:

```
function bubblesort_array(A) {
    const len = array_length(A);
    for (let i = len - 1; i >= 1; i = i - 1) {
        for (let j = 0; j < i; j = j + 1) {
            if (A[j] > A[j + 1]) {
                const temp = A[j];
                A[j] = A[j + 1];
                A[j + 1] = temp;
            } else { }
        }
    }
}
```

(a) What is the order of growth of its runtime for an input array of *n* elements?

(b) Write the function, `bubblesort_list`, that takes as argument a **list** of numbers and uses the bubble sort algorithm to sort the list into ascending order. Your function **must not create any new pair or array**, and **must not use the function `set_tail`**. Its runtime must have the same order of growth as that of `bubblesort_array`.

```
function bubblesort_list(L) {
    // ???
}
```

Example use:

```
const LL = list(3, 5, 2, 4, 1);
bubblesort_list(LL);
LL; // should show [1, [2, [3, [4, [5, null]]]]]
```

3. The ***Game of Life*** is a zero-player game, in which an infinite, two-dimensional grid of ***cells*** evolve according to a set of rules. Each cell can be in one of two possible states, ***alive*** or ***dead***. Every cell's state in the next generation depends on the current states of its eight neighbours, which are the cells that are horizontally, vertically, or diagonally adjacent. The following rules determine a cell's state in the next generation:

   (1) Any live cell with fewer than two live neighbours dies, as if caused by under-population.

   (2) Any live cell with two or three live neighbours lives on to the next generation.

   (3) Any live cell with more than three live neighbours dies, as if by over-population.

   (4) Any dead cell with exactly three live neighbours becomes a live cell, as if by reproduction.

   We represent the grid of cells with a 2D array of numbers. An element value of 0 (zero) represents a dead cell and a 1 (one) represents a live cell.

   (a) Write a function `make_2D_zero_array(rows, cols)` that takes in two positive integer arguments, `rows` and `cols`, and returns a 2D array with `rows` rows and `cols` columns. All elements in the result array are set to 0 (zero). You **must not use any loop** in your function.

```
function make_2D_zero_array(rows, cols) {
    // ???
}
```

   (b) To represent the infinite 2D grid of cells, we use a `n`-by-`n` 2D array, and consider the left and right edges of the array to be stitched together, and the top and bottom edges also. In other words, the array's left-most and right-most columns are considered adjacent to each other, and likewise for its top-most and bottom-most rows. An array that is treated in this way is called a ***toroidal*** array.

   Write a function `num_of_live_neighbours(game, n, r, c)` that takes in as parameters a `n`-by-`n` 2D array `game`, an integer `n` (where `n ≥ 3`), a row number `r` and a column number `c` (where `0 ≤ r < n` and `0 ≤ c < n`). The input 2D array represents the current states of the cells. The function returns the number of live neighbours the cell `game[r][c]` has. Your function should consider the toroidal property of the 2D array when looking for the cell's neighbours.

   Also note that, for an integer `x`, when `x < 0`, the result of `(x % n)` can be negative. A trick to always get a non-negative remainder is to use `((x + n) % n)` if `x ≥ -n`.

```
function num_of_live_neighbours(game, n, r, c) {
    // ???
}
```

(c) Complete the function `next_generation(game, n)` that takes in a `n`-by-`n` 2D array `game`, and an integer `n` (where `n ≥ 3`) as parameters. The input 2D array represents the current states of the cells. The function returns a new `n`-by-`n` 2D array that contains the next-generation states of the cells.

What is the order of growth of your function's runtime in terms of `n`?

```
function next_generation(game, n) {
    const next = make_2D_zero_array(n, n);

    // ???

    return next;
}
```

4. Consider the `cc` (coin change) function presented in Lecture L3:

```
function cc(amount, kinds_of_coins) {
    return amount === 0
        ? 1
        : amount < 0 || kinds_of_coins === 0
            ? 0
            : cc(amount, kinds_of_coins - 1)
              +
              cc(amount - first_denomination(kinds_of_coins),
                 kinds_of_coins);
}

function first_denomination(kinds_of_coins) {
    return kinds_of_coins === 1 ?   5 :
           kinds_of_coins === 2 ?  10 :
           kinds_of_coins === 3 ?  20 :
           kinds_of_coins === 4 ?  50 :
           kinds_of_coins === 5 ? 100 : 0;
}
```

Is function `cc` a good candidate for memoization? Support your answer with an example.

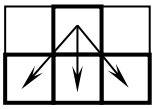
If memoization is suitable for function `cc`, provide the implementation.

What are the orders of growth in *time* and *space* of the memoized version?

5. **[Optional]** This task is optional. Your Avenger will cover this task with you if there is sufficient time left in your Studio session.

   During the rainy season, one of the walls in the house is infested with flies. The wall is covered by $R \times C$ square tiles, where there are R rows of tiles from top to bottom, and C columns of tiles from left to right.

   A house lizard wants to eat as many flies as possible, subject to the following restriction. It starts in any tile in the top row, and eats the flies on the tile. Then, it moves to a tile in the next lower row, eats the flies on the tile, and so on until it reaches the floor. When it moves from one tile to a tile in the next lower row, it can only move vertically down or diagonally to the left or right (see figure below).



   (a) Write a function `max_flies_to_eat(tile_flies)`, that takes as argument `tile_flies`, which is a 2D array (array of arrays) that stores the number of flies on each tile, and returns the maximum possible number of flies the lizard can eat in one single trip from the top to the bottom of the wall. The topmost row of tiles on the wall is Row 0, and `tile_flies[r][c]` contains the number of flies on the tile in Row `r` and Column `c`. Your function should not use memoization.

   Example run:

```
const tile_flies = [[3, 1, 7, 4, 2],
                    [2, 1, 3, 1, 1],
                    [1, 2, 2, 1, 8],
                    [2, 2, 1, 5, 3],
                    [2, 1, 4, 4, 4],
                    [5, 7, 2, 5, 1]];

max_flies_to_eat(tile_flies); // Expected result: 32
```

   What is the order of growth in *time* of your `max_flies_to_eat`?

   (b) Write a function `memo_max_flies_to_eat(tile_flies)`, that is a memoized version of function `max_flies_to_eat`.

   What is the order of growth in *time* of your `memo_max_flies_to_eat`?