CS1101S Practical Assessment Helper Functions

Swap

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

Clone/Copy an Array

```
function copy_array(A) {
  const len = array_length(A);
  const B = [];
  for (let i = 0; i < len; i = i + 1) {
    B[i] = A[i];
  }
  return B;
}
```

Reverse an Array

```
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) {
    swap(A, i, len - 1 - i);
  }
}
//^NEEDS SWAP FUNCTION IN CONJUNCTION

//OR
function arr_reverse(arr) {
  const reversed_arr = [];
  const arr_length = array_length(arr);
  for (let i = 0; i < arr_length; i = i + 1) {
    reversed_arr[arr_length - i - 1] = arr[i];
  }
  return reversed_arr;
}

//DESTRUCTIVE reversal:
function d_arr_reverse(arr) {
  const arr_length = array_length(arr);
  for (let i = 0; i < arr_length/2; i = i + 1) {
    const temp = arr[arr_length - i - 1];
    arr[arr_length - i - 1] = arr[i];
    arr[i] = temp;
  }
  return arr;
}
```

Append Array

```
function arr_append(A, B) {
  const a_len = array_length(A);
  const b_len = array_length(B);
  //requires copy_array function
  const C =  copy_array(A)
  for(let i = 0; I < b_len; i = i + 1){
    C[i + a_len] = B[i];
  }
  return C;
}

//OR
function append_array(arr1, arr2) {
  let final = [];
  for (let i = 0; i < array_length(arr1); i = i + 1) {
    final[i] = arr1[i];
  }
  for (let j = 0; j < array_length(arr2); j = j + 1) {
    final[j + array_length(arr1)] = arr2[j];
  }
  return final;
} //append_array([1, 2, 3, 4, 5], append_array([6, 7], [8, 9, 10]));
```

Array Sorting, Ascending order

```
function sort_ascending(A) {
  const len = array_length(A);
  for (let I = 1; I < len; I = I + 1) {
    const x = A[i];
    let j = I – 1;
    while (j >= 0 && A[j] > x) {
      A[j + 1] = A[j];
      j = j – 1;
    }
    A[j + 1] = x;
  }
}
```

```
//let J = [9,7,5,3,1];
//sort_ascending(J);
//J; --> returns [1,3,5,7,9]
```

Convert Array to String

```
function array_to_string(arr) {
  const len = array_length(arr);
  let str = " ";
  for (let I = 0; I < len; I = I + 1) {
    str = str + stringify(arr[i]);
  }
  return str;
}
```

Convert Array to List

```
function array_to_list(A) {
  const len = array_length(A);
  let L = null;
  for (let I = len – 1; I >= 0; I = I – 1) {
    L = pair(A[i], L);
  }
  return L;
}
```

Convert List to Array

```
function list_to_array(L) {
  const A = [];
  let I = 0;
  for (let p = L; !is_null(p); p = tail(p)) {
    A[i] = head(p);
    I = I + 1;
  }
  return A;
}
```

```
//[1,2,3,4,5] returns list(1,2,3,4,5)
**Note, this function does not work for
list(1,2,3,list(1)), returns [1,2,3,[1,null]]
```

To check if arrays A and B are structurally equal.

```
function equal_array(A, B) {
  if (!is_array(A) || !is_array(B)) {
    return false;
  } else if (array_length(A) !== array_length(B)) {
    return false;
  } else {
    let is_equal = true;
    const len = array_length(A);
    for (let I = 0; is_equal && I < len; I = I + 1) {
      if (is_array(A[i]) || is_array(B[i])) {
        is_equal = equal_array(A[i], B[i]);
      } else {
        is_equal = equal(A[i], B[i]);
      }
    }
  } }
```

Enumerate Array

```
function enum_array(start, end){
  const A = [];
  let counter = 0;
  for(let I = start; I < end + 1; I = I + 1){
    A[counter] = I;
    counter = counter + 1;
  }
  return A; }
```

Flatten an Array (CHECK) LOOKS SUS

```
function flatten_array(arr){
    let index_count = 0;
    let result = [];
    function helper(a) {
      const len = array_length(a);
      for (let i = 0; i < len; i = i + 1) {
        const curr = a[i];
        if (is_array(curr)) {
          helper(curr);
        } else {
          result[index_count] = curr;
          index_count = index_count + 1;
        }
      }
    }
    helper(arr);
    return result;
}

//OR
function flatten(A) {
    return accumulate_array(extend, [], A);
}
let arr = [1,2,3,[4,5,6,7],[2,3,4,[2,3,4]]];
flatten_array(arr);
returns [1, 2, 3, 4, 5, 6, 7, 2, 3, 4, 2, 3, 4]
```

Removes the last element in an array, & returns a copy

```
function remove_last(arr) {
    const popped_arr = [];
    const arr_length = array_length(arr) – 1;
    for (let I = 0; I < arr_length; I = I + 1) {
      popped_arr[i] = arr[i];
    }
    return popped_arr;
}
```

Removes current index of array, & returns a copy

```
function remove_index(arr, index) {
    let removed_arr = [];
    let arr_len = array_length(arr);
    for (let I = 0; I < arr_len; I = I + 1) {
      if (I < index) {
        removed_arr[i] = arr[i];
      } else if (I > index) {
        removed_arr[I – 1] = arr[i];
      } else {}
    }
    return removed_arr;
}
```

Adds an element in front of the array. Returns a copy

```
function add_in_front(element, arr) {
    const unshifted_arr = [element];
    const arr_length = array_length(arr);
    for (let i = 0; i < arr_length; i = i + 1) {
      unshifted_arr[i + 1] = arr[i];
    }
    return unshifted_arr;
}
let arr = [1,2,3,4,5,6];
add_in_front(1000, arr);
returns [1000, 1, 2, 3, 4, 5, 6]
```

Add an element at a specified index, shift everything else back, returns a copy

```
function add_in_index(element, arr, index) {
    const added_arr = [];
    const arr_length = array_length(arr);
    for (let i = 0; i <= arr_length; i = i + 1) {
      if (i < index) {
        added_arr[i] = arr[i];
      } else if (i > index) {
        added_arr[i] = arr[i - 1];
      } else {
        added_arr[i] = element;
      }
    }
    return added_arr;
}
let arr = [1,2,3,4,5,6];  add_in_index(1000, arr, 3); returns [1000, 1, 2, 3, 4, 5, 6]
```

Add Element to back of Array

```
function add_back(element, arr){
  const len = array_length(arr);
  arr[len] = element;
  //return arr;   (return optional)
}
```

Build Array

```
function build_array(n, f){
    const A = [];
    for(let I = 0; I < n; I = I + 1){
      A[i] = f(i);
    }
    return A;
}
```

Accumulate for Arrays

```
function accumulate_array(op, init, A) {
    let x = init;
    for (let i = init; i < array_length(A); i = i + 1) {
      x = op(x, A[i]);
    }
    return x;
}
// accumulate_array((x, y) => x + y, 0, [1,2,3,4,5]);
// fold_left on array
```

Map for Arrays

```
function array_map(f, A){
    const len = array_length(A);
    // requires copy_array function
    function copy_array(A) {
      const B = [];
      for (let I = 0; I < len; I = I + 1) {
        B[i] = A[i];
      }
      return B;
    }
    //end of copy_array
    const B = copy_array(A);
    for(let I = 0; I < len; I = I + 1){
      B[i] = f(B[i]);
    }
    return B;
}
//OR
function map_array(f, arr) {
    for (let I = 0; I < array_length(arr); I = I + 1) {
      arr[i] = f(arr[i]);
    }
    return arr;
}
// map_array(x => x + 2, [1, 2, 3, 4]);
//OR
function arr_map(arr, func) {
    const mapped_arr = [];
    const arr_length = array_length(arr);
    for (let I = 0; I < arr_length; I = I + 1) {
      mapped_arr[i] = func(arr[i]);
    }
return mapped_arr;
```

Array Filter

```
function array_filter(pred, A){
   const len = array_length(A);
   const B = [];
   let counter = 0;
   for(let I = 0; I < len; I = I + 1){
     if(pred(A[i])){
        B[counter] = A[i];
        counter = counter + 1;
     } else {
        continue;
     }
   }
   return B;
}

//OR
function filter_array(pred, arr) {
   let final = [];
   let final_index = 0;
   for (let i = 0; i < array_length(arr); i = i + 1) {
      if (pred(arr[i])) {
         final[i - final_index] = arr[i];
      } else {
         final_index = final_index + 1;
      }
   }
   return final;
}
// filter_array(x => x > 5, [4, 5, 6, 7, 8, 9, 2, 3, 6]);
```

Counting

Permutations:

```
function permutations(ys) {
   return is_null(ys)
      ? list(null)
      : accumulate(append, null,
         map(x => map(p => pair(x, p),
                  permutations(remove(x, ys))),
            ys));
}
```

Choose

```
function choose(n, r) {
    if (n < 0 || r < 0) {
        return 0;
    } else if (r === 0) {
        return 1;
    } else {
// Consider the 1st item, there are 2 choices:
// To use, or not to use
// Get remaining items with wishful thinking
        const to_use = choose(n - 1, r - 1);
        const not_to_use = choose(n - 1, r);
        return to_use + not_to_use;
    }
}
```

Combinations

```
function combinations(xs, r) {
    if ( (r !== 0 && xs === null) || r < 0) {
        return null;
    } else if (r === 0) {
        return list(null);
    } else {
        const no_choose = combinations(tail(xs), r);
        const yes_choose = combinations(tail(xs), r - 1);
        const yes_item = map(x => pair(head(xs), x),
                             yes_choose);
        return append(no_choose, yes_item);
    }
}
```

Remove Duplicates in a List

```
function remove_duplicates(lst) {
  return is_null(lst)
    ? null
    : pair(
      head(lst),
      remove_duplicates(
        filter(
          x => !equal(x, head(lst)),
          tail(lst))));
}
```

Matrix

Transpose Matrix

```
function transpose(M) {
   const rlen = array_length(M);
   const clen = array_length(M[0]);
   const a = [];
   for (let i = 0; i < clen; i = i + 1) {
      const b = [];
      for (let j = 0; j < rlen; j = j + 1) {
         b[j] = M[j][i];
      }
      a[i] = b;
   }
   return a;
}
```

Create Zero Matrix

FOR ARRAYS

```
function create_zero_matrix(r, c) {
   let final = [];
   for (let i = 0; i < r; i = i + 1) {
      final[i] = [];
      for (let j = 0; j < c; j = j + 1) {
         final[i][j] = 0;
      }
   }
   return final;
}
```

FOR LISTS

```
function make_zero_matrix(r, c){
   const lst = build_list(x => x = 0, c);
   let zeromatrix = null;
   let x = 0;
   for(let x = 0; x < r; x = x + 1){
      zeromatrix = pair(lst, zeromatrix);
   }
   return zeromatrix;
}
```

Create Matrix

```
function create_matrix(i, j, f) {
   const ans = [];
   let counter = 0;

   for (let x = 0; x < i; x = x + 1) {
      ans[x] = [];
```

```
      for (let y = 0; y < j; y = y + 1) {
        ans[x][y] = f(counter);
        counter = counter + 1;
      }
    }

    return ans;
}
```

Trees

Tree Sum

```
function tree_sum(tree) {
    return accumulate_tree(x => x,
                           (x, y) => x + y, 0 , tree);
}
```

Accumulate Trees

```
function accumulate_tree(f, op, initial, tree) {
    function accum(x,y) {
        return is_list(x)
            ? accumulate_tree(f, op, y, x)
            : op(f(x),y);
    }
    return accumulate(accum, initial, tree);
}
```

Map Trees

```
function map_tree(f, tree) {
    return map(sub_tree => !is_list(sub_tree)
        ? f(sub_tree)
        : map_tree(f, sub_tree) , tree);
}
```

Scale Tree

```
function scale_tree(tree, k) {
        return map_tree(data_item =>
                        data_item * k,
                    tree);
}
```

Count Data items in Tree

```
function count_data_items(tree) {
    return accumulate_tree(x => 1,
                           (x, y) => x + y, 0 , tree);
}
```

```
let stack = null;
function push(x) {
  stack = pair(x, stack);
}

function pop() {
  const x = head(stack);
  stack = tail(stack);
  return x;
}
```

Interesting Cases:

Tower of Hanoi

```
function hanoi(n, src, dsc, aux) {
  let res = null;
  if (n === 0) {
    return null;
  } else if (n === 1) { res = append(res, list(pair(src, dsc)));
  } else {
    res = append(res, hanoi(n-1, src, aux, dsc));
    res = append(res, list(pair(src, dsc)));
    res = append(res, hanoi(n-1, aux, dsc, src));
  }
  return res; }
```

Count Island

```
function count_islands(emap) {
  const R = array_length(emap);
  const C = array_length(emap[0]);
  const label = [];
  let island_count = 0;
  function label_island(row, col, island_id) {
    if ( row >= 0 && row < R && col >= 0 && col < C ) {
      if (emap[row][col] !== 0 && label[row][col] === 0 ) {
        label[row][col] = island_id;
        label_island(row, col - 1, island_id);
        label_island(row, col + 1, island_id);
        label_island(row - 1, col, island_id);
        label_island(row + 1, col, island_id);
      } else {}
    } else {}   }
  for (let row = 0; row < R; row = row + 1) {
    label[row] = [];
    for (let col = 0; col < C; col = col + 1) {
      label[row][col] = 0;
    }
  }
  for (let row = 0; row < R; row = row + 1) {
    for (let col = 0; col < C; col = col + 1) {
      if (emap[row][col] !== 0 && label[row][col] === 0) {
        island_count = island_count + 1;
        label_island(row, col, island_count);
      } else {}
    }
  }
  return island_count;
}
```

Spiral Matrix

```
function make_spiral(n, m) {
  const matrix = [];
  for (let i = 0; i < n; i = i+ 1) {
    matrix[i] = [];
    for (let j = 0; j < m; j = j+1) {
      matrix[i][j] = 0;
    }
  }
  let top = 0;
  let bot = n-1;
  let left = 0;
  let right = m-1;
  let num = 1;
  while (num <= n*m) {
    for (let i = left; i < right+1; i = i + 1) {
      matrix[top][i] = num;
      num = num+1;
    }
    for (let i = top+1; i < bot+1; i = i+1) {
      matrix[i][right] = num;
      num = num+1;
    }
    if (top !== bot) {
      for (let i = right - 1; i > left-1; i = i - 1) {
        matrix[bot][i] = num;
        num = num + 1;
      }
    }
    if (left !== right) {
      for (let i = bot - 1; i > top; i = i - 1) {
        matrix[i][left] = num;
        num = num+1;
      }
    }
```

```
      left = left + 1;
      top = top + 1;
      right = right -1;
      bot = bot - 1;
   }
   return matrix;
}
```

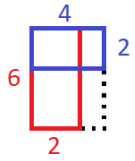
```
Smallest Bounding Area:
const get_x = (aar) => list_ref(aar, 0);
const get_y = (aar) => list_ref(aar, 1);
const get_width = (aar) => list_ref(aar, 2);
const get_height = (aar) => list_ref(aar, 3);

function smallest_bounding_AAR_area(rs) {

   let min_x = Infinity;
   let min_y = Infinity;
   let max_x = -Infinity;
   let max_y = -Infinity;

   for (let p = rs; !is_null(p); p = tail(p)) {
      const aar = head(p);
      const x1 = get_x(aar);
      const x2 = x1 + get_width(aar);
      const y1 = get_y(aar);
      const y2 = y1 + get_height(aar);

      if (x1 < min_x) { min_x = x1; }
      if (x2 > max_x) { max_x = x2; }
      if (y1 < min_y) { min_y = y1; }
      if (y2 > max_y) { max_y = y2; }
   }
   return (max_x - min_x) * (max_y - min_y);
}
//returns the area of the smallest AAR that bounds
//(encloses) all the axis-aligned rectangle in rs
```

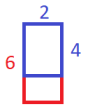


Area enclosed = 6 x 4 =24

```
function optimized_smallest_bounding_AAR_area(rs) {
   let max_longer = 0;
   let max_shorter = 0;

   for (let p = rs; !is_null(p); p = tail(p)) {
      const aar = head(p);
      const width = get_width(aar);
      const height = get_height(aar);
      const longer = math_max(width, height);
      const shorter = math_min(width, height);

      if (longer > max_longer) { max_longer = longer; }
      if (shorter > max_shorter) { max_shorter = shorter; }
   }
   return max_longer * max_shorter;
}
```



Area enclosed = 6 x 2 = 12

```
const get_x = (aar) => list_ref(aar, 0);
const get_y = (aar) => list_ref(aar, 1);
const get_width = (aar) => list_ref(aar, 2);
const get_height = (aar) => list_ref(aar, 3);

function overlap_area(aar1, aar2) {
   // [a, b] and [c, d] are the input intervals.
   function overlap_length(a, b, c, d) {
      return math_max(0, math_min(b, d) - math_max(a, c));
   }

   const x_overlap = overlap_length(
                  get_x(aar1), get_x(aar1) + get_width(aar1),
                  get_x(aar2), get_x(aar2) + get_width(aar2));

   const y_overlap = overlap_length(
                  get_y(aar1), get_y(aar1) + get_height(aar1),
                  get_y(aar2), get_y(aar2) + get_height(aar2));

   return x_overlap * y_overlap;
}
```

https://leetcode.com/problems/rectangle-area/description/%5C

Tree_to stream (array,js)
Stream_to_tree(array.js)
Deep_tree_ref(array.js)
Fast_fib(array.js) (fib in log(n))