CS1101S Final Assessment Cheatsheet AY24/25

by yghern

Recurrence Relations

$$
\begin{aligned}
T(n) &= T(n-c) + O(1) && \Rightarrow O(n) \\
T(n) &= 2\ T(\frac{n}{2}) + O(n) && \Rightarrow O(n \log n) \\
T(n) &= T(\frac{n}{2}) + O(n) && \Rightarrow O(n) \\
T(n) &= 2\ T(\frac{n}{2}) + O(1) && \Rightarrow O(n) \\
T(n) &= T(\frac{n}{2}) + O(1) && \Rightarrow O(\log n) \\
T(n) &= 2\ T(n-1) + O(1) && \Rightarrow O(2^n) \\
T(n) &= 2\ T(\frac{n}{2}) + O(n \log n) && \Rightarrow O(n(\log n)^2) \\
T(n) &= 2\ T(\frac{n}{4}) + O(1) && \Rightarrow O(\sqrt{n}) \\
T(n) &= T(n-c) + O(n) && \Rightarrow O(n^2)
\end{aligned}
$$

Lists: A list is either null or a pair whose tail is a list. A list of a certain type is either null or a pair whose head is of that type and whose tail is a list of that type.

```
function flatten_list(xs) {
    return accumulate(append, null, xs);
}

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

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

function subsets(s) {
    return accumulate(
        (x, s1) => append(s1,
                   map(ss => pair(x, ss), s1)),
        list(null),
        s);
}

function combis(xs, r) {
    if ( (r !== 0 && xs === null) || r < 0) {
        return null;
    } else if (r === 0) {
        return list(null);
    } else {
        const without = combis(tail(xs), r);
        const with = map(x => pair(head(xs), x),
                         combis(tail(xs), r - 1));
        return append(without, with);
    }
}

// Continuation-Passing Style (CPS)
function append_iter(xs, ys) {
    function app(current_xs, ys, c) {
        return is_null(current_xs)
            ? c(ys)
            : app(tail(current_xs), ys,
                x => c(pair(head(current_xs), x)));
    }
    return app(xs, ys, x => x);
}
```

Mutable Lists

```
function d_append(xs, ys) {
    if (is_null(xs)) {
        return ys;
    } else {
        set_tail(xs, d_append(tail(xs), ys));
        return xs;
    }
}

function d_reverse(xs) {
    if (is_null(xs) || is_null(tail(xs))) {
        return xs;
    } else {
        const temp = d_reverse(tail(xs));
        set_tail(tail(xs), xs);
        set_tail(xs, null);
        return temp;
    }
}

function d_remove(v, xs) {
    function helper(ys) {
        if (is_null(ys)) {
            return null;
        } else if (head(ys) === v) {
            return tail(ys);
        } else {
            set_tail(ys, helper(tail(ys)));
            return ys;
        }
    }
    return helper(xs);
}

function d_map(f, xs) {
    if (is_null(xs)) {
        return null;
    } else {
        set_head(xs, f(head(xs)));
        d_map(f, tail(xs));
        return xs;
    }
}

function d_filter(pred, xs) {
    if (is_null(xs)) {
        return null;
    } else if (pred(head(xs))) {
        set_tail(xs, d_filter(pred, tail(xs)));
        return xs;
    } else {
        return d_filter(pred, tail(xs));
    }
}
```

Trees: A tree of a certain data type is a list whose elements are of that data type, or trees of that data type.

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

function flatten_tree(tree) {
    if (is_null(tree)){
        return null;
    } else if (is_list(head(tree))) {
        return append(head(tree),
                      flatten_tree(tail(tree)));
    } else {
        return pair(head(tree),
                    flatten_tree(tail(tree)));
    }
}

function accumulate_tree(f, op, initial, tree) {
    return accumulate(
        (x, ys) => is_list(x)
            ? op(accumulate_tree(f, op, initial, x),
                                 ys)
            : op(f(x), ys),
        initial,
        tree
    );
}
```

Binary Search Trees

```
function insert(bst, item) {
    if (is_empty_tree(bst)) {
        return make_tree(item, make_empty_tree(),
                         make_empty_tree());
    } else {
        if (item < entry(bst)) {
            return make_tree(entry(bst),
                          insert(left_branch(bst),
                                 item),
                          right_branch(bst));
        } else if (item > entry(bst)) {
            return make_tree(entry(bst),
                          left_branch(bst),
                          insert(right_branch(bst),
                                 item));
        } else {
            return bst;
        }
    }
}

function find(bst, name) {
    return is_empty_tree(bst)
        ? false
        : name === entry(bst)
            ? true
            : name < entry(bst)
                ? find(left_branch(bst), name)
                : find(right_branch(bst), name);
}
```

Streams: A stream is either the empty list, or a pair whose tail is a nullary function that returns a stream.

Arrays: An array is a data structure that stores a sequence of data elements.

```
function swap(A, x, y) {
    const temp = A[x];
    A[x] = A[y];
    A[y] = temp;
}
```

Array Searching

```
function linear_search(A, v) {
    const len = array_length(A);
    let i = 0;
    while (i < len && A[i] !== v) {
        i = i + 1;
    }
    return (i < len);
}

function binary_search(A, v) {
    function search(low, high) {
        if (low > high) {
            return false;
        } else {
            const mid =
                math_floor((low + high) / 2);
            return (v === A[mid]) ||
                   (v < A[mid]
                    ? search(low, mid - 1)
                    : search(mid + 1, high));
        }
    }
    return search(0, array_length(A) - 1);
}
```

Sorting Algorithms

Algo	Time Complexity			Space
	Best	Average	Worst	
Sel	$O(n^2)$	$O(n^2)$	$O(n^2)$	$O(1)$
Ins	$O(n)$	$O(n^2)$	$O(n^2)$	$O(1)$
Bub	$O(n)$	$O(n^2)$	$O(n^2)$	$O(1)$
Mer	$O(n \log n)$	$O(n \log n)$	$O(n \log n)$	$O(n)$
Qck	$O(n \log n)$	$O(n \log n)$	$O(n^2)$	$O(\log n)$

Sorting (Lists)

```
function selection_sort(xs) {
    if (is_null(xs)) {
        return xs;
    } else {
        const x = smallest(xs);
        return pair(x,
                    selection_sort(remove(x, xs)));
    }
}
function smallest(xs) {
    return accumulate((x, y) => x < y ? x : y,
                      head(xs), tail(xs));
}

function insertion_sort(xs) {
    return is_null(xs)
           ? xs
           : insert(head(xs),
                    insertion_sort(tail(xs)));
}
function insert(x, xs) {
    return is_null(xs)
           ? list(x)
           : x <= head(xs)
           ? pair(x, xs)
           : pair(head(xs), insert(x, tail(xs)));
}

function bubble_sort(xs) {
    const len = length(xs);
    for (let i = len - 1; i >= 1; i = i - 1) {
        let p = xs;
        for (let j = 0; j < i; j = j + 1) {
            if (head(p) > head(tail(p))) {
                const temp = head(p);
                set_head(p, head(tail(p)));
                set_head(tail(p), temp);
            }
            p = tail(p);
        }
    }
}

function quicksort(xs) {
    return is_null(xs)
        ? null
        : append(
            quicksort(head(
                 partition(tail(xs), head(xs)))),
            pair(head(xs),
                 quicksort(tail(
                 partition(tail(xs), head(xs))))
            )
          );
}
function partition(xs, p) {
    return pair(
        filter(x => x <= p, xs),
        filter(x => x > p, xs)
    );
}

function merge_sort(xs) {
    if (is_null(xs) || is_null(tail(xs))) {
        return xs;
    } else {
        const mid = math_floor(length(xs) / 2);
        return merge(merge_sort(take(xs, mid)),
                     merge_sort(drop(xs, mid)));
    }
}
function merge(xs, ys) {
    if (is_null(xs)) {
        return ys;
    } else if (is_null(ys)) {
        return xs;
    } else {
        const x = head(xs);
        const y = head(ys);
        return (x < y)
            ? pair(x, merge(tail(xs), ys))
            : pair(y, merge(xs, tail(ys)));
    }
}
function take(xs, n) {
    return n === 0
        ? null
        : pair(head(xs),
               take(tail(xs), n - 1));
}
function drop(xs, n) {
    return n === 0
        ? xs
        : drop(tail(xs), n - 1);
}
```

Sorting (Arrays)

```
function selection_sort(A) {
    const len = array_length(A);

    for (let i = 0; i < len - 1; i = i + 1) {
        let min_pos = find_min_pos(A, i, len - 1);
        swap(A, i, min_pos);
    }
}
function find_min_pos(A, low, high) {
    let min_pos = low;
    for (let j = low + 1; j <= high; j = j + 1) {
        if (A[j] < A[min_pos]) {
            min_pos = j;
        }
    }
    return min_pos;
}

function insertion_sort(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;
    }
}

function bubble_sort(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]) {
                swap(A, j, j + 1);
            }
        }
    }
}

function merge_sort(A) {
    merge_sort_helper(A, 0, array_length(A) - 1);
}
function merge_sort_helper(A, low, high) {
    if (low < high) {
        const mid = math_floor((low + high) / 2);
        merge_sort_helper(A, low, mid);
        merge_sort_helper(A, mid + 1, high);
        merge(A, low, mid, high);
    }
}
function merge(A, low, mid, high) {
    const B = [];
    let left = low;
    let right = mid + 1;
    let Bidx = 0;

    while (left <= mid && right <= high) {
        if (A[left] <= A[right]) {
            B[Bidx] = A[left];
            left = left + 1;
        } else {
            B[Bidx] = A[right];
            right = right + 1;
        }
        Bidx = Bidx + 1;
    }

    while (left <= mid) {
        B[Bidx] = A[left];
        Bidx = Bidx + 1;
        left = left + 1;
    }
    while (right <= high) {
        B[Bidx] = A[right];
        Bidx = Bidx + 1;
        right = right + 1;
    }

    for (let k = 0; k < high - low + 1; k = k + 1) {
        A[low + k] = B[k];
    }
}

function quicksort(A) {
    quicksort_helper(A, 0, array_length(A) - 1);
}
function quicksort_helper(A, low, high) {
    if (low < high) {
        const pi = partition(A, low, high);
        quicksort_helper(A, low, pi - 1);
        quicksort_helper(A, pi + 1, high);
    }
    return A;
}
function partition(A, low, high) {
    const pivot = A[high];
    let i = low - 1;
    for (let j = low; j < high; j = j + 1) {
        if (A[j] < pivot) {
            i = i + 1;
            swap(A, i, j);
        }
    }
    swap(A, i + 1, high);
    return i + 1;
}
```

Memoization (1D/2D/Streams)

```
function memoize(f) {
    const mem = [];
    function mf(x) {
        if (mem[x] !== undefined) {
            return mem[x];
        } else {
            const result = f(x);
            mem[x] = result;
            return result;
        }
    }
    return mf;
}

const mem = [];
function read(n, k) {
    return mem[n] === undefined
           ? undefined
           : mem[n][k];
}
function write(n, k, value) {
    if (mem[n] === undefined) {
        mem[n] = [];
    }
    mem[n][k] = value;
}
function mchoose(n, k) {
    if (read(n, k) !== undefined) {
        return read(n, k);
    } else {
        const result = k > n
                       ? 0
                       : k === 0 || k === n
                       ? 1
                       : mchoose(n - 1, k) +
                         mchoose(n - 1, k - 1);
        write(n, k, result);
        return result;
    }
}

function memo_fun(fun) {
    let already_run = false;
    let result = undefined;
    function mfun() {
        if (!already_run) {
            result = fun();
            already_run = true;
            return result;
        } else {
            return result;
        }
    }
    return mfun;
}
```