CS1101S Midterms Cheatsheet AY23/24

by chrisgzf (AY19/20), modified by Zeepheru for AY23/24

Recursive/Iterative: Check if there are deferred operations

```
function fact_iter(n) {
    function mult_remaining(counter , product) {
        return counter === 1
            ? product
            : mult_remaining(counter - 1, product
            * counter);
    }
    return mult_remaining(n, 1);
}

function fib(n) {
    function f(n, k, x, y) {
        return (k > n) ? y : f(n, k + 1, y, x + y);
    }
    return (n < 2) ? n : f(n, 2, 0, 1);
}

function gcd(a, b) {
    return b === 0 ? a : gcd(b, a % b);
}

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

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 and tail are of that type.

```
function reverse(xs) {
    function rev(original, reversed) {
        return is_null(original)
            ? reversed
            : rev(tail(original),
                  pair(head(original), reversed));
    }
    return rev(xs ,null);
}

function append_iter(xs, ys){
    // iterative process
    function app(xs, ys, c) {
        return is_null(xs)
        ? c(ys)
        : app(tail(xs), ys,
              x => c(pair(head(xs), x))
              );
    }
    return app(xs, ys, x => x);
}

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

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

```
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(xs) {
    function h(xs, prev) {
        return is_null(xs)
            ? prev // end of list or tree
            : is_list(xs)
                ? append(flatten(xs), prev) //list
                : pair(xs, prev); // leaf
    }
    return accumulate(h, null, xs);
}
```

Besides the base case, these operations consider two cases. One, when the element is itself a tree, and another when it is not.

Binary Trees: A binary tree of a certain type is null or a list with three elements, whose first element is of that type and whose second and third elements are binary trees of that type.

Binary Search Trees: A binary search tree of Strings is a binary tree of Strings where all entries in the left subtree are smaller than its value and all entries in the right subtree are larger than its value.

```
function insert(bst, item) {
    if (is_empty_tree(bst)) {
        return make_tree(item, make_empty_tree(),
          make_empty_tree());
    } else {
        if (item < entry(bst)) {
            // smaller than i.e. left branch
            return make_tree(entry(bst),
                        insert(left_branch(bst),
                               item),
                        right_branch(bst));
        } else if (item > entry(bst)) {
            // bigger than entry i.e. right branch
            return make_tree(entry(bst),
                        left_branch(bst),
                        insert(right_branch(bst),
                               item));
        } else {
            // equal to entry.
            // BSTs should not contain duplicates
            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);
}
```

Permutations & Combinations

```
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 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;
    }
}
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);
    }
}

function makeup_amount(x, coins) {
    if (x === 0) {
        return list(null);
    } else if (x < 0 || is_null(coins)) {
        return null;
    } else {
        // Combinations that do not use the head coin.
        const combi_A = makeup_amount(x, tail(coins));
        // Combinations that do not use the head coin
        // for the remaining amount.
        const combi_B = makeup_amount(x - head(coins),
                                      tail(coins));
        // Combinations that use the head coin.
        const combi_C = map(x => pair(head(coins), x),
                            combi_B);
        return append(combi_A, combi_C);
    }
}
```

Orders of Growth

(Limits used for succinctness)

Big Theta, **Big Omega**, and **Big O**:

$\theta(g(n)) \iff \exists\, k_1, k_2 \in \mathbb{Z}^+\ \exists\, n_0 \in \mathbb{R}$
$(\forall\, n > n_0 (k_1 \cdot g(n) \le r(n) \le k_2 \cdot g(n)))$

$O(g(n)) \iff \exists\, k \in \mathbb{Z}^+ (\lim_{n\to\infty}(k \cdot g(n) \ge r(n)))$

$\Omega(g(n)) \iff \exists\, k \in \mathbb{Z}^+ (\lim_{n\to\infty}(k \cdot g(n) \le r(n)))$

Order (small to big): 1, $\log n$, n, $n \log n$, n^2, n^3, 2^n, 3^n, n^n

Note: $r(n)$ has OOGs $\theta(r(n))$, $O(r(n))$, and $\Omega(r(n))$.

Common Recurrence Relations

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

Generally, $T(n) = O(n^k) + T(n-1) \implies O(n^{k+1})$

Insertion sort takes elements from left to right, and *inserts* them into correct positions in the sorted portion of the list (or array) on the left. This is analagous to how most people would arrange playing cards.

Time Complexity: $\Omega(n)$ $O(n^2)$

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

function insertion_sort(xs) {
    return is_null(xs)
        ? xs
        : insert(head(xs),
                 insertion_sort(tail(xs)));
}
```

Selection sort picks the smallest element from a list (or array) and puts them in order in a new list.

Time Complexity: $\Omega(n^2)$ $O(n^2)$

```
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) {
    function h(xs, min) {
        return xs === null
            ? min
            : head(xs) < min
                ? h(tail(xs), head(xs))
                : h(tail(xs), min);
    }
    return h(xs, head(xs));
}
```

Quicksort is a divide-and-conquer algorithm. `Partition` takes a pivot, and positions all elements smaller than the pivot on one side, and those larger on the other. The two 'sides' are then partitioned again.

Time Complexity: $\Omega(nlogn)$ $O(n^2)$

```
function partition(xs, p) {
    function h(xs, lte, gt) {
        if (is_null(xs)) {
            return pair(lte, gt);
        } else {
            const first = head(xs);
            return first <= p
                ? h(tail(xs), pair(first, lte), gt)
                : h(tail(xs), lte, pair(first, gt));
        }
    }
    return h(xs, null, null);
}


function quicksort(xs) {
    if (is_null(xs) || is_null(tail(xs))) {
        return xs;
    } else {
        const pivot = head(xs);
        const splits = partition(tail(xs), pivot);
        const smaller = quicksort(head(splits));
        const bigger = quicksort(tail(splits));
        return append(smaller, pair(pivot, bigger));
    }
}
```

Mergesort is a divide-and-conquer algorithm.

Time Complexity: $\Omega(nlogn)$ $O(nlogn)$

```
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);
}

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 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)));
    }
}
```

Personal Additions

```
function accumulate(op, initial, xs) {
    return is_null(xs)
           ? initial
           : op(head(xs),
             accumulate(op, initial, tail(xs)));
}

function remove_duplicates(lst) {
    return accumulate(
                  (x, y) => is_null(member(x, y))
                                ? pair(x, y) : y,
                  null, lst);
}

function subsets(xs) {
    if (is_null(xs)) {
        return list(null);
    } else {
        const no_subsets = subsets(tail(xs));
        const yes_subsets = map(
                  x => pair(head(xs), x),
                            subsets(tail(xs)));

        return append(no_subsets, yes_subsets);
    }
}

function subsets_acc(xs) {
    return accumulate(
                (x, y) =>
                    append(y, map(t => pair(x, t), y)),
                list(null), xs);
}
```

Misc Notes

They do like to ask on **active lists**. In all prior implementations, **active lists** are **FUNCTIONS**. So all operations on them should make use of this $\implies$ use the indexes.

- Also note that an active list called on an index returns a **list** of length 1.

Also more generally, remember to use the **properties** of the data structures given! Eg: list is sorted, left tree is smaller than right tree, etc.

Master Theorem for **recurrence relations** (Citation Needed):
For $T(n) = aT(\frac{n}{b}) + \Theta(n^d),\ a \geq 1,\ b > 1,\ d \geq 0,$

$$a < b^d \implies \Theta(n^d)$$
$$a = b^d \implies \Theta(n^d \log n)$$
$$a > b^d \implies \Theta(n^{\log_b a})$$