CS1101S Cheatsheet

for finals AY23-24

m. **zaidan**

Recursion & Iteration

Recursion: Increasing deferred operations
Iteration: Constant deferred operations
(rule of thumb: if the final call is the application call, it is usually *iterative*)

Definitions

List: either `null` or a pair whose head is of that type and tail is a list of that type.
Streams: either `null` or a pair whose head is of that type and tail is a nullary function that returns a stream of that type.

Orders of Growth

for $r(n)$,

Big O $O(g(n))$**:** if there is a positive constant k such that $r(n) \leq k * g(n)$ for any sufficiently large value of n
Big Theta $\theta(g(n))$**:** if there is positive constants k_1 and k_2 and a number n_0 such that $k_1 * g(n) \leq r(n) \leq k_2 * g(n)$ for any n $> n_0$.
Big Omega $\Omega(g(n))$**:** if there is a positive constant k such that $k * g(n) \leq r(n)$ for any sufficiently large value of n
Order (ascending): 1, log n, n, n log n, n^2, n^3, 2^n, 3^n, n^n

Ignore constants, lower order terms. $O(2n) = O(3n) = O(n)$
For a sum, take the larger term. $O(n) + O(n^2) = O(n^2)$
For a product, multiply the two terms. $O(n) \times O(n) = O(n^2)$

Recurrence Relations

O(1) + T(n-1)	
O(1) + 2T(n/2)	O(n)
O(n) + T(n/2)	
O(1) + T(n/2)	O(log n)
O(log n) + T(n-1)	
O(n) + 2T(n/2)	O(n log n)
O(n) + T(n-1)	$O(n^2)$
$O(n^k)$ + T(n-1)	$O(n^{k+1})$
O(n) + 2T(n-1)	$O(2^n)$

Sorting/Search Algorithms

	Time		Space
Binary Search	θ(log n)	O(log n)	O(1)
Selection Sort	$\theta(n^2)$	$O(n^2)$	O(n)
Insertion Sort	θ(n)	$O(n^2)$	O(n)
Merge Sort	θ(n log n)	O(n log n)	O(n log n)
Quick Sort	θ(n log n)	$O(n^2)$	$O(n^2)$

Rule of Thumb for Abstractions

Is input a list?	if not, don't use
Is length(output) = length(input)?	use *map*
Is items in output = items in input?	use *filter*
Else	use *accumulate*

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 {
        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 = combinations(tail(xs), r);
        const yes = combinations(tail(xs), r
            - 1);
        const yes_item = map(x => pair(head(
            xs), x), yes);
        return append(no, yes_item);
    }
}
```

Rule of Thumb for Environment Model

1. `function` are `const` declarations.
2. `const` appear as := and `var` appear as :
3. Create a frame if there are arguments passed into the `function`
4. Create a frame if there are declarations within a block, otherwise do not create empty frames.
5. Better practice to cancel out old values instead of erasing them.

Sorts and Searches

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


// Selection Sort
function smallest(xs) {
    return accumulate((x, y) => x<y ? x:y,
                      head(xs), tail(xs));
}

function selection_sort(xs) {
    if (is_null(xs)) {
        return xs;
    } else {
        const x = smallest(xs);
        return pair(x, selection_sort(remove(
            x, xs)));
    }
}

// Insertion Sort
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)));
}

// Merge Sort
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)));
    }
}
```

```
const middle = x => math_floor(x/2);


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_sort(xs) {
    if (is_null(xs) || is_null(tail(xs))) {
       return xs;
    } else {
       const m = middle(length(xs));
       return merge(merge_sort(take(xs, m)),
                    merge_sort(drop(xs, m)));
    }
}
```

Trees

```
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);
}

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(T) {
    return accumulate((x, ys) => is_list(x)
                        ? append(flatten_tree
                            (x), ys)
                        : append(list(x), ys)
                            ,
                      null, T);
}
```

Memoisation

```
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;
}
```

Streams

```
function add_streams(s1, s2) {
    return is_null(s1)
    ? s2
    : is_null(s2)
        ? s1
        : pair(head(s1) + head(s2),
            () => add_streams(stream_tail(s1)
                ,
            stream_tail(s2)));
}
function mul_streams(s1, s2) {
    return is_null(s1)
        ? s2
        : is_null(s2)
        ? s1
        : pair(head(s1) * head(s2),
               () => mul_streams(stream_tail(
                   s1),
                                 stream_tail(
                                     s2)));
}

function scale_stream(s, f) {
    return stream_map(x => x * f, s);
}

function shorten_stream(s, k) {
    return k > 0 && !is_null(s)
           ? pair(head(s), () =>
               shorten_stream(stream_tail(s),
               k-1))
           : null;
}

function zip_streams(s1, s2) {
    return pair(head(s1), () => zip_streams(
        s2, stream_tail(s1)));
}
```

T-Diagrams