CS1101S Midterms Cheatsheet AY19/20

by chrisgzf (adapted from https://github.com/ning-y/Cheatsheets/)

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);
}
```

Order of Growth

Big Theta: The function r has order of growth $\theta(g(n))$ if there are 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 O: The function r has order of growth $O(g(n))$ if there is a positive constant k such that r(n) $\leq k * g(n)$ for any sufficiently large value of n

Big Omega: The function r has order of growth $\Omega(g(n))$ if there is a positive constant k such that $k * g(n) \leq$ r(n) for any sufficiently large value of n

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

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 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))));
}
```

Passing the deferred operation as a function in an extra argument is called "Continuation-Passing Style" (CPS).

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);
    }
}
```

```
Function Repeater
((thrice(twice))(function) = repeat
function (2^3) times

rep-x(rep-y)(f) = y ^ x times

twice(twice)(twice)(plus_one)(0); = 2^2^2
rep-x(rep-y(f)) = y * x times

n_times(twice,3)(plus_one)(0); = 2*2*2 =
2^3
```

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));
}
```

```
Orders of Growth:
T(n) = 2T(n/2) + O(n)

=> O(nlogn)
T(n) = T(n/2) + O(n)

=> O(n)
T(n) = 2T(n/2) + O(1)

=> O(n)
T(n) = T(n/2) + O(1)

=> O(logn)
```

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));
    }
}
```

```
Sorting methods

Insertion Sort
Worst n^2. Average n^2. Best n
Sort the tail using wishful thinking,
insert the head

Selection Sort
Worst n^2. Average n^2. Best n^2
Find the smallest element x and remove
it from the list. Sort the remaining
list, and put x in front.


Quicksort
Worst n^2. Average nlogn. Best nlogn
Partition using head of list, then sort
each sublist (lte, gt) then append.

Mergesort
Worst nlogn. Average nlogn. Best nlogn.
Split the list in half, sort each half
using wishful thinking, merge the
sorted lists together.
```

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)));
    }
}
```

Map
```
function map(f, xs) {
  return is_null(xs) ? null :
  pair(f(head(xs)), map(f, tail(xs)));
}
```

Filter
```
function filter(pred, xs) {
return is_null(xs) ? xs
    : pred(head(xs))
        ? pair(head(xs), filter(pred, tail(xs)))
        : filter(pred, tail(xs));
}
```

Accumulate
```
function accumulate(f, initial, xs) {
    return is_null(xs) ? initial
         :f(head(xs),
             accumulate(f, initial, tail(xs)));
}
accumulate(f, null, list(2, 3, 4)) =
f(2, f(3, f(4, null)))

//filter using accumulate
function my_filter(pred , xs) {
    return accumulate( (x,acc) => pred(x) ?
    pair(x,acc) : acc , null , xs);
}

//map using accumulate
function my_map(f, xs) {
    return accumulate((x,ys) => pair(f(x),ys),
     null, xs);
}
```

```
Permutations
// SOLUTION 1:
if (is_null(xs)) {
    return list(null);
} else {
    const s = permutations(tail(xs));
    const t = map(ys => insertions(head(xs), ys), s);
    return accumulate(append, null, t);
}

// SOLUTION 2:
return accumulate((x, ps) => accumulate((p, qs) => append(insertions(x,
p), qs), null, ps), list(null), xs);

// SOLUTION 3:
return accumulate((x, ps) => accumulate(append, null, map(p =>
insertions(x, p), ps)), list(null), xs);
```

```
/* General Helper Functions */

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

/* Miscellaneous Section */

// Misc 1: Sublist
function sublist(xs) {
    if (is_null(xs)) {
        return list(xs);
    } else {
        return append(map(x => pair(head(xs), x),
sublist(tail(xs))), sublist(tail(xs)));
    }
}

// Misc 2: Permutation
function permutation(xs) {
    return is_null(xs)
        ? list(xs)
        : accumulate(append, null, map(x => map(p =>
pair(head(xs), p), permutation(remove(x,
        xs))), xs);
}

// Misc 3: Combination
function combinations(xs, r) {
    if ( (r !== 0 && xs === null) || r < 0) {
        return null;
    } else if (r === 0) {
        return list(null);
    } else {
        const elt = head(xs);
        const front = map(x => pair(elt, x),
combinations(tail(xs), r - 1));
        const end = combinations(tail(xs), r);
        return append(front, end);
    }
}
```

```
Check if 2 sets are permutations of each other
function are_equal_sets(set1, set2) {
       if (length(set1) !== length(set2)) { return false; }
       else {
              return accumulate(
          (x1, y1) => accumulate( (x2, y2) => x1 === x2 ||
y2, false, set2) && y1,
          true, set1);
}
```

```
/* Continuation Passing Style */

function append_iter(xs, ys) {
  function app(xs, ys, c) {
    return is_null(xs)
          ? c(ys) // instead of returning result, return c(result)

  /* wrap the call of pair in pair(head(xs), append(tail(xs), ys)) in function call
  and then cast it back as a function. NOTE: x passed here is the 'wishful thinking' */

          : app(tail(xs), ys, x => c(pair(head(xs), x)));
  }
  return app(xs, ys, x => x);
}

/* Array & Matrix Functions */
// transposing matrix

function transpose(arr) {
    const len = array_length(arr);
    for (let i = 0; i < len; i = i + 1) {
        for (let j = 0; j < i; j = j + 1) {
            const temp = arr[i][j];
            arr[i][j] = arr[j][i];
            arr[j][i] = temp;
        }
    }
    return arr;
}

// reverse an array
function reverse(arr) {
    const len = array_length(arr);
    const maxidx = len - 1;
    for (let i = 0; i < math_floor(len / 2); i = i + 1) {
        const temp = arr[i];
        arr[i] = arr[maxidx - i];
        arr[maxidx - i] = temp;
    }
    return arr;
}
/* Streams stuff */
function sieve(s) {
    return pair(head(s),() => sieve(stream_filter(x => !is_divisible(x, head(s)),
stream_tail(s))));
}
```

```
function coin_change(amount, coin_range) {
      if (amount === 0) {
            return 1;
      } else if (amount < 0 || coin_range === 0) {
            return 0;
      } else {
            return coin_change(amount, coin_range - 1)
            + coin_change(
            amount - highest_value(coin_range),
            coin_range
            );
      }
}
```

```
/* Memoization */

// read and write

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] = [];
    } else {}
    mem[n][k] = value;
}

// stream memoize (or rather
memoize nullary function)

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;
}
```

```
//Selection sort
function selection_sort(xs) {
    if (is_empty_list(xs)) {
        return xs;
    } else {
        return x = smallest(xs);
        return pair(x, selection_sort(
        remove(x,xs))
        );
    }
}

function smallest(xs) {
    function sm(x, ys) {
        return (is_empty_list(ys))
            ? x
            : (x < head(ys))
                ? sm(x, tail(ys))
                : sm(head(ys), tail(ys));
    }

    return sm(head(xs), tail(xs));
}

//BUBBLE SORT
function sort(b) {
    let length_b = array_length(b);
    for (let num_sorted = 0;
    num_sorted < length_b;
    num_sorted = num_sorted + 1
    ) {
        for (let index = 0;
        index < length_b - 1;
        index = index + 1
        ) {
            if (b[index] > b[index+1]) {
                swap(index, index+1, b);
            } else { }
        }
    }
    return undefined;
}
function swap(left_index, right_index, array)
{
    let tmp = array[left_index];
    array[left_index] = array[right_index];
    array[right_index] = tmp;
}
```

```
// QUICKSORT

function quicksort(xs) {
    if (is_empty_list(xs)) {
        return [];
    } else if (length(xs) === 1) {
        return xs;
    } else {
        let partitioned_xs =
        partition(tail(xs), head(xs));
        return append(
        quicksort(head(partitioned_xs)),
        pair(head(xs),
        quicksort(tail(partitioned_xs)))
        );
    }
}

function partition(xs, p) {
    function helper(elem, out_pair) {
        if (elem <= p) {
            return pair(
            append(head(out_pair), list(elem)),
            tail(out_pair)
            );
        } else {
            return pair(
            head(out_pair),
            append(tail(out_pair), list(elem))
            );
        }
    }
    return accumulate(helper, pair([], []), xs);
}

//Insertion sort

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

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

CS1101S Cheatsheet 2017

by ning

Basic Syntax and Built-in Functions

```
for (var i = 0; i < 5; i = i+1) {
    // do something
}

while (i < 10) {
    // do something
}

// join many lists together
accumulate(append, [], xxs);

// clone a list
map(function(x) { return x; }, xs);

// clone an object
JSON.parse(JSON.stringify(obj));
```

- `parseInt(string)`: Interprets the given `string` as an integer, and returns that integer
- `math_pi`: Refers to the number π, the mathematical constant
- `math_max(a, b, c, ...)`: Returns the largest argument given
- `math_floor(x)`: Returns the largest integer smaller than `x`
- `math_ceil(x)`: Returns the smallest integer larger than `x`
- `math_sqrt(x)`: Refers to the square root function, returns $\sqrt{x}$
- `math_pow(base, exponent)`: Returns `base` to the power of `exponent`, i.e. b^e
- `is_number(x)`, `is_boolean(x)`, `is_string(x)`: Returns `true` ifx is the respective primitive data type, else `false`
- `is_function(x)`: Returns `true` if `x` is a function, else `false`
- `is_object(x)`: Returns `true` if `x` is an object, else `false`. Note that functions and arrays are objects
- `is_array(x)`: Returns `true` if `x` is an array, else `false`. Note that the empty array `[]`, also known as the empty list, is an array
- `equal(x, y)`: Returns `true` if `x` and `y` have the same strucutre, and corresponding leaves are `===`, else `false`
- `array_length(x)`: Returns the current length of array `x`
- `apply_in_underlying_javascript(f, xs)`: calls the function `f` with arguments `xs`

Substitution Model

```
function plus_one(x) {
    return x + 1;
}

function twice(f) {
    return function(x) {
        return f(f(x));
    }
}

function n_times(f, n) {
    if (n === 1) {
        return f;
    } else {
        return function(x) {
            return f((n_times(f, n-1))(x);
        }
    }
}

function chain(f, n) {
    if (n === 1) {
        return f;
    } else {
        return (chain(f, n-1))(f);
    }
}
```

`plus_one(plus_one(0))`
$pp(0) \to p(1) \to 2$

`(twice(plus_one))(0)`
$t(p)0 \to pp(0) \to 2$

`(n_times(plus_one,4))(0)`
$n(p,4)(0) \to p(n(p,3))(0) \to pp(n(p,2))(0)$
$\to ppp(n(p,1))(0) \to pppp(0) \to 4$

`((n_times(twice,3))(plus_one))(0)`
$n(t,3)(p)(0) \to ttt(p)(0) \to ttpp(0) \to tpppp(0)$
$\to pppppppp(0) \to p^8(0) \to 8$

`((n_times(chain(twice,3),2))(plus_one))(0)`
$n(c(t,3),2)(p)(0) \to n(c(t,2)(t),2)(p)(0)$
$\to n(c(t,1)(tt),2)(p)(0) \to n(t(tt),2)(p)(0)$
$\to n(tttt,2)(p)(0) \to t^8(p)(0) \to p^{2^8}(0)$
$\to p^{256}(0) \to 256$

`((chain(twice,4))(plus_one))(0)`
$((c(t,4))(p))(0) \to c(t,3)(t)(p)(0) \to c(t,2)(tt)(p)(0)$
$\to c(t,1)(tttt)(p)(0) \to tttttttt(p)(0) \to t^8p(0)$
$\to p^{2^8}(0) \to p^{256} \to 256$

Environment Model

To apply a function f,

1. Create a new frame, A
2. Point A back to the environment in which f was defined
3. In A, bind the parameters of f to the values of the arguments given during the function call
4. Evaluate the body of f within A

When checking through the workings, **make sure the frames all point back to some other frame**.

Lists: a list is either the empty list `[]` or a pair whose tail is a list.

- `pair(x, y)`: Makes a pair from x and y.
- `is_pair(x)`: Returns true if x is a pair and false otherwise.
- `head(x)`: Returns the head (first component) of the pair x.
- `tail(x)`: Returns the tail (second component) of the pair x.
- `set_head(p, x)`: Sets the head (first component) of the pair p to be x; returns undefined.
- `set_tail(p, x)`: Sets the tail (second component) of the pair p to be x; returns undefined.
- `is_empty_list(xs)`: Returns true if xs is the empty list, and false otherwise.
- `is_list(x)`: Returns true if x is a list as defined in the lectures, and false otherwise. Iterative process; time: O(n), space: O(1), where n is the length of the chain of tail operations that can be applied to x.
- `list(x1, x2,..., xn)`: Returns a list with n elements. The first element is x1, the second x2, etc. Iterative process; time: O(n), space: O(n), since the constructed list data structure consists of n pairs, each of which takes up a constant amount of space.
- `length(xs)`: Returns the length of the list xs. Iterative process; time: O(n), space: O(1), where n is the length of xs.
- `map(f, xs)`: Returns a list that results from list xs by element-wise application of f. Recursive process; time: O(n), space: O(n), where n is the length of xs.
- `build_list(n, f)`: Makes a list with n elements by applying the unary function f to the numbers 0 to n - 1. Recursive process; time: O(n), space: O(n).
- `for_each(f, xs)`: Applies f to every element of the list xs, and then returns true. Iterative process; time: O(n), space: O(1), where n is the length of xs.
- `list_to_string(xs)`: Returns a string that represents list xs using the text-based boxandpointer notation [...].
- `reverse(xs)`: Returns list xs in reverse order. Iterative process; time: O(n), space: O(n), where n is the length of xs. The process is iterative, but consumes space O(n) because of the result list.
- `append(xs, ys)`: Returns a list that results from appending the list ys to the list xs. Recursive process; time: O(n), space: O(n), where n is the length of xs.
- `member(x, xs)`: Returns first postfix sublist whose head is identical to x (===); returns [] if the element does not occur in the list. Iterative process; time: O(n), space: O(1), where n is the length of xs.
- `remove(x, xs)`: Returns a list that results from xs by removing the first item from xs that is identical (===) to x. Recursive process; time: O(n), space: O(n), where n is the length of xs.
- `remove_all(x, xs)`: Returns a list that results from xs by removing all items from xs that are identical (===) to x. Recursive process; time: O(n), space: O(n), where n is the length of xs.
- `filter(pred, xs)`: Returns a list that contains only those elements for which the oneargument function pred returns true. Recursive process; time: O(n), space: O(n), where n is the length of xs.
- `enum_list(start, end)`: Returns a list that enumerates numbers starting from start using a step size of 1, until the number exceeds (¿) end. Recursive process; time: O(n), space: O(n), where n is the length of xs.
- `list_ref(xs, n)`: Returns the element of list xs at position n, where the first element has index 0. Iterative process; time: O(n), space: O(1), where n is the length of xs.
- `accumulate(op, initial, xs)`: Applies binary function op to the elements of xs from right-to-left order, first applying op to the last element and the value initial, resulting in r1, then to the second-last element and r1, resulting in r2, etc, and finally to the first element and rn1, where n is the length of the list. Thus, accumulate(op,zero,list(1,2,3)) results in op(1, op(2, op(3, zero))). Recursive process; time: O(n), space: O(n), where n is the length of xs, assuming op takes constant time.

Objects

```
var obj = {'aa': 4, 'bb': true,
    'cc': function(x) { return x * x; } };
obj['aa'] === obj.aa // true
obj['bb'] === obj.bb // true
obj['cc'] === obj.cc // true
obj['cc'](5) === obj.cc(5) // true
```

Pseudo-classical Inheritance

The `new` keyword,

1. Constructs an empty object
2. Calls the constructor function with that object as `this`
3. Returns the object

Class methods can be added within the constructor function, or dynamically by accessing the contructor's `prototype`.

```
function MyClass() {
    this.my_method = function() {
        // do something
    };
}

MyClass.prototype.another_method =
    function() {
        // do something else
    };

function OtherClass() { }

MyClass.Inherits(OtherClass);
```

Inheritance is created using `Inherits`, which sets the `__proto__` of `MyClass` to `ParentClass.prototype`.

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

```
var tree = list(list(1, 2), list(3, 4));

function count_data_items(tree) {
    return is_empty_list(tree)
        ? 0
        : (is_list(head(tree))
            ? count_data_items(head(tree))
            : 1)
        + count_data_items(tail(tree));
}

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

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

Binary Search Trees: A binary tree is the empty list or a list with three element, whose first element is a binary tree, whose second element is a data item, and whose third element is a binary tree.

The first element is called the left subtree and the third element is called the right subtree of the binary tree. The second element is called the value of the binary tree.

A binary *search* tree is a binary tree where all data items in the left subtree are smaller than its value and all data item in the right subtree are large than its value.

```
function find(bst, name) {
    if (is_empty_binary_tree(bst)) {
        return false;
    } else if (name < value_of(bst)) {
        return find(left_subtree_of(bst), name);
    } else if (value_of(bst) < name) {
        return find(right_subtree_of(bst), name);
    } else {
        return true;
    }
}
```

A balanced tree is a tree whose leaves have depths that differ by at most one.

Wishful Thinking is a critical technique in functional programming. Mastery of this technique, in my opinion, guarantees you an A.

1. Divide the solution into distinct steps
2. Find a repeating pattern in the solution steps that can be abstracted
3. Implement the first step in a function f
4. Use the function f within itself recursively to complete the rest of the solution steps

```
function append(xs, ys) {
    if (is_empty_list(xs)) {
        return ys;
    } else {
        return pair(head(xs), append(tail(xs), ys));
    }
}
```

The `append` implementation above takes the first element of `xs`, then pairs it with the next element from `xs`, unless `xs` is the empty list—then it pairs with all of `ys`. The repeating step is the parring of an element with the next. The implementation of this step is with the built-in function `pair`.

Memoization

```
function mfib(n) {
    var mem = [];
    function fib(k) {
        if (mem[k] !== undefined) {
            return mem[k];
        } else {
            var result = (k <= 1)
                ? k
                : fib(k - 1) + fib(k - 2);
            mem[k] = result;
            return result;
        }
    }
    return fib(n);
}
```

Permutations & Combinations

```
function permutations(s) {
    if (is_empty_list(s)) {
        return list([]);
    } else {
        return accumulate(append, [],
            map(function(x) {
                return map(
                    function(p) {
                        return pair(x, p);
                    },
                    permutations(remove(x,s))
                );
            }, s)
        );
    }
}

function powerset(set) {
    if (is_empty_list(set)) {
        return list([]);
    } else {
        var rest_powerset = powerset(tail(set));
        var x = head(set);
        var has_x = map(function(s) {
            return pair(x, s);
            }, rest_powerset
        );
        return append(rest_powerset, has_x);
    }
}
```

Knapsap-like Problems

```
function coin_change(amount, coin_range) {
    if (amount === 0) {
        return 1;
    } else if (amount < 0 || coin_range === 0) {
        return 0;
    } else {
        return coin_change(amount, coin_range - 1)
            + coin_change(
                amount - highest_value(coin_range),
                coin_range
            );
    }
}
```

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.

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

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

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

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

function smallest(xs) {
    function sm(x, ys) {
        return (is_empty_list(ys))
            ? x
            : (x < head(ys))
                ? sm(x, tail(ys))
                : sm(head(ys), tail(ys));
    }
    return sm(head(xs), tail(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.

```
// QUICKSORT
function quicksort(xs) {
    if (is_empty_list(xs)) {
        return [];
    } else if (length(xs) === 1) {
        return xs;
    } else {
        var partitioned_xs =
            partition(tail(xs), head(xs));
        return append(
            quicksort(head(partitioned_xs)),
            pair(head(xs),
            quicksort(tail(partitioned_xs)))
        );
    }
}

function partition(xs, p) {
    function helper(elem, out_pair) {
        if (elem <= p) {
            return pair(
                append(head(out_pair), list(elem)),
                tail(out_pair)
            );
        } else {
            return pair(
                head(out_pair),
                append(tail(out_pair), list(elem))
            );
        }
    }
    return accumulate(helper, pair([], []), xs);
}
```

Bubble sort repeated swaps adjacent elements if they are in the wrong order, until the whole list (or array) is in the right order.

```
function sort(b) {
    var length_b = array_length(b);
    for (var num_sorted = 0;
        num_sorted < length_b;
        num_sorted = num_sorted + 1
    ) {
        for (var index = 0;
            index < length_b - 1;
            index = index + 1
        ) {
            if (b[index] > b[index+1]) {
                swap(index, index+1, b);
            } else { }
        }
    }
    return undefined;
}

function swap(left_index, right_index, array) {
    var tmp = array[left_index];
    array[left_index] = array[right_index];
    array[right_index] = tmp;
}
```

Lists

- A list of certain data type is null or a pair whose head is of that certain data type and whose tail is a list of that data type.
- A list of numbers is null or a pair whose head is a number and whose tail is a list of numbers.

Trees

- A tree of certain data items is a list whose items are said data items or trees of such data items.
- A tree of numbers is a list whose items are numbers, or trees of numbers.

Binary Search tree

- A binary tree of a certain type is the empty list 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.
    - The 1st element is the value of the binary tree. (Entry)
    - The 2nd element is the left subtree. (Left branch whose tree values are smaller than Entry)
    - The 3rd element is the right subtree of the binary tree. (Right branch whose tree values are larger than Entry)

Important Functions:

Useful Identities

pair(__, null); = list(__);
pair(__, list(…)); = list(__, …)

Extra Info

*Use append to put item at the back during recursion
*Use pair to put item in the front during recursion

- Equality do not work on function as each function is treated as unique value
- Should use "= = =" to compare primitive data type only (Boolean, Number, String, Undefined)

map(x => append(list(a), x), xs) - where "x" refers to each element of the list xs
accumulate((x, y) => x + y, null, xs) - where "x" refers to each element of the list xs, "y" is accumulated values
filter(x=> x % 2, xs) – where "x" refers to each element of the list xs

Accumulate folds from right to left*

```
const xs = list(x1, x2, …,xn);
accumulate(op, i, lst);
op(x1, op(x2, op(… op(xn, i)…)))
```

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

member(4, list(1,2,3,4,5,6)) -> list(4,5,6)
build_list(5, x => x) -> list(0,1,2,3,4)

Memoization (Streams)

```
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;
}

const onesB = pair(1, memo_fun(() => ms("B", onesB)));
```

Memoization (Normal use, 2D)

```
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] = [];
  } else {}
  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;
  }
}
```

Important Array Codes

```
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 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);
  }
}
```

```
//Make sure array a is sorted in ascending order
function binary_search(a, v) {
  function search(low, high) {
    if (low === high) {
      return a[low] === v;
    } else {
      const mid = math_floor((low + high) / 2);
      if (v === a[mid]) {
        return true;
      } else if (v < a[mid]) {
        return search(low, mid - 1);
      } else {
        return search(mid + 1, high);
      }
    }
  }
  return search(0, array_length(a) - 1);
}
```

```
function matrix_multiply_3x3(A, B) {
  const M = [];
    for (let r = 0; r < 3; r = r + 1) {
      M[r] = [];
      for (let c = 0; c < 3; c = c + 1) {
        M[r][c] = 0;
        for (let k = 0; k < 3; k = k + 1) {
          M[r][c] = M[r][c] + A[r][k] * B[k][c];
        }
      }
    }
  return M;
}
```

```
function transpose_matrix(arr) {
  const len = array_length(arr);
  for(let i = 0; i < len; i = i+1) {
    for(let j = i; j < len ; j = j+1) {
      let temp = arr[i][j];
      arr[i][j] = arr[j][i];
      arr[j][i] = temp;
    }
  }
  return arr;
}
```

```
function stream_to_array(s, n){
  let arr = [];
  function helper(strm, k) {
    if(k === n){
      arr[k] = 0;
      return arr;
    }
    else {
      arr[k] = head(strm);
      return helper(stream_tail(strm), k + 1);
    }
  }
  return helper(s, 0);
}
```

```
function array_to_stream(arr) {
  const len = array_length(arr);
  function helper(index) {
    if(index >= len) {
      return null;
    }
    else {
      return pair(arr[index], () => helper(index+1));
    }
  }
  return helper(0);
}
```

Mutable Lists

```
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_append(xs, ys) {
  if(is_null(xs)) {
    return ys;
  }
  else {
    set_tail(xs,append(tail(xs),ys));
    return xs;
  }
}
```

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

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

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

Important List Codes

```
function subsets(s) {
  if (is_null(s)) {
    return list(null);
  } else {
    const s1 = subsets(tail(s));
    return append(s1,
               map(ss => pair(head(s), ss), s1));
  }
}
```

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

Sorting (Lists)

```
function insertion_sort(xs) {
    if(is_null(xs)) {
        return xs;
    }
    else {
        return insert(head(xs), insertion_sort(tail(xs))) ;
    }
}

function insert(x, xs) {
    if (is_null(xs)) {
        return list(x);
    }
    else if(x <= head(xs)){
        return pair(x, xs);
    }
    else {
        return pair(head(xs), insert(x, 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))) ;
    }
}

function smallest(xs) {
    function sm(x, ys) {
        if(is_null(ys)) {
            return x;
        }
        else if(x < head(ys)) {
            return sm(x, tail(ys));
        }
        else {
            return sm(head(ys), tail(ys)) ;
        }
    }
    return sm(head(xs), tail(xs)) ;
}
```

```
function bubblesort(L) {
    const len = length(L);
    for (let i = len - 1; i >= 1; i = i - 1) {
        let p = L;
        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);
            }
            else { }
            p = tail(p);
        }
    }
    return L;
}
```

```
function merge_sort(xs) {
    if (is_null(xs) || is_null(tail(xs))) {
        return xs;
    }
    else {
        const mid = middle(length(xs)) ;
        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) ;
        if(x < y) {
            return pair(x, merge(tail(xs), ys));
        }
        else {
            return pair(y, merge(xs, tail(ys)));
        }
    }
}

function middle(n) {
    return math_floor(n / 2) ;
}

function take(xs, n) {
    if(n === 0) {
        return null;
    }
    else {
        return pair(head(xs), take(tail(xs), n - 1));
    }
}

function drop(xs, n) {
    if(n === 0) {
        return xs;
    }
    else {
        return drop(tail(xs), n - 1);
    }
}
```

```
function quicksort(xs) {
    if (is_null(xs)) {
        return null;
    }
    else if (is_null(tail(xs))) {
        return xs ;
    }
    else {
    const left_and_right = partition(tail(xs), head(xs));

    return append(quicksort(head(left_and_right)), pair(head(xs),
                    quicksort(tail(left_and_right)))) ;
    }
}

function partition(xs, p) {
    return pair(filter( x=> x <= p, xs),
    filter(x => x > p, xs));
}
```

Sorting (Arrays)

```
function insertion_sort(A) {
    const len = array_length(A);
    for(let i = 1; i < len; i = i + 1){ //from 1 to n-1
        const value = A[i];
        let hole = i;
        while(hole > 0 && A[hole - 1] > value){
            A[hole] = A[hole - 1];
            hole = hole -1;
        }
        A[hole] = value;
    }
}
```

```
function selectionsort(A) {

    const len = array_length(A);

    for(let i = 0; i < len - 1 ; i = i + 1) { // n-2 times
        let minIndex = i;
        for(let j = i + 1; j < len; j = j + 1){ // n-1 times

            if(A[j] < A[minIndex]){
              minIndex = j;
            }

            else{} // dont need to swap
        }
        const temp = A[i];
        A[i] = A[minIndex];
        A[minIndex] = temp;
    }
}
```

```
function bubblesort (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 { }
        }
    }
}
```

```
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);
    } else {}
}

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(arr, low, high)
{
    if (low < high)
    {
        let pi = partition(arr, low, high);
        quickSort(arr, low, pi-1);
        quickSort(arr, pi+1, high);
    }
    else{}
    return arr;
}


function partition(arr, low, high)
{
    let pivot = arr[high];
    let i = (low - 1);

    for (let j = low; j < high; j = j + 1)
    {
        if (arr[j] < pivot)
        {
            i = i + 1;
            let temp = arr[i];
            arr[i] = arr[j];
            arr[j] = temp;
        }
        else{}
    }

    let temp1 = arr[i+1];
    arr[i+1] = arr[high];
    arr[high] = temp1;
    return i+1;
}

quickSort(arr, 0, length of arr - 1);
```

Orders of Growth

	Approach	Best Case	Average Case	Worst Case
Insertion Sort	*m* = *n* - 1	O(n) sorted increasing list	O(n ^ 2)	O(n ^ 2)
Selection Sort	*m* = *n* - *1*	O(n ^ 2)	O(n ^ 2)	O(n ^ 2)
Merge Sort	*m* = *n/2*	O(n log n)	O(n log n)	O(n log n)
Quicksort	*m* = *n/2*	O(n log n)	O(n log n)	O(n ^ 2) sorted list

Ignore constants. Ignore lower order terms. For a sum, take the larger term.
For a product, multiply the two terms.
Orders of growth are concerned with how the effort scales up as the size of the problem increases, rather than an exact measure of the cost.