LISTS

```
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))));
}
```

ACCUM_CPS

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

function accumulate_cps(f, initial, xs, c) {
    return is_null(xs)
           ? c(initial)
           : accumulate_cps(f, initial, tail(xs),
               wish => c(f(head(xs), wish)));
}
```

BST

```
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)
}
```

Insertion sort takes elements from left to right, andinserts 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)));
}
```

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));
    }
}
```

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));
}
```

TREES

```
function append_iter(xs, ys) {
    return is_null(xs)
           ? ys
           : append_iter(tail(xs), pair(head(xs), ys));
}
```

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

```
function tree_sum(tree) {
return is_null(tree)
? 0
: is_list(head(tree))
  ? tree_sum(head(tree)) + tree_sum(tail(tree))
  : head(tree) + tree_sum(tail(tree));
}
```

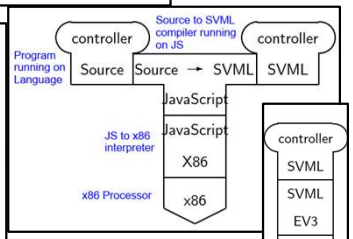
Accum tree
Where f is the map and op is the operator (sum, list, etc)

```
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 tree_flatten(xs) {
    return accumulate(
        (curr, rest) =>
            is_list(curr)
              ? append(tree_flatten(curr), rest)
              : pair(curr, rest),
        null,
        xs
    );
}
```

```
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);
}
```

```
function tree_filter(f, tree) {
    if (is_null(xs)) {
        return null;
    } else {
        return accumulate(
            (curr, rest) =>
                is_list(curr)
                    ? pair(tree_filter(f, curr), rest)
                    : f(curr)
                    ? pair(curr, rest)
                    : rest,
            null,
            tree
        );
    }
}
```



P&C,POWETSET

```
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 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 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);
    }
}
```

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)));
    }
}
```

D_MAP/APP

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

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

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

```
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);
}
```

```
function two_d_memoize(f) {
    const mem = [];
    function read(x, y) {
        return (mem[x] === undefined) ?
            undefined : mem[x][y];
    }
    function write(x, y, value) {
        if (mem[x] === undefined) {
            mem[x] = [];
        mem[x][y] = value;
    }
    function mf(x, y) {
        const mem_xy = read(x, y);
        if (mem_xy !== undefined) {
            return mem_xy;
        } else {
            const result = f(x, y);
            write(x, y, result);
            return result;
        }
    }
    return mf;
}
const perms01memo =
two_d_memoize((n, m) => {
    if (n === 0 && m === 0) {
        return list(null);
    } else {
        const p0 = (n > 0)
            ? map(p => pair(0, p), perms01memo(n - 1, m))
            : null;
        const p1 = (m > 0)
            ? map(p => pair(1, p), perms01memo(n, m - 1))
            : null;
        return append(p0, p1);
    }
});
```

For loop	Order of growth
for (i = 0; i < n; i = i + 1)	O(n)
for (i = 0; i < n; i = i + 2)	O(n)
for (i = n; i > 1; i = i – 1)	O(n)
for (i = 1; i < n; i = i *2)	O(logn)
for (i = n; i > 1; i = i / 2)	O(logn)
for (i = 0; i*i < n; i = i +1)	$O(\sqrt{n})$
for(i = 1; p < n; i = i + 1) {p = p + 1}	$O(\sqrt{n})$
for(i = 1; i < a*n; i = i + b)	O(an/b)

Recurrence Relation	Time Complexity
T(n) = T(n-1) + O(1)	O(n)
T(n) = T(n-1) + O(n)	$O(n^2)$
T(n) = T(n–1) + O(logn)	O(nlogn)
T(n) = 2T(n-1) + O(1)	$O(2^n)$
T(n) = T(n / 2) + O(n)	O(n)
T(n) = T(n / 2) + O(1)	O(logn)
T(n) = 2T(n / 2) + O(n)	O(nlogn)

Generally

$T(n) = T(n-1) + O(n^k)$ → $T(n) = O(n^{k+1})$

$T(n) = 2T(\frac{n}{2}) + O(n)$ → $T(n) = O(n \log n)$ merge sort.

$T(n) = T(\frac{n}{2}) + O(n)$ → $T(n) = O(n)$ not intuitive

$T(n) = 2T(\frac{n}{2}) + O(1)$ → $T(n) = O(n)$ not intuitive

$T(n) = 2T(n-1) + O(1)$ → $T(n) = O(2^n)$

STREAMS

```
function partial_sums(ss) {
    return pair(head(ss), () =>
add_streams(stream_tail(ss), partial_sums(ss)));
}
```

thrice(f) = fff

thrice(thrice) = thrice(thrice(thrice) = thrice(f9) = f27

```
function fibgen(a, b) {
    return pair(a, () => fibgen(b, a + b));
}
```

```
function replace(s, a, b) {
    return is_null(s)
        ? null
        : pair((head(s) === a) ? b : head(s),
            () => replace(stream_tail(s), a, b));
}
```

```
function sieve(s) {
    return pair(head(s),
        () => sieve(stream_filter(
                        x => !is_divisible(x, head(s)),
                        stream_tail(s))));
}

const primes = sieve(integers_from(2));
```

ARRAY SORT

```
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 insertion_sort(A) {
    const len = array_length(A);

    for (let i = 1; i < len; i = i + 1) {
        let j = i - 1;
        while (j >= 0 && A[j] > A[j + 1]) {
            swap(A, j, j + 1);
            j = j - 1;
        }
    }
}
function swap(A, x, y) {
    const temp = A[x];
    A[x] = A[y];
    A[y] = temp;
}
```

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

```
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;
}
function ms(m, s) {
    display(m);
    return s;
}
const onesB = pair(1,
  memo_fun(() => ms("B", onesB)));
```

```
function fstreams(f,s1,s2){
    if (is_null(s1)) { return s2; }
    else if (is_null(s2)) { return s1; }
    else { return
        pair( f(head(s1),head(s2)),
          () => fstreams(f,
              stream_tail(s1), stream_tail(s2)));
    }
}
```

MISC

```
function repeat(f, n) {
return x => n === 0
    ? x
    : f(repeat(f, n - 1)(x));
}
```

```
unction is_hula_hoop(x) {
  let pairs = null;
  function check(y) {
    if (is_pair(y)) {
      if (! is_null(member(y, pairs))) {
        return true;
      } else {
        pairs = pair(y, pairs);
        return check(head(y)) && check(tail(y));
      }
    } else {
      return false;
    }
  }
  return check(x);
}
```

```
function merge_k_sorted_arrays(arr) {
    function merge_k_helper(arr, i, j) {
        if (i === j) {
            //if only 1 array, add to output
            return arr[i];
        }
        if (j - i === 1) {
            //if 2 arrays, merge them
            return merge_two_arrays(arr[i], arr[j]);
        }
        const out1 = merge_k_helper(arr, i, math_floor((i + j) / 2)); //i
        const out2 = merge_k_helper(arr, math_floor((i + j) / 2) + 1, j);
        return merge_two_arrays(out1, out2); // Merge the output array


turn merge_k_helper(arr, 0, array_length(arr) - 1);
```

```
function matrix_multiply(A, B) {
    const r1 = array_length(A);
    const c1 = array_length(A[0]);
    const r2 = array_length(B);
    const c2 = array_length(B[0]);
    if (c1 === r2) {
        const M = [];
        for (let r = 0; r < r1; r = r + 1) {
            M[r] = [];
            for (let c = 0; c < c2; c = c + 1) {
                M[r][c] = 0;
                for (let k = 0; k < c1; k = k + 1) {
                    M[r][c] = M[r][c] + A[r][k] * B[k][c];
                }
            }
        }
        return M;
    }
}
```