Mutable Data

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

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

function d_reverse(xs) {
  if (is_null(xs)) {
    return xs;
  } else if (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 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);
  }
}

function swap(A, i, j) {
  let temp = A[i];
  A[i] = A[j];
  A[j] = temp;
}

function count_pairs(x) {
  let pairs = null;
  function check(y) {
    if (is_pair(y) && is_null(member(y, pairs))) {
      pairs = pair(y, pairs);
      check(head(y));
      check(tail(y));
    }
  }
  check(x);
  return length(pairs);
}
```

Loops & Arrays

```
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 rotate_matrix(M) {
  const n = array_length(M);
  function swap(r1, c1, r2, c2) {
    const temp = M[r1][c1];
    M[r1][c1] = M[r2][c2];
    M[r2][c2] = temp;
  }
  // Do a matrix transpose first.
  for (let r = 0; r < n; r = r + 1) {
    for (let c = r + 1; c < n; c = c + 1) {
      swap(r, c, c, r);
    }
  }
  // Then reverse each row.
  const half_n = math_floor(n / 2);
  for (let r = 0; r < n; r = r + 1) {
    for (let c = 0; c < half_n; c = c + 1) {
      swap(r, c, r, n - c - 1);
    }
  }
}
```

Searching & Sorting

```
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);
}
```

Recursive:

```
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);
}
```

Loop:

```
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 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) {
    let j = i - 1;
    while (j >= 0 && A[j] > A[j + 1]) {
      swap(A, j, j + 1);
      j = j - 1;
    }
  }
}

function insertion_sort2(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]; // shift right
      j = j - 1;
    }
    A[j + 1] = x;
  }
}

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 bubblesort_array(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;
      }
    }
  }
}
```

Memoization

```
const mem = [];
function mtrib(n) {
  if (mem[n] !== undefined) {
    return mem[n];
  } else {
    const result =
      n === 0 ? 0
      : n === 1 ? 1
      : n === 2 ? 1
      : mtrib(n-1) + mtrib(n-2) + mtrib(n-3);
    mem[n] = result;
    return result;
  }
}

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 mtrib =
  memoize(n => n === 0 ? 0
          : n === 1 ? 1
          : n === 2 ? 1
          : mtrib(n - 1) + mtrib(n - 2) + mtrib(n - 3));

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;
  }
}            // O(nk) space & time
```

Streams

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

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

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)));

function partial_sums(s) {
  function helper(acc, stream) {
    return pair(head(stream) + acc, () =>
      helper(acc + head(stream), stream_tail(stream)));
  }
  return helper(0, s);
}

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

function stream_pairs3(s) {
  return (is_null(s) || is_null(stream_tail(s)))
    ? null
    : pair(pair(head(s), head(stream_tail(s))),
        () => interleave_stream_append(
              stream_map(x => pair(head(s), x), stream_tail(stream_tail(s))),
              stream_pairs3(stream_tail(s))));
}
```

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),
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);
    }
}
```

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)));
    }
}
```