Mutable Functions

d_reverse

```
function d_reverse(xs) {
  if (is_null(xs)) {
    return xs;
  } else if (is_null(tail(xs))) {
    return xs;
  } else {
    let temp = d_reverse(tail(xs));
    set_tail(tail(xs), xs);
    set_tail(xs, null);
    return temp; }}
```

d_append

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

d_filter

```
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)); }}
```

d_map

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

Bubble Sort (List)

```
function bubblesort_list(L) {
  const len = length(L);
  for (let i = len - 1; i >= 1; i = i - 1) {
    let k = L;
    for (let j = 0; j < i; j = j + 1) {
      const first = head(k);
      const sec = head(tail(k));
      if (first > sec) {
        set_head(tail(k), first);
        set_head(k, sec);
      } else {}
      k = tail(k); }}
  return L; }
```

Array Processing

Flatten Array

```
function flatten_array(arr) {
  let output = [];
  function append_to_output(arr) {
    for (let k = 0; k < array_length(arr); k = k + 1)
{
      if (is_array(arr[k])){
        append_to_output(arr[k]);
      } else {
        output[array_length(output)] = arr[k];
}}
return output;}
return append_to_output(arr); }
```

Append

```
function append_array(lhs, rhs) {
  const result = [];
  for (let i = 0; i < array_length(lhs); i = i + 1) {
    result[array_length(result)] = lhs[i]; }
  for (let i = 0; i < array_length(rhs); i = i + 1) {
    result[array_length(result)] = rhs[i]; }
   return result;}
```

Reverse

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

Map

```
function map_array(f, arr) {
  const len = array_length(arr);
  function iter(i) {
    if (i < len) {
      arr[i] = f(arr[i]);
      iter(i + 1);
    } else {}}
  iter(0);}
```

Binary Search

```
function binary_search_loop(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);}
```

Selection Sort

```
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;
    } else{} }
  return min_pos; }

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);}}
```

Insertion Sort

```
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;}}}
```

Merge 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);
  } else{ }}

function merge(A, low, mid, high) {
  const B = []; // temporary array
  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 mergeC(xs, xs_len, ys, ys_len) {
  let result = [];
  let result_len = xs_len + ys_len;
  let xi = 0;
  let yi = 0;

  for (let i = 0; i < result_len; i = i + 1) {
          if (xi === xs_len) {
                  result[i] = ys[yi];
                  yi = yi + 1;
          } else if ( yi === ys_len) {
      result[i] = xs[xi];
      xi = xi + 1;
          } else if (xs[xi] <= ys[yi]) {
      result[i] = xs[xi];
      xi = xi + 1;
    } else {
      result[i] = ys[yi];
      yi = yi + 1; }}
return result; }
```

Stream Processing

Map

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

function stream_map_optimized(f, s) {
  return is_null(s)
    ? null
        : pair(f(head(s)),
      memo_fun( () =>
stream_map_optimized(f, stream_tail(s))));}
```

Filter

```
function stream_filter(p, s) {
  return is_null(s)
    ? null : p(head(s))
        ? pair(head(s),() => stream_filter(p,
stream_tail(s)))
        : stream_filter(p, stream_tail(s));}
```

Append

```
function stream_append_pickle(xs, ys) {
  return is_null(xs)
    ? ys()
    : pair(head(xs), () =>
stream_append_pickle(stream_tail(xs), ys));}
```

Add Streams

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

Multiply Streams

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

Scale Streams

```
function scale_stream(factor, stream) {
   return stream_map(x => x * factor, stream); }
```

merge_list_of_streams(list(s1,s2,s3)) returns the stream 1, 2, 3, 11, 22, 33,...

```
function merge_n(ss) {
  return pair(head(head(ss)),
        () => merge_n(append(tail(ss),
          list(stream_tail(head(ss))))); }
```

More

```
function more(a, b) {
  return (a > b)
    ? more(1, 1 + b)
     : pair(a, () => more(a + 1, b)); }
eval_stream(more(1,2), 15);// [1, [1, [2, [1, [2, [3,
[1, [2, [3, [4, [1, [2, [3, [4, [5, [][][][][][][][][][]]]]]]]]]]]]]]]]]
```

Partial Sums

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

Memoized 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;}

function ms(m, s) {
  display(m);
  return s;}

function m_integers_from(n) {
   return pair(n, memo_fun(() => ms("M: " +
stringify(n), m_integers_from(n + 1))));}
```

Memoized Coin Change

```
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 mcc(n, k) {
  if (n === 0){
    return 1;
  } else if (n < 0 || k === 0){
    return 0;
  } else if (read(n,k) !== undefined){
    return read(n,k);
  } else {
    const value = mcc(n, k - 1) + mcc(n -
first_denomination(k),k);
    write(n,k,value);
    return value;}}
```

Rotate Matrix

```
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; }
// Transpose Matrix
   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); }}}
```

Order of Growth

① Loop → if WD inside the loop is constant, for(let i = 0; i < N; i = i + 1) → Θn

② Nested Loop → Θn^2

③ Sequential → add up & give the highest pow

④ If Else → only give the block with the highest pow