**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))));

}

Text, letter

Description automatically generatedText, letter

Description automatically generated

Text, letter

Description automatically generated

Text, letter

Description automatically generated

Text, letter

Description automatically generatedText, letter

Description automatically generatedText

Description automatically generated