Iterative vs Recursive Processes

By: Gan Chin Yao | 2017 Dec

A function that calls itself gives rise to an iterative process if the recursive call is always the last operation that the function carries out. This means that the result of the recursive call is also the result of the function.

Can have both
A recursive function that does iterative processes
A recursive function that does recursive processes

```
function f(x, y) {
    if (x === 0) {
        return y;
    } else {
        return f(x - 1, y + 1);
    }
}
```

Iterative return the final value in the last call, so you will return the final result in the base case, instead of recursion which return a smaller value for the sub-case and then re-calculate all the way back up again

The body of function `f` has one recursive call. The result of the recursive call is the result of the function. This means that the function gives rise to an iterative process.

```
function g(x) {
    if (x === 0) {
        return 0;
    } else {
        return 1 + g(x - 1);
    }
}
```

The body of function `g` has one recursive call. The result of the recursive call is not the result of the function. When the recursive call returns, its result needs to be added to 1. This operation is called a "deferred operation". The presence of such a deferred operation means that the corresponding process is a recursive process.

We say the function gives rise to a recursive process.

Deferred ope
Need to wait

Definition of trees

A tree of data items is a list, whose elements are data items or trees.

Definition of list

A list is either empty, or a pair whose tail is a list.

Fast Power Function: Correctness

```
// raise b to the power of non-neg integer e
function fast_power(b, e) {
    if (e === 0) {
        return 1;
    } else if (is_even(e)) {
        return fast_power(b * b, e / 2);
    } else {
        return b * fast_power(b, e - 1);
    }
}
```

Equality in The Source: `equal`

Specification

- Booleans, strings, functions, numbers: `equal` same as `===`
- Any two empty lists are `equal`
- Two pairs are `equal` iff their heads and tails are `equal`.

```
function equal(x, y) {
  return (is_empty_list(x) && is_empty_list(y)) ||
         (is_pair(x) && is_pair(y) &&
          equal(head(x), head(y)) &&
          equal(tail(x), tail(y)))              ||
         x === y;
}
```

Scaling a list

Let us scale all elements of a list by a factor `f`.

```
function scale_list(items, factor) {
    return is_empty_list(items)
           ? []
           : pair(factor * head(items),
                  scale_list(tail(items),
                             factor)
                  );
}
```

A2: Selection Sort in Source

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

A2: Selection Sort (function `smallest`)

```
// find smallest element of a non-empty list 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));
}
```

B1: Merge Sort

```
function merge_sort(xs) {
    if (is_empty_list(xs) ||
        is_empty_list(tail(xs))) {
        return xs;
    } else {
        var mid = middle(length(xs));
        return merge(merge_sort(take(xs, mid)),
                     merge_sort(drop(xs, mid)));
    }
}
```

B1: Merge Sort (function `merge`)

```
function merge(xs, ys) {
    if (is_empty_list(xs)) {
        return ys;
    } else if (is_empty_list(ys)) {
        return xs;
    } else {
        var x = head(xs);
        var y = head(ys);
        return (x < y)
               ? pair(x, merge(tail(xs), ys))
               : pair(y, merge(xs, tail(ys)));
    }
}
```

Helper functions for Merge Sort

```
// take half, rounded down
function middle(n) {
    // homework
}
// put the first n elements of xs into a list
function take(xs, n) {
    // homework
}
// drop first n elements from list, return rest
function drop(xs, n) {
    // homework
}
```

```
function merge_sort(A) {
    var H = [];
    merge_sort_with_helper(A, 0, array_length(A),H);
}

function merge_sort_with_helper(A, low, high, H) {
    if (low < high) {
        var middle = math_floor((low + high) / 2);
        merge_sort_with_helper(A, low, middle, H);
        merge_sort_with_helper(A, middle + 1, high, H);
        merge_with_helper(A, low, middle, high, H);
    } else {}
}
```

```
function merge_with_helper(A, low, middle, high, H) {
    for (var i = low; i <= high; i = i + 1) {
        H[i] = A[i];
    }

    var i = low;
    var j = middle + 1;
    var k = low;

    while (i <= middle && j <= high) {
        if (H[i] <= H[j]) {
            A[k] = H[i];
            i = i + 1;
        } else {
            A[k] = H[j];
            j = j + 1;
        }
        k = k + 1;
    }

    while (i <= middle) {
        A[k] = H[i];
        k = k + 1;
        i = i + 1;
    }
}
```

```
function insertion_sort(A) {
    var len = array_length(A);
    var i = 1;
    while (i < len) {
        var x = A[i];
        var j = i - 1;
        while (j >= 0 && A[j] > x) {
            A[j + 1] = A[j];
            j = j - 1;
        }
        A[j+1] = x;
        i = i + 1;
    }
}
```

```
function heapSort(arr){
  var len = arr.length,
      end = len-1;

  heapify(arr, len);

  while(end > 0){
   swap(arr, end--, 0);
   siftDown(arr, 0, end);
  }
  return arr;
}
function heapify(arr, len){
   // break the array into root + two sides, to create tree (heap)
   var mid = Math.floor((len-2)/2);
   while(mid >= 0){
    siftDown(arr, mid--, len-1);
  }
}

function siftDown(arr, start, end){
   var root = start,
       child = root*2 + 1,
       toSwap = root;
   while(child <= end){
      if(arr[toSwap] < arr[child]){
        swap(arr, toSwap, child);
      }
      if(child+1 <= end && arr[toSwap] < arr[child+1]){
        swap(arr, toSwap, child+1)
      }
      if(toSwap != root){
         swap(arr, root, toSwap);
         root = toSwap;
      }
      else{
         return;
      }
      toSwap = root;
      child = root*2+1
  }
}

function swap(arr, i, j){
  var temp = arr[i];
  arr[i] = arr[j];
  arr[j] = temp;
}
```

```
function quickSort(arr, left, right){
   var len = arr.length,
   pivot,
   partitionIndex;


  if(left < right){
    pivot = right;
    partitionIndex = partition(arr, pivot, left, right);

   //sort left and right
   quickSort(arr, left, partitionIndex - 1);
   quickSort(arr, partitionIndex + 1, right);
  }
  return arr;
}

function partition(arr, pivot, left, right){
   var pivotValue = arr[pivot],
       partitionIndex = left;

   for(var i = left; i < right; i++){
    if(arr[i] < pivotValue){
      swap(arr, i, partitionIndex);
      partitionIndex++;
    }
  }
  swap(arr, right, partitionIndex);
  return partitionIndex;
}

function swap(arr, i, j){
  var temp = arr[i];
  arr[i] = arr[j];
  arr[j] = temp;
}
```

Squaring a list

Let us *square* all elements of a list.

```
function square_list(items) {
    return is_empty_list(items)
           ? []
           : pair(square(head(items)),
                  square_list(tail(items))
                  );
}
```

Scaling a list

Let us scale all elements of a list by a factor `f`.

```
function scale_list(items, factor) {
    return is_empty_list(items)
           ? []
           : pair(factor * head(items),
                  scale_list(tail(items),
                             factor)
                  );
}
```

Binary Trees

Definition

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

More definitions

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.

Definition of `accumulate`

```
function accumulate(op,initial,sequence) {
    if (is_empty_list(sequence)) {
        return initial;
    } else {
        return op(head(sequence),
                  accumulate(op,initial,
                             tail(sequence)));
    }
}
```

Permutations

```
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));
  }
}
```

Linear search: The `find` function

Specification

For a given value `v` and list `xs` return true iff `v` occurs in `xs`

```
function find(v, xs){
    return (is_empty_list(xs)) ? false
           : (v === head(xs)) ? true
             : find(v, tail(xs));
}
```

```
function selection_sort(A) {
    var len = array_length(A);
    for (var i = 0; i < len - 1; i = i + 1) {
        var j_min = i;
        for (var j = i + 1; j < len; j = j + 1) {
            if (A[j] < A[j_min]) {
                j_min = j;
            } else {}
        }
        if (j_min !== i) {
            swap(A, i, j_min);
        } else {}
    }
}

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

- `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 box-and-pointer 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 one-argument 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 r_1, then to the second-last element and r_1, resulting in r_2, etc, and finally to the first element and r_{n-1}, 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.

Miscellaneous Functions

- `is_number(x)`: Returns `true` if `x` is a number, and `false` otherwise.
- `is_boolean(x)`: Returns `true` if `x` is `true` or `false`, and `false` otherwise.
- `is_string(x)`: Returns `true` if `x` is a string, and `false` otherwise.
- `is_function(x)`: Returns `true` if `x` is a function, and `false` otherwise.
- `is_object(x)`: Returns `true` if `x` is an object, and `false` otherwise. Following JavaScript, arrays are considered objects.
- `is_array(x)`: Returns `true` if `x` is an array, and `false` otherwise. The empty array `[]`, also known as the empty list, is an array.
- `equal(x, y)`: Returns `true` if `x` and `y` have the same structure (using pairs and `[]`), and corresponding leaves are `===`, and `false` otherwise.
- `array_length(x)`: Returns the current length of array `x`, which is 1 plus the highest index `i` that has been used so far in an array assignment on `x`.
- `parse(x)`: returns the parse tree that results from parsing the string `x` as a Source program.
- `JSON.stringify(x)`: returns a string that represents the given JSON object `x`.
- `apply_in_underlying_javascript(f, xs)`: calls the function `f` with arguments `xs`. For example:

```
function times(x, y) {
  return x * y;
}
apply_in_underlying_javascript(times, list(2, 3)); // returns 6
```

Algorithm	Time Complexity			Space Complexity
	Best	Average	Worst	Worst
Quicksort	Ω(n log(n))	θ(n log(n))	O(n^2)	O(log(n))
Mergesort	Ω(n log(n))	θ(n log(n))	O(n log(n))	O(n)
Timsort	Ω(n)	θ(n log(n))	O(n log(n))	O(n)
Heapsort	Ω(n log(n))	θ(n log(n))	O(n log(n))	O(1)
Bubble Sort	Ω(n)	θ(n^2)	O(n^2)	O(1)
Insertion Sort	Ω(n)	θ(n^2)	O(n^2)	O(1)
Selection Sort	Ω(n^2)	θ(n^2)	O(n^2)	O(1)
Tree Sort	Ω(n log(n))	θ(n log(n))	O(n^2)	O(n)
Shell Sort	Ω(n log(n))	θ(n(log(n))^2)	O(n(log(n))^2)	O(1)
Bucket Sort	Ω(n+k)	θ(n+k)	O(n^2)	O(n)
Radix Sort	Ω(nk)	θ(nk)	O(nk)	O(n+k)
Counting Sort	Ω(n+k)	θ(n+k)	O(n+k)	O(k)
Cubesort	Ω(n)	θ(n log(n))	O(n log(n))	O(n)

Random Int

```
MyClass.prototype.getRandomRangedInt = function(min, max) {
    return math_floor(math_random() * (max - min + 1)) + min;
};
```

Sort a number into an ascending list

```
// return a list with a number sorted into ascending lst
function helper(lst, number) {
        var mhead = filter(function(x) {
            return x < number;
        }, lst);
        var mtail = filter(function(x) {
            return x >= number;
        }, lst);
        return append(append(mhead, list(number)), mtail);
    }
e.g. helper(list(3,4,7,8) , 6); => list(3,4,6,7,8)
```

MAP ITERATIVE

```
// iterative alternative
function new_map(f, xs) {
    function helper(xs, acc) {
        if(is_empty_list(xs)) {
            return acc;
        } else {
            return helper(tail(xs),
                          pair(f(head(xs)),
                               acc));
        }
    }
    return reverse(helper(xs, []));
}
```

FILTER

```
function filter(pred, xs) {
    if(is_empty_list(xs)) {
        return [];
    } else {
        if(pred(head(x))) {
            return pair(head(xs),
                        filter(pred,
                               tail(xs)));
        } else {
            return filter(pred, tail(xs));
        }
    }
}
// filter with accumulate
function new_filter(pred, xs) {
    return accumulate(function(e, acc){
        return pred(e)
        ? pair(e, acc) : acc;
    }, [], xs);
}
```

ACCUMULATE

```
function accumulate(op, init, seq) {
    if(is_empty_list(seq)) {
        return init;
    } else {
        return op(head(seq),
                  accumulate(op, init,
                             tail(seq)));
    }
}

are_equal_sets

function are_equal_sets(set1, set2) {
    if(length(set1) === length(set2)) {
        return false;
    } else {
        return accumulate(function(e1,acc){
            return accumulate(
                function(e2,acc) {
                return e2 === e1
                || acc;
            }, false, set2);
        }, true, set1);
    }
}
```

PERMUATATIONS

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

N-PERMUATATIONS

```
function n_permutations(xs, n) {
    if(n === 0) {
        return list([]);
    } else {
        return accumulate(function(e, acc){
            return append(map(function(x) {
                return pair(e, x);
            }, n_permutations(remove(e,xs),
                              n - 1)),
            acc);
        }, [], xs);
    }
}
```

N-COMBINATIONS

```
function n_combinations(xs, n) {
    if (n === 0) {
        return list([]);
    } else if (is_empty_list(xs)) {
        return [];
    } else {

        return append(map(function(e) {
            return pair(head(xs), e);
        }, n_combinations(tail(xs), n-1)),
        n_combinations(tail(xs), n));
    }
}
```

TOWER OF HANOI

```
function make_room(n) {
    return enum_list(1,n);}
function make_house(r1,r2,r3) {
    return list(r1,r2,r3);}
function starting_house(n) { return
make_house(make_room(n),make_room(0),
           make_room(0));}

function make_move(s, t) {return pair(s, t);}
function get_source(m) {return head(m);}
function get_destination(m) {return tail(m);}

function move_kitty(m, h) {
    var s = get_source(m);
    var d = get_destination(m);
    var kitty = head(s);
    if(is_empty_list(d) || kitty < head(d)) {
        // stack operations
        // if source return tail
        // if dest pair with kitty
        // else leave house untouched
    } else {
        return h;
    }
}
// !!-starting from last move to the first
function apply_moves(moves, h) {
    return accumulate(move_kitty, h, moves);
}
// reverse order! 0(2^n)
function find_moves(house, num, r1, r2) {
    if(num === 1) {
        return list(make_move(r1, r2));
    } else {
        var other_r = 6 - r1 - r2;
        var f_half = find_moves(house,
            num-1,r1,other_r);
        var m = make_move(r1, r2);
        var s_half = find_moves(house,
            num-1,other_r,r2);
        return append(s_half,pair(m,f_half));
    }
}

function find_and_move(n) {
    var h = starting_house(n);
    var moves = find_moves(h, n, 1, 2);
    return apply_moves(moves, h);
}
```

COIN CHANGE

```
function makeup_amount(x, l) {
    if(is_pair(l)) {
        return append(map(function(x) {
            return pair(head(l), x);
            }, makeup_amount(x - head(l),
                             tail(l))),
           makeup_amount(x, tail(l)));
    } else if (x === 0) {
        // empty case, there is a solution
        return list([]);
    } else {
        // fail, return nothing
        return [];
    }
}
makeup_amount(22, list(1,10,5,20,1,5,1,50));
```

TREES

is_number_tree

```
function is_number_tree(xs) {
    if (!is_list(xs)) {
        return false;
    } else {
        return accumulate(function(e, acc){
            return acc
            && (is_list(e)
                ? is_number_tree(e)
                : is_number(e));
        }, true, xs);
    }
}
```

map_tree

```
function map_tree(f, xs) {
    return map(function(e) {
        return (is_list(e))
        ? map_tree(f, e) : f(e);
    }, xs);
}
```

accumulate_tree

```
function accumulate_tree(f, init, xs) {
    return accumulate(function(e, acc) {
        return (is_list(e))
        ? f(accumulate_tree(f, init, e),
            acc)
        : f(e, acc);
    }, init, xs);
}
```

STACK

```
function Stack() {
    this.data = [];
}
Stack.prototype.push = function(item) {
    this.data = pair(item, this.data);
};
Stack.prototype.pop = function() {
    if(is_empty_list(this.data)) {
        error("stack is empty" );
    } else {
        var result = head(this.data);
        this.data = tail(this.data);
        return result;
    }
};
Stack.prototype.length = function() {
    var context = this;
    return length(context.data);
};
```

QUEUE

```
var front_ptr = head;
var rear_ptr = tail;
var set_front_ptr = set_head;
var set_rear_ptr = set_tail;

function make_queue() {
    return pair([],[]);
}
function is_empty_queue(q) {
    is_empty_list(front_ptr(q));
}
function front_queue(q) {
    if(is_empty_queue(q)) {
        display("Error: Queue is empty" );
    } else {
        return head(front_ptr(p));
    }
}
function delete_queue(q) {
    if(is_empty_queue(q)) {
        display("Error: Queue is empty" );
    } else {
        set_front_ptr(q,
            tail(front_ptr(q)));
        return q;

    }
}
function insert_queue(q, item) {
    var new_pair = pair(item, []);
    if(is_empty_queue(q)) {
        set_front_ptr(q, new_pair);
        set_rear_ptr(q, new_pair);
    } else {
        set_tail(rear_prt(q), new_pair);
        set_rear_ptr(q, new_pair);
    }
    return q;
}
function dequeue(q) {
    var result = front_queue(q);
    delete_queue(q);
    return result;
}
```

MUTABLE LIST MODIFY LIST INDEX

```
function modify_list_index(xs,index,val2) {
    if(is_empty_list(xs)) {
        error("Out of bounds");
    } else if(index === 0) {
        set_head(xs, val2);
    } else {
        modify_list_index(tail(xs),
                          index - 1, val2);
    }
}
```

REMOVE ITEM FROM LIST

```
function remove_elem(xs, index) {
    function inner(prev, x, index) {
        if(is_empty_list(x)) {
            return error("out of bounds");
        } else {
            if(index === 0) {
                set_tail(prev, tail(x));
            } else {
                inner(x, tail(x),
                      index - 1);
            }
        }
    }
    var temp = pair(1, xs);
    inner(temp, xs, index);
    return tail(temp);
}
```

SWAP ADJACENT ELEMENT

```
function swap(xs) {
    var temp = tail(xs);
    set_tail(xs, tail(temp));
    set_tail(temp, xs);
    return temp;
}
```

MUTABLE REVERSE

```
function mutable_reverse(xs) {
    // VERSION 1
    if (is_empty_list(xs)) {
        return xs;
    } else if (is_empty_list(tail(xs))) {
        return xs;
    } else {
        var temp=mutable_reverse(tail(xs));
        set_tail(tail(xs), xs);
        set_tail(xs, []);
        return temp;
    }
     // VERSION 2 (found in DG 9)
     // note that xs will point to last
     // (previously first first elem)
     function helper(prev, xs) {
       if (is_empty_list(xs)) {
           return prev;
       } else {
          var rest = tail(xs);
          set_tail(xs, prev);
          return helper(xs, rest);
       }
     }
     return helper([], xs);
}
```

MERGE LIST

```
function mergeB(xs, ys) {
    if(is_empty_list(xs) &&
       is_empty_list(ys)) {
        return [];
    } else if(is_empty_list(xs)) {
        set_tail(ys, mergeB(xs, tail(ys)));
        return ys;
    } else if(is_empty_list(ys)) {
        set_tail(xs, mergeB(tail(xs), ys));
        return ys;
    } else if(head(xs) <= head(ys)) {
        set_tail(xs, mergeB(tail(xs), ys));
        return xs;
    } else {
        set_tail(ys, mergeB(xs, tail(ys)));
        return ys;
    }
}
```

STREAM_MAP

```
function stream_map(f, s) {
    return pair(f(head(s)),function() {
        return stream_map(f,stream_tail(s));
    });
}
```

RECURSIVELY DEFINED STREAM

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

var ones = pair(1,
                function() {return ones});
var integers = pair(1, function() {
    return add_streams(integers, ones);
    });
/* Visualization
ones:       1 1 1 1 1 1 ...
integers:   1 2 3 4 5 ...
-----------------------------
intergers:  1 2 3 4 5 6 ...
*/
```

DYNAMIC PROGRAMMING

is_subsequence

```
function is_subsequence(xs, ys) {
    if(is_empty_list(xs)) {
        return true;
    } else if(is_empty_list(ys)) {
        return false;
    } else if(head(xs) === head(ys)) {
        return is_subsequence(tail(xs),
                              tail(ys))
            || is_subsequence(xs,
                              tail(ys));
    } else {
        return is_subsequence(xs,
                              tail(ys));
    }
}
```

META-CIRCULAR EVALUAT

REDEFINING VARIABLES

```
function evaluate_var_definition(stmt,
                                 env) {
    if(has_binding_in_frame(
        var_definition_variable(stmt),
        first_fram(env))) {
        error("Redeclared variable");
    } else {
        define_variable(
            var_definition_variable(stmt),
            evaluate(
                var_definition_value(stmt),
                env),
            env);
        return undefined;
    }
}

// Old definition
function evaluate_var_definition(stmt,
                                 env) {
    define_variable(
        var_definition_variable(stmt),
        evaluate(var_definition_value(stmt)
                 env),
        env);
    return undefined;
}
```

REVERSE APPLICATION ORDER

```
function list_of_values(exps, env) {
    if (no_operands(exps)) {
        return [];
    } else {
        var abs = list_of_values(
                  rest_operands(exps), env);
        return pair(evaluate(
                first_operand(exps),
            env),
                    abc);
    }
}
```

THUNKING

- compute values only when needed
- when applying user defined function, each argument is placed in a "thunk"

```
function list_of_values(exps, env) {
    if(no_operands(exps)) {
        return [];
    } else {
        return pair(
            make_thunk(first_operand(exps),
                       env),
            list_of_values(rest_operands(exps),
                           env));
    }
}
function make_thunk(expr, env) {
    return { tag: "thunk",
             expression: expr,
             environment: env};
}
function force(v) {
    if(is_thunk(v)) {
        return force(
            evaluate(thunk_expression(v),
                     thunk_environment(v)));
    } else {
        return v;
    }
}
function lookup_variable_value(variable,
                               env){
    function env_loop(env) {
        if(is_empty_environment(env)) {
            error("Unbound variable");
        } else if(has_binding_in_frame(
            variable,first_frame(env))) {
            var val = force(
                first_frame(env)[variable]);
            first_frame(env)[variable] = val;
            return val;
        }
    }
```

POWER SET

```
function power_set(xs) {
    if (is_empty_list(xs)) {
        return list([]);
    } else {
        // Either you pick the number,
        // or you don't
        var without_head=power_set(tail(xs));
        var use_head = map(function(l) {
            return pair(head(xs),l);
        }, without_head);
        return append(use_head,without_head);
    }
}
```

COUNT PAIRS

```
function count_pairs(x) {
    var hist = [];
    function helper(x) {
        if(!is_pair(x)) {
            return 0;
        } else {
            if(is_empty_list(member(x, hist)))
                hist = pair(x, hist);
                return 1 + helper(head(x))
                         + helper(tail(x));
            } else {

                return 0;
            }
        }
    }
    return helper(x);
}
```