National University of Singapore
School of Computing
CS1101S: Programming Methodology
Semester I, 2021/2022

Consultation Sheet 1
Recursion, List Processing and Higher Order Functions

For the following problems, work them out on paper. Do not use Source Academy to verify correctness. Please present your workings when going for consultation.

Draw diagrams to illustrate your thought process where applicable.

1. **Fast Power**

   (a) Consider the following function:

   ```
   function power(b, n) {
       return n === 0 ? 1 : b * power(b, n-1);
   }
   ```

   Does *power* give rise to an iterative or recursive process?

   Use the Θ notation to characterize the running time and space consumption of *power* as the argument n grows.

   (b) Given that

   $$b^n = \begin{cases} b^{n/2}b^{n/2} & n\,is\,even \\ b^{(n-1)/2}b^{(n-1)/2}b & n\,is\,odd \end{cases}$$

   Using the above, implement a function `fast_power(b,n)` which computes b^n where n is a natural number.

   Does your implementation give rise to an iterative or recursive[1] process?

   Use the Θ notation to characterize the running time and space consumption of $fast\_power$ as the argument n grows.

[1] Try to do both if time permits

National University of Singapore
School of Computing
CS1101S: Programming Methodology
Semester I, 2021/2022

Consultation Sheet 1
Recursion, List Processing and Higher Order Functions

For the following problems, work them out on paper. Do not use Source Academy to verify correctness. Please present your workings when going for consultation.

Draw diagrams to illustrate your thought process where applicable.

1. **Fast Power**

   (a) Consider the following function:

   ```
   function power(b, n) {
       return n === 0 ? 1 : b * power(b, n-1);
   }
   ```

   Does *power* give rise to an iterative or recursive process?

   Use the Θ notation to characterize the running time and space consumption of *power* as the argument n grows.

   (b) Given that

   $$b^n = \begin{cases} b^{n/2}b^{n/2} & n\,is\,even \\ b^{(n-1)/2}b^{(n-1)/2}b & n\,is\,odd \end{cases}$$

   Using the above, implement a function `fast_power(b,n)` which computes b^n where n is a natural number.

   Does your implementation give rise to an iterative or recursive[1] process?

   Use the Θ notation to characterize the running time and space consumption of $fast\_power$ as the argument n grows.

[1] Try to do both if time permits

2. **List Processing**

   (a) **Zip**. Given two lists `xs = list(x1, x2, ... xn)` and `ys = list(y1, y2, ... yn)`, write a function `zip(xs, ys)` that produces the list `list(pair(x1, y1), pair(x2, y2), ..., pair(xn, yn))`.

   (b) **Reverse**. Given a list `xs = list(x1, x2, ... xn)`, write a function `reverse(xs)` that produces the list `list(xn, xn-1, ..., x1)`.

   Use the Θ notation to characterize the running time and space consumption of `reverse`.

   Please note that this is from your lecture notes. Do not refer to your lecture notes, please show your own understanding.

   Also, please try to produce both the recursive and iterative versions.

   (c) [Challenge] **Multi-zip**. Given a list of lists `ls = list(as, bs, cs, ...)` where each of the sublists has the following format: `xs = list(x1, x2, ... xn)`, write a function `multizip(ls)` that produces the list

```
list(list(a1, b1, c1, ...),
     list(a2, b2, c2...),
     ...,
     list(an, bn, cn, ...))
```

3. Higher order functions

   (a) **Filter, using accumulate**. Given a list `xs = list(x1, x2, ... xn)`, write a function `filter(f, xs)` which produces a sublist of `xs` containing only items in `xs` satisfying `f`
   The definition of accumulate is provided for your convenience:

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

   (b) **Filter tree**. Given a tree of items `tree`, write a function `filter_tree(f, tree)` which produces a subtree of `tree` containing only the data items in `tree` satisfying `f`.

   (c) Modify `filter_tree` to remove empty sub-trees resulting from removed items.

1. **Fast Power**

(a) Consider the following function:

```
function power(b, n) {
    return n === 0 ? 1 : b * power(b, n-1);
}
```

b * b * b * b * ... (n times)

Does *power* give rise to an iterative or recursive process?

Use the Θ notation to characterize the running time and space consumption of *power* as the argument n grows.

Recursive, deferred ops.

power(b, n)

b * power(b, n-1)

b * (b * power(b, n-2))

⋮

∴ linear time & space

$\Theta(n)$ time, $\Theta(n)$ space

(b) Given that

$$b^n = \begin{cases} b^{n/2}b^{n/2} & n \text{ is even} \\ b^{(n-1)/2}b^{(n-1)/2}b & n \text{ is odd} \end{cases}$$

Using the above, implement a function `fast_power(b,n)` which computes b^n where n is a natural number.

Does your implementation give rise to an iterative or recursive[1] process?

Use the Θ notation to characterize the running time and space consumption of *fast_power* as the argument n grows.

Try to do both if time permits

```
function fast-power (b, n) {
    if (n == 0) { // base case
        return 1;
    } else if (n % 2 == 0) { // if n is even.
        return fast_power(b, n/2) * fast_power(b, n/2);
    } else {
        return fast_power(b,(n-1)/2) * fast_power(b, (n-1)/2) * b;
    }
}
```

trick or $b^2, n/2$ or $(b, n/2)^2$.

$O(1) + T(\frac{n}{2}) \geq T(\_\_) + O$

recursive

only different.

∴ Recursive, $\Theta(n^2)$ time & space. $O(n)$

$T(n) = 2T(\frac{n}{2}) + O(1)$ becomes

$= O(1) + 2(T(\frac{n}{2}))$.

$+ 2(O(1) + 2T(\frac{n}{4}))$

$+ 4T(\frac{n}{4})$

$+ \dots$

$= (\frac{1}{2})^n n = 1 \quad 2^k \sum_{i=1}^{n} 2^k = 2^{k+1} - 1$

$= 2 \cdot 2^{\log_2 n} - 1$

$= 2n - 1$

$O(1)(n-1)$

$(n-1)(O(1))$

$O(n-1)$

$\approx O(n)$

GP: $\frac{a(r^n - 1)}{r - 1}$

$O(1) + 2(O(1)) + 4(O(1))$

$+ 8(O(1))$

$+ 16(O(1))$

$O(1)\left(\frac{2^{\log_2 n} - 1}{2 - 1}\right)$

$2^{\log_2 n} = n$

$O(1)(2^{\log_2 n} - 1)$

$= 2^{\log_2 n} - 1 = n - 1$

$O(1) \times (n-1)$

$O(n-1) \Rightarrow O(n)$

```
function fast-power (b, n) {
    function iter(b, n, count) {
        if count == 0 {
            return b;
        } else if (n % 2 == 0) { even case.
            return iter(fast-power(b, (n-1)/2
```

accumulator

recursive

odd case?

$\log_2 n$

$2^{\log_2 n} = n$

$f(f^{-1}(x)) = x$

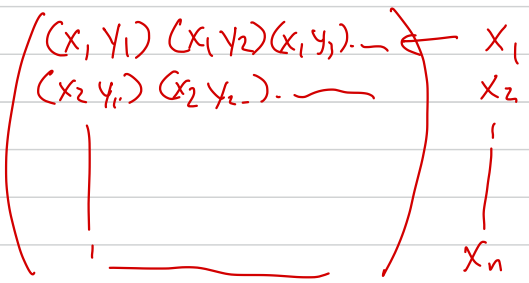
2. List Processing

(a) **Zip.** Given two lists `xs = list(x1, x2, ... xn)` and `ys = list(y1, y2, ... yn)`, write a function `zip(xs, ys)` that produces the list `list(pair(x1, y1), pair(x2, y2), ..., pair(xn, yn))`.

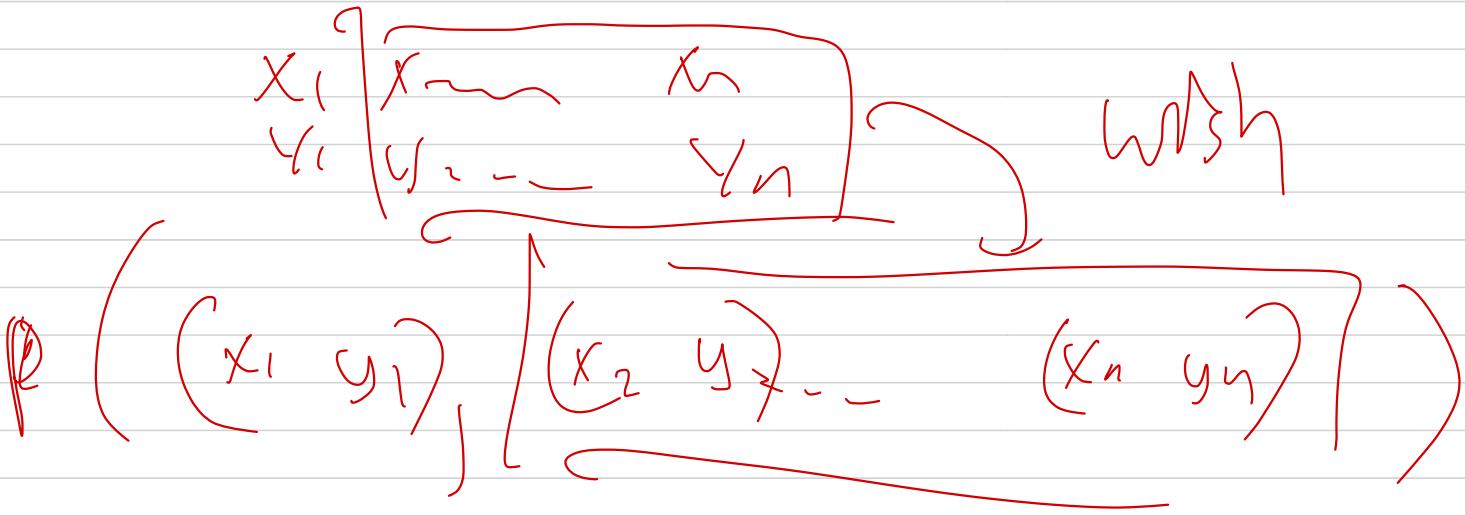
a) function zip(xs, ys) {
return map(x => map(y => pair(x, y), ys), xs);
}.

xs: list(x1, x2, x3 ...)

return list(pair(x1, y1) ...).



return is-null(xs)
? null
: pair(pair(head(xs), head(ys)), zip(tail(xs), tail(ys)));



(b) **Reverse**. Given a list `xs = list(x1, x2, ... xn)`, write a function `reverse(xs)` that produces the list `list(xn, xn-1, ..., x1)`.

Use the Θ notation to characterize the running time and space consumption of `reverse`.

Please note that this is from your lecture notes. Do not refer to your lecture notes, please show your own understanding. :(

Also, please try to produce both the recursive and iterative versions.

list(1, 2, 3, 4, 5)
list(5, 4, 3, 2, 1)

```
function reverse(xs) {
    function iter(current_xs, count) {
        return count < 0
            ? current_xs
            : iter(append(current_xs, list(list_ref(xs, count))),
                   count - 1);
    }
    return iter(list(), length(xs));
}
```

$O(n^2)$ time, $O(n)$ space

```
function reverse(xs) {
    if (is_null(xs)) {
        return null;
    } else {
        const element = list_ref(xs, length(xs));
        return list(element,
                    reverse(filter(x => x !== element, xs)));
    }
}
```

$O(n)$ time & space.

(c) [Challenge] **Multi-zip**. Given a list of lists `ls = list(as, bs, cs, ...)` where each of the sublists has the following format: `xs = list(x1, x2, ... xn)`, write a function `multizip(ls)` that produces the list

```
list(list(a1, b1, c1, ...),
     list(a2, b2, c2...),
     ...,
     list(an, bn, cn, ...))
```

sorts all the first elements of each list to create a list of first elements at the first position of another list.

1. 2 loops — the list itself.
   — the element in each -s.
2. Outer loop.
   go thru each list.
3. Inner loop.
   create a list for each element,
   append the list to the accumulated list
   and append the element to the list

o o o

```
function multizip (ls) {
  return accumulate ( append,
                      null,
                      map ( xs => {  const pos1 = position of xs. in ls
                                     return accumulate ( append,
                                                         null,
                                                         map ( x => { const pos2 = pos of element x. in xs
                                                                      return list_ref(list_ref( xs, pos1), pos2)
                                                         , xs)
                      }, ls)
  );
}
function pos( element, list) {
  return element === head(list)
         ? 0
         : 1 + pos( element, tail(list));
}
```

list
elem
mistake

(x1 — xn)

app (x1 , x2)

not list

3. Higher order functions

(a) **Filter, using accumulate**. Given a list `xs = list(x1, x2, ... xn)`, write a function `filter(f, xs)` which produces a sublist of `xs` containing only items in `xs` satisfying `f`
The definition of accumulate is provided for your convenience:

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

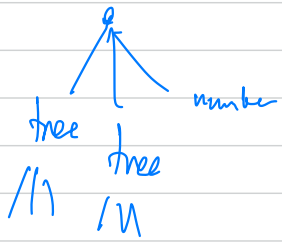
x => x < 5

f(a, f(b, f(c, ... f(z, null) ...

function filter(f, xs) {
return accumulate((x, y) => { if (f(x)) {
return pair(x, y);
} else { return y }.
},
null,
xs);
}

p(c, p(b, p(a, null)))

(b) **Filter tree**. Given a tree of items `tree`, write a function `filter_tree(f, tree)` which produces a subtree of `tree` containing only the data items in `tree` satisfying `f`.




```
function filter_tree ( f, tree ) {

return a tree with elements that satisfies f.
  = elements that does not satisfy f are
    removed from the tree

  function helper ( current_tree, tree ) {
      return is_null (tree)
          ? current_tree
          : is_number ( head (tree))
          ? f( head (tree))
              ? remove from tree
                helper ( remove ( head(tree), tree) , tail(tree) )
              : helper( current_tree, tail (tree))
          : helper ( helper( current_tree, head (tree)), tail (tree));
  }.                                  \ eva this first     \ this later.
  function remove ( element , tree) {
      return is_null (tree)
          ? tree
          : is_number ( head (tree))
          ? equal ( element, head (tree))
              ? tail (tree)
              : tree.
          : append ( remove ( element, head (tree)), remove ( element, tail (tree)) );
  }.
  return helper ( tree , tree );
}.
```

```
function filter_tree (f, tree) {
    if (is_null(tree)) {
        return null;
    }
    else if (is_number(tree)) {
        if (f(tree)) {
            return tree;
        } else {
            return list();
        }
    } else {
        return pair(filter_tree(f, head(tree)), filter_tree(f, tail(tree)));
    }
}
```

(c) Modify `filter_tree` to remove empty sub-trees resulting from removed items.

```
function filter_tree ( f, tree ) {
    function helper ( current_tree, tree )
        return is_null (tree)
            ? current_tree
            : is_number ( head (tree))
            ? f( head (tree))
                ? helper ( current_tree, tail (tree))
                : helper ( remove ( head (tree), tree) , tail (tree) )

            : helper ( helper( current_tree, head (tree)), tail (tree));
                                    \ era this first     \ this later.
    };
    function remove ( element , tree) {
        return is_null (tree)
            ? tree
            : is_number ( head (tree))
            ? equal ( element, head (tree))
                ? is_null( tail (tree) )   ( or rest of the list ).
                    ? null.                : if the rest of the list is null, connect the list.
                    : tail (tree)
                : tree.
            : append ( remove ( element, head (tree)), remove ( element, tail (tree)) );

    };
    return helper ( tree , tree );
};
```

```
function filter_tree ( f , tree) {
    if ( is_null (tree) ) {
        return null;
    } else if ( is_number (head (tree)) ) {
        if (f( head (tree))) {
            return pair ( head (tree), filter_tree ( f, tail ( tree)));
        } else if (is_null (tail (tree))) {
            return null;
        } else {
            return pair (null, filter_tree (f, tail(tree)));
        }
    } else {
        // tree & tree
        return pair( filter_tree (f, head (tree)),
                     filter_tree ( f, tail (tree)));
        }
    }
}
```

now leftover ops.

$T(n) = O(n) + T(n-1)$

$= O(n) + O(n-1) + \dots + 1$

$= 1 + 2 + 3 + \dots + n.$

$= \frac{1}{2}(n^2+n) = n^2$

$T(n) = O(n)\,(\,T(n) + O(n)\,)$

$= n(n + T(n-1))$

$= n(n + (n-1) + (n-2) + \dots)$

$= n(1+2+3\dots)$

$= \frac{n}{2}(n^2+n)$ $\frac{n}{2}(2a + (n-1)d)$

$= n^3.$

$T(n) = O(1) + T(\frac{n}{2})$

$= O(1) + O(1) + O(1) \dots$

$= (\frac{1}{2})^k = n$

$n = 2^k$

$\log_2 n = k.$

$T(n) = O(\log n)\,(\,T(n) + O(\log n)\,)$

id (1 + id (1 + id
n
id(n-1)
id(n-1-1)
id(n-1-1-1)
n