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.