Austin Santoso – CS1101S Cheat Sheet

Lazy processing → only process when called

Order of Growth

"Big Oh" O(): Upper bound. Most common, somewhat useful and often the one used in practice

"Big Theta" Θ(): Tight bound. Most useful, but can be very difficult (if not impossible) to compute

"Big Omega" Ω(): Lower bound. Not very useful since we are usually more interested in worst-case performance rather than best case performance

$2^n > n^2 > n \log(n) > n > \log(n) > 1$

USEFUL FUNCTIONS

Coin changer

```
function first_denomination(kinds_of_coins) {
        return kinds_of_coins === 1 ? 5
        : kinds_of_coins === 2 ? 10
        : kinds_of_coins === 3 ? 20
        : kinds_of_coins === 4 ? 50
        : kinds_of_coins === 5 ? 100
        : 0;
}

function cc(amount, kinds_of_coins) {
   return amount === 0
      ? 1
      : amount < 0 ||
        kinds_of_coins === 0
        ? 0
        : cc(amount, kinds_of_coins - 1)
          +
          cc(amount - first_denomination(
kinds_of_coins),
                kinds_of_coins);
}
```

ABSTRACTION

Lists

append(xs, ys) → combines 2 lists

```
function append(xs, ys) {
        return is_empty_list(xs)
                ? ys
                : pair(head(xs),
                        append(tail(xs), ys));
}
```

reverse(xs) → reverse the order of the list xs

```
function reverse(xs) {
        function rev(original, reversed) {
        return
        is_empty_list(original)
                ? reversed
                : rev(tail(original),
                        pair(head(original),
                        reversed));
        }
        return rev(xs,[]);
}
```

PERMUTATIONS

```
function subsets(s) {
        if (is_empty_list(s)) {
        return list([]);
                } else {
                const s1 = subsets(tail(s));
                const x = head(x);
                return append(s1,
                map(ss => pair(x, ss),
        s1));
} }

function subsets(s) {
  return accumulate(
        (x, s1) => append(s1,
                map(ss => pair(x, ss),
        s1)),
        list([]),
        s);
}

function subsets(xs) {
   if(is_empty_list(xs)) {
      return list([]);
   } else {
      const sub = subsets(tail(xs));
      return append(list([]),
append(map(x => pair(head(xs), x), sub),
tail(sub)));
   }
}
```

WISHFUL THINKING

```
function permutations(s) {
        return is_empty_list(s)
                ? list([])
                : accumulate
                (append, [],
                map(x => map(p =>
        pair(x, p),

                permutations(remov
                e(x, s))),
                        s));
}
```

SORTING

INSERTION SORT

```
function insert(x, xs) {
        return is_empty_list(xs) ?
list(x)
                : x <= head(xs) ?
pair(x,xs)
                : pair(head(xs),
        insert(x, tail(xs)));
}

function insertion_sort(xs) {
        return is_empty_list(xs) ?
xs
                : insert(head(xs),

        insertion_sort(tail(xs)));
}
```

Austin Santoso – CS1101S Cheat Sheet

SELECTION SORT

```
// find smallest element of a non-empty list xs
function smallest(xs) {
        const y = head(xs);
        const ys = tail(xs);
        return is_empty_list(ys) ?
head(xs)
        : y <= head(ys) ?
smallest(pair(y,
                        tail(ys)))
        : smallest(pair(head(ys),
tail(ys)));
}

function selection_sort(xs) {
        if (is_empty_list(xs)) {
        return xs;
        } else {
                const x =
        smallest(xs);
                return pair(x,
                selection_sort(re
        move(x, xs)));
        }
}
```

MERGE SORT

```
function merge(xs, ys) {
   if (is_empty_list(xs)) {
      return ys;
   } else if (is_empty_list(ys)) {
      return xs;
   } else {
      const smallest_x = head(xs);
      const smallest_y = head(ys);
      return (smallest_x < smallest_y)
          ? pair(smallest_x,
merge(tail(xs), ys))
          : pair(smallest_y, merge(xs,
tail(ys)));
   }
}

function merge_sort(xs) {
   if (is_empty_list(xs) ||
is_empty_list(tail(xs))) {
// base case: an empty list or a 1
item list is already sorted
      return xs;
   } else {
      const half_len =
middle(length(xs));
      const LHS = take(xs, half_len);
      const RHS = drop(xs, half_len);
      return merge(
         merge_sort(LHS), // sorted
LHS
         merge_sort(RHS)  // sorted
RHS
      );
   }
}
```

QUICK SORT

```
function partition(xs, p) {
   function
partition_helper(list_smaller,
list_greater, list) {
      if(is_empty_list(list)) {
         return
pair(reverse(list_smaller),
reverse(list_greater));
      }
      else if(head(list) <= p) {
         return
partition_helper(pair(head(list),
list_smaller), list_greater, tail(list));
      }
      else {
         return
partition_helper(list_smaller,
pair(head(list), list_greater),
tail(list));
      }
   }
   return partition_helper([], [], xs);
}

function quicksort(xs) {
   if(is_empty_list(xs)) {
      return [];
   }
   else {
      const first_split =
partition(tail(xs), head(xs));
      return accumulate(append, [],
list(quicksort(head(first_split)),
list(head(xs)),
quicksort(tail(first_split))));
   }
}

function compose(f, g){
         return x => f(g(x));
}
```

ABSTRACTION

```
function thrice(f) {
         return
compose(compose(f, f), f);
}
```

WISHFUL THINKING

Stream Memoization:

```
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;
}
```