Exercise Class I

Shengyi Jiang, Qinlin Chen, Yicheng Huang,
Zhiqi Chen & Zhehao Lin

October 29, 2020

NANJING UNIVERSITY

Table of Contents

Table of Contents

Lab03-05 Maximum Subsequence

Problem: Return the maximum subsequence (not necessarily contiguous) of length at most `l` (e.g., 3) that can be found in the given number `n` (e.g., 20125).

Thought:

1. It's hard to swallow it all once, so how can I divide this problem into smaller ones?
2. I'm given `n` and `l`, where `n` can repeatedly perform `n//10` until reaching `0`, and `l` may decrease itself to `0`.
3. Well, each time I only consider a bit of `n`, which has only two choices: in the maximum subsequence or not.

Solution: For each `n` and `l`, we denote the maximum subsequence in this case as `max_subseq(n,l)`. `max_subseq(n,l)` is the larger one of the following splitted cases.

- `max_subseq(n//10, l-1)*10 + n%10` // the last digit is in `max_subseq(n,l)`
- `max_subseq(n//10, l)` // otherwise

Lab03-05 Maximum Subsequence (cont'd)

Brief proof of solution: For the convenience of presentation, we denote the number n as $n_1 n_2 \dots n_k$, where n_i means the i-th bit of n. We also denote the maximum subsequence of $n_1 n_2 \dots n_k$ in length at most l as $s(n_1 n_2 \dots n_k, l)$. For example, when $n_1 = 2$ and $n_2 = 3$, $s(n_1 n_2, 1) = n_2$.

- If n_k is in $s(n_1 n_2 \dots n_k, l)$, we can conclude that $s(n_1 n_2 \dots n_{k-1}, l-1) n_k = s(n_1 n_2 \dots n_k, l)$ since n_k occupies one length. To prove, if we have another different subsequences $tn_k = s(n_1 n_2 \dots n_k, l)$, we can always replace t by $s(n_1 n_2 \dots n_{k-1}, l-1)$ because the latter one is the maximum subsequences of $n_1 n_2 \dots n_{k-1}$ in length at most $l-1$.
- If n_k is not in $s(n_1 n_2 \dots n_k, l)$, we can conclude that $s(n_1 n_2 \dots n_{k-1}, l) = s(n_1 n_2 \dots n_k, l)$. The proof is similar.
- Combining the above two cases, we choose the larger one, which is the globally maximum solution.

Lab03-05 Maximum Subsequence (cont'd)

Example: The following table shows the calculation procedures of the problem `max_subseq(n=20125, l=3)`. The color blue represents the base case of recursion, while the color red represents the original problem.

Insight: When splitting problems, from red to blue, each step we jump to the one above or left-above. (`max(left-above * 10 + now_last_digit, above)`). On the contrary, the value calculated flows from blue to red.

value flow		l = 0	1	2	3
n	0	↓ ↘*	↓ ↘	↓ ↘	↓
	2	↓ ↘	↓* ↘	↓ ↘	↓
	20	↘	↓* ↘	↓ ↘	↓
	201		↘*	↓ ↘	↓
	2012			↘*	↓
	20125				**goal**

max_subseq (n, l)		l = 0	1	2	3
n	0	**0**	0	0	0
	2	0	**2**	2	2
	20	0	**2**	20	20
	201	0	**2**	21	201
	2012	0	2	**22**	212
	20125	0	5	25	**225**

Lab03-05 Maximum Subsequence (cont'd)

Code Sample:

```
def max_subseq(n, l):
    if l == 0 or n == 0:
        return 0
    case1 = max_subseq(n // 10, l - 1) * 10 + n % 10
    case2 = max_subseq(n // 10, l)
    return max(case1, case2)
```

Table of Contents

HW03-02 Ping-pong

Key points:

- The ping-pong value is locally monotonous (e.g., it decreases from 7 to 0 when the index increases from 7 to 14). $\rightarrow$ **A locally monotonous variable recording current value.**
- The ping-pong value sometimes (when the index k is a multiple of 7 or contains the digit 7) changes its monotonicity. $\rightarrow$ **A variable recording the direction/monotonicity.**
- In a tail recursion manner, it performs well.

HW03-02 Ping-pong (cont'd)

Code Sample: `cur_val` records the current value, and `direc` records the current direction (+1 or −1). `-direc` means changing direction.

```
def pingpong(n):
    def state(cur_index, target, cur_val, direc):
        if cur_index == target:
            return cur_val
        if cur_index % 7 == 0 or num_sevens(cur_index) > 0:
            return state(cur_index + 1, target, cur_val - direc, -direc)
        return state(cur_index + 1, target, cur_val + direc, direc)
    return state(1, n, 1, 1)
```

NANJING UNIVERSITY

Table of Contents

南京大學 NANJING UNIVERSITY

HW03-03 Count Change

Problem: Once the machines take over, the denomination of every coin will be a power of two: 1-cent, 2-cent, 4-cent, 8-cent, 16-cent, etc. *There will be no limit to how much a coin can be worth.* Given a positive integer `total`, a set of coins makes change for `total` if the sum of the values of the coins is `total`. Write a recursive function `count_change` that takes a positive integer `total` and returns **the number of ways** to make change for `total` using these coins of the future.

HW03-03 Count Change (cont'd)

Thought: What do we have? (1) Unlimited kinds of coins with increasing denominations; (2) The goal of summation of coins, *total*. The lower bound of (1) is known (i.e., 1-cent), while the upper bound of (2) is also known (i.e., *total*). So it's not hard to figure out that you should try to use coins with increasing denominations in order, while `goal` is decreasing when using coins. The rest is similar to the problem "Maximum Subsequence" talked above.

Solution: Regarding to 1-cent denomination, we have two choices: use a coin with this denomination or simply not use this denomination. If we use it, we can decrease our `total` by 1 and can further decide whether to use 1-cent denomination; If we do not use it, our `total` is unchanged and we can only use coins with denominations larger than 1 (at least 2-coin) later. The same is true for i-coin. The number of ways is the addition of that of these two choices.

HW03-03 Count Change (cont'd)

Solution (cont'd): Denote `rec_count(min_coin,sub_total)` as the number of ways to make change for `sub_total` using coins with denominations `min_coin`-cent, `2*min_coin`-cent, etc.

```
    rec_count(min_coin, sub_total)
  = rec_count(min_coin*2, sub_total)
+ rec_count(min_coin, sub_total-min_coin)
```

Base Case:

- `sub_total == 0`
  → return 1 (exactly match)
- `sub_total < min_coin`
  → return 0 (no more enough)

Example: When we make changes for 7:

rec_count		*min_coin* 8	4	2	1
sub_total	0	1	1	1	1
	1	0	0	0	1
	2	0	0	1	2
	3	0	0	0	2
	4	0	1	1	4
	5	0	0	0	4
	6	0	0	2	6
	7	0	0	0	**6**

HW03-03 Count Change (cont'd)

Code Sample:

```python
def count_change(total):
    def rec_count(min_coin, sub_total):
        if sub_total == 0:
            return 1
        if sub_total < min_coin:
            return 0
        min_coin_used = rec_count(min_coin, sub_total - min_coin)
        min_coin_unused = rec_count(min_coin * 2, sub_total)
        return min_coin_used + min_coin_unused
    return rec_count(1, total)
```

Table of Contents

HW02-04 Make Repeater

Problem: Implement the function `make_repeater` so that `make_repeater(h, n)(x)` returns `h(h(...h(x)...))`, where h is applied n times.

HW02-04 Make Repeater (cont'd)

Solution: It's easy to define a function that computes the value of $h^{(n)}(x)$. So just define a helper function (that computes the value of $h^{(n)}(x)$) and returns it.

```python
def make_repeater(h, n):
    def repeater(x):
        i = 0
        while i < n:
            x = h(x)
            i += 1
        return x
    return repeater
```

HW02-04 Make Repeater (cont'd)

Solution: A recursive thinking:

- $n = 1$, return h ($n = 0$, return identity)
- $n = k$, we have
  $h^{(k-1)}$=make_repeater(h,n-1) $\Rightarrow$
  $h^{(k)}$=compose(h,make_repeater(h,n-1))

```python
def make_repeater(h, n):
    if n == 1:
        return h
    else:
        return compose(h,
            make_repeater(h, n-1))
```

HW02-04 Make Repeater (cont'd)

Solution: compose is an operator defined on function space $\mathcal{F} \times \mathcal{F}$. Especially, when two operands are f and power of f, compose is commutable. There is a homomorphism between $(f, \mathit{compose})$ and $(\mathbb{N}^+, +)$. Recall that we have defined `accumulate` to abstract similar operations on int, so...

```
def make_repeater(h, n):
    return accumulate(compose,
      identity, n, lambda i: h)
```

Table of Contents

HW03-04 Missing Digits

Problem: Write the recursive function `missing_digits` that takes a number n that is sorted in non-decreasing order. It returns the number of missing digits in n. A missing digit is a number between the first and last digit of n of a that is not in n.

HW03-04 Missing Digits (cont'd)

Solution: The number is sorted in non-descreasing order. We can track the value of current digit d.

- base case: $n < 10$, return 0
- n, d, compute `next_d` and compute `f(n//10, next_d)`.

Be careful with same digits.

```python
def missing_digits(n):
    def helper(n, current_digit):
        if n < 10:
            return 0
        next_digit = (n // 10) % 10
        return max(current_digit - \
        next_digit - 1, 0) + \
        helper(n//10, next_digit)
    return helper(n, n % 10)
```

HW03-04 Missing Digits (cont'd)

You can find that next_d is equal to the last digit of n. So we do not have to exlicitly track it.

```
def missing_digits(n):
    if n < 10:
        return 0
    right_first_digit = n % 10
    right_second_digit = (n // 10) % 10
    if right_second_digit < right_first_digit:
        return missing_digits(n // 10) + \
        (right_first_digit - right_second_digit) - 1
    return missing_digits(n // 10)
```

Table of Contents

Lab02-04 | Heard You Liked Functions

Define a function cycle that takes in three functions f1, f2, f3, as arguments. cycle will return another function that should take in an integer argument n and return another function. That final function should take in an argument x and cycle through applying f1, f2, and f3 to x, depending on what n was.

- n = 0, return x
- n = 1, apply f1 to x, or return f1(x)
- n = 2, apply f1 to x, and then f2 to the result of that, or return f2(f1(x))
- n = 3, apply f1 to x, f2 to the result of applying f1, and then f3 to the result of applying f2, or f3(f2(f1(x)))
- n = 4, start the cycle again applying f1, then f2, then f3, then f1 again, or f1(f3(f2(f1(x))))

And so forth.

Lab02-04 I Heard You Liked Functions (cont'd)

Solution: We have `f1, f2, f3`: $\mathrm{T} \to \mathrm{T}$, we need a function $\mathrm{cycle} : (\mathrm{T} \to \mathrm{T}, \mathrm{T} \to \mathrm{T}, \mathrm{T} \to \mathrm{T}) \to (n : \mathrm{int} \to x : \mathrm{T} \to y : \mathrm{T})$. First, define the inner-most function g that computes the value given `x` and `n`. Second, define function f that take n that returns `g(x, n)`. Last, return `f`.

Lab02-04 I Heard You Liked Functions (cont'd)

```python
T = TypeVar('T')

def g(x: T, n: int, f1: Callable[[T], T],
      f2:Callable[[T], T], f3:Callable[[T], T]):
      res, i = x, 1
      while i <= n:
        if i % 3 == 1:
          res = f1(res)
        elif i % 3 == 2:
          res = f2(res)
        else:
          res = f3(res)
        i += 1
      return res
def f(n: int, f1: Callable[[T], T],
      f2: Callable[[T], T], f3: Callable[[T], T]):
    return lambda x: g(x, n, f1, f2, f3)
def cycle(f1: Callable[[T], T],
          f2: Callable[[T], T], f3: Callable[[T], T]):
  return lambda n: f(n, f1, f2, f3)
```

```python
def cycle(f1, f2, f3):
    def f(n):
        def g(x):
            res, i = x, 1
            while i <= n:
                if i % 3 == 1:
                    res = f1(res)
                elif i % 3 == 2:
                    res = f2(res)
                else:
                    res = f3(res)
                i += 1
            return res
        return g
    return f
```

NANJING UNIVERSITY

Q & A