Computational Structures in Data Science

Midterm Review

Berkeley
UNIVERSITY OF CALIFORNIA

Announcements & Policies

- Midterm:
  - 2 hours, 120 Minutes
  - 5 **Handwritten** Cheat sheets – More than ~3 is counter-productive
    - Must be handwritten!
  - 1 CS88 Provided Reference Sheet (See previous exams page)
- Academic Integrity violations: -100% and no clobber policy.
- **Everything through this week is in scope!**
  - functions, loops, sequences, HOFs, dictionaries, ADTs, recursion, tree recursion

You are not your grades!

Do your best!

My Advice

- Don't rush!
  - Slow is fast and fast is slow
  - BREATHE!
- Skim the exam first
  - It's ok to do questions out of order!
  - Get the stuff you're good without out of the way
  - BUT don't spend too much time planning the exam.
- Read through the question once
  - What's it asking you to do at a high level?
  - What do the doctests suggest?
  - What techniques should you be using?
- Use the scratch space!

Midterm Topics

- Functions
- Higher Order Functions
  - Functions as arguments
  - Functions as return values
- Environment Diagrams
- Lists, Dictionaries
- List Comprehensions, Dictionary Comprehensions
- Abstract Data Types
- Recursion & Tree Recursion

Computational Structures in Data Science

Recursion Review

Berkeley
UNIVERSITY OF CALIFORNIA

Michael Ball | UC Berkeley | https://c88c.org |

The Recursive Process

Recursive solutions involve two major parts:

- Base case(s), the problem is simple enough to be solved directly
- Recursive case(s). A recursive case has three components:
  - **Divide** the problem into one or more simpler or smaller parts
  - **Invoke** the function (recursively) on each part, and
  - **Combine** the solutions of the parts into a solution for the problem.

Advice

- Breaking down the problem:
  - What is the base case? What is the smallest or simplest argument?
  - How many different 'pathways' are there? Tree recursion means 2 or more pathways
- Double-check the return types!
  - What is being 'combined' (e.g. added, min/max, etc). The return value must be compatible.

Count Change Revisited, 61A FA22 Final Q5

- CS61A Fall 2022 Final Exam [61A Exams Page]
- Preview: Adapt Count Change, then figure out how handle additional constraints.

```
coins = (1, 5, 10, 25, 50)
def count_change(amount):
    def helper(remaining, coin_index):
        if coin_index == len(coins) or remaining < 0:
            return 0
        if remaining == 0:
            return 1
        return (helper(remaining - coins[coin_index], coin_index    ) +
                helper(remaining                    , coin_index + 1))
    return helper(amount, 0)
```

Modify this code to solve the following new problem:

Your cash register only has `k` of each type of coin. Implement `count_change_register`, which counts the number of ways to make change for `amount` using at most `k` coins of each type.

```
def count_change_register(amount, k):
    '''Return the number of ways to make change for amount using at most k of each coin type.

    >>> count_change_register(20, 10) # Excludes 20 pennies and excludes 1 nickel + 15 pennies
    7
    >>> count_change_register(20, 2) # 10-10, 10-5-5
    2
    >>> count_change_register(100, 10)
    84
    >>> count_change_register(100, 100)
    292
    '''
    def helper(remaining, coin_index, n):
        if coin_index == len(coins) or remaining < 0 or __(a)__:
            return 0
        if remaining == 0:
            return 1
        return (helper(remaining - coins[coin_index], coin_index    , __(b)__ ) +
                helper(remaining                    , coin_index + 1, __(c)__ ))
    return helper(amount, 0, 0)
```

Select The Correct Option – View Page 13

- 5.1: Fill in Blank (a)

• `True` • `False` • `n > 0` • `k > 0` • `n > k` • `n == k` • `n > 10` • `n == 10`

- 5.2: Fill in Blank (b)

• `k` • `k - 1` • `k + 1` • `n` • `n + 1` • `n - 1` • `0` • `10`

- 5.3: Fill in Blank (c)

• `k` • `k - 1` • `k + 1` • `n` • `n + 1` • `n - 1` • `0` • `10`

- 5.4: What does `count_change_register(25, 2)` return?

• `0` • `1` • `2` • `3` • `4` • `None of these`

Select The Correct Option – View Page 13

- 5.1: Fill in Blank (a)

• `True` • `False` • `n > 0` • `k > 0` **• `n > k`** • `n == k` • `n > 10` • `n == 10`

- 5.2: Fill in Blank (b)

• `k` • `k - 1` • `k + 1` • `n` **• `n + 1`** • `n - 1` • `0` • `10`

- 5.3: Fill in Blank (c)

• `k` • `k - 1` • `k + 1` • `n` • `n + 1` • `n - 1` **• `0`** • `10`

- 5.4: What does `count_change_register(25, 2)` return?

• `0` • `1` **• `2`** • `3` • `4` • `None of these`

Computational Structures in Data Science

Some Practice Questions

Berkeley
UNIVERSITY OF CALIFORNIA

Exam Practice

View exams https://c88c.org/fa25/resources/

- Spring 22 Q7 — Closet Overhead
- Spring 20 Q5 — Atey Ate Already

SP22

7. (5.0 points) Closet Overhaul

You've designed a closet abstract data type to help you organize your wardrobe.

A closet contains two things:

- `owner`: the name of the closet owner represented as a string
- `clothes`: the collection of clothes in the closet represented as a dictionary, where the key is the clothing item name and the value is the number of times the clothing item has been worn.

The `make_closet` constructor takes in `owner` (a string) and `clothes` (a **list** of strings representing clothing items) and returns a closet ADT.

Given this, you've implemented the abstract data type as follows:

```
def make_closet(owner, clothes):
    """ Create and returns a new closet. """
    clothes_dict = {}
    for item in clothes:
        clothes_dict[item] = 0
    return (owner, clothes_dict)

def get_owner(closet):
    """ Returns the owner of the closet """
    return closet[0]

def get_clothes(closet):
    """ Returns a dictionary of the clothes in the closet """
    return closet[1]
```

Given the closet ADT, implement the functions `wear_clothes` and `favorite_clothing_item`. You may not need all the lines provided, and you may need to change the indentation for some lines.

SP20

5. (10 points) Atey Ate Already

It's a lot more fun to think about food than take midterms, so let's look at the cheapest places to fulfill an order. Given the function `total_cost` and assuming it works as described, fill out `find_restaurant` to find the cheapest restaurant to fulfill the order.

Remember: Pay close attention to the doctests to guide your solution.

```
def total_cost(restaurant, order):
        """
        Function that returns the total cost of an order at a certain
        restaurant. Returns -1 if fulfilling the request is not possible.
        >>> total_cost('chipotle', ['burrito', 'taco'])
        11.96
        >>> total_cost('sliver', ['boba'])
        -1.0
        """
        # We have omitted how this function works.
def find_restaurant(restaurants, order):
    """
    Function that returns the cheapest restaurant and price as the first
    element of a list followed by the prices for each of the restaurants.
    In the case that no restaurant can fulfill the order, the first
    element should be ['None found!', -1]. In the case that two
    restaurants have the same price, keep the first restaurant.
    Hint: Use total_cost!
    >>> find_restaurant(['chipotle', 'la burrita'], ['burrito', 'taco'])
    [['la burrita', 9.78], [['chipotle', 11.96], ['la burrita',9.78]]

    >>> find_restaurant(['sliver','cheeseboard'], ['boba'])
    [[None found!, -1.0], ['sliver', -1.0]['cheeseboard', -1.0]]
    """
```

```
def findRestaurant(restaurants, order):
    """
    Hint: Use totalCost In the case that two restaurants have the same price,
    keep the first restaurant.
    >>> findRestaurant(['chipotle', 'la burrita'], ['burrito', 'taco'])
    [['la burrita', 9.78], [['chipotle', 11.96], ['la burrita', 9.78]]]
    >>> findRestaurant(['sliver', 'cheeseboard'], ['boba'])
    [['None found!', -1.0], [['sliver', -1.0], ['cheeseboard', -1.0]]]
    """
    placesList = [ [ restaurant, __________________ ]
                   for restaurant in restaurants ]
    minCost = -1.0
    cheapestPlace = "None found!"
    for place in __________________:
        …
    …
```

```
def findRestaurant(restaurants, order):
    placesList = [[restaurant, totalCost(restaurant, order)]
                  for restaurant in restaurants]
    minCost = -1.0
    cheapestPlace = "None found!"
    for place in placesList:
        if place[1] != -1.0 and (place[1] < minCost or
minCost == -1.0):
            minCost = place[1]
            cheapestPlace = place[0]
    return [cheapestPlace, minCost] + placesList
```

SP20 #5

(10 points) Atey Ate Already

It's a lot more fun to think about food than take midterms, so let's look at the cheapest places to fulfill an order. Given the function `total_cost` and assuming it works as described, fill out `find_restaurant` to find the cheapest restaurant to fulfill the order.

Remember: Pay close attention to the doctests to guide your solution.

```
def total_cost(restaurant, order):
        """
        Function that returns the total cost of an order at a certain
        restaurant. Returns -1 if fulfilling the request is not possible.
        >>> total_cost('chipotle', ['burrito', 'taco'])
        11.96
        >>> total_cost('sliver', ['boba'])
        -1.0
        """
        # We have omitted how this function works.
def find_restaurant(restaurants, order):
    """
    Function that returns the cheapest restaurant and price as the first
    element of a list followed by the prices for each of the restaurants.
    In the case that no restaurant can fulfill the order, the first
    element should be ['None found!', -1]. In the case that two
    restaurants have the same price, keep the first restaurant.
    Hint: Use total_cost!
    >>> find_restaurant(['chipotle', 'la burrita'], ['burrito', 'taco'])
    [['la burrita', 9.78], [['chipotle', 11.96], ['la burrita',9.78]]

    >>> find_restaurant(['sliver','cheeseboard'], ['boba'])
    [[None found!, -1.0], ['sliver', -1.0]['cheeseboard', -1.0]]
    """
```

```
def findRestaurant(restaurants, order):
    """
    Function that returns the cheapest restaurant and price as the first
    element of a list followed by the prices for each of the restaurants.
    In the case that no restaurant can fulfill the order, the first
    element should be ['None found!', -1].
    Hint: Use totalCost! In the case that two restaurants have the same price,
    keep the first restaurant.
    >>> findRestaurant(['chipotle', 'la burrita'], ['burrito', 'taco'])
    [['la burrita', 9.78], [['chipotle', 11.96], ['la burrita',9.78]]
    >>> findRestaurant(['sliver','cheeseboard'], ['boba'])
    [[None found!, -1.0], ['sliver', -1.0]['cheeseboard', -1.0]]
    """
    placesList = [ [restaurant, totalCost(restaurant, order)]
                    for restaurant in restaurants ]
    minCost = -1.0
    cheapestPlace = "None found!"
    for place in range(placesList):
        if place[1] != -1.0 and (place[1] < minCost or minCost == -1.0):
            minCost = place[1]
            cheapestPlace = place[0]
    return [cheapestPlace, minCost] + placesList
```