Midterm Review Sheet

Algorithms

1. **Analyze Complexity – Two principles**
   a) Count the times of iterations
      i. "for loops" will usually go through every element in an iterable object, which usually means *O(n)*. E.g., *for i in range(len(list)):* (It depends on the "thing" in range(), if it's a constant, then the complexity should be *O(1)*).
      ii. "while loops" usually have a control statement, like *i -= 1* or *i //= 2*. The former one usually has *O(n)* complexity, and the latter usually have *O(logn)* complexity. ("+, -" means linear, "*, /" means logarithmic.) But be flexible, always remember, the central point is to count the times of iterations.
   b) Parallel or Nested?
      i. If two structures are nested, then the final complexity should be the multiplication of the two single complexities.
      ii. If two structures are parallel, the final complexity equals to the larger one.

```python
def complexity_example(lst):
    for i in range(len(lst)):
        control_var1, control_var2 = len(lst), len(lst)
        while control_var1 > 0:
            control_var1 //= 2
        while control_var2 > 0:
            control_var2 -= 2
```

Complexity: O(n^2)

2. **Different Algorithms**
   a) Sorting
      i. Bubble Sort – Works like bubbles
         1. In a role, compare each two elements that are next to each other (from left to right), swap the larger one to the right side. Do $n - 1$ roles.
         2. How do we optimize:
            a) If no swap happens in a role, then stop (indicate the array is already sorted).
            b) In the x role, we do not have to check the last $x - 1$ element(s).
         3. Time complexity: *O (n^2)*
      ii. Merge Sort – Divide and conquer
         1. First, cut array to two parts with about equivalent length, sort each of them using merge sort (recurrence). Then, merge the two sorted list (we have two cursors starting from the first index of the two lists, each time compare the two numbers the cursors is pointing at, pop out the smaller element and append it to the large (merged) list, and move its cursor to the next (right) element. Continue do the same thing until all elements are in the large list), and the whole array is sorted.
         2. Time Complexity: *O (n log n)*
   b) Searching
      i. Binary Search
         1. The array should be already sorted.
         2. Each time select the element at the middle of the array, and check the target is in the left or the right, then reduce the scale of the problem to n/2.
         3. Time Complexity: *O (log n)*
   c) Recursion
      i. How can we understand a recursion quickly?
         1. DO NOT TRY TO THINK LIKE COMPUTERS! We only have human brains, don't dive into the recursion and try to figure out how it exactly goes from n -> n-1 -> n-2 -> … -> 1, just make sure two things: 1. Is the base case right? Can the recursion end? 2. How can the algorithms compute the n case from the n-1 case? As long as you make these two things right, the mathematical induction will help you do the rest!

OOP

3. **How We Understand OOP** – Everything is an object! (一切皆对象).
   a) What is an object? A "thing" that should have some **features** and can **interact** with others.
      i. Attributes: What is the feature of the "thing"? What does it look like? (属性)
      ii. Methods: How the "thing" is going to interact with others? (能动性)
   b) The principles of OOP
      i. Encapsulation (封装) – Concealing all the logic inside an object, only provide interface to interact with other objects/functions.
         1. Reflection: getters/setters, variables with "_" (weekly hidden, still can be inherited and accessed) and "__" (Cannot be inherited and accessed).
      ii. Inheritance (继承) – subclass can extend superclass
      iii. Polymorphism (多态) – subclass can overload the methods from its superclass.
4. **How We Implement OOP**
   a) "self"
      i. Add "self." when referring to attributes and methods belongs to class.
      ii. Add "self" at the first argument when you define a method in a class.
      E.g., *def method_name(self, *args):*
   b) Getters and Setters
      i. Getters: Method used to retrieve a certain attribute. Usually just simply *return self.attribute*
      ii. Setters: Method used to assign a new value to a certain attribute. *self.attribute = value*
   c) Predefined Method – "Magic method" that are predefined in python, usually have special use. Looks like *__***__*
      i. *__init__*: Called automatically when initialization.
      ii. *__add__*: Statement like *sum = object_1 + object_2* will call the *__add__* method of *object_1*
      iii. *__str__*: Called only when the statement is a print function.
      iv. *__iter__*: Called when the statement includes iteration, e.g., "for" loops.
   d) Decorators – Add above the method defining statement, to change the functionality of a method.
      i. E,g,. *@property*, to call this method like accessing an attribute

Other

5. **Matplotlib**
   a) Remember to install! *pip install matplotlib*.
6. **Copy**
   a) Please refer to the PowerPoint, that one is much clearer.

Do Well in Midterm!