Beihang University
School of General Engineering

Beijing, November 3rd, 2020
Lin Wei and Yida Ding

Computer Science and Programming
Lab Class 6

Task 1 ***Class Review – Bank Account*** *(15 minutes)*

Implement a Python class `Bank_Account`. This class should model an individual bank account. There should be an initial function which generates an empty bank account with zero RMB, zero Dollars and zero Euros, a function for saving any type of currency, a function for withdrawing any type of currency, a function for remittance between two accounts, a function for checking account balance. In addition, each type of currency has an interest for each year (RMB 1.5%, Dollar 1.2%, Euro 1%). The function names are listed as follows:

1. `__init__(self):`
2. `save_money(self,money_type,money_amount):`
3. `withdraw_money(self,money_type,money_amount):`
4. `remittance(self,other_account,money_type,money_amount):`
5. `check_money(self):`
6. `after_one_year(self):`

Task 2 ***Get execution time*** *(10 minutes)*

First of all, we will introduce a new python package **time**. This module provides various time-related functions.

time.**time**() returns the time in seconds since the epoch as a floating point number. On Windows and most Unix systems, the epoch is January 1, 1970, 00:00:00 (UTC). To find out what the epoch is on a given platform, look at time.**gmtime**(0).

```
>>> import time
>>> print(time.time())
1510226673.87
```

Assmue that you want to get the execution time of the your program that calls the function $max\_of\_3()$, you can store the start time and end time of your program in variable **start_time**, **end_time** respectively and print the difference between them. An example code is shown below.

```
import time

def max_of_3(a,b,c):
    maxi = a
    if b > maxi:
        maxi = b
    if c > maxi:
```

```
        maxi = c
    return maxi

start_time = time.time()    #time starts a function call
max_of_3(4,5,6)
end_time = time.time()    #time ends a function call
print("Execution time is %es" %(end_time - start_time))

Execution time is 1.192093e-06s
```

Task 3 ***Sorting algorithm – Insertion sort*** *(20 minutes)*

Write a function *insert_sort*() to sort an integer list in ascending order with insertion sorting algorithm. Test your function with the several test cases and please compare the execution time of different test cases, with an increasing length of input lists: 100, 1000 elements, 10000 elements, 100000 elements, and 1000000 elements.

Task 4 ***Sorting algorithm – Check whether a list is sorted*** *(10 minutes)*

Write a function *is_sorted*() which checks whether a given list (parameter to the function) is sorted in ascending order. Make sure your function requires linear runtime!

Task 5 ***Sorting algorithm – Specific runtime*** *(40 minutes)*

Find an input list, such that your function *Insert_sort* takes exactly 5.3 seconds to finish sorting. To solve this task, you can take the following steps:

1. Implement a function to compute the execution time of *insert_sort*() for a reversed list ranging from n to 1. This function should receive the length of the list (n) and return the execution time. In addition, please think about why we consider a reversed list here.

2. Please implement the dichotomy method in a recursive way. Use your dichotomy function to obtain the approximate root of $f(x) = 3x^2 + x - 2$ with the accuracy of 0.001 in the range *(0,1)*. **Hint:** In dichotomy, if $f(x_1) * f(x_2) < 0$ for different x_1, x_2, there must be at least one root between x_1 and x_2 for the equation $f(x) = 0$. Therefore, we can explore $\frac{x_1+x_2}{2}$. If $f(x_1) * f(\frac{x_1+x_2}{2}) < 0$, the root is between x_1 and $\frac{x_1+x_2}{2}$; Else if $f(\frac{x_1+x_2}{2}) = 0$, the root is just $\frac{x_1+x_2}{2}$; Else, the root is between $\frac{x_1+x_2}{2}$ and x_2.

3. Please apply the dichotomy method to determine the length of the reversed list n^* which leads to an execution time of 5.3 s. The dichotomy function should receive a lower bound of length n_1, an upper bound of length n_2 and the accuracy $e = 0.1s$, then return the desired length n^*.

Task 6 ***Efficient sorting*** *(if you have finished the other tasks)*

Recall from the lecture that *insert_sort*() requires a quadratic number of steps in the worst case, i.e. the running time is in $O(n^2)$. Can you think of an algorithm for sorting which reduces the running time to linear time? If you cannot find a solution, do not worry: This problem will be addressed in later lectures.