1.목표

change_money_list를 통하여 거스름돈을 최소화한다. 또한, change_money_saved_list에서 계산의 중간 저장을 통해 더 빠르게 결과를 도출하기로 한다.

2.과제를 해결하는 방법

재귀함수에 대한 이해, 결과를 중간 저장하는 것의 필요성, 실행 속도를 줄이는 방법, append 이해

3.과제를 해결한 방법

1) change_money_list

min_coins에 입력된 잔돈을 넣어서, 모두 1원으로 거슬러 준 것으로 초기화한다.
list형 변수인 min_coins_list를 선언해서 거슬러준 돈들을 저장한다.
입력된 돈이 미리 저장된 단위 동전이라면 min_coins_list에 입력된 돈을 저장하고 1, counter, min_coins_list를 반환한다.
그렇지 않다면 list형 변수 coins를 선언하고, coin_value_list에 저장된 단위 동전을 입력된 돈과 비교하면서 작거나 같은 동전들만 coin에 입력한다.
coins에 저장된 coin을 반복한다. 반복하는 동안 list형 변수 new를 선언하고, 반복중인 coin을 new에 저장한다. result를 선언하고 make_change(coin_value_list, coin_change-coin, count=count)를 집어넣어, coin을 쪼개서 재귀를 통해 거스름돈을 찾는다. count=count를 통해 몇 번이나 실행중인지 저장한다. result를 통해 결과를 저장함으로써 실행 시간을 단축시킬 수 있다.
재귀를 통해 result에 저장된 거스름돈의 개수를 num_coins, 거스름돈의 목록과 new를 더한 것을 coins_list, 실행 회수를 count에 저장한다.
반복을 하면서 현재 거스름돈의 개수가 저장된 최소 거스름돈보다 작거나 같다면 최소 거스름돈 개수에 현재 거스름돈의 개수를 저장하고, min_coins_list에 coins_list를 저장하여 최소 거스름돈의 목록을 저장한다.
최종적으로 min_coins, counter, min_coins_list를 반환한다.

2) change_money_saved_list

change_money_saved_list와 유사하나 계산한 것을 저장해놓는 known_result의 차이가 있다. 입력된 돈이 미리 저장된 단위 동전이라면 known_result[coin_change]에 min_coins_list를 입력해 진행된 연산을 저장하고 1, counter, min_coins_list를 반환한다.
그 외의 경우에서 만약 저장된 연산이 있다면 known_result[coin_change]의 길이를 반환해서 거스름돈의 개수를 나타내고, counter와 known_result[coin_change]를 반환하여 실행회수와 거스름돈 목록을 나타낸다. 그 외의 경우에서는 change_money_saved_list와 똑같이 실행하다 현재 거스름돈의 개수와 저장된 최소 거스름돈의 개수를 비교할 때, 현재 거스름돈이 저장된 최소 거스름돈의 개수보다 작거나 같으면 known_result[coin_change]에 현재 거스름돈의 list을 저장한다.
마지막으로 min_coins, counter, min_coins_list를 반환하며 끝낸다.

4.결과화면

1) change_money_list

```
C:₩Users₩HyunTaek₩PycharmProjects₩week04₩venv₩Scripts₩python.exe C:/Users/HyunTaek/PycharmProjects/week04/change_money_list.py
거스름돈: 63
(3, 33160648, [21, 21, 21])
Time:  64.912

Process finished with exit code 0
```

2) change_money_list_test

```
True [21, 21] 42
True [21, 21, 1] 43
True [1, 1, 21, 21] 44
True [10, 10, 25] 45
True [25, 21] 46
True [5, 21, 21] 47
True [5, 21, 21, 1] 48
True [25, 21, 1, 1, 1] 49


Ran 1 test in 4.281s
True [25, 25] 50

OK

Process finished with exit code 0
```

3) change_money_saved_list

```
C:₩Users₩HyunTaek₩PycharmProjects₩week04₩venv₩Scripts₩python.exe C:/Users/HyunTaek/PycharmProjects/week04/change_money_saved_list.py
거스름돈: 45
(3, 154, [25, 10, 10])
Time:  0.000

Process finished with exit code 0
```

4) change_money_saved_list_test

```
True [21, 21, 25, 25] 92
True [21, 21, 21, 5, 25] 93
True [21, 21, 21, 21, 10] 94
True [10, 10, 25, 25, 25] 95
True [25, 25, 25, 21] 96
True [25, 25, 25, 21, 1] 97


Ran 3 tests in 0.109s

True [21, 21, 21, 10, 25] 98
OK
True [10, 1, 25, 21, 21, 21] 99
True [25, 25, 25, 25] 100

Process finished with exit code 0
```

1 개요

다양한 알고리즘은 같은 문제를 해결하더라도 시간 복잡도 혹은 공간 복잡도를 낮추어 성능을 올리는데에 목적이 있다. 이러한 알고리즘의 세계에서 중심이 되는 철학은 '큰 문제는 작은 문제로 나눌 것'이다. 직관적으로 해결되거나 어느정도 감당할 수 있을 정도의 수준으로 문제를 나누고 작은 해답을 합쳐 큰 문제에 대한 해답을 구해낸다. 앞으로 우리가 배울 다양한 알고리즘은 이 철학에서 크게 벗어나지 않을 것이다.

재귀는 함수 혹은 클래스 등이 자기 자신을 다시 호출하는 것을 말한다. 대게 콜스택이 지나치게 증가하여 좋은 프로그래밍 기법이라고 이야기되지 않지만 가독성이나 구현의 편의성을 위하여 성능을 포기할 때가 있다. 특히 피보나치 수열이나 하노이의 탑을 구현하는 문제는 많은 프로그래밍 교재에서 재귀를 연습할 때 다루어진다.

재귀적 기법으로 프로그램을 구현할 때에는 '종료 조건'과 '종료 조건으로 가는 식'이 필요하다. 이러한 특징은 위에서 이야기한 '큰 문제를 작은 문제로 나누는 철학'과 일치한다. 큰 문제를 '종료 조건으로 가는 식'이라는 작은 문제로 나누어 최종적으로 '종료 조건'이라는 가장 작은 문제로 분할하는 것이다. 이번 실습에서는 피보나치 수열보다 종료 조건이 좀 더 복잡한 '거스름돈 계산' 프로그램을 만들어 본다. 더불어 실행 시간과 함수 호출 횟수를 세아려 얼마나 오래 그리고 얼마나 많이 호출되었는지 측정한다.

결과를 살펴보면 63원을 계산하기 위하여 무려 33,160,648번의 함수 호출이 있었다. 또한 17.499초나 걸리고 있다. 왜 이럴까? 손으로 직접 따져보면 금방 알 수 있으니 해보자. 이유는 '같은 연산'을 여러번 반복하기 때문이다. 예제로 입력한 63을 계산하기 위하여 같은 연산을 최대 5,083,416번 반복하고 있다(궁금하면 코드를 추가하여 계산해보자.).

2.2 Recursion with Memory(Caching)

재귀의 단점으로 지목되는 호출 스택은 피할 수 없으니 무시하더라도 '같은 연산'을 여러번 반복하는 것은 비효율적이다. 컴퓨터가 같은 작업을 하는데에 특화되어 있어도 연산 시간이 오래걸리면 사용자에게도 불편하다. 따라서 같은 연산을 가능한 안 하는 방법이 필요하다. 이럴 때 사용하는 방법이 '메모리 혹은 캐싱'이다. 예를 들어 6원을 반환하기 위하여 5원 1개, 1원 1개를 계산하였다고 하자. 이것을 '저장' 해두었다가 나중에 6원을 계산하려고 할 때에 재계산없이 동전 2개가 필요하다고 저장된 값을 '불러올 수'있다. 이러한 방법을 '메모리 혹은 캐싱'이라고 부른다.

위의 코드를 메모리를 적용하여 다시 적어보면 아래와 같다. `known_result`라고 이름진 변수를 매 함수 호출 인자로 전달해주어 계산했었던 결과를 '저장'한다. 그 결과 같은 연산을 하더라도 월등히 빨라진다.