Number Theory Assignment #5

21011712 박준영

1) generate_superinc_knapsack(n)

```
def generate_superinc_knapsack(n) :
    super = list()
    former_sum = 0

    for i in range (0, n):
        list_int = random.randint(former_sum + 1, former_sum + 3)
        former_sum += list_int
        super.append(list_int)

    mod_m = random.randint(2 * super[n - 1] + 1, 2 * super[n - 1] + 100)

    return super, mod_m
```

< the source code >

Explanation)

I've created a list variable "super" which is a list that will contain the elements of the super increasing integers. In order for the sequence to be "super increasing", the last element of a randomly chosen set must be bigger than the sum of the rest of the set. In other words,

$$w_k = w_1 + w_2 + \ldots + w_{k-1} \quad (\text{for } 1 < k \leq n)$$

which means we will need to keep track of the sum (i.e. $w_1 + w_2 \ldots + w_{k-1}$) which in this code via variable "former_sum".

In the for loop, which will go on for *n* times, random integers are selected from a range of *former_sum + 1* to *former_sum + 3*, because the sequence of the integers must be super increasing. The random integer created is saved in the temporary variable *list_int*, and the data in list_int is added to former_sum, and to the list " *super* ".

After creating the full list, we create a new random integer called *mod_m* which is bigger than 2 * w_n; the last element of the list. The highest range of mod_m is set to $2 * w_n + 100$, which was a big enough number so that *mod_m* would have a more free range.

After creating the list " *super* " and the variable *mod_m*, we return them both.

2) knapsack_genkey(n)

```
def gcd(a, b):
    while(a != 0):
        b, a = a, b % a

    return b


def knapsack_genkey(n):
    sk, mod_m = generate_superinc_knapsack(n)
    pk = []
    #mod_r, mod_m이 소수

    for i in range(2, mod_m):
        if (gcd(i, mod_m) == 1) :
            mod_r = i                      #if mod_m and i are relative prime, mod_r = i

    for i in range(0, n):
        pk.append((sk[i] * mod_r) % mod_m)

    sk.append(mod_m)
    sk.append(mod_r)
    return sk, pk
```

< the source code >

Explanation)

First, we need to get the returned data from *generate_superinc_knapsack(n)* and save them into variables *sk* and *mod_m*. Variable *sk* will be the private key, and is equal to the "super" list. Next is the list " *pk* " which will be the public key.

In the for loop, it checks for an integer such that *gcd(i, mod_m) = 1*, and variable *i* will be the relative prime which is saved in another variable named *mod_m*. Then we calculate the public key elements by multiplying every *sk* element by *mod_r* and applying the modular arithmetic to the multiplication. Then we add the number to our list " *pk* ". After we complete creating the public key, we add variables *mod_m* and *mod_r* at the end of our private key so that we can use the data in other later functions.

When the computation is over, the function returns the private key " *sk* " and public key " *pk* "

3) knapsack_encrypt(p, pk)

```
def knapsack_encrypt(p, pk):
    enc_msg_int = format(p, "b")
    string_temp = str(enc_msg_int)

    enc_msg = []
    cipher_c = 0

    if (len(pk) > len(string_temp)):
        contrast = len(pk) - len(string_temp)
        for i in range(0, contrast):
            enc_msg.append(0)

        for i in range(0, len(string_temp)):
            enc_msg.append(int(string_temp[i]))


    for i in range(0, len(pk)):
        cipher_c += pk[i] * enc_msg[i]

    return cipher_c
```

< the source code >

Explanation)

First, we need to change the message "*p*" into binary. We use the *format* function to convert the message into binary conveniently, and then save the data to the variable " *enc_msg_int* ". Then I've made a temporary string variable that saves the string version of " *enc_msg_int* ", so that I could comfortably manipulate the binary data. Then I've made a list variable " *enc_msg* " to store the binary values of *p* used to get the ciphered integer " *c* ". The variable " *cipher_c* " will be used to get the actual encrypted integer, and the default value is set to 0.

If the number of elements is the public key is bigger than the length of the binary string, the binary string must be calculated with more 0s in front of it.

Example) 123 = $(1111011)_2$, and n = 10. So, $(1111011)_2$ must be treated as $(0001111011)_2$

The difference between the two lengths is stored in the variable " *contrast* ", and for the size of *contrast*, 0 is added to the front of the list *enc_msg*. Then, the actual binary value of *p* is added to the string.

For all values in the list *pk* and *enc_msg*, they are multiplied and added to the variable *cipher_c*, which is the encrypted integer. *Cipher_c* is then returned by the function.

4) knapsack_decrypt(c, sk)

```
def knapsack_decrypt(c, sk):
    n = len(sk) - 2

    index_list = []
    p = 0

    inverse_mod_r = pow(sk[n + 1], -1, sk[n])
    subset_c = (c * inverse_mod_r) % sk[n]

    subset_c -= sk[n - 1]
    index_list.append(n)

    i = 0
    while (subset_c != 0):

        if (subset_c <= sk[i]):


            if (subset_c == sk[i]):
                index_list.append(i + 1)
                subset_c -= sk[i]

            elif (subset_c < sk[i]):
                index_list.append(i)
                subset_c -= sk[i - 1]

            i = 0

        if (i >= n):
            break

        i += 1

    for i in range(0, len(index_list)):
        p += 2**(index_list[0] - index_list[i])

    return p
```

< the source code >

Explanation)

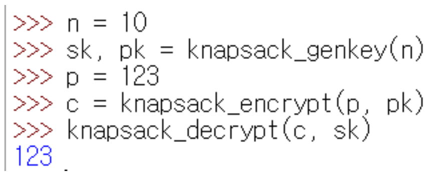
First, I define variable *n* as *len(sk) – 2* since *sk* includes *mod_r* and *mod_m*. Then I declared a list variable called " *index_list* " which is the list that contains the powers of 2 needed to recreate the original message *p*. Because we need to recreate *p*, I've defined the variable *p* and set its default value as 0. The variable " *inverse_mod_r* " is literally the multiplicative inverse of variable *mod_r* which sits at the nth position in *sk*. I've used the *pow* function which is a built-in function that gets us the multiplicative inverse. Then I have calculated the variable *subset_c* which is needed to check what powers of 2 we need to recreate the original message *p*.

I've calculated the first *subset_c* since in every case the *subset_c* is always bigger than the last element in *sk*. Then I have added, or appended the integer *n* which points to the position of where that value of *sk[n − 1]* was. Now, in the while loop we do the same thing as I have done with the first time, constantly checking if it can find the largest element that is smaller than or equal to *subset_c*. If it is smaller than or equal to *subset_c*, it subtracts that value from *subset_c* and adds that position to the list index_list. One thing to note is that when the value of *subset_c* is smaller than the value of *sk* in the ith position, we add the value of *sk[i − 1]*, whereas if the value of *subset_c* is equal to the value of *sk* in the ith position, we add the value of *sk[i]*.

After repeating this process we get a full list of powers of 2. Then we use these elements to the power of 2 and add them all to the variable *p*.

Then the function returns the full original integer *p*.

Results)

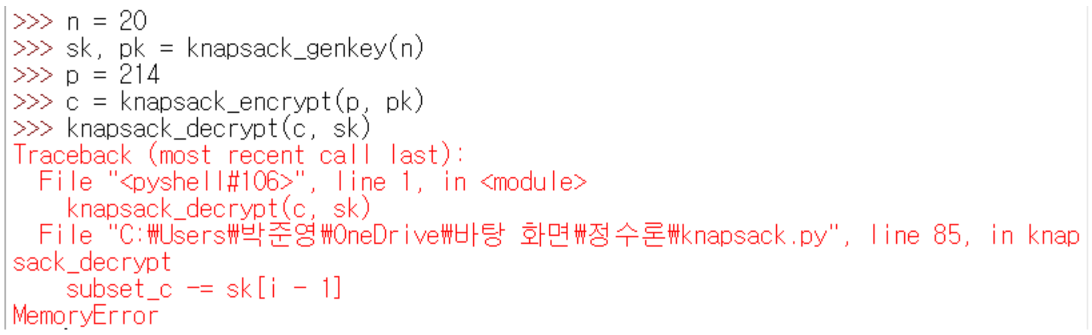


```
>>> n = 10
>>> sk, pk = knapsack_genkey(n)
>>> p = 123
>>> c = knapsack_encrypt(p, pk)
>>> knapsack_decrypt(c, sk)
123
```

(when n = 10, p = 123)

After this example I have tried variously with different n and different p but as n became bigger it always had a memory failure.

I acknowledge how it may seem, as it may seem that I am giving an excuse. So here is a screenshot of another example.



```
>>> n = 20
>>> sk, pk = knapsack_genkey(n)
>>> p = 214
>>> c = knapsack_encrypt(p, pk)
>>> knapsack_decrypt(c, sk)
Traceback (most recent call last):
  File "<pyshell#106>", line 1, in <module>
    knapsack_decrypt(c, sk)
  File "C:₩Users₩박준영₩OneDrive₩바탕 화면₩정수론₩knapsack.py", line 85, in knap
sack_decrypt
    subset_c -= sk[i - 1]
MemoryError
```