Option 2 Refactor and Analyse code

Gia Bao Hoang – a1814824

I. Refractor:

1. GoodHours:

Before: Revision 233

After: Revision 293

Number	Original code	Refractored

1	`int bh = stoi(beforeTime.substr(0,2));`
	`int bm = stoi(beforeTime.substr(3,2));`
	
	`int ah = stoi(afterTime.substr(0,2));`
	`int am = stoi(afterTime.substr(3,2));`
		`int bhour = stoi(beforeTime.substr(0,2));`
		`int bminute = stoi(beforeTime.substr(3,2));`
		
		`int ahour = stoi(afterTime.substr(0,2));`
		`int aminute = stoi(afterTime.substr(3,2));`

2	`string time = "";`
	`if (bh < 10) time += "0";`
	`time += to_string(bh);`
	`If (bm < 10) time += "0";`
	`time += to_string(bm);`
		`string timeToString(int t)`
		`{`
		`  string zero = "";`
		`  if (t < 10) zero += "0";`
		`  return zero + to_string(t);`
		`}`
		
		`string time = timeToString(bhour) +`
		`timeToString(bminute);`

3	`string astr = t.substr(0,i);`
	`int a = 1;`
	`for (int ai = 0; ai < i; ai++) {`
	`  a = a * (astr[ai] - '0');`
	`}`
	
	`string bstr = t.substr(i,4-i);`
	`int b = 1;`
	`for (int bi = 0; bi < 4-i; bi++) {`
	`  b = b * (bstr[bi] - '0');`
	`}`
		`int multiplyString(string s)`
		`{`
		`  int result = 1;`
		`  for (int i = 0; i < s.length(); i++) {`
		`  result *= (s[i] - '0');`
		`  }`
		`  return result;`
		`}`
		
		`a = multiplyString(t.substr(0, i));`
		`b = multiplyString(t.substr(i, 4-i));`

4	`bm++;`
	`if (bm == 60) {`
	`  bm = 0;`
	`  bh++;`
	`  if (bh == 24) {`
	`  bh = 0;`
	`  }`
	`}`
		`bminute++;`
		`if (bminute == 60) {`
		`  bminute = 0;`
		`  bhour++;`
		`}`
		`if (bhour == 24)`
		`  bhour = 0;`

Discussion: (based on Number)

1. Update the name of variable for easier understanding.
2. Move the repeated code into a separate function that is only responsible to convert the given time from integer type to string type (timeToString). This simplifies the main function (howMany).
3. Move the repeated code into a separate function that is only responsible to calculate the product of each digit in each string (multiplyString). This simplifies the addition function that check whether a time is good (isGood).
4. Collapse the nested if statement into two if statements to preserve the functionality of the code segment and enhance readability.

Reflection:

- Naming variables is very important since the name needs to be both short and clarified itself.
- The name of the function should clarify its process. Repeated code can be move into separate additional functions to both simplified the code and enhance readability.
- Be careful with the if statement and its structured. It is best to avoid nested if statements since it can make the code become more confusing and harder to keep track with the logic workflow.

2. SimpleDuplicateRemover:

Before: Revision 209

After: Revision 294

Number	Original code	Refractored

1	`for (int i = n-1; i >=0; i--) {`
	`  if (!unique.count(sequence[i])) {`
	`  temp.push_back(sequence[i]);`
	`  unique.insert(sequence[i]);`
	`  }`
	`}`
	
	`for (int i = temp.size()-1; i >= 0; i--) {`
	`  result.push_back(temp[i]);`
	`}`
		`for (int i = n-1; i >=0; i--) {`
		`  if (unique.count(sequence[i]))`
		`  continue;`
		
		`  result.insert(result.begin(), sequence[i]);`
		`  unique.insert(sequence[i]);`
		`}`

Discussion:

1. Refractor the code segment and simplify the logic of the code. In the original version, the vector temp is the final vector result but in reverse order. By implementing the insert function of vector, we can remove the vector temp and remove the second for loop that iterate the vector temp in reverse order. As a result, the code is simplified.

Reflection:

- The reason I did two for loops at first was to divide the code into different sections, each with a specific purpose. However, doing that also make the code more complicated than it should be. And as I become more proficient with C++, I learn that the insert function would be sufficient to perform what I need and make the code much more concise.
- This problem taught me to make the most out of the tools I have at my hand. By utilizing the insert function of the vector, I could increase the processing time as well as simplify the code while preserving its functionality.

II. Analyse:

1. MonsterValley2

Solution 1: Mine	Strategy: Recursion + Brute Force Efficiency: Low Readability: Medium

```
#include <string>
#include <vector>
#include <iostream>
#include <algorithm>
using namespace std;


class MonstersValley2
{
public:
    int minimumPrice(vector<int> dread, vector<int> price)
    {
        return minimumPriceHelper(dread, price, 0, 0, 0);
    }

    int minimumPriceHelper(vector<int> dread, vector<int> price, int pos, long
t_dread, int t_price)
    {
        if (pos >= dread.size()) return t_price;

        if (t_dread < dread[pos]) {
            t_dread += dread[pos];
            t_price += price[pos];
            return minimumPriceHelper(dread, price, pos+1, t_dread, t_price);
        }

        return min(minimumPriceHelper(dread, price, pos+1, t_dread, t_price),
minimumPriceHelper(dread, price, pos+1, t_dread + dread[pos], t_price + price[pos]));
    }
};
```

Solution 2 : URL	Strategy: Dynamic Programming Efficiency: High Readability: Low

```
#include <iostream>
#include <string>
#include <string.h>
#include <vector>

using namespace std;

class MonstersValley2{
    public:
    long long dp[30][50];
    long long Max(long long a, long long b)
```

```
    {
        if (a > b) return a;
        return b;
    }
    int minimumPrice(vector<int> dread, vector<int> price){
        int n, MaxP, i, j;
        n = dread.size();
        MaxP = 2 * n;
        memset(dp, 0, sizeof(dp));

        for (i = price[0]; i <= MaxP; i++) dp[0][i] = (long long) dread[0];

        for (i = 1; i < n; i++)
            for (j = 1; j <= MaxP; j++) {
                if (j > price[i] && dp[i-1][j-price[i]] > 0)
                    dp[i][j] = dp[i-1][j-price[i]] + (long long) dread[i];

                if ((long long) dread[i] <= dp[i-1][j]) dp[i][j] = Max(dp[i-1][j],
dp[i][j]);

                if (j > 1) dp[i][j] = Max(dp[i][j-1], dp[i][j]);
            }

        i = 1;
        while(dp[n-1][i] == 0) i++;
        return i;
    }
};
```

Solution 3: URL

Strategy: Dynamic Programming + Greedy

Efficiency: High

Readability: Medium

```
#include <iostream>
#include <vector>
#include <algorithm>

using namespace std;

typedef long long ll;
ll pricepower[30][50];

class MonstersValley2 {
    public:
    int minimumPrice(vector<int> dread, vector<int> price) {
        int dsz = dread.size();
        pricepower[0][price[0]] = dread[0];
        for (int i = 1; i < dsz; i++) {
            for (int j = 45; j >= -1; j--) {
                if (pricepower[i-1][j] >= dread[i]) {
                    pricepower[i][j] = pricepower[i-1][j];
```

```
                }
            }

            for (int j = 45; j >= -1; j--) {
                if (pricepower[i-1][j] != 0) {
                    pricepower[i][j+price[i]] = max(pricepower[i][j+price[i]],
pricepower[i-1][j]+dread[i]);
                }
            }
        }

        int res = 0;
        for (int i = 0; i< 45; i++) {
            if (pricepower[dsz-1][i] != 0) {
                res = i;
                break;
            }
        }
        return res;
    }
};
```

Discussion:

Efficiency:

- The input of the problem has a small size, from 1 to 20 elements. Hence, readability is more important than efficiency.
- Solution 1 is a brute force solution using recursion to generate all possible permutation and compare them to get the smallest solution. With each encounter, except when we must bribe the monster to get through, we could either choose to bribe or not bribe it. In another words, each encounter would have two outcomes, making the time complexity of the solution is O(2^n). As the problem is small, it is still sufficient.
- Both solution 2 and 3 are DP approaches using tabulation. In both solutions, we have a 2D – vector, where the element at [i][j] would represent the maximum total dread that can be obtained by spending j coins for the first i encounters. Each solution is implemented in slightly different way. While solution 2 is more efficient, solution 3 is more readable.
- In solution 2, we use a nested for loop to iterate through the 2D vector dp. Element dp[i][j] is first calculated by add on the dread of the current encounter if we have enough money. Then it will be evaluated with dp[i-1][j] (same amount of coins, previous encounter) and dp[i][j-1] (1 coin less, same encounter). This to maximize the total dread while minimize the total cost. The time complexity is O(n*2n) or O(n^2).
- In solution 3, the outer for loop is to iterate through each encounter (i). Then the first inner for loop perform a simple evaluation: at the same encounter, if the total dread with j-1 coin is bigger or equal to the current encounter dread, then the total dread with j coin would be the same value. Then at the second inner for loop, we only go at the element with value (meaning enough money to bribe and gain some dread). Once again, we make sure to get the maximum amount of dread with the minimum number of coins. In this case, the time

complexity is O(n*(45+45)) = O(90n), which seems to be faster than solution 2. However, since n is from 1 to 20, O(90n) is bigger than O(n^2) or even O(n*2n).

Readability:

- All solutions do not have any comments to clarify the approach. Some basic comments can help to increase readability.
- Solution 1 is very easy to understand, and the code is clear and concise. It would be better to change the name of the function minimumPriceHelper to generateGetMin to better explain its functionality.
- Solution 2 is a bit messy and hard to understand. It would be idea to refractor the code and give the variables better names.
- Despite having the same strategy with solution 2, solution 3 is much easier to read as the code is segmented into different section with clear functionality. The variable name also clarified itself. However, it would be more ideal to use a variable with a name instead of the integer 45 for both inner loops, as this number is not mentioned in the question and making the solution confusing.

Reflection:

- With one problem, there are multiple approaches, and there are multiple implementations within one approach. I have learned how to better recognize a DP problem as well as its potential in solving problems.
- In addition, it is always a good idea to check the size and the constraints of the problem to derive a suitable solution. With small size problem, brute force is a great choice, but as the size increase, a better solution is required to process the problem more efficiently.
- Comments and variable names are essential for the problem to clarify itself. Segment your code into different section is good way to separate each smaller functionality of the code, enhance its readability.
- DP approach is a powerful tool if we can master it.