COT5405 - Analys of Algorithms
Homework 2

Qinxuan Shi
UFID: 83518162

1 Problem 1: Weighted approximate common substring

1.1 Pseudo-code of the Algorithm

Algorithm 1 Weighted approximate common substring

Input: String s1, String s2, dp[][]

Output:

We define that the best common substring means that the substring has the heaviest total weight.
Assume that i is the rightmost index of the character in substring s1' of s1, and j is the rightmost index of the character in substring s2' of s2. And we define dp[i][j] to represent the weight of best common substring between s1' and s2'.

```
for i= 0 to s1.length do
    if s2[0] exists in s1 then
        dp[i][0] ← W_{s2[0]}
    else
        dp[i][0] ← 0
    end if
end for
for j= 0 to s2.length do
    if s1[0] exists in s2 then
        dp[0][j] ← W_{s1[0]}
    else
        dp[0][j] ← 0
    end if
end for
for i= 0 to s1.length do
    for j= 0 to s2.length do
        if s1[i] = s2[j] then
            dp[i][j] ← Maximum_weight(i,j)
        else
            dp[i][j] ← dp[i-1][j-1]
        end if
    end for
end for
return max{dp[i][j]}.
```

Algorithm 2 Maximum_weight(i, j)

Input: String s1, String s2, int[] weight, int i, int j, dp[]

Output:

We now can get the substring s1' ended with index i and substring s2' ended with index j. Then we calculate the maximum subarray of them after transform them into weight int[] w.

```
if t=0 then
    dp[0] ← w[0]
end if
for i = 1 to w.length do
    if dp[i - 1] >0 then
        dp[i] ← w[i] + dp[i - 1]
    else
        dp[i] ← w[i]
    end if
end for
return max{dp[i]}
```

1.2 Proof of the algorithm's correctness

Firstly, our goal is to get the best common substring. so we use OPT(i,j) to represent the best common substring s, i means the index of the rightmost character of s in the first string s1 and j means the index of the rightmost

character of s in the second string s2.

Since when two characters are different, we have to minus δ to the total weight, it is clear that the best common substring has the property that $s1[i] = s2[j]$. Then, we are going to change the character into an array of weight with the length of $min\{s1(0,i).length, s2(0,j).length\}$ (because both substring of s1 and s2 are of the same length, the maximum length of the best common substring is the length of the shorter substring).

Bellman equation:

$$\text{OPT(i, j)} = \begin{cases} Maximum\_weight(i,j), & \text{if s1[i] = s2[j]} \\ OPT(i-1, j-1), & \text{if s1[i] not euqal to s2[j]} \end{cases}$$

Then, we are going to explain function Maximum_weight. After transforming the english characters into numbers with the regulation which is starting from the last character of substring (index i in s1 and index j in s2), if these two characters of the substring s1' and substring s2' are the same, the character turns to the weight of the characher, while if the two characters are different, the number in the array of weight becomes -δ.

Now, we get the array of weight, and we are going to find the maximum sum of the array. Exactly, it's the same as the Maximum subarray problem. So we can deal with the problem with the following Bellman equation.

Bellman equation

$$\text{OPT(i)} = \begin{cases} x_1, & \text{if i = 1} \\ max\{x_i, x_i + OPT(i-1)\}, & \text{if i >1} \end{cases}$$

1.3 Algorithm's running time

To the Algorithm Maximum_weight, it is easy to get the complexity by using the KADANE'S algorithm, which is O(n).

According to the pseudo-code, the first for-loop and the second for-loop are going to initialize the dp matrix which is used to memorize the maximum weight of each dp[i][j], so the complexity is O(n).

To the third for-loop, which is used to compute each dp[i][j], we assume that the length of s1 is M and the length of s2 is N, is the complexity of third for-loop is O(MN).It is also easy to know that if we want to find the maximum weight in the dp matrix, it takes O(MN).

To conclude, the total complexity of the algorithm is O(MN), which depends on the length of the two strings.

We could clearly infer from the result picture of 500 points of (MN, time) that $Time = k * MN$, so it is true for the proof that the total complexity of the algorithm is O(MN).

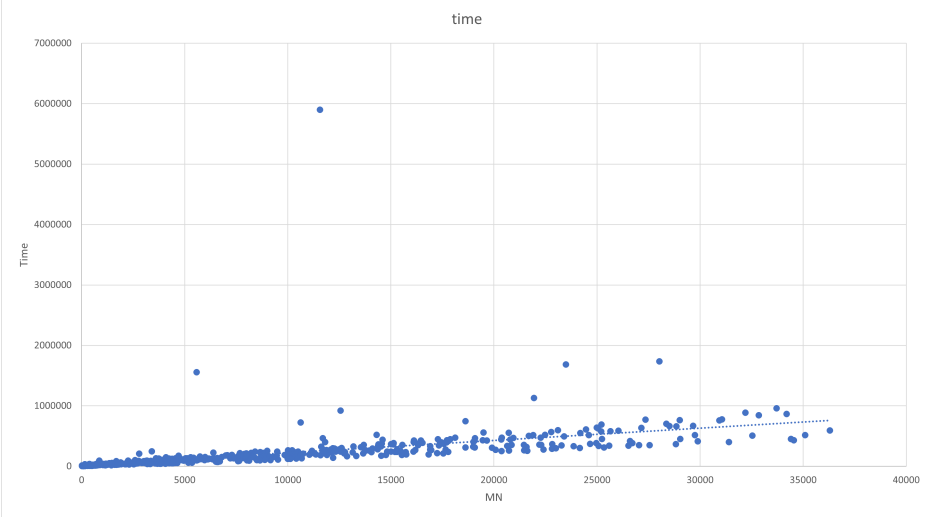

Figure 1:

First experiment is $w_i = 1$ and $\delta = 10$, the last number of the result is the runningtime of milliseconds.

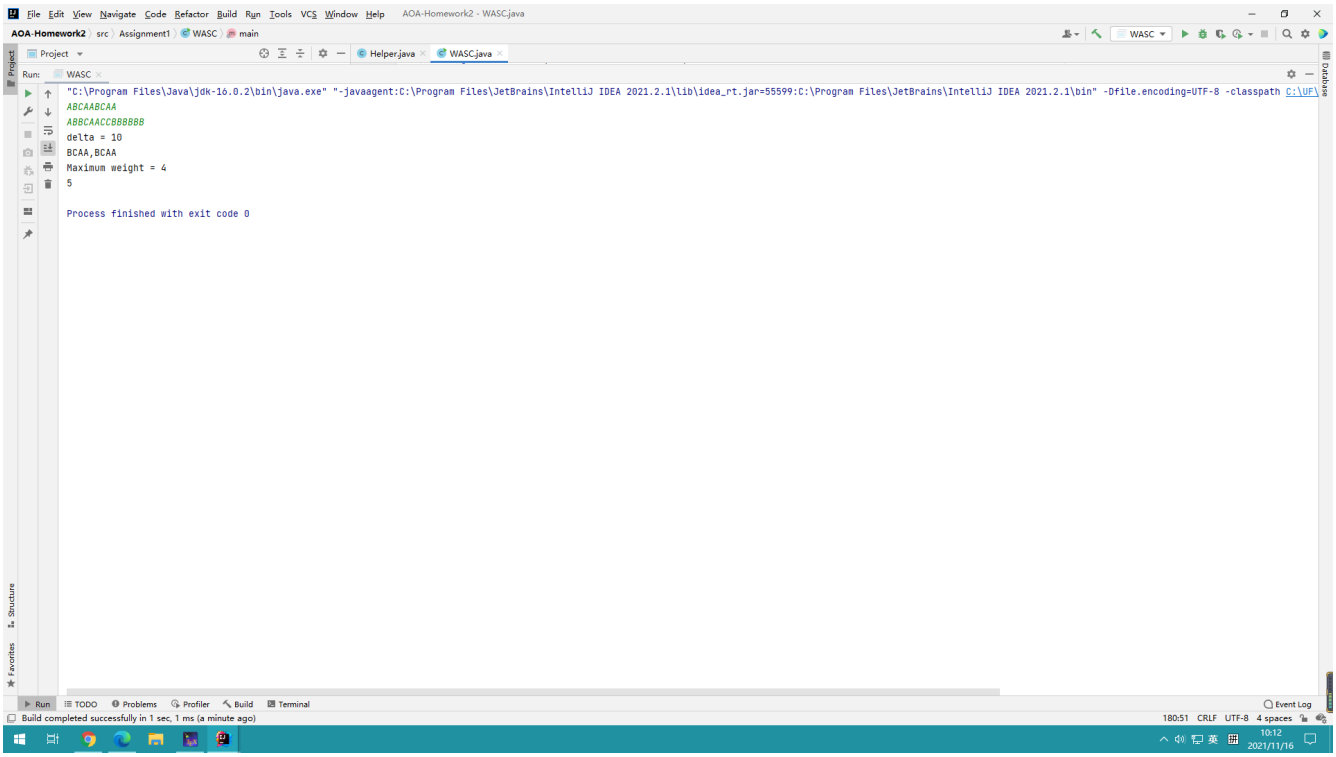

Figure 2:

Here is the ten experiments with different weights and deltas that w_i is proportional to the frequency of the letter in English and δ takes values between the smallest and the largest weight.

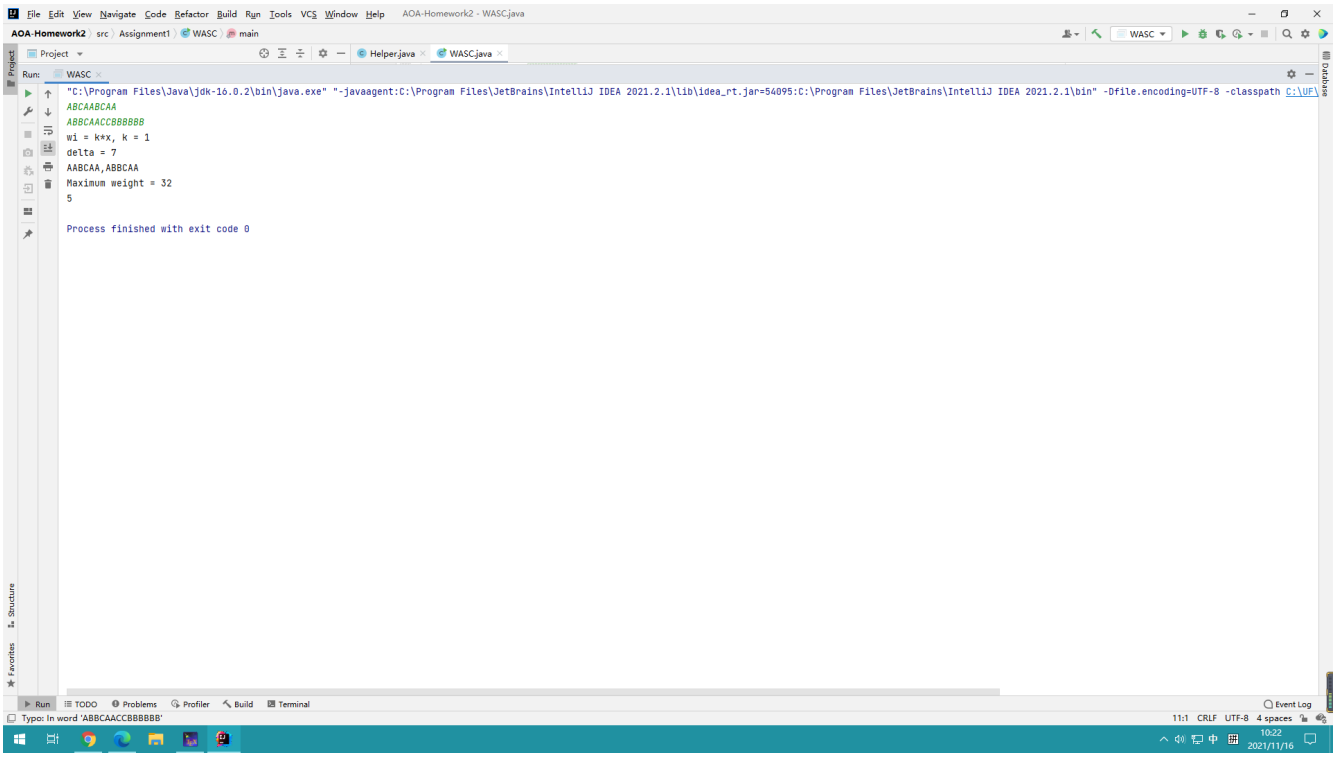

Figure 3:

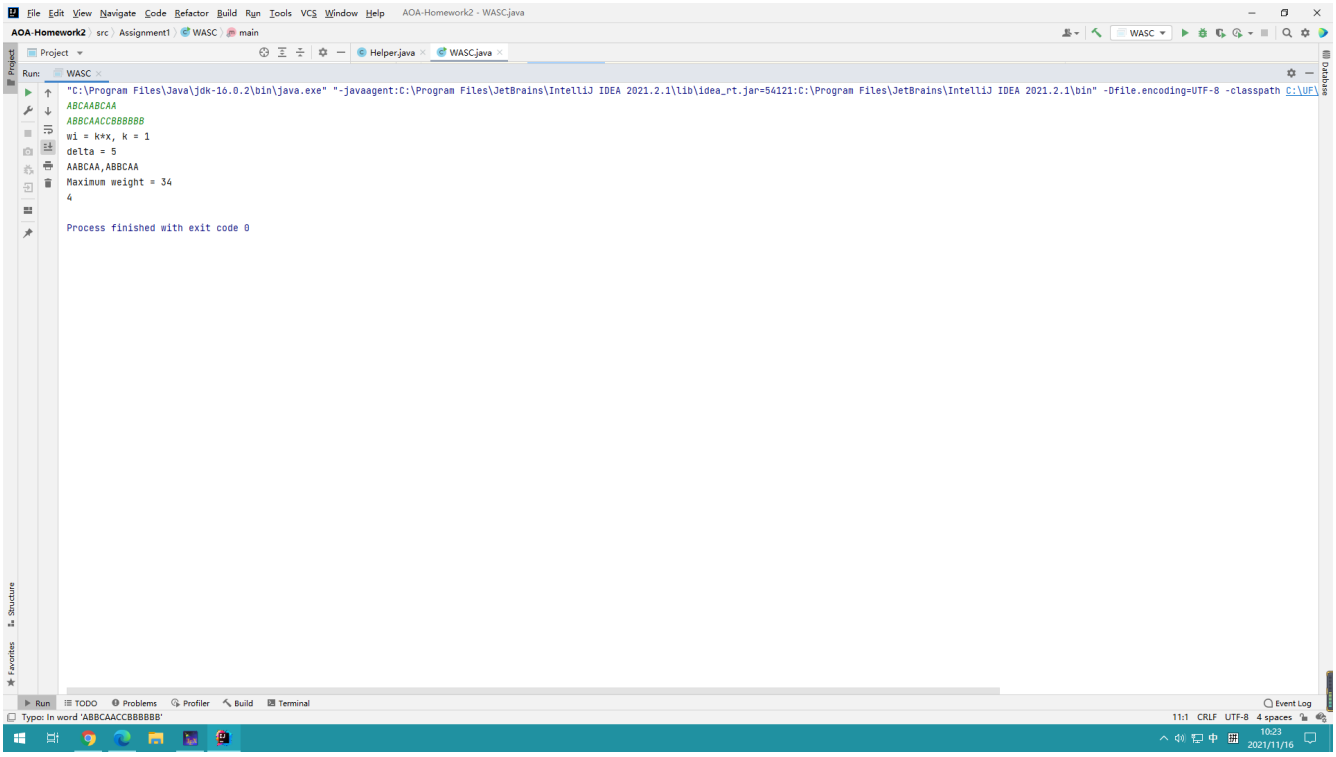

Figure 4:

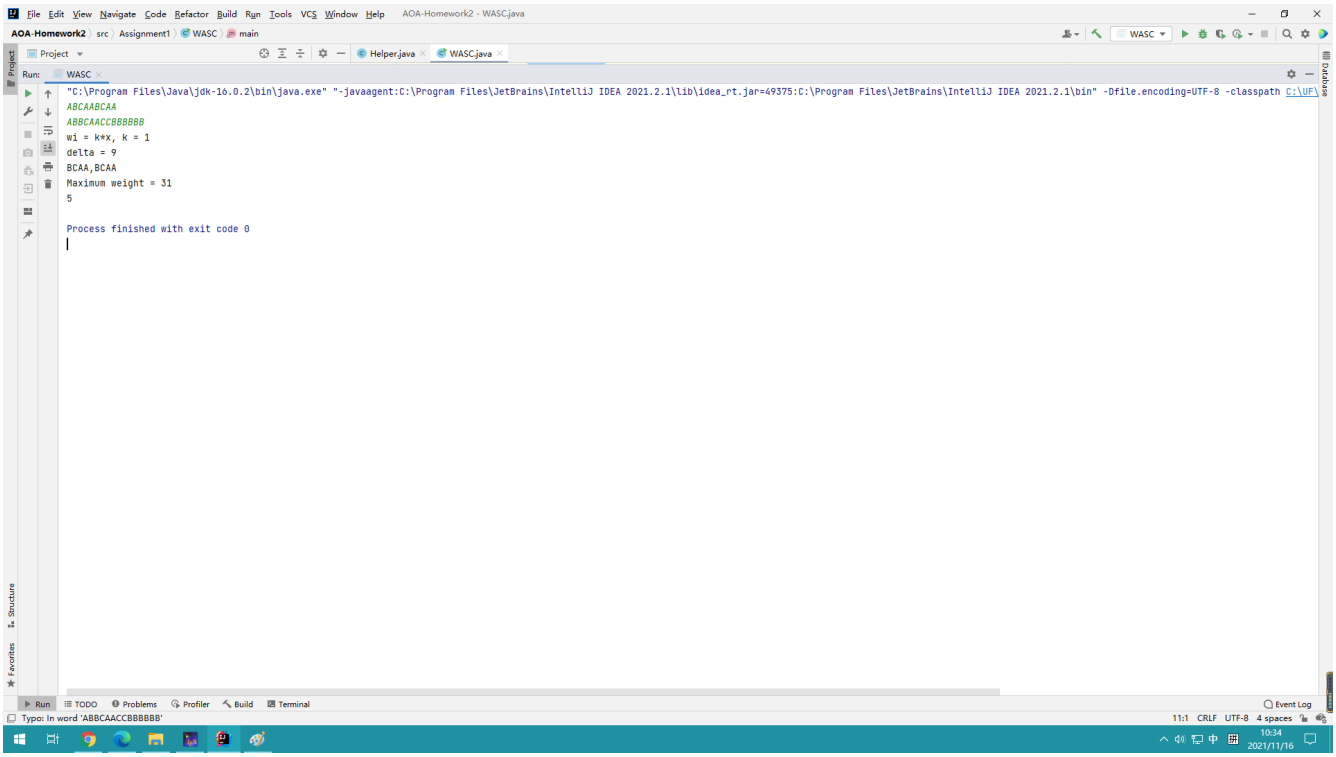

Figure 5:

since the k is 1, there are not so many possible results, so the following results have the k of 2.

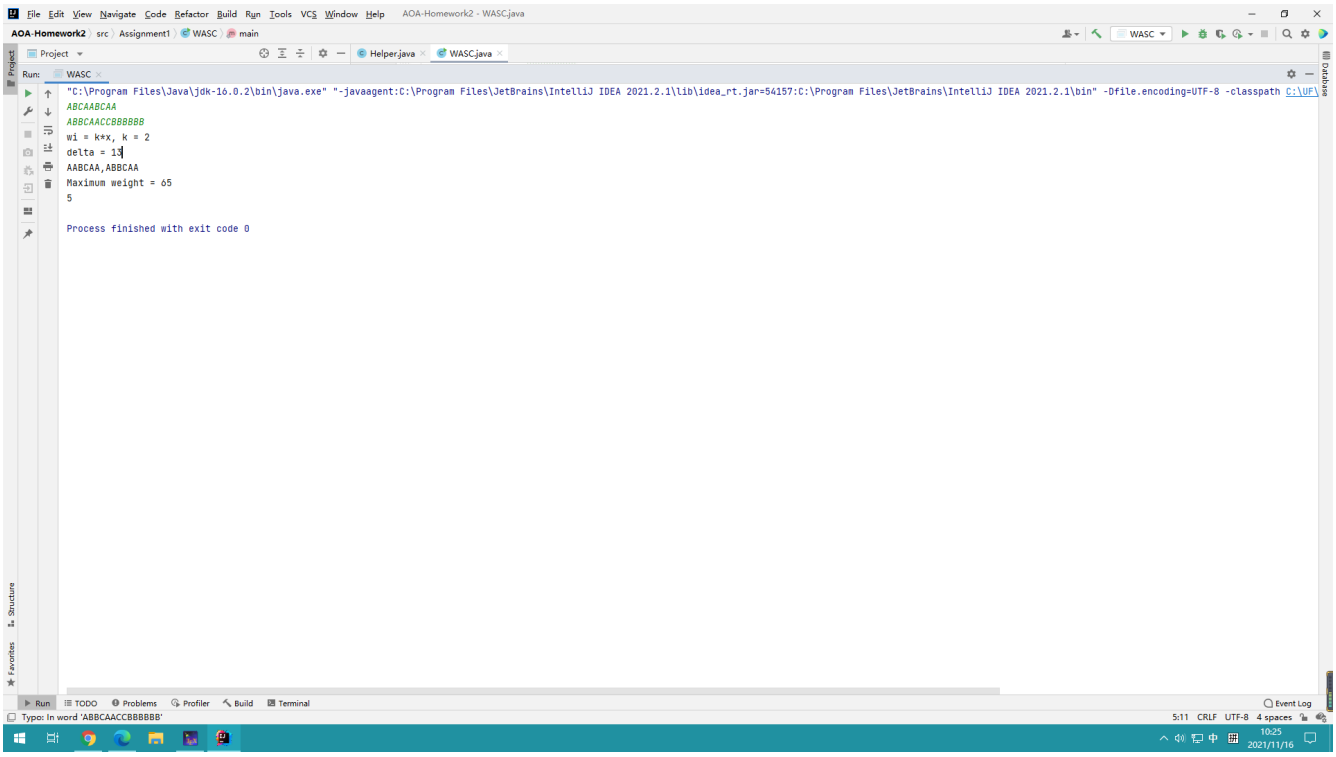

Figure 6:

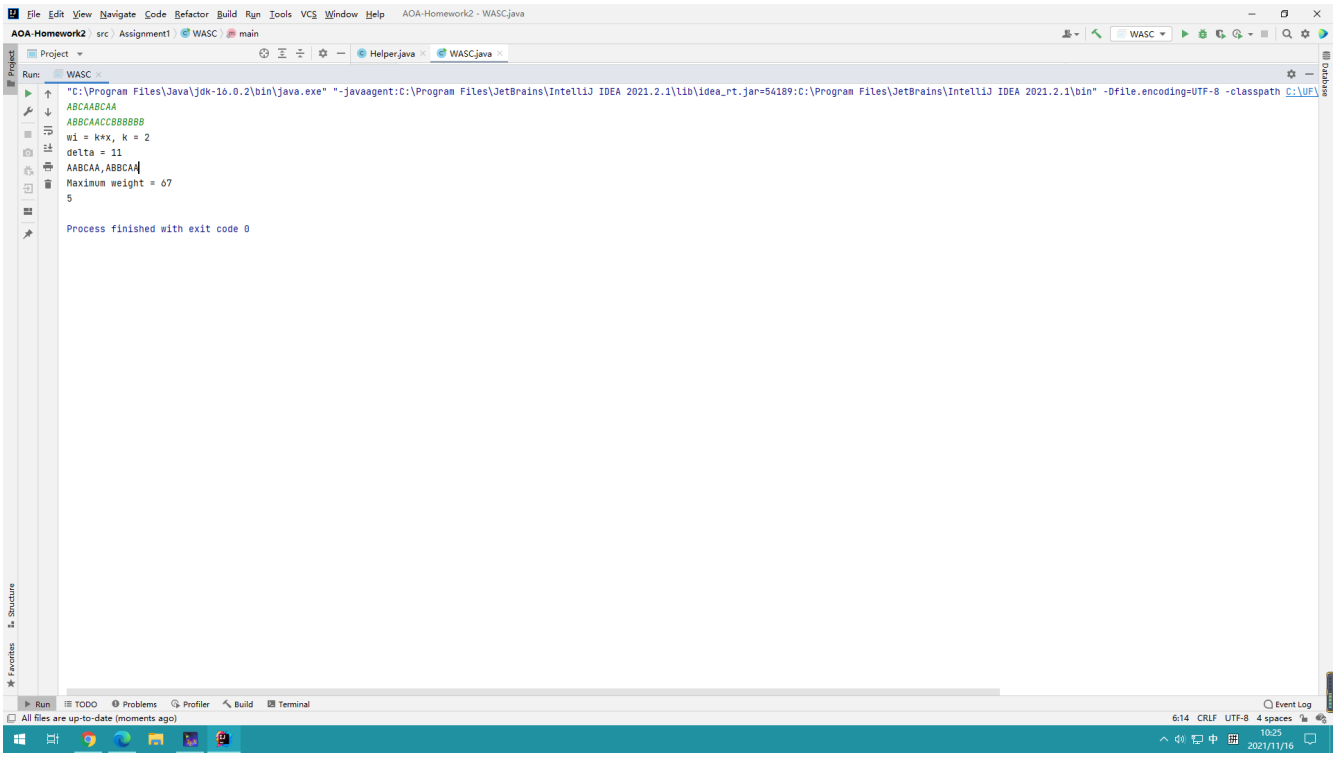

Figure 7:

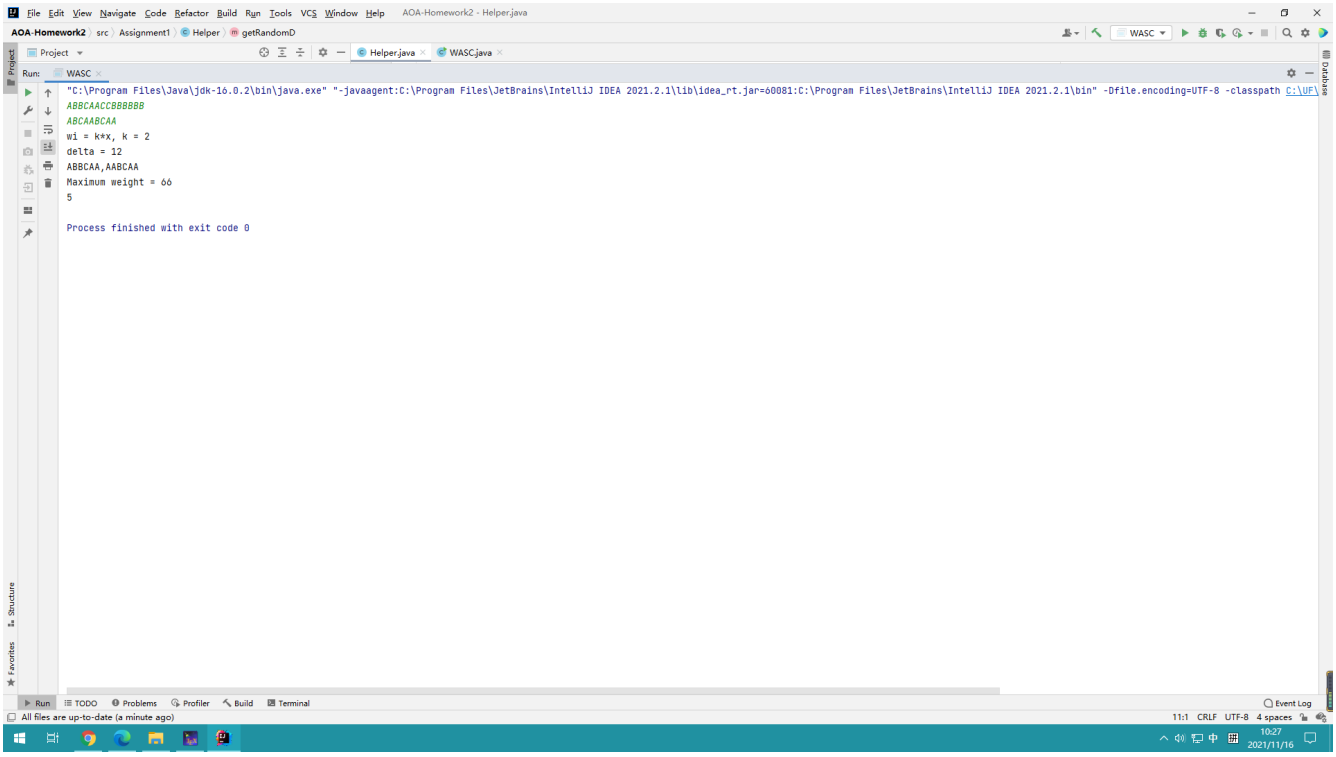

Figure 8:

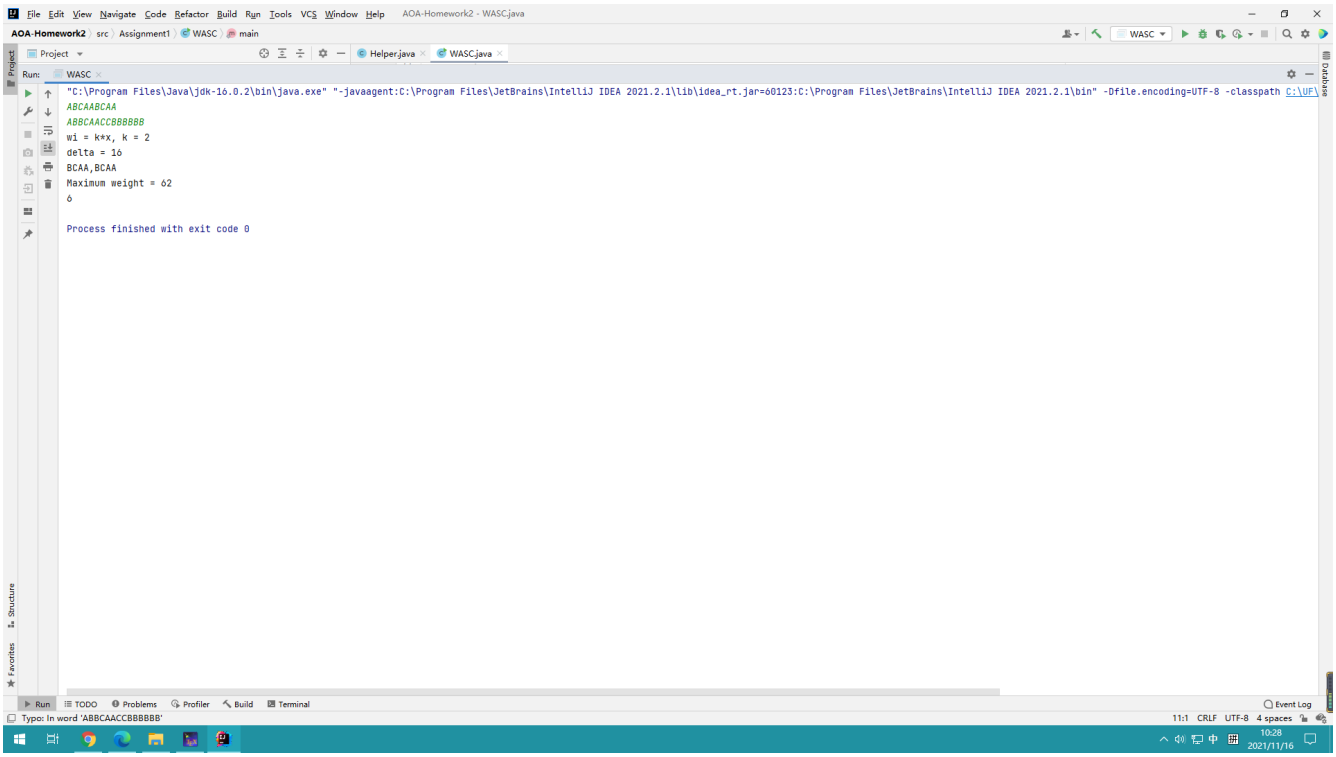

Figure 9:

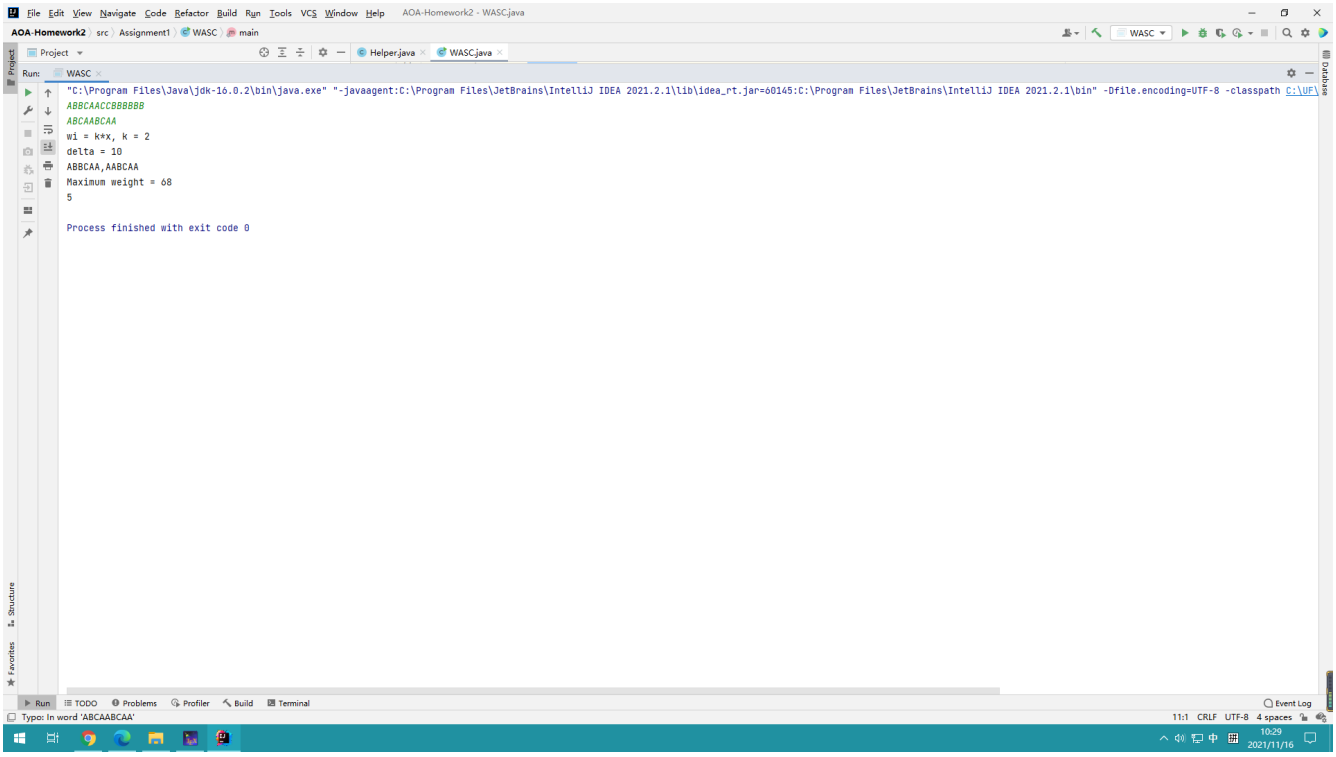

Figure 10:

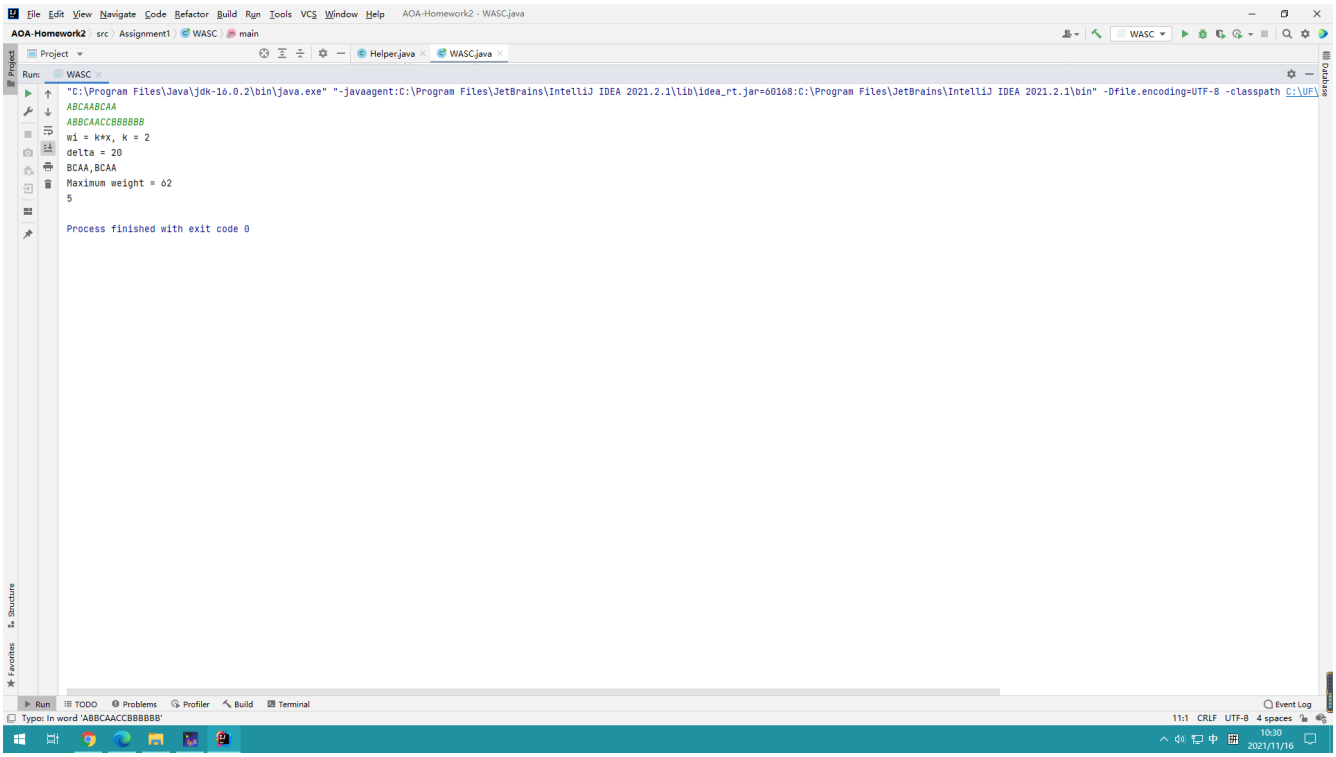

Figure 11:

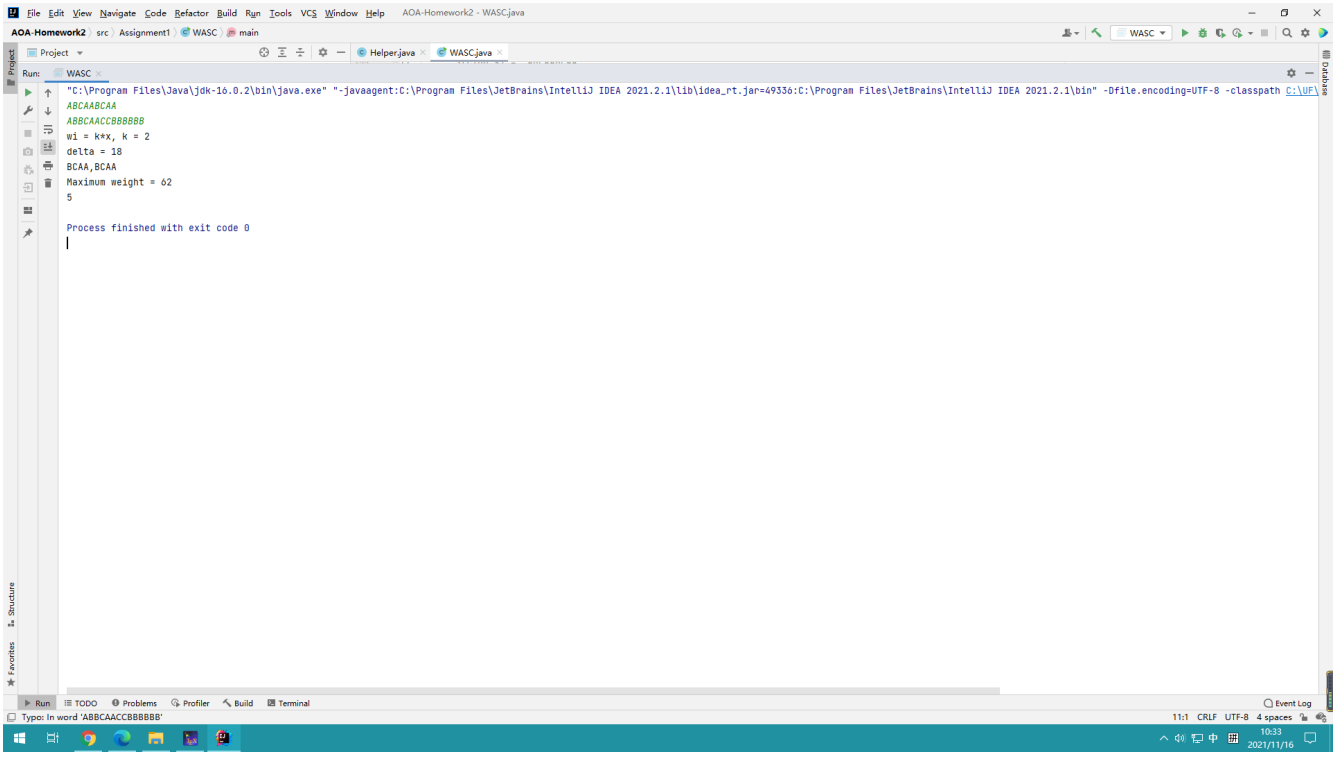

Figure 12:

2 Problem 2: Interval-based constant best approximation

2.1 Pseudo-code of the Algorithm

Algorithm 3 best approximation

Input: a set of points $\{p_1,...,p_n\}$
Output:

```
Initially let M[n] be the array to memorize the minimum cost M[i] for each i, 0≤i≤n.
let M[0] = 0
for j = 1 to n do
    for i = 1 to j do
        compute the least error e_ij for each interval from point p_i to p_j
    end for
end for
for j = 1 to n do
    M[j] = min_{1≤i≤j}(e_ij + δ + M[i-1])
end for
```

2.2 Proof of the algorithm's correctness

Since the goal is to find a partitioning of the interval [1,M] into contiguous intervals such that the error of approximating points in each interval element by the average value of y in the interval is minimized.

Firstly, we can get the equation from the class recording that the minimum error of these points is:

$$E^2 = \sum_{i=1}^{n}(y_i)^2 - \frac{(\sum_{i=1}^{n} y_i)^2}{n}$$

Then, we assume that for the index j, [i,j] is the last interval that we divided, so we can get the bellman equation that relate OPT(j) and OPT(i-1).

Bellman equation:

$$\text{OPT(j)} = \begin{cases} 0, & \text{if j} = 0 \\ min_{1\leq i\leq j}\{e(i,j) + \delta + OPT(i-1)\}, & \text{otherwise} \end{cases}$$

2.3 Algorithm's running time

According to the pseudo-code, if we want to compute the complexity of the Algorithm, we need to consider two parts.

The first part is computing the least error e_{ij} for each p_i to p_j. According to the equation, for each pair of i and j, we need O(n) to compute the sum of y_i^2 and the sum of y, and it only takes O(1) to put them together and get E^2. So to get all e_{ij}, it needs $O(n^3)$.

The second part is the get the M[i] for each i from 1 to n. For each i, we need to compute the minimum cost of the sum of $e(i,j) + \delta + OPT(i-1)$, and it takes O(n) times. So the second part takes $O(n^2)$.

To conclude, the total complexity of the algorithm takes $O(n^3)$.

Here is the memory condition of the program that is solved by using array.

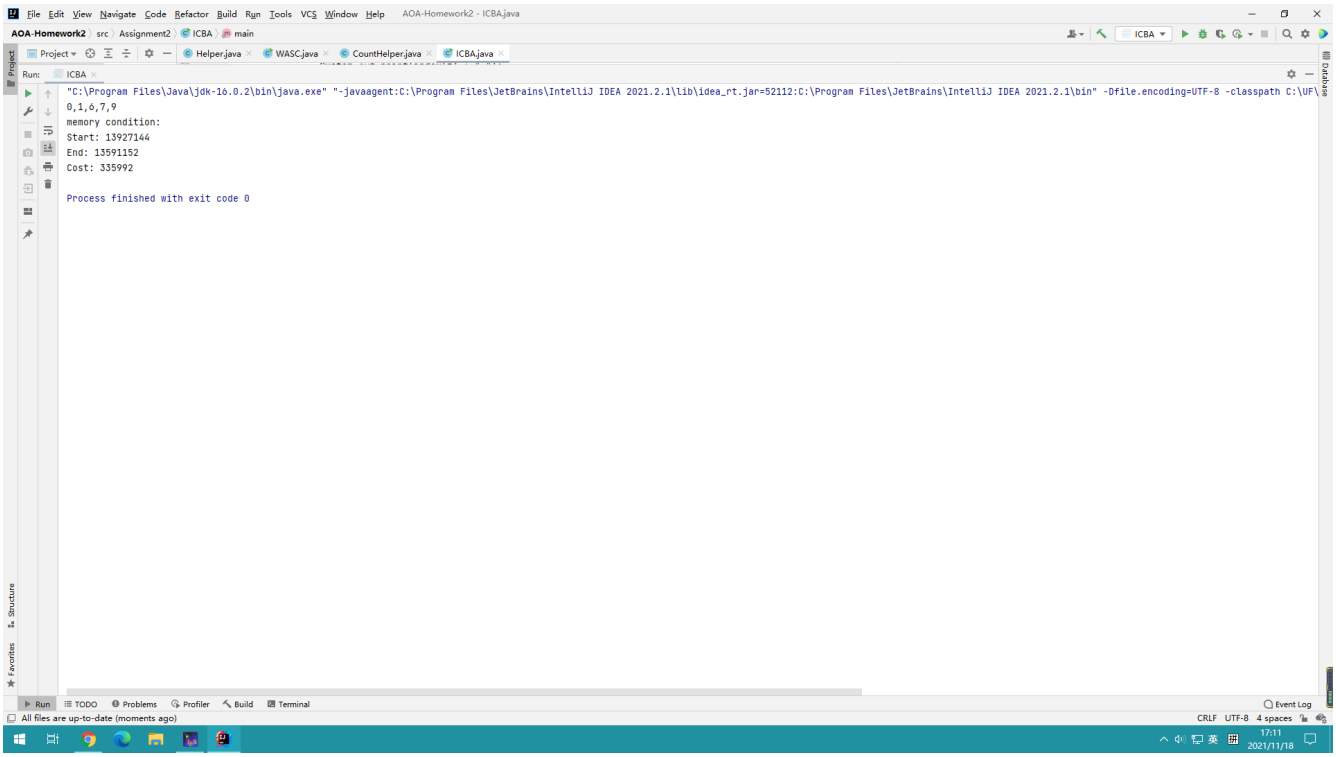

Figure 13:

Here is the memory condition of the program that is solved by using hashmap.

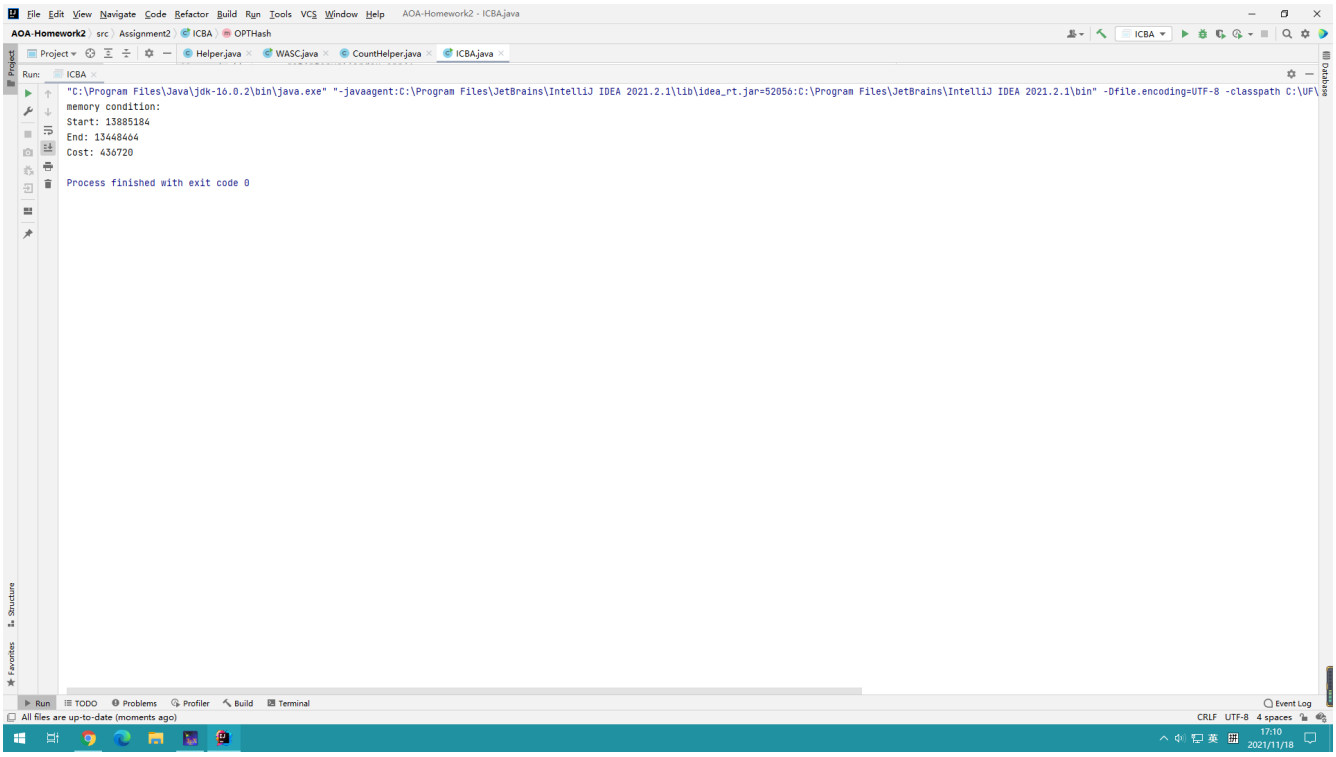

Figure 14: