Problem Set #1 (Algorithms)

Department: 컴퓨터정보공학부

Student ID: 2015722025

Student Name: 정용훈

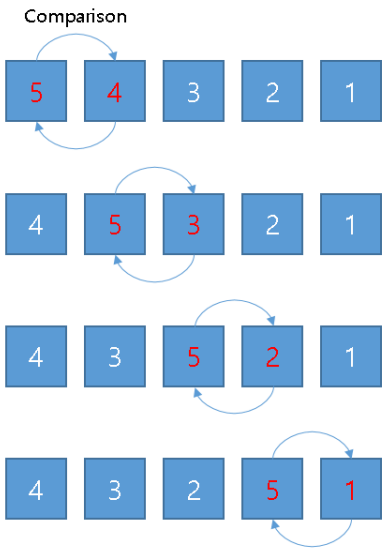
1. Bubble sort pseudocode

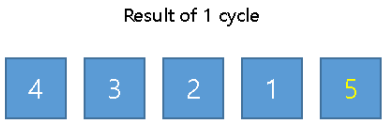
```
for i ← (number of item)-1 to 0 update i ← i-1   //Save number of item and repeat until i is zero
{
   j ← 0 & flag ← 0                              //Save 0 to j and initialize flag to 0
   while j<i update j ← j+1                      //Repeat if j is smaller than i
   {
         if A[j] > A[j+1]                        //Practice condition if A[j] is bigger than A[j+1]
         {
            swap A[j] and A[j+1]  & flag ← 1     //Swap
         }
    }

   if flag is 0                                  //For checking unnecessary comparison
   {
         end sorting                             //End of sorting
   }
}
```

2. Number of comparisons

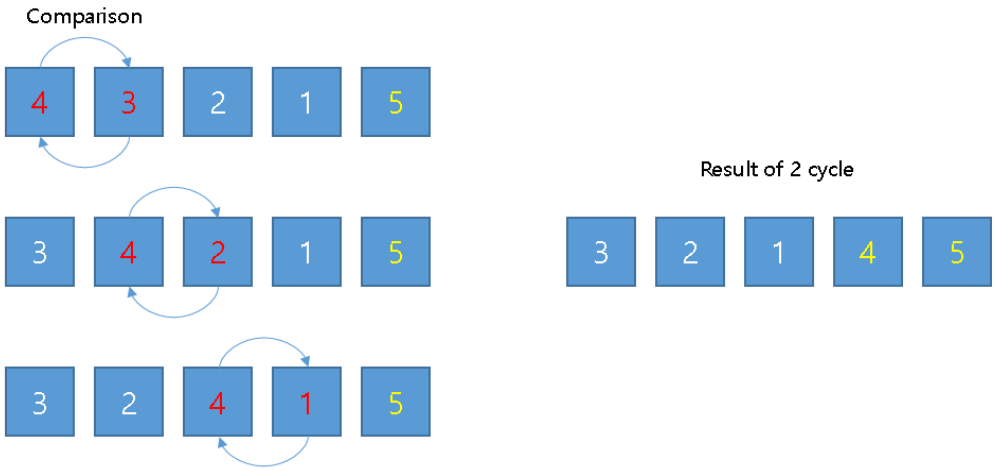
(1) The worst case

The worst case is when items come in descending order! (When I want array of ascending) Because run all cycle for ordering array. Refer below image.





There are **5** items and they need **4** comparison in **first cycle**. You need refer second image.

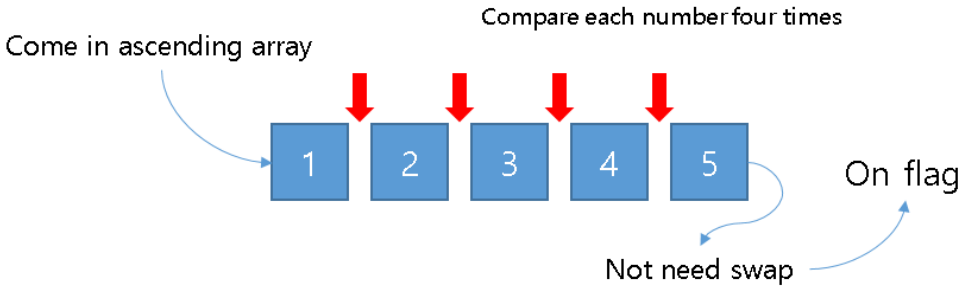


When referring second image, you can see comparison rule of sorting. In worst case you need **(N-1) cycle** and each cycles need **(N-1), (N-2), (N-3).....2, 1** comparison, so generalizing this rule in an equation is as follows.

$$\begin{aligned} \textit{Number of comparison} &= (n-1)+(n-2)+(n-3)\ldots 3+2+1 \\ &= n\,(n-1)/\,2 \end{aligned}$$

(2) The best case

You can weigh the number of comparisons in the best case much more simply. You just need to determine if there is a swap in the array. Refer below image.



You can use flag if array not need swap, then flag exit function of sorting. In best case (When array come in ascending) just need **N-1** time for comparison.

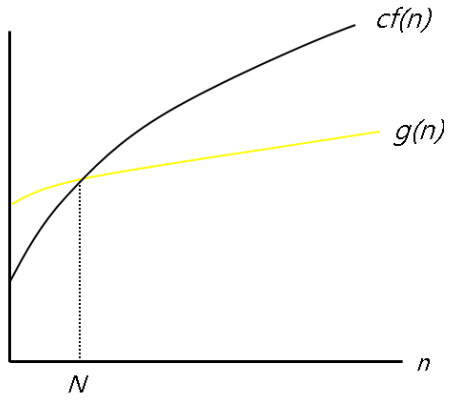
3. running time in Θ-notation

Before justify running time in big theta (**Θ**) notation, you need to know big oh (**O**) and big omega (**Ω**).

First Big-OH's definition

- For a given complexity function g(n) ∈ O(f(n)) satisfies the following

- For all integers n with **n ≥ N**, there are real number c (c>0) satisfied **g(n) ≤ c*f(n)** and an integer N, not negative.

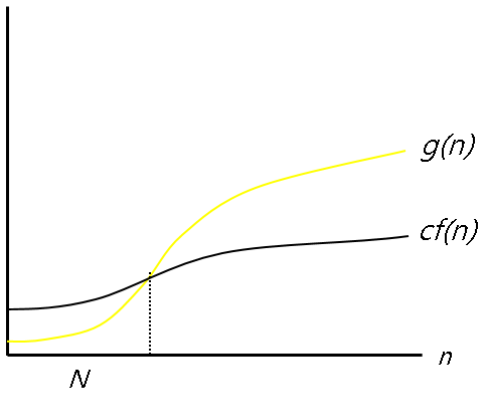


(a) g(n) ∈ O(f(n))

Second Big-omega's definition

- For a given complexity function g(n) ∈ Ω(f(n)) satisfies the following

- For all integers n with **n ≥ N**, there are real number c (c>0) satisfied **g(n) ≥ c*f(n)** and an integer N, not negative.



(b) g(n) ∈ Ω(f(n))

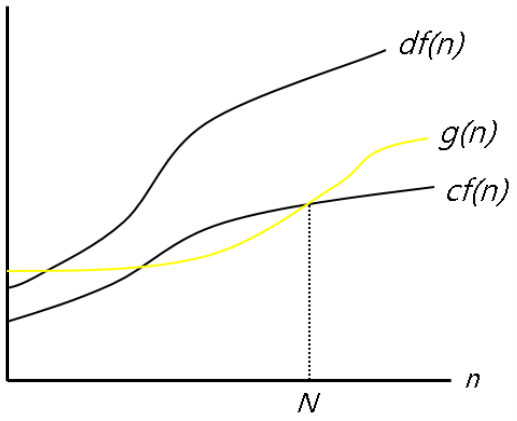
Third Big-theta's definition

Finally you can define Big-theta using the above two definitions.

- For the complexity function f(n), the following is satisfied: **θ(f(n) = O(f(n)) ∩ Ω(f(n))**

- So, for all integers n with **n ≥ N**, there are real number c (c>0) satisfied **c*f(n) ≤ g(n) ≤ d*f(n)** and an integer N, not negative.

- g(n) ∈(f(n)): g(n) is the order of f(n)



(c) g(n) ∈ Θ(f(n))

Ex) T(n)=(n(n-1))/2 is O(n^2), and Ω (n^2). Therefore, T(n)= Θ(n^2)

The above definitions of Big-oh and Big-omega show the principles and notation of Big-theta. Then, let's define the running time for each case of the pseudo-code presented in this task.

(1) The worst case

```
for i ← (number of item)-1 to 0 update i ← i-1    ---1
{
   j ← 0 & flag ← 0                               ---2
   while j<i update j ← j+1                       ---3
   {
         if A[j] > A[j+1]                         ---4
         {
            swap A[j] and A[j+1]  & flag ← 1      ---5
         }
    }

   if flag is 0                                   ---6
   {
         end sorting                              ---7
   }
}
```

1: Decrease from N to last 1 to determine if the code proceeds and the repeat ends. (The actual array number is different since the arrangement starts at 0). Thus, the number of repetitions of the code is $\boldsymbol{n-2}$.

2: Commands inside a repeating statement, which are performed $\boldsymbol{n-1}$ times because they are executed less than once.

3: The number of times the i value repeats depends on update. "n + (n-1) ... 3 + 2" equation can be derived, which can be summarized as $\frac{\boldsymbol{n(n+1)}}{\boldsymbol{2}} - \boldsymbol{1}$. (Be aware that in the actual number of runs, the comparison takes place one more time each.)

4: The fourth code is executed once less each time compared to the number of times the third code is executed, so equations such as "(n-1) + (n-2) ... 2 + 1" are derived and can be summarized as in $\frac{\boldsymbol{n(n-1)}}{\boldsymbol{2}}$.

5: Since the case is worst case, the conditional statement is executed (swap) every comparison time, so it is executed in the same way as code 4. Therefore, the following expressions are derived $\frac{\boldsymbol{n(n-1)}}{\boldsymbol{2}}$.

6: As with code two, it is located inside the repeating statement one, and the expression of $\boldsymbol{n-1}$ can be derived.

7: Not used in Worst case.

If all derived expressions are added, $\frac{\boldsymbol{3}}{\boldsymbol{2}}\boldsymbol{n^2} + \frac{\boldsymbol{5}}{\boldsymbol{2}}\boldsymbol{n} - \boldsymbol{2}$ will be given, and if the expression is marked as Big-theta, it will be as follows => $\boldsymbol{\Theta(n^2)}$

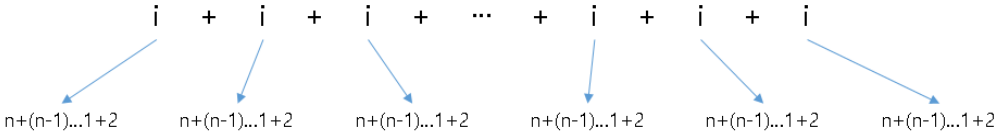
(2) The average case

To obtain the average execution time, you define the number of times each code can run and divide it by the defined number.

1: The first code says, "1, 2, 3 ... n" run times may occur. Therefore, the first-time formula is as follows: $\sum_{k=1}^{n} k = \frac{n(n+1)}{2}$ In conclusion, you divide it by the number of items, the final formula is as follows. $\mathbf{\frac{n(n+1)}{2n}}$.

2: Since code 2 is executed once less than code 1, the following expression is derived. $\sum_{k=1}^{n-1} k = \frac{n(n-1)}{2}$ → $\mathbf{\frac{n(n-1)}{2(n-1)}}$

3: Code three uses two repetition condition. In order to derive a general formula, the appropriate number of repetitions must be added for each case in code 1. Simply put, for each i-value, you can calculate the average by calculating all the possible times at each i-value.



Therefore, the following expressions are derived: $\sum_{k=1}^{n-1}\sum_{i=1}^{k}(n-i+1)$ When you solve an equation, you can: $\frac{n(n-1)(4n+4)}{12}$ divided by the number of runs $\mathbf{\frac{n(n-1)(4n+4)}{12(n-1)}}$

4: Code No. 4 is executed once less than code 3 is executed, and expressions are derived as follows. $(\sum_{k=1}^{n-1}\sum_{i=1}^{k}(n-i))/(n-1) = \mathbf{\frac{n(n-1)(2n-1)}{6(n-1)}}$.

5: For code 5, the code is executed according to the condition after the condition through size comparison has been executed. When averaging is obtained, it is one of the smaller or larger cases, so multiply the general formula by one-half. $\mathbf{\frac{n(n-1)(2n-1)}{12(n-1)}}$.

6: Since code 6 is the same number of times as code 2, the formula shall be as follows. $\mathbf{\frac{n(n-1)}{2(n-1)}}$.

7: Code 7 belongs to code 6. The average number of execution will be omitted and calculated. (As we use Big-theta notation, it does not have much impact.)

Finally, the largest number of execution times obtained in each code is n^2, so the Big-theta notation in the Average case is $\mathbf{\Theta(n^2)}$

4. My program and comments

The program items have been prepared so that they can operate according to their respective problems (5, 6, and 7). First you can check main function.

```
int main()
{
    int button = 0; //For checking select menu
    cout << "**************************" << endl;
    cout << "*      Start Program     *" << endl;
    cout << "*       Algorithms       *" << endl;
    cout << "*       2015722025       *" << endl;
    cout << "*     Jeong yong hoon    *" << endl;
    cout << "**************************" << endl;

    while (1)
    {
        cout << endl;
        cout << "1.Problem 5" << endl; //menu of view
        cout << "2.Problem 6" << endl;
        cout << "3.Problem 7" << endl;
        cout << "4.Exit Program" << endl;
        cout << "Select menu: ";
        cin >> button;

        switch (button)
        {
        case 1:
            Problem5(); //Function of problem 5
            break;
        case 2:
            Problem6(); //Function of problem 6
            break;
        case 3:
            Problem7(); //function of problem 7
            break;
```

```
    case 4:
        cout << "**************************" << endl; //end of program
        cout << "*      End of Program    *" << endl;
        cout << "*       Algorithms       *" << endl;
        cout << "*       2015722025       *" << endl;
        cout << "*     Jeong yong hoon    *" << endl;
        cout << "**************************" << endl;
        return 0;
    }
  }
}
```

In the main function, you can select the execution of the desired problem. Next, let's analyze the code that functions for each problem.

(1) Problem 5

```
void Problem5()
{
    int Best[] = { 1,2,3,4,5,6,7 }; //For showing Best case
    int Worst[] = { 7,6,5,4,3,2,1 }; //For showing Worst case
    int* Average = new int[7];
    Create_array(Average,7); //For showing average case(Create random array)

    cout << endl << "(a)Best case" << endl;
    Bubble(Best, sizeof(Best) / 4, 1); //sorting

    cout << endl << "(b)Worst case" << endl;
    Bubble(Worst, sizeof(Worst) / 4, 1); //sorting

    cout << endl << "(c)Average case" << endl;
    Bubble(Average, 7, 1); //sorting

    delete[] Average;
}
```

Question 5 is to print step-by-step what the comparison and swap takes place based on one sample for each case. Detailed execution results are presented by describing item 5. Below is an example of a simple practice.

```
(a)Best case
---Input Array---
1 2 3 4 5 6 7
-----------------
1 2 3 4 5 6 7
1 2 3 4 5 6 7
1 2 3 4 5 6 7
1 2 3 4 5 6 7
1 2 3 4 5 6 7
1 2 3 4 5 6 7
---Output Array---
1 2 3 4 5 6 7
Size of array: 7
Number of cycle: 1
Number of comparison: 6
-----------------
```

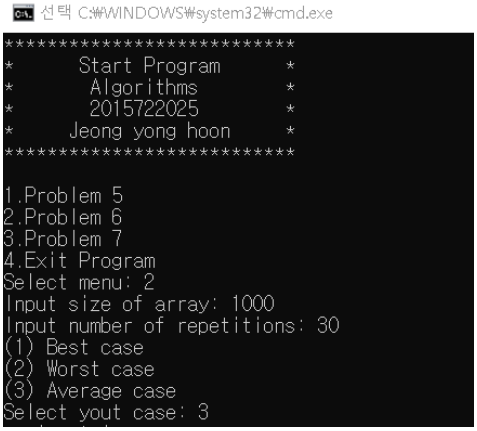
(2) Problem 6

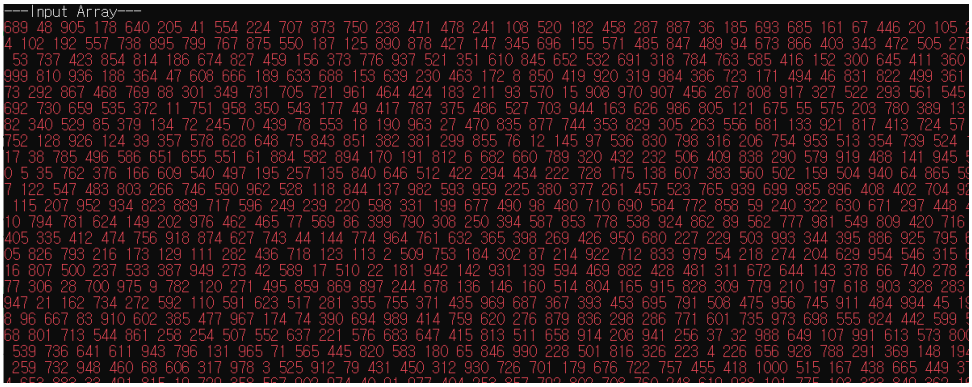
```
void Problem6()
{
    int size, repeat, Case;
    cout << "Input size of array: "; //Get size of array
    cin >> size;
    cout << "Input number of repetitions: "; //Get number of repetitions
    cin >> repeat;
    cout << "(1) Best case" << endl; //Get case of array
    cout << "(2) Worst case" << endl;
    cout << "(3) Average case" << endl;
    cout << "Select yout case: ";
    cin >> Case;

    int* A = new int[size];
    Info = new int[size];

    for (int count = 0; count < repeat; count++)
    {
        if (Case == 1) //Get array
            Best_create_array(A, size);
        else if (Case == 2)
            Worst_create_array(A, size);
        else if(Case == 3)
            Create_array(A, size);

        Bubble(A, size, 2); //practice sorting

        for (int init = 0; init < size; init++)
            A[init] = 0;

        Sleep(1000); //sleep function to prevent duplication of seed
    }
```

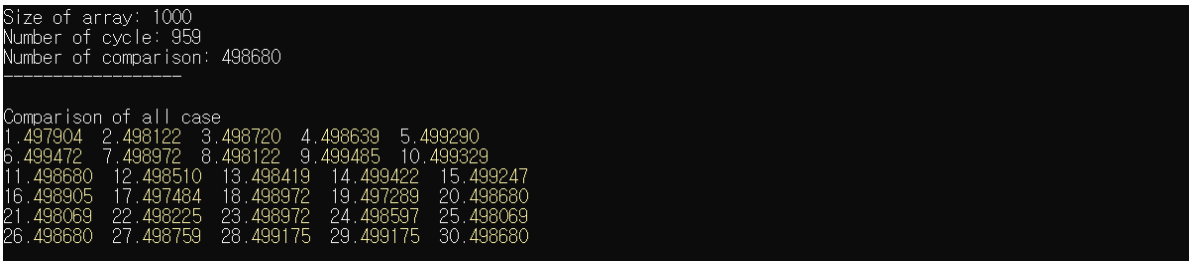
```
    cout << endl;
    cout << "Comparison of all case" << endl; //Output to clean up repeated results
    for (int i = 0; i < Info_count; i++)
    {
        cout << i + 1 << ".";
        textcolor(YELLOW, BLACK);
        cout << Info[i] << "  ";
        textcolor(WHITE, BLACK);
        if ((i + 1) % 5 == 0)
            cout << endl;
    }
    cout << endl;

    delete[] A;
    Info_count = 0;
    delete[] Info;
    return;
}
```

First, the size of the array and the number of repetitions are entered to obtain the sample, and the case of the array is selected. Next, by allowing an option to be printed on a function that outputs information, you can perform a function that outputs a sample that outputs a comparison number to the console. Then, the repeated results are compiled and printed.

The execution of a function can output the following results:







You can easily extract the number of comparisons for the average case by executing the following functions.

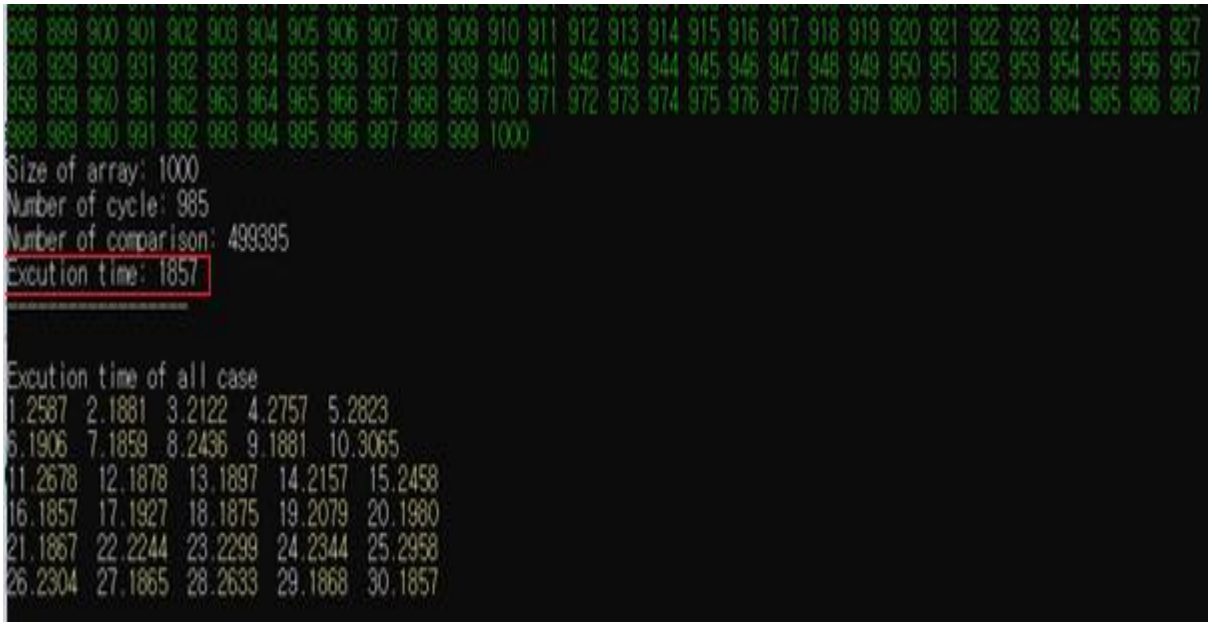
(3) Problem 7

There is a difference in the implementation of question No. 7 and question No. 6. The execution of question No. 7 has an **additional output of the actual execution time.**

You can see that the two functions have different options as factors in proceeding with the Bubble sort. (Option 3 can additionally measure the actual execution time.)

```
Bubble(A, size, 2); //practice sorting
```

```
Bubble(A, size, 3);
```



If you look at the red square, you can see that the actual execution time was added and printed.

(4) Function of Bubble

Next, I will analyze the alignment function used in all the menu.

```
void Bubble(int A[], int size, int option)
{
    int i, j, tmp, flag, comparison = 0, cycle = 0;
    chrono::system_clock::time_point StartTime, EndTime; //value of checking excution time
    chrono::microseconds excution;

    cout << "---Input Array---" << endl;
    textcolor(LIGHTRED, BLACK);
    printArray(A, size, NOP, NOP); //View input array
    textcolor(WHITE, BLACK);
    cout << "-----------------" << endl;

    if (option == 3)
        StartTime = chrono::system_clock::now(); //Information of current time

    for (i = size - 1; i > 0; i--) //For decreasing time of comparing
    {
        cycle++; //check cycle of bubble sort
        for (j = 0, flag = 0; j < i; j++, comparison++) //For comparing all items
        {
            if (option == 1)
                printArray(A, size, j, cycle); //For checking step by step

            if (A[j] > A[j + 1]) //Check integer
            {
                flag = 1;
                tmp = A[j + 1]; //swap
                A[j + 1] = A[j];
                A[j] = tmp;
            }
        }
        if (flag == 0) //For checking condition of orderig
            break;
    }
```

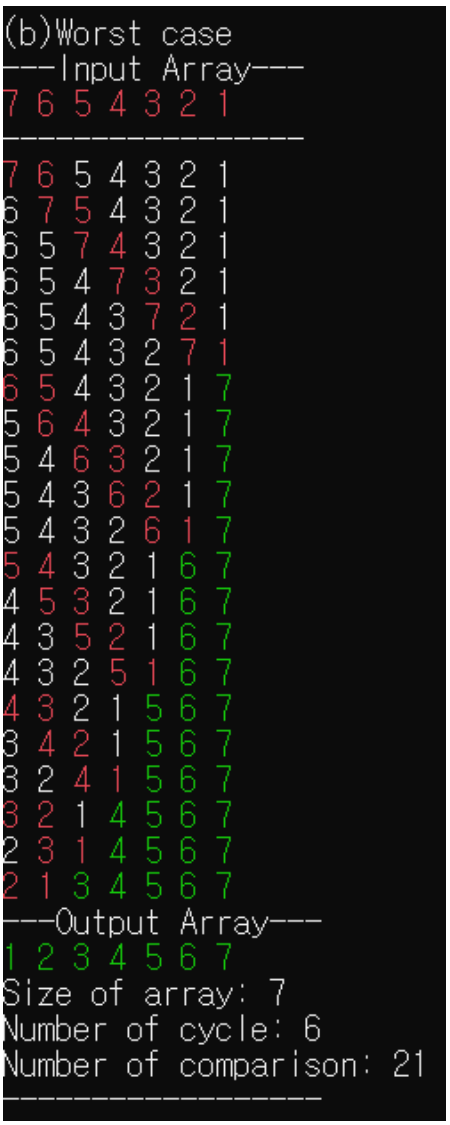
Part of sorting

```
    if (option == 3) //For printing real time of excution
    {
        EndTime = chrono::system_clock::now();
        excution = chrono::duration_cast<std::chrono::microseconds>(EndTime - StartTime);
    }

    if (option == 3) //Print excution time
    {
        cout << "Excution time: " << excution.count() << endl;
        Info[Info_count] = excution.count();
    }

    else if (option == 2) //print number of comparison
        Info[Info_count] = comparison;

    cout << "---Output Array---" << endl; //For printing information of working
    textcolor(LIGHTGREEN, BLACK);
    printArray(A, size, NOP, NOP);
    textcolor(WHITE, BLACK);
    cout << "Size of array: " << size << endl;
    cout << "Number of cycle: " << cycle << endl;
    cout << "Number of comparison: " << comparison << endl;

    if (option == 3)
    {
        cout << "Excution time: " << excution.count() << endl;
        Info[Info_count] = excution.count();
    }

    else if(option ==2)
        Info[Info_count] = comparison;

    cout << "------------------" << endl;
    Info_count++;

    return;
}
```

It is a function that actually aligns based on the array received, the information output varies depending on the option, and is implemented to include color in the text output from the console for readability.

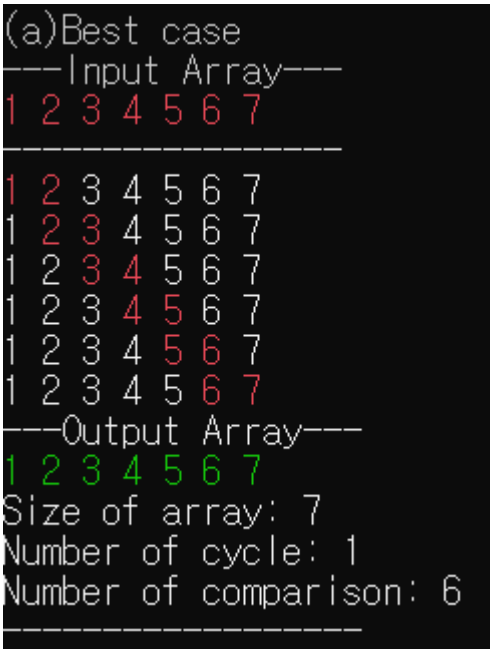
5. Result of step by step

(a) Worst case



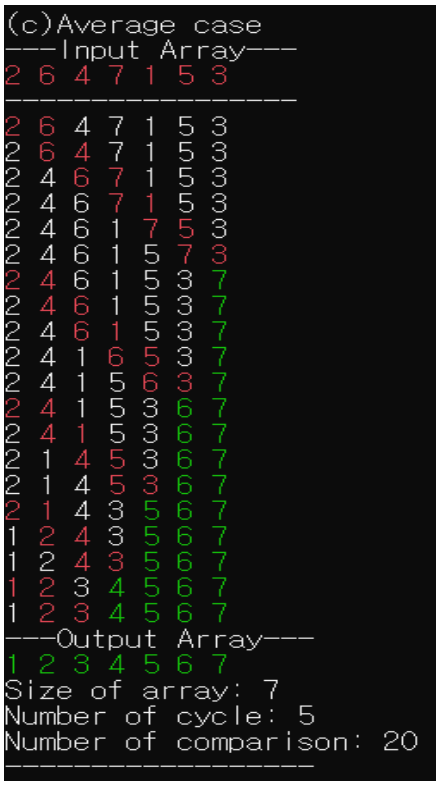
You can check a worst case of bubble sorting. Worst case was made in reverse order of ascending order, red is the process of making comparisons, and green means item that alignment has been completed. You can see that the expression n*(n-1)/2 is formed when looking at the number of comparisons.

(b) Best case



The best case can be arranged in ascending order simply. It can be seen that the comparison is made as shown in the results, but the swap does not occur. Then, it can also be seen that the number of comparisons is formed by the expression (n-1).

(c) Random input sequence



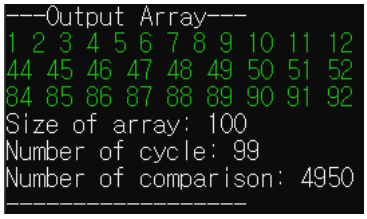
The following is the average case, which shows the step for creating an array with a random. It should be noted that the results can change whenever a function is executed because an array has been created with random.

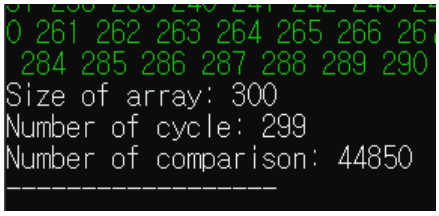
6. Graphs of based on number of comparisons

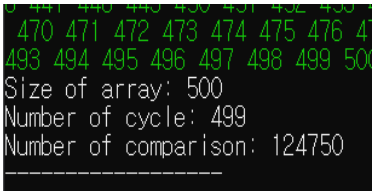
(a) Worst case

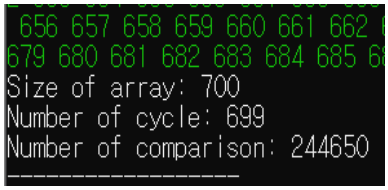
Time unit : micro

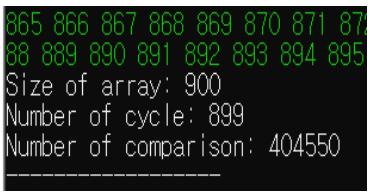
N(items)	100	300	500	700	900	1100	1300	1500
Comparisons	4950	44850	124750	244650	404550	604450	844350	1124250

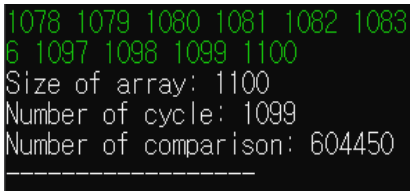


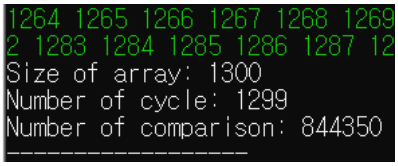


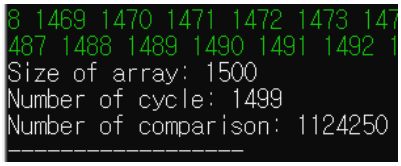


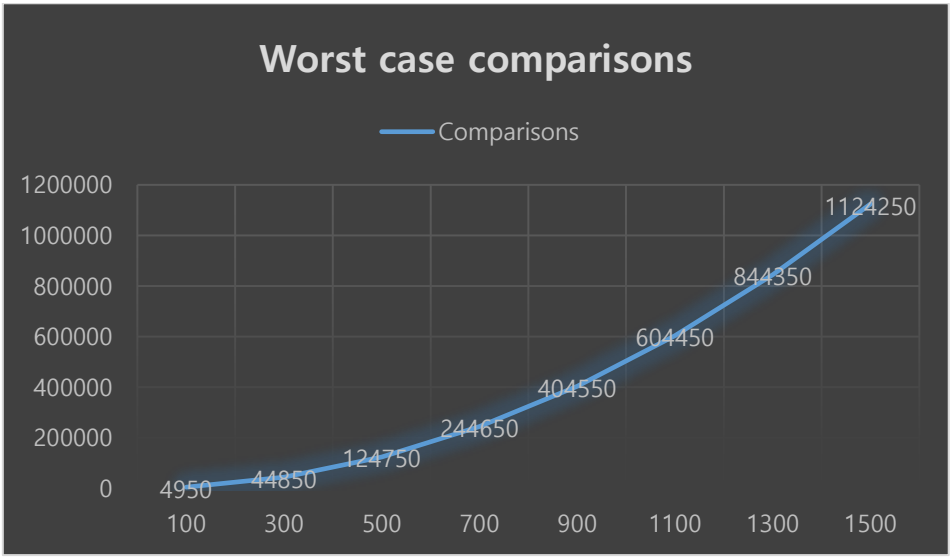








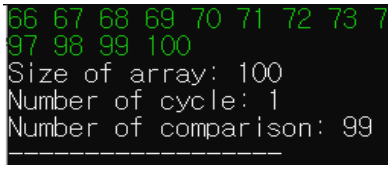


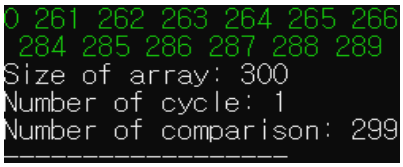


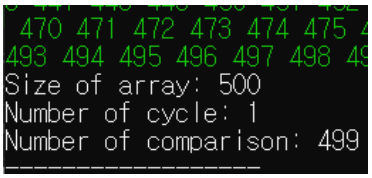
The above table and results are shown in the Worst case. The expression showing the number of comparisons in the Worst case was n*(n-1)/2 and as the value of N increases, it can be seen that the value of comparisons increases in the graph form of N^2.

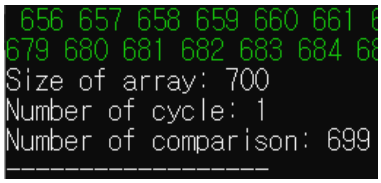
(b) Best case

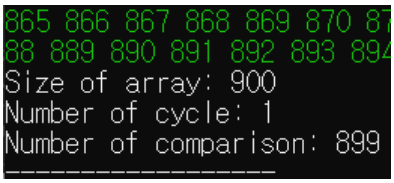
N(items)	100	300	500	700	900	1100	1300	1500
Comparisons	99	299	499	699	899	1099	1299	1499

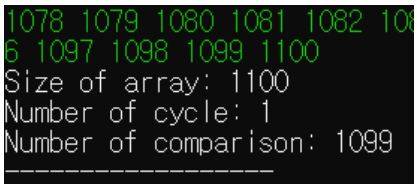


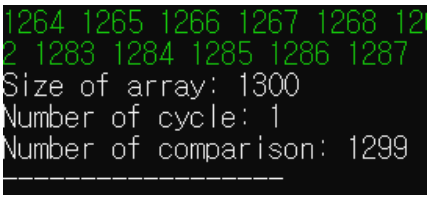


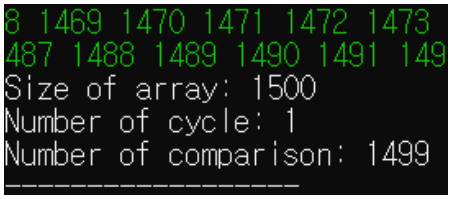


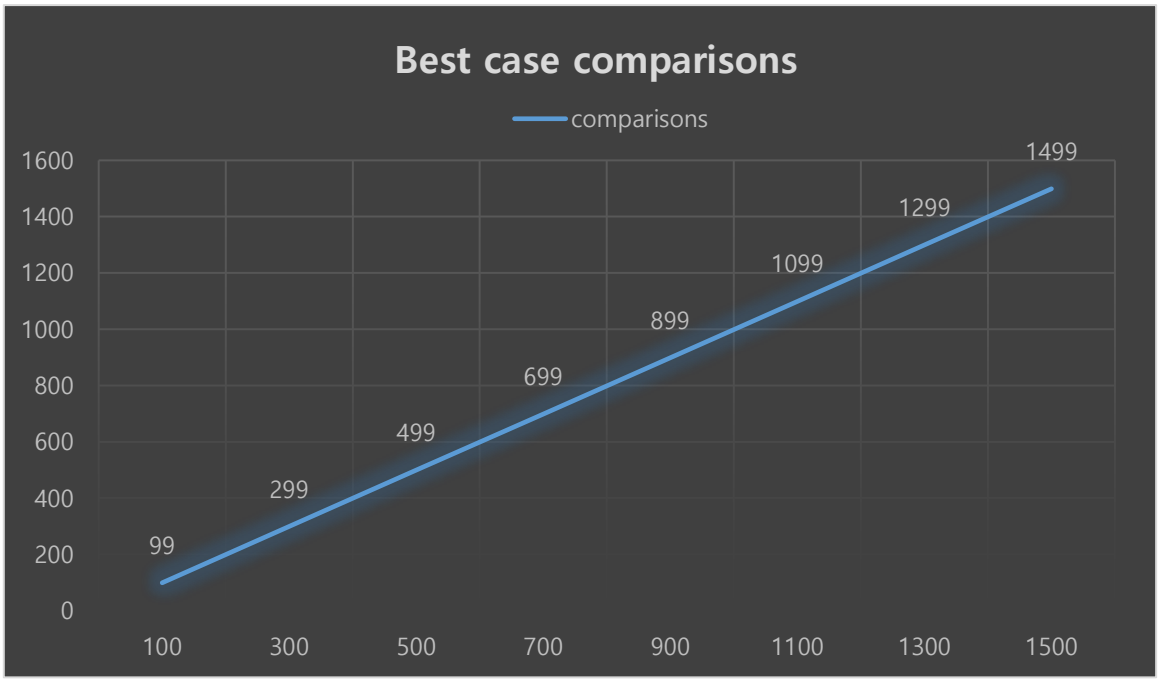








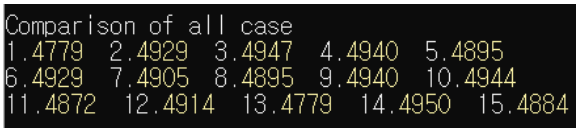


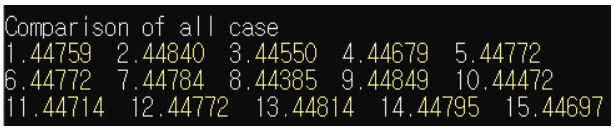


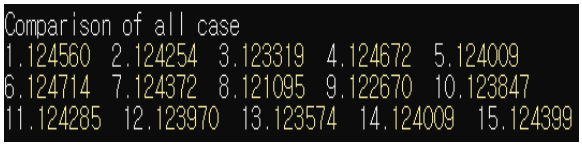
Similarly, the above table and results show the best case. The number of comparisons in the best case is (N-1), which shows that the actual graph also has a straight line graph.

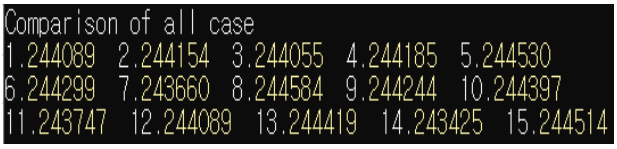
(c) Average case

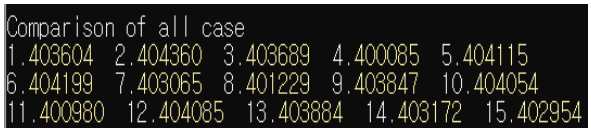
N(items)	100	300	500	700	900	1100	1300	1500
Comparisons(1)	4779	44759	124560	244089	403604	604125	844245	1119200
2	4929	44840	124254	244154	404360	603589	839097	1123785
3	4947	44550	123319	244055	403689	603820	843972	1123754
4	4940	44679	124672	244185	400085	603124	843489	1122597
5	4895	44772	124009	244530	404115	602910	843755	1123430
6	4929	44772	124714	244299	404199	602370	841424	1122480
7	4905	44784	124372	243660	403065	604099	841190	1123304
8	4895	44385	121095	244584	401229	604314	844197	1123754
9	4940	44849	122670	244244	403847	603322	843885	1122765
10	4944	44472	123847	244397	404054	603547	843755	1119785
11	4872	44714	124285	243747	400980	603670	841722	1123389
12	4914	44772	123970	244089	404085	603670	840072	1123509
13	4779	44814	123574	244419	403884	602434	843684	1122593
14	4950	44795	124009	243425	403172	602434	842810	1121549
15	4884	44697	124399	244514	402954	602497	842139	1124130
Average	4900	44710	123850	244159	403155	603328	842629	1122668

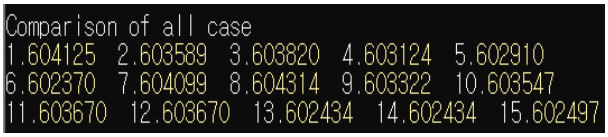


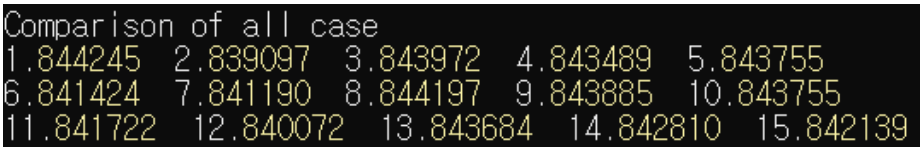


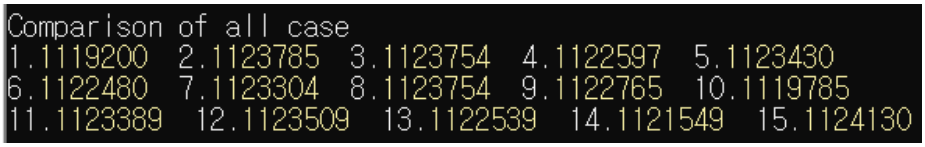


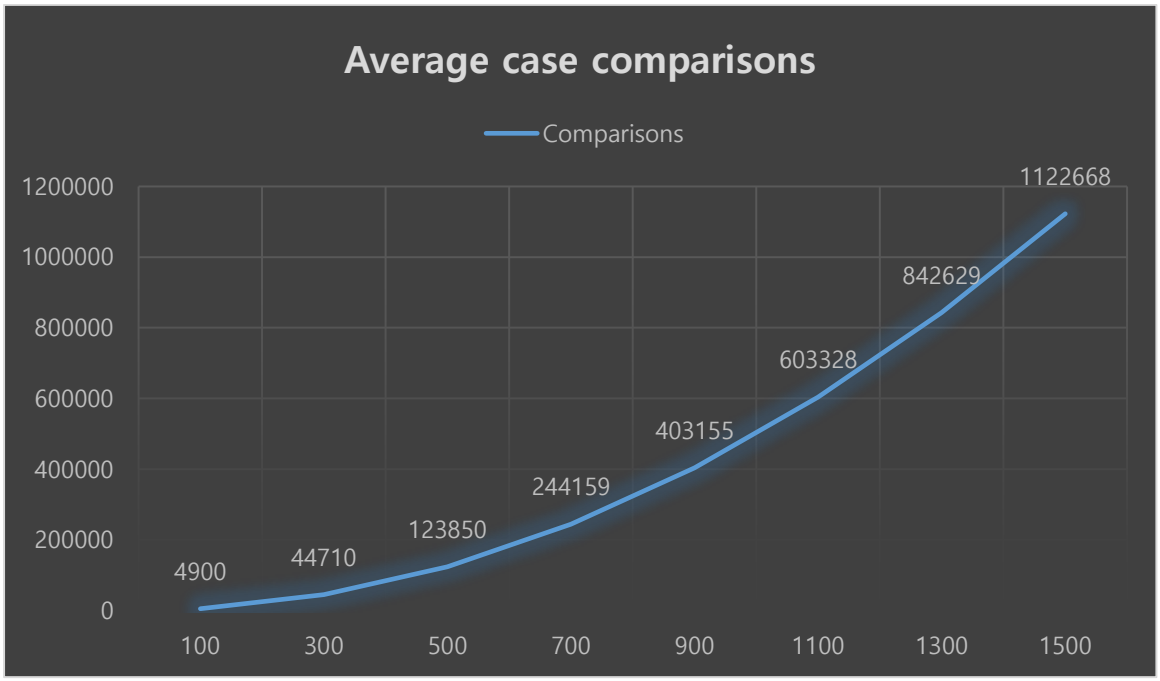












The above table and the results are for the average case (random case) and it can be found that the graph shows number of comparison similar to the worst case. The reason why the shape of the graph is similar to the Worst case is because the same notation was obtained by the number of execution obtained in paragraph 3 by the Big-theta notation, which results in a similar number of comparisons. The actual number of comparisons itself is almost the same.

7. Graphs of based on actual execution time

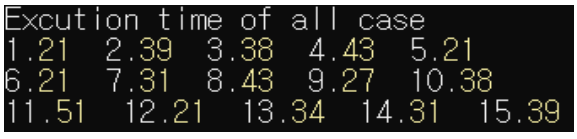
Information about PC:

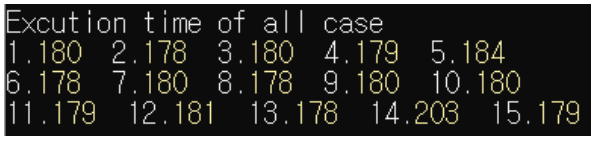
프로세서	Intel(R) Core(TM) i5-7300HQ CPU @ 2.50GHz, 2501Mhz, 4.
설치된 실제 메모리(RA...	8.00GB
총 실제 메모리	7.89GB
사용 가능한 실제 메모리	3.14GB
총 가상 메모리	15.7GB
사용 가능한 가상 메모리	7.44GB

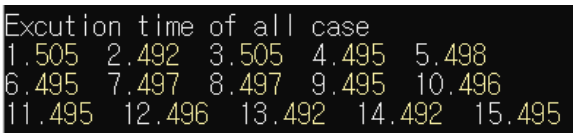
OS 이름	Microsoft Windows 10 Home

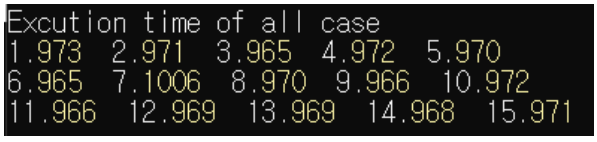
(a) Worst case

N(items)	100	300	500	700	900	1100	1300	1500
Execution time(1)	21	180	505	973	1615	2526	3347	4372
2	39	178	492	971	1602	2386	3377	4540
3	38	180	505	965	1596	2391	3385	4466
4	43	179	495	972	1604	2391	3365	4483
5	21	184	498	970	1601	2423	3494	4473
6	21	178	495	965	1600	2384	3325	4582
7	31	180	497	1006	1602	2380	3335	4481
8	43	178	497	970	1599	2418	3339	4509
9	27	180	495	966	1600	2421	3331	4420
10	38	180	496	972	1613	2391	3319	4450
11	51	179	495	966	1607	2389	3380	4499
12	21	181	496	969	1596	2399	3337	4482
13	34	178	492	969	1599	2430	3407	4438
14	31	203	492	968	1597	2407	3338	4449
15	39	179	495	971	1606	2382	3365	4509
Average	33.2	181.1	496.3	971.5	1602.5	2407.9	3362.9	4476.9









```
Excution time of all case
1.1615  2.1602  3.1596  4.1604  5.1601
6.1600  7.1602  8.1599  9.1600  10.1613
11.1607  12.1596  13.1599  14.1597  15.1606
```

```
Excution time of all case
1.2526  2.2386  3.2391  4.2391  5.2423
6.2384  7.2380  8.2418  9.2421  10.2391
11.2389  12.2399  13.2430  14.2407  15.2382
```

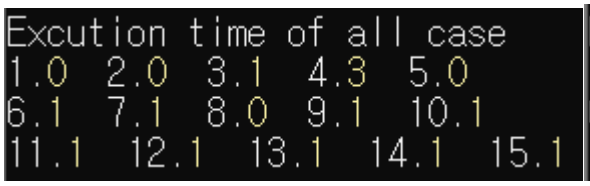
```
Excution time of all case
1.3347  2.3377  3.3385  4.3365  5.3494
6.3325  7.3335  8.3339  9.3331  10.3319
11.3380  12.3337  13.3407  14.3338  15.3365
```

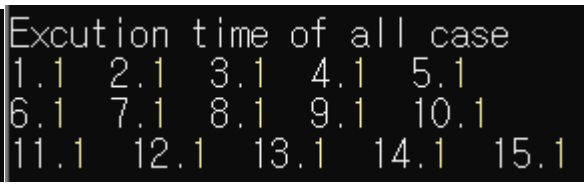
```
Excution time of all case
1.4372  2.4540  3.4466  4.4483  5.4473
6.4582  7.4481  8.4509  9.4420  10.4450
11.4499  12.4482  13.4438  14.4449  15.4509
```

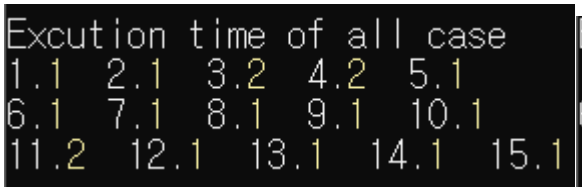
The end time can vary in condition of PC, even if the same arrangement is executed sorting repeatedly. So the expiry time of Worst case also collected several samples of execution time in the same array, averaged them out, sorted them into tables, and graphs. Note that the unit of extension time follows microseconds. If you look at the graph in Worst case, it will look similar to the one in which the comparison is graphically represented, which is related to the big-theta notation n^2.

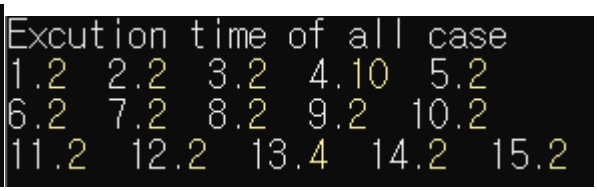
(b) Best case

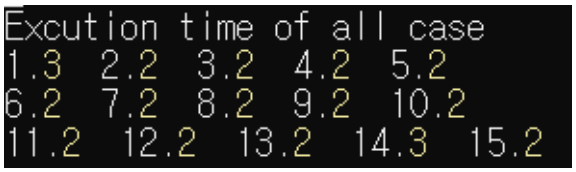
N(items)	100	300	500	700	900	1100	1300	1500
Execution time(1)	0	1	1	2	3	3	3	4
2	0	1	1	2	2	3	3	4
3	1	1	2	2	2	3	3	4
4	3	1	2	10	2	3	3	4
5	0	1	1	2	2	3	3	5
6	1	1	1	2	2	7	3	4
7	1	1	1	2	2	6	4	4
8	0	1	1	2	2	3	3	4
9	1	1	1	2	2	3	3	4
10	1	1	1	2	2	3	4	4
11	1	1	2	2	2	3	3	4
12	1	1	1	2	2	5	3	4
13	1	1	1	4	2	3	3	6
14	1	1	1	2	3	3	3	4
15	1	1	1	2	2	3	3	4
Average	0.87	1.	1.2	2.67	2.13	3.6	3.13	4.2

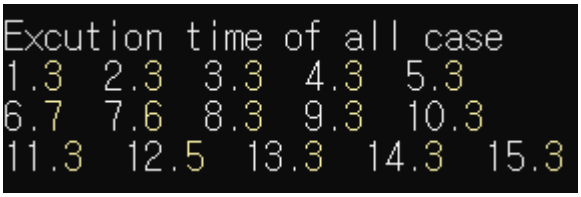


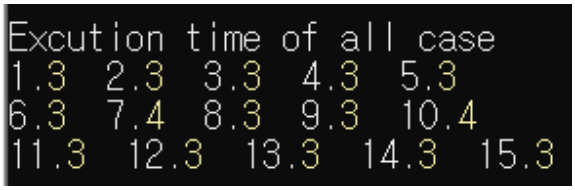


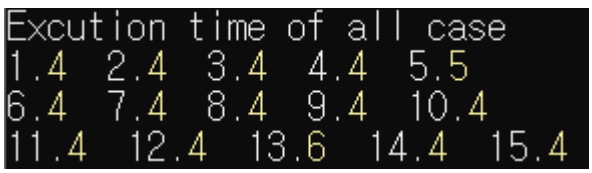


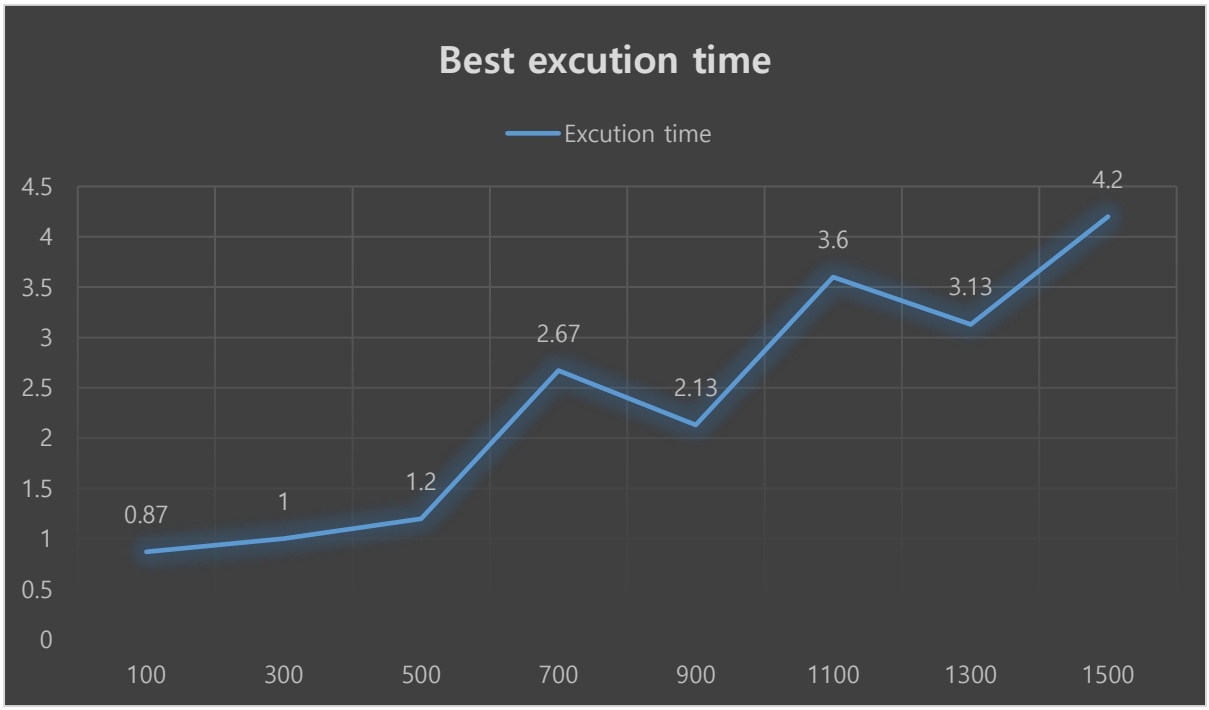








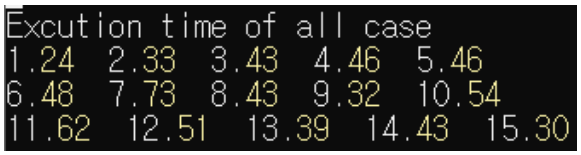


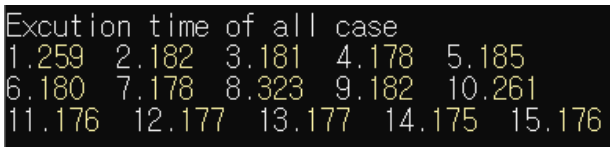


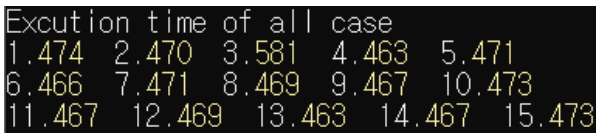
The following graph shows the best case execution time. Similarly, the average time was calculated and recorded because the aligned array did not always have the same execution time. If you look at the graph, you can see that time appears irregular. I learned that this reason is influenced by the computer environment. For example, if you have a power supply and a laptop battery, you can see that the battery is not only slower, it is much more erratic. However, as a result, you can see a graph that rises when there are more items.

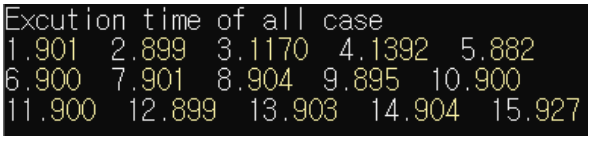
(c) Average case

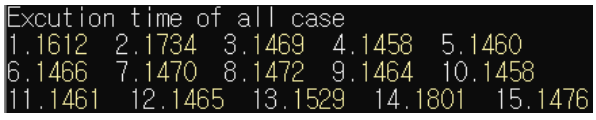
N(items)	100	300	500	700	900	1100	1300	1500
Execution time(1)	24	259	474	901	1612	2165	4232	4516
2	33	182	470	899	1734	2187	3020	4172
3	43	181	581	1170	1469	2472	3305	4123
4	46	178	463	1392	1458	2428	3057	4010
5	46	185	471	882	1460	2170	3509	4121
6	48	180	466	900	1466	2175	3025	4040
7	73	178	471	901	1470	2198	3088	4068
8	43	323	469	904	1472	2194	3068	4127
9	32	182	467	895	1464	2191	3056	4023
10	54	261	473	900	1458	2200	4316	4204
11	62	176	467	900	1461	2189	3017	4082
12	51	177	469	899	1465	2190	3055	4068
13	39	177	463	903	1529	2199	3021	4205
14	43	175	467	904	1801	2355	3034	4058
15	30	176	473	927	1476	2190	3019	4089
Average	44.5	199.3	476.2	951.8	1519.7	2233.5	3254.8	4127

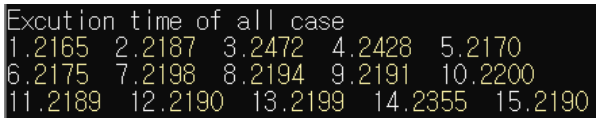


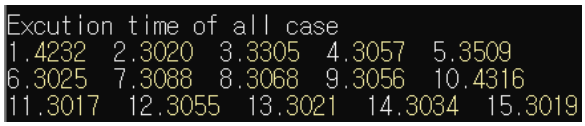


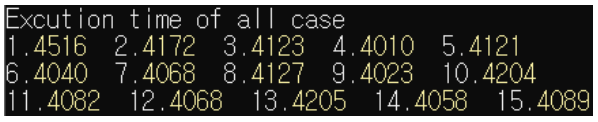


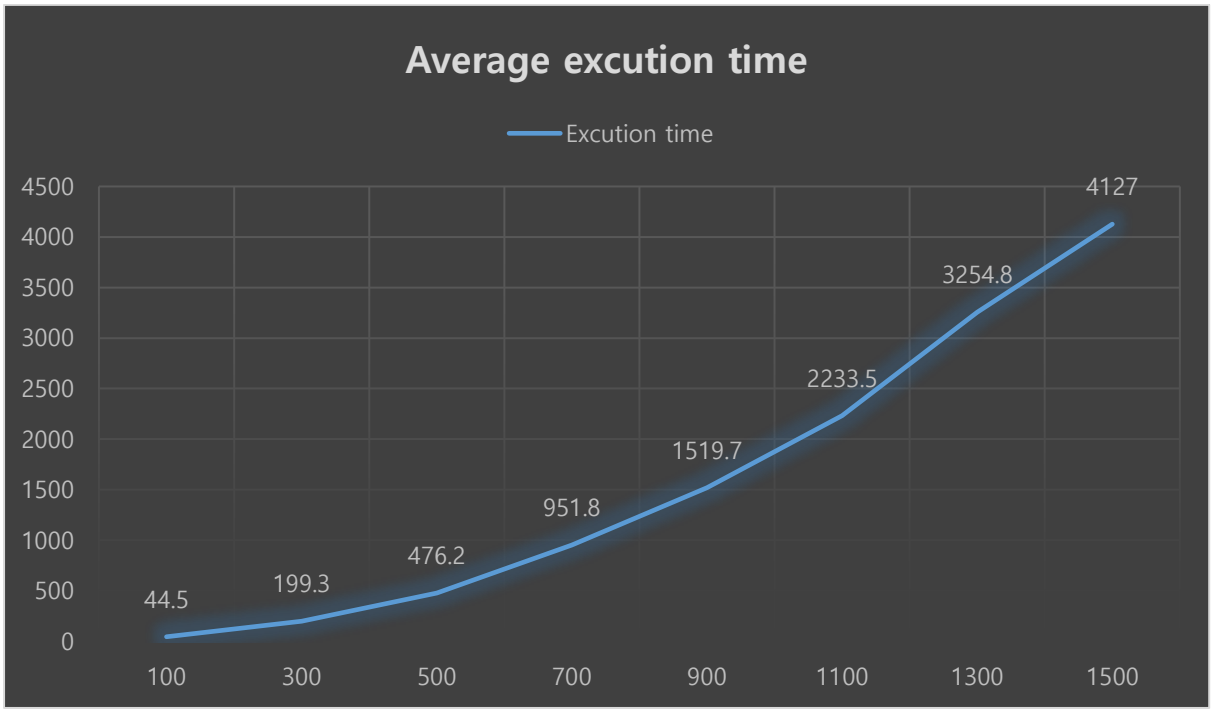












First of all, I did an experiment before writing a report, there was a phenomenon where execution time was more noticeable than Worst case. The next day, when I did the experiment again, I could get less execution time than the Worst case. It has also been learn that this phenomenon is affected by the computer environment (such as battery use) as described above. (Unfortunately, I thought there might be some kind of algorithm or other internal factors, but I couldn't find them). As a result, there is only a slight difference in execution time and the average case also shows a graph of the same shape as the Worst case. This is because the Big-theta notation is the same as n^2.

8. Reference

- ✓ Expression idea for Average case

  https://www.youtube.com/watch?v=euPlXW7dnlI

- ✓ Time-measuring idea

  https://jacking.tistory.com/988