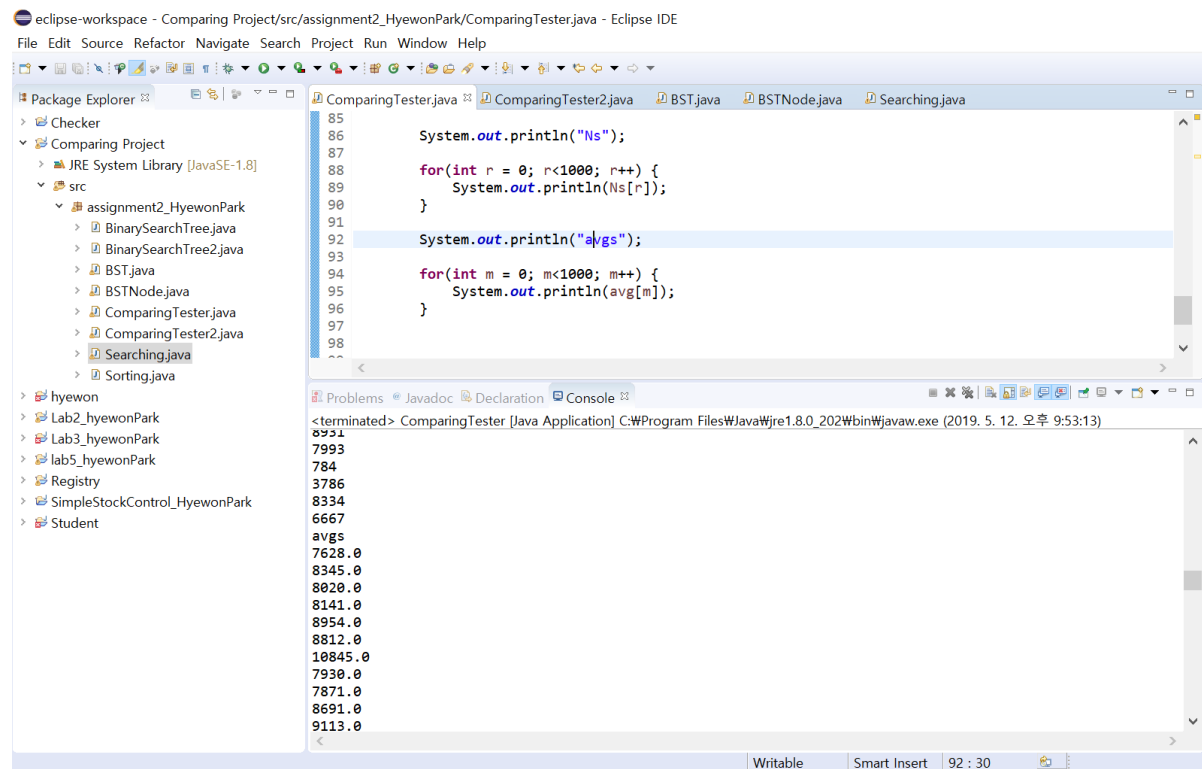


## -With the Task 1

=> compute average length of path to search random node with BST

<Screenshot of T1 output>



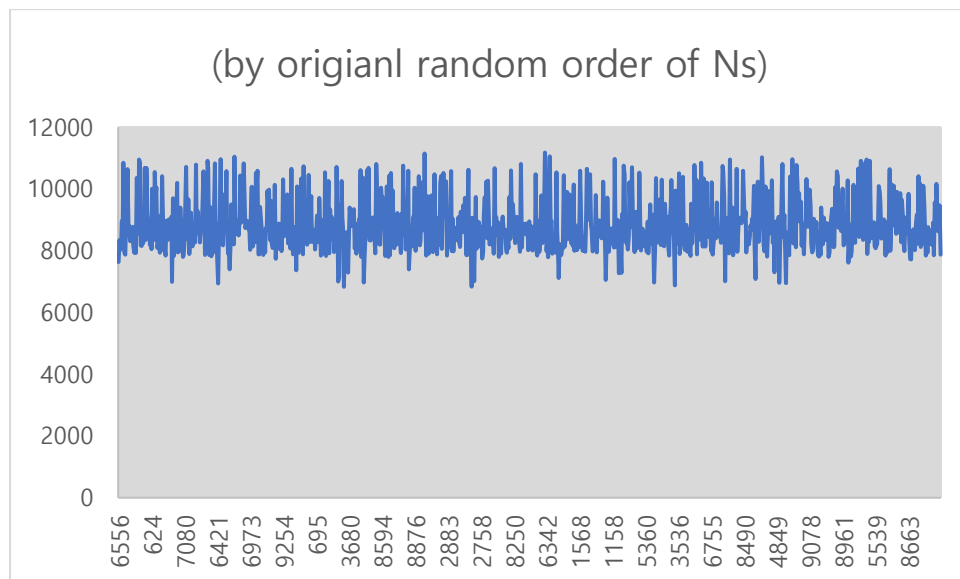
The screenshot shows the Eclipse IDE interface. The Package Explorer on the left displays a project structure with a source folder containing several Java files. The main editor window shows the code for `ComparingTester.java`, which includes two loops: one for printing an array `Ns` and another for calculating and printing an average `avg`. The Console window at the bottom shows the output of the program, including the array `Ns` and the average `avg`.

```
85
86     System.out.println("Ns");
87
88     for(int r = 0; r<1000; r++) {
89         System.out.println(Ns[r]);
90     }
91
92     System.out.println("avg");
93
94     for(int m = 0; m<1000; m++) {
95         System.out.println(avg[m]);
96     }
97
98
```

Console Output:

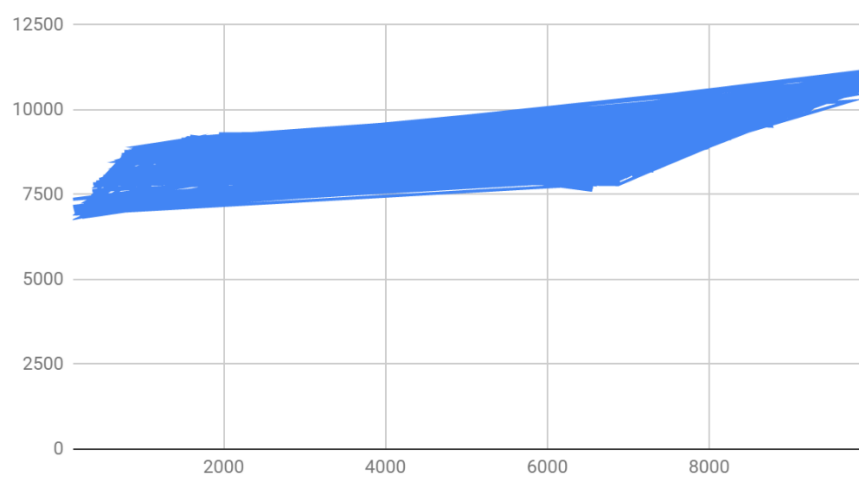
```
<terminated> ComparingTester [Java Application] C:\Program Files\Java\jre1.8.0_202\bin\javaw.exe (2019. 5. 12. 오후 9:53:13)
8951
7993
784
3786
8334
6667
avg
7628.0
8345.0
8020.0
8141.0
8954.0
8812.0
10845.0
7930.0
7871.0
8691.0
9113.0
```

<plots>



(↓ Ns are sorted in order)

BST graph (x: N, y: average path length)



(horizontal axis : tree size N, vertical axis : average number of compares)

## -With Task 2

<Screenshot of T2 output>

=> compute average length of path to search random node with sorted array, with binary search

The screenshot shows the Eclipse IDE interface. The Package Explorer on the left lists the project structure, including the 'src' folder and the 'assignment2\_HyewonPark' package. The main editor displays the code for 'ComparingTester2.java'. The code includes a method to generate a sorted array, a selection sort algorithm, and a loop that performs 1000 binary searches, recording the path length for each. The Console window at the bottom shows the output of the program, which includes the results of the 1000 iterations and the calculated average path length.

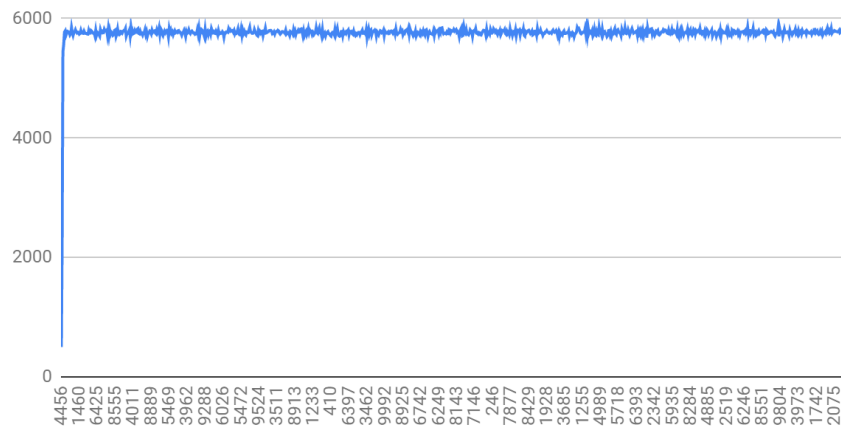
```
52     }
53     //sort the array.
54     selectionSort(array, 0, array.length-1);
55
56
57
58
59     /**
60      * choose the random target and search it by binary search. (do it for 1000 time)
61      */
62     for(int l = 0; l<1000; l++) {
63         int pick = rand.nextInt(N);
64         binarySearch (array[pick], array, 0, array.length-1);
65         results[l] = getCount();
```

<terminated> ComparingTester2 [Java Application] C:\Program Files\Java\jre1.8.0\_202\bin\javaw.exe (2019. 5. 12. 오후 9:59:44)

```
1086
6829
8668
1731
2737
avgs
500.0
2230.0
5440.0
5569.0
5775.0
5796.0
5726.0
5763.0
5786.0
5778.0
5771.0
```

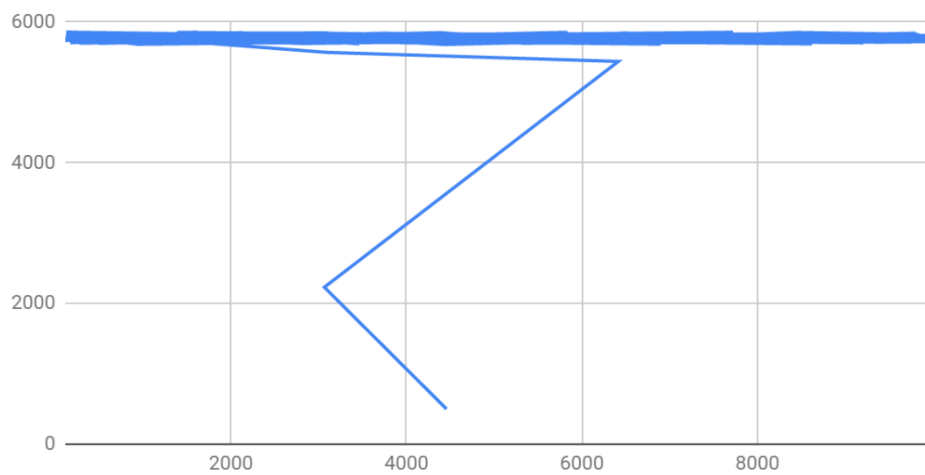
<plots>

Binary Search with Sorted array (with the original random order of Ns)



(↓ Ns are sorted in order)

Sorted Array with Binary Search graph (x: N, y: average path length)

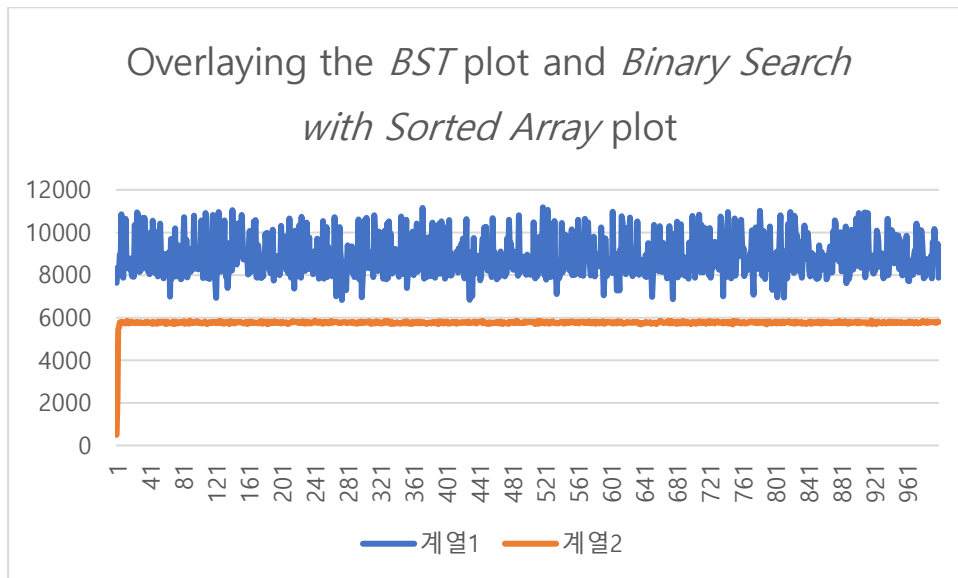


(horizontal axis : tree size N)

(vertical axis : average number of compares)

## -Plotting both results together

(average number of compares)



The Orange graph is for Binary Search with sorted Array, and the blue one is for BST.

It shows that **Binary Search with sorted array** is **more efficient** than Using **BST** for random node.

BST also has time complexity of  $\log N$  when it is well-balanced, but this time we built the BST by inserting random elements, so it became ill-balanced.

But, in case of binary search with sorted array, the array is always sorted, so there is no worry about searching from well balanced or ill balanced array. It's always well sorted.

That's why the average length of path for BST is remarkably higher than binary search tree's one.

<Code Fragments>

## For the Task 2

From the code Professor uploaded on the Blackboard,

I slightly changed the BST file.

=====

```
public BSTNode search (Comparable target) {  
    BSTNode curr = this.root;  
  
    for (;;) {  
        addCount();  
  
        if (curr == null)  
            return null;  
  
        direction = target.compareTo(curr.element);  
        if (direction == 0)  
            return curr;  
  
        if (direction < 0)  
            curr = curr.left;  
        else // direction > 0  
            curr = curr.right;  
    }  
}
```

```
        //getter for count  
        public int getCount() {  
            return count;  
        }  
  
        //setter for count  
        public void addCount() {  
            count++;  
        }
```

```
public void resetCount() {
    count = 0;
}
```

=====

This is the searching method for BST,

And I added the methods getCount(), addCount(), resetCount() to compute the length of the path( to find the random target from BST).

=====

At the testing class, I made instance for BST, the “tree” instance.

```
BST tree = new BST();
```

And to save the results of every computed path, I made the array ‘results’, and to compute the average results for each size N, I made the array ‘avg’. I also made array ‘Ns’, to store the every different random Ns.

```
int [] results = new int [1000];
double [] avg = new double [1000];
int [] Ns = new int [1000];
```

=====

```
for(int i = 0; i < 1000; i++) {
```

```
    tree.resetCount();
```

```
    /**
```

```
     * Build BST by insertion of N random keys.
```

```
     * (Randomly shuffle numbers of 0 to N-1 and insert them
```

```
into BST.
```

```
    */
```

```
    //deciding N(size of the tree)
```

```
    int N = 0;
```

```
    while(N<100) {
```

```
        N = rands.nextInt(10000);
```

```
    }//this makes sure that N is from 100 to 10,000
```

```
    Ns[i] = N;
```

```

//building BST
for(int j = 0; j<N; j++){
    int key = rand.nextInt(N);
    tree.insert(key);
}

/**
 * Randomly pick a node in the BST
 * and search it. (Also computes the path length.)
 */
for(int l = 0; l<1000; l++) {
    int pick = rand.nextInt(N);
    tree.search(pick);
    results[l] = tree.getCount();
}

int sum = 0;
for(int k = 0; k<1000; k++) {
    sum += results[k];
}

avg[i] = sum/1000;

}

```

=====

=>> You can see how I used the arrays I created at the testing.

## For the Task 2

I also used the code that Professor uploaded on blackboard, for Sorting the array and searching with Binary Search.

This time I changed them a lot.

-Highlighted as green part is my explanation!



=====

(There is no Boolean variable for tracing, and also no trace method)  
(And the type Comparable became int.)

```
public static void selectionSort (int[] a, int left, int right) {
    // Sort a[left...right].

    for (int l = left; l < right; l++) {
        int p = l; int least = a[p];
        // ... least will always contain the value of a[p].
        for (int k = l+1; k <= right; k++) {
            //int comp = a[k].compareTo(least);
            if (a[k] < least) {
                p = k; least = a[p];
            }
        }
        if (p != l) {
            a[p] = a[l]; a[l] = least;
        }
    }
}
```

```
public static int binarySearch (int target, int[] a, int left, int right)
{
    // Find which if any component of a[left...right] contains target
    // (where a is sorted).

    int l = left, r = right;

    while (l <= r) {
        addCount();
```

(And also used the getCount(), addCount() and resetCount() method.)

```
        int m = (l + r)/2;

        if (target < a[m])
            r = m - 1;
        else if (target > a[m])
            l = m + 1;
        else // target == a[m]
            return m;
    }
    return NONE;
}
```

### <Potential Limitation>

I think I completed the mission, but it could be better if we could make the BST well-balanced.

To do that, we need to save the Ns and sort them before we put it into BST. But I'm not sure whether if this can ruin the original purpose of this experiment.