PRODUCT SEARCH

Project & Slides Created by:

Edward Andino Sierra

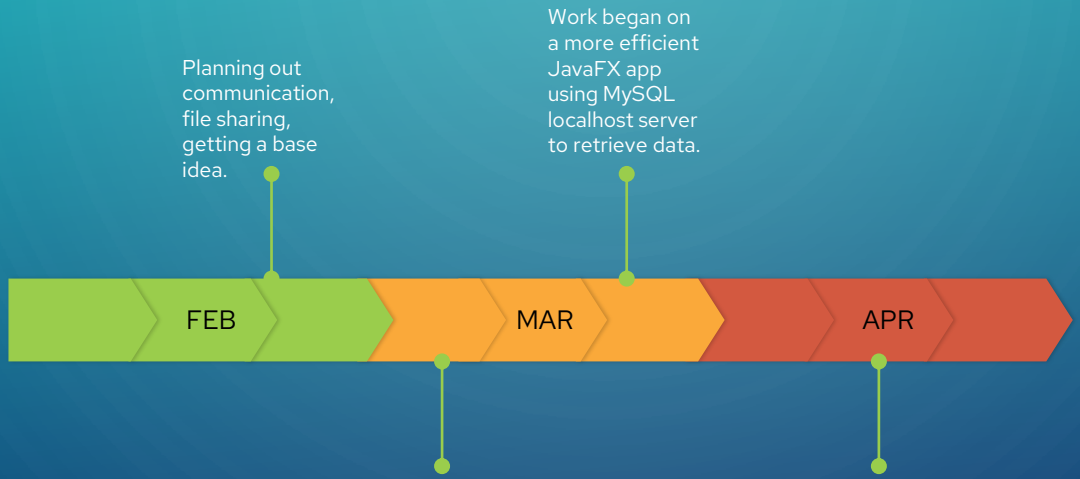
Carlos Mendoza

Ivan Zelenkov

THE APP'S FEATURES

- An app that displays the ID, Brand, ModelNumber, ModelYear, ProductName, and Description of multiple products stored in a database.
- A search bar allows the user instantly search for any product that has any combination of letters with its tuples in common with the search result.
    - The search essentially filters the data as the user types in the search bar; no need for typing and pressing enter.
- Before the search results are displayed, they are sorted.
- Merge Sort and Binary Search were implemented in order to create an efficient running time.

TIMELINE


Planning out communication, file sharing, getting a base idea.
Work began on a more efficient JavaFX app using MySQL localhost server to retrieve data.
FEB
MAR
APR
Originally planned to be a Java web app using Apache. Implementing Sort and Search in a servlet file proved to be difficult.
Final Touches. Potentially analyzing run time in more detail.

LANGUAGES AND SOFTWARE USED

JavaFX is a set of graphics and media packages that enables developers to design, create, test, debug, and deploy rich client applications that operate consistently across diverse platforms.

Scene Builder allows the easy layout of JavaFX UI controls, charts, shapes, and containers so that you can quickly prototype user interfaces

MySQL Server is used to add, delete, and modify data stored in the application.

IntelliJ IDEA Ultimate is designed for full-stack and enterprise development, supporting a wide range of backend and frontend frameworks and technologies. It includes profiling and database tools, an HTTP client, and many more features for professional developers out of the box.

Divide and Conquer Strategy

Using the Divide and Conquer technique, we divide a problem into subproblems. When the solution to each subproblem is ready, we 'combine' the results from the subproblems to solve the main problem.

Divide

- If q is the halfway point between p and r, then we can split the subarray A[p..r] into two arrays A[p..q] and A[q+1, r].

Conquer

- In the conquer step, we try to sort both the subarrays A[p..q] and A[q+1, r]. If we haven't yet reached the base case, we again divide both these subarrays and try to sort them.

Combine

- When the conquer step reaches the base step and we get two sorted subarrays A[p..q] and A[q+1, r] for array A[p..r], we combine the results by creating a sorted array A[p..r] from two sorted subarrays A[p..q] and A[q+1, r].

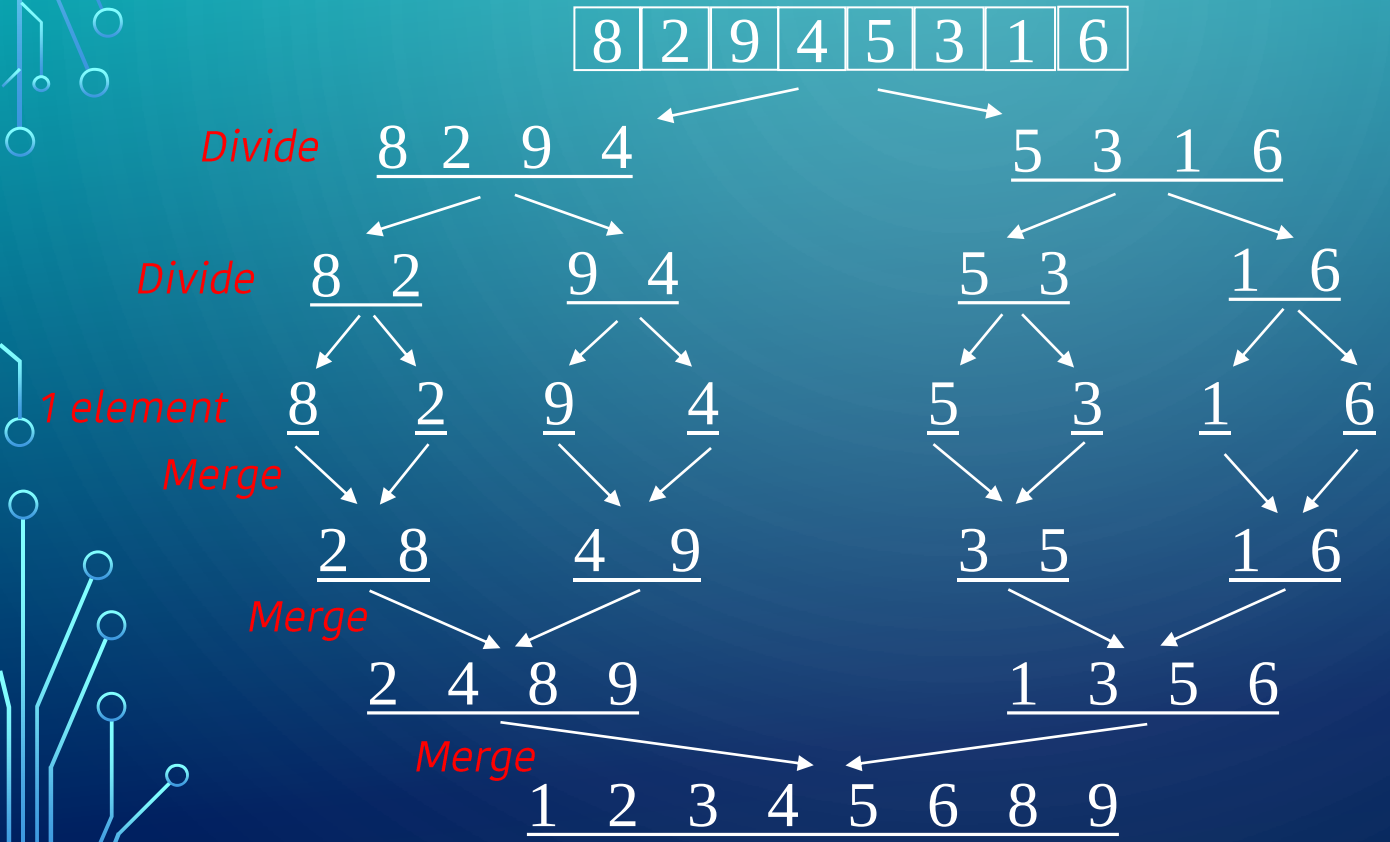
MERGE SORT

MERGE SORT

```
public static void mergeSort(int[] a, int n) {
        if (n < 2) {
                return;
        }

        int mid = n / 2;
        int[] l = new int[mid];
        int[] r = new int[n - mid];

        for (int i = 0; i < mid; i++) {
                l[i] = a[i];
        }

        for (int i = mid; i < n; i++) {
                r[i - mid] = a[i];
        }
        mergeSort(l, mid);
        mergeSort(r, n - mid);
        merge(a, l, r, mid, n - mid);
}
```

```
public static void merge( int[] a, int[] l, int[] r,

        int left, int right) {
        int i = 0, j = 0, k = 0;

        while (i < left && j < right) {
                if (l[i] <= r[j]) {
                        a[k++] = l[i++];
                } else {
                        a[k++] = r[j++];
                }
        }

        while (i < left) {
                a[k++] = l[i++];
        }

        while (j < right) {
                a[k++] = r[j++];
        }
}
```

MERGE SORT



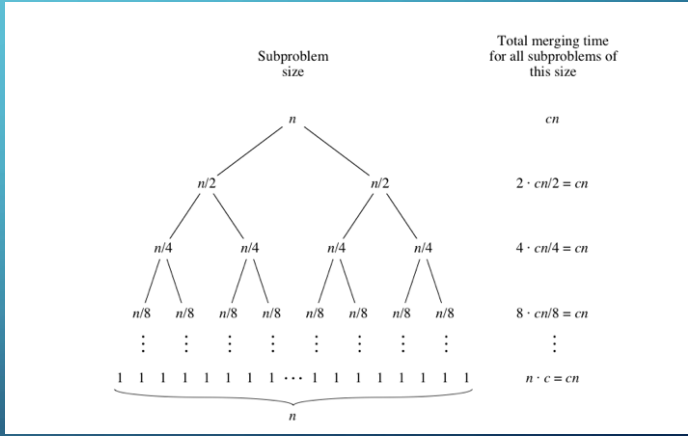
When solved, the time complexity will come to O(N log N)

This is true for the worst, average and best case since it will always divide the array into two and then merge.

The space complexity of the algorithm is O(N) as we're creating temporary arrays in every recursive call.

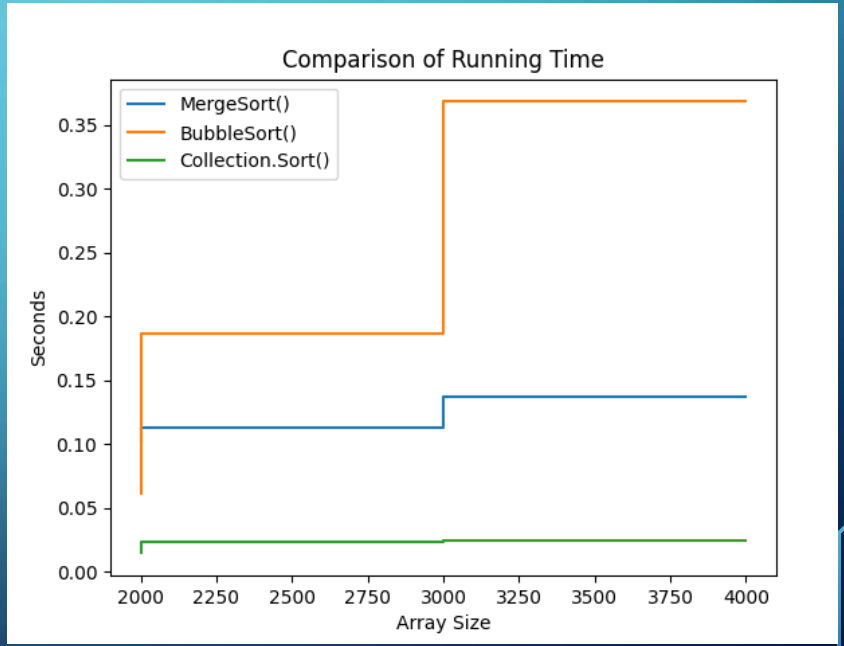
Analysis of merge sort

- l = log2 n +1
- cn = n/subproblems
- Total time cn(log2n+1) = O(nlog2 n)



Using Matplotlib

```
# Import Library

import matplotlib.pyplot as plt
import numpy as np

# Data Coordinates

x = np.array ([2000, 3000,4000]  )
y1 = np.array ([0.1025108, 0.1128249,0.1372066]  ) #mergeSort
y2 =  np.array([0.0612307,  0.1866135,0.3685195] )  #bubble
y3 =  np.array([0.0144089, 0.0228877,0.0243339]  )  #array.sort

# PLot

plt.step( x,y1,label='MergeSort()')
plt.step( x,y2, label='BubbleSort()')
plt.step( x,y3,label='Collection.Sort()')
# Add Title

plt.title("Comparison of Running Time")

# Add Axes Labels

plt.xlabel("Array Size")
plt.ylabel("Seconds")
plt.legend(loc="upper left")
# Display

plt.show()

```

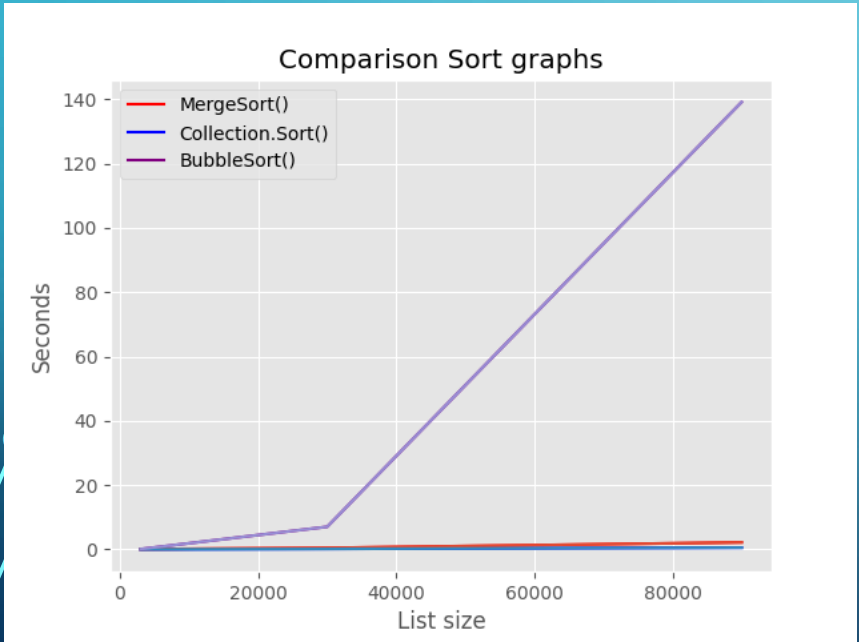


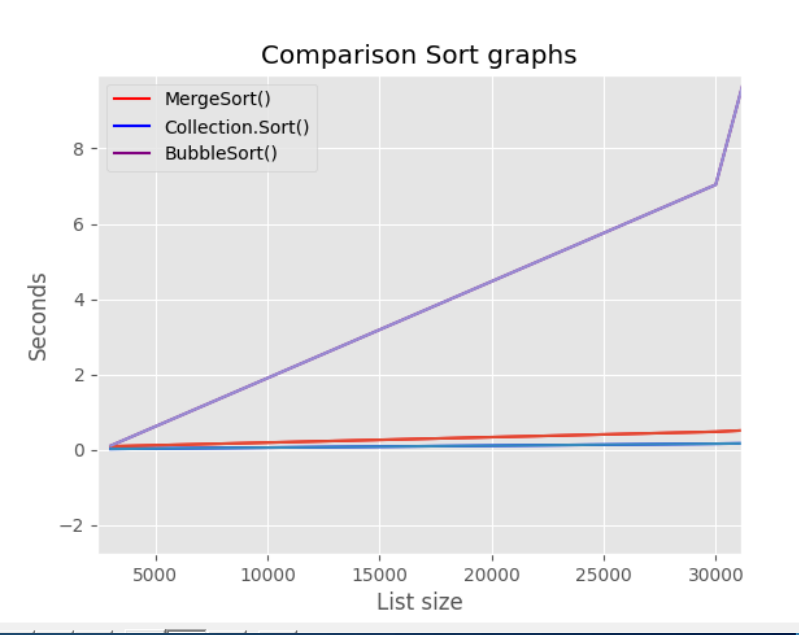
Arrays.sort() and Collections.sort()

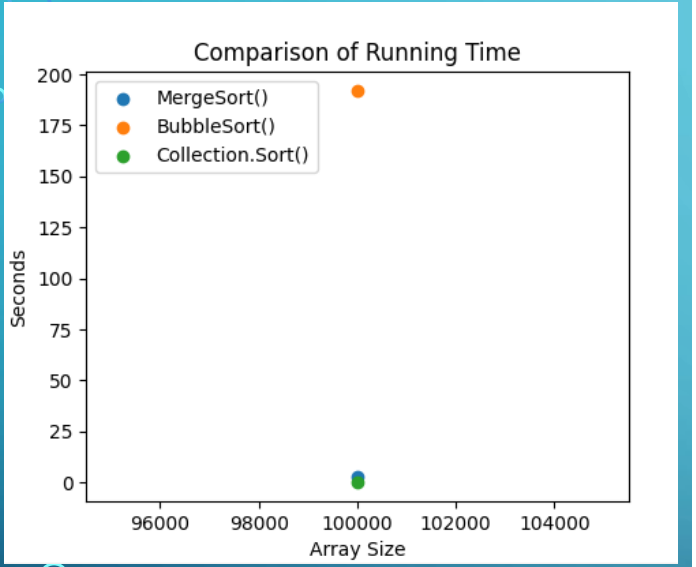
- Arrays.sort uses two sorting algorithms. One is a modification of Quicksort named dual-pivot quicksort, the other an adaptation of MergeSort named Timsort. Both have a time complexity of O(n log n), where n is the total number of items in the array.

- The time complexity including the comparator is O(n * (log n) * c), where c is the time complexity of the comparator. By default, Arrays.sort uses a comparator following the natural ordering of the content for an O(1) time complexity. Programmer-defined comparators with little computational work also resolve to an O(1) time complexity.

Comparison of Different Sorts








```
The Execution time of mergeSort is: 2.7229244  seconds
The Execution time of bubbleSort is: 191.7619817  seconds
The Execution time of Collection Sort is: 0.344165  seconds
Array Size:100000
```

Algorithm	10	20	50	100	1,000	10,000	100,000
O(1)	< 1 sec.	< 1 sec.	< 1 sec.	< 1 sec.	< 1 sec.	< 1 sec.	< 1 sec.
O(log(n))	< 1 sec.	< 1 sec.	< 1 sec.	< 1 sec.	< 1 sec.	< 1 sec.	< 1 sec.
O(n)	< 1 sec.	< 1 sec.	< 1 sec.	< 1 sec.	< 1 sec.	< 1 sec.	< 1 sec.
O(n*log(n))	MergeSort() Array.Sort()		< 1 sec.	< 1 sec.	< 1 sec.	< 1 sec.	< 1 sec.
$O(n^2)$	BubbleSort()		< 1 sec.	< 1 sec.	< 1 sec.	2 sec.	3-4 min.
$O(n^3)$	< 1 sec.	< 1 sec.	< 1 sec.	< 1 sec.	20 sec.	5.55 hours	231.5 days
$O(2^n)$	< 1 sec.	< 1 sec.	260 days	hangs	hangs	hangs	hangs
O(n!)	< 1 sec.	hangs	hangs	hangs	hangs	hangs	hangs
$O(n^n)$	3-4 min.	hangs	hangs	hangs	hangs	hangs	hangs

Display code

BINARY SEARCH

Locates a target value in a sorted ArrayList by successively eliminating half of the array from consideration.

How many elements will it need to examine? O(log N)
Recursive binary search method, the space complexity would be O(log N)

Example: Searching the array below for the value 42.

index	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16
value	-4	2	7	10	15	20	22	25	30	36	42	50	56	68	85	92	103

min

mid

max

BINARY SEARCH CODE

```
// Returns the index of an occurrence of target in a,
// or a negative number if the target is not found.
// Precondition: elements of a are in sorted order
public static int binarySearch(int[] a, int target) {
    int min = 0;
    int max = a.length - 1;

    while (min <= max) {
        int mid = (min + max) / 2;
        if (a[mid] < target) {
            min = mid + 1;
        } else if (a[mid] > target) {
            max = mid - 1;
        } else {
            return mid;    // target found
        }
    }
    return -(min + 1);     // target not found
}
```

BINARY SEARCH

Recursive binarySearch method.

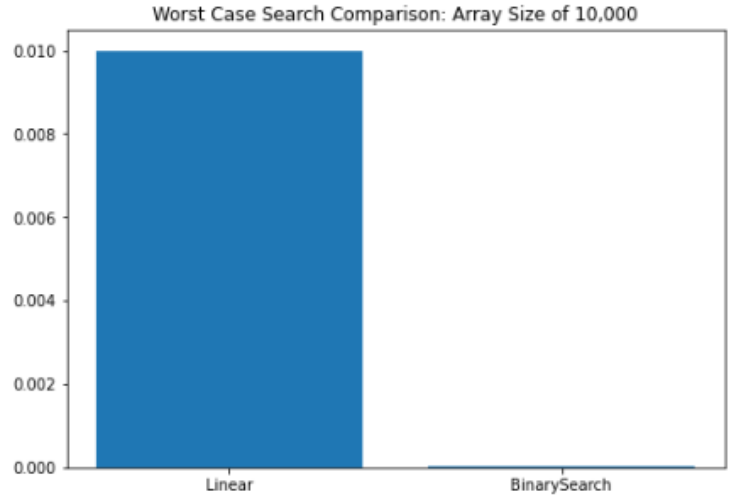
- If the target value is not found, return its negative insertion point.

index	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16
value	-4	2	7	10	15	20	22	25	30	36	42	50	56	68	85	92	103

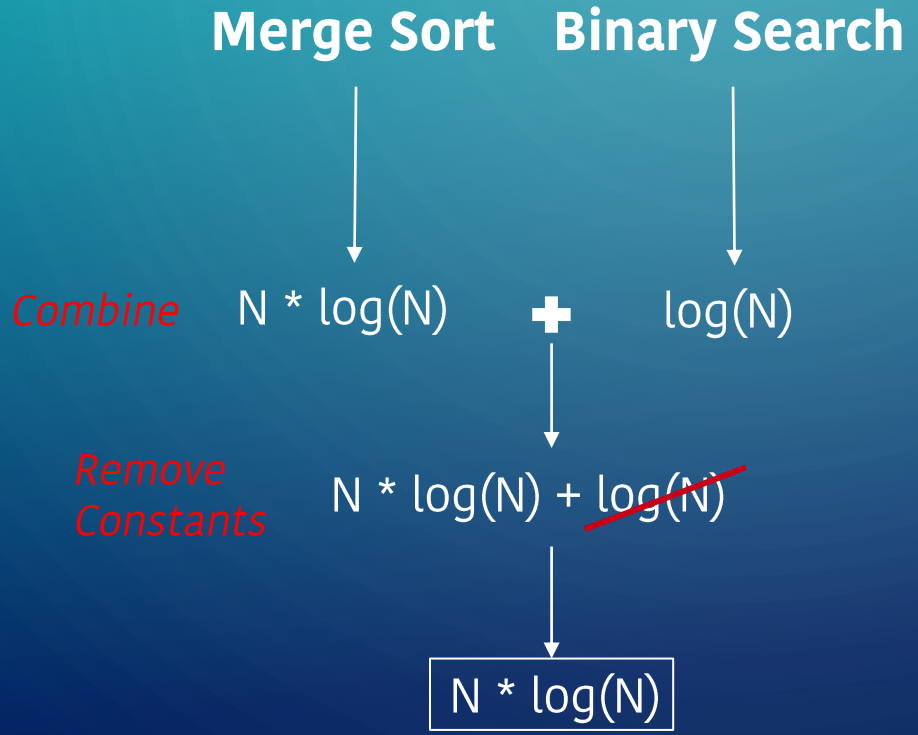
```
int index  = binarySearch(data, 42);  // 10
int index2 = binarySearch(data, 66);  // -14
```

SEARCH ALGORITHM COMPARISON

```
#Linear Search Worst Case Running Time: 0.0099996 seconds
#Binary Search Worst Case Running Time: 0.0000216 seconds
#Linear Search Worst Case: O(n)
#Binary Search Worst Case: O(log(n))

figr = plt.figure()
ax = figr.add_axes([0,0,1,1])
algorithms = ['Linear', 'BinarySearch']
runningTimes = [0.0099996, 0.0000216]
ax.bar(algorithms,runningTimes)
plt.title("Worst Case Search Comparison: Array Size of 10,000")
plt.show()
```



RUNNING TIME FOR COMBINED ALGORITHMS

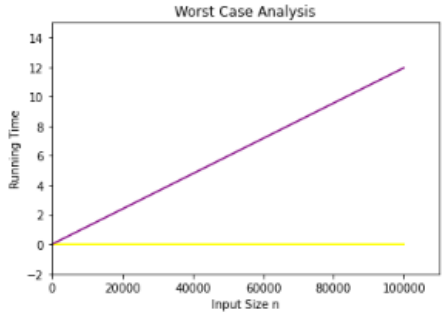


Combining these algorithms results in a running time of O(N log(N)). This means that the running time of Merge Sort and Binary combined is equivalent to just doing Merge Sort.

WORST(AND AVERAGE) CASE RUNNING TIME FOR COMBINED ALGORITHMS

In [6]:
```
import matplotlib.pyplot as plt
%matplotlib inline

#Running Time for MergeSort: 11.9436864 seconds
#Running Time for BinarySearch: 2.16E-5 seconds = 0.0000216 seconds
#Running Time is determined by an array size of 100,000 and string size of 10
plt.axis([0, 110000, -2, 15])
plt.title("Worst Case Analysis")
plt.xlabel("Input Size n")
plt.ylabel("Running Time")

#Plotting MergeSort
a = [0, 100000]
b = [0, 0.0000216]
plt.plot(a,b, color="yellow")

#Plotting BinarySearch
c = [0, 100000]
d = [0, 11.9436864]
plt.plot(c,d, color="purple")
```

Out[6]: `[<matplotlib.lines.Line2D at 0x1da64711300>]`

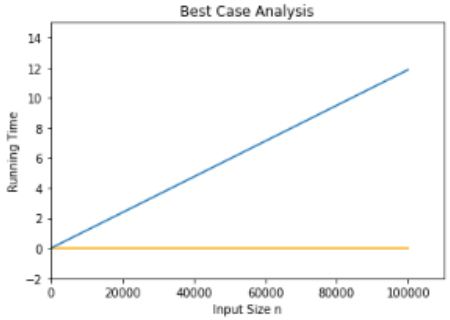


11.9436864 + 0.0000216 is effectively still 11.94 seconds.

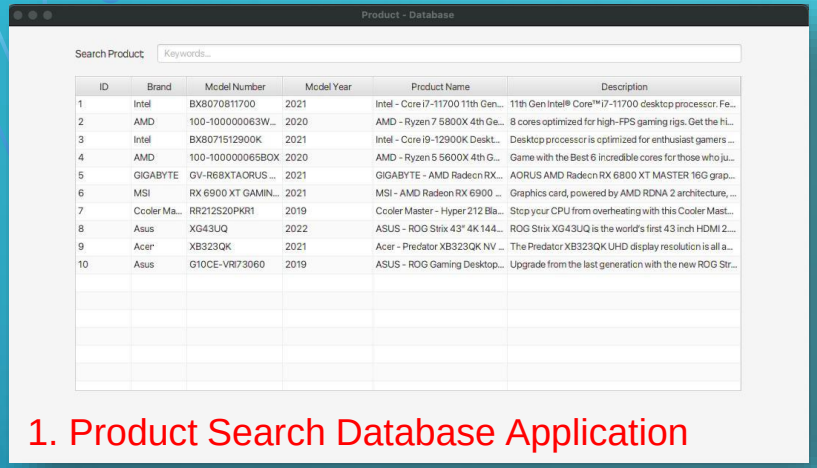
BEST CASE
RUNNING TIME FOR COMBINED ALGORITHMS

```
#Running Time for MergeSort: 11.8606131 seconds
#Running Time for BinarySearch: 2.07E-5 seconds = 0.0000207 seconds
#Running Time is determined by an array size of 100,000 and string size of 10

#Best case in MergeSort is still O(n*Log(n)), no need to change

#Best case in BinarySearch is O(1), a slight difference from O(log(n)), BinarySearch finds middle node immediately in order
#for this case to occur; result parameter was changed from 0 to 50000
plt.axis([0, 110000, -2, 15])
plt.title("Best Case Analysis")
plt.xlabel("Input Size n")
plt.ylabel("Running Time")

#Plotting MergeSort
a = [0, 100000]
b = [0, 0.0000207]
plt.plot(a,b, color="orange")

#Plotting BinarySearch
c = [0, 100000]
d = [0, 11.8606131]
plt.plot(c,d)
```

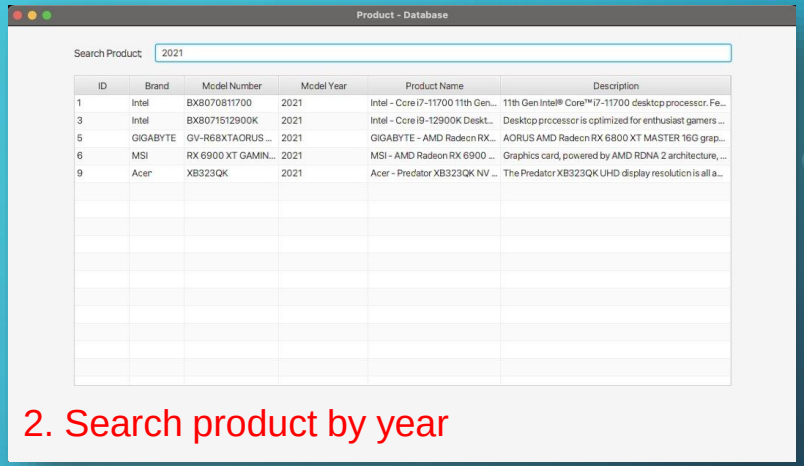
```
Out[1]: [<matplotlib.lines.Line2D at 0x2530a0b9bd0>]
```



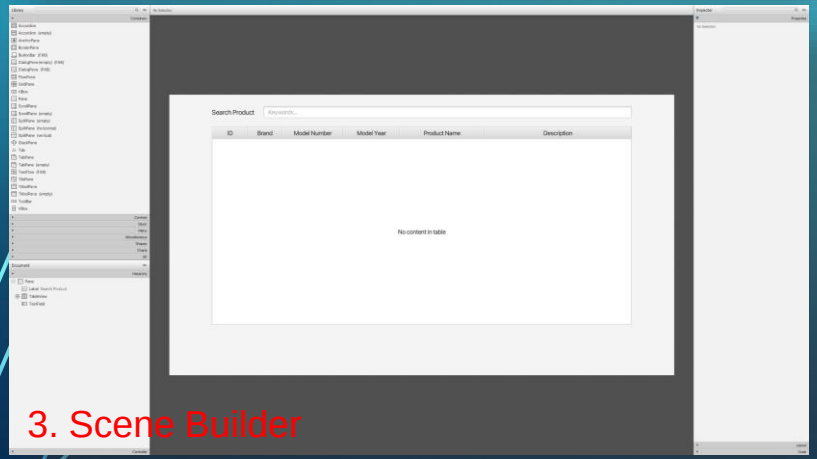
SCREENSHOTS OF CODE AND APP IN ACTION



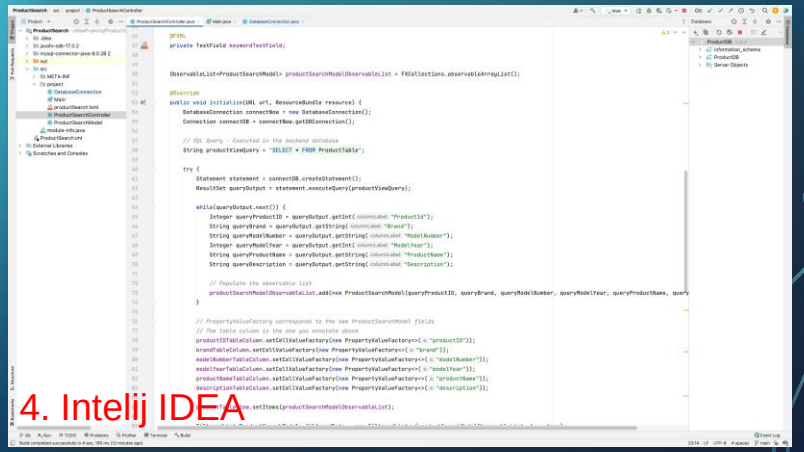
1. Product Search Database Application



2. Search product by year



3. Scene Builder



4. Intelij IDEA

CONCLUSION

- Our app works as expected. The running time is analyzed in detail.
- The project has been successfully completed.

Thank you