**Algorithms and Data Structures**

**Exercise 2: E-commerce Platform Search Function**

**Explain Big O notation and how it helps in analyzing algorithms.**

Big O notation is a way to describe how fast or slow an algorithm is, especially as the size of the input grows. It helps us understand the efficiency of an algorithm by showing how much time or space it might need in the worst-case. For example, an algorithm with O(n) grows step-by-step as the input increases, while one with O(log n) is faster and grows much more slowly. Big O helps developers compare different solutions, spot slow parts of the code, and choose the best way to solve a problem—especially when working with large amounts of data.

**Describe the best, average, and worst-case scenarios for search operations.**

| **Search Type** | **Best Case** | **Average Case** | **Worst Case** |
| --- | --- | --- | --- |
| Linear Search | O(1) | O(n) | O(n) |
| Binary Search | O(1) | O(log n) | O(log n) |

**Search.java**

package searching.java;

import java.util.\*;

class Product {

int productId;

String productName;

String category;

public Product(int productId, String productName, String category) {

this.productId = productId;

this.productName = productName;

this.category = category;

}

public void display() {

System.*out*.println( productId + ","+ productName +","+ category);

}

}

public class search {

public static void main(String[] args) {

Scanner sc = new Scanner(System.*in*);

Product[] productsLinear = {

new Product(105, "Mouse", "Electronics"),

new Product(102, "Notebook", "Stationery"),

new Product(109, "Keyboard", "Electronics")

};

System.*out*.println("Enter Product ID to search (Linear Search): ");

int linearSearchId = sc.nextInt();

boolean found = false;

for (Product p : productsLinear) {

if (p.productId == linearSearchId) {

System.*out*.println("Product found using Linear Search:");

p.display();

found = true;

break;

}

}

if (!found) {

System.*out*.println("Product not found using Linear Search.");

}

Product[] productsBinary = {

new Product(101, "Pen", "Stationery"),

new Product(103, "Charger", "Electronics"),

new Product(106, "Bag", "Travel"),

new Product(110, "Shoes", "Footwear"),

new Product(115, "Watch", "Accessories")

};

Arrays.*sort*(productsBinary, Comparator.*comparingInt*(p -> p.productId));

System.*out*.println("Enter Product ID to search (Binary Search): ");

int binarySearchId = sc.nextInt();

int low = 0, high = productsBinary.length - 1;

found = false;

while (low <= high) {

int mid = (low + high) / 2;

if (productsBinary[mid].productId == binarySearchId) {

System.*out*.println("Product found using Binary Search:");

productsBinary[mid].display();

found = true;

break;

} else if (productsBinary[mid].productId < binarySearchId) {

low = mid + 1;

} else {

high = mid - 1;

}

}

if (!found) {

System.*out*.println("Product not found using Binary Search.");

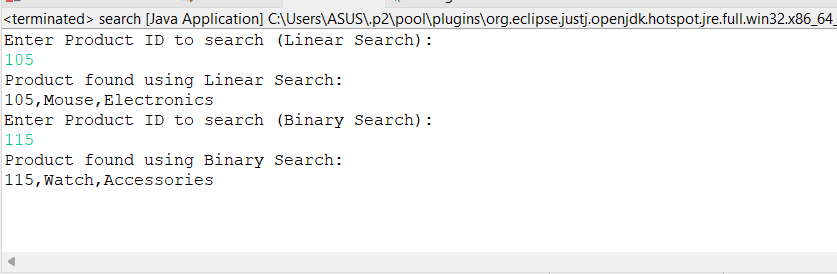
}

sc.close();

}

}

**Output:**

****

**Compare the time complexity of linear and binary search algorithms.**

| **Algorithm** | **Best Case** | **Average Case** | **Worst Case** | **Requires Sorted Data?** |
| --- | --- | --- | --- | --- |
| **Linear Search** | **O(1)** | **O(n)** | **O(n)** | **❌ No** |
| **Binary Search** | **O(1)** | **O(log n)** | **O(log n)** | **✅ Yes** |

* Linear Search checks each element one by one. In the worst case, it may go through the entire list, making it slower for large datasets.
* Binary Search works by repeatedly dividing the sorted array in half. It is much faster but requires the array to be sorted.

**Discuss which algorithm is more suitable for your platform and why.**

* If the data is unsorted and searches are infrequent, then linear search is suitable as it doesn't require sorting.
* If the data is sorted or can be sorted once and reused, and many searches need to be done, then binary search is better because it is much faster as input size grows.

**Exercise 7: Financial Forecasting**

**Explain the concept of recursion and how it can simplify certain problems.**

Recursion is a programming technique where a method calls itself to solve a problem. A recursive function breaks a complex problem into smaller, simpler sub-problems, each resembling the original problem.

Every recursive function has two main parts:

1. Base Case – the condition that stops the recursion.
2. Recursive Case – the part where the function calls itself with a smaller input.

**Code:**

**Futurevalue.java**

package futurevalue;

public class futurevalue {

public static double calculate (double presentValue, double rate, int years) {

if (years == 0) {

return presentValue;

}

return calculate(presentValue \* (1 + rate), rate, years - 1);

}

public static void main(String[] args) {

double presentValue = 5000;

double rate = 0.08;

int years = 6;

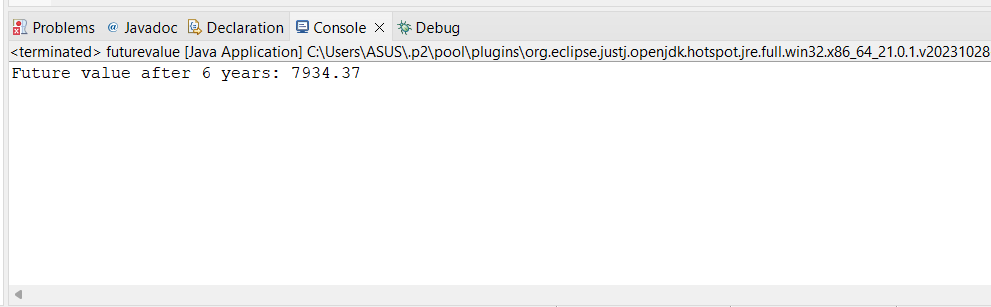
double futureValue = calculate(presentValue, rate, years);

System.out.printf("Future value after %d years: %.2f\n", years, futureValue);

}

}

**Output:**



**Analysis**

The recursive method:

public static double calculate(double presentValue, double rate, int years) {

if (years == 0)

return presentValue;

return calculate (presentValue \* (1 + rat), rate, years - 1);

}

has a linear time complexity:

* Time Complexity: O(n)  
  Because the function calls itself once per year, where n is the number of years.
* Space Complexity: O(n)  
  Each recursive call adds a frame to the call stack. For large n, this can lead to stack overflow.

**Optimization:**

For optimization we can use a iterative approach which gives O(n) time complexity and O(n) space Complexity.

Instead of calling the function again and use a loop. It does the same job but uses less memory and runs faster.

double futureValue(double val, double rate, int years) {

for (int i = 0; i < years; i++) {

val = val \* (1 + rate);

}

return val;

}

* The loop runs once for each year, so the number of operations increases linearly with years.Each iteration does a constant time operation: multiplication and addition → O(1).
* So the total time = years × O(1) = O(n).