**Week – 1**

**Dats structures and Algorithms**

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

**Scenario:**

Working on the search functionality of an e-commerce platform. The search needs to be optimized for fast performance.

**Steps:**

1. **Understand Asymptotic Notation**

**Big O notation** : It is a way to describe how the performance of an algorithm changes as the input size grows. It focuses on the most significant part of the algorithm's behavior as the number of elements (n) becomes very large.

**Various scenerios for search operations :**

* Best case is the most optimistic one . It predicts what’s the least amount of work the algorithm could do.
* Average case is the typical number of steps the algorithm will do most of the time.
* Worst case is the most pessimistic one . It predicts what’s the most amount of work that could happen if the data or query is not in our favor.

**Linear search(Only Works On Unsorted Array ) :**

1. Best case: O(1) — The element is at the first position.
2. Average case: O(n/2) — The element is somewhere in the middle.
3. Worst case: O(n) — The element is at the last position or not present at all.

**Binary search (Only Works On Sorted Data):**

1. Best case: O(1) — The middle element is the one we want.
2. Average and Worst case: O(log n) — Cuts the search space in half each time.
3. **Setup : Create the Product class**

class product {

int productid;

string productname;

string category;

product(int productid, string productname, string category) {

this.productid = productid;

this.productname = productname;

this.category = category;

}

@override

public string tostring() {

return "product{id=" + productid + ", name=" + productname + ", category=" + category + "}";

}

static product linearsearch(product[] products, int idtofind) {

for (product p : products) {

if (p.productid == idtofind) {

return p;

}

}

return null;

}

}

1. **Implementation: Linear Search and Binary Search**

import java.util.\*;

public class ProductSearch {

public static int linearSearchProduct(Product[] products, int targetId) {

for (int i = 0; i < products.length; i++) {

if (products[i].productId == targetId) {

return i;

}

}

return -1;

}

public static int binarySearchProduct(Product[] products, int targetId) {

int left = 0;

int right = products.length - 1;

while (left <= right) {

int mid = left + (right - left) / 2;

if (products[mid].productId == targetId) {

return mid;

} else if (products[mid].productId < targetId) {

left = mid + 1;

} else {

right = mid - 1;

}

}

return -1;

}

public static void main(String[] args) {

Product[] products = {

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

new Product(101, "Smartphone", "Electronics"),

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

new Product(102, "Headphones", "Accessories")

};

int targetId = 110;

int linIndex = *linearSearchProduct*(products, targetId);

if (linIndex != -1) {

System.*out*.println("Found with linear search: " +products[linIndex]);

} else {

System.*out*.println("Product not found (linear)");

}

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

int binIndex = *binarySearchProduct*(products, targetId);

if (binIndex != -1) {

System.*out*.println("Found with binary search: " +products[binIndex]);

} else {

System.*out*.println("Product not found (binary)");

}

}

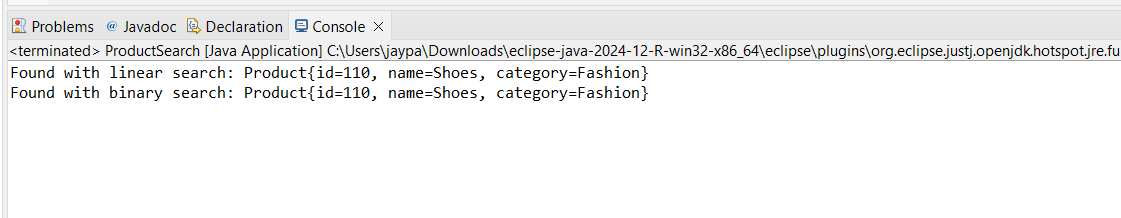
}

1. **Analysis & Discussion**

|  |  |  |  |
| --- | --- | --- | --- |
| **Algorithm** | **Best Case** | **Average Case** | **Worst Case** |
| Linear Search | O(1) | O(n) | O(n) |
| Binary Search | O(1) | O(Logn) | O(Logn) |

1. If we have a large number of products and can afford to keep the product list sorted by productId, then binary search is the most efficient. It scales much better as our data grows.
2. Binary search requires that the list is sorted. This is fine if product data is not changing too often.
3. If the data is frequently changing and we cannot keep it sorted easily, then linear search is simpler and better because sorting every time would be too costly.
4. In most e-commerce platforms where products do not change order every second and are already indexed by some ID or are kept in a database index, binary search is the most suitable

**Output :**



**Exercise 7: Financial Forecasting**

**Scenario:**

Developing a financial forecasting tool that predicts future values based on past data.

1. **Understand Recursive Algorithms:**

Recursion : Recursion is when a function calls itself to solve a problem.

How does Recursion simplify problems?

* It breaks a big problem into smaller, similar problems.
* The function solves one part and then calls itself to do the rest.
* This often results in less code and is easier to understand for problems like calculating factorials, searching in trees, or going through directories.
* Recursion is like telling someone to do part of a task and then repeat the same process for the remaining part.
* It keeps repeating until reaching a simple base case that can be solved directly.

1. **Create a method to calculate the future value using a recursive approach and Implement a recursive algorithm to predict future values based on past growth rates**

public class FutureValueCalculator {

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

if (years == 0) {

return presentValue;

}

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

}

public static void main(String[] args) {

double presentValue = 10000;

double rate = 0.05;

int years = 10;

double futureValue = *calculateFutureValue*(presentValue, rate, years);

System.*out*.println("Future value after " + years + " years: " + futureValue);

}

}

1. **Analysis**

Time Complexity of the recursive algorithm :

calculateFutureValue(presentValue, rate, years)

* Each call reduces years by 1 and then calls itself again.
* This happens years times, one call per year.

Time Complexity = O(years) i.e., O(n)

**Optimizing the recursive solution:**

**Using closed-form formula:**

We can directly get the answer using math:

FutureValue=PresentValue×(1+rate)^years

double futureValue = presentValue \* Math.pow(1 + rate, years);

Time complexity = O(1)

OUTPUT :

