**Week-1 Algorithms Data Structures**

1. **E-commerce Platform Search Function**

**Scenario:**

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

**Solution:**

**Understanding Asymptotic Notation:**

**Big O Notation:**

Big O notation is a way to describe the performance (time or space) of an algorithm as the input size grows. It focuses on the worst-case growth rate, helping developers understand how an algorithm behaves with larger inputs.

**Best, average and worst-case scenarios for search operation:**

|  |  |  |  |
| --- | --- | --- | --- |
| Cases | Description | Linear Search | Binary Search |
| Best | Element at the first position for linear or middle for binary. | O(1) | O(1) |
| Average | Element is somewhere in the middle. | O(n/2) | O(log n) |
| Worst | Element is not present in the array | O(n) | O(log n) |

**Set Up:**

**Code:**

**Product.java**

public 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 String toString()

{

String pro = productId+" - "+productName+" - "+category;

return pro;

}

}

**Implementation:**

**Searching.java**

import java.util.\*;

public class Searching

{

public static Product linearSearch(Product[] products, String name)

{

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

{

if (products[i].productName.equals(name))

{

return products[i];

}

}

return null;

}

public static Product binarySearch(Product[] products, String name)

{

int start=0;

int end = products.length-1;

while (start <= end)

{

int mid = (start + end)/2;

int result = products[mid].productName.compareTo(name);

if (result==0)

return products[mid];

else if (result<0)

start = mid+1;

else

end = mid-1;

}

return null;

}

public static void main(String[] args)

{

Product[] products =

{

new Product(1, "Laptop", "Electronics"),

new Product(2, "Mobile", "Electronics"),

new Product(3, "Shirt", "Fashion"),

new Product(4, "Pressure Cooker", "Cookware"),

new Product(5, "Frying Pan", "Cookware")

};

Product result1 = *linearSearch*(products, "Mobile");

if (result1!= null)

{

System.*out*.println("Linear Search Result: " + result1);

}

else

{

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

}

Arrays.*sort*(products, new Comparator<Product>()

{

public int compare(Product p1, Product p2)

{

return p1.productName.compareTo(p2.productName);

}

});

Product result2 = *binarySearch*(products, "Pressure Cooker");

if (result2 != null)

{

System.*out*.println("Binary Search Result: " + result2);

}

else

{

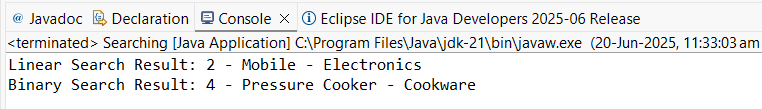
System.*out*.println("Binary Search Result: Product not found");

}

}

}

**Output:**



**Analysis:**

|  |  |
| --- | --- |
| Algorithm | Time Complexity |
| Linear Search | O(n) |
| Binary Search | O(log n) |

We can use Binary Search when the product list is very large. We can sort it only once and can search many times. It is suitable for fast searching of products.

We can use Linear Search when the product list is small. It does not need sorting.

1. **Financial Forecasting**

**Scenario:**

You are developing a financial forecasting tool that predicts future values based on past data.

**Understanding recursive algorithms:**

In Java, Recursion is a process in which a function calls itself directly or indirectly is called recursion and the corresponding function is called a recursive function. Using a recursive algorithm, certain problems can be solved quite easily. A few Java recursion examples are Inorder, Preorder, Postorder traversals, DFS of graph etc.

**Setup and Implementation:**

**Code:**

public class FinancialForecasting

{

public static double futureValue(double principal, double rate, int years)

{

if (years == 0)

{

return principal;

}

else

{

return *futureValue*(principal, rate, years - 1)\*(1 + rate);

}

}

public static void main(String[] args)

{

double principal = 10000;

double rate = 0.05;

int years = 10;

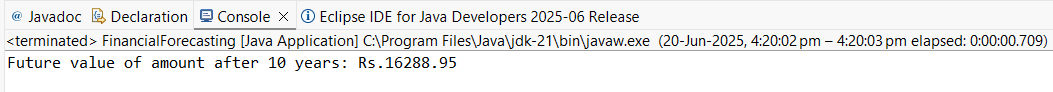
double result = *futureValue*(principal, rate, years);

System.*out*.printf("Future value of amount after %d years: Rs.%.2f\n", years, result);

}

}

**Output:**



**Analysis:**

This code has O(n) time complexity. Each recursive call reduces the n number of years by 1 and therefore the recursive call occurs n times. We can optimize this approach by using iteration or memoization.