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

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

**1. Understand Asymptotic Notation:**

1. **Explaination of Big O notation**

Given two functions f(n) and g(n), we say that f(n) is O(g(n)) if there exist constants c > 0 and n0 >= 0 such that f(n) <= c\*g(n) for all n >= n0.

Big O notation helps us understand how an algorithm behaves as the input size grows. Instead of telling us exactly how long it takes, it gives a rough idea of how fast the time or memory usage increases.

It’s super useful for comparing how efficient different algorithms or data structures are. Big O mainly looks at the **worst-case scenario**, showing the maximum amount of work an algorithm might need to do.

We write it as **O(f(n))**, where f(n) shows how the number of steps grows with the input size n.

1. **How it helps in analyzing algorithms**

**Shows how an algorithm scales:**  
Big O describes how the time or space needed by an algorithm increases as the input size grows.

**Compares algorithm efficiency:**  
It helps us compare which algorithm will perform better as the input gets larger — for example, O(n log n) is generally faster than O(n²).

**Focuses on the worst-case scenario:**  
Big O gives an upper bound, helping us prepare for the most demanding situations the algorithm might face.

**Predicts performance for large inputs:**  
Even without running the code, we can understand how it will behave with millions of data points.

**Ignores hardware and low-level details:**  
It abstracts away things like processor speed and focuses on logic, making it a reliable way to discuss performance.

**Guides design choices:**  
Helps developers select the most suitable algorithm or data structure for a given problem, especially when scalability matters.

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

**1.Best Case Scenario**

Best case scenario for searching is O(1).

It happens when the required element is found at the first attempt

**Example:** In a linear search, the element is the **first item** in the list.

**2.Average Case Scenario**

Average case scenario for searching is O(n).

If the required element is found in between in the array, then the time complexity is O(n/2),which is basically O(n).

**Example:** In a list of 10 items, the item might be the 5th one on average.

3.Worst case Scenario

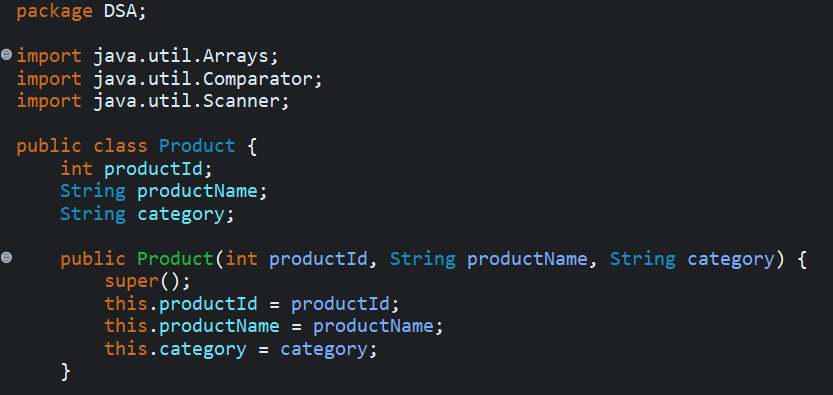
Worst case time complexity of searching is O(n).

If the required element in present at the last index or nth index, then the whole array is to be traversed.

**Example:** Linear search has to check every item before confirming.

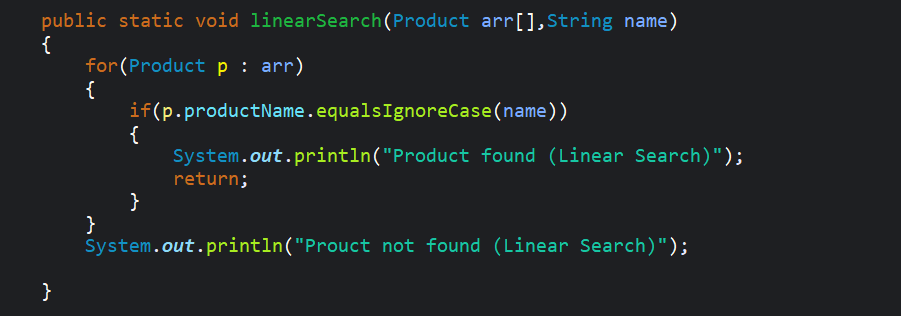
**2. Setup:**

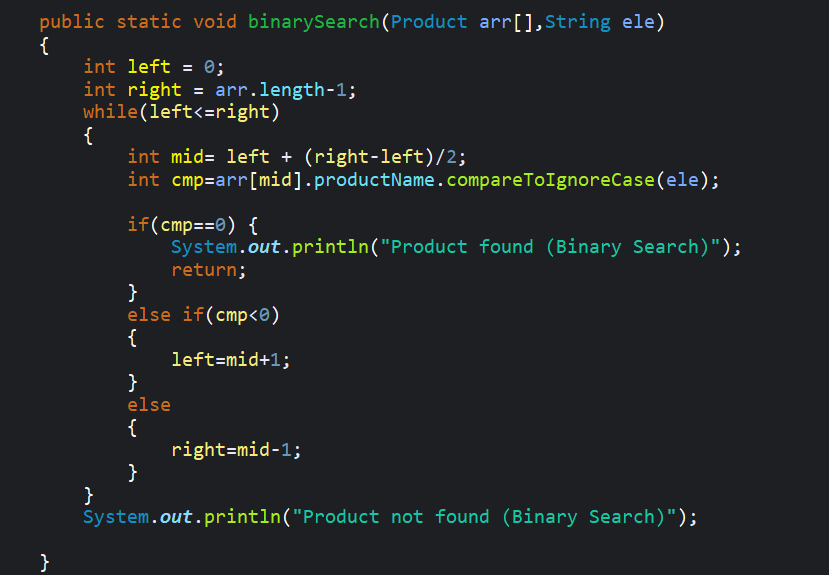
Create a class **Product** with attributes for searching, such as **productId, productName**, and **category**.



**3. Implementation:**

* Implement linear search and binary search algorithms.

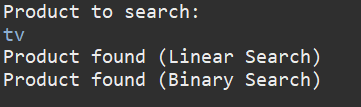




* Store products in an array for linear search and a sorted array for binary search.



**Code Output**

****

**Code**

package DSA;

import java.util.Arrays;

import java.util.Comparator;

import java.util.Scanner;

public class Product {

int productId;

String productName;

String category;

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

super();

this.productId = productId;

this.productName = productName;

this.category = category;

}

public static void linearSearch(Product arr[],String name)

{

for(Product p : arr)

{

if(p.productName.equalsIgnoreCase(name))

{

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

return;

}

}

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

}

public static void binarySearch(Product arr[],String ele)

{

int left = 0;

int right = arr.length-1;

while(left<=right)

{

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

int cmp=arr[mid].productName.compareToIgnoreCase(ele);

if(cmp==0) {

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

return;

}

else if(cmp<0)

{

left=mid+1;

}

else

{

right=mid-1;

}

}

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

}

public static void main(String[] args)

{

Product products[]= {

new Product(1,"TV","electronics"),

new Product(2,"Saree","fashion"),

new Product(3,"Chocolate","food"),

new Product(4,"Ball","sports"),

new Product(5,"Sofa","home")

};

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

System.***out***.println("Product to search:");

String n=sc.nextLine();

sc.close();

*linearSearch*(products,n);

//sorts products array in alphabetic order based on productName

Arrays.*sort*(products, Comparator.*comparing*(p -> p.productName));

*binarySearch*(products,n);

}

}

**4.Analysis:**

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

|  |  |  |
| --- | --- | --- |
| Property | Linear Search | Binary Search |
| Data Requirement | Works on unsorted or sorted data | Works on sorted data |
| Best Case | O(1)(element found at beginning) | O(1)(element found at mid) |
| Average case | O(n) | O(log n) |
| Worst Case | O(n)(element found at last or not found) | O(log n)(Search space is halved in each iteration) |
| Efficiency | Less efficient | Much faster and efficient |
| Search metho | Checks each element one by one | Repeatedly divides the search range in half |
| Use case | Suitable for small unsorted data | Suitable for large sorted data |

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

For a large numbers of users to use this e commerce application, the binary search is more efficient . It divides the search space into halves in each iteration which will hugely impact in time complexity reduction.

**Exercise 7: Financial Forecasting**

**Scenario:**

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

**Steps:**

**1. Understand Recursive Algorithms:**

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

Recursion is a programming technique where a function calls itself to solve a problem.

It keeps breaking the problem into smaller versions of itself, until it reaches a point where it stops — known as the base case.

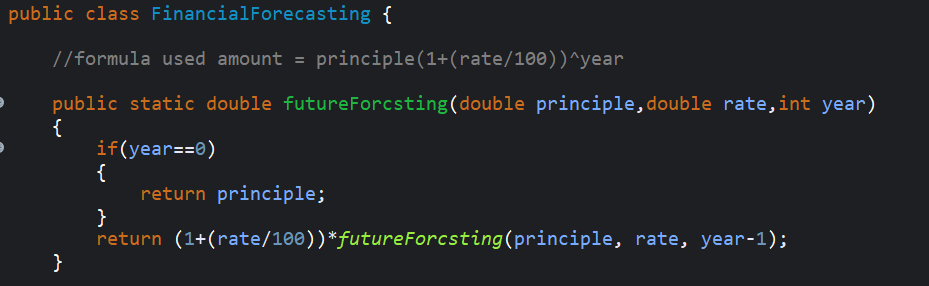
Recursion helps simplify problems that have a repeating or nested structure, where each part of the problem looks similar to the whole. It allows us to write cleaner, shorter, and more readable code for such problems.

A recursive function has two parts:

1. **Base Case** – when to stop recursion.
2. **Recursive Case** – the part where the function calls itself with a smaller input.

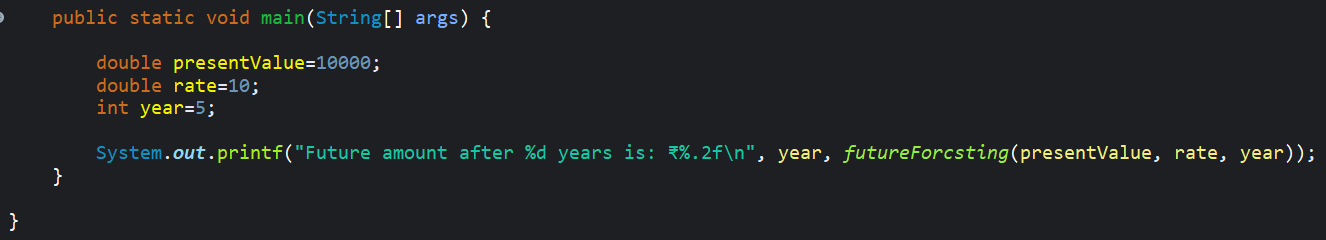
**2. Setup:**

* Create a method to calculate the future value using a recursive approach



**3.Implementation:**

* Implement a recursive algorithm to predict future values based on past growth rates.



**Code Output**

****

**Code**

package DSA;

public class FinancialForecasting {

//formula used amount = principle(1+(rate/100))^year

public static double futureForcsting(double principle,double rate,int year)

{

if(year==0)

{

return principle;

}

return (1+(rate/100))\**futureForcsting*(principle, rate, year-1);

}

public static void main(String[] args) {

double presentValue=10000;

double rate=10;

int year=5;

System.***out***.printf("Future amount after %d years is: ₹%.2f\n", year, *futureForcsting*(presentValue, rate, year));

}

}

**4. Analysis:**

* Discuss the time complexity of your recursive algorithm.
  + 1. In each step the calculation is reduced by 1 year. So the number of total calls are n.
    2. So the time complexity of this function is O(n).
* Explain how to optimize the recursive solution to avoid excessive computation.
  + 1. We can store the previously calculated results in a variable for better implementation.