**Exercise 2: 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.

**Big O Notation:**

Big O notation describes the upper bound of an algorithm’s time and space complexity in terms of input size n. We mainly consider the worst case scenario of the algorithm to find its time complexity in terms of Big O. Can be used to compare the efficiency of different algorithms or data structures.

**O(1)** -> Time is constant, no matter how large the input is.

**O(n)** -> Time grows linearly with input.

**O(logn)** -> Time grows slowly even if input is large.

**O(nlogn)** -> Time grows faster than linear but slower than quadratic.

**O(n2)** -> Time grows quadratically with input.

**O(2n)** -> Time doubles with each additional input.

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

**Best Case:** The item is found immediately, such as the item is the beginning element in search operation on the list. Minimum time taken.

**Average Case:** The item is found somewhere in the middle of the data. It takes moderate time to find the required item.

**Worst Case:** The item is not present or is the last element in the list. Requires scanning the entire list, so maximum time is taken.

**Product.java:**

**package** ecommerceSearchFunction;

**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() {

**return** "ID: " +productId+ ", Name: "+productName+", category: "+category;

}

}

**Search.java:**

**package** ecommerceSearchFunction;

**import** java.util.Arrays;

**import** java.util.Comparator;

**public** **class** Search {

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

name=name.toLowerCase();

**for**(Product p:products) {

**if**(p.productName.toLowerCase().equals(name)) {

**return** p;

}

}

**return** **null**;

}

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

name=name.toLowerCase();

**int** left=0,right=products.length-1;

**while**(left<=right) {

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

String temp=products[mid].productName.toLowerCase();

**if**(temp.equals(name)) {

**return** products[mid];

}

**else** **if**(temp.compareTo(name)<0) {

left=mid+1;

}

**else** {

right=mid-1;

}

}

**return** **null**;

}

**public** **static** **void** main(String[] args) {

Product[] products= {

**new** Product(1, "Television", "Electronics"),

**new** Product(2, "Book", "Stationary"),

**new** Product(3, "Mobile", "Electronics"),

**new** Product(4, "Sofa", "Home Decor"),

**new** Product(5, "T-Shirt", "Clothing")

};

System.***out***.println("Linear Search: "+*linearSearch*(products, "mobile"));

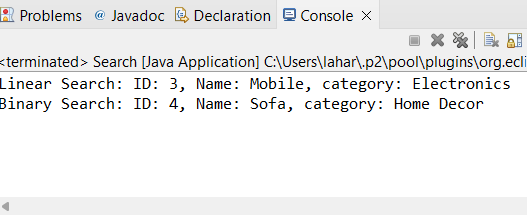
Arrays.*sort*(products, Comparator.*comparing*(p->p.productName.toLowerCase()));

System.***out***.println("Binary Search: "+*binarySearch*(products, "sofa"));

}

}

**Output:**

****

**Analysis:**

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| Algorithm | Best Case | Average Case | Worst Case | Space Complexity | Requires Sorted Data |
| Linear Search | O(1) | O(n) | O(n) | O(1) | No |
| Binary Search | O(1)(mid hit) | O(logn) | O(logn) | O(1) | Yes |

* Linear Search checks each item in an array one by one until match is found.
  + Best Case: First item matches.
  + Worst Case: Last item matches or item is not found at all.
  + Doesn’t require the array to be sorted.
* Binary Search splits the search space in half each time.
  + Much faster (O(logn)) compared to linear search for large data sets.
  + Requires data to be sorted.
  + Best for static datasets where sorting is done once and many searches happen afterward.

Binary Search is more suitable for an E-commerce Platform Search Function because

1. Binary Search is faster for larger datasets.
2. The number of comparisons grows logarithmically with number of products, this ensures fast and consistent performance even as data grows.
3. If data is pre-sorted binary search provides excellent speed.

**Exercise 7: Financial Forecasting**

**Scenario:**

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

**Recursion:**

1. Recursion is a programming technique where a function calls itself to solve a problem. Instead of solving the whole problem at once, recursion breaks it down into smaller sub problems that are easier to manage.
2. A typical recursive function has two key parts:
   1. Base Case:

The condition under which the recursion stops.

* 1. Recursive Case:

The function calls itself with a smaller or simpler input.

**FinancialForecasting.java:**

**package** financialForecasting;

**public** **class** FinancialForecast {

**public** **static** **double** futureValue(**double** initial, **double** growthRate, **int** years) {

**if**(years==0)

**return** initial;

**return** (1+growthRate)\**futureValue*(initial, growthRate, years-1);

}

**public** **static** **void** main(String[] args) {

**double** initial=10000;

**double** growthRate=0.1;

**int** years=3;

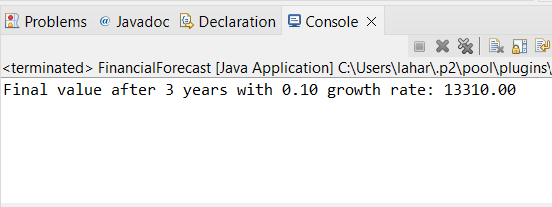
**double** finalValue=*futureValue*(initial, growthRate, years);

System.***out***.printf("Final value after %d years with %.2f growth rate: %.2f",years,growthRate,finalValue);;

}

}

**Output:**



The time complexity of the recursive financial forecasting algorithm is O(n), where n is number of years. The function calls itself once for every year until years==0. The space complexity is also O(n) due to the call stack.

A simple loop gives the same result without consuming stack memory. The time complexity of the iterative approach is O(n) and space complexity is O(1). There will be no risk of stack overflow. Faster and safer for large inputs.

**public** **static** **double** futureValueIterative(**double** initial, **double** growthRate, **int** years) {

**double** finalValue=initial;

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

finalValue\*=(1+growthRate);

}

**return** finalValue;

}

Can also use the inbuilt Math.pow() function for exponentiation, the time complexity is O(logn) internally (due to fast exponentiation). And space complexity is O(1).

**return** initial\*Math.*pow*(1+growthRate, years);