Best Practices for Runtime Analysis & Big O Notation

When designing algorithms, follow these best practices to ensure optimal time and space complexity.

🟢 General Algorithm Best Practices

1. **Choose the right data structure**
   - Use `HashMap` for O(1) lookups instead of `ArrayList` (O(N)).
   - Use `TreeSet` for ordered data retrieval with O(log N) complexity.
2. **Optimize loops & conditions**
   - Avoid unnecessary nested loops ($O(N^2)$ or worse).
   - Use **break/return** early to reduce iterations.
3. **Use sorting efficiently**
   - Prefer **Merge Sort** (O(N log N)) over **Bubble Sort** ($O(N^2)$).
   - If frequent sorting is needed, consider **TreeMap** (O(log N) per operation).
4. **Avoid excessive recursion**
   - Recursive Fibonacci ($O(2^n)$) is inefficient; use **Memoization (O(N))**.
   - Use **Tail Recursion** when applicable.
5. **Reduce redundant computations**
   - Store results of expensive operations using **caching/memoization**.
   - Example: Use **Dynamic Programming** to avoid recomputation.

🔵 Time & Space Complexity Optimization

1. **Precompute values when possible**
   - If computing the same value multiple times, store it in a **lookup table**.
2. **Use efficient searching methods**
   - Instead of Linear Search (O(N)), use **Binary Search (O(log N))** for sorted data.
3. **Use efficient iteration techniques**
   - Instead of `for (int i = 0; i < list.size(); i++)`, use **for-each loops**.
   - Example: `for (String s : list)`
4. **Optimize memory usage**
   - Use `StringBuilder` instead of `String` for concatenation.
   - Use **primitive types (int, double) over wrapper classes (Integer, Double)** when possible.
5. **Profile & benchmark performance**

- Use Java's `System.nanoTime()` or `JMH` (Java Microbenchmark Harness) to measure execution time.

1. Problem Statement: Search a Target in a Large Dataset

Objective:

Compare the performance of **Linear Search (O(N))** and **Binary Search (O(log N))** on different dataset sizes.

Approach:

1. **Linear Search:** Scan each element until the target is found.
2. **Binary Search:** Sort the data first (O(N log N)), then perform O(log N) search.

Comparative Analysis:

Dataset Size (N)	Linear Search (O(N))	Binary Search (O(log N))
1,000	1ms	0.01ms
10,000	10ms	0.02ms
1,000,000	1s	0.1ms

Expected Result:

Binary Search performs much better for large datasets, provided data is sorted.

Sol:

```
import java.util.Arrays;

import java.util.Random;

public class Main {

    public static void main(String[] args) {

        int[] sizes = {1000, 10000, 1000000};

        Random rand = new Random();
```

```
    for (int n : sizes) {
        int[] arr = new int[n];
        for (int i = 0; i < n; i++) {
            arr[i] = rand.nextInt(1000000);
        }
        long linearStartTime = System.nanoTime();
        linearSearch(arr, rand.nextInt(1000000));
        long linearEndTime = System.nanoTime();
        long linearTime = (linearEndTime - linearStartTime);
        Arrays.sort(arr);
        long binaryStartTime = System.nanoTime();
        binarySearch(arr, rand.nextInt(1000000));
        long binaryEndTime = System.nanoTime();
        long binaryTime = (binaryEndTime - binaryStartTime);
        System.out.println("Dataset Size: " + n);
        System.out.println("Linear Search Time: " + linearTime / 1000000.0 + "ms");
        System.out.println("Binary Search Time: " + binaryTime / 1000000.0 + "ms");
        System.out.println();
    }
}
public static void linearSearch(int[] arr, int target) {
    for (int i = 0; i < arr.length; i++) {
        if (arr[i] == target) {
            return;
```

```
        }
      }
    }
    public static int binarySearch(int[] arr, int target) {
      int left = 0, right = arr.length - 1;
      while (left <= right) {
        int mid = (left + right) / 2;
        if (arr[mid] == target) {
          return mid;
        } else if (arr[mid] < target) {
          left = mid + 1;
        } else {
          right = mid - 1;
        }
      }
      return -1;
    }
}
```

---

2. Problem Statement: Sorting Large Data Efficiently

Objective:

Compare sorting algorithms **Bubble Sort ($O(N^2)$), Merge Sort ($O(N \log N)$), and Quick Sort ($O(N \log N)$)**.

Approach:

1. **Bubble Sort:** Repeated swapping (inefficient for large data).

2. **Merge Sort:** Divide & Conquer approach (stable).
3. **Quick Sort:** Partition-based approach (fast but unstable).

Comparative Analysis:

Dataset Size (N)	Bubble Sort ($O(N^2)$)	Merge Sort (O(N log N))	Quick Sort (O(N log N))
1,000	50ms	5ms	3ms
10,000	5s	50ms	30ms
1,000,000	Unfeasible (>1hr)	3s	2s

Expected Result:

- **Bubble Sort** is impractical for large datasets.
- **Merge Sort & Quick Sort** perform well.

Sol:

```
import java.util.Arrays;

import java.util.Random;

public class Main {

    public static void main(String[] args) {

        int[] sizes = {1000, 10000, 1000000};

        Random rand = new Random();

        for (int n : sizes) {

            int[] arrBubble = generateRandomArray(n);

            int[] arrMerge = arrBubble.clone();

            int[] arrQuick = arrBubble.clone();

            long bubbleStartTime = System.nanoTime();

            bubbleSort(arrBubble);

            long bubbleEndTime = System.nanoTime();
```

```
        long bubbleTime = (bubbleEndTime - bubbleStartTime);
        long mergeStartTime = System.nanoTime();
        mergeSort(arrMerge, 0, n - 1);
        long mergeEndTime = System.nanoTime();
        long mergeTime = (mergeEndTime - mergeStartTime);
        long quickStartTime = System.nanoTime();
        quickSort(arrQuick, 0, n - 1);
        long quickEndTime = System.nanoTime();
        long quickTime = (quickEndTime - quickStartTime);
        System.out.println("Dataset Size: " + n);
        System.out.println("Bubble Sort Time: " + bubbleTime / 1000000.0 + "ms");
        System.out.println("Merge Sort Time: " + mergeTime / 1000000.0 + "ms");
        System.out.println("Quick Sort Time: " + quickTime / 1000000.0 + "ms");
        System.out.println();
    }
}
public static void bubbleSort(int[] arr) {
    int n = arr.length;
    for (int i = 0; i < n - 1; i++) {
        for (int j = 0; j < n - 1 - i; j++) {
            if (arr[j] > arr[j + 1]) {
                int temp = arr[j];
                arr[j] = arr[j + 1];
                arr[j + 1] = temp;
```

```
            }
        }
    }
}
public static void mergeSort(int[] arr, int left, int right) {
    if (left < right) {
        int mid = (left + right) / 2;
        mergeSort(arr, left, mid);
        mergeSort(arr, mid + 1, right);
        merge(arr, left, mid, right);
    }
}
public static void merge(int[] arr, int left, int mid, int right) {
    int n1 = mid - left + 1;
    int n2 = right - mid;
    int[] leftArr = new int[n1];
    int[] rightArr = new int[n2];
    System.arraycopy(arr, left, leftArr, 0, n1);
    System.arraycopy(arr, mid + 1, rightArr, 0, n2);
    int i = 0, j = 0, k = left;
    while (i < n1 && j < n2) {
        if (leftArr[i] <= rightArr[j]) {
            arr[k] = leftArr[i];
            i++;
```

```
        } else {
            arr[k] = rightArr[j];
            j++;
        }
        k++;
    }
    while (i < n1) {
        arr[k] = leftArr[i];
        i++;
        k++;
    }
    while (j < n2) {
        arr[k] = rightArr[j];
        j++;
        k++;
    }
}
public static void quickSort(int[] arr, int low, int high) {
    if (low < high) {
        int pivot = partition(arr, low, high);
        quickSort(arr, low, pivot - 1);
        quickSort(arr, pivot + 1, high);
    }
}
```

```
public static int partition(int[] arr, int low, int high) {
    int pivot = arr[high];
    int i = low - 1;
    for (int j = low; j < high; j++) {
        if (arr[j] <= pivot) {
            i++;
            int temp = arr[i];
            arr[i] = arr[j];
            arr[j] = temp;
        }
    }
    int temp = arr[i + 1];
    arr[i + 1] = arr[high];
    arr[high] = temp;
    return i + 1;
}
public static int[] generateRandomArray(int n) {
    Random rand = new Random();
    int[] arr = new int[n];
    for (int i = 0; i < n; i++) {
        arr[i] = rand.nextInt(1000000);
    }
    return arr;
}
```

}

---

3. Problem Statement: String Concatenation Performance

Objective:

Compare the performance of **String (O(N²)), StringBuilder (O(N)), and StringBuffer (O(N))** when concatenating a million strings.

Approach:

1. **Using `String` (Immutable, creates new object each time)**
2. **Using `StringBuilder` (Fast, mutable, thread-unsafe)**
3. **Using `StringBuffer` (Thread-safe, slightly slower than StringBuilder)**

Comparative Analysis:

Operations Count (N)	String (O(N²))	StringBuilder (O(N))	StringBuffer (O(N))
1,000	10ms	1ms	2ms
10,000	1s	10ms	12ms
1,000,000	30m (Unusable)	50ms	60ms

Expected Result:

- **StringBuilder & StringBuffer** are much more efficient than `String`.
- Use **StringBuilder** for single-threaded operations and **StringBuffer** for multi-threaded.

Sol:

```
public class Main {

    public static void main(String[] args) {

        String str = "Hello";

        StringBuilder sb = new StringBuilder("Hello");

        StringBuffer sbf = new StringBuffer("Hello");

        int[] sizes = {1000, 10000, 1000000};

        for (int n : sizes) {
```

```
        long stringStartTime = System.nanoTime();
        String resultString = concatenateUsingString(str, n);
        long stringEndTime = System.nanoTime();
        long stringTime = (stringEndTime - stringStartTime);
        long stringBuilderStartTime = System.nanoTime();
        StringBuilder resultStringBuilder = concatenateUsingStringBuilder(sb, n);
        long stringBuilderEndTime = System.nanoTime();
        long stringBuilderTime = (stringBuilderEndTime - stringBuilderStartTime);
        long stringBufferStartTime = System.nanoTime();
        StringBuffer resultStringBuffer = concatenateUsingStringBuffer(sbf, n);
        long stringBufferEndTime = System.nanoTime();
        long stringBufferTime = (stringBufferEndTime - stringBufferStartTime);
        System.out.println("Operations Count: " + n);
        System.out.println("String Time: " + stringTime / 1000000.0 + "ms");
        System.out.println("StringBuilder Time: " + stringBuilderTime / 1000000.0 + "ms");
        System.out.println("StringBuffer Time: " + stringBufferTime / 1000000.0 + "ms");
        System.out.println();
    }
}
public static String concatenateUsingString(String str, int n) {
    String result = str;
    for (int i = 0; i < n; i++) {
        result += " World";
    }
```

```
        return result;
    }
    public static StringBuilder concatenateUsingStringBuilder(StringBuilder sb, int n) {
        for (int i = 0; i < n; i++) {
            sb.append(" World");
        }
        return sb;
    }
    public static StringBuffer concatenateUsingStringBuffer(StringBuffer sbf, int n) {
        for (int i = 0; i < n; i++) {
            sbf.append(" World");
        }
        return sbf;
    }
}
```

---

4. Problem Statement: Large File Reading Efficiency

Objective:

Compare **FileReader (Character Stream)** and **InputStreamReader (Byte Stream)** when reading a large file (500MB).

Approach:

1. **FileReader:** Reads character by character (slower for binary files).
2. **InputStreamReader:** Reads bytes and converts to characters (more efficient).

Comparative Analysis:

File Size	FileReader Time	InputStreamReader Time

1MB	50ms	30ms
100MB	3s	1.5s
500MB	10s	5s

Expected Result:

- **InputStreamReader** is more efficient for large files.
- **FileReader** is preferable for text-based data.

Sol:

```
import java.io.*;

public class Main {

    public static void main(String[] args) {

        String filePath = "largeFile.txt";

        compareFileReading(filePath);

    }

    public static void compareFileReading(String filePath) {

        long fileReaderTime = readFileWithFileReader(filePath);

        long inputStreamReaderTime = readFileWithInputStreamReader(filePath);

        System.out.println("FileReader Time: " + fileReaderTime + " ms");

        System.out.println("InputStreamReader Time: " + inputStreamReaderTime + " ms");

    }

    public static long readFileWithFileReader(String filePath) {

        long startTime = System.currentTimeMillis();

        try (FileReader fileReader = new FileReader(filePath); BufferedReader bufferedReader =
new BufferedReader(fileReader)) {

            while (bufferedReader.read() != -1) {}

        } catch (IOException e) {
```

```
            e.printStackTrace();
        }
        return System.currentTimeMillis() - startTime;
    }
    public static long readFileWithInputStreamReader(String filePath) {
        long startTime = System.currentTimeMillis();
        try (InputStreamReader inputStreamReader = new InputStreamReader(new
FileInputStream(filePath)); BufferedReader bufferedReader = new
BufferedReader(inputStreamReader)) {
            while (bufferedReader.read() != -1) {}
        } catch (IOException e) {
            e.printStackTrace();
        }
        return System.currentTimeMillis() - startTime;
    }
}
```

---

5. Problem Statement: Recursive vs Iterative Fibonacci Computation

Objective:

Compare **Recursive (O(2^n))** vs **Iterative (O(N))** Fibonacci solutions.

Approach:
Recursive:

```
public static int fibonacciRecursive(int n) {
    if (n <= 1) return n;
    return fibonacciRecursive(n - 1) + fibonacciRecursive(n - 2);
}
```

Iterative:

```
public static int fibonacciIterative(int n) {
    int a = 0, b = 1, sum;
    for (int i = 2; i <= n; i++) {
        sum = a + b;
        a = b;
        b = sum;
    }
    return b;
}
```

Comparative Analysis:

Fibonacci (N)	Recursive ($O(2^n)$)	Iterative (O(N))
10	1ms	0.01ms
30	5s	0.05ms
50	Unfeasible (>1hr)	0.1ms

Expected Result:

- **Recursive approach** is infeasible for large values of N due to exponential growth.
- **The iterative approach** is significantly faster and memory-efficient.

Sol:

```
public class Main {

   public static void main(String[] args) {

      int n = 50;

      long recursiveTime = measureTime(() -> fibonacciRecursive(n));

      long iterativeTime = measureTime(() -> fibonacciIterative(n));

      System.out.println("Recursive Time: " + recursiveTime + " ms");

      System.out.println("Iterative Time: " + iterativeTime + " ms");

   }
```

```
    public static long measureTime(Runnable method) {
        long startTime = System.currentTimeMillis();
        method.run();
        return System.currentTimeMillis() - startTime;
    }
    public static int fibonacciRecursive(int n) {
        if (n <= 1) return n;
        return fibonacciRecursive(n - 1) + fibonacciRecursive(n - 2);
    }
    public static int fibonacciIterative(int n) {
        int a = 0, b = 1, sum;
        for (int i = 2; i <= n; i++) {
            sum = a + b;
            a = b;
            b = sum;
        }
        return b;
    }
}
```

---

6. Problem Statement: Comparing Different Data Structures for Searching

Objective:

Compare **Array (O(N)), HashSet (O(1)), and TreeSet (O(log N))** for searching elements.

Approach:

1. **Array:** Linear search (O(N)).
2. **HashSet:** Uses hashing (O(1) on average).
3. **TreeSet:** Balanced BST (O(log N)).

Comparative Analysis:

Dataset Size (N)	Array Search (O(N))	HashSet Search (O(1))	TreeSet Search (O(log N))
1,000	1ms	0.01ms	0.1ms
100,000	100ms	0.01ms	10ms
1,000,000	1s	0.01ms	20ms

Expected Result:

- **HashSet** is fastest for lookups but requires extra memory.
- **TreeSet** maintains order but is slightly slower than HashSet.

Sol:

```
import java.util.*;

public class Main {

    public static void main(String[] args) {

        int[] array = new int[1000000];

        HashSet<Integer> hashSet = new HashSet<>();

        TreeSet<Integer> treeSet = new TreeSet<>();

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

            array[i] = i;

            hashSet.add(i);

            treeSet.add(i);

        }

        int target = 500000;
```

```
    long arraySearchTime = measureTime(() -> arraySearch(array, target));
    long hashSetSearchTime = measureTime(() -> hashSetSearch(hashSet, target));
    long treeSetSearchTime = measureTime(() -> treeSetSearch(treeSet, target));
    System.out.println("Array Search Time: " + arraySearchTime + " ms");
    System.out.println("HashSet Search Time: " + hashSetSearchTime + " ms");
    System.out.println("TreeSet Search Time: " + treeSetSearchTime + " ms");
}
public static long measureTime(Runnable method) {
    long startTime = System.currentTimeMillis();
    method.run();
    return System.currentTimeMillis() - startTime;
}
public static boolean arraySearch(int[] array, int target) {
    for (int num : array) {
        if (num == target) return true;
    }
    return false;
}
public static boolean hashSetSearch(HashSet<Integer> hashSet, int target) {
    return hashSet.contains(target);
}
public static boolean treeSetSearch(TreeSet<Integer> treeSet, int target) {
    return treeSet.contains(target);
}
```

```
}
```