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.

```
import java.util.*;

class LinearAndBinary{

    public static void main(String[] args){

        Scanner sc = new Scanner(System.in);

        executionTime(1000, 999);
        executionTime(10000, 9999);
        executionTime(1000000, 99999);
    }
```

```java
    public static void executionTime(int size, int target){


        int[] arr = new int[size];

        for(int i=0;i<arr.length;i++)
            arr[i] = i+1;

        long ini = System.nanoTime();

        binarySearch(arr, target);

        System.out.println("Time taken for binary search on a dataset of 
"+size+" elements is : "+(System.nanoTime()-ini)/1000000.0+" milli seconds");

        ini = System.nanoTime();

        linearSearch(arr, target);

        System.out.println("Time taken for linear search on a dataset of 
"+size+" elements is : "+(System.nanoTime()-ini)/1000000.0+" milli seconds");

    }

    public static boolean binarySearch(int[] arr, int target){

        int low = 0, high = arr.length-1;
        while(low <=high){

            int mid = (low+high)/2;

            if(arr[mid] == target)
                return true;
            else if(arr[mid] > target)
                high = mid - 1;
            else
                low = mid + 1;
        }
        return false;

    }

    public static boolean linearSearch(int[] arr, int target){
```

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

            if(arr[i] == target)
                return true;

        }
        return false;

    }
}
```

---

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.

```
import java.util.*;

class BubbleMergeQuick{

    public static void main(String[] args){

        executionTime(1000);
        executionTime(10000);
        executionTime(1000000);


    }

    public static void executionTime(int size){

        int[] arr = new int[size];
        for(int i=0;i<size;i++)
            arr[i] = (int)(size*Math.random());

        long ini = System.nanoTime();
        bubbleSort(arr);
        System.out.println("Time taken by bubble sort for size "+size+" is : 
"+(System.nanoTime()-ini)/1000000.0+" milli seconds");

        ini = System.nanoTime();
        mergeSort(arr, 0, arr.length-1);
        System.out.println("Time taken by merge sort "+size+" is : 
"+(System.nanoTime()-ini)/1000000.0+" milli seconds");

        ini = System.nanoTime();
        quickSort(arr, 0, arr.length-1);
        System.out.println("Time taken by quick sort "+size+" is : 
"+(System.nanoTime()-ini)/1000000.0+" milli seconds");

    }

    public static void bubbleSort(int[] arr){

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

            boolean swaps = false;
```

```java
            for(int j=1;j<arr.length;j++){
                if(arr[j]<arr[j-1]){
                    int temp = arr[j];
                    arr[j] = arr[j-1];
                    arr[j-1] = temp;
                    swaps = true;
                }

            }
            if(!swaps)
                break;

        }

    }
    public static void mergeSort(int[] arr, int left, int right){

        if(left>=right)
            return;

        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[] l = new int[mid-left+1];
        int[] r = new int[right-mid];

        int li=0, ri=0;
        for(int i=left;i<=mid;i++)
            l[li++] = arr[i];
        for(int i=mid+1;i<=right;i++)
            r[ri++] = arr[i];

        li = 0;
        ri = 0;
        int in=0;

        while(li<l.length&&ri<r.length){
```

```
        if(l[li]<=r[ri])
            arr[in++] = l[li++];
        else
            arr[in++] = r[ri++];

    }

    while(li<l.length)
        arr[in++] = l[li++];
    while(ri<r.length)
        arr[in++] = r[ri++];

}

public static void quickSort(int[] arr, int left, int right){

    if(left >= right)
        return;

    int ind = partition(arr, left, right);

    quickSort(arr, left, ind-1);
    quickSort(arr, ind+1, right);

}

public static int partition(int[] arr, int left, int right){

    int idx = left;

    for(int i=left;i<right;i++){

        if(arr[right] > arr[i]){
            int temp = arr[i];
            arr[i] = arr[idx];
            arr[idx] = temp;
            idx++;
        }

    }
    int temp = arr[idx];
    arr[idx] = arr[right];
    arr[right] = temp;
```

```
        return idx;
    }
}
```

---

3. Problem Statement: String Concatenation Performance

Objective:

Compare the performance of **String ($O(N^2)$), 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 ($O(N^2)$)	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.

```
import java.util.*;

class StringConcatenationPerformance{

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

```
        String str = "";
        StringBuilder stb1 = new StringBuilder();
        StringBuffer stb2 = new StringBuffer();

        long ini = System.nanoTime();
        for(int i=0;i<1000;i++)
            str += "hello";
        System.out.println("Time taken by String to concatenate 1000 strings is
: "+(System.nanoTime()-ini)/1000000.0+" milli seconds");

        ini = System.nanoTime();
        for(int i=0;i<1000;i++)
            stb1.append("hello");
        System.out.println("Time taken by String Builder to concatenate 1000
strings is : "+(System.nanoTime()-ini)/1000000.0+" milli seconds");

        ini = System.nanoTime();
        for(int i=0;i<1000;i++)
            stb2.append("hello");
        System.out.println("Time taken by String Buffer to concatenate 1000
strings is : "+(System.nanoTime()-ini)/1000000.0+" milli seconds");


        ini = System.nanoTime();
        for(int i=0;i<10000;i++)
            str += "hello";
        System.out.println("Time taken by String to concatenate 10000 strings
is : "+(System.nanoTime()-ini)/1000000.0+" milli seconds");

        ini = System.nanoTime();
        for(int i=0;i<10000;i++)
            stb1.append("hello");
        System.out.println("Time taken by String Builder to concatenate 10000
strings is : "+(System.nanoTime()-ini)/1000000.0+" milli seconds");

        ini = System.nanoTime();
        for(int i=0;i<10000;i++)
            stb2.append("hello");
        System.out.println("Time taken by String Buffer to concatenate 10000
strings is : "+(System.nanoTime()-ini)/1000000.0+" milli seconds");


        ini = System.nanoTime();
        for(int i=0;i<1000000;i++)
            str += "hello";
```

```
        System.out.println("Time taken by String to concatenate 1000000 strings
is : "+(System.nanoTime()-ini)/1000000.0+" milli seconds");

        ini = System.nanoTime();
        for(int i=0;i<1000000;i++)
            stb1.append("hello");
        System.out.println("Time taken by String Builder to concatenate 1000000
strings is : "+(System.nanoTime()-ini)/1000000.0+" milli seconds");

        ini = System.nanoTime();
        for(int i=0;i<1000000;i++)
            stb2.append("hello");
        System.out.println("Time taken by String Buffer to concatenate 1000000
strings is : "+(System.nanoTime()-ini)/1000000.0+" milli seconds");

    }


}
```

---

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.

```
import java.util.*;
import java.io.*;

class LargeFileReadingEfficiency{

    public static void main(String[] args){

        exectionTime("1mbfile.txt");
        exectionTime("100mbfile.txt");
        exectionTime("500mbfile.txt");

    }

    public static void exectionTime(String path){

        try{
            FileReader f1 = new FileReader(path);

            FileInputStream fi1 = new FileInputStream(path);
            InputStreamReader i1 = new InputStreamReader(fi1);

            long ini = System.nanoTime();
            int inp = -1;
            while((inp = f1.read()) != -1 ){}

            System.out.println("Time taken for the execution of File Reader for
file "+path+" is : "+(System.nanoTime()-ini)/1000000.0+" milli seconds");

            ini = System.nanoTime();
            while((inp = i1.read()) != -1){}

            System.out.println("Time taken for the execution of
InputStreamReader is for file "+path+" is :
"+(System.nanoTime()-ini)/1000000.0+" milli seconds");
        }
        catch(FileNotFoundException e){
            System.out.println(e);
        }
        catch(IOException e){
            System.out.println(e);
```

```
        }


    }


}
```

---

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.

```java
import java.util.*;

class RecursiveVsIterativeFibonacci{

    public static void main(String[] args){

        executionTime(10);
        executionTime(30);
        executionTime(50);

    }

    public static void executionTime(int n){

        long ini = System.nanoTime();

        System.out.println(iterative(n));
        System.out.println("Time taken for the execution of iterative fibonacci for value : "+n+" is : "+(System.nanoTime()-ini)/1000000.0+" milli seconds");

        ini = System.nanoTime();
        System.out.println(recursive(n));
        System.out.println("Time taken for the execution of recursive fibonacci for value : "+n+" is : "+(System.nanoTime()-ini)/1000000.0+" milli seconds");

    }

    public static int iterative(int n){

        int a=0, b=1;
        for(int i=2;i<=n;i++){
            int c = a+b;
            a = b;
```

```
            b = c;
        }
        return b;

    }

    public static int recursive(int n){

        if(n == 1 || n == 0)
            return n;

        return recursive(n-1)+recursive(n-2);

    }


}
```

---

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.

```
import java.util.*;

class ArrayHashSetTreeSet{

    public static void main(String[] args){

        executionTime(1000, 500);
        executionTime(100000, 50000);
        executionTime(1000000, 500000);


    }
    public static void executionTime(int n, int target){

        int[] arr = new int[n];
        HashSet<Integer> hs = new HashSet<>();
        TreeSet<Integer> ts = new TreeSet<>();

        for(int i=0;i<n;i++){
            arr[i] = i+1;
            hs.add(i+1);
            ts.add(i+1);
        }

        long ini = System.nanoTime();
        for(int i=0;i<arr.length;i++){
            if(arr[i] == target)
                break;
        }
        System.out.println("Time taken by array to search target in size "+n+"
is :"+(System.nanoTime()-ini)/1000000.0+" milli seconds");

        ini = System.nanoTime();
        hs.contains(target);
        System.out.println("Time taken by HashSet to search target size "+n+"
is :"+(System.nanoTime()-ini)/1000000.0+" milli seconds");

        ini = System.nanoTime();
        ts.contains(target);
        System.out.println("Time taken by TreeSet to search target size "+n+"
```

```
is :"+(System.nanoTime()-ini)/1000000.0+" milli seconds");
    }



}
```