ALGORITHMS

An algorithm is a step-by-step set of instructions or a set of rules designed to perform a specific task or solve a particular problem. It is a well-defined procedure that takes input, processes it, and produces the desired output. Algorithms are used in various fields, including computer science, mathematics, and everyday problem-solving.

In the context of computer science, algorithms are crucial for designing computer programs and solving computational problems efficiently. They serve as the blueprint for solving a problem, specifying the exact steps that need to be taken to reach the solution. Algorithms can range from simple and straightforward to complex and sophisticated, depending on the nature of the problem they are intended to address.

Key characteristics of algorithms include:

1. Input: Algorithms take input data or parameters.

2. Output: They produce output or a result.

3. Definiteness: Each step in the algorithm must be precisely defined.

4. Finiteness: The algorithm must terminate after a finite number of steps.

5. Effectiveness: Every step in the algorithm must be executable and should contribute to solving the problem.

6. Generality: Algorithms should be designed to solve a general class of problems rather than specific instances.

Algorithms play a fundamental role in the field of computer science, shaping the development of software and influencing the efficiency of various computational processes.

List of Some Algorithms

Creating a comprehensive list of algorithms is a challenging task, as there are numerous algorithms designed for various purposes in computer science. However, here's a list of some common algorithms that cover a broad range of topics within the field:

1. **Binary Search**
2. **Bubble Sort**
3. **QuickSort**
4. **Merge Sort**
5. **Insertion Sort**
6. **Selection Sort**
7. **Depth-First Search (DFS)**
8. **Breadth-First Search (BFS)**
9. **Dijkstra's Algorithm**
10. **A* Search Algorithm**
11. **Floyd-Warshall Algorithm**
12. **Dynamic Programming**
13. **Greedy Algorithms**
14. **Kruskal's Algorithm**
15. **Prim's Algorithm**
16. **Heap Sort**
17. **Radix Sort**
18. **Hashing (Hash Tables)**
19. **Linear Search**

20. KMP (Knuth-Morris-Pratt) Algorithm
21. Rabin-Karp Algorithm
22. Topological Sort
23. Minimum Spanning Tree
24. Bellman-Ford Algorithm
25. Ford-Fulkerson Algorithm (Max Flow)
26. Euclidean Algorithm (GCD)
27. RSA Algorithm (Cryptography)
28. Quicksort
29. B-Trees
30. Trie Data Structure
31. AVL Trees
32. Red-Black Trees
33. Bloom Filters
34. Suffix Trees
35. Monte Carlo Algorithm
36. Las Vegas Algorithm
37. Strassen's Matrix Multiplication
38. Boyer-Moore Algorithm
39. Graph Isomorphism Algorithm
40. K-means Clustering
41. PageRank Algorithm
42. Genetic Algorithms
43. Simulated Annealing

44. Traveling Salesman Problem (TSP) Algorithms

45. Fisher-Yates Shuffle

46. RSA Encryption Algorithm

47. Huffman Coding

48. Dinic's Algorithm (Maximum Flow)

49. Turing Machine (not an algorithm but a theoretical concept)

50. Brent's Cycle Detection Algorithm

This list includes algorithms related to sorting, searching, graph algorithms, dynamic programming, cryptography, and more. Keep in mind that there are many more algorithms, and their relevance can depend on specific application domains and problem-solving contexts.

Time complexity

Time complexity is a measure of the amount of time an algorithm takes to complete as a function of the size of the input. It quantifies the efficiency of an algorithm in terms of the time it requires to process the input data.

Because there are various ways to solve a problem, there must be a way to evaluate these solutions or algorithms in terms of performance and efficiency (the time it will take for your algorithm to run/execute and the total amount of memory it will consume).

This is critical for programmers to ensure that their applications run properly and to help them write clean code.

This is where Big O Notation enters the picture. **Big O Notation is a metric for determining the efficiency of an algorithm. It allows you to estimate how long your code will run on different sets of inputs and measure how effectively your code scales as the size of your input increases.**

Space complexity

Space complexity is a measure of the amount of memory or space an algorithm requires to solve a specific problem as a function of the input size. It is an important aspect of algorithm analysis, along with time complexity. Space complexity is often expressed in big O notation.

There are several components contributing to the space complexity of an algorithm:

1. Fixed Memory Usage (Constant Space): Some algorithms use a constant amount of memory regardless of the input size. These are typically represented as O(1), meaning the space requirements remain constant.

2. Variable Memory Usage (Linear Space): If the amount of memory used grows linearly with the size of the input, it is represented as O(n), where n is the input size.

3. Logarithmic Space: Algorithms with space complexity proportional to the logarithm of the input size are represented as O(log n). These algorithms often involve divide and conquer strategies.

4. Quadratic or Polynomial Space: Some algorithms use space that is a polynomial function of the input size, such as O(n^2) or O(n^k).

It's important to consider both time and space complexity when analyzing algorithms. In certain applications, minimizing space usage might be critical, especially in resource-constrained environments.

For example, if an algorithm uses an array to store data, and the size of the array is directly proportional to the input size, the space complexity could be O(n). If the algorithm uses a fixed-size array regardless of the input, the space complexity is O(1).

Keep in mind that space complexity analysis doesn't always consider the space used by the input itself, as that is often considered part of the input rather than the algorithm's space complexity.

Big O Notation

Big O, also known as Big O notation, represents an algorithm's worst-case complexity. It uses algebraic terms to describe the complexity of an algorithm.

Big O defines the runtime required to execute an algorithm by identifying how the performance of your algorithm will change as the input size grows. But it does not tell you how fast your algorithm's runtime is.

Big O notation measures the efficiency and performance of your algorithm using time and space complexity.

In big-O notation, time complexity is expressed in terms of the upper bound of the growth rate of the algorithm's running time concerning the size of the input. The notation O(f(n)) represents an upper bound on the time complexity, where "f(n)" is a mathematical function describing the growth rate and "n" is the size of the input.

Common time complexities include:

1. Constant Time (O(1)):

- The algorithm's running time remains constant regardless of the size of the input. Example: accessing an element in an array.

2. Logarithmic Time (O(log n)):

- The running time grows logarithmically as the size of the input increases. Example: binary search in a sorted array.

3. Linear Time (O(n)):

- The running time is directly proportional to the size of the input. Example: linear search in an unsorted array.

4. Linearithmic Time (O(n log n)):

- Common in efficient sorting algorithms like merge sort and heap sort.

5. Quadratic Time (O(n^2)):

- The running time is proportional to the square of the size of the input. Example: bubble sort.

6. Cubic Time (O(n^3)):

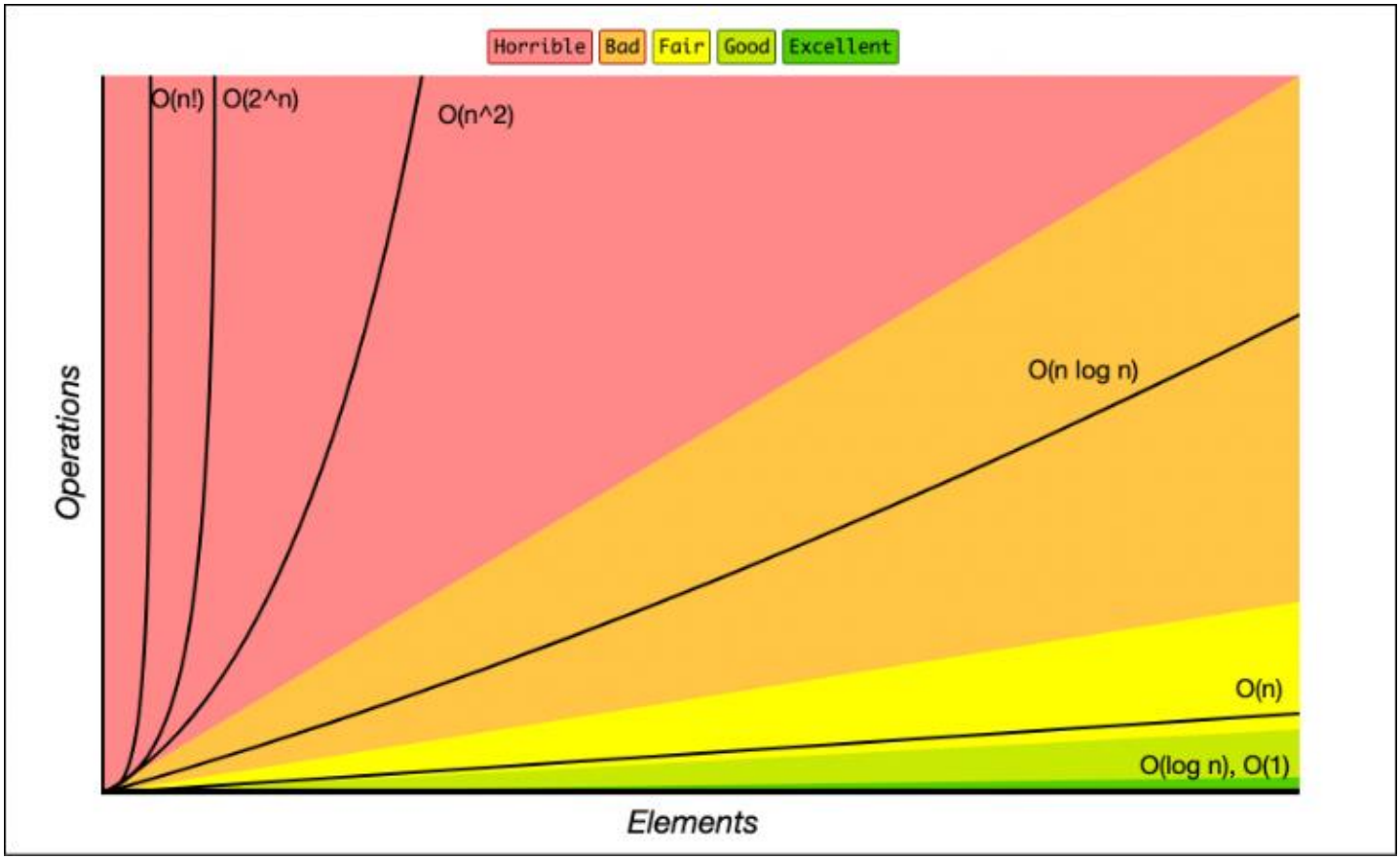
- The running time is proportional to the cube of the size of the input.

7. Exponential Time (O(2^n)):

- The running time grows exponentially with the size of the input. Often associated with inefficient recursive algorithms.

Understanding and analyzing time complexity helps developers choose efficient algorithms for solving problems, especially when dealing with large datasets. It provides a high-level understanding of how the algorithm's performance scales with input size, which is crucial in designing and optimizing algorithms for various applications.

Big_O_Complexity_Chart



The Big O chart above shows that O(1), which stands for constant time complexity, is the best. This implies that your algorithm processes only one statement without any iteration. Then there's O(log n), which is good, and others like it, as shown below:

O(1) - Excellent/Best

O(log n) - Good

O(n) - Fair

O(n log n) - Bad

O(n^2), O(2^n) and O(n!) - Horrible/Worst

You now understand the various time complexities, and you can recognize the best, good, and fair ones, as well as the bad and worst ones (always avoid the bad and worst time complexity).

When your calculation is not dependent on the input size, it is a constant time complexity (O(1)).

When the input size is reduced by half, maybe when iterating, handling recursion, or whatsoever, it is a logarithmic time complexity (O(log n)).

When you have a single loop within your algorithm, it is linear time complexity (O(n)).

When you have nested loops within your algorithm, meaning a loop in a loop, it is quadratic time complexity (O(n^2)).

When the growth rate doubles with each addition to the input, it is exponential time complexity (O2^n).

Consider: Big_O_TimeComplexityDemo.java

1) Constant Time: O(1)

When your algorithm is not dependent on the input size n, it is said to have a constant time complexity with order O(1). This means that the run time will always be the same regardless of the input size.

For example, if an algorithm is to return the first element of an array. Even if the array has 1 million elements, the time complexity will be constant if you use this approach:

```
public static int constantTime_firstElement(int[] array) {   return array[0];  }
```

The function above will require only one execution step, meaning the function is in constant time with time complexity O(1).

But there are various ways to achieve a solution in programming. Another programmer might decide to first loop through the array before returning the first element:

```
public static int linearTime_firstElement(int[] array) {
        for (int i = 0; i < array.length;) {
           return array[0];
        }
return -1; // This line is unreachable in this example but added for completeness
}
```

there is a loop, this is no longer constant time but now linear time with the time complexity O(n).

2) Linear Time: O(n)

when the running time of an algorithm increases linearly with the size of the input. This means that when a function has an iteration that iterates over an input size of n, it is said to have a time complexity of order O(n).

For example, if an algorithm is to return the factorial of any inputted number. This means if you input 5 then you are to loop through and multiply 1 by 2 by 3 by 4 and by 5 and then output 120:

```
public static int linearTime_calcFactorial (int n) {
        int factorial = 1;
        for (int i = 2; i <= n; i++) {
           factorial *= i;
        }
        return factorial;
    }
```

The fact that the runtime depends on the input size means that the time complexity is linear with the order O(n).

3) Logarithm Time: O(log n)

This is similar to linear time complexity, except that the runtime does not depend on the input size but rather on half the input size. When the input size decreases on each iteration or step, an algorithm is said to have logarithmic time complexity.

This method is the second best because your program runs for half the input size rather than the full size. After all, the input size decreases with each iteration.

A great example is binary search functions, which divide your sorted array based on the target value.

For example, suppose you use a binary search algorithm to find the index of a given element in an array:

```
public static int linearTime_binarySearch(int[] array, int target) {
        int firstIndex = 0;
        int lastIndex = array.length - 1;
        while (firstIndex <= lastIndex) {
            int middleIndex = (firstIndex + lastIndex) / 2;
            if (array[middleIndex] == target) {
               return middleIndex;
            }
            if (array[middleIndex] > target) {
               lastIndex = middleIndex - 1;
            } else {
               firstIndex = middleIndex + 1;
            }
         }
           return -1;
        }
```

In the code above, since it is a binary search, you first get the middle index of your array, compare it to the target value, and return the middle index if it is equal. Otherwise, you must check if the target value is greater or less than the middle value to adjust the first and last index, reducing the input size by half.

Because for every iteration the input size reduces by half, the time complexity is logarithmic with the order O(log n).

4) Quadratic Time: O(n^2)

When you perform nested iteration, meaning having a loop in a loop, the time complexity is quadratic, which is horrible.

A perfect way to explain this would be if you have an array with n items. The outer loop will run n times, and the inner loop will run n times for each iteration of the outer loop, which will give total n^2 prints. If the array has ten items, ten will print 100 times (10^2).

In this example where you compare each element in an array to output the index when two elements are similar:

```
public static String quadraticTime_matchElements(char[] array) {
    for (int i = 0; i < array.length; i++) {
        for (int j = 0; j < array.length; j++) {
            if (i != j && array[i] == array[j]) {
                return "Match found at " + i + " and " + j;
            }
        }
    }
    return "No matches found 😔";
}
```

In the example above, there is a nested loop, meaning that the time complexity is quadratic with the order O(n^2).

5) Exponential Time: O(2^n)

You get exponential time complexity when the growth rate doubles with each addition to the input (n), often iterating through all subsets of the input elements. Any time an input unit increases by 1, the number of operations executed is doubled.

The recursive Fibonacci sequence is a good example. Assume you're given a number and want to find the nth element of the Fibonacci sequence.

The Fibonacci sequence is a mathematical sequence in which each number is the sum of the two preceding numbers, where 0 and 1 are the first two numbers. The third number in the sequence is 1, the fourth is 2, the fifth is 3, and so on... (0, 1, 1, 2, 3, 5, 8, 13, …).

This means that if you pass in 6, then the 6th element in the Fibonacci sequence would be 8:

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

In the code above, the algorithm specifies a growth rate that doubles every time the input data set is added. This means the time complexity is exponential with an order O(2^n).

6) Linearithmic Time (O(n log n)):

This means that the running time of an algorithm is proportional to n times log n, where n is the size of the input. This is a common time complexity for algorithms that divide the input into smaller parts and then process them recursively, such as merge sort or quick sort. For example, if an algorithm is to sort an array of numbers using merge sort, it will split the array into two halves, sort each half recursively, and then merge the two sorted halves. The splitting and merging steps take O(n) time, and the recursive calls happen log n times, so the overall time complexity is O(n log n).

```
public static void mergeSort(int[] arr, int low, int high) {
  if (low < high) {
    // Find the middle point
    int mid = (low + high) / 2;
    // Sort first and second halves
    mergeSort(arr, low, mid);
    mergeSort(arr, mid + 1, high);
    // Merge the sorted halves
    merge(arr, low, mid, high);
  }
}
public static void merge(int[] arr, int low, int mid, int high) {
  // Find sizes of two subarrays to be merged
  int n1 = mid - low + 1;
  int n2 = high - mid;
  // Create temp arrays
  int[] L = new int[n1];
```

```
int[] R = new int[n2];
// Copy data to temp arrays
for (int i = 0; i < n1; i++)
  L[i] = arr[low + i];
for (int j = 0; j < n2; j++)
  R[j] = arr[mid + 1 + j];
// Merge the temp arrays
// Initial indexes of first and second subarrays
int i = 0, j = 0;
// Initial index of merged subarray
int k = low;
while (i < n1 && j < n2) {
  if (L[i] <= R[j]) {
    arr[k] = L[i];
    i++;
  } else {
    arr[k] = R[j];
    j++;
  }
  k++;
}
// Copy remaining elements of L[] if any
while (i < n1) {
  arr[k] = L[i];
```

```
    i++;
    k++;
  }
  // Copy remaining elements of R[] if any
  while (j < n2) {
    arr[k] = R[j];
    j++;
    k++;
  }
}
```

7) Cubic Time (O(n^3)):

This means that the running time of an algorithm is proportional to n cubed, where n is the size of the input. This is a high time complexity that indicates that the algorithm is very slow and inefficient. This could be the result of applying an O(n^2) algorithm to n items, or applying an O(n) algorithm to n^2 items, or having three nested loops that iterate over the input. For example, if an algorithm is to multiply two square matrices of size n x n, it will need three nested loops to compute each element of the product matrix, so the overall time complexity is O(n^3).

```
public static int[][] matrixMultiply(int[][] A, int[][] B) {
  // Assume A and B are square matrices of size n x n
  int n = A.length;
  // Create a new matrix to store the product
  int[][] C = new int[n][n];
```

```
 // Loop through each row of A
 for (int i = 0; i < n; i++) {
  // Loop through each column of B
  for (int j = 0; j < n; j++) {
   // Initialize the element C[i][j] to zero
   C[i][j] = 0;
   // Loop through each element of A[i] and B[j]
   for (int k = 0; k < n; k++) {
    // Add the product of A[i][k] and B[k][j] to C[i][j]
    C[i][j] += A[i][k] * B[k][j];
   }
  }
 }
 // Return the product matrix
 return C;
}
```

The running time grows rapidly with the input size, making it less efficient for larger datasets.