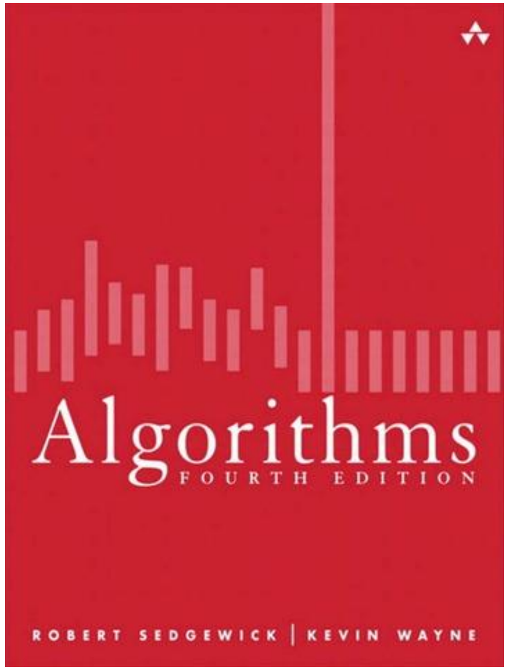
KMITL

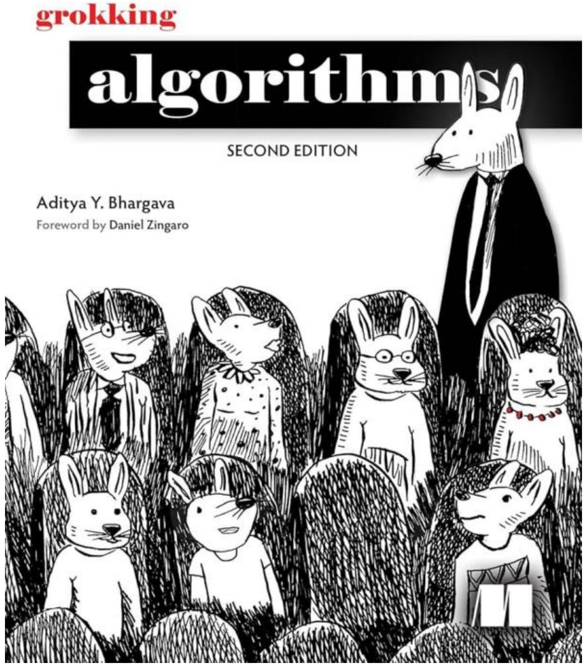
CS PROF DEV 2024

BIG O NOTATION

Reference



GeeksforGeeks



Geeksforgeeks.com
Grokkingg algorithm second edition by Aditya Y.Bhargava
Algorithm FOURTH EDITIONS by Robert Sedgewick and Kebin Wayne

Wheat and chessboard problem

- Once upon a time there was an Indian king who wanted to reward a wise man for his excellence.
- The wise man asked for nothing but some wheat that would fill up a chess board.
- But here were his rules: in the first tile he wants 1 grain of wheat, then 2 on the second tile, then 4 on the next one…each tile on the chess board needed to be filled by double the amount of grains as the previous one
- The naïve king agreed without hesitation, thinking it would be a trivial demand to fulfill, until he actually went on and tried it…

- So how many grains of wheat does the king owe the wise man?

Wheat and chessboard problem

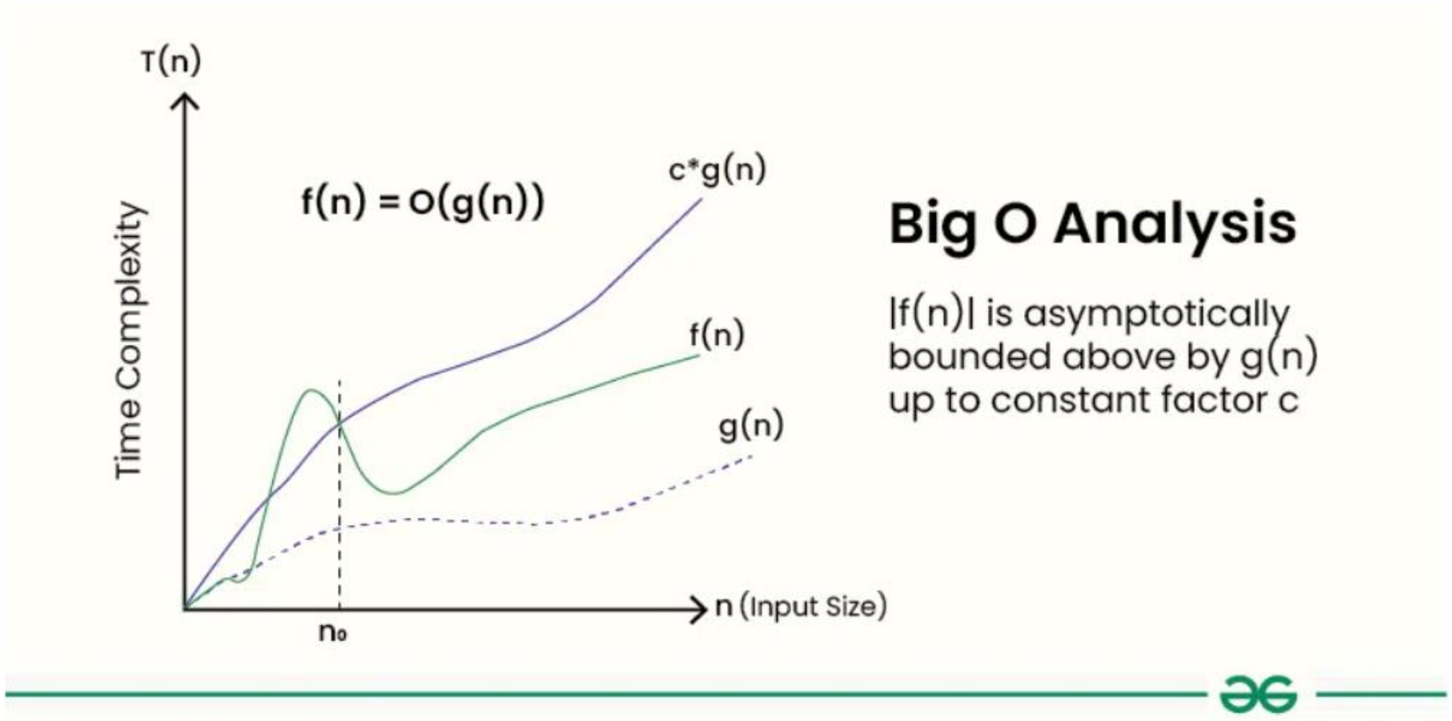
- We know that a chess board has 8 squares by 8 squares, which totals 64 tiles. So the last tile should have a total of 2^{63} grains of wheat. If you do a calculation online, for the entire chessboard, you will end up getting $1.8446744*10^{19}$ – that is about 18 followed by 18 zeroes.

- The numbers grow quite fast later for exponential growth don’t they?
- The same logic goes for computer algorithms.

- As we will see in a moment, the growth of 2^n is much faster than n^2. Now, with n = 64, the square of 64 is 4096. If you add that number to 2^{64}, it will be lost outside the significant digits.

Big O Notation

Big O notation is a powerful tool used in computer science to describe the time complexity or space complexity of algorithms. It provides a standardized way to compare the efficiency of different algorithms in terms of their worst-case performance. Understanding Big O notation is essential for analyzing and designing efficient algorithms.



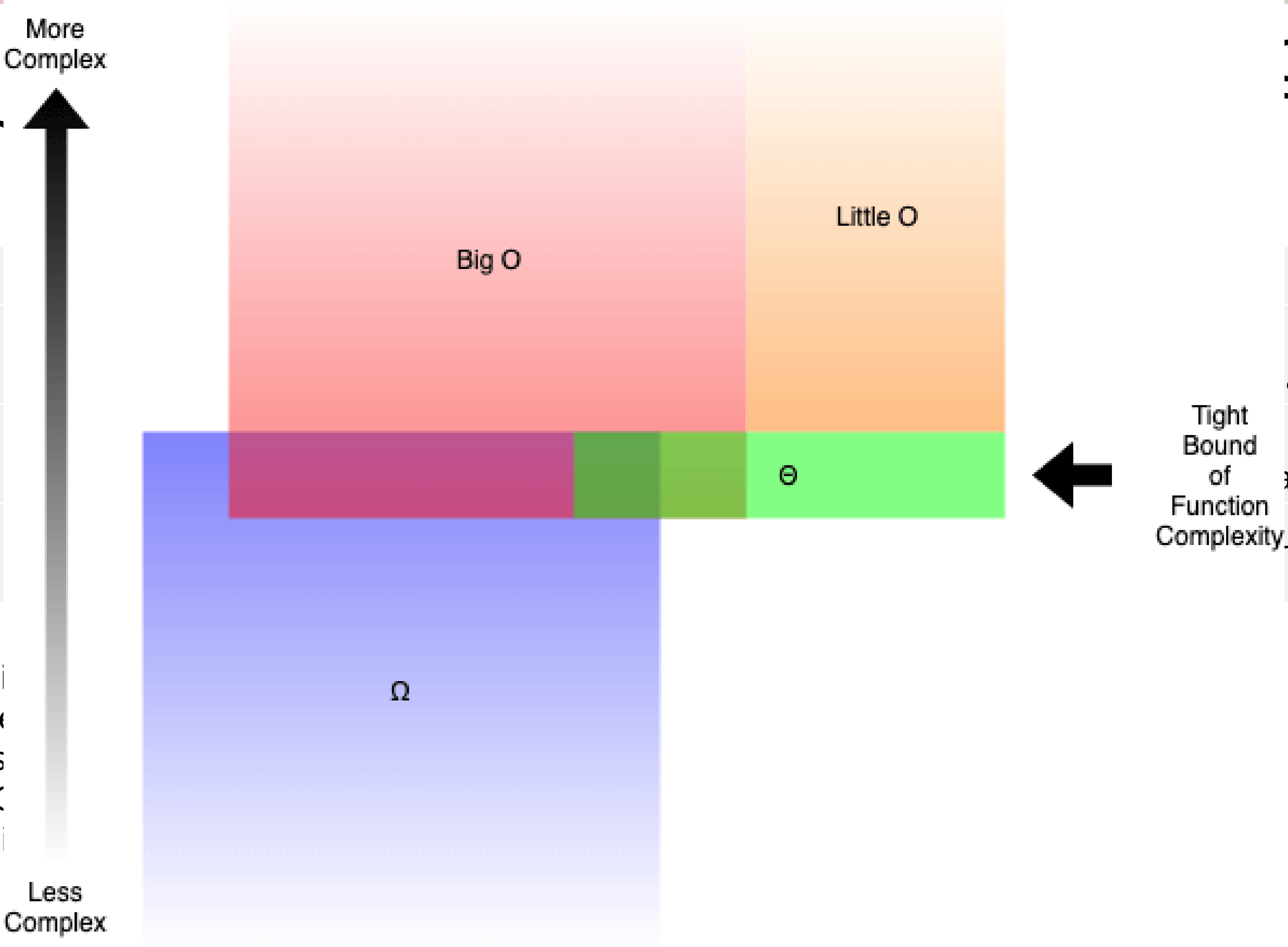
Comparison of Big O Notation, Big Ω (Omega) Notation, and Big θ (Theta) Notation

Notation	Definition	Explanation
Big O (O)	$f(n) \leq C * g(n)$ for all $n \geq n_0$	Describes the upper bound of the algorithm’s running time in the worst case.
Ω (Omega)	$f(n) \geq C * g(n)$ for all $n \geq n_0$	Describes the lower bound of the algorithm’s running time in the best case.
θ (Theta)	$C_1 * g(n) \leq f(n) \leq C_2 * g(n)$ for $n \geq n_0$	Describes both the upper and lower bounds of the algorithm’s running time.

In each notation:

- f(n) represents the function being analyzed, typically the algorithm’s time complexity.
- g(n) represents a specific function that bounds f(n).
- C, C_1, and C_2 are constants.
- n_0 is the minimum input size beyond which the inequality holds.=

Compari ga) Notatior



Notation	
Big O (O)	ase.
Ω (Omega)	ase.
θ (Theta)	unds

In each notati

- f(n) represe
- g(n) repres
- C, C_1, and (
- n_0 is the mi

Comparison of Big O Notation, Big Ω (Omega) Notation, and Big θ (Theta) Notation

[illegible]
[illegible] **dominant term** [illegible]
[illegible]

Logarithmic Time Complexity: Big O(log n) Complexity

Logarithmic time complexity means that the running time of an algorithm is proportional to the logarithm of the input size.

```
int binarySearch(int arr[], int l, int r, int x) {
    if (r >= l) {
        int mid = l + (r - l) / 2;
        if (arr[mid] == x)
            return mid;
        if (arr[mid] > x)
            return binarySearch(arr, l, mid - 1, x);
        return binarySearch(arr, mid + 1, r, x);
    }
    return -1;
}
```

Linear Time Complexity: Big O(n) Complexity

Linear time complexity means that the running time of an algorithm grows linearly with the size of the input.

```
boolean findElement(int arr[], int n, int key) {
    for (int i = 0; i < n; i++) {
        if (arr[i] == key) {
            return true;
        }
    }
    return false;
}
```

Quadratic Time Complexity: Big O(n^2) Complexity

Quadratic time complexity means that the running time of an algorithm is proportional to the square of the input size.

```
static void bubbleSort(int arr[], int n) {
    int i, j, temp;
    boolean swapped;
    for (i = 0; i < n - 1; i++) {
        swapped = false;
        for (j = 0; j < n - i - 1; j++) {
            if (arr[j] > arr[j + 1]) {
                temp = arr[j];
                arr[j] = arr[j + 1];
                arr[j + 1] = temp;
                swapped = true;
            }
        }
        if (swapped == false)
            break;
    }
}
```

Cubic Time Complexity: Big $O(n^3)$ Complexity

Cubic time complexity means that the running time of an algorithm is proportional to the cube of the input size.

```
void multiply(int mat1[][], int mat2[][], int res[][]) {
    int N = mat1[0].length;
    for (int i = 0; i < N; i++) {
        for (int j = 0; j < N; j++) {
            res[i][j] = 0;
            for (int k = 0; k < N; k++)
                res[i][j] += mat1[i][k] * mat2[k][j];
        }
    }
}
```

Polynomial Time Complexity: Big O(n^k) Complexity

Polynomial time complexity refers to the time complexity of an algorithm that can be expressed as a polynomial function of the input size n. In Big O notation, an algorithm is said to have polynomial time complexity if its time complexity is O(n^k), where k is a constant and represents the degree of the polynomial.

Algorithms with polynomial time complexity are generally considered efficient, as the running time grows at a reasonable rate as the input size increases. Common examples of algorithms with polynomial time complexity include linear time complexity O(n), quadratic time complexity O(n^2), and cubic time complexity O(n^3).

Exponential Time Complexity: Big O(2^n) Complexity

Exponential time complexity means that the running time of an algorithm doubles with each addition to the input data set.

```
void solve_hanoi(int N, String from_peg, String to_peg, String spare_peg) {
    if (N < 1)
        return;
    if (N > 1)
        solve_hanoi(N-1, from_peg, spare_peg, to_peg);
    System.out.println("move from " + from_peg + " to " + to_peg);
    if (N > 1)
        solve_hanoi(N-1, spare_peg, to_peg, from_peg);
}
```

Factorial Time Complexity: Big O(n!) Complexity

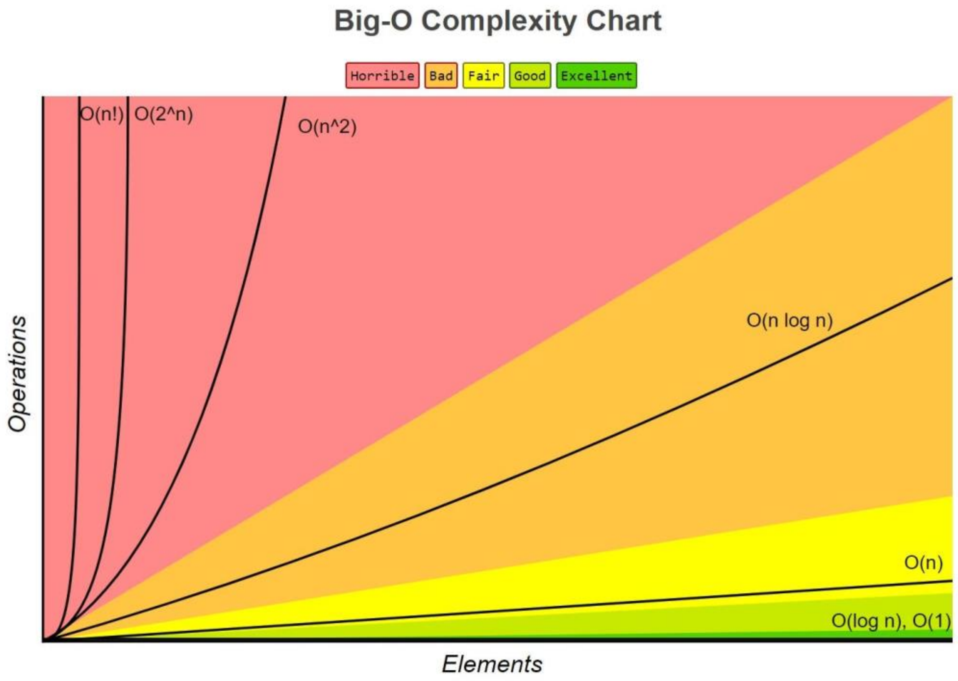
Factorial time complexity means that the running time of an algorithm grows factorially with the size of the input. This is often seen in algorithms that generate all permutations of a set of data.

```
public static List<List<Integer>>
generatePermutations(List<Integer> nums) {
    List<List<Integer>> result
    = new ArrayList<>();
    permute(nums, 0, result);
    return result;
}

private static void swap(List<Integer> nums,
int i, int j) {
    int temp = nums.get(i);
    nums.set(i, nums.get(j));
    nums.set(j, temp);
}
```

```
private static void permute(List<Integer> nums,
int start, List<List<Integer>> result) {
    if (start == nums.size() - 1) {
        result.add(new ArrayList<>(nums));
        return;
    }

    for (int i = start; i < nums.size(); i++) {
        swap(nums, start, i);
        permute(nums, start + 1, result);
        swap(nums, start, i);
    }
}
```

Mathematical Examples of Runtime Analysis

Size (n)	log n	n	n log n	n^2	2^n	n!
10	1	10	10	100	1024	3628800
20	2.996	20	59.9	400	1048576	2.432902e+1818



Big O for Common Data Structure

Data Structure	Time Complexity								Space Complexity
	Average				Worst				Worst
	Access	Search	Insertion	Deletion	Access	Search	Insertion	Deletion	
Array	O(1)	O(n)	O(n)	O(n)	O(1)	O(n)	O(n)	O(n)	O(n)
Stack	O(n)	O(n)	O(1)	O(1)	O(n)	O(n)	O(1)	O(1)	O(n)
Queue	O(n)	O(n)	O(1)	O(1)	O(n)	O(n)	O(1)	O(1)	O(n)
Linked List	O(n)	O(n)	O(1)	O(1)	O(n)	O(n)	O(1)	O(1)	O(n)