Big-O Notation – Analysis of Algorithms

(how fast does an algorithm grow with respect to N)

O(l) - constant time

This means that the algorithm requires the same fixed number of steps regardless of the size of the task.

```
bool IsFirstElementNull(String[] strings) {
    if(strings[0] == null)
        return true;
    return false;
}
```

- o finding a median value in a sorted array
- o accessing any element in an array or ArrayList
- o adding an element to the end of an ArrayList
- o addFirst, addLast, getFirst, getLast, removeFirst, & removeLast operations on a LinkedList

O(log n) - logarithmic time

- o Binary search in a sorted list of n elements;

O(n) - linear time

This means that the algorithm requires a number of steps proportional to the size of the task.

```
bool ContainsValue(String[] strings, String value) {
    for(int i = 0; i < strings.Length; i++)
        if(strings[i] == value)
            return true;
    return false;
}
```

- o Traversal of a list (a linked list or an array) with n elements;
- o Finding the maximum or minimum element in a list, or sequential search in an unsorted list of n elements;
- o Calculating the sum of n elements in an array, ArrayList, List, or Set
- o Calculating iteratively n-factorial; finding iteratively the nth Fibonacci number

O(n log n) - "n log n " time (a.k.a. superlinear time, a.k.a. Linearithmic time)

- o More advanced sorting algorithms - quicksort, mergesort, heapsort

O(n^2) - quadratic time

The number of operations is proportional to the size of the task squared.

```
bool ContainsDuplicates(String[] strings) {
    for(int i = 0; i < strings.Length; i++) {
        for(int j = 0; j < strings.Length; j++) {
            if(i != j && strings[i] == strings[j])  // Don't compare with self
                return true;
        }
    }
    return false;
}
```

- o Selection sort and Insertion sort
- o Traversing a two-dimensional array
- o Comparing two two-dimensional arrays of size n by n
- o Finding duplicates in an unsorted list of n elements (using two nested loops).

O(n^3) - cubic time

- o Conventional algorithm for matrix multiplication

$O(a^n)$, $a > 1$ - exponential time

Denotes an algorithm whose growth will double with each additional element in the input data set.

- Recursive Fibonacci implementation
- Towers of Hanoi
- Generating all permutations of n symbols
- Traveling salesman problem – dynamic programming solution

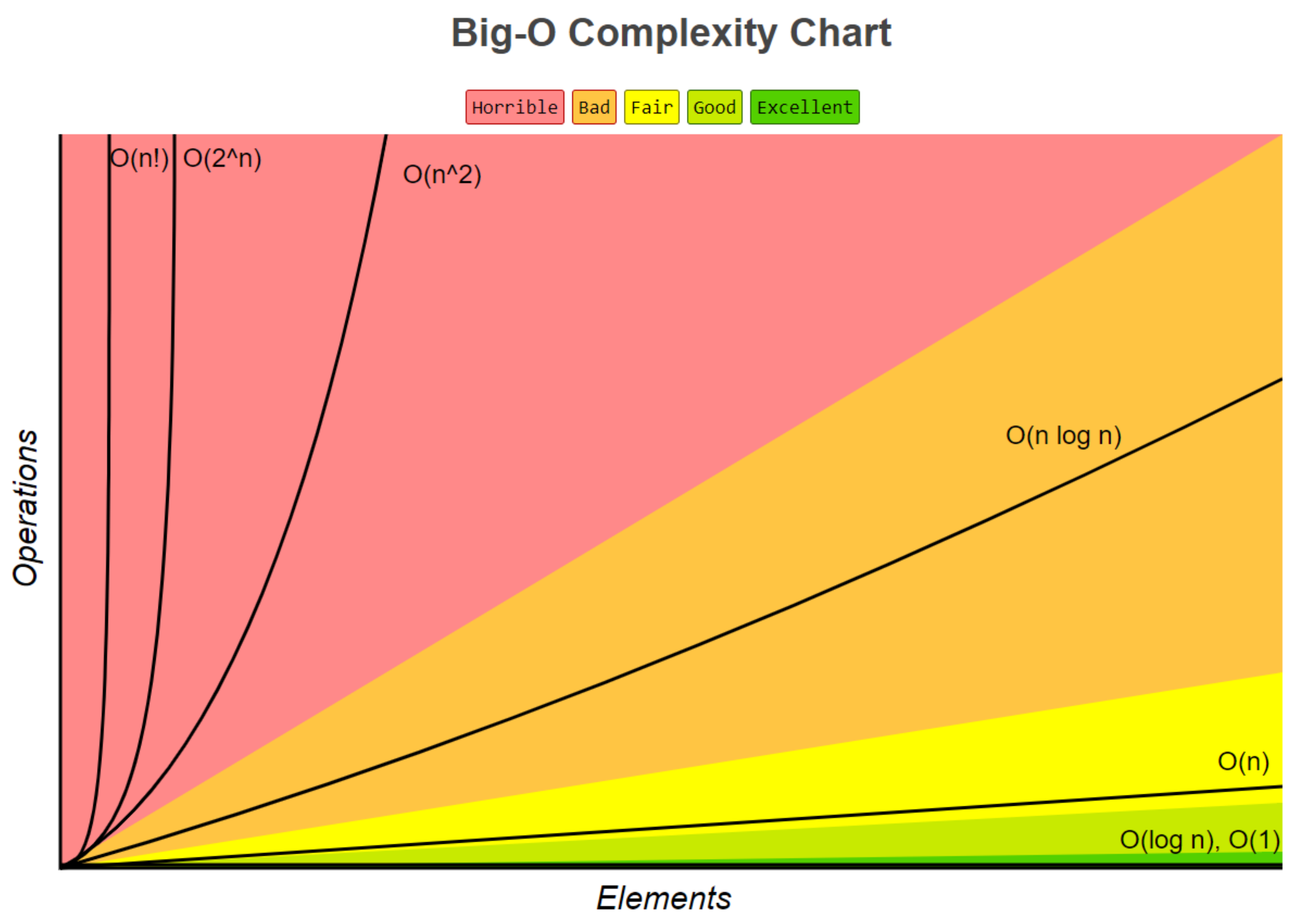
O(n!) — factorial time

- Traveling salesman problem – brute force solution
- Determinant Expansion by Minors

The best time in the above list is obviously constant time, and the worst is factorial time which quickly overwhelms even the fastest computers even for relatively small n. **Polynomial** growth (linear, quadratic, cubic, etc.) is considered manageable as compared to exponential or factorial growth.

Order of asymptotic behavior of the functions from the above list (using the "<" sign informally):

$$O(l) < O(\log n) < O(n) < O(n \log n) < O(n^2) < O(n^3) < O(a^n) < O(n!)$$



n	LOGARITHMIC ($\log_2 n$)	LINEAR (n)	QUADRATIC (n^2)	EXPONENTIAL (2^n)
100	7	100	10,000	Off the charts
1000	10	1000	1,000,000	Off the charts
1,000,000	20	1,000,000	1,000,000,000,000	Really off the charts

Big O values of Lists

The table summarizes the complexity classes of all the methods in the classes that implement the **List** interface.

Method	**ArrayList**	**LinkedList**
add (at index)	O(N)	O(N)
add (at at end)	O(1)*	O(1)
addAll (at end)	O(M)*	O(M)
addAll (at index)	O(MN)*	O(MM)
clear	O(N)	O(1)
contains	O(N)	O(N)
containsAll	O(MN)	O(MN)
equals	O(N)	O(N)
get	O(1)	O(N)
indexOf	O(N)	O(N)
isEmpty	O(1)	O(1)
iterator listIterator listIterator (at index)	O(1)	O(1)
lastIndexOf	O(N)	O(N)
remove (at index)	O(N)	O(N)
remove	O(N)	O(N)
removeAll	O(MN)	O(MN)
retainAll	O(MN)	O(MN)
set	O(1)	O(N)
size	O(1)	O(1)
sublist	O(N)	O(N)
toArray	O(N)	O(N)

M is the number of elements in the collection passed as a parameter to the method.

An * means **amortized** complexity. That is, when we **add** a value in an array, most of the time we perform some constant number of operations, independent of the size of the array. But every so often, we must construct a new array object with double the length, and then copy all the array's current elements into it. If we pretended that each **add** did more operations (but still a constant number, just a bigger constant), that number would dominate the actual number of operations needed for all the doubling/copying.

Using backing arrays, and length doubling (actually, most "empty" collections start with an initial capacity of 10 and increase by 1.5 times), the array length for storing **N** elements is never more than **1.5N** words of memory; using linked lists, storing **N** elements always requires **2N** words of memory. So array implementations of list always are better, in storage capacity, than linked implementations.

A word about O(log n) growth:

As we know from the Change of Base Theorem, for any a, b > 0, and a, b $\neq$ 1

$$\log_b n = \frac{\log_a n}{\log_a b}$$

Therefore, $\mathbf{\log_a n = C \log_b n}$ where C is a constant equal to $\log_a b$.

Since functions that differ only by a constant factor have the same order of growth, **O(log$_2$ n)** is the same as **O(log n).** Therefore, when we talk about logarithmic growth, the base of the logarithm is not important, and we can say simply **O(log n).**

A word about Big-O when a function is the *sum* of several terms:

If a function (which describes the order of growth of an algorithm) is a *sum* of several terms, its order of growth is determined by the fastest growing term. In particular, if we have a polynomial

$$\mathbf{p(n) = a_k n^k + a_{k-1} n^{k-1} + \ldots + a_1 n + a_0}$$

its growth is of the order n^k:

$$\mathbf{p(n) = O(n^k)}$$

Example:

```
{perform any statement S1}                         O(1)
for (i=0; i < n; i++)
{
        {perform any statement(s) S2}              O(n)

        {run through another loop n times}         O(n^2)
}
```

Total Execution Time: $1 + n + n^2$ therefore, $\mathbf{O(n^2)}$