Time and Space Complexity

Key-Terms

Complexity Analysis

The process of determining how efficient an algorithm is. Complexity analysis usually involves finding both the **time complexity** and the **space complexity** of an algorithm.

Complexity analysis is effectively used to determine how "good" an algorithm is and whether it's "better" than another one.

Time Complexity

A measure of how fast an algorithm runs, time complexity is a central concept in the field of algorithms and in coding interviews.

It's expressed using **Big O** notation.

Space Complexity

A measure of how much auxiliary memory an algorithm takes up, space complexity is a central concept in the field of algorithms and in coding interviews.

It's expressed using **Big O** notation.

Big O Notation

The notation used to describe the **time complexity** and **space complexity** of algorithms.

Variables used in Big O notation denote the sizes of inputs to algorithms. For example, **O(n)** might be the time complexity of an algorithm that traverses through an array of length **n**; similarly, **O(n + m)** might be the time complexity of an algorithm that traverses through an array of length **n** and through a string of length **m**.

The following are examples of common complexities and their Big O notations, ordered from fastest to slowest:

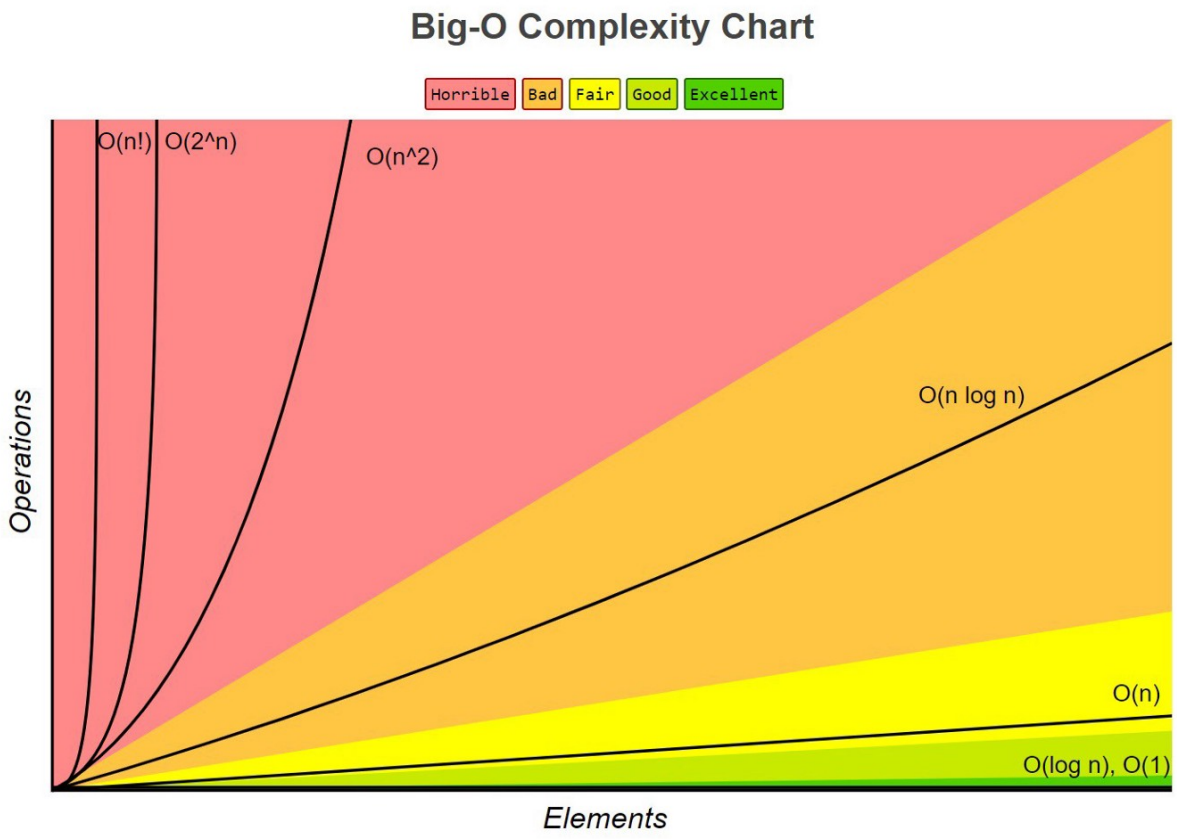
- **Constant**: O(1)
- **Logarithmic**: O(log(n))
- **Linear**: O(n)
- **Log-linear**: O(nlog(n))
- **Quadratic**: $O(n^2)$
- **Cubic**: $O(n^3)$
- **Exponential**: $O(2^n)$
- **Factorial**: O(n!)

Note that in the context of coding interviews, Big O notation is usually understood to describe the **worst-case** complexity of an algorithm, even though the worst-case complexity might differ from the **average-case** complexity.

For example, some sorting algorithms have different time complexities depending on the layout of elements in their input array. In rare cases, their time complexity will be much worse than in more common cases. Similarly, an algorithm that takes in a string and performs special operations on uppercase characters might have a different time complexity when run on an input string of only uppercase characters vs. on an input string with just a few uppercase characters.

Thus, when describing the time complexity of an algorithm, it can sometimes be helpful to specify whether the time complexity refers to the average case or to the worst case (e.g., "this algorithm runs in O(nlog(n)) time on average and in $O(n^2)$ time in the worse case").

Gráfico



Log

Logarithm

A mathematical concept that's widely used in Computer Science and that's defined by the following equation:

$\log_b(x) = y$ if and only if **$b^y = x$**

In the context of coding interviews, the logarithm is used to describe the complexity analysis of algorithms, and its usage always implies a logarithm of base **2**. In other words, the logarithm used in the context of coding interviews is defined by the following equation:

$\log(n) = y$ if and only if **$2^y = n$**

In plain English, if an algorithm has a logarithmic time complexity (**O(log(n))**, where n is the size of the input), then whenever the algorithm's input doubles in size (i.e., whenever **n** doubles), the number of operations needed to complete the algorithm only increases by one unit. Conversely, an algorithm with a linear time complexity would see its number of operations double if its input size doubled.

As an example, a linear-time-complexity algorithm with an input of size 1,000 might take roughly 1,000 operations to complete, whereas a logarithmic-time-complexity algorithm with the same input would take roughly 10 operations to complete, since **$2^{10} \sim= 1{,}000$**.

- caso base

- em computação, a base sempre é 2

- $2^1 = 2$
  - log(n) = y = 2
- $2^2 = 4$
  - log(n) = y = 4
- $2^3 = 8$
  - log(n) = y = 8
- Ao dobrar o n, o y só aumenta em 1, ou seja, quando o input dobra, log do input só aumenta em 1
  - a complexidade aumenta muito pouco com o aumento do input
  - $2^3 = 2^1.2^2$

O(1)

- constante
    - o tempo não muda com o aumento do input

```
int a[] = {1, 2, 3, 4, 5};
return 1 + a[0];
```

O(log(n))

- se ao dobrar ou reduzir para metade o input ou alguma variável apenas uma operação é realizada, provavelmente a complexidade é de log(n)

```
while (x > 0) {
   x /= 2;
}

i = 1
while(i < n)
    i = i * 2
```

```
Iteration |   x
----------+-------
    0     |   x
    1     |  x/2
    2     |  x/4
   ...    |  ...
   ...    |  ...
    k     |  x/2^k
```

O(n)

- linear
    - conforme o input aumenta, a complexidade de tempo aumenta linearmente
        - como se caso o input aumentasse em 1, o tempo aumentaria em 1

```
int a[] = {1, 2, 3, 4, 5};
int sum = 0;
for(int i = 0; i < 5; i++)
  sum += a[i];
```

On(nlog(n))

- itera pelo array e performa operações que dividem pela metade ou dobrem o conteúdo com apenas uma operação
- merge sort e quick sort

```
/* C program for Merge Sort */
#include <stdio.h>
#include <stdlib.h>

// Merges two subarrays of arr[].
// First subarray is arr[l..m]
// Second subarray is arr[m+1..r]
void merge(int arr[], int l, int m, int r)
{
  int i, j, k;
  int n1 = m - l + 1;
  int n2 = r - m;

  /* create temp arrays */
  int L[n1], R[n2];

  /* Copy data to temp arrays L[] and R[] */
  for (i = 0; i < n1; i++)
    L[i] = arr[l + i];
  for (j = 0; j < n2; j++)
    R[j] = arr[m + 1 + j];

  /* Merge the temp arrays back into arr[l..r]*/
  i = 0; // Initial index of first subarray
  j = 0; // Initial index of second subarray
  k = l; // Initial index of merged subarray
  while (i < n1 && j < n2) {
    if (L[i] <= R[j]) {
      arr[k] = L[i];
      i++;
    }
    else {
      arr[k] = R[j];
```

```
      j++;
    }
    k++;
  }

  /* Copy the remaining elements of L[], if there
  are any */
  while (i < n1) {
    arr[k] = L[i];
    i++;
    k++;
  }

  /* Copy the remaining elements of R[], if there
  are any */
  while (j < n2) {
    arr[k] = R[j];
    j++;
    k++;
  }
}

/* l is for left index and r is right index of the
sub-array of arr to be sorted */
void mergeSort(int arr[], int l, int r)
{
  if (l < r) {
    // Same as (l+r)/2, but avoids overflow for
    // large l and h
    int m = l + (r - l) / 2;

    // Sort first and second halves
    mergeSort(arr, l, m);
    mergeSort(arr, m + 1, r);

    merge(arr, l, m, r);
  }
}

/* UTILITY FUNCTIONS */
/* Function to print an array */
void printArray(int A[], int size)
{
  int i;
  for (i = 0; i < size; i++)
    printf("%d ", A[i]);
  printf("\n");
}

/* Driver code */
int main()
{
  int arr[] = { 12, 11, 13, 5, 6, 7 };
  int arr_size = sizeof(arr) / sizeof(arr[0]);
```

```
  printf("Given array is \n");
  printArray(arr, arr_size);

  mergeSort(arr, 0, arr_size - 1);

  printf("\nSorted array is \n");
  printArray(arr, arr_size);
  return 0;
}
```

O(n²)

- quadrática
  - para cada n no input, itera uma vez por todo o input
    - [1, 2, 3]
      - n = 1
        - itera por 1, 2 e 3
      - n = 2
        - itera por 1, 2 e 3
      - n = 3
        - itera por 1, 2 e 3
- conforme o input aumenta, a complexidade de tempo aumenta de forma quadrática
  - If our array has `n` items, our outer loop runs `n` times and our inner loop runs `n` times for each iteration of the outer loop

```
// printa todas as combinações de pares
  // 1 e 1, 1 e 2, 1 e 3, 1 e 4, 1 e 5, 2 e 1, 2 e 2, ...
int a[] = {1, 2, 3, 4, 5};
for(int i = 0; i < 5; i++)
{
  for(int j = 0; j < 5; j++)
    printf("%d = %d\n", arr[i], arr[j]);
}
```

O(2^n)

- quando se analisa o número de combinações possíveis dentro de um input
- Uma palavra tem 2 bit, um bit pode ser 0 ou 1
  - as possibilidades de palavras existentes são $2^2 = 4$
  - {00}, {01}, {10}, {11}
- muitas vezes são algoritmos recursivos que resolvem um problema de tamanho N resolvendo em um problema menor N-1

```
int Fibonacci(int n)
{
    if (n <= 1)
        return n;
    else
        return Fibonacci(n - 1) + Fibonacci(n - 2);
}
```

O(n!)

- Permutações de uma string
  - ABC
    - ABC ACB BAC BCA CBA CAB

```
// C program to print all permutations with duplicates allowed
#include <stdio.h>
#include <string.h>

/* Function to swap values at two pointers */
void swap(char *x, char *y)
{
    char temp;
    temp = *x;
    *x = *y;
    *y = temp;
}

/* Function to print permutations of string
This function takes three parameters:
1. String
```

```
2. Starting index of the string
3. Ending index of the string. */
void permute(char *a, int l, int r)
{
int i;
if (l == r)
    printf("%s\n", a);
else
{
    for (i = l; i <= r; i++)
    {
        swap((a+l), (a+i));
        permute(a, l+1, r);
        swap((a+l), (a+i)); //backtrack
    }
}
}

/* Driver program to test above functions */
int main()
{
    char str[] = "ABC";
    int n = strlen(str);
    permute(str, 0, n-1);
    return 0;
}
```

```
from itertools import permutations
l = list(permutations(range(1, 4)))
print(l)
```