Leonardo Gabriel Martínez Uribarren
MS. in Computer Science and Engineer
lgmu.cursos@hotmail.com

Required Readings

- Data Structures and Algorithms in C++, Fourth Edition. Pages: 51-56[1].
- Ideas That Created the Future: Classic Papers of Computer Science. Pages: 440-443. Big Omicron and Big Omega and Big Theta (1976), Donald E. Knuth[2].
- Zahlentheorie, Paul Bachmann. Pages: 401[3].

1 Computational Complexity

The Computational Complexity measures the degree of difficulty of an algorithm. Its study take their foundations in the ceaselessly pursuit to find an efficient solution to a given problem. It can be done by analyzing the resources an algorithm needs to be completed. Considering resources such as energy consumption, memory usage, and how many clock cycles are required.

1.1 Qualitative approach

> A solution that finish its task fast using real input data is an efficient algorithm[4].

Considering that a given algorithm is correct and it finished. There are still a few observations concerning to such definition.

Firstly, it is impossible to know if a deficient implementation is running rapidly because of a specific hardware. And finally, what is the meaning of "real input data" and how its growth will impact in terms of performance.

For those reasons, it is imprudent to express that an algorithm is both good or bad just based on the efficient definition.

1.2 Quantitative approach

A first approach to find an efficient solution is to compare several algorithms running on the same hardware. The most efficient is the one which got the best performance in all areas.

Even though is the most precise way to measure computational complexity, this approach have some problems too, Although it gets the most efficient algorithm from a set of algorithms, a real efficient algorithm may not be included in it. Furthermore, how many algorithms someone needs to compare so as to obtain an efficient one. The task may become a highly time consuming.

Another approach is to analyse an individual algorithm in terms of its growing running time, relating the input to the computer's clock cycles. This perspective takes into consideration the hardware and the input data, quantifying the time an algorithm takes to be completed in terms of the input[1].

> An algorithm is efficient if it has a polynomial running time[4].

The previous definition requires a method to evaluate an algorithm and obtain a polynomial (P). This process is done by analysing the algorithm line by line and assign an algebraic equation to each ($p_1, p_2, p_3, ..., p_n$), where the value is represented in terms of the input n (n is the number of times a block of code will be executed). And the algorithm polynomial is the summation of each equation obtained.

$$P = p_1 + p_2 + p_3 + ... + p_n \tag{1}$$

For example. The polynomial (P) for the next block of code is:

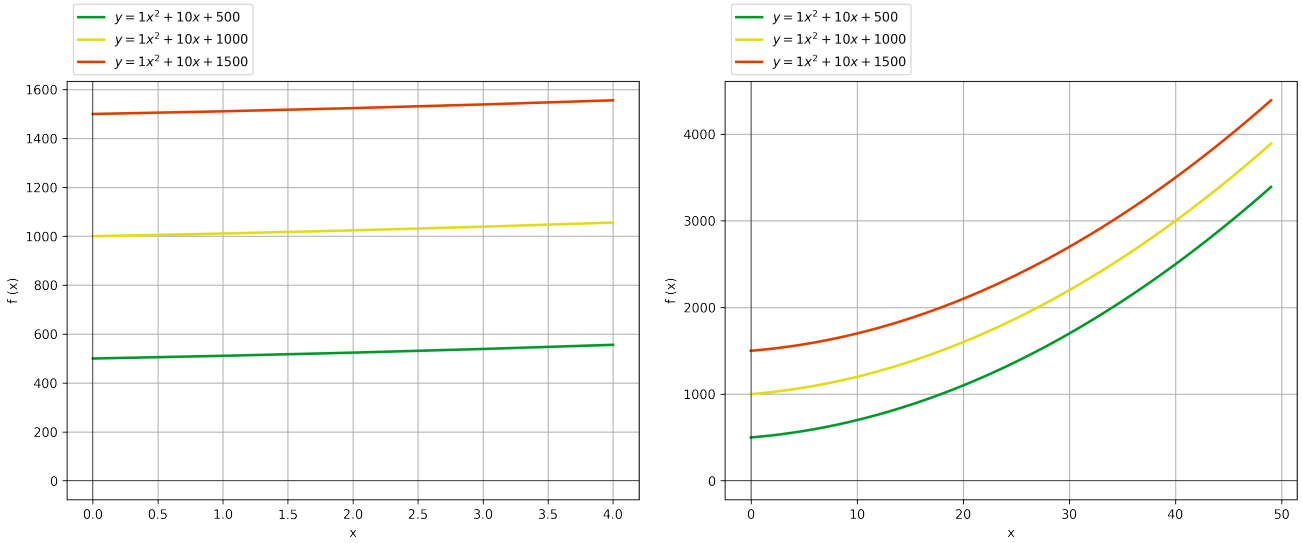
```
1  int main()
2  {
3      int a = 10, b = 20;              // c1
4      cout << a+b << endl;             // c2
5      for (int i=0; i<a; i++)          // n
6          cout << i*a*b << endl;       // c3
7      return 0;                        // c4
8  }
```

$$\therefore \quad P = c_3 n + c_1 + c_2 + c_4$$

[1] A similar approach is done to analyse the complexity of an algorithm considering the memory usage. Called Space Complexity.

1.3 Big O notation

Consider the follow equation: $f(x) = ax^2 + bx + c$; $\{a, b, c \in \mathbb{R} \mid c > b > a\}$. When x has a lower value, c defines the growing ratio of $f(x)$. As the value of x increases, the quadratic term starts defining the growing ratio of the equation.



For algorithm analysis introduction, the only important term is the highest order one[2] so as to group an algorithm based in its order growth. For the same equation: $f(x) = ax^2 + bx + c$, it is said that $f(x)$ belongs to the quadratic set of equations. $\{f(x) \in x^2\}$.

This efficiency approximation is called *asymptotic complexity* and Big O notation is an upper bounded asymptotic complexity defined as:

$$\{f(n) \text{ is } O(g(n)) \mid \exists\ n \in n_0 > 0, c > 0; \quad f(n_0) \leq cg(n_0)\}$$

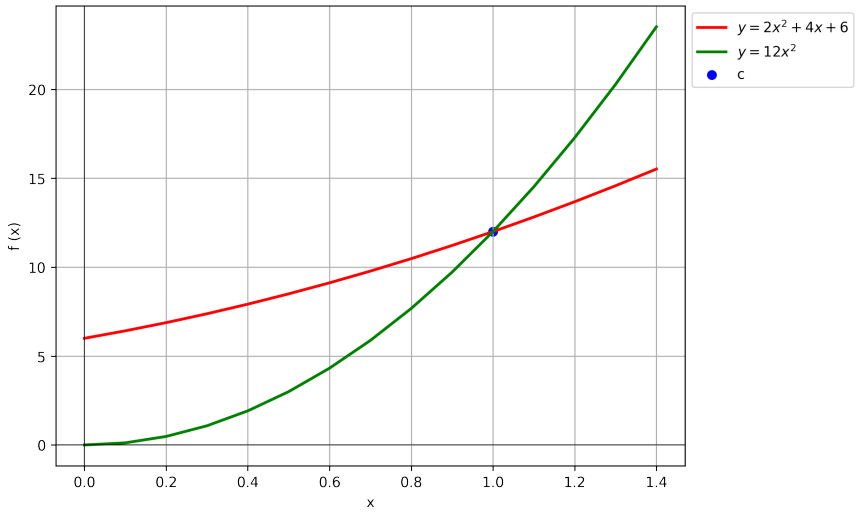
For example:

If $a = 2, b = 4, c = 6$. $f(n) = O(n^2)$

$$2n^2 + 4n + 6 \leq cn^2$$
$$\frac{2n^2}{n^2} + \frac{4n}{n^2} + \frac{6}{n^2} \leq c$$
$$2 + \frac{4}{n} + \frac{6}{n^2} \leq c$$

For $x = 1$

$$2 + \frac{4}{1} + \frac{6}{1} \leq c$$
$$2 + 4 + 6 \leq c$$
$$12 \leq c$$



[2]For Θ analysis and precise algorithms analysis requires to take into consideration every variable term including constants.

1.3.1 Complexity Examples

```
int a = (1+2+5);
int b = a * a * a;
int *c = &b;
```

$O(c)$

```
for (int i=0; i<n; i++)
        // Code
```

$O(n)$

```
for (int i=0; i<n; i++)
    for (int j=0; j<m; j++)
        // Code
```

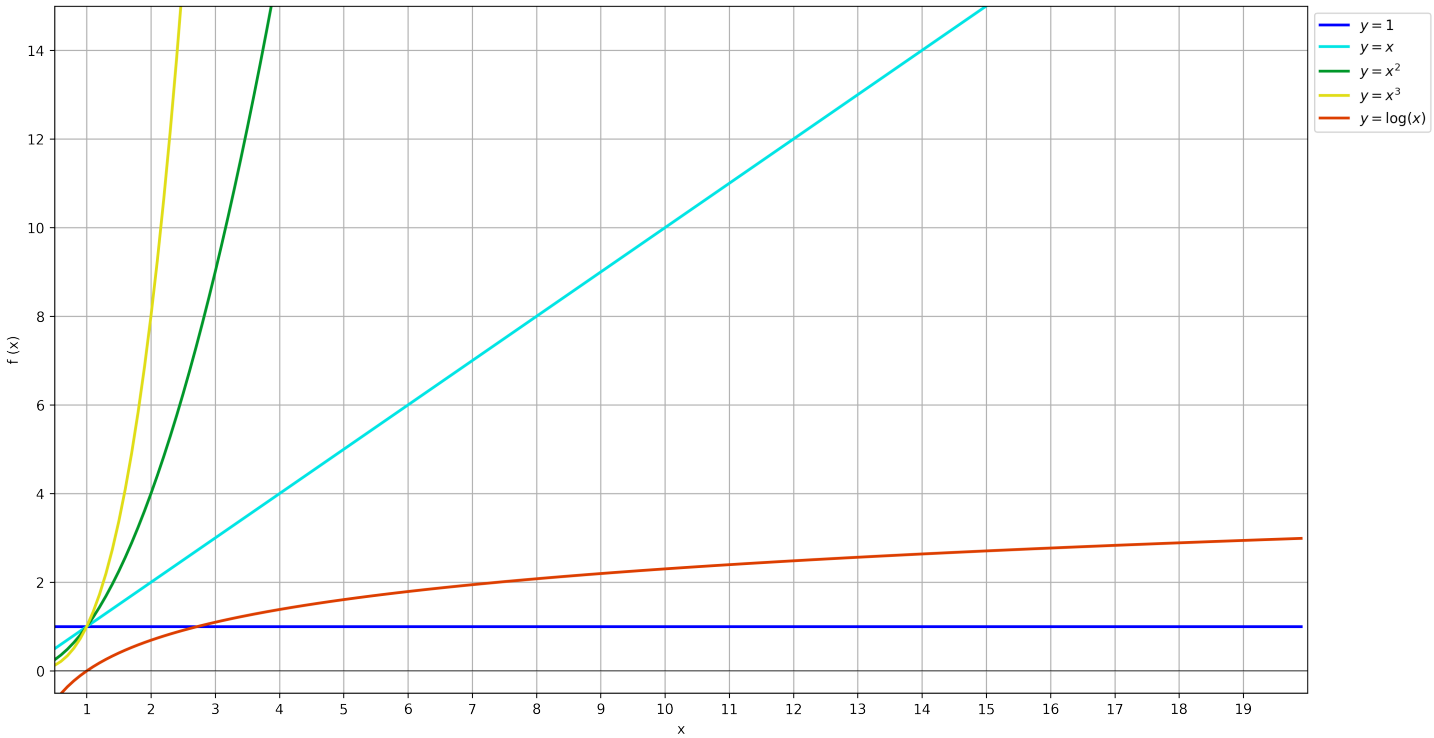
$O(n^2)$

```
for (int i=0; i<n; i++)
    for (int j=0; j<m; j++)
        for (int k=0; k<o; k++)
            // Code
```

$O(n^3)$

```
int binarySearch(int arr[], int x)
{
    int l = 0;
    int r = sizeof(arr) / sizeof(arr[0]);
    int m;
    while (l <= r)
    {
        m = l + (r - l) * 0.5;
        if (arr[m] == x) return m;
        if (arr[m] < x)  l = m + 1;
        else             r = m - 1;
    }
    return -1;
}
```

$O(log(n))$



1.4 Exercises

A Get the Big O notation for the next polynomials:

1. $f(x) = x^3 + x^2 + x^4 + x^2 + x^3$

2. $f(x) = (c_1 + c_2)^2 * (c_1 - c_2)^2 * (c_1 * c_2)^2$

3. $f(x) = \frac{x^2}{x^3}$

4. $f(x) = \frac{5x^5}{20x^2} + \frac{14x^6}{42x^3}$

5. $f(x) = 1 + 2 + 3 + ... + n$

---

1) this polynomial complexity is x power of four 2) this polynomial just have constants, its complexity is constant. 3) the reduced fraction is x power one over two, its complexity is n power one over two. 4) the reduce fraction is x power 3, its complexity is n power of three. 5) this problem is a linear series, reduced is n square.

B Do the following tasks.

(a) Get the Computational Complexity for the following code.

(b) Modify the function *bubble_sort* to obtain an inverse result.

```cpp
#include <iostream>
using namespace std;

void bubble_sort(int *arr, int size)
{
    for (int i = 0; i < size-1; i ++)
        for (int j = 0; j < size-2-i; j ++)
            if (arr[j] < arr[j+1])
            {
                int temp = arr[j+1];
                arr[j+1] = arr[j];
                arr[j] = temp;
            }
}

void printArr(int arr[], int size)
{
    for (int i = 0; i < size-1; i ++)
        cout << arr[i] << " ";
    cout << endl;
}

int main()
{
    int arr1[] = {2, 7, 1, 8, 3, 4};
    int size = sizeof(arr1) / sizeof(arr1[0]);
    printArr(arr1, size);
    bubble_sort(arr1, size);
    printArr(arr1, size);
    return 0;
}
```

References

[1] Adam Drozdek. *Data Structures and Algorithms in C++, Fourth Edition*. Cengage Learning, 2013. ISBN: 9781285415017.

[2] Harry R. Lewis. *Ideas That Created the Future: Classic Papers of Computer Science*. The MIT Press, Feb. 2021, pp. 440–443. ISBN: 9780262363174. DOI: 10.7551/mitpress/12274.001.0001. URL: https://doi.org/10.7551/mitpress/12274.001.0001.

[3] Paul Bachmann. *Zahlentheorie*. LEIPZIG, 1894, p. 401.

[4] Jon Kleinberg & Eva Tardos. *Algorithm Design*. Always Learning. Addison Wesley, 2014. ISBN: 9781292023946.

[5] Ronald L. Rivest & Clifford Stein Thomas H. Cormen Charles E. Leiserson. *Introduction to Algorithms, Fourth Edition*. MIT Press, 2022. ISBN: 9780262046305.

a) bubble_sort function's complexity is quadratic and printArr function's complexity is lineal, therefore the complexity is n square. b) to obtain an array in inverse order is required to iterate the array in inverse order.