Calculating algorithm complexity

By Stefan Maxim,

Research Symposium 2024

with function interpolation

The Harker School

Abstract

Traditionally, calculating an algorithm's complexity is done by looking at the runtime terminal statements and counting the number of operations. However, algorithms with non-polynomial complexity are particularly hard to estimate this way as one would usually resort to brute-force calculation. Our method proposes a simple way to approximate the actual complexity of an algorithm. First, we make a table with the number of operations that a function makes for a certain input n starting at 1, and with each incrementation of n corresponding to a new value in our table (Ex: n=1, 2, 3, 4, 5 =>f(n)= 1, 2, 4, 7, 11...). Then we made a calculated column with n-1 values, where each number is the difference between the upper two complexity values,
(1, 2, 3, 4 in this case, since it represents 2-1, 4-2, 7-4, 11-7). This process was repeated until we got a table with one value, and then treated it as the starting constant for the operation complexity equation (Example: if there were 5 tables, then the lone value at the bottom was the 4th derivative of the operation function). We tested this method with a java program that interpolated the data from the tables into a function that served as a reasonable estimate for what the complexity behaves like for a certain range of data points. For an input of X test data points we can model the operation complexity of a sample size of 2X data points with an average accuracy of 70%.

Introduction

The complexity of a given algorithm provides an approximate **estimation of its runtime**, expressed as a function of the input size (n). Such estimation is done **independent of the particular implementation** details. The runtime T(n) is measured as the number of elementary steps of constant duration needed to execute the algorithm.

Goals:

- Predict the behavior and runtime of an algorithm without implementing it
- Compare multiple algorithms in order to select the best one for the considered purpose

Assumptions:

- Determining the algorithm complexity is an approximate evaluation
- Exact behavior and runtime are hard to compute given the large number of influencing factors

In AI, algorithm complexity is crucial. [1] investigates the speed of algorithm improvement and the corresponding complexity.

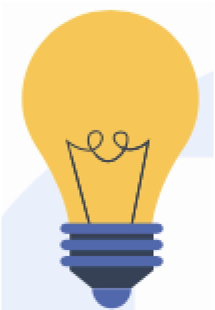
Background

Usual steps:

1. **Identify time consuming operations**:
   - Evaluate **which operations** in the given algorithm contribute significantly to the **overall runtime**
2. **Count operations**:
   - Estimate **how many times** each key operation is executed as a function of the **input size (n)**
3. **Expression simplification**:
   - Simplify the expressions from step 2, focusing on the **higher powers of (n)** that give the largest time contribution (ignore lower order terms in power of (n))
4. **Use Big O notation**:
   - Transform the simplified expression from step 3 in Big O notation, that provides an **upper bound** for the algorithm **runtime growth rate** as a function of input size (n)

Issues:

- Takes long time
- Slow



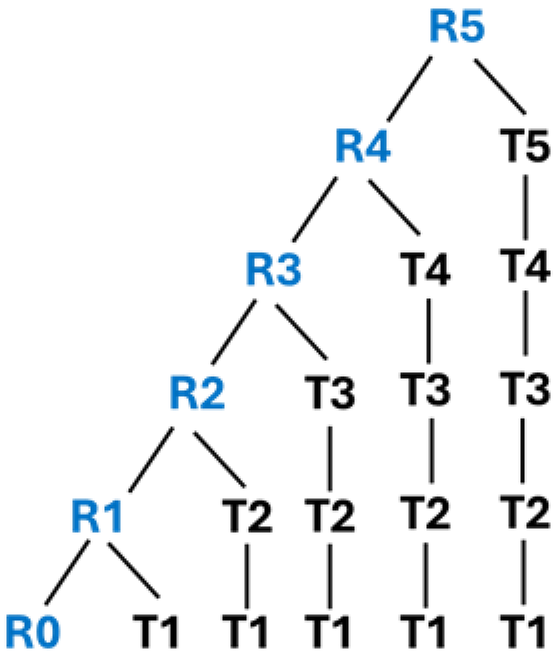
Introducing solution with example

Algorithm:

R(n) {
T(n)
R(n-1)}

T(n){
T(n-1)}

Tree calls:



Integration

ValueTable:

N :	0	1	2	3	4	5
calls :	0	2	5	9	14	20
diff :		2	3	4	5	6
diff :			1	1	1	1

$$f''(n) = 1$$

$$f'(n) = \int 1 dn = n + c$$

$$f(n) = \int n+c \, dn = \frac{n^2}{2} + cn + d$$

$$O(n) \approx \text{highest power} = \frac{n^2}{2} \approx n^2$$

$$\longrightarrow O(n) = n^2$$

Math - finite differences

Derive difference quotient from Taylor's polynomial [edit]

For a *n*-times differentiable function, by Taylor's theorem the Taylor series expansion is given as

$$f(x_0+h)=f(x_0)+\frac{f'(x_0)}{1!}h+\frac{f^{(2)}(x_0)}{2!}h^2+\cdots+\frac{f^{(n)}(x_0)}{n!}h^n+R_n(x),$$

Assuming that $R_1(x)$ is sufficiently small, the approximation of the first derivative of *f* is:

$$f'(x_0)\approx\frac{f(x_0+h)-f(x_0)}{h}.$$

This is similar to the definition of derivative, which is:

$$f'(x_0)=\lim_{h\to 0}\frac{f(x_0+h)-f(x_0)}{h}.$$

except for the limit towards zero (the method is named after this).

Brook Taylor(1685 -1731)

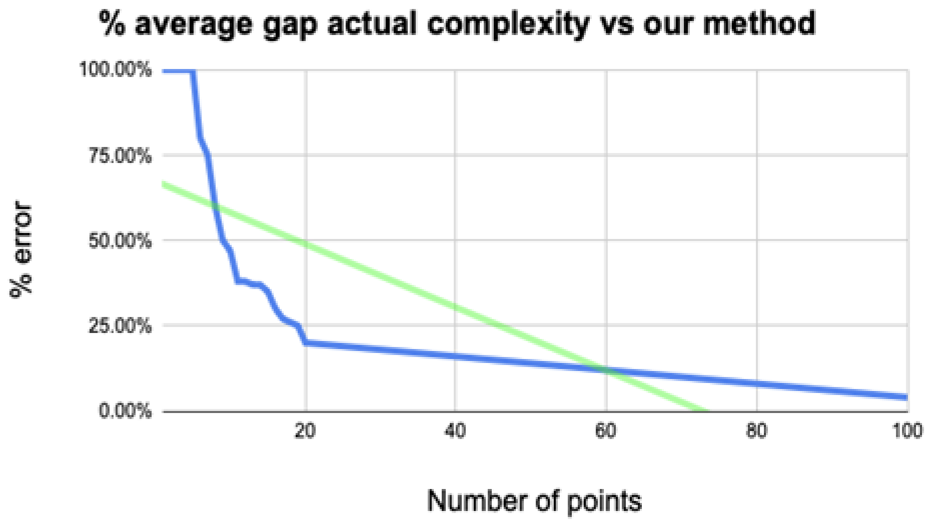
Algorithm

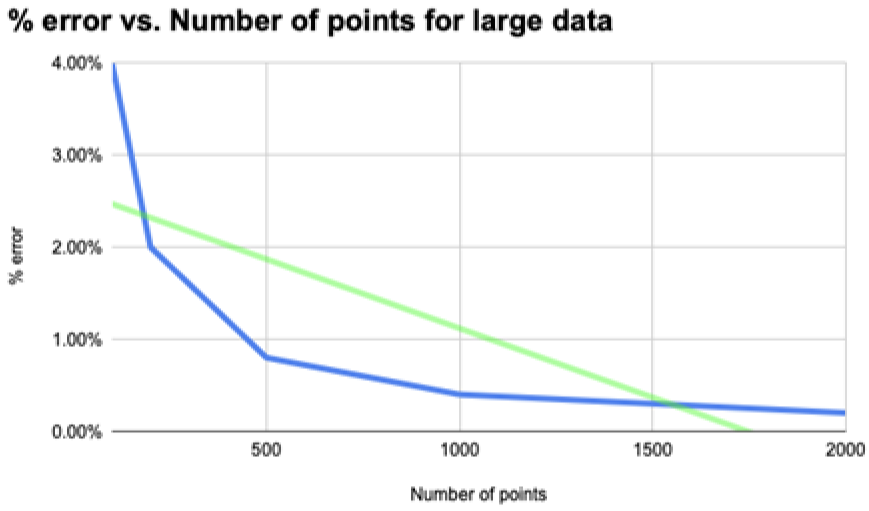
```java
/**
 * Method that takes an ArrayList of numbers and returns an ArrayList of number that are the distances between each of the terms
 * Exe: Takes [0, 1, 3, 6, 10, 15], and returns [1, 2, 3, 4, 5], since those are the distance betwen 0 and 1, 1 and 3, 3 and 6, etc
 * @param prevArrayList the origional ArrayList from which to find the distances
 * @return An ArrayList of the distances between each of the terms in the og AL
 */
public ArrayList<Integer> arrDeriv(ArrayList<Integer> prevArrayList)
{
    ArrayList<Integer> derived = new ArrayList<Integer>();
    for(int i = 0; i < prevArrayList.size()-1; i++)
    {
        derived.add(prevArrayList.get(i+1) - prevArrayList.get(i));
    }

    return derived;
}
```


```
+1.0x^(0)
1 , 2 , 4 , 8 , 16 ,
1 , 2 , 4 , 8 ,
1 , 2 , 4 ,
1 , 2 ,
1 ,
PREDICTED FUNCTION: +0.041666666666666664x^(4)+-0.08333333333333333x^(3)+0.5x^(2)+0.41666666666666696x^(1)+1.0x^(0)

PREDICTED COMPLEXITY:57.5
REAL COMPLEXITY:64.0
THE ERROR  PERCENT IS:10.15625%
```

Results

Polynomial function of grade 4



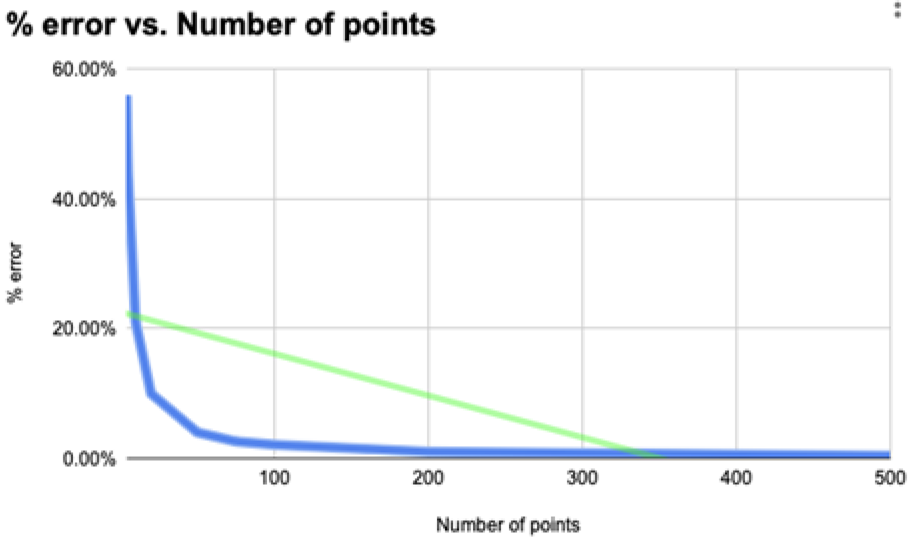


Results

Polynomial function of grade 2 with only 3 input points

Sample pyramid with 10 points

```
1 , 4 , 9 , 16 , 25 , 36 , 49 , 64 , 81 , 100 ,
3 , 5 , 7 , 9 , 11 , 13 , 15 , 17 , 19 ,
2 , 2 , 2 , 2 , 2 , 2 , 2 , 2 ,
0 , 0 , 0 , 0 , 0 , 0 , 0 ,
0 , 0 , 0 , 0 , 0 , 0 ,
0 , 0 , 0 , 0 , 0 ,
0 , 0 , 0 , 0 ,
0 , 0 , 0 ,
0 , 0 ,
0 ,
```

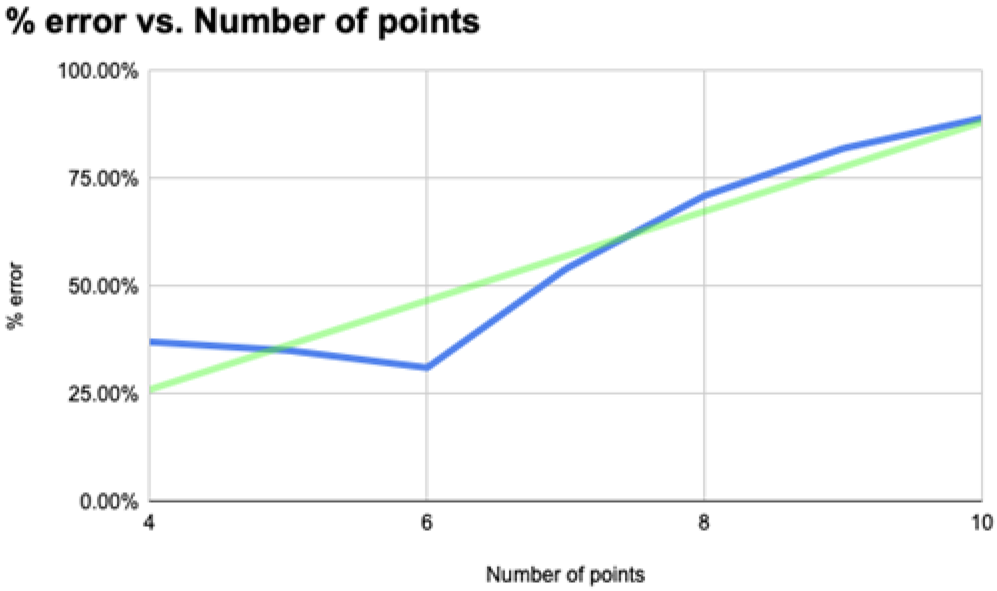


Results

2^n function with only 3 points

Sample pyramid with 3 points

```
2 , 4 , 8 ,
2 , 4 ,
2 ,
```



Conclusion & Further Research

- The computational complexity of algorithms ⇒performance impact
    - speed, efficiency, and resource utilization
- Polynomials are easier to estimate even at high order
- Finite differences is a very useful method to evaluate derivatives

  The 2^n and log algorithms need an adjusted approach

Future work

- Math proof for the proposed solution
- Determine the error rate in many more classes of functions
    - n logn
    - binary

Thanks!

References

1. Y. Sherry and N. C. Thompson, "How Fast Do Algorithms Improve? [Point of View]," in *Proceedings of the IEEE*, vol. 109, no. 11, pp. 1768-1777, Nov. 2021, doi: 10.1109/JPROC.2021.3107219.
2. Swapnil Phalke; Yogita Vaidya; Shilpa Metkar, “Big-O Time Complexity Analysis Of Algorithm”, in Proceedings of 2022 International Conference on Signal and Information Processing (IConSIP), Aug. 2022.
3. J. Hartmanis, R. E. Stearns, “On the computational complexity of algorithm”, American Mathematical Society, May 1965, pp. 285-306.
4. K. Cordwell, et al, “On algorithms to calculate integer complexity”, arXiv 1706.08424v4, Dec. 2018
5. J. Arias de Reyna & J. van de Lune, *Algorithms for determining integer complexity*, arXiv:1404.2183 [math.NT].
6. V. V. Srinivas and B. R. Shankar, *Integer complexity: Breaking the* θ(n2) *barrier*, World Academy of Science,Engineering and Technology 2 (2008), no. 5, 454 - 455.