Algorithm Analysis

Zoe Siegelnickel Palak Yadav

May 23, 2022

Abstract

Algorithms are the foundation of technology today. From medicine to education and beyond, algorithms serve to solve complex problems. This paper explores several types of recursive algorithms and compares them using the conventional notation of time complexity. We analyze algorithms such as the Karatsuba algorithm and the Strassen algorithm, two kinds of algorithms that reduce the time it takes to multiply numbers.

1 Introduction

1.1 General Definitions

Algorithms are the foundation of society today. From medicine to education and beyond, algorithms serve to solve complex problems often by organizing large sets of data to identify patterns and solutions.

Algorithms are essentially a set of procedures a computer executes to convert a set of input to a desired set of output. For example, schools use applications to create schedules for students. Course selections of all the students are inserted into an algorithm that creates an unique schedule for each student by taking into account factors such as class size, teacher availability, sections and more. This is one example of the type of problems algorithms attempt to solve.
The applications of algorithms are numerous and can be found in almost all fields today. In medicine, for example, DNA and protein modeling is an important task to develop drugs and vaccines. A computer must analyze millions of bases of the DNA to locate patterns or specific genes. An example of such an application is the Human Genome project.

1.2 Time Complexity

In a competitive world where countless solutions are being worked on to solve a problem, there must exist a universal way for comparing and determine how an algorithm behaves.

To illustrate this in an intuitive way, let's consider the time it takes for a car and a person to travel 10 feet. If both start from rest, the person will likely

reach the 10 feet mark first because the car will take extra time to turn on. However, if the car and the person were required to travel 10 miles, the car will definitely reach first. Though the type of car and the speeds of different individual will vary, the general behavior of each over a long period of time will help us determine which mode of transportation is a better option. A similar idea can be applied to algorithms.

Rather than only looking at how algorithms behave for small input size, the general behavior of the function as the input size increases must be considered. Though it may seem reasonable to simply measure the time it takes to run an algorithm, this is not an accurate reflection of the algorithm's efficiency because factors such as the type of hardware, other applications running at the same time, and more will cause the run time to vary from device to device. Therefore, a machine independent way of comparing algorithms must exist.

An algorithm's run time primarily depends on the input size. As the input size increases, the time it takes to execute the algorithm also increases. Each algorithm has a certain correlation between the input size and the run time, where the run time is described in terms of the input size.

The terms run time and input size can be defined more precisely:

Definition 1.1 (Run Time) *The run time of an algorithm is the number of steps or operations executed; it is machine independent.*

Definition 1.2 (Input Size) *The number of elements in the input is the input's size.*

To measure the run time for an algorithm, three cases must be considered: worst-case, best-case, and average case scenarios. Each case is analyzed by setting an asymptotic bound on the algorithm. Setting an asymptotic bound on the algorithm indicates how the function will behave over time.

The three most common bounds that can be set on an algorithm are as following:

Definition 1.3 (Big O $O(n)$) This gives the upper-bound of an algorithm's run time, describing the worst-case scenario run time.

$$\begin{gathered}\text{For a given function } g(n),\\ O(g(n)) = f(n) : \text{there exists positive constants } c \text{ and } n_0 \text{ such that}\\ 0 \leq f(n) \leq cg(n) \text{ for all } n \geq n_0\end{gathered}$$

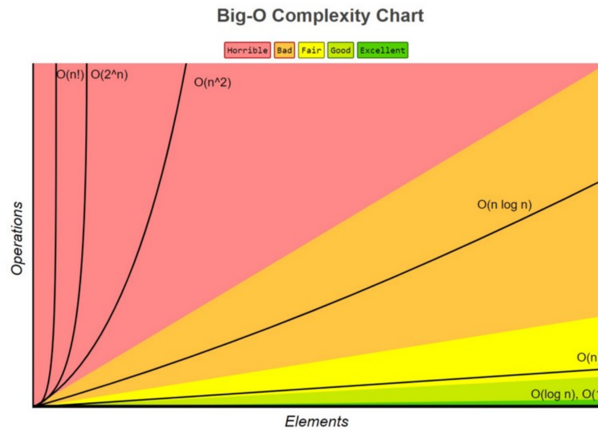

Figure 1: The graph represents various relationships that can exist between the input size and the worst-case run time.

Definition 1.4 (Big-Omega Ω) This sets the lower bound on an algorithm and indicates the best-case scenario when the algorithm preforms the least amount of work.

$$\text{For a given function } g(n),$$
$$\Omega\ (g(n)) = f(n) : \text{there exists positive constants } c \text{ and } n_0 \text{ such that}$$
$$0 \leq cg(n) \leq f(n) \text{ for all } n \geq n_0$$

Definition 1.5 (Theta Θ) This describes the best and worst case scenario. It sets both an upper and lower bound, giving the exact bound.

$$\text{For a given function } g(n),$$
$$\Theta(g(n)) = f(n) : \text{there exists positive constants } c_1, c_2, \text{ and } n_0 \text{ such that}$$
$$0 \leq c_1 g(n) \leq f(n) \leq c_2 g(n) \text{ for all } n \geq n_0$$

1.3 Search Algorithms

Search algorithms are used to located specific elements in an array. The most basic search algorithm is the **linear search**

Suppose we have a list of n integers, such as $[1, 2, 3, 4, 5, 6, 7, 8, 9]$ and the integer 7 must be found.

In linear search, the algorithm will look through every single input to find the desired value. In the best-case scenario, the desired value can be the first item in the list, and the run-time can be expressed as $O(1)$. This is referred to as constant time, where the number of operations to find the value is only one.

In the worst-case scenario, the desired value will be the last item in the list, and the algorithm will need to analyze every single item in the input, resulting in the run-time of $O(n)$. This is called the linear time because as the input size increases, the run-time increases linearly.

Binary Search, also known as half interval search, is a more efficient type of searching algorithm than linear search. Binary search must be performed on a sorted array, where the inputs are organized in ascending order. The target value is compared to middle term in the array. The algorithm eliminates the half that does not contain the target value, depending on if the target value is greater than or less than the middle term. The process of comparing and eliminating continues until the target value is found, or the algorithm reaches a point where it can no longer continue and the term is not found.

This procedure can be described mathematically in the following way:

Consider an sorted array S with n elements:
$[S_1 \leq S_2 \leq S_3 \leq S_4 ... S_{n-1}]$
The target value T must be identified in the set. Each element in the set is assigned an index, starting at 0 and going up to n

1. Set min $= 0$ and max $= n - 1$
2. If $min > max$, the search ends and the target is not included in the array.
3. Compute the middle term m by solving $(min + max)/2$
4. If $S_m = T$, the target has been found and the search ends.
5. If $S_m < T$, set min to $m + 1$ and repeat from step 2
6. If $S_m > T$ set max to $m - 1$ and repeat from step 2.

As the size of the input increases, the number of operations required to find the target values increases by $log_2 n$. For example, when n is 4, the worst-case scenario requires 2 operations. When n is 512, the worst-case scenario requires at 9 operations to find the target. The time complexity of the binary search

can be expressed as $O(log_2 n)$. On large scales, the binary search is significantly faster than the linear search.

1.4 Sorting Algorithms

Sorting algorithms are used to sort large arrays of unorganized data. Some common sorting algorithms include insertion sort, bubble sort, selection sort, merge sort and more.

Bubble Sort

Suppose we have an unsorted array $B[1, 4, 3, 5, 8, 2]$ and we want to arrange it in ascending order. Bubble sort examines each element in the array and compares it to the adjacent element. If the adjacent element is smaller, the two elements swap, and the greater element is now on the right side. This process pushes the largest element in the array to the right. The algorithm repeatedly swaps the elements until all elements are correctly sorted.

$$[\mathbf{1,\ 4}, 3, 5, 8, 2]$$
$$[1, \mathbf{4,\ 3}, 5, 8, 2] \text{ swap}$$
$$[1, 3, \mathbf{4,\ 5}, 8, 2$$
$$[1, 3, 4, \mathbf{5,\ 8}, 2]$$
$$[1, 3, 4, 5, \mathbf{8,\ 2}] \text{ swap}$$
$$[1, 3, 4, 5, 2, 8]$$

The algorithm will perform several more iteration to move all the remaining elements in the right position. The algorithm must go through an additional pass without any swapping to know that the array has been sorted. In the best case scenario when the smallest number is furthest to the right, at least n swaps must occur.

For large arrays, the bubble sort is not the most efficient algorithm because it has a time-complexity of $O(n^2)$.

2 Recursive Algorithms

2.1 Introduction

Organizing large sets of data is a vital aspect of computer science, and data scientists implement recursive algorithms to solve problems involving large inputs. When computing a large problem, recursive algorithms divide the larger problem at hand into smaller sub-problems by recalling itself. Generally, it divides the input until the base case is reached and performs the algorithm's objective on easily solvable instances. This method reduces the number of operations a computer must perform, but increases the memory required when applied at large scales.

2.2 Fibonacci Sequence

The Fibonacci Sequence is an excellent example to demonstrate recursive algorithms. The Fibonacci Sequence is a list of numbers where each number is the sum of the two numbers that precede it.

Where n is the n-th number in the series and the base values are 0 and 1, the Fibonacci Sequence can be described as:

$$F_0 = 0$$
$$F_1 = 1$$
$$F_n = F_{n-1} + F_{n-2} \text{ for } n \geq 2$$

Below are the first few numbers of the sequence:

$$0, 1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89, 144...$$

The recursive approach involves dividing a large problem into sub-problems, and recursively calling the function upon each of the smaller sub-problems. Eventually, the base cases are reached and recursive calls can no longer be applied.

For example to find F_5, the following procedures can be performed:

$$\begin{aligned} F_5 &= F_4 + F_3 \\ &= (F_3 + F_2) + (F_2 + F_1) \\ &= (F_2 + F_1 + F_0 + F_1) + (F_1 + F_0 + F_1) \\ &= (F_1 + F_0 + F_1 + F_0 + F_1) + (F_1 + F_0 + F_1) \end{aligned}$$

On a large scale, calculating the nth term in such a manner is unfeasible and time consuming. This is where dynamic programming plays a key role in reducing the number of operations, however, we we will not be exploring that in this paper.

3 Karatsuba Algorithm

Discovered by Anatoly Karatsuba in 1960, the Karatsuba algorithm is a fast way of multiplying two numbers. Multiplying large numbers efficiently is an important task for computer scientists, however the traditional, naive way of multiplying numbers involves multiplying each digit in one number to each digit in the second number, requiring n^2 single-digit computations. As the size of multiplication increases, the time required to solve using the naive way increases dramatically. However, the Karatsuba multiplication algorithm uses the divide and conquer procedure to reduce the time complexity from $O(n^2)$ to $O(n^{log_2 3}) \approx O(n^{1.58})$

3.1 Naive Method of Multiplication

First, let's analyze the naive method of multiplying numbers that is traditionally taught in schools.

For example, two even-digit number are chosen

$$x = 1345$$
$$y = 5624$$

To find the product, each digit in y must be multiplied to each digit in x. For example, the 4 in x must be multiplied to every single digit in 1345, then 2 must be multiplied to every single digit in 1345 and so on and so forth.

$$\begin{array}{c} 1345 \\ *\ 5624 \\ \hline 5380 \\ 26900 \\ 807000 \\ +6725000 \\ \hline 7564280 \end{array}$$

A total of four sub-products must be solved to find the final answer, and each "sub-product" involves four computations. In total, 16 single-digit computations must be carried out to find the product of x and y.

If n represents the number of digits in each number, the naive method of multiplying requires n^2 single digit computations to find the product of any two n-sized numbers.

3.2 Applying Karatsuba Algorithm

The Karatsuba Algorithm applies the divide and conquer procedure to reduce the time required to multiple two n-digit numbers. If the numbers are odd, zeros can be added on the left to make it an even number. This algorithm divides the problem into three-sub problems, and involves some basic arithmetic and shifts to join the results to produce the final product. Instead of executing 16 computations (n^2), the Karatsuba algorithm only utilizes 12 computations ($n^{1.58}$)

The numbers x and y can be expanded and be expressed in the following manner:

$$x = 1300 + 45$$
$$y = 5600 + 24$$

Each number can be divided into its lower and higher "bits"

The bits of x are as following:

$$\text{low bit} = 45 = a$$
$$\text{high bit} = 13 = b$$

The bits of y are as following:

$$\text{low bit} = 24 = c$$
$$\text{high bit} = 56 = d$$

The product of x and y can written as:

$$xy = (1300 + 45)(5600 + 24)$$

The distributive property can be applied to simplify the expression:

$$xy = (1300 * 5600) + (1300 * 24) + (5600 * 45) + (45 * 24)$$

$$xy = (13 * 56) * 10^4 + (13 * 24 + 56 * 45) * 10^2 + 45 * 24$$

Once again, 4 products must be solved, each involving 4 single-digit multiplications. A total of 16 computations (n^2) must be performed to find the product.

The Karatsuba Algorithm takes advantage of recursions to compute this at a faster rate.

The Karatsuba algorithm divides the problem into three sub-problems. Since three terms are being added, the algorithm divides the sub-products in the following way:

1. The product of the higher bits $ac = 13 * 56$
2. The product of the lower bits $bd = 45 * 24$
3. The middle term $(ad + bc)$ is found by solving the expression

$$(a + b)(c + d) - ac - bd = (13 + 45)(56 + 24) - 13 * 56 - 45 * 24 \tag{1}$$

The expression in the third sub-problem can be simplified to prove that it indeed equals to $(ad + bc)$

$$(a + b)(c + d) - ac - bd = (ad + bc)$$

$$(ac + ad + bc + bd) - ac - bd$$
$$ad + bc$$

The Karatsuba algorithm applies recursions to "recall" the values of ac and bd in the third sub-problem, and subtracts that from the product of xy

3.3 Generalized Karatusba Algorithm

The following procedures can be applied to the product of any two n-sized numbers. If the numbers contain odd digits, a zero can be added on the left to make it even.

$$x = a * 10^{n/2} + b$$
$$y = c * 10^{n/2} + d$$

The multiplication of x and y can be shown as:

$$xy = (a * 10^{n/2} + b)(c * 10^{n/2} + d) \tag{2}$$

When all the terms are multiplied, the result is the following:

$$xy = ac * 10^n + (ad + bc) * 10^{n/2} + bd$$

To apply the Karatusba algorithm, create the following three sub-problems, where b represents the base of the numbers:

$$H = ac$$
$$L = bd$$
$$M = (a + c)(b + d) - H - L$$
$$xy = H * b^n + M * b^{n/2} + L$$

3.4 Analyzing Run Time

The Karatsuba Algorithm can be applied until the problem is reduced to the multiplication of single-digit numbers. Since only three multiplication of size $n/2$ are being carried in addition to some basic arithmetic, the algorithm reduces the time complexity from $O(n^2)$ to:

$$O(n^{log_2 3}) = 3T(n/2) + O(n) \tag{3}$$

On a large scale, the Karatusba algorithm provides a dramatic improvement over the naive algorithm. For example, to multiply two 570 digit numbers, the naive method would require 324900 single-digit computations, where as the Karatsuba algorithm would use approximately 23332 single-digit multiplications.

4 Akra Bazzi method and Algorithm Analysis

4.1 Recurrence Relations and Merge Sort

Merge sort is a type of recursive algorithm that uses the divide-and-conquer approach to reduce the time it takes to sort arrays in the following way:

1. Divides the array of n elements in half
2. Sorts the two sub arrays recursively
3. Merges the two sorted sub-arrays, resulting in one sorted array

The time complexity of merge sort can be expressed as :

$$T(n) = 2T(n/2) + \Theta(n) \tag{4}$$

We can use the Akra Bazzi method to prove that the merge sort has a time complexity of $O(nlogn)$. It is much more efficient than various sorting algorithms, such as insertion sort and quick sort.

To start, we will define a recurrence relation as an expression of a certain term in a sequence as a function of a set number of terms preceding it. The Master theorem can give us the asymptotic complexity, or time for the worst case scenario of a recurrence relation.

The theorem takes the form

$$T(n) = aT\frac{n}{2} + f(n) \tag{5}$$

where $aT\frac{n}{2}$ represents the time it takes to recurse down to a workable problem size, with each problem being size $\frac{n}{b}$, and $f(n)$ representing the time it takes to work with the small problem. However, this theorem cannot be applied to recurrences that aren't split into evenly sized sub problems, so we can use the Akra Bazzi method to generalize to more recurrences;

$$T(x) = \sum_{i=1}^{k} a_i T(b_i x) + g(x) \tag{6}$$

- a_i is a constant greater than 0.
- p is a real number such that
- b_i is a constant is some value between 0 and 1.

$$\sum_{i=1}^{k} a_i b_i^p = 1 \tag{7}$$

We can use lemma 1 from calculus to introduce our constants:

$$c_3 g(x) \geq x^p \int_{b_i x}^{x} \frac{g(u)}{u^{p+1}} du \geq c_4 g(x)$$

We can then apply theorem 1 of the Akra - Bazzi method:

$$T(x) = \Theta\left(x^p \left(1 + \int_1^x \frac{g(u)}{u^(p+1)} du\right)\right) \tag{8}$$

4.2 Proof (as presented by Tom Leighton)

We will split the domain of x into 2 intervals:

$$I_0 = [1, x_o]$$

$$\text{for } j \geq 1 : I_j = (x_0 + j - 1, x_0 + j)$$

We will show that there is a constant c_5 such that for all $x > x_0$

$$T(x) \geq c_5 x^p \left(1 + \int_1^x \frac{g(u)}{u^{p+1}}\right)$$

4.3 Proof:

$$T(x) = \sum_{i=1}^{k} a_i T(b_i x) + g(x) \geq \sum_{i=1}^{k} a_i c_5 (b_i x)^p \left(\int_1^{b_i x} \frac{g(u)}{u^{p+1}} du\right)$$

We can separate out the constants $c_5 x^p$ and as isolate the wanted interval:

$$c_5 x^p \sum_{i=1}^{k} a_i b_i \left(1 + \int_1^x \frac{g(u)}{u^{p+1}} du - \int_{b_i x}^{x} \frac{g(u)}{u^{p+1}} du\right)$$

$$\geq c_5 x^p \sum_{i=1}^{k} a_i b_i \left(1 + \int_1^x \frac{g(u)}{u^{p+1}} du - \frac{c_4}{x^p} g(x)\right) + g(x)$$

Because we know that

$$\sum_{i=1}^{k} a_i b_i^p = 1$$

we can simplify as such:

$$= c_5 x^p \left(1 + \int_1^x \frac{g(u)}{u^{p+1}} du - \frac{c_4}{x^p} g(x)\right) + g(x)$$

$$= c_5 x^p \left(1 + \int_1^x \frac{g(u)}{u^{p+1}} du\right) + g(x) - c_5 c_4 g(x)$$

$$= \left(1 + \int_1^x \frac{g(u)}{u^{p+1}} du\right)$$

□

5 Matrix Multiplication

The idea of recursion used in the Karatsuba Algorithm can be applied to multiply matrices faster.

5.1 Standard Multiplication

Supposing we start with two n by m matrices;

$$\begin{bmatrix} 1 & 2 \\ 4 & 5 \end{bmatrix}$$

$$\begin{bmatrix} a & b \\ d & e \end{bmatrix}$$

Traditional matrix multiplication would dictate that we multiply every row in the first matrix by every column in the second, and then add up values according to the example below:

$$\begin{bmatrix} 1a + 2d & 1b + 2e \\ 4a + 5d & 1b + 2e \end{bmatrix}$$

This quickly becomes inefficient for larger matrices, so we can implement a recursive program to divide our original input matrix into 4 quadrants, and continue this pattern until we reach blocks of size 2 by 2. From there, we can multiply out the matrix using traditional multiplication, giving us a time complexity of $O(n^2)$: notation.

$$\begin{bmatrix} 1 & 2 & 3 & 4 \\ 5 & 6 & 7 & 8 \\ 9 & 10 & 11 & 12 \\ 13 & 14 & 15 & 16 \end{bmatrix} \text{ becomes } \begin{bmatrix} 1 & 2 \\ 5 & 6 \end{bmatrix} \begin{bmatrix} 3 & 4 \\ 7 & 8 \end{bmatrix} \begin{bmatrix} 9 & 10 \\ 13 & 14 \end{bmatrix} \begin{bmatrix} 11 & 12 \\ 13 & 14 \end{bmatrix}.$$

Where we our original input n is divided into 8 sub-problems, each half the size of $A + B$, with the addition of the sub-problems coming out to $(n/2)^2$.

5.2 Strassen Algortihm

Strassen's method similarly starts by recursively subdividing matrices until they reach size n by 2. By creating new matrices and maximizing the efficacy of the distributive property, Strassen's reduces the multiplicative operations to 7.

Using the same example matrices;

$$\begin{bmatrix} 1 & 2 \\ 4 & 5 \end{bmatrix} \begin{bmatrix} a & b \\ d & e \end{bmatrix}$$

our 7 new matrices become:

$$M_1 : \left[(1+5)(a+e)\right]$$

$$M_2 : \left[(4+5)(a)\right]$$

$$M_3 : \left[(1)(b-e)\right]$$

$$M_4 : \left[(5)(d-a)\right]$$

$$M_5 : \left[(1+4)(e)\right]$$

$$M_6 : \left[(4-1)(a+d)\right]$$

$$M_7 : \left[(4-5)(d+e)\right]$$

Because every matrix only includes 1 multiplication, we have brought down the number of multiplications from 8 to 7. Similarly to the Karatsuba algorithm, this brings down the time complexity from n^3 to $n^{2.81}$.

Given these matrices, the formula for our complete multiplication is as follows:

$$\begin{bmatrix} M_1 + M_4 - M_5 + M_7 & M_3 + M_5 \\ M_2 + M_4 & M_1 - M_2 + M_3 + M_6 \end{bmatrix}$$

Example:

If we start with the following matrices:

$$\begin{bmatrix} 1 & 2 \\ 3 & 4 \end{bmatrix} \begin{bmatrix} 5 & 6 \\ 7 & 8 \end{bmatrix}$$

Our first step is to create our 7 new matrices:

$$M_1 = (1+4)(5+8) = 65$$

$$M_2 = (3+4)5 = 35$$

$$M_3 = 1(6-8) = -2$$
$$M_4 = 4(7-5) = 8$$
$$M_5 = (1+2)8 = 24$$
$$M_6 = (3-1)(5+6) = 22$$
$$M_7 = (2-4)(7+8) = -30$$

Now we add according to the algorithm:

$$\begin{bmatrix} (65+8-24+-30) & (-2+24) \\ (35+8) & (65-35+-2+22) \end{bmatrix} = \begin{bmatrix} 19 & 22 \\ 43 & 50 \end{bmatrix}$$

6 Using the Akra Bazzi

The time complexity for the Strassen algorithm can be represented as such;

$$T(x) = 7T\left(\frac{x}{2}\right) + \Theta(n)$$

where we divide up our original matrix into 7 sub-problems, each with size $\frac{x}{2}$. We can find the time complexity of the Strassen algorithm using the Akra Bazzi method; We remember that:

$$\sum a_i b_i^p = 1$$

Using the strassen equation;

$$a_i = 7$$
$$b_i = 1/2$$
$$7 * \left(\frac{1}{2}\right)^p = 1$$
$$\frac{1}{2^p} = \frac{1}{7}$$
$$p = log_2 7$$

This provides us with a mathematical relationship between the original size of our problem, and the number of sub-problems we need to create in order to get to our final size. We can plug our p value into theorem 1 of the Akra Bazzi method to get a final time complexity:

$$T(x) = \Theta\left(x^{(log_2 7)}\left(1 + \int_1^x \frac{g(u)}{u^{(log_2 7+1)}} du\right)\right)$$

7 Acknowledgement

We would like to express our gratitude to our mentor John D Shackleton for his guidance and support.

References

[1] Cormen, Thomas H., et al. *Introduction to Algorithms.* MIT Press; McGraw-Hill, 1990.

[2] Bender, Edward A., and Williamson, Stanley G. *Mathematics for Algorithm and Systems Analysis.* Courier Corporation, 2005.