Number Theory Handbook - 1

Analyzing the Time Complexity of Algorithms

Yuxin Xue

Department of Mathematics

University of Washington

This handbook is available free on :

https://doi.org/10.6028/NIST.HB.XXXX

March 2022

Abstract

In computer science, the time complexity is the amount of time that the computer takes to run an algorithm. With the development of computer, it is getting common for people to write the algorithm by themselves and it will be helpful if they know the time complexity of their algorithm. Finding out the time complexity will help people to understand and developer their algorithms better. Although there is easy way for people to estimate the time complexity by remembering the time complexities of basic statements, it is still meaningful to estimate the time complexity by counting the elementary operations. In this handbook, we will show how to estimate the time complexity by using the mathematical methods. Before reading this handbook, we assume readers has basic understanding about the different number base system and coding experiences.

Key words

Big O Notation;

Number Bases;

Length of Number;

Time estimates;

Time complexity.

Contents

1. Big O Notation

Before getting to the time complexity, let's firstly define the big O notation. The big O notation is invented by Paul Bachmann and Edmund Landau as a member of Bachmann–Landau notation (asymptotic notation). The big O notation is used to describe the behavior of an function when it getting to a limit value. In computer science, we use it to bound the function within in a specific range that helps us to know the running time of that function.

Definition 1.1 (Big O notation)**.** Suppose $f(n)$ and $g(n)$ are two defined, positive functions for all $n \geq n_0$, n_0 is an initial point. $f(n) = O(g(n))$ if there exist a constant C such that $f(n) \leq C * g(n)$ for all n.

Example 1.1. *Let* $n \geq 0$*,* $f(n) = 2n^2 + 3n + 1$*,* $g(n) = n^2$*, we say* $f(n) = O(g(n))$ *since* $f(n) = 2n^2 + 3n + 1 \leq 3n^2 = 3 * g(n)$ *by letting* $C = 3$.

Remark 1.1. Usually we choose $g(n)$ as a simpler function than $f(n)$ and $g(n)$ does not increase much faster than $f(n)$. This is because we want to use $g(n)$ to estimate the value of $f(n)$. If $g(n)$ grow much faster than $f(n)$, it will not bring us a good estimation. For example, the following are correct but not good big O estimation:

$$n^2 = O(n^3 + n^2 ln(n) + 1)$$
$$n^2 = O(e^{n^2})$$

Remark 1.2. The function $g(n)$ gives us the estimation of how $f(n)$ increases with the increase in n. For example $f(n) = O(n^2)$ means if n is doubled, then $f(n)$ increased by factor of 4. $f(n) = O(n^3)$ means if n is doubled, then $f(n)$ increased by factor of 8.

Theorem 1. *From the definition of big O notation, If the*

$$\lim_{x \to \infty} f(x)/g(x) = C$$

where C is any constant, then we can write $f(g) = O(g)$.

Example 1.2. *Let δ be any positive number, then $ln(n) = O(n^{\delta})$. The reason is if we let n becomes the really large number, for example, $n = 2^1000000$, then $ln(n) = 1000000 * ln(2) = 693147.1...$. Then we can easilly to find a number that $ln(n) \leq n^{\delta}$.*

Definition 1.2. For $n_1, n_2, ... \geq 0$, consider positive, defined functions $f(n_1, n_2, ...)$ and $g(n_1, n_2, ...)$ if there exist constant C such that $f(n_1, n_2, ...) \leq C * g(n_1, n_2, ...)$

Example 1.3. *Let $f(n_1, n_2)$ stands for the area of ellipse are in the x-y plane with semi-major axis x and semi-minor axis y. Then it is not hard to show that $f(n_1, n_2) = \pi * n_1 * n_2 \leq 4 * n_1 * n_2 = O(4n_1n_2)$.*

2. Length of Number

From the past math or computer science classes, we may already learn theories of the base of numbers. For example, the number system we mostly use is decimal number system, which is a base-10 system. In computer science, we also use the binary number system, a base-2 number system, and a octal number system. A base-8 number system. So, what is the largest K digits number N, and smallest K digits number M, to the base B? Let B equal to 10, and K equal to 10 and B equal to 6, we observe the largest number N is $999999 = 10^6 - 1$, M is $100000 = 10^5$. Therefore, we get the following theorem.

Theorem 2. *The largest K digits number to the base B is* $B^K - 1$*, the smallest K digits number to the base B is* B^{k-1}.

Then, it leads to the following result.

Theorem 3. *The number of digits required to write N to the base B is* $[log_B * N] + 1$*, where* $[n]$ *is the largest integer smaller than or equal to n.*

It is easy to show that $log_B = lnN/lnB$. And by the previous theorem, the length, or bitlength of a number, which is the bits that a number has, can be calculated.

$$length(N) = 1 + [log_2(N)] = 1 + [lnN/ln2]$$

Example 2.1. *Let A be a number with length n, B be a number with length m. The length of A+B is max(length(A),length(B)) or max(length(A),length(B))+1.*

Example 2.2. *Length(A)=n, length(B)=m, therefore*

$$2^{n-1} \leq A < 2^n 2^{m-1} \leq B < 2^m$$

Therefore, $2^{n+m-2} \leq AB < 2^{n+m}$.

Remark 2.1. Usually, we don't want to find the exact length of a number. What we want to find is the bound of the length that can help us to estimate the time later.

Example 2.3 (Find the length of n!)**.** *We can estimate the length of n! by using the big O notation. Notice that*

$$length(n!) \leq n(length(n)) = O(n * ln(n))$$

*Therefore we can say the length of n! is O(n*ln(n)).*

3. Time estimates

From this section, we will only consider the binary number system, which is the number to the base 2. Firstly, let's start with a simple calculation.

Let A = 1111000, B = 0011110. What is the value of A+B?

Algorithm 1 Addition of binary numbers

```
Require: $n, m \geq 0$
  $l \leftarrow max(length(n), length(m))$
  $i \leftarrow 1$
  $Result \leftarrow 0$
  while $i \leq l$ do
      $a \leftarrow$the i-th digit of n, $a \leftarrow 0$ if length(n) <i
      $b \leftarrow$the i-th digit of m, $b \leftarrow 0$ if length(m) <i
      if $a = 0$ and $b = 0$ then
          the i-th digit of Result = the i-th digit of Result + 0
      else if ($a = 1$ and $b = 0$) or ($a = 0$ and $b = 1$) then
          if the the i-th digit of Result = 0 then
              the i-th digit of Result = 1
          else if the the i-th digit of Result = 1 then
              the i-th digit of Result = 1
              the (i+1)-th digit of Result = 1
          end if
      else if $a = 1$ and $b = 1$ then
          if the the i-th digit of Result = 0 then
              if the (i+1)-th digit of Result = 0 then
                  the (i+1)-th digit of Result = 1
              else if the (i+1)-th digit of Result = 1 then
                  the (i+1)-th digit of Result = 0
                  the (i+2)-th digit of Result = 1
              end if
          end if
      end if
  end while
```

Definition 3.1 (Bits Operation)**.** As shown in the previous algorithm, we do this procedure l times. If we doing this once, it is call a bit operation. And we define the time of an algorithms takes to run is the bit opearions it will take.

Remark 3.1. Time(A+B) = max(ln(A),ln(B))

Then let's consider the more complicated example. What is the time for multiplying a number n to a number m. We assume length(m) < length(n).

Algorithm 2 Multiplication of binary numbers

```
Require: $n, m \geq 0$
  $l \leftarrow max(length(n), length(m))$, $k \leftarrow min(length(n), length(m))$
  $i \leftarrow 1$
  $Result = 0$
  while $i \leq k$ do
      if the i-th digit of m is 1 then
          result = result + n with (i-1)-th 0 appended at the end of n
      else if the i-th digit of m is 0 then
          result = result + 0
      end if
  end while
```

Remark 3.2. Therefore, we see that the for the multiplication of a l bits to k bits number, we have k additions and l bits operations for each addition. Therefore

$$Time(m * n) = O(ln(m)ln(n))$$

Example 3.1 (find the time of n!)**.** *From the previous section, we know length(n!)=O(nln(n)). For each multiplication, we are multiple a number with length O(n) to a number with length at most* $O(nln(n))$. *It requires* $O(nln^2(n))$ *bits operations, and we need to do this O(n) times. By the time of multiplication we calculated before.*

$$time(n!) = O(n) * O(n * ln^2(n)) = O(n^2 * ln^2(n)).$$

4. Conclusion

From the previous sections, we already know how to use the big O notation, the length estimation and the time estimation of addition and multiplication. Since all algorithms are implemented by combining additions and multiplications, we are able to analyze the time complexities by using the results we got before. I hope this handbook can teach you a new method to estimate the time complexity of an algorithm. Although this method may be less efficient and requires more calculations, but it all about knowing the math theorems behind the algorithms. This will help you understand the algorithm, complexity and number theory better.

References

[1] Neal Koblitz, University of Washington (2021) Number Theory Class Notes

[2] Fillion, Corless, Robert M., Kotsireas, Ilias, ACMES Conference. (2019). Algorithms and complexity in mathematics, epistemology, and science : proceedings of 2015 and 2016 ACMES Conferences.

[3] Kimberly Carey, Minnesota State University, Mankato. (NA) How to write a summary. https:// mathematicstrategies.weebly.com/written-summaries.html

[4] Bachmann, Paul (1894). Analytische Zahlentheorie [Analytic Number Theory] (in German). Vol. 2. Leipzig: Teubner.

[5] Mohr, Austin. ”Quantum Computing in Complexity Theory and Theory of Computation” (PDF). p. 2. Retrieved 7 June 2014.

[6] Michael Sipser (1997). Introduction to the Theory of Computation. Boston/MA: PWS Publishing Co. Here: Def.7.2, p.227

[7] Cormen, Thomas; Leiserson, Charles; Rivest, Ronald; Stein, Clifford (2009). Intro-

duction to Algorithms (Third ed.). MIT. p. 53.

[8] Donald Knuth (June–July 1998). ”Teach Calculus with Big O” (PDF). Notices of the American Mathematical Society. 45 (6): 687. (Unabridged version)