How to find time complexity of an algorithm

Ask Question

The Question

How to find time complexity of an algorithm?

What have I done before posting a question on SO ?

I have gone through this, this, this and many other links

But no where I was able to find a clear and straight forward explanation for how to calculate time complexity.

What do I know ?

Say for a code as simple as the one below:

```
char h = 'y'; // This will be executed 1 time
int abc = 0; // This will be executed 1 time
```

Say for a loop like the one below:

```
for (int i = 0; i < N; i++) {
    Console.Write('Hello World !');
}
```

int i=0; This will be executed only **once.** The time is actually calculated to `i=0` and not the declaration.

i < N; This will be executed **N+1** times

i++ ; This will be executed **N** times

So the number of operations required by this loop are

{1+(N+1)+N} = 2N+2

Note: This still may be wrong, as I am not confident about my understanding on calculating time complexity

What I want to know ?

Ok, so these small basic calculations I think I know, but in most cases I have seen the time complexity as

O(N), O(n2), O(log n), O(n!).... and many other,

Can anyone help me understand how does one calculate time complexity of an algorithm? I am sure there are plenty of newbies like me wanting to know this.

complexity-theory time-complexity

edited Jan 13 '15 at 1:35
Benjamin Gruenbaum
133k 44 296 364

asked Jun 14 '12 at 11:21
Yasser
22.3k 28 135 212

39 Bonus for those interested: The Big O Cheat Sheet bigocheatsheet.com – msanford Jun 9 '13 at 22:12

34 A beautiful question! – user2963623 Jul 4 '14 at 13:23

1 Check this blog out: mohalgorithmsorbit.blogspot.com. It covers both recursive and (especially) iterative algorithms. – Mohamed Ennahdi El Idrissi Mar 5 '15 at 0:10

2 isn't i ++ equivalent to i = i+ 1 ,if yes then it should be executed 2N times right? – Mudit Juneja Aug 15 '15 at 14:35

Thank you! Really nicely asked! – Prakash Raman Oct 7 '15 at 16:32

10 Answers

You add up how many machine instructions it will execute as a function of the size of its input, and then simplify the expression to the largest (when N is very large) term and can include any simplifying constant factor.

For example, lets see how we simplify `2N + 2` machine instructions to describe this as just `O(N)`.

Why do we remove the two `2` s ?

We are interested in the performance of the algorithm as N becomes large.

Consider the two terms 2N and 2.

What is the relative influence of these two terms as N becomes large? Suppose N is a million.

Then the first term is 2 million and the second term is only 2.

For this reason, we drop all but the largest terms for large N.

So, now we have gone from `2N + 2` to `2N`.

Traditionally, we are only interested in performance *up to constant factors*.

This means that we don't really care if there is some constant multiple of difference in performance when N is large. The unit of 2N is not well-defined in the first place anyway. So we can multiply or divide by a constant factor to get to the simplest expression.

So `2N` becomes just `N`.

Stanford (one of the best CS schools on the planet) is just starting a free online course on analysis of algorithms, I suggest if you are interested you join this course:

https://www.coursera.org/course/algo

Sign up is still open for a few days.

edited Mar 4 '15 at 0:17
nbro
3,573 4 17 54

answered Jun 14 '12 at 11:25
Andrew Tomazos
28k 21 104 190

10 hey thanks for letting me know "why O(2N+2) to O(N)" very nicely explained, but this was only a part of the bigger question, I wanted someone to point out to some link to a hidden resource or in general I wanted to know how to do you end up with time complexities like O(N), O(n2), O(log n), O(n!), etc.. I know I may be asking a lot, but still I can try :{) – Yasser Jun 14 '12 at 11:33

Well the complexity in the brackets is just how long the algorithm takes, simplified using the method I have explained. We work out how long the algorithm takes by simply adding up the number of machine instructions it will execute. We can simplify by only looking at the busiest loops and dividing by constant factors as I have explained. – Andrew Tomazos Jun 14 '12 at 11:36

2 yes, the link seems interesting, will try it out – Yasser Jun 14 '12 at 12:10

1 I have added a fantastic link below, explains clearly. – anirban chowdhury Jan 18 '13 at 10:05

1 Giving an in-answer example would have helped a lot, for future readers. Just handing over a link for which I have to signup, really doesn't help me when I just want to go through some nicely explained text. – bad_keypoints Jan 2 '16 at 4:48

This is an excellent article : http://www.daniweb.com/software-development/computer-science/threads/13488/time-complexity-of-algorithm

The below answer is copied from above (in case the excellent link goes bust)

The most common metric for calculating time complexity is Big O notation. This removes all constant factors so that the running time can be estimated in relation to N as N approaches infinity. In general you can think of it like this:

```
statement;
```

Is constant. The running time of the statement will not change in relation to N.

```
for ( i = 0; i < N; i++ )
     statement;
```

Is linear. The running time of the loop is directly proportional to N. When N doubles, so does the running time.

```
for ( i = 0; i < N; i++ ) {
  for ( j = 0; j < N; j++ )
    statement;
}
```

Is quadratic. The running time of the two loops is proportional to the square of N. When N doubles, the running time increases by N * N.

```
  else if ( target > list[mid] )
    low = mid + 1;
  else break;
}
```

Is logarithmic. The running time of the algorithm is proportional to the number of times N can be divided by 2. This is because the algorithm divides the working area in half with each iteration.

```
void quicksort ( int list[], int left, int right )
{
  int pivot = partition ( list, left, right );
  quicksort ( list, left, pivot - 1 );
  quicksort ( list, pivot + 1, right );
}
```

Is N * log (N). The running time consists of N loops (iterative or recursive) that are logarithmic, thus the algorithm is a combination of linear and logarithmic.

In general, doing something with every item in one dimension is linear, doing something with every item in two dimensions is quadratic, and dividing the working area in half is logarithmic. There are other Big O measures such as cubic, exponential, and square root, but they're not nearly as common. Big O notation is described as O ( ) where is the measure. The quicksort algorithm would be described as O (N * log (N)).

Note that none of this has taken into account best, average, and worst case measures. Each would have its own Big O notation. Also note that this is a VERY simplistic explanation. Big O is the most common, but it's also more complex that I've shown. There are also other notations such as big omega, little o, and big theta. You probably won't encounter them outside of an algorithm analysis course. ;)

answered Jan 18 '13 at 10:04
anirban chowdhury
4,188 4 22 41

7 Great answer. I found it even easier to understand than the accepted answer. Interestingly, my vote made your answer's number of upvotes equal to that of the accepted answer. :D – The Peaceful Coder Dec 29 '14 at 15:38

2 The *quicksort* algorithm in the worst case has a running time of N^2, though this behaviour is rare. – nbro Mar 4 '15 at 8:23

1 IIRC, little o and big omega are used for best and average case complexity (with big O being worst case), so "best, average, and worst case measures. Each would have its own Big O notation." would be incorrect. There are even more symbols with more specific meanings, and CS isn't always using the most appropriate symbol. I came to learn all of these by the name Landau symbols btw. +1 anyways b/c best answer. – hiergiltdiestfu May 8 '15 at 7:43

How to calculate for this - i (0 to N) and j (0 to i). Loop j is inside loop i. How to calculate the complexity in this case? – Bhavuk Mathur Aug 23 '16 at 14:49

2 Clearest answer ever – A.J Oct 16 '16 at 22:17

Taken from here - Introduction to Time Complexity of an Algorithm

1. Introduction

In computer science, the time complexity of an algorithm quantifies the amount of time taken by an algorithm to run as a function of the length of the string representing the input.

2. Big O notation

The time complexity of an algorithm is commonly expressed using big O notation, which excludes coefficients and lower order terms. When expressed this way, the time complexity is said to be described asymptotically, i.e., as the input size goes to infinity.

For example, if the time required by an algorithm on all inputs of size n is at most $5n^3 + 3n$, the asymptotic time complexity is $O(n^3)$. More on that later.

Few more Examples:

- 1 = O(n)
- n = O(n2)
- log(n) = O(n)
- 2 n + 1 = O(n)

3. O(1) Constant Time:

An algorithm is said to run in constant time if it requires the same amount of time regardless of the input size.

Examples:

- fixed-size queue: enqueue and dequeue methods

4. O(n) Linear Time

An algorithm is said to run in linear time if its time execution is directly proportional to the input size, i.e. time grows linearly as input size increases.

Consider the following examples, below I am linearly searching for an element, this has a time complexity of O(n).

```
int find = 66;
var numbers = new int[] { 33, 435, 36, 37, 43, 45, 66, 656, 2232 };
for (int i = 0; i < numbers.Length - 1; i++)
{
    if(find == numbers[i])
    {
        return;
    }
}
```

More Examples:

- Array: Linear Search, Traversing, Find minimum etc
- ArrayList: contains method
- Queue: contains method

5. O(log n) Logarithmic Time:

An algorithm is said to run in logarithmic time if its time execution is proportional to the logarithm of the input size.

Example: Binary Search

Recall the "twenty questions" game - the task is to guess the value of a hidden number in an interval. Each time you make a guess, you are told whether your guess is too high or too low. Twenty questions game implies a strategy that uses your guess number to halve the interval size. This is an example of the general problem-solving method known as binary search

6. O(n2) Quadratic Time

An algorithm is said to run in quadratic time if its time execution is proportional to the square of the input size.

Examples:

- Bubble Sort
- Selection Sort
- Insertion Sort

7. Some Useful links

- Big-O Misconceptions
- Determining The Complexity Of Algorithm
- Big O Cheat Sheet

edited Feb 25 '15 at 13:49
nbro **3,573** 4 17 54

answered Mar 27 '14 at 13:14
Yasser **22.3k** 28 135 212

Note: the first link is broken. – Ziezi Aug 23 '16 at 5:59

Time complexity with examples

1 - Basic Operations (arithmetic, comparisons, accessing array’s elements, assignment) : The running time is always constant O(1)

Example :

```
read(x)                              // O(1)
a = 10;                              // O(1)
a = 1.000.000.000.000.000.000        // O(1)
```

2 - If then else statement: Only taking the maximum running time from two or more possible statements.

Example:

```
age = read(x)                               // (1+1) = 2
if age < 17 then begin                      // 1
```

```
end;
```

So, the complexity of the above pseudo code is T(n) = 2 + 1 + max(1, 1+2) = 6. Thus, its big oh is still constant T(n) = O(1).

3 - Looping (for, while, repeat): Running time for this statement is the number of looping multiplied by the number of operations inside that looping.

Example:

```
total = 0;                                   // 1
for i = 1 to n do begin                      // (1+1)*n = 2n
      total = total + i;                     // (1+1)*n = 2n
end;
writeln(total);                              // 1
```

So, its complexity is T(n) = 1+4n+1 = 4n + 2. Thus, T(n) = O(n).

4 - Nested Loop (looping inside looping): Since there is at least one looping inside the main looping, running time of this statement used O(n^2) or O(n^3).

Example:

```
for i = 1 to n do begin                      // (1+1)*n  = 2n
   for j = 1 to n do begin                   // (1+1)n*n = 2n^2
       x = x + 1;                            // (1+1)n*n = 2n^2
       print(x);                             // (n*n)    = n^2
   end;
end;
```

Common Running Time

There are some common running times when analyzing an algorithm:

1. O(1) – Constant Time Constant time means the running time is constant, it's *not affected by the input size*.
2. O(n) – Linear Time When an algorithm accepts n input size, it would perform n operations as well.
3. O(log n) – Logarithmic Time Algorithm that has running time O(log n) is slight faster than O(n). Commonly, algorithm divides the problem into sub problems with the same size. Example: binary search algorithm, binary conversion algorithm.
4. O(n log n) – Linearithmic Time This running time is often found in "divide & conquer algorithms" which divide the problem into sub problems recursively and then merge them in n time. Example: Merge Sort algorithm.
5. $O(n^2)$ – Quadratic Time Look Bubble Sort algorithm!
6. $O(n^3)$ – Cubic Time It has the same principle with $O(n^2)$.
7. $O(2^n)$ – Exponential Time It is very slow as input get larger, if n = 1000.000, T(n) would be 21000.000. Brute Force algorithm has this running time.
8. O(n!) – Factorial Time THE SLOWEST !!! Example : Travel Salesman Problem (TSP)

Taken from this article. Very well explained should give a read.

edited Feb 25 '15 at 14:33
nbro
3,573 4 17 54

answered Apr 19 '14 at 9:36
Yasser
22.3k 28 135 212

In your 2nd example, you wrote `visitors = visitors + 1` is `1 + 1 = 2`. Could you please explain to me why you did that? – Sajib Acharya Dec 31 '15 at 9:09

2 @Sajib Acharya Look it from right to left. First step: calculate `visitors + 1` Second step: assign value from first step to `visitors` So, above expression is formed of two statements; first step + second step => 1+1=2 – Bozidar Sikanjic Jan 12 '16 at 9:46

1 @BozidarS, now I get that. Thanks for the help. :) – Sajib Acharya Jan 12 '16 at 9:59

thanks!!! awesome explanation :D – Paulo Costa Jan 30 '16 at 18:27

It is marvelous explanation . Thanks – Sourov Datta Dec 8 '16 at 5:53

Although there are some good answers for this question. I would like to give another answer here with several examples of `loop`.

- **O(n)**: Time Complexity of a loop is considered as *O(n)* if the loop variables is incremented / decremented by a constant amount. For example following functions have *O(n)* time complexity.

  ```
  // Here c is a positive integer constant
  for (int i = 1; i <= n; i += c) {
      // some O(1) expressions
  }
  ```